THE VISIONARY

PROGRAMMER'S HANDBOOK

or

Quilling the Great Adventure

by John Olsen

The Visionary Programmer's Handbook

or

Quilling the Great Adventure

by John Olsen

Printed in the USA
First Edition, September, 1991

ISBN 0-938385-28-3
UPC No. 0-10225-91150

Published by
Oxxi, Inc.
P O Box 90309
Long Beach, CA 90809-0309
USA

Phone (213) 427-1227 FAX (213) 427-0971

Dedication

To Ron, who first introduced me to adventure gaming way back in 1978.

and

To Kevin, who designed Visionary from the ground up, who answered over a thousand of my most stupid questions without a complaint, and who opened up a new world of adventure programming to computer users.

Table of Contents

Chapter 23: The MainLoop.SUB File

Chapter 24: The GetDrop.SUB File

Chapter 25: The Cannibal.SUB File

Preface

What makes a great adventure game? What makes it fun to play? Why do some games become famous and others not? What makes one game sell thousands of copies, another sell just a handful, while others can't even get published? How can you create an adventure game that will be fun to play? How can you make it challenging but not too hard? And how can you use Visionary to create your game? All these things will be revealed in this book.

We'll look at how to get started. We'll examine the different kinds of puzzles. We'll look at the special literary skills required to make the adventure come alive in the mind of the player. We'll see a variety of techniques and tips culled from over ten years of experience in writing and playing adventures. We'll see how to create the game-play logic that ties it all together. And we'll see how to use the specific commands and various features of Visionary to your best advantage.

If you've never written an adventure before, this book will be an invaluable aid in completing your first game. If you are an experienced adventure author, this work will help you hone your skills to produce a superior game.

This book is designed for users of Visionary, the Aegis Interactive Gaming Language. Using Visionary is certainly not a requirement for writing adventures, but it is recommended for all but the hardiest programmers.

Even if you plan on writing your adventure "from the ground up" in BASIC or assembly language, the first half of this book will be most helpful. The second half will give specific routines and examples of how to use the Visionary commands. It will give you the skills necessary to create the great adventure that has been inside you just waiting to get out!

Keys, Commands and Source Code

I have attempted to present information in a consistent manner throughout the manual. Certain type faces are reserved for specific kinds of information. This is designed to help you quickly decide which material is important to you.

Keys

Where single keys on the keyboard have names longer than one character, the name is enclosed in square brackets: **[Esc]**. This means press the **[Esc]** key. Multiple-keypress sequences are presented with a

hyphen between each key and the next: **[Ctrl]-C**. This means "hold down either of the **[Ctrl]** keys and press the **[C]** key before releasing the **[Ctrl]** key."

Styles

Below is a guide to the style standards in the manual.

General Information and Hints

> » General information is presented in boxed format. The » character means you can skip this discussion without seriously affecting your use of the program.

Commands and Other User-Entered Text

If you enter text from a line formatted like this:

```
Text in this format is to be entered as shown
```

you should type it character-for-character, including all spaces and punctuation.

Source Code Listings

Source code listings in Appendix A are presented with line numbers. These numbers do not appear in the actual source code which is included on the *Cannibal* disk, but are shown with the source listing for your reference only.

```
1   This line of source runs over the end of the printed page
          line, but the overlapping portion is not numbered
2   while this begins the actual second line of the source
3   and the line following this one is blank
4
5   etc.
```

Introduction

I first played a computer adventure in 1979, less than a year after I had purchased my first computer. That game was *AdventureLand* by Scott Adams, a true classic. It was my first experience with an adventure game, and it was wondrous!

Looking back at it now, I can see its warts. It took over eight minutes to load from a cassette, and completely filled the entire 16K memory of my TRS-80. The parser was primitive; only two words were allowed per command: GET KEY. CHOP TREE. KILL DRAGON. Input was strictly limited to verb and noun. There were no graphics. The display was split with the room description at the top of the screen, and all other input and output shown on the lower portion of the screen. The messages were short and frequently cryptic. And there weren't a lot of rooms or plot by today's standards. But I was entranced.

The plot, for those of you who never had the pleasure of enjoying this adventure, involved gathering thirteen treasures in a fantastic land of dragons, forests, and caverns. There was a wild assortment of objects in the land that defied logic. Paul Bunyan's blue ox Babe, jeweled crowns, Aladdin's lamp, and the magic mirror from Snow White were just a few of the things found in *AdventureLand*. You could visit volcanic chasms, blue lakes, and smelly swamps. If you took a wrong turn, you could even find yourself lost in a computer memory chip. When playing this adventure, you frequently had to give logic a vacation, and just go with the situation.

A year later, when I first tried my hand at writing my own adventure, I began to appreciate the immense task that Scott Adams had accomplished. Into a mere 16K of memory, he had squeezed the adventure interpreter and the data necessary for an entire adventure game. The interpreter had to contain all the routines for loading and saving games in progress, the parser for dealing with the player's input, the screen routines for the split-screen window, as well as the logic for moving to different rooms, getting, dropping, and using the variety of different objects. The data had to contain all the programming steps, the messages, the location descriptions, and the vocabulary list.

Scott Adams went on to produce a total of thirteen adventures, with titles such as *The Count, Ghost Town, Golden Voyage, Mission Impossible, Mystery Fun House, Pirate Adventure, Pyramid Of Doom, Savage Island, Strange Odyssey,* and *Voodoo Castle*. These games were later converted for use on Apple and Atari computers, and a few were finally made available on the Commodore 64.

None are available today for the Amiga. And perhaps it is just as well. They were products of an earlier and more innocent age. They would

not stand up well in their original text-only state, with their limited vocabulary and shorter storylines.

Still, I remember with pleasure the arrival of each new adventure. Usually the playing of a new game was a group process. About five of us would get together for a long evening of adventuring. With one person doing the typing, and the others all yelling out suggestions, we would play until the wee hours of the morning. Usually we would get stuck every hour or so. Then we would sit around and toss around a variety of more and more outlandish suggestions. Sometimes, by sheer luck or accident, we would strike the lucky combination and solve one of the puzzles.

I loved adventures. I loved playing them. But my mind kept toying with plots for my own adventures. These adventure stories were frequently inspired by the juvenile radio programs of the 1940's like *Jack Armstrong, Fu Manchu,* and *I Love a Mystery*. I was yet to be born when these programs aired over the radio, but I had heard audio tapes of them. The fantastic stories were perfectly suited to computerized adventures.

Then there were the Saturday afternoon serials at the movie theater. Although they also played before I was born, I had watched 16mm film prints of some of them. Twelve and fifteen chapter serials like *Captain Marvel* and *Daredevils of the Red Circle* were a perfect inspiration for the type of computer adventures that I had in mind.

I decided that I was going to write an adventure game. But I had no idea of how to go about it. There were no adventure authoring languages like Visionary at that time. My only choice was to write in the BASIC language. And that meant that I had to write everything: the parser, the input routines, the output routines, as well as the movement and other logic. But I had a pretty good knowledge of BASIC, and so undaunted I began writing my first adventure during my Christmas vacation of 1980.

The plot of my inaugural adventure was taken from the old horror movies. Its working title was *Frankenstein Adventure*. The plot had you as the player, discovering you were the long lost relative of Dr. Frankenstein. As his only heir, you had inherited his mansion. When you arrived, you found a letter from him telling you that he wanted you to complete his creature and bring it to life. The story was set in and around the old mansion. There was the obligatory graveyard and crypt. And of course, there was the windmill out in the center of the foggy old bog.

The plot twists in this first adventure are as effective today as they were ten years ago. You had to place a heart and liver in the creature. To do this required a visit to the cemetery, where you removed the organs from a corpse. But a hungry wolf barred your way as you attempted to leave the cemetery, and your only recourse was to give up the liver.

The wolf snatched it up and ran off. This left you with a creature in the cellar laboratory without a liver. But at a later time, the wolf appeared again, and this time you were able to kill it. You discovered that it was a werewolf, and in death it reassumed its human form. It was from this second human corpse that you obtained the liver you needed.

Another plot twist was the ending of the adventure. Throughout the entire story, you were under the impression that your goal was to complete the creature and bring it to life. You spent a great deal of time finding the various tools, instruments, and organs in order to complete the creature. Yet when you finally connected the electrodes and threw the electrical switch, the creature came to life in a shower of sparks and started toward you with murderous intent. It was at this point that you discovered your adventure was not over. Your final goal was to destroy the creature and save your own life. The way in which this task was accomplished was taken directly out of one of the Frankenstein movies. You lured him into the swamp, as seen in the climax of *The House of Frankenstein*.

The adventure was completed and debugged during my two-week Christmas break in 1980. It was my intent to submit it for publication, and sell it. But first I gave a copy to a friend to play. He loved it. But he did ask, "Why can you keep killing the wolf?" Sure enough, he had found a bug. I had the wolf appearing when the player visited the far corner of the cemetery. But I forgot to stop creating the wolf after you had killed it. Hence, the wolf kept reappearing and reappearing. It was a lesson well learned: always have someone else play your game after you are finished writing it—no matter how sure you are that all the bugs have been eliminated, someone else is sure to find one that you missed!

I submitted my first adventure to *CLOAD*, a cassette-based magazine for the TRS-80. They had previously bought some of my other, non-adventure games, and they snapped up *Frankenstein Adventure*. Several months later, it appeared in one of their issues. My first published adventure! Within a matter of days I started getting letters. Everyone loved it. I got letters from all over, even from other countries. Some were in foreign languages that I couldn't read, but had to have interpreted. Some people would ask for help. Others would simply write expressing their appreciation for the thrilling experience. And although the volume of letters dwindled, I still received letters for many years after that, as copies of my program continued to be circulated.

I was captivated. Writing adventures was more fun that playing them! I immediately started plotting other adventure stories. First there was one based on the jungle settings of the Tarzan novels, entitled *The Elephant's Graveyard Adventure*. Then followed a sequel using a few of the same locations, *King Solomon's Mines Adventure*. Both were published by *CLOAD* magazine in 1981. Others followed. The *Lost City*

adventure was so big, it had to be done in two parts. Keep in mind, these games were all written in BASIC to run in a maximum of 16K memory. The *Arabian Nights* adventure took place in Baghdad. *Shipwrecked* took place on a desert island. All of these were published by *CLOAD* magazine, or *SOFTSIDE* magazine, a printed magazine with a companion disk. For a time, adventure games were so popular that *SOFTSIDE* offered an **Adventure of the Month Club.** Several of my adventures saw their first publication as an Adventure of the Month Club offering.

I continued writing several new adventures each year on the TRS-80 through the early 1980's until the TRS-80 stopped production and the demand for software tapered down to a trickle. The Commodore 64 was the new kid on the block, so I upgraded my computer to a Commodore 64. I spent some months converting the first of my adventures over to the C-64. The BASIC languages used by Commodore and Tandy were similar but had substantial differences. I prepared to convert the rest of my adventures to run on the C-64, but before the task was completed, a wonderful thing appeared—an adventure-authoring system for the Commodore 64.

In early 1984, AdventureWriter was released. This adventure-authoring system allowed anyone to write adventures, with no programming knowledge. Not only that, the adventures you wrote with it were not in BASIC—they were in machine language. All messages were encoded so that even examining the disk sectors wouldn't reveal any clues. The parser was still somewhat limited, but with proper programming you could parse more complicated sentences. For example, you could say: "PUT THE ENVELOPE IN THE MAILBOX", something you could never do with Scott Adam's adventures, or with my BASIC ones. But best of all, the adventures could be much larger and still run with the instantaneous speed of machine language. No more pauses after the player typed his input. The adventures now reacted immediately.

AdventureWriter actually originated in England. It was written in 1983 by Graeme Yeandle and released by Gilsoft under the name **The Quill.** It was released under the name **AdventureWriter** in the U.S. by CodeWriter Corp. who also released Apple, IBM, and Atari conversions of it under the same name. One of the nice things about this spread of products was that once an adventure was written using AdventureWriter or The Quill, it was relatively simple to convert it to run on an Apple, IBM or Atari. From a marketing standpoint, this was a big plus—for a small investment in time, the same adventure could be sold on four times as many computers. And that translated into larger royalties.

At that point that I tossed out all further plans to convert my adventures to the Commodore 64 in BASIC. I started using AdventureWriter immediately to convert my old BASIC adventures over to the C-64. The program was a joy to use. It allowed me to create a pro-

gram much more sophisticated than I was previously able to write. My first conversion was the two-part *Lost City Adventure*. Since AdventureWriter allowed so much bigger adventures, both parts were combined into one giant adventure and given the title *Revenge of the Moon Goddess*.

I re-wrote my original *Elephant's Graveyard Adventure* and its sequel *King Solomon's Mines Adventure*, combining them into another giant adventure under the title of *Perils of Darkest Africa*. Then I sat down to write my first original story on AdventureWriter. I took my inspiration from an old radio mystery in which the dead apparently came to life and walked at night. My story took place in a Louisiana cemetery on the edge of a bayou. The result of several month's work was *Night of the Walking Dead*. It was to become one of my most critically acclaimed works. One reviewer wrote that he actually felt chills run down his spine when playing the game late at night. He even went so far as to write that it compared with Stephen King's novels. (I was flattered, but realized he was exaggerating).

All three adventures were published by CodeWriter Corp., the company that also published AdventureWriter, in a package called *The Thriller Series*. The software package was released in time for the Christmas rush of 1984, and could be found in the software section of all the major toy stores. Because of demand for a sequel, I converted of some of my old stories from the TRS-80 days. By the following summer a second package was released, called *Thriller II*. It contained *Shipwrecked, Son of Ali Baba*, and *Frankenstein's Legacy*. This last was my very first adventure, upgraded and expanded and now available on the Commodore 64.

There were more adventures and other publishers in my future. My next adventure was born from a desire to write an adventure that was so big, that it would tax the limits of AdventureWriter. From this inspiration came *Three Hours to Live*. It had 254 rooms, the maximum that AdventureWriter allowed. The completed adventure was so big that I could not convert it for the Apple computers, since the Apple has a lower memory limit on AdventureWriter than does the C-64.

And the adventures kept coming and coming. A few were written especially for use in my computer classes. Most were written with commercial possibilities in mind. There was *Eye of the Inca*, a treasure hunt inside an ancient Central American temple. *The Sea Phantom* was a ghost story that took place on the stormy New England coast. *Merlin's Gold* had you searching through Camelot looking for the wizard's gold. *The Magic Forest* was a "prequel" to *Merlin's Gold*. In it, you released Merlin from a magic spell. And then there were the two "sampler" adventures entitled *I Was a Cannibal for the FBI* and *The Secrets of Funland*. These were smaller adventures meant to be a sample of what an adventure is. The puzzles and plots of both were rather simple as adventures go.

I bought my Amiga in 1986, but had to wait until 1990 to write my first adventure on it. Until that time, there was no adventure-authoring language available on the Amiga. At least, there was none I found suitable. Oh, there were glimmers of hope—in 1986, AmigaVenture was released, but it was a BASIC adventure framework rather than an adventure language. In 1987 two similar programs, ADL and ADVSYS, were released. Both were adventure-authoring languages, but were complicated and didn't allow for anything more than pure text. Each was originally written for the IBM and ported over to the Amiga. And none of the above were released commercially. They were either public domain or shareware.

In 1988, Sunrize Industries advertised the first commercial adventure creation program called *Adventure Workshop*. I got to try it out, when the company contacted me to do the beta-testing for the program. It offered some sound and graphics capabilities, as well as a language that was easy to understand and use. Unfortunately, the program was never released commercially. If I wanted to write adventures on the Amiga, I would haveto use one of the systems I found lacking, or do it "from scratch" in BASIC. Then along came **T.A.C.L., The Adventure Construction Language**. This was the early precursor and "baby brother" to Visionary.

I first saw **T.A.C.L.** advertised in December of 1989. The following morning I was on the phone ordering it. I was told the program wasn't quite ready. The manual hadn't come back from the printers yet. But I put in my order anyway. It arrived during my Christmas break, and I spent most of my free time getting acquainted with the language.

During the first three months of 1990, I wrote three brand new adventures using T.A.C.L. The three adventures were all horror stories, and made up the *Nightmares from the Crypt* trilogy. The first was *Rings for Bony Fingers*. In it, you had to find the ten rings and replace them on the fingers of a skeleton. In *Ghostriders of El Diablo*, the second in the trilogy, you lifted an Indian curse from the old ghost town of El Diablo. The third and final adventure was entitled *Dr. Death's House of Horrors*, set in a large Victorian mansion that housed a wax museum. All three adventures were tied together by a central theme. In each, you were asleep having a nightmare. Each nightmare was different, but in each you had to find the way to awake before you died in your sleep.

In the summer of 1990, I heard the first whisperings of a new adventure language called **Visionary**. It would have its roots in T.A.C.L., but was to be much more extensive, with full graphic and sound capabilities. I was called upon to beta-test the software before its release, and found it to be a superb language. This is the language I use today.

Visionary's ease of use is the reason for this book—I foresee that many new adventure-game writers are out there, simply waiting for the tools to start creating. I hope this book wil be one of those tools.

There are many more adventures brewing in my imagination. As computers evolve and become more sophisticated, so will my adventures. I have no idea what computer I will be using years from now. But one thing I do know—I will be writing adventures on it.

John R. Olsen

August, 1991

Chapter 1: The Plot

You have an idea for an adventure. And that's where it all starts—with an idea. A suggestion. A thought nagging at your mind. "Gee, I'd like to see an adventure set in ancient Egypt." Or maybe a wish to see something that you've never seen before. "Why don't they ever make an adventure where you get to travel around in time?"

Whatever gets you starting to think about an adventure story, it's the first step. For many people it goes no further than that. But not you. You actually want to write an adventure. And you can. Using the interactive gaming language Visionary, you can write your own adventure with relative ease. The technology has arrived.

Get to Know Visionary

So how do you start? How do you actually make a computer game out of the story that you have in your head? First of all, get to know Visionary. Know its capabilities. Know what it can do, and what it can't. Do this before making any maps or writing out any location descriptions. By knowing what this adventure authoring system is capable of before you start planning your adventure, you can work around the limitations inherent in any system. It saves you the time required to go back and redesign parts of your adventure later.

Plotting Your Course

The next step in writing your adventure is the plot. Grab a pencil and some paper and start jotting down notes as you plot out your story. Let's say you want to write a story about ancient Egypt. Start making a list of things that will give the feel of ancient Egypt. Guards. Chariots. Pyramids. Flowing Nile. Barge on the Nile. Mummy. Tanna leaves. Hot sun. Sand. Full moon shining at night. Jackals baying at the moon. Oasis. Temple of Osiris. Slaves. Rock quarry.

Let your imagination run wild, as you search for things to give the feel of ancient Egypt. Use any memories you have of old movies. Maybe pictures. Stories you heard years ago. TV shows. Check out some books from the library. Non-fiction books. Fiction books. What you are doing is making a list of things that you can put in your adventure. Things that will make it fuller and richer, and give the player the feeling of really being there, in ancient Egypt.

What's the Story?

Now it's time to start pulling the story together. You must decide where the adventure will start and how the story will progress. Perhaps you will place the adventurer in the waterfront district of the port city of Alexandria. Or perhaps you want the player to begin out in the desert in the Valley of the Kings. Usually the plot of the story will help dictate where the player will begin.

Every adventure must end, and you must choose a suitable ending to your adventure. The player must have a final goal. And when this goal is reached, the player should be rewarded with a congratulatory message, telling him that she has completed the adventure. But you as the author must decide what constitutes the adventure's end. Does it end when she has collected all thirteen treasures? When he has awakened the princess? Or maybe when he has destroyed the last of the infidels? Sometimes making this decision is one of the hardest. You may have a great idea for an adventure story, but you must also choose a suitable final goal.

The Plot

Then there is the story—the plot. You need to do more than just decide the adventure will be about ancient Egypt. You must carefully craft a story. Perhaps this is a love story about the search for a kidnaped princess. Or maybe it's a treasure hunt, where the player is a grave robber trying to collect the treasures of the ancients. Or maybe the player is trying to prove he is the long lost son of the Pharaoh. As soon as you have decided, come up with a plot outline.

A plot outline will tell the story as you want the adventure player to experience it. For example: the player is fishing on the river bank of the Great Nile, trying to catch his dinner. He snags a bottle, which when opened releases a genii. Instead of granting him three wishes, the genii tells him that he is the true son of Pharaoh, stolen away at birth. It tells him that proof can be found in the House of the Dead, where the embalmers ply their craft. He visits there and is sent on a series of quests, finally resulting in finding a papyrus document proving his claim. He takes it to the palace, where he is welcomed with open arms by his long lost mother, the wife of Pharaoh. He has finished the adventure and won.

The Tone

You must also set the tone of your adventure. Is this a dark and somber adventure? A light romantic one? Perhaps a frivolous escapade. Or a pun filled comedy? Whatever you choose, you must be consistent. You should not change tone in the middle of an adventure. Imagine this. You are playing a scary adventure, lost deep within the ancient pyramid with a living mummy close on your heels. Suddenly Bozo the

Clown jumps out of a sarcophagus and hits you in the face with a cream pie. You've lost the mood. As the author, it is your responsibility to tell a smooth and even toned story. Of course, if you were playing an adventure set in a Fun House with rolling floors and squirting air hoses, then having Bozo jump out and hit you in the face with a pie would be most appropriate.

The Target Audience

The tone of your adventure and indeed even the plot itself will be frequently dictated by your targeted audience. Before you get too far into the planning of your adventure, you must decide who you intend to play the game. Is this going to be an adventure for kids? Or mainly for adults? For beginning adventure players? Or experienced adventurers? If your adventure is mainly intended for the first time adventurer, you will probably want to keep your plot simple, and the tone light. If your game is aimed at the intermediate to advanced player, then you may want to darken the tone, and certainly extend the plot. Everything you do in creating your adventure should reflect your target audience. As you will see when we start designing the challenges and describing the rooms and objects, this target audience will continue to have a major effect on what you choose.

The Map

At the same time you are doing all of this, a map should be drawn. At first, it may be a rough and incomplete map. But as you start to form the story, you will find the map will start to fall into place. And it will prevent you from making serious logical errors later when doing the actual programming. If you intend to allow the player to go north from the temple antechamber into the great hall, then going south from the great hall should take him back to the antechamber. Drawing a map as you plan your adventure will make sure these obvious directions don't become confused. Without a map it is amazingly easy to start inserting conflicting directions into your adventure.

Of course, you may want to make logical errors intentionally. A maze is a good example of that. Many adventures make appropriate uses of mazes. Originally, mazes were part of "cavern" type adventures. The player wandered through the maze trying to find her way to the other side and some objective. Traveling north in the maze would take her to some room, but traveling from there back south might not return her to the first room. The lack of logic was acceptable, because she was in a maze.

The use of mazes has since been expanded to be used in many other contexts. They can even be in wide open spaces. Perhaps the player is lost out in the desert. The sun beats down on the sand dunes, confusing him until he can't tell which way is which. He goes north and then

south. But he finds he is not back where he started. Under normal circumstances, this would not be logical. But since he has lost all sense of direction, the illogical becomes logical.

You've started with an idea. Then you fleshed it out with a plot. You've set the tone and chosen your target audience. Finally, you have a rough map sketched. The next step will be designing the puzzles that are at the heart of every adventure. This may be the single most important aspect of writing an adventure, and will be fully discussed in the next chapter.

Chapter 2: The Puzzles

The next step in creating your adventure game will be to design the various puzzles that will intertwine with your plot. You have designed your plot, set the tone of the adventure, chosen your target audience, and sketched a map. Now you will design the puzzles.

The Purpose of Puzzles

At this point in the creation of your adventure game, you have not touched your computer. Be patient. You are still laying the groundwork for your adventure. It's always tempting to just sit down at your computer and start writing your game. But if you do, you will find yourself going back and making changes to a story that was not completely thought out. Experience has shown that you will end up wasting less time if you finish your plan first. It could easily take twice as long to complete your adventure if you rush into it without proper planning.

The most satisfying part of playing an adventure is solving the puzzles. Overcoming challenges. Arising victorious despite overwhelming odds. Showing cleverness in solving the puzzles. A great feeling of accomplishment is felt when successfully completing each phase of an adventure.

On the other hand, a feeling of great frustration is felt when playing a poorly designed adventure with impossible puzzles. You want the player to feel good, not bad, about your adventure. That doesn't mean making the puzzles easy; the player wants to feel she has worked to accomplish something. It means making the puzzles a challenge. The player wants to feel that the glory which comes with solving the puzzles was earned, not that an easy solution was provided.

Adventure games rely on puzzles or challenges. The game should constantly make the player ask himself, "what do I do next?" or "how do I do that?". There is always some task to accomplish. There should be a variety of smaller goals to achieve on the way to the reaching the larger final goal. For example, the player's final goal might be to release the princess from a magic spell. One smaller goal might be to find the ancient manuscript which lists the ingredients for an antidote to awaken the princess. Other goals could be gathering the necessary ingredients to make the potion. The player might have to find the feather of the giant Roc. Or collect sand from the Kalahari. Or obtain the web from a black widow spider.

Designing Your Puzzles

Once you have listed the goals on paper, try to design a variety of different ways to reach achieve them. Don't make them all the same. You would not want the player to simply travel to the bird's nest to find a feather, then simply travel to the Kalahari desert to get the sand, and finally simply travel to a damp cave to get the web. Make the solution to each puzzle different. Make getting the feather harder. Perhaps the player will have to capture the bird in a trap first. Maybe traveling to the Kalahari is usually difficult. And the black widow might always bite the player causing death, unless he has the foresight to drink an anti-venom potion first.

Layer the Goals

The small goals should have further goals beneath them. To make your adventure more real and more exciting, you should have layers and layers of goals. Take the example of obtaining the feather from a giant Roc. Try to make it more difficult by splitting the task into further small goals. Let's say the player cannot find a feather that has been shed from the bird. She can only pluck one from the bird itself. So she builds a giant trap. But building the trap requires a lot of netting. She has to get this somewhere. Perhaps there is a stand in the bazaar that sells netting. But she has no money. So she has to get a job hauling animal carcasses to the glue factory. See how there are several different goals she must reach in order to get the feather. Get a job. Get money. Buy the netting. Build the trap. Trap the bird. And then pluck the feather. Small goals like these make reaching the larger goal more fulfilling. Upon obtaining the feather, the player experiences a real feeling of accomplishment.

Allow False Solutions

Let the player do things that serve no purpose in the final solution to the adventure. If looking for money, allow the player to get a job tending bar. Allow him to steal it from a shopkeeper. You may wish to have her caught for stealing and thrown in jail. You may wish to have him fired from her bartending job. In this way, you will force the player to get the money by hauling the carcasses. But provide various options. An adventure in which nothing can be done unless it is part of the solution, becomes a narrow, limited and boring adventure.

By allowing the player to do more than the bare minimum, you keep her on her toes. When she succeeds in some task, she never knows if it is an important part of the solution or not. By allowing the player to get a job tending bar, the player is under the impression that money can be earned that way. It is only later that she finds out there was another way to earn the money she needed. And she may even go back

and try to regain the job tending bar, figuring that she wouldn't get fired the second time.

Let's see how you can further enhance the problem of the Roc feather. You're not just going imply that players need to trap the bird. Make the players come to that conclusion alone, without prompting. Perhaps the first instinct will be to kill the bird. By all means you should design your adventure to allow the attempt. Provide weapons. Perhaps a sword or a spear. Make getting these weapons additional goals. Perhaps the sword has to be purchased with the money earned somewhere else. Or you could have players win weapons in an arm-wrestling contest. When they try to use the weapons on the Roc, they fail. Perhaps the bird is too agile—it avoids all sword thrusts. Or its feathers act as armor; the sword bounces off. The important thing here is that you have given several false solutions to this goal, and allowed the adventure players to find the correct one on their own.

Let's see how these techniques can be used in obtaining sand from the Kalahari. Offer the player several ways to travel to the Kalahari. Let's say the designed way is by magic carpet—but you also offer a camel caravan and a fast horse as other means of travel. Allow the player to discover the fast horse can't finish the journey. Perhaps it expires from heat when only halfway there, and she is returned to the city by friendly Bedouins. If you don't wish to allow the player to reach the Kalahari by camel caravan, find some way to prevent it. The important thing is to at least let the player try.

If you intend to allow the player to reach the Kalahari by magic carpet, there are a variety of smaller goals you can design. One is finding the magic carpet. Another is figuring out how to use it. You might have the carpet hidden in a locked trunk. A further goal would be to find the key. Then once the carpet has been found, somehow you have to let the player know it is a magic carpet, not just a normal one. Perhaps you could place a dim aura glowing about it. Or a small voice might tell the player that the carpet is magic. There might even be a small message embroidered on the bottom. Once the player realizes the carpet is magic, the magic words to operate it must be found. You might have these words revealed by a ragged seer sitting cross-legged in the sand, telling fortunes. You could have the words be written in an old book. You could even have them appear inside a fortune cookie! The important thing is that the player has had a fun time in the quest.

Make Puzzles Consistent

You want to make these puzzles consistent with the plot of your adventure. There is no sense in finding an aspirin listed in the ingredients of the magic potion. It's inconsistent with the historical setting. You wouldn't have the player travel to the north pole to find the feather of the Roc. It's inconsistent with the geographical setting. Releasing the princess from her magic spell by the use of an antidote potion is consis-

tent. Releasing her from the spell by winning the Great Roman Chariot Race is inconsistent. Make your puzzles consistent with the setting, the time frame, and the tone of your adventure.

Your puzzles should move the plot along. Each time one goal is reached, the player should feel he has come a little closer to the final goal. When the player has finally obtained the Roc feather, he knows he is now one third of the way to making the potion necessary for the revival of the princess. This leaves him motivated to continue playing your game. He feels success, and sees he is making progress.

Likewise, when your player returns with sand from the Kalahari desert, she knows she is two thirds of the way to the ultimate goal. Without this system that allows players to evaluate their own progress, players will frequently just give up playing the game. However, if the puzzles keep taking them closer and closer to his final goal, the players will keep playing your game all the way to the very end.

Types of Puzzles

There are different types of puzzles that you can place in your adventure. Some are built around the avoidance of death. Some are designed around acquiring or using some object. Some are created to get past an obstacle. And some puzzles are simply based on choosing the correct items to carry. You should place a variety of these types of puzzles in your game.

Obtaining and Using Puzzle Objects

The puzzles described so far have been about acquiring an object. The player was trying to get money, or some netting, or a sword, or a feather. Puzzles of this type are probably the most common ones in adventures. You will probably use them more frequently than any other. Whether your player is trying to obtain a treasure or is trying to obtain a tool to reach the treasure, seeking some object is the most frequently used type of puzzle found in adventures.

There are several different ways to make an object difficult to obtain. One is to make the object visible, yet difficult to reach. For example, you could have a gem visible inside a cage of tigers. The player can see the object, but he can't obtain it. The advantage to this type of puzzle is that the player knows what he must get.

Another way to make an object difficult to obtain is to hide it. You force the player to search a variety of places, seeking the correct hiding place. In some cases, you will let the player know what he is looking for. In other cases, the player won't know what he is looking for. You may let the player know he is looking for a key that will unlock a door. You might do this by simply stating " the door is closed and locked, and you don't have the key." In this case, the player will begin to search for the key, knowing what he is looking for.

In other cases, you might have the player searching for some way to avoid the tigers. In this case, the player doesn't know what she is looking for. She only knows that she wants to get the jewel without being killed by the tigers. So she searches for some means to accomplish this, whether she finds a gun, some poison, a suit of armor, or even a long string with a wad of bubble gum on the end.

Discovering how to use objects can be a related type of puzzle. For example, the player may have a flashlight, but no batteries. The player can't use the flashlight until he finds some batteries. You might even give the player dead batteries, and then require him to find and use a battery charger before he can turn on the flashlight. Or you might present the player with a broken object, say a broken-down old pump organ. It can't be played until the bellows are repaired. The player will have to find some glue or a needle and thread before he can fix the bellows and play the organ. Remember that finding objects isn't always the end of a goal. Sometimes the player must discover how to use the object as well.

Death Traps

Some puzzles are built around the avoidance of death. Many adventures have one or more death traps. There may be a trap door that drops the player into a pit of alligators. A poison dart may be shot from a small hole in the wall. A giant boulder may come crushing down. Anyone who is familiar with the old movie serials or the more recent blockbuster adventure movies will be familiar with the concept of death traps. These are always fun to have in your adventure. They add a little spice, a thrill, and a taste of danger.

Never Kill the Player Without Warning!

If you intend to use any death traps in your game, there are several things you will need to remember. Never kill the player without warning. **Never**. It's unfair to the player. Let's say the player is walking down a long tunnel when suddenly a huge sword drops down from above and kills him. The player will now have to start the entire game over, or pick up from the last position he has saved. And the player is justified in feeling unfairly cheated. Death that is a complete surprise marks a poor adventure, one which will usually be abandoned without being completed.

The player should always be able to complete your adventure game without ever being killed, if he is prudent. That doesn't mean you have to place obvious warnings before every death trap. There's not much fun to an adventure if every death trap has a bright red neon sign pointing to it. But you have to give the player some type of warning, even if it is vague. The player should be able to look back and realize his mistake and know that he was warned but failed to recognize the warning. You might have a cave drawing, depicting someone being slain by a sword dropping from the ceiling. You might have a fortune teller earlier in the game who warned him to "beware of dark places

where death falls from above." You might just have a message appear on the screen as he is a step away from the death trap, warning that "you hear a scraping sound from above." The warnings don't have to be obvious. But there should always be some type of warning.

Once you have designed your death trap, try to find some unique way to get past it. Give your player a long spear so she can pole vault over the pit of alligators. Let her carry a wax dummy that can be pushed in front of her and be struck by the dropping sword. Look for unusual ways to solve the problems. You could even turn a death trap into a means of escape. A pit in the floor of the underground maze could drop the player hundreds of feet to a swirling underground river, in which she drowns. But if she had worn a life preserver, the current would have carried her out of the underground maze to the valley below. What appeared to be a death trap actually turned out to be the only safe exit.

Don't over-use death traps. You could design a maze filled with death traps—one room might have a trap door, the next room might have a dropping sword, the one after that might have an electrified floor, and so on. Although they are fun and exciting, death traps can quickly become boring, especially if the player has been in that part of the adventure before and is trying to move on to the next part. It is usually best to spread them out in your adventure. If you choose to use death traps, use them sparingly, and find clever and unique ways to avoid the trap.

Similar to death traps are various ways of dying by hunger or thirst. Sometimes you may want to make the player hungry or thirsty. Perhaps the player wanders out into the desert. You might wish to make sure he takes water with him, or else perish. If you plan on keeping track of time as your adventure is played, you may wish to require the player to eat at least once a day. The puzzles here could be to find a source of food or water. You might even force the player to find a way of carrying the water out into the desert with him.

As mentioned before, though, always give the player some warning. Never have him suddenly die from thirst or hunger. Give several warnings, such as "You are starting to get thirsty", "You are parched!", "You are dying from thirst" and "You won't be able to make more than three moves without some water." Try to allow the player enough moves to go back and get water at the first warning. One advantage to these types of puzzles is that they frequently require the player to carry food or water along, and that adds to the inventory, forcing the player to carry fewer other items. As will be shown later, limiting the number of items in a player's inventory can result in another type of puzzle to solve.

Sleep can be used in similar ways to eating and drinking. Just as you can make the player feel hungry, you can make him sleepy. This can be done by keeping track of time passing, and making your player sleepy

at night time. There might be a variety of reasons to want your player to fall asleep. Perhaps you will have him die from the cold, unless he sleeps indoors. You could have him robbed of his valuables while asleep, unless he locks the door. You could make him weaker if he refuses to sleep, thus causing him to lose an arm wrestling contest.

Here are some of the things to consider when you use sleep in your adventure. When do you want the player to sleep? Every night for six hours, or just after drinking a mysterious potion? What will be the consequences if she doesn't sleep? Will she fall asleep anyway after so many turns, or will she stay awake but just be weaker? Must she do anything special before going to sleep? Does she need to lock herself in the house, or wear a crucifix to ward off a vampire? Do you want any positive consequences if she sleeps? Perhaps she will have a dream giving him some vital clue? Or do you just want to avoid negative consequences by sleeping? Will she simply avoid theft, weakness or death? Using sleep in your adventure is not required, but it can enhance your game.

Obstacles

Some puzzles are based on getting past an obstacle. These are similar to death traps, but death is not involved. Instead, something non-fatal is in your way. It could be something as simple as a locked door. Or perhaps you have to find a way across a river. These puzzles are not like a death trap. You won't die if you try to swim across. Usually in cases like this, the adventure game simply won't let the player cross. It might give a message like "You can't swim, and decide against trying," or "The current's too swift, so you turn back." It's a non-fatal obstacle that the player has to overcome. Try to place several of these types of puzzles in your adventure.

As with any puzzle, your job in creating the solution to an obstacle puzzle is to try to mislead the player. The player might be trying to get inside a castle. The obstacle is perceived to be the drawbridge. The player can't seem to find a way to lower it. As the author, you place various "red herrings" in the player's path. There could be a horn to blow, but no one responds. Maybe a bell to ring, but again no one answers. Perhaps even a rope to pull on, which unfortunately still doesn't lower the drawbridge. However, the perceived obstacle is not the true obstacle. Perhaps the drawbridge can never be lowered. You have used the "red herrings" to mislead the player into thinking that the drawbridge is the way to enter the castle. Actually, the solution to entering the castle could be as simple as building a ladder to scale the walls.

» **Clues** Always give the player a hint, some clue as to the solution to the problem. Never leave the player totally frustrated as to the solution to any problem, or the player may give up and leave the adventure. You don't want to make the clues too obvious; you can be subtle.

In the example, perhaps the top of a ladder is just visible, sticking above the walls of the castle. Or perhaps the player saw a small ladder out in the forest earlier in the adventure. You could even have a band of elves run giggling through the woods carrying a tall ladder, after the player has tried the more obvious "red herrings." However you choose to have the player get past your obstacle, make sure that you have left some clue.

Inventory Limits

Some puzzles are built around the player selecting the right objects to carry along. What should he take into the castle? What will he need? What won't he need? The reason that this can be considered a puzzle is that usually the player can only carry a limited number of items. Inventory limits are frequently an integral part of puzzles. The player needs to choose carefully what he will carry. You can make it impossible to complete the adventure without using six items even though he can only carry five. This requires taking some of the items to one location dropping them, and going back for the rest.

As mentioned previously, if one of the items in the player's inventory is a required item like food or water, then it cuts down on the number of other items that can be carried. This is one way to make solving a puzzle a bit trickier. As the game designer, you know how many items are required to solve each puzzle. So you may decide to set the inventory limit in such a way that carrying the required food or water necessitates making a second trip to carry all the other required objects.

You could even make the second trip more difficult, by placing obstacles in the player's path. For example, a troll might stop the player each time she crosses a bridge, and demand payment before the player is allowed to continue. You might create enough gold to allow payment only once, yet require the player to make two trips across the bridge to carry all of the items she needs to complete some goal. That forces the player to find some alternate way of bringing back the second batch of objects. Perhaps she will have to swim across the river, kill the troll, or find a second source of gold. All of this became necessary because there was one extra object that the player needed but couldn't carry due to the inventory restriction.

We've covered the basic types of puzzles you can put in your adventure. There are death traps, obstacles, objects, and inventory related puzzles. These four general types can frequently be combined and modified. For example, the avoidance of a death trap in the form of a pit of alligators might require the building of a trampoline. To do this, the player needs to acquire the necessary parts and tools. And since he is limited to carrying no more than a certain amount, he must carefully manage his inventory.

Don't be afraid to try something new. Use your imagination—be inventive.

Design Guidelines for Puzzles

Regardless of what type of puzzle you design, there are several guidelines to follow. Keep puzzles logical. Don't make them impossible. Give clues. Allow objects to have several uses. Mislead the player regarding the value of an object. And finally, vary the types and difficulties of the puzzles you choose. Following each of these guidelines will make your adventure a more rewarding game to play. Let's examine each guideline further.

Keep Them Logical

Keep the puzzles logical. You expect the player of your adventure to behave logically. You expect him to try to solve your puzzles logically. So don't throw puzzles at him that defy logic. If you had designed a medieval adventure, it would not be logical to create an obstacle in the form of a futuristic metal robot. It would be even more illogical to solve the puzzle of getting past the robot by feeding it a cheeseburger, since robots don't eat. You shouldn't expect the player of your adventure to attempt illogical things to solve your puzzles.

If you were designing a time travel adventure, a robot would be a logical obstacle to create. But using a cheeseburger to get past it is still not a logical solution. You don't want to force the player to guess solutions from thin air. It would not only take forever to find the right combination, it would also be extremely frustrating. It would be better to have some other way of getting past the robot; some way that is tricky but more logical. Perhaps the player could short it out with electricity. Or rust it with water. Or lure it into a death trap meant for the player. When the player overcomes the robot, he will feel much more satisfied if he has used logic to do it, than if he relied on pure luck.

You can use solutions to puzzles that seem illogical, if you later show that they are indeed logical. It's best if you have given the player some clue before he encounters the problem. Take the problem of the bear that was encountered in Scott Adams' classic *AdventureLand*. The player stood on a narrow ledge, confronted by a bear. The bear

wouldn't let the player pass. The solution to the puzzle was to scare the bear by yelling. The bear was so startled that it fell off the ledge.

At first this solution seems illogical. No player could be expected to solve the puzzle in such an unexpected manner. But in this case, the player was given a clue: "Don't waste honey, get mad instead." So although the solution was illogical, it was still acceptable. The player had been given a clue, and was not expected to find such an unusual solution by himself. To have expected him to solve it without any help would have been unfair to the player.

If your game has multiple solutions, one answer should be better than the others

The above mentioned hint "Don't waste honey, get mad instead" illustrates another technique in designing your puzzles. You can design a puzzle with several solutions, but only one that is best. In the example taken from *AdventureLand*, there were two solutions to the bear puzzle. The player could scare the bear, or he could feed it the honey. After the bear ate the honey, it went to sleep allowing the player to pass. But this alternate solution was not the best one, because it required the player to give up the honey. And in that adventure, the honey was one of the thirteen treasures the player was supposed to collect. Keep this technique in mind when you design your adventure. Allow several solutions to some puzzles, but make each have drawbacks except one.

Provide a Possible Solution

Don't make your puzzles impossible. Don't get so tricky in your design that even an advanced player couldn't figure out the solution. Remember who your audience is. If you are designing an adventure for beginners, keep the puzzles easier. If you intend for it to be played by seasoned adventurers, then make the puzzles harder. But remember that it is easy to get carried away and make them too hard. Any adventure that is too hard is an adventure that won't get played.

Provide Clues

That doesn't mean you have to throw out a good idea, just because the puzzle is too hard to solve. You'll just make it easier to solve by giving the player clues and hints. Let's say you have a terrific idea for an obstacle; a wolf bars the player's way out of the graveyard.

You want to make the solution a tough one. The wolf can't be fed or killed. The secret is that this is no normal wolf, but a werewolf. It can only be killed by a silver headed cane. To make the puzzle harder, you provide the player with "red herrings." The player finds some raw meat which she can try to feed the wolf. She finds a revolver with which she can try to kill the wolf. Of course, neither works.

The puzzle as it stands is too hard. The player would probably never attempt to strike the wolf with the cane on her own, after failing with the revolver and raw meat. It would not be logical. So as the game

designer, it is your responsibility to make the puzzle a bit easier by providing some clues to the solution. Make sure the player knows the cane has a silver head. Go further and describe it as a "silver head fashioned into the shape of a wolf's head." You might even have the wolf snarl and glance uneasily at the cane, if the player carries it. All of these clues and hints can lead the player to discover the solution to your puzzle.

Another way to give clues and hints is by the use of the HELP command. This is one of the few traditional commands you will find in nearly every adventure. Just as a player can type "inventory" to find out what she carries, the player can usually type "help" to get some clues. When the player does request help, your game should respond with some appropriate clue.

You can use the HELP command to provide clues and hints

Don't just come out and give the solution to the current problem. Don't say "Strike the werewolf with the cane." There's no fun in simply being told what to do. It's much more satisfying for the player to figure it out for herself, after reading the hint. Make it a clue which must be deciphered. More appropriate would be, "This is not a normal wolf. It appears afraid of your cane." This type of clue would not be given in the normal description of the wolf. It should only be given if the player asked for help. And naturally, it should only be given under the proper circumstances, when the player carried the cane and was confronted by the wolf.

When you design the helps in your game, be sure they occur at the proper time. If the player stands by a locked door and asks for help, your game should not give the response about the wolf. It should instead give some clue to the whereabouts of the key to the door. Plan ahead where you feel the player will need help, and make hints available that will assist the player without giving too much away.

Always have a "default" help. Remember that the player may ask for help any time during the playing of your adventure. And you frequently may not want to give him any help. But you need some response for his request. You need a default response, like "Sorry, you get no help right now. Maybe later. Just remember to examine everything." That way the player knows that there may be helps later, and not to stop asking for them. It also reminds him of the basic adventure playing strategy of examining all objects.

Give Objects Several Uses

Another suggestion to follow in creating a great adventure game, is to give objects several uses. A flashlight, for example, has the obvious use of providing light. But don't stop there. Surprise the player by giving the flashlight other uses. It could be used as a weapon, to knock out a guard. It could scare off wild animals in the night. It might have a secret message engraved on the lens, which would only show up when

the flashlight is turned on. Perhaps the player could remove the batteries to use for some other purpose. The important thing is that you have created other uses for the flashlight, in addition to the expected one. Use this concept frequently in your adventures.

Mislead the Player about Objects

Mislead the player as to the value of an object. All too often, adventure games give the player objects which have only one purpose. And once that purpose has been served, the object is no longer needed. For example, the player may find a key which she uses to unlock a door. Usually the player will assume that the key has no further value, since it has been used to solve a puzzle. But as the game designer, you could fashion other uses for the key. Perhaps later it will be used to open a padlock, be fed to a metal eating monster, or be melted down and fashioned into a magic ring. You have tricked the player into thinking the object is not needed any more, when it really has further value. This is a clever technique that should be used whenever possible.

Vary the Types of Puzzles

Vary the types of puzzles you choose. Don't make your player find object after object after object, with no other type of puzzle offered. He will all too quickly become bored. Toss in a death trap. Limit the number of items that can be carried in the player's inventory. After the player finds an tool, force him to discover a way to use it. Then create some obstacle to overcome. Keep giving the player a variety of experiences. It will keep him interested.

Give your game lots of puzzles, and vary them. Not only vary their type, but vary their size. Space the small puzzles in between large ones. Some goals have many other smaller goals beneath them. These are large puzzles. Others have very few smaller goals. Mix these smaller puzzles in with the larger ones. Getting past a closed drawbridge may consist of many small goals. The player must get an axe, chop down some trees, purchase hammer and nails, and build a ladder, all before she can scale the castle walls and get past the closed drawbridge. Getting past a locked door, on the other hand, could have very few small goals. The player finds the key hidden in an urn and opens the door. To keep your adventure fresh and enjoyable, put smaller puzzles in between the larger ones.

If you follow all these guidelines in creating the puzzles for your adventure, you will have designed a more interesting and playable game. Remember that there are few hard and fast rules. Use your own best judgement when you create your game design. When the game is finished and tested, if you find some of the puzzles just don't work, you can always go back and modify them.

One final word of caution. Don't get carried away with your puzzles. Don't make them too hard. Don't make them too complicated. Don't require so many steps to reach one goal that the player forgets what he is after. Don't make the answers to each puzzle so complicated, illogical or unusual that the player gives up, or can't find it. Remember that you are creating a work of entertainment, not frustration. You want to design something that people will enjoy and want to play.

The puzzles are probably the main part of any adventure game. As such, there has been a lot to write about them. If you design them well, they will fit into your plot smoothly and make a wonderful adventure. They will become the things your game is known for.

Chapter 3: The Places

In the previous chapters, we have discussed designing your adventure story, choosing your target audience, sketching a map, and creating the puzzles or challenges. All this was done without even touching your computer. You were laying the groundwork for your adventure. The next steps in creating your adventure game will involve using Visionary. Boot it up and let's begin putting your adventure into the computer.

The Room

When playing an adventure, the first thing the player sees is the location, frequently just called the "room." As such, these become one of the most important parts of the game.

If you are designing a classic text adventure, the room will be described in words of your choice, and these room descriptions will set the mood of the story. If you are designing a graphics adventure, the artwork will describe the room for you—illustrating the old saying, "a picture is worth a thousand words".

As the game author, you control what the player experiences. If the location is well described or drawn, the entire game is more exciting and fun to play. Although much of what is important to a text description also applies to a graphic adventure, most of this chapter will focus on text descriptions of the rooms and how to get the most out of them. The way you describe these locations in words is of vital importance to a text game.

Descriptions

When you start describing each room—remember, we'll call it a room even if it's an outdoor location—there are a variety of things you need to consider. The first is accuracy. You want nothing that is unclear or ambiguous.

Exits

Start with the obvious exits. Be sure you describe them all in your room description. If there are doorways to the north, east and west, be sure you describe them in a way that the player can understand where they are and what they look like. It isn't fair to the player to have obvious exits that aren't mentioned. If the player can freely go north, make sure he knows he can go north. Don't make him discover it by chance or experimentation. However, this only applies to **obvious** exits from which you can leave without any special action.

If there are **hidden** exits, then you won't want to list them in the room descriptions. You would never say, "There is the entrance to a tunnel hidden beneath the statue." Part of the fun for the adventure player is finding that tunnel for himself using the clues that you leave him. But if the exits are only partially hidden, you may want to hint at their existence in the room description. "Standing in the temple, you feel a breeze coming from above. It may indicate a tunnel entrance on the ledge." This notifies the player that if he can get up on the ledge, he will find an exit.

The Rest of the "Room"

You not only need to make sure the exits are accurate, you also need to make sure the rest of the room is accurately described, too. Let's take for example the description of an Egyptian temple room. You must accurately describe the entire room. Describe the smooth stone floor, the skylight in the high ceiling, and the etchings covering the marble walls. Describe the wide altar, the statue behind the altar, and the stone benches lining the walls. Historical accuracy is important. Don't describe the benches as plastic; it wasn't invented yet. You want a complete and accurate description of the room.

If something in the room can change, then it should not be listed in the room description. Doors are a good example. If there is a wide open doorway to the east, then include it in your room description. It won't change. You might describe it as "an open archway leading east." The important thing is that the player knows it is open and will stay that way. On the other hand, if you design the room with a door that can be closed and locked, then you must be careful when describing it in the room description. If you describe it as a closed door, the room description will no longer be accurate when the player has opened the door.

Things That Change

There are several ways to deal with objects that can change. In the case of doors, you have two choices. One way is to describe the door in the room description as simply a door; not open, not closed, just a door. The player can then examine the door by typing "LOOK AT THE DOOR" and you can have your adventure give an appropriate description like, "the door stands open" or "the door is closed and locked." Of course, you will design your game so that the player will see a similar message, if he tries to travel through a door that is closed.

A second way to deal with a changing object (like the door) is to not include it in the room description, but rather make it an object instead. You could actually have two objects in your program: "a closed door to the east" and "a door standing open to the east." In this way, when the adventure describes the room followed by the objects, the player will

not only be aware of the door, but also will automatically know if it is open or closed. Only one of the two objects will appear in the room at any one time, and will be swapped for the other when the player opens or closes the door.

This second method has other uses as well. It can be used with anything that is stationary in a room and can not be picked up. It can be used to show "a stone statue with its arm raised" or "a stone statue with its arm at its side." It can be used to show "a wide altar" or "a wide altar with a trap door leading downward." This method can also be used with objects that are movable. You could have two objects, "a dry torch" and "a burning torch" which can be carried around. When you either light your torch or douse it, the object in your possession is swapped for the other one.

A third method that Visionary has made possible, is to have a single object and make use of its attributes. The object "door" could have the attribute of "open." Each object can have up to 32 attributes, which are either "yes" or "no." If the door is open, you can set the door "open" attribute to "yes." Likewise, if the door is closed, you can set the door "open" attribute to "no." When the room is described, the proper description of the door can be given, depending on whether the "open" attribute is "yes" or "no."

Similarly, the object description for the door can be modified depending on the status of the attribute. This method will work equally well with other objects, both movable and nonmovable. A single nonmovable object like an altar can be either described as "a wide altar" or "a wide altar with a trap door leading downward" depending on one of the altar's attributes. The torch mentioned above could have a "burning" attribute, which could be checked before describing the torch as either "a dry torch" or "a burning torch." And the statue mentioned previously could be a single nonmovable object with a "arm" attribute. Further details on attributes will be discussed later, when we begin to work with the adventure objects and their descriptions.

Use Descriptions to Enhance the Game

Be sure not to limit your descriptions to only those items that are an important part of the game. Perhaps the stone benches are an important part of the game because sitting on them opens a trap door. Then naturally they should be described. But don't stop with just those items. Include things that are purely for "window dressing." Describe the altar even if it actually plays no part in your game. Describe the steps leading up the altar, even if they serve no purpose. Doing so helps make a fuller and richer adventure and makes the room seem more real and exciting.

The way in which you describe a room is another important factor. Remember that you are creating a new world for the game player. So

don't leave your descriptions short and bare. Make the player feel the tone, the mood. Make him feel what you want him to feel. Here's a description barren of mood: "You are standing in a long tunnel." Now compare it with this: "You stand bent over in a low tunnel that leads into darkness. The flames of your torch cast flickering shadows on the glistening wet rock walls." The second description makes the player feel more like he is actually there. It's certainly going to make the adventure a lot more fun to play.

Similarly, if you plan on designing a graphic adventure, don't make the room graphics sparse. Make them rich in detail, to enhance the mood of your game. Don't just draw a long dark tunnel. Add a flickering source of light. Put shadows in the corners of the tunnel floor. Show the masonry between the rocks in the rough walls. Give your drawings texture. Your final game will look so much better, and create a much more impressive image in the players' minds.

Don't describe only what the player can see, but also include the things he can hear and smell. "The sound of dripping water echoes in the distance. A musty smell fills your nostrils." Each time you add another of the senses to your description, you make the world of your adventure seem more real. We have five senses: sight, hearing, smell, taste, and touch. Yet it is surprising how frequently adventure authors only use the single sense of sight. Try to fit sight, sound, and smell into every room description. Tell the player what he sees, and also tell him what he hears and smells.

Usually taste and touch are not appropriate for room descriptions. But they are certainly desirable when the player takes some action, like eating food or wearing gloves. There are times when you can include the sense of touch in a room description. "You stand outside the temple in the hot desert sun. The burning rays bake against the back of your neck. A light breeze cools your skin." On the other hand, taste is nearly always absent from a room description, since it requires a specific action from the player (such as "eat the wafer" or "drink the potion"). Taste can be an important part of any adventure, so don't forget to use it as you design the other sections of your adventure.

You may wish to add some hints to your room descriptions. These hints could warn the player of certain potentially dangerous actions. In a text game, tell the player that "Etchings on the wall depict an Egyptian being struck by a sword as he opens the small north door." In a graphics game, actually **show** the etching on the wall. You may want the player to examine the etching further (usually by clicking the mouse button on the appropriate part of the scenery) before he is shown the picture, but he should be shown it. This would serve as a warning to the player that a death trap is ahead, and that perhaps wearing a suit of armour would be advisable.

Or you could add hints to the room description that tempt the player to try some action. "You see handprints on the dusty statue, as though

someone had been pushing against it." In a graphics game, show the handprints, and allow the player to examine them more closely, at which time you print out a message similar to the one above. You might include this type of message in a room description if you wanted to suggest that the player push the statue aside. The room descriptions and drawings are certainly not the only places to leave hints for the player, but many times they can be appropriate ones.

Don't put things in your room descriptions or show things in your room graphics, unless you plan on allowing the player to take at least the most rudimentary actions with them. You should allow the player to attempt to get them, examine them, and use them. If you have a temple room containing a stone bench, then you must be prepared to allow the player to get it. Or at least try to. You might wish to simply reply, "The bench is too heavy, and won't budge." But you have at least acknowledged the player's attempt.

Likewise you must plan ahead to allow the player to examine an object, even if the response is "You see nothing special." And you must plan to allow the player to use it. In the case of a stone bench, anticipate the player will try to "SIT ON THE BENCH." Even if all you are going to do is to reply "OK", you must plan on some appropriate response. If you don't plan on allowing the player to at least examine an object, then don't show it in your room graphic or mention it in your room description.

Connecting the Rooms

The next thing to do when designing your locations is to connect them. On your original map, you had them connected. Traveling north from the anteroom moved the player to the great hall. You have finished carefully describing the obvious exits in your room descriptions. Now it is time to make sure your game will allow the player to travel in those same directions. Using Visionary, this job becomes very easy. Just above your room descriptions, you can list any default directions. These are the directions in which the player can go when the adventure begins. They may change as the game progresses, but for now you will list which rooms connect which other rooms as the game begins.

As you list which rooms can be reached from the current one, do not connect the current room to any other room that has a hidden entrance or other closed passage. Only include the obvious exits that can be used. For example, if there is an open archway to the north that leads to the temple, then be sure to connect the anteroom to the temple by going north. However, if there is a secret panel that allows the player to travel south into a hidden room, do not include it in your room connections. You do not want to allow the player to travel south into the hidden room. Until the player finds and opens the secret panel, the rooms are not connected. Keep these types of things in mind as you connect your rooms.

Room connections can be changed as the adventure is being played. You will always have the option of connecting or disconnecting rooms in the middle of the game. There will always be passageways that open and close. Perhaps locked doors become unlocked, connecting two previously unconnected rooms. Or there may be a cave-in in a previously open tunnel, disconnecting rooms that were previously connected. All of these things will take place after the adventure has started, and will depend on the actions of the player or on some random event. They will be dealt with separately, so don't include them as you are creating this part of your adventure. Just keep them in mind for later reference.

When creating your locations with Visionary, you will use text files. You can use one or more files to hold all the locations in your adventure. For each room in a Visionary text game, you will probably want to include two room descriptions. One is a complete description of the room that the player will see when first visiting the room. The second is a short description for the player to read when the room is visited any time after that. In a Visionary graphics game, the shorter description should be sufficient, since the player has the room graphic to look at. The room file will also contain the default directions that the player may travel to exit the room. And it may contain certain attributes for each room, such as if the room is dark or if the room is flooded. The next chapter will explain more about these variables called attributes.

The Store Room

When you design your rooms, be sure to include one that cannot be visited. It will be used as a **store room** for objects that are not currently being used in the game. For example, if the player will eventually find a diamond buried in a mine, the diamond is stored in this store room until it is found. When the player first enters the mine, he doesn't see the diamond, because it is not stored in that room. It is stored in the special room you have designed for just that purpose.

After the player has dug in the mine, you will move the diamond from the storage room to the mine, and will tell the player he has found the diamond. Since this special room is only used to store objects that are not currently being used, it will have no exits. Likewise, no room will exit to the store room. And it is not necessary to give the room any special description, since the player will never visit there. But even though the room has no description and no exits, it is still a vital part of any adventure and should not be forgotten.

Game Graphics

Graphics can play a part in any adventure. You may or may not plan to use graphics in your adventure. If you don't plan on using any graphics, then the text is even more important. Textual descriptions are all you have to create a thrilling adventurous experience.

Probably the best-known company for creating strictly-text adventures is Infocom. They knew the value of creating a complete world in the imagination using only the power of the printed word. They did a superb job of describing the rooms of their adventures so that the player would create a mental picture that was extremely complete. You may not be able to approach their reputation for excellence, but it is a worthy goal to reach for.

Even Infocom eventually surrendered to the increasing popularity of graphic adventures, and finally began releasing their adventures with visual depictions of each location in addition to the text descriptions. If you choose to add graphics to your adventure, then it is imperative that the art work for each room match the text description you gave it. If your text describes a temple complete with altar, statue, and stone benches, then be sure your graphics include the altar, statue, and stone benches. And watch the small details. If your text describes a green slime on the wall, make sure the graphics don't show a yellow slime.

You may wish to have more than one graphic for each location. In the event you have a door which can be open or closed, it would be advisable to have two graphic pictures of the room, one showing the door open and one showing the door closed. Depending on whether the door is open or not, you would choose the appropriate graphic screen. Granted, this takes more work, but by planning your adventure ahead before you even sit down at the computer, the amount of work is lessened. And the effort will certainly be worthwhile. It will result in a nicer, more professional looking adventure game.

Check your grammer and spelling in the room descriptions

Two final things to watch carefully when describing your rooms are spelling and grammar. Nothing can be more distracting than poor grammar or spelling. "The only two things you can see is an altar and some stone benches." This is an example of one of the most common grammatical errors. The word "is" stands out as a grating error. It not only breaks the mood, but also gives your game an amateurish look. Similarly, spelling is important. Keep a dictionary handy by your computer, and use it whenever in doubt. If you are going to use exotic words and objects that you are unfamiliar with, look them up in a dictionary or encyclopedia first. It may take a little extra time, but it is time well spent.

When it comes to the room descriptions, there are a variety of things you must consider. Accuracy is important; historical accuracy as well as accuracy in depicting objects and room exits. Fill the room with well described objects that make the game richer. Use the the five senses in each room description to set the tone of the adventure. And check your spelling and grammar. Once you have finished all the room descriptions in your game, you will be ready for the next step.

You have now completed the room descriptions and the room connections. If someone were to play your game at this point, he could walk around in the world you have created, and read the room descriptions

or view the room graphics. But he couldn't do anything else. He could not see most of the objects, unless they happened to be included in the room description or graphic. And he certainly couldn't manipulate any objects at this point. That's the next step in creating your adventure. In the next two chapters you will learn about variables which affect all objects, and then begin adding the objects to your game.

Chapter 4: The Variables

Variables are an integral part of any adventure. They will not only be used in conjunction with objects but also with messages and program logic. So before we go any further with our discussion of creating adventure games, it is appropriate to spend some time examining the purpose and many varied uses of variables. This chapter will look at the different ways you can get the best use out of the variables in your adventure.

Variable Values

A numeric variable is a single letter or a short word that has a numerical value. For example, this line of Visionary code:

```
HOT := 10
```

sets a variable named HOT to a value of ten. Notice the symbol ":=" is the way Visionary assigns values to variables. Your adventure can assign that value, change the value, print the value, and even compare it to other values. Depending on the value of the variable, your game can then do different things. If the player says "eat the pie", your game can give different replies depending on the value of certain variables. The player might be told that "it tastes delicious" or "you burn your tongue and drop it" depending on the value of a variable.

A string variable is a variable that contains words or other alphabetical and numerical characters. For example, this line of Visionary code:

```
$MISTAKE := "You have made some error here."
```

sets a variable named $MISTAKE to stand for the sentence above. As with numeric variables, string variables can be assigned, changed, printed, and compared. Let's examine many of the ways that numeric and string variables can be used in your adventure.

Inventory Variables

The most common use of a variable is in the player's inventory limit. Generally in any adventure, the player is limited in the number of items that can be carried. As the game writer, you choose a variable name, say INVENTORY to stand for the number of objects currently in the player's inventory. When the player starts out, he carries nothing, so you set INVENTORY to zero at the start of the game. Each time the player picks up an object, you add one to INVENTORY (called **incrementing** the variable). And each time he drops an object, you subtract one from INVENTORY (**decrementing** the variable). Before

allowing the player to pick up an object, your game can check the variable to see if the player is carrying too much. If he is, you can refuse to let him pick up any more.

A built-in ITEMS variable is an easy way count the items carried

Visionary already has a built-in variable called **ITEMS** which keeps track of the number of objects in the player's inventory. It automatically increments and decrements as objects are picked up and dropped. All you have to do is decide on an inventory limit and compare it to ITEMS before allowing the player to pick up an object. The following example shows how this can be done with an axe.

```
IF ITEMS   6 THEN
   T You can't carry any more.
ELSE
   GRAB AXE
ENDIF
```

A related use of variables is to set the maximum number of objects allowed in the players inventory as a variable, not as a number. In the above example, the inventory limit is six. You could accomplish the same thing by replacing the constant "6" with a variable "MAXIMUM", as shown below.

```
IF ITEMS   MAXIMUM THEN
```

Then you would define MAXIMUM at the beginning of your adventure as being equal to six—this is usually done in the .ADV file. The above line will then act exactly as the previous one. The advantage to using a variable is that it makes it easier to modify the game later. If you change your mind, and want the maximum inventory to be five items, you have much less work to do. You need only change the one line at the beginning of your adventure where you set MAXIMUM to six. If you chose to use the constant "6" instead, then you would have to go back and change every occurrence of the "6" in the entire program. Using the variable saves you time.

Another reason to use a variable in the above example is that MAXIMUM can then be changed while the game is in progress. Perhaps the player finds a magic pill of strength. You could increase the value of MAXIMUM allowing him to carry more. Or perhaps he must carry less if he is weakened in a fight with a giant. In this case, you would decrease the value of MAXIMUM. By comparing the ITEMS variable with another variable like MAXIMUM, you can make your inventory limits more versatile.

In addition to using the reserved variable ITEMS, you may want to keep track of the player's inventory in a second variable as well. You could use it to differentiate between the items carried and the items worn. Let's say your adventure game allows the player to wear glasses, a hat, boots and gloves. It isn't logical to count these items as carried. If you did, then limiting the player to six objects would allow only two additional items to be carried. Logically, the glasses, hat, boots, and

gloves don't fill the player's arms when they are worn, so they should not be included in the inventory limit. If you define a variable WORN to be used together with the variable ITEMS, it would nicely solve the inventory problem. Remember that ITEMS is automatically incremented whenever an object is picked up or dropped. You, as the game writer, have to do nothing to ITEMS. However, whenever one of the four objects is put on, you would need to increment WORN, as shown below:

```
WORN := WORN + 1
```

And likewise, whenever one of the four objects was removed, the variable WORN would be decremented. To check for the inventory limit, you simply subtract the two variables and compare the result with the MAXIMUM allowed limit. An example of the Visionary code follows.

```
IF ITEMS - WORN  MAXIMUM THEN
 T You can't carry any more.
ELSE
 GRAB AXE
ENDIF
```

In the above example, let's suppose the player is wearing the boots, the hat, the glasses, and the gloves. Let's further suppose the player also carries a compass, some flint, and a steel knife. As each of the items was originally picked up, Visionary automatically incremented ITEMS. So at this point, the value of ITEMS is seven. As the hat, boots, glasses and gloves were each worn, you incremented WORN. That means WORN is now equal to four. In the above example, we subtract the value of WORN (4) from the value of ITEMS (7) and compare it with MAXIMUM. It means that the player actually carries three items (since the other four are worn). So when ITEMS - WORN is compared with MAXIMUM, the player is allowed to pick up the axe. By keeping track of a player's inventory in two variables ITEMS and WORN, you can allow a more sophisticated adventure, where objects that are worn don't add to the inventory limit.

There is one pitfall of which you need to be aware. When the player chooses to remove his hat, be sure to check MAXIMUM again. You wouldn't want to allow him to remove his hat if he already carried the maximum number of objects. To do so would leave him carrying more objects than your limit. Assuming this limit has not been exceeded, be sure to decrement WORN when the hat is removed, so to keep an accurate count of how many objects are still worn.

The State of Objects

Variables have many other uses in addition to inventory limits. You can use them to describe the state of an object. Let's say for example that the player carries a goblet. If she examines the goblet, she may see various things inside it. You can use a variable, say GOBLET, to tell

what it contains. If GOBLET is zero, then it is empty. If GOBLET is one, then it contains sand. If GOBLET is two, then it contains poisoned water. And if GOBLET is three, then it contains good water. When the player says "EXAMINE THE GOBLET" you can then give the correct description, depending on the value of the variable GOBLET.

A variable could be used to describe the state of a non-movable object, as well as a movable one. Perhaps the player stumbles across a large old treasure chest that can't be carried. You could define a variable, say CHEST, to describe the state of the chest. Zero could mean that the chest is locked shut. One could mean it is unlocked but shut. And two could mean that it is unlocked and open. Then if the player said "OPEN THE CHEST", you could give him the proper response depending on the value of the variable CHEST. If the value of CHEST is two, you could say "IT ALREADY IS." However, if the value is one, you would say "OK" and change the value of CHEST to two, indicating that the chest is now open. On the other hand, if CHEST equalled zero, you would reply "SORRY IT'S LOCKED." If the player simply asked to "LOOK AT THE CHEST" then you could accurately describe it as closed or open by checking the value of the variable.

Doors and open doorways can be easily checked using variables. A door can be open or closed, locked or unlocked. Using a variable, you can respond appropriately to any command the player makes. Similarly, you could have an open doorway for the player to pass through. Perhaps there is a magic spell that keeps the player from entering the next room. You could use a variable to determine if the spell had been cast yet, or if the spell had been removed. The player could be allowed or denied access to the room depending on the value of the variable.

The types of variables discussed above are called **attributes** in Visionary. They are special variables that can have only two values, zero and one. Attributes will be explained in more detail later in this chapter, but first let's look at other ways to use variables.

Counting Moves

Variables can be used to keep track of the player's turns. Frequently, you will want to keep a record of how many moves the player has taken. In a text game, a move is considered to be made any time the player types a command and the computer gives some response. In a graphic game, a move is considered to be made when the player clicks on a mouse button.

Even if the computer doesn't understand what the player wants to do, a move has been made. You may want to count these moves. Perhaps the player is merely curious to know how many moves have been taken. By creating a variable MOVES, you can increment it in the automatic logic section of your adventure (as part of a Non Player Character).

That way, each time a move is made, the variable MOVES is increased by one. The player can ask "HOW MANY MOVES HAVE I TAKEN", and your game can then print out the value of the MOVES variable.

Visionary provides a useful pre-defined variable called MOVES

The concept of counting moves is an important one in adventures. Visionary has reserved a special variable called **MOVES** that automatically keeps track of the number of moves a player has made. You as the game writer don't have to define the variable or increment it. It is automatically taken care of. When you wish in your adventure, the value of MOVES can be printed out for the player.

Many times, the adventure player will want to complete your game in the fewest moves as possible. At the end of the game, therefore, when the player has achieved the final goal and won, you should tell him how many turns it took to complete the adventure. This can frequently lead a player into trying your game a second time, even though he has already completed the adventure. He may want to try to beat his old score, and reach the goal in as few turns as possible.

Limiting Player Turns

There can be various reasons you might want to limit the number of turns in your game. Perhaps the player must escape from a giant alien maze within three hours. If you choose to count each turn as one minute, you will allow the player 180 moves in which to complete the adventure. By checking the MOVES variable, you can stop the game after 180 moves, and declare that the player has lost.

Another reason that you might want to limit the number of turns in your game, is that it can permit a potential publisher to sample your program without being able to complete the game. If when you have finished your adventure you decide to send a copy to a publisher for approval, you might feel more secure if you do not send a complete working version of your game until you have signed a contract. That way if for some reason your program falls into the wrong hands, a pirated version of the full game will not be released. Your game could only be played for, say, 100 turns, before it stops with some message advising the player that it is a submission copy, not an officially licensed one.

You might also use this technique if you decide to release your game as "share-ware". You could then release into public domain copies of your completed game that would only allow 200 turns. An announcement at the beginning of the game would caution players that the share-ware version of the game is a sample of the full and complete game, and that the game cannot be completed in 200 moves. After 200 moves, the game would stop and tell the player that the complete game can be obtained by sending you an appropriate fee for a fully working version. In this way the player can get a taste of your game and decide if he wants to purchase a share-ware copy from you.

You should be sure to reset variables whenever appropriate

There are other reasons to keep track of the number of player's moves in your game. You can use a variable to count the turns backwards, from some set amount down to zero. For example, if your adventure included a desert, you could use a variable to keep track of the turns that the player spends out in the desert until she dies from thirst. You could define a variable THIRST which starts with a value of ten. Whenever the player is in a desert location, you would decrement THIRST. If it reaches zero, then you stop the game and pronounce that the player has died from thirst. If the player drinks from the canteen, then you would reset THIRST back to ten. If the player leaves the desert, you would also reset THIRST to ten. If you didn't reset THIRST, it would be conceivable that the player could leave the city and enter the desert for nine moves, then re-enter the city. When leaving the city the second time, she would die immediately upon entering the desert. To prevent such things from happening, always check to see which variables need to be reset.

As mentioned in the chapter on creating an adventure plot, if you plan on permitting your player to die from thirst, be sure you give adequate warnings. Never kill the player abruptly without advance notice. Use variables to accomplish this. For example, when THIRST equals five, you might tell the player, "You are getting thirsty." Then again when THIRST equals three, "You are dying of thirst." And at one, "You are about to expire from thirst." These warnings give the player plenty of time to drink from his canteen. And in case he has forgotten to take the canteen into the desert with him, always give the first warning in time for him to return to safety, if he acts promptly.

Keeping Count

Using variables that count backwards can be a useful technique in constructing your adventure. You might wish to count the turns until a magic potion wears off. Or you might wish to keep track of the number of moves until the cannibals arrive on the island. Perhaps you will allow the player to stay in a room full of crocodiles for only five turns before they attack. Or maybe the player dives under water, and can only hold her breath for eight turns. You will find many ways to use variables that decrement. They can be as varied as the adventures you will create.

Non-Player Character Section and Variables

Variables are not automatically incremented or decremented, except for the reserved variables like MOVES that are built into Visionary. The others, you must deal with yourself. You should place the programming code for these variables in the Non-Player Character portion of your adventure. This section is executed after every move the player makes.

It doesn't matter what the player says or does, this section of the program is always executed. It is here that you should check to see if the player is in the desert, and if necessary, decrement THIRST. It is here that you should check to see if a player is under water, and if so, decrement AIR. And it is here that you must check the variables after decrementing them, to see if the player has died and if the adventure is over.

You may wish to increment a variable to act as a clock. The concept is similar to decrementing a variable as described above. But in this case, you will constantly increment the variable and use it to tell time. If the player finds an old pocket watch, he can look at it and tell the time. Let's say you choose a variable called TIME. When the player says "LOOK AT THE WATCH" your adventure can tell him the current time of day. Just be sure you don't allow the clock to run past 12. Remember to reset your variable to one, when it reaches thirteen. You could even define a second variable to keep track of morning or afternoon.

There are more reasons to use a clock than strictly cosmetic ones. Of course, it is nice to have a watch that the player can use to tell time. But you can also use the time feature to cause specific things to happen in your game. Perhaps the ghosts disappear at dawn. Or the tower clock strikes at midnight. Maybe the player won't die of thirst in the desert if he travels through it at night. As you can see, a time variable or two can be used in a variety of ways.

Keeping Score

Not all variables should increment or decrement on every turn. You will want some variables to change only after certain actions by the player. For example, a variable could be used to keep track of the player's score. Let's say that every time the player achieves some smaller goal on his way to the final one, the score is increased. Maybe ten points are awarded for every new room that is visited. Or the player may get 50 points for each treasure found.

As the game writer, you define a variable at the beginning of the game, such as SCORE. When the game starts, it has a value of zero. Each time the player finds a treasure, a certain number of points is added to SCORE. You might wish to have some treasures worth more points than others. At any time, if the player asks for the SCORE, your game can announce how many points have been earned.

Some adventures restrict where the SCORE can be given. You might want the player to be standing in a certain location before he can find out his score. In Scott Adams' classic *AdventureLand*, the player could only find his score when he stood in the root cellar. The treasures weren't added to his score if they were carried. They had to be

dropped in the room. A sign in the root cellar said, "Leave treasures here and say SCORE."

Let the player know how to find out the SCORE

You could do something similar in your game. But if you do, be sure to let the player know what is expected. Somehow, let him know what he has to do to get his score. A frequent device is a sign, such as the one in the root cellar of *AdventureLand*. It tells the player exactly what to do in order to gain points. In the above example, he had to drop them in the root cellar. Carrying them in the root cellar didn't count. To find out his score, he then had to say the word "SCORE." Feel free to vary this in any ways that you wish. Perhaps you won't require the player to be in any certain room to increase his score; the score could increment as soon as a treasure is found. Perhaps you want him to ask for his score in a different way, say by talking to a magic mirror.

Whatever you choose, make sure the player knows what you expect. This could be done with a sign. A genii could pop out of a bottle and tell the player the proper way to determine his score. You could simply put the information in an announcement at the beginning of your adventure. Just be sure the player knows.

Keeping Track

A variable can be used to keep track of the bullets left in a gun. Let's say the player finds a loaded revolver. She can use it to kill jungle animals, or perhaps signal someone far off. But you can't let her keep shooting the gun forever, when logically it should eventually run out of bullets. Using a variable like BULLETS, you could set the number of BULLETS to six when the gun is first found. Each time the player gives the command to shoot something, your adventure should check to make sure the player holds the gun, and that BULLETS is not zero. You then print some appropriate message such as "BANG" and decrement BULLETS. If BULLETS equalled zero, you would print some alternate message telling the player, "YOU ARE OUT OF BULLETS."

Depending on the reason for shooting the gun, you might want to use further variables here. If, for example, the player was shooting at crocodiles, you might want to change some variable that keeps track of the number of crocodiles still alive. Or you might wish to reset a variable that was being decremented, counting the turns the player had left with the crocodiles before they ate her.

Variables can also be used to keep track of the weight of the player's inventory. In many adventures, the player has a certain limit to the number of objects he can carry. But it doesn't matter how much they weigh. If the player can carry a maximum of six objects, they could all be light objects or all be heavy objects. This is not the most realistic method of limiting the player's inventory. An alternative method is to

assign weight to each object. In this way, you can allow the player to carry a maximum number of pounds, not objects.

If you choose this method, you can no longer use the built-in variable ITEMS. But you could easily define a variable WEIGHT to keep track of the weight that the player carries. You could assign each object a number, which indicates its weight. Before allowing the player to pick up an object, you should see if it would make the player's total weight exceed your chosen maximum.

As mentioned earlier, if you use a variable like MAXIMUM, instead of a constant like 50, then the maximum weight can be changed in the middle of the game. The following Visionary code shows how the program might look if the player wanted to pick up an axe weighing ten pounds.

```
IF WEIGHT + 10  MAXIMUM THEN
 T You can't carry that much weight.
ELSE
 GRAB AXE
 WEIGHT := WEIGHT + 10
ENDIF
```

Let's see how it works. When the player gives the command to pick up the axe, the axe's weight is temporarily added to the weight the player already carries. If the new temporary total exceeds the maximum weight allowed, the player is prohibited from picking up the axe. If not, the player is allowed to take the axe. Not only is this more realistic, it also prevents the player from putting objects in a back pack or a suitcase, which would allow more to be carried. Putting objects inside other objects is a common way for players to carry more than the maximum number of objects. If you wish to prohibit this, assigning weight to all objects is an excellent method.

Handle Random Events

Another use for variables is to provide random messages or sound effects to the player as the game progresses. A variable can be set to some random number, rather than some specific value. If it is then decremented each turn, a message or sound effect of your choice can be given when it reaches one.

For example, sounds and messages that help set the stage and help create a mood can be set randomly. After a random number of turns, a text game can say things like, "A bird chirps in the distance." A graphic game could actually play the sound of a bird chirping after that random number of turns. Every once in a while the player may hear, "the wind as it whips through the tree tops."

These types of random sounds and messages really enhance an adventure. They add realism and depth to your game. They shouldn't always occur at the same time and in the same place. Set a variable at ran-

dom, and decrement it on each turn. When it reaches one, the message is printed or the sound is played. You don't want the message to be printed when the variable is zero, or the message or sound will continue to be given on every successive move, since the variable will stay at zero. By presenting it when the variable is one, it will only appear once. You can then either let the variable remain at zero, or reset it for another time.

Variables can not only control random messages and sounds, they can also control random events. You may wish to have a rain shower occur randomly. In that case, you must decide what the random parameters are (say somewhere between 25 and 35) and assign the proper value to your variable. The following example shows how this can be accomplished in Visionary.

```
RAIN := RANDOM 10
RAIN := RAIN + 25
```

This assigns a random number between zero and ten to RAIN, then adds 25 to the result. In the Non Player Character files (the ones that execute after every move) you can decrement RAIN until it reaches one, at which time you create the rain shower for the player. In a text game, you could simply print a message telling the player that it is raining. In a graphic game, you could overlay some rain clouds and falling rain over the current picture. You could even add some sound effects, including thunder and splashing raindrops. You might wish to write your adventure so that RAIN is only decremented when the player is outdoors, ensuring that it won't rain when he is indoors (it rarely rains indoors).

Flags

Let's examine some special variables called flags. Generally, a flag is a variable that has only two values, zero or one. A flag can be set to one or "unset" back to zero. A flag can tell you if the player has done something. It can signal if the player has dug in the sand. A flag can indicate an event has occurred. It can indicate if a magician has placed curse on the player. A flag can also describe the state of some object. It can tell you whether a door is open or closed. Even though flags may be limited to only two values, on or off, set or unset, zero or one, yes or no, they are extremely valuable in any adventure.

In Visionary, there are 32 flags assigned to each object and to each room. These flags are called **attributes.** If you have an object called DOOR, you could name one of the attributes CLOSED. Each attribute can be set (meaning yes) or unset (meaning no). If the attribute CLOSED is set, then the door is considered to be closed. If the attribute is unset, then the door is considered to be open. By using these flags, you can tell if the player is permitted to travel through the doorway or not. If you use a second flag named LOCKED, you can make

sure the door is not LOCKED when the player attempts to open it. As discussed earlier, this same logic can be duplicated with a single variable that has three values. Using two attributes instead, is simply another way of accomplishing the same thing.

Two-valued attributes like DARK and VISITED are called flags. You can add your flags quite easily

One of the most practical uses of flags is to tell if a room is dark or not. If a room is dark and the player doesn't carry any source of light, then you don't want him to see the room, or see any of the objects in the room. In a text game, you want the normal room description to be replaced with something like "YOU CAN'T SEE ANYTHING IN THE DARK." In a graphics game, a simple black screen can suffice. On the other hand, if the room is not dark, or if the player carries a lit torch, you want the game to show the location and the objects. This is accomplished with a flag. It is such a useful flag that Visionary has built it into each room as an attribute. Each room has an attribute named **DARK** that starts unset (in other words not dark) unless the you specify otherwise. You can always control whether the room is dark or not, and by appropriate programming, you can control what the player can see under any given circumstances.

Another use for a flag is to tell if a room has been visited or not. In this way, a text adventure game can give a full description of the location when it is first visited, and give a shortened description thereafter. Visionary has also included this flag as one of the attributes available for each room. The attribute is named **VISITED**.

If it is set (in other words, if the room has already been visited) you can give the player a brief reminder of what room he is in. If it is unset, then the entire room description is given and the attribute is set. If the player says "LOOK", the full room description is given again. The Visionary program automatically takes care of this by unsetting the VISITED attribute temporarily so that the player is given the full room description.

In addition to the 32 attributes available for each object and room, you can create as many more flags as you wish. The only difference is that attributes are defined along with the object or room definitions, while the other flags must be defined at the beginning of the adventure. Just remember to treat a flag as any variable, and give it a name and a value. Let's examine some different uses of flags.

Let's say in your adventure, a dead body lies on an autopsy table. The abdomen has been opened, allowing the player to look inside the corpse. Flags can be used to indicate which organs remain inside. You might have three flags, HEART, LIVER, and SPLEEN. If the player looks inside the corpse, your game could describe the organs as present if the appropriate flags were set. If the player chooses to remove one of the organs, then the associated flag is unset. This assures that when the body is described, the missing organs will not be described too.

Sometimes, picking up an object can set a chain of events in motion. A flag can be used to make sure the chain of events does not occur a second time. Let's say for example that you want Moby Dick to appear far out to sea, after your game has been played for a while. But you want to make sure the player has found the harpoon and has the means to kill the whale before you make it appear.

The easiest way is to make the whale appear in the ocean when the player picks up the harpoon on the ship. But if the player succeeds in killing the whale, you don't want it to appear a second time if the player should happen to drop the harpoon and pick it up again. That's where a flag is used. Create a flag called WHALE. If the flag is unset (meaning the whale has not appeared) then create the whale and set the flag, when the player gets the harpoon. If the player should ever pick up the harpoon again in the game, the flag would be found set and the whale would not be created a second time.

If dropping an object causes something to happen, check a flag first. If dropping a sacred skull in the pygmy village causes the natives to run scared and disappear into the jungle, then set a flag. You don't want the same thing to occur if the player should pick up the skull and drop it a second time. It would not make sense to tell the player that the pygmies run scared again, since the pygmies had already left. By checking a flag to make sure the skull had not been dropped yet, you can make sure the appropriate message appears only once.

Digging in the ground and finding treasures requires the use of flags. Perhaps there is a magic ring buried in the sand dunes. When the player digs in the dunes, you want the ring to appear—but only if it is still buried. You don't want the ring to appear if it has already been found and has been used.

The easiest way to deal with this is to give an attribute to the ring called FOUND. If the player says "dig" and the ring is FOUND, your game will respond "You find nothing." Otherwise, it will reply "You find a ring in the sand." In this case, be sure you not only place the ring at the current location, but also set the ring's FOUND attribute so that it can't be found a second time.

You can even have something happen only after you have visited a certain location. Let's say that after the player has visited the grave in the far corner of the cemetery, a wolf appears at the gate to the cemetery. He is required to kill the wolf. Use a flag to prevent the wolf from appearing every time he visits the grave. Once it has been killed, you don't want it to continue to appear whenever he visits that grave.

Flags are especially useful in giving the player "help". When the player types "help", you want to present him with some clue to help him get past the current puzzle. You may wish to have several helps for the same problem. Flags will tell you which clues have been used, and which ones are available.

Flags can also be used to keep track of previous player actions

Perhaps the player is stuck trying to open a safe. If he asks for "help", you may wish to respond "You have held the combination in your hands." But the question is, has the player really held the combination? You can't know that for sure, unless you use a flag. If the combination was written in invisible ink on a piece of notebook paper, then set a flag if the player picks up that paper. If there is a later request for help, you can check the flag and give that clue if the flag is set. Perhaps you want to give a second clue, if the first one is not sufficient. Use another flag. When the first clue is given, set the flag. Then when the request for "help" is made a second time, check the flag. If the flag is set, don't give the first clue. Give the second clue instead.

By setting flags after certain events have happened, you can give more appropriate clues to the player. Let's say that one of the first puzzles the player must solve is to figure out how to get inside a house. Perhaps there is little she can do until she is inside, but the door is locked. When the player asks for "help", you want to give her a clue such as "check inside the envelope." Once she has found the key and entered the house, this clue is longer of any value.

Perhaps the player will ask for a hint on what to do next—for example, you may want to advise him to "visit the kitchen." Use a flag to determine which clue to give. Once the front door has been unlocked and opened, set a flag. When the player asks for help, check that flag. If it is not set, then you know he has not yet entered the house, and you should give the first clue. If on the other hand, the flag is set, then you know he has solved the first puzzle, and you should skip the first clue and give him the second one instead. If you use flags for each of the major turning points in your adventure story, you will always be able to give the appropriate hints.

There are many other uses for variables, flags and attributes. They are only limited by your imagination. They provide a vital method of checking program logic to prevent things happening that you don't want to happen, and to cause other things to happen that you want to occur. They can prevent the player from traveling in certain directions, and from seeing things you don't want him to see. They can cause messages to appear at random, and objects to appear as the result of the player's actions. As you write your adventure, you will use many variables. You won't be able to avoid it.

Chapter 5: The Objects

As you write your adventure, the next step is to describe the objects. Some of the objects have already been described, as they were part of the location descriptions. But for the most part, the objects in your game have yet to be created.

Object Types

Objects can be manipulated in various ways. A sword can be picked up. A candy bar can be eaten. A hammer can pull nails. A door can be opened. A boulder can be rolled aside.

Not all objects can be picked up by the player. Some, like the door, are stationary, but can still be manipulated. Others, like the boulder, are too heavy to carry. But the one thing all objects have in common is that they can be used. The player can do something to them. They can be picked up, dropped, examined, or used in a variety of other ways.

There are two basic types of objects in adventure games, movable objects and nonmovable objects. **Movable objects** can be picked up by the player and carried along. **Nonmovable objects** cannot be picked up by the player, but may be manipulated in other ways.

A hammer is a good example of a movable object. The player can pick it up, and put it down. He can carry it with him from one location to another. He can manipulate it in various ways. He can pull nails with it. He can break a window with it. He might even find it to be a magic hammer that sings when stroked. Whatever its uses, the hammer is considered to be a movable object because it can be moved from one location to another.

A giant boulder is an example of a nonmovable object. Although the boulder cannot be carried by the player, it is still considered an object, because it is part of the game that the player can refer to. It can be manipulated. The player can examine it. She can write on it. She can climb it. Perhaps she can even roll it aside, to reveal a treasure beneath it. The only functional difference between the boulder and the hammer is that the boulder cannot be picked up by the player and the hammer can.

When you design movable objects for your adventure, you should consider an inventory limit. Perhaps you will place no limit on the number of objects that the player can carry. Most adventures, however, have some inventory limit to make solving the puzzles more of a challenge.

As mentioned in the previous chapter, there are two basic ways of limiting a player's inventory. You can allow no more than some set

number of objects to be in the inventory at any one time. Or you can limit the total weight that the player can carry. If you choose the latter, you must choose the weight of each movable object in your game. You must decide if you will limit the player's inventory, and if so in which manner. This decision must be made before you start programming the objects into your game.

As the player attempts to pick up each object, you must evaluate if the limit has been exceeded. Insert a routine to perform this evaluation, into the action section of each object. An example of a Visionary routine to do this given in Chapter 4.

The Object Files

In Visionary the objects, both movable and nonmovable, are listed together in one or more text files that you create. These files must contain the description of each object, as well as the name and any attributes you wish to assign to the object. Some objects may need no attributes. But most will require at least one attribute.

For example, if a pearl lies locked inside a box, you would create an object named "pearl". You could give it an attribute called FOUND. Before the player has opened the box, the pearl is not in the room with him. It is stored in the storage room. The concept of a storage room was mentioned in the previous chapter on rooms. When the player first opens the box, the pearl is moved to the current room and the attribute FOUND is set. In that way, if the player should close the box and open it again, he would not find another pearl. You accomplish this by checking the flag each time the box is opened, and only creating the pearl if the flag is unset.

A sample piece of Visionary code shows how this routine could be included in the file for the box.

```
ACTION OPEN
IF PEARL NOT FOUND THEN
   PLACEOBJ PEARL, THISROOM
   T A pearl drops out of the box.
   SET PEARL, FOUND
ELSE
   T It is open and empty.
ENDIF
ENDACT
```

Another common attribute you will give to many objects is WORN. Anything like a hat, shoes, a wrist watch, glasses, a ring, a necklace or gloves can be worn. You want the player to be able to wear these items, since it would be illogical to prohibit it. Yet when you give a description of the object, you want that description to indicate if the item is worn or not.

Let's say the player has found a set of ear plugs. When she takes inventory, she should see "a pair of ear plugs." After she has put them in her ears, the inventory should show "a pair of ear plugs worn securely in your ears." When the player puts the ear plugs in her ears, you should set the attribute WORN. Then when you write the inventory descriptions, be sure that you check the WORN attribute and print the correct description. These inventory descriptions are included in the code block for the object PLUGS, as shown below:

```
CODE
 IF PLUGS WORN THEN
  T a pair of ear plugs worn securely in your
    ears
 ELSE

  T a pair of ear plugs
 ENDIF
ENDCODE
```

In addition to the object descriptions, the WORN attribute will be used at other times in your adventure. Perhaps wearing the ear plugs is the only thing that can save the player from the seductive song of the mermaids. When the player gets to that part of the story, you need only check the WORN attribute for the ear plugs to see if he is drawn to his doom or not.

The object files also contain any adjectives that describe the objects. This allows the player to say "look at the old photograph" rather than "look at the photograph". The difference may not seem important to you now, but it adds a touch more depth and realism to your adventure.

A second important reason to use adjectives for your objects is that these adjectives differentiate between objects that have the same name. Perhaps there are two photographs, an old one and a new one. If the player holds both of them, he can examine either by specifying the adjective. He can say, "look at the new photograph," or "look at the old photograph." To give another example, you might have two boxes placed in a room, one a red box and the other green. You can tell which one the player wants to open by checking the adjective. If he asks to "open the box", Visionary will ask him to be more specific, since there are two different boxes. The adjectives "red" and "green" allow the player to specify which box he means.

Objects in Multiple "Roles"

There is an useful variation on the above situation, if the objects being described are nonmovable. Let's say the boxes are large and nailed down. If they are in different rooms, you might want the player to be able to say "open the box" without receiving a request to be more specific. It's logical that if there is only one box in the room, then the player should not have to say "open the red box."

A simple way around this problem is to create only one object named box. When the player enters the first room, place the object "box" in that room. Since there is only one box in the adventure, the command "open the box" will not require using an adjective "red." Visionary will not be unsure which box is meant, since there is only one box in the game. When the player moves to the second room containing the other box, you will place the object "box" in the second room. To the player, it will appear to be a second and separate box. But to your program, it will be the same box. The player can once again say "open the box" instead of "open the green box." Visionary will accept this command, since there is only one box.

If you choose this alternate method of having a object playing two roles, eliminating the need for different adjectives, you are then presented with additional problems. You may need to keep track of which box is locked and which is unlocked, or which is open and which is closed. Maybe it's important to keep track of which box is empty, which contains a pearl and which contains a ruby.

The solution to these problems lies in using attributes. Give the box various attributes such as RED-LOCKED, GREEN-LOCKED, RED-OPEN, GREEN-OPEN, RED-EMPTY, and GREEN-EMPTY. If the player asks to open the box, determine which box he means by checking which room he is in, and then examine the appropriate attribute RED-LOCKED or GREEN-LOCKED to see if he can open it.

If he asks to "look inside the box", you again determine which room he is in when he made the request. If he is in the room with the red box, check the attribute RED-EMPTY. If the attribute is set, tell the player that "the box is empty." Otherwise, tell him that "it contains a giant pearl." Although it sounds as though it takes a lot of extra work to avoid the necessity of using an adjective, it actually isn't. You still would need the same type of attributes and would have to write similar amounts of code, whether you used two separate boxes with adjectives to differentiate them, or a single box which will be moved from one room to another.

The concept of placing a nonmovable object in several different rooms can be used for other purposes as well. Let's say in your adventure, there is a stretch of beach along which the player can explore. You may have several different locations, all at the water's edge. You might anticipate that the player would try to "drink the water" or "get some water" or "swim in the ocean" at the various locations. An easy way to permit this, is to create a single object called "water". Give it some synonyms, like "ocean" and "sea". Then move that object to each of the water front locations as the player moves into them. This is done by placing the following line into the code block of each room file.

```
PLACEOBJ WATER, THISROOM
```

When the player enters the room, the water is automatically placed there too. The player can then try to get some water, drink the water, or swim in the water. You can anticipate those requests and program the appropriate responses into the object file for WATER.

Let's look at another example of where this technique comes in handy. Let's say you have a thirty room adventure that takes place out in the desert. The player may want to "look at the sky" to see if it looks like rain, or to see if the buzzards have returned to circle above him. "Sky" must be an object, or the player will not be able to look at it. An easy way to deal with this is to create one object called "sky" and to move it to the player's current room. Simply put the following line into the code block of each room file:

```
PLACEOBJ SKY, THISROOM
```

When the player asks to look at the sky, this object file is executed and the appropriate description is displayed. Attributes can be checked and the description of clouds, rain, or buzzards can be given if they exist.

It's to your advantage to place the movable objects in a separate file from the nonmovable objects. If there is an excessive amount of objects, you may want to split the file into several smaller files. But it is still advisable to keep the movable objects together in files separate from the nonmovable objects.

There are several reasons for keeping the two types of objects in separate files. First, it makes each object easier to find when you need to go back later and make modifications to it. You will frequently need to change parts of an object file. Perhaps you discover the need to add an attribute to an object. Or you may decide to change an adjective. If you have kept the nonmovable objects separate from the movable ones, it speeds up the search for the object you want to find.

Object Descriptions

In a text adventure, there are three times that an object is described. One is when the player first sees the object after entering a room. When he first enters a room, a full complete description of the room is given followed by any visible objects. He might be told that "you see a brass key lying on the floor."

The second time an object is described is when the player asks for "inventory". This is usually a brief description, unlike the first one. It usually consists of a few words, like "a brass key."

The third time a player will see a description of the object is when he specifically asks to "examine the key." Usually this is the most detailed description of the three. The player might be told that "the key is small and ornate, the brass is tarnished." In some cases, the player may see a further description. If, for example, he holds a letter, he can

"read the letter" to receive a further description. He might be given as much as several paragraphs to read.

When you create the file for each object, you decide on the different kinds of descriptions. If the player has the object, then give the short description that will only be seen in an inventory list. If the player does not have the object in his possession, then give the full sentence description that he will see when first entering the room. These two descriptions must be included in the CODE section of the object file. The third, longer description would be placed in the ACTION section of the file. The player will be given the longer description, only if he asks to "examine" the object.

Build Your World With Words

Always remember that in your text adventure game, you are building a world with words. Even when you write a graphics adventure, you will usually want to describe objects with words, since the picture alone may not be sufficient. For example, a picture of a bottle doesn't tell the player as much as a description like "the brittle glass bottle is empty."

Even graphic adventures will frequently rely on word descriptions. When you describe the objects in your world, make sure you bring them to life with your words. Spend some time carefully describing each object. When designing an inventory description, you want to keep it short. Limit it to one line on the video screen.

But choose your words carefully so that the player will feel the mood and tone of this new world. Don't just say "a strange old knife." You could call it "an antique knife with an ornately carved handle." When designing an object description to be seen when the player first enters a room, again try to limit yourself to a single line on the video screen. It will look more aesthetically pleasing if, when there are several objects in the room, the object descriptions don't exceed one line.

However, when giving a full complete description of the object after the player requests to "look at the knife," then don't feel the need to limit yourself. Give a full detailed description of the knife. The game is going to be a lot more exciting and fun to play if the object descriptions are more complete.

When designing the description for each object, keep in mind both the difficulty level of your game and its target audience. If you have designed your adventure to be for the beginner, your object descriptions may be bolder and more obvious. A futuristic ray gun might be described as, "a sturdy ray gun with a large trigger just begging to be pulled." If instead the game is designed for an experienced adventure player, the objects can be described differently, in a more subtle manner. The same ray gun could be "a long metal rod with a small stud on one end."

Likewise, if your game is intended for more knowledgeable adults, you may not have to describe an object as closely as if it were intended for younger adventurers. In an adventure targeted at adults, an object

could be simply described as "an old scimitar." Objects like these may not be familiar to younger players, so you might want to change the description to make the nature of a scimitar more obvious. You could describe it as, "a sharp old scimitar, its wide curved blade gleaming." What you have done is to change the description of the object, depending on the experience of the target audience. Always keep this in mind as you create your object descriptions.

The "Nothing" Variable

When the player takes "inventory," your adventure game will list all the objects she carries. If she doesn't carry anything, it will not list anything. To make this look a bit nicer, make one movable object called 00NOTHING. Place it in the player's inventory, but don't allow her to drop it. When the player asks for the inventory, check the ITEMS variable. If it is equal to one, then the only thing the player carries is 00NOTHING. In that case, use an object description that says, "You carry nothing."

If the variable ITEMS is greater than one (indicating the player carries something else), then use an object description that says "You are carrying:". Since 00NOTHING starts with a double zero, it will come alphabetically before any of the other movable objects. Visionary always lists objects alphabetically, hence the words "You are carrying:" will appear first above the other items in the player's inventory. By creating 00NOTHING as the first object, you have made the inventory listing look more attractive. The object 00NOTHING serves no other purpose. The player cannot get it, drop it, examine it, or use it in any fashion. It's only there to make the inventory listing look better.

Allow Object Manipulation

When writing your adventure, you should allow the player to manipulate each object in all the obvious ways, as well as any special ways that you have designed. The obvious ways of manipulating an object include examining it, taking it, and dropping it.

The player should further be allowed to use the object in any other obvious ways. Let's say the player finds a candy bar. He must be allowed to pick it up, assuming he has not reached his inventory limit. He must also be allowed to drop it. That may sound obvious, but it can be easily overlooked. The player must be allowed to examine the candy bar. He needs to know if it "looks fresh and tasty" or if it "is old and stale."

In addition to these three actions, which you should allow for every movable object, the player should be allowed to use the object in any appropriate manner. In the case of the candy bar, he should be allowed to eat it, feed it to a goat, or use it as bait. Even if these actions serve no purpose in the solution of your adventure, they should be allowed. Otherwise, your adventure will seem too limited and boring. A guideline to follow is that if you have placed something in your adven-

ture, then allow the player to manipulate it. If you aren't going to allow him to manipulate it, then don't put it in the game.

Let the Player Examine Objects

Always allow the objects to be examined. This includes both movable objects as well as nonmovable objects. When describing an object at the player's request, it is an opportune time to give the player some hints. If the player has asked to "look at the boulder", you could respond that "it is quite large and heavy." If you want the player to discover something sitting on top of the boulder, you could go further and give the player a hint by saying, "it looks like you could climb on top of it." If there is something beneath the boulder, you could tell him, "It is much too heavy to carry, but you might be able to roll it aside."

Remember that when the player asks for a longer description of an object, you are given an excellent opportunity to nudge the player in the right direction.

Let Movable Objects Be Picked Up

Always allow the movable objects to be picked up, even if the object is simply part of the "window dressing." Sometimes you will place an object in your adventure, not because it serves any purpose in the ultimate solution, but rather because it is appropriate to find the object at some location.

For example, let's say the player has entered an old prospector's shack. You would not want the shack completely empty. It would seem appropriate to have a few objects inside, like an old lantern on the table, or a blackened fireplace poker on the hearth. The lantern might play an important part in the adventure, being instrumental in the solution to some puzzle later. The fireplace poker, on the other hand, might be strictly there to give the shack a richer tone. But you must allow both objects to be picked up. Even though the poker is unimportant to the solution of the adventure, it is part of the game and you must allow the player to acquire it.

Let Movable Objects Be Dropped

Always allow the movable objects to be dropped. Even when, for the best solution of the puzzle, an object should not be dropped, allow it to be dropped anyway. Perhaps you have left a pile of clothes on the floor for the player to find. You want them to be found, picked up, and put on hangers. Dropping the clothes back on the floor really has no place in the final solution to your adventure. But it should be permitted anyway. If you don't permit the player to drop the clothes, it only makes your adventure easier to solve, because the only other alternative is to hang them up. However, if you do allow the clothes to be

dropped, then you not only cloud the solution to the puzzle a bit more, but you make the world of your adventure a little more realistic. A general rule of thumb is, if an object is movable, always allow it to be dropped.

Anticipate the Action of Play

There are many ways for objects to be manipulated in addition to being examined, picked up, and dropped. Try to anticipate what the player will try, and create actions for each of them. If the player has a bottle, anticipate he will try to open it, put things inside it, seal it closed again, or even break it. Allow the player to manipulate the object in all the normal ways. Let him open the bottle, even if it is empty. Let him put a rolled-up note in the bottle, even if it has nothing whatsoever to do with the solution of your adventure. Give him the means to seal it back up again. Allow him break it, or at least try to break it. You must allow him to manipulate the bottle in all the normal ways.

If you don't want the player to manipulate an object in some normal expected fashion, then you must devise some logical reason to prevent it. If the player has found a glass bottle, it is logical to expect him to try to break it. If you wish to prevent this, you must find some reason why he can't succeed. In a poorly-designed adventure, if the player tried to break the bottle, he would receive a response, "You can't." But in a well-designed adventure, you will tell him the reason that he can't. You may reply that "You don't have anything hard enough to break the bottle." Or you could simply say, "You don't feel destructive at the moment." Either way, you have done more than just deny the player the ability to break the bottle. You have explained why he can't break it.

Don't be concerned about the player trying unusual or illogical things. You needn't anticipate he will try to eat the bottle. You don't have to include anything in your game to prevent him from eating the bottle. Feel free to ignore the more outlandish actions. Just be sure you don't overlook any of the more obvious and logical ones.

On the other hand, if you want the player to actually eat the bottle, then provide the necessary programming code to accommodate that action. But at the same time, be sure you let the player know that she is expected to eat the glass bottle. Don't expect her to try such an unusual and bizarre action on her own. If you want her to eat it, give some clue. This can be done in a "help" message, or in the object description. If she asks for help, you could respond, "You are so hungry, you could eat glass." If she asks to examine the bottle, you could reply that "it appears to be a Hollywood stunt prop made of sugar." The important thing to remember is that if you require the player to treat an object in an unusual way, you must lead her to do it, not just expect her to try it on his own.

Let Objects Be Put Inside Other Objects

As mentioned above, you should allow objects to be put inside other objects. Allow the player to put a note inside the bottle. Require him to put batteries in flashlight before he can use it. To hear a recording, make him put a cassette in a recorder and then push "play." To keep a gun dry, he should put it in a plastic bag. The player will feel a much truer sense of being actually able to manipulate things in your make-believe world, when he is allowed to put some things inside others.

When you allow objects to be placed inside other objects, be sure to set flags, so you can determine the correct descriptions to give. For example, if the player asked to "examine the flashlight", you would want to respond that either "it needs batteries" or "it contains batteries." By using an attribute named FULL for the flashlight, you could easily give the proper response depending on whether the flag was set (and the flashlight is full of batteries) or if the flag was unset (and the flashlight was not full of batteries). Look at the following lines that produce this result:

```
ACTION LOOK, EXAMINE, SEARCH
 IF FLASHLIGHT IS FULL THEN
  T It contains fresh batteries.
 ELSE
  T Bad luck... no batteries!
 ENDIF
ENDACT
```

Likewise, if the player tried to turn on the flashlight, you would either tell him he was successful if the flag was set, or tell him that nothing happened if the flag was unset. When you permit objects to be placed inside other objects, use flags keep track of their status.

If you don't allow all the objects in your adventure to be manipulated, regardless of their importance, then the player will soon learn to disregard any objects that can't be manipulated in most normal fashions. This will make the puzzles in your adventure much less challenging, and the game much less interesting to play. By adding non-essential objects to the game and allowing the player to manipulate them, you make the puzzles more difficult. The player will not be able to tell the important objects from the unimportant ones.

Some Tips for Using Objects

When you create objects in your adventure, it is important to include objects that are nonmovable as well as the more common movable variety. One reason you should include them is that they can change in some way. A door, for example, is a nonmovable object. The player cannot pick it up and take it along. But it still needs to be included as an object, since it can change from an open door to a closed door. By making the door an object, the player can examine it, and see if it is

open or closed. The most common way to accomplish this is with an attribute. Name an attribute for the door, called OPEN. If the attribute is set, tell the player that the door is open, and allow it to be used as an opening. If it is unset, tell him it is closed, and deny him access.

Another reason to include nonmovable objects is if you anticipate that they will be referred to by the player. Take for example the bottle. If the player said, "drop the bottle" then Visionary would be able to respond properly and place the bottle at the current location. If on the other hand, the player said "drop the bottle on the floor," Visionary would not be able to understand it.

The reason is the word "floor." Your Visionary game would not be able to understand that sentence unless you had defined FLOOR as an object. It is obviously a nonmovable object. The player will not pick it up and carry it around. You may not even include any attributes, descriptions, or actions for the object. But by creating it, and placing it in the current room, the player is allowed more ways of expressing game commands. It makes the adventure more user-friendly, and as mentioned before, it makes for a better game.

Some nonmovable objects are not optional, they are mandatory. Consider the objects necessary if the player wants to "push the boat into the water." You must have created a boat and also created water. Unless you make "water" an object, Visionary will not be able to understand the player's request, and will respond "I don't know what a water is." To allow the player to succeed, you must create "water" as an object.

When Objects Change

Sometimes, a movable object will change. Just as a nonmovable object like a door can change from closed to open, a movable object like a torch can change from unlit to lit.

There are several ways you can change the condition of an object in your adventure. One way is to actually create two objects. Let's say you create a dry torch and a burning torch. Place the dry torch at the location at which you intend it to be found by the player. The second object, the burning torch, should be stored in the store room. When the player lights the torch, place the dry torch in the store room and place the burning torch in the current location.

An alternative to using two separate objects and swapping one for the other is to use a single object and one attribute. Design an object "torch", and name one of its attributes BURNING. The proper description could be given by checking the attribute. Likewise, a darkened room could be fully described if the attribute is set, or it could only be described as "You can't see in the dark" if the attribute is unset. Using two objects, or one object and an attribute, are equivalent

methods to achieve the same result. Both work quite well and the choice can be left up to you, the author.

Invisible Objects

There are special cases where you will have to create two versions of an object. One version will be a nonmovable object and the other will be movable.

Let's say for example that the player is walking through sand dunes filled with tall beach grass. You may want the player to use some of the beach grass for a certain purpose, to be able to take some of the grass. Not all of it; just some of it. When you write the room description, don't mention the grass. It will be more obvious that the grass can be used, if you list it after the location description as an object. If the grass was only listed in the location description, the player would probably not attempt to take any of it. Remember, you don't want to make your adventure impossible to figure out. Listing the grass as an object is a subtle hint that it is to be used in your game.

The first thing to do in this case is to create a nonmovable object that will appear immediately after the room description. You might say, "You see bunches of tall beach grass growing throughout the sand dunes." When you create this object, do not call it GRASS. You want to save that specific name for the second object. Give the first object a slightly different name, say GRASS1.

Then create the second object. This one will be a movable object called GRASS. Place it in the current room. When you write the room description of this object, check to make sure the movable object is not in the sand dunes. If the location is anywhere else you could say, "You see a handful of beach grass lying here." If however the GRASS is at the sand dune location, then print nothing. The first nonmovable object's description will be sufficient, and the movable object GRASS becomes invisible at this location. When you write the description for GRASS in the player's inventory, you could simply say "a handful of beach grass."

Now let's see how this works. When the player stands in the sand dunes, she sees beach grass growing all over the place. When she says "get the grass", give her the movable object GRASS which was in the room at the time, but was invisible. When she takes inventory, she sees "a handful of beach grass." When she looks at the room description again, she still sees "beach grass growing all over the place." If she takes the handful of grass and drops it in any other location, it will show up as "a handful of beach grass lying here." If however she drops it in the sand dunes, it will not show up separately. Rather, it will be included by the statement, "You see bunches of beach grass growing all over the place."

This same technique of an invisible object can be used if the player is standing on a gravel path, and wants to pick up some of the gravel. You create two objects. Place a nonmovable object GRAVEL1 in the room, which is described as "A gravel path beneath your feet." Place a movable object GRAVEL in the same room. Make sure the movable object GRAVEL is described only when it is not in this location. When it is in the same location as the GRAVEL1 (the gravel path) it becomes invisible. In that way, the player can take some of the gravel without affecting the object description. And if he drops it back in the original location, it blends back in with the rest of the gravel, becoming invisible.

Let's examine another variation on the invisible object. Let's say you have a snack bar in your adventure. The player can visit the snack bar and buy any of twenty different items available there. Rather than list all twenty objects as visible objects, make them invisible. Rather than say, "You see a bag of popcorn on the counter," "There is some cotton candy here," "An ice cream bar lies here," and so on, make the objects invisible. That way, they will not clutter up the screen when the location is first described. They will remain invisible when they are at this location, as long as they have not yet been purchased. Let's see exactly how this can work.

Use one attribute called SOLD for each object at the snack bar. If SOLD is unset for any object, then make it invisible. That is, don't give it any description. When the player buys the object, set the attribute SOLD and place the object in the player's inventory. Of course, you must take care of additional things. Be sure the player has the appropriate money, and remove it from the inventory. And always check the inventory limit before completing the sale, to make sure the player doesn't carry more than you want him to.

Let's say the player wanders up to the snack bar. Since the twenty objects available for purchase are invisible, he must have some way of knowing what he can buy. One way is to place a sign on the wall of the snack bar, listing what's available and at what price. Another way is to have the player talk to the clerk behind the counter. She can give him a quick list of what's still available. If he asks for peanuts, check his inventory limit. If it hasn't been reached, then remove the appropriate amount of money from his inventory, give him the peanuts, and set SOLD for peanuts.

When you write the descriptions for the peanuts, check the SOLD attribute. If it is set, describe "A bag of peanuts lies here" for the room description or "a bag of peanuts" for the inventory description. If the attribute is unset, then print nothing. In this way, you will have the peanuts and all the other objects available and present in the current location, but they will be unseen and unobtainable until they are sold.

You can use the concept of invisible objects in other similar situations as well. When the player goes up to a ticket booth to buy a concert

ticket, the ticket can already be at the location, but invisible. It will become visible only when she buys it with a twenty-dollar bill. In a jungle adventure, a python can lie hidden in a tree. It will be invisible until the player examines the tree, at which time an attribute for the python is set and it becomes visible. As you design your adventure, you will find many situations where it is advantageous to use invisible objects.

The main reason for having an invisible object, in addition to those mentioned already, is that the player is permitted to manipulate the object even though it is not visible. That's because the object is actually at the current location. The player can refer to it, and Visionary will understand him. If you chose not to place the object at the current location, then the player could not refer to it. If, for example, the player went up to the ticket booth and said "buy a ticket", Visionary would respond "I don't know what a ticket is." That's because there was no ticket at the location. In Visionary, if the object is not present, it cannot be referred to without additional programming—you can refer to an object that is not present if you want to use the special vocabulary file, but that usually takes more programming steps, and slows down the execution of the program. For further details on the vocabulary file, see the next chapter.

By having the object present but invisible, the player can ask for it, examine it, and otherwise refer to it. You can easily write the code to permit these actions as long as the object is present. The use of invisible objects is a powerful technique for you to use in your game.

As you have seen, there are a large variety of things that can be done with the objects you place in your adventure. Objects are the corner stone of any adventure. Without them, there's nothing to do but wander through the rooms. With them, your program finally becomes a real game. We have now completed the major portions of your adventure. In the next chapter, we will examine your adventure's vocabulary and how it affects the objects.

Chapter 6: The Vocabulary

When choosing your adventure's vocabulary, you must take special care. By judicious use of verbs, nouns, and their synonyms, you will be able to anticipate almost any normal input from the player. You don't want to make the player search for some specific word, when any one of several will do. There is nothing more frustrating to an adventure game player than playing a guessing game, trying to find the exact words to do some simple task. Don't force the precise words, "climb the stairs" to be used—design your vocabulary to permit the player to use variations such as "climb the steps," "climb up the steps," "climb up", or even simply "up." It will make your game much more playable.

Vocabulary refers to the words in your adventure. It includes the nouns, and verbs, as well as the adjectives and prepositions. When you write your adventure, you need to be constantly concerned about using the appropriate vocabulary. It is through the vocabulary that the player will interact with your game, and come to feel a part of the new and exciting world that you have designed. The player should be able to easily move about in your world, manipulate objects, and solve the puzzles you have set out. A well-chosen and carefully thought-out vocabulary makes for a smooth playing and a thoroughly enjoyable game. A poorly-chosen, carelessly-planned vocabulary makes a frustrating game that won't get played.

Vocabulary in a Graphic-Only Game

If you are designing a pure graphic adventure with no typing allowed, you will be concerned with selecting a limited number of verbs, on which the player can click the mouse pointer. Because no typing is accepted in such a game, you must anticipate every normal request from the player, and provide a list of the verbs on the screen.

You may find that selecting some vague words will allow you more freedom. For example, if the word "use" can be clicked on, the player can then click on the picture of the gun to "shoot", click on the picture of the shovel to "dig", or he can click on the picture of a key to "unlock." In this case, you do not have to provide the verbs "shoot", "dig", or "unlock." The one word "use" satisfies the need for the three more specific ones.

For this reason, there is little need for taking further care with the game vocabulary in a graphics-only game. There is no need to choose adjectives. The player will not be typing in "red box" or "blue box," but simply click on the picture of the desired box. And similarly, there is no need to worry about prepositions. If the player wants to put an

object "in" someplace, or "on" something, a click on the first object and then the second object is all that is needed. Likewise, there is no need to anticipate special commands, since no typing is allowed in a pure graphics game.

Text Adventure Games

But in a text-only game, or a combination graphics and text game, the subject of vocabulary is a vital one. The remainder of this chapter will examine more closely the special care you must take in designing the vocabulary for a text game, or text-graphics hybrid.

As you write your text adventure game, you will design the vocabulary in several areas. One of those is a special vocabulary file, which we will examine later in this chapter. Another is in subroutines. But the most common place where you will work with the vocabulary is with objects. Let's examine a sample file for an object and see how the vocabulary is integrated into the game.

Object Vocabulary

In each object file, you will choose the nouns that represent the object. You will also decide if any adjectives are necessary to describe each object. But the most important role for vocabulary is in the actions programmed for each object. It is here that you decide what manipulations will be permitted with each object and what verbs will be accepted to initiate those manipulations. It is here that you will decide if a candy bar can be eaten, and if so which verbs will be accepted. You may only wish to accept "eat the candy bar." Or you may wish to accept additional variations such as "swallow the candy bar," "chew the candy bar," or "enjoy the candy bar."

Let's examine an object file, step by step, and see in which parts you need to be concerned about the vocabulary.

The Object Name

When you first define an object, you must choose its name. This is the name that you will refer to when writing the game. Then you must select the nouns that the player will use when referring to the object. Usually, these two words are the same. The difference is that there is only one name given the object for the game to refer to. There can, however, be many names that the player refers to. This is where synonyms are used.

Synonyms

For each object in your adventure, synonyms are important. It is important to list as many synonyms as possible. For example, if one of the

objects in your adventure is named "pistol," you should also include the alternate words "gun" and "revolver" as synonyms.

Synonyms allow the player to be a little looser in what he says as he plays your game. He can say, "get the gun," "shoot the pistol," or "load the revolver." Granted, it is not as precise to accept so many different words for the pistol, but it is certainly more user-friendly. And that's important in any adventure. You don't want to frustrate the player with a search for the exact word. You must allow a little leeway in the words he chooses.

When you begin to design the object of the gun, you give it the name "pistol" for the purpose of writing the adventure. Use only this noun, no others. Whenever you want your game to check to see if the gun is present, you refer to it as "pistol", not as "gun." After you have given the object its name, then list all the names the player can use. Include "gun" and "revolver" as well as any other appropriate synonyms like "firearm." This list of nouns is checked when the game is being played. When the player has typed a command involving the gun, your game will match his noun with one of the several nouns that stand for the pistol. It won't matter if he calls it a "pistol", a "gun", or a "revolver," the game will understand what he means. Be sure you list all other logical synonyms for the name of the object.

Adjectives

After selecting the object name and its synonyms, the next thing is to list any adjectives. If you have used adjectives in your object descriptions, then they must be included in the list of adjectives. If for example, you have described a knife as "a sharp knife" then you must include the word "sharp" in the adjective list. If you fail to include the word "sharp" then the adventure will fail to understand the player if he commands "pick up the sharp knife." Visionary would respond, "I don't know what sharp means."

Since you included the word "sharp" in your object description, expect the player to use it. That means it must be included in the adjective list. If your adjective list doesn't match your object descriptions, then the game will not respond smoothly, and the player will encounter frustrations in playing your game. And this is the kind of frustration you want to avoid.

Avoid Using Double Adjectives

Beware of double adjectives in your objects. Visionary can't handle more than one. For example, let's say you create a broom that is described as "a dusty old broom." It would seem logical to then include both words "dusty" and "old" in the adjective list. In that way, the player could say "get the old broom," "get the dusty broom," or "get the dusty old broom." Unfortunately, if you attempt this you will create more problems for the game player than you will solve.

The difficulty is that Visionary cannot handle more than one adjective at a time. It would respond properly if the player commanded "get the old broom" or "get the dusty broom." However, if the player attempted to use both adjectives exactly as she saw the object described, Visionary would not be able to comply with her request. If she said, "get the dusty old broom," Visionary would reply "You can't see a dusty old here."

The best solution then, is to use single adjectives to describe the objects. If more than one adjective becomes absolutely necessary, use them all in the adjective list and create a special vocabulary file for the command that uses the double adjectives. This special vocabulary file will be covered later in this chapter.

The next three parts of every object file do not involve vocabulary. They take care of attributes of the object, starting location for the object, and descriptions of the object. All three are important to your adventure, and have been examined in previous chapters. But since they are not affected by the player's input, they are of no concern to our discussion of vocabulary. For the purposes of this chapter, we will skip these three parts of the object file, and move on to the actions, where vocabulary plays its biggest role.

Object Action

The action section of each object file is the largest. It is the last section of the file and contains all the actions that can be taken with the object. This is where the verbs are listed.

Prepositions will also be used here. This section is where the player tries to manipulate the object. This is where you decide what will happen when he says "look at the ring" "get the ring," "drop the ring," and other phrases as he attempts to manipulate the ring.

Look

Generally the first action you should include is "look," since it applies to all objects. Always allow objects to be examined. This includes both movable objects as well as nonmovable ones. The player will be able to learn much about the object by asking to "look at the ring." Don't forget to include the synonyms "examine," and "view." All synonyms are listed after the verb "look", on the same line separated by commas.

Some additional synonyms may also be appropriate depending on the object. Consider the word "search." "Search the ring" is not synonymous with "look at the ring," "examine the ring," and "view the ring." However, if the object is a book, then "search the book" is synonymous with "examine the book." By searching or examining the book, the player might find a flower pressed in between the pages. In this case, "search" is a synonym for "examine."

When working with objects that contain writing, you must decide if you want "look" and "read" to be synonymous. If the player asks to "look at the book," you might want to show him what the book says, just as if he asked to "read the book."

Or you may choose to treat the two verbs as totally separate actions. If the player asks to "look at the book," you might want to describe the cover and mention that "there's plenty to read inside." She would have to ask to "read the book," to see the exact words contained in it. Whichever you decide, be consistent in your adventure. Treat all objects that can be read in the same manner, whether they are books, newspapers, advertisements, signs, or stone tablets.

Depending on the object being examined, you might also want to include additional synonyms for "look," such as "frisk," "observe," "scrutinize," "scan," "inspect," "check," "study," or "analyze." If for example, the player is playing detective in your game and has caught a thief, he might expect "frisk" to be synonymous with "examine." On the other hand, if the player is a scientist who has discovered some laboratory rats, he might expect "observe" to be synonymous with "examine." The point is that when you list the synonyms for "look", vary the list depending on the nature of the object being looked at. Use any that are appropriate for that object. Remember that the more synonyms you include, the more user-friendly your adventure will be. It makes for a better game.

Get

The next action after "look" is usually "get." This is where the player can take the object. In the case of nonmovable objects, this is where the player attempts to take the object, only to be told that "it's too heavy," or "it's nailed down."

In the case of movable objects, this is where you will check to see if he already carries the object, and if not, if he can carry it without exceeding his inventory limit. When you choose your vocabulary for this section, be sure to include the synonyms "take," and "grab."

As with "look" and all other actions, the synonyms for the verb are listed after it, on the same line separated with commas. Other synonyms like "steal," "lift," and "carry" may be appropriate depending on the game you are creating. If your player is a jewel thief, it would be appropriate to include "steal" as a synonym for "get." Don't expect to use the same synonyms for "get" for every object. They will vary, depending on the nature of the object that the player is trying to take.

When the player attempts to "get" or "take" a nonmovable object, you can use a larger selection of synonyms. You can use words that normally are not synonymous with "get" but which all result in the same response. Let's say a large boulder blocks the player's path. If the player tries to move it, he is told, "it weighs tons, and won't budge."

Since this response is appropriate for any attempt to move the boulder, feel free to list many verbs which normally would not be considered equivalent to the word "get." Include the words "get," "take," "grab," "push," "pull," "slide," "move," "roll," and "lift." No matter which of these verbs are used by the player, he is told that the boulder won't budge.

» A general rule: When you deny the player the ability to do something, include many more synonyms for the action than would usually be used.

Drop

The next logical action after "get" is usually "drop." In this section, the player is allowed to drop any object she is carrying. It is here that you check to make sure the object is not already dropped. It is also in this area where you decrement the player's inventory if necessary. (It is not necessary if you are using the built-in variable ITEMS to keep track of the number of items in the player's inventory). You will not find a lot of synonyms for the word "drop" so the vocabulary for this section of your game will be limited.

Be careful when choosing the few limited synonyms for "drop." Sometimes it depends on the player's location. When you are programming the actions to drop each object, you must decide if "throw" should be a synonym for "drop." If the player is standing in a meadow and says, "throw the bottle," then you would want your game to respond as if he had said, "drop the bottle." In this case both words are synonyms and you would reply, "OK," remove the bottle from the player's inventory, and place it at the current location. If, however, the player was standing on a cliff overlooking the ocean, then the situation would be different, and "throw" no longer has exactly the same meaning as "drop." In this situation, if the player said "drop the bottle," you would expect it to fall at his feet and remain at his location. If on the other hand he said, "throw the bottle," you would expect it to fly over the cliff and no longer be at his location.

The way to deal with cases like this is to decide what you want to happen to the bottle. If you want identical things to happen, then make "throw" a synonym for "drop." Otherwise, make them two separate actions. One action, "drop," would always drop the bottle. The other action, "throw," would sometimes drop the bottle and other times move it to the base of the cliff. Using conditional statements, you could have the bottle stay at the current location or move to a different one, depending on whether the player stands in a meadow or on a cliff.

The word "remove" may or may not be equivalent to "drop." It depends on how you wish you design your game. Let's say that the player is wearing a hat. You have an attribute for the hat called WORN, and it has been set. If the player asks to "drop the hat," then he obviously wants to remove the hat from his head and drop it on the ground. In this case, you would drop the hat and unset WORN. If, however, he asks to "remove the hat" then he probably intends to remove the hat from his head but continue to carry it.

If you want to permit this, then you must not make "remove" a synonym for "drop." You must make it a separate action. The problem here is that you will then have to create a separate action for "remove," in which you check his inventory limit to ensure that removing the hat doesn't cause his inventory limit to be exceeded. One way to avoid this is to make "remove" a synonym of "drop." If you choose to do this, then when the player says, "remove the hat" you will drop the hat at the current location, check WORN and if it is set, unset it and tell the player that the hat "falls to the floor." In this way, you don't have to worry about his inventory already being at the limit before removing the hat.

If the player still wishes to carry the hat, he is able to say "get the hat" after removing it. If it exceeds his inventory limit, it will be caught at that time. You may choose to make "remove" a synonym for "drop," or you may wish to treat it as a separate action. Both methods have their own advantages, and the decision is yours to make.

Wear

The next action that you should deal with is "wear." Of course, it only applies to a few of the movable objects in your adventure, but this is an appropriate place to include it if it applies. Designing the proper vocabulary for "wear" is a bit tricky. The reason is that the only common alternate ways to say "wear the hat" are "put the hat on" or "put on the hat." List "put" as a synonym for "wear." Visionary will treat them the same, and allow all three statements to be used synonymously. Since "on" is a recognized preposition, all three commands will work.

The problem, however, arises if the player should want to "put the hat on the mannequin." If the mannequin is not an object presently in the player's location, Visionary will give an error message, "I don't know what a mannequin is." If the mannequin is an object that you have defined to be in this location, then the player will end up wearing the hat instead of the mannequin. To prevent this from happening, you must anticipate which objects the player may try to put the hat on, and separate those actions with the OBJNOUN command. An example of Visionary code below shows how this is done.

```
ACTION  WEAR, PUT
 IF OBJNOUN IS MANNEQUIN THEN
  T You can't put it on the mannequin.
 ELSE
  T It looks nice on you.
  SET HAT, WORN
 ENDIF
ENDACT
```

In this example, if the player types any preposition such as "on," and uses "mannequin" as the object noun, she is prohibited from putting the hat on it. Of course, there is nothing to prevent you from making this an important part of your game. Perhaps it is required that the player put the hat on the mannequin. In that case, you would change the example, and set an attribute called ON-MANNEQUIN. You would change the descriptions of the hat and the mannequin depending on the status of the attribute.

An important thing to remember when making "put" a synonym for "wear" is that you must anticipate the player may try to use the hat with other objects as well. She may try to "put the hat on the shelf" or "put the hat in the box." Both will result in the hat being worn by the player unless you specifically program around them as shown above. All this can make for a large amount of programming, but allows the player a wider range of actions.

An alternate method that is not as user-friendly but is easier to program, is to make "put" and "wear" separate actions. Don't make them synonymous. Program the verb "wear" to put the hat on. Program the verb "put" so that only a simple message is given, such as "Try to WEAR the hat." In this way the player is prompted to use a verb different from "put," if indeed he wishes to wear the hat. And if he was trying to do something different, he then knows he can't do it. This alternate method is not the recommended one, but may be preferred if your game has too many objects for the hat to be put in, or put on, or put between, or put under. It does make things a lot simpler.

An action associated with wearing objects, is removing objects. If the player can wear a hat, she must be able to remove the hat. As discussed earlier, you may wish to make the verb "remove" synonymous with "drop." In this way, when the player removes the hat, she not only stops wearing it, she also stops carrying it.

Frequently, however, you will wish to keep the verb "remove" as a separate action from "drop." You will usually want the player to be able to remove a hat, some gloves, or other objects while still carrying them in the inventory. As also mentioned previously, this places upon you the additional responsibility to check the player's inventory limit before allowing her to remove the hat. If you choose to keep "remove" separate from "drop," then it is appropriate to place it after "wear" in the object file.

There are problems in designing the vocabulary for "remove" that are similar to those encountered with "wear." In addition to asking to "remove the hat", the player may also ask to "take the hat off" or "take off the hat." We are now presented with a situation where the verb "take" has a meaning exactly opposite from normal. The preposition "off" is what makes the difference. So you will have to go back to the action containing "get, take, grab" and modify it.

First, write the necessary code to "remove" the object. Make this a separate action from the others, with no additional synonyms. Be sure to include the necessary inventory checks when taking this action. Once you have finished writing all the code for the verb "remove," then go back and modify the "take" action. At the beginning of the "take" action, check for the preposition "off." If it appears, then insert a duplicate copy of all the code that appeared under the "remove" action. An example of the Visionary code follows.

```
ACTION GET, TAKE
 IF PREPOSITION IS OFF THEN
  insert duplicate code here
 ELSIF PLAYER HAS HAT THEN
  T You already have it.
 ELSE
  GRAB HAT
  T OK.
ENDIF
ENDACT
```

In the above example, if the player says "take off the hat" then Visionary recognizes that he has used the preposition "off" and it executes the same code that would have been executed if the player had said "remove the hat." If the player doesn't use the preposition "off", the you assume he wants to take the hat, and you continue with the normal actions for "take." The above example was intentionally kept simple. Don't forget to always check the inventory limit before allowing the player to take anything.

Subroutines for Actions

Since having duplicate sections of code, one set under the "take" section and the other under the "remove" section, is not only redundant but takes up extra space, you may prefer to use a subroutine. You may prefer to place the code for removing an object in a subroutine, and then call the subroutine from the two separate places. This saves you time, and keeps the program smaller.

We haven't said much about subroutines until now. For more information on subroutines and how to use them, see Chapter 8, Subroutines.

Object-Dependent Actions

We have discussed the most common vocabulary words that the player will use. She will try to examine objects, take objects, and drop objects. Some objects, like a book, she will try read. Others, like sunglasses, she will try to wear and remove. In addition to these, you should allow other common actions as well. Allow every object to be manipulated in normal ways.

Before completing the object file, be sure you include some actions that respond to normal anticipated vocabulary words. If the object is book, allow the player to burn it, if fire is present in your adventure. If the object is a bottle, allow the player to put something inside it. All these actions are normal and should be anticipated. Design your vocabulary accordingly. Remember to use as many synonyms as you can to make the game easier to play. Let the player say "light the book" as well as "burn the book." Let the player "insert the note in the bottle" as well as "put the note in the bottle," "place the note in the bottle," and "stick the note in the bottle."

Let's examine the example of a note in a bottle further. The vocabulary involved in the command to put a note in a bottle is a bit more complicated than usual. Let's see the best way to accomplish it. Obviously, the verb is "put." It is less obvious which noun to use. Is it "note" or "bottle?" Should this action be included in the file for the object "note" or for the object "bottle?" Both objects must be present in order for the feat to be accomplished, so the programming code will be executed regardless of which object file contains the action. However, you will find it a bit easier to write the code if you include it as an action on the note, rather than the bottle. This is because the note is the direct object. The bottle is the indirect object, acted upon by the preposition "in."

Now that we have decided to place the action in the file for the note, we need to start by listing all the synonyms. As stated repeatedly before, always include as many synonyms as possible. In this example, it is appropriate to use "insert," "place," "stick," "push," and "shove." After listing the synonyms, you must check to see if the player wants it placed in the bottle, rather than in something else. Don't bother checking to see what preposition was used. It doesn't matter whether the player said "put the note in the bottle" or "into" or "inside." The only thing you need to check is that "bottle" was used after some preposition. Look at the following Visionary code.

```
ACTION PUT, INSERT, PUSH, STICK, PLACE, SHOVE
 IF OBJNOUN IS BOTTLE THEN
  T OK, the note is in the bottle.
  PLACEOBJ NOTE, STORAGE
  SET NOTE, IN-BOTTLE
 ELSE
  T You can't put it there.
 ENDIF
ENDACT
```

Remember, this section of code would appear in the object file for the note, not the bottle. As you can see above, if the player asked to place the note inside the bottle, he was told he had succeeded. The note was removed from the current location and placed in the storage room (a concept that was discussed previously). Then a flag was set. An attribute named IN-BOTTLE was created for the note.

By checking this flag, you can later give the proper description for the bottle. You can say "the bottle is empty" if the attribute is unset, or "the bottle contains a note" if it is set. You can do a variety of other things as well, if you know that the note is either in the bottle or not. Perhaps the player is rescued only if the note is inside the bottle.

The Vocabulary Action File

You want to create an adventure with a complete vocabulary, one that will permit more sophisticated sentences. By using prepositions and object nouns, you can do much more than the simple two word commands of primitive adventure games. The player is not limited to saying things like "get bottle" and "write note." The enhanced vocabulary allows him to make more complex requests. He can "put the hat on the mannequin" or "insert the note into the bottle."

However, there may be times that you want to permit even more complex commands. You may want to let the player ask for things that even Visionary could not normally do—and you can. You can even exceed Visionary's vocabulary limits, and understand special commands from the player that normally would not be possible. The secret is in the vocabulary action file.

The **vocabulary action file** is a special file, separate from the object files. In this file, you can include any anticipated input from the player that might not normally be picked up by Visionary in the actions of an object file. As mentioned before, using two adjectives is not permitted in Visionary. The player can't refer to "the dusty old broom." The vocabulary action file is a way around this limitation.

Normally, the player cannot refer to two objects at once. He cannot say, "place the note and the ring inside the bottle." But you can permit it, if you include it in the vocabulary action file. There are some reserved words like **JUMP** and **LOAD**, which normally cannot be used by the player. But they can if you program them into the vocabulary

action file. The vocabulary action file is a unique place to deal with special vocabulary problems. Let's examine it further.

The vocabulary action file is made up of actions, and programming code to be executed if the actions are requested. This looks much like the action sections in each object file. The difference is that the action sections of object files are only checked if the object is present with the player, either in his location or in his inventory. The action sections of the vocabulary file are checked after every move, regardless of where the player is or what he carries.

Visionary will check here before anywhere else, so that a player's request to "place the note and the pencil inside the bottle" can be acted upon, if it is included in the vocabulary file. When you add anticipated sentences into the vocabulary section, you must include the verbs, nouns, adjectives and pronouns. You may leave out only words like "a," "an," and "the."

Let's see some of the special ways the vocabulary action file can be used. Take the example of the note in the bottle. The way in which this example was designed will not allow the player to take the note out of the bottle. If he said, "take the note out of the bottle," Visionary would say "I don't see any note here." Remember that the note was placed in the storage room.

One way around this problem would be to make the note invisible, as discussed in the last chapter. The problem with that is, it would require that you continue to move the invisible note from one location to another as the bottle also moves. It would have to follow the bottle, so it could be removed when the player requested it.

A much simpler solution is to use the vocabulary file. Include an action for "take the note out of the bottle." When the player makes this request you only have to check to see if the player has the bottle and if the attribute IN-BOTTLE is set. If so, take the note from the storage room and place it in the current location. This is an example of how the player can refer to objects that are not at his location. Normally, such a thing cannot be done. But with the vocabulary action file, it can.

Another time you will need to use the vocabulary action file is when there is no object referred to. Remember that when the player inputs a command, Visionary checks through all the objects present to find a matching command. If the player simply asked to "wait", Visionary would not find an object present with a matching command. Put "wait" in the vocabulary file. Since Visionary checks the vocabulary file first, it will find "wait" and will give the response you chose.

This "wait" command would be an excellent way to make time pass quicker, if your adventure used a clock. Similarly, if the player wanted to "sleep" the vocabulary file could intercept that command and deal with it appropriately. If the player said, "sit down," you could set any necessary flag and respond "OK." The vocabulary file is an excellent

place to insert single word commands and other commands that are not actions made upon an object.

Reserved Words

Reseved words like LOAD and JUMP must be defined in the .VOC file if they might be used in player commands

There are several reserved words in Visionary which normally cannot be used with an object. The vocabulary action file allows their use. The word "load" normally signals Visionary that the player wishes to load a saved game, and causes a window to appear for selection of the proper saved position. This causes problems if you have a gun that needs ammunition. If your player finds an empty rifle and later finds some shotgun shells, he would logically respond "load the gun". If you try to place the verb "load" in an action inside the object file for "gun" you will find that it doesn't work. If the player types "load the gun," the window appears so he can load a saved game. By using the vocabulary action file, loading the gun is simply accomplished. Make the action, "load the gun" and use exactly the same code that you would have used in the "gun" object file.

The word "jump" is another reserved word in Visionary. If the player is standing on a high diving board and types "jump off the diving board," Visionary will respond "ILLEGAL COMMAND." This is because the word "jump" is used in debugging. But your player can still jump off the diving board, if you use the vocabulary action file. Create an action called "jump off the diving board" and program whatever response you wish. Perhaps you will have the player die when he hits the dry bottom of the drained pool. Perhaps he will find a treasure under the water. Whatever you decide, you can make it happen in the vocabulary file.

We've spent a lot of time examining the vocabulary of your adventure. We've looked at the common commands like "look," "get," and "drop." We've looked at synonyms and how they can change. We've looked at how to use more sophisticated commands with prepositions and indirect objects. And we've looked at how the vocabulary file can increase the sophistication of your adventure. But through all these things, there has been one common thread. When you create the vocabulary for your adventure, make it easy for the player to find the right words. If she has the right idea, don't make her go searching for an exact phrase in order to try her idea. Use synonyms and alternate phrasing to permit as wide a variety of commands as possible. Remember, your game will only be enhanced by allowing the player many equivalent ways to express commands. The better the vocabulary, the better the game.

Chapter 7: Messages

In your adventure, you must constantly keep the player informed of what is happening. You do this by presenting messages telling what is occurring as the plot unfolds and changes take place in player status and in the surroundings of the make-believe world. **Messages** are an important part of any adventure game, whether a text or graphic adventure. Without them, the player would have only the room and object descriptions or graphics as a guide. The player would have no idea what else is taking place in the game.

There are a variety of purposes for messages in your adventure—most apply to graphic adventures and **all** apply to text games. This chapter will examine these messages and see how to best make use of them.

Action Messages

Respond to each player command

A message must appear when the player takes any action. Regardless of what action the player chooses, your game must reply with some message. Even a short reply like "OK" is acceptable, in response to some simple command like "get the axe." But some reply is required. In a poorly-designed adventure, if the player asked for "inventory" and carried nothing, there would be no reply. In your well-designed adventure, the same situation would be met with the message, "You aren't carrying anything." Make sure you respond to any command made by the player, regardless of the nature of the command or the brevity of your response. You must let the player know you recognize the command.

Update player or game status

Whenever the player takes some action, your game must give a message updating the current situation. If the player asks to "chop down the tree," you must program some response. You might reply, "the tree falls with a crash," to let him know he succeeded. In a graphics adventure, you could show the tree falling, accompanied by the sound of the crash. Alternately, you might tell him, "You can't. You don't have an axe." Either way, you have printed some message to advise the player of the result of his request.

The message can go farther and give the player additional information as well. The request to chop the tree might be met with, "as the tree falls to the earth, you see something fly out of the treetop and sink into the swamp." This type of message is doubly important in a graphics adventure, since the player might not notice it in the graphics. This message has not only told the player that he succeeded in chopping down the tree, but that there was something in the tree top, and per-

haps he should have climbed the tree and found it, before cutting the tree down.

Inform player of unexpected results

A message can also tell your player of some unexpected result from her action. After requesting to cut down the tree, you could tell her that "It falls across the river, creating a natural bridge to the other side." Here, the message not only tells the player that she has been successful in his bid to chop down the tree, but that she has solved the problem of how to cross the river. In a graphics adventure, this type of message would be less important, since the tree could be actually shown spanning the river. It never hurts, though, to give some response, even in a graphic adventure. Chopping down a tree is just one example of the actions a player can take. Keep in mind that whatever the request, a message must always be given, acknowledging the request and updating the current status of the world.

Clues and Help Messages

Another time that your game will print messages, is when the player specifically requests them. There are times when the player will ask the computer for information. Your game must be prepared with an appropriate message. The player might ask for "help." Generally, you will have a variety of messages available giving clues when the player has requested them. Choose the proper message depending on the current situation, the player's location and which puzzles have already been solved.

Another way that the player may specifically ask the computer for a message is, "What time is it?" This is a common request in an adventure in which time is an important factor. You must have some reply ready, whether it is a specific time on a clock or a general time of day. Depending on the plot of your story, you may wish to answer the player with "It is 2:10 PM" or "It is mid-afternoon." This specific example best applies to a text game. In a graphics game, it would be simpler to have a picture of a clock on-screen at all times. But even in a graphics game, there are many times that the player will ask the computer for information.

The most common way in which the player will specifically request a message from the computer is by asking to examine an object. It is a standard behavior to examine an object as soon as it is found. In a text game, this is usually accomplished by typing "look at the bottle." In a graphics game, the player usually only has to click on the picture of the bottle.

Most experienced adventurers are wary and will ask to examine an object even before picking it up. You must be ready with the appropriate message to describe every object in the world of your creation, whether it can be picked up or not. The message must tell the player everything important about it, unless you have a reason for hiding something.

Even then, be sure you play fair with the adventurer. As mentioned before, don't spring any fatal surprises on the player without some hint or clue first.

Another way a player will specifically ask the computer for a message is when he asks for "help." In return, he expects a message that will give him a clue. The request for help is made when the player is stuck and can't figure out how to accomplish a certain task. It may even come from overall frustration, and be a general cry for help.

Clues shouldn't tell too much

There are several things to consider when you design these messages. First, don't blatantly tell the player what to do. Don't simply say, "pry the door open with the crowbar." You want to make it a bit of a puzzle or a riddle, so that the player will feel the glow of success and a sense of accomplishment. Reply instead, "You may need some leverage here." When he figures out how to open the door, he will feel good about himself. He was smart enough to figure it out, even if it did take a clue from the computer. By telling him exactly what to do, you make him feel dumb. But giving him a clue and letting him decipher it makes him feel smart. The player will enjoy playing your game a lot more if you don't make him feel dumb.

Provide more than one clue per situation

A second thing to consider when you write your clues is to design more than one per situation. If the puzzle is especially tricky, you may want to split the solution into several parts, and give separate messages for each part. You might even want to mislead the player at first. If she has encountered a padlock and has asked for help, you might reply, "Check the keys in the pantry." After you have verified that she has tried unsuccessfully to use the keys to unlock the padlock, you could have a second clue available. The next time she asks for help, you could give her the message, "If the keys don't work, try breaking it with something." Remember to be a little vague. Don't be too specific, or it will take a lot of the fun out of the game.

Default Message

When you design your clue messages, have a default message. This is a message given to the player when you want to give no help. If you can tell from the player's location, from the puzzles already solved, or from flags set, that the player has not currently encountered any difficult puzzle, you may wish to simply respond, "Sorry, you don't need any help here." You may wish to encourage the player to "examine everything" or to "try again later." But as stated before, you must give some reply to the player's request, even if that reply is a standard default message.

Timed and Random Messages

Messages can not only appear when the player takes some action or asks for a message, but they can also appear at some set time. In this

case, it will not matter what the player has or has not said or done. The message appears because it is time for the message to appear.

At the stroke of midnight, "the skeleton rises from its coffin and stalks into the night." This message advises the player of some important event taking place in the game. This can be especially important in a graphics game, when the action takes place off-screen. If the player is depending strictly on the graphics to provide him with information, he will miss important events. In such cases, giving the player such a message is vitally important.

Random messages can add realism

Some messages don't advise of any important event, rather they add a touch of realism to the game. These messages frequently appear at random. At various times, the player is given messages like, "a bird chirps in the trees" or "a tumbleweed rolls by in the hot wind." Messages like these, whether occurring randomly or at set times, appear without any specific action by the player. They are not given in response to some action, or as a response to his request for information. They usually have no place in a graphics game, since such random occurrences can be better presented graphically, showing an animated sequence with a tumbleweed rolling past, or they can be presented with digitized sounds of birds chirping on cue. But in a text game, this type of message can help make the game seem more realistic.

Warning messages should appear in both graphic and text adventures, when the player is about to encounter danger. These are usually triggered by some action the player has taken or some location he has reached. Remember, you should never allow the player to die without warning. Give some warning messages. When the player reaches a center of a cave, print a message that "you feel a strange foreboding wind coming from the north." If the player picks up a half-buried shovel in the cave, the message might be "you hear a rumbling sound, as though the rocks above your head were about to cave in."

Give the player adequate warning of danger ahead

Arriving at locations and using objects aren't the only way to trigger a warning message to the player. You can also have subtle warnings in the object descriptions. You could describe the cave opening with "a boulder, teetering on the edge above your head." Just make sure the player knows of the danger and is allowed to avoid it. Never kill the player without some type of warning first.

Message Style

Make your messages clear and explanatory. If she asks to "open the door," don't just say "You can't." Tell her why she can't. Say, "You can't. It's locked, and you don't have a key." Or, "You can't. The key you have doesn't fit." Or even, "You can't. It's locked." But at least give the player some explanation.

Give the player some explanation when a requested action is not allowed

If you don't allow your messages to be explanatory, the player will become frustrated. Don't expect her to be able to figure out how to open the door, unless she knows why it won't open. Without an explanation, she is left trying to guess. Is it locked? Is it nailed shut? Is there a magic spell on it? Is the doorknob missing? The player won't know unless you provide the information.

Humor is appropriate if you are designing a funny adventure. In that case, make your messages humorous. Go all out. Use puns, jokes, and humorous situations. Look for various ways to enhance your messages. Don't just say "a clown jumps out and hits you in the face with a pie, and then closes the door again." Embellish it, "Bonzo the clown springs from behind the door and smacks you in the face with a cream pie, then pulls the door closed again while chuckling heartily." An adventure game with lots of laughs can be most enjoyable to play. If you choose a light tone for your game, don't pull any punches. Feel free to let yourself go.

Messages in Subroutines

As you write the messages for your game, you will find yourself using some of the same messages over and over. "You don't have it." "Sorry, you are carrying too much." "It already is." These messages appear frequently in adventures, whenever the player tries to use something he doesn't have, tries to pick up something that can't be carried, or tries to do something that already has been done.

Use subroutines for commonly-used messages

To save yourself the effort of typing them over and over, and to save memory as well, use subroutines for commonly used messages. Every time you check a new object to see if it can be picked up, you must first see if the player already carries it. If he does, you will respond, "You already have it." You will find yourself writing this message over and over, for each object you define. How much easier it is to simply call a subroutine instead of typing the message each time, and have the message in the subroutine. You will save time, and the resulting game will be smaller.

Subroutines are an excellent place for commonly used messages, and will be examined in the next chapter.

Messages With Variables

Some messages will need to include a number. For example, "You have taken 56 turns." The number 56 will change, as the game is played. Luckily, it is easy to print messages containing variables. The "@" symbol in front of a variable name is the way in which Visionary allows the printing of variables. Look at the following example of Visionary code.

```
T You have earned @SCORE points.
```

When the player asks for her score, the above message tells her how many points she has earned. Similarly, you can print out the value of other variables in your messages. You can tell the player that "the time is 10:30 AM" or "you can hold your breath only 4 more minutes."

Sometimes you will have a message that is going to appear several times with minor variations. To save space, as well as time spent typing all the variations, split the message into several parts and make the unchanging part a subroutine. Let's say the player has a magic lamp, and can get various treasures by rubbing it. A sample message is shown below.

```
Poof! A red genii appears in a puff of white
smoke. It smiles and hands you some gold coins.
```

If the player rubs the lamp again, let's say he gets a ring. The third time, he gets a necklace. The fourth time, some jewels. And so on. Each time he rubs the magic lamp, the above message must be repeated, with the appropriate changes. To save typing, split the message into two parts. The first part, which never changes, can be placed into a subroutine. When the player rubs the lamp, call the subroutine, followed by the second half of the message that describes what the genii gives him. This not only saves you a lot of typing, it also saves memory. The longer the message, the more memory you will save.

Text Styles

Consider the "look" of your messages. There are a variety of things you can do to affect the way a message is presented. The type style can be changed in your messages. You can change the fonts that you use. The default font on your Amiga is **Topaz**. It is an easy matter to switch to a different font.

In a text game, you can use the **Fast Fonts** program found on your Workbench disk to change fonts. Follow the directions in your Amiga manual and modify your startup sequence so that a different font will be used after your game disk is booted. In a graphic game, where text is printed to the graphic screen, use Visionary's three "font" commands. You are allowed to load as many as eight different fonts into memory at once, and use them on the graphic screen. Using a different font will give your messages a special look.

Other ways to change the look of your messages include use of italics, bold print, underlining, and reverse characters. All of these styles can be accessed from within Visionary. It gives a nicer look to your program if you will use some of these features. Don't overuse them, or the game will take on a cluttered look. But when appropriate, use italics to place stress on a word or phrase. Bold print can be used to emphasize words in a different way. You might want to emphasize a certain word in a help message by placing it in bold letters. Reverse characters would be appropriate as part of your adventure title.

Don't overlook the various ways to use these changes in type style. Your messages will take on a whole new appearance, and your game will be enhanced.

Spelling and Grammar

The final point to be made about your messages, is a reminder to check your spelling and grammar. You should have been checking the spelling and grammar of your location and object descriptions as they were written. Don't forget to watch your messages as well. Nothing can make an excellent game look amateurish as easily as errors in spelling and grammar. They are so easy to fix. Don't overlook this simple yet important task.

Both text and graphic adventures rely heavily on messages. They keep the player updated on what is happening in the game. Use lots of them. Make them descriptive. Add humor if appropriate. Remember, you are building a world with words. Even if you plan on using graphics, the messages in your game are vital in creating a world that is alive, vibrant, and exciting. Take great care in creating your messages. Your efforts will be rewarded many times over.

Chapter 8: Subroutines

Subroutines are the sections of your game that can be used over and over by various portions of your program. By using subroutines, you will speed up your work as you write your game, and also will keep the memory size of your game down. Subroutines are important to adventure game creating, and will be examined in this chapter.

Messages from Subroutines

The simplest use of a subroutine is in a single-line message. As mentioned in the last chapter, you will frequently find certain messages appear over and over throughout your program. You may find the phrase "You already have it" used many times. "Sorry, you are carrying too much" is another common message you will use frequently. By placing messages such as these in subroutines, they can be called when needed, without taking up as much space. Your resulting adventure takes less memory, and this leaves you more room for a larger adventure.

Keep a separate file of these subroutines. Give each of them meaningful names. You might give the name "DoorIsClosed" to the subroutine containing the message, "The door is closed and securely locked." When the player tries to open the door or examine the door, call "DoorIsClosed." By using meaningful names like this, it's easy to remember them as you write your program. Whenever you come across a place where the message is appropriate, you can easily recall its name without effort.

Not only is it easy to remember the name as you are writing the code, but it's also easy to recognize the purpose of the subroutine when you are looking at the code a year later. You may not anticipate the need to look at the program code after the game is finished, but it occurs more frequently than you might expect. By using meaningful subroutine names, you will make that future job easier.

Action Subroutines

Often your subroutines will be composed of more than a single line. Entire blocks of code can be placed in a subroutine. Let's say for example, that the player asks to "dig." If he wanders out into the desert, he might type "dig in the sand." If he returns to the jungle, he might ask to "dig in the ground." Or if he runs across a native burial ground, he might want to "dig up the grave."

If all three situations occur in a single game, you must place the "dig" action in more than one place. You must create an object "sand" and

allow for the action "dig." You must create an object "ground" and again allow for the action "dig." A "grave" must be created, and you must permit the action "dig." You should also anticipate the player will type the single word "dig." In this case, the action "dig" should be included in the vocabulary file. A relatively large block of code is necessary to take care of the action "dig."

Rather than duplicate the "dig action" block of code four times, once for the sand, once for the ground, once for the grave, and once for the single word command "dig," place the code in a subroutine and call the subroutine each time it is required. Give the subroutine an easy-to-remember name like "DIG." In the object file for "sand," under the action "dig" you only need to "CALL DIG." Likewise in the object files for "ground" and "grave" you only need to call the subroutine. And finally, in the vocabulary file, if the player types the single word "dig," you again "CALL DIG." By using a subroutine in this example, you have saved a large amount of memory.

Nested Subroutines

You can call a subroutine from within another subroutine. In programming, this is called **nesting**: the call for the second, nested subroutine is included in the subroutine code activated by the first subroutine call.

Perhaps there are several different locks to be opened in your adventure. As the player discovers the special method to open each lock, you call a subroutine. The subroutine would increment a variable, to tell you how many total number of locks have been opened. It would print out a message, letting the player know he succeeded in opening a lock. And it could call another subroutine, one which would play the digitized sound of a key turning in a lock. In this case, a subroutine has called another subroutine.

Avoid jumping out of the subroutine

You can nest subroutines in this fashion up to 128 deep. That's a limit which you will likely never even approach. Be careful, however, not to jump out of the middle of a subroutine. For example, never go to a room from within a subroutine. Always exit with the "ENDSUB" command.

A common use of subroutines is when you have an game action which the player can phrase in two different ways. You would place the code for that action in a subroutine, rather than duplicating the entire block under each different way of phrasing.

For example, "remove the hat" and "take off the hat" are two different ways that the player could ask to do exactly the same thing. Rather than write the necessary code for removing the hat and placing it both under the "remove" action and the "take" action, call a subroutine from both those places. If you use this method consistently in your adventure, you will save an appreciable amount of memory.

Let's look at a very handy subroutine that checks to see if a room is dark or not. There are times when some of the rooms in your adventure will be in the dark. Not all the rooms will be dark, just some of them.

Let's say you have designed an island that has a volcanic mountain on it. The player can visit a cave in the mountain. When the player is outside, it is light and she can see. When the player enters the cave, it is dark and she cannot see unless she has a lit torch. When you design the files for each room, you need to decide whether to print the location description, or print "You can't see in the dark." The following VISIONARY subroutine accomplishes this task.

```
SET THISROOM, DARK
IF PLAYER HAS TORCH  OR  TORCH IN THISROOM THEN
 UNSET THISROOM, DARK
ENDIF
```

Notice that the IF statement is in two parts. The room is first set to dark. If the player carries the torch, it is then considered lighted. If the torch is in the room but not carried, the room is also considered lighted. Otherwise, the room remains dark.

The above subroutine is placed in the code section of any room file that should be dark. Whenever the player enters a room, the code section of the room file is executed. You would call this subroutine first, before you describe the room. After the subroutine has been executed, then describe the room as the player will be able to see it, if the DARK attribute is unset. Describe the room as "You can't see in the dark." if the DARK attribute is set. By calling this subroutine, you can easily determine which normally darkened rooms should be described and which should not.

» A subroutine which tests for darkness would not placed in a "room" location which would normally be lit at all times. For example, the above subroutine is not used to test for darkness at any of the outdoor locations on the island in the sample game, but is needed only in the cave locations.

You will find a variety of uses for subroutines as you complete work on your game. The general rule is to use a subroutine for any piece of code that is needed more than once. It will pay off in time and effort saved, as well as making a more memory-efficient game.

Chapter 9: Automatic Actions

Automatic actions are those pieces of code that need to be executed after each move the player makes. By using automatic actions, you can easily keep things happening in the background of your game. In a text game, you can modify the way the text is formatted with automatic actions. You can even keep track of timed events in your adventure. This chapter will examine these and other ways that automatic actions can be used in your game.

Automatic Events

There are times when you want your adventure to take certain actions, regardless of the player's input. Instead of waiting for the player to type a certain command or click on a certain item, you want the game to act in a certain way no matter what the player types. It doesn't even matter if the player's input is understood, you will want certain things to take place. Even if the game responds, "I don't understand that," you want some special actions to continue uninterrupted.

Time is a good example. You want time to continue regardless of what the player does. It doesn't matter if your timer is a sophisticated one that marks hours, minutes, and AM/PM, or if it is a simple turn counter. Whatever type of clock you have designed, you want it to continue incrementing on each turn, regardless of any player actions. You want this action to automatically take place on every turn, hence the name **automatic action**.

Automatic actions and events that happen no matter what the player does are also "characters" in your game, defined in the NPC Files

Automatic actions should be placed in the non-player character or **NPC file** sections of your program. It is here that they will be executed after every turn. NPC files are similar to normal object files. They include title, name, attributes, initial room, code sections, and actions. The basic difference between these files and the usual object files is that they are executed after every turn. Object files, on the other hand, execute only when each particular object is in the player's current location or in his inventory. The particular importance of NPC files is that the player's location is disregarded, and the files are executed on each turn. This makes them invaluable for holding actions that you want constantly executed.

Counting Time

Since time is the most frequent use of NPC files, let's examine a simple timer that you might use in your adventure. Keep in mind that Visionary already uses a simple timer that keeps the number of turns updated in the MOVES variable. Each time the player makes a move, the vari-

able MOVES is incremented automatically. The necessary code is not placed in an NPC file, it is all done internally by Visionary. If this is the only timer you need, then you don't need to create a special one.

However, you will frequently find the need for special timers. You may want one that counts backward. You may want one that starts counting after a certain action has taken place. You may want one that keeps track of "fake" hours and minutes, one minute per move. Let's see how to implement these ideas in an NPC file.

Counting Backward

When you design the NPC file that will hold your timer, choose a title you will easily remember. Usually you will want to use an identical title and name like TIMER. Place the NPC in the storage room location that you have created for unused objects, a room which the player never visits.

For the purpose of automatic actions, there is no need to define attributes. All these actions will take place in the CODE section of the file to ensure they are executed upon each move the player makes. As such, there will be no actions following the CODE section. The CODE section itself, however, will be large. It will contain all the timers and other automatic actions for your game.

There is no need to define more than one NPC file. All the automatic actions for your entire game can be placed in the CODE section of one NPC.

Let's see how a timer that counts backward would work. You will be using a variable, which we will call TIMER. Be sure you define this variable at the beginning of the game, in the ADV file. Since TIMER will count backwards, you will have to choose at what number you want it to start, and include that in the definition at the beginning as well. Let's say that you will give the player exactly 100 moves to escape from a maze before dying. In that case, you must define TIMER to be 100 in the ADV file. In the CODE section of the NPC file, you would place lines as shown below.

```
TIMER := TIMER - 1
IF TIMER = 10 THEN
 T You only have ten minutes left.
ELSIF TIMER = 5 THEN
 T You will die in five minutes!
ELSIF TIMER = 0 THEN
 T AARRG!  You are dead.
 QUIT
ENDIF
```

As you can see above, the timer is decremented, each time this section of your program is executed. And since this is in the CODE section of a NPC file, it is executed after each move the player makes. On each move, the variable TIMER is reduced by one. The player dies when it

reaches zero, after a warning at ten and again at five. Perhaps if the player escapes the maze before the timer reaches zero, he wins the game. Or perhaps he goes on to other challenges. In this case, you should set the variable TIMER to negative one, so it will never reach zero and kill him after he has escaped.

Timers can be used to limit total number of moves allowed

The basic idea of a timer that counts backward has many uses. Perhaps you wish to keep your game limited to a maximum number of moves. In this way, part of your adventure puzzle is not only to find some treasures but to do so within a set time limit. If you allow the player 200 moves to find five treasures, you have added to the challenge of the game.

You may also wish to limit the number of moves the player can make, to ensure no player will be able to finish the game. As mentioned in the chapter on variables, you may want to submit a partially working copy of your finished game to a publisher. By choosing a time limit that is too small to complete the game, you allow the potential publisher to test your game and try it without being able to complete it. This concept can also be used for share-ware demos.

The basic idea in all the above is to limit the number of moves the player can make when playing your adventure. Regardless of the reason, the player is only permitted a certain number of turns before the game terminates. To do this, you place a decrementing timer in an NPC file.

Let's look at another timer that counts backward. Suppose your player is swimming in a lagoon, and can dive beneath the surface to explore the lagoon's bottom. You want to create a timer that will keep track of her turns beneath the water, since she can only hold her breath so long.

Let's say we have two locations, SURFACE and UNDERWATER, and a variable BREATH. Both SURFACE and UNDERWATER will be defined in the room files, and BREATH will be defined in the ADV file at the beginning of your game. The following Visionary code permits the player to stay underwater for six turns.

```
IF PLAYER IN UNDERWATER THEN
 BREATH := BREATH - 1
 IF BREATH = 2 THEN
  T You are running out of air.
 ELSIF BREATH = 1 THEN
  T You better surface!
 ELSIF BREATH = 0 THEN
  T You die from lack of oxygen.
  QUIT
 ENDIF
ELSE
 BREATH := 6
ENDIF
```

Notice that this section of code only executes when the player is underwater. When the player is not underwater, BREATH is made equal to

six, she breathes normally, and most of this code is skipped. When the player moves underwater, you start decrementing BREATH. The player dies after six turns underwater. Notice that two warnings are also given before the player dies. As mentioned frequently, warnings are vital to any adventure where the player can die.

Special-Circumstance Counters

Some timers operate only in specific conditions

The concept of a timer that only counts under certain circumstances is a valuable and frequently used one. It is considered an automatic action, even though it doesn't count automatically on every turn. The important thing is that it counts automatically under certain conditions. In the above example, the timer counted only when the player was under water.

There are many other possible uses for this type of counter. You may design your adventure character to be claustrophobic who can only spend a certain number of turns inside a cave before going mad. Maybe the player accidentally drinks poison and has only twenty turns to find the antidote. Perhaps your player has cast a magic spell that wears off after ten turns. Or the quick energy of a candy bar may wear off after eight moves.

In each of these examples, you will set up a timer in an NPC file and decrement a variable only under certain conditions. You only decrement the variable if the player is inside the cave, or has drunk the poison, or has cast the spell, or eaten the candy bar.

Counting Forward

The types of timers discussed so far have been decrementing ones. They count backwards. The first one described, constantly decremented as the game progressed. The second type decremented only under the condition that the player was in a certain location. The specifics of each of the previous routines above can be modified so that the timers count forward, with similar results. Whether you design the timer to increment or decrement, the concept remains unchanged. It is an automatic action that must be placed in an NPC file.

Let's see how a different type of counter can be designed. This counter will still be automatically incremented in the NPC file, but will do more than just count turns. It will keep track of hours, minutes, and AM/PM. In the following example of Visionary code, we need three variables: HOUR, MINUTE, and AM. Each of these must be defined in the ADV file at the beginning of your adventure. They will each start at zero. Now look at the routine to keep track of time.

```
MINUTE := MINUTE + 1
IF MINUTE = 60 THEN
 MINUTE := 0
 HOUR := HOUR + 1
 IF HOUR = 13 THEN
  HOUR := 1
  AM := 1 - AM
 ENDIF
ENDIF
```

In the above routine, a minute is added to the clock for every move the player makes. When the minute hand reaches 60, it increments the hour hand and reset the minute hand to zero. When the hour hand reaches 13, it resets to one and changes the AM flag to one if it was zero, or zero if it was one. We may decide to let it be morning when AM equals zero, and afternoon when AM equals one. That way, if your adventure starts with all three variables equal to zero, then the game starts at midnight.

The clock routine shown above automatically ticks away invisibly on every turn the player makes. A clock like this isn't much use by itself. It serves no purpose unless somewhere within the game, the player is appraised of the time. Perhaps your adventure provides a pocket watch that can be examined. Perhaps there is an old clock in the town square that can be seen. The way you can use the results from a clock routine to report the time to the player in a text adventure is shown below. When the player asks to look at the clock, your game can execute this code.

```
IF AM = 0 THEN
 T The time is @hour   : @minute   AM.
ELSE
 T The time is @hour   : @minute   PM.
ENDIF
```

Notice that two cases are necessary, since the time can be morning or afternoon. The "@" sign is required to print out the value of the variables "hour" and "minute". See Chapter 7, Messages, for more details.

In a graphics game, the above code could be replaced with lines which will draw a clock on the screen and update the hands on the clock. Since having so many different drawings for the hands in different positions would take up a lot of space in your graphic file, you might prefer to use a digital clock. In this way, you could draw the clock face on the screen, and then using Visionary's TEXT command, write the digits on the face of the clock. The second method would take much less memory and be easier to program. But the first method would be more appropriate in a game set in a time period before the invention of digital watches. It's a matter of taste and programming ability.

Timed Events

If you plan on having certain events occur in your game at specific times during the day, use a clock routine. If you want a corpse to rise from the grave at midnight, place the code in the same NPC file, after the clock routine. After the clock has incremented, check to see if all three variables are zero. If so, midnight has struck, and it is time for the corpse to rise.

At this point, you may want to print some warning message to the player, or place the object of a moving corpse at the graveyard location. You could even start a new timer into action, counting the moves until the corpse stalks and eventually catches up with the player. Since you want all this action to take place specifically at midnight regardless of what moves the player is in the process of making, you must place the code for these actions in an NPC file. They are automatic actions that you want to take place automatically at midnight, regardless of what else is currently happening in the game.

Many different events can be programmed to take place automatically, depending on your clock. That's why the concept of a clock or timer is to vital to most adventures. In a text game, informational messages can appear at different times of the day. At six o'clock in the morning, you can print "It's dawn." At noon, you can remind the player that, "the sun is directly overhead." And when it's eight at night, you can say, "the sun is slowing sinking in the west." In a graphics game, the same thing can be done by changing your graphics to show the daylight passing. By checking the clock, you can create messages and graphics that reflect the time of day.

Random Counts

You may wish to place random messages in the NPC files. These are generally messages that have no bearing on the actual game play of your adventure. They don't help the player in any way. They don't affect any objects, locations, or actions. Rather, they are presented to create a more realistic atmosphere in your make-believe world. You may wish to tell the player "you hear the crackle of a campfire in the distance." Perhaps "there is the shadow of a bird passing over your head."

Messages such as these do a lot to add to the tone of your adventure. They are controlled by setting a variable with a random number and using a counter to decrement the variable. Examine the following example.

```
COUNTER := COUNTER - 1
IF COUNTER = 0 THEN
 T A cricket chirps away merrily.
 COUNTER := RAND / 500
ENDIF
```

When you define your variables at the beginning of the game, you might assign COUNTER to be 15. With every move the player makes, this routine is automatically executed and the variable is decremented. When the counter reaches zero, the message is given and the counter is randomly reset to a number between 0 and 19. Remember that RAND will always generate a random number between 0 and 9999, so dividing by 500 yields a whole number between 0 and 19.

Random messages can be used to indicate time of day, or give other information

The above example is intentionally kept simple. You would probably want to select one message from a variety of messages. You might also want to ensure the random number is not too small to prevent the messages from coming too close together. You might want to further check the clock to print random messages that reflect the time of day. If the clock indicates it is dark, you might use a random message like, "in the distance you hear a coyote howling at the moon." If it's daytime, "a bird's chips fill the morning air." If it's around noon, you might tell the player "the hot sun bakes down from directly above."

Each of these messages make the your adventure fuller and richer in detail. They add a touch of realism to your world and make the game more interesting to play.

Many automatic actions are not dependant on your clock routine. But they are frequently dependant on their own timer. An example previously mentioned was that of the corpse that rises from its grave at midnight to chase the player. One way of handling the chase is with a separate timer. As the timer increments automatically after each turn, the corpse gets closer and closer until either the player destroys it or it destroys him. You could set up the timer as shown below.

```
IF CORPSE > 0 THEN
 CORPSE := CORPSE - 1
 PLACEOBJ CORPSE, THISROOM
 IF CORPSE = 9 THEN
  T The corpse is after you!
 ELSIF CORPSE = 3 THEN
  T He's nearly upon you!
 ELSIF CORPSE = 0 THEN
  T He's got you!  You're dead!
  QUIT
 ENDIF
ENDIF
```

In this example, we have a variable and object with the same name. The variable CORPSE is normally zero. When it is in that state, the routine is skipped on each turn. But when it is midnight and you bring the corpse to life, you set the variable CORPSE to 10. At this point, the above routine will be executed automatically on every turn and allow the player nine moves to destroy the corpse.

Notice that the object CORPSE will follow the player from room to room, as a result of the PLACEOBJ command. Placing the object in the room with the player is important so that the object can be referred

to. The player can say "look at the corpse," "talk to the corpse," "hit the corpse," or "destroy the corpse." None of these commands will be understood if you fail to place the object CORPSE in the current room. If the player fails in destroying the corpse, he dies on the ninth turn. If he succeeds, you must set the variable CORPSE to zero as part of the action that destroys it. In that way, the above routine will no longer be executed. If you forget to zero the variable, the player will be killed even though he destroys the walking corpse.

An interesting variation on the above situation is having the corpse continue to stalk the player without any time limit. As long as the player keeps moving and stays out of the corpse's way, he will remain alive. If he stops in any room and allows the corpse to catch up with him, he will be caught and the adventure will end. Examine the following Visionary code and see how this is accomplished.

```
IF CORPSE IS CHASING THEN
 IF CORPSE IN THISROOM THEN
  T It has caught you!  You're dead.
  QUIT
 ELSE
  T Look out!  It's coming!
  PLACEOBJ CORPSE, THISROOM
 ENDIF
ENDIF
```

In this routine, we have used an attribute for the object CORPSE that was defined in the object file as CHASING. When the corpse is brought to life at midnight, this attribute is set. The above routine is only executed if the attribute is set. Before moving the corpse into the player's current room, check to see if it is already there, indicating that the player was in this room during the last move. If so, the player is caught and dies. If not, the corpse is moved to the current room in preparation for the next turn.

Special Uses for Automatic Actions

Clocks and counters are not the only uses for automatic actions. With a little forethought, you can find many uses for actions that occur independent of the player's input.

One-Time Events

Some messages should not be random. You want them to appear only once, at a specific time. Perhaps when the player first enters a room, you want to tell her "You smell something delicious coming from the kitchen." This can't be put in the room description, because it would appear more than once. Make it an automatic action, and keep it from repeating by using a flag.

```
IF PLAYER IN DININGROOM  AND  SMELL = 0 THEN
 T You smell something delicious.
 SMELL := 1
ENDIF
```

As usual, be sure to define all variables like SMELL first in the ADV file. In the above example, the message will be given when the player enters the dining room, and has not seen the message. After it is given, the variable SMELL is set to 1 so that it will never be printed again.

Text Formatting

A special use of automatic actions is in text formatting. Let's say you are writing a text adventure and want to place a blank line before the player types her line of input. The easiest way is to place it at the end of the NPC file. After everything else in the file is done, and all messages given, the blank line will be printed just before the program returns to the player's control.

If at times you want the blank line suppressed for various reasons, use a conditional statement and a variable. The following shows how this can be easily accomplished with Visionary.

```
IF BLANK = 1 THEN
 BLANK := 0
ELSE
 T
ENDIF
```

In this example, the variable BLANK is normally zero, and a blank line is normally printed. If for some reason you wish to suppress this for one turn, simply set the variable to one in the appropriate place. When this routine is encountered at the end of the NPC file, the blank line will not be printed, but the variable will be reset to zero for the next turn.

Single-Location Movable Objects

Occasionally you may want an object to be manipulated only at one location. You may want the player to be able to pick up the object, but not take it to another location. Let's say the player is at a shooting gallery. A rifle is chained to counter. The player can pick it up and put it down, as well as shoot with it. But it's chained down, and the player is not allowed to leave the shooting gallery with it. Use an automatic action to make sure that if she leaves, the rifle doesn't leave too.

```
IF PLAYER IN GALLERY THEN
ELSE
 IF PLAYER HAS GUN THEN
  DROP GUN
  PLACEOBJ GUN, GALLERY
 ENDIF
ENDIF
```

If the player is in the gallery, this routine does nothing. If she isn't in the gallery, you check to see if she carries the gun. If so, you make her drop it and place it back in the gallery. The command DROP GUN is necessary to keep the inventory variable ITEMS accurate. Without this command, the gun would still be put back in the gallery, but ITEMS would not be decremented properly.

Automatic Creation of Objects

Let's look at another use for automatic actions. In your adventure, perhaps you want a good fairy to appear after the player has found ten treasures. Since you don't know which treasure will be found last, you can't place the code to create the good fairy in the object file for a treasure. Use an automatic action in the NPC instead. Each time a treasure is first found, set an attribute like FOUND and increment a variable like TREASURES. Only increment the variable if the FOUND attribute is unset, to prevent the variable from being incremented additional times as the player repeatedly picks up and drops any one treasure. Then make the following an automatic action in an NPC file.

```
IF TREASURES = 10 THEN
 T A good fairy appears before you.
 PLACEOBJ FAIRY, THISROOM
TREASURES: = 11
ENDIF
```

This section of code will be checked after every move the player makes. When all ten of the treasures have been found, you print a message to the player, place the object of a fairy in the current room, and change the variable TREASURES to eleven so that the routine will not be executed again and again on every successive turn.

Automatic Movement of Objects

In some cases, objects must be moved from room to room automatically. Automatic actions are quite useful in these situations. Let's say the player finds a vase with a magic ring in the bottom of it. You must have two objects defined, VASE and RING. In this way, the player can remove the ring, discard the vase, or even wear the ring. But as long as the ring remains inside the vase, you are presented with a problem. When the player gets the vase, you don't want to add the ring to his inventory. Yet, when he moves from one room to another, you want the object of the ring to follow the vase.

This is important. The ring must be in the same location as the vase, or else when the player asks to "get the ring" he would be told "I don't know what a ring is." This is due to the fact the only objects that are in the player's location or inventory are checked for actions such as "GET." For these reasons, you want the ring to stay in the same room as the vase, as long as it is inside the vase. The problem is easily solved

by defining an attribute for the vase called "FULL" and placing the following routine in the NPC file.

```
IF PLAYER HAS VASE  AND  VASE IS FULL THEN
 PLACEOBJ RING, THISROOM
ENDIF
```

Notice that this routine is only activated when the player carries the vase and the vase also carries the ring. If both situations are true, then the ring is placed in the current room. As long as the vase is full, you will want the ring to remain invisible, and not be listed among the items that can be seen at this location. To do this, place the following lines in the code block of the object file for the ring.

```
IF PLAYER HAS RING THEN
 T a gold ring
ELSIF VASE IS FULL THEN
ELSE
 T There is a gold ring lying here.
ENDIF
```

Remember, the code block of the ring file is only executed under two circumstances, when the player takes inventory and when the room containing the object is described. The lines in the above example take care of all three of these situations. The ring will be described if the player takes inventory. The ring will also be described if it is in the room with the player, but not if the vase if full.

You've been given many different examples of automatic actions. You've seen timers that count forwards and those that count backwards. You've seen timers that count every move, and those that count only under certain circumstances. You've seen actions that have been initiated by counters, and counters that have been initiated by actions. You've seen how blank lines can be inserted into the text automatically. You've seen how objects can be automatically created and moved from one location to another. These are only a few of the many ways you can use automatic actions in your adventure game. The more you use them, the more uses you will find for them.

Chapter 10: Sounds and Pictures

Until now, little has been said specifically about graphics and sound in your adventure. Perhaps you want to design pure text adventures, like the old Infocom-style games. If this is your intention, you may want to browse quickly through this chapter, since most of it doesn't apply to pure text games.

But in today's market, more and more adventures support graphics and sound. You are encouraged to use both in your game, even if in the most rudimentary sense. You will find they can enhance almost any adventure. This chapter will examine various ways to use graphics and sound in your game, from the simplest to the most sophisticated.

Basics of Game Graphics and Sounds

At their simplest, graphics and sound can be viewed as icing on a cake. They can make your game look more appealing and play nicer. You can design a graphic screen for each room in your game, and display it either when the player first enters the room or whenever it is specifically requested. You can design a main title screen, and have music playing while it is displayed. You might include some sound effects to be used at appropriate times. This by itself, might be all you need in the way of graphics and sounds in your game. But more is available, if you wish. Before we look at the more sophisticated uses, let's start with the basics.

One of the first things you need to do is get a paint program and learn how to use it. This can be a time-consuming task. Don't expect to master the techniques of creating computer art in a weekend. If you are not an artist, don't despair. Work together with others who have more artistic ability. Let the artists know what you want, and work with them to design the graphic screens the way you want them to look.

If you don't know any computer artists, contact your local computer club or user group. They frequently have special interest groups for computer art, and can put you in contact with someone who has the skills you lack. Another possibility is your local computer store. They may know of some budding computer artists who would be willing to work with you on your adventure. Additionally, you may wish to leave a call for art work on a local BBS or on larger telecommunications services like PLink, BIX and CompuServe. Whether you do the art work yourself or have others do it, the next step is to add the graphic screens to your adventure.

The Title Screen

Title graphics create the player's first impression of your game

Let's consider the title screen. This is the first thing the player will see when starting your adventure. It should have your game title in the foreground along with your name as the author. The background should be something generally descriptive of your adventure. If your adventure takes place in a haunted castle, you might wish the graphic to show a night time scene with a castle sitting on a high cliff with a full moon shining down on it. If your adventure takes place in a jungle city, you could show the vine-covered stone walls of a lost city. Choose your background carefully, as this title screen will tell the player a lot about your game before play even starts.

Pick an effective way of displaying the title and the credits over the background. If you wish to have the title screen showing while your Visionary game loads, you can use the utility program **LOADSCREEN** that comes on the Visionary disk.

In addition to showing a standard IFF graphic screen, this utility also allows color cycling which you can use for various effects including simple animation.

You may wish to have the title fade onto the screen, then scroll upward as your other credits appear. To do this, you will have to let the Visionary game load, and then take over the scrolling of the credits. Scrolling credits give your game a nice professional look, and can be easily accomplished using the Visionary graphics commands.

Game credits should include your name, as well as the names of any others who helped in the creation of your adventure. If an artist assisted you with the graphics, include the artist's name. If others helped you with the story, with play testing, or with digitizing any sounds, be sure to include them as well.

When the credits are completed, you may wish to allow them to scroll off the top of the screen. You could fade them out. Or use your imagination and design your title screen in other ways.

Animation is another thing to consider. You may wish to have animated titles or backgrounds in your opening screen. Luckily, Visionary makes it easy to do all of these things. With a full assortment of graphics commands, Visionary allows you unlimited ways to create your graphics screens.

Give the player a way to skip the opening credits

Although you are justifiably proud of your opening graphics and want your name to be seen in the credits, don't force the player to sit through the entire opening sequence every time your game is booted. Allow the sequence to be aborted in the middle, so the player can jump directly to the beginning of your adventure.

Think about other games you have played. Most allowed you to skip all the opening credits and get down to the business of playing the game. You may have run across a few that forced you to wait until all

the credits were finished, and you undoubtedly found it very frustrating. Don't force the players of your game to view all the credits. Allow them to abort by pressing either a mouse button or a key on the keyboard.

Title Sound Effects

A silent title screen is a dead screen. You need some music or sound effects playing in the background. With Visionary, you can play music several different ways. The music can be digitized, and you can have your game play the sound sample. Sound samples take an extraordinary amount of memory, and thus are not recommended for title music.

Visionary also supports songs written in the **MED** format. MED, which stands for Music Editor, is a freely-distributable program that allows you to create music for your Amiga, by using an electronic keyboard and MIDI interface, by using the Amiga keyboard as a music keyboard, or by entering musical notes.

You can probably find MED on a local BBS, as well as most sources of freely-distributable software like the Fred Fish disks. Visionary has a built-in MED player which is superb for title music or any other music needs in your adventure.

If for some reason you wish to use a different music program, you can use it by calling it from Visionary's external DOS function. You can use this DOS function to call a SMUS player or other music player.

If you are competent to create your own music, MED will produce a nice score. Otherwise, you may want to enlist the aid of a musician who can assist you in creating the music for your adventure. As mentioned above, if you need to find someone capable of writing music on the Amiga, your local user's group, BBS, or computer store should get you started.

Game Graphic Screens

At the very least, you will need three graphic screens in your adventure. In addition to the opening title screen, you will need a "win" and a "lose" screen.

When the player falls into a death trap, she loses and the game is over. At this point, you will want some type of graphic screen. Perhaps you will design a simple skull glowing in the dark, and play a funeral dirge in the background. You may want to extend it further with animation. Perhaps the skull will slowly smile. But at the simplest, you need to let the player know she has died.

You also need a "win" screen which your player will see when he successfully completes your game. If you want something simple, use the same graphic as the opening screen and superimpose "THE END" in large letters. Compose some upbeat music to play over the graphic. If

you want to get more complicated, feel free to design a completely different screen, animate it, or add some digitized sound effects.

These three screens—title, game won and game lost—are important. The only time they should be skipped is if you plan on a strictly text adventure with no graphics at all.

You may be satisfied with only three graphic screens in your adventure, or you may wish to expand the graphics further. The next logical step is to add graphics for each room in your game.

Every time the player enters a new location, a graphic showing what he sees should be available. This graphic screen should include all the nonmovable objects as well. Don't include movable objects, since the player may very well pick them up and remove them from the room.

In a text-and-graphic hybrid game, generally show only nonmovable objects in the graphic room screen

The exception to this rule is if you plan on having an entirely graphic-oriented game, where the player can point the mouse and click the button to pick up an object. This is certainly possible to do with Visionary, as the sample game that came with this book illustrates. Although it is possible, it is much more sophisticated and requires additional programming skill. If all you want is a graphic scene pictured for each location the player enters, follow the general rule that suggests you only show nonmovable objects in your graphic room screens.

Don't show a room graphic each time the room is visited. Remember it takes a small amount of time to load the screen from the disk. As the player moves through previously visited rooms, it would slow the game down appreciably to force him to see the graphic each time he passed through the room. The best way to display room graphics is to show them when the player first visits a room. Thereafter, do not display the graphic when the player enters the room.

Display the room graphic only on the first visit

But you should also provide a command to see the graphic again, if for some reason the player wishes to. A typical command to accomplish this is "VIEW." You could use "LOOK" for this purpose, but that verb is usually reserved for the text description of a room. It is usually best to have two separate verbs, one to redescribe the room location in words, and one to re-display the room scenery graphics. As the game designer, you have to make that choice.

Sounds and Music

Sound can be a true enhancement to your adventure. Music during the opening and closing titles is most effective. Usually you will not want music playing during the game, but only during the opening and closing titles. If you choose to have music in the background during the game, lower the volume so it blends in better. Additionally, digitized sound effects can add a lot to your game.

Sound effects can be used in many places in your adventure. When the player opens a door, you might want the sound of squeaking hinges. And the sound of a door slamming shut would be appropriate if the

player asks to shut the door. If the player finds a whistle, he should hear something if he tries to blow it. Likewise if starting a car, he should hear the engine. And he should hear the horn honk if he tries that action. There are many actions which could have sounds associated with them.

Some sounds will not be connected to any specific action by the player. Some will be used in the background regardless of the action a player takes. If the player stands in an old study, you might want the sound of an old grandfather clock ticking in the background. If the player is on a sandy beach, the sounds of waves crashing on the shore should run in the background. A background of birds chirping would be appropriate for a meadow, and machine noises would be excellent if the player was in a factory. Whether the sounds are part of the location background, or are in response to some player action, you should add digitized sounds to your game whenever memory limitations make it possible.

Use Visionary commands to manage sounds

Visionary has a full range of commands to manipulate and play digitized sounds, making your job easy. You can load up to 50 different digitized sound effects into memory at once. You can then choose to play any one of them, through either the left or right channel. You can determine how many time the sample plays, from once to continuously.

The volume is also at your control, which can be a very useful feature. If the player in your game is standing on the beach, you may have the sound of waves playing continuously in the background. As she walks inland, away from the ocean, you can reduce the volume of the crashing waves. And when you wish, you can completely turn off the sound of the waves.

If you don't know much about digitized sounds, you don't need to worry. They are generally straight-forward to work with, and require a lot less time and effort than graphics or music creation. You can obtain digitized sounds from a variety of sources, including library disks from the Visionary Users Group, local BBS's as well as public domain disks.

You can even make your own sound effects, if you purchase a sound digitizer. A wide variety of sound digitizing hardware is available for well under $100. Digitizers are easy for a beginner to use, and will allow you to capture any sounds you wish. You can digitize live sounds, or choose sounds from TV, video tape, or records. Check your local library for an excellent assortment of sound-effects records, which you can borrow and digitize.

Memory Use

As you use graphics and sounds, watch the memory usage of your program. It is easy to create a program which requires large amounts of memory. If possible, you will want your program to run on any 512K Amiga. In this way, it is playable by many more people.

A second of digitized sound can take up to 10K of Chip RAM to play

To achieve this limit, you may have to curb your graphics or sounds. Try to keep as few graphics screens in memory as possible. Each one can easily eat up 50K or more of memory. Watch the length of your digitized sounds. These types of sounds are the most impressive and realistic, but also require the most memory. A second of digitized sound can take nearly 10K of memory. You could easily fill 100K with a single ten-second sample.

If you are digitizing your own sounds, you can alleviate this problem by changing the sampling rate. Unfortunately, this lowers the fidelity of the sound and generally is a poor solution. A better solution is to keep the samples short, and repeat them. A single chirp of a bird will take less than one second. Repeating the one second sample at random times gives the effect of a bird chirping without the additional memory consumption. And to save as much memory as possible, always free up a graphics or sound buffer when it is no longer needed.

Although adding digitized sounds to your game makes it much more enjoyable to play, you should keep close track of the amount of chip memory you have available. Remember that all graphics and digitized sounds must reside in chip ram. If you want your game to be playable on all Amiga computers, then you must design all the graphics and sounds to work with a half-megabyte of Chip RAM. Admittedly, some Amiga computers have a full megabyte of Chip RAM, even two megabytes. But to be compatible with as many Amiga computers as possible, you should choose the lowest common denominator, a half-megabyte of Chip RAM.

Use the CHIPMEM variable to monitor the available graphics memory

The best way to keep track of how much chip ram you are using, is to use Visionary's built-in variable **CHIPMEM**. As you are play-testing your game, you can have the value of CHIPMEM be printed out either continuously, or when you issue a special command. By closely monitoring the amount of chip memory you have left, you can design your game to use as many graphics and sounds as possible, while still maintaining full compatibility with all Amiga computers.

Mouse-Responsive Graphics

Let's examine a more sophisticated use of graphics. Imagine a screen that is mostly graphics. There are several lines at the bottom of the screen for the input of text, but the majority of the screen is taken up by a room graphic. On the right side of the screen is a compass and a list of common verbs. In such a game, the player could use the mouse to click on the compass to go north, instead of typing "GO NORTH." The player could use the mouse to click on the verb "EXAMINE" and then move to the room graphic and click on the picture of a sofa, instead of typing "EXAMINE THE SOFA."

In each case, the adventure would respond in the same manner as if the player had actually typed the words. This type of adventure is possible with Visionary. In fact, the sample Visionary game that comes

with this book is designed in this fashion. Admittedly, it takes more work to write this kind of adventure, but it can be done if you are willing to spend the time and effort. The second half of this book will examine exactly how to create such a game in great detail, and explain many special techniques to make the creation easier.

An adventure such as the one described above, or the game which results from the source code in this book, is more graphic than text, but is becoming quite popular among game players. If you choose to change a mainly text adventure to this type of game, you will have to go back and make extensive changes to your program.

You will first have to create the room graphics, then go back and add commands that check to see if the mouse has been clicked on one of the fifty allowable zones within the graphic screen. These zones could be compass buttons, buttons that say HELP or INVENTORY, or they could contain part of your scenery graphics. If your location graphics contain a sandy beach, the ocean, and the sky, you could have a zone for each.

In this way, if the player clicks anywhere on the beach, your game will know it and can respond with a description of the sand. Likewise, if the player clicks anywhere on the water, you may want to know that and have your game respond in some way. Naturally, you won't know exactly where these zones are until the graphics are completed. You won't know the exact location of the beach, or the sky, or the HELP button on the screen until it has been drawn.

After the graphics are finished, you can measure the location of each object on the screen and write the routines for the mouse to click on them. It takes a lot of extra work to create this type of adventure. Visionary takes as much of the work out of it as possible, but you still have to create the graphics and decide which areas on the screen will give some response when clicked upon. It won't be easy, but the purpose of this book is to make it easier. You can see a sample game, with the same features as described above. You can look at the source code and see just how it was accomplished. It may take extra work, but the end product will be much more professional and will be worth all the effort.

Both graphics and sound can add a lot to your adventure. That's not to say they are required. Pure text adventures have been around for years, and continue to be enjoyed by adventure gamers all over the world. If you choose to enhance your game with graphics and sound, you may wish to start cautiously with only a minimum of screens and sounds. After you have been successful, you may wish to increase the graphics and sounds in your game. You could even create a game that requires no text input at all! With Visionary, time and talent, you can do wondrous things to your game.

Chapter 11: Dungeon Adventures

This chapter will take a look at how you can design a totally-graphic dungeon adventure similar to Dungeon Master™ and other maze games. These are very popular types of games, and can be easily programmed using Visionary. We'll see how to use some shortcuts in graphics to make the game take less memory and run faster. And we'll examine the methods you can use in creating your characters, their personal attributes, and the interaction between them.

Game Graphics

Let's start with the graphics that show the dungeon tunnels. These location scenes can be created in memory from a set of tunnel pieces, which means you can save a lot of disk space and the game will execute faster. If you have a 400-room dungeon, it would take many disks to contain all the location scenes if the scene for each room was saved in a separate disk file.

Not only that, but it would take you a very long time for you create all those scenes with your paint program. And the resulting game would play sluggishly, since each time the player moved to a new room in the dungeon, a new file would have to be loaded from the disk.

A much better way to design your dungeon game is to create a few graphic files with tunnel pieces and then combine them in memory to create each dungeon room. You might want to create a series of walls in one file, showing variations of openings and solid walls on both the left and right side of the tunnel. You might also have several different scenes of the ceiling, one unbroken, and another showing an opening for a staircase. Similarly, you might design several different scenes showing the floor of the tunnel, one showing an unbroken floor and others showing stairs leading downward in different directions. These pieces can then be combined in any way you want to create the proper look for each room in the tunnel system of your adventure.

Hidden Screens

When the game is being played, the pieces can be put together in a hidden screen and then be displayed on the visual screen that the player sees. This is easily done with Visionary's **COPY** command. Your game can run through a series of conditional statements that build the floor, the walls, and the ceiling of the current room based upon what doors and staircases are present. Each piece can be copied onto a blank space, as the scene of the room is built up. Then when

the room scene is finished, you can copy the final result to the screen for the player to see.

With COPY, objects appear, room graphics shift as the palyer moves

Graphic copying is executed within Visionary very quickly, and the player will notice only the smooth movement of your game. A technique known as **double buffering** is used to eliminate any trace of flickering that would otherwise occur during the copy process. Double-buffering is explained below.

Let's next look at how to create the maze of tunnels that are always found in dungeon-type games. A simple approach is to use an array. This array can keep track of not only which rooms lead where, but also which rooms contain monsters, weapons or treasures.

You can use Visionary's "pixel" and "rectangle" commands to simulate the reading and writing of the array elements, then the actual array itself will be displayed as a picture on a graphic screen. It will picture the maze as a bird's-eye view. In this overhead view of the maze, each pixel represents a single room in the maze. You can read the pixels surrounding the current room, to see which directions are open to the player.

In addition, you can have the color of the pixels tell you things about the room, such as which monsters, weapons or treasures are in the room. In that way, when the player moves into a new room, your program can check the pixel color of that room to determine what to place in the room, and can check the pixel colors of the adjacent rooms to see where to draw the exits. All of this can be done extremely fast when using a pixel array to keep track of the rooms and their contents.

Player's Directional View

Another consideration is the directional view of the player. The above-mentioned array will tell you which rooms are connected, so you can draw the proper view of the tunnels. But consider that there are four views of every room. If the player stands facing north, a different view is seen than if the player faces south, east or west.

Adjust the viewpoint of the game graphics to the direction of the player's gaze

When your game draws the current room, and the view down the hallway of the maze, you need to take the direction of the player's view into consideration. There are a variety of methods you can use in your programming. You can keep track of the directions in variables, and use conditional statements to draw the sides and end of the hall correctly. Or you can use an array to keep track of the surrounding rooms and the direction that the player is viewing them. Just be sure you don't forget to adjust the view to the direction the placer faces.

Multiple-Character Control

Next, we should consider how many characters you will want your player to control. Will the player control a single player who explores the dungeon, or a team of three or four hardy adventurers? Control-

ling a single character is obviously the easiest, but permitting a team of several characters can be accomplished with a bit more effort.

If there will be more than one character, you need to design some way for the player to switch characters. You could show graphic representations of each character at the top of the game screen, and define a click zone about each one. When the player clicks on any of the zones, a subroutine should be called which will clear the variables used, and set them properly for the various attributes for the new character.

By having several characters in your game, you can allow one character to be killed but permit the party to continue with the game. Allowing more than one character to take part in the game is a nice addition to these types of games.

Player Attributes

Next, we should consider what attributes you want the player's character or characters to have. A half-dozen or so should be sufficient, although there is actually nothing to limit to your creativeness. The values for these attributes can be saved in standard numeric variables. If you plan on having a team of several characters be in your game, you may prefer to save your individual characters' attributes in arrays, to make it easier to call up the individual values.

Let's say you decide on attributes for "strength", "stamina", "intelligence" and "experience", which are standard player attributes in role-playing games. With a single character in the game, you could simply use numeric variables with these names. As the player vanquishes each monster, you can increase the "experience" variable. If it's a tough fight, you can decrease the "strength" variable, and so on.

However, if you plan on allowing several characters to be in the game, you may find it easier to use an array, where the values of each element are colors in your graphic screen. In that way, when the player switches to a different character, your game can simply switch to a different line in the array.

Player Inventory

Now that you have designed the player's attributes, there some other things you need to design for the player. Inventory is one of these. You need to create some method for the player to pick up objects, whether they are treasures or weapons. Can the player simply walk over them and pick them up automatically? You could require the player to click on them in some manner, in which case you need to define some click zones.

Then you must decide how to show the character's inventory. Will it be by means of graphic pictures, or simply a list of words? Depending on what you design here, you will have to provide some way to drop the objects again. If you choose to show objects graphically, you will need

to design some way to click on the pictures to drop them. If a list of words is shown instead, you may want to design a list that scrolls, so the player can press return to select the one to drop.

Whatever you choose, you need to design an easy-to-use way of taking objects, showing the objects held, and dropping the objects.

Game Animation

Animation is always desirable in any dungeon adventure. This can include doors which are be shown opening as well as monsters shown moving about.

The easiest way to accomplish monster movement is to have the various poses for the monster in a graphic file that is loaded from disk and saved in a hidden screen. When the player encounters a monster, your program would copy a picture of the monster onto the dungeon room, then keep copying different poses onto the dungeon room until the player kills the creature.

One way to do this is to copy the section of the background where the monster is to be placed into a hidden graphics buffer. Then copy the picture of the monster onto the game screen. To show the monster moving, you then copy the background back to the game screen, copy the new section of background where the monster will move to, into the hidden buffer, then copy the new pose of the monster on the game screen.

Double-Buffering

The problem with the method described above is that, while it is easy to program, it also shows the monster flickering, since the monster is not being shown for the part of the time while the background is being restored and the new section is being copied.

A better way to show the animated monster is to use a method called double-buffering. You will basically do the same thing as described in the previous paragraph, but it will all be done on a hidden graphic screen. When the picture is complete, the game then flips to view that buffer. The process is then repeated, with the new pose for the monster being created as described above, but again in the hidden buffer, and then the view is flipped to this buffer when it is ready. In this way, no flickering is seen. The game may move a trifle slower, since an extra step is taking place, but at the speeds which Visionary copies graphics, this will never be noticeable.

Items other than monsters can be animated in a dungeon-type game. A large stone door can swing open with a grinding sound. A studded iron door can screech open. Or a drawbridge can be lowered with a thud. The basic methods of animation as described above can be used

to create moving doors, as well as mice running across the floor, shadows flickering against the wall, or water bubbling in a pool.

The Graphic Interface

Designing the game interface, the way the player interacts with your game, was one of your first tasks, even before the programming started. The game interface developed from the map you devised, the plot you created, and the puzzles you included in your adventure design. Now it's time to translate that concept into the **graphic interface**.

Let's see what you will need in the graphic interface for a dungeon adventure. You will need a place for the player to click the mouse in order to move. Usually, some type of compass is used for such movement. It need not be an actual compass with "north", "south", "east" and "west", but should at least consist of arrows pointing the way that the player can go. Or the player might simply click on the picture of the dungeon itself: clicking in the center to move forward, on the left side to turn left, on the right side to turn right, and on the bottom to move backwards.

If you plan on having more than one character in the adventure, you will need some way for the player to choose which character is currently being controlled. As mentioned above, this can be easily done by showing a picture of the character, and allowing the player to click on the one desired.

Also as mentioned above, you will need to design some way to get, display, and drop objects. And then you will also have to design some way to use the weapons that the player finds. You may wish display a rectangle on the screen that shows the player's weapons. If the player clicks on one of the weapons, that is the object currently in use. The player can then choose either to fight with it, or drop it.

The Goal

What is the goal in your game? Perhaps the final goal is to reach the bottom-most level of the dungeon and kill the ultimate monster. Usually in this type of game, the monsters are harder and harder to kill as you descend deeper into the dungeon. However as your character gains more experience, more strength, and more powerful weapons, these monsters can be overcome. That means your player's sub-goals will be to complete each level of the dungeon, to vanquish all the monsters there and find all the treasures, weapons and magic potions in that level.

You might change the goal of your game, and have the player reach the bottom of the dungeon in order to rescue the princess. Then you can repopulate all the upper levels of the dungeon with even more powerful monsters, weapons, and magic potions and make the player fight all the

way back to the surface with the princess. The plot design is up to you. Just make sure you have it designed on paper before you start your programming.

This chapter certainly hasn't encompassed every single problem you may encounter in creating a dungeon-type maze game. Others may crop up as you write your game. But it has perhaps given you some insight into the type of things you need to consider when you use Visionary to make your dungeon to end all dungeons. As with any adventure game, a dungeon game that is well-designed on paper first, is a dungeon game that is much easier to program.

Chapter 12: Artificial Intelligence in Non-Player Characters

Visionary allows the inclusion of non-player characters in your game. These are characters other than those the player control.

In a standard adventure game, usually the player is transported to a new world and started upon a quest. In this case, the player is the character. In role-playing adventures, the player assumes a new identity, that of a special character with individual special attributes. In such games, the player can sometimes assume the persona of several different characters, all of whom may make up a party of hearty adventurers who will work together to complete their quest.

But in both types of adventures, you will find other characters who are not controlled by the player. These will be characters the player encounters in the game, and interacts with. This chapter will examine ways in which you can create and use non-player characters in your adventure games.

Let's start with a simple example. In the game *Frankenstein's Legacy*, written over ten years ago by this author, Dr. Frankenstein's creature was brought to life by the player. With a shower of electrical sparks, the creature jerked to life and started toward the player with a lumbering gait. The murder in its eyes convinced the player that some way must be found to lure the creature to its doom, or else the player would fall victim to its violence. The creature followed the player from room to room until the player stopped and was caught, or the creature was destroyed.

This is an example of a non-player character, or **NPC**, at its simplest. The creature started out as a nonmovable object, and changed to an NPC when it was brought to life. If this game were written in Visionary today, the creature would be considered an NPC after it came to life, and the NPC file would program its moves.

Level of Intelligence

All NPC's have artificial intelligence. The level of that intelligence is up to the game programmer. The programmer can choose to give each NPC a low level of intelligence, or he can choose to make its intelligence quite high. As you can imagine, the higher the level of intelligence, the more programming is required. More decisions have to be made for the intelligent NPC, and this requires additional programming of conditional statements.

The level of intelligence of Dr. Frankenstein's creature in *Frankenstein's Legacy* was very low. The creature just followed the player from room to room. If the player kept on the move, he was safe. If he stopped for any reason, he would be captured and killed. The only intelligence the creature possessed was the ability to follow the player. If the player spoke to it, the creature would not respond. If the player tried to attack the creature, it would not respond. If the player tried to interact with the creature in any way, it would not respond. It was only programmed to follow and kill.

This is the simplest kind of intelligence to program into any NPC. In this case, a routine in the NPC file was checked to see if the creature was in the same room with the player. If it was, the player was killed. If it was not, then the creature was placed in the current room, and the game continued. This level of intelligence was appropriate for Frankenstein's creature, but may be inappropriate for other NPC's in your adventures.

The Effect of Intelligence Level

Let's consider other levels of intelligence. Consider a hypothetical game where the player buys some canvas sails from a shopkeeper. The shopkeeper only needs a low level of intelligence. At the simplest, the shopkeeper will greet the player as she enters the shop, ask her what she wants to buy, and then ignore any further attempt to converse unless the player asks to buy sails. This is simply programmed by placing the routine in either the action section of the NPC or the vocabulary action file.

The next level of intelligence would be to have the shopkeeper respond to requests to buy other items. Perhaps a list of items for sale is placed on the wall of the shop, and the player can ask for any of them. You can program the NPC to check for the item that the player requests, and reply "Sorry, fresh out" to all requests except the request for the sails.

For intelligent NPC actions, anticipate how the player will try to interact with the NPC

A still-higher level of intelligence would involve creating those other items as objects, and allowing the shopkeeper to sell any of them. At another level you could check what the player is asking the shopkeeper, and have the shopkeeper programmed to respond appropriately to some anticipated questions. You might additionally program the NPC to allow the shopkeeper to move about. This might force the player to track down the shopkeeper, who is somewhere in the back of the shop.

Let's see some ways that you can program greater intelligence into the shopkeeper in the example above. You must first anticipate how the player will try to talk to the shopkeeper. If you want him to directly address the shopkeeper, you must still anticipate he may try some general command like "talk to the shopkeeper." In this event, you might program your game to respond "go ahead and talk to him." This

would be best programmed into the object file for the shopkeeper, under the action for "talk."

When the player says "I want to buy some sails" your game must be able to understand what the player has requested. You can enter the desired actions in both the object file for the sails, as well as the vocabulary action file, for variations that won't be caught elsewhere. Then you can program the shopkeeper to give the player the sails in return for a bag of gold, or refuse to sell them if the player lacks the gold.

For greater realism, hide objects until the player's action reveals them

If you want the player to be able to select from several items, use a slightly different technique. Place all the objects that can be sold in the shop. But make them invisible. That means in a graphic adventure, don't show pictures of them. And in a text adventure, don't describe them in each object's code block. In that way, when the player refers to any of the items, the appropriate object file is executed and the action entry for "buy" is executed. Each action entry might first check for the presence of the shopkeeper, and tell the player "there's no one here to buy from" if the shopkeeper is absent. Then it might have the shopkeeper make various replies, depending on whether the player can afford the particular item he has requested.

By using Visionary's "rand" variable, you can even have the shopkeeper give different responses randomly. Each random response should have the same final meaning, but should be phrased differently.

Sophisticated Interaction

If you want to permit more sophisticated game action between the player and the shopkeeper, you might wish to allow the player to talk to the shopkeeper about other things than his purchases. The non-player character of the shopkeeper isn't a very interesting person, if it only talks to the player in order to sell something.

The more limited your NPC's responses are, the less intelligent it will appear

You may wish to design your game to allow other verbal interaction. This will require additional code to anticipate and respond to common dialog. If you want the player to be able to say "What's new" to the shopkeeper and receive a specific response, you will have to program it into your vocabulary action file. Similarly, you can permit the player to say things like "Nice weather" or "Hello" to the shopkeeper, and have the shopkeeper respond in some appropriate way. Since none of these commands act upon a specific object file, you should place them in the vocabulary action file.

NPC Movement

It was also suggested that perhaps the shopkeeper be allowed to move about, rather than stay in one place. This movement is programmed into the NPC file. First, you as the programmer must decide what

rooms the shopkeeper will be allowed to move into. Let's say, you decide to limit movements to the shop itself, the back room, the alley behind the shop, and the upstairs apartment.

Check to make sure NPC movement actually resulted from a random NPC move

There are several ways to move the shopkeeper randomly, but a simple one is to use the "moveobj" command to move the shopkeeper in a random direction. Then check the "error" variable or the "objectPOS" variable to see if the shopkeeper was successfully moved. This is necessary, since "moveobj" will not allow an object to be moved in an illegal direction. If the move was not made, keep the routine in a loop until some move is actually accomplished.

Random NPC Actions

Let's examine some other techniques that can be used with non-player characters. The "rand" variable has some uses other than the above example to choose a random direction for the shopkeeper. A non-player character's response to the player can be randomized, to give a more intelligent feel to the NPC. If every time the player says "hello" to the shopkeeper, the shopkeeper responds back "hello" it's like talking to a robot. If you are feeling ambitious, you can program several similar replies to each action the player makes. Your NPC can then respond differently, every time the player takes that action.

Let's say for example that the player says "hello" to the shopkeeper. The shopkeeper might have a list of possible replies that included "hello", "howdy", "greetings", "mornin'", "hello yourself" or "come on in." Each time the player greets the shopkeeper with "hello" the shopkeeper could give a different reply. You could keep track of which reply had been used already in a variable. Then you could either have the shopkeeper give a reply in some predetermined order, or you could choose it randomly, using the "rand" variable.

Monsters can move randomly, and have random effects on the rooms they pass through

If you are creating a dungeon-type maze game, you might want to have monsters roaming around in the maze of tunnels. In some maze games, the monsters are always at one location. After the player becomes familiar with the game, she knows exactly where to go to find the monster. You can make your game more difficult by varying this. Make the monster move about randomly throughout the maze. In this way, the game becomes more challenging for the player, who never knows when the monsters will appear. The programming for this type of movement is again done in the NPC file. Since the NPC file is executed after every turn, the monsters can be moved one room per turn. This can be done in the same way described above for the shopkeeper, by using the "moveobj" command and the "rand" variable to move the monster randomly.

You can not only have your monsters move randomly, but they can also take other actions randomly, while out of the sight of the player. You might decide to have the monsters steal any treasure they come across in their random movements. Or if two monsters randomly happen to

enter the same room, you might want them to magically join to become a "super monster."

Moving monsters could destroy objects that they encounter. Imagine how the player will feel if she enters a new room in the dungeon and finds a magic shield that has been smashed by a visiting ogre earlier. This adds a bit of realism that will not be found otherwise.

Player and NPC Interaction

Interaction with non-player characters like monsters usually takes the form of either casting magic spells or fighting. Although you want your non-player characters to be able to move about in other rooms without the player's knowledge, you will usually want them to come face to face, eventually. And when they do, you must program your game to permit some type of interaction.

Magic spells are the easiest type of interaction to program, because they are only used once. If casting the spell doesn't work for the player, that's it.

Keep track of both player's and NPC's attributes during and after interactions

Fighting is different. If the player thrusts with his sword and it fails the first time, he usually is given additional chances to stab the monster. Use the NPC object files to create the programming that will check to see how the player fares in his battle. Remember to use variables to keep track of the monster's attributes as well as the player's attributes. In this way, you can make the battle seem more real. After each move of the player, your NPC file can check the variables and print out the result to the player, telling him how he fared and how the monster reacted, before the player's next turn.

NPC People

Using people instead of monsters can make the interaction of your non-player characters more difficult. People are usually expected to be more intelligent than monsters, so you must do more programming to give your program the artificial intelligence that these NPC's require.

People NPCs are usually expected to be able to talk with the player

You will probably want to allow the player to talk to other people, something usually not permitted of monsters. This means your program must anticipate the most common verbal interactions, and give appropriate responses. Of course, the player can still fight with people, as he does with monsters. Your NPC files should include programming to allow the people to fight back. But in addition, the NPC files should allow the player to talk to the non-player character and give some type of response in return.

As with monsters, when people are used as non-player characters they should be allowed to move about in the background and take actions that affect the game. And frequently they will be permitted more actions than monsters. You should program your NPC files so that the

people can wander about in the adventure and move objects. They can pick up things in one room, and then leave them in another room. They might attack the player and steal some objects, which could be left elsewhere. They might also change things, like lock doors that had been left open, cause cave-ins to seal passageways, or leave clues written in blood on the mirrors.

There is really no limit to what the non-player characters in your adventure can do. The more time you spend in programming, the more sophisticated actions they can take. Just be sure to use easy-to-understand variables to keep track of the characters and their attributes. And use the "rand" variable to vary the directions the characters move, as well as the responses they make. By adding non-player characters to your game, it can be a fuller and more exciting experience.

Chapter 13: The Finishing Touches

You're finally done. Your adventure is finished and completely playable. You've written the source code, you have compiled it, and you have fixed all known bugs. You can play it and win, or play it and die. But you're not quite done, yet. You still must go back and fine-tune your game. Then there is play-testing. And you have to make a final decision on the distribution of your masterpiece. In this chapter, we will look at these final steps as you prepare your game for release.

Fine-Tuning

Now is the time to go back and add those unnecessary messages. Add the ones that weren't necessary and thus were probably overlooked. As mentioned in previous chapters, these add detail to your game and create a more realistic make-believe world. So now is the time to add the bird chirps, the coyote howls, the scent of pine, and all the other random messages that add richness to the tone of your game.

If you have a graphic game, consider replacing the random messages with random sound effects. It is much more impressive to actually hear the coyote howl than just to be told it howls.

Add unnecessary objects to mislead the player and enhance the game's atmosphere

You might also consider adding objects at this final stage. Up to this point, you may have been focusing on the objects necessary to the solution to the various puzzles in your game. You may not have spent much time considering extraneous objects which serve no purpose except to mislead the player and add to the game's atmosphere. Now is the time to add these unnecessary objects. Add the candy wrapper found lying on the floor. Include the six-inch piece of cotton thread found in the dirt. Make sure that these objects can be picked up, dropped, examined, and manipulated in any normally-expected way. Always consider how the player might try to use these new objects.

Think through the possible actions with the new objects. You wouldn't want to create something which could unwittingly be an alternate solution to some puzzle. If you intend for the player to cut down a tree with a chain saw, don't create an axe at the last minute to act as an extra unnecessary object. If you do, you must be prepared to allow the player to use it instead of the chain saw. Adding new objects at this point should be for the purpose of enriching your game, but not altering the solutions to the puzzles.

The more ways an object can be used, the more programming is required to add it to the game

Don't create an object that can be manipulated in too many ways. For example, it is a lot easier to add a candy wrapper than it is to add matches. There are a limited number of ways to manipulate the candy wrapper. There are many more things you can do with matches, however. If you add matches to your adventure, you will have to go back and modify the action sections of every object, in anticipation that the player will try to burn the objects. Add things like matches only if you are willing to spend the additional time to properly modify all the objects that can be affected by them.

Play-Testing

Naturally, you have been playing your own game as it has grown. As you have made additions and changes, you have played and tested them. But now is the time to go back and play the entire game again. In fact, play it several times. Each time you play the game, you will be checking for different things.

Play to Win

First, play the game to win. Make sure that all the treasures can be obtained, all the puzzles solved, and all the challenges met. Don't worry about all those other things you added to your game. Ignore the parts that are unnecessary to the ultimate solution.

The first time through, you want to make sure that the game can be won. Since you know exactly what it takes to win your adventure, this run-through is the fastest one. If you find any errors, mark them down. Keep a written record of them. Then use this record to go back and fix the errors you have found. After you are sure the game can be won, you are ready to play it again.

Play to Lose

The second time you play your own game, you want to intentionally lose. You want to test out all the ways that the player can die. There is probably more than one way to die in your adventure. The player could run out of time and be caught by cannibals. He might fall into a pit of army ants. He might die of thirst in the desert. Whatever ways you have designed, you must now test.

Play the adventure and intentionally try to die. Test each death trap to see if you die where you should. When dying from thirst, check to make sure you die after the appropriate number of turns only. Make sure you don't die if you drink water barely in time.

This second run-through will take appreciably longer than the first. But as with the first, keep a written record of any errors you find. When done, go back and fix them. That leaves you ready for the third and final reason for play-testing your game.

Play All the Options

Try every game option—use the unnecessary objects, argue with the talking NPCs, try all the synonyms

By now, you are probably sick and tired of playing your own game. But the third run-through is perhaps even more important than the first two. This is the one that will take the longest. This time, try everything. Whatever you have written into your game, you must test. Make sure that all movable objects can be taken and dropped. Check to make sure your inventory is correctly being incremented and decremented. Make sure you can't exceed any inventory limits you created. Examine every object, to make sure it gives the correct description. Use every object in all the ways you programmed. If you can think of alternate ways that a player might ask for something, go back and add it as a synonym.

If you suddenly think of something that a player might try that you didn't anticipate, now is the time to go back and add that action to your game. When you have finished trying every possible action and combination of actions that a typical adventure player might try, you are done with the run-throughs.

Outside Testing

A fresh eye may see things you're too close to the game to spot

The next step is to have someone else test your adventure. You may think your game is ready to go, but other players will find things you have overlooked. Choose some people you trust, players who like adventures. You need to trust these people not to hand out copies of your game without your permission. And there is no sense in having them test your game if they hate to play adventures. Ask them to keep a written record of anything that seems wrong. Note spelling errors. List errors in logic. If something just doesn't seem right, have them write it down.

Be prepared for criticism. It can hurt to hear bad things about your game, especially after you have spent so much time on it. But don't take it personally. Remember, it is going to make your game even better.

Spend the time to go back and fix the things your testers listed. If one tester suggests things you disagree with, discuss them with all the testers. The suggestion may be right, considering that the tester is looking at your game with the same fresh eyes as your future players will have. You, on the other hand, are looking at it through the eyes of its designer. You may feel some emotional attachment, and may resent any question about the way you designed it. Trust your testers. Listen to their advice, and fix the problems they discover. Then ask your game testers to play the revised game one more time. Get a final report and make any last-minute changes.

Finalize the Title

Since the start, when your game was in the planning stages, you have probably had a working title for it. Now is the time to select the final title. The title is a very important part of your game. If you intend to commercially release your game, the title along with the artwork on the box is the first thing a customer will see. If your game is released to public domain or as shareware, the title will again be the first thing the customer sees. The title could be the thing that sells your game. It could be the one thing that makes someone decide to try your game instead of some other. So choose your title accordingly.

Make the title descriptive of the adventure story. Make it sound exciting. A title like "Monster Adventure" may be fine for a working title, but "Monster Island" will sell more copies. Likewise, a working title of "Jungle Adventure" could be changed to "Lost City of the Jungle."

The final title should entice the user to play your game

Whatever you choose for your title, make sure it not only tells the prospective player the general setting of the story, but entices him to want to play it. Once you have selected the title, put it at beginning of your adventure. Place it in bold print. And don't forget your name beneath it. If you are using an opening graphics screen, it must go there too. You certainly expect credit for all your hard work.

Copyright Notices

Along with the title and author's credits, you should include any copyright notices. To ensure your legal rights, you should include copyright notices in several places. Put one at the beginning of your adventure before the title, and perhaps one on the graphics title screen as well. Use both the word and symbol for copyright, followed by your name and the current year. Even if you plan on releasing your game as shareware or public domain, you should still include the copyright notices.

You should also include the identical copyright notice in a remark line in your code before compiling the game. Although this will never been seen by the game player, its inclusion in the code qualifies as "legal notice" of your intention to retain your copyright. You are allowed to retain copyright even for materials released without charge, as freely-distributable software. Only if you release your game into the public domain will a copyright notice not be needed.

The legal format for copyright notices is

```
Copyright (C) {year}, {copyright holder}
```

For a game produced in 1992 by a programmer named V. I. Sionary, this copyright notice would appear thus:

```
Copyright (C) 1992, V. I. Sionary
```

The copyright symbol © is usually substituted for the text string (C) in this copyright notice in printed material, but either notation is acceptable.

Final Checks

Now that you're done, go back and take one last look. Yes, again! Check the entire program for spelling and grammar. Look at your room and object descriptions. Are they complete? Are your helps clear enough without giving away the solution to a puzzle? Did you keep the player from doing things at the wrong time or in the wrong place? For example, can she shoot the gun if she isn't holding it?

Use a different password for each game you develop

Check the saved game feature. It's important to make sure that your game only allows saved positions to be loaded if they were from **your** adventure, not some other Visionary game. Again, Visionary has anticipated this problem and has taken care of it for you. Each adventure you create will have a unique code that is included in each saved game, to prevent saved places from other games from being loaded into your game. This unique code is based on the password of your adventure, so be sure you use a different password for each adventure you write.

Game Distribution

Now that the game is finished, debugged, and fine-tuned, the next step is distribution. You wrote the adventure so other people could play it, not so it could sit on your shelf. What does your game need before it is released? Will you release it as public domain, as shareware, or try to get it published?

Copy Protection

If you haven't given any thought to copy protection up to this point, now is the time. How do you feel about it? Most users are against it, but many publishers still use it to help prevent loss of revenue due to piracy. Software piracy is less prevalent in business and productivity software, but is more of a problem with game software. If you are thinking about selling your game, you should think about copy protection.

Disk protection is possible, but tough to do unless you are an expert. You have to decide which method you will use to make the disk uncopyable but still playable. Disk protection also makes duplication harder for a publisher. If you intend to sell your program through a publisher, it may increase duplication costs to use disk protection. A final consideration is that disk protection is not that effective. There are a variety of software programs available which permit the user to copy disks even though they are protected. There are even hardware systems available that permit the user to copy protected disks. You

may not wish to go to a lot of trouble designing a protected disk, if that protection is relatively easy to defeat.

A method of program protection that seems to be gaining popularity is manual-style protection. In this method, your game asks the player to answer a question based on the game manual or the game box. If she answers the question correctly, you can assume she has a legitimate copy of your program, and permit her to play it. If she answers incorrectly, you may give her a second chance or simply assume she has a pirated copy. In that case, you can lock up the game, denying access to this player.

Manual-style protection requires that your game have a manual

One advantage of manual-style protection is that it is easy to insert into your adventure. Put the question at the beginning of the story, and again in the middle. The player can't continue with the game until the correct answer about something in the user manual is given. Another advantage for the publisher is that there is no additional cost for special duplication. The advantage to the customer is that the game disk can be backed up as many times as needed.

This method of protection is not infallible, especially since it is so easy to photocopy any manual. It also assumes that you will write and print an manual to go with your game. Considering this, you may wish to use no protection at all. The program is your own creation, so the decision is yours.

Distribution Methods

If your game is good enough, you may find a publisher who wants to put it on the market and pay you royalties. But that's not the only way to distribute an adventure. Another possibility is to submit it to a magazine. Several Amiga magazines come with a companion disk, and others are strictly on disk with no companion magazine. They are always looking for programs to fill their disks. Another possibility is to release your program as shareware, and allow your program to be distributed through those channels. Of course, you can always release it into public domain for the enjoyment of every Amiga owner.

Contacting a Publisher

If you decide to try a publisher, pick one you trust, one who can best sell your game. Don't feel you must choose the one who offers the largest royalty. Pick the one who will give your game the widest market. Remember, it's better to have 15% of a big pie than 30% of a small one.

There are questions to ask, as you choose which publisher to submit your program to. Does the firm advertise in appropriate places? Does it have overseas sales? Has it been around for a while? You may not want a publisher who is new and doesn't have the contacts to get your game distributed into the stores. You should try to avoid a firm which

has overextended itself with advertising and may be in danger of bankruptcy.

As soon as you have decided on the publisher you intend to contact, write a letter to the firm. Don't send your program yet. You letter must convince the publisher that you have a product he can use. Let him know what your program is, what it does, and any special features. Try to get a signed non-disclosure agreement which ensures both your rights as well as the publisher's, before sending him your disk.

Then go ahead and send the program via insured mail. Be prepared to wait. Publishers get a lot of submissions, and it takes time to review them all. If the publisher likes your game, expect that changes will be requested. Rarely—if ever—is a program accepted without at least some slight revisions.

If you are offered a contract, read it carefully. Understand what you are being offered. Will your royalties be 15% of gross, or retail, or wholesale? Will you receive a set dollar amount per unit sold, regardless of the selling price? Can the contract be cancelled? If so, how and by who? Is there a provision for an audit of sales if you request it? If you are unsure or uncomfortable, see a lawyer, who can explain any parts of the contract which you question. If all goes well, you will find a publisher who likes your adventure and with who you are comfortable doing business. Then work with him to polish the program and get it ready for to market.

Shareware and Public Domain

Instead of selling your adventure to a software or magazine publisher, you may wish to release it as shareware. Encourage users to copy your disk and give copies to their friends. If they play it and like it, you ask them to send you a fee for it.

If you choose this route, include your name and address in an easily accessed area. Usually, it is best to place this information at beginning of the game. It can also be printed in the adventure manual, if your game includes one. Mention that the program is shareware, is copyrighted, and how much money you expect if they should decide to use your program. Tell the user what she gets in return for mailing you the fee. You may wish to offer a map or hints, or even a complete solution in return for the fee. You may wish to send the user an entirely new adventure game. Shareware seems to be a lot more effective if you offer something else for the fee, not just the game the user already has.

For shareware releases, use an address that will not change

Be sure to use an address that will not change for several years. The post office will only forward your mail for one year, and you may continue to hear from users for years and years as your program continues to circulate.

Release to public domain is giving your game away for free

You could even give your game away for free. That, in essence, is what you do when you release your game into public domain. It means you are giving it away. You give up all rights to the program. Anyone can copy it for no charge. Anyone can resell it, if they wish. By putting your program in public domain, you are giving anyone the right to do anything they want with it. This is not recommended unless you are extremely philanthropic.

Your game is done. You have finished writing it, debugging it, and have decided on its final distribution. Now what? Sit back and enjoy the accolades from the players who love your professionally-designed adventure. Bask in the glow from the applause and the cheers. Read the magazine articles by reviewers who love your game. And start thinking about your next epic. Now that you've discovered how much fun it is to write adventures, you won't want to stop. You probably already have ideas in your head for several more adventures. Jot them all down, pick one, and start on your next game. Happy adventuring!

Chapter 14: I Was a Cannibal for the FBI

The second half of this book will is about the example game *I Was a Cannibal for the FBI* which is on the disk included with this book. The source code for the game can be found on the disk, and it is also printed in Appendix A in the back of this book, for your reference.

The game was written specifically for this book, and uses a variety of special routines which will be explained in detail. After you have read Part 2 of this book and have read through the source code for the game, you will have acquired a wealth of tricks, tips, and other techniques that you can use with Visionary to create even better adventures.

Play the Game

Still haven't looked at the game? Read no further—play it now!

By now you should have booted up the disk and played the game. If for any reason, you still haven't looked at the game, you should try it out before reading any further. Put the disk in drive df0: and reboot your computer. The game will automatically start, first displaying the title screen and then loading and executing the game itself. You can exit the game by winning, dying, or clicking on **QUIT**.

It is not necessary that you win the game. It is only necessary that you have tried the various features, and will understand what we are talking about as we go on. Winning the game, or even knowing how to win, are not required for the rest of this book.

The adventure was designed to be an easy one to solve. It was designed for a beginning level adventurer. However, don't feel bad if you haven't solved it yet. At this point, you are probably more interested in learning how to write a game, than you are in learning how to solve this specific one. If you can't solve the adventure at this point, you have several options. You can look at the source code on the disk, and track down the solution. Or if you prefer, you can look in Appendix B for the complete (and simple) solution.

» If for some reason you have a defective disk that won't load, please contact Oxxi/Aegis for a replacement disk. The address is in the Technical Support Appendix. There is a nominal charge for a replacement disk.

A Text Editor

The second half of this book assumes you are using a text editor like ED or TurboText to create your source code, and then using **VCOMP** to produce the two **.GAM** and **.WRD** files.

Available separately from Oxxi/Aegis is a graphic editor called **VIE**. If you are using the VIE program, you don't need to use a text editor or VCOMP, since VIE combines them to make it easier to write your program and catch and correct errors.

Whichever you use, the source code listed in Appendix A of this book will produce the same game. It is also assumed you have some general ability in using CLI or Shell. If you feel you need additional help with CLI/Shell commands, please see your AmigaDOS manual.

Graphic Interface

If you use the Visionary graphic interface to develop your games, you are limited by the choices offered in the interface. Even if your first games were developed using this utility, chances are good that you'll now want to try some tricks the graphic interface doesn't provide. For this, you'll need to use a text editor and develop your own object, room, NPC and other CODE files.

The Game Interface

Now that you have looked at the *Cannibal* game, you know it combines both text and graphics. It is not a pure text game, and it is not a pure graphics game. It is a hybrid combination, combining the features of both.

The game was intentionally designed in this fashion. Text games are the easiest to write. Graphic games are harder. But hybrid games are the hardest to write, because they must allow dual input. The player must be able to type "Save" and have the game respond the same way as if he had clicked on the **SAVE** button. In theory this may sound rather simple, but as you will see later in this book, it is more complicated than it sounds.

Objects and Inventory

Simple mouse drags are used for the GET and DROP actions

Let's examine some of the specific features of the game, and explain why they were used instead of other methods that would achieve the same goal. Picking up and dropping objects is accomplished by simply dragging them from the location window to the inventory window, or back. Notice that the player's inventory is limited to six items. Once it is full, the player cannot pick up any additional items.

Also notice that there is no limit to the number of items that can be in a room. Each item in the room appears on the side of the location

window. If there are more than five objects in the room, the window will scroll to show the rest of them. Because of this feature, the window containing the objects is usually referred to as the scroll-bar window. The scrolling of the window and the moving of objects by dragging them are both very important techniques you may want to use in your adventures. Later in this book, an entire chapter will be devoted to the routines that accomplish these feats.

Command Input

The player can input commands to the game either by typing them in the text window or by clicking on the buttons. The most common commands can be accomplished by clicking on buttons. Less-common commands must be entered from the keyboard. For example, examining the boulder can be easily accomplished by simply clicking on the picture of the boulder. But moving the boulder cannot be done with the mouse. The player must type "MOVE THE BOULDER" from the keyboard. When you write your own game, you may wish to do away completely with the text input. In this case, you would need a "MOVE" button in order to move the boulder.

EXAMINE actions use a single mouse click

In this game, examining objects is done by simply clicking on them. If the player wishes to examine the ladder, he only has to click on the picture of the ladder, and he is given a more detailed description of it. If he wishes to examine the bottle or the boulder, he only has to click on their picture.

Notice that this works with both movable objects in the scroll-bar window and the inventory window, as well as nonmovable objects in the location window. Although this is intuitive for the player of the game, it is not simply accomplished in one section of the source code. Examining objects by clicking on them is actually handled in three separate areas of the source code. We will be examining exactly how this is done in a future chapter.

The Compass

Direction of movement is handled with a intuitive click on a compass button

Another feature of the game which deserves mention is the compass. There are six buttons on the compass, one each for north, south, east, west, as well as up and down. When the player is permitted to travel in any of those six directions, the letter on the button is highlighted in orange. When the player can't travel in a certain direction, the letter is "ghosted" in gray. This provides as easy way for the player to tell which directions can be travelled, without having to display the information in the text window. Presenting the information to the player in this fashion is more difficult to program, but results in a game that is more user-friendly and easier to understand. The routines to accomplish this feature are fully explained later in this book.

Visual feedback is provided by changing how the button looks when clicked

Also notice that when the compass buttons and any other buttons are pressed the letters on them change to yellow, and the button appears to be pressed inward. When the mouse button is released, the button appears to pop back out again. This is another cosmetic nicety. It isn't required. The game could have been designed without this feature and would have played just as well. But it looks better. It gives the player some visual feedback that the computer really did accept his click on the button. Writing a game without visual changes in the buttons is certainly much easier to do. But taking the time to add the feature makes your game more professional. The way to make the buttons change as they are pressed in will also be explained in a later chapter.

Graphic Techniques

If you played the game long enough, you may have been captured by the cannibals and killed. At that point, the location window held a scene of you in a pot of boiling water. The picture was animated, with bubbling water, dancing flames, and rolling clouds of smoke. This was accomplished with color cycling.

Color cycling can provide an illusion of animation, without the memory requirements

Color cycling is something that can be automatically built into each IFF graphic, using most paint programs. Visionary supports color cycling, but care must be taken in using it. Without taking great care, the illusion would have been broken, and other colors in the scene would have cycled as well. The whole idea would have failed, if while the pot boiled, you also saw the buttons changing colors, or the mouse pointer changing colors.

Using color cycling takes careful design, and planning ahead. A later chapter will go into more detail on how the animation via color cycling was accomplished in this game.

Visible and Invisible Features

There are many other features in this game. Some are visible features, like different colored fonts in the text window, and others are invisible features which you didn't see.

One example of an invisible feature is the loading of the location scenes into RAM for faster display, if the computer you are using has enough memory. When this game loads, it checks the memory of the computer to see if there is enough unused memory. When there is, all 15 of the location scenes are loaded into RAM. As the player moves from location to location, new scenes are loaded and displayed each time. If these scenes are in RAM, the game plays much faster and smoother. If the computer doesn't have enough free RAM, the scenes will load from disk instead.

This is just one example of an invisible feature that the player won't even be aware of. But using this feature in your own adventure will make it play much smoother and give it a more professional feel.

Source Code

If you examine the game disk from CLI (or SHELL), you will find a directory named **SourceCode**. Inside this directory, you will find all eleven text files that were compiled with Visionary to create the game *I Was a Cannibal for the FBI*. These same eleven files are also printed in Appendix A in the back of this book.

Looking at these files will show you all the routines and techniques that were used to make the game into the hybrid text-graphic adventure that you have played. The rest of this book will refer to various parts of these files, as we look at how the game was created, and how it accomplishes the displays, the animated buttons, and all the other game features.

Starting in the next chapter, we will begin to dissect the game, look at the source code, and reveal more of the secrets of writing adventures in Visionary.

Chapter 15: The Idea

As discussed in the first three chapters, any adventure starts as an idea in your mind. That's how *I Was a Cannibal for the FBI* began. Several years ago, when I was writing adventures on the Commodore 64, I had been writing full-length adventures for average and advanced game players, and decided to take a break by writing a small simple adventure for beginners.

Since a game of this nature obviously would not attract a publisher wishing to pay money for it, I decided to use it as an advertisement for the BBS I was running. It would be a program that could be uploaded to any BBS for its users to download. When the BBS users played the game, they would find several notices about my own BBS in various locations. This would encourage them to use my BBS and would increase my user base. This was my original motivation for writing *I Was a Cannibal for the FBI.*

The Concept

Next I had to ask myself, why would someone calling a BBS want to download my game? I decided that if the title was catchy, people would be interested enough to spend the time to download it. But a title that is catchy to one person is dull and uninteresting to another. The important thing here is to **know your audience.**

I knew that a large part of my audience would be between the ages of 10 and 25. This target audience would have older if I had been writing for IBM's, but remember this was during the time that the Commodore 64 was at its peak. The majority of BBS users for the Commodore 64 were in the younger age bracket.

I needed a title that would catch the eye of such a person, and make him want to download the game. I decided on something that would appear gross and disgusting, a sure-fire attention-grabber for this age group. Cannibalism jumped into my mind. Having the player forced into cannibalism as part of his job seemed like the makings of a fun adventure.

Of course, now that you have played the adventure, you realize there is nothing gross or disgusting about it. But that was not the point. The purpose was to "hook" the BBS user into downloading my program. Hence, I chose my title based on the similar movies of the 1950's, like *I Was a Communist for the FBI.* I just changed "Communist" to "Cannibal" and I had a title that I felt would attract my audience into downloading it.

The Plot, Setting and Game Goals

So in this case, the title came first. Then I had to decide what I could do in a game with such a title. The word "cannibal" conjured up images of African jungles. But since I was resolved to write a small game with limited locations, I discarded Africa in favor of a south Pacific island. In Africa, it would be difficult to find a reason for limiting the number of locations the player could visit. However, on an island, the limited number of locations would seem logical, and would not present a problem.

After I had settled on an island locale, I had to choose the goal of the adventure. The rest of the plot would have to wait until I had chosen a goal. Then the plot could be written with that goal in mind. Since the game had cannibals in the title, a obvious goal seemed to be to escape death from the cannibals. I decided to limit the number of turns the player could make before ending the game in a loss. For the player, this would remove the luxury of casually wandering around the island, trying to escape. Since I was trying to program a simple, uncomplicated game, I decided not to have the cannibals present on the island until the very end. After appropriate warnings to the player, they would show up after about 100 turns, and the game would end.

The next step was to choose the solution. How was the player to escape the cannibals? How was she to win the game? My choice was to make the solution constantly visible, if the player knew where to look. The answer was in the old tramp freighter slowly steaming its way past the island. If only the player could get out to the ship, she would be rescued from the island. Allowing the player to swim out to the ship would have presented little challenge, so although I decided to allow the player to swim, I made her become fatigued and return to shore. To solve the adventure, I decided the player would have to paddle out to the ship. And to make it more difficult, I hid the oars.

Once I had the basic solution in mind, I then began modifying it to make the game more challenging. I replaced the oars with a shovel. The player would never find any oars, but would find a shovel. And this shovel could be used in place of oars to paddle out to the ship. And to make that part even harder, I made the shovel in two parts, the blade and the handle. The player had to put them together to create a working shovel.

The next step in making the puzzle harder, was to hide both parts of the shovel. The handle was buried in the sand dunes, and the player had to dig with his hands in the sand to find it. The blade was made more difficult to find, by placing it in a rock room inside a cave. The room could not be entered without using a ladder, since it was high off the ground. And entrance to the cave was barred by a large boulder that the player had to move.

And to complicate things further, the player could not push the boulder out of the way with her normal strength. She required a burst of energy from a candy bar to achieve this feat. And finally, I hid the candy bar in the top of the palm tree. So even though the solution was simply in the shovel, I made it difficult to obtain.

Of course, I didn't want the shovel to be too obviously the solution, so I allowed the player to treat it as an ordinary shovel, and dig up the ground. By allowing him to dig and find various buried items, I misdirected his attention away from using the shovel as a paddle.

At the same time, allowing the player to dig and find buried items allowed me to introduce other "red herrings" to further misdirect the player. I knew it wouldn't take the player long to figure out that his salvation lay in the old tramp steamer, so I planted a variety of false solutions which would mislead the player into believing he had found a way to signal the freighter.

Plenty of false solutions make the game more challenging

I planted a flare gun in the adventure, but it contained no flares. It was my intent that the player might search in vain for some flares so she could signal the ship. I also planted some wet matches on the beach, along with some driftwood in the hope that the player might try to start a signal fire. But I made sure that the matches never dried out, and wouldn't burn anything. I created a two-way radio for the player to find. But she couldn't use it to radio the ship, because it lacked batteries. I even placed a radio battery at a different location, but the battery was dead. It was my intention to fool the player into trying to contact the ship in a variety of other ways, and forget about the somewhat obvious method of rowing out to the ship.

I decided that by adding other objects, I could further confuse the issue. I created a rowboat in addition to the canoe. Both lacked oars, but the rowboat had a hole in the back and wasn't seaworthy. I put wooden planks on the rooftop of the old shack, encouraging the player to try to remove them and build a raft. I presented the player with a message that the boards were nailed down, so that he would assume a hammer should be able to pull them up. Then I hid a hammer in the cave so that the player would find it and think he had finally found the way to get the boards. But when he tried it, he found the nails were driven in too securely, and would not come out.

The advertising messages were made an integral part of the game

I also designed three different messages for the player to find. These messages would advertise my BBS and encourage players to call. I even told them that the solution to the game was to be found in the bulletins on the BBS, in case they got stumped. The first message was the easiest to find. I left an advertising flyer on the roof of the shack. To find it, the player only had to climb the ladder and find it lying on the roof. The second message was chiseled into the stone walls of the rock room in the cave.

This was a bit trickier to find, since the player had to solve the problem of moving the boulder in order to enter the cave. The third message was left inside a bottle found on the beach. It couldn't be removed until the player broke open the bottle. And I made sure that any of the hard objects that the player found would break the bottle. The coconut, the hammer, the chisel, or the shovel would all break the bottle. When the player succeeded, a piece of paper would flutter to the ground, containing the third message. When I rewrote the adventure for the Amiga using Visionary, I replaced these three messages with advertisements for Visionary and this book.

Unnecessary objects add flavor to the game

And there were other items that I added to the game, which really served no purpose other than to create a fuller adventure, make it a bit more realistic, and give the player more objects to manipulate. There was a candle, which the player could try to light. There was a human skull, which the player could assume was left behind by the cannibals. A dead sea gull was found on top of the boulder, perhaps where it had built a nest. And a coconut was found in the top of the palm tree. All of these objects gave the player something to find, something to manipulate, and something to try using to assist in the quest to be rescued.

The Programming Process

At this point, after deciding on the plot, the goal, the location, and the puzzles, I finally sat down and entered the game into the Commodore 64 computer. Since the game was completely thought out in advance, it look very little time to actually do the programming.

The text game was completed, sent to one BBS after another, and it served its purpose very well. When I found I was going to be writing a book on Visionary, I decided that this same game would make an excellent example. So the same plot was used again, but this time with an entirely new approach. This time the game was going to be graphically oriented.

The first steps had already been completed. The next one was to decide on the exact format of the graphic game, and make any necessary modifications to the plot to accommodate them. The next chapter will discuss the designing of the graphic interface, how it would look and how it would work from the player's perspective.

Chapter 16: The Graphic Interface

Taking a graphic approach to the adventure presented me with some new and unique problems. These problems had to be addressed and solved before I could even start any programming.

Planning the Graphic Interface

The first decision I had to make was exactly how I wanted the game to look. What would players see when they played the game? What would they do? How would they use the mouse to play the adventure? Looking at other graphic adventures provided some answers. Others were of my own invention, born of special needs that I had not seen addressed elsewhere.

Look for game ideas in other games

When you write your own adventures, don't be afraid to look at other state-of-the-art adventure games for ideas. Often they not only give you ideas you will want to use, but also show you things you will want to avoid. You may like the way another game presents buttons for the player to click on, for example, but dislike the way it hides windows and pulls them out only when requested. Your adventure can use the features you like, and avoid the ones you didn't.

The Location Window

In my adventure, I knew that I wanted several windows, each with a different function. I knew that I wanted a large window to show the graphic representation of the current location. When players entered the shack, I wanted them to be able to see the interior. In all cases, I wanted them to be able to see where they were. So a location window was mandatory. It seemed that since this was probably the single most important window on the screen, that it should be the largest. I chose to put it in the upper left corner of the screen.

At this point, I needed to decide if I wanted the game to be all graphic, or allow text input as well. I knew allowing text input would be more difficult to program, but at the same time it would serve as an excellent example for this book. I decided that most commonly used commands should be on buttons that could be pressed, removing the need for typing them. But still, a text window would be necessary.

The Text Window

The need for a text window was obvious. There would be many times when I would want the game to tell the player things. The game would have to warn the player before the cannibals arrived. It would have to

respond to various commands from the player. When the player tried to push the boulder, the game needed a place to report, "it moves slightly, but rolls back."

Of course, since I intended to accept text input as well as mouse input, I also needed a place for the player to type commands. So I decided to design a text window that would lie beneath the location window, and would have room for six lines of text. That would leave five lines of text for messages, and one line for the player to type on. It seemed like a reasonable compromise.

Inventory Handling

The next logical step for my game design was to create an inventory window. This was logical for what I wanted to create—don't feel that when you write you own adventure that you need one. In your game, perhaps the player could easily find out what is in the inventory by clicking on a button and reading a text list of the items carried.

In my case, I wanted to have movable objects visible at the location window, and that seemed to lead to the conclusion that there should be a small window for inventory. That way, the player could see what objects were in the inventory as well as what objects were in the room. I envisioned the player picking up objects by using the mouse to drag the objects out of the location window into the inventory window.

To drop an object, the player would simply drag it back out of the inventory window and into the location window. Having decided that this was the way I wanted my adventure to look, I knew that I would need a smaller inventory window in addition to the larger location window.

Command Buttons

I knew that some buttons would be required, since I didn't want to force the player to type all the commands. The most common commands would be placed on buttons that could be simply clicked to accomplish that command. I placed these buttons on the right side of the screen under the inventory window.

The next decision I had to make was to decide which common commands I would include on buttons, and which ones would need to be typed. If the player could get and drop objects by moving their pictures with the mouse, I would not need a GET or DROP button. At this point I had not clearly defined exactly how the player would get and drop objects by using the mouse, but I knew that was what I wanted.

With GET and DROP out of the way, there were some obvious choices. I created buttons for LOAD, SAVE, and QUIT. The player might want to save the game at its current state, and later load it again. Hence the need for the **LOAD** and **SAVE** buttons. And although the

player could quit at any time simply by turning off his computer, I figured having a special button for quitting would allow the player to return to CLI. The **QUIT** button also made the game look more professional.

I decided **HELP** would be appropriate, to remind the player that help was always available. And **DIG** would encourage the player to dig, which I wanted him to do. Remember that by encouraging the player to dig, I was misdirecting him away from the true purpose of the shovel. And in sandy locations, the shovel wasn't even required for digging. These five buttons seemed sufficient for the common commands, and they were placed in the area to the right of the text window.

Designing the Compass

While I was concentrating on buttons, I knew that a compass would have to appear somewhere on the screen. I wanted the compass to serve two purposes, to indicate to the player which directions were available for travel, and to allow movement in those directions by clicking on the direction buttons rather than typing the directions on the keyboard.

In my adventure I did not use the directions of north-west, north-east, south-west or south-east. That left only six directions for my compass: north, south, east, west, up and down. To tell the player which of those directions he could travel, I decided to "ghost" the directions which were inactive. I did this by making "ghosted" directions a dull gray, in contrast to the normal bright orange color of the other buttons. Finally, I placed the entire compass between the inventory window and the other action buttons.

Active buttons give the player feedback that the button commands are being processed

Then there was the matter of how I wanted buttons to look and behave when pressed. Visionary let me easily design zones on the screen that could be recognized when the player clicked inside them. This is, in general, how all buttons work. But I wanted to make the buttons look like they were actually moving when they were clicked. It certainly wasn't necessary. The buttons didn't need to change appearance in any way, when they were used. But I decided that the player needed the additional feedback, showing him the buttons moving, and reassuring him that his mouse clicks had been recognized by the computer. So I decided to make the buttons appear to move inward when the mouse was clicked.

Along with this, I decided that the words on the button should "light up" when the button was depressed, to additionally reinforce the notion that the button was being activated. I wanted the button to stay depressed as long as the mouse button was depressed, and then to pop back out when the mouse button was released. This then, was the design for the buttons, how they would look and behave.

Handling the Examine Command

One of the most common commands that any adventure player makes is to examine the various objects in the game. I wanted to design a simple way to accomplish this in my game, so that the player would not have to do any typing.

I decided that if the player simply clicked on any object, the text window should give a description of the item clicked on. So instead of typing "Examine the Ladder", the player only had to click on the picture of the ladder. Instead of typing "Read the Letter", the player would simply click on the picture of the letter.

I decided that nearly everything should be examinable, so I planned to allow the player to click on any movable object or nonmovable object—even any part of the scenery—and still receive some description. As you may have noticed in playing the game, you can examine the sand, the water, and the sky in addition to the movable and nonmovable objects.

The Get and Drop Actions

The next decision I made was how to get and drop objects. I knew I wanted them to appear on-screen so the player could see them, and I wanted the player to be able to move them from the location to his inventory and back, simply by using the mouse to drag them out of one window and into the other.

But if I made these objects appear in the location window, there was a problem with making them look right. There were nineteen movable objects, and I had to consider the possibility that the player might carry all these objects into one location. The question I had to resolve was, "how will this look?" I couldn't overlay each one on top of each other, or the player wouldn't be able to see what was at the bottom of the pile.

I considered assigning each one a spot on the screen, so that if it was dropped, it would appear in one specific place. But while a certain spot was fine for one location, say the sand dunes, it was not acceptable for another location, say the meadow. Perhaps the position I chose was taken up by the shack in the meadow. Then dropping the bottle in the sand dunes would look normal, but dropping it in the meadow would make it look like it was sitting on the side of the shack.

Another possibility was to have different spots for each location. I could assign a specific place on the screen so that dropping the bottle in the sand dunes might place it on the left side of the location window, about half way down. And when in the meadow, a different place could be assigned for the bottle to appear, say in the center of the location window, near the bottom. This method would require nineteen different spots to drop the nineteen objects in the game. And

each location scene would be required to have its own special spots for the objects.

I knew the amount of programming to accomplish this would be staggering. It would require setting up arrays for each location, to give the x and y coordinates for each of the nineteen objects. Even though this approach would take an enormous amount of programming, I considered it. But after some testing, I found there were other problems with placing objects on the scenery in this manner.

One of the problems with placing objects in the location scenery was that sometimes the sizes were inconsistent. Most of the location scenes would be shown from the same perspective and distance. Most of the scenes would be what in the movies is called a "long shot." This means the perspective would be as if the player were standing back looking at a large area. But several scenes would be a "medium shot", seen from a closer distance, and one scene needed to be a "close up." When the player sits in the top of the palm tree, the view is a close-up view. When the player stands by the canoe, the view is a medium shot.

Now consider an object, such as a bottle, being placed in the location scenery for these places. A bottle that would look normal-sized by the shack would look too small by the canoe, and far too small in the tree top. By placing the objects in the scenery, the perspective was off. There were several ways around this problem.

If objects will be placed in your a screen, consider how perspective and position may change from screen to screen

To make objects in the locations look normal size, I could either change the scenery or change the objects. I could change the scenery art so that it all was a "long shot." But this would require all locations be drawn as if they were being viewed from the same distance. While this was acceptable for many locations like the sand dunes and the meadow, it would force the beach scene with the canoe to look distant and bare. The canoe would have to be much smaller and way off in the distance. It would not longer appear that the player was standing by the canoe, and therefore would not make sense that the player could reach down and push it into the water.

The close up shot of the tree top would be even worse. To show the tree top from the same perspective would require the eye of the viewer to be away from the tree top by quite some distance. It would make any dropped objects look the right size, but wouldn't appear as though the player was sitting in the tree top. Instead, it would appear that the player was suspended in mid-air in back of the tree top. So changing the scenery was not the solution.

The other possibility was to change the objects. I could have three different-sized objects in memory, and place the appropriate one on the scenery when it was supposed to be there. For example, I could have three different bottles that all looked alike, but were different sizes. When the player dropped the bottle in the meadow, the game would place the small drawing of the bottle in the appropriate spot on

the screen. If the player dropped the bottle on the beach, the program would put the medium sized bottle on the sand beside the canoe. And if the player was sitting in the top of the palm tree, the large sized bottle would appear sitting on the palm fronds beside you.

All of this was possible, but it was starting to grow exceedingly large for such a simple game. To use this approach would require 57 drawings for the nineteen movable objects, three drawings for each object. And each of these drawings would have to be assigned to one of 228 possible X,Y coordinates—each of the nineteen objects needed to be assigned to special places on twelve different location scenes. Even though my goal of putting movable objects on the location scenes was getting way too complicated, I still was considering trying it. But further problems arose with the concept.

Resolution and Palette Problems

The low-resolution screen is 320 pixels across by 200 pixels down in NTSC, 320x256 in PAL, and allows 32 colors. I chose this over a higher resolution because I wanted the additional colors that only low-res could give me.

Another reason I chose low-res was that a screen in low-res takes less chip memory, and I knew I was going to be using a lot of chip memory for the various screens that would be in memory all at once, as well as the digitized sound effect that I wanted to include. All of these things use chip memory, and I needed to make sure my game would run on a minimum Amiga with 512K of Chip RAM.

But when using low-res, **aliasing** can be a problem. I discovered anti-aliasing problems with the low-resolution screen. Aliasing refers to edges of curved and diagonal lines looking jagged because the pixels are larger than in other resolutions, and **anti-aliasing** is the technique used to reduce these "jaggies".

One way to reduce them is to add colored pixels in the corners of the jagged edges that are a mixture of the object color and the background color. Doing this makes the curves and diagonals look much smoother. But the problem I encountered was that while an object would look fine in one location, it would look awful in another location. If the bottle were drawn so it would look good on the sand, then it would look terrible on the tree top. The sand-colored pixels added to reduce the jagged edge when the bottle lay on the sand made the bottle look horrible on the green background of the tree top, even though the bottle looked wonderful against the tan background of the sand.

There was a solution for this problem too, but it got even more complicated. I could have different drawings for each location that would properly match the colors of the backgrounds. This would mean at least six different pictures of each small-sized object, six more of each object in the medium size, and six more in the large size. That way I

could have the program draw the correct object if it existed in any room. It would be the correct size to match the perspective, it would have the correct anti-aliasing colors to reduce the jagged edges, and it would be placed in the correct position on the screen to appear in the location properly.

As I considered all the dizzying possibilities of eighteen different drawings for each of nineteen different objects, which could be placed in 228 different places on the screen, I realized there was an easier way. Such a simple idea, having the movable objects appear in the location scene, couldn't be that hard. And by now you have played the game, and you've seen how I resolved the problem.

I decided to place all the movable objects for any given location on the side of the location in a white area that would scroll upwards or downwards if necessary to show all the objects in that place. This solved all of my problems and still accomplished my goal of showing the objects on-screen for the player to view. Using this method, only one drawing of each object was necessary. No different-sized objects were required. All the objects could be anti-aliased toward a white background, so no different colored objects were needed. And since each object appeared in a given area, no special coordinates were needed for each specific location.

The white bar has room to show up to five objects that are in the current room. If more than five are present, the bar can be scrolled to show the rest. By taking this approach, all the movable objects would appear on the screen, and could be moved by the player from the room location to the inventory or back. With this, the hardest nut to crack, resolved, I was finished designing the graphic interface.

Now I had the game completely designed on paper and in my head. I knew what I wanted it to look like, and how it would play. The next step was to create the graphics. This had to be done before any programming, because much of the programming depended on knowing exactly where on the graphic screen various things were. In the next chapter, I'll cover the graphics, music and sound effects that went into *Cannibal*.

Chapter 17: Sound and Graphic Files

Since I was creating a graphics adventure, I knew the graphics would have to come next. I needed to know exact pixel locations, so that as I wrote the rest of the adventure, I could correctly refer to them, and get things moving properly. For example, to pick up the bottle, I wanted the player to simply click on the bottle and drag it over to the inventory window. This required the use of Visionary's COPY command.

To use the COPY command, I had to know which of the 25 possible screens contained the picture of the bottle. Then I had to know the coordinates of the upper left corner and the lower right corner of a rectangle that contained the picture. I also had to know which screen the picture of the bottle was to be drawn. And I needed to know the coordinates of the upper left corner of the rectangle where the bottle was to be drawn. I also had to know the coordinates of the scroll-bar window and the inventory window, so I would know if the player was clicking on the scroll-bar window, the inventory window, or neither.

Before I could start writing the source code for the adventure, I needed to know the precise pixel coordinates of all the graphics that would be used in the game. For these reasons, the graphics were necessarily the next step in the creation of the game.

The Graphics

Lacking all but the most rudimentary artistic skills myself, I was lucky to find an excellent artist to do my art work for the game. I discovered Erik Hermansen by leaving a notice on some of the larger Portland area bulletin board systems.

If you also lack artistic ability, I can recommend this as one of several ways to find someone do help you with your game. You will be looking for a rare combination of talents, someone who has artistic ability as well as a person familiar with computer art, and with paint programs. If you have a telephone modem and can call a local BBS or two, this is an excellent way to seek artists for your game.

You can also check with local colleges and universities who frequently have computers in their art departments. They may be able to put you in contact with some art students who possess the skills you need. Other sources include your local computer user groups and computer stores. They will frequently be able to put you in touch with someone who does computer art work.

The Window Template

Erik worked closely with me in designing the graphics screens for use in *Cannibal*, using Deluxe Paint III. Our first task was to design the window template that would be the screen the player saw. This screen contained three windows, the location window, the inventory window, and the text window. It also contained the buttons that the player would click on to move or take other action. This screen, named "window.pic", can be found on the disk, in the **Video** directory.

As we were designing the window template, we were also taking into consideration the possible sizes of the movable objects. They had to be big enough to be recognizable, but at the same time small enough to easily fit in the scroll-bar window and in the inventory window. We finally settled on a size of 15 by 17 pixels. Erik would design each of the nineteen movable objects to fit inside an invisible rectangle that was 15 pixels across by 17 pixels down.

Having settled on this size, we were then free to correctly size the inventory window to hold six of these objects, with 1 pixel width in between. We were also able to design the scroll-bar window on the side of the location window to permit up to five objects be displayed, again with a 1-pixel border.

Coordinates

Keep track of coordinates as you create the game graphics

The next step was to keep a written record of coordinates for each window and button on the window template. In Visionary, the COPY command needs three sets of X,Y coordinates each time you use it. It needs the upper left corner and lower right corner coordinates for the rectangle containing the part of the screen to copy from, and the coordinates for the upper left corner only, for the screen it will copy to. Since all the graphics would be copied to **window.pic**, not from **window.pic**, it was only necessary to record the X and Y coordinates of the upper left corner of the location window, the inventory window, the text window, the scroll-bar window, and all the buttons. These numbers would be vital later in programming, to make sure that the graphics would be drawn on the window template correctly.

You may need to correct for a discrepancy between the coordinates in your painting program and those in Visionary

Depending on the paint program you use, there may be a discrepancy in the way in which the y coordinate is measured. Visionary uses the standard "over-down" method of figuring coordinates, where the upper left corner is (0,0). Earlier versions of Deluxe Paint III, the paint program we used, force you to use the "first quadrant" approach of a mathematical coordinate plane, where the lower left corner is (0,0). To give an example, the upper left corner of **window.pic** would have coordinates of (0,0) in Visionary, but would have coordinates of (0,255) in Deluxe Paint III.

Normally, a low-res screen like **window.pic** would have 320x200 resolution, and the upper left corner would have coordinates of (0,199) in Deluxe Paint. However, in an effort to make this game more visually pleasing when played on a PAL computer, I added an extra 56 pixels of length to the screen, making the resolution 320x256. So the corrected coordinates for the upper left corner in Deluxe Paint III are (0,255).

The Hidden Screen

We had now designed the main screen, the window template, and knew exactly where everything was located. The next step was to design a screen which would remain hidden from the player's eyes, but would contain the pictures of the objects and the pictures of the buttons in both states: pushed in and popped out. Parts of this screen would be copied to the visible screen when the player moved objects or clicked on the buttons.

If you look on the disk in the Video directory, you will find this screen as a file named **buttons.pic**. It can be viewed with any graphics viewer or paint program. What you will see is all nineteen objects sitting in a line across the bottom of the screen. Above them are pictures of the buttons. Above that is a blank rectangle that is used for a hidden buffer, and beside that is a larger picture of the ladder. Let's look at each of these and I will point out some of their special features.

Object locations on the hidden screen were designed to be a simple multiple of the object number to make programming easier

In placing the object pictures at the bottom of the button.pic screen, I had to plan ahead. I knew that I would eventually want to refer to each object by a number, so that I could run that number through a simple formula and come up with the X and Y coordinates of the object's location on the **button.pic** screen. I placed the left edge of each object exactly 15 pixels from the left edge of the previous object. In this way, the X-coordinate would be a multiple of 15 of the object number. For example, object number one was the ladder and its location would start at zero. Object number two was the bottle, and it began at fifteen. Once I knew the object's number, I could subtract one, and multiply the result by fifteen to obtain the X-coordinate. Since the objects were all drawn in a line from left to right, the Y-coordinate for each would always be 183. By designing the **button.pic** screen in this fashion, I was paving the way to make future programming easier.

Object Edges

While you are looking at the objects, notice that the edges of the objects may look strange. This is because all the objects sit against a black background. During the game play, however, the objects will be placed upon the white background of either the scroll-bar window or the inventory window. When placed against a white background, the objects will look normal.

The reason for the strange appearance on the black background of the hidden screen is the pixels added to the edges of the object for the purposes of anti-aliasing. By adding pixels in the corners that are a combination of white and the object border color, the object takes on a smoother and less jagged look. The result looks strange when viewed against black, but looks fine against the white background where it is intended to be displayed.

The Buttons

Next, notice the buttons. Each action button has a twin. They come in pairs, such as two versions of the **HELP** button: one shows the button in its normal position, and the other shows it in the depressed position. All buttons except the direction buttons have two pictures, one showing the button out, the other showing the button in.

Again planning ahead, I placed the depressed button picture immediately below the normal button picture. This was done in anticipation of the future programming required to make the button look like it was being pushed inward. When using the COPY command, the X-coordinate would not have to be changed. The Y-coordinate would only change by 13 each time.

The only exception was the direction buttons. Since I had completely planned out the graphic interface in advance, I knew that each direction button could take on three possible shapes. I designed three buttons for each direction button. If the player was allowed travel in the indicated direction, the button would appear in the normal "out" display with the direction in orange. If the player could not travel in that direction, the button would still appear out, but the direction would be "ghosted" in gray. The third picture would show the button as it would appear if depressed, with the shadows on the edges reversed, and the direction in yellow.

By positioning the depressed version of the button in between the normal and ghosted versions, I knew I would later be able to easily return the button to its previous condition by adding either 13 or -13 to the Y-coordinate of the button. This value, which I called an offset, could be set depending on the current status of the button and the directions, as soon as the mouse click was detected. Then it would only take a single routine to return the direction button to normal when the player released the mouse button. I'll explain more about this routine in a later chapter on the **MainLoop.SUB** file.

Above the buttons to the right is a larger picture of the ladder. I knew that in certain locations, I would want the ladder to appear in the location window propped up against the shack, the tree, the boulder, or the cave. This would be in addition to the smaller picture showing in the scroll-bar window. So the ladder was placed here, where it could be accessed at any time, to overlay it on the location scenery.

The large white rectangle in the upper left corner of buttons.pic was designed as a hidden buffer for overlaying the ladder on the scenery before the scene was copied to the window template.

A hidden buffer lets the ladder appear in appropriate screens without noticeable delay

I designed the sequence of events to be as follows. When the player entered the meadow, the meadow scenery was loaded from the disk into screen buffer 1. Players wouldn't see this happen, because they would be looking at the window template on screen number 0. After the meadow scenery was loaded into screen buffer 1, I copied it to the white buffer in the buttons.pic file, which was in screen number 2. If the ladder was present at the current location, I did an overlay of the ladder onto my hidden buffer. Then I copied the hidden buffer to screen number 0, into the location window. All this took place rapidly—from the player's perspective, after **South** was clicked, the meadow with the ladder against the shack was immediately displayed.

I could have eliminated the middle step, and the need for the white rectangle on **buttons.pic**. I could have loaded the meadow scenery into screen buffer 1, and then copied the meadow directly into screen buffer 0 and the location window. Finally, I could have copied the ladder directly onto the scenery if it was present. The end result would have been the same, but it would not have looked as smooth. The meadow would have appeared in the location window first, and then the ladder would have appeared a fraction of a second later. It would have been noticeable, and would have looked less professional.

So I chose to do my overlays in a hidden buffer first, and then copy the final scene onto the location window. The end result looked nicer, but took extra programming steps and took some space on the **buttons.pic** file. For a more detailed look at how objects like the ladder are overlaid on the scenery, see the complete explanation for the subroutine I called **ReDrawScreen**, in Chapter **.

The Game Graphics

At this point, Erik had presented me with two completed graphic files exactly to my specifications. I knew the X and Y coordinates of every window, object, and button in both files. The next step was to design the art work for each of the location scenes. Each room would have at least one location scene—some would have two. I needed two different versions of the east beach scene. One showed the canoe beached on the sand, and the other shows the canoe floating at the surf's edge. Similarly, I needed two versions of the scene by the boulder, one showing the boulder in front of the cave and the other showing the boulder rolled aside.

In addition to the scenes for each of the rooms, I also needed a scene when the player won and when the player lost. The winning scene would show the player having left the island, and on his way to safety. The losing scene would show the player caught by the cannibals, sitting in a large iron pot filled with boiling water.

Each of these scenes were designed to be exactly 227 pixels long by 120 pixels wide. Once they were completed, it was time to do some more measuring.

Counting Pixels

Counting pixels is tedious work, but it had to be done if I wanted the player to be able to click on the scenery and receive a description for that part of the scene. Remember, it was my intent that the player would be able to examine things by simply clicking on them.

That not only meant the movable objects in the scroll-bar window and the inventory window, but the nonmovable objects in the location window as well. If players were standing on the beach by the rowboat, I wanted them to be able to click on the picture of the rowboat and receive some description of the vessel. So for the scene showing the west end of the beach by the rowboat, I had to keep a written record of the X and Y coordinates for the upper left corner of a rectangle containing the rowboat, and similar coordinates for the lower right corner of the rectangle containing the rowboat. I would need these numbers later when I started programming the click zones for the room files.

In case the player wanted to examine other parts of the location, I had to measure pixels for all the other things in the scenes. Standing by the rowboat, for example, I wanted the player to be able to click on objects other than just the rowboat. I needed to find the double set of coordinates for the sky, the sand, and the ocean. Or if the player was standing in the sand dunes, I wanted to allow a click to examine any of the four plants present, the ocean as seen between the sand dunes, the sky, or the dunes themselves. Again, this meant I needed to know the coordinates for each of these areas. In that way, when the time came to program these click zones into the room files, I would have the exact numbers available for each examinable object.

Use the VCOORD utility to make pixel-counting easier

A utility program named **VCOORD** that comes with Visionary makes counting the pixels much easier. It allows you to load your graphics and find the coordinates by simply clicking the mouse button on the picture. The coordinates are then displayed in the upper left corner of the screen, or can be written to a file or sent to your printer. When using the optional **VIE** editor, available separately, the coordinates will be recorded by simply clicking the mouse on the graphic.

Sound Effects and Music

Having completed the graphics, the sound effects were next. These were easy, fun, and a nice change of pace from counting pixels. I had a short list of sound effects that were required. I would need some ocean sounds for the two beach scenes. I wanted some bird sounds for the meadow. And I wanted some sort of cave sounds for inside the

cave. This last one was a bit difficult, until I finally settled on a hollow dripping sound that I felt was typical of a damp cave.

These would be the main sound effects in the game. In addition, I wanted two more sound effects for when the cannibals captured the player. I wanted the sound of a bubbling pot, and the sound of a man's scream. This completed my list of sound effects. The next task was to find them.

Finding or Making Sound Effects

My first step was to look at sound effects that already existed as digitized sounds available from a variety of public domain sources. It was from these sources that I found a good dripping sound, and an excellent death scream.

When you can't find the sound effects you need, it is easy to make your own. That's what I eventually did for several of them. Since I could not find some of the sounds I needed already digitized, I used an audio digitizer to create the rest. Audio digitizers are readily available in the $75 range, and are very easy to use. You simply plug them into the back of your computer, connect to your audio source, and run the software. Many digitizers come with their own software, which may be limited to digitizing the sound. After sounds are captured by the digitizer, they often need to be edited. For this purpose an audio editor is needed. I recommend AudioMaster III because of its excellent sequencing abilities as well as full editing capabilities.

Repeated sounds can be randomized to avoid "broken-record" repetition

For the sounds that I personally digitized,, I used several sources. For the birds chirping, I found a record containing chirping birds. I digitized a short sample, and then looped through various parts using the sequencing of AudioMaster III. This made a short sample lasting only a few seconds into one that played for several minutes. By randomly choosing the bird chirps in the sequencing, I was trying to avoid the "broken-record" sound of birds chirping in exactly the same way, over and over.

For the ocean waves, I used a "white noise" generator on a clock radio, that was supposed to lull you to sleep with the sound of the ocean. Using a microphone, I digitized the white noise while manually adjusting the volume to give the effect of crashing waves. Then I processed this sample with AudioMaster III to add sequencing of the waves, to make the sample run longer and make the waves crash more randomly, and less in a recognizable pattern. That left me with the bubbling pot sound.

Creating the digitized sample for the bubbling pot was one of the high points of that week. I couldn't find the sample already in existence, so I decided to create my own sound effect from scratch. The basic concept was that I would blow bubbles in water with a straw while holding a microphone above the water to catch the bubbling sound.

With some inventive experimentation, you can often create the sound effects you need

The first thing I discovered was that different containers create different sounds. Using a glass of water created bubble sounds that were too high pitched and had a "ringing" sound as though someone were lightly tapping on the glass. I switched to a large metal pan and found that the bubbles had a lower pitch, but there was still a metallic "ringing" sound to the bubbles. I finally settled on a large soft plastic bucket that eliminated the "ringing" sound while it kept the lower bubbling sound.

I next discovered that blowing into a single straw didn't give enough bubbling action, so I put two straws in my mouth. Blowing lightly didn't give me the big bubbles and the necessary large "plopping" noises, so I found the best result was obtained by blowing hard and getting large splashy bubbles. Of course, I wasn't able to maintain blowing air out at that rate for very long, but I knew I could loop a short sample to make it appear much longer.

So away I blew, microphone held above the plastic bucket, water bubbling like crazy and splashing all over the place, and me probably looking like the big bad wolf trying to blow down one of the little pigs houses. And when I was done, I had it! I had a perfect digitized sample of a bubbling pot. I also had a kitchen that was wet from top to bottom and a microphone that is still drying out to this very day. It was so funny I had to laugh—but it was also a laugh of triumph. I had created the perfect sound effect for my game.

The point of this story is that you ought to try your own hand at creating sound effects, too. It's easy to do, and often a lot of fun as well.

Playing Sounds

Except for the death scream, I wanted all the sound effects to play continuously. I didn't want the player to hear fifteen seconds of ocean waves, and then no more. As long as he stood at the ocean, I wanted him to hear the waves. And when he moved away from the beach toward the meadow, I wanted the sounds of the waves to diminish as he moved farther and farther away. Likewise, I wanted the player to hear the birds chirping in the distance as he approached the meadow, and then hear the volume increase as he reached the meadow by the shack.

To keep the sounds running continuously, I knew I could set the Visionary PLAY SOUND command to a continuous setting. But to avoid the "broken-record" sound, I processed the digitized sounds with AudioMaster III first, so various parts of the short samples could be played in a sequence that would appear random, and would last long enough so that when my game started them a second time, no one would notice. While I had done this with the sounds I personally digitized as I was working with them, I now went back to the drip sound that I had found on a public domain disk and sequenced it so the drip would sound a bit more random.

Music

With the sound effects finished, that left me with music. There were two places in my game that I wanted music. I wanted some title music to be played while the game was loading and the main title was being displayed. I also wanted some music to be played at the game's conclusion, when the player won and was being transported back to civilization.

Visionary supports music modules created with MED (Music Editor), and using this type of music saves an enormous amount of memory compared with digitized music of the same length. Erik Hermansen, who did such a nice job with my graphics, created two musical scores for me using MED. The first one, as you have by now heard, was the native drums that plays at the game loads. The second one, which you have only heard if you have won the game, is played when you have reached the safety of the old freighter. The first piece with the drums was so effective, that I also decided to put it at the end of the game, when the cannibals arrive on the island.

With all the graphics and sounds completed and out of the way, the next step would be to start the actual programming, using Visionary. The first thing to program would be the **Cannibal.ADV** file. The next chapter will take a look at how it was constructed.

Chapter 18: The Cannibal.ADV File

When I sat down at the computer, I knew that the **Cannibal.ADV** file would be the first one I needed to create. The Visionary manual covers the format of the **{game}.ADV** file, so I needn't explain it again here. But if you will look at the source code for the **Cannibal.ADV** file, either in Appendix A or on the disk, you will notice a large number of variables.

These were not all entered when I started the game—in fact, very few were entered at that time. But as the game was being written, I came back here and added variables as I needed them. When you write your own game, you will find yourself doing much the same thing. You will be coming back to the **Cannibal.ADV** file many times to make additions as your game grows.

Password

The first entry in the **Cannibal.ADV** file is your personal password. This has several purposes, and hence should be chosen carefully. One of the purposes of the password is to correctly decode your encoded graphics and sounds. Many of you will want your graphics and sounds encoded (using the VCODE program found on your Visionary disk) so that after your game is finished and released, others cannot make modifications to your pictures.

When you encode your graphic and sound files, you must use the same password as in the **Cannibal.ADV** file. Without this password, your game will not be able to correctly decode those files. If you choose an obvious password, anyone with the VCODE program could unscramble your files, modify them, and encode them again. So for the sake of security, be sure your password is unusual.

Use a unique password for each Visionary game

Your password should also be unique. That is to say, you should not use the same password on two different adventure games. This is because the password is also used in identifying saved game positions. Consider what would happen if there was no way of identifying the game the positions were saved from. Let's say you were playing a game called *Magician* and saved your place, and then started playing a second game called *Jungle*. If you tried to load your place using the file saved from *Magician* there would be nothing to stop you. But the game would crash, because all the screens would be different, the variables would be different, and the objects would be in the wrong places.

To keep that from happening, Visionary checks the saved game with the password in the **{game}.ADV** file. If they don't match, the saved position will not be loaded. This important safeguard would be com-

promised if you used the same password in both games. By having a different password for each game you write, you will ensure the only previous positions that can be loaded into your games are ones that really belong there.

The Variables

The variable block follows the password in the **Cannibal.ADV** file. It may look intimidating when you look at the source code for *Cannibal*, but remember that it started out small, and grew as the need for variables arose.

If you are writing a strictly text adventure, your variables will probably be few. Since Visionary has built-in variables for **SCORE**, **MOVES** and **ITEMS**, you might not even need to declare any variables at all. In a graphic adventure, however, there is a need for a wide assortment of variables. If you examine the source code for the **Cannibal.ADV** file, you will see comments beside each variable. These comments give you some general idea of what the variable is used for. As I go through the development of the game, I will expand on these comments when necessary. For now, just look over the list and you will see that variables are used mostly for graphics purposes. Many of the variables keep track of positions on the graphic screens for buttons, objects, scenes, and text placement.

When I began the writing of this game, I filled in the **Cannibal.ADV** file with only the barest of information. I chose a password. I knew I would need three variables, so I inserted them. I knew I would have to write a **.ROOMS** file, so I entered it into the ROOM block. Then I had to make some decisions about the object files.

Object Files

I knew I wanted to have three different types of object files. There would be some actions that I wanted to take place on every turn. I wanted to check my timer on every turn and see if it was time for the Cannibals to land, or time for the player to be captured. To ensure that this check for time occurred at the end of every move, I had to place them in an **NPC file**. As the Visionary manual points out, there is a special object file called NPC which always executes at the end of each turn. So I would need an NPC file.

I have found that all the other objects in my adventures can be split into two groups, one for nonmovable objects and one for movable objects. There is no reason you cannot use a single file, but I have discovered that it's easier to find things later if I have the files kept separate.

It may help to have separate files for movable and nonmovable objects

In addition, the nonmovable object files are usually shorter because the objects can't be manipulated as much. The movable object files are longer because they permit the getting and dropping of objects as well as other uses of the objects like burning, opening, reading, breaking, locking and unlocking.

I put the three different file names in the OBJECT block of the **Cannibal.ADV** file. As you can see from the source code, I later split the movable objects into two separate files because the single file was getting too big.

Subroutines

Subroutines play an important part in any adventure, especially a graphics one. When I started writing the adventure, I only saw the need for a single file of subroutines. I entered the file name **Cannibal.SUB** in the subroutine block.

Some time later, I decided it would be easier to place some of the subroutines in separate files. I created a **GetDrop.SUB** file of subroutines that were involved in making objects appear in the scroll-bar window and the inventory window. These subroutines also took care of things like moving the objects around the screen behind the mouse pointer, redrawing the scrollbar as items were scrolled upwards or downwards, and making the scroll-bar window arrows appear when necessary.

I created a **StartUp.SUB** file for the subroutines that were only called once at the beginning of the game, to load the initial graphics and sounds. And I created a **MainLoop.SUB** file for the large subroutine that checked for any keypress or mouse click, took any appropriate action, and then started over again.

Vocabulary

The next step was to fill in the VOCAB block. I knew there would be many special commands that would not fall elsewhere in my source code, and so I created a file **Cannibal.VOC** to take care of these situations. Take a quick look at the source code for this file, and you will see the types of commands it includes. A detailed explanation of what they do and why they are there instead of someplace else will be given later.

The Initial Room

The last thing I had to do in the **Cannibal.ADV** file was fill in the name of the room where my game started. Since the game was completely designed on paper before sitting down to the computer, this was easy. I knew the player would start on the beach by the rowboat. All I had

to do was to choose an official name for the room and place it after the **INITROOM** line.

Choose descriptive room names

Since this room was at the west end of the beach in my game map, I picked the name "west_end_of_beach" to make it easier to remember. Having completed as much of the **Cannibal.ADV** file as I could, I was ready to go on to the next step in creating my adventure game.

Chapter 19: The Cannibal.ROOMS File

The next step was to create the actual rooms or locations that the player would visit, like the beach, the shack, and the cave. All the rooms went in the **Cannibal.ROOMS** file. I designed twelve rooms, plus two "fake" rooms.

Hidden, Unused and Special-Purpose Rooms

The first fake room was the one I wrote first. I named the room "Unused" to remind me that the purpose of this room was to store objects that were currently not being used. For example, I had buried some driftwood in the sand by the rowboat. I would place the driftwood object in this unused room until the player dug it up. At that time, I would move the object from the unused room to the player's current location. This room was also used for the storage of objects that were destroyed, like the bottle when broken, or the candy bar when eaten.

At a later time, I decided that I could use the "Unused" room for another purpose as well. I wanted to allow the player to attempt to swim out in the ocean. It would not have been logical to deny the player access to the ocean, especially when standing at the edge of the surf with the water stretching before him. What excuse could I have used for refusing to let the player try to swim? Tell him he didn't know how to swim? Pretty lame. Tell him he couldn't go that way? Not logical; he could see it before him. But at the same time, I didn't want swimming to lead the player anywhere, especially not out to the freighter where the game would have been won.

I chose to allow the player to swim about in the ocean a bit, then tire and return to shore. To do this, I had to create a new room. Since there was no need to actually show this room graphically, I knew I could use the "Unused" room for this purpose. I added the code to the "Unused" room so that when the player stood by the rowboat and tried to go north, he would be moved to the "Unused" room, and immediately be sent back to the location by the rowboat, where he would be given a message telling him that after swimming a bit he had tired and returned to shore.

If you will examine the source code for the **Cannibal.ROOMS** file, lines 4-9 show how simple the "Unused" room file really is. The file consists of a two-line code block that sets a room attribute and then forces the player back to the "west_end_of_beach." The room attribute, called **ForcedReturn**, is for the purpose of returning the player to the beach without printing the room description again.

When the player is sent back to the "west_end_of_beach", the lines starting at 53 unset the attribute so that it won't be activated normally again, and call a subroutine to print the message telling him he returns to shore after a swim. If you're curious about the subroutine that prints the message, feel free to look it up in the **Cannibal.SUB** file, starting at line 628. Be warned, however, that many subroutines call other subroutines which call other subroutines. It can get a bit dizzying unless you know why things were set up in that fashion. We will be looking at subroutines in a later chapter, and you may wish to wait until then.

The "Unused" room is the simplest type of room file, consisting only of a very small code block. The only other room like it in my adventure is "Ocean2" found at line 78. "Ocean2" serves a similar purpose to "Unused"—it is where a player is sent when he attempts to swim out from the east end of the beach by the canoe. As soon as the player arrives here, a similar attribute is set for the "east_end_of_beach" and he is sent back to the beach by the canoe.

Repeated code is often a good candidate for a subroutine

When the "ocean2" room file is executed, that attribute is cleared and the message about swimming is given. Since the same message was given from two different rooms, "west_end_of_beach" and "east_end_of_beach", a single subroutine was used to save space. The alternative was to place the same message in both rooms, however that was less efficient. Whenever you find yourself writing the same block of code more than once, remember you can make it a subroutine and you will save a lot of space.

The Initial Room

The second room that I wrote was the "west_end_of_beach." This was probably the largest of the room files. Since it is the room the player is in when the game starts, it must also contain the necessary code to set up the game. This set-up includes such things as loading the various graphic screens, loading sound effects, loading the fonts, and defining the click zones for the buttons.

Since this was such a large amount of code, I decided to make it a separate subroutine which could be called from the room file only once, when the game first started. In the next chapter, we will take a complete look at the **StartUp.SUB** file and see what it includes. For now, let's look at the rest of this room file, and the other room files.

Descriptive room names will make your task easier

The first part of the room file is the room name. I chose "west_end_of_beach" because it was descriptive. You could just as easily call your rooms by names such as "Room1" and "Room2", but it is easy to lose track of which room is which when the names are less descriptive. You may be able to remember them by numbers today, but in six months when you go back to look at your source code again, you will have forgotten which room number is which. With descrptive room names, you will make your job easier in the long run.

Room Attributes

The attribute block comes next in the room file. Remember that these attributes are simply variables with only two possible values, Y(es) or N(o). For those of you familiar with other programming languages, they are essentially flags. But they are more user-friendly, and they are "attached" to a room. By that, I mean that they are values associated with a particular room.

For example, look at the first attribute for this room, called **Started**. I created this attribute to describe whether the game had just been started or not. Since this room is the first one executed when the game begins, this is the logical place for such an attribute. Since the attributes are attached to the rooms, you can have several attributes with exactly the same name, each attached to a different room.

The same name can be use for an attribute in different rooms

The attribute **ForcedReturn** is a good example of this. I have used the name ForcedReturn as an attribute for both the "west_end_of_beach" and the "east_end_of_beach." There is no conflict in using identical names for these attributes, since each will only be checked against the room name that I specify. If you will examine lines 53 and 122 in the **Cannibal.ROOMS** file, you will see how each is used with the room name.

Notice that both attributes are set to N, meaning "No." Remember that these are set to either Y or N the very first time the room file is executed, and never again. The second or third time the player visits the room, the attribute block will be skipped. In this way, if any attribute has changed value, say from N to Y, it will stay changed when the room is re-entered. I set both values to N to indicate that the game had not yet been initialized (started) and that the player had not been forced to return to this location after a swim in the ocean. These were the only attributes that I needed, but remember that you can use up to 32 attributes if you wish.

Automatic Attributes

Before I move on to the default block, let me remind you of some attributes that are automatically set in Visionary without the need for you to do anything. Each room has the attributes VISITED and DARK. Both start out set to N unless you specify otherwise, and neither needs to be listed in the attribute block unless you want to set the initial value to Y. The VISITED attribute always starts at N and automatically switches to Y after the room has been first visited.

By checking this attribute, you can decide whether to print a complete room description, or a short summary of the description. The DARK attribute also starts at N, and can be used to tell if a source of light (a burning torch or a flashlight) is needed in order to give the room description. You might use this if you have a dungeon, where you want the player to have a torch before she can see where she is.

In my *Cannibal* adventure, I didn't need the DARK attribute. I used the VISITED attribute in nearly every room, but since I was satisfied with the default value, I didn't declare it in the attribute block. For more information on the VISITED and DARK attributes, see your Visionary manual.

Default Directions

The next block in this room file is the default directions. As you can see, starting at line 20, there are three ways the player can exit this room. She can move south to the sand dunes, east to the east end of the beach, or north to the unused room. Remember that the player thinks she can move north into the ocean, but actually is moved to the unused room just long enough for an attribute to be set, and then is sent back to the beach.

Here is another case where descriptive room names will make your job easier. It's much easier to tell where the player will end up when moving south if the room is named "sand_dunes" than if it is named "room3." It's not only easier for you to keep track of your game as you write it today, it is also much easier when you look at it months from today.

As is pointed out in the Visionary manual, the default directions are only executed once, when the room is first visited. That means if the directions are changed later in your game, they will not be re-initialized back to these default values when the player comes back into this room. A good example of this feature in the *Cannibal* game is the use of the ladder. When the game starts, the player can climb the ladder that is against the shack to reach the roof of the old shack. You will see in line 202 that the default for **up** is "top_of_the_shack." That means if the player tried to go up, he will be taken to the roof top.

But consider what happens when the ladder is moved. The player should not be able to go up to the shack roof if the ladder is not present. When the ladder is moved, the directions available for the player are modified so that **up** is no longer a viable action. And that's the way you want it to stay, even if the player walks to the sand dunes and then back to the meadow by the shack. When the player returns to the shack the second time, the default directions block is not executed. The **up** direction remains unmodified, so that if the ladder has been removed and the **up** direction is no longer available, it will remain that way when re-entering the room.

The Code Block

The next, and final, block of this room file is the code block. It is the largest block and contains a wide variety of things.

Remember that the code block is always executed any time the room is entered, whether for the first time, or the twentieth time. This is the

appropriate place to put your room descriptions, define your click zones, and set variables. Let's examine the code block for the "west_end_of_beach."

Click Zones

The first thing I did in the code block was to clear the click zones for all buttons used in any previous room, then define the click zones for the current location. Line 28 calls a subroutine ClearButtons which clears the click zones for button numbers 34-44. If you wish to examine the subroutine, you will find it in the **Cannibal.SUB** file starting at line 144. We'll examine this subroutine in a later chapter, but for now let's just look at the reason for clearing the buttons.

Note that not all the buttons are cleared. I used button numbers 0-14 for such things as clicking on the compass, clicking on action buttons like **HELP**, **DIG** and **QUIT**, and clicking in the scroll-bar window and the inventory window. I wanted these always active, and did not clear them when the player moves from room to room.

However, there were items that would appear in the graphic scenes of each room that I wanted the player to be able to examine by clicking on them. For example, I wanted the player to be able to click on the picture of the canoe and receive a description of it. But when standing in the cave and clicking on the same screen location, I didn't want the player to read the description of the canoe. So in the code block, I first cleared the buttons that were assigned to nonmovable objects in the scenery.

Some areas of your screen may be assigned a lower priority than others

After clearing the old click zones, I defined the new zones for the boat, the sand, the ocean and the sky. Each was assigned to a particular button number. The numbers are important, because they are **prioritized**. The location window's coordinates are 6,6 for the upper left corner and 231,71 for the lower right corner. Notice that I defined the sky to take up the whole location window, but I also defined it to be the lowest priority of the four zones. In that way, anywhere the player clicks which is not otherwise defined will display the description for the sky.

If you examine the coordinates for the four click zones carefully, you will realize that the four rectangles they define overlap each other. By taking advantage of the priorities, I could save a lot of programming time.

For example, let's look at the rowboat and the sand. I defined a large rectangle that contained the sandy beach. Part of it also included the rowboat, but that was unintentional. Then I defined a smaller rectangle for the rowboat, which was overlaid on the other rectangle. By using a smaller button number, I knew that clicking on the portion of the screen that was defined to be both rowboat and sand would only describe the rowboat.

If the click zones were not prioritized, I would have had to define the click zone for the boat in the center of the screen, then define a click zone for the sand to the left of the boat, another zone for the sand to the right of the boat, and a final zone for the sand below the boat. Visionary's feature of prioritizing buttons makes defining click zones much easier. Keep this in mind when you design your adventures.

Nonmovable Object Placement

Having finished defining click zones, I next placed some nonmovable objects in the room. These objects, the ocean, the sand, the sun, and the sky, have to be in the room so that they can be described.

When Visionary's parser sees the command "examine the sand" it looks to the object file for "sand" and prints out the description I gave it. That means the object "sand" has to be in the current room. And since there is sand in more than one room, I just keep moving it to the current room as the player enters it. That means if the player was elsewhere, perhaps in the sand_dunes, and enters this room next to the rowboat, I place the object "sand" in this room so that if the player tries to examine it, get it, or otherwise manipulate it, my game will find the object and respond appropriately as I wish.

Any object that would appear in only one location, the shack for example, would not need to be placed in the room in this fashion. The single object location could be specified in the **INITROOM** part of the object file.

We'll be looking at the object files in much more detail in a future chapter, so I'll say no more here. But this is the place to define objects that appear to be stationary but in reality need to move from one room to another.

Initial Variable Settings

In line 39, I set the variable DIG to zero. There are four things that can happen when the player tries to dig. If he is in a sandy area, he can dig without any tools. If he is in a dirt area, he needs a shovel to dig. And if he is in a rock or wood area, he will never be able to dig. I used the variable DIG to tell which type of area the current room was.

The code block of each room contains a line defining DIG. When the player attempts to dig, I only have to check the value of the variable and give the proper response. Just remember, if you use this type of technique you must go back to the .ADV file and include DIG in your list of variables. If you try to compile your game without it, you will receive an error message.

In line 41, I set a variable called **RoomNumber** to one. This number actually refers to the graphic picture that the computer will show in the location window. This variable is added to the graphics filename, so

the computer will load the correct file from disk and display it. If you wish to see how this is done, look at the file called **Cannibal.SUB** at line 308. If it looks confusing, don't worry. The entire routine will be explained later in the chapter on subroutines. For now, you need only remember that the RoomNumber variable stands for which picture to show for the current room.

Some rooms, like the "east_end_of_beach" by the canoe, have two different values for RoomNumber depending on whether the canoe is on the sand or in the water. When the player enters this room, the computer will know which picture to display depending on the value of the variable.

The StartUp Call

The next seven lines make up a conditional statement checking to see if the game has started. If the game has not started, a subroutine called **StartUp** is called, and various things are done to get the game started. In the StartUp subroutine, the title music is played, the graphic scenes showing the windows and buttons are loaded, the sound effects are placed into memory, and the permanent buttons are defined.

We'll be looking at this subroutine in detail later, but if you want to take a quick look now, look at the file called **StartUp.SUB** to see the entire routine. This StartUp only takes place once, at the beginning of the game. After the game has begun, this section of the room file will always call a subroutine called **ReDrawScreen**, which draws the new scenery in the location window, draws any objects in the scroll-bar window, and updates which of the compass buttons should be active and which should be ghosted. This subroutine will also be examined later.

Sound Effects

Turn off sound on entering a new room, if the player can enter from any room with sound effects

After drawing the scenery in the location window, I turn on the sound effects for the ocean and turn off the sound effects for the birds. It should be pointed out that it is important to turn off the birds, even though the sounds may not be running anyway. I needed to consider that the player could have entered this room from several different directions, and if she entered from the sand dunes, the bird sounds would be playing and would need to be stopped.

Also notice that I set the volume for the ocean sounds to 40, rather than full volume to 64. I wanted the ocean to be in the background, so kept the volume down a bit. At the same time, I wanted the death scream at the end of the game to stand out, so I had to keep all the other sounds proportionally softer.

Room Descriptions

The next section of the code block gives one of three possible descriptions of this room. If the player was swimming in the ocean, she is told

that he tires and returns to the beach. Notice that in line 54, the attribute **ForcedReturn** is unset back to N so that the swimming message will not be given when the player next enters the room from a different direction. The subroutine **NoSwim** prints out the actual message.

If the room has not been visited yet, a full description of the room is given. Lines 57-062 show how this is accomplished. A string variable, **$tx**, is set to the line of words that I want printed out on the screen, and then I call a subroutine **print** which actually puts the text on the screen.

The print subroutine also takes care of some mechanical tasks such as changing the color to blue and scrolling the other lines up one line to make room for the new line of print. It checks to see if more than five lines have been recently printed, and if so, pauses while the player reads the text before clicking the mouse button to continue. This print subroutine is a very important subroutine to the game, and will be examined later.

Finally, if the room has already been visited, the game prints out a shorter room description in line 64.

Check for Player Input

The next lines from 68-70 jump to a subroutine from which the game never returns. It is at this point that I wanted the game to start checking for input from the player. The input would either be made by typing on the keyboard or by clicking on the mouse. This input would be accepted, acted upon, and then start checking for input again. I wanted the game to continue in this loop forever, until the game was ended.

A subroutine with no return checks for player input

So the subroutine I was jumping to here, would never return again. Of course this room might be visited many times during the game play, and I never wanted this section of code to execute again, or it would send the player into a never-ending loop. So the subroutine **StartUp2** is only called when the room attribute **started** is unset. And the first thing I did in the subroutine was to set the **started** attribute to make sure it would never be called again. You can see this in the file **StartUp.SUB** in line 119 if you wish to jump ahead.

This completes the end of the code block and the room file for the "west_end_of_beach." As you have seen, this room file is not a typical one, because it contains additional code for getting the game started and properly initialized. When you create your own adventure, it is probably easier to write normal room files and then come back to the starting room and add the extra code for initializing your game later. In this way, you can get the feel for how the game is coming together, by seeing each of the rooms as they are created to fit into your plot.

The actual details of getting the game started can come after you have finished the room files and the objects files. That was what I did, and

it seemed to work smoother that way. However, I have described the initialization here (out of order), since I wanted to give a complete explanation of the room files. To get a clearer picture of room files, let's look at some of the other rooms in the file.

Special-Purpose Rooms

The next room is the special-purpose room "ocean2" This room is similar to the "unused" room which was discussed at the beginning of this chapter. Its only purpose is to force the player back to the beach when he tries to swim in the ocean. In this case, this location forces the player back to the beach by the canoe. This room file is not a typical one, but rather can be viewed as a special-purpose room file.

A Typical Room File

The next room file, "east_end_of_beach" is a good example of a typical room file. It starts with an easy-to-remember, descriptive name. There are some attributes, in this case to see if the player has been forced back to this room from the ocean. Default directions are chosen, allowing the player to travel to other rooms.

The code block contains the necessary information to clear the buttons and redefine them for the current scene. Some objects, like the sand and ocean, are placed at this location, so they can be easily examined. The DIG variable is set to indicate that this is a sandy location where the player can dig without tools. And finally, one of three possible room descriptions is given, depending on whether the player has entered the room from the ocean, has entered for the first time, or has been there before.

One noteworthy point about this room file is the fact that the canoe can be either on the beach or in the water. When the player visits this section of beach by the canoe, the canoe can be pushed into the water. When that happens, an attribute attached to the canoe is set. While the canoe is on the beach, the attribute **InWater** is set to N, but it changes to Y when the player pushes it into the water. Look at lines 107-116 and you will see that the **RoomNumber** and click zones are changed according to where the canoe sits.

If according to the attribute **InWater**, the canoe sits in the water, then location picture 4 is displayed, showing the canoe out in the ocean. The click zones are defined to fit the location of the canoe, which is a bit higher on the screen. If the canoe sits on the beach, however, location picture 2 is displayed where the canoe is on the sand, and the click zones are defined lower on the screen to match the canoe position.

After defining which picture to show, and where the canoe will be located, the subroutine to redraw the screen is called.

Handling Diagonal Click Zones

You might notice that I used two click zones for the canoe, instead of one. The reason is that the canoe runs diagonally. To enclose it in a single large rectangular zone would mean that large triangles in the upper left corner and lower right corner would show up as water, but would respond to clicks like they were the canoe.

To prevent that problem, I chose to use two smaller click zones which encompassed most of the canoe without taking up too much of the water. I used the same principle with the ladder, elsewhere in this game. It also was a long shape that ran diagonally, and required two smaller click zones rather than a single larger one. This is a technique you may find useful in your own adventures, depending on the particular shape of an object.

Since the rest of the room files are similar in construction, I'll skip over most of them, pausing only for those that contain special features that will be of interest. Some rooms have no attributes listed, because none are necessary beyond the automatic ones of VISITED and DARK. The code block of each room file generally starts by clearing the buttons for the click zones, and then re-defining them for that scene. The room number is then set, and the screen is drawn for that number.

Frequently, sounds are either started, stopped, or changed in volume at this point. If entering a particular room doesn't change the sound effects no matter how you enter that room, then sound commands are not necessary.

The sounds are generally followed by objects that are to be placed in the room. These are background objects that the player may want to examine or otherwise refer to. At this point I also define the variable DIG in order to determine whether to allow the player to dig in this room or not.

Handling Multiple-Scene Rooms

Conditions can be tested to determine which view of a room will be displayed

The last part of the code block gives the two variations on the room description. Look at the room file for "by_the_boulder" starting at line 345, and you will see another case where two different scenes can be shown for the same room. If the boulder is moved, scene number 11 is shown, and the click zone for the boulder is adjusted sideways a bit. An additional click zone for the cave opening is added, and the object for the cave opening is added to the room, allowing the player to examine it. If the boulder is still in front of the cave, then scene number 9 is shown, and the single click zone for the boulder is defined.

This is another example of where a conditional statement can help you display different views of a single scene. In your own adventure, you might use a similar routine to show a front porch with the door closed or with the door open. Routines similar to this one can be very powerful, and you will find many uses for them.

When you have finished your room files, you are ready to move on to the next step in writing your adventure. But don't feel you have to complete the whole room file as you see it in my source code. You will probably do as I did, and write the general rough outline first. Then as the need arises, you can come back and add things to your room files. Your room files do not have to be complete before you can go on to the next part of adventure writing. But at least you should get the rooms defined with their default directions and the room descriptions. As the game starts to take form, you can always come back and make modifications.

With the room files completed, it's time to look at the **StartUp.SUB** which was mentioned previously. This is the subroutine that is called from the first room the player enters, in this case "west_end_of_beach." In the next chapter, I'll explain how I created my **StartUp.SUB** file and why each part plays an important part in the game.

Chapter 20: The StartUp.SUB File

There are a variety of things that must be done at the beginning of any game. In *Cannibal*, I had to load some graphics, sound effects, songs, fonts, print some opening text, and define some click zones. To make this source code easier to find and deal with, I put it all in a subroutine called **StartUp.SUB** that would only be called once. This chapter will take a detailed look at this special one-time-only subroutine. It will also look at some of the special features of Visionary that this subroutine affects.

The Text-Only Screen

As you are aware, Visionary has two screen modes. The screen can be in the text or graphics mode. The text mode shows a text-only screen. The graphics mode is the one that I chose for *Cannibal*, so that I could show scenes of each location, and so that the player could click on buttons.

But there are still times when *Cannibal* will switch to the text mode. This is primarily done when the player loads or saves his position in the game. At that point, Visionary automatically switches to the text mode and presents the player with a requester window to choose the device and filename for the saved position.

Even in a graphic adventure, you may need a text screen

Another time the game will use the text screen instead of the graphics screen is when Visionary is forced to pop up any other requester. For example, if Visionary is trying to access a disk named *Cannibal* in order to load a location scene, but you have taken the disk out of the drive, it will switch to the text screen and put a requester box asking you to insert *Cannibal* in any drive.

So there are times that the player will see the text screen, even though the game was designed to use the graphics screen only. And since the player will occasionally see the text screen, I wanted to customize its look.

Text Colors

I wanted to control what the player saw on the text screen. To do that, I wanted to control the four colors of the text screen. These four colors are numbered 0-3 and can be adjusted using the TextPalette command.

Color 0 is used for the background and is normally gray, having RGB (red, green, blue) values of 8,8,8. Color 1 is used for the text the game prints back to the player. It is normally black, with RGB values of

0,0,0. Color 2 is used for text the player types, and is also used in the layering gadgets in the upper right corner of the screen. It is normally white, and has RGB values of 15,15,15. The final color, Color 3, is used for the scrollbar at the top of the text screen and for the text prompt. It is normally red, and has RGB values of 8,0,0.

Scrollbar Handling

Since neither the scrollbar or the prompt were required during my game, I decided to hide them by making them the same color as the gray background. I did this in line 6 of the source code for **Start-Up.SUB**.

The next two code lines, 8 and 9, also control the visuals on the text screen. The main purpose of the command **scrollbar off** is to prevent the player from grabbing the scrollbar with the mouse and dragging it downward to see the screen behind the text screen. But there are two additional features as well. When Visionary turns off the scrollbar, it also eliminates the layering gadgets in the upper right corner and eliminates the printout of the room name in the upper left corner.

If I had not turned off the scrollbar, my attempts to hide it by changing its color to the background gray would have been thwarted. Although the bar would have blended into the background, the white of the layering gadgets would have still appeared, and the black title of the room would have appeared on the left. By turning off the scrollbar, I eliminated all three problems at once.

Visionary automatically creates text screen menus

The other command, **menus off**, serves the purpose of defeating the default pull-down menus Visionary provides to allow the player to quit, load, and save a game, or start a new game. I wanted all these options in my game to be controlled from the graphic screen, not by pull-down windows from the text screen. However, please note that both of these are very powerful features of Visionary, and you may wish to leave them enabled in your game, to save you a lot of extra programming.

Loading the Game

I knew that when my game was finished, I would want to put it on a disk that would boot in drive df0: of an Amiga. I planned on having a title screen show while the game loaded, and wanted some music to play in the background while the title showed and the rest of the game loaded. To achieve that goal, I loaded the music and played it as the next step in the loading sequence. Lines 11-13 show how easily this is accomplished.

Erik Hermansen did the music for *Cannibal* as well as all the graphics. Once I had his MED module containing the song, I only had to load the song into buffer 0, switch Visionary from digitized sounds to MED songs, and play the song in buffer 0.

Visionary's default condition is set up to play digitized sound samples when the game begins. Since it can't play both MED modules and digitized samples at the same time, I had to use the **EnableMusic** command to switch it to playing MED music.

Also notice that although the music was simply named **title.mus** I had to specify the path name for the file. I planned on placing all my audio effects, both digitized sounds as well as MED modules, in one directory called Audio. That directory name had to be included in the file name. I also specified the disk by name, so that if the player had booted to Workbench first, and then started the game by clicking on an icon, the game would still be able to find the music. For these reasons, as well as others which will be explained later, all files loaded by Visionary were given very explicit path names.

Using the Ram Disk

Lines 15-20 show a special routine that I wrote to speed up the game play where possible. One thing that could slow a game down is the time it takes the computer to load location scenes from a disk drive. For example, when the player moves from the beach to the sand dunes, the computer must load the scene of the sand dunes from the disk drive into the computer before it can display it on the screen.

Your game can check for enough RAM space to pre-load graphics and sounds

If the player had installed my game on a hard drive, this time would have been negligible. However, I realized that many players of the game would not be using a hard drive. The next-best thing would be to store all the scenes in RAM, so they would load instantly and would not require any slow disk access. That's what this special routine does. It first checks to see if there is room in RAM, by accessing the FastMem variable. If there is at least 200K of fast RAM available, then all the location scenes are loaded into RAM before the game begins.

To make sure that the game knows the location scenes are currently available in RAM, I set a variable **$DEVICE** to the proper pathname which would be added to the front of any location scene filename. If there was enough room in RAM, the variable was set to "RAM:" and all the screens were loaded into RAM. Otherwise, the variable was set to "Cannibal:Video/", which refers to the disk and directory where the scenes are to be found.

There were two other parts to making this technique work. I had to be sure I used this $DEVICE variable when loading the scenery files, and I also had to be sure to recheck this variable after any game position had been saved. The loading of scenery files takes place in line 308 of the file called **Cannibal.SUB**. I'll be describing this file in detail later.

The Saved Game Handler

The saved game feature of Visionary was the other place where I needed to check the value of the $DEVICE variable. The values of all

variables are saved when the player position is saved, and that includes the value of $DEVICE.

While normally this will cause no problem, there are special cases where this could cause a serious problem. For example, if the player is using a machine that has plenty of free memory, the scenery files are loaded into RAM and $DEVICE tells the game that the scenes are in RAM. If the player's place is saved, then loaded back in again, the game will think the scenes are in RAM. And usually they will be.

But the player may come back to the game while multi-tasking some other programs, so there is no longer enough free RAM to hold the scenes. When the game loads, scenes are not placed in RAM, and $DEVICE is set to "Cannibal:Video/". After the player's saved position has been loaded, $DEVICE will be set to "RAM:" as it was when the position was originally saved. When the game tried to fetch the scenery file from RAM: it would not find it, and the game would then crash.

This type of problem could also occur if the player started the game on a computer with lots of free RAM, then saved it and took it elsewhere to play on a system with minimal free RAM. Again, when loading the saved position, the game would expect to find scenes in RAM when they were not, and the game would crash.

To keep this problem from happening, a check needs to be made immediately after a saved game position is loaded. You will find such a check in the **MainLoop.SUB** file, in lines 174-179. I'll be describing this check routine in detail later, but for now just notice that I attempt to load the current location scene from RAM, and then check the **ERROR** variable. If it is set, I know the files are not in RAM, and set $DEVICE accordingly. Likewise, if no error is detected, I know the scenes are safe in RAM and properly adjust the value of the $DEVICE variable. It takes a bit of extra effort to place the scenery location files in RAM and use them properly, but the game runs much faster that way, and the effort is well worthwhile.

Loading Graphics and Sounds

The next parts of the **StartUp.SUB** file involve loading graphics and sounds. First I load the current scenery file, **Loc1**, into screen buffer 1. Screen buffer 1 is where I decided to load all the location scenes. I wanted to refer to each of them by number, so that a single routine could load any scene just by changing the value of a variable.

Notice that each file is loaded with three lines of code. The first sets the "**$filename**" variable. The second line loads the filename into the correct buffer. The third line calls a subroutine to check for any loading error.

The LoadingError Subroutine

Special error-handling routines will make your game more user-friendly

Look at the end of the **StartUp.SUB** file at line 127 for the subroutine that checks for errors. In general, it checks the **ERROR** variable and if it finds no error, simply returns to the loading of files. On the other hand, if it finds an error has occurred, it prints an error message on the text screen, closes the graphic title screen and shows the text screen, and then waits for the player to press **[Return]** before quitting.

To specifically accomplish this, the routine first switches to Pen 2 by typing on the text screen the commands in line 130, changes the color of Pen 2 to white in line 131, clears the screen of all other text in line 132, moves the cursor down to the center of the screen and prints the error message in line 142, and finally changes back to Pen 1. All of these commands are done with "T" lines, the normal text output command for the text screen.

Then the title screen is closed, using the DOS command to call up the **CloseScreen** utility program that comes with Visionary, and the text screen is displayed with the **ScreenMode text** command. Since I previously chose to disable the scrollbar and change the color of Pen 3 to the background color, the only thing the player will see at this point will be a totally gray screen with a white message displayed in the center. The game then waits for the player to press [Return] in line 146 before aborting in line 147.

Buttons

After checking for a loading error, line 26 of this routine then loads a file called **buttons.pic** into screen buffer 2. This screen will never be seen in its entirety; rather pieces of it will be used as required. It contains pictures of all the buttons that can be pressed in the game. It contains two of each action button, one as it looked pressed inward and the other as it looks normally. It also contains three versions of the compass buttons, one normal, one pushed in, and one ghosted.

All the objects that will show up in the location window and the inventory window appear in this screen, and will eventually be copied from here onto the visible screen. It also contains a large picture of the ladder and rectangular area where graphic overlays will take place.

Sounds and Music

The next four files loaded are all sounds. The sounds of bubbling water is placed in sound buffer 0. This is the sounds that the player hears when he is captured by the cannibals and thrown into a large iron pot of boiling water. The ocean sounds are loaded into sound buffer 1. These play constantly in the background when the player is at the beach or nearby. The sounds of birds chirping in a meadow are loaded into sound buffer 2. These will be played at the meadow and surround-

ing locations. The sound of water dripping is loaded into sound buffer 3. This sound will be heard in the cave.

There is one other sound, a man screaming, which is not loaded into memory yet. This sound is used only when the player loses, and I decided to conserve valuable Chip RAM by not loading it at this time. It will be loaded and played once, only if the game is lost. The fact that there will be a delay while this sound file is loaded from disk is negligible, since the sound of bubbling water will be heard while the file is loading.

The Game Screen

After the sounds, the graphic file **window.pic** is loaded into screen buffer 0. This is the screen the player will see. It contains the text window, the location window, the inventory window and the compass and action buttons. I designed this screen to stay in front, always before the player's eyes, while text is printed to it and scenery is copied to it. All the other graphic screens will be forever hidden, only used to help in the display of the game screen in screen buffer 0.

The next lines take care of loading a specially designed font, the **Artesian** font. The first three lines look similar to the other file-loading lines discussed above, in that they set the filename, load the file, and check for an error. Since Visionary is capable of using any size font on the graphic screen, it is necessary to specify not only the name of the font, but the size as well. In this case, the font loaded into font buffer 0 is size 8. The fourth and final line of this routine tells the game to use the font in font buffer 0 on screen 0, the screen that the player will see.

I designed the **Artesian** font for use with the low-res screen of 320x200 pixels (or 320x256 in PAL). Remember that the reason for choosing low-res was to allow more colors while at the same time taking up less Chip RAM. An unfortunate side effect of this choice was that normal fonts looked too fat. They took up so much space, there was room for only 30 characters in the text window. I solved the problem by designing my own font to be thinner, allowing 40 characters across the text window.

If you also find the need to design your own font, the simplest way is to use the font editor found on the Extras disk that came with your original 1.3 Workbench disk. It will be found in the tools drawer under the name **FED**, which is short for "font editor." A variety of other commercial and freely distributable font editors, all of which will allow you to easily design your own fonts, are also available.

After loading the font, I was ready to print out some text in the text window. To do this, I started by setting the color that the text would appear in. In line 55, I set the color of any text printed to screen 0 to blue.

For easy reference, define color numbers as variables with the same name as the color

Visionary takes the colors from the palette of the currently-displayed screen. In this case, it means that the colors of the palette for screen 0, the file called **window.pic** would be chosen. I could have just as easily have set the color to 13, which is the palette number for blue. But it might be hard to remember at a later date what color 13 was. So I defined a variable in the .ADV file called "blue" and assigned the value of 13 to the variable. In that way, when I wanted to refer to the color blue, I could say "blue" instead of "13". At the same time, I also defined some other variables for white, red, green, and brown. All of these were colors that I knew I would be referring to later, and defining them now would save me time, effort, and confusion later.

As mentioned previously, I created a subroutine to type my text on the graphic screen. It would take care of scrolling the five lines of text above it upward and printing the currently desired text on the sixth line. Another subroutine called **Linefeed** would simply move the text upward and print a blank line. These were the routines I used to print out the opening messages of text in the text window. As you can see in lines 56-63, I simply set the value of my text string and call the print subroutine. This print subroutine is a very important part of the game, and will be examined in great detail in a later chapter.

Hidden Buffers

After printing the copyright notice and the title in the text window, the next thing was to create several hidden graphics buffers. To quickly recap the Visionary manual, there are three types of copies that can be done with graphics: draw, overlay, and XOr.

Copying in the overlay mode requires a "mask" buffer, where the object to be overlaid can be temporarily placed while a mask is created. In this way, only the object to be overlaid will appear, and not its background. Only one mask buffer is required, so I chose to make it screen number 24.

A mask buffer is needed for overlay copy mode

To conserve Chip RAM, I created the screen to be as small as possible and still hold any object that would need to be overlaid. Since the large picture of the ladder was the largest overlay copy that would be required, I used coordinates of 55 pixels by 78 pixels. Notice this is done in line 65, and I also had to specify that there five bit planes in the graphics (meaning 32 colors, which is 2 to the 5th power) and that the screen was in lo-res (meaning 320x200 NTSC or 320x256 PAL). Then after creating the screen, I had to tell Visionary that this was the one I wanted used as the mask buffer.

Not all graphic screens are loaded from disk. The mask buffer is one example. I created it in memory from scratch. My scrollbar (to hold the objects that appear in the location window) is another example. It was also created in memory. In lines 68-71 you will see that I created another screen numbered 23 for my scrollbar. Since I wanted it to be

large enough to contain 20 objects, each 15x17 pixels with one pixel in between, I created screen 23 to be long and thin.

Screens can be larger than the normal display limits of the monitor

It is quite possible to create screens larger than the player can see on the normal screen, and that's exactly what I did in creating screen 23 to be 15 pixels across and 361 pixels down. Then I filled the entire screen with white, by setting the mode to "draw" for that screen, setting the color for screen 23 to white, and drawing a rectangle from the upper left corner to the lower right corner. The scrollbar screen was now ready for use. We'll see exactly how it is used when we examine the **GetDrop.SUB** in a later chapter.

Ready to Play

Although it has taken a long time to explain all this, the computer has executed the code and loaded all the files, printed out the text, and created the screens in about 30 seconds. If the game is being played from a hard drive, it would take less than five seconds. And it is now that the game is ready to present itself to the player's eyes.

In the next five lines, I tell Visionary that I want screen 0 (the previously loaded **window.pic** file) to be the one shown. Then I set the screenmode to graphics, which forces the screen to the front where it can be seen. The title picture which has been showing all this time, is now forced to the back and is hidden. Next, I turn off the MED music that has been playing with the DisableMusic command, and start up the sound effects in sound buffer 1. The second zero tells the game to keep playing the sound effects in a continuous loop, so until the player visits another room and another sound command is received, the ocean waves will continue to crash.

Then I close the title screen which has been hidden behind screen 0, in order to free some Chip RAM. This is done by using the versatile DOS command and the small **CloseScreen** utility program which comes with Visionary.

Setting the Scene

The next lines of code print some additional text in the text window, which has been waiting for the player to press the mouse button to continue. These six lines of text are in the standard form of a text string followed by a call to my print subroutine. Their purpose is to set up the plot of the game, give players a quick background on who they are and what they is trying to do.

Following this section, fifteen permanent click zones are defined. I used low numbers for the buttons, 0-15, so that they would have top priority over all other button clicks. When the mouse is pointed inside the click zone coordinates indicated and the button is clicked, the named subroutine is called. Notice I chose simple names for that sub-

routines that would help me remember their purpose. I only varied from simplicity in button 7, and used the name DIG2 since there was already a subroutine named DIG when I got around to programming this button.

The last thing I did in this subroutine was to place an object called **"00nothing"** in the room, and have it placed in the player's inventory. We'll learn more about this object when we examine the object files in a future chapter. For now, it will be sufficient to say that "00nothing" is an invisible object that the player always carries. The double-zero at the beginning of the name forces it to be the first object in the list of objects.

The StartUp2 Subroutine

At this point, the game has basically started. The program jumps back to the "west_end_of_beach", where the subroutine was originally called from. At this point the room description is given. Since the room has not been visited before—specifically, the VISITED attribute is not set—the full room description is given. Then the program is told to jump to the next subroutine in the **StartUp.SUB** file, the one called **StartUp2**.

The subroutine StartUp2 is short but important. It first sets the attribute "started" for the "west_end_of_beach" where the game starts. With this attribute set, the program will not jump to the StartUp section of the code again whenever the player visits "west_end_of_beach", locking the program up into an endless loop.

Next, this subroutine sets the VISITED attribute for the room. This attribute is set automatically when the player leaves a room in the normal way. But in this case, the player has not left the room normally; the ENDROOM statement has not yet been executed. So in this one special case, the VISITED attribute must be set manually by my program. If I hadn't done that, the next time the player entered the room, the full room description would have been printed instead of the shortened one.

Calling MainLoop and LoadingError

The third and last thing the subroutine does is to call the subroutine named **MainLoop**. In this routine, the game will check to see if the player has typed something or has pressed the mouse button, and if he has, it will try to do whatever the player has requested. Then it will loop back and check again. This will continue until the game ends with the player winning, losing, or quitting. The **MainLoop** subroutine is the heart of the adventure, and as such deserves a chapter all of its own. We will be looking at it after the next chapter.

The third and last subroutine in this file is the **LoadingError** subroutine, which was examined previously and needs no further explanation. This completes the explanation of the **StartUp.SUB** file.

With the **StartUp.SUB** file completed, the next step is to write the object files. Without these object files, you can still wander around in your game, but nothing can be manipulated, not even examined. In the next chapter I'll cover the different types of object files that I used in *Cannibal*, and how they were constructed.

Chapter 21: The NonMovable.OBJ and NPC.OBJ Files

There are three basic types of object files, nonmovable objects, movable objects, and non-player-character objects. When I had so many movable objects in my game that the file became too large, I split the file into two parts. So my source code for objects is actually in four files, not three. This chapter will look at all four files, and see how objects are created. We will also look at specialized routines for dealing with specific objects.

When Visionary compiles the source code for a game, it alphabetizes the objects by name. In that way, its internal search for objects is optimized. What I have done in *Cannibal* is to take advantage of this fact to force many of the objects to be listed in a certain order of my choice. In that way, when I know exactly where the object is in the list of objects, I can not only refer to it by name, but I can also refer to it by number.

Object names that begin with numbers will appear at the start of the alphabetized object list

Being able to refer to an object by number allows me to use a powerful technique where many routines can refer to objects by a single variable name. To make that routine act on any object I wish, I only need to set the value of the variable to match the object, and then call the routine. We'll be looking at some of these routines in the chapter on the **GetDrop.SUB**, but for now, this will explain why some of object names start with a two digit number. By starting them with such a number, they will be placed at the start of the list of objects, when Visionary alphabetizes the object list. Hence, I know that "00nothing" will be the first object on the list, and "01ladder" will be immediately after it.

The NonMovable.OBJ File

Keeping this special trick in mind, let's begin by looking at the file called **NonMovable.OBJ**. It contains all the objects that are normally considered part of the background. These objects can not usually be manipulated in more than the most rudimentary ways. Usually, they can be examined, and that's about all. Normally speaking, they can not be picked up, eaten, broken, or burned.

There is no reason that these types of objects must be kept separate from the other types. Visionary places no restrictions on which objects must be in which file. The restriction is one of my own making, to allow me to keep my files structures a bit better. It also makes finding things easier. It was my decision to place all these background objects together in a single file called **NonMovable.OBJ**.

The "00nothing" Object

When you look at the **NonMovable.OBJ** file, you will see that the first object is named "00nothing." There is very little to this object, and as you can see, the object file is very small. Its purpose is probably more valuable to a text adventure that it is to this graphic game. I originally included it so that made the look of an inventory listing appear nicer.

Visionary's built-in **INVENTORY** command will list out all the objects that the player carries, but it does no more. I wanted the inventory list to start by saying, "You are carrying:" and then list the objects. In the event that the player carried no objects, however, I wanted the game to respond, "You carry nothing." Visionary's built-in "inventory" command does neither of these two things. It simply types a list of the objects carried, and in the event that the player carries no objects, it doesn't do anything. By adding the "00nothing" object, I was able to customize the look of the "inventory" command.

Let me describe how the "00nothing" object can help customize the INVENTORY command in your Visionary game. Notice in the source code for **NonMovable.OBJ** at lines 10-11 the code block is empty. That's because in my game, I decided to use an inventory window on the graphic screen to show the inventory. But let me describe how you could add a few simple lines to the code block to customize your "inventory" command in a text game. Look at the sample code below:

```
CODE
IF ITEMS > 1 THEN
 T You are carrying:
ELSE
 T You carry nothing.
ENDIF
ENDCODE
```

At the beginning of the game, I make sure that the player carries the object "00nothing." The player is never allowed to drop the object, and actually is never even aware it is in the inventory. By giving the object the name "00nothing," when Visionary compiles your source code, alphabetizing forces this object to be at the top of the list. Thus if the player does not carry more than one item, the one object the player carries must be the "00nothing" object, and the game types "You carry nothing." If the player carries more than one item, the line "You are carrying:" is typed before the list of objects. Regardless of the situation, the inventory list looks a bit nicer.

Even though I chose not to use the standard Visionary inventory method, substituting a visual method on the graphics screen, I still left in the "00nothing" object so that the other objects that I intended to number would start with 01 rather than 00. I did not need to number all the objects in the game. Rather, I wanted to refer to all the movable objectsObjects;by number by number. So I left "00nothing" and made the other movable objects numbers 01 through 19. In that way, I

could refer to the movable objects either by name, or by number. I could make a loop which could search for any of the nineteen objects or I could use a variable to specify one particular object depending on how I set the value of the variable. Therefore, the object "00nothing" played a small but important part in my adventure.

Typical Nonmovable Object Files

The file starting at line 17, which defines the object "sand", is a more typical object file for a nonmovable object. The file starts with the name of the object, which the game and the programmer will use. Following this is a list of names which the player can use, which are all **synonyms**. This is followed by the room where the game should initially place the object.

Since I planned on moving the "sand" object from room to room, as the player entered any room containing sand, the initial room was irrelevant. I could just as easily have placed it in one of the beach locations.

The code block is empty, since I didn't want the game to print anything special when the player saw the room description. I did place a comment in the code block reminding me that an addition reason that this object existed was to permit the player to type "put the ladder on the ground". If such a line is typed by the player, Visionary expects both objects "ladder" and "ground" to be present, or it will respond with an error message. Creating "sand" and the synonym "ground" allowed such a command from the player. And even though later as the game was being designed, I decided that the ladder would always be either on the ground or against an object, and that the player would have no choice in the matter, I left the object in the source code so that the other actions could still be performed on it.

Actions

After the code block, three actions are listed. You should always allow the player to examine the object, even if it is part of the background. In this case, I allowed the player to use three different words with the same meaning, "look," "search" and "examine."

Expect the player to try to EXAMINE and GET each object—other typical actions will depend on the object

Notice that if the player tried to examine the beach sand, I didn't give much of a description. You don't have to either. But always give some description, no matter how small. Even a response like "You see nothing special here" or "Looks like beach sand to me" is better than no response at all. I then anticipated that the player might try to get some of the sand, so wrote an action block that tells the player he can't get any sand.

The third action that I anticipated the player would try would be to dig in the sand. I created a final action block to take care of this situation. I realized that the player might type "dig in the sand" or simply "dig." The first case would be caught by this section of code. The second would have to be caught by the vocabulary action file, a special file that we will be examining in a later chapter. To keep from typing the exact same lines of source code both here and in the vocabulary action file, I made a DIG subroutine and called it from both places. And that completes a look at the typical nonmovable object file.

Sometimes I didn't need to use even three simple actions. Other times, I found the need to add extra actions. Look, for example, at lines 138-173 which define the object "ocean." In addition to the standard actions of **look** and **get**, I anticipated the player might try to **drink** from the ocean or **swim** in the ocean. I gave a short message if the player tries to drink the sea water. I had already anticipated that the player might try to swim out to sea, and had taken care of that situation if the player clicked on the **North** button. To disallow doing the same thing by typing "swim in the ocean," I added the **swim** action and called the same subroutine as would be called if the player clicked **North** on the compass.

To force player actions, you can define object synonyms that are not synonymous in English

Since I wanted my game to permit text input as well as mouse input, I occasionally used synonyms in my action blocks that really did not have the same meaning. For an example of this, look at the "hole" object in lines 231-245. This is the hole that is in the cave, the one that lead into the rock room. I made **look**, **examine** and **go** all synonymous for the hole. Obviously "look at the hole" and "go into the hole" have two different meanings. But since I wanted to give the same response to both commands, and deny the player the ability to enter the hole, I listed them all in the same action block as synonyms. It's a good trick to remember, when you have dissimilar words that you want to receive the same response. It can save you some programming steps, and if used consistently, can make your final game much smaller.

Define two objects for something the player sees as one object, to allow a change its actions or attributes in the game's context

Another special case occurs in lines 408-431 where you will see an object called "ladder1." I found the need for two different ladders in my game. Actually, there was only a single ladder, but I needed two different objects for it. One object would be the ladder as normally seen. This would be the one that the player would see leaning against the shack. This would be the movable object that the player could pick up and move into his inventory. But I needed to create a second object for when the player was on the roof of the shack and could see the ladder leading down to the meadow. I didn't want this to be the same object which could be picked up. Not when the player was on the shack.

So I created a second object called "ladder1" which could be put on the shack roof, in the boughs of the tree, and on the top of the boulder. This object would appear to the player to be the same object, but this

one could not be taken. Notice that when examined, the same description as the other ladder is given. But when the player types "get the ladder" he is told to leave it alone. This technique of using two objects for what the player sees as only one object can be used in a variety of ways. Keep it in mind when you create your own games.

Moving NonMovable Objects

Most nonmovable objects have no attributes. Let's look at several different objects that do require attributes, and see the different ways that attributes can be used with objects. The first object file we will look at is for the boulder in lines 460-522. The boulder starts out in front of the cave, and during the game is rolled aside to permit the player to enter the cave. Since I wanted to know the status of the boulder in order to give the proper description, among other things, I created an attribute for the boulder called **moved**.

When the game starts, the boulder has not been moved, so the attribute starts at N. Any time the player tries to examine the boulder, either by clicking on the picture of the boulder or by typing "look at the boulder" one of two descriptions can be given, depending on the status of the **moved** attribute. As you can see in lines 743-487, if the boulder is moved the description is different that if it is not. All of this is made easy by creating an attribute and checking it before printing out the description.

This status of this attribute must be changed when the boulder is moved out of the way. When I designed the game, I decided that the player would not be able to move the boulder away from the cave entrance using his normal strength. He first had to eat the candy bar to gain some temporary extra energy which to move the boulder. Let's look at the source code from lines 494-515 and see the wide variety of things that have to be done when the player moves the boulder.

First of all, I check to see if the player is strong enough in line 495. If not, the game jumps down to the bottom of the action block and with the message "the boulder shifts slightly but rolls back." The subtle point of this message is that it tells the player that moving the boulder is not impossible, but can't be accomplished just yet. In this way, the player won't give up the idea as being something the game won't allow, but will try to figure out some other way to move it.

In fact, after eating the candy bar, the player will be given a burst of energy that lasts only four moves, enough to allow the boulder to be rolled away. When the candy bar is eaten, the variable **energy** is set to 4. Each time the player makes a move, the **energy** variable is decremented by one as part of the NPC.OBJ file. We'll be looking at that file later. But line 495 checks to see if the player has any extra energy before permitting the boulder to move.

If the player is strong enough, a message is printed telling the player that the boulder rolls over. Then the attribute **moved** is set to **Y** so that the game will always remember the boulder has been moved. The next thing to do is to clear the **energy** variable to prevent a message from being displayed when the energy runs out before the boulder is moved. You'll see this message in the **NPC.OBJ** file later.

Now that the boulder has been moved, the game needs to change the directions to allow the player to move east into the cave. Normally, this would be a simple two line command, consisting of a DIRECTIONS command followed by a LINK command. The possibility that the ladder may be presently leaning against the boulder complicates this slightly. As you can see in lines 500-504, the directions leading from this location depend on whether the ladder, object "01ladder", is in the room.

If the ladder is in the room with the boulder, there are three directions that the player can go: **North** to the meadow, **East** into the cave, or **Up** to the top of the boulder. If the ladder is not present, there are only two directions the player can go, **North** and **East**. After setting the directions, the rooms must be linked. After all, it makes no sense to tell the computer that the player can go east, unless you also tell it where the player will be, when he goes east. Line 505 links the inside of the cave with the outside of the cave.

The next thing to do is to actually show the boulder rolled aside. Since I was creating a graphics game, I didn't want to just tell the player that the boulder had moved—I needed to **show** it had moved. To do this required several steps. First I had to change the picture of the location to the one showing the boulder moved out of the way. I numbered this as picture 11, and set that value in line 506.

Then I needed to adjust some click zones. Since the boulder was no longer in the same position on the screen, I redefined button 34 to the new position. Since the opening to the cave was now visible, I defined a new click zone for button 35 to be the cave opening.

Next I placed the object "cave" in the room, so that when the player typed "look in the cave", Visionary would be able to give a response. If I had not placed the "cave" object in this room, asking to "look into the cave" would have generated the response "I don't see a cave here." And finally, since everything was now in readiness, I called my subroutine to redraw the screen showing the boulder rolled away, and show the compass button for east highlighted.

Changing the position of a nonmovable object, like the boulder, can be an important part of any adventure. Just don't overlook any important steps in doing so. As you have seen from this one example, you must first check to see if the changing of the position is allowed. Then you must print a message to the player, set an attribute, change the directions affected by the change, redefine graphic picture numbers and

click zones, move necessary objects to the current room, and finally show the new graphic scene.

A similar routine is used in my game when the player pushes the canoe into the water. Let's take a look at how it is accomplished, and you will see many similarities to the previous routine.

In lines 676-783 you will find the source code for the "canoe" object. Just as the boulder had an attribute called **moved**, the canoe has an attribute called **InWater**. The purpose is basically the same, in that it keeps track of the status of the canoe. When the game starts, the canoe is not in the water, so the attribute is set to N. Notice the standard action blocks for examining and taking the canoe. The third action block, however, has some unique features that deserve special mention.

Look at the action block for **get** and **sit** starting in line 701. If the player says "get in the canoe" or "sit in the canoe", it is clear that the player wants to sit down in the canoe. If, however, he leaves out the word "in" and types "get the canoe" the command takes on an entirely different meaning. This action block shows how to deal with the double meaning of **get**. Notice that I checked for the presence of three different prepositions "in," "into" and "inside." If any of these were present, then I assumed that the player wanted to actually sit down in the canoe, and called a subroutine to tell the player it is a comfortable fit. If on the other hand, none of those three prepositions were used, then I assumed the player was trying to pick up the canoe.

The next action block at line 714 shows how the canoe can be moved. Unlike the boulder which could be rolled over but not back, the canoe can be pushed into the water and pulled back on shore again. Since verbs like "move" and "slide" can be used synonymously for both "push" and "pull" I decided to use a single action block for both actions. That's why you will see that I made "slide," "move," "push," "pull" and "put" all part of the same action. To determine which the player wanted to do, I only had to check the status of the "InWater" attribute. If the canoe was already in the water, I pulled it out. If the canoe was out of the water, I pushed it in. In each case, I set the RoomNumber variable so that the proper graphic scene would be shown, redefined the click zones for the new position of the canoe (notice I used two smaller click zones, rather that one large one - as explained previously), called the subroutine to redraw the screen, and either set the "InWater" attribute to Y or unset it to N.

The action block for moving the canoe into the water is similar to the action block for moving the boulder away from the cave entrance. In each case, an attribute is used. In each case, a different graphic scene is shown, depending on the status of the object. And in each case, click zones are redefined, and the screen is redrawn. But also notice some differences. When the boulder is moved, the exits are affected. Since the player can suddenly go in a new direction, the directions have

to be redefined and the new room linked with the old one. This isn't necessary when moving the canoe, since the exits stay the same.

To win the game, the player has to row the boat to safety. The action block starting at line 732 shows exactly how this is done. If the player asks to "row the canoe" or "paddle the canoe," the game checks to see if the canoe is in the water and the player has the shovel to row with. If they are, then the game concludes with the player winning. If they are not, then the game responds either that there are no oars or that the canoe is beached on the sand. Let's look more closely at what happens when the player wins.

When the Game Is Won

Starting in line 735, you can see all the things that the game does when the player wins. The first thing is to clear the timer. The timer is a variable that increments on every move the player makes. When it reaches 90, the cannibals arrive on the island and the player is given a warning message. After that additional warnings are given periodically until the player finally is captured and loses after a total of 103 moves on the timer.

If I had not cleared the timer, some embarrassing things could happen. For example, the player could win, and then be told that the cannibals have arrived. Or even worse, he could win right as the timer hit 103, be told he won, only to be then told that he lost. Clearing the timer keeps such things from happening.

Five lines of text are then printed to the text window telling the player that he has escaped from the island and won. While the player reads the message, the final scene showing the player safely being carried away from the island is displayed on the screen. A MED song is loaded from disk into song buffer 1, the sound effects are turned off and MED is enabled in line 749, then played in the following line. At this point, my print subroutine will automatically stop scrolling the text and ask the player to press the mouse button to continue. We'll be looking at the print routine later, so won't go into the mechanics of it now.

While the final fanfare music plays and the player presses the mouse button, the last four lines of text are displayed in the text window. At this point, the variable **CountLines** is set to 1 to keep the message "Press Mouse Button to Continue" from appearing. This is a feature built into the print subroutine as we will see later. Then I change the text color to brown and print out the message telling the player to click the mouse button to exit the game.

The variable **TextColor** is one that I defined in the .ADV file and used in my print subroutine so that I could change the color the text printed in the text window easily. Likewise, "brown" was a variable defined in the .ADV file so that I could refer to the color by name rather than by

palette number. By defining variables like these in your games, you will find it much easier to change features in your game without having to remember specific numbers.

Lines 764-767 show you one of several ways you can check for a mouse click. Using the **LeftButton** command of Visionary, you can check the status of the button. As you can see, I put it in a **while loop** which would wait until the status of the button changed. First I used a while loop to make sure the mouse button had been released from any previous presses, just in case the player had pressed the mouse button earlier and had not released it yet. Then I used a second while loop to wait until the player pressed the button down.

At that point, I set the MainLoop variable to 1. Remember that the concept of the MainLoop was discussed previously, although we have not examined the subroutine in detail yet. Basically, the loop goes around forever checking for player input and acting on it. The game stays in this loop while the value of the variable MainLoop is zero. By setting the value to 1 at this point, I am telling the game to exit the main loop and stop. When we later examine the MainLoop.SUB file, you will see that the game does a few other clean-up activities upon exit before actually returning control to the computer.

Non-Winning Actions

Now that you have seen now the game is won, let's look at the last action block for the canoe. This action starting in line 779, anticipates that the player will try to burn the canoe with the matches. It jumps to a standard subroutine to tell the player that the wet matches won't burn anything.

One of the cardinal rules of adventure writing is that if you are going to create an object, you must allow the player to use it in all normally expected ways. Since I created some matches, I had to expect the player would try to use them. And that meant that I had to anticipate attempts to burn things. So you will find an action block similar to this in many objects. If it was logical to expect the player to attempt to burn something, I inserted this action block. I didn't insert it on all objects, since I didn't expect the player to try to burn the sand, the water or the battery. But I inserted the action into any object file that could logically be burned, like the paper, the candle and the driftwood.

The NPC.OBJ File

Now that we've taken a fairly complete look at the **NonMovable.OBJ** file, let's take a quick look at the **NPC.OBJ** file. We've referred to it several times in this chapter as being the place where the timer checks to see if the cannibals have arrived, as well as being the place where the energy from the candy bar is slowly reduced. It's a very small file, and can be quickly explained.

The non-player character file, called the **NPC.OBJ** file, is different from other object files in that it always executes after every turn. So when you want things to happen after every turn, this is where they should be placed. If you have a second character in your adventure, other than the player, this is where you would have it move. If, for example, you were going to have a pirate appear and follow the player around until he was given a bottle of rum, the NPC file is where you would place the code to create the pirate, place him in the player's current room, and make him beg for rum. Since this is a character in the adventure who is not the player, it is called a non-player character, and the actions governing it are placed in this special object file that will always be executed after every turn the player makes.

You can have as many NPC objects as you want, just as you can with other objects. But notice in line 4 of the **NPC.OBJ** file that instead of calling them objects, they are referred to as NPC. For *Cannibal*, I chose to have only one NPC object, and called it "status." As with normal objects, you must specify name and initroom. The code block is usually the main part of an NPC file, but in my game there are no action blocks required. The code block I wrote had two functions: one was to increment the timer and check to see if the cannibals had arrived on the island, the other was to check to see if the player had eaten the candy bar, and if so to reduce the energy level for each turn taken.

Originally, I placed the routines for incrementing the timer and checking its value here in the NPC. However, I wanted the timer to be incremented when the player moved an object from the location to his inventory, or back. The picking up and dropping of objects was programmed to by-pass the regular moves, meaning the NPC file was not executed and the timer was not incremented for **get** and **drop** actions. To solve the problem, I made the timer section into a subroutine called **CannibalsArrive** and called it from both the NPC as well as the **get** and **drop** parts of my source code. That way the timer was incremented and checked regardless of whether the player was making a normal move, or picking up or dropping objects. We'll be looking at the timer routine in the chapter on subroutines, so I won't go into the details here.

Let's look at the second part of the NPC file. I only wanted to decrement the **energy** variable after the player had eaten the candy bar and the variable had been set to 4. When the game starts, the value is initialized to zero. I use a simple conditional statement that checks to see if the energy is greater than zero. If it is, then I know the player has eaten the candy bar, and I decrement the variable. Once it reaches zero, I print a message to the player telling him that the quick burst of energy has passed, and then never decrement the variable again. Remember that when the player successfully moves the boulder, the

variable **energy** is cleared to prevent the message from being printed after the boulder has been moved.

That concludes our look at the **NPC.OBJ** file and the **NonMovable.OBJ** file. Both are constructed from simple concepts which will help build your adventure. In the next chapter, we will examine the other objects in *Cannibal*, the movable ones. Because these are generally manipulated more than the nonmovable ones, they will take a bit more explanation.

Chapter 22: The Movable.OBJ Files

There are two files of movable objects used in *Cannibal*. This chapter will look at them both, and examine some of the ways that I used objects in my game. I'll also be pointing out special routines that you can modify for use in your own games.

The Numbered Object Names

One of the first things you will notice about these movable objects is that I have given each object name a two-digit prefix. This is done so each of the movable objects will be forced to the front of the list of objects when the game is compiled. I used double digits, including leading zeros where necessary, to ensure that the Visionary compiler alphabetized the objects in the order that I intended.

Visionary commands allow objects to be referred to by number as well as by name. This was originally intended to make it easier to create loops to easily permit player commands like "drop all" or "get all." It was not intended to permit the player to select one specific object by number, due to the fact that the alphabetizing of the object names could make an object's number change as other objects were added or deleted from the game. What I did was to take advantage of the fact that all object names are alphabetized, by creating names that would force the objects to be certain numbers. In Visionary, object numbers start with zero, not one, so I created the nonmovable object named "00nothing" to fill that spot.

The reason that I wanted to be able to refer to objects by number mainly had to do with getting and dropping the objects. Other ways of manipulating the objects, such as examining, reading, eating, breaking and burning, could all be easily handled with the normal action blocks within the object files. In my game, however, getting and dropping the objects would be done strictly with mouse input, rather than text input.

I decided, for example, not to permit the player to pick up the bottle by typing "get the bottle" on the keyboard. The mouse would have to used to drag the picture of the bottle from the location window to the inventory window. This routine must take place outside the object files, which usually respond to text input. It was important that I be able to refer to each movable object by number. In that way, I could build a single routine to move any object from the location to the inventory, and use it by setting a variable to the object number and then execute the routine. It was for these reasons that I chose to number the movable objects, but not the other nonmovable ones. As we look at the movable object files, notice all the objects are numbered from 1-19.

A Typical Object File

Look at the source code for the file named **Movable1.OBJ** and you will see that object number 01 is the ladder. As with any object file, movable or nonmovable, it starts with the name for the object that game will use, "01ladder." This is followed by the name that the player will use, and any synonyms. After this, comes adjectives that I anticipate the player may use to describe the ladder. Next comes the attribute block. I did not need any attributes for the ladder, so you will see the attribute block is empty. It is not even necessary to include the attribute block if it is empty, but I placed it here to remind you where it normally would go. The next line lists which room the object initially begins in, when the game begins. In the case of the ladder, it will always appear in the meadow when the game starts.

The Code Block

Even if there are no text descriptions of an object, its file must contain a code block

Notice the code block is also empty. In a text game, I might put some text in the code block to tell the player that there is a ladder in the room. This would also be the place to place the text that would be printed when the player takes inventory. However, since I designed a graphic game, neither of these things is necessary. The player can see that a ladder is present in the room, and does not have to be told in a text message. Likewise, the player can see the inventory and does not need a text message about what it contains. For these reasons, the code block is empty. If, however, you are designing a game where you wish to use the code block, you could use some lines similar to these:

```
IF PLAYER HAS 01LADDER THEN
 T  You carry a ladder.
ELSE
 T  There is a ladder here.
ENDIF
```

Whether you choose to fill the code block or not, every object file must have a code block. It is not optional, as was the case with the attribute block. Even if it is empty, you must include it.

The Action Blocks

Starting at line 18, you will see five action blocks for the various things the player can try to do with the ladder. The first thing that I anticipated the player might try is to get the ladder. Even though I have provided the player with a simple way to get the ladder by using the mouse to drag it into the inventory window, I have permitted the player to input commands from the keyboard. So I had to anticipate an attempt to get the ladder by typing the command instead of using the mouse. This action block simply calls a subroutine that reminds the player to use the mouse for taking objects, rather than typing the command. I could have allowed the player to pick up the object in either

way, but decided not to write the extra programming necessary since it was unlikely that the player would choose this method of taking the ladder.

Always provide a way for the player to examine objects

The next action block, which starts at line 22, is a standard one that allows the player to examine the object. This type of action block should appear for every object you create in your game, whether it is a movable object, or a nonmovable part of the background. As you can see in this action block, the description that the player receives when trying to examine the object does not have to be specific. But it can occasionally give subtle nudges to the player. With the message "it should hold you," the player is given a subtle push to try to climb the ladder. Remember that if you want a player to take some particular action, even if it is some misdirection not aimed at the actual solution to your adventure, you can offer a variety of hints and clues in various places in your game. The object description is an excellent place to leave such hints.

The next action allows the player to type "drop the ladder." Again, as with **get**, the game jumps to a subroutine which simply reminds the player that objects should be dropped by using the mouse rather than keyboard text commands.

Try to anticipate how the player may rephrase allowable actions

An action similar to **drop** is **put**. I anticipated that the player might type the command "put the ladder against the shack." I used synonyms of **put**, **set**, **lay** and **lean** for this action. I also anticipated these verbs might be used in a slightly different way, such as "put the ladder on the ground." The action block starting at line 33 shows how both of these possibilities were combined into one block.

I decided that if the ladder is placed in the meadow, it will always be against the shack, and never on the ground. The player really has no control over it. If the ladder is currently in the player's inventory, a message to use the mouse to drag it is given. Then it won't matter whether the player wanted to place the ladder on the shack or on the ground, the computer will place it where it should be placed.

If the ladder is not in the player's inventory, the player will be told that the ladder stays where it is, meaning it is not to be moved from the shack to the ground or back. Of course, if the ladder is neither in the player's inventory or the current location, Visionary will take care of it automatically by giving the player a message that the ladder is not there.

The last action block for the ladder starts at line 43. This action provides the player with an alternate way to climb the ladder. Normally I expect the player to climb the ladder by simply clicking on the **Up** button. However, as repeated many times before, you must always anticipate what logical actions the player may try to do. And it is logical to expect the player to move by typing rather than by clicking. This last action block permits the player to type "climb the ladder" whenever

the ladder is present in the room. Notice that I have used the variable **RoomNumber** to check to see what room the player is currently in, and then moved the player to the appropriate room. I could have just as easily used the room name instead of the **RoomNumber** variable by saying:

```
IF PLAYER IN MEADOW THEN
```

Both methods achieve the same result. However, if the player is not in one of the rooms where the ladder can be climbed, the player is told it can't be climbed because "it's not leaning against anything."

This completes our look at the first movable object. It was typical of most movable objects in many respects. We will now take a look at some of the other movable objects, skipping over the parts that are somewhat typical and stopping at any special features for that particular object.

More-Complex Object Definitions

Starting in line 66 you will see the object file for the bottle. Notice that it will be number 02 in Visionary's list of objects. Also notice that I provided several synonyms for the word bottle, as well as two adjectives.

Object Attributes

This object has an attribute attached to it. I created an attribute **sealed** to indicate that when the game starts that the bottle is sealed shut. It was my original intent to allow the bottle to be opened later in the game, at which time the attribute would be set to **N**. I later decided it would be easier to open the bottle simply by smashing it with a hard object like the coconut or the hammer. At that point, the attribute was no longer necessary, since as long as the bottle survived intact, it would always remain sealed. The attribute code was left in the object file as another example of how attributes can be given to objects.

The **initroom** command places the bottle at the east end of the beach when the game begins. Notice that, again, there is nothing in the code block for this object. You will find that all the code blocks for the movable objects are empty, since *Cannibal* is a graphic game and doesn't use that feature of Visionary. And after the code block come the action blocks.

Specialized Object Actions

Nearly all movable object files will start with three basic action blocks for **get**, **drop** and **examine**. You'll find these here in the file for the bottle, followed by some actions that are more specialized and specific to the bottle.

Since the bottle is sealed, I had to anticipate that the player would try to open it. I allowed three different verbs to be used in attempting to open the bottle, **open**, **unseal** and **uncork**. If the player has the bottle in the inventory, a message that the bottle won't open is given when any of these three actions is attempted. If however, the bottle is simply at the location and not in the player's inventory, a subroutine is called which gives the standard response "You don't have it."

The only way to open the bottle is to break it open. This action is accommodated in the final action block. If the player tries to break, smash, or hit the bottle, a subroutine is called that checks to see if the player holds any of the six objects that are hard enough to break the bottle, then breaks it and replaces the bottle with the paper that is inside. We will be looking at the subroutine used to swap the intact bottle with its contents in greater detail in Chapter 25.

Handling Prepositions

The object file for the battery which starts at line 151 has some interesting features that deserve a closer look. An attribute is used to make sure the battery doesn't appear more than once in the course of the game. The object file has another interesting feature, concerned with handling two different uses of prepositions in commands.

An attribute called **found** is defined in line 158. This is necessary because the battery is hidden when the game starts. The battery is stored in the **Unused** room until the player digs in the meadow with the shovel.

When the player stands in the meadow with the shovel and says "dig" I check the **found** attribute to see if the battery has been found yet. If the attribute is set to **N**, I tell the player a battery has been found, place the battery in the meadow, show it in the location window, and finally set the attribute to **Y**. If the player should try to dig in the meadow any further, checking the attribute reveals that the battery has already been found, so the message the player receives is "you find nothing." Without an attribute like **found**, the battery would keep popping up every time the player dug in the meadow.

Placing Objects Inside Other Objects

Let's next look at how some objects can be placed inside other objects. In *Cannibal*, the player can try to put the battery inside the two-way radio. Look at the source code starting at line 181. Notice there are three conditional statements. First I check to see if the player has the battery, since the object file will also be executed if the battery is at the player's location but is not being held. Next I check to see if the player specifically asked to put the battery in the radio, as opposed to in the bottle or on the roof. This is done by checking the object noun. The

last thing I check is to make sure that the player is actually has the radio in the inventory, and it is not simply at the current location.

If all three conditions are met, the game prints out a message that reminds the player that the battery is dead, and such an action is useless. If all three conditions are not met, one of three subroutines is called. The subroutine called will give the player one of three messages: "you don't have the radio," "you don't have the battery" or that the battery can't be put into the non-radio object.

This action block could have been written differently. To create an action block that would allow the player to successfully put the battery in the radio, the source code would have to be modified. Let's look at how that could be accomplished. First, you would want to create an attribute for the battery called **InRadio**. The action block might look like the one below:

An attribute like InRadio is useful when you anticipate one object being "stored" inside another

```
ACTION PUT, INSERT
 IF PLAYER HAS 04BATTERY THEN
  IF OBJNOUN IS 05RADIO THEN
   IF PLAYER HAS 05RADIO THEN
    SET BATTERY, INRADIO
    T OK.
    DROP BATTERY
    PLACEOBJ BATTERY, UNUSED
   ELSE
    T Pick up the radio, first.
   ENDIF
  ELSE
   T You can't put the battery there.
  ENDIF
 ELSE
  T Pick up the battery, first.
 ENDIF
ENDACT
```

With this code, there is no need to check to see if the battery is already inside the radio. If it is, the battery would not actually be *in* the current room, and Visionary would automatically respond that "There is no battery here." Since we know that the battery is not in the radio, we only need to check to see if the player holds the battery, whether the command specified putting it into the radio and not elsewhere, and whether the radio is in the player's inventory.

If these conditions are met, the **InRadio** attribute is set, the battery is dropped from the inventory, and moved to the unused room where such objects are stored. If you leave out the line "DROP BATTERY," the action block will appear to work properly, but Visionary's built-in **ITEMS** variable will not be properly decremented. Also, if any of the three conditions are not met, an error message can be printed.

When you choose to allow the player to put objects inside other objects, like the battery in the radio, be sure to use an attribute like the **InRadio** one used above. In that way, when the player examines the

radio, your game can check the battery's attribute and tell the player either that the radio has no battery or that there are batteries inside.

If you allow an object to be put inside another, plan whether you will allow for it to be taken back out again

Other ramifications of such actions, include taking the batteries out of the radio. You must allow the player to remove the batteries, by creating an additional action block. Since the game will not find the batteries in the current room when the player tries to remove them, the action block cannot be included in the action block for the battery object file. You will have to place it in the special vocabulary file that Visionary permits for just these reasons. More details on the vocabulary file will be given in Chapter 26. Just remember that when the player successfully removes the batteries from the radio, you must unset the **InRadio** attribute back to **N**.

Prepositions in Action Commands

Let's next look at the object file for the radio, starting with line 204. It's pretty much a standard object file, except for the last action block. There are several ways for the player to ask to turn on the radio, with the commands: "use the radio," "play the radio," "turn the radio on," or "turn on the radio."

The command "turn on the radio" will not be caught by the action block you are looking at, since the word "radio" comes after "on" and thus is viewed by Visionary as an object noun. By making **turn** synonymous for **use** and **play**, "turn the radio on" will be accepted. To make "turn on the radio" acceptable to Visionary, you must add a similar routine to the special vocabulary file mentioned above. This file would specifically include the action "turn on radio" and have the same lines of code follow it that occurred in the object file for the radio.

Handling Adjectives

The object file for the candy bar starts at line 241, and contains a special situation. When defining this object, I ran up against a problem with the name. I wanted the word "candy" to be both an adjective and a noun. I wanted the player to be able to say "eat the candy bar" as well as say "eat the candy." In the first case, the word "candy" is an adjective that describes the noun "bar." In the second case, "candy" is the noun.

Visionary won't allow you to use the same work for both adjective and noun, so this caused a problem. I solved the problem by making "candy" an adjective and then adding a special vocabulary action entry to take care of the other variation on the command.

In this way, the object file for "06bar" will catch the command "eat the candy bar" as well as "eat the Snicker," "eat the Snicker bar," and "eat the Snicker candy bar." The final variation "eat the candy" was added to a vocabulary action entry, so that the game would understand the

player regardless of the choice of words. Since the actual code for eating the candy bar was rather lengthy, I put it in a subroutine rather than duplicate it both here and in the vocabulary file. We'll be looking at that subroutine in Chapter 26.

Anticipating Player Actions

Let's move on to the second group of objects found in the file called **Movable2.OBJ**. The first object in this file defines the dead sea gull. I included an entry that anticipates the player may try to eat the dead bird. This is the last action block in the object file. It is doubtful that the player will try to eat a dead bird, but the precedent has been set. The player was allowed to eat the candy bar, and was permitted to drink the salt water of the ocean. So it is only logical to assume that some player may extend the concept to consuming the bird.

I had to anticipate and make an appropriate response for that action. Remember, you must allow the player to take any logical action with the objects in your adventure. It would not be fair for the player to ask to eat the sea gull only to be told that "You can't" or "I don't understand you." Usually the best way around this problems is to tell the player in a short message that the action can't be performed, and also **why**. In this case, the player is told the sea gull can't be eaten it because it is "too disgusting".

Combining Objects

Next consider the object file for the shovel blade, starting at line 78. It contains a routine for putting the shovel blade on the shovel handle to make a complete shovel. I want to examine this routine in detail, since it is an important part of the graphic interface.

Your game may require two objects to be combined to create a third, single object

What the player sees on the graphic screen is two objects in the inventory window which then become a single object. The player starts with the two separate parts of the shovel, the handle and the blade. Both parts will show in the inventory window. When the player asks to put the shovel blade on the handle, both objects will disappear and be replaced by the picture of a complete shovel. Let's see how the game uses the Visionary commands to make this happen.

Look at the action block for the shovel blade, starting at line 104. The first thing I did was to choose four different verbs which would allow the player a wide variety of ways to express the command: "put the blade on the handle," "connect the blade to the handle," "stick the blade on the handle" or "push the blade onto the handle." Next, I check to see that the player actually wants the blade to go on the handle, not somewhere else.

Once I have checked to see that the blade is the object noun of the command, I check to see if the player has the shovel handle. If there is

a problem with any of these things, the main part of the routine is skipped, and a subroutine is called that either tells the player "you don't have it," or "you can't put the blade there." If everything is in readiness, the main section of the routine is executed.

Once two objects are combined, the parts must be removed from the inventory and replaced with the new object

Several things take place rapidly when the shovel blade is put onto the handle. First a message is printed out that not only tells the player that the action was successful, but also suggests that the shovel could be used. Next, the handle is destroyed. This is done by setting a variable **ObjNum** to 12. This variable keeps track of the object number, which for the shovel handle is 12. Then a subroutine which takes care of placing the object in the **Unused** room is called. This subroutine, which will be examined in detail in chapter 24, finds and erases the picture in the inventory window, and modifies the array that keeps track of which inventory spots are open. The shovel blade is then destroyed in the same way. The variable **ObjNum** is set to 11, which is the number of the blade, and the same subroutine is called. At this point, both the blade and the handle have been removed, and the next step is to place the entire shovel in the player's inventory.

Rather than hunt for an empty spot in the inventory window, I simply use the spot previously vacated by the shovel blade. First, I write the new object number in the inventory array.

Visionary does not support data structures like arrays directly. But there is an easy way to read and write to arrays in an indirect manner, using Visionary's ability to see and change individual pixels on any graphic screen. This will be explained in more detail in Chapter 23 when we look at getting and dropping objects. But for now, we'll quickly note that the array is created as a series of colored pixels on graphic screen 2, where each color stands for a different object. There are 19 objects, which means I had more than enough colors, since I was using the 32 color mode. In line 114, I set the mode for graphic screen 2 to draw. In the next line, I set the color to be used on that screen to the variable **ObjNum**. Finally, I drew a one-pixel rectangle on screen 2, at the coordinates that stood for the empty spot in the inventory window left by the shovel blade. The X and Y coordinates were obtained previously by the **DestroyObject** subroutine used to remove the shovel blade.

After adjusting the array on screen 2, the next step was to actually draw the picture of the shovel in the inventory window. Remember that the picture of the shovel, as well as all the other objects is found in the graphic file called **buttons.pic** where they are all lined up neatly at the bottom on the screen, each starting exactly 15 pixels from the start of the previous one. And they are all in numerical order; that is to say the ladder is first, the bottle is next, followed by the paper, the battery and so on. Given the object number for the shovel, and using some simple arithmetic, I could figure out exactly where the picture of the shovel was. Given the X and Y coordinates of the inventory array, and again

doing some simple arithmetic, I could figure out exactly where to place the picture of the shovel in the inventory window.

Before doing the actual copying of the picture, I set the mode for screen 0 to overlay. In this way, the only the picture of the shovel would be copied, not the background. Then in line 118 the actual copy took place.

Let's look at the mathematics of the copy command in line 118. Once you understand the principal of using simple expressions like these in your copy commands, you will have a powerful tool for creating your own graphics games at your disposal. Let me reproduce the line below:

```
copy 2,ObjNum * 15 - 15,183,ObjNum * 15 -
  1,199,0,x * 16 + 266,y * 18 + 17
```

Since I'm copying from screen 2 with all the object pictures to screen 0 with the visible game screen, the first number is 2. Next comes the expression "ObjNum * 15 - 15" which gives the X coordinate for the upper left corner of the shovel. I multiplied the **ObjNum** variable by 15, because each object picture started 15 pixels from the previous one. From this, I subtracted 15 since I started the first object (the ladder) at 0, not 15.

The number 183 indicates the Y coordinate of the upper left corner of the shovel. Since all the objects lie in a straight line, they all share this coordinate. The expression "ObjNum * 15 - 1" gives the X coordinate of the lower right corner of the shovel. Again, I multiplied by 15 since each object was 15 pixels wide, and subtracted one to get to the last pixel for the object. The number 199 refers to the Y coordinate for the lower right corner of the object. All objects lie on the bottom of the screen, and share this coordinate. At this point, all information to tell the computer where the picture of the shovel currently is is present.

The second part of this line tells the computer where the picture of the shovel should be copied. The zero tells the computer to copy the picture to screen 0. The expression "x * 16 + 222" takes the array value for X and changes it into an X coordinate for the upper left corner of where the shovel will be placed. I multiplied by 16 because each object was 15 pixels wide and I wanted one extra pixel between the objects. Adding the 222 slid the object over from the left edge of the screen into the inventory window.

The expression "y * 18 + 17" takes the array value for Y and changes it into the Y coordinate for the upper left corner of the place where the shovel will be copied. I multiplied it by 18 because each object is 17 pixels long, and I wanted an extra blank pixel between the objects. By adding 17, I slid the object down from the very top of the screen to within the inventory window. With the completion of this single line of code, the picture of the shovel was placed in the inventory window.

After copying the picture of the shovel from screen 2 to the visible game screen, there were only two things left to do. The last two things were place the object in the current room and grab it. By doing this, the internal workings of Visionary knew that the object was in the player's inventory, and the built-in **ITEMS** variable was correctly incremented. Without these two steps, the picture of the shovel would have appeared on the screen, but the game would have not always known it was there.

The above routine for putting the shovel blade on the handle has taken quite some space to explain, but it is an important routine and deserves the effort you have given to understanding it.

An Example of Poor Programming

There are actually three different ways of putting the shovel together. The player can say "put the blade on the handle," "put the handle in the blade," or simply "make a shovel." In the first example, the word "blade" is the object noun and so the action should take place in the object file for the blade. In the second example, the word "handle" is the object noun, and the action should be placed in the object file for the handle. In the third example, the word "shovel" is the object noun, and since the object is not present when the command is given, the "make a shovel" action must be placed in the vocabulary action file in order for the Visionary game to understand it.

This routine to put the shovel together will be used in three different places, in the "blade" file, in the "handle" file, and in the vocabulary action file. If you will look at the source code, you will see that I have placed it in all three locations.

Use a subroutine to handle code that is repeated several times

This is an example of **poor** programming, intentionally done to show you what **not** to do. Since the exact same routine is used three times, it would have been much better to write the routine only once, as a subroutine. This subroutine could then have been called from the three separate places where it was needed. By leaving such a large routine in the program three times, a lot of space was wasted unnecessarily. Remember to use subroutines to make your game more efficient.

Looking through the rest of the movable objects in this file reveals little new about their construction. They are all pretty routine, and have no special features that need additional attention. We have finished now looking at the way movable objects are used in *Cannibal*. Next, we will be looking at the main loop, which is the heart of the game. In Chapter 23, we will see how it controls actions, and allows the player to access the rooms and objects that were created so far.

Chapter 23: The MainLoop.SUB File

Once my game has started, it continues in a loop checking for the player's input and acting on it. This is the heart of my adventure, and can be found in the **MainLoop.SUB** file. This chapter will look at how the player can input commands from either keyboard or mouse.

Although this file is the small compared to the ones examined in previous chapters, it is vitally important. For that reason, I intend to go through each section slowly and in detail. Follow each step as closely as you can, and you will be rewarded with some powerful and valuable routines you can use in your own games.

A General Overview

Before starting our microscopic examination of the main loop, let's take a more general look at the entire routine overall. It basically provides a small word processor, to allow the player to input commands from the keyboard. This would not be necessary in a text game, since text commands are handled by Visionary's internal routines. But I planned a graphic game, which meant that I needed routines to would allow the player to type on a graphic screen.

To be more specific, I had to write the source code that would check to see if the player had pressed a key, then print that letter on the graphic screen. I had to build words from the individual key presses, taking into account that the player might use the [BackSpace] key to remove errors. I also had to notice when the player pressed [Return], and send the finished line of text to Visionary to be acted upon.

The design of the main loop is split into two sections. The first section is the text input section. This is where the game watches the keyboard for the player's input, and also checks the mouse for any button clicks. Once a command has been entered either by a mouse click or by pressing [Return], the program moves to the second section of the main loop.

In the second section, the computer executes the command given by the player and reports back to the player any errors that have occurred. The second section also takes care of some special situations that occur if the player requests to load or save a game position. Once this is done, the program jumps back up to the start of the loop and continues the process all over again.

One of the first things you will notice when you look at the **MainLoop.SUB** file is that it contains a single subroutine. Admittedly, this is a rather large subroutine, but it is composed of only one subroutine

called **MainLoop**. The while loop at line 8 is the beginning of the main loop. The loop, which ends at line 186, is only exited when the player wins, loses, or quits. If that should happen, then the lines 190-192 are executed, which empty RAM: of the location scenes that were stored there and then quit. Quitting exits the program and returns the player to the computer's operating system. But until this happens, the game continues to loop through lines 8-186.

At the beginning of the main loop in line 8, you will see that I chose a variable named **MainLoop**. Since I set no value in the .ADV file for the variable, it defaults to zero. And it stays zero throughout the whole game, ensuring that the game keeps repeating the loop over and over again. Only when the game is won, lost, or quit is the value of **MainLoop** set to one, so that the loop will be exited and the game will be stopped.

Checking the Input Window

The beginning of the loop contains some initialization steps necessary before going into the loop that will check for the player's input from the keyboard.

When the initialization loop begins, I need to move up any existing text in the text window by one line, and leave a blank line for the player to type on. A subroutine is called to move up the text and create a blank line at the bottom of the text window. Part of this subroutine will check to see if six lines have been printed, and if so it will stop in order to permit the player to read the text before it scrolls out of the window. It will print a message prompting the player to click the mouse button to continue. The number of lines printed so far is kept in a variable called **CountLines**.

In order to initialize this routine for the next time around and at the same time to keep the message from being displayed in place of a blank 6th line, I first set the variable to -1 in line 10. Then the **Linefeed** subroutine is called, which takes care of sliding the text up and blanking the sixth line.

Setting Up for Special Characters

The next five lines from 13-17 set up some variables that will be used in the next loop. The next loop will check for keypresses and build up these keystrokes into words and eventually a full sentence. It will exit when the **[Return]** key is pressed. For this reason, I have created a variable called **Return** which will be the trigger for the next loop. In line 13, I set this variable to 1, and it will stay this value until the player presses [Return], at which time it will be set to zero and the loop will be exited.

I next set a variable called **TextPosition** to 9. When each move starts, the cursor will be on the left side of the text screen, exactly 9 pixels from the edge of the entire screen. As the player types, this variable will increase, until the player threatens to leave the boundary of the text window, at which time the game will stop accepting keystrokes.

The next variable in line 15 is the string variable **$sentence**. This will contain the sentence that the player types, as it is built up letter by letter. Since the player's move has just started, I have set the value of the string variable to null.

[Return] and [BackSpace] characters are handled differently than other text

The next two variables are used to check for certain special keystrokes. In line 16 I define a string variable **$return** to be the [Return] character. This is done by using the back-slash key followed by "r". As documented in the Visionary manual, there are several special characters that can be accessed by using the back-slash. In this case, I want to have the keystroke stored in a variable so that I can compare it with the player's keystroke. In that way I will know when [Return] has been pressed, and it is time to exit this input loop.

In line 17 I define another string variable for the [BackSpace] character. Again, notice that I use the back-slash method of defining the character. This will be used to compare the player's keystroke with the backspace, so that when the player makes a mistake it can be corrected, and my input routine will respond properly. Having defined these five variables, we are nearly ready to enter the input loop.

Text Color

Text color can be used to provide extra information

Before entering the input loop, there are two more lines, 19 and 20, to consider. I decided that the text that appeared in the text window should be different colors. As you have seen, the text input by the player is green. The computer sends its messages in blue. Warnings about cannibals landing are printed in red, and "prompt" messages telling the player to "press the mouse button to continue" are in brown.

In line 19, I set the color for the pen on screen 0 to green. Remember, **green** is actually a variable that I had previously set to 28 in the .ADV file. In the palette for screen 0, color number 28 is the green. By doing this, I know that all successive text printing on the graphic screen will be done in green. The next command actually prints some text to the graphic screen.

The Prompt Character

The **text** command in line 20 prints a single text character to the graphic screen. In a normal font, the character would be the "~" symbol. However, I decided that the "~" would become the cursor. I wanted a cursor to show in the text window where the text input occurs.

The prompt character can be changed from the default *tilde*

Rather than draw a small solid rectangle using Visionary's graphics commands, I chose instead to change the font so an otherwise-unused character could become the cursor. So when I designed the special font for this game, I changed the shape of the "~", in the font file itself, to a solid rectangle. Thus, what line 20 really does is to print a rectangular cursor on screen 0, at coordinates 9, 192. Line 20 is the way that I set the cursor to the beginning of the bottom line in the text window, in anticipation of the player's typed input.

The while loop in line 24 is the beginning of the input loop. This loop checks for keystrokes as well as mouse clicks, and ends at line 96. It continues looping while the value of the variable **return** is not zero, finally exiting when the variable is set to zero.

In line 26, a character is accepted from the keyboard, and stored in the string variable, **$letter**. Whether a key has been pressed or not, the routine continues to the next line. The next line detects whether a key was pressed or not, by checking the length of the string variable. The variable **temp** will be 1 if any key was pressed, or 0 otherwise. The line after that stores the ASCII value of the character input, into the variable **val**.

Handling Unwanted Keystrokes

You need to provide for keys which will not add to the text command string

Line 29 masks out unwanted keystrokes. The method used is the simple logic statement "val > 127". Remember that **val** contains the ASCII value of the character input by the player. If it the ASCII value is less than 127, then the character is one I want the game to accept. Otherwise, it could include unwanted characters such as the function keys or cursor keys. I don't want my input routine to accept those, so this line will help me eliminate them. If the keystroke is one that I want to accept, then **val** will be less than 127 and the expression "val 127" will be true, and evaluate to 1. Otherwise, the expression is false, or 0.

What I am actually doing in line 29 is multiplying **temp** by either 0 or 1. So now, **temp** will be 1 only if a key is pressed **and** it is an acceptable key.

In line 30, the character typed by the player is compared with the variable **$return** to see if the player pressed [Return]. And in line 31, the character typed by the player is compared with the variable **$backspace** to see if the [BackSpace] key was pressed. At this point, I have four variables to use. I have stored the character in the variable **$letter**, I know whether it is an acceptable character by checking **$temp**, and I know if it is either a [Return] or a [BackSpace].

Checking for Mouse Input

In lines 33-46 I temporarily disregard the keyboard and concentrate on any mouse clicks that were recently made. Later on in the **MainLoop**

routine, at line 90 to be exact, the **readbuttons** command is given. When that happens, the computer checks to see if any buttons have been clicked on, and if so jumps to the routine named in the associated **click** command, then it jumps back to the line immediately after line 90.

The lines of source code from line 33-46 will act on any mouse click caught on the previous pass through the while loop. These lines will make the button appear to move inward on the graphic screen, as though it had been actually pushed in. Then they will cause the button to pop back out again, as soon as the player's finger is lifted from the mouse button. Let's see exactly how this happens.

The line at 33 checks to see if a button on the game screen was pressed. It does this by checking a variable called **ButtonUsed**. This variable is set in any subroutine that is accessed by a click zone. If no button was used in the previous pass through the while loop, the variable will be 0.

If an object was moved from the location window to the inventory window or back, the **ButtonUsed** will be set to 9. If the player clicked on the **LOAD** button, the variable would be set to 4. If the player chose **SAVE** instead, the **ButtonUsed** variable would be set to 5. By pushing any of the other action buttons or compass buttons, the variable will be set to 1. And if any part of the scenery located in the location window was clicked on, the variable will be set to 2. You can now see that line 33 will catch any time the player pressed on one of the buttons, but ignores clicks that move or examine objects. Once the routine detects that a button has been used, it executes the next eight lines of code.

The first part of this routine, found in line 34, sets the **return** variable to 0 in order to fool an upcoming routine into thinking the player has typed the command and pressed [Return]. In this way, commands such as **help** or **N** can be input from either the keyboard or the mouse.

In line 35, the button is actually shown pressed inward. This is done by copying the picture of the button in the down position from screen 2 to the proper spot on screen 0. The copy command expects the X and Y coordinates for the upper left corner and lower right corner of the object to be copied, and the upper left corner of the place it is to be copied to. Each of these coordinates in line 35 is a variable. The values of the variables are set in the subroutines that were called as soon as the click zones were activated, and reflect the location of the button that was pressed as well as the location of its alter images on screen 2.

After the button has been pressed inward, I didn't want it to pop up again until the player's finger is lifted from the mouse button. To accomplish this, I use a simple while loop to check the status of the mouse button and keep going around in the loop as long as it is being held in.

However, I discovered a strange quirk in the Amiga that required a short pause before this while loop. It has something to do with the blitter being busy with the previous copy command, and briefly locking up the computer before the while loop. The symptoms of the problem were that the mouse pointer would occasionally freeze up for a second, immediately after clicking the mouse button. It only happened occasionally, and never for more than a second. But it was annoying to have the mouse temporarily freeze up while I was trying to move it, and then finally to have it jump across the screen to the current location of the mouse.

A pause after COPY may be needed to prevent brief mouse "freezes"

I discovered that a simple solution to the problem was to place a brief pause after the copy command, but before the while loop. A pause of 10 or 15 is sufficient to completely eliminate the problem, while at the same time being short enough to escape human notice. It apparently gives the blitter enough time to complete its work with the copy command and avoid any freeze-up, but the pause is not noticeable by the player. Keep this tip in mind when you write your own games. If you notice an occasional brief freeze-up in the mouse pointer, track down the copy command that is being executed when the freeze-up occurs, and place a short pause after it. It can make the game run smoother.

Lines 37 and 38 keep the game in a while loop until the player's finger is removed from the mouse button. Then the routine moves on to the last three lines. These lines adjust the two Y coordinates for the button that is about to pop back up again. Usually, the picture of the button in its "up" state lies immediately on top of the picture of the same button in its "down" state. So when preparing to show the button popping back up again, only the Y coordinate of the button on screen 2 needs to be changed. Each button is exactly 13 pixels tall, so usually it is only necessary to add -13 to the Y coordinate, and then use the same copy command as in line 35.

However, the compass buttons are the exception to this. When the compass buttons pop back up, they can be either highlighted, or ghosted. They are highlighted if the directions are active and the player can go that way. If the directions are inactive and the player can't travel in that direction, they are ghosted. When drawing the buttons on screen 2, I placed three nearly-identical buttons on top of each other. For example, the top button might show N in the highlighted state, the middle button shows N as it appears when pushed in, and the bottom button shows N in the ghosted state.

When the player presses a compass button, I want it to pop back up to its proper state. I do this by adding -13 to the Y coordinate to return the button to the highlighted state, and by adding 13 to return it to the ghosted state. The variable **offset** is set to either 13 or -13 in the **ReDrawScreen** subroutine which we will be examining in a later chapter. Lines 39 and 40, then, set the new Y coordinate for the picture of the button, in preparation for showing the button popping back up. Line

41 sets the variable **offset** back to the standard default of -13, since the popped-up version for most buttons is 13 pixels above the pushed down version. At this point, the button has been shown pushed down, and preparation has been made to show it popping up again.

In lines 44-46, the button is shown popping up again. Only if the variable **ButtonUsed** equals 1 does the button pop up. This means that if the player pressed the **LOAD** or **SAVE** button, their popping up is not handled here. Their pop-up display is handled elsewhere, since I felt it was more appropriate to show them popping up *after* the load or save operation had been completed. Line 45 shows the same copy command as in line 35. Since only the two Y coordinates have been changed, the copy will place the proper picture of the button back on the graphic screen.

In the last two sections of code, if it was discovered that a button had been pressed, that button was shown being pushed inward and then popping back out again. The variable **return** was also set to 0. The reason for setting it to 0 is shown in the next routine.

Text Input

The next routine, going from line 48-77 is a long one, and comprises the actual text input routine. Before looking at each line individually, let's get a general understanding of what the routine does.

First it checks the most recent keypress. If it finds the [Return] key was pressed, it erases the rectangular cursor from the line. If the key was the [BackSpace] key, it either shortens the sentence and adjusts the text showing, or in the event that the cursor is at the left border, it does nothing. If this routine finds that the keypress threatens to extend beyond the right border of the text window, it also does nothing, and ignores the keypress.

Finally, if this routine finds that an acceptable keypress was made, it adds it to the sentence, both on the screen and in memory. That's basically the substance of this routine. Let's look at each part in more detail.

The [Return] Character

The keyboard input routine starts at line 48 by checking to see if the [Return] key was pressed. This could happen either by the actual keypress being caught back in line 30, or by the mouse clicking on one of the action or compass buttons being caught back in line 34. Either way, I want the cursor to disappear.

To do this, I will simply draw a small white rectangle on top of the green rectangular cursor. As you can see, this takes four simple steps. First the color for screen 0 is changed to white. Don't forget it is actually being changed to color number 8, but I defined the variable **white**

back in the .ADV file so that I wouldn't have to remember the numbers.

Then I set the mode for screen 0 to draw. In line 51, I actually draw the small rectangle at the current text position. The variable **TextPosition** was initialized earlier in line 14 and has been kept current as each key has been pressed, as will be seen in the next few lines.

Finally in line 52, I place the screen in the overlay mode in preparation for future text commands. And that's all that this routine does when it detects that [Return] has been pressed. More action will be taken later, but for now my game only requires that the cursor be erased.

The [BackSpace] Character

If the player has pressed the **[BackSpace]** key, the next lines of code starting at line 53 will be executed. First I check the length of the sentence that I have been building with each keystroke the player makes. If I find the length of the sentence is 0, then I know that the cursor is at the left border of the text window, and I do absolutely nothing; I ignore the [BackSpace] key. Otherwise, I start the process of **deleting** the previous character.

A special routine is needed to handle character deletion in a text window

To delete a character takes several steps. First, in line 57 I shorten the length of the sentence by one. This only affects the variable that keeps track of the length of the sentence. Then I adjust the variable **TextPosition** backwards by 6 pixels. This variable should always point to the position where the next character will be printed. The next line actually shortens the sentence itself, by using the newly-modified variable **sentence** that tells the current length of the sentence. It takes the left portion of the variable **$sentence**, which is left when the line string is shortened by one, and stores it back in the same variable again. This may not be clear, so let's take a look at a specific example.

Let's say the player presses [BackSpace] after typing the sentence "EAT BIRT". The sentence is stored in the string variable **$sentence** and the length of 8 is stored in the numeric variable **sentence**. When [BackSpace] is pressed, the variable **sentence** is changed from 8 to 7. Line 59, the left 7 characters of the string variable **$sentence** are stored back in the variable again. The variable **$sentence** will now hold the shortened words "EAT BIR".

After the actual string variable is shortened in line 59, I erase the letter from the graphic screen along with the cursor that follows it. This is done in line 60 by changing the color for screen 0 to white, to match the text window background. I switch to the draw mode, so the white rectangle I am about to draw will cover the deleted text. Line 62 actually draws a double wide rectangle to erase both the character and the cursor.

Then I must move the cursor back to the new position. This is done by switching to the overlay mode, to prevent any of the black background

color showing through on the white text window. The color is set to green, and the cursor is placed at the current text position. That's all it takes to backspace in the text input routine.

Adding to the Text Input

If the player has typed something and it is not [Return] or [BackSpace], I know that the character should be added on to the current text in the text window.

In line 67 I check to see if the text position has exceeded the width of the text window. Without a check of this sort, the player could easily keep typing past the end of the text window and onto the buttons on the right side of the screen. It is at this point in the text input routine that I need to check to see if the keystroke will be allowed. If the text position exceeds 240, the keystroke is ignored. It is important to note that this part of the routine comes *after* the check for [Return] and [BackSpace]. If this part of the routine were placed earlier, the player could type into a corner from which there would be no escape.

The last section of the text input routine checks to see if an acceptable keystroke was made, and if so it adds it both to the string variable **$sentence** as well as to the graphic text window. Line 68 checks to see if a keypress has been made that is acceptable. Line 70 adds the character to the sentence. Remember that to connect two string variables together—technically called "concatenating two strings"—you simply place both string variables inside quotes, each preceded by a "@" symbol.

Erase the cursor *before* a new text character is added to the line

I can't just place the letter on the screen, on top of the cursor. In the draw mode, it would give the letter inside a black rectangle (the black background showing through). In the overlay mode, it would give the letter inside a green rectangle (the green cursor showing). So I have to erase the cursor first, then place the letter on the screen, then place the cursor in the new position.

You can see how easy this was in lines 70-75. I set the color to white, the mode to draw, and draw a white rectangle over the cursor to erase it. Then I set the color to green, set the mode to overlay (to overlay the green letter on the white text window without letting any of the black background show through), and place both the letter and cursor in the text window. The last thing to do in line 76 is to update the new text position variable. Each letter in my specially designed font is five pixels wide, with one pixel extra between letters. That's why I add 6 to the variable **TextPosition**.

Keep in mind that all the above code for the text input is executed in a split second as the game looks for a single keypress and processes it correctly. When no key was pressed, but a mouse click was detected instead, remember that the above routine was tricked into believing that [Return] had been pressed, and it erased the cursor. The next

routine checks for such an eventuality and prints the text to the screen, just as if the player had used the keyboard instead of the mouse.

Mouse Input

The routine at lines 79-88 prints the command that comes from any mouse click into the text window. If for example, the player wants to examine the rowboat, the command can either be typed, "examine the row boat", or it can be executed by simply clicking on the picture of the rowboat.

When the player clicks on the picture of the rowboat, I want the words "examine the rowboat" to appear in the text window exactly as if the player had typed them. In this way, the player knows exactly has been done by the mouse-click on the rowboat—it has been examined.

Echoing Mouse Commands to the Text Window

The routine starting at line 79 takes care of this **echoing** of the command to the text window.

Text that will be echoed after a mouse click is set up in the subroutine called by the click command

In the example of the rowboat, I have defined a click zone that contains the rowboat. I specified the exact pixel locations of the invisible rectangle that contain the rowboat. I also specified the name of the subroutine that should be called when this zone was clicked in. If you want to jog your memory, you can look back at the **Cannibal.ROOMS** file at line 30, and you will see the name of the subroutine called is **SeeBoat**. This is a very simple subroutine that only does two things. It sets a string variable **$tx** to contain "examine the row boat" and sets the **ButtonUsed** variable to 2. Then it jumps back here to the main loop and lets this routine take care of the rest.

This text echo routine first blanks out the last line in the text window. This covers the case when the player may have been in the middle of typing something, but decided to use the mouse instead. This routine, in lines 81-83, will erase any such partial text input.

The mode is set to draw, although it would also work with overlay. The color is set to a matching white, and a long thin rectangle is drawn in line 83 which blanks out the last line in the text window. Line 84 sets the color to green, and line 85 calls a subroutine which actually prints the text stored in the **$tx** variable.

In preparation for actually sending the command to the game to act on, the string variable **$sentence** is set to the same string as the **$tx** variable. Finally, the variable **return** is set to 0 in line 87. It may or may not already be set to 0 depending on whether a button was pressed, or an object was clicked on.

The **readbuttons** command in line 90 is all that is required to make the computer check for any mouse clicks. Everything else was set up previously, by defining the click zones in the room file and by writing the subroutines called by them in the subroutines file. We will be looking at the subroutines called by the click commands later, but they are generally very simple. Since everything else is set up, the only thing that the main loop's input routine needs to do, is this one command to see if any click zones have been activated recently. If any click zones have been activated, the assigned subroutine is called and the program jumps back here to line 91.

In line 91, I empty the queue in case more than one button was pressed while the computer was processing. Because of the speed of the computer, it is very unlikely that more than one mouse click will occur during the split second that the computer executes the subroutine and returns here, but it is wise to include such a line just in case.

The last thing I do before ending the input loop is to check to see if an object was moved from the location window to the inventory window or back. If it was, then the variable **ButtonUsed** will be set to 9, and I'll force the input loop to exit by setting the **return** variable to 0. I do this because I want the getting and dropping of objects to count as a turn.

Ending the Input Loop

Line 96 marks the end of the input loop. At this point, the loop either jumps back up to line 24 and starts all over again, or it exits and continues onto the next section where the sentence input will be acted upon. If [Return] has not been pressed, it will loop over and over until either a button is pressed or a sentence is typed. The game spends most of its time here, waiting for the player to do something. But once the player's input occurs, the game moves on to the next section.

Executing the Commands

The next section uses Visionary's powerful **GHOST** command to take what the player has requested and make the game act on it. The GHOST command makes the Visionary game do exactly the same thing that it would do if the player typed the command onto the text screen and pressed [Return]. But since I designed a graphics game, not a text game, I could not use the normal text screen and the built-in text input routines. The GHOST command gives me an easy way to accomplish the same thing. But first, there are a few housekeeping chores to take care of.

If the player has simply moved an object from the inventory to the current room, there is no need to ghost any command. So in line 100 I check to make sure the player didn't get or take any object. Otherwise, the **ButtonUsed** variable would be 9, and this whole section would be

skipped. As long as the variable is less than 9, the lines from 102-180 will be executed. Let's look at the four different routines in that area.

In line 102, the color is changed to blue. This is done so that any text that is printed to the text window will now be blue. I wanted these messages from the computer to be a different color than text typed by the player. Line 102 takes care of this color change.

The LOAD Command

In the next section, I check to see if the player has typed or clicked on **LOAD**. If so, I will switch to the text screen, and use Visionary's built in requester system for selecting drives and filenames.

In line 104 I check to see if the sentence typed was **LOAD**, and if it was, the variable **dummy** will be set to 0. Then I check the variable and if it is 0, I start by setting the **ButtonUsed** variable to 4. This may already be 4, if the player clicked on the **LOAD** button, but I must consider that perhaps the player used normal text input instead. Either way, the variable is now set to 4.

Then using the **T** command to type on the text screen, I clear the screen in line 107 using one of Visionary's special "back-slash" characters mentioned previously. I move down to the center of the screen and type "Please Wait for Loading."

The LOAD button pops up only when the load function is complete

Remember that if the **LOAD** button was depressed above in lines 33-42, it was not shown popping back up again in lines 44-46. The reason is that I want the button to stay down while the load takes place. Now is the time I set the three sets of X and Y coordinates necessary to show the **LOAD** button popping back up, which will take place later in line 173. The final part of this routine is to switch control over to the text screen in line 124.

The SAVE Command

The next section of code from lines 127-149 does exactly the same thing for saving as the above section did for loading. There are some necessary differences, however.

You don't usually need to save a mask buffer with a saved game

The **ButtonUsed** variable is set to 5, to indicate what command was given. The X and Y coordinates are different, since a different button will be shown popping back up. And there is a special command in line 147. By unloading screen 24 before saving the player's position, I have saved some time and memory. Screen 24 is my mask buffer, originally defined in the **StartUp.SUB** file. It contains nothing that needs to be saved when the game position is saved, so by unloading this one screen before the game is saved, the time it takes to save the game is shortened. And the saved game file is smaller.

However, the mask buffer is important to the working of the game, and I make sure to open the mask buffer again after the saving or loading

of any game position. We'll see exactly how that is accomplished in line 170, to be explained a few paragraphs below.

At this point in the game, I am ready to GHOST the player's command to the computer. First, I check to see if the player has typed or clicked on **QUIT** in line 151. If the **temp** variable is 0, then all I have to do is set the variable **MainLoop** to 1 and I know the main loop will exit and the game will end. If the player has input anything other than **QUIT**, I ghost that command to the computer at line 155. Since I am ghosting the value of a variable, I need to place the "@" sign in front of the variable.

TURN lets you count each command as a turn, even if player location doesn't change

Also notice that I use the **TURN** option at the end of the GHOST command. This forces the Visionary code to take a complete turn after receiving the ghosted command. It forces the game to check the vocabulary action file and the NPC file. Since I am forcing the game into a never-ending loop, these files would not be properly executed as they would otherwise. The reason I am forcing the game into a never-ending loop is so that I can allow text to be input and printed to a graphics screen.

Error-Trapping

After ghosting the player's command to the computer, it is important to check for any errors. If for example, the player asked to "EXAMINE THE ROWBOAT" while in the meadow, Visionary would report an error to the text screen. I want to echo that error to the graphic screen.

In line 156 I check the error variable for any error. If I find one I next see if the player was trying to load or save a game position. If that is the case, I print out a special error message in line 158. Otherwise, I set the **$tx** variable to the **$LastError** variable and call the usual print routine.

Back to the Graphics Screen

Remember that if the player was trying to load or save a game position, my game switched to the text screen before ghosting the command. Now that the command has been successfully ghosted and any error reported, I need to switch the player back to the graphic screen.

In line 166 I check to see if the command was to load or save a position. If so, I start by clearing the text screen in line 167, just to make sure that the load/save message is removed. It should be noted again that if Visionary has to pop up any requester windows, it always switches back to the text screen to show them.

For example, if *Cannibal* tries to load a MED song from the disk and the disk has been removed from the drive, the game will switch to the text screen and present a typical requester box telling you to put the disk in the drive. By clearing the text screen after a load or save, as I

have done in line 167, I make sure that the text screen will be empty in case any unexpected requesters pop up.

Clear the text screen when you switch back to the graphics screen

I next switch the game back to the graphics screen in lines 168 and 169. Having unloaded the mask buffer before saving in line 147, I now put it back in place in lines 170 and 171. If you use this technique to save memory and time in your adventures, don't forget these two lines of code. It's easy to remember to create the screen, but forget to assign it as a mask.

Lines 172 and 173 show the **LOAD** or **SAVE** button popping back up, by setting the mode for screen 0 to draw and the copying the button using the X and Y coordinates set just before the command was ghosted. At this point the game is nearly ready to go back into the input loop again. But there is one more vital routine to include.

Using RAM:

The routine that runs from lines 174-179 is vital, since my game will sometimes load the location scenes into RAM.

In the **StartUp.SUB** file I checked the available RAM in the computer. If there was enough room, I loaded the location scenes into memory. The game can play with the location scenes either on disk or in memory, but it plays much faster if they are in memory, due to the absence of disk-access time.

Check for use of RAM: after a game position is loaded, and reset the appropriate variables

After loading a previously-saved game, it is important to once again check to see if the location scenes are in RAM or not. If it tries to load them from RAM, and they aren't there, the game will crash. You can't trust the **$device** variable that was saved with the game, because the status of the computer memory may have changed since the game was saved. The technique of keeping files in RAM is a valuable one, and I recommend it to create a better game. But if you are going to use it, you must also check RAM again after a game is loaded.

Let's look at lines 174-179 and see how they work. The idea is really quite simple. I just look into RAM to see if the location scenes are there, and then set the **$device** variable accordingly. In line 174, I try to load the current location scene from RAM into screen buffer 1. Then I check the error variable in the next line.

If an error has occurred, then I assume the screen does not exist in RAM, and I set the string variable **$device** to the disk pathname "Cannibal:Video/". If no error occurred when I tried to load the screen from RAM, then I set the **$device** variable to **RAM:**.

Exiting the MainLoop

The last step in the entire main loop is to set the **ButtonUsed** variable back to 0. In this way, when the input loop is entered again, the but-

tons won't be activated unintentionally. The main loop ends in line 186, and as long as the **MainLoop** variable remains zero, the loop continues forever. The **MainLoop** variable is only set to 1 when the player wins, loses, or quits. When that happens, the game passes out of the main loop.

When the main loop is exited, there is one last thing I do before actually quitting. Since I may have placed all the location scenes into RAM for faster access, I need to delete them again. In line 190, I remove all the files using Visionary's **DOS** command. I send any system messages to NIL: so the player won't receive a message like "deleting LOC1" on the screen. By using the "#?" wildcards, I can delete all the files with a single statement. And then with the **QUIT** command, the game stops and sends the player back to the operating system.

» Always clean up memory! Never end a game with memory tied up for any reason.

This has been a long chapter to describe a short subroutine. Because of the importance of the subroutine, it was necessary to go into great detail. I hope you have found many routines you will be able to use in your own Visionary games. Some of the concepts presented may have sounded complicated at first, but once examined in detail, should have proven to be actually quite simple.

The next chapter will cover another subject that also appears complicated at first. We will be looking at the movement of graphic objects from the location window to the inventory window. Again, these routines will be very valuable in your own games.

Chapter 24: The GetDrop.SUB File

Remember when you played the *Cannibal* game, how easy it was to get objects and put them in your inventory? You only had to point the mouse at it, grab it by holding down the mouse button, and drag it over into the inventory window. Dropping objects from your inventory back to the current location was just as easy.

However, accomplishing this apparently simple task required creating some specialized Visionary routines for my game. I placed all these routines in a separate subroutine file called **GetDrop.SUB**, and this chapter will examine that file. With a few modifications, you should be able to use these routines in your own game to make the moving of objects fast and easy.

Get and Drop Subroutines

The first two subroutines in the **GetDrop.SUB** file are very simple ones. They are called whenever the player tries to get or drop an object by using keyboard input. In this way, when the player types "get the ladder", the game will provide a reminder to use the mouse to move the picture of the object to the inventory window. It is important to note that I could have allowed the player to get and drop objects by using text input, but chose not to. If you want to permit text commands for these operations in your games, it is a simple matter to grab the object and then call a subroutine which will remove the object from the location window and display it in the inventory window.

In fact, there's nothing that says you must even *have* an inventory window or a location window. Remember, with Visionary you are in control of the game, and you can make it play any way you want.

GetObject

Starting in line 30, you will find the **GetObject** subroutine. This is a major subroutine. It's large, and takes care of a variety of tasks. This is the subroutine that is called when the player clicks the mouse button anywhere within the scrollbar window on the side of the location window.

The routine checks to see which object was clicked on, if any. It then allows that object to be moved around on the screen, following along after the mouse pointer. And when the mouse button is finally released, the routine decides whether the object should be placed in the inventory window or back in the scrollbar window. The routine will give a description of the object if the player was not attempting to move the object, but was rather just clicking on it for a description.

And finally, the routine takes care of some general housekeeping matters such as incrementing the timer and checking to see if the cannibals have arrived during this move.

When the Player Enters a Room

Before we start to examine the **GetObject** routine more closely, it will be necessary to get a general understanding of what happens when the player enters a room.

Let's say the player starts the game and moves east to the other end of the beach by the canoe. The game not only displays the location scene for the beach, but also displays any objects at that location. In order to display any such objects, the game runs through a loop that checks for the presence of each movable object. As they are found, their pictures are copied to a long white strip in memory, on screen 23. This screen is created in memory and is not loaded from a disk file. And the screen is never seen in its entirety, since it is long enough to allow all nineteen movable objects to be displayed one below the other.

However, the scrollbar window shows a section of screen 23 that can hold the pictures of five objects. Should there be more than five objects in the room, the scrollbar window can display them all by copying different parts of screen 23 to screen 0, the screen that the player sees. The building of screen 23 to hold all the movable objects, and the displaying of part of screen 23 in the scrollbar window, is all done nearly instantaneously as part of the routine that redraws the screen. One of the things that **GetObject** must do is to determine which of the 19 possible objects that could be on screen 23 has been selected by the player's mouse click.

Arrays

I've also used several arrays in *Cannibal*. One array is used to keep track of the objects that are currently drawn on screen 23. Remember that I can refer to any of the movable objects by number as well as by name. Object 1 is the ladder, object 2 is the bottle, and so on.

Having object numbers in an array makes them easy to retrieve

Each time a new room is entered by the player and screen 23 is freshly drawn showing all the objects present, I write the object numbers to an array so that I know which objects are on screen 23, and in what order. This is necessary because I allow the player to slide the scrollbar window up or down to show any of the objects in the room, and then allow the selection of any object displayed, regardless of its position within the scrollbar window. When the player clicks on somewhere in the scrollbar window, I need to know what object number was selected in order to make the correct picture follow the mouse pointer around on the screen. By placing the object numbers for the objects in an array as they are being drawn on screen 23, I have the important information saved in a manner that can be easily retrieved when I need it.

I created a second array to keep track of which locations within the inventory window are empty and which are full. There are two reasons for doing this. First, when the player tries to move an object into the inventory window, I need to know where the picture of the object can be placed. By using a simple 3x2 array, I can check to see if a particular spot is empty. If the value of the array for that spot in the inventory window is 0, I know it is empty and I can place the object picture there.

The second reason for using an array for this task is that when the player tries to drop an object, by moving it from the inventory window to the location window, I need to know what object was clicked on. To do that, I simply check the array for that spot. It is 0, I know the player clicked on an empty spot. Otherwise, the object number is in that array element, and I know which object the player wants to move. The second array, then, is a simple array that is three across and two down, just like the inventory window.

Pixel Arrays

As pointed out previously, Visionary does not directly support data structures like arrays. However, it is quite easy to create arrays using the colored pixels on any hidden graphic screen. I can check the color of any pixel, and change it if I wish. Hence, if I use pixels to create an array, I can read the values from the array by using the **PIXEL** command, and write the values to the array by using the **RECTANGLE** command.

Using this method of creating arrays creates extremely compact data storage. In my *Cannibal* game, I only needed a small unused space in the upper left corner of graphic screen 2. This is the screen that contains the pictures of the buttons and the movable objects. I use a small amount of space, 3 pixels long and 2 pixels wide, to create my inventory array. And I use a space below that, that is 19 pixels long and only 1 pixel wide, to create my scrollbar array.

As we look at the **GetObject** subroutine, we will see how the arrays and the long scrollbar on screen 23 fit together to allow objects to be moved from the scrollbar window to the inventory window.

The GetObject Action

The **GetObject** subroutine is called when the player clicks anywhere within the scrollbar window. In the first three lines from 32-34, I find which position within the scrollbar window was chosen, which position on screen 23 corresponds to that, and what object number lies at that position.

In line 32, I define a variable **Pic** which will be 0 if player clicks on the top position within the scrollbar window, on down to 4 if the player clicks on the bottom position. This is done by checking the Y coor-

dinate of the mouse pointer, subtracting 21 since the window is down 21 pixels from the top of the screen, and dividing by 18 since each object is 17 pixels long with one extra pixel between objects. Since Visionary variables are integer variables, I don't have to worry about any decimal remainders after dividing by 18. I know they will be dropped off.

In line 33, I find the position on screen 23 that corresponds to the position on the scrollbar window. Remember that if more than five objects are in the room, the player is allowed to scroll the scrollbar window to see the rest. That doesn't mean, for example, that just because the player clicks on position 0, he is also clicking on position 0 on screen 23. The scrollbar window may have been moved downward, and the actual position could be something else.

A special variable keeps track of scrollbar position

To keep track of such possibilities, I created a variable **SBPosition** which always contains the current scrollbar window position. For example, if the variable is 0, the scrollbar window is at the top of screen 23. If the variable is 1, the scrollbar window has moved down one object, and shows the next five objects on screen 23. I use this variable **SBPosition** in line 33 by adding it to the variable **Pic** to find the exact position on screen 23 that the player has selected, and store it in the variable **ChosenPic**.

But adding the two variables together, I can expect **ChosenPic** to have a value anywhere from 0 (if the player picks the top picture and the scrollbar window is moved all the way to the top) on down to 18 (if the current room contains all 19 of the movable objects and the player moves the scrollbar window all the way to the bottom and then clicks on the bottom picture).

In line 34, I look at the array and see what object number is stored in the position that player chose. Using the PIXEL command to find the color of the pixel, I can find the value of the array element.

Notice that I must specify screen 2, the X coordinate of **ChosenPic**, and the Y coordinate of 2. This is because I chose to create my array as a series of colored pixels that run from left to right, that is 2 pixels down from the top of screen 2. Also notice that I must specify the name of a variable into which the color value of the pixel is to be read. I chose the variable name **ObjNum** because the values stored in the array are the object number.

At this point in the routine, the player has clicked somewhere in the scrollbar window, and I now know the number of the object selected. Again, **ObjNum** could contain any number from 1-19. If it is 1, I know the player has clicked on the picture of the ladder. If it is 2, I know the player chose the bottle, and so on.

In line 36, I check to see if the player clicked on a picture instead of a blank spot. As I manipulate the objects and arrays, I am careful to keep all empty spots set to 0 in the array. In this way, I can check the

variable **ObjNum** and see if the player clicked on an object in the scrollbar window, or clicked on an empty position. In line 36, I check the value of the variable, and if I find it is 0 the program will skip the rest of the subroutine and exit at line 136. If however, the value is not 0, I start preparing to move the picture of the object around the screen, following after the mouse pointer.

Moving the Object Graphic

Let's take a general look at how the picture of the chosen object is moved about the screen. Understanding the general idea of how it works will make it easier to explain the mechanics of actually doing it.

As long as the mouse button is held down, I want the picture of the object to follow the mouse pointer around the screen. In this way, the player can drag the picture anywhere on the screen, before letting go of it by releasing the button. That means I need to check to see where the mouse currently is pointing, and draw the object picture at that position. Then I'll loop back and check to see where the mouse is pointing again, and draw the picture there.

The only problem with doing this is that it will keep drawing pictures of the object on the screen without erasing them. If the player slides the object to the left, it will leave a whole line of duplicate objects pictured on the screen. To prevent this from happening, I need to memorize the graphics in the background, so that they can be restored as the object is moved around.

It is not necessary to memorize the entire graphic screen. Only the section that the object overlaps need be remembered. In this case, since all my objects fit in a 15-by-17 pixel rectangle, only a box of this size needs to be memorized. The 15x17 rectangle is copied to a hidden area in memory, and then copied back again after the object has moved on.

So in general, the loop will find out where the mouse points, copy the screen section to a hidden buffer, and copy the picture of the object to where the mouse points. Then when the mouse pointer moves, the routine will copy the background from the hidden buffer back to the screen to erase the picture of the object, and start the loop over again by copying a new section of the screen to the buffer and placing the object picture at the mouse pointer's location. This will continue until the mouse button is released.

Because the picture of the object will not stay on the screen the whole time, it will tend to flicker. This is because the picture is drawn in one spot, the erased and moved to another spot. And after it is erased, but before it is redrawn, the computer will be copying the old background to the screen, and copying the new background to the buffer. During this brief time, the object will not be shown on the screen at all. That's

why the object will appear to flicker very rapidly while it is being moved.

Reduce flicker as objects move by only redrawing them when required

To keep it from flickering while it is held stationary on the screen, I have saved the old X and Y coordinates of the mouse pointer. These are compared with the new X and Y coordinates of the mouse pointer, and if they are the same, the redrawing of the background and object are skipped. This means that when the player temporarily stops sliding the object around on the screen, it reassumes its normally solid look. I did this because I thought it looked nicer that way. When you create your own games, it is certainly not necessary if you wish to save a bit of space and don't mind the constant flicker.

Lines 38-40 start by placing a white rectangle in the hidden buffer mentioned above. The hidden buffer is in the lower right corner of screen 2, right after the pictures of all the objects. By placing a white rectangle here, I make sure that when the player picks up an object by grabbing the picture, a white area will be left behind in the scrollbar.

In lines 42-46 I save the coordinates from where the picture is being dragged. These are the coordinates in the scrollbar where the picture originates. Lines 42 and 43 save the X and Y coordinates in variables that will change as the mouse is moved about the screen. Line 44 saves the old Y coordinate in a variable that will not be changed as the mouse moves about, in case it is necessary to place the object back in the scrollbar window when the player releases it. Lines 45 and 46 check the current mouse coordinates and subtract a little from them so that the mouse pointer will point to the center of the object rather than the corner.

Lines 50-62 contain the loop that was described above. This loop continues as long as the mouse button is held down, and keeps memorizing the background and drawing the picture of the object wherever the mouse is currently pointing. Line 51 checks to see if either the X or Y coordinate of the mouse pointer has changed. If not, the whole drawing routine is skipped and the loop is started over again. Otherwise, the drawing routine is executed.

The drawing routine for the picture of the object is just as described in general terms above. In lines 52 and 53, the current mouse coordinates are stored, after adjusting them to keep the pointer in the middle of the picture rather than the corner. In lines 54 and 55, the old background is restored from the hidden buffer on screen 2 to the game screen 0 at the old mouse coordinates.

Remember that the first time this routine is executed, the white rectangle created in line 40 will be the background copied to the old coordinates in the scrollbar window. Then a rectangular 15x17 pixel piece of the background is copied from screen 0 to the hidden buffer. In this way, when the object has moved past this spot, the background can be properly restored.

Line 56 actually copies the picture of the object to the screen. Since each object can be referred to by number, and since they were drawn all nicely lined up in my **buttons.pic** file, it's easy to now move the right one to the screen. Line 56 takes the object number, as stored in the variable **ObjNum** and copies it from screen 2 where **buttons.pic** was loaded, to screen 0 at the current mouse coordinates. Then in lines 57 and 58, the old mouse coordinates are updated and the loop starts all over again. As you can see, once the concept has been explained, the whole routine is really quite simple.

Releasing the Object

Now let's see what happens when the player lets up the mouse button, and the object is released.

Generally speaking, I check for four different possibilities. If the player intended not to move the object, but to simply click on it for a description, I check to see if the object was not moved more than a few pixels, and in this case, I give the player the object's description.

If the object was moved more than a few pixels, I know the player was trying to do something else. If the mouse was pointing in the inventory window when the button was released, I either place the object in that window, or if it is full I place it back in the scrollbar window. And if the mouse was not in the inventory window when the button was released, again I move the picture of the object back to the scrollbar window. In any case, before doing one of these four things, I must erase the picture of the object one last time, before moving it to the exact location I want it in. This is done in lines 64-66.

Click or Drag?

I first will consider the possibility that the player clicked on the object only in order to examine it. In this case, the object would not actually be moved at all, or at least very little. In lines 68-71 I subtract the mouse coordinates when the object was released, from the original coordinates of the object. These amounts are stored in the variables **x1** and **y1** and tell how far, in pixels, the object was moved.

Sometimes the object will move slightly when the player clicks to examine it

Depending on whether the player moved the object to the left or the right, it is possible these values could be negative as well as positive. To make it easier to compare the values, I found the absolute values of the two variables in lines 69 and 71. These lines may look a bit strange, but they will effectively strip off the negative sign from any negative number. Since Visionary does not have a command to determine the absolute value, I used the short routine you see here.

In line 73, I check to see if the change in movement in either direction is less than 8. If it is, I assume the player was attempting to examine the object, not move it. An alternative to using the absolute value routines at line 69 and 71 would be to write line 73 as follows:

```
IF  X1 < -8 and X1 > 8  and  Y1 < -8  and  Y1
> 8   THEN
```

It may be simpler to understand, and if so you may prefer to use it rather than do absolute values. Both accomplish the same task but in different ways. The choice is yours.

If the movement was less than 8 pixels, lines 74-77 are executed. Line 74 recopies the picture of the object back to the correct spot in the scrollbar window, just in case it was moved slightly. Line 75 places the name of the object in a string variable **$temp.** Remember that all this time, I have been referring to the object by number, not by name. At this point, I want to send a command to the main loop that will be ghosted. This means I have to refer to the object by name, not by number.

Line 75 uses Visionary's **ObjName** command to make this possible. Line 76 sets the text string variable to "examine the bottle" or "examine the ladder" or whatever the **$temp** is set to. Finally line 77 sets the **ButtonUsed** variable to 2, in order to signal to the main loop that a button has been clicked, and a command needs to be echoed to the text window and then ghosted to Visionary. Setting the variable to 2 also lets **MainLoop** know that there is no picture of a button that needs to be drawn either pressed inward or popping back outward.

Moving Objects to Inventory

In line 78, I check to see if the object was released inside the inventory window. If the most recent mouse coordinates indicate the player wanted to add the object to the inventory, I need to see if there is room or not. If there is, I need to find an empty spot in the window where I can draw the picture of the object. Otherwise, I need to draw the picture of the object back in the scrollbar window.

Line 79 checks the **ITEMS** variable to see if the inventory limit has been exceeded. If it has, the program skips down to line 127 where the picture is copied from screen 2 back to the scrollbar window section of screen 0. If there is room in the inventory, the game starts executing the next rather large routine.

Getting some objects may change other game options

The routine that actually places the picture of the object in the inventory window runs from lines 80-126 and is in basically three sections. The first part is to find a blank spot in the inventory window. The second part is to draw the object there. And the third part is to adjust the directions the player can move, in the event that the object picked up was the ladder. Remember that picking up the ladder changes the possible directions. When the ladder is in the meadow, for example, the player can go up to the roof of the shack. When the player picks up the ladder, it can no longer be climbed to get up onto the shack roof. Let's examine each of these three parts.

The lines from 80-95 loop through the 3x2 array and find the first available empty inventory spot. Since this is a two dimensional array, two nested loops are used. Line 84 actually reads the array element and stores the value in the variable **Slot**. Line 85 checks to see if the array element is 0, and if so saves the variables in **X** and **Y**, then sets the loop variables to their maximum in order to immediately exit the loop.

Once an empty spot in the inventory window is found, line 96 actually does the copy of the picture from screen 2 to screen 0. Notice the use of the variables in this line. Knowing the object number saved in the variable **ObjNum** makes finding the coordinates for the object on screen 2 a simple mathematical task.

Knowing the X and Y values for the empty inventory spot, the exact coordinates on screen 0 for placement of the picture in the inventory window can be easily computed. The X coordinate is multiplied by 16, since each object takes up 16 pixels in width (15 for the object and 1 blank space). To this, 266 is added since the inventory window starts 266 pixels from the left edge of the screen. Similarly the Y coordinate is multiplied by 18, since each object is 17 pixels long, plus one for a blank line. And 17 is added to the result, since the inventory window is 17 pixels down from the top of the screen.

Once the picture has been drawn in the inventory window, the 3x2 inventory array needs to be updated. Lines 98 and 99 set the mode and the value of the array element. Line 100 writes the value to the array by drawing a one pixel rectangle on screen 2. And line 101 places the object in the player's internal Visionary inventory, so that the **items** variable will be kept current.

In line 102, I check to see if the ladder was the object just placed in the inventory window. Remember that in certain rooms, such an action will affect the directions the player can travel. A series of conditional IF-THEN statements occur between lines 103-118 which check the room number, modify the directions if needed, and call the subroutine to redraw the screen.

It's necessary to redraw the screen because once the ladder is picked up and placed into the inventory window, it should no longer appear propped up against the shack, the tree, the boulder, or the cave entrance. The **ReDrawScreen** subroutine is one that we have not examined yet. It takes care of redrawing the location scene, the scrollbar window, as well as the status of the compass buttons.

Updating the Scrollbar

The next thing is to update the scrollbar array, to take into account the fact that the object has been removed from the room and should no longer be placed in the scrollbar. Line 120 sets up the mode for drawing, and line 121 uses a **COPY** command to slide the whole array to the left one pixel.

The variable **MaxMov** in line 121 is used to keep track of the maximum number of movable objects that exist in the game. In this game, it was defined in the .ADV file to be 19, since there are 19 movable objects. I use it in the COPY command to make sure that enough pixels are moved to account for any number of objects regardless of what room the player is in, and regardless of how many objects are in that room.

After updating the scrollbar array, the scrollbar itself must be updated. Line 123 shows how Visionary's COPY command is used to move the pictures of the movable objects on screen 23 up, in order to eliminate the object just placed into the inventory window.

Notice that by multiplying the variable **ChosenPic** by 18 and then adding 18, we will be starting with the object just below the one chosen to be moved. I continue down to the very bottom of the long thin screen 23, in order to assured that there will be a white spot at the end, and not two pictures of the last object. This section is copied onto itself, so that the chosen picture is covered up. In this way, it disappears from the scrollbar, and the other objects move up into the void.

Line 124 decrements the variable **ObjTotal** which keeps track of the total number of objects in the current room. Then the subroutine **DisplaySB** is called which copies the correct part of the scrollbar on screen 23 to the scrollbar window on screen 0. Since this routine is called from several places in my program, I decided to make it a subroutine to avoid duplicating large sections of code in several areas of the source code.

After updating the visual display, the subroutine **CannibalsArrive** is called in line 126, in order to increment the timer and properly check to see if the cannibals have arrived and if the player has lost the game yet. This subroutine is also called when an object is moved from the inventory window back to the scrollbar window, and also when the player makes other moves which are ghosted to Visionary and which are caught by the NPC file.

Line 128 shows the COPY command that is used if the program was unable to place the object in the inventory window because it was already full. Line 132 shows the same line of code, this time used to place the picture of the object back in the scrollbar window if it was mouse button was released when the object was outside the inventory window. Since the same piece of source code was used more than once, I could have make it into a subroutine. I chose not to, since it was only a single line and it made the program easier to follow this way.

That ends the **GetObject** subroutine to move an object from the location window's scrollbar window to the inventory window. Remember the whole thing started in the **MainLoop** file. After checking for keypresses, the routine checked to see if any of the click zones defined

in the **StartUp.SUB** file had been clicked in. If the player clicked in the scrollbar window, this **GetObject** subroutine is called.

DropObject

Next we will look at the **DropObject** subroutine. It is very similar to the one above, but works backwards. This is the subroutine that is called if the player clicks inside the inventory window.

This subroutine checks to see what object has been clicked on, and then allows the player to grab that object by holding down the mouse button. If the object is moved only minimally, the object description is given to the player. If the object is moved outside the inventory window, it is placed in the scrollbar window, otherwise it is placed back in the inventory window where it originated. We'll look at this subroutine next, but in less detail.

Since the **DropObject** subroutine is similar to the **GetObject** subroutine, we won't need to spend so much time explaining the fine details. As before, I started by finding the number of the object clicked upon. This time, I checked the mouse coordinates and did some subtracting and dividing to give me the inventory array coordinates. In line 146, I then read the value of the array element into the variable **ObjNum** using Visionary's PIXEL command. If this value is 0, I know the player has clicked on an empty spot, and the program jumps from line 148 to line 244 and skips the entire routine.

The initial white background is saved in lines 150-152. The old and new mouse coordinates are defined in lines 154-159 in preparation to allow the object to move about on the mouse pointer. Notice in lines 156 and 157 both original X and Y coordinates are saved in temporary variables so that the object can be put back in the inventory window in the exact spot from which it came, in the event it is necessary.

Lines 163-175 show the loop which draws the picture of the object on the screen, following the mouse pointer wherever it does. It restores the previous background, saves the background at the new location of the mouse, and then draws the object there.

If the object has been moved since the last time through the loop, the game continued to restore the background, save the new background, and draw the object. This continues as long as the mouse button is held down. When the player's finger is lifted from the mouse button, I then have to decide whether to place the object back in the inventory window or place it in the scrollbar window.

Lines 177-179 restore the background to normal, one last time before placing the picture of the object in its final resting place. If the player has moved the object anywhere over into the location window, line 181 will start the section of code to place it in the scrollbar window, otherwise the code jumps down to line 211 where the object is put back in the inventory window.

If the object is to be placed in the scrollbar window, there are several things to be done. First, I free up the inventory array where the object used to be. This is done in lines 182-184 by drawing a black pixel at the previously computed X and Y coordinates in the inventory array.

Then the object is officially dropped from the player's internal Visionary inventory, in order to keep the variable ITEMS accurate. If the object moved back to the room location is the ladder, I have to check to see if this is one of the rooms where the ladder creates a new direction available for the player to take. Lines 178-207 show the places where this can happen. In each case, the directions have to be changed, the room must be linked with the new direction, and the screen needs to be redrawn.

After the object has been officially moved from the inventory window, a subroutine called **AddObject** is called in line 209. This subroutine will be examined in detail later in this chapter. For now, it need only be pointed out that this subroutine takes care of adjusting the scrollbar array, adjusting the scrollbar on screen 23, and displaying the proper part of the scrollbar to the scrollbar window on screen 0. In line 210, the **CannibalsArrive** subroutine is called again, which increments the timer and checks to see if the player has lost yet.

Starting at line 211, you will see the section of source code that is executed if the object was not dragged into the location window. In that case, I place it back in the inventory window. This is done in line 212, by using the COPY command to copy the picture of the object from screen 2 to the old coordinates (temporarily saved in the **temp1** and **temp2** variables) in the inventory window on screen 0.

Lines 213-216 check the distance that the object was moved, making sure it is not a negative number. Line 217 checks to see if the object was moved less than 8 pixels. If that was the case, the variable **$temp** is defined to be the name of the object which I have only been referring to by number. This name is added to the text string in line 219, and line 220 will force the main loop to ghost the previously defined text string. In this way, the player can "examine the hammer" by clicking on the picture in the inventory window.

We have now concluded our examination of the **GetObject** and **DropObject** subroutines. These are the main parts of getting and dropping any object by using the mouse. There are still some smaller support routines which need explaining, and these will be covered next.

Refreshing Windows after a Move

The next few routines take of such tasks as copying the proper part of the scrollbar on screen 23 into the scrollbar window on screen 0. They will also allow easy adding of objects to the scrollbar, and removing objects from the inventory window. Routines will also create the arrows on the ends of the scrollbar window that only appear when there

are more than 5 objects in the current room, and the scrollbar window can be scrolled. Let's start with the **ReDrawScrollBar** subroutine.

ReDrawScrollBar

The purpose of the **ReDrawScrollBar** routine is to check the movable objects when the player enters a new room. It runs through all the movable objects, and places any of those present in the scrollbar on screen 23. It also creates the scrollbar array which contains the object numbers that correspond to the pictures shown in the scrollbar window. Then it displays the objects in the scrollbar window on screen 0.

The first thing to do is to blank out the pictures on screen 23. This is the long thin screen where all the objects present in the current room will be drawn one above the other. Lines 232-234 draw a long white rectangle on screen 23. Two variables are then initialized in lines 235 and 236. The variable **ObjNum** is set to 1, so that the search for movable objects will start with the ladder (object number 1). The variable **ObjTotal** is set to 0, to indicate that there are currently no objects found in the room.

The loop from lines 239-251 will be executed 19 times, and will check each object to see if it is in the current room. Remember that the objects will be referred to by number, not by name. That was one of the important reasons in giving the objects such strange names. The two digit prefix forces Visionary to alphabetize the objects in the order indicated by the numbers.

If the object is in the room, line 242 copies the picture from screen 2 to the scrollbar in screen 23. Notice the COPY command uses the variables **ObjNum** and **ObjTotal** to find where the picture is on screen 2, and where it should be copied to on screen 23. Then the object number is written to the scrollbar array in line 245. Line 243 sets the mode to draw, and line 244 sets the color to be the object number.

Finally, the **ObjTotal** variable in incremented, to indicate the addition of an object to the room. Lines 248-249 clear the rest of the array, and line 250 increments **ObjNum** for the next pass through the loop. At the end of the subroutine, another subroutine is called in line 253 which does the actual displaying of the scrollbar into the scrollbar window.

DisplaySB

The subroutine **DisplaySB** takes care of actually copying the correct part of the scrollbar into the scrollbar window, and activates the arrow buttons if they should exist. The reason that I made this a separate subroutine is so I could call it when an object was dropped or when a new room was entered.

Lines 261-266 make sure the variable **SBPosition** is correctly adjusted. Remember that this variable keeps track of which position on screen 23's scrollbar is currently being shown at the top of the scrollbar win-

dow. These lines make sure that as many of the objects are seen as possible, to prevent objects being off the scrollbar window to the top while white unused spaces are shown at the bottom (as could occur when taking objects out of the scrollbar window).

Another purpose of these lines is to ensure the scrollbar window is never scrolled upwards past 0. Then the subroutine **DrawArrows** is called to draw the arrows at the top or bottom of the scrollbar window, if necessary. Finally, lines 270-271 copy the correct portion of screen 23 to the scrollbar window on screen 0.

AddObject

The **AddObject** subroutine is used when an object is added to the scrollbar window. This can occur when an object is moved from the inventory window back to the location window, and can also occur when the player digs and finds something.

In either case, the object is always placed at the start of the list. Everything else is moved down one to make room, and the object is inserted at the beginning. Then the top of the scrollbar window is displayed to show the newly-added object. Lines 279-282 show how the scrollbar array is moved down one, and the new object is added to the front of the list.

The variable **ObjTotal** is increased to keep track of the total number of objects in the current room. Lines 285-290 take care of sliding the objects down on screen 23 and adding the new object to the top of the list. Notice that before drawing the object at the top of the scrollbar in line 290, I have to erase the top position with a white rectangle in line 288.

Before displaying the new object in the scrollbar window, the variable **SBPosition** must be set to 0. The **DrawArrows** subroutine is called to draw the arrows if necessary, and then the top of the scrollbar is actually copied to screen 0 in line 279. This is a short but helpful subroutine.

The Scrollbar Arrow Subroutines

The **ClickUpArrow** subroutine is the one used when the player clicks on the up arrow on the scrollbar window. It slides the objects upward as long as the mouse button is held down, or until the end of the objects is reached. First, in lines 307-308, the picture of the arrow is changed to show it depressed. After a short pause to keep the mouse pointer from temporarily freezing up, the scrollbar window is moved up one full picture.

Rather than move it in a single jerk, I decided to show it smoothly moving, pixel by pixel. So instead of doing a single **copy**, I did 18 copies, each one moving the scrollbar up just one pixel. You can see

how this was done in the loop from lines 311-314. It takes a little extra programming, but looks much nicer.

As soon as the scrollbar has been moved up one full position, the **SBPosition** variable is updated in line 315. Line 316 calls a subroutine to see if the up and down arrows should still be shown, or if either should be blanked and deactivated. This updates the current status of the two variables **UpArrowActive** and **DownArrowActive**.

At the end of the loop, the queue is emptied with the **readbuttons empty** command, in the rare event that extra mouse clicks have taken place during the loop.

Lines 305 and 306 are good examples of some shortcuts you can use in your programming. Shown below are two different ways of writing each line:

```
WHILE LEFTBUTTON              or
WHILE LEFTBUTTON = 1 DO

IF UPARROWACTIVE              or
IF UPARROWACTIVE = 1 THEN
```

A variable value of 1 also means "true"

Since the value 1 means "true", you can use the shorter version. This is useful any time you are checking for a value of 1. When the **DrawArrows** subroutine (to be described shortly) is called, the variable **UpArrowActive** is set to 1 if the up arrow on the scrollbar window can be seen and pressed. The shortcut shown above is a slightly faster way to detect if the arrow is present.

The **ClickDownArrow** subroutine at line 325 is nearly identical to the previous **ClickUpArrow** subroutine. It checks to see if the variable **DownArrowActive** is set to 1, and if so allows the scrollbar window to scroll downward. Notice that the routine to smoothly scroll the window is in a while loop that goes from 18 down to 0, instead of 0 up to 18. This is because the window will be moving in the opposite direction from the previous subroutine. Other than that, the two subroutines are nearly identical.

The **DrawArrows** subroutine at line 347 checks to see if there are more than 5 objects in the room. If so, it draws arrows at the top or the bottom of the scrollbar window, depending on whether the extra objects are off the top of bottom of the window.

The subroutine also sets two variables **UpArrowActive** and **DownArrowActive** so that the other subroutines will know if the arrows are drawn or not. Lines 349-351 initialize the two variables and set the drawing mode in case the arrows need to be drawn. Line 353 checks to see if there are objects below the scrollbar window, by adding the current scrollbar position in **SBPosition** to the size of the scrollbar window.

The size of the scrollbar window is always 5, and is set in the variable **SBSize** in the .ADV file. If this is less than the total number of objects

in the room, I know there are more objects below. In that case, the **DownArrowActive** variable is set to 1 and the picture of the arrow is copied from screen 2 to screen 0. Otherwise, a blank spot is copied to where the arrow should be.

If the scrollbar window is not at position 0, I know in line 360 that more objects appear above. In this case, I set the appropriate variable and draw the appropriate picture of the arrow. This subroutine is short and simple, but needs to be called whenever the object pictures are scrolled or whenever a new room is entered.

DestroyObject

The last subroutine in this file is the **DestroyObject** subroutine. This subroutine will take any object number it is given in the **ObjNum** variable, and will search the inventory window for that object. It will destroy it and make the necessary corrections to the inventory array.

Lines 376-389 show two nested loops which check the top and bottom level of the inventory window. This is done by checking the inventory array, stored on screen 2. Notice it is very similar to the search routine at lines 82-95 in the **GetObject** subroutine discussed earlier in this chapter. The value of the array element is read in line 378 and compared with the object being searched for in line 379. When it is found, the coordinates are saved and the loop is exited by forcing both loop variables to their maximum.

In lines 391-393 the array element is changed to 0, to reflect the fact that the object no longer is in the inventory. Lines 395-397 draw a white rectangle over the picture of the object in the inventory window. Finally the object is officially dropped in line 399, and is placed in the **Unused** room where objects are stored when they are not currently needed.

In the *Cannibal* adventure, this **DestroyObject** subroutine is used three times. When the player eats the candy bar, it must first be in the inventory, and then must be destroyed.

When the player breaks the bottle open, it must also be destroyed using this subroutine. In addition, a piece of paper is created at that point, using the **AddObject** subroutine described above.

The other place where this subroutine is used is when the player puts the shovel handle into the blade and makes a complete shovel. Before making the shovel, the player must be holding both pieces. This subroutine is called twice, once to destroy the handle and then to destroy the blade. Finally, the shovel is placed in the inventory window at the same coordinates as the last destroyed object.

» All of the subroutines examined in this chapter are vital to the mouse control of objects. The beauty of subroutines is that once written, they can be called from anyplace in your game. And they can be used over and over again in future games, saving you appreciable amounts of development time. If you are still having trouble understanding any part of these subroutines, come back to this chapter after you have finished the rest of the book. Perhaps after you have a clearer picture of how things fit together in the whole game, you will have a better feel of how these subroutines are used, and can be used by you.

In the next chapter, I will explain many of the general subroutines that I used in the *Cannibal* game. These will include the print routines that allow text to be printed on a graphics screen, and the redraw screen routines which are called whenever the player enters a new room. Both play an important part in the creation of a graphic adventure game.

Chapter 25: The Cannibal.SUB File

Subroutines play a large part in any adventure. As you will see in this chapter, I used a large variety of subroutines extensively in the creation of *Cannibal*. We'll be looking at them in detail, in the hope that you will find many that you will be able to apply to your own adventures.

CannibalsArrive

The first subroutine in the **Cannibal.SUB** file is the subroutine named **CannibalsArrive**. The first thing the subroutine does is to increment a timer. Visionary has a similar timer that automatically increments the variable "moves" after every move. I chose not to use this one, since it increments after normal moves, and my special methods to permit the mouse to get and drop objects bypass the normal moves. So I found it more convenient to use my own timer.

After incrementing the timer in line 6, I set a variable **TextColor** to red, so that any messages printed about the cannibals would stand out. Then I checked to see if it was time for the cannibals to land in line 8.

After 90 moves, the game will print out a warning message that the cannibals are about to land on the island. Warning messages continue to be printed periodically until the player is captured when the timer is 103, as you can see in line 42 of the source code.

Until then, each of the sections of code is very similar. A linefeed is generated by calling a subroutine, then the message is printed, and finally the **ButtonUsed** variable is set to 9. By doing this, the main loop's input routine is restarted after the message is printed.

When the cannibals actually arrive on the island in line 25, the jungle drums originally heard at the opening of the game are started up again. This is a MED music file that was originally loaded into memory when the opening title was being displayed. When the game started, the music was turned off, but was not removed from memory. In this way it could be used again later. And finally, notice that in line 40 the text color is always set back to blue after any cannibal messages are given.

End-Game Actions

Let's take a look at what happens when the player loses the game. When the timer hits 103 in line 42, the room number is changed to 15 and the subroutine to redraw the screen is called. Scene 15 is a special one that is shows the player sitting in a pot of boiling water. This scene was carefully drawn so that the colors could be cycled to give the appearance of animation. The bubbles, the flames, and the smoke are all

animated. But the colors were carefully chosen to avoid any conflict with the colors used elsewhere on screen 0. When the colors are cycled, **all** colors will cycle. I didn't want the player to see any colors in the buttons to change, or have the mouse pointer changing colors. So it was important to keep certain colors that were used on screen 0 outside the cycling range.

Color cycling requires special handling of screen and button colors

The other colors were then changed using Visionary's **PALETTE** command to match the special palette of colors required to create the animation. Lines 47-49 blank out the scrollbar window, the inventory window and the text window by drawing white rectangles. This is done so that the colors in objects and text will not be seen color cycling. Then the new colors for this scene are set in lines 50-66. And finally, the color cycling is turned on in line 67.

At this point, the animated death scene is showing in the game. The next step is to turn on the sound effects for the death scene. Line 68 disables the music, and returns control of the audio device to digitized sound samples. Line 69 plays the sound of the bubbling water that was loaded when the game began, and has been sitting unused all this time.

Before printing out a closing message, the **CountLines** variable is set to -1. This prevents my text printing routine from printing the usual "Press mouse button to continue" message. The message is printed, telling the player the game is lost, and one last sound sample is loaded into the computer. Line 82 loads a digitized sample of a scream. This is played as loud as possible in line 83.

Notice that the volume is set all the way up, and the command is set to play the scream only once. Because of the time taken by the disk access, the death scene shows along with the sound of the bubbling water, then there is a brief pause before the scream. This was done intentionally to create a pause before the scream, and also to save memory by not requiring the sound sample to be in memory until the end of the game.

Before Exiting

The game is now over, but it has not yet closed down the screen and returned to the operating system. I decided to wait for the click of a mouse button before exiting the game. I first cleared all click zones by calling the subroutine **ClearButtons** in line 84. I did this so that no matter where the next click was, there would be no response. Then in lines 85-86 I wait until the player's finger has been lifted from the button, just in case it is still being held down from a previous move. Lines 87-88 then wait until the player clicks on the mouse button before setting the **MainLoop** variable to 1, which will force the game to exit when it returns to the main loop of the game.

The Print Routine

The next **print** subroutine is the one that I call whenever I want text to be printed on the graphic screen, in the special text window that I created. First I call a subroutine that creates a line feed. This is the subroutine that also checks for more than 6 lines of print, and then pauses so the player can read it. The second thing I do is to call the subroutine that actually prints the text. By keeping these two subroutines separate, I can call them individually in other places, as well as together for general text purposes. The last thing the **print** subroutine does in line 99 is to clear the queue of button clicks, in case any buttons were pressed during the printout of text to the text window. The **print** subroutine is short and simple.

LineFeed

The next subroutine is the **LineFeed** subroutine that is called by the **print** subroutine and is also called from other places. This routine moves up five lines of text in the text window, and makes room for a new line to be printed on the bottom line. If more than five lines have been printed successively, then this routine will pause and wait for the player to read the message and then press the mouse button to continue. Let's look into this subroutine in more detail.

The first thing that the **LineFeed** subroutine does is to copy a large section of the text window to a slightly higher position. Lines 106-107 move the bottom five lines of text upward 9 pixels distance. Each line of text takes 9 pixels, 8 pixels for the font, and a one pixel wide blank line in between lines of text. Lines 108-109 blank out the sixth line by drawing a white rectangle over it. Then the color of the text is set to whatever value **TextColor** was given when the subroutine was called. This is done in preparation of printing out the text, even though this subroutine doesn't actually do any of the printing itself.

The next thing that this subroutine does is to count the number of lines of text that have been printed recently. It increments the count in line 112, by incrementing the **CountLines** variable. Then it checks this variable against **MaxLines** which was set to 5 in the .ADV file. If it's not time to pause, the subroutine is ended. Otherwise, the routine prints a message in line 117 and waits for further input. Notice that line 115 sets the drawing mode to "overlay." This is so that the text will appear properly on the white text window. If it were set to "draw", the text would appear in the same place, but would be set against the black background. Line 116 sets the color to brown, so that the text prompting the player for mouse input will be brown, and contrast with the blue and green text usually seen. Line 117 actually shows how text is written to the graphic screen. The command is TEXT followed the screen number, the X and Y coordinates, and the text to be printed. The prompting message will always be printed at this location.

When I designed this routine, I wanted the game to wait until the player either clicked on the mouse button or pressed a key on the keyboard. The next routine shows how this is done. I first make sure the mouse button is released, by using the while loop in lines 118-119. Then I enter another while loop from lines 121-126. In line 122 I check the status of the mouse button and save it in the variable **temp1.** This variable will be 0 if the mouse button is up, and 1 if the button is down. Next I get a character from the keyboard in line 123 and check it's length in line 124. The length is saved in the variable **temp2.** This variable will be 0 if no key was pressed or will be 1 if some key was pressed. In line 125 I add these two values together, knowing that the sum will only be 0 if nothing was pressed, either mouse or key. Since the sum is saved in the variable **temp** and the loop variable is also **temp**, the loop will continue until either the mouse or a key is pressed to change the value of the variable.

Once the player presses the mouse button or presses a key, the routine goes on to line 127. Here, I change the color to white and draw a white rectangle on the bottom section of the text window to blank out the previously displayed message. Then I change the color in line 129 back to the normal text color, as saved in the **TextColor** variable. The variable keeping track of the lines printed is reset back to 1, and the subroutine is exited.

The **PrintText** subroutine does the actual printing of the text to the graphic screen after the other text has been scrolled upward to make room for it. As you can see, it only takes two lines. Line 138 sets the mode to overlay, to keep the letters showing up properly on the white text window. Line 139 then prints whatever is contained in the text string **$tx** at coordinates (9,192) on the screen. These are the coordinates for the last line of text in the text window. Notice the specific syntax that I use to print the value of the string variable. I enclose the string variable in quotes, and place the "@" symbol in front of it. Without the "@" symbol, this line of code would print out **$tx** to the text window, instead of things like "examine the ladder".

Button Subroutines

The **ClearButtons** subroutine is another short but useful routine. It is called when entering a room, to clear all the click zones assigned to buttons 34-44. These are the zones for the nonmovable objects, and allow the player to click on them for a description. This prevents accidents from happening like the player receiving a description of grass after clicking on the picture of the boulder. Of course buttons can always be redefined when entering a room, but sometimes there are buttons that are not needed in some room. These buttons won't be redefined, and unless cleared will contain the name of the subroutine assigned in the previous room. This subroutine is a simple loop that

uses Visionary's **REMOVE** command to clear the subroutines assigned to the buttons from 34 to 44.

The next eleven subroutines are called when the player clicks on one of the buttons on the screen. Six of them are for the compass buttons, and the other five are for the larger action buttons. They are all similar in content. Let's look at the first one, **GoNorth**, which will serve as an excellent example for these eleven buttons.

GoNorth

The first line in the **GoNorth** subroutine is to define the value of the **offset** variable. It will be either -13 or 13. Remember, as described in a previous chapter, the **N** button will be shown pushed in and then popping back out again. The **offset** variable will tell my main loop routine whether to look 13 pixels above or 13 pixels below the picture of the pushed-in button, to find the pushed-out button. Usually the offset is -13, and the pushed-out button will be 13 pixels above the pushed-in picture. However, if the direction north is ghosted, then the offset is set to 13, and my main loop routine will find the proper picture of the pushed-out button 13 pixels below the pushed-in version. For more information on this design, go back and check the chapter on the **MainLoop.SUB** for additional details.

In line 155, the **offset** variable is set to **GoN**. This variable is either -13 or 13, and will be set by the **ReDrawScreen** subroutine described later in this chapter. Part of that subroutine is go check to see if the player can go north, and if so set the **GoN** to -13, so as to show the highlighted **N** button. If the player can not go north, then it will set the **GoN** to 13, so as to show the ghosted **N** button.

After setting the **offset** variable, three pairs of X and Y coordinates are defined. These are used in the **COPY** command to copy the picture of the pushed-in button to the same position as the current button. Remember that the COPY command needs three sets of coordinates in order to work. It needs the upper left coordinates and lower right coordinates of a rectangle that will be copied from. It also needs the upper left coordinates of the position that will be copied to. After setting the proper coordinates, the text string that will be ghosted to the interpreter is defined. In this case, it is the single letter **N**. Finally the **ButtonUsed** variable is set to 1, to let the main loop know that a command has been given from the mouse, and that a screen button needs to be shown moving in and then back out again.

The other ten button subroutines all follow the same pattern. Only the compass button routines need the **offset** variable, since they are the only ones that can have two different pushed-out pictures. You will notice that the last five button subroutines do not have the **offset** variable assigned. This is because I designed it to always default to -13 unless otherwise set. All of these button subroutines have the six coordinates defined, followed by the string variable defined that will be

ghosted by the main loop, and finally the **ButtonUsed** variable is set to 1, so that the command will be ghosted and the button will be shown moving. The only exceptions are the **Load** and **Save** subroutines, which use different values for the **ButtonUsed** variable so that the main loop will show them pushed-in, but will not show them pushed-out again until after the game position is loaded or saved.

ReDrawScreen

The next subroutine after all the button subroutines is the **ReDrawScreen** subroutine in line 303. This is the subroutine that has been previously mentioned throughout this book. Now it's time to see how this important subroutine works. Let's take an overall look first, before getting down to specifics.

In general, the subroutine loads a disk file that shows the new location, then checks the compass directions to see which way the player can move, then copies the location scene into a special overlay buffer where the ladder can be added if necessary, and then finally displays the scene in the location window and draws the proper compass buttons in their places. Now let's go back and see how all this is done.

The first part of the subroutine from lines 306-316 loads the disk file containing the location scenery. Line 306 checks to see if the player has moved to a new room, so that the new disk file can be loaded. Remember that this subroutine is also called other times than when the player moves to a new room, which means it may not be necessary to load the location scene from disk. It may already be in memory.

This section of the **ReDrawScreen** subroutine only loads the scenery from disk if a new room has been entered. The routine starts by clearing the text screen, in case any disk error causes a requester window to pop up on the text screen. If for example, the disk has been removed, the game will switch to the text screen and show a small window requesting that the player place the disk in the drive. In the event one of these types of requesters is displayed, I don't want any previous text messages to be displayed. Typing "\f" to the text screen prevents it.

The file name is set in the variable "$filename" in line 308. Remember that all the location scenery is in individual files named "Loc1", "Loc2", and so on. Line 308 sets the file name to "loc" with the device name in front of it, and the room number behind it. Depending on the value of the two variables **$device** and **RoomNumber**, the final value of the string variable "$filename" can be things such as "ram:loc12" or "Cannibal/Video:loc6". In line 309, the file is loaded into screen buffer 1, and the subroutine **LoadingError** is called which checks for any error and prints out an error message if necessary. The **LoadingError** subroutine was described earlier in the chapter on the **StartUp.SUB**.

In line 311, the **LastRoomNumber** variable is updated, so that when the **ReDrawScreen** subroutine is next called, I will know what location

scene is currently in memory. The next line starts preparing to show the objects in the scrollbar window. The scrollbar position variable **SBPosition** is set to 0, in order that the top of the scrollbar be displayed first.

The "show screen 0" and "screenmode graphics" commands in line 313 and 314 will force the graphic screen back to the front in the event that a requester window moved the text screen to the front. Then the subroutine is called in line 315 that shows the objects that are in the new room displayed in the scrollbar window. Remember that all this code is only executed when the player actually moves to a new room, not when the screen is redrawn without loading a new scenery file.

Changed Options

The next six sections of code check to see what directions the player can move, and properly sets the offsets so that each compass button will be either highlighted or ghosted. In line 319, the direction for north is checked to see if the player can go that way. If so, the offset variable **GoN** is set to -13, which means that the picture of the pushed-up button will be found 13 pixels above the pushed-in button.

If the player cannot go north, the variable **GoN** is set to 13, indicating that the picture of the pushed-up button will be found 13 pixels below the picture of the pushed-in button. This routine is then repeated for each of the compass buttons, in lines 325-353. At the end of these six routines, all the offsets are set for the compass buttons, but it's still not time to draw anything on the visible game screen yet.

Before copying the location scenery into the location window, I must check to see if the ladder needs to be shown laying against the shack, the tree, or some other place. This will be done in a hidden screen buffer, and the final result will be moved to the visible game screen. The first step is to copy the location scene from screen buffer 1, into which it was loaded, over to screen buffer 2, where the ladder can be overlaid if necessary. Lines 355-357 do the copy, and prepare to do an overlay copy.

Removing Unused Click Zones

Line 359 removes the click zone for the ladder since it will likely be missing from the location scene. If the player is in one of the rooms where the ladder can be seen propped up, the click zone will be defined next. The conditional statement at line 361 jumps down to line 378 if the ladder is not in the room. If the ladder is present, then the room numbers are checked one by one, the larger picture of the ladder is copied from one part of screen 2 onto the location scenery in another part of screen 2, and the click zone is defined. In each room where the ladder is visible, it will be placed at different coordinates, and the click zone will be defined accordingly. You will see five similar

sections of source code in lines 362-377, each doing a copy and defining a click zone for a different room.

The final section of the **ReDrawScreen** subroutine does the actual copies, moving the location scene to the visible screen's location window and drawing the proper six compass buttons in their places. These lines of code can be seen from lines 380-387. The mode is set to draw, then the location window is copied to screen 0 in line 381, and the compass buttons are copied to screen 0 in lines 382-387.

Notice that when copying the compass buttons, the proper offset variables are added to the Y coordinates for the source of the copy, so that the highlighted picture of the button or the ghosted picture of the button will be shown, whichever is accurate.

Common-Message Subroutines

Some common messages are contained in the next five subroutines. These are messages that are used so frequently that it was easier for me to place them in subroutines to be called when needed. They are used in nearly every movable object file, and as you will see in the source code from lines 393-424 simply use the print subroutine to display messages to the text window. The messages are "it already is", "you can't do that", "you don't have it", "you already have it" and "OK."

The next group of subroutines from line 428 all the way down to line 585 are all called when nonmovable objects are clicked on. When the player wants to examine part of the location scene, the mouse button can be clicked when the pointer is on the object in question. The click zones defined in each room file send the game to these subroutines. The subroutines are simple in the extreme.

The exact text string is set in the variable **$tx** so that it can be echoed to the screen and ghosted to the parser, just as if the player had typed the words instead. Then the **ButtonUsed** variable is set to 2, to notify the main loop that a line of text waits to be input. The value of 2 also tells the main loop that no button on the screen needs to be shown being pushed in or popping back up.

Remember that when the main loop checks for keypresses, it checks once each time through the loop for mouse clicks. If a defined zone has been clicked in, the program jumps to one of these subroutines and then returns to the main loop, in order to process the information.

Breaking Objects

When the player tries to open the bottle, it won't open. Chances are the next action will be an attempt to break it, in which case this next

subroutine is called. The **BreakBottle** subroutine at line 589 permits the player six different ways to break the bottle.

The player needs to be holding something hard in order to break the bottle – the bottle will be broken only if the inventory contains the coconut, the shovel handle, the shovel blade, the complete shovel, the hammer, or the chisel. If the player holds even one of these objects, the subroutine **SmashBottle** is called which does the actual breaking. Otherwise lines 603-606 tell the player "you have nothing hard enough" to break the glass bottle. Now let's look at the next subroutine and see how the bottle breaks.

The **SmashBottle** subroutine at line 612 calls a subroutine to destroy the bottle, then calls a subroutine to add the paper found inside the bottle to the room, and then prints a message to the text window. Line 613 sets the **ObjNum** variable so that the call to the subroutine **DestroyObject** will know which object to search the inventory window and the inventory array for. The **DestroyObject** subroutine was examined in the chapter on **GetDrop.SUB**.

When the bottle is broken, a paper bearing a message will be found inside. Line 615 places the paper in the current room. Line 616 sets the value of **ObjNum** in preparation to call the next subroutine. Line 617 calls the **AddObject** subroutine which places the picture of the paper at the top of the scrollbar, and also adjusts the scrollbar array on screen 2.

Even a graphics adventure sometimes needs text to explain things that happen.

Once the picture of the bottle has been destroyed and the picture of the paper has been displayed, lines 618-623 print a message in the text window explaining to the player what has happened. Without this message, the player wouldn't know what happened to the bottle that was in the inventory window, and why a piece of paper suddenly appeared in the scrollbar window.

NoSwim

The **NoSwim** subroutine at line 628 is called whenever the player tries to swim out from either section of the beach. Remember there were two special room files in the **Cannibal.ROOMS** file that are briefly visited when the player tries to swim in the ocean. Both rooms set an attribute and then send the player back to the beach. Upon returning to the beach, this subroutine is called and the attribute is cleared. The reason for the attribute is to keep the room description from being displayed again when being forced back to the beach.

If the player should attempt to sit down in the rowboat or the canoe, nothing really happens except that a message is printed. No attributes or variables are set, when the player sits down. They aren't needed in the game, since the view of the location and everything else remains the same whether the player is standing or sitting. The only thing that needs to be done is to recognize that the player has asked to sit, and to

acknowledge that the command has succeeded. The two subroutines at lines 641 and 650 are called when the player tries to sit in the rowboat and canoe. As you can see, they simply say in different ways that the player is sitting.

The Dig Subroutine

The **dig** subroutine starts at line 657 and is called whenever the player clicks on the **DIG** button or types the command from the keyboard. The subroutine is a large one, because the player must be permitted to dig (or at least attempt to dig) in every location.

There is a variable **dig** tset in each room file, that tells the game about the ability to dig in the room. If the **dig** variable is 3, then the player can't dig in the room under any circumstances, and the message doesn't really say why. This is used if the player should try to dig in illogical places like inside the wood shack, on the roof of the shack, or in the top of the tree.

If the variable **dig** is set to 2, the player can't dig again, but is told instead the reason is that the rock is too hard. This is used in the two cave locations. If **dig** is 1, then the player can dig only if the complete shovel exists in the inventory. This applies to the meadow, the ground by the tree and the ground by the boulder. Finally, if **dig** is 0, then the player can dig without any tool. These are places where the player can dig in the sand by hand, such as the sand dunes and the two beach locations.

The first six lines of this subroutine deny the player the ability to dig under any circumstances. They are called when the player either has tried to dig in rock, or somewhere more bizarre. But starting at line 665, the player can dig if the shovel is in the inventory. Just having the shovel handle or the shovel blade isn't good enough. Even having both parts isn't enough—the player must put them together to make the complete shovel before digging in the active areas.

If the player is in the meadow by the shack, digging uncovers the radio battery. Digging by the boulder finds the flare gun. By the palm tree, digging reveals the chisel. The lines from 667-700 make up the source code for these three areas. The code is divided into three similar routines, one for each location.

The lines from 667-677 are executed when the player digs in the meadow. If the **found** attribute is set for the battery, then I know the player dug here previously and already found the battery. In that event, line 669 calls a subroutine that tells the player nothing further is found when digging. If the **found** attribute is not set for the battery, then a message is printed telling the player what has been found, the object technically known as "04battery" is placed in the room, the **found** attribute is set so that the battery won't be found a second time,

and the **AddObject** subroutine is called to show the picture of the battery in the scrollbar window.

A similar routine is then executed from lines 678-688 which permits the player to find the flare gun by the boulder. And in lines 689-700 another routine is executed which checks to see if the player finds the chisel by the palm tree. Line 702 is the default that is executed if the player does not carry the shovel.

Starting in line 705 the program checks to see if the player is standing in a sandy area and can dig without any tools. Again, there are three similar routines for the three sandy locations. The routine at line 706 creates the matches when the player digs at the east end of the beach, in the same manner as described above. The section starting at line 717 finds the driftwood buried in the sand by the rowboat. And the part of the subroutine starting at line 728 makes the shovel handle appear when the player digs in the sand dunes. All three of these routines work in the same way as described above.

The wet matches are a misdirection used to make the game solution a bit trickier to find.

The next subroutine is called **DigNothing** and is called from several places within the **dig** subroutine. It simply prints a message that nothing is found when digging. The subroutine after that is even shorter. It is called whenever the player attempts to burn any of the objects, and prints a reminder that nothing can be burned with wet matches. The matches, you will remember, always remain wet in the adventure. They never dry out, even though the player is not supposed to know that, and will hopefully try to dry and use them.

BreakCoconut

The **BreakCoconut** subroutine is called in case the player tries to open the coconut with the hammer or chisel. It's purpose it to simply deny the player the ability to break the coconut. But notice that it doesn't simply say "You can't"—it tells the player **why** the coconut can't be broken. It's important to remember in designing your own adventure games that you must always tell the player why something can't be done. It not only reduces the player's frustration but also can be used to give clues and hints for what you want the player to try.

EatCandy

The next to the last subroutine in the **Cannibal.SUB** file is the **EatCandy** subroutine at line 773. This is called from the movable object file and the special vocabulary action file. When called, it destroys the candy bar, gives the player temporary energy and prints a message to the screen. Line 774 checks to make sure the player actually has the candy bar in the inventory. This is important, because the subroutine can also be called if the candy bar is simply in the same room as the

player. But if the candy bar is in the same room, the **DestroyObject** subroutine won't work properly.

So I first need to check to make sure the player actually holds the candy bar, and print out a message saying "you don't have it" otherwise. Then the **energy** variable is set to 4. Remember that this is decremented in the **NPC.OBJ** file, to make sure the player pushes the boulder soon after eating the candy bar. The object number is set, and the subroutine to destroy the candy bar is called. Finally, the player is given a message that eating the candy has made the player stronger—a hint to try again to move the boulder.

Examine Shack

The last subroutine is a straightforward message subroutine that is called when the player attempts to examine the floor of the shack. And with that routine the entire subroutine file is completed. As you have seen, a large variety of subroutines are present in this one file.

I have also separated out some specialized subroutines that I wanted to be able to find easier. These included the **GetDrop.SUB**, **StartUp.SUB** and **MainLoop.SUB** files. When you create your own games, feel free to split up your subroutines in any way that makes your job easier. Unless you are writing a book like this, no one else is going to see your source code, so your only consideration should be what is easiest for you.

We have now examined nearly all the source code for *Cannibal*. The only file left untouched is the special vocabulary action file. This is where I have added commands that would not be picked up elsewhere by the parser. The next chapter will take a close look at what types of things are considered vocabulary action file entries.

Chapter 26: The Cannibal.VOC File

The vocabulary action file is a specialized file where you can anticipate commands from the player that will not be caught elsewhere. Usually they are not caught elsewhere because they are not actions made on a recognizable object.

You can easily expand the abilities of the Visionary "parser"

These may be single word commands like "dig" or "jump" which are verbs without a noun. They may be commands that would confuse the parser with directions, like "get up" or "look down". They may also be variations on a single command, like "turn on the radio" or "turn the radio on". This chapter will take a close look at what types of commands go in this file, and why.

Debugging Actions

The first three vocabulary actions in this file are special ones. They are never intended to be used by the player, but are to help me keep track of things as I debug the program.

Endgame

The first action is **endgame** and serves to jump to the end of the game. If I am testing the game, and want to check to make sure the ending works properly, I can simply type "endgame" and the .VOC file will find this action and set me at the end of the game. The timer will be set so that the cannibals are about to arrive. I am placed on the east end of the beach. The canoe is set in the water, and the shovel is placed in the room.

At this point I can either choose to wait a few turns and allow the game to end in a loss, or I can paddle the canoe and end the game in a win. This command came in handy many times as I was checking the animation of the death scene, the loading of the music, and other things at the end of the game.

Status

The next vocabulary action is **status**. When I am debugging *Cannibal* I can type "status" to find out the value of some important variables. It tells me how much fast memory and chip memory is left, as the game plays. This is important to check, in order to make sure that the game is compatible with all Amiga computers, even those with only a half megabyte of chip RAM.

The routine also prints out the current device name, which will let me know if the computer has properly copied the screens into RAM and is

loading them from there. The timer and items held are also printed out, since there are times I need to know how soon the cannibals will arrive, or if the inventory window is correctly displaying the same number of items that the game is keeping track of.

View Hidden Scenes

The third action is one that I created to allow me to see the screens that are normally hidden from the player's eyes. By typing "cycle" I can be shown graphic screen buffer 1, which contains the location scenery as loaded from the file.

Then I can next see screen buffer 2, which contains the "buttons.pic" file which contains all the buttons and movable objects. Screen buffer 23 shows me the top part of the long thin scrollbar. Screen buffer 24 shows me the small overlay buffer that Visionary uses to create masks for overlays. Then the text screen is shown, and finally I am cycled back to the game screen at screen buffer 0.

Movement between the screens is done by pressing the mouse button. Notice that between each "show screen" command are two "while" loops to make sure the button has been pressed and released. This "cycle" routine has come in handy many times, when things weren't working right and I wanted to view the various screens to help pinpoint the problem.

» Normally, all debugging routines would be removed before releasing the game. You don't want a player accidentally stumbling across these undocumented features that were only intended for your use. But I have left them in the program so that you can see examples of routines that you can create to help you in the programming of your own game.

Help

I have strongly recommended earlier in this book that you always include "helps" for the player. When the player gets stuck, typing a HELP command should produce some pertinent clue about getting past the trouble spot.

The action in line 57 shows how these helps are programmed into your game. Since the word "help" is a single verb and acts on no defined object, it must be defined here in the vocabulary action file. Notice that I have also listed some alternate ways to ask for help, such as "hint", "clue", and others.

Define context-sensitive help messages in the .VOC file

In *Cannibal*, I have only used a single help message, given any time the player asks for help. In a full-sized adventure, a single, repeated help message would not be recommended—it would be better to check the room the player is currently in, and give separate hints that are designed specifically for the current room or the present puzzle. "Context-sensitive help" is easy to do by checking the current room number, or by checking for the presence or absence of certain objects. Whatever you decide for your own game, remember that all "helps" go here in the vocabulary action file.

Supplementing Game Actions

The next vocabulary action is one that supplements the action "sit in the canoe." The action entry for this command can be placed in the file for the "canoe" object and will execute properly. But if the player chose to type "sit down in the canoe", the word "down" will cause the parser problems and will generate an error.

Handle directions used as prepositions

In order to make the game more user-friendly and permit the use of this exact wording, I placed the possible variations on this command in the vocabulary file. Notice that there are variations on this command, all of which will not be caught by the parser in the "canoe" file. All of them can be listed here, and help make the game play smoother without requiring the player to find a specific combination of words to accomplish the act of sitting down in the boat.

Usually, I would expect the player to try digging by clicking on the **DIG** button on the screen. On a DIG command, the **dig** subroutine is called, and the game checks to see is found. However, I also anticipated that the player might try typing the word "dig" instead of clicking the button. The vocabulary action **dig** at line 95 will catch such keyboard input and call the proper subroutine.

Building a shovel can be accomplished three ways. Two of them refer to an existing object, and can be placed in those object files. The third method refers to a non-existent object, the shovel. Since the object does not exist, the vocabulary action file must be used to allow this third type of input to be accepted.

Let the player refer to an object which does not yet exist

If the player types "put the shovel handle in the blade", the object file for the handle will be searched for the verb "put" and the shovel will be properly created. If the player types "put the blade on the handle", the object file for the blade will be searched for the verb "put" and the shovel will again be properly created. But if the player types "put the shovel together", the program will search object file for the shovel, and find that the shovel is not present in the current room. It will then return an error to the player.

To keep this from happening, I have designed this routine to be used when the player enters any of several commands that refer to the assembling of the shovel. The routine used here is the same as found in

both the "handle" and "blade" files, and need not be explained again here. For more information, go back to the chapter on **Movable.OBJ** files.

Allow commands that imply some other action in the context of the game

I anticipate the player will try to swim to freedom. So I permit it in the game, and tell the player that "you tire and return to the beach." I even encourage the player to try it, by highlighting the N compass button when the player is standing on either end of the beach. I would normally expect the player to use the mouse to click on the compass button, but there is always the chance the player will type the command to swim on the keyboard. The vocabulary action in line 132 anticipates three different variations on the keyboard input, and gives the same response as if the player were to use the mouse and compass buttons instead.

Adding the "get up" command in line 147 was done for the sake of completeness. Since I have allowed the player to sit down in the canoe, I must also allow the "get up" action. Since nothing was really done when the player sat down, other than acknowledging the command, nothing need be done for "Get up" either. A simple "OK" is all that's needed.

Permit expected player actions

The object file for the bottle permits several ways to state the command to break the bottle. There are two variations on the command which would not be caught by the object file, due to the word "open." For that reason, I have added these two variations to the vocabulary file in lines 154-156. The reason again is to make the game more user-friendly and give the player more leeway to phrase the command.

For exactly the same reason, I have added two more ways to try to break the coconut in lines 160-162. Again, the word "open" is permitted in "break open the coconut" and "break the coconut open." Notice that you can leave the word "the" out of the vocabulary action and it will be accepted if used.

Use noun-words as adjectives

The entry at line 166 was required because I wanted to use the word "candy" as an adjective in the movable object file. The object was referred to as "bar" to permit the player to type "candy bar". That meant that the action to "eat the candy bar" could be placed in the object file for the "bar", but the action to "eat the candy" would have to be placed here in the vocabulary action section. For more details, go back and read the part of the **Movable.OBJ** chapter on the candy bar.

The word "jump" is another one of those verbs that has no accompanying noun. Since there is no object file in which to place it, the command to jump should be placed here in the vocabulary action file. Since there were a variety of places in *Cannibal* where the player might try to jump, it was important to include the command "jump" and not leave it out of the game.

Provide a response to player actions

The command to "jump" will never get the player anywhere in the game. But that's no reason to ignore it. You must always anticipate any logical action the player may make and plan an appropriate response to it. So even though jumping does the player no good, I felt it important to include it here.

You can see from the source code listed in this section, that a variety of responses are given, depending on the location where the player tries to jump. All of the responses tell the player that nothing happens, but they tell it in different ways.

From the top of the palm tree, the player can see an ocean going freighter far out to sea. This will be the source of eventual rescue. If the player should either click on the picture of the ship, or type "look out to sea" the routine at line 197 will be executed. It will give the player a hint to help find the solution to the game.

The "look around the shack" command in line 208 will be given if the player either types it, or clicks on the inside of the shack. In either case, the "ExamineFloor" subroutine is called and the description is given. The command is included here in the vocabulary action file because the word "around" makes it difficult for the parser to catch it elsewhere.

Include your credit and/or copyright notice

The action at line 214 will allow the player to identify the author of the game, by typing "author author" or simply "author." It is doubtful that most players will even be aware of this possibility, but is included just in case. Its presence in the compiled code of the game with a copyright notice may also be used to satisfy the legal requirement for including a copyright notice within the compiled code of a software product.

If the player clicks on the "quit" button, I must also anticipate that the QUIT command may also be entered from the keyboard. If it is, the action listed in line 227 will catch it and make the game end.

Since the inventory window always is visible and shows the player what is carried, it is doubtful that any player would ever try to take inventory the "old" way by typing on the keyboard. But I always try to anticipate the player's unusual moves, and so the routine at line 233 will catch the inventory request and remind the player the inventory can be seen by looking in the inventory window.

Some Other Uses of .VOC Files

Before we quit our examination of the vocabulary action files, let's look at some other possible uses of this special file. If I had designed my *Cannibal* plot just a bit differently, there would be additional reasons to use the vocabulary action file. Let's consider what would happen in some of those hypothetical situations.

Handle actions involving one object placed inside another object

In the chapter on movable objects, we discussed the possibility of putting the batteries in the radio and removing them again. Although this is not allowed in *Cannibal*, it serves as an excellent example of why there is a need for the .VOC file. When the player asks to remove the batteries from the radio, Visionary would normally reply "There are no batteries here." This is because when the player uses the subject noun of "batteries", Visionary checks the object file and finds that the batteries are not in the current location. Remember, that when the batteries have been placed inside the radio, they are no longer actually in the same room with the radio. Instead, they have been moved to the "unused" room that stores such objects. The only way your game knows the batteries are in the radio, is that you have set an attribute for the batteries, such as "InRadio."

So in this example, the player's command to "remove the batteries from the radio" will not be successful if the action block is placed in the object file for the batteries. It should be placed instead in the vocabulary file. This file is checked after every move, so if the game can't find a way to respond to the request, it will be caught here.

Provide for all the normal ways of phrasing a command

Another example of where the vocabulary file can come in handy is when you have a radio or a flashlight that the player can turn on. There are basically two different ways the player can make this request: "turn the radio on" or "turn on the radio." Only the first way will be acceptable when used in the radio's object file. In the second way, the word "on" comes before "radio" and changes the way Visionary sees the line. The simplest way to solve the problem is to add a vocabulary action entry that allows the player to say "turn on radio." In this way, you have allowed the player more variety in command phrasing in your game, and it becomes more user-friendly.

As you gain more experience designing adventures, you will find more and more occasions to use the vocabulary action file. Some will be subtle differences, while others will present an obvious need. This is an excellent file to finish your game. When everything else is done, always go back to the vocabulary action file and check to see if there are any special commands that have not been covered elsewhere. If so, this is the place to put them.

We've now completed our look at all the source code files that went into the creation of the *Cannibal* adventure. The next chapter will try to pull everything together and see how the entire game is constructed of these parts.

Chapter 27: Putting It All Together

Now that we've taken an individual look at each file of source code that makes up the *Cannibal* game, let's take a step backwards and see how it all fits together to form a complete game. We will be taking a more generalized view of the game that we have in the last few chapters.

Initializing

Each of the files has a specific purpose to the final game. When the game first starts, it executes the **Cannibal.ADV** file which among other things tells it in which room the game will start. The program then executes that room file. In *Cannibal*, the initial room is the "west_end_of_beach".

When the program reaches there, I take care of all the initialization routines. All the graphics, music and sound are loaded into the game, and the basic click zones that will be used throughout the entire game are defined. To make sure that this initialization routine is never called again, I create and set a room attribute. Whenever the room is entered again, through the normal play of the game, the attribute will be checked and the initialization routine will be skipped.

The MainLoop

The next step is to jump to the main loop of the game. Since Visionary is only able to jump to subroutines, this main loop was made into a subroutine, even though it will only be called once and will never be exited until the game is over. The file **MainLoop.SUB** is split into two sections. Each section is loop itself. The two sections consist of a loop within a loop. The larger outer loop gets a player command and then ghosts the command to the Visionary parser. The inner loop checks for individual keypresses and builds the player's command one letter at a time. This inner loop also checks for mouse clicks on any defined click zone, and if it finds any, it acts as if a command was typed from the keyboard.

All the other files are ones that are called after either a click zone is selected with the mouse, or after a typed command is ghosted to the Visionary parser. If the player picks up and object from the location window and puts it in the inventory, the parser in the main loop is bypassed, and the action is handled strictly within the routines found in the **GetDrop.SUB** file. Otherwise, all mouse clicks are dealt with by subroutines and then the parser acts on the string variable passed to it by the subroutine.

Except for getting and dropping objects, all mouse clicks will eventually be handled by the parser. When click zones are defined, they specify a subroutine to be called when the zone is activated by clicking the mouse button. Generally the subroutines called in this way only need to set a string variable **$tx** that contains the command to be parsed and need to set a numeric variable to tell the main loop that a command is ready to be parsed. Then the inner loop that accepts the commands is exited, and the command is ghosted to the parser.

Visionary's parser takes care of splitting the command into separate words and acting on them. If the command is to move in a specific direction, the program first checks to make sure the move is a legal one. If the direction has been defined, and the rooms properly linked, the program executes the room file for that particular room. Otherwise, it will report an error to the text window. It also sets an error variable and the **$LastError** string variable.

After ghosting the command to the parser, I check this error variable, and print the contents of the **$LastError** variable to the graphic screen, so that the player can see what was wrong. If the room file is executed, my program forces a new scene to be displayed in the location window, and redraws the scrollbar that contains the objects present in the new room. Then the **NPC.OBJ** file is called, since it is called after every move. And finally control is returned to the main loop immediately after the ghost command that was just performed.

If the command input by the player was not a direction command, the parser splits it into parts so that it will know how to properly act on the command. It weeds out the articles, like "a", "an" and "the". It identifies the verb, which tells the parser what action is requested. It identifies the subject noun, which tells the parser the object file to execute. In the simplest cases, like "break the bottle" this is all that is necessary. The parser identifies the verb "break" and the subject noun "bottle." Then it checks the room to see if the object is present, either in the room or in the player's inventory.

If the object is not present, an error is set and reported back to the player. Otherwise, the object file is searched for the action that matches the verb. If it is found, the action block is executed, taking whatever action the programmer designed. If the action is not found, another type of error is reported to the player.

It is important to note that after the object file is checked, the special vocabulary action file is checked. In this way, special actions are caught that would otherwise produce an error. If the command typed by the player is found in the vocabulary action file, any previous errors set in the object file are cleared, and whatever is programmed into the specific action in the vocabulary action file is executed. For example, if the player said "break the bottle open" and the exact action is not found in the object file for the bottle, the entry in the vocabulary action file will be executed and no error will be generated.

The NPC.OBJ File

And finally, after both the object file and vocabulary action file have been checked, Visionary will execute the **NPC.OBJ** file. Remember that this file is always checked after every move, so that characters in the game other than the player, can be moved to new locations. It is also an excellent place to check for timers and create random events.

After the **NPC.OBJ** file is executed, then Visionary passes control back to the programmer. Notice that everything described in these last four paragraphs is done internally by Visionary after the command is ghosted. As a programmer, you don't have to do anything. But as a programmer, you are responsible to understand the inner workings of Visionary so you can write your programs more efficiently.

After Visionary has returned control of the program back to my program, it jumps back up to the start of the main loop again, and starts checking for further input. When you look at the general construction of the game from a wider viewpoint, it really is a simple idea.

The Call Schedule

Let's summarize how the files are used, one last time. The **Cannibal.SUB** file is first called. It then jumps to the **Cannibal.ROOMS** file. Then I take control and make it jump first to the **StartUp.SUB** file and then to the **MainLoop.SUB** file, where the game is actually played. While the game is being played, any use of the mouse to get and drop objects calls the **GetDrop.SUB** file. If the player moves to a new room, the **Cannibal.ROOMS** file is called again.

Any other input is parsed by Visionary's parser, and the **Nonmovable.OBJ**, **Movable1.OBJ** and **Movable2.OBJ** files are called depending on what object was used in the command. These files often call the **Cannibal.SUB** file. Then the **Cannibal.VOC** file is automatically called by Visionary, and likewise the **NPC.OBJ** file is also automatically executed. Then the game picks up back in the **MainLoop.SUB** file where it left off. This accounts for all eleven of the source code files used in the *Cannibal* game.

The next and final chapter will remind you of some shortcuts and special techniques that I used in *Cannibal*. It will also introduce some additional tricks that I didn't use in my game, but which you may find useful in your own creations.

Chapter 28: Additional Tips and Tricks

This chapter will list some special tips on using Visionary, and will include some techniques I decided not to use in the final version of *Cannibal*, but which you may find useful. Keep them in mind as you write your own games. They can make your job as a programmer easier. They can also make your game look nicer, play smoother, and react faster.

ASCII Codes

Check the Visionary manual for more information on in-line formatting of ASCII codes. The "\F", "\R" and "\B" codes used in the **MainLoop.SUB** file are examples of these codes. Although they were originally intended to be used with the text screen, they can be used elsewhere in your programs.

You may remember I used some of them in my main loop to see if the player had pressed the [BackSpace] or [Return] keys. Not only can they be printed to the text screen, they can also be included in string variables, as I did to check for the [BackSpace] and [Return].

I found the "\F" form-feed command useful in clearing the text screen of unwanted text, so that any requester windows would look cleaner. And finally remember that any ASCII character can be printed, by using the back-slash followed by a three digit number. I was able to put a quotation mark in some of my messages by using "\034", something that I would have been unable to do any other way. Keep these special ASCII codes in mind as you create your own games.

You can use the back-slash key to send an escape code, in order to change pen numbers. What are pen numbers, and why would you want to change them? The text screen has four colors. Color 0 is the gray color of the background, and defaults to RGB values of 8,8,8. Color 1 is the black color of the text that the game prints, and defaults to RGB values 0,0,0. Color 2 is the white color of the text that the player types, and defaults to RGB values of 15,15,15. Color 3 is the red color of the bar at the top of the text screen, and defaults to RGB values of 8,0,0. You can change these using Visionary's **PALETTE** command. You can also change the normal game output from color 1 to color 2 by using this simple line:

```
T \027[32m
```

This switches the output pen from 1 to 2, by printing an escape code and the code to switch pens. In this way, the color of the game output will be white, the same as the player input. You can switch to any pen you want, by using the numbers 30-33 for pens 0-3. This technique was used in the LoadingError subroutine found at the end of the **Start-Up.SUB** file.

Debugging

The next tip has to do with the cross-reference file Visionary can produce. It is very useful in finding errors reported to you by DBUG. When you are testing your game, errors are bound to appear. When they do, they will be printed in the special DBUG window, that you can see by using the layering gadgets. These errors will also be written to the .ERR file, so you can examine them later.

Each error will tell you three things: the error number, a brief description of the error, and the line in which the error occurred. To check the line number, you will need to use the .XRF file. This is a file which VCOMP will create when compiling your code into the .GAM and .WRD files. It will only be created if you specifically request it as a compile time option. My advice is to always request it. It only increases the compile time marginally, and is well worth it. The .XRF file will list your entire source code, with line numbers. When errors are reported, you can check the .XRF file and easily find the exact line which caused the problem. Using this information along with the error number and description will make correcting the error a simple task.

Odds and Evens

This next technique involves odd and even. If for some reason you want to know if a variable is odd or even, there are two simple, short routines that do the job different ways.

```
IF VARIABLE AND 1 THEN
 T This number is odd.
ELSE
 T This number is even.
ENDIF
```

or

```
IF 1 - (VARIABLE AND 1) THEN
 T This number is even.
ELSE
 T This number is odd.
ENDIF
```

Either one can be used in a variety of ways, not restricted to their use in conditional statements.

External DOS Commands

What if you are multi-tasking some other program, and want to stop it with a **[Ctrl]-C** command? This may happen if you are using an external animation player or sound player—you find that if it was being played normally from CLI, you could exit by holding down the **[Ctrl]** key and pressing C. But how do you stop the external process from within your Visionary game?

The answer lies in Visionary's DOS command. You need to know the **process number**. This refers to the number that appears on the CLI screen from which the external program was run. It usually looks something like "2" except the number may be different. In this example, the DOS command would be:

```
DOS "break 2 c"
```

This line will send the command to CLI and stop whatever process 2 was executing.

Saving Memory

Let's next consider ways to make your saved game files smaller, so they will take less disk space and will load faster.

When Visionary saves a game position, it saves all graphic and sound files as well as the values of all variables, attributes, rooms, etc. However, if the graphic screens have not been modified since they were originally loaded, they are NOT saved. When the game is loaded again, the unmodified graphic files will be loaded from their original files.

Don't write to a screen unless you have to

The fact that all modified graphic screens are saved in the .SAV file, allows you to use techniques to keep the .SAV file size at a minimum. First, don't write to a graphic screen in memory unless you have to. In that way, it won't be included in the .SAV file. For example, in *Cannibal* I place my overlay buffer on screen 2. That means screen 2 (the screen containing the file **buttons.pic**) must be included in the .SAV file. A better way would be to create special screens in memory to do the overlay.

Unload unwanted screens from memory before a SAVE

Then just before a game is saved, unload that screen from memory so that it won't be saved. Remember, it doesn't actually contain anything that must be saved anyway. Then after the save is over, create the screen again. Remember to create the screen again after a load. In this way, your .SAV file will be appreciably smaller.

Encoding Files

As long as we are talking about graphic files, let's look at the techniques of encoding your graphic and sound files.

It's wise to encode them in your game, so that when it is released for others to play, no unauthorized changes can be made. Without the encoding, anyone could load the graphics files into a paint program and make any modifications they wished. Or they could change the digitized sounds with a sound editor.

It's important to remember that the password used in the .ADV file must match the password you use when you use the **VCODE** program to encode your graphics and sounds. It's also important to use a different password on each game you write. In this way, each set of graphics and sounds have their own password and they are much more secure. VCODE is an important utility program available on your Visionary disk, and I recommend that you use it.

The Title Screen

The **LoadScreen** utility program that is on your Visionary disk is useful for displaying your title screen. It can be displayed while your Visionary game is loading and getting set up to start. Notice that it allows some options like color cycling, which permits a type of animation in your title screen if you wish it.

Don't encode your title screen

Since **LoadScreen** is a separate program from Visionary, it will not decode graphic screens that have been encoded with VCODE. For this reason, your title screen should be the only one that is not encoded. The best way to keep your title screen visible until the right moment is to link your game with VLINK using the **-g** option. This keeps the game's graphic screens in the background, while the title screen is displayed.

Close the title screen when the game starts

When your game is ready to be seen, you can force it to the front with a **ScreenMode Graphics** command from within your Visionary game. That puts the title screen in the background out of the way. Then to free up the Chip RAM that is no longer needed, you should use the DOS command to call the **CloseScreen** utility. It will close the graphic screen and give your game more chip ram to use.

Changing the Mouse Pointer

While we are on the subject of graphics, here is another tip regarding the color of the mouse pointer. In a Visionary game, the mouse pointer is in the shape of a hand. You can change the color of the hand by changing the colors of your palette. Color 17 controls the main color of the hand. Color 18 controls the highlights in the hand.

Changes in a 32-color screen palette may affect the mouse pointer

If you are using a 32-color screen, be careful which colors you use in your palette for colors 17 and 18, or the hand may not show up well against your graphics. You want it to contrast with the rest of the screen, no matter where the mouse is pointed. It may take some experimenting to find the correct colors. When you are working in the high-resolution mode, you only have 16 colors in your palette, so you need not worry about any conflict with colors. You can still set the colors of the hand, by using Visionary's **PALETTE** command. But you won't have to worry about any "hand" changes also changing your other graphics.

PAL and NTSC

Another subject related to graphics is the NTSC standard as compared to the PAL standard. A low-resolution screen in the NTSC standard is 320x200 pixels. In PAL standard, a low-resolution screen is 320x256 pixels. Higher-resolution modes vary proportionally.

Your game should work with both NTSC and PAL

Visionary can tell which type of system your game is being played on, and makes that information available in the **VideoMode** variable. If the variable is equal to 0, the mode is NTSC. If the variable is 1, the mode is PAL. You should try to design your game to work with either standard. This can be done by checking the **VideoMode** variable and then modifying the names of the graphic files you load, to load the ones of the proper length. You can also check the **VideoMode** variable in order to modify your click zones, to accommodate the longer PAL screen. Admittedly, this will all take more work, but it will result in a more compatible game. Visionary gives you the proper tools, and it is up to you to use them.

The Music Editor

Finally, let's take a quick look at **MED**, the music editor which can be used to create music for your Visionary game. Using MED, you can create music that has a high-quality sound, but takes only a fraction of the space needed by a digitized sound sample.

The beauty of MED is that it is freely distributable and easy to learn to use. You don't have to buy any music software, you can just obtain if for free from a wide variety of sources. If you decide you like MED, and intend to keep and use it, you should register with the author and send in a small fee.

MED is easy to use. It comes complete with instructions for use, in a text file that accompanies the MED program. Even a novice can learn to write music with MED. I know, because I did. I am not a musician. I've never played a musical instrument in my life—unless you count a record player! But within a day of receiving MED and going through the directions, I had created my first piece of music. It may not have

been a masterpiece, but it did use all four voices and several different instruments. And it sounded good to my ears.

Since Visionary supports MED songs, I recommend you obtain a copy and use it. It is available from any source of public domain material, including Fred Fish disks and bulletin board systems.

This is not a complete list of shortcuts and techniques for use in Visionary. But it lists some of the things I have used in the games I have written. If you discover others, document them in your source code, so you won't lose track of them.

And write me at Oxxi. I'd love to hear about your clever solutions, neat tricks and imaginative uses of Visionary's abilities—I might even publish them in my regular column *The Sorcerer's Den* in every issue of **Enchanted Realms** magazine.

Visionary has opened a new world of interactive games programming. Let's all take advantage of that fact and use this new language to its fullest.

Appendix A: Source Code for the Cannibal Game

The .ADV File

```
1
2   ;---------- Cannibal.ADV ----------
3
4   ADVENTURE
5
6   PASSWORD jroj
7
8
9   VAR
10  timer           0       ; keep track of moves
11  energy          0       ; becomes 4 when you eat the snicker
12  dig             0       ; ability to dig in 0=sand 1=dirt 2=rock
        3=wood/tree
13
14  RoomNumber      1       ; room number 1-16
15  LastRoomNumber  1       ; to see if a new scene needs to be loaded
16  $device                 ; will be null, or "ram:" if enough ram
        space
17  $filename               ; filename variable for loading files
18  ChosenPic               ; 0-18 for picture in room array chosen
        to move
19  pic                     ; 0-4 for position on scrollbar
20  ObjNum                  ; 1-19 for movable objects
21  ObjTotal                ; total movable objects in this room
22  SBPosition              ; 0-14 for scroll bar position
23  SBSize          5       ; scroll bar size, will hold 5 objects
        max.
24  MaxMov          19      ; maximum movable objects = 19 (1-19)
25  Slot                    ; temporary variable for openings in
        inventory
26  NewX                    ; new MouseX position
27  NewY                    ; new MouseY position
28  OldX                    ; old mouseX position
29  OldY                    ; old mouseY position
30  $tx                     ; text string to be printed
31  temp            0       ; temporary variable for various uses
32  temp1           0       ; temporary variable for various uses
33  temp2           0       ; temporary variable for various uses
34  $temp                   ; temporary string for various uses
35  $letter                 ; single character for getstring
36  $sentence               ; built up sentence of input
37  sentence                ; length of the $sentence
38  TextPosition            ; position text on graphic screen
```

```
39  MaxLines          5       ; maximum lines in text window before
        pause
40  CountLines                ; counter for lines displayed in text
        window
41  $return                   ; RETURN cannot be defined here
42  return                    ; =1 when RETURN is pressed
43  $backspace                ; BACKSPACE cannot be defined here
44  backspace                 ; =1 when backspace is pressed
45  MainLoop                  ; main loop while variable
46  TextColor        13       ; default text color (blue)
47  white             8       ; palette color for white
48  blue             13       ; palette color for blue
49  red               9       ; palette color for red
50  green            28       ; palette color for green
51  brown            18       ; palette color for brown
52  dummy                     ; dummy variable
53  val                       ; value of the input character
54  UpArrowActive         0   ; =1 if scrollbar up arrow shows
55  DownArrowActive       0   ; =1 if scrollbar down arrow shows
56  ButtonUsed                ;
57  x1                        ;\
58  y1                        ; \
59  x2                        ;  \ variables for block copies
60  y2                        ;  / showing buttons depressed
61  x                         ; /
62  y                         ;/
63  GoN              13       ; offset to add to N button, (lighted)
64  GoS              13       ; offset to add to S button, (lighted)
65  GoE              13       ; offset to add to E button, (lighted)
66  GoW              13       ; offset to add to W button, (dark)
67  GoU              13       ; offset to add to U button, (dark)
68  GoD              13       ; offset to add to D button, (dark)
69  offset          -13       ; offset to pop button up
70
71  ENDVAR
72
73  ROOM
74   Cannibal.rooms
75  ENDROOM
76
77  OBJECT
78   NonMovable.obj
79   Movable1.obj
80   Movable2.obj
81   NPC.obj
82  ENDOBJECT
83
84  SUB
85   Cannibal.SUB
86   GetDrop.SUB
87   MainLoop.SUB
88   StartUp.SUB
89  ENDSUB
90
91  VOCAB
92   Cannibal.VOC
93  ENDVOCAB
```

```
 94
 95   INITROOM  west_end_of_beach
 96
 97   ENDADVENTURE
 98
 99  ;-------------------------------------
100
```

The .ROOMS File

```
1
2   ;--------------- Cannibal.rooms ---------------
3
4   Room Unused ; where unused objects can be temporarily stored
5   code
6    set west_end_of_beach, ForcedReturn ; if player tries to swim,
         he goes
7    go west_end_of_beach                ; here and is forced back to
         beach
8   endcode
9   Endroom
10
11  ;---------------------------------------
12
13  room west_end_of_beach
14
15  attrib
16   started N
17   ForcedReturn N
18  endattrib
19
20   default
21   s sand_dunes
22   e east_end_of_beach
23   n unused
24  enddefault
25
26  code
27
28  call ClearButtons
29
30  click 34, 140,63, 225,105, SeeBoat
31  click 35, 6,74,   231,124, SeeSand
32  click 36, 6,45,   231,71,  SeeOcean
33  click 37, 6,6,    231,71,  SeeSky
34
35  placeobj ocean, thisroom
36  placeobj sand, thisroom
37  placeobj sun, thisroom
38  placeobj sky, thisroom
39  dig := 0
40
41  RoomNumber := 1
42
43  If west_end_of_beach not started then
44   call StartUp
45  else
46   call ReDrawScreen
47   play sound 1, 0,0,40,0 ; ocean
48   stop sound    1        ; birds
49  endif
50
51
```

```
52
53  if west_end_of_beach is ForcedReturn then
54   unset west_end_of_beach, ForcedReturn
55   call NoSwim
56  elsif thisroom not visited then
57   $tx := "You are on the west beach, by a battered"
58   call print
59   $tx := "rowboat.  The warm waters of the Pacific"
60   call print
61   $tx := "gently lap ashore."
62   call print
63  else
64   $tx := "You are on the west end of the beach."
65   call print
66  endif
67
68  if west_end_of_beach not started then
69   call startUp2
70  endif
71
72  endcode
73
74  endroom
75
76  ;-------------------------------------
77
78  room ocean2 ; used if player tries to swim out from
          east_end_of_beach
79
80  code
81   set east_end_of_beach, ForcedReturn
82   go east_end_of_beach
83  endcode
84
85  endroom
86
87  ;-------------------------------------
88
89  room east_end_of_beach
90
91  attrib
92   ForcedReturn N
93  endattrib
94
95  default
96   w west_end_of_beach
97   n ocean2
98  enddefault
99
100 code
101
102 call clearButtons
103
104 click 36, 6,6,    231,82,  SeeOcean
105 click 37, 6,82,   231,124, SeeSand
106
107 if canoe is InWater then
```

```
108     RoomNumber := 4
109     click 34, 70,39,  140,63,  SeeCanoe
110     click 35, 118,23, 170,50,  SeeCanoe
111    else ; if canoe is beached
112     RoomNumber := 2
113     click 34, 33,68,  113,107, SeeCanoe
114     click 35, 87,53,  137,88,  SeeCanoe
115    endif
116    Call ReDrawScreen
117
118    placeobj sand, thisroom
119    placeobj ocean, thisroom
120    dig := 0
121
122    if east_end_of_beach is ForcedReturn then
123     unset east_end_of_beach, ForcedReturn
124     call NoSwim
125    elsif thisroom not visited then
126     $tx := "You are on the east end of the beach."
127     call print
128     $tx := "A shallow lagoon is north.  A native"
129     call print
130     if canoe is InWater then
131      $tx := "canoe floats in the shallows."
132      call print
133     else
134      $tx := "canoe lies beached at your feet."
135      call print
136     endif
137    else
138     $tx := "You are at the east end of the beach."
139     call print
140    endif
141
142    endcode
143
144    endroom
145
146    ;-------------------------------
147
148    room sand_dunes
149
150    default
151     n west_end_of_beach
152     s meadow
153    enddefault
154
155    code
156
157    call ClearButtons
158
159    click 34, 55,6,    75,21,   SeePlant
160    click 35, 94,88,   126,111, SeePlant
161    click 36, 207,54,  231,74,  SeePlant
162    click 37, 217,89,  231,99,  SeePlant
163    click 38, 154,58,  191,65,  SeeOcean
164    click 39, 6,22,    100,33,  SeeDunes
```

```
165   click 40, 6,34,    134,56,  SeeDunes
166   click 41, 6,57,    231, 125,SeeDunes
167   click 42, 6,6,     231,55,  SeeSky
168
169   RoomNumber := 3
170   Call ReDrawScreen
171   play sound 1, 0,0,20,0 ; ocean
172   play sound 2, 1,0,10,0 ; birds
173
174   placeobj sand, thisroom
175   placeobj dunes, thisroom
176   placeobj plant, thisroom
177   placeobj sky, thisroom
178   placeobj ocean, thisroom
179   dig := 0
180
181   if thisroom not visited then
182    $tx := "You are walking through the sand dunes."
183    call print
184   else
185    $tx := "You are at the sand dunes."
186    call print
187   endif
188
189   endcode
190
191   endroom
192
193   ;---------------------------------------
194
195   room meadow
196
197   default
198    n sand_dunes
199    s by_the_boulder
200    w in_the_shack
201    e top_of_the_cliff
202    u top_of_the_shack
203   enddefault
204
205   code
206
207   call ClearButtons
208
209   click 34, 6,35,   22,124,  SeeShack
210   click 35, 23,49,  44,124,  SeeShack
211   click 36, 39,76,  115,119, SeePlant
212   click 37, 112,68, 126,78,  SeePlant
213   click 38, 140,87, 231,124, SeePlant
214   click 39, 140,53, 158,57,  SeeOcean
215   click 40, 51,38,  128,124, SeeDunes
216   click 41, 129,51, 231,124, SeeDunes
217   click 42, 6,6,    231,52,  SeeSky
218
219   RoomNumber := 6
220   Call ReDrawScreen
221   stop sound    0       ; ocean
```

```
222   play sound 2, 1,0,40,0 ; birds
223
224   placeobj plant, thisroom
225   placeobj sand, thisroom
226   placeobj sky, thisroom
227   placeobj dunes, thisroom
228   placeobj ocean, thisroom
229   dig := 1
230
231   if thisroom not visited then
232    $tx := "You are standing in a meadow, by an old"
233    call print
234    $tx := "wooden shack.  The door is missing, and"
235    call print
236    $tx := "no one is in sight."
237    call print
238   else
239    $tx := "You are in the meadow."
240    call print
241   endif
242
243   endcode
244
245   endroom
246
247   ;-------------------------------------
248
249   room top_of_the_shack
250
251   default
252    d meadow
253   enddefault
254
255   code
256
257   call ClearButtons
258
259   click 34, 195,74, 206,92,  SeeLadder
260   click 35, 216,80, 226,105, SeeLadder
261   click 36, 6,67,   231,124, SeeRoof
262   click 37, 156,38, 185,58,  SeePlant
263   click 38, 6,6,    231,75,  SeeSand
264
265   RoomNumber := 5
266   Call ReDrawScreen
267
268   placeobj plant, thisroom
269   placeobj ladder1, thisroom
270   placeobj sand, thisroom
271   dig := 3
272
273    if thisroom not visited then
274     $tx := "You are on the top of the shack.  The"
275     call print
276     $tx := "weathered boards are nailed down"
277     call print
278     $tx := "securely, to form the roof."
```

```
279     call print
280    else
281     $tx := "You are on the top of the shack."
282     call print
283    endif
284
285   endcode
286
287   endroom
288
289   ;-------------------------------------
290
291   room top_of_the_cliff
292
293   default
294    w meadow
295   enddefault
296
297   code
298
299   call ClearButtons
300
301   click 34, 97,76,   112,98,  SeePlant
302   click 35, 153,96,  162,113, SeePlant
303   click 36, 190,59,  211,77,  SeePlant
304   click 37, 61,74,   102,108, SeeRocks
305   click 38, 96,61,   114,81,  SeeRocks
306   click 39, 190,82,  218,124, SeeRocks
307   click 40, 115,13,  137,74,  SeeTree
308   click 41, 78,6,    140,35,  SeeBoughs
309   click 42, 6,106,   190,124, SeeSand
310   click 43, 105,59,  194,106, SeeSand
311   click 44, 6,6,     231,124, SeeSky
312
313   RoomNumber := 7
314   Call ReDrawScreen
315   play sound 2, 1,0,10,0 ; birds
316   play sound 1, 0,0,10,0 ; ocean
317
318   placeobj sand, thisroom
319   placeobj boughs, thisroom
320   placeobj rocks, thisroom
321   placeobj plant, thisroom
322   placeobj sky, thisroom
323   dig := 1
324
325   if thisroom not visited then
326    $tx := "You find yourself at the top of a cliff."
327    call print
328    $tx := "Except for the palm tree, very little"
329    call print
330    $tx := "vegetation is found here. Far below, the"
331    call print
332    $tx := "waves crash against the cliff."
333    call print
334   else
335    $tx := "You are at the top of the cliff."
```

```
336    call print
337   endif
338
339   endcode
340
341   endroom
342
343   ;-------------------------------------
344
345   room by_the_boulder
346
347   default
348    n meadow
349   enddefault
350
351   code
352
353   call ClearButtons
354
355   click 36, 6,6,    94,63,   SeeSky
356   click 37, 84,83,  198,100, SeeSand
357   click 38, 82,101, 130,114, SeeSand
358   click 39, 60,115, 103,124, SeeSand
359   click 40, 6,6,    231,124, SeeRocks
360
361   if boulder is moved then
362    RoomNumber := 11
363    click 34, 57,35,  141,91, SeeBoulder
364    click 35, 142,26, 172,78, SeeCave
365    placeobj cave, thisroom
366   else ; boulder is in front of opening
367    RoomNumber := 9
368    click 34, 126,12, 185,82, SeeBoulder
369   endif
370   Call ReDrawScreen
371   play sound 2, 1,0,10,0 ; birds
372   stop sound    0        ; drip
373
374   placeobj sand, thisroom
375   placeobj rocks, thisroom
376   placeobj sky, thisroom
377   dig := 1
378
379   if thisroom not visited then
380     $tx := "You are standing at the south end of"
381    call print
382    $tx := "the meadow by the rock cliff.  A giant"
383    call print
384    if boulder is moved then
385     $tx := "boulder lies beside the opening to"
386    else
387     $tx := "boulder lies in front of an opening to"
388    endif
389    call print
390    $tx := "a cave in the cliff wall."
391    call print
392   else
```

```
393    $tx := "You are back beside the boulder."
394    call print
395   endif
396
397   endcode
398
399   endroom
400
401   ;-------------------------------------
402
403   room in_the_tree
404
405   default
406    d top_of_the_cliff
407   enddefault
408
409   code
410
411   call ClearButtons
412
413   click 34, 74,9,   85,13,   SeeShip
414   click 35, 93,100, 123,119, SeeRocks
415   click 36, 14,93,  231,124, SeeBoughs
416   click 37, 112,61, 138,76,  SeeRocks
417   click 38, 135,47, 169,63,  SeeRocks
418   click 39, 6,10,   231,124, SeeOcean
419
420   RoomNumber := 8
421   Call ReDrawScreen
422
423   placeobj boughs, thisroom
424   placeobj rocks, thisroom
425   placeobj ocean, thisroom
426   dig := 3
427
428   if thisroom not visited then
429    $tx := "You are sitting in the top branches of"
430    call print
431    $tx := "the tall palm tree.  Way out to sea, an"
432    call print
433    $tx := "old ocean freighter slowly makes its"
434    call print
435    $tx := "way east.  Gray smoke rises from its"
436    call print
437    $tx := "smoke stacks."
438    call print
439   else
440    $tx := "You are in the top of the palm."
441    call print
442   endif
443
444   endcode
445
446   endroom
447
448   ;-------------------------------------
449
```

```
450   room in_the_shack
451
452   default
453    e meadow
454   enddefault
455
456   code
457
458   call ClearButtons
459
460   click 34, 125,6,  188,45,  SeeWindow
461   click 35, 35,44,  205,55,  SeeTable
462   click 36, 6,6,    231,124, SeeInterior
463
464   RoomNumber := 12
465   Call ReDrawScreen
466   play sound 2, 1,0,10,0 ; birds are softer inside shack
467
468   placeobj sand, thisroom
469   dig := 3
470
471   if thisroom not visited then
472    $tx := "You are standing inside the shack.  It"
473    call print
474    $tx := "has been abandoned for years.  Dust has"
475    call print
476    $tx := "settled on everything.  Light streams"
477    call print
478    $tx := "in the window, illuminating the inside."
479    call print
480   else
481    $tx := "You are inside the shack."
482    call print
483   endif
484
485   endcode
486
487   endroom
488
489   ;-----------------------------------------
490
491   room in_the_cave
492
493   default
494    w by_the_boulder
495   enddefault
496
497   code
498
499   call ClearButtons
500
501   click 34, 10,11,  39,49,   SeeHole
502   click 35, 86,37,  114,59,  SeeWall
503   click 36, 6,6,    231,124, SeeRocks
504
505   RoomNumber := 13
506   Call ReDrawScreen
```

```
507   play sound 2, 1,0,5,0 ; birds faintly heard in cave
508   play sound 3, 0,0,20,0 ; drip
509
510   placeobj sand, thisroom
511   placeobj cave, thisroom
512   placeobj rocks, thisroom
513   dig := 2
514
515   if thisroom not visited then
516    $tx := "The small opening high in the wall of"
517    call print
518    $tx := "the cave looks big enough to crawl"
519    call print
520    $tx := "through, but its ten feet off the"
521    call print
522    $tx := "ground. Rough pictures are chiseled into"
523    call print
524    $tx := "the wet rock walls."
525    call print
526   else
527    $tx := "You are inside the cave."
528    call print
529   endif
530
531   endcode
532
533   endroom
534
535   ;-------------------------------------
536
537   room rock_room
538
539   default
540    d in_the_cave
541   enddefault
542
543   code
544
545   call ClearButtons
546
547   click 34, 11,6,    81,34,    SeeFissure
548   click 35, 92,39,   126,55,   SeeWall
549   click 36, 6,6,     231,124,  SeeRocks
550
551   RoomNumber := 14
551   Call ReDrawScreen
553   play sound 3, 0,0,40,0 ; drip
554   stop sound    1          ; birds
555
556   placeobj cave, thisroom
557   placeobj sand, thisroom
558   placeobj rocks, thisroom
559   dig := 2
560
561   if thisroom not visited then
562    $tx := "You are inside a small rock room."
563    call print
```

```
564    $tx := "There is writing carved into the wall."
565    call print
566    $tx := "The ladder leads down through the"
567    call print
568    $tx := "opening."
569    call print
570   else
571    $tx := "You are inside the rock room."
572    call print
573   endif
574
575   endcode
576
577   endroom
578
579   ;-------------------------------------
580
581   room top_of_the_boulder
582
583   default
584    d by_the_boulder
585   enddefault
586
587   code
588
589   call ClearButtons
590
591   click 34, 6,6,    231,124, SeeIsland
592
593   RoomNumber := 10
594   Call ReDrawScreen
595
596   dig := 3
597
598   if thisroom not visited then
599    $tx := "You are sitting on top of the large"
600    call print
601    $tx := "boulder.  You can survey the entire"
602    call print
603    $tx := "island from here."
604    call print
605   else
606    $tx := "You are on top of the large boulder."
607    call print
608   endif
609
610   endcode
611
612   endroom
613
614   ;-------------------------------------
615
616
```

The StartUp.SUB File

```
1
2   ;--------------- StartUp.SUB ---------------
3
4   sub StartUp
5
6    TextPalette 3,8,8,8 ; change color of prompt & bar to gray
7
8    scrollbar off ; also prevents front/back gadgets from being seen
9    menus off
10
11   load song 0, "Cannibal:audio/title.mus" ; load MED song
12   EnableMusic ; turn off digitized sound and turn on MED
13   play song 0 ; play MED song
14
15   if FastMem > 200 then ; if there's enough ram, put scenes in ram
16    $device := "ram:"
17    dos "copy >NIL: Cannibal:Video/loc#? to ram:" ; copy all loc's
          to ram:
18   else
19    $device := "Cannibal:Video/"
20   endif
21
22   $filename := "Cannibal:video/loc1"
23   load screen 1, $filename
24   call LoadingError
25
26   $filename := "Cannibal:video/buttons.pic"
27   load screen 2, $filename
28   call LoadingError
29
30  $filename := "Cannibal:Audio/bubbles.snd"
31   load sound 0, $filename
32   call LoadingError
33
34   $filename := "Cannibal:Audio/ocean.snd"
35   load sound 1, $filename
36   call LoadingError
37
38   $filename := "Cannibal:Audio/birds.snd"
39   load sound 2, $filename
40   call LoadingError
41
42   $filename := "Cannibal:Audio/drip.snd"
43   load sound 3, $filename
44   call LoadingError
45
46   $filename := "Cannibal:Video/window.pic"
47   load screen 0, $filename
48   call LoadingError
49
50   $filename := "artesian.font"
51   load font 0, "artesian.font",8
52   call LoadingError
```

```
 53    font 0,0 ; use font 0 on screen 0
 54
 55    color 0,blue
 56    $tx := "    Copyright @1991  by John Olsen"
 57    call print
 58    call Linefeed
 59    $tx := "     I WAS A CANNIBAL FOR THE FBI"
 60    call print
 61    $tx := "              by John Olsen"
 62    call print
 63    call Linefeed
 64
 65    create screen 24, 55, 78, 5, lores ; ladder size
 66    mask 24
 67
 68    create screen 23, 15, 361, 5, lores ; for scrollbar
 69    mode 23, draw
 70    color 23, white
 71    rect 23, 0,0, 14, 360 ; fill scrollbar with white
 72
 73    show screen 0
 74    screenMode graphics ; shows screen w/ window, scenery, text
 75    DisableMusic ; turn off title MED music
 76    play sound 1, 0,0,40,0 ; ocean
 77    DOS "run CloseScreen Video/title.pic"
 78
 79    $tx := "That's right, the FBI.  They sent you"
 80    call print
 81    $tx := "undercover to a cult of cannibals.  But"
 82    call print
 83    $tx := "you were discovered and imprisoned on"
 84    call print
 85    $tx := "this neighboring island.  They will be"
 86    call print
 87    $tx := "coming for you soon. And they're hungry!"
 88    call print
 89    $tx := "You must escape before they return."
 90    call print
 91    call Linefeed
 92
 93    click 0,  273,64,  285,76,  GoNorth
 94    click 1,  273,97,  285,109, GoSouth
 95    click 2,  288,81,  300,93,  GoEast
 96    click 3,  259,81,  271,93,  GoWest
 97    click 4,  307,64,  319,76,  GoUp
 98    click 5,  307,97,  319,109, GoDown
 99    click 6,  259,115, 319,127, Help
100    click 7,  259,133, 319,145, Dig2
101    click 8,  259,151, 319,163, Load
102    click 9,  259,169, 319,181, Save
103    click 10, 259,187, 319,199, Quit
104    click 11, 236,21,  250,109, GetObject
105    click 12, 266,17,  312,51,  DropObject
106    click 13, 236,6,   250,17,  ClickUpArrow
107    click 14, 236,113, 250,124, ClickDownArrow
108
109    placeobj 00nothing, thisroom
```

```
110    grab 00nothing
111
112
113    endsub
114
115    ;-------------------------------------
116
117    sub StartUp2
118
119     set west_end_of_beach, started
120     set west_end_of_beach, visited ; since room loop not completed
121     call MainLoop
122
123    endsub
124
125    ;-------------------------------------
126
127    sub LoadingError
128
129    if error > 0 then
130     t \027[32m ; this changes text output to pen 2 (normally pen 1)
131     TextPalette 2,15,15,15 ; this changes pen 2 to white
123     t \f
133     t
134     t
135     t
136     t
137     t
138     t
139     t
140     t
141     t
142     t     ERROR: can't load file "@$filename "!  Press RETURN to
          abort.
143     t \027[31m ; this changes text output back to pen 1
144     DOS "CloseScreen Cannibal:Video/title.pic"
145     ScreenMode text
146     getstring $temp
147     quit
148    endif
149
150    endsub
151
152    ;-------------------------------------
153
```

The NonMovable.OBJ File

```
1
2   ;-------------- NonMovable.obj --------------
3
4   object 00nothing
5
6   name nothing
7
8   initroom unused
9
10  code
11  endcode
12
13  endobject
14
15  ;---------------------------------------
16
17  object sand
18
19  name ground, sand, dirt
20
21  initroom unused
22
23  code
24  ; so player can say PUT THE LADDER ON THE GROUND
25  endcode
26
27  action look, examine, search
28   $tx := "It's just normal beach sand."
29   call print
30  endact
31
32  action get, take, grab
33   $tx := "Leave the sand alone!  The next thing,"
34   call print
35   $tx := "you'll try getting the sun!"
36   call print
37  endact
38
39  action dig
40   call Dig
41  endact
42
43  endobject
44
45  ;---------------------------------------
46
47  object floor
48
49  name floor
50
51  initroom in_the_shack
52
53  code
```

```
endcode

action look, examine
 call ExamineFloor
endact

endobject

;--------------------------------------

object sky

name sky

initroom unused

code
endcode

action look, examine, search
 $tx := "The sky is clear and blue, with only a"
 call print
 $tx := "few white puffy clouds floating about."
 call print
endact

action get, take, grab
 $tx := "It's too high.  Why are you holding your"
 call print
 $tx := "hands up?"
 call print
endact

endobject

;--------------------------------------

object sun

name sun

initroom unused

code
endcode

action look, examine, search
 $tx := "It's so bright that you hesitate to look"
 call print
 $tx := "into it.  But you can feel the warmth."
 call print
endact

action get, take, grab
 $tx := "Carefull, you'll burn yourself.  You"
 call print
 $tx := "decide that the sun is out of reach."
```

```
111     call print
112    endact
113
114    endobject
115
116    ;-------------------------------------
117
118    object island
119
120    name island, rock
121
122    initroom west_end_of_beach
123
124    code
125    endcode
126
127    action look, examine, search
128     $tx := "This appears to be a small rock island"
129     call print
130     $tx := "out beyond the surf."
131     call print
132    endact
133
134    endobject
135
136    ;-------------------------------------
137
138    object ocean
139
140    name ocean, water, sea, lagoon
141
142    initroom unused
143
144    code
145    ; so player can say PUSH THE CANOE IN THE WATER
146    endcode
147
148    action look, examine, search
149     $tx := "The sun sparkles on the blue waters of"
150     call print
151     $tx := "the Pacific as the waves gently lap"
152     call print
153     $tx := "ashore.  You see a speck on the horizon."
154     call print
155    endact
156
157    action get, take, grab
158     $tx := "You get a small amount in your hands,"
159     call print
160     $tx := "but all it does is get your hands wet."
161     call print
162    endact
163
164    action drink, swallow
165     $tx := "Yeck, salty!"
166     call print
167    endact
```

```
168
169   action swim
170    call NoSwim
171   endact
172
173   endobject
174
175   ;-----------------------------------
176
177   object plant
178
179   name plant, plants, grass
180
181   initroom unused
182
183   code
184   endcode
185
186   action look, examine, search
187    $tx := "The thin green beach grass waves about"
188    call print ;
189    $tx := "in the slight breeze."
190    call print
191   endact
192
193   action get, take, grab
194    $tx := "It is strongly rooted, and won't come"
195    call print
196    $tx := "up.  You decide to leave it alone."
197    call print
198   endact
199
200   endobject
201
202   ;-----------------------------------
203
204   object dunes
205
206   name dunes, sanddunes
207
208   initroom meadow
209
210   code
211   endcode
212
213   action look, examine, search
214    $tx := "The sand dunes are tall and rounded."
215    call print
216    $tx := "They look like they would be fun."
217    call print
218   endact
219
220   action get, take, grab
221    $tx := "You're here, and they are over there."
222    call print
223    $tx := "Now, how do you expect to get them?"
224    call print
```

```
225   endact
226
227   endobject
228
229   ;-----------------------------------
230
231   object hole
232
233   name hole, opening
234
235   initroom in_the_cave
236
237   code
238   endcode
239
240   action look, examine, go
241    $tx := "It's pretty high up.  Too high to jump."
242    call print
243   endact
244
245   endobject
246
247   ;-----------------------------------
248
249   object roof
250
251   name roof
252
253   initroom top_of_the_shack
254
255   code
256   endcode
257
258   action look, examine, search
259    $tx := "Nothing special about this roof.  Seems"
260    call print
261    $tx := "pretty secure."
262    call print
263   endact
264
265   action get, take, grab
266    $tx := "Everything's nailed down!"
267    call print
268   endact
269
270   endobject
271
272   ;-----------------------------------
273
274   object window
275
276   name window
277
278   initroom in_the_shack
279
280   code
281   endcode
```

```
282
283   action look, examine
284    $tx := "The glass is missing from the window"
285    call print
286    $tx := "frame.  Light streams through."
287    call print
288   endact
289
290   endact
291
292   endobject
293
294   ;-------------------------------------
295
296   object table
297
298   name table
299
300   initroom in_the_shack
301
302   code
303   endcode
304
305   action look, examine
306    $tx := "It's a plain table, of simple design."
307    call print
308    $tx := "You can see it's nailed to the floor."
309    call print
310   endact
311
312   action get, take, grab
313    $tx := "You can't.  It's nailed down!"
314    call print
315   endact
316
317   endobject
318
319   ;-------------------------------------
320
321   object tree
322
323   name tree, palm
324
325   adj palm, tall
326
327   initroom top_of_the_cliff
328
329   code
330   endcode
331
332   action look, examine
333    $tx := "It's slick brown bark leads upward to"
334    call print
335    $tx := "the green fronds at the top."
336    call print
337   endact
338
```

```
339   action climb
340    $tx:="You try, but you slide back down."
341    call print
342   endact
343
344   action chop, cut
345    $tx:="You'll need something sharp to do that."
346    call print
347   endact
348
349   action burn, light
350    call NoBurn
351   endact
352
353   endobject
354
355   ;-------------------------------------
356
357   object boughs
358
359   name boughs, fronds
360
361   initroom unused
362
363   code
364   endcode
365
366   action look, examine, search
367    $tx := "The wide green fronds look smooth and"
368    call print
369    $tx := "feathery.  Quite comfy!"
370    call print
371   endact
372
373   action get, take, grab
374    $tx := "Leave the greenery alone."
375    call print
376   endact
377
378   endobject
379
380   ;-------------------------------------
381
382   object shack
383
384   name shack
385
386   adj old, wood
387
388   initroom meadow
389
390   code
391   endcode
392
393   action look, examine
394    $tx:="It is a strange old rundown shack.  It"
395    call print
```

```
396     $tx := "has a single open doorway."
397     call print
398   endact
399
400   action light, burn
401     call NoBurn
402   endact
403
404   endobject
405
406   ;-----------------------------------
407
408   object ladder1
409
410   name ladder
411
412   initroom unused
413
414   code
415   endcode
416
417   action look, examine
418     $tx:="It looks old and weather-beaten.  But it"
419     call print
420     $tx:="should hold you."
421     call print
422   endact
423
424   action get, take, grab
425     $tx := "Leave it there, so you can climb back"
426     call print
427     $tx := "down it.  No sense bringing it up here."
428     call print
429   endact
430
431   endobject
432
433   ;-----------------------------------
434
435   object rocks
436
437   name rocks
438
439   initroom unused
440
441   code
442   endcode
443
444   action look, examine
445     $tx := "The rocks appear to be of volcanic"
446     call print
447     $tx := "origin."
448     call print
449   endact
450
451   action get, take, grab
452     $tx := "Sorry, they're stuck there."
```

```
453     call print
454    endact
455
456    endobject
457
458    ;-------------------------------------
459
460    object boulder
461
462    name boulder, rock
463
464    attrib
465     moved N
466    endattrib
467
468    initroom by_the_boulder
469
470    code
471    endcode
472
473    action look, examine
474     $tx := "It is big, over ten feet high.  It sits"
475     call print
476     if boulder is moved then
477      $tx := "beside the opening to the cave."
478      call print
479     else
480      $tx := "directly in front of a cave opening,"
481       call print
482      $tx := "barring your way.  But a good hard push"
483       call print
484     $tx := "might topple it over on its side."
485     call print
486     endif
487    endact
488
489    action get, take, carry
490     $tx := "It weighs tons!  You can't carry it."
491     call print
492    endact
493
494    action move, slide, push, pull, move
495     if energy > 0 then
496      $tx := "It rolls over, revealing the entrance."
497      call print
498      set boulder, moved
499      energy := 0
500      if 01ladder in thisroom then
501       directions by_the_boulder, n e u
502      else
503       directions by_the_boulder, n e
504      endif
505      link by_the_boulder, e, in_the_cave
506      RoomNumber := 11
507      click 34, 57,35,  141,91, SeeBoulder
508      click 35, 142,26, 172,78, SeeCave
509      placeobj cave, thisroom
```

```
510     call ReDrawScreen
511    else
512     $tx := "It shifts slightly, but rolls back."
513     call print
514    endif
515   endact
516
517   action climb
518    $tx := "You can't get a foothold."
519    call print
520   endact
521
522   endobject
523
524   ;-----------------------------------
525
526   object island1
527
528   name boulder, rock, island
529
530   attrib
531   endattrib
532
533   initroom top_of_the_boulder
534
535   code
536   endcode
537
538   action look, examine
539    $tx := "All you can see is the boulder that you"
540    call print
541    $tx := "are standing on, and the island below."
542    call print
543    $tx := "Out on the ocean, you can see a speck on"
544    call print
545    $tx := "the horizon.  Is it a ship?"
546    call print
547   endact
548
549   endobject
550
551   ;-----------------------------------
552
553   object cave
554
555   name cave, wall, pictures, picture, writing, writings, carving,
          carvings, message
556
557   initroom unused
558
559   code
560   endcode
561
562   action look, examine, read
563    if player in in_the_cave
564     $tx := "The crude pictures depict men eating"
565     call print
```

```
566     $tx := "raw human flesh.  This disgusting"
567     call print
568     $tx := "display shows them chewing on human"
569     call print
570     $tx := "arms and legs.  You feel sick to your"
571     call print
572     $tx := "stomach!"
573     call print
574     elsif player in rock_room
575     $tx := "\34 Buy THE VISIONARY PROGRAMMERS HANDBOOK"
576     call print
577     $tx := "by John Olsen, and learn how to write"
578     call print
579     $tx := "games like this using VISIONARY, the"
580     call print
581     $tx := "Aegis Interactive Gaming Language.\34"
582     call print
583    else ; if player outside, looking in cave
584     $tx := "It looks like there's enough light"
585     call print
586     $tx := "inside to see fairly well.  And you can"
587     call print
588     $tx := "easily fit through the opening now."
589     call print
590    endif
591   endact
592
593   endobject
594
595   ;-------------------------------------
596
597   object fissure
598
599   name fissure, opening
600
601   attrib
602   endattrib
603
604   initroom rock_room
605
606   code
607   endcode
608
609   action look, examine
610    $tx := "It's too high up to reach, and too"
611    call print
612    $tx := "small to crawl through, but it allows"
613    call print
614    $tx := "enough light into the room to let you"
615    call print
616    $tx := "see the writing on the wall clearly."
617    call print
618   endact
619
620   endobject
621
622   ;-------------------------------------
```

```
623
624   object rowboat
625
626   name rowboat, boat
627
628   adj battered, row
629
630   initroom west_end_of_beach
631
632   code
633    ; so player can say LOOK ROWBOAT
634   endcode
635
636   action look, examine, search
637    $tx := "The oars are missing, and there's a"
638    call print
639    $tx := "gaping hole in the rear."
640    call print
641   endact
642
643   action take, grab, move, slide, push, pull
644    $tx := "It's too heavy to move."
645    call print
646   endact
647
648   action get, sit
649    if prep is "in" then
650     call SitBoat
651    elsif prep is "into" then
652     call SitBoat
653    elsif prep is "inside" then
654     call SitBoat
655    else
656     $tx := "It's too heavy to move."
657     call print
658    endif
659   endact
660
661   action row, paddle
662    $tx := "This boat has a hole in the back.  It's"
663    call print
664    $tx := "not going to float."
665    call print
666   endact
667
668   action light, burn
669    call NoBurn
670   endact
671
672   endobject
673
674   ;-------------------------------------
675
676   object canoe
677
678   name canoe, boat
679
```

```
680   adj native
681
682   attrib
683    InWater N
684   endattrib
685
686   initroom east_end_of_beach
687
688   code
689   endcode
690
691   action look, examine, search
692    $tx := "The oars are missing."
693    call print
694   endact
695
696   action take, grab
697    $tx := "It's too heavy to carry."
698    call print
699   endact
700
701   action get, sit
702    if prep is "in" then
703     call SitCanoe
704    elsif prep is "into" then
705     call SitCanoe
706    elsif prep is "inside" then
707     call SitCanoe
708    else
709     $tx := "It's too heavy to carry."
710     call print
711    endif
712   endact
713
714   action slide, move, push, pull, put
715    if canoe is InWater then
716     call OK
717     RoomNumber := 2
718     click 34, 33,68,  113,107, SeeCanoe
719     click 35, 87,53,  137,88,  SeeCanoe
720     call ReDrawScreen
721     unset canoe, InWater
722    else
723     call OK
724     set canoe, InWater
725     RoomNumber := 4
726     click 34, 70,39,  140,63,  SeeCanoe
727     click 35, 118,23, 170,50,  SeeCanoe
728     call ReDrawScreen
729    endif
730   endact
731
732   action row, paddle
733    if canoe is InWater then
734     if player has 13shovel then
735      timer := 0 ; so no further messages will be printed from
          NPC.obj
```

```
736       $tx := "Using the shovel, you row the canoe out"
737       call print
738       $tx := "to the freighter.  You are taken aboard"
739       call print
740       $tx := "and enjoy your leisurely trip back to"
741       call print
742       $tx := "civilization and your job at the FBI."
743       call print
744       $tx := "Congratulations on escaping the island!"
745       call print
746       RoomNumber := 16
747       call ReDrawScreen ; show EndTitle
748       load song 1, "Cannibal:audio/win.mus"
749       EnableMusic
750       play song 1 ; play music when you win
751       $tx := "You have won this sample adventure.  Now"
752       call print
753       $tx := "start writing your own adventure games"
754       call print
755       $tx := "with VISIONARY, the Aegis Interactive"
756       call print
757       $tx := "Gaming Language from Oxxi Aegis."
758       call print
759       CountLines := 1 ; to keep other mouse message from appearing
760       call Linefeed
761       TextColor := brown
762       $tx := "      Press mouse button to exit."
763       call print
764        while leftbutton = 1 do ; make sure button is up first
765        endwhile
766        while leftbutton = 0 do ; then if button is pushed down,
            exit
767        endwhile
768       MainLoop := 1
769      else
770       $tx := "You have no oars."
771       call print
772      endif
773     else
774      $tx := "Not while it is beached on the sand."
775      call print
776     endif
777   endact
778
779   action burn, light
780    call NoBurn
781   endact
782
783   endobject
784
785   ;-------------------------------------
786
787   object boards
788
789   name boards, nails
790
791   initroom top_of_the_shack
```

```
792
793   code
794   endcode
795
796   action get, take, grab, pry, remove, pull
797    if player has 14hammer then
798     $tx := "The nails are securely fastened deep in"
799     call print
800     $tx := "the boards.  They won't budge."
801     call print
802    else
803     $tx := "You'd need a hammer for that job."
804     call print
805    endif
806   endact
807
808   action look, examine
809     $tx := "The wide old nails penetrate deep into"
810     call print
812     $tx := "the dry old boards."
812     call print
813   endact
814
815   action light, burn
816    call NoBurn
817   endact
818
819   endobject
820
821   ;-------------------------------------
822
```

The NPC.OBJ File

```
1
2   ;---------------- NPC.obj ----------------
3
4   npc status
5
6   name status
7
8   initroom unused
9
10  code
11
12  call CannibalsArrive ; increment timer and see if Cannibals have
        arrived
13
14  if energy > 0 then
15   energy := energy - 1
16   if energy = 0 then
17    call Linefeed
18    $tx := "You feel a drain, as the quick energy"
19    call print
20    $tx := "from the Snicker passes."
21    call print
22   endif
23  endif
24
25  endcode
26
27  endNPC
28
29  ;---------------------------------------
30
```

Moveable1.OBJ

```
1
2   ;---------- Movable1.obj ----------
3
4   object 01ladder
5
6   name ladder
7
8   adj wood, wooden
9
10  attrib
11  endattrib
12
13  initroom meadow
14
15  code
16  endcode
17
18  action get, take, grab
19   call get
20  endact
21
22  action look, examine
23   $tx:="The old ladder looks weather-beaten."
24   call print
25   $tx:="But it should hold you."
26   call print
27  endact
28
29  action drop
30   call drop
31  endact
32
33  action put, set, lay, lean
34   if player has 01ladder then
35    $tx := "Use the mouse to drag it over."
36    call print
37   else ; 01ladder is at this location
38    $tx := "That's where it stays."
39    call print
40   endif
41  endact
42
43  action climb
44   if RoomNumber = 6 then
45    go top_of_the_shack
46   elsif RoomNumber = 7 then
47    go in_the_tree
48   elsif RoomNumber = 13 then
49    go rock_room
50   elsif RoomNumber = 9 then
51    go top_of_the_boulder
52   elsif RoomNumber = 11 then
53    go top_of_the_boulder
```

```
 54    else
 55     $tx:="You can't.  It's not leaning against"
 56     call print
 57     $tx := "anything."
 58     call print
 59    endif
 60   endact
 61
 62   endobject
 63
 64   ;-------------------------------------
 65
 66   object 02bottle
 67
 68   name bottle, potion, liquid
 69
 70   adj glass, sealed
 71
 72   attrib
 73    sealed Y
 74   endattrib
 75
 76   initroom east_end_of_beach
 77
 78   code
 79   endcode
 80
 81   action look, examine, search, view
 82    $tx:="You can see something inside the bottle."
 83    call print
 84    $tx:="You can't quite make out what it is."
 85    call print
 86   endact
 87
 88   action get, take, grab
 89    call get
 90   endact
 91
 92   action drop
 93    call drop
 94   endact
 95
 96   action open, unseal, uncork
 97    if player has 02bottle then
 98     $tx := "The seal is tight, and the bottle won't"
 99     call print
100     $tx := "open."
101     call print
102    else
103     call NoHave
104    endif
105   endact
106
107   action break, smash, hit
108    call BreakBottle
109   endact
110
```

```
111   endobject
112
113   ;-----------------------------------------
114
115   object 03paper
116
117   name paper
118
119   adj piece, old
120
121   initroom unused
122
123   code
124   endcode
125
126   action look, examine, read
127    $tx := "The old piece of paper says:"
128    call print
129    $tx := "Breaking this bottle will bring you"
130    call print
131    $tx := "luck.  Maybe good; maybe bad."
132    call print
133   endact
134
135   action get, take, grab
136    call get
137   endact
138
139   action drop
140    call drop
141   endact
142
143   action burn, light
144    call NoBurn
145   endact
146
147   endobject
148
149   ;-----------------------------------------
150
151   object 04battery
152
153   name battery
154
155   adj dead, radio
156
157   attrib
158    found N
159   endattrib
160
161   initroom unused
162
163   code
164   endcode
165
166   action look, examine
167    $tx := "It's a dead old radio battery."
```

```
168    call print
169    $tx := "It's really useless."
170    call print
171   endact
172
173   action get, take, grab
174    call get
175   endact
176
177   action drop
178    call drop
179   endact
180
181   action put, insert
182    if player has 04battery then
183     if objnoun is 05radio then
184      if player has 05radio then
185       $tx := "But it's dead.  Only put a charged"
186       call print
187       $tx := "battery in the 2-way radio."
188       call print
189      else
190       call NoHave
191      endif
192     else
193      call CantDoThat
194     endif
195    else
196     call NoHave
197    endif
198   endact
199
200   endobject
201
202   ;--------------------------------------
203
204   object 05radio
205
206   name radio
207
208   adj 2-way, two-way
209
210   initroom in_the_shack
211
212   code
213   endcode
214
215   action look, examine
216    $tx := "This two-way radio looks to be in"
217    call print
218    $tx := "excellent shape.  But unfortunately,"
219    call print
220    $tx := "there is no battery in it."
221    call print
222   endact
223
224   action get, take, grab
```

```
225    call get
226   endact
227
228   action drop
229    call drop
230   endact
231
232   action use, play, turn
233    $tx := "It won't work without a battery."
234    call print
235   endact
236
237   endobject
238
239   ;-----------------------------------
240
241   object 06bar
242
243   name bar, snicker, snickers
244
245   adj candy
246
247   initroom in_the_tree
248
249   code
250   endcode
251
252   action look, examine
253    $tx := "Mmmmmm...  A Snickers candy bar!  It"
254    call print
255    $tx := "looks fresh and tasty."
256    call print
257   endact
258
259   action get, take, grab
260    call get
261   endact
262
263   action drop
264    call drop
265   endact
266
267   action eat, chew, consume, swallow, enjoy
268    call EatCandy
269   endact
270
271   endobject
272
273   ;-----------------------------------
274
275   object 07candle
276
277   name candle
278
279   adj half-burned
280
281   initroom in_the_shack
```

```
282
283   code
284   endcode
285
286   action look, examine
287    $tx := "The wax is hard, and the wick is black."
288    call print
289   endact
290
291   action get, take, grab
292    call get
293   endact
294
295   action drop
296    call drop
297   endact
298
299   action burn, light
300    call NoBurn
301   endact
302
303   endobject
304
305   ;----------------------------------------
306
307   object 08skull
308
309   name skull
310
311   adj bleached, human
312
313   initroom top_of_the_cliff
314
315   code
316   endcode
317
318   action look, examine
319    $tx := "This bleached old skull could be a"
320    call print
321    $tx := "remnent of some past cannibal feast."
322    call print
323   endact
324
325   action get, take, grab
326    call get
327   endact
328
329   action drop
330    call drop
331   endact
332
333   endobject
334
335   ;----------------------------------------
336
337
```

The Movable2.OBJ File

```
1
2   ;————————— Movable2.obj —————————
3
4   object 09seagull
5
6   name seagull, bird
7
8   adj dead
9
10  initroom top_of_the_boulder
11
12  code
13  endcode
14
15  action look, examine
16   $tx := "This dead old seagull is stiff and"
17   call print
18   $tx := "starting to smell a bit."
19   call print
20  endact
21
22  action get, take, grab
23   call get
24  endact
25
26  action drop
27   call drop
28  endact
29
30  action eat, consume, chew, swallow
31   $tx := "Yeck!  No way!  You'd rather starve!"
32   call print
33  endact
34
35  endobject
36
37  ;————————————————————————————
38
39  object 10coconut
40
41  name coconut
42
43  adj hard, old
44
45  initroom in_the_tree
46
47  code
48  endcode
49
50  action look, examine
51   $tx := "The hard old coconut is probably not"
52   call print
53   $tx := "edible; it looks hard and dried out."
```

```
 54    call print
 55   endact
 56
 57   action get, take, grab
 58    call get
 59   endact
 60
 61   action drop
 62    call drop
 63   endact
 64
 65   action break, open
 66    call BreakCoconut
 67   endact
 68
 69   action eat
 70    $tx := "Open it first..."
 71    call print
 72   endact
 73
 74   endobject
 75
 76   ;-----------------------------------------
 77
 78   object 11blade
 79
 80   name blade
 81
 82   adj shovel
 83
 84   initroom rock_room
 85
 86   code
 87   endcode
 88
 89   action look, examine
 90    $tx := "It's a wide shovel blade, without any"
 91    call print
 92    $tx := "handle."
 93    call print
 94   endact
 95
 96   action get, take, grab
 97    call get
 98   endact
 99
100   action drop
101    call drop
102   endact
103
104   action put, connect, stick, push
105    if objnoun is 12handle then
106     if player has 12handle then
107      $tx := "OK.  The completed shovel looks useful."
108      call print
109      ObjNum := 12
110      call DestroyObject
```

```
111       ObjNum := 11
112       call DestroyObject
113       ObjNum := 13
114       mode 2, draw
115       color 2, ObjNum
116       rect 2, x,y, x,y ; put new object in inventory array
117       mode 0, overlay
118       copy 2,ObjNum * 15 - 15,183,ObjNum * 15 - 1,199,0,x * 16 +
             266,y * 18 + 17
119       placeobj 13shovel, thisroom
120       grab 13shovel
121      else
122       call NoHave
123      endif
124     else
125      call CantDoThat
126     endif
127    endact
128
129    endobject
130
131    ;-----------------------------------
132
133    object 12handle
134
135    name handle
136
137    adj shovel
138
139    attrib
140     Found N
141    endattrib
142
143    initroom unused
144
145    code
146    endcode
147
148    action look, examine
149     $tx := "It's a long wood handle that will fit"
150     call print
151     $tx := "into a shovel blade."
152     call print
153    endact
154
155    action get, take, grab
156     call get
157    endact
158
159    action drop
160     call drop
161    endact
162
163    action put, connect, stick, push
164     if objnoun is 11blade then
165      if player has 11blade then
166       $tx := "OK.  The completed shovel looks useful."
```

```
167      call print
168      ObjNum := 12
169      call DestroyObject
170      ObjNum := 11
171      call DestroyObject
172      ObjNum := 13
173      mode 2, draw
174      color 2, ObjNum
175      rect 2, x,y, x,y ; put new object in inventory array
176      mode 0, overlay
177      copy 2,ObjNum * 15 - 15,183,ObjNum * 15 - 1,199,0,x * 16 +
            266,y * 18 + 17
178      placeobj 13shovel, thisroom
179      grab 13shovel
180     else
181      call NoHave
182     endif
183    else
184     call CantDoThat
185    endif
186   endact
187
188   action light, burn
189    call NoBurn
190   endact
191
192   endobject
193
194   ;-------------------------------------
195
196   object 13shovel
197
198   name shovel
199
200   initroom unused
201
202   code
203   endcode
204
205   action look, examine
206    $tx := "It's a pretty handy looking shovel."
207    call print
208   endact
209
210   action get, take, grab
211    call get
212   endact
213
214   action drop
215    call drop
216   endact
217
218   action use
219    $tx := "Do you mean dig?  Please be specific."
220    call print
221   endact
222
```

```
endobject

;-------------------------------------

object 14hammer

name hammer

adj old

initroom in_the_cave

code
endcode

action look, examine
 $tx := "It's a quite servicable old hammer."
 call print
endact

action get, take, grab
 call get
endact

action drop
 call drop
endact

action use
 $tx := "Please be more specific.  For example,"
 call print
 $tx := "PULL THE NAILS or BREAK THE BOTTLE."
 call print
endact

endobject

;-------------------------------------

object 15chisel

name chisel

adj dull

attrib
 found N
endattrib

initroom unused

code
endcode

action look, examine
  $tx := "It's an old chisel, somewhat rusty."
 call print
```

```
280   endact
281
282   action get, take, grab
283    call get
284   endact
285
286   action drop
287    call drop
288   endact
289
290   action use
291    $tx := "Please be more specific.  For example,"
292    call print
293    $tx := "BREAK THE COCONUT."
294    call print
295   endact
296
297   endobject
298
299   ;-----------------------------------
300
301   object 16driftwood
302
303   name driftwood, wood
304
305   adj dry
306
307   attrib
308    found N
309   endattrib
310
311   initroom unused
312
313   code
314   endcode
315
316   action look, examine
317    $tx := "It's weathered driftwood, quite dry."
318    call print
319    $tx := "It would make great kindling..."
320    call print
321   endact
322
323   action get, take, grab
324    call get
325   endact
326
327   action drop
328    call drop
329   endact
330
331   action burn, light
332    call NoBurn
333   endact
334
335   endobject
336
```

```
337   ;-----------------------------------------
338
339   object 17gun
340
341   name gun
342
343   adj flare
344
345   attrib
346    found N
347   endattrib
348
349   initroom unused
350
351   code
352   endcode
353
354   action look, examine
355    $tx := "It's an emergency flare gun, but there's"
356    call print
357    $tx := "no flare in it."
358    call print
359   endact
360
361   action get, take, grab
362    call get
363   endact
364
365   action drop
366    call drop
367   endact
368
369   action use, shoot
370     $tx := "But it has no flare in it.  You can't."
371    call print
372   endact
373
374   endobject
375
376   ;-----------------------------------------
377
378   object 18matches
379
380   name matches, book, match
381
382   adj wet
383
384   attrib
385    found N
386   endattrib
387
388   initroom unused
389
390   code
391   endcode
392
393   action look, examine
```

```
394     $tx := "The book of matches is wet..."
395     call print
396   endact
397
398   action get, take, grab
399     call get
400   endact
401
402   action drop
403     call drop
404   endact
405
406   action burn, light, strike
407     $tx := "Nothing...  Wet matches don't burn."
408     call print
409   endact
410
411   action dry
412     $tx := "You might try leaving them somewhere"
413     call print
414     $tx := "in the sun for a while..."
415     call print
416   endact
417
418   endobject
419
420   ;-----------------------------------
421
422   object 19flyer
423
424   name flyer
425
426   adj advertising
427
428   initroom top_of_the_shack
429
430   code
431   endcode
432
433   action look, examine, read
434     $tx := "The advertising flyer says:"
435     call print
436     $tx := "\34 For more information on how this game"
437     call print
438     $tx := "was made, read THE VISIONARY PROGRAMMERS"
439     call print
440     $tx := "HANDBOOK by John Olsen.  Available from"
441     call print
442     $tx := "your computer store, or Oxxi Aegis."
443     call print
444   endact
445
446   action get, take, grab
447     call get
448   endact
449
450   action drop
```

```
451     call drop
452    endact
453
454    action burn, light
455     call NoBurn
456    endact
457
458    endobject
459
460    ;-------------------------------------
```

The MainLoop.SUB File

```
1
2   ;--------------- MainLoop.SUB ---------------
3
4   sub MainLoop
5
6   ;=============================================================================
7
8   WHILE MainLoop = 0 DO
9
10  CountLines := -1
11  call LineFeed
12
13  return := 1
14  TextPosition := 9
15  $sentence := ""
16  $return := "\r"
17  $backspace := "\b"
18
19  color 0, green   ; make cursor green
20  text 0,9,192,"~" ; character modified to be cursor
21
22  ;-----------------------------------
23
24  while return # 0 do
25
26   getchar $letter
27   length $letter, temp ; temp = 0 if NO letter pressed, temp = 1
         if letter
28   value $letter, val
29   temp := temp * (val < 127) ; temp is 1 only if keypress is
         alpha/numeric
30   compare $letter, $return, return
31   compare $letter, $backspace, backspace
32
33  if ButtonUsed = 1 or ButtonUsed > 3 then
34   return := 0 ; set to zero so as to erase cursor and exit loop
35   copy 2, x1,y1,x2,y2, 0, x,y ; draw button in down position
36    pause 15 ; to keep mouse pointer from temporarily freezing
37    while leftbutton = 1 do    ; wait till button released
38    endwhile
39   y1 := y1 + offset
40   y2 := y2 + offset
41   offset := -13
42  endif
43
44  if ButtonUsed = 1 then
45   copy 2, x1, y1, x2, y2, 0, x,y ; draw button popping up
46  endif
47
48   if return = 0 then ; return was pressed
49    color 0,white
50    mode 0, draw
51    rect 0, TextPosition,184, TextPosition + 5,192 ; erase cursor
```

```
52     mode 0,overlay
53    elsif backspace = 0 then ; backspace was pressed
54     length $sentence, sentence
55     if sentence = 0 then ; The sentence has length zero, so do
           nothing!
56     else
57      sentence := sentence - 1
58      TextPosition := TextPosition - 6
59      left $sentence, sentence, $sentence
60      color 0,white
61      mode 0,draw
62      rect 0, TextPosition,184, TextPosition + 11,192 ; erase
           letter & cursor
63      mode 0,overlay
64      color 0,green
65      text 0,TextPosition,192,"~" ; type cursor
66     endif
67    elsif TextPosition > 240 then ; outside text window
68    elsif temp > 0 then ; accept the keypress
69     $sentence := "@$sentence @$letter" ; add letter to sentence
70     color 0,white
71     mode 0, draw
72     rect 0, TextPosition,184, TextPosition + 5,192 ; erase cursor
73     color 0,green
74     mode 0,overlay
75     text 0,TextPosition,192,"@$letter ~" ; type letter and cursor
76     TextPosition := TextPosition + 6
77    endif
78
79   if ButtonUsed > 0 then
80    ; blank any previous input typed before typing button contents
81    mode 0,draw ; or overlay
82    color 0,white
83    rect 0,    9,185,    249,192
84    color 0,green
85    call PrintText ; echo the button name to the text window
86    $sentence := $tx
87    return := 0
88   endif
89
90   readbuttons ; check for mouse button presses between letters,
         and go there
91   readbuttons empty ; in case further buttons were pressed while
         away
92   if ButtonUsed = 9 then
93    return := 0
94   endif
95
96   endwhile ; end of input loop, ie. return = 0
97
98   ;-------------------------------------
99
100  if ButtonUsed < 9 then ; ..................................
101
102  color 0, blue
103
104  compare $sentence, "load", dummy
```

```
105   if dummy = 0 then
106    ButtonUsed := 4 ; so that player can click or type LOAD
107    t \f
108    t
109    t
110    t
111    t
112    t
113    t
114    t
115    t
116    t
117    t                                Please Wait for Loading
118    x1 := 137
119    y1 := 151
120    x2 := 197
121    y2 := 163
122    x  := 259
123    y  := 151
124    Screenmode text ; switch to black text screen with white
         writing
125   endif
126
127   compare $sentence, "save", dummy
128   if dummy = 0 then
129    ButtonUsed := 5 ; so that player can click or type SAVE
130    t \f
131    t
132    t
133    t
134    t
135    t
136    t
137    t
138    t
139    t
140    t                                Please Wait for Saving
141    x1 := 198
142    y1 := 151
143    x2 := 258
144    y2 := 163
145    x  := 259
146    y  := 169
147    unload screen 24
148    Screenmode text ; switch to black text screen with white
         writing
149   endif
150
151   compare $sentence, "quit", temp
152   if temp = 0 then
153    MainLoop := 1
154   else
155    ghost "@$sentence" turn
156    if error > 0 then
157     if ButtonUsed > 3 then
158      $tx := "SAVE GAME ERROR: play at your own risk."
159     else
```

```
160       $tx := $lasterror
161      endif
162      call print
163     endif
164    endif
165
166    if ButtonUsed > 3 and ButtonUsed < 9 then
167     t \f
168     show screen 0
169     ScreenMode graphics
170     create screen 24, 55, 78, 5, lores ; for ladder overlays
171     mask 24
172     mode 0, draw
173     copy 2, x1, y1, x2, y2, 0, x,y ; draw button popping up
            (load/save)
174        load screen 1, "ram:loc@RoomNumber" ; try to load it, see
             if it's there
175        if error > 0 then                    ; if it's not then
176         $device := "Cannibal:video/"        ; set path name to video/
177        else                                 ; otherwise
178         $device := "ram:"                   ; set path name to ram:
179        endif
180    endif
181
182    endif ; ..................................................
183
184    ButtonUsed := 0
185
186    ENDWHILE ; end of main loop
187
188    ; ============================================================
189
190    DOS "delete >NIL: ram:loc#?" ; clear ram: after game ends
191
192    quit
193
194    endsub
195
196    ;-------------------------------------
197
```

The GetDrop.SUB File

```
1
2   ;--------------- GetDrop.SUB ---------------
3
4   sub Get
5    $tx := "Use the mouse to drag it over to the"
6    call print
7    $tx := "inventory window.  If there are more"
8    call print
9    $tx := "than 5 objects at this location, you"
10   call print
11   $tx := "will have to press the up and down"
12   call print
13   $tx := "arrows to scroll the objects."
14   call print
15  endsub
16
17  ;---------------------------------------
18
19  sub Drop
20   $tx := "Use the mouse to drag it out of the"
21   call print
22   $tx := "inventory window, and into the location"
23   call print
24   $tx := "window."
25   call print
26  endsub
27
28  ;---------------------------------------
29
30  sub GetObject
31
32  Pic := ( MouseY - 21 ) / 18 ; returns value 0-4 for picture
        chosen
33  ChosenPic := Pic + SBPosition ; returns value 0 - (MaxMov - 1)
        (0-18)
34  pixel 2, ChosenPic,2,ObjNum ; read object number of chosen pic
        into ObjNum
35
36  IF ObjNum > 0 THEN ; if the spot clicked on is not empty then
37
38   mode 2, draw
39   color 2, white
40   rect 2, 285,183, 299,199 ; copy white to background in hidden
        buffer
41
42   OldX := 235              ; old x value from where object is coming
43   OldY := 21 + 18 * Pic ; old y value from where object is coming
44   temp := OldY
45   NewX := MouseX - 8 ;  value set early, in case of a really
        quick click
46   NewY := MouseY - 9 ;                that bypasses the while
        loop below
47
```

```
48    pause 15 ; to keep blitter happy and avoid temporary freeze ups
49
50    While Leftbutton = 1 do ; while button is held down, move object
51     if oldX # MouseX - 8 or oldY # MouseY - 9 then ; if mouse
          moved
52      NewX := MouseX - 8 ; offset by -8 so you are grabbing the
          obj center
53      NewY := MouseY - 9 ; offset by -9 so you are grabbing the
          obj center
54      mode 0, draw
55      copy 2, 285,183, 299,199, 0, oldX,oldY ; restore old
          background
56      copy 0, NewX, NewY, NewX + 14, NewY + 16, 2, 285,183 ; copy
          bkgd
57      mode 0, overlay
58      copy 2, ObjNum * 15 - 15,183, ObjNum * 15 - 1,199, 0, NewX,
          NewY
59      oldX := NewX
60      oldY := NewY
61     endif
62    endwhile
63
64    mode 0, draw
65    copy 2, 285,183, 299,199, 0, oldX,oldY ; restore old background
66    mode 0, overlay
67
68    x1 := NewX - 235              ; x distance that object was moved
69    x1 := (x1 > 0) * 2 * x1 - x1 ; absolute value of x1
70    y1 := NewY - temp             ; y distance that object was moved
71    y1 := (y1 > 0) * 2 * y1 - y1 ; absolute value of y1
72
73    if x1 < 8 and y1 < 8 then; moved very little, so player wanted
          to EXAMINE
74     copy 2, ObjNum * 15 - 15,183, ObjNum * 15 - 1,199, 0,
          235,temp ; put it back
75     ObjName ObjNum, $temp ; put the name of ObjNum in $temp so we
          can...
76     $tx := "examine the @$temp"
77     ButtonUsed := 2
78    elsif NewX > 257 and NewY < 45 then ; inv window
79     if items < 7 then ; if there's room in inventory, add the
          object
80      temp1 := 0 ; 3 spaces across inventory window
81      temp2 := 0 ; 2 spaced down inventory window
82       while temp2 < 2
83        while temp1 < 3
84         pixel 2, temp1, temp2, Slot ; see if inventory slot is
           taken
85         if Slot = 0 then ; empty slot found
86          x := temp1 ; remember the location of the empty slot
87          y := temp2 ; remember the location of the empty slot
88          temp1 := 3 ; force the loop to exit
89          temp2 := 2 ; force the loop to exit
90         endif
91         temp1 := temp1 + 1
92        endwhile
93        temp2 := temp2 + 1
```

```
 94        temp1 := 0
 95       endwhile
 96      copy 2,ObjNum * 15 - 15,183,ObjNum * 15 - 1,199,0,x * 16 +
           266,y * 18 + 17
 97      ; put object ObjNum in inventory window
 98      mode 2, draw
 99      color 2, ObjNum ; set value, before writing to array
100      rect 2, x, y, x, y; write ObjNum to array
101      grab ObjNum
102      if ObjNum = 1 then ; if grab ladder we must also move the
           BIG ladder
103       if RoomNumber = 6 then ; by shack
104        directions meadow, n s e w
105        call ReDrawScreen ; so as to make the ladder disappear
106       elsif RoomNumber = 7 then ; by tree
107        directions top_of_the_cliff, w
108        call ReDrawScreen
109       elsif RoomNumber = 9 then ; by upright boulder
110        directions by_the_boulder, n
111        call ReDrawScreen
112       elsif RoomNumber = 11 then
113        directions by_the_boulder, n e
114        call ReDrawScreen
115       elsif RoomNumber = 13 then ; by hole in cave
116        directions in_the_cave, w
117        call ReDrawScreen
118       endif
119      endif
120      mode 2, draw
121      copy 2, ChosenPic + 1,2,MaxMov,2, 2, ChosenPic,2 ; slide up
           array
122      mode 23, draw
123      copy 23, 0,ChosenPic * 18 + 18, 14,360, 23, 0,ChosenPic * 18
           ; slide up obj
124      ObjTotal := ObjTotal - 1
125      call DisplaySB ; show new scrollbar with object missing
126      call CannibalsArrive ; increment timer and check for cannibals
127     else ; if over inventory limit, put back in scrollbar
128      copy 2, ObjNum * 15 - 15,183, ObjNum * 15 - 1,199, 0,
           235,temp
129      ; put object ObjNum back in scrollbar at location pic
130     endif
113    else ; if object released outside inventory window, put in
           scrollbar
123     copy 2, ObjNum * 15 - 15,183, ObjNum * 15 - 1,199, 0, 235,temp
133     ; put object ObjNum back in scrollbar at location pic
134    endif
135
136   ENDIF
137
138   endsub
139
140   ;-----------------------------------
141
142   sub DropObject
143
144   x := ( MouseX - 266 ) / 16 ; returns value 0-2
```

```
145   y := ( MouseY - 17 ) / 18 ;  returns value 0-1
146   pixel 2, x,y,ObjNum ; read object number of chosen pic into
          ObjNum
147
148   IF ObjNum > 0 THEN ; if the spot clicked on is not empty then
149
150    mode 2, draw
151    color 2, white
152    rect 2, 285,183, 299,199 ; copy white to background in hidden
          buffer
153
154    OldX := 266 + 16 * x  ; old x value from where object is
          coming
155    OldY := 17 + 18 * y   ; old y value from where object is
          coming
156    temp1 := OldX
157    temp2 := OldY
158    NewX := MouseX - 8 ; read these values early, in case player
          clicks so
159    NewY := MouseY - 9 ;     fast that the following while loop is
          skipped
160
161    pause 15 ; so that temporary freeze ups are avoided
162
163    While Leftbutton = 1 do ; while button is held down, move
          object
164     if OldX # MouseX - 8 or OldY # MouseY - 9 then ; if mouse
          moved
165      NewX := MouseX - 8 ; offset by -8 so you are grabbing the
          obj center
166      NewY := MouseY - 9 ; offset by -9 so you are grabbing the
          obj center
167      mode 0, draw
168      copy 2, 285,183, 299,199, 0, OldX,OldY ; restore old
          background
169      copy 0, NewX, NewY, NewX + 14, NewY + 16, 2, 285,183 ; copy
          new background
170      mode 0, overlay
171      copy 2, ObjNum * 15 - 15,183, ObjNum * 15 - 1,199, 0, NewX,
          NewY
172      OldX := NewX
173      OldY := NewY
174     endif
175    endwhile
176
177    mode 0, draw
178    copy 2, 285,183, 299,199, 0, oldX,oldY ; restore old
          background before...
179    mode 0, overlay
180
181    if NewX < 260 then ; put object in scrollbar
182     mode 2, draw
183     color 2, 0
184     rect 2, x, y, x, y; write zero (black) (empty) to array
185     drop ObjNum
186      if ObjNum = 1 then ; if dropped ladder
187       if RoomNumber = 6 then
```

```
188       directions meadow, n s e w u
189       link meadow, u, top_of_the_shack
190       call ReDrawScreen
191      elsif RoomNumber = 7 then
192       directions top_of_the_cliff, u w
193       link top_of_the_cliff, u, in_the_tree
194       call ReDrawScreen
195      elsif RoomNumber = 9 then
196       directions by_the_boulder, n u
197       link by_the_boulder, u, top_of_the_boulder
198       call ReDrawScreen
199      elsif RoomNumber = 11 then
200       directions by_the_boulder, n e u
201       link by_the_boulder, u, top_of_the_boulder
202       call ReDrawScreen
203      elsif RoomNumber = 13 then
204       directions in_the_cave, w u
205       link in_the_cave, u, rock_room
206       call ReDrawScreen
207      endif
208     endif
209    call AddObject
210    call CannibalsArrive ; increment timer and check for cannibals
211   else ; put object back in inventory
212    copy 2,ObjNum * 15 - 15,183,ObjNum * 15 - 1,199,0,temp1,temp2
213    temp1 := NewX - temp1                    ; x distance mouse
          was moved
214    temp1 := (temp1 > 0) * 2 * temp1 - temp1 ; absolute value of
          temp1
215    temp2 := NewY - temp2                    ; y distance mouse
          was moved
216    temp2 := (temp2 > 0) * 2 * temp2 - temp2 ; absolute value of
          temp2
217    if temp1 < 8 and temp2 < 8 then; moved very little, so
          assume EXAMINE
218     ObjName ObjNum, $temp ; store name of ObjNum in $temp so we
          can send msg
219     $tx := "examine the @$temp"
220     ButtonUsed := 2
221    endif
222   endif
223
224  ENDIF
225
226  endsub
227
228  ;-------------------------------------
229
230  sub ReDrawScrollBar
231
232  mode 23, draw
233  color 23, white
234  rect 23, 0,0, 14, 360 ; blank out previous contents of scroll
          bar
235
236  ObjNum := 1 ; start with object number 1 (ladder)
237  ObjTotal := 0 ; start with picture position 0, top of scrollbar
```

```
238
239   while ObjNum < MaxMov + 1 do ; loop through all 19 objects,
          put in scrollbar
240    if ObjNum in thisroom then
241      mode 23, overlay
242      copy 2, ObjNum * 15 - 15,183, ObjNum * 15 - 1,199,  23,  0,
          ObjTotal * 18
243      mode 2, draw                    ; save the ObjNum in an
          array, so later
244      color 2, ObjNum                 ; when the player clicks on
          an object in
245      rect 2, ObjTotal,2, ObjTotal,2 ; the scrollbar, we will know
          which one
246      ObjTotal := ObjTotal + 1        ; it is.  Works as long as
          ObjNum < 32
247    endif
248    color 2, 0
249    rect 2, ObjTotal,2, MaxMov,2 ; zero out rest of array
250    ObjNum := ObjNum + 1
251   endwhile ; example: 7 objects, stored in 0-6, ObjTotal = 7
252
253   call DisplaySB
254
255   endsub
256
257   ;-------------------------------------
258
259   sub DisplaySB
260
261   if SBPosition > ObjTotal - 5 then ; makes sure scroll bar is
          scrolled
262    SBPosition := ObjTotal - 5       ; to show as many objects as
          possible
263   endif
264   if SBPosition < 0 then
265    SBPosition := 0
266   endif
267
268   call DrawArrows ; puts scrollbar arrows on screen if needed
269
270   mode 0, draw ; draw scroll bar on screen
271   copy 23, 0, SBPosition * 18,   14, SBPosition * 18 + 89,    0,
          235, 21
272
273   endsub
274
275   ;-------------------------------------
276
277   sub AddObject
278
279   mode 2, draw
280   copy 2, 0,2,ObjTotal,2, 2, 1,2 ; slide array down one
281   color 2, ObjNum
282   rect 2, 0,2,0,2 ; add object to front of list
283
284   ObjTotal := ObjTotal + 1
285   mode 23, draw
```

```
286   copy 23, 0,0, 14, 342, 23, 0, 18 ; slide down objects on
          scrollbar
287   color 23, white
288   rect 23, 0, 0, 14, 17
289   mode 23, overlay
290   copy 2, ObjNum * 15 - 15,183,  ObjNum * 15 - 1,199,  23,   0, 0
291
292   SBPosition := 0
293
294   call DrawArrows ; puts scrollbar arrows on screen if needed
295
296   mode 0, draw ; draw scroll bar on screen
297   copy 23, 0, SBPosition * 18,   14, SBPosition * 18 + 89,    0,
          235, 21
298
299   endsub
300
301   ;-----------------------------------------
302
303   sub ClickUpArrow
304
305    while leftbutton ; move scrollbar as long as the mouse button
          is depressed
306     if UpArrowActive then
307      mode 0, draw
308      copy 2, 275, 139, 287, 149,   0,   236, 6 ; show up arrow
          depressed
309      pause 5 ; to keep keep mouse pointer from freezing
310      temp := 0
311      while temp < 18 do
312       temp := temp + 1
313       copy 23,0,SBPosition * 18 - temp,14,SBPosition * 18 + 89 -
          temp,0,235, 21
314      endwhile
315      SBPosition := SBPosition - 1
316      call DrawArrows ; draws arrows if appropriate, blanks it
          otherwise
317     endif
318    endwhile
319   readbuttons empty
320
321   endsub
322
323   ;-----------------------------------------
324
325   sub ClickDownArrow
326
327   while leftbutton ; keep scrolling while mouse button depressed
328    if DownArrowActive then
329     mode 0, draw
330     copy 2, 260, 139, 272, 149,   0,  236, 114 ; show down arrow
          depressed
331     pause 5 ; to keep keep mouse pointer from freezing
332     SBPosition := SBPosition + 1
333     temp := 18
334     while temp > 0 do
335      temp := temp - 1
```

```
336      copy 23,0,SBPosition * 18 - temp,14,SBPosition * 18 + 89 -
           temp,0,235, 21
337     endwhile
338     call DrawArrows ; draws arrows if appropriate, blanks it
          otherwise
339    endif
340   endwhile
341   readbuttons empty
342
343   endsub
344
345   ;-------------------------------------
346
347   sub DrawArrows
348
349   DownArrowActive := 0
350   UpArrowActive := 0
351   mode 0, draw ; in case we need to draw arrows on scrollbar
352
353   if SBPosition + SBSize < ObjTotal then ; more objects are below
354    DownArrowActive := 1
355    copy 2, 260, 126, 272, 136,   0,   236, 114 ; show down arrow
         normal
356   else
357    copy 2, 290, 126, 302, 136,   0,    236, 114 ; blank down
         arrow
358   endif
359
360   if SBPosition > 0 then ; more objects are above
361    UpArrowActive := 1
362    copy 2, 275, 126, 287, 136,   0,   236, 6 ; show up arrow
         normal
363   else
364    copy 2, 290, 126, 302, 136,   0,    236, 6 ; blank up arrow
365   endif
366
367   endsub
368
369   ;-------------------------------------
370
371   sub DestroyObject ; used for EAT SNICKER or BREAK BOTTLE
372
373   temp1 := 0 ; 3 spaces across inventory window
374   temp2 := 0 ; 2 spaced down inventory window
375
376    while temp2 < 2
377     while temp1 Å 3
378      pixel 2, temp1, temp2, Slot ; check contents of each slot
379      if Slot = ObjNum then ; object has been found
380       x := temp1
381       y := temp2
382       temp1 := 3
383       temp2 := 2
384      endif
385      temp1 := temp1 + 1
386     endwhile
387     temp2 := temp2 + 1
```

```
388     temp1 := 0
389    endwhile
390
391   mode 2, draw
392   color 2, 0
393   rect 2, x,y, x,y ; zero (black) out the inventory array element
394
395   mode 0, draw      ; blank out object in inventory window by
          painting
396   color 0, white    ; over it with a white rectangle
397   rect 0, 266 + 16 * x, 17 + 18 * y, 280 + 16 * x, 33 + 18 * y
398
399   drop ObjNum
400   placeobj ObjNum, unused
401
402   endsub
403
404   ;-------------------------------------
```

The Cannibal.SUB File

```
 1
 2  ;---------- Cannibal.SUB ----------
 3
 4  sub CannibalsArrive
 5
 6   timer := timer + 1
 7   TextColor := red
 8   if timer = 90 then
 9  call Linefeed
10    $tx := "The cannibals will land in only a few"
11    call print
12    $tx := "minutes!"
13    call print
14    ButtonUsed := 9
15   elsif timer = 97 then
16    call Linefeed
17    $tx := "The cannibals are nearly here."
18    call print
19    ButtonUsed := 9
20   elsif timer = 100 then
21    call Linefeed
22    $tx := "The cannibals have landed!"
23    call print
24    ButtonUsed := 9
25    EnableMusic
26    play song 0
27   elsif timer = 101 then
28    call Linefeed
29    $tx := "Here come the cannibals!  And they"
30    call print
31    $tx := "look hungry!"
32    call print
33    ButtonUsed := 9
34   elsif timer = 102 then
35    call Linefeed
36    $tx := "Those drums!  They're driving you mad!"
37    call print
38    ButtonUsed := 9
39   endif
40   TextColor := blue
41
42   if timer = 103 then
43    RoomNumber := 15 ; sitting in pot of water
44    call ReDrawScreen ; SHOW THE DEATH SCENE
45    mode 0, draw
46    color 0, white
47    rect 0, 235,21, 250,109 ; blank objects, so they aren't seen
          color cycling
48    rect 0, 266,17, 312,51  ; blank inventory so color cycling
          doesn't show
49    rect 0, 8,140,  248,192 ; blank text window, so cycling isn't
          seen in red
50    palette 0, 2, 5,8,15
```

```
51      palette 0, 4,7,7,7
52      palette 0, 5,6,6,6
53      palette 0, 6,4,4,4
54      palette 0, 7,3,3,3
55      palette 0, 9,5,8,15
56      palette 0, 11,8,13,15
57      palette 0, 12,5,8,15
58      palette 0, 14,4,4,6
59      palette 0, 15,5,5,7
60      palette 0, 16,6,6,9
61      palette 0, 19,11,9,7
62      palette 0, 20,11,5,2
63      palette 0, 21,8,3,2
64      palette 0, 22,4,2,1
65      palette 0, 23,0,0,0
66      palette 0, 25,14,11,9
67      cycle on
68      DisableMusic
69      play sound 0, 0,0,24,0 ; play bubbling non-stop
70      CountLines := -1 ; so all 6 lines will print without "more"
            message
71      $tx := "The cannibals have caught you!  They"
72      call print
73      $tx := "tie you up and dump you in a large iron"
74      call print
75      $tx := "pot.  A red fog swims before your eyes"
76      call print
77      $tx := "and you lose consciousness."
78      call print
79      call Linefeed
80      $tx := "The adventure is over.  You are dead."
81      call print
82      load sound 1, "Cannibal:audio/death.snd"
83      play sound 1, 1,1,64,0 ; play scream once
84      call ClearButtons
85       while leftbutton = 1 do ; make sure button is up first
86       endwhile
87       while leftbutton = 0 do ; then if button is pushed down, exit
88       endwhile
89      MainLoop := 1
90     endif
91
92    endsub
93
94    ;-------------------------------------
95
96    sub print
97     call LineFeed
98     Call PrintText
99     ReadButtons empty ; empties click queue to ignore clicks during
           a print
100   endsub
101
102   ;-------------------------------------
103
104   sub LineFeed
105
```

```
106    mode 0,draw
107    copy 0,   9,149,   249,193,   0,   9,140 ; move 5 lines up
108    color 0,white
109    rect 0,    9,185,    249,192 ; blank 6th line
110    color 0, TextColor
111
112    CountLines := CountLines + 1
113
114    if CountLines  MaxLines then
115     mode 0, overlay
116     color 0, brown
117     text 0,9,192,"   - press mouse button for more -"
118      while leftbutton = 1 do ; make sure button is up first
119      endwhile
120      temp := 0
121      while temp = 0 do
122       temp1 := leftbutton ; temp1 = 0 if NO button was pressed, 1
            otherwise
123       getchar $letter
124       length $letter, temp2 ; temp2 = 0 if NO key was pressed, 1
            otherwise
125       temp := temp1 + temp2 ; temp = 0 if NO key and NO button
            pressed
126      endwhile
127     color 0,white
128     rect 0,    9,185,    249,192 ; blank 6th line
129     color 0, TextColor
130     CountLines := 1 ; count 1 because this line is blanked and
          used
131    endif
132
133   endsub
134
135   ;-----------------------------------
136
137   sub PrintText
138    mode 0, overlay
139    text 0,9,192,"@$tx"
140   endsub
141
142   ;-----------------------------------
143
144   sub ClearButtons
145    temp := 34 ; clear the buttons for non-movable objects
146   while temp < 45 do
147    remove temp
148    temp := temp + 1
149   endwhile
150   endsub
151
152   ;-----------------------------------
153
154   sub GoNorth
155    offset := GoN
156    x1 := 59
157    y1 := 138
158    x2 := 71
```

```
159     y2 := 150
160     x  := 273
161     y  := 64
162     $tx := "n"
163     ButtonUsed := 1
164   endsub
165
166   ;-----------------------------------------
167
168   sub GoSouth
169     offset := GoS
170     x1 := 72
171     y1 := 138
172     x2 := 84
173     y2 := 150
174     x  := 273
175     y  := 97
176     $tx := "s"
177     ButtonUsed := 1
178   endsub
179
180   ;-----------------------------------------
181
182   sub GoEast
183     offset := GoE
184     x1 := 85
185     y1 := 138
186     x2 := 97
187     y2 := 150
188     x  := 288
189     y  := 81
190     $tx := "e"
191     ButtonUsed := 1
192   endsub
193
194   ;-----------------------------------------
195
196   sub GoWest
197     offset := GoW
198     x1 := 98
199     y1 := 138
200     x2 := 110
201     y2 := 150
202     x  := 259
203     y  := 81
204     $tx := "w"
205     ButtonUsed := 1
206   endsub
207
208   ;-----------------------------------------
219
220   sub GoUp
211     offset := GoU
212     x1 := 111
213     y1 := 138
214     x2 := 123
215     y2 := 150
```

```
216     x  := 307
217     y  := 64
218     $tx := "u"
219     ButtonUsed := 1
220    endsub
221
222    ;-----------------------------------------
223
224    sub GoDown
225     offset := GoD
226     x1 := 124
227     y1 := 138
228     x2 := 136
229     y2 := 150
230     x  := 307
231     y  := 97
232     $tx := "d"
233     ButtonUsed := 1
234    endsub
235
236    ;-----------------------------------------
237
238    sub Help
239     x1 := 137
240     y1 := 138
241     x2 := 197
242     y2 := 150
243     x  := 259
244     y  := 115
245     $tx := "help"
246     ButtonUsed := 1
247    endsub
248
249    ;-----------------------------------------
250
251    sub Dig2
252     x1 := 198
253     y1 := 138
254     x2 := 258
255     y2 := 150
256     x  := 259
257     y  := 133
258     $tx := "dig"
259     ButtonUsed := 1
260    endsub
261
262    ;-----------------------------------------
263
264    sub Load
265     x1 := 137
266     y1 := 164
267     x2 := 197
268     y2 := 176
269     x  := 259
270     y  := 151
271     $tx := "load"
272     ButtonUsed := 4
```

```
273   endsub
274
275   ;-------------------------------------
276
277   sub Save
278    x1 := 198
279    y1 := 164
280    x2 := 258
281    y2 := 176
282    x  := 259
283    y  := 169
284    $tx := "save"
285    ButtonUsed := 5
286   endsub
287
288   ;-------------------------------------
289
290   sub Quit
291    x1 := 259
292    y1 := 164
293    x2 := 319
294    y2 := 176
295    x  := 259
296    y  := 187
297    $tx := "quit"
298    ButtonUsed := 1
299   endsub
300
301   ;-------------------------------------
302
303   sub ReDrawScreen
304
305
306   if LastRoomNumber # RoomNumber then ; load new location scenery
307    t \f
308    $filename := "@$device loc@RoomNumber" ; add device & RoomNumber
309    load screen 1, $filename
310    call LoadingError
311    LastRoomNumber := RoomNumber ; update previous room number to
          one just left
312    SBPosition := 0 ; set scroll bar position (for movable
          objects) back to top
313    show screen 0
314    screenmode graphics
315    call ReDrawScrollBar
316   endif
317
318
319   if player CanGo 0 then
320    GoN := -13
321   else
322    GoN := 13
323   endif
324
325   if player CanGo 1 then
326    GoS := -13
327   else
```

```
328     GoS := 13
329    endif
330
331    if player CanGo 2 then
332     GoE := -13
333    else
334     GoE := 13
335    endif
336
337    if player CanGo 3 then
338     GoW := -13
339    else
340     GoW := 13
341    endif
342
343    if player CanGo 8 then
344     GoU := -13
345    else
346     GoU := 13
347    endif
348
349    if player CanGo 9 then
350     GoD := -13
351    else
352     GoD := 13
353    endif
354
355    mode 2, draw
356    copy 1, 0,0, 227,118, 2, 6,6 ; copy from scene buffer (1) to
           buffer 2
357    mode 2, overlay ; prepare to overlay ladder on scene in buffer
           2, if there
358
359     remove 15 ; remove click zone for ladder
360
361    if 01ladder in thisroom then
362      if RoomNumber = 7 then ; by tree
363       copy 2, 235,1, 288,76, 2, 118, 12
364       click 15, 130, 21, 164,79, SeeLadder
365      elsif RoomNumber = 6 then ; by shack
366       copy 2, 235,1, 288,69, 2, 28, 56
367       click 15, 36, 61, 75,124, SeeLadder
368      elsif RoomNumber = 9 then ; by upright boulder
369        copy 2, 235,1, 288,76, 2, 156, 21
370        click 15, 166,28, 206,91, SeeLadder
371      elsif RoomNumber = 11 then ; by overturned boulder
372        copy 2, 235,1, 288,76, 2, 108,24
373        click 15, 118,45, 151,84, SeeLadder
374      elsif RoomNumber = 13 then ; by hole in cave
375       copy 2, 235,1, 288,76, 2, 15,36
376       click 15, 23,51, 64,98, SeeLadder
377      endif
378    endif
379
380     mode 0, draw
381     copy 2, 6,6, 231,124, 0, 6,6 ; draw location from buffer (2)
           in window (0)
```

```
382    copy 2,  59, 138 + GoN,  71, 150 + GoN, 0, 273,64 ; draw N
           button
383    copy 2,  72, 138 + GoS,  84, 150 + GoS, 0, 273,97 ; draw S
           button
384    copy 2,  85, 138 + GoE,  97, 150 + GoE, 0, 288,81 ; draw E
           button
385    copy 2,  98, 138 + GoW, 110, 150 + GoW, 0, 259,81 ; draw W
           button
386    copy 2, 111, 138 + GoU, 123, 150 + GoU, 0, 307,64 ; draw U
           button
387    copy 2, 124, 138 + GoD, 136, 150 + GoD, 0, 307,97 ; draw D
           button
388
389    endsub
390
391    ;-----------------------------------
392
393    sub AlreadyIs
394     $tx:="It already is!"
395     call print
396    endsub
397
398    ;-----------------------------------
399
400    sub CantDoThat
401     $tx := "You can't do that."
402     call print
403    endsub
404
405    ;-----------------------------------
406
407    Sub NoHave
408     $tx:="You don't have it."
409     call print
410    endsub
411
412    ;-----------------------------------
413
414    Sub HaveIt
415     $tx:="You already have it."
416     call print
417    endsub
418
419    ;-----------------------------------
420
421    Sub OK
422     $tx := "OK."
423     call print
424    endsub
425
426    ;-----------------------------------
427
428    sub SeeBoat
429      $tx := "examine the row boat"
430      ButtonUsed := 2 ; set to 2 so MainLoop.SUB will act as
           command was typed
431    endsub
```

```
;-------------------------------------

sub SeeCanoe
  $tx := "examine the canoe"
  ButtonUsed := 2
endsub

;-------------------------------------

sub SeeLadder
  $tx := "examine the ladder"
  ButtonUsed := 2
endsub

;-------------------------------------

sub SeeTree
  $tx := "examine the tree"
  ButtonUsed := 2
endsub

;-------------------------------------

sub SeeBoughs
  $tx := "examine the fronds"
  ButtonUsed := 2
endsub

;-------------------------------------

sub SeeOcean
  $tx := "look at the ocean"
  ButtonUsed := 2
endsub

;-------------------------------------

sub SeeShip
  $tx := "look out to sea"
  ButtonUsed := 2
endsub

;-------------------------------------

sub SeeSand
  $tx := "examine the sand"
  ButtonUsed := 2
endsub

;-------------------------------------

sub SeePlant
  $tx := "examine the grass"
  ButtonUsed := 2
endsub
```

```
489   ;-------------------------------------
490
491   sub SeeDunes
492     $tx := "examine the dunes"
493     ButtonUsed := 2
494   endsub
495
496   ;-------------------------------------
497
498   sub SeeShack
499     $tx := "examine the shack"
500     ButtonUsed := 2
501   endsub
502
503   ;-------------------------------------
504
505   sub SeeWindow
506     $tx := "examine the window"
507     ButtonUsed := 2
508   endsub
509
510   ;-------------------------------------
511
512   sub SeeTable
513     $tx := "examine the table"
514     ButtonUsed := 2
515   endsub
516
517   ;-------------------------------------
518
519   sub SeeInterior
520     $tx := "look around the shack"
521     ButtonUsed := 2
522   endsub
523
524   ;-------------------------------------
525
526   sub SeeSky
527     $tx := "examine the sky"
528     ButtonUsed := 2
529   endsub
530
531   ;-------------------------------------
523
533   sub SeeHole
534     $tx := "look at the hole"
535     ButtonUsed := 2
536   endsub
537
538   ;-------------------------------------
539
540   sub SeeWall
541     $tx := "look at the wall"
542     ButtonUsed := 2
543   endsub
544
545   ;-------------------------------------
```

```
546
547   sub SeeFissure
548     $tx := "look into the fissure"
549     ButtonUsed := 2
550   endsub
551
552   ;-------------------------------------
553
554   sub SeeRocks
555     $tx := "look at the rocks"
556     ButtonUsed := 2
557   endsub
558
559   ;-------------------------------------
560
561   sub SeeIsland
562     $tx := "look over the island"
563     ButtonUsed := 2
564   endsub
565
566   ;-------------------------------------
567
568   sub SeeBoulder
569     $tx := "examine the boulder"
570     ButtonUsed := 2
571   endsub
572
573   ;-------------------------------------
574
575   sub SeeCave
576     $tx := "look in the cave"
577     ButtonUsed := 2
578   endsub
579
580   ;-------------------------------------
581
582   sub SeeRoof
583     $tx := "examine the roof"
584     ButtonUsed := 2
585   endsub
586
587   ;-------------------------------------
588
589   sub BreakBottle ; there are six ways to break the bottle
590    if player has 10coconut then
591     call SmashBottle
592    elsif player has 12handle then
593     call SmashBottle
594    elsif player has 11blade then
595     call SmashBottle
596    elsif player has 13shovel then
597     call SmashBottle
598    elsif player has 14hammer then
599     call SmashBottle
600    elsif player has 15chisel then
601     call SmashBottle
602    else
```

```
603     $tx := "You don't have anything hard enough to"
604     call print
605     $tx := "break it."
606     call print
607    endif
608   endsub
609
610   ;-----------------------------------------
611
612   sub SmashBottle
613    ObjNum := 2 ; bottle is number 2, hence the name 02bottle
614    call DestroyObject
615    placeobj 03paper, thisroom
616    ObjNum := 3 ; paper is number 3, hence the name 03paper
617    call AddObject ; place the paper in the scroll bar
618     $tx := "It smashes into a thousand splinters."
619     call print
620     $tx := "A piece of paper flutters to the"
621     call print
622     $tx := "ground."
623     call print
624   endsub
625
626   ;-----------------------------------------
627
628   sub NoSwim
629     $tx := "You splash about in the salty waters"
630     call print
631     $tx := "for a bit, then return to the beach."
632     call print
633     $tx := "You are too weakened to swim out"
634     call print
635     $tx := "farther."
636     call print
637   endsub
638
639   ;-----------------------------------------
640
641   sub SitBoat
642     $tx := "It's not too comfortable, sitting in"
643     call print
644     $tx := "the splintery old rowboat."
645     call print
646   endsub
647
648   ;-----------------------------------------
649
650   sub SitCanoe
651     $tx := "OK.  It is a comfortable fit."
652     call print
653   endsub
654
655   ;-----------------------------------------
656
657   sub Dig
658
659    if dig = 3 then
```

```
660      $tx := "You can't dig here."
661      call print
662     elsif dig = 2 then
663      $tx := "The rock's too hard."
664      call print
665     elsif dig = 1 then
666      if player has 13shovel then
667       if player in meadow then
668        if 04battery is found then ; battery was previously found
             when digging
669         call DigNothing
670        else
671         $tx := "You find a dead radio battery."
672         call print
673         placeobj 04battery, thisroom
674         set 04battery, found ; so if you dig again, you won't
             "find" it again
675         ObjNum := 4 ; battery is object 4, because it was named
             04battery
676         call AddObject ; put battery in scroll bar
677        endif
678       elsif player in by_the_boulder then
679        if 17gun is found then
680         call DigNothing
681        else
682         $tx := "You find an emergency flare gun."
683         call print
684         placeobj 17gun, thisroom
685         set 17gun, found
686         ObjNum := 17
687         call AddObject
688        endif
689       else ; if player in top_of_cliff
690        if 15chisel is found then
691         call DigNothing
692        else
693         $tx := "You discover a dull old chisel."
694         call print
695         placeobj 15chisel, thisroom
696         set 15chisel, found
697         ObjNum := 15
698         call AddObject
699        endif
700       endif
701      else
702       $tx := "You need a shovel to dig here."
703       call print
704      endif
705     else ; if dig = 0
706      if player in east_end_of_beach then
707       if 18matches is found then
708        call DigNothing
709       else
710        $tx := "You find a book of wet matches."
711        call print
712        placeobj 18matches, thisroom
713        set 18matches, found
```

```
714       ObjNum := 18
715       call AddObject
716      endif
717     elsif player in west_end_of_beach then
718      if 16driftwood is found then
719       call DigNothing
720      else
721       $tx := "You find a piece of dry driftwood."
722       call print
723       placeobj 16driftwood, thisroom
724       set 16driftwood, found
725       ObjNum := 16
726       call AddObject
727      endif
728     elsif player in sand_dunes then
729      if 12handle is found then
730       call DigNothing
731      else
732       $tx := "You find the wooden handle to a shovel."
733       call print
734       placeobj 12handle, thisroom
735       set 12handle, found
736       ObjNum := 12
737       call AddObject
738      endif
739     else
740      call DigNothing
741     endif
742    endif
743
744   endsub
745
746   ;-------------------------------------
747
748   sub DigNothing
749     $tx := "You find nothing, even after digging"
750     call print
751     $tx := "for quite a while."
752     call print
753   endsub
754
755   ;-------------------------------------
756
757   sub NoBurn
758     $tx := "Not without some good dry matches."
759     call print
760   endsub
761
762   ;-------------------------------------
763
764   sub BreakCoconut
765     $tx := "It's hard as a rock.  Maybe even"
766     call print
767     $tx := "harder!  It won't break open."
768     call print
769   endsub
770
```

```
771   ;-----------------------------------------
772
773   sub EatCandy
774    if player has 06bar then
775     energy := 4
776     ObjNum := 6
777     call DestroyObject
778     $tx := "Yummmm.  You get a sugar rush."
779     call print
780     $tx := "Talk about quick energy!"
781     call print
782    else
783     call NoHave
784    endif
785   endsub
786
787   ;-----------------------------------------
788
789   sub ExamineFloor
790    $tx := "It's old and dirty, but you don't see"
791    call print
792    $tx := "anything unusual."
793    call print
794   endsub
795
796   ;-----------------------------------------
797
```

The Cannibal.VOC File

```
 1
 2  VOCAB ;---------- Cannibal.voc ----------
 3
 4  action endgame
 5   timer := 95
 6   set canoe, InWater
 7   placeobj 13shovel, east_end_of_beach
 8   LastRoomNumber := 1 ; to force a ReDrawScreen
 9   go east_end_of_beach
10  endact
11
12  ;-------------------------------------
13
14  action status
15   $tx := "@FastMem  K FastMem   @ChipMem  K ChipMem"
16   call print
17   $tx := "$device = @$device   timer = @timer"
18   call print
19   $tx := "items = @items"
20   call print
21  endact
22
23  ;-------------------------------------
24
25  action Cycle
26   Show Screen 1 ; location scenery
27   While LeftButton = 1 do ; waits to make sure mouse button is
         down
28   endwhile
29   While LeftButton = 0 do ; waits to make sure mouse button is
         released
30   endwhile
31   Show Screen 2 ; button.pic
32   While LeftButton = 1 do ; waits to make sure mouse button is
         down
33   endwhile
34   While LeftButton = 0 do ; waits to make sure mouse button is
         released
35   endwhile
36   Show Screen 23 ; scroll bar
37   While LeftButton = 1 do ; waits to make sure mouse button is
         down
38   endwhile
39   While LeftButton = 0 do ; waits to make sure mouse button is
         released
40   endwhile
41   Show Screen 24 ; overlay buffer
42   While LeftButton = 1 do ; waits to make sure mouse button is
         down
43   endwhile
44   While LeftButton = 0 do ; waits to make sure mouse button is
         released
45   endwhile
```

```
 46    screenmode text
 47    While LeftButton = 1 do ; waits to make sure mouse button is
           down
 48    endwhile
 49    While LeftButton = 0 do ; waits to make sure mouse button is
           released
 50    endwhile
 51    screenmode graphics
 52    Show Screen 0
 53   endact
 54
 55   ;-------------------------------------
 56
 57   action help, hint, clue, give me clue, give me help, give me hint
 58    $tx := "Most common actions can be accomplished"
 59    call print
 60    $tx := "with the mouse.  To EXAMINE or LOOK AT"
 61    call print
 62    $tx := "an object, just click on it.  You GET"
 63    call print
 64    $tx := "and DROP objects by dragging them from"
 65    call print
 66    $tx := "the location window to the inventory"
 67    call print
 68    $tx := "window, and back.  But you can always"
 69    call print
 70    $tx := "use the keyboard to type commands. Some"
 71    call print
 72    $tx := "less common actions (like EAT THE CANDY"
 73    call print
 74    $tx := "BAR) can only be achieved by using the"
 75    call print
 76    $tx := "keyboard.  Enjoy yourself!"
 77    call print
 78   endact
 79
 80   ;-------------------------------------
 81
 82   action sit, sit down, sit down in canoe, sit down in boat
 83    if player in west_end_of_beach then
 84     call SitBoat
 85    elsif player in east_end_of_beach then
 86     call SitCanoe
 87    else
 88     $tx := "You sit down and relax for a minute."
 89     call print
 90    endif
 91   endact
 92
 93   ;-------------------------------------
 94
 95   action dig
 96    call Dig
 97   endact
 98
 99   ;-------------------------------------
100
```

```
101   action make shovel, build shovel, put shovel together, assemble
          shovel
102
103    if player has 12handle then
104     if player has 11blade then
105      $tx := "OK.  The completed shovel looks useful."
106      call print
107      ObjNum := 12
108      call DestroyObject
109      ObjNum := 11
110      call DestroyObject
111      ObjNum := 13
112      mode 2, draw
113      color 2, ObjNum
114      rect 2, x,y, x,y ; put new object in inventory array
115      mode 0, overlay
116      copy 2,ObjNum * 15 - 15,183,ObjNum * 15 - 1,199,0,x * 16 +
            266,y * 18 + 17
117      placeobj 13shovel, thisroom
118      grab 13shovel
119     else
120      $tx := "You can't do that yet."
121      call print
122     endif
123    else
124     $tx := "You can't do that yet."
125     call print
126    endif
127
128   endact
129
130   ;-------------------------------------
131
132   action swim, swim north, swim n
133    if player in west_end_of_beach then
134     t
135     call NoSwim
136    elsif player in east_end_of_beach then
137     t
138     call NoSwim
139    else
140     $tx := "There's no water here."
141     call print
142    endif
143   endact
144
145   ;-------------------------------------
146
147   action get up, stand up
148     $tx := "OK."
149     call print
150   endact
151
152   ;-------------------------------------
153
154   action break open bottle, break bottle open
155    call BreakBottle
```

```
156  endact
157
158  ;-----------------------------------------
159
160  action break open coconut, break coconut open
161   call BreakCoconut
162  endact
163
164  ;-----------------------------------------
165
166  action eat candy
167   call EatCandy
168  endact
169
170  ;-----------------------------------------
171
172  action jump, jump up, jump hole, jump tree, jump shack, jump
         boulder
173   if player in in_the_cave then
174    $tx := "You can't reach the hole by jumping."
175    call print
176   elsif player in by_the_boulder then
177    $tx := "You can't reach the top by jumping."
178    call print
179   elsif player in top_of_the_cliff then
180    $tx := "Be careful about jumping, when this"
181    call print
182    $tx := "near the edge of the cliff."
183    call print
184   elsif player in meadow then
185    $tx := "You can't reach the top of the shack by"
186    call print
187    $tx := "jumping."
188    call print
189   else
190    $tx := "Wheee..."
191    call print
192   endif
193  endact
194
195  ;-----------------------------------------
196
197  action look out to sea
198   $tx := "The old freighter moves slowly, which"
199   call print
200   $tx := "gives you an idea.  It's too far to swim"
201   call print
202   $tx := "but if you could row out to it..."
203   call print
204  endact
205
206  ;-----------------------------------------
207
208  action look around the shack
209   call ExamineFloor
210  endact
211
```

```
212   ;-----------------------------------
213
214   action author author, author
215    $tx := "written 8/15/91 by"
216    call print
217    $tx := "John Olsen"
218    call print
219    $tx := "Using Aegis Visionary"
220    call print
221    $tx := "From Oxxi Inc. (213)427-1227"
222    call print
223   endact
224
225   ;-----------------------------------
226
227   action quit
228    quit
229   endact
230
231   ;-----------------------------------
232
233   action i, inv, inventory, take inventory, get inventory
234    $tx := "You can carry up to six items.  They are"
235    call print
236    $tx := "shown in the Inventory window.  Click on"
237    call print
238    $tx := "any item to examine it more closely."
239    call print
240   endact
241
242   ;-----------------------------------
243
244   ENDVOCAB
245
246
```

Appendix B: The Solution to I Was a Cannibal for the FBI

The True Solution

Use the shovel to row the canoe to safety—that's the solution in a nutshell. Remember the spot you saw out to sea, on the horizon? It was visible from various locations when you clicked on the ocean. And when you were in the top of the palm tree, you could tell that the spot was an old ocean going freighter slowly puffing its way along. By paddling the canoe out to the ship, you are picked up and rescued from the cannibals, and returned to your job at the FBI and the safety(?) of civilization.

That presents the question, "Where do you get the shovel?" Actually, you find it in two parts, and put them together. You find the handle to the shovel hidden in the sand of the dunes. When you dig there, you will find the shovel handle. The blade to the shovel is in the rock room in the cave.

When you put the blade on the handle, you have a fully functional shovel. The shovel can be used to dig in the meadow, by the palm tree, or by the boulder. And in each of these locations, digging with the shovel produces some buried object. But none of them plays any part in the ultimate solution to the game. They just provide a secondary reason for the existence of the shovel, and help conceal its true purpose of acting as paddle for the canoe.

Entering the Cave

Perhaps this brings up another question, "How do I get in the cave?" You have to push the boulder out of the way. And that requires the extra strength that only the candy bar can temporarily give you. The extra energy from the candy bar will only last four moves, so you need to eat the candy bar immediately before trying to move the boulder. Once the boulder is tipped over, you can enter the cave, put the ladder against the hole in the wall, and enter the rock room.

The Optimum Path

Let's go through the solution in order, step-by-step from the very beginning to the very end. We will ignore the many other things that you can do, which play no part in the ultimate solution of the game. They were included to enrich the playing experience, and to add some "red herrings" intended to throw you off the scent of the actual solu-

tion. But the only steps necessary to complete the adventure are actually quite few. They can be listed in a short paragraph.

When the game starts, you are on the beach. Travel south to the meadow, and get the ladder. Take the ladder east to the palm tree, and place it against the tree. Climb the ladder and take the candy bar that you find up there. Climb back down, take the ladder with you, and go west and south to the boulder. Eat the candy bar and push the boulder aside. Enter the cave and put the ladder against the hole in the wall. Climb the ladder and enter the rock room. Take the shovel blade that you find there, climb back down, and head back to the beach. Stop at the sand dunes on your way back and dig in the sand. You will find a shovel handle buried there. Put the handle in the blade, to make a complete shovel. Return to the east end of the beach, by the canoe. Push the canoe into the water, sit down, and paddle your way to safety. You have won.

The winning path took less than 40 moves, depending on what you counted as a move—but remember, every action is also counted as a turn, so each time you take an object you don't need, or give any action command that doesn't move you toward the solution, it is also counted as a turn.

There are many other things that you can find in the game. And there are many things you can do with them. But they had nothing to do with the actual solution. Each had a concrete reason for being there. Each had a logic of its own. For a more complete explanation of why they were there, see chapter 15 on how the game was designed.

Appendix C: Bibliography

Books

The following books are recommended reading for adventure authors. They are listed by title, author, publisher, and date of publication. Some may be out of print, in which case you should check your public library, your local user-group library, or used-book stores.

Adventure Gamewriters Handbook for the Commodore 64, J. Walkowiak, Abacus Software, 1985.

Book of Adventure Games, The, Kim Schuette, Arrays Inc., 1984.

Book of Adventure Games II, The, Kim Schuette, Arrays Inc., 1985.

Captain 80's Book of Basic Adventures, Robert Liddil, 80 NorthWest Publishing Inc, 1981.

Compute!'s Guide to Adventure Games, Gary McGath, Compute! Books, 1984.

Computer Adventures—The Secret Art, Gil Williamson, Amazon Systems, 1990.

Golden Flutes & Great Escapes—How to Write Adventure Games for the Commodore 64, Delton T. Horn, Dilithium Press, 1984.

Keys to Solving Computer Adventures Games, M.K. Simon, Prentice Hall, 1988.

Quest for Clues, Shay Adamms, Origin Systems Inc., 1988.

Quest for Clues 2, Shay Adamms, Origin Systems Inc., 1989.

Quest for Clues 3, Shay Adamms, Origin Systems Inc., 1990.

Shortcut Through AdventureLand, A, Jack Cassidy, Datamost, 1984.

Shortcut Through AdventureLand II, A, Richard Owen Lynn, Datamost, 1984.

Writing Basic Adventure Programs for the TRS-80, Frank Dacosta, Tab Books, 1982.

Magazines

The following magazines are also strongly recommended for adventure authors:

Enchanted Realms, Digital Expressions, P.O. Box 33656, Cleveland, OH 44133.

Questbusters, P.O. Box 5845, Tucson, AZ 85703.

Oxxi

The publisher of the Aegis Visionary program is Oxxi, Inc. By writing care of them, you can communicate with me, as well as with the creator of the Visionary program, Kevin Kelm.

Oxxi provides a technical support service for the Visionary language as well as for all their other products. Check the instructions in your Visionary manual for access to this service.

You can write to me at the Oxxi address:

John Olsen
C/O Oxxi, Inc.
P O Box 90309
Long Beach, CA 90809-0309
USA

Address correspondence to Kevin Kelm at:

Kevin Kelm
C/O Oxxi, Inc.
P O Box 90309
Long Beach, CA 90809-0309
USA

Index

What makes a great adventure game?

What makes a game fun to play? Why do some games become famous and others not? What makes one game sell thousands of copies, another sell just a handful, while others can't even get published? How can you create an dventure game that will be fun to play? How can you make it challenging but not too hard?

❦

John Olsen gives us an entertaining.view of adventure gaming from the other side, sharing his experience and insight into the art of writing games. If you've never written an adventure before, this book will be an invaluable aid in completing your first game. If you are an experienced adventure author, this work will help you hone your skills to produce a superior game.

❦

Olsen covers how to get started, examines different kinds of puzzles and reveals special tricks that will make an adventure come alive in the mind of the player. You'll see how to create the game-play logic that ties it all together. You'll learn a variety of special techniques and tips culled from more than ten years of experience in writing and playing adventures.

❦

This book is designed for users of Visionary, the Aegis Interactive Gaming Language. Using Visionary is certainly not a requirement for writing adventures, but it is recommended for all but the hardiest programmers. In the second half of the book, you'll learn use the specific commands and various features of Visionary to your best advantage.

❦

Included in this book is a disk containing source code for a complete mini-adventure, "I Was a Cannibal for the F.B.I.", plus all the sound and graphics required to make this an entertaining illustration of the principles of designing an adventure game.

❦

Whether you plan to use Visionary, or write your adventure "from the ground up" in BASIC or assembly language, John Olsen shows you the skills you need to create the great adventure that has been inside you just waiting to get out!

0 10225 91150

ISBN 0-938385-28-3
PaperDisk Publishing
Signal Hill, CA