VISIONARY™

The Aegis Interactive Gaming Language

Developed by Kevin Kelm
Published by Oxxi, Inc.

1 BOOK
2 DISKS

Aegis Visionary

The Aegis Interactive Gaming Language

Developed by Kevin Kelm
Published by Oxxi, Inc.

Manual by Jim Curlee, with contributions
by Patricia Cummings and JoAnn Houston

Printed in the USA
First Edition, September, 1991

ISBN 0-938385-21-6
UPC No. 0-10225-91145

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

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

OXXI, INC. PROGRAM LICENSE AGREEMENT

Carefully read all the terms and conditions of this agreement. If you do not agree to these terms and conditions, promptly return this package and the other components of this product to the Commodore dealer where the purchase was made.

LICENSE:

You have the non-exclusive right to use the enclosed programs only on a single computer. You may physically transfer the programs from one computer to another provided that the programs are used on only one computer at a time. You may not electronically transfer the programs from one computer to another over a network. You may not distribute copies of the programs or accompanying documentation to others. You may not modify or translate the programs or their documentation.

BACKUP AND TRANSFER:

You may duplicate the programs solely for backup purposes. You must reproduce and include the copyright notice on any backup copy. You may transfer and license the product to another party only if the other party agrees to the terms and conditions of this agreement and completes and returns a registration form to Oxxi, Inc.. If you transfer the programs you must at the same time transfer the documentation and the backup copy or copies, or transfer the documentation and destroy the backup copy or copies.

TERMS:

This license is effective until terminated. You may terminate it by destroying the programs and documentation and all copies thereof. This license will also terminate if you fail to comply with any term or condition of this agreement. You agree upon such termination to destroy all copies of the programs and documentation.

PROGRAM DISCLAIMER:

Oxxi, Inc., Aegis Development and Kevin Kelm make no warranties, either expressed or implied, with respect to the programs described herein, their quality, performance, merchantability, or fitness for any particular purpose. These programs are sold "as is". The entire risk as to their quality and performance is with the buyer (and not the creators of these programs; Kevin Kelm, Oxxi, Inc. and its division, Aegis Development; their distributors and their retailers), who assumes the entire cost of all damages. In no event will Kevin Kelm, Aegis Development or Oxxi, Inc. be liable for direct, indirect, incidental, or consequential damages. Some States or Countries do not allow the exclusion of limitation of implied warranties or liabilities for incidental or consequential damages, so the above limitation or exclusion may not apply. Further, Oxxi, Inc. reserves the right to make changes from time to time in the content hereof without

obligation of Kevin Kelm, Aegis Development and Oxxi, Inc. to notify any person of such revision or changes.

DISKETTE LIMITED WARRANTY:

Oxxi, Inc. warrants to the original licensee that the diskette on which the programs are recorded shall be free from defects in material and workmanship only for a period of ninety (90) days from the date of original purchase. If a defect covered by this warranty occurs during this 90-day warranty period, and it is returned to the dealer from whom it was purchased not later than five (5) days after the end of such 90-day period, the dealer shall, at the dealer's option, either repair or replace the diskette.

This warranty is in lieu of all other express or statutory warranties, and the duration of any implied warranty, including but not limited to the implied warranties, of merchantability and fitness for a particular purpose, is hereby limited to said ninety (90) day period. Oxxi, Inc.'s liability is limited solely to the repair or replacement of the defective product, in its sole discretion, and shall not in any event include damages for loss of use or loss of anticipated costs, expenses or damages, including without limitation any data or information which may be lost or rendered inaccurate, even if Oxxi, Inc. has been advised of the possibility of such damages.

Some states do not allow a limitation on how long an implied warranty lasts, so the above limitation may not apply to you. Some states do not allow the exclusion or limitation of incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Further, Oxxi, Inc. reserves the right to make changes from time to time in the content hereof without obligation of Oxxi, Inc. to notify any person of such revision or changes.

TRADEMARKS:

Amiga, AmigaDOS and **Amiga Workbench** are trademarks of Commodore-Amiga, Inc.

Deluxe Paint and Deluxe Paint III are trademarks of Electronic Arts.

Aegis, Aegis SpectraColor, Visionary, SoundMagic, AudioMaster, TurboText and **Aegis Visionary** are trademarks of Oxxi, Inc.

Acknowledgements

I would like to thank Kevin Kelm for his ideas, creativity and perseverance to see this project through to completion. Much gratitude is also due John Olsen, the Visionary Guru—without his many hours of beta-testing Visionary, it would not be a product today. My sincere appreciation goes to Patricia Cummings and Joann Houston for their contribution to this manual.

Jim Curlee
Product Manager
September 1991

I would like to dedicate Visionary to my family for their support through the years.

Kevin Kelm
September 1991

Table of Contents

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

Lines in the source code listings in Appendix C are displayed with an * before the start of each new line. These asterisk characters do not appear in the actual source code, but are shown with the source listing for your reference only.

```
 *  This line of source runs over the end of the printed page
line, but the overlapping portion is not numbered
 *  while this begins the actual next line of the source
 *  and the line following this one is blank
 *
 *  etc.
```

Chapter 1: Introduction

What You Can Do with Visionary

Welcome to Aegis Visionary "The Aegis Interactive Gaming Language", the most advanced Adventure Writing Language available for the Commodore-Amiga computers. Visionary has been designed to provide you with necessary tools to easily produce a commercial quality adventure game.

Your adventures can take on many styles, including text-only, graphic-only, text-graphic hybrids—you can even create dungeon-style games. Whether Visionary is used for professional, recreational, or educational purposes, it offers Amiga users the highest level of performance and ease-of -use available.

Feature Overview

Visionary offers these powerful features:

- 65,000 rooms with 32 attributes per room
- 65,000 objects including non-playing characters who can interact with players 32 attributes per object
- 65,000 subroutines
- 65,000 action blocks
- 128,000 variables
- 4,294,967,295 characters of text

Plus

- Stereo sound
- 25 IFF graphic screen buffers in memory
- 25 IFF sound buffers in memory
- 50 prepositions (user declared)
- 10 articles (user declared)
- 70 powerful programming commands
- 19 mathematical operations
- Re-definable function keys
- Larger-than-page scrolling
- Instant image blitting
- Speech output

Visionary supports

- Total mouse-driven adventure games
- HAM (Aegis SpectraColor)
- Standard IFF images
- Amiga IFF-ANIM format from DOS command
- Aegis AudioMaster III sequenced sounds
- MED/MIDI music
- NTSC and PAL

Visionary Compiler and Utilities

Aegis Visionary is really eight products in one:

- Visionary Language
- Visionary Compiler
- Visionary Debugger
- Visionary Linker
- VCode Utility
- LoadScreen Utility
- CloseScreen Utility
- VCoord Utility

Visionary Language

The Visionary language provides 70 extremely powerful commands tailored for the creation of adventure games. Each command replaces the multiple lines of code it would take to create the same result in ordinary computer languages. The language is very "English" like—even to the extent of having pronouns, modifiers, and so forth, which have function similar to the "parts of speech" of normal English grammar.

Visionary Compiler

Once code is created with the Visionary Language, it is compiled into executable code, which can be processed by the Amiga computer. This is done with the aid of the Visionary compiler, VCOMP. VCOMP is a two-pass compiler to help optimize your code.

Visionary Debugger

You can debug your code after compiling with the Visionary debugging utility, DBUG. The debugger allows you to execute the game step-by-step without "dying", so you can explore all of the facets of your game to test for inconsistencies or other problems.

Visionary Linker

With the Visionary Linker, called VLINK, you can convert your game into a "run-time" version, which can be played without having to run it through a copy of Visionary. The linked version of your program can be distributed without requiring your game users to purchase a copy of Visionary.

VCode Utility

VCode allows the Visionary programmer to encrypt the IFF images and sound samples used in the game to prevent them from being modified in any way. VCode also allows files to be decoded – but only by the author.

LoadScreen Utility

LoadScreen lets the Visionary programmer display an unencrypted IFF image while the game loads, usually at the time of boot-up. LoadScreen also performs some "housekeeping" tasks like running timers and color cycling.

CloseScreen Utility

CloseScreen gives you a way to halt the display of the title screen, even if the game allowed the player to close it with some other command such as a mouse click.

VCoord Utility

The **VCoord** utility allows the graphic screen designer to easily "count pixels", to determine the Visionary coordinates of a desired zone or screen area. These coordinates are used in defining click zones, specifying areas for graphic commands, copying images from one buffer to another, and a number of other places where a precise screen location is specified.

VIE – the Visionary Interactive Editor

Source code can be entered via any Amiga editor such as Oxxi's TurboText™. A graphical editor called VIE (Visionary Interactive Editor) for Visionary, with many built-in features to assist with the creation of code, is available as a separate product from Oxxi. The VIE program will allow you to select graphic elements using the mouse, rapidly define your adventure "map" in an environment much like an ordinary paint program, and use point-and-click simple commands to develop your game's player interface. Code developed using the VIE will automatically be compatible with the Visionary system.

System Requirements

Visionary has been thoroughly tested to insure it works correctly and reliably. The software was run for several months on a wide range of hardware configurations with various peripherals. A strict quality-assurance process has been applied throughout the development stage to test all levels of software operation.

Visionary requires Amiga KickStart version 1.2 or later, and Amiga Workbench version 1.3 or later for correct operation. Visionary works equally well under AmigaDOS 1.3 and AmigaDos 2.0.

Visionary requires a minimum of 1 Mbyte of memory and one disk drive, but you also have to keep in mind the memory requirements of your game. The more extensive your program, the more memory you will need to run it.

About This Manual

This manual is intended to explain the tools used to create an adventure game using Visionary. In addition, it outlines the general procedures to use these tools to create a game and provides a reference for the Visionary Language. It is not intended to be a tutorial for programming in Visionary. Detailed tutorials with examples, including source code for a sample game, are available in *The Visionary Programmer's Handbook*, by John Olsen. John Olsen's book addresses the needs of the beginner programmer, as well as those of the expert programmer who wishes to develop commercial game software with the Visionary program.

Last-minute information on changes, additions, and enhancements made to Visionary after the this manual was printed, can be found in a **ReadMe** file located on the distribution disk. If present, this file can be viewed by simply double-clicking its icon from Workbench. Please take the time to read this file—it may save you trouble by pointing out changes in the way the software operates.

What's In the Box

Pleas check the contents of the Visionary package to insure that all documentation, the program disk, and supplementary information material have been included and are in proper working condition. The package should include:

- This Manual
- Two 3.5-inch floppy disks
- A warranty registration card, bound at the front of this manual

If any of the above items are missing, please contact Oxxi Customer Service for an immediate replacement. You will find technical support information in Appendix E, Technical Support.

Warranty Registration

We are committed to give you the best after-sales support possible. However, we can only help you if we know who you are. Before proceeding any further, remove the warranty registration card bound in this manual, fill in the necessary information, and mail it to us. When we receive your registration card, you will be entitled to:

- Technical Support
- Updates to Visionary when they are produced, at a nominal cost
- Substantial discounts on all Oxxi/Aegis software products
- Periodic newsletter to all Oxxi/Aegis software users

Please complete the warranty registration card and mail it today.

Making Backups

It is essential that a working copy of the Visionary master disk be made immediately. The data stored on computer disks is susceptible to electrical and magnetic fields, and can be damaged by physical contact with dust, fingerprints or moisture.

By creating a backup, or working copy, the master disk can be kept safe from any harm that might otherwise come to it during day to day use. Before making a backup, please insure that the master disk is write-protected. If the write-protect hole is closed, the disk is write-enabled—in other words, you can write to it. If the write-protect hole is open, the disk is write-protected. Insure that the hole is open on the master disk at all times.

Installation

While Visionary will work on a floppy system, you will find several advantages to work on a hard disk drive. To install Visionary to a hard drive, run the installation program from the CLI/Shell with the following command:

```
execute HD_Install {pathname}
```

where {pathname} is the drive and directory location where you want to install Visionary.

To create your games, you will need access to a text editor such as Oxxi's TurboText™. Follow the installation instructions for your particular editor to install your editor on your hard disk drive.

Selling Your Adventures

You can sell the adventure games you develop using Visionary, and we ask no licensing fee or royalty payments. However, the adventure must display the following credit at the start of the adventure:

```
DEVELOPED USING AEGIS VISIONARY,
PUBLISHED BY OXXI, INC 1991
   (213) 427-1227
```

Chapter 2: Off to a Running Start

Most of the discussion in this chapter is geared to the first-time game programmer. It should help you get started with Visionary, and give you a quick look at how the Visionary commands are used to make your game come to life.

The tutorial assumes you know a little about programming. Perhaps you've written a program in BASIC, or a script file to be executed with AmigaDOS **IconX**. Maybe you've even written programs in a compiled language before, but want some guidance with the commands specific to Visionary.

This tutorial does not attempt to teach you programming, or game design. Two example Visionary games are provided in the Visionary package. In addition to working through the tutorial, you should examine the way these programs accomplish the game action and handle potential game problems. John Olsen's excellent book, *The Visionary Programmer's Handbook*, provides many more-extensive examples, game design tips, and techniques for problem-solving and handling graphics. Many libraries also carry textbooks and other general references for the first-time programmer.

Source Code

Unless a program is written in machine language, it exists in two stages. The files containing the commands which will be interpreted by the Visionary program, the files you write to create your game, are called **source code**. Your source code is then **compiled**, **debugged** and **linked** to produce the **executable file**, the stand-alone game program.

To produce the source code files for your game, you will need a text editor. The Amiga comes with a usable text editor called ED—your AmigaDOS Manual contains instructions for starting and using the ED text editor. You may prefer to use a commercial text editor such as the Oxxi program TurboText™, or you can use a word-processor like WordPerfect™ that allows text to be saved in ASCII format.

In the text editor, press the **[Return]** key at the start of a line to make a blank line. Blank lines in source code are ignored by the Visionary compiler, so they take up no room in the compiled code. In the source code file, they serve to "open out" the lines of code, making them easier for you to read.

Press the **[SemiColon]** key at the start of a new line. The semi-colon character indicates that nothing following on the line will be read or noticed by

the computer. In programming, this is called a **comment**. Comments are intended for the human readers of the code—leaving them out will not affect the way the program handles the code, but may make your source code more difficult to read later.

```
; The semi-colon marks this line as a comment
```

The **[Space]** character at the start of a line is also ignored by the compiler. Commonly, related sections of code such as loops and multiple-line definitions are indented using a space at the beginning of each line. This serves to help the human reader of the source code spot the lines that go together. The Visionary compiler ignores these leading spaces.

```
DEFAULT
 e East_Room
 w West_Room     ; comments after code
 n North_Room    ; are also ignored
 u UpStairs
ENDDEFAULT
```

Like blank lines and comments, these **line indentions** can be left out. Their main purpose is to help you navigate through your own code. Often, you will come back to re-work your code several times during the course of developing a game program. Your choice to omit these programming aids now may mean extra hours spent trying to decipher your own program later.

Setting the Scene

The game we will develop during this tutorial was written by John Olsen. His game, *The Magic Potion*, uses a simplified version of the plot, source code and graphics John used in developing his Visionary game *I Was a Cannibal for the FBI*.

Rooms

The *Potion* game borrows the setting from *Cannibal*—a deserted tropical island—but its map shows only four locations or **rooms** where the player can be after a move. Play starts at a deserted beach by a palm tree. A second location has an abandoned shack whose roof the player can climb to, providing the third location. The palm tree can also be climbed, for the fourth location.

So we'll start with the file that describes the rooms and their relationships to each other. This file can have any name, but it must end with the extension **.ROOMS**.

In your text editor, start a new file which we'll call Potion.ROOMS. Each file will start with a blank line, followed by a comment line containing the

name of the file, and a second blank line on the third line of the source code **listing:**

```
;-- Potion.rooms --
```

In a .ROOMS file, the next line names the room. We'll first name each of the rooms, separating one room from the next in the source code for our human eyes with a blank line, a comment line with dashes, and another blank line.

```
room ByTree
endroom

;--------------

room ByShack
endroom

;--------------

room InTreeTop
endroom

;--------------

room ShackRoof
endroom

;--------------
```

This ROOM/ENDROOM pair of commands will "bracket" the other code we add later to describe and define each room.

Room Attributes

Once we have decided on our game locations, we next have to define the most basic **attribute** of those locations. What directions can the player move? After a command to move east, what room will the player be in? Can the player even move east from this location?

The DEFAULT command sets the possible directions allowed at the location. At the start of the game, by the palm tree, the player needs to be able to move west to the shack. At the shack, for example, we'll give the player the ability to move both west (back to the palm tree) and up, to climb to the top of the shack. Once in the room which defines the top of the shack, the player needs to be able to move down, back to the location by the shack. Also, since we decided the player could climb the tree, a command for getting down again needs to be provided in the room **InTreeTop**.

These codes are added to the .ROOMS file after each ROOM command, before the ENDROOM command. Notice that a blank line on either end of the DEFAULT section serves to set it off from the rest of the file, and the direction lines are indented to further emphasize their inclusion in the group.

```
room ByTree

default
 w ByShack
 u InTreeTop
enddefault
endroom

room ByShack

default
 e ByTree
 u ShackRoof
enddefault
endroom

room InTreeTop

default
 d ByTree
enddefault
endroom

room ShackRoof

default
 d ByShack
enddefault
endroom

;----------
```

Save the Potion.ROOMS file. You'll be coming back to it again as your game takes on form.

Objects

We have the "scenery" in place for our game—now, what about props? Most adventure games provide objects which the player must collect, move, drop, and otherwise handle to solve the puzzle posed by the game. Our simple *Potion* game provides several **movable** objects which the player can manipulate, as well as **nonmovable** objects, which contribute to the atmosphere, but which the player cannot pick up and carry from place to place.

Nonmovable Objects

We have already mentioned several objects in passing, in describing the scenery of the rooms. A palm tree by a beach implies sand, for example, and palm fronds at the top of the tree. The beach also implies ocean, sky, perhaps some clouds or the sun. The palm tree itself is an object, as are any other plants you include in this picture. At the shack, you have the shack itself.

These are all **nonmovable objects**, which must be defined in file with the extension **.OBJ**. We will create a file for these objects, and another just for movable objects.

Start a new file which will be named NonMovable.OBJ. Notice that it, like the .ROOMS file, starts with a blank line, followed by a comment line containing the name of the file. Then each code section will be set up like we did for the rooms. However, instead of a ROOM/ENDROOM command pair, the OBJECT and ENDOBJECT commands will bracket the sections of code that describe each object.

A line between the OBJECT and ENDOBJECT commands will give the **synonyms** which the player will be allowed to use as names for this object in commands. Sand, for example, might also be referred to in the context of this game as **dirt**, the **ground**, or the **floor**, as well as by its object name, **sand**. The player is less likely to use a synonym for objects like **sun** and **sky**, so no synonyms need to be provided for these objects.

```
;-- NonMovable.obj --

object sand
name ground, floor, sand, dirt
endobject

;-----------

object TreeTop
name boughs, fronds, greenery, top
endobject

;-----------

object sky
name sky
endobject

;-----------

object sun
name sun
endobject

;-----------
```

```
object island
name island, rock
endobject

;------------

object ocean
name ocean, water, sea
endobject

;------------

object plant
name plant, plants, grass
endobject

;------------

object dunes
name dunes, sanddunes
endobject

;------------
object roof
name roof
endobject

;------------

object tree
name tree, palm
endobject

;------------

object shack
name shack
endobject

;------------
```

Notice that we have put some other objects in this file—island, for example, was added because your player may try to use the word in a command. By defining it in the object file, we prevent your game from stupidly saying "I see no island here" in response to such a command.

Adjectives

Some object's names may be used with adjectives in a command—the palm tree, for example. To climb the palm tree, you may want to allow the player to say "Climb tree" or "Climb palm tree" or even "Climb tall tree". The

ADJ command allows you to define word that may be used as adjectives in a command involving the object.

The object code for the palm tree, for example, would be edited to include the two adjectives **palm** and **tall** by adding an ADJ command line:

```
;————————————

object tree
name tree, palm
adj palm, tall
endobject

;————————————
```

Object Actions

If the only actions the player can take are movements from one room to another, the game is going to be very boring. In addition to describing objects, we need to provide some way for the player to interact with them. Since these are nonmovable objects, we don't expect the player to be able to take them along from room to room—but expect the player to try.

Two actions will be common for every object you describe or display in your game. The player will try to **examine** or **take** each object you include in the game setting. If for no other reason, this will be attempted because other adventure games have used seemingly harmless objects as "killers", and made an otherwise useless-looking object the key to solving the game.

So next we should plan what the game will respond when the player tries to examine, and to take, each non-movable object. A brief description is all that's needed for each item. For **sand**, for example, we might want to respond "It's just normal beach sand" to a command to look at or examine the sand. If the player asks to examine the **sun**, you might respond "It's so bright that you hesitate to look into it. But you can feel the warmth."

For the line about the sand, you should have no problems. But how will you print the line about the sun? Since it is longer than the width of your Amiga line, you need to provide some way to show the rest of the line. For now, we'll assume that no line will be longer than 40 characters. We'll place the first 40 characters of a line to be printed out for the player in a **string variable**. We'll call the string variable **$tx**—the $ character at the start of the variable name indicates that it will be a string variable. The **value** of the variable, in this case a text string, will be defined using the Visionary "equals" sign **:=** to make the variable name, on the colon side, equal to the value, the text in quotes, on the right side.

Then we'll pass the string variable to a **subroutine** which will print it out on the screen, and also keep track of whether the screen is already full, and some other house-keeping chores. We'll discuss creating the actual sub-

routine later on, but for now, we'll use the name **print** to **call** the subroutine.

By calling the subroutine after each string variable, we can use the same variable over and over. In one line of code, $tx is set equal to "It's so bright that you hesitate to look", this text line is printed by the subroutine, then the variable $tx is set equal to a new value, "into it. But you can feel the warmth."

So the section for the sun would now appear like this:

```
;———

object sun
name sun

action look
 $tx := "It's so bright that you hesitate
    to look"
 call print
 $tx := "into it.  But you can feel the
    warmth."
 call print
endact

endobject

;———
```

What happens if the player command is not "Look at the sun" but is "Examine sun" instead? As this action is currently coded, the Visionary game would respond "I don't understand the word 'examine'". You can add action synonyms, just as you added object synonyms. Three action synonyms would change the **action look** line in the sun object description to

```
action look, examine, search
```

You might wish to write a similar response if the player attempts to **take** a nonmovable object such as the sand. Consider what happens when this action is not handled in your code. Instead of a "natural" response, the game simply tells the player "You can't do that". The response is the same whether the player tries to take the sun, or a drink of water.

You can use the same subroutine, print, to send the string variable $tx to the user so that your game appears to respond to the player's command. On a command to take the sun, for example, a "natural" response would be "Careful!, you'll burn yourself. You decide the sun is out of reach." This gives your game a much more intelligent feel than the simple "You can't do that", repeated each time the player tries to take (or **get** or **grab**) a nonmovable object.

The Initial Location

Each object has an initial location, the **initroom**, where it is "stored" until the player acts on it. Since these are nonmovable objects, the player can't carry them to another room and drop them there, so the initroom will be the location where the object will always be located during the game. For an item like the palm tree, which is only "visible" in one room, ByTree, the initroom will be ByTree. But how will we handle objects like the sun, sand, and plants which are visible in each room?

The solution is a "hidden" room, which the player can never visit, but which will hold all the objects that need to be accessible from each room. For this game, the hidden room is named **unused**, but it might also be called **hidden, invisible** or any other name that helps you remember the player will never be able to get there.

The MoveObj command is used to move the object from the hidden room into the room where the player first encounter it, and then to move it into the next room so that it appears to be in both places.

The Code Section

Each object description also contains a **code** section, bracketed by the CODE and ENDCODE commands. Objects may have no commands in the code section, and the whole code section may be left out.

So the completed object code for the sun might look like:

```
;————————

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."
  call print
endact

endobject

;————————
```

Expected Actions

In addition to the typical actions **take** and **examine**, some objects will invite special actions. By including a tree, you set the player up to try climbing it. When water is present, most players will try to drink it. Your game needs a way to handle these actions with game objects.

For the tree, you might want to have the game respond "You try, but you slide back down" when the player tries to climb. For drinking the ocean, we'll just have the game tell the player "you get a small amount in your hands, but all it does is get your hands wet." This may be enough to communicate that it is pointless to try drinking the ocean.

These actions are handled in exactly the same way as the **take** and **examine** actions. First the action and its synonyms are defined, then the game response is set as the string variable $tx, and finally the string is printed by calling the subroutine print.

So the tree object code might now look like:

```
;————————

object tree
name tree, palm
adj palm, tall

initroom ByTree

code
endcode

action look, examine, search
  $tx := "It's slick brown bark leads upward
    to"
  call print
  $tx := "the green fronds at the top."
  call print
endact

action get, take, grab
  $tx := "Sure, I suppose you intend to pull
    it up"
  call print
  $tx := "by the roots?  No way!"
```

```
  call print
endact

action climb
 $tx:="You try, but you slide back down."
 call print
endact

endobject

;———
```

» To see the entire **NonMovable.OBJ** source code, refer to Appendix C, The Tutorial Game Files. We will not look any more at this file. If you will be compiling the *Potion* game for yourself, you will want to enter the code listed in this Appendix, line for line, save it as a file called NonMovable.OBJ and use it in compiling the complete game. This tutorial will cover the process of compilation later.

Movable Objects

In addition to the nonmovable objects, which are essentially part of the scenery, Visionary lets you add **movable objects** to your game. These can be picked up by the player, moved from room to room, dropped in other locations than they were in initially, used on other objects, even combined with other objects to make a whole new object. Most of these actions are beyond the scope of this tutorial.

But we've set up two rooms that are "at the top of" of objects described in the other two rooms—the top of the shack roof, and the top of the tree—and now we need to provide some plausible way for the player to climb up to these locations. An object like a **ladder** could be provided in one room, and the game can let the player move it to the other location to use there, as well.

To start the movable object file, a new file which will be named **Movable.OBJ** will have the standard information in the first three lines:

```
— Movable.OBJ —
```

Like nonmovable objects, each movable object description will start with the command OBJECT and the name for the object, and end with the command ENDOBJECT. NAME, ADJ, ACTION/ENDACTION,

CODE/ENDCODE, and INITROOM commands will also be included. Let's start defining the ladder by setting up a **template** for the object, which will include each command or command pair we expect for an object file:

```
; ———

object

name

adj

attrib
endattrib

initroom

code
endcode

action
endaction

endobject

; ———
```

Object Attributes

Movable objects are a lot more likely to need some additional information than initial location and allowed actions—for example, since the ladder can be moved, it might be placed against the shack or against the tree, to allow the player to climb. Your game needs to keep track of what object the ladder is leaning against, or if it is not leaning against anything.

Potion keeps track of these position **attributes** with two **flag** variables, **AgainstShack** and **AgainstTree**. The initial values of these variables are set between the ATTRIB/ENDATTRIB command pair, right after the NAME and any ADJ command. For the ladder, which starts out placed against the shack, the initial values of these attributes are **Y** for **AgainstShack** and **N** for **AgainstTree**. We'll plug these variable names and values in the lines between ATTRIB and ENDATTRIB, and add the names and adjectives we'll use for the ladder at the same time.

```
; ———

object ladder

name ladder

adj wood, wooden
```

```
attrib
 AgainstShack Y
 AgainstTree N
endattrib

initroom ByShack

code
endcode

action
endaction

endobject

; ———
```

Inventory

When the player **takes** an object, it is said to be in the player's **inventory**. The objects in the inventory move with the player from room to room. Generally, adventure games only allow objects in inventory to be manipulated in other ways than the **take** and **examine** actions we've already discussed. Movable objects in the same room with the player must be moved into the inventory in order to be used or "carried" to the next room.

So the **take** action for a movable object is different than that for a nonmovable object. When the player couldn't carry the object to the next room, you didn't need anything more than a printed comment to tell the player why. For the ladder, however, you need commands to place the ladder object into the player's inventory when the **take** command is given.

The player can see a printed list of objects that have been taken by giving the command INVENTORY. This command is handled automatically by Visionary—you don't have to write code to allow this to happen.

The IF Command

You also need to know if the player already has picked up the ladder, whether it is leaning against the shack or the tree, or lying on the ground. Obviously, if the player is "holding" the ladder, it can't be climbed. Likewise, if it is lying on the ground under the tree, the player can't use it to climb.

The command that handles this test is the IF command, and its "accomplices", ELSIF, THEN, ELSE and ENDIF. Visionary allows the very English sentence "If the player has the ladder, then {take some action}" to be coded:

```
if player has ladder then
 {take action}
endif
```

The IF...THEN/ENDIF command set brackets the actions that will be taken if the first test is "passed." You can test for more than one condition with the ELSIF...THEN command, and provide for a different action if all the preceding tests fail with the ELSE...THEN command. ELSIF and ELSE commands always occur between the IF...THEN/ENDIF command set.

To see how these commands are used in a Visionary game, let's look at how the CODE section of the ladder object will be handled. This section contains the descriptions of the object that will be printed out by your Visionary game when the INVENTORY command is given, and lines that will be added to the room description if the objects are present in the room when the player asks to **examine** it. When the player has the ladder, the response to the INVENTORY command is a brief line about the object, "a wooden ladder". If the ladder is leaning against the shack, the line "A ladder is propped against the shack" will be printed after the general room description. "A ladder is propped against the tree" is part of the room description if that ladder has been moved to that room and placed against the tree. Finally, since the ladder can be lying on the ground, a line "A wooden ladder lies here" is needed to complete the room description for that situation.

First we'll test the simple case, when the player holds the ladder in inventory:

```
if player has ladder then
 $tx := "a wooden ladder"
```

Notice that we're using the "generic" string variable $tx, which we can pass to the subroutine **print**. Next, we'll test for the case where the ladder is leaning against something:

```
ELSIF ladder is AgainstShack then
 $tx := "A ladder is propped against the
    shack."
ELSIF ladder is AgainstTree then
 $tx := "A ladder is propped against the
    tree."
```

The phrase "if ladder is AgainstTree" is shorthand for the test that determines if the AgainstTree variable is set to Y. So these two ELSIF commands test the case where the player does not have the ladder, to see which of the two flags set in the ATTRIB section of this object is currently set to Y.

Finally, if all three of these tests fail, we'll print the fourth line. When the ladder is in the same room as the player, and is not in the inventory, or propped against either the shack or the tree, it must be lying on the ground. The ELSE line defines the action that will be take in this case—the string variable will be set to the phrase "A wooden ladder lies here."

```
ELSE
 $tx := "A wooden ladder lies here."
```

Since the four possible conditions have been covered, we close the IF testing with the ENDIF command, and send the string variable to the subroutine for printing. The completed CODE section looks like this:

```
code

if player has ladder then
 $tx := "a wooden ladder"
ELSIF ladder is AgainstShack then
 $tx := "A ladder is propped against the
    shack."
ELSIF ladder is AgainstTree then
 $tx := "A ladder is propped against the
    tree."
else
 $tx := "A wooden ladder lies here."
endif
call print

endcode
```

Set and Unset Flags

When the player tries to **take** a movable object, you need a way to place it in the player's inventory. If the ladder was against the shack, the value in the flag **AgainstShack** must also be changed from Y to N, since it is no longer against the shack. Generally, when a flag or two-value variable has the value Y, it is said to be **set**. When its value is N, it is **unset**. The Visionary command to change a Y value to N is UNSET, and changing it back to Y again is done with the SET command.

Also, since the player no longer has the option to climb the ladder once it has been taken, the direction **up** must be removed from the direction options which were initially set for this room in the .ROOMS file. This is accomplished with the DIRECTIONS command. Allowable directions are simply listed after the command, along with the room to which the direction command moves the player.

The Visionary command that adds the object to the player's inventory is GRAB. After the object is added to the inventory, you will also want to send the player a message that the **take** command worked. This is done by setting the string "OK" and sending the variable to the print subroutine.

In the case where the player already holds the ladder, you might use the print subroutine to send the message "You already have it." However, you can expect to have to send this identical message for any case where the player tries to **take** an object already in the inventory. Code that will be repeated over and over is better off in a subroutine. We'll call this one

HaveIt, and make a mental note to add the subroutine to the code when we are writing these subroutines.

For now, we have really begun to flesh out the ladder description. At this point, it looks like this:

```
; ------

object ladder

name ladder

adj wood, wooden

attrib
 AgainstShack Y
 AgainstTree N
endattrib

initroom ByShack

code

if player has ladder then
 $tx:=" a wooden ladder"
ELSIF ladder is AgainstShack then
 $tx:="A ladder is propped against the
    shack."
ELSIF ladder is AgainstTree then
 $tx:="A ladder is propped against the tree."
else
 $tx:="A wooden ladder lies here."
endif
call print

endcode

action get, take, grab
 if player has ladder then
  call HaveIt
 else
  directions ByTree, w
  directions ByShack, e
  unset ladder, AgainstShack
  unset ladder, AgainstTree
  grab ladder
  $tx:="OK."
  call print
 endif
endact

; ------
```

The GO Command

Another action that needs to be added to the ladder object file is **climb**. First we need to test if the ladder is leaning against something in the "room". We can do that easily with one IF and one ELSIF command set, to see if either of the attribute flags for the ladder is set. If neither is set, we can tell the player who asks to climb the ladder "You can't. I'ts not leaning against anything."

In the case where the ladder is leaning against the tree—the variable AgainstTree is set—we want the player to move to the room InTreeTop. The command which accomplishes this is **GO InTreeTop**. The whole action block for the climb action thus looks like this:

```
action climb
 if ladder is AgainstShack then
  go ShackRoof
 ELSIF ladder is AgainstTree then
  go InTreeTop
 else
  $tx:="You can't.  It's not leaning against"
  call print
  $tx := "anything."
  call print
 endif
endact
```

Object Nouns

In order to allow the player to drop the ladder, or put it on the ground, we can add the action command set:

```
action put, set, lay, lean, prop
 if player has ladder then
  drop ladder
  $tx:="OK."
  call print
 endif

endact
```

However, what happens when the player tries to **lean** the ladder **against the tree**? In this case, **tree** is the object of the preposition **against**. Visionary handles a case like this with the OBJNOUN statement. You can test any command to see what OBJNUON was used with a preposition.

In "lean ladder against {object}", we want to know if the OBJNOUN was **tree** or **shack** so we can handle the cases where the player can now climb the ladder to a new room. In other situations, such as the command "lean the ladder against the sanddunes", you need to tell the player that the command did not succeed.

For the tree, this section of the action code would follow the ENDIF statement for the **lean** action, and come before the ENDACT statement. The following code shows how this is done.

```
action put, set, lay, lean, prop
if player has ladder then
 drop ladder
 $tx:="OK."
 call print
endif

if objnoun is tree then
 if ladder is AgainstTree then
  call AlreadyIs
 else
  set ladder, AgainstTree
  directions ByTree, w u
  $tx:="It leans against the tree and leads
    into"
  call print
  $tx:="the branches."
  call print
 endif
ELSIF objnoun is shack then
 if ladder is AgainstShack then
  call AlreadyIs
 else
  set ladder, AgainstShack
  directions ByShack, e u
  $tx:="It leans against the shack."
  call print
 endif
ELSIF objnoun is sand then
else
 $tx:="You can't do that."
 call print
endif

endact
```

» To see the entire **Movable.OBJ** source code, refer to Appendix C, The Tutorial Game Files. We will not look any more at this file. If you will be compiling the *Potion* game for yourself, you will want to enter the code listed in this Appendix, line for line, save it as a file called Movable.OBJ and use it in compiling the complete game.

The Vocabulary File

We've been talking about actions that are pretty straight-forward. Take. Drop. Lean. Examine. Even when the commands included prepositional phrases, like "lean the ladder against the shack", they were still pretty simple.

But how will your game handle a command like "save" or "quit"? The Visionary program automatically provides menus for standard actions like these, but you can also allow the player to continue typing program commands from the keyboard. The file that handles these commands is the .VOC file. We'll call ours **Potion.VOC**, and start it like all the other files:

```
; — Potion.VOC —
```

After the starting lines, each action is coded in its own section. Some of these actions are included in the vocabulary file to let the player have lots of different ways to express a command. for example, in asking for help or a clue in the middle of a game, your player might try any one of the following:

- help
- hint
- clue
- give me a hint
- give me a clue
- help me
- give me help

To any one of these, you may want to print a standard response—this is what the tutorial game does. More likely, you would want to give **context-sensitive help**, a different clue depending on where the player was in the game, what was in inventory, or some other condition you might set. This is easy enough to do by establishing variables that hold a value you can set and unset, and test to determine their condition. The tutorial game uses a simple phrase to guide the player back to using the commands provided in the game code. The code to do this looks like this:

```
action help, hint, clue, give me clue, give
   me help, help me, give me hint
 $tx := "Type sentences, or use buttons
   instead."
 call print
endact
```

The GHOST Command

When you compile a game in Visionary, the compiler automatically adds standard menus to the game. You can override these so that they don't

show up, but the game will have commands like LOAD, SAVE, QUIT and NEW accessible anyway. You can make them available for use in the commands typed by the player with the GHOST command.

This command causes the text in the string that follows it to be interpreted as one of the player's commands. It is useful for other purposes, too—suppose you wanted the player to "fall down" a hidden trapdoor whenever the direction West was used in a particular room. The direction would never appear in any indicator, but when the player gave the command "west", it would be interpreted as "down" instead.

First the action is named. For **save**, which allows the player to save the current position and status in the game, you would ghost the save command as itself. Adding a default name for the saved file will prevent a file requester from appearing. This might be useful in developing a graphics game where the player could only save the game by clicking a button. Since there is no way for the filename to be typed in such a game, the filename would have to be provided.

For the **save** action, then, the action block in the .VOC file looks like this:

```
;———————

action save, save game, save position,
    store, store game
 ghost "save SaveGame"
 $tx := "OK.   Saved."
 call print
endact

;———————
```

» To see the entire **Potion.VOC** file, refer to Appendix C, The Tutorial Game Files. We will not look any more at this file. If you will be compiling the *Potion* game for yourself, you will want to enter the code listed in the Appendix, save it as a file called Potion.VOC, and use it in compiling the complete game.

Subroutines

A **subroutine** is a small program within a program, which is usually written to handle commands or command sets that will be required over and over again in the program. So far we have set up our code to call two different subroutines to perform repeated actions. Now we'll talk about how these subroutines are defined using the commands available from the Visionary program.

Visionary subroutines are all contained in files with the extension **.SUB** at the end of the filename. Each subroutine starts with the command SUB and ends with the command ENDSUB. The actions performed whenever the subroutine is called are listed between these two commands.

For example, we made a mental reminder to write the subroutine **HaveIt** at this point. This routine would "automate" the printing of the line "You already have it" whenever the player asked to **take** something that was already in the inventory. Since printing a string should be a familiar action by now, we'll simply list the subroutine:

```
;————————

Sub HaveIt
 $tx:="You already have it."
 call print
endsub

;————————
```

This code would go in a file called **Potion.SUB**, which would start with the standard line-comment-line pattern we have used with the other files.

Nested Subroutines

The print subroutine is a little more complex. You need a way to tell when the end of the print line has been reached. At the end of the line, you have to **scroll** the display up one line, to get ready for the next line to be printed. You might need to provide for more lines to be printed than there is room for on the screen. This will not be a problem in a pure-text adventure, where you are using the entire screen for text, but in a game that includes both text and graphic interaction, your "window" for text may be much smaller.

So the **print** subroutine will actually call two other subroutines. This is called **nesting subroutines**, because the other subroutines are executed before the ENDSUB command of the first has been executed. Visionary allows multiple nesting of subroutines.

We'll call the subroutine that handles the end of print lines **LineFeed**, and the one that checks for enough room in the text window **PrintText**. For now, it's enough to simply set up these calls:

```
sub print
 call LineFeed
 Call PrintText
endsub
```

The StartUp.SUB file

Subroutines can be very short, or they can be large files with their own separate names. One large subroutine needed for almost every game is the

one which handles conditions at the start of the game. This subroutine is only executed once, but having it in its own file allows you to quickly alter the startup conditions.

The TEXTPALETTE Command

The colors used in writing text to the Visionary text screen can be defined using the TEXTPALETTE command. A "pen" number between 0 and 3 can be defined as a Red, Green, Blue color value to set the four text pen colors. These would be defined once at the start of the game, so they are a natural addition to the StartUp.SUB file.

If you plan to have a graphic interface overlaying the Visionary-produced text screen, you would also turn off the text screen's scrollbar and menu bar. So the StartUp.SUB file would start like this:

```
;-- StartUp.SUB --

sub StartUp

TextPalette 0,0,0,0 ; set all pens to black
TextPalette 1,0,0,0
TextPalette 2,0,0,0
TextPalette 3,0,0,0

scrollbar off ; also prevents front/back
    gadgets from being seen
menus off ; prevents right mouse button from
    switching to text screen
```

Load Files

Visionary allows you to load digitized sound files, sequenced music files from AudioMaster™, or music created with the MED Music Editor program. These files can be stored in memory until you are ready to play them with the PLAY command. When you want to play the sounds or songs with no delay for loading, they should be loaded into memory at the start of the game.

For the *Potion* game, we might want to load sound files that will produce ocean wave noise, and perhaps some birds chirping. We'll also plan to write a subroutine, **LoadingError**, to handle the case where the game doesn't find the files on startup. For our game, then, we'd add the following commands to the StartUp.SUB file:

```
$filename := "ocean.snd"
load sound 1, $filename
call LoadingError
```

```
$filename := "birds.snd"
load sound 2, $filename
call LoadingError
```

We've set up the sound of the ocean as sound 1, and the birds as sound 2. Music and sound handling is covered in greater detail in Chapters 7 and 8.

» The rest of the file **StartUp.SUB**, will be left for you to examine on your own. It should be entered exactly as it appears in Appendix C if you will be compiling the *Potion* game.

Graphic Adventures

Originally, all adventure games were text only. You started that game, and all your interaction with it passed through the keyboard. You typed "W" and the description of the new location informed you that you had moved west. To "see" what you carried, you commanded "INVENTORY" and the program listed everything you have picked up.

All-graphic adventure games, on the other hand, have no way for the player to type commands—the game programmer provides active **click zones** on the graphic screen which perform actions when they are clicked. The graphic image that underlies a zone defined as the SAVE button, for eaxmple, might carry a picture of a button with the word SAVE on it.

Screen Buffers

When you look at a game screen for a graphic adventure, you see only one image, which changes in response to your commands. But this image is often built up from several screens of information, with portions of the hidden screens swapped to the front screen, the game screen, as the player commands are processed.

All these screens reside in memory, in an area referred to as the **screen buffer**. Visionary allows up to 25 screens to be held in memory. Each of the screens used in Visionary games is given a number, from 0 through 24. The Visionary graphics screen is always created as a full, undraggable screen with no title bar. The screens can be created in either of two ways: they can be loaded from disk, or opened by the program.

Loading a Screen

Visionary supports IFF image files, including those generated by HAM paint programs such as Aegis SpectraColor™. If the system has sufficient memory, new images can be brought in at any point by using the LOAD

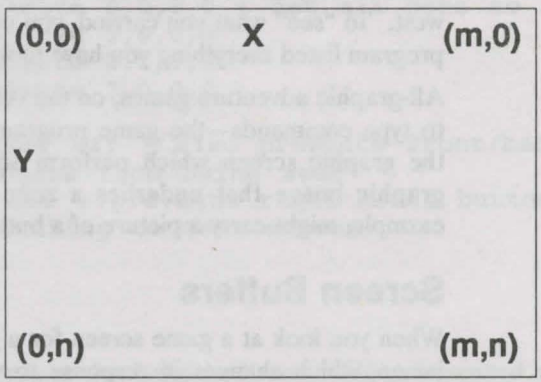
SCREEN command. The command specifies the screen buffer number, and a path and file name for the file to be loaded.

Visionary Coordinates

Let's say we have loaded to screen buffer 0, the image seen by the player, a graphic which includes a SAVE button. To allow the button to be clicked to provoke the SAVE action, the programmer needs to define **where** the click zone is, **what** action is performed when the zone is selected, **whether** the zone changes in appearance after the action is performed and **how** the zone changes.

The upper left corner of the screen is a pixel at coordinates (0,0). The first number of the coordinate pair is the X or horizontal position. The second is the Y or vertical position. By starting at the upper left corner and defining down as positive, right as positive, Visionary avoids negative coordinates in the main screen.

The Visionary coordinate screen looks like this:



The Visionary Coordinate System

In LoRes, **m** is 319 because low resolution screens provide 319 horizontal pixels. In HiRes, **m** is 639. For an NTSC system, **n** can be 199 in standard display, or 399 in Interlace. A PAL system will have **n** = 255 in standard display, 511 in Interlace.

The Visionary utility VCOORD is provided to make locating coordinates in your graphics screens much easier. VCOORD is covered in detail in Appendix B, Visionary Utilities.

Click Zone Location

You can point at any location on the screen by giving its (X,Y) coordinates. A rectangle can be defined by giving its upper left and lower right corner's

coordinates. A rectangle (0,0) by (100,100) would be 100 pixels across, 100 pixels high, and its upper left corner would be in the upper left corner of the screen. If you had a scenery picture with this rectangle defined as a click zone right where the yellow of the sun was displayed on the screen, you could use the zone as a place to allow the player to give click commands involving the sun.

A simple button to allow the player to SAVE the current game position might be defined as a zone within a rectangle defined by the Visionary screen coordinates (259,13) and (319,25). This zone would be in the upper right-hand corner of the screen for a lo-res graphic, or just to the left of the middle near the top edge of the screen for a high-res graphic.

When the button is clicked, it might change color to show it has been selected. Visionary also allows you to **swap** images within defined zones on the screen—this capability would allow you to swap an image of a pressed-in button for the original graphic showing it popped up, whenever the player clicked on the button.

The COLOR Command

To change screen colors over the entire screen, you would use the Visionary command PALETTE. To change the current pen color, however, we can use the COLOR command. Visionary refers to individual colors in the palette by pen number or color number. These phrases are used interchangeably to specify a position in the palette.

The number of colors available in your palette depends in part on the choices you make about **resolution** in your graphic screens. For example, a HIRES screen on the Amiga can only have 16 colors. A LORES screen, on the other hand, can have 64. Since color numbers start at 0, this means that a game with HIRES graphic screens and the maximum number of colors will have a maximum color number or pen number of 15.

You can also define the number of **bit planes** for your graphics. This refers to the number of bits that will be available for binary coding of the Red, Green and Blue color values. Each bit plane increases the number of available colors by a power of 2—a 2-BitPlane image, for example, has a maximum number of colors of 4, while a 3-BitPlane image has 8 colors available. Visionary allows you to directly set the current drawing color to any of the palette colors up to a maximum of 32 colors with the COLOR command.

The RECT Command

But let's say we want to produce a text window for our game, in which the player will enter all commands. We'll make this area smaller than the entire screen, to allow us some room for showing some graphic gadgets and a game scene in a graphic window.

We will want to draw a rectangle in the chosen color, without changing the rest of the screen colors. We want this rectangular area to be a different color than the rest of the **graphic interface** we provide for the player of our game. We'll also use this capability to further develop the PrintText subroutine we planned for in the **print** subroutine, so it will correctly **refresh** the text window, scrolling old text out of sight instead of up into the game graphics.

This would be done with the RECT command. This command also specifies the screen buffer in which drawing is to be done, then gives coordinates for the upper left and lower right corners of the rectangle. By changing the pen color, drawing the rectangle, and changing the pen color back again, we can produce a rectangle of the desired color.

The COPY Command

We also want to scroll the text up as new lines are added to the window—essentially copying the image in the window in a higher position. This is accomplished with the Visionary COPY command. With COPY, you can move images into memory and swap them into other screen buffers quickly and easily.

The COPY command specifies a rectangle on the **source buffer**, using Visionary coordinates, then points to the upper left corner of the rectangle's location of the **destination buffer**. The images in the two buffers need to have the same number of bit planes.

You can use the COPY command in one of three **modes**, DRAW, OVERLAY, or XOR. For this simple subroutine, we'll use the draw mode, which replaces anything under the copied rectangle with the contents of the rectangle. This will have the effect of "wiping out" the text that already exists in the window under the lines we copy.

Applying Visionary Graphics Commands

Let's look at how these concepts—screen buffer, rectangle draw, copy and color change—are used to print text in a small text window on top of the graphic screen.

We'll set our text window to allow no more than 6 lines of text to be displayed. As each line is printed to the window, we'll need to do several housekeeping chores. The cursor has to be moved down until the maximum number of lines are in the window. When the seventh line is to be printed in the window, the previous lines have to be scrolled up to make room for it. The top line has to be removed, otherwise it will be printed outside the area we have decided is to be the text window.

The subroutine **LineFeed** will handle the scrolling of lines of text. It copies the contents of each line into a position one line up, creating a blank line for the new text to be printed into.

```
sub LineFeed
 mode 0,draw
 copy 0, 7,149, 247,193,  0,  7,140 ; move 5
   lines up
 color 0,0
 rect 0, 7,185, 247,192 ; blank 6th line
 color 0, TextColor
endsub
```

The third line in this subroutine copies the contents of a rectangle in screen buffer 0, the screen seen by the player, to a rectangle one line space higher on the same screen. Then, since the copy statement left a "remnant" of the previous lines in line 6, we use the COLOR command to change the drawing color to color number 0, and draw a rectangle of that color over the remnant of the copied text. What the player sees happens so fast, it looks like a smooth upward scroll of the text each time the [Return] key is pressed.

The Graphics commands are covered in much greater detail in the Command Reference chapter, and described in an orderly way in Chapter 6. Some advanced techniques for using the Visionary graphics commands are detailed in Chapter 10.

The Game Definition File

Programming uses a lot of variables. The more like a "natural" human language the programming language comes, the easier it is to use. Visionary puts these two facts together to allow you to define an amazing number of variables which can be used to make your work easier.

These variables can be defined "on the fly" as you write code, of course. If you need a variable in one room of your game to test how many times an object has been in the player's inventory, you might define a variable TAKEN and use the program commands to increment it each time the object is grabbed by the player. That kind of variable can be in the object file, and it will always be "seen" by the program, because it applies to that object.

What about a variable which might be used in any of the rooms, for any of the objects, or at any point in the game? For example, let's say you want to define 16 variables to be used in specifying colors. Now, in the COLOR command, instead of using the value 0, you could just say **white** instead. If you later change your palette so that color 0 isn't white any more, you would only need to change the color numbr in one place, where you defined the variable **white** as being equal to 0.

The .ADV File

The place where these general variables are defined is the game adventure file, which has the extension **.ADV**. In the lines between the command pair VAR/ENDVAR, you can define system-wide variables for use in your other program files. To set up our color numbers, for example, we would start the .ADV file like this:

```
;— Potion.ADV —

ADVENTURE

PASSWORD jro

VAR
 MaxLines          5       ; maximum lines in
   text window before pause
 CountLines                _ ; counter for lines
   displayed in text window
 Defeat            0       ; set to 1 to defeat
   text window pause
 white            27 _  _ ; palette color for
   white
 blue              8       ; palette color for
   blue
 green             4       ; palette color for
   green
 brown            23    _  ; palette color for
   brown
ENDVAR
```

Notice that the .ADV file starts like all the others, with some information to be read by humans, then states ADVENTURE and sets the game PASSWORD. The password is very important, and it will be discussed later. For now, enter it as shown.

We have also defined two other variables. **CountLines** and **MaxLines** will be used to list messages in the text window that are more than six lines long. If we used the **LineFeed** subroutine to show them, the message would scroll out of the window faster than the player could read it. Setting a maximum number of lines as a variable in the .ADV file lets us simply use the variable name when we write this subroutine. If we decide later to allow seven lines instead of six, we only have to change the value in one place—here in the .ADV file.

The **CountLines** variable has no value set for it. When no value is "plugged into" a variable, it is set at 0 by default. So when the .ADV file is compiled by Visionary, **CountLines** is set equal to 0 at the start of the program.

File Pointers

Also in the .ADV file are the pointers to the other files needed to create a Visionary game. Between the ROOM/ENDROOM command pair, each .ROOMS file will be listed. The .OBJ files used for the game are listed between the OBJECT/ENDOBJECT command pair. Files containing sub-routines are listed between the commands SUB and ENDSUB, and finally, the .VOC filename is listed between a VOCAB/ENDVOCAB set.

For our *Potion* game, this section of the .ADV file looks like this:

```
ROOM
Potion.rooms
ENDROOM

OBJECT
NonMovable.obj
Movable.obj
ENDOBJECT

SUB
Potion.SUB
MainLoop.SUB
StartUp.SUB
ENDSUB

VOCAB
Potion.VOC
ENDVOCAB
```

The Initial Room

The final command of the adventure file specifies the room in which the player is located when game play starts. This INITROOM command is very similar to the ones we've already used for defining the room in which a movable objects is first located.

The final command in the adventure file, ENDADVENTURE, simply closes the command "bracket" opened by the ADVENTURE command.

We've completed our overview of the source code files for the tutorial game. Chapters 3 and 4 cover the creation of these files in much more detail, and discuss how the compiler uses the source code files to produce the compiled game files.

» We won't be coming back to this filein the tutorial. If you will be compiling the *Potion* game, be sure to enter the Potion.ADV file exactly as it is shown in Appendix C.

Compiling a Game

Once you have created the program files, you are ready to compile your program. The process of compiling translates the Visionary command language to machine-readable code. In the process, the source code files will be **encoded** using the password from the adventure file.

The Visionary Compiler, VCOMP, is the utility which turns your Visionary source code into its binary, encoded format. The VCOMP command is given in the CLI/Shell—it essentially calls the VCOMP program, then passes it the name of your adventure file containing the pointers to all the other files for your game.

» The *Potion* game files as listed in this tutorial are incomplete. Before compiling, you can enter the versions of these files from Appendix C, and save them with the filenames shown in the second line of each source code listing. You can also load them from the Catacoombs disk. There are eight files, named Potion.ADV, Potion.ROOMS, NonMovable.OBJ, Movable.OBJ, Potion.SUB, MainLoop.SUB, StartUp.SUB and Potion.VOC. These names are not case-sensitive, that is, they can be all capitals or all lower-case letters—Visionary will ignore the letter case.

For our tutorial adventure, the CLI command to start the compiling process would be:

```
VCOMP Potion.ADV
```

If the adventure file is complete, and the VCOMP command is entered properly, the Visionary compilation process will start. The compiler is extremely fast, so it takes only a short time to compile even complex programs.

The file which is produced when you compile will be named .GAM and .WRD. This can then be used by the Debugger and finally the Linker to create a stand-alone version of the game.

Cross-Referencing Your Code

Even when the adventure file is complete, you may have some variables you left undefined, subroutines you called but forgot to write, or other programming errors that will cause your compiled program to crash or misbehave. A useful VCOMP command **switch** turns on cross-referencing, which will generate a file with the same name as the game but with an ".XRF" suffix.

```
VCOMP Potion.ADV -x
```

The -x option turns on the cross-reference generator. Use of "-X" by itself in the VCOMP command will generate all sections of the cross reference file, but you can cross-reference any section on its own by giving its section letter after the -x switch.

If you specify any of the following letters, all cross-reference sections **except** the ones you specify after -x in the command line will be omitted; that is, **only** the sections you specify will appear. The single exception is the "F" option. Option "F" forces a form feed to be inserted between each of the selected cross reference sections. Unlike the other letters, using "-XF" in the command does not turn off any of the other sections.

Letter	Cross-Reference Section
A	Articles (of speech)
C	Code (program)
O	Objects
P	Prepositions
R	Rooms
V	Variables
F	Form Feeds

So if, for example, we give the command

```
VCOMP Potion.ADV -xr
```

the Potion.XRF file that will be generated will contain only the names of the five rooms defined in the single .ROOMS file, Potion.ROOMS.

The Visionary utilities DBUG and VLINK are discussed in more detail in Chapter 4 of this manual.

The Visionary Debugger

Debugging is the process of having the computer help you search for errors in your source code. The Visionary debugger utility, DBUG, allows you to develop, play, and root out the bugs and errors in new games.

While you are running the DBUG program, you will actually be playing your compiled game. Instead of crashing, halting, or otherwise misbehaving, however, when the DBUG play runs across a problem or **bug** in your code, it allows you to look at the source of the trouble, and even provides some additional information to help you correct the error.

To run DBUG, you would use the CLI to set the stack to 20,000 and call the DBUG program and pass it the name of your compiled game file. For our tutorial game, these commands would be:

```
stack 20000
DBUG Potion.GAM
```

The ".GAM" suffix is optional, but we'll include it here.

The DBUG utility will automatically generate a list of DBUG errors, which will be saved to a file {gamename}.ERR. An explanation of each error that can result from DBUG is shown in Appendix A.

DBUG Commands

While in the DBUG program, you can send commands to the program for greater control of the debugging process. DBUG commands interrupt the game play, and perform the specified command. They are entered in the text interface of the game, just as game text commands would be.

The DBUG command **JUMP ByShack** in the *Potion* game, for example, would cause the play to move to the room ByShack, without requiring you to give the correct direction command to move to that room as you play the game.

JUMP {room name}

This command sends the debugger/player to the specified room.

SET {room|object} {attrib#} [Y/N]

The SET command in DBUG prints the status of the given attribute number for the given room or object. Optionally, you can set the attribute to Y or N by supplying the value at the end of the command.

This could be used to check your room descriptions for a room which has been visited, for example, since we know that the VISITED attribute is always attribute 1, by giving the command

```
SET ByTree 1 Y
```

and then asking the game to **look**. The game finds the room has already been visited, since the VISITED variable is set, so the brief description is printed.

EQU {variable} [value]

EQU prints the current value of the given variable. If the optional [value] is supplied, the variable will be set to that new value.

String variables can also be used in the EQU debugging command. The format is the same as above, but because this is a string variable, a dollar

sign must precede the variable's name. If a string in single or double quotes follows the variable's name, the variable will be set to that value.

So to test a string variable that contains the player's name, for example, you might enter the DBUG command

```
EQU $playerName "V. I. Sionary"
```

From that point, each time the game printed the playerName variable, the string "V. I. Sionary" would be used. It is not possible to use Visionary's formatted variables and codes in strings entered via the debugger.

PLACE {object} [{room name}]

The PLACE command prints the name of the room where the given object can be found. The optional [{room name}] command moves the object to that room. In the tutorial game, entering the DBUG command

```
PLACE corkscrew InTreeTop
```

will cause the corkscrew to be moved to the room InTreeTop.

ROOM [{room name}]

When the ROOM command is given by itself, the complete status of the current room is given. With the optional room name, the complete status of the specified room will be printed.

OBJECT {object name}

The OBJECT command prints the complete status of the specified object.

Break Execution

A **Break Execution** menu item has been added to DBUG. Selecting this menu option will stop execution of all programs for the remainder of the turn. The prompt will be given back, and normal processing then resumes as DBUG waits for an input.

Break Execution is useful for breaking out of a runaway loop. Selecting this **may** leave resources allocated that would otherwise be closed by the programmer's software, but as always, they will be freed at the end of the session.

The Visionary Linker

After the compiler creates the .GAM and .WRD files which the debugger can use, they must be linked to create the final, executable, distributable program. This is done with the Visionary Linker, VLINK. The VLINK command allows you to create a final game program with its word-file

linked to it or separate, and you can generate an icon for the game and create it to start in either text or graphics mode.

You may chose not to incorporate the .WRD file, stores all of the text in the game, with the executable file. An advantage of linking it with the executable is that text output during game play will be radically faster. An advantage of **not** linking it comes when the .WRD file is so large that the final program would be inconveniently large. Extra-large .WRD files sometimes are a product of a large text adventure, where text output speed may not be a major contributor to the quality of the game.

When a game starts up in Graphics Mode, it is very much like performing a SCREENMODE GRAPHICS command as the first statement in your program, except that as the game loads and starts up, the text screen will open up in the BACKGROUND, so that the first screen the player sees is whatever graphics screen you care to show. This can be handy for a nice clean startup effect in an all-graphic game.

To create a final game file for the *Potion* game, have it start up in graphics mode, and link the word file with the game file, the command would be:

```
VLINK Potion.GAM -ig
```

The ".GAM" extension is optional, and so are the options. The allowable VLINK options do the following.

i	does not generate an icon for this game
w	does not link the word file with the game (keep it separate)
g	causes game to start up in Graphics Mode

When you will link the two files, the .WRD file must be in the same directory as the .GAM file. The executable game file will be saved in the same directory.

The output of VLINK will be a large file—the base size of your game program will be approximately 200K, plus the size of your .GAM file, plus the size of the .WRD file (if desired). For example, a 30K .GAM file and a 10K .WRD file will link to form a final game executable file of 200+30+10, or approximately 240K. Be sure before linking that you have enough disk space for the output file in addition to the .GAM and .WRD files.

Your Own Game

Now that you have been through the four steps of creating a Visionary game—writing source code, compiling, debugging and linking—you're ready to step out on your own. The next chapters will discuss how to use individual commands for text, music, graphics, and program controls, and give short examples for each command. An advanced programming section includes another tutorial which covers some very sophisticated programming techniques which are possible with the Visionary commands, and a

command reference section lists commands in alphabetical order with command syntax and usage.

Credits

The Visionary language provides you with a powerful set of tools for developing your own game programs. Once you have a stand-alone, executable file, it can be distributed free as a public-domain game, or published either as shareware or as a commercial program. The choice is up to you.

Remember that even though we do not ask for a licensing fee or royalty payments, if you publish an adventure developed or compiled using Visionary, it must display a credit at the start of the adventure that reads:

```
DEVELOPED USING AEGIS VISIONARY
PUBLISHED BY OXXI, INC 1991
    (213) 427-1227
```

Resources

While this tutorial has provided a quick look at the possibilities of Visionary, there are many other sources of information about adventure games and programming. A full list of resource books is provided at the end of Appendix E, Technical Support. This reference list was taken with permission from *The Visionary Programmer's Handbook* by John Olsen.

The Spark

Visionary provides you with some excellent tools, but in the end, the creation of your game is up to you. Use your imagination. Don't be afraid to try something you've never seen before—it might be the gimmick that sells your game.

Good adventure games all start with a spark, an idea in someone's mind. With Visionary, you've got a way to fan that spark into a roaring fire. Go ahead. Fan the flame. Build the hottest adventure game you ever dreamed about!

Chapter 3: Conventions and File Formats

Adhering to Standards

While Aegis Visionary is one of the easiest-to-learn computer languages available on any computer, there are some rules you should keep in mind while creating your adventures. This chapter explains some of these trivial details required to enter code correctly.

As you are creating your adventures, you will be creating at least five separate files which will be linked together to create the actual run-time game.

Order of Operator Precedence

Aegis Visionary operators are largely algebraic. As the program statements are executed, there is a specific order of precedence which dictates which operator/function is processed before other operators/functions. The Aegis Visionary statements are processed from left to right following the Order of Precedence shown below:

Operator precedence in descending order:

1 *, /
2 MOD
3 +, -
4 <, <=, =, >=, >, #, IN, HAS, IS, NOT
5 AND, OR
6 :=

So a statement which included both the MOD operator and the AND operator would process the MOD operator first. In a statement with the MOD operator and multiplication, however, the * operator would be processed first, then the MOD operator.

Description Notation

In the next section, we will describe Visionary formally. To do so we will use a formal description standard similar to Backus-Naur Form (BNF) Statement notation. Statement names that are to be used directly will be printed in uppercase, while the parameters and arguments to the commands will be printed in between curly brackets. For example,

```
PLACEOBJ {object_name} {room_name}
```

means that PLACEOBJ is the command, the phrase {object_name} should be replaced with the name of a specific object in your adventure, and that {room_name} should be replaced with the name of a specific room in your adventure. The brackets indicate that whatever lies between them should not be taken literally, but can assume the name of any valid identifier of the same class. This is called a **variable**.

Throughout the manual we use the curly brackets by way of illustration—you should never literally type them into your program statements. Another illustration symbol we use throughout the manual is ("|"). This is used as an OR indicator, to show that a choice is available. **Either** one or the other of the items listed may be used, but not both. Like the brackets, the OR symbol is not to be typed into your programs; it is printed here by way of illustration to help you understand Visionary's format.

Capitalization

Visionary is case-insensitive, but it is wise to adopt some scheme of capitalization and stick with it throughout your project. One that works very well is to write all the system words in all capitals, then capitalize only appropriate letters in your own identifiers. This method allows you to scan your program and quickly tell the difference between your words and Visionary's words.

Indentation

It is a good habit to indent one or two columns each time you enter another level of IF statements and when you first enter a new block. This also makes the program more readable. Indenting too much can make your program difficult to read when lines of text run off the edge of the screen.

Comments

Another good practice is to **comment** the more complex sections of your program, or at any point where the function of your code is not blatantly obvious. You won't need a comment for every line, but a comment here and there will clear things up nicely. Comments start with a semi-colon ";", and all characters in the line after the semi-colon are ignored by Visionary.

Visionary Nomenclature

We will be using a set of words to describe specific items associated with the Visionary adventure game construction system. The following is the list

of terms we will frequently use and their definition within the context of Visionary.

{text}	any line of text
{word}	any word containing no spaces
{variable_name}	the name of an integer variable, from 1 to 19 characters)
{file_name}	a standard AmigaDos filename, can include the path name as well
{room_name}	the name of a room (1 to 39 characters)
{object_name}	the name of object or NPC 1 to 19 characters
{subroutine_name}	the name of a subroutine 1 to 19 characters
{attribute_name}	the name of attribute, 1 to 19 characters
{adjective}	any word describing an object,1 to 19 characters)
{noun}	any noun, 1 to 19 characters
{verb}	any single-word verb
{direction}	One of the allowed directions: N|S|E|W|NE|NW|SE|SW|U|D
{direction#}	The number, 0 through 9, refers to one of the allowed directions
{expression}	what is acted upon by a command
{string variable}	text string used as a variable
{literal string}	a text string inside single or double quotes

Visionary Sentence Structure:

One of the most important parts of an adventure game is the command parser. A parser examines the program statements and **parses** it, analyzing the individual components of the statement and placing them into such categories as verb, noun, adjective, and so on. Once the elements of the line have been categorized, the program can make the correct response to the component.

Visionary sentences are generally structured to start out with a verb. The program then looks for a noun, any adjectives associated with the noun, and then for prepositions and objects of the prepositions.

The Visionary author must first define the vocabulary used by the program. If a program sentence uses a word that is not defined, an error is returned when the program is compiled. Your program should watch for this error, and when it occurs, tell your player to "Try again."

The following sentence structures are standard, and should serve for almost all situations. Other sentence structures can be built in through your program, using vocabulary blocks you create in the Vocabulary File.

Definitions of Components

The following types of sentence components—you may remember them as "parts of speech" from grammar class in school—are recognized by Visionary. They are defined as:

ATTRIBUTE	An inherent characteristic
PREPOSITION	A word that combines with a Noun or Pronoun to form a phrase.
PRONOUN	A word used as a substitute for a Noun.
NOUN	Person, place, thing.
VERB	Action
ADJECTIVE	A word that typically serves as a modifier of a Noun.
MODIFER	To limit the meaning of, especially in a grammatical construction.
ARTICLES	Short modifiers like "a", "an", "the", and "that" placed in front of a noun or an adjective modifying the noun

Standard Sentence Structures

In the following generic sentences, articles are omitted, even when the example statements include them. This is standard notation. The examples are shown in all caps, but this is not necessary in your code.

{verb} {noun}

```
GRAB DAGGER
```

{verb} {adjective} {noun}

```
GRAB the MAGIC POTION
```

{verb} {noun} {preposition} {noun}

```
KILL the MONSTER WITH the DAGGER
```

{verb} {adjective} {noun} {preposition} {noun}

```
STEAL the SECRET SCROLL FROM the MONSTER
```

{verb} {adjective} {noun} {preposition} {adjective} {noun}

```
DECODE SECRET SCROLL WITH the DECODER RING
```

{verb} {preposition} {noun}

```
LOOK UNDER the ROCK
```

{verb} {preposition} {adjective} {noun}

```
SEARCH THROUGH the METAL CHEST
```

Typical Special Case Sentences

The following sentence structures must be specifically implemented by the adventure writer in the Vocabulary file. They are not handled automatically by the Visionary parser.

{verb} {preposition}

```
LAY DOWN
STAND UP
```

{verb}

```
YELL
SLEEP
```

A well-written adventure will include a witty, or at least appropriate, responses to some of the more common player requests. For example, if the player typed "STAND UP" when the game character was already standing, the program could reply, "You make a fool of yourself trying to stand up before you discover that you already are."

Explanation of the Visionary Files

Visionary uses a system of files to keep all source code organized, and to keep individual files from becoming too large. There are five classes of **source** files.

The first classification or main file, known as the **adventure** or .ADV file, is the most important and the easiest to write. This file basically provides the system with key information such as the names of the other files, the password, variable names, and which room the player will begin the adventure in. There can only be one adventure file per game. The other four classes may have any number of files per adventure. The other four classes are: room, object, subroutine, and vocabulary.

The **room** files contain all the code describing the rooms in the adventure. Each file may contain multiple rooms. There may also be any number of files, and you may name them whatever you like (although it is often helpful to use a "Room" or ".ROOM" extension to keep your files straight). The first two attributes of a Room File are defined automatically, but objects have no predefined attributes. The "zero" attribute for all rooms is the DARK attribute which is pre-defined for your convenience. The VISITED attribute is also set and reset automatically.

The **object** files are similar to the room files, only they define each object used in the adventure. Like the room files, it is a good idea to use an "Objects" or ".OBJ" extension for these files.

The **subroutine** files are special sections of the code that preform specific functions, and can be called from anywhere else in the adventure. This

makes it easy to perform a particular function repeatedly. It is recommended to use the ".SUB" extension for these files.

The **vocabulary** files define all nonstandard phrases, words, and special functions. It is recommended to use the ".VOC" extension for these files.

Formal Description of Visionary File Layouts

While reviewing the formal description for the file layouts, we have used the following symbols to assist with the illustration of the file layout.

- All optional blocks are surrounded by a pair of square brackets "[ ]".
- Editorial comments appear as standard Visionary comments, beginning with a semicolon ";".
- Options appear as two or more identifiers surrounded by parenthesis "( )" and separated by the OR symbol, "|".
- Blank lines and line indention are optional

The Adventure File

```
ADVENTURE {adventure_name}
PASSWORD {game_password}
[
ARTICLE
 {list of valid articles, separated by
    spaces or lines}
ENDARTICLE

]
[
PREP
 {list of valid prepositions, separated by
    spaces or lines}
ENDPREP
]
[
VAR
 {int_variable_name} [value]
${string_variable_name} ["value"]
 .
 .
.
ENDVAR
]
ROOM
 {room_filename}
 .
 .
.
ENDROOM
[
```

```
OBJECT
 {object_filename}
 .
 .
 .
ENDOBJECT
]
[
SUB
 {subroutine_filename}
 .
 .
 .
ENDSUB
]
[
VOCAB
 {vocabulary_filename}
 .
 .
 .
ENDVOCAB
INITROOM {room_name}
ENDADVENTURE
```

The Room File

All **room** file names must appear in the adventure file.

```
ROOM {room_name}
[
ATTRIB
 {attribute name} [(Y | N)]
 .
 .
 .
ENDATTRIB
]
[
DEFAULT
 {direction abbreviation} {room_name}
 .
 .
 .
ENDDEFAULT
]
[
CODE
 {program statements}
ENDCODE
]
[
ACTION {list of actions, separated by commas}
```

```
 {program statements}
ENDACT
]
; You may follow ENDACT with another ACTION
    declaration
ENDROOM
; you may follow ENDROOM with another ROOM
    declaration
```

The Object File

The **object** file name must appear in the adventure file.

```
(OBJECT | NPC) {object_name}
[
NAME {list of synonyms, separated by commas}
]
[
ADJ {list of adjectives, separated by commas}
]
INITROOM {room_name}
[
CODE
 {program statements}
ENDCODE
]
[
ACTION {list of verbs, separated by commas}
 {program statements}
ENDACT
]
; you may follow ENDACT with another ACTION
    declaration
(ENDOBJECT | ENDNPC)
; you may follow ENDOBJECT or ENDNPC with
    another OBJECT or NPC declaration.
```

The Subroutine File

The name of the **subroutine** file must appear in the adventure file.

```
SUBROUTINE {subroutine_name}
 {program statements}
ENDSUB
; you may follow ENDSUB with another
    SUBROUTINE declaration
```

The Vocabulary File

The name of the vocabulary file must appear in the adventure file.

```
VOCAB
ACTION {list of actions, separated by commas}
 {program statements}
```

```
ENDACT
; you may follow ENDACT with another ACTION
    declaration
ENDVOCAB
```

Chapter 4: How to Use the Visionary Compiler

The Visionary Compiler

Once you have entered your program statements with your editor, you are ready to compile your program to create machine readable code. VCOMP is the Visionary Compiler, which turns your Visionary source code (program) into a binary, encoded format.

Command line format for VCOMP is:

```
VCOMP {game name}[.ADV] [-x[.......]]
```

where {game name} is the filename of the main adventure file, possibly followed by its ".ADV" suffix. The "-x" option turns on the Cross-Reference generator, creating a file with the same name as the game but with a ".XRF" suffix. Entering the above command properly, will turn on the Visionary compiler. The compiler is extremely fast, and it will take only a short period of time to compile even complex programs.

The file which is produced when you compile will be named .GAM and .WRD. This can then be used by the Debugger and finally the Linker to create a stand-alone version of the game.

Use of "-X" by itself in the VCOMP command will generate all of the sections of the cross reference file. If you specify any set of the following letters, all cross-reference sections **except** the ones you specify will be squelched; that is, **only** the sections you specify will appear.

The Visionary compiler will automatically generate a list of compiler errors, which will be saved to a file {gamename}.ERR.

Letter	Cross-Reference section
A	Articles (of speech)
C	Code (program)
O	Objects
P	Prepositions
R	Rooms
V	Variables
F	Form Feeds

Option "F" forces a form feed (clear screen) to be inserted between each of the selected cross reference sections. Unlike the other letters, using "–XF" does not turn off any sections.

EXAMPLES:

```
VCOMP MyGame
VCOMP MyGame.GAM
VCOMP MyGame -x
VCOMP Mygame.GAM -xapcrovf
```

The Visionary Debugger

The Visionary Debugger, DBUG, allows you to develop, play, and root out the bugs and errors in new games. DBUG requires more stack space than most programs, so you will need to reset the stack to 20,000 before starting.

Command line format for DBUG is:

```
stack 20000
DBUG {game name}[.GAM]
```

where {game name} is the filename of the game file generated by the compiler, VCOMP, and is optionally followed by its ".GAM" suffix.

EXAMPLES:

```
DBUG MyGame
DBUG MyGame.GAM
```

Execution of DBUG automatically generates a list of errors in the file {gamename}.ERR. The line numbers in this {gamename}.ERR file refer approximately to the line numbers given in the .XRF file generated by the compiler, helping you to pinpoint where the error occurred. An explanation of errors that can be reported from DBUG are shown in Appendix A.

Command Shortcuts During DBUG "Play"

The command shorthand provided for the player will also be very useful as you "play" the game during the DBUG process. Rather than typing GO NORTH or MOVE NORTH, you can simply type NORTH, or even shorter, N. These direction abbreviations are:

Command	Allowed Abbreviations
Move North	North, N
Move South	South, S
Move East	East, E
Move West	West, W
Move Northwest	Northwest, NW
Move Northeast	Northeast, NE
Move Southwest	Southwest, SW
Move Southeast	Southeast, SE
Move Up	Up, U
Move Down	Down, D

Other commands provide additional information. **SCORE**, for example, will show the value stored in the system-assigned variable **Score**. **INVEN-**

TORY or its abbreviation **I** will list all the objects for which the (Player HAS {object}) expression is true, in other words, all the objects in the inventory.

QUIT will quit the game. You can add the Yes response to the original QUIT command line to pass it to the command, forcing the game to quit without confirming, with the command

```
QUIT, Y
```

LOOK or **L** will cause the current room's CODE block to be executed. **PRINTER (ON|OFF)** will toggle printer output. **SPEECH (ON|OFF)** toggles the Amiga voice.

You can also save and load games. The command

```
SAVE [filename]
```

saves the current game to a file. The filename is optional, and can include the full path. Either of the two commands

```
LOAD [filename]
RESTORE [filename]
```

will load the specified game file. In both SAVE and LOAD, the filename is optional—when no filename is given, a file requester appears so you can select a file.

Visionary automatically links some functions to keys on the keyboard, allowing a single keypress to execute a function or command. The default "key bindings" are:

Key	Function or Command
[F1]	North
[F2]	South
[F3]	East
[F4]	West
[F5]	Up
[F6]	Down
[F7]	SAVE
[F8]	LOAD
[F9]	LOOK
[F10]	INVENTORY

These commands can be given during DBUG "play" by a single keypress of the appropriate function key. If you have used the DEFINE command to change the defaults in your game, these keys may not have the same functions stored in them. The command **DEFINE**, without any parameters after it, will display the current functions defined in the function keys.

To re-define the functions stored in the function keys, you can enter the DEFINE statement with parameters during DBUG "play":

```
DEFINE [{FKey number} "text"]
```

where "text" is the command string inside double quotes which will be assigned to the function key. If you were going to specifically define function key 1 to move north, for example, you would use the following DEFINE statement:

```
DEFINE 1 "MOVE NORTH"
```

Additional debugger commands in "play" while DBUG is running are:

JUMP {room name}

JUMP moves the debugger/player to the specified room exactly as if the player had moved there with the direction commands as in the normal course of play.

SET {room|object} {attrib#} [Y/N]

The SET command prints the status of the given attribute number for the given room or object. If a Y or N is supplied, the variable is set to this value.

EQU {variable} [value]

The EQU command prints the current value of the specified variable. If a value is supplied, the variable is set to the new value.

DBUG debugger can accept string variables as an argument to the EQU debugging command. The format is the same as above; a dollar sign must precede the variable's name. If a string in single or double quotes follows the variable name, the variable will be set to that value. It is not possible to use Visionary's formatted variables and codes in strings entered via the debugger.

PLACE {object} [{room}]

PLACE prints the name of the room where the specified object can be found. If a room name is supplied, the object will be moved to that room.

ROOM [{room}]

ROOM prints the complete status of the given room, or the current room if no room name is specified.

OBJECT {object name}

This command prints the complete status of the specified object.

Break Execution

The **Break Execution** menu item will stop execution of all programs for the remainder of the turn. The prompt will be given back, and normal process-

ing will resume as DBUG waits for input. Selecting this **may** leave resources allocated that would otherwise be closed by the programmer's software, but as always, they will be freed at the end of the session. Break Execution is used to halt runaway loops and other program bugs where the text interface would otherwise be unavailable.

The Visionary Linker

VLINK is the Visionary Linker, which turns games that have been compiled and debugged into the final, executable, (and distributable) format, combining the ".GAM" file with the ".WRD" file into one program file.

The command line format for VLINK is as follows:

```
VLINK {gamename[.GAM]} [-iwg]
```

The .GAM extension is optional, since Visionary links a file with this extension, and the three options are also optional. The {gamename} is the name of the game you wish to link.

The VLINK command options do the following:

i	do not generate an icon for this game
w	do not link the word file with the game, keep it separate
g	cause game to start up in Graphics Mode

When linking, the .WRD file **must** be in the same directory as the .GAM file, and the executable game will be put into the same directory. You may chose not to incorporate the .WRD file (which stores all of the text in the game) with the executable. An advantage of linking it with the executable is that text word output will be radically faster. One advantage of **not** linking it would be if the .WRD file is so large that the final program would be inconveniently large (something that might happen in a large text adventure).

Causing the game to start up in Graphics Mode is very much like performing a SCREENMODE GRAPHICS command as the first statement in your program, except that as the game loads and starts up, the text screen will open up in the **background**, so that the first screen the player sees is whatever graphics screen you care to show. This can be handy for a nice clean startup effect in an all-graphic game.

Finally the output of Vlink **will** be a large file; the base size of your game program will be approximately 200K, plus the size of your .GAM file, plus the size of the .WRD file, if it will be linked to the .GAM file.

! **Be sure before linking that you have enough disk space for the output file in addition to the .GAM and .WRD files.**

Chapter 5: Variables and Flow Control

Flow control deals with how to change the flow of execution of the program as the game progresses. The game progress should change as the player enters actions and responses. When creating your adventure program, you should anticipate the range of actions the player will try and and responses those actions will invoke at different points in the game, and build your game accordingly.

Variables

One of the critical elements you can use to exert control over the flow of the game is the use of variables. A **variable** is a word used as a name for a value which have change as the program is running—a variable value, in other words. For example, we might have a **numeric variable** named SCORE. SCORE, which initially starts at zero, is incremented by one each time a "hit" is made in the game.

Variables can also be used to store text **strings**. A string variable named TEMPERATURE, might start with a value—the text stored in it—of "cold", then be changed to "hot" in the course of play, then later to "boiling" as the program continues.

These two variable types, string and numeric, provide the "hooks" on which you can hang the changing play of your game. You can test a numeric variable such as SCORE, perhaps to see if the player's score is large enough to end the game with a win. You can also compare a string variable with a text string, perhaps to see if the TEMPERATURE has changed to "boiling" yet.

System Variables

Certain variables are automatically declared and maintained by the system. The following variables are built into Visionary. You can use them as variables in your own programs without defining them—Visionary automatically defines them.

Numeric System Variables

SCORE	Success rate of the player
ITEMS	The number of items in the player's Inventory
MOVES	Incremented by the system every turn.
RAND	Random number always returns value from 0 to 9999

MAXOBJ	Number of objects in adventure.
MAXROOM	Number of rooms in adventure.
LASTMOVE	Values 0..9, the player last moved N,S,E,W,NW,NE,SW,SE,U,D.
CHIPMEM	The amount of available chip memory read out in "K".
FASTMEM	The amount of available fast memory read out in "K".
MOUSEX	The value will always reflect the current position of the of the mouse pointer. If a graphics screen is open, the coordinates will reflect the mouse position within it, otherwise it will reflect the mouse position within the text screen.
MOUSEY	The value will always reflect the current position of the of the mouse pointer. If a graphics screen is open, the coordinates will reflect the mouse position within it, otherwise it will reflect the mouse position within the text screen.
LEFTBUTTON	This variable will contain a 1 if the left mouse button is pressed, or a 0 if it is not.
BUTTONPRESSED	This variable reflects the number of the last CLICK button that the player pressed, or 1000 if none.
TIME	HH.MM, represent the current time hour with hour values 0 to 23 and minutes 0 to 59
ERROR	Is settable by the programmer, and will normally be 0. If some operation, like a disk load of an image or sound, opening the speech device, or creating a screen, FAILS somehow, whether due to lack of memory or it couldn't find the file, the file was an incorrect format, or the player types unknown command, this variable will be set to 1. ANY statement that can generate an error during runtime debugging will cause the ERROR variable to be set to 1. Its value is returned to 0 after every successful turn.
VIDEOMODE	Assumes at the beginning of run, either a "0" for NTSC or a "1" for PAL. The variable can be written to as any other variable thereafter.
objPOS	The room number the object is in. Each object declared has a POS variable assigned to it with the variable name "objnamePOS". In other words, the room number where the object **ladder** is located is stored in the variable **ladderPOS**.

All other variables used by the adventure must be defined by you. Basically there are two classes of variables, numeric and string. Numeric variables are defined by mathematical formulas using operators following the Order of Precedence. String variables are text strings that are altered by the progress of the game.

String Variables

The format for defining variables in the VAR block in the main .ADV file is as follows:

```
${name}
```

In all references, a string variable will be preceded by the standard dollar sign "$". The initial value of this variable will be either be set to an empty or **null** string, or to the text string in single or double quotes following it. The maximum length of a string variable is 79 characters.

No "interpreting" of the string content is done in the VAR block of the .ADV file. The string will be just as you typed it. You will not be able to include in-line variable and screen code formatting, it won't work. This is done so you won't have to wade through the interpreted "garbage" in the string to find the actual text when you're looking at the cross-reference file.

Use of String Variables

Any text string displayed on the screen can have a string variable embedded in it—this process displays the content of the string variable as part of the text string.

At the place in the displayed text string where you want the string variable included, you would enter the at-symbol "@" first, followed by the dollar sign to indicate that this is a string variable, and then the variable name. The "@" indicator can also be used to embed numeric variables as well.

For example, at some point in the program you may display the text string "The water is _____." In the blank you want to display the current setting of the string variable TEMPERATURE mentioned above. You would code the statement to display the text string as follows:

```
T The water is @$TEMPERATURE .
```

If the current setting of the string variable TEMPERATURE was hot, the string would be displayed on screen as:

```
The water is hot.
```

Note the trailing space between the embedded string variable and any text which follows it. This space is very important. Without it, the Visionary compiler can't separate the variable name from the text string it is embedded in. If the trailing space was left out of the string above, your debugger wouldn't be able to find the variable "$TEMPERATURE."

anywhere in the code. The trailing space is stripped out when the variable is embedded in the text string.

String Commands

You can change the contents of a string variable with the Visionary string commands. These commands are:

- LET
- COMPARE
- ROOMNAME
- OBJNAME
- GETSTRING
- GETCHAR
- GETNUM
- GHOST

LET

The LET command allows you to set the contents of a string variable, too. It takes the following format:

```
[LET] ${name} := (${stringvar}|"text")
```

The actual word LET in the command is optional. The text may be any standard text string, including formatted variables.

For example, in BASIC, given the variable A$ which contains "Hello", the following command:

```
LET A$ = A$ + ", " + A$
```

would change the value stored in A$ to "Hello, Hello". In Visionary, the following statement does the same thing:

```
$A := "@$A , @$A"
```

Again, note the trailing space after @$A, before the comma. Since the second @$A has no portion of text string following it – the double-quote is the end-of-string indicator – it does not require a trailing space.

The equate operator is also legal in a LET command:

```
LET $A := $B
```

COMPARE

```
COMPARE (${stringvar}|{literal}),
    (${string}|{literal}), {int var}
```

Compare will give the integer variable at the end of the command one of three values, depending on the relationship of the first string variable or literal, quoted text string and the second. If the string on the left evaluates to be **less than** the second, according to the ASCII value of its contents, -1

is returned. If it is greater, 1 is returned, and if the two strings are equal, 0 is returned.

Whether the letters in the string are upper or lower case is not important—like most Visionary commands, COMPARE is case-insensitive. Internally, the program converts all text in the string to one case before making a comparison. So comparing "HELLO" to "hello" would return the value 0, becasue the program sees these two strings as equal.

ROOMNAME

```
ROOMNAME ({room}|{expression}|THISROOM),
   ${stringvar}
```

Puts the name of the specified room, or the current room if "THISROOM" is true, into the string variable given. The room number can also be derived by evaluating an expression, and its name stored in the string variable.

Examples:

```
ROOMNAME Haunted_Library, $ARoom   ;this
   one's kinda silly.
ROOMNAME 7, $AnotherRoom
ROOMNAME THISROOM, $MyRoom
ROOMNAME (TimesVisited * 2), $AnotherRoom
```

OBJNAME

```
OBJNAME ({object}|{expression}), ${stringvar}
```

Substitutes the name if the given object or object number into the given string variable. What really happens, is that the first name given to that object is returned. If the object doesn't have any names, then the object's system name is returned.

GETSTRING

```
GETSTRING ${stringvar}
```

Accepts input from the player through the keyboard and sends the text into the string variable given. Up to 77 characters may be input. The prompt "-" is printed at the beginning of the input line.

GETCHAR

```
GETCHAR ${stringvar}
```

Accepts one alphanumeric keypress and sends it to the text string variable given. Any ASCII value may be sent, and the statement will wait until the player does press a key.

GETNUM

```
GETNUM {int var}
```

Accepts a number from the player, in the range of -32768 and 32767, and puts it in the integer variable given. The prompt "-" is printed at the beginning of the line.

GHOST

```
GHOST (${stringvar}|"text") [, TURN]
```

Causes the argument, whether a string variable or a literal text string, to be passed to the Visionary player command interpretter as if the player had really typed it. The command is then executed (if no errors) and control returns.

In-Line Formatting

All strings and text statements allow in-line formatting of ASCII codes. This is done by inserting a backslash "\", followed by any one of the following:

nnn	a 3-digit DECIMAL number from 0 to 255. This creates one character of that ASCII code number
N	\N in a text string causes a carriage return and line feed
R	\R causes a carriage return only. The cursor goes back to the beginning of the line
F	\F causes a form feed, or CLEAR SCREEN to be generated. If the printer is on, a new page will roll in.
T	\T will cause a tab of 8 characters to appear
B	\B will generate a backspace
V	\V will generate a vertical tab, or 8 \N's
G	\G will generate a screen "beep", or bell
\	to print a backslash, simply type \\, which will print **one** backslash

LENGTH

```
LENGTH (${stringvar}|"text"), {intvar}
```

Puts into the specified integer variable the number of characters in the string variable or literal text string.

LEFT

```
LEFT (${stringvar}|"text"), {number},
    ${stringvar}
```

Moves the left-hand portion of the first string expression to the string variable on the right. The middle numeric expression specifies how many of the characters to copy.

MID

```
MID (${stringvar}|"text"), {start}, {number,}
    ${stringvar}
```

Moves the middle of the first string to the string variable on the right. Copying starts at the character indicated by the value of the expression {start}, and moves {number} characters.

RIGHT

```
RIGHT (${stringvar}|"text"), {number},
    ${stringvar}
```

Moves the right-hand portion of the first string expression to the string variable on the right. The middle numeric expression specifies how many of the characters to copy.

VALUE

```
VALUE (${stringvar}|"text"), {intvar}
```

If the string holds an integer number, its numeric value is stored in the given integer variable. If an error occurs (the string was not really a valid integer), the ASCII value of that character will be stored in the given integer variable.

UPCASE

```
UPCASE ${stringvar}
```

Sends all alphabetic characters in the string variable into upper case.

DOWNCASE

```
DOWNCASE ${stringvar}
```

Sends all alphabetic characters in the string variable into lower case.

System String Variables

The system sets and maintains a set of string variables that are reset automatically as the game is played. These are:

$LastLine	The player's last text command, verbatim.
$LastDir	Contains, at all times, the name of the last direction player moved.

$SubjNoun	The subject noun in the last sentence, or ""
$ObjNoun	The object noun in the last sentence, or ""
$SubjAdj	The subj. noun's adjective in the last sentence, or ""
$ObjAdj	The obj. noun's adjective in the last sentence, or ""
$Verb	The verb in the last sentence, or "".

The last five variables, $SubjNoun, $ObjNoun, $SubjAdj, $ObjAdj and $Verb, are non-null only if the sentence was interpreted as a non-vocabulary action, that is, if the player's command applied to one of an object's ACTION blocks. Their values will be unpredictable if there is an error in the command line.

System Error String Variables

Nine additional error-related string variables are declared automatically by the compiler. These variables are for dealing with error conditions when the player types a command that is not understood by the command line parser. Each is suited only for specific situations, but can be customized by the game designer, say, for languages other than English. Error number 2 is actually a report on the player's current score. In order, they are:

$Error1	"You can't go @$LastDir from here."
$Error2	"Your current score is @Score ."
$Error3	"You can't see a @$SubjAdj @$SubjNoun here."
$Error4	"I don't understand."
$Error5	"You can't see a @$SubjNoun or a @$ObjNoun here."
$Error6	"Specify which @$SubjNoun you mean next time."
$Error7	"Specify which @$ObjNoun you mean next time."
$Error8	"You can't see a @$ObjAdj @$ObjNoun here."
$Error9	"You can't do that to a @$SubjNoun ."

Flow Control Commands

Another mechanism the Visionary Language provides for altering the program interatction with the player depending upon the actions and input of the player is a set of commands which serve as flow control commands. Flow control commands follow the basic tenet of logic: "IF A is true THEN B occurs ELSE C occurs."

The Flow Commands catagory consists of seven commands:

- IF
- ELSIF
- ELSE
- AND
- OR
- WHILE
- END

The IF and WHILE commands utilize the other Flow Control commands. Each of these powerful commands is described in turn below.

IF

The IF command is an extremely powerful Flow Control command. The execution of the program will branch dependent upon the meeting of certain conditions you define. A series conditions can be strung together using the ELSIF command in conjuction with the IF command, or multiple simulataneous conditions can be required using the AND command with the IF command.

In the IF command sytax always contains a THEN after the condition. Statements after THEN will be executed if the condition is met. Where the condition is not met, the program drops down to look for either an ELSIF command, an ELSE command or and END command. The Flow Control Commands AND and OR are used with the condition expression of the IF command.

IF

SYNTAX:

IF {condition} THEN
...
[ELSIF {condition} THEN]
...
ENDIF

COMMENTS:

The IF statement allows a program to decide which sections of code it will execute. The {condition} can be any legal Visionary expression. Legal expressions are:

{const}	(simple constant)
{var}	(simple variable)
{expression} * {expression}	(multiplication)
{expression} / {expression}	(division)
{expression} MOD {expression}	(modulus)
{expression} + {expression}	(addition)
{expression} - {expression}	(subtraction)
{expression} < {expression}	(less than comparison)
{expression} <= {expression}	(less than or equal comparison)
{expression} = {expression}	(equal to comparison)
{expression} >= {expression}	(greater than or equal ")
{expression} > {expression}	(greater than comparison)
{expression} # {expression}	(not equal to comparison)

{object|PLAYER|expression IN THISROOM|expression}	(is object in room)
PLAYER HAS {object|expression}	(player has object (or object #) comparison)
PLAYER CANGO {expression}	(can the player go a direction # comparison)
{PREP|OBJNOUN|object|room} IS {"preposition"|object|attribute}	(attribute comparison
{PREP|OBJNOUN|object|room} NOT {PREPOSITION|object|attribute}	(inverse attribute comparison)
{expression} AND {expression}	(conjugate two simple expressions)
{expression} OR {expression}	(conjugate two simple expressions)
{int variable} := {expression}	(equate)

NOTES:

- THISROOM, as used with the "IN" operator, implies whatever room the player is currently in.
- PLAYER IN THISROOM will **always** return true, or 1.
- PREP IS|NOT "preposition": see if player's last line contained the given preposition, from the list of **valid** prepositions. Note: the preposition **must** appear in quotes!
- OBJNOUN IS|NOT {object}: see if the object noun from the player's last line was the given object.
- PLAYER CANGO {expression}: the expression is expected to be a value between 0 and 9, inclusive, representing N, S, E, W, NE, NW, SE, SW, U, D.

The default order of precedence for the operators, from highest to lowest by line, is:

PRECEDENCE	OPERATOR(S)
1	*, /
2	MOD
3	+, -
4	<, <=, =, >=, >, #, IN, HAS, IS, NOT, CANGO
5	AND, OR
6	:=

The order of precedence may be changed with the use of paretheses, "(" and ")". To negate a mathematical expression, subtract it from 0. To negate a boolean or logical expression, subtract it from 1.

An IF statement may also be followed by one or more ELSIF clauses, and then may be followed by a single ELSE clause, if needed.

EXAMPLES:

```
if (health / 2 > 67) AND (PLAYER HAS
  Golden_Sword) THEN
 ...
ELSIF 1-(Chamber IS DARK) OR (PLAYER HAS
  Lamp) OR (PLAYER CANGO 0) THEN
 ...
 ELSIF ((value > -5) AND (value < 16)) OR
  (value = 22) THEN
 ...
 ELSE
 ...
 ENDIF
```

ELSIF

SYNTAX:

ELSIF {condition} THEN

COMMENTS:

Like IF, ELSIF can accept any legal Visionary expression as a condition. AND and OR commands can also be used with the condition expressions and the ELSIF statement always ends with THEN.

EXAMPLES:

```
ELSIF (Magic Sword in INVENTORY) THEN
T You can slay dragon.
ENDIF
```

ELSE

SYNTAX:

ELSE

COMMENTS:

When previous conditions in an IF or ELSIF command fail, then what follows the ELSE command prevails.

EXAMPLES:

```
IF (Chamber is DARK) THEN
T Turn on light switch.
ELSE
GO Closet
ENDIF
```

AND

SYNTAX:

IF {condition} AND {condition} AND {condition} THEN

COMMENTS:

The AND command is used when multiple conditions must be met.

EXAMPLES:

```
IF (Chamber is DARK) AND (Player HAS Lamp)
   AND (Player HAS Matches) THEN
 T Light Lamp
ENDIF
```

OR

SYNTAX:

OR

COMMENTS:

The OR command is used when any one of several conditions of an IF command can be met in order for the program to execute code following the THEN.

EXAMPLES:

```
IF (Player HAS Gun) OR (Player HAS Knife)
   THEN
 T Slay Dragon
ENDIF
```

END

SYNTAX:

ENDIF

COMMENTS:

When an END command is used, the program stops executing the current section of code and returns to the main program. END can be used with many other commands including IF and WHILE.

EXAMPLES:

```
IF (chamber is DARK) THEN
 T Strike a Match
ENDIF
```

WHILE

A second powerful Flow Control command is the WHILE command. It makes the program do something as it is executing a certain section of code found in the expression for the WHILE command. The program will continue to "do" what is has been instructed by the expression of the WHILE command fail or evaluates to "0". The WHILE command can utilize the commands AND, OR and END in the same manner as the IF command.

SYNTAX:

WHILE {expression} DO

.

.

.

ENDWHILE

COMMENTS:

WHILE allows the code that follows it to be executed a number of times, repeating indefinitely until the expression in the WHILE statement is evaluated as "0" or "false". Like the IF command, it has a special syntax which includes a special associate command DO. Every WHILE command is followed by the word DO after the expression of the WHILE command.

EXAMPLES:

```
count := 1;
WHILE count < 100 DO
 T This is a test!
 count := count + 1
ENDWHILE
```

Chapter 6: Graphics Handling

What Graphics Can Do

In this chapter, we will discuss the commands and techniques associated with using graphics in Visionary games. Your adventures will be more vivid with the use of graphics. Not only can you use graphics as the background for your adventures as various rooms, you can graphically **show** objects for use by the player. Players can interact with objects depicted in the graphics by clicking on a defined "hot spot" to cause certain events to unfold.

Graphical commands are divided into four different types—setup, drawing, effects and interaction—according to their uses in Visionary programs. These commands are listed below.

Command	Topic
CREATE SCREEN	Setup
LOAD SCREEN	Setup
SHOW SCREEN	Setup
UNLOAD SCREEN	Setup
SCROLLBAR	Setup
MENUS	Setup
COLOR	Drawing
COPY	Drawing
LINE	Drawing
MASK	Drawing
MODE	Drawing
PALETTE	Drawing
RECT	Drawing
TEXT	Drawing
CYCLE	Effects
DISSOLVE	Effects
FADEFROM	Effects
FADETO	Effects
SCROLLTO	Effects
CLICK	Interaction
READBUTTONS	Interaction
REMOVE	Interaction

Setup Commands

Visionary allows the programmer to have up to 25 graphics screens, numbered 0 through 24, in memory simultaneously. These **screen buffers** can be declared and used in any numerical order you like; you may use screen 24 without first defining screens 0 through 23.

The Visionary graphics screen is always an non-draggable screen with no title bar, therefore it will always start at the upper left corner of the physical display, at the position you set in the Amiga Preferences program for your system.

These screens can be created in two ways, by loading a graphics file into memory or by creating the screen with program commands.

LOAD SCREEN

Visionary supports IFF image files, including HAM images generated by Aegis SpectraColor™ and others. Within RAM limits, these images can be imported from disk at any point during the game using the command

```
LOAD SCREEN {buffer number},
    (${stringvar}|"filename")
```

The buffer number specified must be in the range 0 to 24, inclusive, and the name of the file to be loaded must be given in terms of a literal text string or the name of a string variable which contains the filename. For instance:

```
LOAD SCREEN 0, "PICS:Rooms/Dungeon"
LOAD SCREEN count, $PicName
```

If everything is OK—the file is present and there is sufficient CHIP RAM to hold the image—the IFF image will be loaded into memory and that buffer number will thereafter be associated with that image, at least until it is reassigned by loading another image over it or by unloading it.

CREATE SCREEN

Visionary also allows you to open graphic screens that start out blank rather than forcing you to load an image from disk. This is accomplished with the command

```
CREATE SCREEN {buffer number}, {X}, {Y},
    {depth}, (HIRES|LORES|LACE|NOLACE|HAM|
    HALFBRITE)
```

Once again, as in all graphics commands, the buffer number must be in the range of 0 through 24. X is the width in pixels of the screen. Valid choices for the X expression are 320 in lo-res and 640 for hi-res images. Y is the height of the image in pixels. Under NTSC, the screen standard used in the USA, Canada, and parts of Australia, the valid choices are 200 or 400 with Interlace. Under the PAL screen system which is standard in Europe, the valid choices for {Y} are 256 or 512 in Interlace.

The screen's depth is the number of bitplanes the image uses, which is determined by the number of colors in the image. Valid choices for depth are 1 through 6. The following is a table of the valid choices for depth and their associated meaning.

Bitplanes	Colors	Resolution Modes	Video Modes
1	2	Lores, Hires	Color
2	4	Lores, Hires	Color
3	8	Lores, Hires	Color
4	16	Lores, Hires	Color
5	32	Lores	Color
6	64/4096	Lores	HAM/Extra Halfbrite

Although the screen's video modes will to some degree be automatically set by the X and Y values specified, you can provide a series of words separated by spaces to explicitly set the video modes. Any combination can be used, though in the event of conflicting words, only the last will be used, and certain modes are not available in all resolutions. If no modes are specified, LORES and NOLACE are automatically assumed.

SHOW SCREEN

Simply loading a screen into memory does NOT mean it is immediately displayed. To cause an already-defined screen to be displayed, the

```
SHOW SCREEN {buffer number}
```

command is used. The image is displayed in a screen immediately **behind** the standard text screen, which is not affected by this command. No more than one graphic screen will be displayed at a time; any currently-displayed image will be superceded.

To remove a graphic screen from the display without removing it from memory, the SHOW SCREEN command may be given any buffer number larger than 25. Visionary will attempt to show a screen buffer that does not exist, which will cause the previously-displayed screen to go away.

SCROLLBAR (ON|OFF)

This command allows you to explicitly permit or forbid the player to drag the text screen up and down. SCROLLBAR ON is the default setting.

This is one of two commands which are commonly left **out** of a program until after the debugging process is complete. If SCROLLBAR OFF is active during debugging, you will not be able to drag the graphic screen out of the way to enter DBUG commands in the text interface—unless you have provided a text window in your adventure, you will not be able to debug your program.

MENUS

This command dictates whether the Menu bar is available to the player. MENUS ON is the default.

```
MENUS (ON|OFF)
```

The MENUS OFF command is one of two commands usually disabled until after the debugging process is complete. When MENUS and SCROLLBAR are both OFF, access to the back screen, which may have the text interface, is disabled.

UNLOAD SCREEN

If you have run out of screen buffers because you already have 25 loaded, or perhaps because you want more memory for, say, sound samples, you can use the

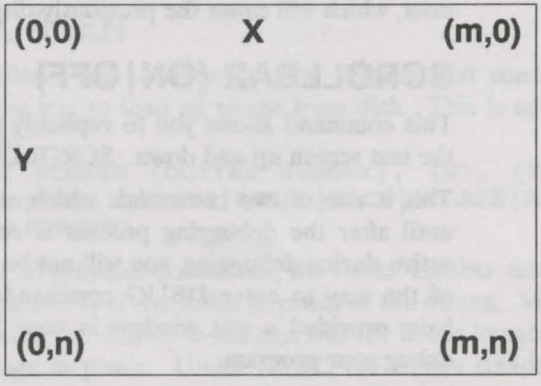
```
UNLOAD SCREEN {buffer number}
```

command to de-allocate all memory associated with a given screen buffer. If this buffer is currently being displayed, the displayed screen will go away and the Visionary text screen will move to the front, the top of the display.

Drawing Commands

Now that you know the commands used to setup and maintain graphic screens, we can discuss rendering images in them. Visionary allows you to modify the contents of any graphic buffer whether it is currently displayed or not, and cut and paste images between them.

Areas of any graphic screen are referred to using a system of X and Y coordinates with the zero point at the upper left corner of the screen. The coordinate system in Visionary screens works as follows:



The Visionary Coordinate System

The **m** value can be either 319 in low resolution, or 639 in hi-res.

The **n** value for an NTSC system can be 199 for the standard display or 399 for Interlace.

The **n** value for a PAL system can be 255 for the standard display or 511 for Interlace.

PALETTE

It is possible to define and redefine the color palette within each screen you have open with the

```
PALETTE {buffer number}, {pen number}, {R},
    {G}, {B}
```

command. This sets the Red, Green and Blue (RGB) values for the given pen number in the screen buffer you specify. The RGB values can take on any value between 0 and 15, inclusive.

Some sample values:

Color	R	G	B
Black	0	0	0
White	15	15	15
Red	15	0	0
Orange	15	8	0
Yellow	15	15	0
Green	0	15	0
Blue	0	0	15
Violet	15	0	15
Lt. Grey	12	12	12
Dk. Grey	5	5	5

COLOR

```
COLOR {buffer number}, {pen number}
```

command, which sets the color of all following line and rectangle operations in the given buffer to the pen number you supply. This pen number has 0 as a lower bound, and an upper bound that is based on the depth (number of bitplanes) in that screen:

BitPlanes (Depth)	Maximum pen number
1	1
2	3
3	7
4	15
5	31
6	15(HAM)/63(Extra Halfbrite)

Although extra halfbrite mode displays 64 colors, only 32 of those are directly settable; the upper 32 are half as bright as the lower 32.

Be warned that HAM mode is NOT easy to draw in. Read the Amiga Hardware Manual for more information.

The default color in any newly-opened screen buffer is 1.

MODE

There are three drawing modes in Visionary graphics. They are DRAW, XOR, and OVERLAY. Only the first two apply to lines, rectangles and text.

The format of the MODE command is:

```
MODE {buffer number}, (DRAW|XOR|OVERLAY)
```

The DRAW mode causes all drawing operations to entirely replace the background, only the area behind the entity being drawn, with the new color or image. TEXT will be rendered with an opaque block behind the letters, wiping out anything they are on top of. The default drawing mode in all new buffers is DRAW.

XOR, short for "Exclusive Or", mode causes the result to be a combination of the source and the destination. See table below.

OVERLAY mode is used only for COPY operations, and causes any holes (color 0) in the source image to be transparent to the background in the destination image. For the LINE and RECT commands, it behaves exactly like DRAW mode, and for the TEXT command, it causes the text to be overlayed onto the background, with the background color transparent. See COPY.

The following table shows the resultant image values based on the original destination values and the source values for each of the three drawing modes.

		source value	
		0	1
DRAW	dest 0	0	1
	value 1	0	1
XOR	dest 0	0	1
	value 1	1	0
OVERLAY	dest 0	0	1
	value 1	1	1

LINE

The simplest draw operation is the

```
LINE {buffer number}, {X1}, {Y1}, {X2}, {Y2}
```

command. It causes a line to be drawn in the specified screen buffer between the endpoints (X1, Y1) and (X2, Y2). Single pixel POINTS can be drawn by supplying the same (X,Y) coordinates for both endpoints.

RECT

The next draw operation is

```
RECT {buffer number}, {X1}, {Y1}, {X2}, {Y2}
```

which draws a rectangle in the given screen buffer between the upper left corner (X1, Y1) and the lower right corner (X2, Y2).

TEXT

Text may also be rendered into the graphics screens with the

```
TEXT {buffer number}, {X}, {Y},
    (${stringvar}|"text")
```

command. (X,Y) specifies the screen coordinates of the left side of the baseline of the text, which is the **bottom** of the normal characters, excluding characters with descenders, like j's and g's. The text string may contain the same inline-formatted variable values as for the text screen command "T", but ANSI codes for text style changes will not work.

PIXEL

This command reads the pen color of the indicated graphics buffer at the designated x,y position and stores it in the integer variable given.

```
PIXEL {buffer#} {x}, {y}, {int_variable}
```

Block Transfer Commands

We have covered all of the rendering operations except those used in copying image blocks between screen buffers.

COPY

Visionary allows you to move whole images about in memory easily and quickly. This is done with the command

```
COPY {source buffer}, {X1}, {Y1}, {X2},
    {Y2}, {dest. buffer}, {X}, {Y}
```

which frames a block with the upper left corner at (X1,Y1), the lower right corner at (X2,Y2) in the source buffer, and then copies that whole image to the coordinates (X,Y) in the destination buffer. If the values specified

would cause the box to overlap the edges of the screens, the block's size is automatically cut down.

> **! This operation will fail if the source buffer and the destination buffer do not have the same number of bitplanes (depth). Usually the source and destination buffers will have the same video modes, though this is not required.**

DRAW and XOR modes are very straight-forward – just give the command. If you have the memory, the screens will be handled as you direct. For OVERLAY mode, however, you will first have to execute a MASK command.

MASK

This Mask is used by the system to decide how to make color 0 transparent. What really happens during an OVERLAY COPY is that the source image block is manipulated in a hidden buffer to produce a Mask which specifies each pixel that contains color 0.

Since only you know how big your COPY operations will be, it is left up to you to perform at some point in the setup of your game the

```
MASK {buffer number}
```

command, which tells the system to build this mask image in the pre-allocated buffer you specify. This buffer must be at least as large as the largest block you wish to COPY. It must not only have the same X and Y extent, but it must also have the same number of bitplanes. It may have the same resolution, although this is not required.

Video Effects Commands

Visionary also provides a number of facilities for color-cycling and for making nice transitions from screen to screen.

CYCLE

Many paint programs such as Aegis SpectraColor and Deluxe Paint allow the artist to specify color ranges that will be "color cycled" to create some effect. Visionary reads the color-cycling information from the IFF image file and puts it under the programmer's control with the command

```
CYCLE (ON|OFF|ONCE|RESTORE)
```

CYCLE is a command that operates globally—any currently displayed screen will be affected if it has active color ranges. If no graphic screen is being displayed, no effect will be seen.

Setting	Description
ON	Activates color cycling
OFF	Deactivates color cycling
ONCE	Causes one "tick" of the clock which counts down toward the next color cycle. Slower cycle ranges may not move immediately.
RESTORE	Returns the original color palette to the current image, including any changes made with the PALETTE command.

DISSOLVE

One of the most flexible transitions is the

```
DISSOLVE {from buffer}, {to buffer}, {type}
```

command, which causes a form of pixel dissolve between two graphic buffers, actually changing the contents of the "to" buffer, which should be the one currently being displayed.

> **! The two buffers MUST be of the same depth and dimensions, though they needn't have the same video modes or color palettes. The "from" buffer's palette will be copied to the "to" buffer when the operation finishes.**

With {type}, you can specify a particular dissolve type. This must be a value between 1 and the number of bytes in a single bitplane of the image, minus one:

Resolution	Bytes per bitplane	Maximum Type value	
NTSC			
	320 x 200	8000	7999
	320 x 400	16000	15999
	640 x 200	16000	15999
	640 x 400	32000	31999
PAL			
	320 x 256	10240	10239
	320 x 512	20480	20479
	640 x 256	20480	20479
	640 x 512	40960	40959

The general formula for the maximum type value is (X / 8) * Y - 1.

The dissolve type specifies the order in which the bytes in the "from" image are to be moved. Because of the way this works, no even values or multiples of 5 are allowed; they will not transfer the entire image.

A value of 1 will cause a top-to-bottom wipe. If the maximum type value is used, a bottom-to-top wipe will occur. Other values to try include 3, 7, 11 and Maximum Type value divided by 2 and rounded up to be odd, although any odd value not divisible by 5 will work.

FADES and SCROLLS

Color fades are also possible with Visionary. The

```
FADETO {buffer number}, {R}, {G}, {B},
    {delay}
```

and

```
FADEFROM {buffer number}, {R}, {G}, {B},
    {delay}
```

commands allow you to send the entire color palette of any screen, displayed or hidden, to and from the Red, Green and Blue values specified—see the color table in the description of the PALETTE command. Between each of the 15 color changes comprising the fade, the computer will pause for the number of vertical blanks indicated by {delay}. A vertical blank is 1/60th of a second for NTSC machines and 1/50th of a second for PAL machines. A value of 0 for delay will make the change happen instantaneously.

SCROLLTO

The

```
SCROLLTO {buffer number}, {X}, {Y}
```

command allows Visionary to perform larger-than-page scrolling. Given a low-resolution, non-Interlaced screen is larger than 320 x 200, SCROLLTO allows you to specify the coordinates within the screen that are to be displayed at the upper left corner of the display. If values given would put the lower right corner of the screen too far into the center of the screen, they are cut down so that the screen never scrolls off the display. This can be done to hidden buffers as well as the one currently displayed.

Graphic Interaction Commands

What would all of these great commands be worth if the player could not communicate with the game graphically?

CLICK

Visionary supports mouse interactions by allowing the programmer to specify that up to 50 regions on the screen are buttons to be pressed by the player clicking on them with the left mouse button.

```
CLICK {button number}, {X1}, {Y1}, {X2},
   {Y2}, (subroutine|object|room)
```

allows you to set, as a button number specified by a value between 0 and 49, a rectangular region defined by the upper-left corner (X1,Y1) and the lower-right corner (X2,Y2) which will react in a defined way to a mouse click. Following that can be the name of a Visionary Subroutine, an Object or a Room.

When the player clicks in this zone, whatever code or subroutine block name you specified will be executed. The graphical region will not automatically change visibly when the player clicks the button.

Buttons remain active until they are removed or redefined. Buttons are also prioritized; that is, they can overlap each other and create more-complex interactions. For example, suppose you want several buttons on the screen, so that the player can click on these for actions. Suppose also that you want the entire screen to be considered a button. Visionary graphic buttons are numbered 0 through 49. In the same order, the system checks each button's bounds when a mouse-click occurs. With the following CLICK definitions:

```
CLICK 0, 10,11, 70,20, ActionHit
CLICK 1, 10,30, 70,40, ActionEat
CLICK 2, 10,50, 70,60, ActionGrab
CLICK 3, 0,0, 319,199, ScreenPressed
```

the first three CLICK commands define the action buttons, and the fourth sets up a lower priority, higher number button that treats the entire screen as a click zone, calling the subroutine, ScreenPressed, when the player clicks on any region within the screen but not within one of the action buttons.

There are many possible uses for screen buttons. Following the lead of Icom Simulations™ games like ShadowGate™ and Uninvited™, there could be a small box with the current scene in it, with the scene clickable and objects within the scene clickable. We could also have an on-screen compass with the current valid directions highlighted and clickable.

Except for considerations of priority, buttons do not have to be defined in any particular order. If you like, you can define button number 49 without first defining buttons 0 through 48, for example.

READBUTTONS

The READBUTTONS command checks for mouse clicks and executes them. During a WHILE execution, buttons are not read. Use of EMPTY after the Readbuttons command empties the mouse-click input queue.

```
READBUTTONS [EMPTY]
```

REMOVE

The command

```
REMOVE {button number}
```

will deactivate the given button.

Graphics-Related Variables

Visionary automatically declares some graphics-related variables for your use. They are READ-ONLY variables (their values may not be set by you, only read) that deal with the player's input.

ButtonPressed

This variable contains the value 0...49, the number of the graphics button most recently pressed by the player. It will contain a value of 1000 until the player first a button.

LeftButton

This variable will always contain a 1 or a 0, depending on the state of the left mouse button. If the player is currently holding it down, LeftButton will be a 1. Otherwise, LeftButton will contain a 0.

MouseX

This variable will always reflect the mouse pointer's X position on the screen. This value is relative to the upper-left corner of the graphics screen if there is one open, or the text screen of there is NOT a graphics screen open. The value will be in the range of 0 to whatever the maximum X value is for the type of graphics screen you are displaying, or 639 for the text screen (See the figure in section III, "Visionary Graphics: Drawing Commands"). If there is only the text screen open, and the player pulls the screen down, weird values may result if this variable is referenced when the player moves the mouse above the text screen.

MouseY

This variable always reflects the mouse pointer's Y position on the screen, using the same criteria as MouseX, above.

Chapter 7: Audio Commands

Introducing Audio

Sound is another important element in games. It can be used to establish mood, to introduce clues to the player, and to surprise the player.

Internally, Visionary supports playback of sampled or **digitized** sounds using the Amiga's stereo sound output. You have a channel for playback to the left speaker and one for playback to the right speaker.

Many products are available to sample sound and edit the resulting sound file on the Amiga Computer. One such product is the Aegis AudioMaster™ series sound-editing software. With AudioMaster you can replicate length sound samples using the looping feature, and save disk space. If you need digitizing hardware, the Aegis SoundMagic™ product will give you outstanding quality sound and it includes AudioMaster as well.

The Visionary audio commands are:

- Load Sound
- Play Sound
- Stop Sound
- Unload Sound

Audio Command Summary

Visionary gives you a maximum of 25 sound buffers. You can load a maximum of 25 sound samples into memory at any given time. The sound buffers are numbered 0 through 24.

Visionary supports four commands for handling sound samples. They work with any IFF-8SVX sampled sound file smaller than 128K.

LOAD SOUND

The LOAD SOUND command causes the system to load into the given sound buffer number the specified IFF-8SVX file. The sound file can be named either explicitly or in a string variable.

```
LOAD SOUND {buffer #},
    (${stringvar}|"filename")
```

Examples:

```
LOAD SOUND 0, $AudFile ;loads buffer 0 with
    the filename in the string variable
    LOAD SOUND AudBuf,
    "SOUNDS:Thuds/Scream.SND"
```

If there is an error, as when the file is not being accessible or there is not enough memory, the command will abort and the sound will not be loaded.

PLAY SOUND

Once a sampled sound has been loaded into memory, it can be played. This is accomplished with the PLAY SOUND command.

```
PLAY SOUND {buffer #}, {channel #},
    {iterations}, {volume}, {period}
```

The channel number is expected to be either 0 or 1. The 0 is the left speaker, 1 indicates the right speaker.

The {iterations} value specifies the number of times you want the sound to be repeated. If you supply a value of 0, the sound will be repeated forever, or until it is halted with the STOP command.

The {volume} can be set to any value between 0 and 64. A value of 0 is "volume off", while 64 is full volume.

PLAY SOUND expects you to supply a {period} setting. Indirectly, this is a value that determines the frequency, or pitch, of the sound. Technically, considered at the hardware level, it is a measure of the number of "bus cycles" to wait between pumping out a new sample—so the smaller the value, the higher the pitch will be.

The system imposes a lower limit for the {period} of 124. The sound will be reduced to little more than rumbles and clicks with any value over 1500, so any value in this range 124 to 1500 is probably desirable. A period of 0 will use the sound's natural pitch. For more information on the correlation between sampling period and sampling frequency, see the *Amiga Hardware Manual*.

If a sample is already playing on the selected channel when this command is issued, the old sound will stop playing and the new sound will start.

STOP SOUND

Once a sound has started playing on some channel, it is possible to shut it off in mid-stream. This is done with the

```
STOP SOUND {channel #}
```

command, which stops any sound currently playing on the channel number specified. The channel number can be either 0 or 1.

UNLOAD SOUND

When you are finished with a particular sound, you should remove it from memory to make room for other data. This is accomplished with the command

```
UNLOAD SOUND {buffer #}
```

This will free all Chip memory associated with the sample and make space for other data. Once this command is run on a sound buffer, the sound that was in it is lost and unplayable. The buffer is then ready to hold some new audio sample.

Audio Hints & Techniques

Sound effects can help to build atmosphere in a game. Doors slamming, ghoulish laughter, distant screams, even the monotonous hum of computers or a ship's engines can all heighten the player's appreciation of the game.

You should probably decide early in the development process just how many samples you are willing to deal with, since this is an issue that helps determine how many disks your game is going to occupy. Keep in mind that incessant disk access will bother the player, but that it may not be feasible to load all of the samples into memory at once.

Music

You might find that you want some music to play in the background while the game progresses. This might be done in two ways.

First, you could load a looping sequence, sound sample or a snippet of music, and play it with an iteration count of 0. The music would then loop continuously in the background.

Another option would be to get hold of a SMUS sound player and call it through Visionary's DOS command. Other players, like the public domain SoundTracker/NoiseTracker might work well, too.

The creation of music for your game is covered in the next chapter.

Chapter 8: Music Commands

Adding Music

Visionary contains some very powerful features allowing you to integrate music into your programs. Visionary supports MED (Music EDitor) MMD0 song files, which are very flexible and powerful. Following is a brief explanation on how to set up such music.

Finding MED

MED is a shareware music creation system written by Teijo Kinnunen which is well worth its shareware price. It should be available through the Fred Fish collection and many other outlets.

Using MED

Though it can be a little technical, MED is an excellent tool. Not only does it allow the composer to play music on a MIDI keyboard and enter notes into the song without too many mouse and Amiga-keyboard interactions, but it can also **play** music through the MIDI port.

MED can generate a number of different music file formats, including what is called the "MMD0" format, which is the one Visionary reads. We won't get into the details of actually creating music with MED – see the MED documentation for that – but we will help you save your score in the correct format for Visionary.

Once your song is finished, select the FILES button in the MED window, and enter, in the text boxes provided, the path and filename of the new save-file, then select the small "SAVE" button near the top. A selection of file types should appear. Select "MODULE". That's all there is to it. The file will be saved in the Visionary-compatible "MMD0" format.

Making Beautiful Music Together

In your Visionary program, you must also do a few things to make the music happen. First you must load the song. Do this with the LOAD SONG command. When you are ready to actually play the song, you must first execute a ENABLEMUSIC command.

```
ENABLEMUSIC
```

When the ENABLEMUSIC command is executed, any sound samples that are currently playing will cease. If the song contains external MIDI tracks,

you should use the ENABLEMUSIC MIDI command, otherwise you should use ENABLEMUSIC NOMIDI.

Now you can execute the PLAY SONG command. When you do, the song will begin to play in the background. The PLAY SONG command will exit immediately after executing.

When you decide you want the music to stop, simply call the STOP MUSIC command. The system will remember its state, and later you may call the CONTINUE SONG command to resume play where it left off. This will work even after executing a DISABLEMUSIC, followed by an ENABLEMUSIC command.

When you are through with the music or want to run some sound effects, you must execute the DISABLEMUSIC command

```
DISABLEMUSIC
```

which will stop playing any song in progress and return the system to sampled sound mode.

```
PLAY
```

If you PLAY another song while a song in progress has the obvious result; the new song simply starts playing.

As with sounds, it is possible to load and unload songs without being in the right mode (ENABLEMUSIC vs. DISABLEMUSIC).

Chapter 9: General Commands

General programming commands are the guts of the program. They make things happen. With general commands, objects are added to inventory, picked up and moved, rooms are linked and allowable movements to and from rooms are defined.

In this chapter we will not list specific examples for each of these commands. Specific examples can be in the Command Reference in Chapter 11 under each command listing. We will give a brief description of each command in turn.

The General Commands are:

- CALL
- DIRECTIONS
- DOS
- DROP
- GO
- GRAB
- LINK
- LOAD
- MOVE
- MOVEOBJ
- PAUSE
- PLACEOBJ
- QUIT
- SET
- SPEECH
- STOP
- T
- UNLOAD
- UNSET

CALL

The CALL command allows Visionary to execute sub-programs or subroutines within the program. Sub-programs can be in the subroutine file, an object's code block or a room's code block.

```
CALL {subroutine|objectroom}
```

DIRECTIONS

This command dictates the allowable exit directions from a room.

DOS

The DOS command allows Visionary to execute programs external to the main program. Anything typed in the DOS command expression is handled just as if you were entering it as a command line in the CLI or Shell. Note the usage of quotation marks.

```
DOS ({stringvar}|"doscommand")
```

DROP

The DROP command removes an object from the player's inventory.

```
DROP {object|expression}
```

GO

After the Player's current turn, the player's location is changed to the given room or room number.

```
GO {room|expression}
```

The {expression} can be evaluated to give a result that is then used to point to a room by its number.

GRAB

GRAB adds an object to the player's inventory.

```
GRAB {object|expression}
```

LINK

LINK causes the first room listed to be linked to the second room by the specified direction. If THISROOM is used, the current room will be part of the link command.

```
LINK {roomname|expression},
    {N|S|E|W|NE|NW|SE|SW|U|D|expression},
    {roomname|expression}
```

Rooms can be specified by room number, supplied as the value of an expression.

LOAD

LOAD enables the program to load screens, sounds, or fonts into one of the 25 screen buffers, 25 sound buffers, or 10 font buffers. The type of load to be performed must be specified. Load type defines the file type that can

be accepted by the LOAD command. For example, LOAD SOUND expects an IFF-8SVX file rather than an IFF-ILBM picture.

```
LOAD SCREEN {expression for buffer#},
    (${stringvar}|"filename")
LOAD SOUND {expression for buffer#},
    (${stringvar}|"filename")
LOAD FONT {expression for buffer#},
    (${stringvar}|"filename"), {height}
LOAD SONG {expression for buffer#},
    (${stringvar}|"filename")
```

MOVE

MOVE allows program control of the player's next move. By specifying one of the direction abbreviations or the number of a direction, the player will be moved in that direction after the current turn is over. Another turn will be taken to process the player's new location.

```
MOVE ({N|S|E|W|NE|NW|SE|SW|U|D|expression})
```

MOVEOBJ

With this command, you can let the player move objects to other rooms. If it is not a legal move (the current room does not have an exit in that direction) a run-time error will be displayed. The object's new position will be reflected by its {object}POS variable, which will be the new room number.

```
MOVEOBJ {object|expression}
```

PAUSE

The PAUSE command will delay the execution of the next program statement by the number of 1/50's of a second specified.

```
PAUSE {expression}
```

PLACEOBJ

With this command you can direct the placement of an object in a particular room. If the expression THISROOM is used, the object will appear in the current room.

```
PLACEOBJ {object|expression},
    {room|THISROOM|expression}
```

QUIT

Use of the QUIT command causes the game to end after the current turn.

```
QUIT [GAME]
```

If the word GAME is omitted, the QUIT command also ends the whole session and quits the program.

SET

With this command, a given attribute for a room or object is set to "Y" or "1". For rooms, there are two system-declared attributes, DARK and VISITED. The SET and UNSET commands do not allow a numeric expression to be used for the room or object number.

```
SET {room|object}, {attribute}
```

SPEECH

The speech command dictates whether entered text will be output by the Amiga's voice via the narrator device. The default option is OFF.

```
SPEECH (ON|OFF)
```

T

The T command prints the text string which follows the T command to the screen.

```
T {text string}
```

UNLOAD

The UNLOAD command removes an audio file, image file or font file from a specific buffer number. As with LOAD, you must specify which type of file is being removed as well as the buffer number.

```
UNLOAD SCREEN {buffer#}
UNLOAD SOUND {buffer#}
UNLOAD FONT {buffer#}
UNLOAD SONG {buffer#}
```

UNSET

UNSET gives a room or object attribute the value of "N" or "0". As with SET, the object or room number must be specifically defined, and cannot be given by evaluating an expression.

```
UNSET {room|object}, {attribute}
```

Chapter 10: Advanced Topics

Advanced topics are exactly as they are described, advanced. The coding tricks presented here are not for the beginner. When you are starting your first game, you should concentrate on getting the basic down, making your Visionary game do what you want. Then, as you get more accustomed to the abilities and strengths of the Visionary language, you can add some of these techniques to your repertoire.

Optimizing Visionary Code

There are several steps that you can take to ensure that your Visionary program is as fast as it can possibly be. Often, there are many was to get something done in Visionary. Only some are better than others.

Statement Concatenation

Consider the following code fragments, and assume that everything else has been taken care of.

```
IF PLAYER HAS Magic_Sword THEN
 COLOR mybuf, 9
ELSE
 COLOR mybuf, 0
ENDIF
```

These lines of code could be completely replaced with the following faster, smaller, easier-to-write statement:

```
COLOR mybuf, 9 * (PLAYER HAS Magic_Sword)
```

Do you see why this code has the same effect? Visionary comparisons, like the "PLAYER HAS..." operation, all evaluate to either 1 or 0. That is, the player **either** has or doesn't have the Magic_Sword.

The standard in programming is that "true" expressions evaluate to 1, while "false" expressions evaluate to 0. So the "PLAYER HAS Magic_Sword" expression has a value of 0 if the player doesn't have the sword, and 1 if the sword is currently in the player's inventory. When the expression has a value of 1, 9 times that expression is 9. When it equals 0, 9 times the expression is 0.

The shorter statement allows us to set COLOR to either 9 or 0 in a single statement, depending on whether or not the player has the sword. If you are crafty enough, far more can be done with this concept.

Assignment Expressions

Assignment operators also return a value, though they do so under slightly different conventions. Instead of just 0 or 1, an assignment operator will return whatever value was stored in the integer variable on the left. For instance, if you were to say

```
count := 5
```

the whole expression "count := 5" would return the value 5. This can be quite useful. For instance:

```
i := j := k := l := 10;
```

would summarily set the values of all four integer variables to 10.

A Lot More

Optimization of code is a practice and a puzzle in itself. There are many more clever ways to optimize Visionary programs through crafty programming, but these are the basics.

When you are just starting to learn the language, don't be too concerned with optimization—concentrate on making your program do what you want. As you come to be a stronger Visionary programmer, you will start to see ways to accomplish the same program tasks more efficiently.

Arrays in Visionary

An **array** is a contiguous list of variables under the same name. Although Visionary is not designed to support such a feature, storage arrays can be implemented. The technique used in Visionary relies on the fact that you can create, read to, and write from screen buffers.

Storage Array Implementation

Any given pixel location within a screen can hold a value from 0 to 63. Several pixels may be necessary to store your data, depending on the range of values you want to store. If you only want to store 0 and 1 (binary) values, 1-bitplane screen will be adequate. To store data with a maximum value of 12, on the other hand, you would need a 4-bitplane screen, whose maximum value is 15.

The following examples demonstrate two different ways to create a storage array in Visionary, one using a simple one pixel/one datum arrangement, and a second example showing how larger data values can be stored using more than one pixel for each value.

One Pixel, One Value

Suppose we want to store 10,000 values that range from 0 to 9. A 2-bitplane screen will only allow a maximum value of 7 to be stored, so we will need a screen 4 planes deep. Each **index** or value location in the array will be composed of 1 pixel.

A low-resolution NTSC screen is 320 pixels across, and 200 pixels high. Since that screen provides 64,000 pixels, we know this lores screen will be sufficient. Each row will hold 320 values, so we will need 32 lines to hold all 10,000 values.

The code details are as follows:

```
; SETUP:
    CREATE SCREEN {buf}, 320, 200, 4, LORES
; WRITE: (given: VALUE to store at INDEX
         position in the array)
    row := index / 320                ;get row
         address
    col := index MOD 320              ;get
         column address
    COLOR {buf}, value                ;store
         value
    LINE {buf}, col,row,col,row
; READ:  (given INDEX position in array,
         find VALUE at position)
    row := index / 320                ;get row
         address
    col := (index MOD 320) * 2        ;get
         column address
    PIXEL {buf},col,row,value         ;get value
; SHUTDOWN:
    UNLOAD SCREEN {buf}
```

Arrays with Values > 64

Next, suppose we want to store 10,000 values with values between 0 and 999. A 6-bitplane screen store a maximum value of 64 in a single pixel. So each index in the array will be stored in 2 pixels. Since 64x64 gives us a maximum value of 4,096 for our 2-pixel-per-datum array, we'll have plenty of room to store a value of 999.

As before, we can use a 320x200 lores screen. Each row will hold 160 values, so we will need 63 lines to store all 10,000 values.

The code details are as follows:

```
; SETUP:
    CREATE SCREEN {buf}, 320, 200, 6, LORES
; WRITE: (given: VALUE to store at INDEX
         position in the array)
```

```
        row := index / 160                ;get row
               address
        col := (index MOD 160) * 2        ;get
               column address
        COLOR {buf}, value / 64           ;store
               high 6 bits of value
        LINE {buf}, col,row,col,row
        COLOR {buf}, value MOD 64         ;store
               low 6 bits of value
        LINE {buf}, col+1,row,col+1,row
; READ:  (given: INDEX position in array.
               output: VALUE at pos)
        row := index / 160                ;get row
               address
        col := (index MOD 160) * 2        ;get
               column address
        PIXEL {buf},col,row,temp          ;get high
               6 bits
        PIXEL {buf},col+1,row,value       ;get low
               6 bits
        value := value + 64 * temp       ;calculate
               final value
; SHUTDOWN:
        UNLOAD SCREEN {buf}
```

READ and WRITE can be set up as subroutines so they can be accessed from any point in your Visionary program.

In the multi-pixel array situation, in order to store values greater than 63 we can "chain" pixels together so that collectively, they represent a value. Since Visionary's allowed range of integers values is -32768 to 32767, it will never be necessary to use more than 3 pixels for a single value. Such a large value as the maximum allowed in Visionary could be represented as

```
PIX(col,row)*4096 + PIX(col+1) * 64 +
   PIX(col+2).
```

Storing Negative Numbers

Negative number storage complicates the issue a bit. One way would be to say that the high-order pixel can represent a value of up to 32, with one bit free for the sign. So for a READ, the code would now look like this:

```
PIXEL {buf},col,row,temp
IF temp > 32 then
 temp := 32 - temp
ENDIF
```

The rest of the pixels, if any, are read as normal. WRITE for negative numbers could be handled like this:

```
temp := value ; (/ 32)? ; add bit selection
    if necessary for > 1 pixel!
IF value < 0 THEN
```

```
    temp := 32-temp
ENDIF
COLOR {buf}, temp
```

Resetting a Range within an Array

By drawing a rectangle with the RECT function, it is possible to initialize or reset the values of a **whole range** within the array to a certain value all at once. If more than 1 pixel is used to represent a value, we can use the LINE function to set all of the values at once. This requires that pixels of the same order are arranged in columns, so drawing a vertical line can set all of the values in that column at the same time.

From here you can consider setting up more complex structures. Visionary allows you to simulate 2 or more dimensional arrays, even complex data structures. It's up to you.

Handling Multiple Screens

If you seriously intend to use graphics in your Visionary program, you will likely end up using multiple graphics screen, whether you use them for different displays, scratch or work buffers, or a combination of purposes.

Screen Transitions

Transitions between two static screens can be accomplished in several ways. If the screens you are moving between are of the same display resolution and depth, and have the same color palette – not necessary, but helpful – you might want to use a **dissolve**. The DISSOLVE command is covered in the basic command discussions in earlier chapters.

For screens that are of different resolutions or use different color palettes, you need a different way to transition. You could use what in video is called a "jump cut", an instantaneous switch to the new display.

You might also use a color fade out/fade in combination. This might be accomplished with the following sequence, switching from screen buffer 0 to screen buffer 1:

```
SHOW SCREEN 0              ; displays screen 0
SCREENMODE GRAPHICS     ; moves the text
   screen out of the way
   ...
FADETO 0, 15,8,0, 5     ; visibly fades
   screen 0 to orange slowly
FADETO 1, 15,8,0, 0     ; fades hidden screen
   1 to orange quickly
SHOW SCREEN 1              ; swap in the
   completely-orange screen 1
```

```
FADEFROM 1, 15,8,0, 5 ; visibly fades screen
    1 from orange slowly
FADEFROM 0, 15,8,0, 0 ; restores the palette
    from the original screen
```

Another transition technique would be a flicker fade. This rapidly alternates the display between the source screen and the destination screen. Example switching from screen 5 to screen 8:

```
source   := 5
dest     := 8

i        := 1
current  := source
swapper  := dest - source
count    := 31                    ; do 31 swaps—this
    number MUST be odd!
WHILE i  count DO
SHOW SCREEN current
current := current + swapper
swapper := 0 - swapper
; you can insert a PAUSE here if you want
    it slower
ENDWHILE
```

Multiple Images Per Screen

When you are dealing with graphics, you might have a need for a set of smaller images that get swapped in and out of the display screen. You **could** perform an IFF load of each screen as the need arises, or you could try putting a number of such templates in a single work screen.

This work screen could be loaded once and never displayed, and it would contain some set of smaller graphic blocks, like scenes or objects, or perhaps people or frames of animation. This work screen buffer can be of immense proportions—just watch your Chip memory consumption. Large screens can be designed in programs such as Aegis SpectraColor or any other paint program that produces IFF images.

When you need to access one of these blocks, use the COPY command to place the desired portion of the work buffer into any buffer you want. This technique greatly reduces disk-access time, and greatly increases the player's enjoyment of the game. You could extend the technique, loading in a large buffer of templates when the player enters a new section of the game, so that memory demands can be kept to a minimum.

» As you design a work screen, be certain to write down the X,Y coordinates of the corners of the blocks so that you can COPY to the correct rectangle later. If you have an image, you can load that image into the VCOORD utility and find the coordinates.

Animation

Animation covers not only the use of ANIM-format animations generated by paint programs and rendering software, but **algorithmic animation** as well. This complex-sounding term simply means the displaying of a sequence of images to cause the illusion of motion. And, of course, as a Visionary programmer, you also have to ability to make animated graphics.

For completely non-interactive, "canned" animation, you can use the DOS command to call an ANIM player, which will play any standard IFF animation. When you need interactive animation, you must explicitly program it to happen. To adequately generate the illusion of motion, the "frame rate", the number of images rendered per second, should be in the range of 8 to 20.

First, you need to decide exactly what will happen in the animation. For our examples, we will animate a blimp flying across the screen. You will find the IFF image of the blimp and all the following examples in the Animation_Demo drawer on the Visionary disk.

Blank Background

Is the background where the animation is to happen blank, or does it contain imagery? If the background is blank, the animation job is much easier. We need only to draw the blimp at position {objectX}, wait for some brief time period of time, erase the blimp's image, add some value to {objectX}, and repeat this process until the blimp image has reached its destination. If the blimp resides at (0,0) in buffer 7, and we render to buffer 0, a LORES screen, at Y position 30, we can do this as follows:

```
;————————— simple blit with simple
   background
MODE 0, DRAW
COLOR 0, 0
WHILE 1-LeftButton DO
 ObjectX := 0
 WHILE ObjectX < 319 AND 1-LeftButton DO
  COPY 7, 0,0, 49,24, 0, ObjectX,30    ;
    draw blimp
  PAUSE 5
```

```
    RECT 0, ObjectX,30, ObjectX+20,54    ;
      erase blimp
     ObjectX := ObjectX + 10
   ENDWHILE
  ENDWHILE
```

This process can be made even faster, By using a self-erasing blit. This eliminates the need to draw a rectangle behind the blimp to erase it. The new program would be:

```
;————————— self-erasing blit with simple
   background

MODE 0, DRAW
WHILE 1-LeftButton DO
 ObjectX := 0
 WHILE ObjectX < 319 AND 1-LeftButton DO
  COPY 7, 0,26, 69,48, 0, ObjectX,30   ;draw
   blimp
  PAUSE 5
  ObjectX := ObjectX + 10
 ENDWHILE
ENDWHILE
```

The PAUSE 5 operation in both of these programs is used so that the blimp's image is displayed longer than it is blank, so that flickering is reduced. Unfortunately, it also "chops up" the animation. For an explanation of how to eliminate this problem, see **double buffering**, below.

Blitter on Complex Background

What if the background is **not** blank? In this case, it is necessary to preserve the background so that trails are not left behind the moving object. In Visionary, this is handled with a COPY in OVERLAY mode.

An example follows, using the same blimp flying above, except that now it is flying in front of "scenery". The same buffer, 7, is also used now as a temporary holding area for the image behind the blimp. Since we are rendering in OVERLAY mode, we need a MASK buffer, for which we will use buffer 6.

```
;————————overlay blit with complex
   background

CREATE SCREEN 6, 60,50,1,LORES NOLACE
MASK 6
MODE 0,DRAW
COPY 7, 0,152, 319,195, 0,0,18          ;
   copy in the background
WHILE 1-LeftButton DO1
 ObjectX := 0
 WHILE ObjectX < 319 AND 1-LeftButton DO
```

```
    COPY 0, objectX,30, objectX+49,54, 7, 94,
      52 ;save background
    MODE 0, OVERLAY
    COPY 7, 0,0, 49,24, 0, objectX,30      ;
      draw blimp
    PAUSE 5
    MODE 0, DRAW
    COPY 7, 94,52, 143,76, 0, objectX,30 ;
      restore background
    objectX := objectX + 10
   ENDWHILE
  ENDWHILE
```

OVERLAY copy mode is somewhat slower than DRAW mode, but it is necessary in this case to get the proper merging of images.

Double Buffering

There is something fundamentally wrong with the sample program segments above. The problem lies in the fact that they do all of their rendering directly in the buffer being displayed, which can generate a great deal of flicker, making the animation unclean. You can get away with this in some situations, but how do we do flicker-free animation?

The technique is called **Double Buffering**—we render to a hidden buffer while displaying a finished one. We then swap the freshly-rendered buffer into the display and modify the one we swapped out. The keeps the display crisp and clean, and eliminates the need for lengthy PAUSE commands that destroy our frame rates.

Double buffering does have its costs, however. In order to use it, we need to maintain two separate sets of SAVE buffers if we animate in front of imagery, and two separate SAVE positions.

The following example shows our flying blimp again, in front of a blank screen, modified to make use of double buffering. The two display buffers we will be using are buffers number 0 and 1, and the blimp still comes from (0,0) in buffer 7.

```
;————————————— double buffering

MODE 0, DRAW
COLOR 0,0
CREATE SCREEN 1,320,100,1, LORES NOLACE
MODE 1, DRAW
COLOR 1,1
TEXT 1, 0,10,"Click the left mouse button to
   exit!"
COLOR 1,0
WHILE 1-LeftButton DO
 oldx1 := 0;
 oldx2 := 0;
```

```
ObjectX := 0
Step := 2
WHILE ObjectX < 319 AND 1-LeftButton DO

 SHOW SCREEN 1              ; show buffer 1
 RECT 0, oldx1,30, oldx1+50,54        ; erase
  from hidden buffer 0

 ObjectX := ObjectX + Step
 COPY 7, 0,0, 49,24, 0, ObjectX,30    ; draw
  to hidden buffer 0
 oldx1 := ObjectX;

 SHOW SCREEN 0              ; show buffer 0
 RECT 1, oldx2,30, oldx2+50,54        ; erase
                            from hidden buffer 1

 ObjectX := ObjectX + Step
 COPY 7, 0,0, 49,24, 1, ObjectX,30    ; draw
                            to hidden buffer 1
 oldx2 := ObjectX;
ENDWHILE
RECT 0, oldx1,30, oldx1+50,54      ;
                            erase from hidden
                            buffer 0
RECT 1, oldx2,30, oldx2+50,54        ;
                            erase from hidden
                            buffer 1
ENDWHILE
```

This example is highly customized for this specific situation. When generating your own animation sequences, you will need to take care to have it all right. If you do not, you will see flickering remnants of the old images. The double-buffering technique is used on the Catacomb demo game included with Visionary. Examine the source code included on that disk to get another example of double-buffering at work.

Image Cycling

Let's really get fancy—suppose we wanted to have our blimp flash the word "Visionary" as it flew across the screen. This is simply a matter of changing which buffer-7 image we render to the display screens. The following is a **non-double-buffered** example of how that might be displayed on a screen with no background:

```
;————————————————— image cycling

MODE 0, DRAW
COLOR 0,0
WHILE 1-LeftButton DO
 ObjectX := 0
 Image := 0
```

```
 WHILE ObjectX < 319 AND 1-LeftButton DO
  COPY 7, 0,50+Image*23, 49,72+(Image*23),
    0, ObjectX,30
; draw blimp
  Image := (Image + 1) MOD 4 ; set image to
    next in sequence
  PAUSE 5
  RECT 0, ObjectX,30, ObjectX+10,54 ; erase
    back end of blimp
  ObjectX := ObjectX + 5
 ENDWHILE
ENDWHILE
```

It is left to you as an exercise to modify this routine to work with double-buffering on a screen with a background.

These advanced graphic techniques can be combined to produced animation of anything you can conceive, and possibly some you can't imagine right now!

Incorporating Player Input in a Game

One way to make your game program appear more "friendly" is to ask the player's name, and then use it in the course of the game, so the program appears to be addressing the player directly.

This ability to pass input from the player into a variable in the game, rather than simply having the game action respond to it, can be used in a variety of ways. For example, suppose you have a game where the wording of some responses during play depends on the player's gender. Perhaps you want the program to say "Most women wouldn't want to!" if the female player tries to eat a moose, but "Most men wouldn't want to!" if the player is male.

The following example shows how to ask the player's name, gender, and age, storing these values in variables that can be used later in your program.

```
;-------------------
SUBROUTINE AskStats
; following asks player's name, puts it into
    string variable $PName
T Brave Warrior, enter your name:
GETSTRING $PName

; following asks player's gender, puts it
    into integer
variable Gender
; Gender = 0 means Male, Gender = 1 means
    female
Done = 0
WHILE Done = 0 DO
 T Are you Male, or Female? (M/F)
```

```
 GETCHAR $Temp ; get the character
 UPCASE $Temp  ; makes sure it's a capital
   letter
 COMPARE $Temp, "M", tempint
 COMPARE $Temp, "F", Gender
 IF tempint = 1 OR Gender = 1 THEN
  Done = 1
 ELSE
  T I rather doubt your gender is @$Temp .
   Try again.
 ENDIF
ENDWHILE

; following asks player's age, puts it into
   the integer variable PAge:
PAge = 0
WHILE PAge > 0 AND PAge < 112 DO
 T Brave Warrior, enter your age:
 GETNUM PAge
 IF Page < 1 OR PAge > 111 DO
  T That just doesn't seem right.  Try again.
 ENDIF
ENDWHILE

ENDSUBROUTINE
;-----------------------
```

Keep in mind that these examples are just one way to get player input—one rather clever adventure game started by presenting the choice of two doors to go through—into the gents' or the ladies' toilet. The program proceeded after that point on the assumption that your gender had been revealed by the restroom you chose!

Chapter 11: Command Reference

AND

SYNTAX:

IF {condition} AND {condition} AND {condition} THEN

COMMENTS:

The AND command is used when multiple conditions must be met.

EXAMPLES:

```
IF (Chamber is DARK) AND (Player HAS Lamp)
   AND (Player HAS Matches) THEN
 T Light Lamp
ENDIF
```

CALL

SYNTAX:

CALL {subroutine|object|room}

COMMENTS:

CALL allows the Visionary program to execute sub-programs. The name of the program to call can be a subroutine, object's CODE block, or a room's CODE block.

EXAMPLES:

```
CALL CheckIfDead
CALL Ray_Gun
CALL Hall_of_Magic
```

CLICK

SYNTAX:

CLICK {button#}, {x1},{y1},{x2},{y2}, {subroutine|object|room}

COMMENTS:

CLICK allows a new form of interaction with a Visionary program; it sets up a region of the graphic screen to be sensitive to mouse clicks so that when the player clicks in the given rectangular region, a subroutine, object CODE block, or room CODE block is executed.

A button number must be specified, between 0 and 49, inclusive. A coordinate of (x1,y1) represents the upper-left corner of the box, and (x2,y2) represents the lower-right corner.

CLICK regions can be layered and prioritized. For instance, if you want the whole screen to be sensitive to a click, but also want smaller buttons elsewhere on the screen, you can declare the smaller buttons first, with lower button numbers, and then the whole screen as a higher button number. Since the list of regions is searched from 0 to 49 for matching regions, the smaller buttons will be checked first. If these tests fail—for example, the player wasn't clicking in the defined regions—then it will get to the full screen button and execute its program.

> **! Each time the CLICK command is issued, any previous user-clicks in the graphics screen are discarded and ignored.**

EXAMPLES:

```
CLICK 0,    5, 10, 100,150, ActionHit
CLICK 1,   52, 82, 238,182, Magic_Sword
CLICK 2, 285,123, 432,192, Library
```

COLOR

SYNTAX:

COLOR {buffer#}, {color}

COMMENTS:

COLOR sets the current drawing color in the given buffer to the given color. The valid buffer range is 0 to 49, and the valid color range depends on the number of colors in the image, starting at 0 and ending at 31 at the MOST.

EXAMPLES:

```
COLOR 0, 7
COLOR i, linecolor
```

CONTINUE

SYNTAX:

CONTINUE SONG

COMMENTS:

This command causes the last MED song to continue playing where it left off if the song was stopped previously. (See STOP SONG.) This command will work ONLY in ENABLEMUSIC mode.

EXAMPLES:

```
CONTINUE SONG
```

COMPARE

SYNTAX:

COMPARE (${stringvar}|"text"), (${stringvar}|"text"), {int var}

COMMENTS:

This command COMPAREs the text string or string variable on the left with that on the right. If the first is less than the second, the integer variable assumes the value of -1. If the first is larger than the second, the integer variable is set to 1. If the strings are equal, the variable is set to 0.

EXAMPLES:

```
COMPARE $MyName, $YourName, result
COMPARE "Bosco Brainly", $YourName, result
COMPARE $YourName, "Bosco Brainly", result
```

COPY

SYNTAX:

COPY {src buf#}, {x1},{y1}, {x2},{y2}, {dest buf#}, {x},{y}

COMMENTS:

COPY allows the Visionary programmer to move rectangular blocks of graphic images around within a buffer, or from buffer to buffer. Given the buffer number that will be the source of the block, the rectangular coordinates (x1,y1) = upper left, (x2,y2) = lower right, and the destination buffer number plus the offset (x,y) into that buffer, COPY will cut out the block of size (x2-x1+1) wide and (y2-y1+1) tall, clip it if necessary to fit into the destination buffer, and then perform the move. Here the drawing mode becomes very important. If it is XOR for the destination buffer, the

block will be **exclusively-ORed** with the destination on a plane-by-plane basis. If the drawing mode is DRAW, the block will be copied verbatim.

If, however, the mode is OVERLAY and a MASK command has been performed, the source rectangle is filtered so that color 0 is transparent, so that when merged with the destination buffer, whatever image was in the destination location will show through color 0 of the copy block. This is especially useful for, say, drawing objects into a room. If the destination buffer has fewer bitplanes, only that number will be transferred. This can result in weird-looking images, but sometimes such an operation may be wanted. Please see MASK.

EXAMPLES:

```
COPY 0, 100,50, 125,100, 0, 200,150
would copy a region 26 by 51 to the same
    buffer, 0.
COPY 3, 0,0, 100,100, 3, 0,0
would copy a region 101 by 101 from buffer
    0 to buffer 3.
```

CREATE SCREEN

SYNTAX:

CREATE SCREEN {buf#}, {x}, {y}, {d}, (HIRES|LORES|LACE|NOLACE|HAM|HALFBRITE)

COMMENTS:

CREATE SCREEN lets the programmer create a graphic screen without having to load an IFF image. Give it the buffer number to put the empty screen into, x,y represent the dimensions of the screen, choices for X are 320 or 640, choices for Y are 200 or 400 for NTSC, 256 or 512 for PAL. D represents the depth of the image, 1 to 6. 6 only applies for HALFBRITE and HAM, and 4 is the maximum for a HIRES screen. You may use ANY combination of the mode words, but if contradicting ones, such as LORES and HIRES, are used together, only the last one will be used. The screen defaults to LORES NOLACE.

If there is not enough memory available, the Error variable will be set to a NON-ZERO value.

NOTE: The X and Y sizes of screens are only suggestions; you may use a screen of any dimensions you like.

EXAMPLES:

```
CREATE SCREEN 0, 320,200, 6, HAM
CREATE SCREEN 1, 640,400, 3, HIRES LACE
CREATE SCREEN i+2, 34, 119, 6, LORES NOLACE
    HALFBRITE
```

CYCLE

SYNTAX:

CYCLE (ON|OFF|ONCE|RESTORE)

COMMENTS:

This command determines the state of color cycling as defined by the paint program in which the image was originally rendered. If ON is used, up to 8 active color cycling ranges, if there are any, will cycle at their intended speed. If OFF is used, any current color cycling will cease, and the colors will stay where they are. If ONCE is used, the cycling clock will "tick" once. This means that if the color ranges have the maximum speed, the colors will move once, but if the range is slower, the tick will count toward making the next color switch. If RESTORE is used, the palette will return to the default colors, useful after finished cycling. The cycling takes place only for the currently displayed buffer, if any.

EXAMPLES:

```
CYCLE ON
CYCLE OFF
CYCLE ONCE
CYCLE RESTORE
```

DIRECTIONS

SYNTAX:

DIRECTIONS {room|expression}, (N|S|E|W|NE|NW|SE|SW|U|D|{expression})

COMMENTS:

This command causes the exits from the given room or room number to be **only** those listed after. If no direction abbreviations are given or the expression evaluates to 0, the room will have no exits. Any number of exits may be specified with this command, and in any order. If a numeric expression is used for the directions list, the bear in mind that N represents bit 0 and D represents bit 9, so the range of values is 0 through 1023.

To include a direction, include the value in the bit set.

EXAMPLES:

```
DIRECTIONS Hall_Of_Magic, N U D SW
DIRECTIONS Gun_Turret, D
DIRECTIONS Locked_Cell
DIRECTIONS GunTurret, 89
DIRECTIONS 12, N U D SW
DIRECTIONS i, j
```

DISABLEMUSIC

SYNTAX:

DISABLEMUSIC

COMMENTS:

Turns off access to the MED Music system, and reactivates the sampled sound system. This command and ENABLEMUSIC are necessary because the MED Music system and the sampled sound system compete for resources, namely hardware audio channels. Therefore, it becomes impossible to allow both to be active at once.

At system startup, MED Music is OFF and sampled sounds are ON—this assumes Visionary could access the audio channels and no other process is using them.

EXAMPLES:

```
DISABLEMUSIC
```

DISSOLVE

SYNTAX:

DISSOLVE {source buf}, {dest buf}, {add}

COMMENTS:

DISSOLVE does a byte or pseudo-pixel dissolve between two buffers. The source buffer is the one from which data is to be copied, destination buffer is of course the one to which the data are to be copied, and "add" is the dissolve type; it **must** assume an odd value that is **not** a multiple of 5. There is a good reason for this—just follow the rules and be happy. Also, the two screens **must** be of the same dimensions and view modes. It would also help for their palettes to be the same, but this is not a strict requirement.

The add value represents the value that is added to the screen pointer at each iteration. The whole screen just won't be reached if it is even or a multiple of 5. The reason is that most screens are of even length. A lores screen is 8000 bytes (base size), a 640 x 200 screen is 16000 bytes, and a hires, interlace screen is 32000 bytes.

EXAMPLES:

```
DISSOLVE 0, 1, 1  ; top-to-bottom wipe from
   buf 0 to 1 in lores
DISSOLVE 0, 1, 7999; bottom-to-top wipe from
   0 to 1 in lores
DISSOLVE 0, 1, 1999 ; a neat effect
DISSOLVE 0, 1, 3  ; an even neater effect
```

DOS

SYNTAX:

DOS (${stringvar}|"doscommand")

COMMENTS:

DOS allows the Visionary program to execute programs external to the main game player. This can be very useful in setting up the environment for the game.

If the DOS call fails, the Error variable will be set to a **non-zero** value.

EXAMPLES:

```
DOS "Assign PICS: mygame:pics"
DOS "dir dh0: opt a"
```

DOWNCASE

SYNTAX:

DOWNCASE ${stringvar}

COMMENTS:

DOWNCASE shifts all alphabetic characters in the string to lower case.

EXAMPLES:

```
DOWNCASE $MyName
```

DROP

SYNTAX:

DROP {object|expression}

COMMENTS:

This causes the listed object to be removed from the player's inventory. After that, its position variable will hold the number of the room in which the object was dropped. A mathematical expression may be substituted to represent an object number.

EXAMPLES:

```
DROP Magic_Sword
DROP (i+1)
```

ELSE

SYNTAX:

ELSE

```
See also IF
```

COMMENTS:

When previous conditions in an IF or ELSIF command fail, then the commands following the ELSE command are executed.

EXAMPLES:

```
IF (Chamber is DARK) THEN
T Turn on light switch.
ELSE
GO Closet
```

ELSIF

SYNTAX:

ELSIF {condition} THEN

```
See also IF
```

COMMENTS:

Like IF, ELSIF can accept any legal Visionary expression as a condition. AND and OR commands can also be used with the condition expressions. The ELSIF statement always ends with THEN.

EXAMPLES:

```
ELSIF (Magic_Sword IN INVENTORY) THEN
T You can slay dragon.
```

ENABLEMUSIC

SYNTAX:

ENABLEMUSIC (MIDI|NOMIDI)

COMMENTS:

ENABLEMUSIC turns on the MED Music system and deactivates the sampled sound system. See DISABLEMUSIC for details about why MED is disabled. The only parameter, MIDI or NOMIDI, selects whether the system should seize access to the serial MIDI device and play instruments through the keyboard(s) or sequencer(s) present. That is, some MED songs have built-in MIDI information, allowing them to play up to 20 channels simultaneously, including the Amiga's 4 internal voices.

It is generally best **not** to use MIDI, unless you are certain that the song you want to play uses it. If it doesn't, nothing will happen outside the Amiga's voices.

NOMIDI is the default; if this parameter is left blank, NOMIDI will be assumed.

EXAMPLES:

```
ENABLEMUSIC
ENABLEMUSIC NOMIDI
ENABLEMUSIC MIDI
```

END

SYNTAX:

END

COMMENTS:

When an END command is used, the program stops executing the current section of code and returns to the main program. END can be used with many other commands including IF and WHILE.

EXAMPLES:

```
IF (chamber is DARK) THEN
 T Strike a Match
ENDIF
```

FADEFROM

SYNTAX:

FADEFROM {buffer}, {r}, {g}, {b}, {delay}

COMMENTS:

FADEFROM is to be used **after** a FADETO. It restores the color palette to the values that existed before a FADETO over 15 iterations and waits the specified amount of time in jiffies between iterations. A delay of 0 will make the whole thing happen almost instantaneously.

One of these **must** be performed for every FADETO if you want to get your original color palette back. See FADETO for more info.

EXAMPLES:

```
FADEFROM mybuf, 0,0,0,20 ;returns the
    palette from black in 5 seconds.
```

FADETO

SYNTAX:

FADETO {buffer}, {r}, {g}, {b}, {delay}

COMMENTS:

FADETO sends all of the colors of the given buffer to the specified R, G, B (red, green, blue) values over 15 iterations. It also waits the specified time in jiffies between iterations.

This should **not** be done more than once before a FADEFROM operation is performed; the system memorizes the color palette at the beginning of this operation as the colors to return the display to during a FADEFROM. If you fade to black twice, you will lose the color palette until you do a CYCLE RESTORE when that screen is active.

If this sounds convoluted, it really isn't. Fades **can** happen in buffers that are not visible – this makes the fade transition possible.

EXAMPLES:

```
FADETO 0, 15, 15, 6, 4   ; sends colors to
    light yellow in 1 second.
```

FONT

SYNTAX:

FONT {screen buffer}, {font number}

COMMENTS:

FONT allows you to set the text font in the specified graphics buffer to be set to the font number specified. The screen buffer must be in the range of 0 to 24, and the font buffer must be in the range of 0 to 9. Font buffer 0 **always** starts out with the default Topaz 8 font, and font buffer 1 **always** starts out with the default Topaz 9 font.

EXAMPLES:

```
FONT myscreen, 2
FONT 0,0
```

GETCHAR

SYNTAX:

GETCHAR ${stringvar}[, WAIT]

COMMENTS:

GETCHAR stops program execution to let the player enter a single keypress. That key is then stored in the given string variable, and program execution resumes. No characters are echoed to the display.

If the word WAIT follows the command, Visionary will pause until the user presses a key. If the word WAIT is **not** present, Visionary will simply test for a key being pressed. If a key was pressed, its value will be passed out as normal, but if a key was **not** pressed, GETCHAR will return a value of 0 , since an empty string has a length equal to 0.

Special keys, like the functions keys F1 through F10, the arrow keys and the HELP key are mapped to special values, as follows:

KEY NAME	ASCII VALUE
Up Arrow	128
Down Arrow	129
Left Arrow	131
Right Arrow	132
HELP	130
F1	133
...	...
F10	142

EXAMPLES:

```
GETCHAR $AChar
GETCHAR $MyInput, WAIT
```

GETPALETTE

SYNTAX:

GETPALETTE {buffer}, {color}, red_var, green_var, blue_var

COMMENTS:

GETPALETTE allows a Visionary program to find out what red, green, and blue color values are currently in a given color or **pen** number in a given buffer. Buffer values range from 0 to 24, the valid color range is 0 to 31, regardless of the number used in the buffer, and the red, green, and blue components of that color, having a value from 0 to 15, will be stored in the given integer variables.

> **!** **You must supply three integer variables in which the Red, Green and Blue color values will be stored.**

EXAMPLES:

```
GETPALETTE 0, 5, r,g,b
GETPALETTE mybuf, colornum, red, green, blue
GETPALETTE i, j, k, l, m
```

GETNUM

SYNTAX:

GETNUM {int var}

COMMENTS:

GETNUM stops program execution to let the player enter an integer number on the command line. The value is then put into the given integer variable, and execution resumes. The number is echoed to the display as it is typed.

EXAMPLES:

```
GETNUM AValue
```

GETSTRING

SYNTAX:

GETSTRING ${stringvar}

COMMENTS:

GETSTRING stops program execution to let the player enter a text string on the command line. The string is then put into the given string variable, and execution resumes. The string is echoed to the display as it is typed.

EXAMPLES:

```
GETSTRING $PlayerName
```

GHOST

SYNTAX:

GHOST (${stringvar}|"text")[, TURN]

COMMENTS:

GHOST causes the text of the given string variable or the given literal string to be interpreted by the command line interface as one of the player's commands. This can be **very** useful.

If the command is not recognized, the Error variable will be set to a **non-zero** value.

If the word TURN follows the command string or variable, then Visionary will immediately execute all relevant programs, including the current room, all objects in the current room, and all NPCs. This allows games to fetch keyboard input in, say, a graphics screen, skipping the normal input facilities, and still be able to "take a turn" like the standard facilities. The "Moves" variable **will not be incremented**. Also, if the command string is null, i.e. its length is 0, GHOST will not do **anything**, whether TURN is specified or not.

EXAMPLES:

```
GHOST $Command
GHOST "QUIT Y"
GHOST "SAVE DH0:games/@$PlayerName "
GHOST "INVENTORY"
GHOST $Command, TURN
```

GO

SYNTAX:

GO {room|expression}

COMMENTS:

GO causes the player's location to change to that of the given room or room number **after** the current turn. Another turn will take place to process the player's new location.

EXAMPLES:

```
GO Hall_Of_Magic
GO Elevator
```

GRAB

SYNTAX:

GRAB {object|expression}

COMMENTS:

This causes the listed object to be placed in the player's inventory. Its POS variable will thereafter hold the value of -1. A mathematical expression may be substituted to represent an object number.

EXAMPLES:

```
GRAB Magic_Sword
GRAB 7
```

IF

SYNTAX:

IF {condition} THEN
...
[ELSIF {condition} THEN]
...
ENDIF

COMMENTS:

The IF statement allows a program to decide which sections of code it will execute. The {condition} can be any legal Visionary expression. Legal expressions are:

{const}	(simple constant)
{var}	(simple variable)
{expression} * {expression}	(multiplication)
{expression} / {expression}	(division)
{expression} MOD {expression}	(modulus)
{expression} + {expression}	(addition)
{expression} - {expression}	(subtraction)
{expression} < {expression}	(less than comparison)
{expression} <= {expression}	(less than or equal comparison)
{expression} = {expression}	(equal to comparison)
{expression} >= {expression}	(greater than or equal comparison)
{expression} > {expression}	(greater than comparison)
{expression} # {expression}	(not equal to comparison)
{object|PLAYER|expression IN THISROOM|expression}	(is object in room comparison)
PLAYER HAS {object|expression}	(player has object (or object #) comparison)

PLAYER CANGO {expression}	(can the player go a direction # comparison)
{PREP|OBJNOUN|object|room} IS {"preposition"|object|attribute}	(attribute comparison)
{PREP|OBJNOUN|object|room} NOT {PREPOSITION|object|attribute}	(inverse attribute comparison)
{expression} AND {expression}	(conjugate two simple expressions)
{expression} OR {expression}	(conjugate two simple expressions)
{int variable} := {expression}	(set equal to, equate)

NOTES:

Some special information about these operators:

- THISROOM, as used with the "IN" operator, implies whatever room the player is currently in.
- PLAYER IN THISROOM will **always** return true, or 1.
- PREP IS|NOT "preposition" will see if the player's last line contained the given preposition, from the list of **valid** prepositions. Note: the preposition **must** appear in quotes!
- OBJNOUN IS|NOT {object}: see if the object noun from the player's last line was the given object.
- PLAYER CANGO {expression}: the expression is expected to be a value between 0 and 9, inclusive, representing N, S, E, W, NE, NW, SE, SW, U, and D, respectively.

Precedence

The default order of precedence for the operators, from highest to lowest by line, is:

PRECEDENCE	OPERATOR(S)
1	*, /
2	MOD
3	+, -
4	<, <=, =, >=, >, #, IN, HAS, IS, NOT, CANGO
5	AND, OR
6	:=

As with standard math formula statements, the order of precedence may be changed with the use of parentheses, "(" and ")". The expression inside the deepest set of nested parentheses will be evaluated first.

To negate a mathematical expression, subtract it from 0. To negate a boolean or logical expression, subtract it from 1.

An IF statement may also be followed by one or more ELSIF clauses, and then may be followed by a single ELSE clause, if needed.

EXAMPLES:

```
IF (health / 2 > 67) AND (PLAYER HAS
  Golden_Sword) THEN
 ...
ELSIF 1-(Chamber IS DARK) OR (PLAYER HAS
  Lamp) OR (PLAYER CANGO 0) THEN
 ...
 ELSIF ((value > -5) AND (value < 16)) OR
  (value = 22) THEN
 ...
 ELSE
 ...
 ENDIF
```

LEFT

SYNTAX:

LEFT (${stringvar}|"text"), {expression}, ${stringvar}

COMMENTS:

LEFT, given the first text string and a count of the number of characters desired, moves that number of characters, up to the maximum of 79, into the given string variable, the last parameter.

EXAMPLES:

```
LEFT $MyName, 5, $FirstName
LEFT "Mary had a little lamb", 10, $AString
```

LENGTH

SYNTAX:

LENGTH (${stringvar}|"text"), {intvar}

COMMENTS:

LENGTH puts the length, in characters, of the given text string into the given string variable.

EXAMPLES:

```
LENGTH $MyName, len
LENGTH "Warrior", count
```

LET

SYNTAX:

LET {variable} := {expression}

LET ${stringvar} := (${stringvar}|"text")

COMMENTS:

LET allows the program to set a variable to the value of the expression following it, or to set the contents of a string variable. As in BASIC syntax, the word "LET" is optional. The same expression syntax as the IF statement is used. In text strings, if a literal string is used, any of the standard in-line variables or codes may be used.

EXAMPLES:

```
LET count := count / ((67 * value) MOD 16)
LET count := result + (PLAYER IN Gun_Turret)
    * intelligence
     count := count + 10;
LET $MyName := "Kevin Kelm"
     $Title       := "Hacker with a Plan"
     $PlayerName := "Bosco Brainly is the
    @$Title"
```

LINE

SYNTAX:

LINE {buffer#}, {x1}, {y1}, {x2}, {y2}

COMMENTS:

Line draws a line in the given buffer, between the 2D points (x1,y1) and (x2,y2) in the current color for that buffer.

EXAMPLES:

```
LINE 0, 0,0, 319,199
LINE i, x, y, x2, y2
```

LINK

SYNTAX:

(all on the same line)
LINK {room|THISROOM|expression}, ((N|S|E|W|NE|NW|SE|SW|U|D)|{expression}), {room|THISROOM|expression}

COMMENTS:

LINK causes the first room listed to be linked in the specified direction to the second room listed. Room links survive whether the specified direction is enabled or not, and remains linked to the same room until deliberately changed. At game startup, if a direction was not included in the DEFAULT block, the direction will be linked to no room.

EXAMPLES:

```
LINK Hall_Of_Magic, D, Pit_of_Terrors
LINK Gun_Turret, D, Ammunition_Chamber
LINK 17, 5, 12
```

LOAD

SYNTAX:

LOAD SCREEN {expression}, (${stringvar}|"filename")
LOAD SOUND {expression}, (${stringvar}|"filename")
LOAD FONT {expression}, (${stringvar}|"filename"), {height}
LOAD SONG {expression}, (${stringvar}|"filename")

COMMENTS:

LOAD allows the program to load screens, sounds and fonts into one of the 25 screen buffers, 25 sound buffers and 10 font buffers in the computer's memory, assuming there is enough free memory to do so. Buffer numbering starts at 0.

LOAD SCREEN expects the file to be an IFF-ILBM picture of any resolution up to 640x400 and any legal Amiga display mode.

LOAD SOUND expects the file to be an IFF-8SVX sampled sound file.

LOAD FONT expects the file to be found in the current fonts: directory. You **must** include the ".font" suffix in the filename. Only disk-based fonts—those that appear in the FONTS: directory—may be used.

LOAD SONG expects the file to be an MMD0-MED Music file, in which the first 4 bytes in file are "MMD0", and it expects a value from 0 through 9 to be the buffer number.

If there is any kind of error loading the entity, the Error variable will be set to a **non-zero** value.

EXAMPLES:

```
LOAD SCREEN 0, "PICS:Scene1"
LOAD SOUND  j+2, $EffectName
LOAD FONT   3, "Emerald.font", 20
LOAD SONG   9, "GameSong"
```

MASK

SYNTAX:

MASK {buffer#}

COMMENTS:

MASK lets the programmer specify which graphic buffer is to be used by the system as a work area for generating OVERLAY COPY operations. The buffer specified **might** not exist at the time this command is issued, but it had better exist by the time the COPY command is issued. The MASK buffer **must** be at least as deep as the destination buffer, and it **must** be at least as large (x,y) as the block that is being moved.

EXAMPLES:

```
MASK 0
MASK MyMaskBuf
```

MENUS

SYNTAX:

MENUS (ON|OFF)

COMMENTS:

This command lets the program control whether or not the player is allowed to deal with menu selections. MENUS ON is the default, but if MENUS OFF is used, no menus will appear in the text screen's menu bar. This also disables [Amiga]-Q and other keyboard combinations.

> **!** **Like SCROLLBAR OFF, MENUS OFF should not be inserted until the program is debugged and ready to go. If the menus are turned off and the keyboard command "QUIT" is overridden, there will be no way to exit the program without a reboot unless the program makes deliberate allowance for it.**

EXAMPLES:

```
MENUS OFF
MENUS ON
```

MID

SYNTAX:

MID (${stringvar}|"text"), {start expr}, {number expr}, ${stringvar}

COMMENTS:

MID, given the source text string on the left, the starting position (in characters) and the number to move, sends that portion of the string the to destination string variable on the right.

EXAMPLES:

```
MID $MyName, 5,5, $AString
MID "Mary had a little lamb", 11, 6, $Word
    ; word then = "little"
```

MODE

SYNTAX:

MODE {buffer#}, (XOR|DRAW|OVERLAY)

COMMENTS:

MODE allows the programmer to specify what drawing mode to use for rendering in the given buffer. XOR will cause an Exclusive-OR operation with the following logical function:

XOR	source 0	value 1
dest 0	0	1
value 1	1	0

on a bit-by-bit, plane-by-plane basis. DRAW will cause the current color to be put into effect, and the drawn image will be solid. COPY commands will also be solid. OVERLAY acts just like DRAW in all operations except COPY, which then uses the MASK buffer to generate a "Blit-Mask", so that wherever there is color 0 in the source image, the destination image will show through. Please also see MASK.

EXAMPLES:

```
MODE 0, XOR
MODE 7, DRAW
MODE i, OVERLAY
```

MOVE

SYNTAX:

MOVE ((N|S|E|W|NE|NW|SE|SW|U|D)|{expression})

COMMENTS:

MOVE allows program control of the player's next move. By specifying one of the direction abbreviations or the number of such an abbreviation, the player will be moved in that direction **after** this turn is over, and another turn will be taken to process the player's new location. If a direction that the player cannot go (because the room hasn't an exit in that direction), a run-time error will be generated.

Directions can also be given as a direction number. North corresponds to 0, South to 1, and so forth through 9, which is the same as the Down command. These move numbers are automatically mapped into the first nine function keys, as well.

Move	Number	Function Key
North	0	
South	1	
East	2	
West	3	
NE	4	
NW	5	
SE	6	
SW	7	
UP	8	
DOWN	9	

EXAMPLES:

```
MOVE N
MOVE SE
MOVE D
MOVE 3
MOVE (j+1) / 2
```

MOVEOBJ

SYNTAX:

MOVEOBJ {object|expression}, ((N|S|E|W|NE|NW|SE|SW|U|D)|{expression})

COMMENTS:

This command attempts to move the given object or object number in the given direction or direction number from whatever room it is currently in. If this is not possible, as when the current room has no exit in that direc-

tion, a run-time error will occur. The object's new position will be reflected by its position variable, which will be the new room's number.

EXAMPLES:

```
MOVEOBJ Magic_Sword, N
MOVEOBJ Lamp, SW
MOVEOBJ (i+1), N
MOVEOBJ Lamp, 7
```

OBJNAME

SYNTAX:

OBJNAME ({room}|{expression}), ${stringvar}

COMMENTS:

OBJNAME takes the name of the given object or object number, and pumps it into the given string variable.

EXAMPLES:

```
OBJNAME AnObjNum, $ThatObject
```

OR

SYNTAX:

OR

COMMENTS:

The OR command is used when any one of several conditions of an IF command can be met in order for the program to execute code following the THEN.

EXAMPLES:

```
IF (Player HAS Gun) OR (Player HAS Knife)
    THEN
 T Slay Dragon
ENDIF
```

PALETTE

SYNTAX:

PALETTE {buffer#}, {color}, {r}, {g}, {b}

COMMENTS:

PALETTE allows the programmer to change the settings of colors in the active buffers, like changing a red to a cyan. The R,G,B setting **must** be in the range of 0 to 15, inclusive.

EXAMPLES:

```
PALETTE 1, 0, 0, 15, 15
would set color 0 in buffer 1 to a bright
   cyan or greenish-blue.
```

PAUSE

SYNTAX:

PAUSE {expression}

COMMENTS:

PAUSE will delay execution of the next statement for the number of 1/50's of a second specified by the expression.

EXAMPLES:

```
PAUSE 9
PAUSE seconds * 50
```

PIXEL

SYNTAX:

PIXEL {buffer}, {x}, {y}, {int_variable}

COMMENTS:

PIXEL reads the pen number or color of the indicated graphics buffer at the position x,y and stores it in the integer variable given.

EXAMPLES:

```
PIXEL 0, 50,100, color
PIXEL i, j,58, pen_num
```

PLACEOBJ

SYNTAX:

PLACEOBJ {object|expression}, {room|THISROOM|expression}

COMMENTS:

This command sends the given object or object number directly to the given room, room number, or the player's current room if THISROOM. The position variable will reflect the object's new position.

EXAMPLES:

```
PLACEOBJ Magic_Sword, Torture_Chamber
PLACEOBJ Ray_Gun, Engine_Room
PLACEOBJ 17, THISROOM
PLACEOBJ (j-1), Library
PLACEOBJ 12, 9
```

PLAY

SYNTAX:

PLAY SOUND {buffer#}, {channel#}, {iterations}, {vol}, {period}
PLAY SONG {buffer#}

COMMENTS:

PLAY SOUND sends the sound in the given buffer, if loaded, to the given channel number, 0 to 3, and plays it for the given number of iterations. If 0 is given for iterations, the sound will repeat indefinitely. Channel 0 is for the left speaker, and channel 1 is for the right speaker. Volume is in the range 0 to 64 with 64 being loud—a value of 0 will use the sound's natural volume.

Period is essentially a pitch setting; 124 is the **highest** possible pitch, while a value of 800, for example, would be **really** low-pitched. In theory, any value as high as 32767 could be used, but the sound will be mush, with lots of clicks and pops. In addition, a period value of 0 will use the sound's natural pitch. This command will work **only** in DISABLEMUSIC mode.

PLAY SONG causes Visionary to play the given song, which is sequenced independently of the main program and therefore plays while the program goes on to do other things. This command will work **only** in ENABLEMUSIC mode.

EXAMPLES:

```
PLAY SOUND 0, 3, 5, 1, 64, 144
PLAY SONG  songnum
```

QUIT

SYNTAX:

QUIT [GAME]

COMMENTS:

If you use the command QUIT GAME, the game will come to an end after the current turn.

If you simply use the command QUIT, not only will the game come to a halt, but the entire program will exit back to the CLI or WorkBench, whichever it was started from.

EXAMPLES:

```
QUIT
QUIT GAME
```

READBUTTONS

SYNTAX:

READBUTTONS [EMPTY]

COMMENTS:

If a Visionary program is working in a totally graphic environment, it will probably be necessary to continue processing the mouse interactions, that is, any CLICK buttons clicked on by the player. This is **not done** within a WHILE loop, though menus are still processed. When you want to allow these buttons to execute their programs, use the READBUTTONS command. It will check for clicks and execute the correct program if a click was in the right zone.

If the word EMPTY follows this command, the input queue is **not** processed. Instead, the queue is emptied; any clicks the user queued up before the execution of this command are eliminated. This can be used after performing some lengthy operation so that the buttons the user idly pressed will not be processed.

EXAMPLES:

```
READBUTTONS
READBUTTONS EMPTY
```

RECT

SYNTAX:

RECT {buffer#}, {x1}, {y1}, {x2}, {y2}

COMMENTS:

RECT draws a solid rectangle in the given buffer number between the upper-left corner (x1,y1) and the lower right corner (x2,y2).

EXAMPLES:

```
RECT 12, 100, 50, 200, 75
```

would make a rectangle in the current color, in screen buffer 12, starting at the location 100, 50 and which is 101 wide and 26 tall.

REMOVE

SYNTAX:

REMOVE {button#}

COMMENTS:

REMOVE deactivates a screen CLICK button. Valid numbers for this button range from 0 to 49.

EXAMPLES:

```
REMOVE 0
REMOVE Count
```

RIGHT

SYNTAX:

RIGHT (${stringvar}|"text"), {expression}, ${stringvar}

COMMENTS:

RIGHT takes the given number of characters from the end if the given source string and plugs it into the destination string on the right.

EXAMPLES:

```
RIGHT $MyName, 5, $LastName
RIGHT "Mary had a little lamb", 4, $Sheep
```

ROOMNAME

SYNTAX:

ROOMNAME ({room}|{expression}|THISROOM), ${stringvar}

COMMENTS:

ROOMNAME takes the name of the given room, room number, or the player's current room if THISROOM is used, and pumps it into the given string variable.

EXAMPLES:

```
ROOMNAME ARoomNum, $CurrentRoom
ROOMNAME THISROOM, $ARoom
```

SCREENMODE

SYNTAX:

SCREENMODE (TEXT|GRAPHICS)

COMMENTS:

This command allows your Visionary program to switch between TEXT display mode, in which the standard Visionary text screen is visible, or GRAPHICS display mode, in which whatever screen buffer was last shown will be visible. If no screen buffer has been shown, then this command will cause no visible effect until a screen buffer is shown with the SHOW command, which displays the screen buffer.

SCREENMODE TEXT will not close a currently-displayed graphics screen; it will only move it behind the standard Visionary text screen.

While the system normally starts up under SCREENMODE TEXT, it is possible to force a game into SCREENMODE GRAPHICS at the beginning, after it has been linked, by linking with the **-g** linker option. See VLINK.

EXAMPLES:

```
SCREENMODE TEXT
SCREENMODE GRAPHICS
```

SCROLLBAR

SYNTAX:

SCROLLBAR (ON|OFF)

COMMENTS:

SCROLLBAR gives the program control over whether or not the player can drag the text screen around. SCROLLBAR OFF turns off this ability, SCROLLBAR ON enables it. At startup, the default is SCROLLBAR ON.

> ! **Like MENUS OFF, SCROLLBAR OFF should not be inserted until the program is debugged and ready to go. If the scrollbar is turned off, you will not be able to drag down the game screen to access Workbench.**

EXAMPLES:

```
SCROLLBAR ON
SCROLLBAR OFF
```

SCROLLTO

SYNTAX:

SCROLLTO {buffer}, {x}, {y}

COMMENTS:

SCROLLTO allows you to move the x,y offset into a very large screen so that the Amiga displays that portion of the image. For instance if you had a 640x200 screen displayed in 320x200 resolution, the player would see the left half of the whole image. By moving the upper right corner of the image from the default of 0,0 to 320,0, you will see the right half of the image. This operation can be done even on screens that are not currently visible—when displayed they will assume the most recently-set position. Though you can specify an offset as large as you like, the system will not allow you to move the x,y position PAST the point where you would see garbage on the right or bottom edges of the screen. That is, in a 320x200 video mode, when you SCROLLTO a 640x400 image, the highest values you can specify for the x,y offset are 320,200.

EXAMPLES:

```
SCROLLTO 0, 320,0
SCROLLTO i, MouseX, MouseY
```

SET

SYNTAX:

SET {room|object}, {attribute}

COMMENTS:

SET causes the given attribute for the given room, or for an object which is present in the current room, to be SET to "Y", or 1. For rooms, the two system-declared attributes, DARK and VISITED, may also be SET with this command. SET and UNSET are the only two program statements that do not allow a numeric expression to replace the room or object number. This is because Visionary has no way of knowing if an expression is intended as a room or object.

EXAMPLES:

```
SET Ray_Gun Broken
SET Gun_Turret Working
SET Hall_of_Magic DARK
```

SHOW

SYNTAX:

SHOW SCREEN {buffer#}

COMMENTS:

If in SCREENMODE GRAPHICS, SHOW SCREEN causes the given screen buffer, if loaded, to be displayed. If an invalid buffer number, such as 50, is used, the display will return to text only.

EXAMPLES:

```
SHOW SCREEN 0
SHOW SCREEN (cbuf * 2)
```

SPEECH

SYNTAX:

SPEECH (ON|OFF)

COMMENTS:

SPEECH gives the program control over whether or not the Amiga's narrator device is delivering speech output for all of the text printed to the text screen with the "T" command. At startup, the default is SPEECH OFF.

EXAMPLES:

```
SPEECH ON
SPEECH OFF
```

STOP

SYNTAX:

STOP SOUND {channel#}

STOP SONG

COMMENTS:

STOP SOUND causes any sound output on the given channel, from 0 to 3, to cease playback. This command will work **only** in DISABLEMUSIC mode.

STOP SONG causes the MED song currently playing, if any, to stop. This command will work **only** in ENABLEMUSIC mode.

EXAMPLES:

```
STOP SOUND 0
STOP SONG
```

T

SYNTAX:

T {text string}

COMMENTS:

T prints the given text string to the text interface where the game player can see it. To modify the text's style, the following commands can be embedded in the text string:

```
~B  : make following text bold
~I  : make following text italic
~N  : make following text normal
~R  : make following text reversed
~U  : make following text underlined
```

When a ~N command is issued, all the other modes are turned off.

It is also possible to print the values of variables within the text line. To do so, precede the variable name with the "at" symbol "@", and follow the variable's name with the space character. The space following the @variable will not be printed.

EXAMPLES:

```
T This is a test!
T ~UThis is an underlined (~Rand
          REVERSED!~N~U) test!!~N
T Intelligence = @intell , Constitution =
          @const , Hitpoints = @hp .
```

TEXT

SYNTAX:

TEXT {buffer#}, {x}, {y}, "text"

COMMENTS:

TEXT is very similar to the "T" command, except that extra parameters are provided to let the programmer specify where the text is to be drawn, that is, which screen buffer to put it in, and where in that screen buffer the baseline of text is to be. The X value specifies the left-most edge of the text, and the Y value specifies the baseline; this is the bottom of the normal text, while characters with descenders like "y" and "g" have tails that appear below the baseline.

The TEXT is rendered in the current font for that buffer. Each buffer defaults to font 0.

EXAMPLES:

```
TEXT 1,113, 40, "@$PlayerName , you have @HP
    Hit Points left."
```

TEXTPALETTE

SYNTAX:

TEXTPALETTE {color}, {r}, {g}, {b}

COMMENTS:

TEXTPALETTE allows the programmer to change the settings of colors in the standard text screen, like changing a red to a cyan. The R,G,B setting **must** be in the range of 0 to 15, inclusive, and the color number is in the range 0 to 3, inclusive.

The default text screen colors are:

color number	red	green	blue
0	8	8	8
1	0	0	0
2	15	15	15
3	8	0	0

EXAMPLES:

```
TEXTPALETTE 0, 0, 15, 15
would set color 0 to a bright cyan or
   greenish-blue.
```

UNLOAD

SYNTAX:

UNLOAD SCREEN {buffer#}
UNLOAD SOUND {buffer#}
UNLOAD FONT {buffer#}
UNLOAD SONG {buffer#}

COMMENTS:

UNLOAD causes whatever audio, image, font or song data loaded into the specified buffer to be de-allocated and deactivated. Valid buffer values for screens and sounds are from 0 to 24. Valid buffer values for fonts and songs are 0 to 9.

If the currently open SCREEN is unloaded, it is removed from the display, which enters SCREENMODE TEXT.

If a currently playing SONG or SOUND is unloaded, playback is stopped before it is removed from the system.

EXAMPLES:

```
UNLOAD SCREEN 0
UNLOAD SOUND  (count DIV 9)
UNLOAD FONT   myfont
UNLOAD SONG   songnum-1
```

UNSET

SYNTAX:

UNSET {room|object}, {attribute}

COMMENTS:

UNSET causes the given attribute for the given room, or for an object which is present in the room, to be SET to "N", or 0. For rooms, the two system-declared attributes, DARK and VISITED, may also be UNSET with this command. SET and UNSET are the only two program statements that do not allow a numeric expression to replace the room or object number. This is because Visionary has no way of knowing if an expression is intended as a room or object.

EXAMPLES:

```
UNSET Ray_Gun Broken
UNSET Gun_Turret Working
UNSET Hall_Of_Magic DARK
```

UPCASE

SYNTAX:

UPCASE ${stringvar}

COMMENTS:

UPCASE shifts all of the alphabetic characters in the string to upper case.

EXAMPLES:

```
UPCASE $MyName
```

VALUE

SYNTAX:

VALUE (${stringvar}|"text"), {intvar}

COMMENTS:

if the string holds an integer number, its numeric value is stored in the given integer variable. If an error occurs, as when the character was not really a valid integer, the ASCII value of the first character will be stored in the integer variable instead.

EXAMPLES:

```
VALUE Count, $AString
VALUE Number, "-137"
VALUE asciival, "hello!"
```

WHILE

SYNTAX:

WHILE {expression} DO

. ..

ENDWHILE

COMMENTS:

WHILE allows the code that follows it to be executed a number of times, repeating indefinitely until the expression in the WHILE statement is evaluated as "0" or "false". Every WHILE command is followed by the word DO after the expression of the WHILE command.

EXAMPLES:

```
count := 1;
WHILE count < 100 DO
 T This is a test!
 count := count + 1
ENDWHILE
```

Appendix A: Error Codes

VCOMP Errors

Probable cause and possible solution is provided for each of the following compiler errors. Some errors are **internal** errors, that is, something your program code is doing is inherently incompatible with the Visionary compiler, even though it appears to be okay on the surface. These errors should seldom occur.

Most errors have a simple solution, however, even when the probable cause sounds simple, finding the source may not be. Use the tools provided to help you find the cause—for example, a spelling error may be found by searching the .WRD and .XRF files.

0: CANNOT RE-OPEN FILE.

Probable Cause: AmigaDOS was unable to open the file for Pass 2

Solution: This error may occur when the compiler was interrupted or when a file was changed after compiling. Don't use the .GAM and .WRD files, except with the DBUG and VLINK utilities.

1: Undefined room.

Probable Cause: Room not yet created or name misspelled

Solution: Check spelling or create the room

2: Undefined object.

Probable Cause: Object not yet created or name misspelled

Solution: Check spelling or create the object

4: Undefined attribute.

Probable Cause: Attribute not yet created or name misspelled

Solution: Check spelling or create the atttribute

6: Division by Zero.

Probable Cause: An expression is divided by the constant 0

Solution: Rework the expression

7: Illegal operation.

Probable Cause: An operator in an expression was invalid

Solution: Rework the expression

10: Undefined object or room.

Probable Cause: The room or object was misspelled or not defined

Solution: Check spelling or define the room or object

11: Unknown direction.

Probable Cause: A direction was given that was not valid

Solution: Use only the valid directions, which are N, S, E, W, NW, NE, SW, SE, U, D

12: Illegal statement.

Probable Cause: A statement was misspelled or invalid

Solution: Check spelling and usage, refer to manual

13: Object name already used.

Probable Cause: More than one instance of an object was attempted

Solution: Rename one of the objects

14: Object name already used as room name.

Probable Cause: The given name was already used for a room

Solution: Rename one of them

15: "OBJECT" or "NPC" expected.

Probable Cause: The first word of an object definition must be OBJECT or NPC

Solution: Use the appropriate word, check spelling

16: Attribute declared twice.

Probable Cause: More than one instance of an attribute was attempted

Solution: Rename one of the attributes

17: Too many attributes.

Probable Cause: More than 31 total attributes declared

Solution: Reduce the number of attributes

18: "INITROOM" expected.

Probable Cause: The word INITROOM was not found
Solution: Insert it

20: Verb list expected.

Probable Cause: A list of verbs was expected
Solution: Supply them

21: Room declared twice.

Probable Cause: More than one instance of a room was attempted
Solution: Rename one of the rooms

22: "ROOM" expected.

Probable Cause: The first word of a room def. must be ROOM
Solution: Supply the word

23: Cannot re-define room.

Probable Cause: This is an internal Visionary error
Solution: Contact Technical Support

25: Cannot open VOCAB file.

Probable Cause: The path/filename provided was not valid
Solution: Check spellings

26: "VOCAB" expected.

Probable Cause: The first word of a vocab file must be VOCAB
Solution: Insert it, check spelling

27: "ACTION" or "ENDVOCAB" expected.

Probable Cause: The word ACTION or ENDVOCAB was expected next
Solution: Insert the appropriate word, check spelling

28: "ADVENTURE" expected.

Probable Cause: The first word of a .ADV file must be ADVENTURE

Solution: Insert it, check spelling

29: "PASSWORD" expected.

Probable Cause: The second line of a .ADV must be PASSWORD

Solution: Insert it, check spelling

30: Password string expected.

Probable Cause: The word PASSWORD must be followed by the game password

Solution: Insert one

31: Illegal password. Try another.

Probable Cause: The password given was not sufficient to protect the game—usually it is too short

Solution: Use another, longer password

32: Variable already declared.

Probable Cause: More than one instance of a variable was attempted

Solution: Rename one of the variables

33: Cannot open file.

Probable Cause: The path/filename given was invalid

Solution: Check spelling and validity

34: "ENDADVENTURE" expected.

Probable Cause: The last word in the .ADV file must be ENDADVENTURE

Solution: Insert it

35: Cannot open master file.

Probable Cause: The path/filename.ADV given was invalid

Solution: Check spelling and validity

36: OUT OF MEMORY.

Probable Cause: The system is out of RAM

Solution: If multitasking, shut down some programs

38: THISROOM attribs MUST be system attribs "DARK" or "VISITED".

Probable Cause: DARK and VISITED are the only attribs that work with THISROOM

Solution: Use only DARK and VISITED

40: Block not ended properly.

Probable Cause: The appropriate END for a block was not found

Solution: Insert the appropriate ENDxxxx

41: SYSTEM ERROR: Cannot track object into inventory.

Probable Cause: Internal error

Solution: Contact Technical Support

42: Text line is too long.

Probable Cause: Line length may not exceed 127 characters

Solution: Shorten the line

43: Subroutine name already used.

Probable Cause: More than one instance of a subroutine was attempted

Solution: Rename one of the subroutines

44: "SUB" or "SUBROUTINE" expected.

Probable Cause: The first word of a subroutine def. must be SUBROUTINE

Solution: Insert it

45: Unable to re-define subroutine.

Probable Cause: Internal error

Solution: Contact Technical Support

46: Undefined code block.

Probable Cause:

Solution:

47: Cannot open .WRD file.

Probable Cause: An AmigaDOS error prevented access to the file
Solution: Shut down other programs or reboot and try again

54: ACTION list too long.

Probable Cause: A list of verbs mey be UP TO 79 characters in length
Solution: Shorten the line

55: Expression too complex.

Probable Cause: More than 50 operators or 50 operands were used
Solution: Shorten the expression

56: Mismatched parenthesis.

Probable Cause: Too many "(" or ")"
Solution: Rework expression

57: Unknown operand.

Probable Cause: An operand in the expression was not valid
Solution: Check spelling and validity

58: ")" expected.

Probable Cause: ")" was expected
Solution: Rework expression

59: Expression contains incompatible operation(s).

Probable Cause: An operator was incompatible with one of its operands
Solution: Rework expression

60: EQUATE expression expected.

Probable Cause: "var := " was expected
Solution: Rework expression

61: Missing operand(s).

Probable Cause: An operator must had two operands
Solution: Rework expression

62: ATTRIBUTES already declared.

Probable Cause: More than one ATTRIB block was found in the same room or object

Solution: Join the two or delete one

63: DEFAULT block already declared.

Probable Cause: More than one DEFAULT block was found in the same room

Solution: Join the two or delete one

64: CODE already declared.

Probable Cause: More than one CODE block was found in the same room or object

Solution: Join the two or delete one

65: Name already used as variable name.

Probable Cause: The name used is already a declared variable

Solution: Rename one of them

66: NAME block already declared.

Probable Cause: More than one NAME block was found in the same object

Solution: Join the two or delete one

67: ADJ block already declared.

Probable Cause: More than one ADJ block was found in the same object

Solution: Join the two or delete one

68: INITROOM already declared.

Probable Cause: More than one INITROOM was found in the same object

Solution: Discard one

69: Quotation mark expected.

Probable Cause: Either a " or a ' was expected

Solution: Insert one

70: "SOUND", "SCREEN", "SONG" or "FONT" expected.

Probable Cause: One of these four words was expected
Solution: Check spelling

71: Expression expected.

Probable Cause: A numerical expression was expected but not found
Solution: Insert one

72: Filename expected.

Probable Cause: A filename was expected but not found
Solution: Insert one

73: "SCREEN" expected.

Probable Cause: The word SCREEN was expected but not found
Solution: Insert it

74: "ON", "OFF", "ONCE" or "RESTORE" expected.

Probable Cause: One of these four words was expected
Solution: Check spelling

75: "XOR", "DRAW", or "OVERLAY" expected.

Probable Cause: One of these three words was expected
Solution: Check spelling

76: Screen type expected.

Probable Cause: No screen types were found
Solution: Add screen types

77: "SCREEN" expected.

Probable Cause: The word SCREEN was expected
Solution: Insert it

78: ENDxxx expected.

Probable Cause: A specific END was expected but not found
Solution: Check spelling and block structure

79: "ON" or "OFF" expected.

Probable Cause: One of these two words was expected
Solution: Check spelling

80: Variable name expected.

Probable Cause: The name of a variable was expected
Solution: Check spelling

81: String variable expected.

Probable Cause: The name of a string variable was expected
Solution: Check spelling

82: Undefined variable.

Probable Cause: An invalid variable name was used
Solution: Check spelling and validity

83: ":=" Expected.

Probable Cause: The equate symbol, ":=" was expected
Solution: Rework expression

84: Text literal or string variable expected.

Probable Cause: One of these was expected
Solution: Insert it

85: Too many articles.

Probable Cause: Too many articles were declared
Solution: No more than 10 are allowed

86: Too many prepositions.

Probable Cause: Too many prepositions were declared
Solution: No more than 50 are allowed

87: Premature end of line.

Probable Cause: More was expected on the line, but it ended
Solution: Insert the missing stuff

89: Name cannot be a keyword.

Probable Cause: Entity names may not be the names of Visionary keywords
Solution: Rename the entity

90: "SONG" expected.

Probable Cause: The word SONG was expected
Solution: Insert it

91: "MIDI" or "NOMIDI" expected.

Probable Cause: One of these two words was expected
Solution: Use one of them

92: "TEXT" or "GRAPHICS" expected.

Probable Cause: One of these two words was expected
Solution: Use one of them

93: Unrecognized parameter.

Probable Cause: A parameter supplied to a statement made no sense
Solution: Check spelling and validity

94: Literal string expected.

Probable Cause: A quoted text string was expected
Solution: Supply one

1000: INTERNAL ERROR REDEFINING OBJPOS VAR.

Probable Cause: All errors > = 1000 are internal errors
Solution: Contact Technical Support

1010: Precedence evaluator problem.

1011: Illegal operation sub-type.

DBUG Errors

Like the VCOMP errors, finding the source of DBUG errors may be simple, or it may involve some real searching. The probable cause and a brief solution is given for each error.

1: Division by Zero.

Probable Cause: The divisor evaluated to 0.

Solution: Rework the expression or add a zero check

2: Unable to execute DOS function.

Probable Cause: An AmigaDOS error prevented the call of a DOS command

Solution: Check spelling, make sure the file is present

3: Object not in player's inventory.

Probable Cause: This is not necessarily an error

Solution: Depends what you are doing

4: Requested direction has no room linked to it.

Probable Cause: No room was LINKed in the direction of motion

Solution: Add a LINK statement, or amend a DIRECTIONS statement

6: SYSTEM ERROR in Variable Formatting.

Probable Cause: Internal error

Solution: Contact Technical Support

8: Object can't move that direction in its current room.

Probable Cause: No room was LINKed in the direction of motion—this is not necessarily an error

Solution: Add a LINK statement, or amend a DIRECTIONS statement

9: ELSE not found in IF block.

Probable Cause: An ELSE was found where it didn't belong

Solution: Remove it

10: Illegal Statement.

Probable Cause: Internal error or file corrupt

Solution: Try a backup copy, recompile, or call Technical Support

11: Linked direction not enabled.

Probable Cause: A LINK was made for a disabled direction

Solution: This is not **necessarily** an error, but the information is included so you can decide if this is what you wanted to do

12: Position variable set to invalid room number.

Probable Cause: An object's namePOS variable was set to a number not representing a valid room number

Solution: Rework the expression

13: Sound buffer index out of range.

Probable Cause: Valid sound buffer range is 0 to 24

Solution: Use a valid number

14: Error loading sound sample.

Probable Cause: The file could not be loaded from disk

Solution: Check filename, also make sure it's an IFF 8SVX file

15: Sound channel number out of range.

Probable Cause: Valid channel number range is 0 to 1

Solution: Use a valid number

16: No sample in sound buffer.

Probable Cause: No sample is loaded into that buffer

Solution: Use a different buffer or find out why

17: Screen buffer index out of range.

Probable Cause: Valid screen buffer number range is 0 to 24

Solution: Use a valid number

19: Buffer not active.

Probable Cause: No screen is loaded into that buffer

Solution: Use a different buffer or find out why

20: Color number out of range.

Probable Cause: The given screen buffer cannot support that pen number

Solution: Use a different number, or rework expression

21: Color setting out of range.

Probable Cause: The given screen buffer cannot support that pen number

Solution: Use a different number, or rework expression

22: Click region number out of range.

Probable Cause: Valid click region number range is 0 to 49

Solution: Use a valid number

23: Mask buffer not deep enough for this OVERLAY COPY.

Probable Cause: Mask buffer must be as deep as SOURCE buffer

Solution: Make mask buffer deeper

24: Mask buffer not large enough for this OVERLAY COPY.

Probable Cause: Mask buffer x,y must be as large as COPY's x,y size

Solution: Make mask buffer larger

25: Direction number out of range.

Probable Cause: Valid direction number range is 0 to 9

Solution: Use a valid number

26: Dissolve screens not alike.

Probable Cause: Screens used in a DISSOLVE must have same x,y,depth

Solution: Modify one or both buffer dimensions

30: Font buffer index out of range.

Probable Cause: Valid font buffer number range is 0 to 9

Solution: Use a valid number

31: Could not access font.

Probable Cause: Font could not be found in the current FONTS: directory or no CHIP memory

Solution: Check filename, spelling, and size of font

33: Error loading song.

Probable Cause: A MED song could not be loaded or no memory

Solution: Check path/filename and spelling

34: Song buffer not active.

Probable Cause: No song is loaded into the given buffer number

Solution: Use a different number of find out why

35: Error opening MED Music player.

Probable Cause: The file MEDPlayer.library was not in LIBS: or another program is already using it

Solution: Make sure the file is present or shut down other programs

37: Sampled-Sound system not active.

Probable Cause: The music system is now running; no sampled sounds may be played

Solution: Use DISABLEMUSIC to allow sampled sounds to play

38: MED Music system not active.

Probable Cause: The sampled-sound system is running; no songs may be played

Solution: Use ENABLEMUSIC to allow songs to play

39: No song to continue.

Probable Cause: In a CONTINUE SONG, no song had been playing before

Solution: This is not necessarily an error

40: Font not valid.

Probable Cause: Not font was loaded into the current font buffer

Solution: Use a different font number or find out why

41: Error loading screen.

Probable Cause: The path/filename was not an IFF ILBM or no memory or not enough CHIP memory

Solution: Check filename and check available memory

42: SUBROUTINE CALL-STACK OVERFLOW.

Probable Cause: More than 128 levels of code-block execution was attempted. Should not happen without recursion

Solution: Avoid recursion

97: Bad Command.

Probable Cause: The command sent to the command interpreter was invalid

Solution: Check spelling

98: Could not open Narrator device!

Probable Cause: The file Narrator.device was not in DEVS:, or something already has exclusive access to the sound channels

Solution: Check for the file, or shut down other programs

99: OUT OF MEMORY.

Probable Cause: No memory left

Solution: Shut down other programs or reduce number of buffers loaded at the same time

1000: INTERNAL ERROR: BAD CYCLE INFO.

Probable Cause: All errors > = 1000 are internal errors

Solution: Call Technical Support.

1001: SYSTEM ERROR: String lookup out of range.

1002: SYSTEM ERROR: Alias lookup out of range.

1003: SYSTEM ERROR: Object lookup out of range.

1004: SYSTEM ERROR: Room lookup out of range.

1005: SYSTEM ERROR: Variable lookup out of range.

1006: SYSTEM ERROR: Adjective lookup out of range.

1007: SYSTEM ERROR: Subroutine lookup out of range.

1008: SYSTEM ERROR: String variable lookup out of range.

1009: SYSTEM ERROR: in String Variable Formatting.

1010: SYSTEM ERROR: Vocab lookup out of range.

Appendix B: Visionary Utility Programs

LoadScreen

LoadScreen allows the Visionary programmer to let the player view an IFF image while the game loads, usually at the time of boot-up. The opening screen or **title screen** image must not be encoded.

The command line format for LoadScreen is:

```
RUN LoadScreen {filename} [-CMTxxx]
WAIT {some number of seconds}
```

The reason for two commands is that LoadScreen does not terminate after loading the image; it sticks around to run timers and color cycling.

The switches are:

C	Cycle colors
M	Left mouse button exits—this is **not** a default!
T*xxx*	Timer waits *xxx* seconds and exit.

These options may be used in any combination with each other. Note that there seems to be no default way to exit. This is because LoadScreen will, by default, wait for a message from the DOS command "CloseScreen", usually called from within the Visionary program. **In a bootable game disk, LoadScreen must be in the C: directory.**

CloseScreen

CloseScreen is a DOS command that the Visionary programmer needs to shut down a title screen loaded in the startup script when a game is booted.

Its command line format is:

```
CloseScreen {filename}
```

The filename used must **exactly** match the filename supplied to LoadScreen, or it will not work.

The CloseScreen command will still work even if switches were used with LoadScreen to allow the user other methods of terminating the display. If the screen has already been closed, no action will be taken. CloseScreen is used as a Visionary command, to let you close a screen that was opened with the LoadScreen utility. **In a bootable game disk, CloseScreen must be in the C: directory.**

VCODE

VCode allows the Visionary programmer to protect his IFF images and sound samples from prying eyes and ears by encrypting the contents of those files. The program does not need to be modified in any way. Visionary detects the encryption and decodes the files as needed. VCode also allows files to be decoded—but only by the author.

The command line format for VCode is:

```
VCode {input filename} {game password}
    [{output filename}]
```

or

```
VCode {input filename} -C
```

In the first case, VCode will load the input file and automatically determine if the file needs to be encoded or decoded. The {game password} **must** be the same password found in the .ADV file for the game. If you specify the optional {output filename}, the processed data will be written to that file. Otherwise, it will be written directly over the old data in the input file.

The second case above allows the user to supply the switch "-C" instead of a password. This will make VCode **check** to see whether the file has been encoded or not. No output will be generated.

Both IFF-ILBM (image) files and IFF-8SVX sampled sound files may be encoded.

VCOORD

The **VCOORD** utility is run from the CLI/Shell. It returns to the CLI the X and Y screen coordinates of the mouse pointer when you clicked the mouse button. These coordinates can then be used in defining click zones, specifying areas for graphic commands, copying images from one buffer to another, and any other place where a precise screen location is required.

The command to run VCOORD is:

```
VCOORD<filename>[-p]
```

When the option -p is given, VCOORD will print the coordinates of the mouse click to the CLI window as well as displaying them on the screen. This gives you a separate record, since with each subsequent mouse click, VCOORD replaces the coordinates from the previous click in your graphic window.

PrepGameDisk

The **PrepGameDisk** utility formats a blank disk in df0: and installs it so that it will be a bootable disk, makes the directories needed for a bootable disk, copies the files from your system which are necessary to make a bootable

disk and then copies the files needed for every Visionary game, such as LoadScreen and CloseScreen, and the MED player library to the disk.

The command to run PrepGameDisk is:

```
execute PrepGameDisk
```

After the utility is run, you'll want to rename the prepared disk and create the startup-sequence as directed in the prompts from the PrepGameDisk utility. You should copy the executable game file, and any commands from the c: directory that your game or the startup-sequence will use. Be sure to copy to the disk fonts: directory any special fonts your game uses as well.

Appendix C: The Tutorial Game Source Files

The source code listings are provided for your convenience in exploring one solution to the tutorial game. Entering the source code line by line into a text editor will help you get a good feeling for how the code is developed. The asterisk * at the start of each new line of code is there to indicate which lines are new lines, and which are overlapping text from the prveious line. Do not enter the asterisk as part of the code.

If you prefer, you can simply load the source code files from the Catacoombs disk into your text editor. The graphics and sound files for the tutorial game are stored on the Visionary disk.

The Potion.ADV File

```
*
*    ;---------- Potion.ADV ----------
*
*    ADVENTURE
*
*    PASSWORD jro
*
*
*    VAR
*    RoomNumber    1  ; room number 1,2,3,4
*    $filename        ; filename variable for loading files
*    ScreenX          ; x position of current location screen
*    ScreenY          ; y position of current location screen
*    $tx              ; text string to be printed
*    temp          0  ; temporary variable for various uses
*    $temp            ; temporary string for various uses
*    $letter          ; single character for getstring
*    $sentence       ; built up sentence of input
*    sentence      ; length of the $sentence
*    TextPosition      ; position text on graphic screen
*    MaxLines     5  ; maximum lines in text window before pause
*    CountLines       ; counter for lines displayed in text window
*    Defeat     0  ; set to 1 to defeat text window pause
*    $return       ; RETURN cannot be defined here
```

```
*     return
*     $backspace      ; BACKSPACE cannot be defined here
*     backspace
*     MainLoop        ; main loop while variable
*     TextColor    8  ; default text color
*     white     27    ; palette color for white
*     blue     8   ;  palette color for blue
*     green      4    ; palette color for green
*     brown     23    ; palette color for brown
*     dummy          ; dummy variable for load/save check
*     val            ; value of the input character
*     ButtonUsed     ;
*     x1          ;\
*     y1          ; \
*     x2          ;  \ variables for block copies
*     y2          ; / showing buttons depressed
*     x           ; /
*     y           ;/
*     GoN    _13     ; offset to add to N button, if lighted
*     GoS     13     ; offset to add to S button, if lighted
*     GoE     13     ; offset to add to E button, if lighted
*     GoW     13     ; offset to add to W button, if lighted
*     GoU     13     ; offset to add to U button, if lighted
*     GoD     13     ; offset to add to D button, if lighted
*     offset  -13     ; offset to pop button up
*
*
*     ENDVAR
*
*     ROOM
*     Potion.rooms
*     ENDROOM
*
*     OBJECT
*     NonMovable.obj
*     Movable.obj
*     ENDOBJECT
*
*     SUB
*     Potion.SUB
*     MainLoop.SUB
*     StartUp.SUB
*     ENDSUB
*
*     VOCAB
```

```
*     Potion.VOC
*     ENDVOCAB
*
*     INITROOM  ByTree
*
*     ENDADVENTURE
*
*     ;-------------------------------------
*
*
```

The Potion.ROOMS File

```
*
*     ;---------------- Potion.rooms ----------------
*
*     room unused
*
*     code
*     endcode
*
*     endroom
*
*     ;-------------------------------------
*
*     room ByTree
*
*     attrib
*      started N
*     endattrib
*
*     default
*      w ByShack
*      u InTreeTop
*     enddefault
*
*     code
*
*     If ByTree not started then
*      call StartUp
*     endif
*
*      RoomNumber := 1
```

```
*      ScreenX := 0
*      ScreenY := 0
*      Call ReDrawScreen
*      play sound 1, 0,0,40,0 ; ocean
*      stop sound    1        ; birds
*
*     call ClearButtons
*
*     click 34, 16,5,   40,20,   SeeSun
*     click 35, 116,13, 186,42,  SeeTreeTop
*     click 36, 116,37, 147,105, SeeTreeTrunk
*     click 37, 237,46, 247,53,  SeeIsland
*     click 38, 5,46,   249,89,  SeeOcean
*     click 39, 5,91,   249,131, SeeSand
*     click 40, 5,5,    249,45,  SeeSky
*
*     placeobj treetop, thisroom
*     placeobj ocean, thisroom
*     placeobj sand, thisroom
*     placeobj sun, thisroom
*     placeobj sky, thisroom
*
*     if thisroom not visited then
*      $tx:="You stand by a single tall palm tree."
*      call print
*     else
*      $tx:="You're back by the tree."
*      call print
*     endif
*
*     if ByTree not started then
*      call StartUp2
*     endif
*
*     endcode
*
*     endroom
*
*     ;-------------------------------
*
*     room ByShack
*
*     default
*      e ByTree
*      u ShackRoof
```

```
*     enddefault
*
*     code
*
*     RoomNumber := 2
*     ScreenX := 245
*     ScreenY := 0
*     call ReDrawScreen
*     play sound 1, 0,0,20,0
*     play sound 2, 1,0,20,0
*
*     call ClearButtons
*
*     click 33, 193,94, 219,110, SeePlant
*     click 34, 119,73, 135,86,  SeePlant
*     click 35, 72,62,  79,71,   SeePlant
*     click 36, 59,51,  70,65,   SeePlant
*     click 37, 119,39, 163,69,  SeeTreeTop
*     click 38, 5,27,   33,110,  SeeShack
*     click 39, 31,35,  249,75,  SeeDunes
*     click 40, 5,73,   249,131, SeeSand
*     click 41, 23,5,   248,37,  SeeSky
*     click 42, 64,37,  118,52,  SeeSky
*
*     placeobj plant, thisroom
*     placeobj treetop, thisroom
*     placeobj sand, thisroom
*     placeobj sky, thisroom
*
*     if thisroom not visited then
*      $tx:="You are by an old run down shack."
*      call print
*     else
*      $tx:="Back by the shack."
*      call print
*     endif
*
*     endcode
*
*     endroomR*
*     ;-------------------------------------
*
*     room InTreeTop
*
*     default
```

```
*     d ByTree
*    enddefault
*
*    code
*
*    RoomNumber := 3
*    ScreenX := 0
*    ScreenY := 127
*    call ReDrawScreen
*
*    call ClearButtons
*
*    click 36, 163,98, 182,131, SeeLadder
*    click 37, 102,95, 127,131, SeeLadder
*    click 38, 93,58,  119,83,  SeeSun
*    click 39, 5,86,   249,131, SeeTreeTop
*    click 40, 5,5,    249,84,  SeeSky
*
*    placeobj sky, thisroom
*    placeobj sun, thisroom
*    placeobj ladder1, thisroom
*    placeobj treetop, thisroom
*
*    if thisroom not visited then
*     $tx:="You are sitting in the palm branches."
*     call print
*    else
*     $tx:="In the tree top."
*     call print
*    endif
*
*    endcode
*
*    endroomR*
*    ;-----------------------------------
*
*    room ShackRoof
*
*    default
*     d ByShack
*    enddefault
*
*    code
*
*    RoomNumber := 4
```

```
*     ScreenX := 245
*     ScreenY := 127
*     call ReDrawScreen
*
*     call ClearButtons
*
*     click 33, 42,15,  59,51,   SeeDriftwood
*     click 34, 61,105, 78,126,  SeeHole
*     click 35, 179,47, 195,63,  SeePlant
*     click 36, 210,80, 220,97,  SeeLadder
*     click 37, 231,86, 241,112, SeeLadder
*     click 38, 11,79,  238,131, SeeRoof
*     click 39, 50,54,  163,79,  SeeRoof
*     click 40, 5,5,    250,82,  SeeSand
*
*     placeobj plant, thisroom
*     placeobj ladder1, thisroom
*     placeobj sand, thisroom
*
*     if thisroom not visited then
*      $tx:="You are sitting on the shack's roof."
*      call print
*     else
*      $tx:="On the roof."
*      call print
*     endif
*
*     endcode
*
*     endroomR*
*     ;-------------------------------
*
```

```
*
*  ;————————————— Movable.obj —————————————
*
*  object ladder
*
*  name ladder
*
*  adj wood, wooden
*
*  attrib
*   AgainstShack Y
*   AgainstTree N
*  endattrib
*
*  initroom ByShack
*
*  code
*
*  if player has ladder then
*   $tx:=" a wooden ladder"
*  elsif ladder is AgainstShack then
*   $tx:="A ladder is propped against the shack."
*  elsif ladder is AgainstTree then
*   $tx:="A ladder is propped against the tree."
*  else
*   $tx:="A wooden ladder lies here."
*  endif
*  call print
*
*  endcode
*
*  action get, take, grab
*   if player has ladder then
*    call HaveIt
*   else
*    directions ByTree, w
*    directions ByShack, e
*    unset ladder, AgainstShack
*    unset ladder, AgainstTree
*    grab ladder
*    call ReDrawScreen
*    $tx:="OK."
```

```
*    call print
*   endif
*  endact
*
*  action look, examine
*   $tx:="It looks old and weather-beaten.  But it"
*   call print
*   $tx:="should hold you."
*   call print
*  endact
*
*  action drop
*   if player has ladder then
*    drop ladder
*    $tx:="OK."
*    call print
*    call ReDrawScreen
*   else
*    call NoHave
*   endif
*  endact
*
*  action put, set, lay, lean, prop
*   if player has ladder then
*    drop ladder
*    $tx:="OK."
*    call print
*   endif
*
*   if objnoun is tree then
*    if ladder is AgainstTree then
*     call AlreadyIs
*    else
*     set ladder, AgainstTree
*     ;link ByTree, u, InTreeTop
*     directions ByTree, w u
*     $tx:="It leans against the tree and leads into"
*     call print
*     $tx:="the branches."
*     call print
*    endif
*   elsif objnoun is shack then
*    if ladder is AgainstShack then
*     call AlreadyIs
*    else
```

```
*     set ladder, AgainstShack
*     ;link ByShack, u, ShackRoof
*     directions ByShack, e u
*     $tx:="It leans against the shack."
*     call print
*    endif
*   elsif objnoun is sand then
*   else
*    $tx:="You can't do that."
*    call print
*   endif
*   call ReDrawScreen
*  endact
*
*  action climb
*   if ladder is AgainstShack then
*    go ShackRoof
*   elsif ladder is AgainstTree then
*    go InTreeTop
*   else
*    $tx:="You can't.  It's not leaning against"
*    call print
*    $tx := "anything."
*    call print
*   endif
*  endact
*
*  endobject
*
*  ;-----------------------------------------
*
*  object bottle
*
*  name bottle, potion, liquid
*
*  adj glass, magic
*
*  attrib
*   sealed Y
*  endattrib
*
*  initroom InTreeTop
*
*  code
*   placeobj cork, thisroom
```

```
*   if player has bottle then
*    $tx:=" a glass bottle"
*    call print
*   else
*    $tx:="A glass bottle is here."
*    call print
*   endif
*  endcode
*
*  action look, examine, search, view
*   $tx:="You can see some liquid inside."
*   call print
*  endact
*
*  action get, take, grab
*   if player has bottle then
*    call HaveIt
*   else
*    grab bottle
*    call ReDrawScreen
*    $tx:="OK."
*    call print
*   endif
*  endact
*
*  action drop, throw
*   if player has bottle then
*    drop bottle
*     $tx:="OK."
*     call print
*     call ReDrawScreen
*   else
*    call NoHave
*   endif
*  endact
*
*  action open, unseal, uncork
*   if player has bottle then
*    if bottle is sealed then
*     if player has corkscrew then
*      $tx := "The cork pulls out of the bottle, and"
*      call print
*      $tx := "falls to brittle bits.  A sweet aroma"
*      call print
*      $tx := "eminates from the bottle opening."
```

```
*      call print
*      unset bottle, sealed
*     else
*      $tx := "The cork won't come out.  Maybe if you"
*      call print
*      $tx := "had a corkscrew?"
*      call print
*     endif
*    else
*     call AlreadyIs
*    endif
*   else
*    call NoHave
*   endif
*  endact
*
*  action drink, swallow, embibe, taste
*   call DrinkBottle
*  endact
*
*  action break, smash, hit
*   $tx:="It won't break."
*    call print
*  endact
*
*
*  endobject
*
*  ;-------------------------------------
*
*
*  object corkscrew
*
*  name corkscrew, screw
*
*  adj cork
*
*  initroom ShackRoof
*
*  code
*   if player has corkscrew then
*    $tx:=" a corkscrew"
*    call print
*   else
*    $tx:="A corkscrew is here."
```

```
*    call print
*   endif
*  endcode
*
*  action look, examine, search, view
*   $tx:="The corkscrew has a blue plastic handle"
*   call print
*   $tx:="and a spiral stainless steel shaft."
*   call print
*  endact
*
*  action get, take, grab
*   if player has corkscrew then
*    call HaveIt
*   else
*    grab corkscrew
*    call ReDrawScreen
*    $tx:="OK."
*    call print
*   endif
*  endact
*
*  action drop, throw
*   if player has corkscrew then
*    drop corkscrew
*     $tx:="OK."
*     call print
*     call ReDrawScreen
*   else
*    call NoHave
*   endif
*  endact
*
*  endobject
*
*  ;-----------------------------------------
*
*
```

```
*
*  ; ———————— NonMovable.obj ————————
*
*  object 1nothing
*
*  name nothing
*
*  initroom unused
*
*  code
*
*  if items = 1 then
*   $tx:="You have nothing in your inventory."
*   call print
*  else
*   $tx:="You carry the following items:"
*   call print
*  endif
*
*  endcode
*
*  endobject
*
* ;————————————————————————————
*
*  object sand
*
*  name ground, floor, sand, dirt
*
*  initroom unused
*
*  code
*  endcode
*
*  action look, examine, search
*   $tx := "It's just normal beach sand."
*   call print
*  endact
*
*  action get, take, grab
*   $tx := "Leave the sand alone!  The next thing,"
```

```
*    call print
*    $tx := "you'll try getting the sun!"
*    call print
*   endact
*
*   endobject
*
*   ;-------------------------------------
*
*   object TreeTop
*
*   name boughs, fronds, greenery, top
*
*   adj tree
*
*   initroom unused
*
*   code
*   endcode
*
*   action look, examine, search
*    $tx := "The wide green fronds look smooth and"
*    call print
*    $tx := "feathery.  Quite comfy!"
*    call print
*   endact
*
*   action get, take, grab
*    $tx := "Leave the greenery alone."
*    call print
*   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."
*    call print
*   endact
*
*   endobject
*
```

```
*   ;-------------------------------------
*
*   object island
*
*   name island, rock
*
*   initroom ByTree
*
*   code
*   endcode
*
*   action look, examine, search
*    $tx := "This appears to be a small rock island"
*    call print
*    $tx := "out beyond the surf."
*    call print
*   endact
*
*   action get, take, grab
*    $tx := "It's way off in the distance, out to"
*    call print
*    $tx := "sea.  There's no way you can get it."
*    call print
*   endact
*
*   endobject
*
*   ;-------------------------------------
*
*   object ocean
*
*   name ocean, water, sea
*
*   initroom unused
*
*   code
*   endcode
*
*   action look, examine, search
*    $tx := "The bright sun reflects off the clear"
*    call print
*    $tx := "blue surface of the salty waters."
*    call print
*   endact
*
```

```
*   action get, take, grab
*    $tx := "You get a small amount in your hands,"
*    call print
*    $tx := "but all it does is get your hands wet."
*    call print
*   endact
*
*   endobject
*
*   ;-------------------------------------
*
*   object plant
*
*   name plant, plants, grass
*
*   initroom unused
*
*   code
*   endcode
*
*   action look, examine, search
*    $tx := "The thin green beach grass waves about"
*    call print ;
*    $tx := "in the slight breeze."
*    call print
*   endact
*
*   action get, take, grab
*    $tx := "It is strongly rooted, and won't come"
*    call print
*    $tx := "up.  You decide to leave it alone."
*    call print
*   endact
*
*   endobject
*
*   ;-------------------------------------
*
*   object dunes
*
*   name dunes, sanddunes
*
*   initroom ByShack
*
```

```
*   code
*   endcode
*
*   action look, examine, search
*    $tx := "The sand dunes are tall and rounded."
*    call print
*    $tx := "They look like they would be fun."
*    call print
*   endact
*
*   action get, take, grab
*    $tx := "You're here, and they are over there."
*    call print
*    $tx := "Now, how do you expect to get them?"
*    call print
*   endact
*
*   endobject
*
*   ;-------------------------------------
*
*   object hole
*
*   name hole, opening
*
*   initroom ShackRoof
*
*   code
*   endcode
*
*   action look, examine, search
*    $tx := "You can't see anything through the"
*    call print
*    $tx := "small weather-beaten opening."
*    call print
*   endact
*
*   action get, take, grab
*    $tx := "That's like trying to eat the hole out"
*    call print
*    $tx := "of a donut!  Forget it!"
*    call print
*   endact
*
```

```
*   endobject
*
*   ;-------------------------------------
*
*   object roof
*
*   name roof
*
*   initroom ShackRoof
*
*   code
*   endcode
*
*   action look, examine, search
*    $tx := "Nothing special about this roof.  Seems"
*    call print
*    $tx := "pretty secure, except for that hole."
*    call print
*   endact
*
*   action get, take, grab
*    $tx := "It's a bit too big to get, and besides,"
*    call print
*    $tx := "everything's nailed down!"
*    call print
*   endact
*
*   endobject
*
*   ;-------------------------------------
*
*   object driftwood
*
*   name driftwood, wood
*
*   adj drift
*
*   initroom ShackRoof
*
*   code
*   endcode
*
*   action look, examine, search
*    $tx := "It's nothing special, just some old"
```

```
*    call print
*    $tx := "driftwood lying below."
*    call print
*   endact
*
*   action get, take, grab
*    $tx := "Leave the driftwood alone!  Remember,"
*    call print
*    $tx := "you are trying to find the magic potion!"
*    call print
*   endact
*
*   endobject
*
*   ;-----------------------------------
*
*   object cork
*
*   name cork, seal, stopper
*
*   initroom unused
*
*   code
*   endcode
*
*   action get, take, grab, remove, pull
*    if player has bottle then
*     if bottle not sealed then
*      $tx := "The bottle is open!"
*      call print
*     elsif player has corkscrew then
*      $tx := "The cork pulls out of the bottle, and"
*      call print
*      $tx := "falls to brittle bits.  A sweet aroma"
*      call print
*      $tx := "eminates from the bottle opening."
*      call print
*      unset bottle, sealed
*     else
*      $tx := "The cork won't come out."
*      call print
*     endif
*    else
*     $tx := "Get the bottle, first."
*     call print
```

```
*    endif
*   endact
*
*   endobject
*
*   ;-----------------------------------
*
*   object tree
*
*   name tree, palm
*
*   adj palm, tall
*
*   initroom ByTree
*
*   code
*   endcode
*
*   action look, examine, search
*    $tx := "It's slick brown bark leads upward to"
*    call print
*    $tx := "the green fronds at the top."
*    call print
*   endact
*
*   action get, take, grab
*    $tx := "Sure, I suppose you intend to pull it up"
*    call print
*    $tx := "by the roots?  No way!"
*    call print
*   endact
*
*   action climb
*    $tx:="You try, but you slide back down."
*    call print
*   endact
*
*   action chop, cut
*    $tx:="You'll need something sharp to do that."
*    call print
*   endact
*
*   endobject
*
```

```
*   ;-------------------------------------
*
*   object shack
*
*   name shack
*
*   initroom ByShack
*
*   code
*   endcode
*
*   action look, examine
*    $tx:="It is a strange old rundown shack.  It"
*    call print
*    $tx := "has no doors or windows!"
*    call print
*   endact
*
*   action get, take, grab
*    $tx := "Since when are you strong enough to pick"
*    call print
*    $tx := "up an old shack?  Don't be silly!"
*    call print
*   endact
*
*   endobject
*
*   ;-------------------------------------
*
*   object ladder1
*
*   name ladder
*
*   initroom unused
*
*   code
*   endcode
*
*   action look, examine
*    $tx:="It looks old and weather-beaten.  But it"
*    call print
*    $tx:="should hold you."
*    call print
*   endact
*
```

```
*   action get, take, grab
*    $tx := "Leave it there, so you can climb back"
*    call print
*    $tx := "down it.  No sense bringing it up here."
*    call print
*   endact
*
*   endobject
*
*   ;-----------------------------------
*
*
*
```

The StartUp.SUB File

```
*
*     ;-------------- StartUp.SUB --------------
*
*     sub StartUp
*
*     TextPalette 0,0,0,0 ; set all pens to black
*     TextPalette 1,0,0,0
*     TextPalette 2,0,0,0
*     TextPalette 3,0,0,0
*
*     scrollbar off ; also prevents front/back gadgets from being
seen
*     menus off ; prevents right mouse button from switching to
text screen
*
*      $filename := "buttons.pic"
*      load screen 2, $filename
*      call LoadingError
*
*      $filename := "scenes.pic"
*      load screen 1, $filename
*      call LoadingError
*
*      $filename := "opening.snd"
*      load sound 0, $filename
*      call LoadingError
*
*      $filename := "window.pic"
*      load screen 0, $filename
*      call LoadingError
*
*      $filename := "artesian.font"
*      load font 0, "artesian.font",8
*      call LoadingError
*      font 0,0 ; use font 0 on screen 0
*
*      $filename := "ocean.snd"
*      load sound 1, $filename
*      call LoadingError
*
*      $filename := "birds.snd"
*      load sound 2, $filename
```

```
*     call LoadingError
*
*     color 0,blue
*     $tx := "     Copyright @1991  by John Olsen"
*     call print
*     call BlankLine
*     $tx := "             THE MAGIC POTION"
*     call print
*     $tx := "              by  John Olsen"
*     call print
*     call BlankLine
*
*     click 0,270,10,283,22,GoNorth
*     click 1,270,37,283,49,GoSouth
*     click 2,285,24,298,36,GoEast
*     click 3,255,24,268,36,GoWest
*     click 4,302,3,315,15,GoUp
*     click 5,302,45,315,57,GoDown
*     click 6,255,68,315,80,Inv
*     click 7,255,86,315,98,Get
*     click 8,255,103,315,115,Help
*     click 9,255,135,315,147,Load
*     click 10,255,152,315,164,Save
*     click 11,255,169,315,181,Quit
*
*     placeobj 1nothing, thisroom
*     grab 1nothing
*     directions ByTree, w
*
*     play sound 0, 0,1,64,0
*     pause 50
*     DOS "CloseScreen title.pic"
*
*     show screen 0
*     ScreenMode graphics
*
*     create screen 24, 100, 100, 5, lores ; for overlays
*     mask 24
*
*    endsub
*
*    ;-------------------------------------
*
*    sub StartUp2
*
```

```
*     set ByTree, started
*     set ByTree, visited
*     $tx := "A glowing green scorpion stings you on"
*     call print
*     $tx := "the ankle, and then skitters off.  You"
*     call print
*     $tx := "hear it's tiny voice laughing, \34 Drink"
*     call print
*     $tx := "the magic potion before it's too late.\34"
*     call print
*     unload sound 0
*     call MainLoop
*
*    endsub
*
*    ;-------------------------------------
*
*    sub LoadingError
*
*    if error > 0 then
*     t \027[32m ; select pen 2
*     TextPalette 2,15,15,15 ; set pen 2 to white
*     t \f
*     t
*     t
*     t
*     t
*     t
*     t
*     t
*     t
*     t
*     t       ERROR: can't load file "@$filename "!  Press RETURN
to abort.
*     t \027[31m ; switch back to pen 1
*     ScreenMode text
*     getstring $temp
*     quit
*    endif
*
*    endsub
*
*    ;-------------------------------------
*
*
```

The MainLoop.SUB File

```
*   ;-------------- MainLoop.SUB --------------
*
*   ; to enable LOAD and SAVE, see lines 153, 161, 170, 171
*
*   sub MainLoop
*
*
;=================================================================
==
*
*   WHILE MainLoop = 0 DO
*
*   call LineFeed
*
*   return := 1
*   TextPosition := 7
*   $sentence := ""
*   $return := "\r"
*   $backspace := "\b"
*
*   color 0,green
*   text 0,7,192,"_" ; character modified to be cursor
*
*   ;-------------------------------------
*
*   while return # 0 do
*
*    getchar $letter
*    length $letter, temp ; temp = 0 if NO letter pressed,
temp = 1 if letter
*    value $letter, val
*    temp := temp * (val < 127)
*    compare $letter, $return, return
*    compare $letter, $backspace, backspace
*
*   if ButtonUsed > 0 and ButtonUsed # 3 then   ; make sure GET
pops up if
*    mode 0, draw                              ; it had been
previously
*    copy 2, 137,26, 197,38, 0, 255, 87         ; selected
before this button
```

```
*  endif
*
*  if ButtonUsed = 1 or ButtonUsed > 3 then
*   return := 0 ; set to zero so as to erase cursor and exit
loop
*   copy 2, x1,y1,x2,y2, 0, x,y ; draw button in down position
*   pause 5 ; to prevent occasional pointer freezes
*    while leftbutton = 1 do    ; wait till button released
*    endwhile
*   y1 := y1 + offset
*   y2 := y2 + offset
*   offset := -13 ; necessary in case player pressed ghosted
direction
*  endif
*
*  if ButtonUsed = 1 then
*   copy 2, x1, y1, x2, y2, 0, x,y ; draw button popping up
*  endif
*
*   if return = 0 then ; return was pressed, so erase cursor
*    color 0,white
*    mode 0, draw
*    rect 0, TextPosition,184, TextPosition + 5,192 ; erase
cursor
*    mode 0,overlay
*   elsif backspace = 0 then ; backspace was pressed
*    length $sentence, sentence
*    if sentence = 0 then ; The sentence has length zero, so
do nothing!
*    else
*     sentence := sentence - 1
*     TextPosition := TextPosition - 6
*     left $sentence, sentence, $sentence
*     color 0,white
*     mode 0,draw
*     rect 0, TextPosition,184, TextPosition + 11,192 ; erase
letter & cursor
*     mode 0,overlay
*     color 0,green
*     text 0,TextPosition,192,"~" ; type cursor
*    endif
*   elsif TextPosition > 240 then ; outside text window
*   elsif temp > 0 then ; accept the keypress
*    $sentence := "@$sentence @$letter" ; add letter to
sentence
```

```
*     color 0,white
*     mode 0, draw
*     rect 0, TextPosition,184, TextPosition + 5,192 ; erase
cursor
*     color 0,green
*     mode 0,overlay
*     text 0,TextPosition,192,"@$letter ~" ; type letter and
cursor
*     TextPosition := TextPosition + 6
*    endif
*
*  if ButtonUsed > 0 and ButtonUsed # 3 then
*    ; blank any previous input typed before typing button
contents
*    mode 0,draw
*    color 0,white
*    rect 0,   7,185,   247,192
*    color 0,green
*    call PrintText ; echo the button name to the text window
*    $sentence := $tx
*    return := 0
*   endif
*
*   readbuttons ; check for mouse button presses between letters
*
*   CountLines := 0
*
*   endwhile ; end of input loop, RETURN has been pressed, ie.
return = 0
*
*   ;-------------------------------------
*
*   compare $sentence, "load", dummy
*   if dummy = 0 then
*    ButtonUsed := 4 ; so that player can click or type LOAD
*    t \027[32m ; switch to pen 2
*    TextPalette 2,15,15,15 ; set pen 2 to white
*    t \f
*    t
*    t
*    t
*    t
*    t
*    t
*    t
```

```
*    t
*    t
*    t                              Please Wait for Loading
*    t \027[31m ; switch back to pen 1
*    x1 := 198
*    y1 := 26
*    x2 := 258
*    y2 := 38
*    x  := 255
*    y  := 136
*   endif
*
*   compare $sentence, "save", dummy
*   if dummy = 0 then
*    ButtonUsed := 5 ; so that player can click or type SAVE
*    t \027[32m ; switch to pen 2
*    TextPalette 2,15,15,15 ; set pen 2 to white
*    t \f
*    t
*    t
*    t
*    t
*    t
*    t
*    t
*    t
*    t
*    t                              Please Wait for Saving
*    t \027[31m ; switch back to pen 1
*    x1 := 259
*    y1 := 0
*    x2 := 319
*    y2 := 12
*    x  := 255
*    y  := 153
*    unload screen 24
*   ; Screenmode text ; switch to black text screen with white
writing
*   endif
*
*   color 0,blue
*
*   compare $sentence, "quit", temp
*   if temp = 0 then
*    quit
```

```
*   elsif ButtonUsed < 4 then ; if enabling LOAD/SAVE, change
line to:  else
*    ghost "@$sentence" turn
*    if error # 0 then
*     $tx := $lasterror
*     call print
*    endif
*   endif
*
*   if ButtonUsed > 3 then
*    $tx := "this feature is disabled in the demo" ; remove to
enable LOAD/SAVE
*    call print                                    ; remove to
enable LOAD/SAVE
*    show screen 0
*    ScreenMode graphics
*    create screen 24, 100, 100, 5, lores ; for overlays
*    mask 24
*    mode 0, draw
*    copy 2,  x1, y1, x2, y2,  0,  x, y ; draw SAVE/LOAD
button popping up
*   endif
*
*   mode 0, draw
*   ButtonUsed := 0
*
*   ENDWHILE ; end of main loop
*
*
;=================================================================
===
*
*   quit
*
*   endsub
*
*   ;-----------------------------------
*
```

The Potion.SUB File

```
*
*     ;--------------- Potion.SUB ---------------
*
*     sub print
*      call LineFeed
*      Call PrintText
*      ReadButtons empty ; empties click queue to ignore clicks
during a print
*     endsub
*
*     ;------------------------------------
*
*     sub LineFeed
*      mode 0,draw
*      copy 0, 7,149, 247,193,  0,  7,140 ; move 5 lines up
*      color 0,white
*      rect 0, 7,185, 247,192 ; blank 6th line
*      color 0, TextColor
*     endsub
*
*     ;------------------------------------
*
*     sub PrintText
*
*      CountLines := CountLines + 1
*
*      if CountLines > MaxLines and defeat = 0 then
*       mode 0, overlay
*       color 0, brown
*       text 0,7,192,"   - press mouse button for more -"
*        while leftbutton = 0 do
*        endwhile
*       color 0,white
*       rect 0,  7,185,  247,192 ; blank 6th line
*       color 0,blue
*       CountLines := 1 ; count 1 because this line is blanked and
used
*      endif
*
*      mode 0, overlay
*      text 0,7,192,"@$tx"
```

```
*
*    endsub
*
*    ;-------------------------------------
*
*    sub BlankLine
*     $tx := ""
*     call print
*    endsub
*
*    ;-------------------------------------
*
*    sub ClearButtons
*     temp := 33 ; clear the buttons for non-movable objects
*    while temp < 43 do
*     remove temp
*     temp := temp + 1
*    endwhile
*    endsub
*
*    ;-------------------------------------
*
*    sub GoNorth
*     offset := GoN
*     x1 := 53
*     y1 := 13
*     x2 := 66
*     y2 := 25
*     x  := 270
*     y  := 11
*     $tx := "n"
*     ButtonUsed := 1
*    endsub
*
*    ;-------------------------------------
*
*    sub GoSouth
*     offset := GoS
*     x1 := 67
*     y1 := 13
*     x2 := 80
*     y2 := 25
*     x  := 270
*     y  := 38
```

```
*      $tx := "s"
*      ButtonUsed := 1
*     endsub
*
*     ;-----------------------------------------
*
*     sub GoEast
*      offset := GoE
*      x1 := 81
*      y1 := 13
*      x2 := 94
*      y2 := 25
*      x  := 285
*      y  := 25
*      $tx := "e"
*      ButtonUsed := 1
*     endsub
*
*     ;-----------------------------------------
*
*     sub GoWest
*      offset := GoW
*      x1 := 95
*      y1 := 13
*      x2 := 108
*      y2 := 25
*      x  := 255
*      y  := 25
*      $tx := "w"
*      ButtonUsed := 1
*     endsub
*
*     ;-----------------------------------------
*
*     sub GoUp
*      offset := GoU
*      x1 := 109
*      y1 := 13
*      x2 := 122
*      y2 := 25
*      x  := 302
*      y  := 4
*      $tx := "u"
*      ButtonUsed := 1
```

```
*     endsub
*
*     ;-----------------------------------------
*
*     sub GoDown
*      offset := GoD
*      x1 := 123
*      y1 := 13
*      x2 := 136
*      y2 := 25
*      x  := 302
*      y  := 46
*      $tx := "d"
*      ButtonUsed := 1
*     endsub
*
*     ;-----------------------------------------
*
*     sub Inv
*      x1 := 137
*      y1 := 13
*      x2 := 197
*      y2 := 25
*      x  := 255
*      y  := 69
*      $tx := "i"
*      ButtonUsed := 1
*     endsub
*
*     ;-----------------------------------------
*
*     sub Get
*      mode 0,draw ; or overlay
*      color 0,white
*      rect 0,  7,185,  247,192 ; erase any previous text input on
line
*      color 0,green
*      x1 := 137
*      y1 := 39
*      x2 := 197
*      y2 := 51
*      x  := 255
*      y  := 87
*      copy 2, x1,y1,x2,y2, 0, x,y ; draw button in down position
```

```
*      $tx := "get ~"
*      TextPosition := 31
*      $sentence := "get "
*      call PrintText
*      ButtonUsed := 3
*     endsub
*
*     ;-------------------------------------
*
*     sub Help
*      x1 := 198
*      y1 := 13
*      x2 := 258
*      y2 := 25
*      x  := 255
*      y  := 104
*      $tx := "help"
*      ButtonUsed := 1
*     endsub
*
*     ;-------------------------------------
*
*     ; to enable LOAD and SAVE, see the file MainLoop.sub
*
*     sub Load
*      x1 := 198
*      y1 := 39
*      x2 := 258
*      y2 := 51
*      x  := 255
*      y  := 136
*      $tx := "load"
*      ButtonUsed := 4
*     endsub
*
*     ;-------------------------------------
*
*     ; to enable LOAD and SAVE, see the file MainLoop.sub
*
*     sub Save
*      x1 := 259
*      y1 := 13
*      x2 := 319
*      y2 := 25
```

```
*      x  := 255
*      y  := 153
*      $tx := "save"
*      ButtonUsed := 5
*     endsub
*
*     ;-----------------------------------
*
*     sub Quit
*      x1 := 259
*      y1 := 39
*      x2 := 319
*      y2 := 51
*      x  := 255
*      y  := 170
*      $tx := "quit"
*      ButtonUsed := 1
*     endsub
*
*     ;-----------------------------------
*
*     sub ReDrawScreen
*
*     if player cango 0 then
*      GoN := -13
*     else
*      GoN := 13
*     endif
*
*     if player cango 1 then
*      GoS := -13
*     else
*      GoS := 13
*     endif
*
*     if player cango 2 then
*      GoE := -13
*     else
*      GoE := 13
*     endif
*
*     if player cango 3 then
*      GoW := -13
*     else
```

```
*      GoW := 13
*     endif
*
*     if player cango 8 then
*      GoU := -13
*     else
*      GoU := 13
*     endif
*
*     if player cango 9 then
*      GoD := -13
*     else
*      GoD := 13
*     endif
*
*      mode 2, draw
*      copy 1, ScreenX,ScreenY, screenX + 244, screenY + 127, 2,
75,73
*      mode 2, overlay
*
*      remove 12 ; remove click zone for ladder
*
*      if ladder in thisroom then
*       if ladder is AgainstTree then
*        copy 2, 0,91, 28,190, 2, 148 + 75, 10 + 73; was 72,199
*        click 12, 148 + 5, 10 + 5, 176 + 5,104 + 5, ClickLadder
*       elsif ladder is AgainstShack then
*        copy 2, 0,91, 28,190, 2, 12 + 75, 26 + 73; was 72,199
*        click 12, 14 + 5, 26 + 5, 36 + 5, 120 + 5, ClickLadder
*       elsif RoomNumber = 1 then ; ladder on ground by tree
*        copy 2, 0,45, 99,72, 2, 140 + 75, 93 + 73
*        click 12, 140 + 5, 93 + 5, 239 + 5, 120 + 5, ClickLadder
*       else ; if RoomNumber = 2 then ; ladder on ground by shack
*        copy 2, 0,45, 99,72, 2, 59 + 75, 94 + 73
*        click 12, 59 + 5, 94 + 5, 158 + 5, 121 +5, ClickLadder
*       endif
*      else
*      endif
*
*      remove 13 ; remove click zone for bottle
*
*      if bottle in thisroom then
*       if RoomNumber = 1 then ; by tree, use small bottle
*        copy 2, 138,54, 148,69, 2, 96 + 75, 97 + 73
*        click 13, 96 + 5, 97 + 5, 106 + 5, 112 + 5, ClickBottle
```

```
*       elsif RoomNumber = 2 then ; by shack, use small bottle
*        copy 2, 138,54, 148,69, 2, 167 + 75, 107 + 73
*        click 13, 167 + 5, 107 + 5, 177 + 5, 122 + 5, ClickBottle
*       elsif RoomNumber = 3 then ; in tree, use big bottle
*        copy 2, 0,0, 23,39, 2, 116 + 75, 52 + 73
*        click 13, 116 + 5, 52 + 5, 139 + 5, 91 + 5, ClickBottle
*       else ; if RoomNumber = 4 then ; shack roof, use little
bottle
*        copy 2, 138,54, 148,69, 2, 143 + 75, 50 + 73
*        click 13, 143 + 5, 50 + 5, 153 + 5, 65 + 5, ClickBottle
*       endif
*      endif
*
*      remove 14 ; remove click zone for corkscrew
*
*      if corkscrew in thisroom then
*       if RoomNumber = 1 then ; by tree, so use small corkscrew
*        copy 2, 151,57, 164,68, 2, 80 + 75, 112 + 73
*        click 14, 80 + 5, 112 + 5, 93 + 5,123 + 5, ClickCorkScrew
*       elsif RoomNumber = 2 then ; by shack, so use small corkscrew
*        copy 2, 151,57, 164,68, 2, 190 + 75, 114 + 73
*        click 14, 190 + 5, 114 + 5, 203 + 5, 125 + 5,
ClickCorkScrew
*       elsif RoomNumber = 3 then ; in treetop, so use large
corkscrew
*        copy 2, 25,0, 50,23, 2, 190 + 75, 93 + 73
*        click 14, 190 + 5, 93 + 5, 216 + 5, 116 + 5,
ClickCorkScrew
*       else ; if RoomNumber = 4 then ; on roof, so use small
corkscrew
*        copy 2, 151,57, 164,68, 2, 77 + 75, 82 + 73
*        click 14, 77 + 5, 82 + 5, 90 + 5, 93 +5, ClickCorkScrew
*       endif
*      endif
*
*      mode 0, draw
*      copy 2, 75,73, 319,199, 0, 5,5 ; draw scenery in window
*      copy 2,  53, 13 + GoN,  66, 25 + GoN, 0, 270,11 ; draw N
button
*      copy 2,  67, 13 + GoS,  80, 25 + GoS, 0, 270,38 ; draw S
button
*      copy 2,  81, 13 + GoE,  94, 25 + GoE, 0, 285,25 ; draw E
button
*      copy 2,  95, 13 + GoW, 108, 25 + GoW, 0, 255,25 ; draw W
button
```

```
*      copy 2, 109, 13 + GoU, 122, 25 + GoU, 0, 302, 4 ; draw U
button
*      copy 2, 123, 13 + GoD, 136, 25 + GoD, 0, 302,46 ; draw D
button
*
*
*     endsub
*
*     ;-------------------------------------
*
*     sub ClickLadder
*      if ButtonUsed = 3 then
*       $tx := "get the ladder"
*       ButtonUsed := 1
*      else
*       $tx := "examine the ladder"
*       ButtonUsed := 2
*      endif
*     endsub
*
*     ;-------------------------------------
*
*     sub ClickBottle
*      if ButtonUsed = 3 then
*       $tx := "get the bottle"
*       ButtonUsed := 1
*      else
*       $tx := "examine the bottle"
*       ButtonUsed := 2
*      endif
*     endsub
*
*     ;-------------------------------------
*
*     sub ClickCorkScrew
*      if ButtonUsed = 3 then
*       $tx := "get the corkscrew"
*       ButtonUsed := 1
*      else
*       $tx := "examine the corkscrew"
*       ButtonUsed := 2
*      endif
*     endsub
*
```

```
*    ;-----------------------------------
*
*    sub AlreadyIs
*     $tx:="It already is!"
*     call print
*    endsub
*
*    ;-----------------------------------
*
*    Sub NoHave
*     $tx:="You don't have it."
*     call print
*    endsub
*
*    ;-----------------------------------
*
*    Sub HaveIt
*     $tx:="You already have it."
*     call print
*    endsub
*
*    ;-----------------------------------
*
*    sub SeeTreeTop
*     if ButtonUsed = 3 then
*      $tx := "get the boughs"
*      ButtonUsed := 1
*     else
*      $tx := "examine the boughs"
*      ButtonUsed := 2
*     endif
*    endsub
*
*    ;-----------------------------------
*
*    sub SeeTreeTrunk
*     if ButtonUsed = 3 then
*      $tx := "get the tree"
*      ButtonUsed := 1
*     else
*      $tx := "examine the tree"
*      ButtonUsed := 2
*     endif
*    endsub
```

```
*
*    ;-----------------------------------
*
*    sub SeeIsland
*     if ButtonUsed = 3 then
*      $tx := "get the island"
*      ButtonUsed := 1
*     else
*      $tx := "look at the island"
*      ButtonUsed := 2
*     endif
*    endsub
*
*    ;-----------------------------------
*
*    sub SeeOcean
*     if ButtonUsed = 3 then
*      $tx := "get the ocean"
*      ButtonUsed := 1
*     else
*      $tx := "look at the ocean"
*      ButtonUsed := 2
*     endif
*    endsub
*
*    ;-----------------------------------
*
*    sub SeeSand
*     if ButtonUsed = 3 then
*      $tx := "get the sand"
*      ButtonUsed := 1
*     else
*      $tx := "examine the sand"
*      ButtonUsed := 2
*     endif
*    endsub
*
*    ;-----------------------------------
*
*    sub SeePlant
*     if ButtonUsed = 3 then
*      $tx := "get the plant"
*      ButtonUsed := 1
*     else
*      $tx := "examine the plant"
```

```
*       ButtonUsed := 2
*      endif
*     endsub
*
*     ;-------------------------------------
*
*     sub SeeDunes
*      if ButtonUsed = 3 then
*       $tx := "get the dunes"
*       ButtonUsed := 1
*      else
*       $tx := "examine the dunes"
*       ButtonUsed := 2
*      endif
*     endsub
*
*     ;-------------------------------------
*
*     sub SeeShack
*      if ButtonUsed = 3 then
*       $tx := "get the shack"
*       ButtonUsed := 1
*      else
*       $tx := "examine the shack"
*       ButtonUsed := 2
*      endif
*     endsub
*
*     ;-------------------------------------
*
*     sub SeeSky
*      if ButtonUsed = 3 then
*       $tx := "get the sky"
*       ButtonUsed := 1
*      else
*       $tx := "examine the sky"
*       ButtonUsed := 2
*      endif
*     endsub
*
*     ;-------------------------------------
*
*     sub SeeSun
*      if ButtonUsed = 3 then
*       $tx := "get the sun"
```

```
*       ButtonUsed := 1
*      else
*       $tx := "look into the sun"
*       ButtonUsed := 2
*      endif
*     endsub
*
*     ;-------------------------------------
*
*     sub SeeLadder
*      if ButtonUsed = 3 then
*       $tx := "get the ladder"
*       ButtonUsed := 1
*      else
*       $tx := "examine the ladder"
*       ButtonUsed := 2
*      endif
*     endsub
*
*     ;-------------------------------------
*
*     sub SeeHole
*      if ButtonUsed = 3 then
*       $tx := "get the hole"
*       ButtonUsed := 1
*      else
*       $tx := "look into the hole"
*       ButtonUsed := 2
*      endif
*     endsub
*
*     ;-------------------------------------
*
*     sub SeeRoof
*      if ButtonUsed = 3 then
*       $tx := "get the roof"
*       ButtonUsed := 1
*      else
*       $tx := "examine the roof"
*       ButtonUsed := 2
*      endif
*     endsub
*
*     ;-------------------------------------
*
```

```
*     sub SeeDriftwood
*      if ButtonUsed = 3 then
*       $tx := "get the driftwood"
*       ButtonUsed := 1
*      else
*       $tx := "look at the driftwood"
*       ButtonUsed := 2
*      endif
*     endsub
*
*     ;-------------------------------------
*
*     sub DrinkBottle
*      if bottle is sealed then
*       $tx := "The liquid is corked securely inside"
*       call print
*       $tx := "the bottle."
*       call print
*      else
*       defeat := 1
*       $tx := "The thick sweet liquid burns your throat"
*       call print
*       $tx := "but quickly takes effect.  You feel"
*       call print
*       $tx := "wonderful!  The magic potion has worked!"
*       call print
*       call BlankLine
*       $tx := " Congratulations on finishing this demo!"
*       call print
*       TextColor := brown
*       $tx := "      Press mouse button to exit."
*       call print
*        while leftbutton = 0 do
*        endwhile
*       MainLoop := 1
*      endif
*
*     endsub
*
*     ;-------------------------------------
*
```

```
*
*     ;---------- Potion.voc ----------
*
*     VOCAB
*
*     ;-------------------------------------
*
*     action FastMem, FastRam, ChipMem, ChipRam, FreeMem, Avail,
Mem, Memory
*      $tx := "@FastMem  Kbytes FastMem   @ChipMem  Kbytes ChipMem"
*      call print
*     endact
*
*     ;-------------------------------------
*
*     action Cycle
*      Show Screen 1 ; location scenery
*      While LeftButton = 1 do ; waits to make sure mouse button
is down
*      endwhile
*      While LeftButton = 0 do ; waits to make sure mouse button
is released
*      endwhile
*      Show Screen 2 ; button.pic
*      While LeftButton = 1 do ; waits to make sure mouse button
is down
*      endwhile
*      While LeftButton = 0 do ; waits to make sure mouse button
is released
*      endwhile
*      Show Screen 24 ; overlay buffer
*      While LeftButton = 1 do ; waits to make sure mouse button
is down
*      endwhile
*      While LeftButton = 0 do ; waits to make sure mouse button
is released
*      endwhile
*      Show Screen 0
*     endact
*
*     ;-------------------------------------
*
```

```
*     action help, hint, clue, give me clue, give me help, give me
hint
*      ;          ----------------------
*      $tx := "Type sentences, or use buttons instead."
*      call print
*      $tx := "Examples: to examine the tree, either"
*      call print
*      $tx := "type LOOK AT THE TREE or just click on"
*      call print
*      $tx := "the picture of the tree.  To pick up the"
*      call print
*      $tx := "ladder, either type TAKE THE LADDER or"
*      call print
*      $tx := "click on the GET button and then click"
*      call print
*      $tx := "on the picture of the ladder.  To drop"
*      call print
*      $tx := "the ladder, do not use the buttons."
*      call print
*      $tx := "You must type DROP THE LADDER.  You may"
*      call print
*      $tx := "also PUT THE LADDER AGAINST THE SHACK,"
*      call print
*      $tx := "etc.  You can go west by typing GO WEST"
*      call print
*      $tx := "or by clicking on the W button on the"
*      call print
*      $tx := "compass.  To save a game in progress,"
*      call print
*      $tx := "type SAVE or click on the SAVE button."
*      call print
*      $tx := "To start a game that was saved, type"
*      call print
*      $tx := "LOAD or click on the LOAD button."
*      call print
*      $tx := "Remember, the buttons are only a very"
*      call print
*      $tx := "few things that this adventure allows."
*      call print
*      $tx := "Use your imagination, and type anything"
*      call print
*      $tx := "you want to do, and the computer will"
*      call print
*      $tx := "respond.  Enjoy the adventure, and buy"
*      call print
```

```
*      $tx := "Visionary from OXXI/Aegis so you can"
*      call print
*      $tx := "write your own adventures.  They can be"
*      call print
*      $tx := "pure text, pure graphics, or (like this)"
*      call print
*      $tx := "a hybrid combination."
*      call print
*     endact
*
*     ;-------------------------------------
*
*     action drink, drink out of bottle
*
*     if player has bottle then
*      call DrinkBottle
*     else
*      call NoHave
*     endif
*
*     endact
*
*     ;-------------------------------------
*
*     action author author, author
*      $tx:="written 5/5/91 by John Olsen"
*      call print
*      $tx:="P.O. Box 181, Newberg, OR 97132"
*      call print
*     endact
*
*     ;-------------------------------------
*
*     action quit
*      $tx := "quit"
*      call print
*     endact
*
*     ;-------------------------------------
*
*     action save, save game, save position, store, store game
*      ghost "save SaveGame"
*      $tx := "OK.  Saved."
*      call print
*     endact
```

```
*
*     ;-------------------------------------
*
*     action load, load game, restore, restore game, restore
position
*      ghost "load SaveGame"
*     endact
*
*     ;-------------------------------------
*
*     ENDVOCAB
*
*
```

Appendix D: ASCII Codes for Visionary

Dec	Oct	Hex	Character
0	0	0	^@ NUL
1	1	1	^A SOH
2	2	2	^B STX
3	3	3	^C ETX
4	4	4	^D EOT
5	5	5	^E ENQ
6	6	6	^F ACK
7	7	7	^G BEL
8	10	8	^H BS
9	11	9	^I HT
10	12	A	^J LF
11	13	B	^K VT
12	14	C	^L FF
13	15	D	^M CR
14	16	E	^N SO
15	17	F	^O SIR
16	20	10	^P DLE
17	21	11	^Q DC1
18	22	12	^R DC2
19	23	13	^S DC3
20	24	14	^T DC4
21	25	15	^U NAK
22	26	16	^V SYN
23	27	17	^W ETB
24	30	18	^X CAN
25	31	19	^Y EM
26	32	1A	^Z SUB
27	33	1B	^[ESC
28	34	1C	^\ FS
29	35	1D	^] GS
30	36	1E	^^ RS
31	37	1F	^_ US
32	40	20	Space
33	41	21	!
34	42	22	"
35	43	23	#
36	44	24	$
37	45	25	%
38	46	26	&
39	47	27	'

Dec	Oct	Hex	Character
40	50	28	(
41	51	29	)
42	52	2A	*
43	53	2B	+
44	54	2C	,
45	55	2D	-
46	56	2E	.
47	57	2F	/
48	60	30	0
49	61	31	1
50	62	32	2
51	63	33	3
52	64	34	4
53	65	35	5
54	66	36	6
55	67	37	7
56	70	38	8
57	71	39	9
58	72	3A	:
59	73	3B	;
60	74	3C	<
61	75	3D	=
62	76	3E	>
63	77	3F	?
64	100	40	@
65	101	41	A
66	102	42	B
67	103	43	C
68	104	44	D
69	105	45	E
70	106	46	F
71	107	47	G
72	110	48	H
73	111	49	+I
74	112	4A	J
75	113	4B	K
76	114	4C	L
77	115	4D	M
78	116	4E	N
79	117	4F	O

Dec	Oct	Hex	Character
80	120	50	P
81	121	51	Q
82	122	52	R
83	123	53	S
84	124	54	T
85	125	55	U
86	126	56	V
87	127	57	W
88	130	58	X
89	131	59	Y
90	132	5A	Z
91	133	5B	[
92	134	5C	\
93	135	5D	]
94	136	5E	^
95	137	5F	_
96	140	60	`
97	141	61	a
98	142	62	b
99	143	63	c
100	144	64	d
101	145	65	e
102	146	66	f
103	147	67	g
104	150	68	h
105	151	69	i
106	152	6A	j
107	153	6B	k
108	154	6C	l
109	155	6D	m
110	156	6E	n
111	157	6F	o

Dec	Oct	Hex	Character
112	160	70	p
113	161	71	q
114	162	72	r
115	163	73	s
116	164	74	t
117	165	75	u
118	166	76	v
119	167	77	w
120	170	78	x
121	171	79	y
122	172	7A	z
123	173	7B	{
124	174	7C	|
125	175	7D	}
126	176	7E	~
127	177	7F	DEL
128	200	80	[CursorUp]
129	201	81	[CursorDown]
130	202	82	[Help]
131	203	83	[CursorLeft]
132	204	84	[CursorRight]
133	205	85	[F1]
134	206	86	[F2]
135	207	87	[F3]
136	210	88	[F4]
137	211	89	[F5]
138	212	8A	[F6]
139	213	8B	[F7]
140	214	8C	[F8]
141	215	8D	[F9]
142	216	8E	[F10]

Appendix E: Technical Support

1 Before you call for technical support for *Visionary*, first try to resolve the problem on your own. Decide exactly what the problem is—does it arise in compiling, loading graphics or sound/song files, or are you simply having problems getting your game to do some particular task? Check the solutions given in Appendix A if you have a VCOMP or DBUG error.

2 Okay, you're at the end of your rope, and nothing you try seems to solve your problem.. Time to call for some technical support. Get your pen and paper handy, look up your serial number and get clear in your mind a brief description of the problem. **IF YOU HAVE NOT REGISTERED YOUR PROGRAM, YOU WILL NOT BE ELIGIBLE FOR TECHNICAL SUPPORT.**

You may want to write your serial number here in the manual for reference:

Serial Number: _____

3 **IF YOU ARE CALLING FROM THE U.S.:**
In order to efficiently provide technical support to you when you need it, **without** increasing the cost of Aegis software, we have contracted with **Computer Technical Support Services** to answer your technical support questions.

1-900-776-6994

The call costs **$1.00** for the first minute, and **$2.00** for each additional minute.

Be prepared to briefly state your technical support problem, and note the answer or solution the technician gives you.

Programming problems often require some research or testing—instead of asking you to wait on line, you may be asked to call back for the answer or solution to your problem.

If you do not have access to 900 lines, you can still use mail or FAX to get technical support.

4 **IF YOU ARE CALLING FROM OUTSIDE THE U.S.:**
Please call Oxxi Customer Service for technical support if you are calling from outside the U.S. The phone number is:

213-427-1227

Be prepared to briefly state your technical support problem, and note the answer or solution the technician gives you.

5 **TECHNICAL SUPPORT BY FAX:**
To receive technical support by FAX, you can FAX or mail us a brief description of your technical support problem. Include at least the following information:

Your program serial number
Your FAX or phone number
The problem listing or section of code

You can mail this information to the address below, or FAX it to:

213-427-0971

6 All correspondence regarding *Visionary* should be addressed:

Oxxi, Inc.
Visionary Support
P O Box 90309
Long Beach, CA 90808-0309
USA

Resources

Book of Adventure Games, The, 2 volumes, Kim Schuette, Arrays Inc., 1984.

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.

Quest for Clues, 3 volumes, Shay Adamms, Origin Systems Inc., 1988.

Visionary Programmer's Handbook, The, John Olsen, Oxxi, Inc, 1991.

MAGAZINES:

Enchanted Realms, Digital Expressions, P.O. Box 33656, Cleveland, OH 44133.

Questbusters, P.O. Box 5845, Tucson, AZ 85703.

Index

Post Office Box 90309
Long Beach, CA 90809-0309
USA
Phone (213) 427-1227
Fax (213) 427-0971

ISBN 0-938385-21-6