S.C.U.M.M. Tutorial

"Rising to the top of the Software Cesspool"

Wallace Poulter

Circa 1991

Table of Contents

PREFACE

Scumm stands for Script Creation Utility for Maniac Mansion.
Originally first developed for the game Maniac Mansion, in 1987, the Scumm system has developed into the mainstay of the company's product line. Somewhat appropriately for a system developed by Lucasfilm, Scumm uses concepts such as Actors, Costumes, Cameras, Sound Effects, and Scripts.

Scumm was originally designed for the C64 by Ron Gilbert. While we program on an IBM clone, Scumm is not tied to any particular hardware and Scumm programs will run with little change on other machines.

All of the utilities that we use in Scumm are named after various disgusting bodily fluids such as flem, byle, and mmucus. This can lead to interesting conversations at restaurants. Asking a co-worker if they have "flemed" something yet and receiving the reply that they are awaiting it to be "byled."

It is possible to do almost anything in Scumm, although not always elegantly.

As with all endeavors, the contribution of others has helped bring this manual to fruition.

Thanks are in order to Ron Gilbert, David Fox, and Aric Wilmunder, for constructive comments and information.

Also thanks to the Scummlettes: Ron Baldwin, Jenny Sward, Dave Grossman, Tim Schafer; and the Scumm Babies; Sean Clark, Mike Stemmle, Tony Hsieh and Tami Borowick.

Finally thanks to Noah Falstein, Hal Barwood, and Kalani Streicher for their help during Scumm-U and beyond.

Chapter 1 Introduction

A long time ago in a Galaxy far far away, Ron Gilbert created the scumm system on a Commodore 64. Despite this the Scumm System has grown to be the mainstay of the Lucasfilm game line.

Scumm is an acronym and stands for Script Creation Utility for Maniac Mansion.

Scumm was developed to alleviate designers from the mundane task of programming and allow them to concentrate on the game's creative design rather than its technical issues. As such each Scumm statement launches a multitude of C and assembly code that is of little concern to the Scumm programmer.

While much has changed internally with the system since Maniac Mansion, Scumm has also evolved graphically from the point n click interface that Lucasfilm pioneered with Maniac Mansion.

Zak MacKraken (scumm system 2.0) was the first IBM based system and allowed easier animation.

Indiana Jones (scumm system 3.0) added improved costumes, improved character walking animation and special case animation.

Loom (scumm system 3.5) added the ability to place text on the main screen.

The Secret of Monkey Island I (scumm system 4.0) added the ability to have the actors scale.

Before starting to understand Scumm we recommend that you play a number of the Scumm system products to get a good feeling for the style and ambiance.

This manual is meant to help the reader learn how to program in Scumm. It contains a tutorial to get the new Scumm programmer started as soon as possible.

The manual is organized as follows.

Chapter One: Scumm Introduction
A brief look at the history and changes of the system over the first 5 Scumm games: Maniac Mansion, Zak McKracken, Indiana Jones, Loom, and The Secret of Monkey Island.

Chapter Two: Tutorial
The structure of Scumm. Multi-tasking. The interface, adding objects, verbs. An introduction to using flem, byle, brief, and windex.

Chapter Three: Statements
Listing of all the statements in Scumm. Includes a description, syntax, side effects, and examples.

Chapter Four: Function
Listing of all the functions in Scumm. Includes a description, syntax, side effects,

and examples.

Chapter Five: System Variables
Listing of all the System Variables in Scumm. Includes a description, syntax and any side effects.

Chapter Six: Errors
A look at common mistakes, non-fatal errors, trouble shooting, and messages to ignore.

Chapter Seven: Macros
How to create and use these very effective routines.

Chapter Eight: Tools/Utilities
What you need to know about flem, byle, windex, and mmucus from the Scumm programmer's perspective.

Chapter Nine: Quick Study Guide
How to create a room, actor, costume, global variable, local variable etc.

Chapter Ten: Byle (In depth)
A look at our animation editor.

Chapter Eleven: Explanation of Headers
A look at the 4 headers, Main-scripts.scu, etc.

Chapter Twelve: Brief
A look at this commercial text editor with Scumm.

Appendix A: Definition of Terms

Appendix B: Structure

Appendix C: Reserved Words

Appendix D: List of Classes

Appendix E: Directories

Appendix F: System Variable Slot Order

Appendix G Latest Boot Script

Appendix H: Project Specific Information (Monkey I)

Index
A complete index including reserved words in alphabetical and group listings, cross references of usage, glossary of terms, and a quick study ref guide.

Chapter 2 Tutorial

2.1.0 Introduction

As befitting a company with the name Lucasfilm, the Scumm system involves such concepts as Actors, Costumes, Cameras and Scripts.

The basic graphic concept in Scumm is the "room." Each screen that the user sees is such a "room." This can be a room in the traditional sense such as Indy's office at Barnett College (in the Indiana Jones: Graphic Adventure Game), but it can also be the exterior of Barnett College or the different levels within the Zeppelin maze.
In essence, Scumm system products are made up of nothing more than multiple rooms. Indiana Jones and Loom have 112 and 97 rooms respectively.

A room is actually just a group of object definitions. Stored along with these definitions are a number of sounds, actors, and the graphic room data.

2.1.1 Why Scumm?

Why Scumm is usually the first question that is asked by a new Scumm programmer. Why didn't we use C or some other high level language, and what advantages does Scumm give us in doing these projects. There are a number of reasons.

Scumm does a lot of very useful memory management for the Scumm programmer. If, during the running of a Scumm program, the programmer needs a particular item, such as a costume for an actor, a room, or a script, just use it and the Scumm system accesses the disk for you and brings it into memory. The Scumm memory management system will throw items out of memory that are not in use or are very old. The system will just remove the items so the Scumm programmer does not have to worry about whether it is in memory. There are, of course, certain rules and procedures that we will address later. The system is smart enough that the Scumm programmer does not have to worry about on which disk the item resides. The Scumm system will access the appropriate disk and retrieve all the necessary items and place it in memory. This is something that C would be only be able to do with overlays. These Scumm games are at least 2.5 - 5 megabytes of code and graphics - too much to fit into memory at

once. That is a significantly sized C program to have sitting around. Scumm's ability to interchange all of the elements in and out is a really big advantage.

Most significantly we use the Scumm system because of its multi-tasking abilities. Scumm can run several sections of code. The way that large computers, such as Unix, multi-task is that they are not actually running everything at once there is only one microprocessor. In reality it is time-slicing, sharing stuff. The way this works is the time slice is built off so many jiffies (a sixtieth of a second), or so many milli seconds runs of each processor.

In Scumm that actually doesn't work well because you are setting up graphics. It would be a problem if the system pulled away from a script when the room is half rendered. This would lead to a screen coming up with half the objects rendered and the next frame it would render the second half. As the Scumm programmer you have to control when the program time slices to the next script.

The system will run all of the instructions it finds in order until it finds a break. Then the system will time slice over to the next script. This script runs until it hits a break and it time slices to the next script and so on until all the scripts have been run. Once it reaches this state the system generates a video screen, and starts all over again.

This is something that we guarantee is really going to cause lots of problems. It always does. If you have programmed in C or PASCAL you are accustom to thinking about code going one instruction after another. When you call a sub-routine, in C or PASCAL, the program goes to the sub-routine, runs the sub-routine, and when the sub-routine is complete, the program returns.

In the diagram below, the C program would run down till it hit the sub-routine. It would then run the entire sub-routine, return from whence it came and continue on.

What Scumm does is it gets to the sub-routine branch and then, both branches of, the original and the sub-routine, will start to run simultaneously.

If the first sub-routine kicks off another sub routine then the program would have 3 pieces of code moving almost simultaneously. This is a new way of thinking about programming and it does take a little getting used to.

When the Scumm programmer writes their first Scumm scripts / (code), they will

almost invariably produce giant Scumm scripts. This one script will be attempting to control multiple items such as a clock, dripping water and a window shade. What the Scumm programmer really has to do is take each discrete item within a room and give the item its own little script. The clock should have a script that just worries about making the pendulum go back and forth and making the appropriate sound. The script once activated runs all on its own and the Scumm programmer never has to worry about it again, so long as you never leave the room. Another script would contain the code to activate the dripping of water from the roof. So more than any other reason multi-tasking is why we use Scumm.

The commands in Scumm have been custom created for our Graphic Adventure type games. These are very high-level commands. When the actors walk around the room the Scumm programmer does not have to worry about the walk animation, or worry about the actors moving their arms. A click of the cursor on an X and Y position and the actor walks there. The camera will follow the actor and scroll the room. The actors walk around chairs and up and down stairs and the Scumm programmer does not have to be concerned with the details.

2.1.2 Structure of Scumm

This is the structure of a typical room file. It includes a number of scripts followed by a room definition.

```
[include "filename"]
[script script-name {
        [statements]
}]
define variable-name=value
room "room-name" room-name {
[define first-script-in-room = max-scripts]
define script
[script script-name {
        [statements]
}]
        sounds {
                "SFX/sound-name" sound-name
        }
        costumes {
                "costumes\actor-name" actor-costume-name
        }
        enter {
                [statements]
        }
        exit {
```

```
        [statements]
    }
    [order {
        [object-name]
        [object-names]
    }]

    object real-object-name {
        name is "[visual-object-name]"
        [dependent-on object-name being object-state]
        [class is class-state [class-state]]


        [verb verb-name|default{
            [statements]
        }]
    }
}
```

To help us with the learning of the Scumm system we have enlisted the aid of sam and max. sam and max, a large Dog and Rabbit, are free lance police in an off-the-wall comic book drawn by one of our artists Steve Purcell. Steve was kind enough to draw a number of characters, objects and rooms for use in Scumm-U.

Make sure you are inside the Scumm-U directory, if not,

v Type cd\scummu

v Type mm

+ Press

_

This will run the program that is already within Scumm-U. What you should see is sam, the large dog, in the office of sam and max. Using the cursor you can click around the room and walk sam around. This is a really simple room. There are no objects wired up, except for the door. Until we do so there is nothing to touch. At the moment we can only touch the door. We are unable to open, close or do anything else with the door. The first task in Scumm-U is to add additional "wiring" to the door.

To get out of the program

+ Press

_

To bring up the code for the office of sam and max

v Type b office.scu

This is the Scumm script for the office. The script is reproduced below.

```
include "header.scu"
include "system.scu"
include "main-scripts.scu"
include "boot.scu"

room "office" office {

	sounds {
		"SFX/doorclos" door-close
		"SFX/dooropen" door-open
	}

	costumes {
		"costumes\sam" sam-skin
	}

	enter {
	}

	exit {
	}

	object front-door {
		name is "front door"

		verb walk-to {
		}
	}
}
```

2.1.3 Structure Explanation

The top 4 lines are "includes",

```
include "header.scu"
include "system.scu"
include "main-scripts.scu"
include "boot.scu"
```

These are including other files, the header, system, boot, and main-scripts. The four includes are the scripts that control the user interface. The entire user interface, the verbs (see xyz), clicking on, the sentence construction is all written in Scumm. That's why Loom has a radically different user interface than Indiana Jones. We just wrote a different interface for it. These are all at the system level and we will get into that later.

```
rooms office "office"
```

This defines the "room" called office. The word in quotes is its file name. "Rooms" are not always traditional rooms such as the office. The Zeppelin maze, the College Exterior (both in Indiana Jones), and the Lucasfilm logo are examples of other forms of "rooms."

```
sounds {
        "SFX/doorclos" door-close
        "SFX/dooropen" door-open
}
```

The sounds structure defines the sound effects that are included in this room. This is another thing that trips people up. The doorclos sound effect is enclosed in the office but you can use the sound effect anywhere. If the actor is in another room, for example, the police station, you can still use the close door sound effect. If the sound is not in memory, the Scumm system will retrieve it from disk no matter what room is showing on the screen.

You may be wondering why would we associate the sounds with different rooms rather than simply having one room with all the sounds in it.

The reason is because of floppy disks. We try to place sounds in the rooms that they will be used in to avoid multiple disk swops. The product will get shipped on 6,8,10, floppy disks, whatever it happens to be in the future. For example, room 13 is sitting on floppy disk 1 and defined within that room 13 is the gunshot sound effect. The game is currently playing the police station which is on floppy disk 4 and the player uses the gun. The Scumm system will say "ah a gunshot" and will produce the message "please insert disk one." Loading the sound effect at this point really screws the whole dramatic timing of everything so you try to group the sound effects with the appropriate rooms.

The Scumm system was developed, for Maniac Mansion, on a C64. The disk on a C64 is not much faster than tape and having the sound effects in close proximity to the room was important because the head only had to seek a little way to access the new item. This is something that may change as technology progresses. When we produce hard disk only versions in the future then all sounds could go in one file.

```
costumes {
        "costumes\sam" sam-skin
}
```

Costumes work exactly the same way as sounds. Here we are defining the costume of sam as sam-skin and just as a naming convention we usually call them skins. We tack skin on the end, to differentiate because there is an actor called sam and then we have a costume called sam-skin. The Secret of Monkey Island contains some exceptions. The Rat costume uses the addition "fur" for example.

Let's explain the difference between actors and costumes.

An actor is a person who walks around the room. The actor can hold things in an inventory. A costume is what the actor looks like. An actor can look like anything by changing the costume to sam, max, indy, a nazi, anything you want just by changing the actor's costume. Think of it as the actor is a skeleton and a costume is the skin. An actor can have no costume in which case the actor is invisible. However the actor still walks around and can have an inventory etc.

```
enter {
}

exit {
}
```

You have probably noticed that in playing a Scumm game, when you walk from a room, they usually the room irises down (sometimes we just cut to black) and then the new room irises up. Before a new room irises up it runs the enter code and just after the room irises down it runs the exit code. These are in place so you can do some setup. The enter code can check to see situations going on in the game and get the room all set up the way it needs to be before it irises up. Similarly the exit code can be used to terminate items such as continuous music.

```
object front-door {
        name is "front door"
```

```
                verb walk-to {
                }
        }
}
```

The string in quotes is what the player sees when they pass the cursor over the object. The Scumm programmer can change the name as the program is running so it could be "broken front door". Later on we will look at a command that allows you to change that. (see page xyz)

Within the object is a verb "walk" which, in this example contains no code. All objects are made up of verbs. The verbs can be literally anything the Scumm programmer wants. A maximum of 255 verbs can be associated with any given object. The verbs open, close, push, pull, etc. are what we decided to call the verbs for Scumm-U and Indiana Jones.

2.1.4 The Interface

The interface is such that when you click on the open in the interface and then click on the front-door object, the system runs the open verb for the object door. The Scumm programmer has to put the code that opens up the door inside the open verb for the door. Similarly with walk-to. Clicking on walk-to door activates the code within the verb walk-to of the object door. The Scumm programmer is responsible for taking the actor through the door and out into the new room. There is no big matrix that connects rooms together. It is the verb walk-to code of an object that tells it to which room it is connected. With two different rooms, the verb walk-to code for the two different doors, that are to connect, point to each other. They are two separate doors(objects), not 1 object.

2.2.1 The say-line command (see 5.7.3)

The say-line command is used to print out an actor's spoken message to the screen. Say-line causes the speaker's mouth to animate. The color of the message can be optionally specified (see actor statement page xyz.)

```
syntax:         say-line [actor-name] "string"
example:    say-line Indy           "Good Morning!"
```

Position your cursor on the line right after the verb walk-to.

```
                object front-door {
                        name is "front-door"
```

```
                verb walk-to { - cursor here
                }
        }
}
```

+ Press

_

This should create a new line.

v Type say-line "The door is closed."

The code should look as follows.

```
object front-door {
        name is "front-door"

        verb walk-to {
                say-line "the door is closed"
        }
}
}
```

Exit into DOS

+ Press

_

This following stands for make and the system will compile the office.

v Type mk.

To bring the room up.

v Type mm

Walk to the door and the line that you typed should appear.

2.2.2 New Object bat

Go into the office.scu again. Type b office.scu. Move the cursor down below the

brace that closes the objects (next to last brace)

```
        object front-door {
                name is "front-door"

                verb walk-to {
                }
        } <- cursor here
}
```

Open up a new line

```
        +       Press
```

_

Add the object called Bat.

```
        v       Type

        object Bat {
                name is "bat"
        }
```

2.2.3 New Verb look-at

Insert the cursor within the object bat after the name is "bat".

```
        object Bat {
                name is "bat" - cursor here
        }

        +       Press
```

_

Your code should look like this

```
        object Bat {
                name is "bat" - cursor here

                verb look-at {
                        say-line "it is a bat"
```

```
        }
}
```

You should end up with 3 close braces at the bottom. This is because the first brace closes the verb, the second brace closes the object and the third brace, that was already there, closes the room.

Scumm is not as good as C. In C you can write an entire program on one line. Scumm requires carriage returns at the end of lines but how things are tabbed does not matter. It is not possible to put open brace, an instruction and a closed brace on the same line as you do in C. Scumm has to break it up.

```
        {say-line "it is a bat"}
```

This is wrong.

```
        {
say-line "it is a bat"
}
```

This is

Exit out

```
+       Press
```

_

but don't mk yet There is another step to do first which is we to define the object. The system has to know where the object is and to do that we have a program called flem.

2.2.4 Set State / Set Use

type flem

Flem should come up with a screen that says office.
Click on the name office once and then click on the box edit objects. The office will appear on the screen.

Pull down the "objects" menu and click on new object. This will automatically give the Scumm programmer the number of the new object. The object numbers start at 50 and the front-door has already been assigned that number. The bat will be object number 51. Click on "ok" and type in the name "bat". Again click on "ok.

Move the cursor in place over the picture of the"bat." Hold down the left mouse button and draw a rectangle around the bat. Release the button and click once to get a sound. This has defined that area for the object bat. The "set state" button

should be on the right hand side of the screen. Click with the left mouse button and the button will change to "set use." Moving the cursor onto the screen and you will find that the cursor is a cross shape. By clicking with this cross the Scumm programmer can set the position of the actor when he interacts with the object. Decide where you want sam to stand when he is looking at the "bat". Preferably off to one side of it. Press the left mouse button to set this "use position."

Below the "set state/set use" button are 4 facing options. These are Left, Right, Back and Forward. When the actor is standing at the "use position" of the object he will face whichever direction that the Scumm programmer has chosen as appropriate. Click on whichever one you wish, in relation to where you chose the "use position".

Finally pull down the file menu. Select "quit" and reply yes to the "Do you want to save objects before quitting" prompt. This exits you back into DOS.

v Type mkall

This will "make" (compile) the files. However not all the files, only those that have been touched.The system is smart enough to look at the dates and the system only compiles what has been changed.

An interesting point, the two bats are different.

```
object Bat {
        name is "bat"
```

The first bat (Bat) is the objects name. This is how the system recognizes the bat. This must be the same as the object and in flem. The second bat ("bat") is what is printed on the screen. This could be wooden bat, broken bat, small bat, aluminum bat it doesn't matter.

Once the program has compiled type mm. When the cursor scans over the bat you should see the name "bat". Click on look-at, followed by click on bat. The actor sam will walk over to the bat, stand at the use position you defined. sam will face the direction that was designated and will say the line of dialog.

What the Scumm system will do when the user activates look-at bat is the system finds the object bat, walks the actor over to the object, turns the actor to face the right direction and then, and only then, the system runs the code. The Scumm programmer can be guaranteed that the actor will be standing in the exact position with the correct facing before the subsequent code is run.

This can be important if the Scumm programmer is going to do some weird special case animation with the actor, because usually the actors need to be standing in just the right place before that can be done. There are special kinds of objects that can be created where the code gets run immediately. The actor does not actually have to walk to the object. In Monkey Island there is a clock. Using the command look-at clock will run the code immediately without the actor actually walking over there. That's a different kind of object and we will get to that later.

2.2.5 Adding a more complicated object front-door

A more complicated object is the door This object is going to open and close which is changing graphic states.

Exit out and go into flem. Go into office edit objects, click on the word front-door.

As you move the cursor around the room you should get a rectangle. That is the size of the rectangle for the door. At the bottom is a slider bar. Slide this bar all the way over to the right. The object part of the room is the area encompassed by the purple screen. These are all the the second states of objects that the artists have drawn in the room.

Move the slide bar over about half way between normal room and the "objects" portion of the room. In the very left hand corner of the objects section is a door that is open. Move the rectangle on top of graphic open door and click once. You should hear a beep. What we have done is defined what is called state 1 for the object. All objects in Scumm have 2 states, state 0 and state 1. There are only two states per object. State 0 is what the artist drew into the background, in this instance the closed door. State 1 is the open door. You have just defined state 1 so now the door has 2 graphic states.

Something to notice on the screen is front-door. It says 50 front-door 2880-447. This is the amount of memory that the door took up. It originally took up 2880 bytes and the system crunched it down to 477 bytes. It is important to realize that one image of the door is .5k. It is important because when you are doing things with graphics, graphics are everything. Text is relatively free. You can have 477 letters of a text block and it would take up the same space as the door.

2.2.6 New Verb open

Create a new verb called open which should be done under walk-to.

```
object front-door {
```

```
        name is "front-door"

        verb walk-to {
        } - cursor here
    }
}
```

The order of verbs within a object is not important.
It should look like this,

```
    verb open {
    }
```

2.2.7 The state-of command

Sets the state of the object. Only one state can be set within each state-of command.

syntax: state-of [object-name] is [STATE]
example: state-of window is OPEN

Place a state-of command within the verb open.

```
    v    Type state-of front-door is OPEN.
```

Your code should look like this.

```
    object front-door {
        name is "front-door"

        verb walk-to {
            say-line "The door is closed."
        }

        verb open {
            state-of front-door is OPEN
        }
    }
```

Exit out and type mkall.

The reason we have to make all (mkall) is because we defined an image, in flem, and the system must go back. Once the program has compiled

v Type mm.

Click on open and click on the door. You will notice something weird happens. That is because we didn't quite define the object right. When we clicked on door it wasn't in the right position. so alt-x out go back into flem. now the one you want to move down is not the image but the door. Click on the word front-door and move the rectangle one square down.

Exit out and

type mkall.

If there is still a problem go back into flem. Move vertical slider bar down and put the rectangle down another one position. Put the cursor on the word front-door and press the right hand mouse button. This is how you can check your states by toggling back and forth.

2.2.8 New Verb Close

Add a new verb called close underneath the open verb. Type state-of front-door is CLOSED within this verb. This should be what your code looks like.

```
        verb open {
                state-of front-door is OPEN
        }

        verb close {
                state-of front-door is CLOSED
        }
}
```

Run a mk and once the program has compiled

type mm.
Check you work.

2.2.9 The if command

There are two forms of the if statement. One with else and one without. These are pretty self-explanatory to anyone who knows any other structured language. There are restrictions in that complex conditions using OR or AND can not yet be constructed.

```
syntax:          if (exp16 = val16) {
                         [statements]
                 }

                 or

                 if (exp16 = val16) {
                         [statements]
                 }else{
                         [statements]
                 }

example:    if (class-of window is LOCKED) {
                         say-line "It's latched from the other side."
                 }

                 if (class-of me is UNLOCKED) {
                         say-line "It looks like a first edition."
                 }else{
                         say-line "It's inscribed, `To a special fan\
                         Yours, Adolph`."
                 }
```

Use the if-else argument within the open verb so that if the door is already open the actor will say a line to that effect. Otherwise the door will be opened.

2.2.10 New Verb Use

Add another verb. Put this new verb right under the close.

```
        verb close {
                state-of front-door is CLOSED
        } <- press ctrl-return here
}
```

The verb is use. Set up the use verb using the if statement. Toggle the state-of door depending on the current state. If the door is open, close it and vice a versa. Your code should look like this.

```
        verb use {
                if (state-of front-door is OPEN) {
                        say-line "It is already open!"
                } else {
                        state-of front-door is OPEN
```

```
        }
}
```

2.2.11 Redefine object bat

Go into flem, office, edit objects. Click on the object named bat Move the slider bar over to object portion of the room. The object next to the third bozo is a blank wall. As you did before move the cursor over, and draw a box over this image. Click the mouse button and you should hear a beep. You have redefined the bat. It is likely that when you defined the bat earlier that you did not draw the box the same size as the blank wall. This is not unreasonable as you had no idea what was coming next. As you move the cursor, the blank wall box size will continue to display. Click this over the bat in the room. Click with the right hand mouse button on the word bat to confirm correct positioning. The bat should vanish.

2.2.12 New Verb Pick-up

Exit out of flem. Within the bat object we need to create a verb called pick-up.

```
object Bat {
        name is "bat"

        verb look-at {
                say-line "it is a bat"
        } <- press ctrl-return here
}
```

Here is what this should look like.

```
object Bat {
        name is "bat"

        verb look-at {
                say-line "it is a bat"
        }

        verb pick-up {
        }
}
```

2.2.13 The pick-up-object statement

This command picks up an object and puts it into the selected actor's inventory; (sets ownership) and automatically flips the state of the object on the screen. The most common usage of this command is within the pick-up verb.

syntax: pick-up-object [object-name]
example: pick-up-object phone

Inside the verb pick-up

```
v       Type pick-up-object bat

verb pick-up {
        pick-up-object bat
}
```

The key command was pick-up-object and this command activated a large number of things. Within memory is the room office (the one that you are in). There is a bit of code embedded in this room for the object bat. This contains all of the code for the verbs that you (the Scumm programmer) have defined. This also contains a section which was the image of the bat. When the pick-up-object command was activated, Scumm took the object and spliced it out of the room and put the object into memory as a separate entity. When the room office leaves memory, the bat will remain. Unlike sounds, which are global, objects are local and the Scumm programmer cannot run code on objects if they are not in memory. Once the pick-up-object command is activated the object appears in the actors inventory.

Exit out and because we changed a graphic component, we need to do a mkall compile. Once this is complete, run the program and pick-up the bat.

2.2.14 The owner-of function

Used to determine who's holding an object. The value returned will either be the actor's number (only the first 12 actors can have an inventory) or nobody or nuked (see page xyz).

syntax: owner-of [object name] is [actor-name]
example: owner-of hat is sam

This is a way of checking if an item is in inventory.
Currently within the look-at verb of the object bat, we have a say-line.

```
verb look-at {
```

```
        say-line "it is a bat"
    }
```

Change that to an if-else statement that checks if the object is already in inventory. If it is say an appropriate say-line, otherwise say the original say-line.

```
    verb look-at  {
        if (owner-of bat is sam) {
            say-line "It is a bat in my inventory"
        } else {
            say-line "It is a bat"
        }
    }
```

2.2.15 New object bozo

Define a new object. Again this automatically brings up the next available object number. Do this three times. Name the objects bozo-1, bozo-2 and bozo-3. As you did previously with the front-door, draw a box around the bozo-1 image in the objects portion of the screen. Click the mouse button. Position the box over the bozo drawing in the regular part of the screen and again click the mouse button. By clicking on the word bozo-1 with the right mouse button you should see the bozo move. Repeat this with bozo-2 and bozo-3.

This will leave you with three bozo images defined and three bozo objects all on top of each other.

Note when you are toggling the graphic states with the right hand mouse button it is a little confusing, because the system draws in order. If you toggle bozo-1 and then bozo-3 you will not see anything. First you need to toggle bozo-1 back to its original state. Then it will be possible to toggle bozo-3 and see the different states.

Exit flem and go into office,

We need to enter the code for the objects bozo-1, bozo-2 and bozo-3.
Do this under the object bat.

```
    verb   pick-up {
        pick-up-object bat
    } <- cursor here press ctrl-return
}
```

Type the following and then repeat for object bozo-2 and object-3.

```
object bozo-1 {
        name is "bozo"
}
```

For bozo's 2 and 3 right after the name.
Type class is UNTOUCHABLE

```
object bozo-2 {
        name is ""
        class is UNTOUCHABLE
}
```

Any object that has a class of untouchable will not register when the cursor moves over it. The object is invisible to the user.
There is no need to type name is "bozo" in bozo-2 and bozo-3 as they are UNTOUCHABLE. However you do need a name is "" in each object at the least.

2.2.16 New Verb Push

Within the object bozo-1 define the verb push.

```
object bozo-1{
        name is "bozo" <- cursor here press ctrl-return
}
```

Your code should look as follows.

```
object bozo-1{
        name is "bozo"

        verb push     {
        }
}
```

2.2.17 The start-script statement

Starts the execution of a new script, automatically loading it onto the heap if it's not already there.

syntax: start-script [script-name]
example: start-script phone-ring

Add the start-script statement to the verb push within the object bozo-1.
Type start-script move-bozo

```
object bozo-1{
        name is "bozo"

        verb push     {
                start-script move-bozo
        }
}
```

2.2.18 Scripts

Find the section of code with room office "office" and put in blank lines (ctrl-return) before the sound structure.

```
room "office" office {
<- cursor here
sounds {
        "SFX/doorclos" door-close
        "SFX/dooropen" door-open
}
```

Type define move-bozo = 200.
Below this type.

```
        script move-bozo {
                do {
                        draw-object bozo-1
                        break-here
                        draw-object bozo-2
                        break-here
                        draw-object bozo-3
                        break-here
                        draw-object bozo-2
                        break-here
                }
        }
```

This is a local script because it is defined (the script) inside the room. If you had defined it outside the room it would have been a global script and anyone can use it. but since the only place that there is a bozo is the office, we make it local.

You may be wondering why we defined the script as = 200.

The system can only have 200 global scripts, but can have unlimited amount of locals. Each script has to have a number. There is no link process within the Scumm system. In contrast when you compile a C program it compiles it links everything. In Scumm each room is individually done. Since there is no link the system can't run through everything one last time and assign every script a unique number. This means the Scumm programmer to give them unique numbers.
Local scripts start at 200. The global varaibles start at 1. You can have up to a maximum of 55 local scripts in each room. Start at 200 in each room.

Finally don't ever put local scripts calls (start-script) inside objects you are taking with you. Local scripts stay with the room.

2.2.19 The break-here statement

At this point we need to talk a little about multi tasking, because when we say push bozo the system is going to run this script. The draw-object (see page xyz) and break here statements are enclosed by a do (see page xyz) statement. This do has no qualifier such as do-until (see page xyz) Therefore once it is activated the script will run forever while the office remains the current room. The script will be multi tasking in the background.

With the script listed above the system reads draw-object bozo-1, then a break and it generates a frame. Moving on the system reads draw-object bozo-2, then a break and again draws a frame. This continues until the office is no longer the current room or a stop-script statement (see page xyz) is activated.

Exit out compile (mkall) and push the bozo.

As an aside there is a way to get scripts to be running twice but the default is only running once. You can recursivly (rec see page xyz) start a script and then you'll have two copies of it and three and four or or six.
It's just multiple instances of the script all running at the same time. That's how in Indiana Jones, we did the nazi's walking around the castle. There is one script controlling the nazi's and we started it 4 times. We kept track of this using local variables (see page xyz)
So let us stop the bozo.

2.2.20 New Verb pull

Define the verb pull and put this after push within the object bozo-1.

```
object bozo-1{
        name is "bozo"

        verb push     {
                start-script move-bozo
        } <-cursor here and press ctrl-return
}
```

This should have become second nature by now.

```
object bozo-1{
        name is "bozo"

        verb push     {
                start-script move-bozo
        }

        verb pull {
        }
}
```

2.2.21 The stop-script statement

Stops the execution of a script, but doesn't remove it from the heap.

```
syntax:                 stop-script [script-name]
example:        stop-script phone-ring
```

Just like the start-script statement place this inside the pull verb.

```
verb pull       {
        stop-script move-bozo
}
```

Exit out and mk. Run the program and try pushing and pulling the bozo.

2.2.22 The script-running function

Used to tell if the specified script is currently running:

```
syntax:                 script-running [script-name]
example:        script-running tv-static
```

You will find that you can push bozo again. This is because you have not taken into account whether the script move-bozo was already running. In this instance, we no longer need the pull verb. Delete the pull verb and then insert the following within the verb push.

```
verb push     {
        if (script-running move-bozo) {
                stop-script move-bozo
        } else {
                start-script move-bozo
        }
}
```

When we push the bozo what the code is asking is if the script move-bozo is running then stop the script, otherwise start the script.

It is possible to reverse this test. Inside of an if statement simply say not script-running rather than script-running.

If you want to have the same options with the pull verb you do not have to duplicate the above code. The following will accomplish the activation of the code below for both push and pull

```
verb push pull        {
        if (script-running move-bozo) {
                stop-script move-bozo
        } else {
                start-script move-bozo
        }
}
```

2.2.23 The draw-object statement

We passed over this when we wrote the move-bozo script.
Draw-object allows us to incorporate screen animation's using multiple objects. The object can either be drawn at it's original position, or a new X,Y location can be specified. Note that the object's state 0, the one drawn into the background, should not be used in animation cycles since changing it's state will cause an entire screen redraw (too slow for animation).

syntax: draw-object [object-name]
example: draw-object phone-1

You may have noticed that when you stop the bozo it may be leaning rather than in the original position. The bozo should be replaced in its original position. After the stop-script statement you need to add draw-object bozo-2 and that will put the bozo in its original position.

```
verb push pull        {
        if (script-running move-bozo) {
                stop-script move-bozo
                draw-object bozo-2
        } else {
                start-script move-bozo
        }
}
```

You may also have noticed that there are two commands doing essentially the same thing. These are state-of and draw object.
The statement state-of door is open and draw object bozo-1 both cause an image to be shown. Let's explain the difference.

State-of is very slow, because state-of deals with objects. These may be completely contained within another object or they could be overlapping another object.
For example you open a cabinet with a flashlight inside. Pick up the flashlight and when you close the cabinet the system will not try to draw the flashlight on top of the cabinet.

State-of is a very sophisticated object image manipulator which takes time.

Draw-object just blasts the image onto the screen. It does not matter what was there before. Animation sequences that are constantly moving are ideal for draw object. The bozo could have been done with state-of and that would have worked quite well. However draw object is significantly faster.

2.2.24 New Object window

Go into flem and bring up the office.
Create a new object and call it window.

Scroll over to object portion of the screen and draw a box over the broken window image. Click once and then move over to the regular portion of the screen and place the box over the window. Again click and the object window states are defined. This particular image is a little tricky. Use the right hand mouse button to confirm that the image is in the correct position.

Set the use position for the window and the facing. Make sure this is not directly in front of the window. You need to be able to see the result.

Exit flem and add the object window after the object bozo-3.

```
        object bozo-3 {
                name is ""
                class is UNTOUCHABLE
        } <- cursor here press ctrl-return
}
```

There is no need to add a verb or class is statement to the window. Your code should look like this.

```
        object bozo-3 {
                name is ""
                class is UNTOUCHABLE
        }

        object window        {
                name is "window"
        }
}
```

2.2.25 The class is statement

Every object in the game can have up to 24 classes. Earlier in the tutorial you saw a class called UNTOUCHABLE. Classes are local variables. These are just bits, that are either on or off.

There are only 4 classes in the system and the another 20 can be defined in the game, such as full empty hot cold etc., can be set by the Scumm programmer.

Classes sometimes use prepositions. The verb use is the only verb that activates prepositions. There is prep-with, prep-on, prep-in and prep-to. The preposition prep-with would alert the system to insert a "with" after the first object had been clicked on. classes are in capitals but prepositions are lower-case.

Add the class PICKUPABLE to the object bat. Classes follow the name is statement.

```
        object bat     {
                name is "bat" <- cursor here press ctrl-return
```

```
        verb look-at   {
                if (owner-of bat is sam) {
                        say-line "It is a bat in my inventory"
                } else {
                        say-line "It is a bat"
                }
        }
```

Your code should look like this.

```
        object bat      {
                name is "bat"
                class is PICKUPABLE
```

2.2.26 The noun2 function

noun2 is a variable that holds whatever is used after the preposition.
This allows the Scumm programmer to activate code only if the first object is used with the correct second object.

Define a new verb use within the object bat.

```
        verb pick-up {
                pick-up-object bat
        } <- cursor here press ctrl-return
}
```

Add the verb use with an if statement that ascertains whether the second object is the window.

```
        verb pick-up {
                pick-up-object bat
        }

        verb use        {
                if (noun2 is window){
                        state-of window is OPEN
                }
        }
}
```

Exit out to dos and mkall.
Run the program and pick up bat and use the bat with the window. You will notice that when you click on use bat it will now add "with" automatically.

Exit out to dos and then run the program again. This time don't pick up bat first. Instead say use bat with window. sam walks over and picks up the bat and then breaks the window.

This is the effect of the class PICKUPABLE. If neither of the objects are in inventory the system will try and pick up one of the objects first. If you remove the PICKUPABLE from the object bat, sam would walk over to the bat and say some line such as "I can't pick that up."

2.2.27 New Object closet-door

You should have the knowledge to do the following.
Create a new object called closet-door. Define the images in flem, remember to set a use position and facing direction. Then within the office room set up the object closet-door with three verbs, open, close, and use. Obviously open and close where appropriate and toggle for the use verb depending on the current state.

If you have problems check the earlier examples.
Your code within the office room should look like this.

```
object closet-door    {
        name is "closet"

        verb open     {
                if (state-of closet-door is OPEN)   {
                        say-line "It is already open!"
                } else {
                        state-of closet-door is OPEN
                }
        }

        verb close    {
                if (state-of closet-door is CLOSED)         {
                        say-line "It is already closed!"
                } else {
                        state-of closet-door is CLOSED
                }
        }
        verb use      {
                if (state-of closet-door is OPEN)   {
                        state-of closet-door is CLOSED
```

```
                } else {
                        state-of closet-door is OPEN
                }
        }
}
```

2.2.28 The me function

Me is always the current object which is a nice shortcut.
The code that we used for the verb open in the object closet-door is below.

```
object closet-door    {
        verb open     {
                if (state-of closet-door is OPEN)   {
                        say-line "It is already open!"
                } else {
                        state-of closet-door is OPEN
                }
        }
```

Within the if-else statement the closet-door could have been replaced by the function me.

```
object closet-door    {
        verb open     {
                if (state-of me is OPEN)     {
                        say-line "It is already open!"
                } else {
                        state-of me is OPEN
                }
        }
```

2.2.29 Predefined script open-door

The system has some predefined scripts. One of these is open-door.
The open-door script takes the place of the if-else statement that we had used previously. Here is the open verb code before we substituted the me function above.

```
object closet-door    {
        verb open     {
                if (state-of closet-door is OPEN)   {
                        say-line "It is already open!"
```

```
                } else {
                        state-of closet-door is OPEN
                }
        }
```

Substitute the if-else with open-door closet-door

```
object closet-door    {
        verb open     {
                open-door closet-door
        }
}
```

The me function still holds with the open-door script.

```
object closet-door    {
        verb open     {
                open-door me
        }
}
```

2.2.30 The sleep-for statement

This command is similar to break-here. Sleep-for causes the system to time-slice over to the next script but will not go onto the next instruction for the designated amount of time.

```
syntax:         sleep-for [number] [time-unit]
example:    sleep-for 15   seconds
```

2.2.31 Exercise One

Write a script that makes the telephone ring 15 seconds after the front door is closed. Create the telephone as multiple (3) objects in a similar manner to the bozo.
Remember to define the script and start it after the move-bozo script. Launch the script (start-script) from within the verb close of the object front-door but do not put sleep-for within the verb. Make the sleep-for the first line of the new script.

Try this before you read any further.

2.2.32 The for statement

This command is similar to the for next loop in BASIC.

syntax: for var = val16 to val16 [++] {[statements]}
example: for foo = 1 to 3 ++ {

(foo is a defined variable already within the system.)

Using the for statement set up the phone to ring 3 times.

2.2.33 Exercise Two

Once you have done that take out the for loop. Instead of making the phone ring 3 times, add the fourth image of the phone (i.e. create another object) and make the phone ring until sam picks it up. When sam picks up the phone, have the phone stop ringing, show the new object image and have sam say "huh no-one there." Then put the phone back down.

Try this before you read any further.

The best way to do exercise one and exercise two is to create 2 scripts, one called phone-ringing and one called answer the phone. The phone-ringing script is launched from the verb close within the object front-door. This should be set up in an endless loop. The script answer-phone should be launched from within the pick up verb of the object phone. This script, answer-phone, should check to see if the phone script was running via the script running statement.

If the script phone-ringing was running then stop the phone-ringing-script and I change the state of the phone object. Finally run through the dialog with sam, replace the phone and exit the script.

If the script phone-ringing wasn't running, generate a say-line that says the phone isn't ringing, I don't need to pick it up.

2.2.34 Exercise Three

Set up the television set. You will need to define 4 objects. These are the blank television set and the 3 different interference patterns. Set up the appropriate code so that turning on the television will activate the rolling patterns. Also allow the television to be turned off.

The television object should have a use position while the objects tv-1 through tv-3 should not have a use position and should be UNTOUCHABLE.

Try this before reading any further.

When you try to turn off the television you will find that you are unable to. You cannot turn the television off by using state-of tv is gone. The reason for this is that you have a script that is drawing the different interference states. When the script running the television is terminated, one of the interference states will remain on the screen. This could be any one of the three states. Therefore you need to set each of the interference objects to off. If it helps, think of these objects as little pieces of paper that are stuck on the screen and you need to pull them off to see what's underneath.

2.2.35 The draw-object statement Revisited

Draw-object changes the state-of an object from state 0 to state 1. This you learned earlier. However draw-object has an interesting side effect.

Any other object that occupies the exact same space and size is automatically set to state 0.

2.2.36 The order structure

The order structure is necessary in situations where objects on top of each other need to be accessed.

The order structure should be placed after the exit structure.

```
exit {
}
<- cursor here press ctrl-return
object front-door {
        name is "front door"
```

Define the coat hanger as an object. Try and pick up the coat hanger and you will find that you are unable to touch it. Instead the closet door will be the only item that you can touch. In effect the door is still in front of the coat hanger.

The way to solve this is to reverse the order of the objects so that the coat hanger is in front of the closet door. This tells the system that they are in reverse order so they come out properly.

```
exit {
}
```

```
order {
coat-hanger
closet-door
}

object front-door {
        name is "front door"
```

2.2.37 The dependent-on statement

This order reversing does however cause one other problem. Pass the cursor over the closed closet door and you will see that the coat-hanger can be touched by the system. There is another thing you need to do.

This is the dependent-on statement and is placed after the name is"" statement. The statement is set up as follows.

dependent-on closet-door being OPEN

s CLOSED can be used in place of OPEN.

Dependent-on tells the system that the object coat-hanger should be invisible until the object closet door is open.

2.2.38 Exercise Four

So now use hanger with television and when you do that I want one of those antennas to appear above the television set. but then what I want is when it is the original position you throw it up to alternate between static and mickey and then when you push it the other way the static should go away leaving just the mickey mouse ears doing the little dance. and if you push it back again then it will go back to the static and mickey mouse ears you will have to do it with 3 scripts.

s You can change states, such as UNTOUCHABLE, at anytime.

s There should be a space between classes not a comma.

Try this before reading any further.

variables (global and local)

The Scumm system has two kinds of variable. These are global and local.

A global variable can be used throughout the game. A local variable is specific to a script. It is possible to have a local variable, with the same name, in another script, in the same room, and they will not clash.

s Local scripts are ok through breaks but not through successive calls.

I want to talk about them in conjunction with the television set so I want to show you the way to do the television set and I want you to rewrite your television sets this way using variables. and the way to do it is you should have only 1 script that controls the television and when you turn on your television you start the script and when you turn off the television you stop the script and then that script checks a variable which is the television status and decides whether it should roll it, whether it should mickey mouse it or a combination of both and this script is just constantly checking the variable so when you put on the antenna all you need to do is set the variable and the script that is running in the background will notice the variable change and automatically do what it is suppose to do and this way you don't have to have a bunch of code starting this script and stopping all these scripts and changing all the objects. each of the different antennas and all of the different positions. It really just localizes it. what you want to do when you are programming in Scumm is to think about each little objects brain, the television has a little brain which is the television script and then that little script has to look at all the factors that might control it like antennas and stuff like like that rather than everything out here trying controlling it the little brain is working. now to do variables go into brief and load in variables.scu. this is a list of variables the first 50 are system variables under no circumstances should you touch these. so go to the end of the file and you'll see a list called user variables and after that is a little thing at the end is bit variable. bit variables are just 1 bit variables either true or false, on or off right above that are user variables. user variables are ? . they are 16 bit assigned number. now if you want to add your own variables you notice the last user variable says "variable dialing-ypos" if you want to add your own variable just add it after that so lets add variable tv-status so now we have defined the variable tv-status. it is a global variable and now we can use. If you use them it is just like C. Foo = 10 or foo = bar you can use a variable in any place you use a number. you can do this but you have to put () around (see illo). check wether and/or variable has been implanted re foo. you can do addition, multiplication, subtraction and division you can do and or or. I may not have implemented that yet. This is what you cannot do (see illo). anyplace it is asking for a number it must simply be one number or one variable. The only place you can do complex expressions is within simple assignments. Foo = (complex) The expressions can be as complex as any other language but

it must be surrounded by parentheses. variable foo is not a special variable but we treat it like that. It is a temp variable and you should never plan on it retaining its value through a given break. so you can set foo to ? and then use and then there is a break, foo might be destroyed by something else. so it is global.. we can say if f=n or if f is n or not= to or is not. you can do < > <= >= not= all the standards. if foo = or if not foo.
this is how you do a local variable, when you define a script. we have a script move bozo. you just say local variable I. for I is 1,2,3 etc this is now a local variable. it is only local to this script and it is different from I in another script and they will hang around through breaks but not through successive calls. If I say start-script move-bozo I = 0 but if I say start-script move-bozo again then I will revert to 0.
To do comments place ; everything after is a comment.
Instead of doing multiple if statements you can do 1 case statement. after last of you can put default and that will catch if non of the actual cases are tripped.

s It is important to remember that you must add variables at the end. If you put a variable in the middle of the list, everything else will get shifted and then there numbers will be off. If you ever add a variable in the middle then you will have to re-compile everything. That doesn't mean doing a mkall it, means you have to compile every room in the game. In monkey island that tak

Sound Effects

Let's go to sound effects. go to office.scu. it says door-open and door-close. If you want to add a new sound effect if you wish to add the bell just type "sfx/bell" bell (check) and what this does is it associates bell with this file. now go into header and at the end it says sounds. door-open is 1 door-closed is 2 make bell=3. Then the command start-sound bell. last thing you need to do is brief mm.bat file replace adlib with i. the other command is stop-sound. some sound effects are looping and will go on forever and to stop them you need to do a stop-sound. but in this instance the bell is a single shot.

gap

Making an object in Flem

type dp. select mode c and pull down file menu. and double click on art and then on office.lbm. on the menu bar that looks like a box with a cross in it. It is above the magnifying glass click on it with the right button and when you do that a little box comes up that says grid spacing etc. click on adjust. now you should have a grid pattern that you are moving around. take this grid and move it all the way to the upper left hand corner and what you need to do is

gap
(place the grid in the top left hand corner and click. remember to move the slide bar up also)

but don't forget the grid because flem works on 8 pixel grids and if you don't stamp the object down right on a grid you won't be able to pick up the object in byle .

check order from now on.

also there is a limit of 70 objects in a room so you might want to keep that in mind as you are adding everything you can think off. The whole picture including the main screen and the graphic elements can only be 5 screens wide and you can divide that any way you want as to what is room and what is object area.
exit back to the prompt. we had some trouble yesterday with the newspaper and under the newspaper is a key and we tried to treat it like the closet door, but because the closet door goes to an open state.
(Ron) It is exactly like the closet door. Your problem is the order. The key has to be in front of the newspaper that doesn't seem right because the key is under the newspaper it seems logical that the newspaper should be on top but just like the clothes hanger needs to be in front of the closet door in the order statement it is the same way. The first state of the newspaper is the newspaper and the second state is the key and the first state of the key is the key and the second state is the blank floor. The key should be dependent on the newspaper being gone. and then have the key come before the newspaper in the order statement. It is exactly like the closet. There is no difference between the two.
Only one order statement.

Adding a new room

So let's talk about adding a new room. we already have art for this called sam-hall. so the first thing we want to do is brief a file called rooms.ifo. now what rooms.ifo does is that it is a master list of all the rooms in the game so as your game goes on you will be maintaining the list adding and deleting rooms. This is where you keep them all. so go to the line after office and add the room sam-hall then tab and =2 and then type disk 1. what disk 1 does is it tells the programmer what floppy disk the room resides on. That's so when it says I need room 4 the program can say please insert disk 6. so it knows what disk it is on. we put them all on disk 1 to start off with and at the end we divide them up. after that type size 144 this is just specifying the vertical size of the room how many scan lines tall the room is after that if we had a z clipping plane we would type zplane depends, but we don't have any so skip that. so press return this next part is really stupid but there cannot be a blank line at the bottom of the file. when the cursor is below sam-hall press the plus sigh a couple of times and it will delete the blank lines

you should get a cursor that looks like an underline rather than a block and that shows that there are no blank lines. then exit out now go into your art directory and run dpaint. and load in sam-hall. so this is the new room and off to the edge we have some nice objects. we have a rat eating a candy bar, we have a dead guy in a door and we have a rat exploding. so a bunch of new objects for you to wire up. now exit now why you are in the art directory type addroom. you need to run this addroom program any time you add a new room to the game or take away a room or change the name of a room, anything that does that you need to do addroom. what this is doing is that you notice when you type mk that it recompiles the office automatically. It is doing that based on time stamps. It realizes that the time stamp on the .scu is newer than the time stamp on the .lfl file. and so it says it must have been changed so I need to remove it. What the command addroom is doing is building all those time stamp dependency files. for you in all of your different directories. so all you have to do is type mk and it will pick up all the changes you have made. so now when you are in the art directory just like when we added an object to the office you had to type mk in the art directory before you went to flem you have to do the same thing here. so type mk in the art directory and now this will take sam-hall and crunch it down and once that is done bring it into flem. and you should see a sam-hall in the flem list now.

Editing Boxes

something else we need to do is when you first bring up flem you will notice that there are two boxes, one says edit objects and the other says edit boxes. we need to edit boxes. so what I need you to do is click on edit boxes, then on sam-hall since this already has boxes we'll get to change them. go to box 0 click on it then pull down the boxes stuff menu and use delete box. Delete all 4 boxes and we'll start over. what the boxes are are they tell you where people can walk. people only walk inside off boxes, never outside. so if you have a room and you have a table in the middle and you want him to walk around the table you would define 4 boxes.

diagram

and then when he walks from a to d he'll walk around. if you want to see complicated boxes look at the boxes in the office. the program figures out automatically the path it should take. you just have to set up where he can walk.

Rules for Boxes

so we are going to do this for sam-hall. so lets create some boxes click on 0 and then move the cursor up onto the screen and click. A box should appear on the screen. click on the right button will allow you to move the box around the screen. so what you want to do is set the box down with the bottom of the box at the bottom of the screen. remember to scroll the screen so that you are at the very

bottom of the screen. when your box is at the bottom take your cursor and there are these little handles on the corners of the box. just click on the bottom left handle and slide it into the corner and you should stretch the box. so all it is is a polygon you want to keep it .5 inch away from left and right walls as the characters walk is based on their centers and so he will stand at the far end of the box and that will be where his center is. which means he will be standing up on the wall if the box is too near the wall. so I will probably line it up at the edge of the carpet. now take the right hand side and stretch it across the screen to where the crayon drawing of max is on the wall. we could do this room with only one box but we will do it with two. now when you stretch it across this edge (right) must be vertical. It cannot be slightly off line. It must be absolutely straight up and down. The line has to go straight through the centers of the handles. now this is just in this instance. once you have box 0 lined up then click on box 1. scroll the screen over and then drop that box down and with this box the left hand side must be exactly vertical and must overlap the box 0 right hand vertical edge by exactly one pixel. then stretch the right hand side of box 1 to the far side of the screen. (overlap should show only 1 line. they are on top of each other).
Now the reason that these two boxes are overlapped like this is that the computer needs to know that they are connected. It needs to know that it can walk from box 0 to box 1. and the only way it can tell that is if you overlap them. If the two sides occupy the exact same space on that one line that's how the program knows they are connected. If they were separated by a pixel sam would not be able to walk from box 0 to box 1. he would get to the edge and stop and there are instances where you want to do this. If there is a fence running down the middle of the screen you just separate the boxes and he won't be able to walk through the fence. If you overlap by more than one pixel it still doesn't recognize that bit it will recognize one that is 40 pixels high connected to one that is 20 pixels high if the lines are both vertical and occupy the same line. The rules are pretty simple (see diagram) you can have a box which is a line and come in at an angle. In a triangle it will be connected only if two corners are together at that connection point. If you want to check if a box is connected look under boxes stuff. there is the option 'show box connections' select that and you should see a line between the middle of box 0 and the middle of box 1. that tells you that those two boxes are connected. now leave the option on and take the cursor and move one of the points on the left hand side of box 1 and you will find that the red line disappears. because the boxes are no longer connected. we suggest when you are drawing boxes for a room that you leave the option on and when you are moving boxes as soon as you see the line appear you know you have a connected box. so after you have done this go up to the boxes stuff menu and select the option that says box connectivity. It will come up with dialog which says that this will destroy the old connectivity. press yes. what this did the program figures out where characters are going to walk at compile time not run time. when you click in the game from box 0 to box 12 the program has already figured out the path that he is going to take. It figured it out when you said do box

connectivity. what it does is it builds a little matrix in memory of all of the boxes and how to get to every other box in the game. and the reason is that because it is a very time consuming task because you are basically doing a maze algorithm and I can't do that at run time because when you clicked somewhere it would pause while the program figured it out and then you would start walking. If you have 20 or 30 boxes in a room it could take a couple of seconds to do that connectivity that you did. It happens so quickly. there are rooms in Monkey Island that have 60 boxes in them. It takes a minute and a half to figure out the connectivity.

Creating object in new room

now that we have the boxes defined pull down the file menu and click on edit objects and that will bring up the familiar objects screen. lets add an object, the doorway. so create an object called sam-hall-door. don't call it door because somewhere else you might have an object called door. now it will say object (number 74 or whatever). there is a master list of objects and the numbers follow one after another. now the naming scheme we use for doors, because once you get into the game there will be hundreds of doors you can be sitting there wondering that you have use door this and door that. so we have a naming scheme. That is the room you are in plus the room it is going to. so sam-hall-office-door would be the name of the door in one direction and office-sam-hall-door would be the name in the other. (actually it would be hall-office, because the name is too long.) what this allows you to do is it allows you to tell easily know what is one door and what the other is. so create an image for it,

Imagemin
scroll your screen over so that the dividing line between the purple section and the normal section is in the middle of the screen. the way it knows between objects and the regular screen is imagine and there is a little button that says set state when you click on it it says set use. when you click on it with the right button it says set imagine. move the cursor to the screen and you will see a vertical line. this is the imagine. move the vertical bar to the border of the main room and the object area and click. That sets the imagine. you will need to do that before you define any object in a room because if you don't you will get into a situation where you define an image and then go over to the door and then you realize that the image is not defined anymore. because it doesn't know where the imagine is so you have to set that imagine first. then you wont have any problems. exit out, it will ask you if you wish to save objects and say yes.

Building a room

go back to your Scumm-U directories and I want you type this command.
skelroom sam-hall > sam-hall.scu.
what this is going to do is it is going to build a room for you. It will include the

door you defined, it will include the room headers and all the header stuff that you have to bring into it. you don't really need to do this, you could bring a blank room into brief and start typing. this makes it easier, normally when we get a room, we know all the objects so we bring the room into flem, we define 30 objects and then skelroom and it has already built those objects for you so you don't have to type them all. But if you already have the office and you add a new object do not run skelroom because it will destroy your old office and leave you with an office with just a bunch of empty objects for it. so don't run that on previous rooms that exist. so run skelroom, then brief sam-hall.scu then you have a little room with the objects defined. That's all there is to adding a new room. ok now don't edit this.

Come-out-door statement

go back to the office and find the front-door and go to the walk-to verb and you have a statement that says something like if state of front-door is closed say "I can't walk through closed door". Put an else statement on that if and inside that type come-out-door hall-office in-room sam-hall. What that does is that Scumm is going to take your main character, it is going to go to this room and it is going to position him at the use position of this object facing the opposite direction of the use direction so for doors you have him facing to the back so what this will do is it will position him at the door and face him to the front like he walked through the door, because you assume that when he comes out of the the door you want him facing the opposite direction. so once you have added this compile with a mkall and then try to walk through the door. So now you may have noticed that when you walked through the door that the door in sam-hall was closed. so inside the open code in the office when you open the door in the office you also need to open the door in sam-hall and you can do that from the office, you use the state-of statement. state-of works for objects even if they are not in the same room. and that will open both doors at once. now remember that there is a predefined script called open door and you would pass at a door and it would automatically open it. If you pass two parameters to that open door script just pass the name of both doors and that script will automatically open both doors for you. There are a bunch of system scripts one of them is called open door and you pass it the two doors. Come-out-door must be the last statement in your verb. when it runs that come-out-door it gets rid off the current room and brings in the new one and any statements after the come-out-door code are not going to be run also some of you when you open and close doors you get a sound associated with it. open door (the script) is making that sound now if you want to go back into the office you need to put open door inside the open and close verbs of the doorway or a come-out-door. so there is nothing special about doors, you could walk to the telephone and have it come out in the hallway. It is just an object with a walk to verb associated with it and a come out.
(remember object names in flem in lower case)

Exit and Enter

something you may notice is when you walk back into the office the television set may not be working anymore. the static was rolling when you left but when you returned it was not. It is because the script that was controlling the television set is a local script and leaving a room kills local scripts. This is what exit and enter are for you need to check if the television set was on so that on reentering you can start the television back up if necessary. And the exit code should turn off music when you leave unless you want it to follow you wherever you go.

Actor and Object trade-off

actors are a class of thing like an object or a room and they can move from room to room. they can have positions, they can clip behind etc. even though we call then actors we don't always use them for people or animals. a lot of times we use them to represent things like fires. you know in a burning fireplace or smoke, lots of different things we would use an actor for. actors have some trade offs that objects don't have. It is slower to draw an actor rather than an object. so if you have a fireplace that's going to go through 3 cycles of flame it is going to be faster to create 3 objects and use draw object than to use an actor although it will take less memory to use an actor and the reason is this, if you have 3 objects of the flame and its different states you also carry around the entire background with those objects and that can get expensive. If this is a busy background with lots of differs. With the fire all you need is just the flame for the fire for now. so it takes less memory now it is a little tricky if you have an actor and an object and they look identical for example if you were to take the exact image used for the 3 objects and you put them into an actor it would be more expensive. It would take more memory. The compression that is used to compress images is much more efficient with objects than it does than actors. so given two images that are identical an object is going to compress much better than an actor is, about 1/3 better, but the flame is still cheaper as an actor only because we don't have to carry around all the background. so even though it doesn't compress as well it is still better to use just because you don't have to carry the background but it is slower and that is something you have to be concerned about.

Elevation

actors automatically clip when one is standing behind the other. it is all based on vertical position. so if you have one actor standing here (see diagram) and another standing behind him, he is going to clip behind because he is higher up the screen. It is the only thing that is used to determine who gets drawn in front of whom is just how high they are on the screen there are some tricks that you can use. there is a thing called elevation. This just causes him to levitate off the floor. so if his normal position is here (see diagram) you set an elevation for him he'll show up here elevated off. but when it is figuring out the priorities off who to draw first it is using the base point not the elevated level.

actor foo elevation 10. Elevation is part of the huge actor statement. the default is elevation 0. for those of you who have played maniac mansion the plant that was in the room, there was a giant man eating plant and the kids could climb up the plant but they always were in front of the plant no matter how they climbed and the reason for that is that the plants position was up here at 0 and we moved the elevation down with a minus and that meant that the kids were always in front of the plant. now I don't know if you have discovered but in the current version of Scumm there is a bug, you can't do negative numbers. you cannot specify negative 10, it crashes. so how you do elevation is that it is just a 16 bit number so for now it is 65535 -10 whatever that comes out to be, you need to figure the number out in advance, you can't do complex expressions. so that is what elevation is and how you can overcome the vertical clipping.

Actor Classes (changed as of March 1991)

actors have classes like objects do. objects have the touchable, untouchable etc all those classes associated with them and actors have a group of classes associated with them as well. they are always clip on and always clip off. if you set an actor's class to always clip on he will always clip no matter what box he is in no matter where he is. if there is a z plane he will clip behind it even if the box says don't z clip he will do so anyway. there is always clip off which just turns that off. there is never clip on which means no matter what he is doing he will never clip behind anything and there is an off for that as well. you can find all of these in the header.scu If you say always clip off and never clip off then the actor will do what the box says to do. But if one of those is on then it will override what the box says and do what the on says and if they are both on we can't guarantee what is going to happen. It will do one of them, whatever the last one is that the system decided to check. This is used for such things as birds and it is going to fly across the screen but you don't want it to clip behind things because he is going to be floating in the air, so the clipping information probably isn't valid for him. Let's say you want a title to pop up on the screen but you don't want it to clip there is another class called ignore boxes on and ignore boxes off, if an actor is ignore boxes on he is not going to pay attention to any boxes in the room. If you tell him to walk there he is just going to head there he is going to walk up the walls and up the ceiling and that is great for things like birds. now whenever you say ignore boxes on you better set one of the clipping ones, because the system doesn't know what to do with this guy any more because he is not in a box so he is going to randomly clip based on what he was doing last. so those two usually go in concert when you say ignore boxes on you usually say never clip on as well. so when the bird flies across the screen we don't have to worry about it or if we wanted the bird to fly in front of something that is clipped but there is nothing behind it then we would say ignore boxes on and always clip on and then as he flew across the screen he would clip. there are a whole bunch of classes (insert) There is one called, I don't know the exact syntax (check), flip-walk-x and flip walk-y. which means when the actor is walking to the right he is facing right but if

you say flip-walk-x he will actually walk to the left and do a little moon walk. same thing for front and back you can flip now the reason is, if you have a screen that is looking down, which all the screens are, the camera's pointed down, coming toward the screen is forward away from it is back but if for some reason you position the camera below that orientation gets flipped, there is a scene in loom with the dock, you are actually looking at the dock from below so now coming toward the screen is up and going away from the screen is down. so we have to flip the y orientation of the actors. so when they walk up they are actually facing forward not backwards and the boxes individual boxes can be flip x and flip y. so what it was in loom, we didn't set bobbin to be flip x or flip y we set the box that is on the dock to be a flip y box. and so whenever bobbin stepped into that box he would also face the opposite direction that the system thought he should and then it looked right.
question? how do you set the boxes?
ron you need to go into brief and twiddle with the bits but all these things should go into flem later. so that is the fundamentals about actors.

Adding a new costume

so now lets move onto costumes. costumes are listed in the costume struct. which is similar to the sound struct.
costumes {"costumes\max" max-skin} and you need to define this inside the header.scu you will find sam-skin =1 just add max-skin =2 and you would do this for every costume that you add. There are only 14 actors and if you wished you could put them all in one room but the program will run pretty slowly you should probably put no more than 5 actors in a room. so since there are only 14 actors, actors need to share what they are and usually we will take our first 5 or 6 actors and they will be our main characters and we will leave them the way that they are like in Monkey Island, guybrush is always number 1 and the crew members you choose are always 2,3 and 4 then about actor 6 through 10 we define as extras. so it is actor 5 we do a simple define in the file then we say define extra1=6 etc etc, in the header. Then if you need an actor for something you just use one of the extras. but you have to be careful you have to make sure that the extra is not already being used in the room or worse you use an extra 5 to be the fire but what you don't realize is that you have a guy walking from room to room and suddenly he is going to enter this room and he is also extra 5 because what is going to happen is that when he decides to enter the room the fire is going to vanish and he'll walk in and through giving animation commands to the fire they will be given to him instead and he will start freaking out. you have to be real careful about your extra's and make sure you break them up.

Using default within actor

question? so where are you adding a costume onto a previous actor. there is a command called actor It's a very long command which has lots and lots of

options when you define a new actor say max the first thing you should do is default, That should really be the first command you give when you are going to bring someone in for the very first time. What default does is that default cleans all the optional parameters that might have been set the elevation, all of his class bits, his talking color all of these strange things that may be set for a guy you want cleaned and rather than having to clear each one by hand which is a real pain, you just say default, because if this actor had been used previously as the plant his elevation would already have been set. then max would have that elevation and you would have spent a lot of time trying to figure out why max was not on the screen max was above the screen because his elevation had been set and you would be surprised you can waste 3 hours trying to figure why max doesn't appear and it is simply that his elevation is off the screen. If you have a fire burning you would probably put it in the enter code, because you come in and run the enter code and you set up the fire actor and you start him animating and then you go in. someone like guybrush the main character in monkey Island is set up once at the beginning of the program and never happens again. he changes costume but we don't globally redefine him so it all depends how you will be using these guys. This all has to be on one line but you can put a \ at the end of the line so that it is all one line of code written on multiple lines. If you were doing say-line \ must be outside of quotes, so then we say costume max-skin this sets max's costume to max-skin. so it is default costume max-skin. this is the simplest you can get away with just that. There are a lot of other options on there. any time you want you can give this command actor max costume frog-skin The second you give that command max is going to change into the frog-skin (which you would have defined in the costume structure just like max)

Changing Costumes

Question In indy when you throw up the censored box does the actor go into the void.
ron we change the costume behind the censored sign
Question is this done with draw object.
no the sign is an actor, the censored sign is just thrown up and the first actor is clipped behind it some of the other options you have are elevation like we described earlier like actor max elevation 10,

Actor command explanation

you've got sound now sound is the sound he makes when he walks. When the artist choreographs and draws the actor there is a special little code that you drop into the chirography list called make a sound and you usually drop that in the list right where the foot hits the ground and then as we are animating through every time we see those marks we will make a sound effect and you define it using the actor sound command so you can have the footsteps be anything you want. there is talk-color when someone says something what color they say it in. There is

step distance which takes two parameters and x and a y which tells how many pixels they should walk left and right and front and back. so you can make someone take little steps or make them take big steps just by changing their step distance. there is name when you say actor max name is "etc" this is the name that will be seen when the cursor goes over and touched them. You have to set some classes before actors are touchable. These are giveable or talkable and they need to be giveable or talkable for their name to show up. Talkable just means this is someone you can talk to and that means you have some kind of dialog associated with them or they are giveable which means you can give items to them. The classes can be done just before or after actor statement. and you just say class of max is just like you did with objects.
question If in a game we want to have something other than giveable and talkable to operate an actor such as use bandage on etc?
The entire interface is written in Scumm so the giving and talking classes are just scripts that are looking at those so there is nothing intrinsic in the system so you can set up your own classes by rewriting the system script a little for example punchable, this is a punchable person.

Animations

There are 4 other commands called walk-animation, talk-animation, stand-animation and init animation. when someone walks the system gives them animation 3 and when they stand it gives them animation 2. you can change those around if you wish. by default we tell the artists when people walk and talk and stand always put it in these animations but there are certain reasons why you might want to refactor for example let's say we have a costume of max walking and the artist also draws in that same costume, max limping so what we can do is to set up the walk choreography to be 3 and the limp choreography to be 10 so when sam uses the baseball bat on max then we can redirect the walk to be 10 and now whenever max walks he will be limping. Inside the actors costume is where all the choreography is stored. so when he walks it knows what cells to paint and when. actor max walk-animation 10. now when you say actor max default it resets everything back that will set his walk to 3 if that is what it is.
Quick comment on byle. Byle is really complicated and I could spend a whole week describing it. In its simplest form there is a choreography that's an action that someone can take. and the artist can set up the action and they can be fairly complicated like a guy exploding and his head blowing off and that is one action and it is called a choreography. and then you can hit that one choreography and it will go. walking is the same way. walking is a choreography which is just repetitive. so when the system says walk it starts doing the walk choreography. there are two types of animation ones that loop and one shots. walking is a looping animation.

Say-line punctuation

There is some punctuation associated with say-line. One of the punctuations is a colon (:) Followed by new text and the new text is always on a new line. colon just means print this line and wait and then when it times out print this line you can also use a comma (,) which will print the next line immediately underneath the previous line. The other punctuation is a plus (+) sign and the plus sign is the last thing in the say line, for those of you familiar with basic it is like the semi-colon in basic, it will print up this message and then stop and it will wait for you to do the next say-line and it will just continue it off the end. It is important to realize that this first message will get printed first if there is a break so if you print up the first thing with a plus sign and then there is a bunch of breaks you are going to see the first message thrown there and then when the breaks are done the second one will get thrown up. This is really dangerous
don't put breaks between say-lines with plus unless they are in a cut-scene where you are sure nothing is going to happen because what you might do is you print the first message which is "I'm going to the " and a plus sign and then the player clicks on something weird and then sam says something before he has time to finish the say-line and it will plaster it right up there. so you should be careful with these. we usually use these when we want to print up a sentence and we want to imbed something inside the sentence like who we are talking to you. you know we can print up 'gee I see your name is' and we do a plus sign and we will do a bunch of if statements and then we will do say-line to finish the guys name but there are never any breaks in between you can do some weird special effects where you have a little loop with an A with a plus on the end and a break and a loop and it will just print a bunch of a's across the screen but you have to be careful with those because if anything interupts it it is going to get printed up.

do-animation

you can trigger the choreography (sometimes called animation) by using the statement do-animation. we can say do-animation max 10 and that will just hit max with the animation 10.
question so when we hit him with the baseball bat we say do-animation max 6 and he would do the animation?
right whatever the animation it is he does it. If it is a looping animation it will do it and if it is a one shot it will do it and stop. If you have a costume let's say the normal sam-costume and he is standing here and you have the artist draw max getting his head knocked off and that is a separate costume so max is just going to be standing there and when sam uses the baseball bat on him what you do is you give it that actor command, actor max costume head-bashing. the first cell of that costume should be max standing there just like he is in his normal costume so when they get put on top of each other it is seamless. you don't see it happen and you need to make sure that they are standing in the exact same place. Because if the artist drew one off from the other max is going to jump over when

you change costumes. so you have to be careful that that doesn't happen. so what you do is change his costume to max getting his head knocked off then you bash it with the good animation 20 which you have defined as getting hit and then max will go through the animation of getting hit in the head and his head will fly off. Then when that animation is done you need to do something with him maybe the animation ends with max's bloody body on the floor and you can just leave it or maybe it needs to go into another special case animation but you do need to step it through the whole process. animation happens concurrently while everything else is going on just like scripts are multi tasking animation happens one frame at a time.

Walk statements

There is a whole group of commands called walked. you can say walk max to 100,75 and that will take max no matter where he is and walks him to 100,75. There is a command called put actor max at 100,75. this will put him at 100,75 it will not walk him it just pops him there. If you have some animation going on it will pop him to the new place and the animation will keep going on. you can put a little for loop around here with i instead of 75 and you can slide him across the room if you don't really want walk because walk does some funny things, if you are facing this way and you say walk over here he is going to turn and then he is going to take his steps and you may not want that to happen. so if you can do something like lets say max gets hit in the head and he vibrates across the floor while that's happening, what you do is you give him the vibrate command which would just be jumping in place and then you do a little for loop where you moved him across the floor. The put has a little optional parameter at the end of it say put actor max at 100,75 in-room office if you leave off this in-room it will put him at 100,75 in whatever room he is currently in. so if you are in the hallway and max is in the office and you say put max at 100,55 he will go to 100,55 in the office not the hallway and that can lead to some confusion because you are going along telling him to appear here but he is not and the reason is because he is in whatever room he was in last so if he is going to do a transition into a new room you need to say in-room office and that will bring him into the room for the first time.

Put-actor

There is also the option put actor at object the problem with this and it messes people up sometimes is objects are not valid unless they are in the current room but if you are in the hallway you cannot put max at an object in the office, but you can put max at co-ordinates in another room. be careful with co-ordinates because they will trap the co-ordinates into boxes so make sure the x and y you give is in a box.
question how do you change the actor so you can switch from running sam around to running max around.

selected-actor

there is a variable called selected-actor and it contains the actor that is currently active. If you just set selected actor to be max then when you click max will walk around instead of sam but the camera is not going to follow. there is a command that you have to give called camera-follow, it tells the camera what to do, just say if you have a little script that changes control to somebody what you need to say selected-actor = max camera follow max. then the camera will follow max instead of sam. now the camera follow command is pretty smart. If max is in another room it will iris out of the current room and iris up in the new room.

camera

There is a command called camera at and a position and it will snap to that position and it is no longer following sam. That kills the follow. there is another command called camera-pan-to and that will cause the camera to slide across the screen to the destination rather than just pop to the destination. suggestion rig up the telephone so that when it rings the camera pop's over to the telephone and you see it ringing so even if you are on the far side of the screen it will pop over to watch the phone ring but when you are done you need to say camera-follow selected-actor to have the camera return to the original location otherwise the camera will just hang on the side of the screen with the phone. when you are switching characters in a game such as maniac mansion or Indy you should use selected-actor everywhere not sam or max because when something is happening you don't know who might be there doing it. It could be sam or max picking up the phone but if you just use selected-actor as the variable then you are guaranteed that it will happen on whoever is the selected-actor. you can't say camera-pan-to selected-actor. there is a way to show

actor-x function

where an actor is that is actor-x. you can say foo = actor-x sam and this will return the actor's x position then you can say camera-pan-to foo and it will pan over there.

cut-scene

Cut scenes can be very useful and also very dangerous. a cut-scene is like a structure. when you hit cut-scene the system is going to freeze every other script that is running and then you run commands and when the cut-scene ends it will unfreeze the scripts that were frozen so when you do a cut-scene you are guaranteed that you have complete control over the machine, nothing is going to interfere with it. This cut-scene for the telephone ringing you want it to go there wherever you are in the game. If sam is over in a hospital and the phone rings you still want to cut to the telephone and all you do is cut-scene, current-room is

office, camera-at give it the position of the telephone and then sleep for 5 seconds and then end the cut-scene. when the cut-scene ends it will return the camera back to where it came from which was in the hospital following sam, because cut-scenes know what was going on before and they push it all on a little stack do there thing and then pop it off the stack and return the system to where it was and when the cut-scene ends it automatically does a camera=follow selected-actor for you so if you were already in the office the current-room office has no effect and then it pops to the camera-pan then sleeps and then does a camera follow.

selected room

There is a variable called selected room. selected room is just whatever room you are in so if you want to see if max is in the room with you you can say if the actor-room max is selected-room then he is with you and you can do something else.

closest-actor

There is a function that returns the closet actor from one to another so you could say foo= closest actor to max and it will return the closest actor.

proximity

there is a command called proximity which will tell you the number of pixels between two actors. you can also put objects in proximity and can check for the distance between sam and the door. That is useful let's say you have a rat somewhere , an object, and it has a state of standing up and going down. If the proximity of sam and the rat is less than 50 then he is up otherwise he is down. There is something you wanted to know earlier about walking away from the telephone what you could do is when he is on the phone you have a little script running and all it is doing is checking the proximity between the telephone and the selected-actor and if the proximity is ever not 0 you put the phone down. You could say if the proximity of sam and max is not 0 you could do a do and a break until proximity of sam and max is not 0.
If var = var, you have if var = val (like 8 or 10). if function = var, like actor-room etc and then if function =val and those are basically the only if's you can do state-of and class-of which are essentially functions you can't say if 10 = the proximity of sam and max you have to put the function first.

Windex

Explaining windex

Inside mm.bat you need to add a lower case w or you can do mm w and that will start it up as well you should see a message that comes up and says windex

debugging system. and that means windex is running. if you press the num-lock key while the program is running that breaks so then what you should see is windex command and that just halts the system and if you continue to press the num-lock key it single steps the program every time you press num-lock it does a single step. It is just going to do one instruction at a time and you can see on your screen that it tells you what script is running.This isn't source level debugging but windex does as best of a job as it can. so what you have got is scr which means script and a number like 28 which means we are currently looking at script 28. The next number is a slot number, just ignore that and the next number after that is the off set into the script and you don't have to worry about that and after that are the instructions that it is executing, a couple of things about the instructions. you will notice. It will say if (d57) that just means variable 57 and so you will have to look inside your variables file to find out what 57 is because it doesn't go in and give you a table. after that there is a curly brace and a number and then another curly brace that is the value in d57 so you can see what is in d57. as you single step through the program as it gets done with one script when it gets a break and goes onto the next script that will be reflected in windex. this isn't going to look exactly like you wrote the code because if you say if foo = 10 and a bunch of code and you say else and you do some code you are not going to see this else in your single stepping, because what the compiler actually creates for that else statement is what you are seeing. It is going to generate the if and then your code and it is going to put a jump which jumps over the else code so you are going to see a bunch of jumps that you didn't put and a lot of times your code is going to compile things that you didn't type. you need to be aware that this is going to happen don't follow this verbatim especially if you use macro's because macro's compile in line and you may have typed one command like set dialog command which actually expands into 20 instructions, so as you are single stepping through some things might not make sense. some of the other things you can do are examine variables. if you type v28 and press return it is going to tell you the value of variable 28. now if you want to know local variables you need to type l2 and that will give you the value of local variable 2 but that is the current running script. If you want to see all of the local variables just type locals if you type locals what that is going to do is it is going to print out all the currently running scripts on the edge of the screen and all the local variables across the right of the screen that don't equal 0. so for some people when you type local you just got script 28 printed on the screen with nothing after it that just means there were no local variables in script 28 that were in use. and in use is defined as not 0 you could still have a local variable that you were using but if it just happened to be 0 it will not be printed out on the list. so typing local is a quick way to examine all the local variables.

debug

There is a command called debug. now if you put a number like debug 2 inside windex. There is also a command called debug. So inside windex if you type

debug 2 and run your program when windex sees this statement it is going to stop. It will be running normally and then all of a sudden the windex screen will come up because you had the debug level set in windex to the same debug level as this. If you type debug 1 inside windex it won't stop. Debug 0 clears out the debug now if you wanted it too stop you can put a debug 0 in your code and it will stop because the debug default is 0 so you can put in brakes, if you want to see what the value of some variables is at a certain place because it doesn't seem to be working just put a debug 0 there and then it will stop and you can examine the variables when you are in the single step mode the mouse doesn't work right you can't really click on the screen and get things to happen like touching objects because windex has its own mouse so if you want to trace like when you are clicking on an object the best way to do that is to walk the guy away from the object click on the object and as he is walking back break out of windex and single step over there. because you really can't click the mouse. Run gets you back. and if you press num-lock it breaks you out.
There are a whole bunch of other commands for example type heap and this will tell you everything in memory this just tells you the scripts, the rooms, the costumes everything that is currently inside the pc's memory and it shows you the size. if you had picked up an object that was in inventory you would see something that said inventory. Type obj and this lists all the objects. It shows you their names, shows you their numbers, that little field to the right with the period and the ones those are your classes. so you can see what classes objects have, you can find out what state the objects are in, on the right it says whether they are in state 1 or state 2, the owner says who owns them and 15 means they are in the room. If one of the objects had been picked up by someone it might say owner one. you can send all your output to a file. If you type out and then a file name everything that appears on the windex screen will also be written to a file and then when you exit the program you can bring that file into brief. sometimes what we do when we are doing something complex you can press the star and it goes into trace mode, so if you type out foo and press * mode and play through a little section of the game the entire trace of that section is stored off in a file and then you can put it in brief and look at it. there is no way to wait-for-animation you just have to know you just say break here 10 which is an ok syntax but when you compile your program it will do 10 break here's. It is a macro. There is a way to turn boxes off you can tell it it is no longer a box but then you have to redo the whole connectivity matrix and there is a command which will let you do that on the fly but it is still being debugged. <noah> can you say stop-script me
no you can't use me within a script.close, off are 0 open and on are 1

cut-outs

You have .zb1,2,3,4 and those are the different z-planes that we use for clipping, to show depth. We can have up to 4 of these "Z planes" and that file contains that clipping information. With these it is possible for an actor to walk behind a desk or walk behind a tree because it is just a clip.

Chapter 3 Statements

Actor Related (3.1)

3.1.1 actor

The actor statement is used to initialize and change the facets of an actor. It is used when initially defining an actor and whenever an update is needed. The statement can be multiple lines if the situation warrants.

Syntax:

```
actor actor-name     [costume costume-name]
                                                    [sound sound-effect]
                                                    [color color-name is color-name]
                                                    [talk-color color]
                                                    [elevation number]
                                                    [walk-animation choreography]
                                                    [stand-animation choreography]
                                                    [talk-animation
choreography,choreography]
                                                    [init-animation choreography]
                                                    [step-dist x-coord,y-coord]
                                                    [name string]
                                                    [width number]
                                                    [default]
                                                    [scale number]
```

Example:

```
actor bobbin              costume bobbin-swim
actor indy          sound splat
actor donovan             color light-green is black
actor brody         talk-color light-grey
actor bobbin              elevation 0
actor bobbin              walk-animation swimming
actor bobbin              stand-animation treading
actor bobbin              talk-animation start-talking, stop-talking
actor bobbin              init-animation treading
actor indy          name "Indy"
actor indy          width 16
actor indy                scale 50 step-dist 16,12
```

Here are the individual components of the actor statement. When initializing an actor, you should use at least costume, name, and talk-color.

costume
Sets the current costume for the actor. The costume change, if any, occurs in the next frame.

sound
This is the sound that an actor can make during an animation sequence. Usually this would be the sound an actor makes when it walks, but it could be the sound of a german shepherd chewing or a shovel hitting the dirt. When the artist draws and choreographs the actor there is a special little code that is dropped into the choreography, and at the appropriate time it will trigger the sound effect which is defined with actor sound.

color
This component of the command substitutes one color for another within the actor's costume. For example, yellow is green would change everything usually drawn in yellow to green. This is appropriate for different video cards and for reuse of a costume such as the Nazi's hair and uniform in Indy 3.

talk-color
Talk-color is the color of the character's words, text, or speech when you see it on the screen. It is recommended that you use one of the 16 EGA palette colors for talk-color. Even with the upgrade of the system to 256 colors, still use the one of the 16 EGA colors. This will prevent extra work when producing a EGA version.

elevation
Actors automatically clip behind each other based on their vertical "y", position. This command can be used, to change the elevation of an actor, without changing its "y" co-ordinate. The automatic clipping will still use the base "y" position for its calculations. This can also be used to raise or lower an actor without worrying about walk boxes. The animation of the ladder in the Zepplin Hall of Indy 3 is such a situation.

walk-animation
When the actor walks, the animation listed within choreograph 2 is activated. Using this command allows you to use animation from a different choreography. This would be appropriate if the character developed a limp for example. The default location for the walk-animation is choreography 2 (see discussion of byle). Remember to use choreography names rather than numbers for code clarity.

stand-animation
The default location for the stand-animation is choreography 3 (see discussion of byle). When the actor is standing, the animation listed within choreography 3 is activated. Using this command allows you to use animation from a different choreography. Remember to use choreography names rather than numbers for code clarity.

talk-animation
Talk-animation has two parameters. The first is for start-talking and the second is for stopping that talking. The default location for the talk-animation is choreography 4 (see discussion of byle). When the actor is talking, the animation within choreography 4 is activated. When the lines go away, choreography 5 is used. This is usually a closed mouth cell. Using the talk-animation command allows you to use animations from different choreography cells. Remember to use choreography names rather than numbers for code clarity.

init-animation
The default location for the init-animation is choreography 1 (see discussion of byle). When an actor is put into a room using put-actor, the init-animation is activated. A costume change will also initiate init-animation. This command allows you to use animation from a different choreography instead of choreography 1. Remember to use choreography names rather than numbers for code clarity.

step-dist
Step-dist takes two parameters, x and y, which tell how many pixels the actor should be moved, with each step, when walking. The x controls walk steps left and right and y controls walk steps front and back. Step-dist is tied to the actor's scale.

name
Name sets the name that is seen on the screen, when the cursor touches the character (see .i.class-of:see also; GIVEABLE and TOUCHABLE.)

width
The actor's on-screen width in pixels. Defaults to 16. Very rarely will you use this, for actors of normal proportion The exception is very large actors (Biff in Indy 3) and wide actors (German Shepherd in Indy 3) Width is mainly used by the system for calculating in such scripts as follow-actor, where the size of the actor width is important. It has nothing to do with when the cursor is on the actor.

scale
The actor's scale can be set anywhere from 0 to 255. The 255 factor is the normal size of the actor. When scale is set to 0 (invisible), the step-dist is

automatically 0. The actor will not move but the system thinks it is trying to move. If invisibility is called for, use an "invisible" costume, not 0 scale.
Although the actor's step-dist works in conjunction with scale, it may be appropriate to change the step-dist if you change the scale, because the scaling perspective sometimes looks off.

default
Sets the actor command back to the system defaults. These are:

talk-color white
elevation 0
walk-animation 2
stand-animation 3
talk-animation 4, 5
init-animation 1
scale 255
step-dist 8,2
width 16

Here are examples of actual scumm system code. From these you can see the many different lengths and make-ups that the actor statement can take.

Examples:

```
actor bobbin        costume bobbin-swim elevation 0 step-dist 4,2\
                                    init-animation treading walk-animation
swimming\                                   stand-animation treading

actor indy          costume wet-indy-skin sound splat\
                                    talk-color yellow width 16 name "Indy"

actor brody         costume clip-board-brody sound footstep\
                                    talk-color light-grey width 16 name
"Marcus"

actor elsa          costume elsa-skin sound footstep\
                                    talk-color light-cyan step-dist 8,2 width
16\                                 name "Elsa"

actor bishop        color yellow is white

actor donovan       costume donovan-skin\
                                    step-dist 8,2\
                                    width 16 \
```

```
                                        talk-color light-blue\
                                        color light-blue is blue\
                                        color light-green is black\
                                        color yellow is white\
                                        color light-purple is black\
                                        name "Donovan"

actor sea-monk      elevation -108
```

Things to remember:

- Presently within the system there is no way to check what the current scale or name (string) are.

- It is possible to use define or variable names in place of any of the actor components.

  Examples:
  actor cauldron-actor elevation cauldron-elev
  actor selected-actor costume guy-qtip

3.1.2 class-of

v As of Mar 21 1991, the specific statement class-of, has been removed and incorporated into the actor statement.

Both objects and actors have classes associated with them. Actor classes are involved with clipping and box information.

Syntax:
class-of actor-name is actor-class [actor-class...]

Example:
class-of troll is IGNORE-BOXES-ON ALWAYS-CLIP-ON

There are 3 classes which have an on and off state.

always-clip-onloff.i.class-of:actor related:always-clip-onloff;
Setting an actor's class to always-clip-on will make an actor always clip, no matter what box it is in, and no matter where it is. If there is a z plane the actor will clip behind it. Even if the box says don't z clip the actor will do so anyway. Always-clip-off just turns that off.

never-clip-onloff.i.class-of:actor related:never-clip-onloff;
Setting an actor's class to never-clip-on will make an actor, no matter what the actor is doing, never clip behind anything. There is an off for this as well. A good example of this would be a bird, that is going to fly across the screen. You would not wish the bird to clip behind objects, because it is going to be floating in the air.

ignore-boxes-onloff.i.class-of:actor related:ignore-boxes-onloff;
Setting an actor's class to ignore-boxes-off will cause the actor to not pay attention to any boxes in the room. Instructing the actor to walk to an x,y coord will cause the actor to head directly to that point. This is useful for actor's such as birds and again there is an off for this as well.

Things to remember:

- Setting an actor's class to always-clip-off and never-clip-off will cause the actor to react, based on the boxes default status. However, if one of the classes is set to on, then it will override the boxes default. If both classes are set to on there is no guarantee what is going to happen. Probably the system will do one of them, with the likelihood that it will be the last one that the system checked.

- Whenever you set ignore-boxes-on it is advisable to set one of the clipping classes. The system does not know what to do with the actor, because it is not in a box. Therefore the actor is going to randomly clip based on what it was doing last. Usually, when you say ignore-boxes-on you will say never-clip-on as well.

3.1.3 come-out-door

This is used for room changes. An actor walks through a door (an object) in one room and comes out of a door(an object) in another room. The actor will be placed at the object's use-position.i.Use position:see also; facing the opposite direction. Come-out-door is equivalent to put-actor at-object,.i.put-actor [at-object] [at x-coord, y-coord] [in-room]:see also; usually into a new room, followed by a .i.camera-follow:see also; .i.selected-actor:see also;. The statement must be the last statement in an object, or local script as this launches the room exit code. Any other code within the object, or local script will not be implemented. Adding a walk statement to the same line is also possible. This is used to bring the actor further into the scene on a room transition.

Syntax:

```
come-out-door       object-name in-room room-name
come-out-door       object-name in-room room-name\
                                              [walk x-coord,y-coord]
```

Example:
come-out-door col-hall-outside-door in-room col-hall
come-out-door ghost-ship3 in-room hellcliff walk 114,111

Things to remember:

- Come-out-door updates the inventory and runs the appropriate exit and enter code.

- Remember not to use this in a local .i.cut-scene:see also;. The main reason is that a come-out-door changes the .i.current-room:see also;. This runs the exit code for the room you're in, and the enter code for the room you're entering. If you use a come-out-door within a local cut-scene, the script is killed when you exit the room, but the cut-scene status is unchanged. The .i.cursor:see also; will still be set to soft-off, the .i.userput:see also ;is still soft-off and the cut-scene state is 1. The cut-scene will never end. This will generate an error message saying an active cut-scene has stopped.

- Look at room-scroll (3.8.5) for an example of a problem with a negative position of a door.

3.1.4 do-animation [face-towards]

This command tells an actor which previously choreographed animation sequence, to perform. Sequences can be as simple as having the actor open his mouth, or as complex as jumping up in the air, spinning three times, and then flapping his arms.The face-towards section of the command causes an actor to face towards another actor. The actor will go through all the intermediate facings on the way.

Syntax:
do-animation actor-name choreography
do-animation actor-name face-towards actor-name

Example:
do-animation ghost-deck-crew flip-head
do-animation nazi face-towards indy

If animation sequences are strung together they must, usually, be broken up so that each frame is seen by the user. The example below shows the .i.break-here:see also; command as an effective tool for this purpose. Break-here is

based on frames. The command .i.sleep-for:see also;will also work, but this uses "real" time instead. This means that on a slow computer it is possible for the animation to take longer than was allowed in the sleep-for command.

Example:
```
do-animation indy punch-high
start-sound getting-hit
break-here
break-here
do-animation indy block-med
break-here
do-animation butler fall-down
```

Here is an example of multiple do-animation statements, which don't cover each other up, but work together. See Indy 3 file Radio-ro.scu. The do-animations set different parts of the radioman's body.

Example:
```
if (state-of radio is HERE) {
        do-animation radioman init-anim
        do-animation radioman on-radio
        do-animation radioman twist-knob
} else {
        do-animation radioman turn-back
        break-here
        do-animation radioman fix-radio
}
```

Things to remember:

- Each animation sequence must be choreographed with the animation frames ahead of time using byle. In most cases, complex sequences are built from simple sequences, and then chained together using multiple do-animations and break-heres.

- There are a number of directions that can be used with the do-animation statement. These are, face-front, face-back, face-left, face-right, turn-front, turn-back, turn-left and turn-right. The "face" commands position the actor immediately to the direction indicated while the "turn" commands show the turning stages of animation.

        Examples:
        do-animation hermit turn-left

do-animation meathook face-right

- Putting do-animation face-towards into a loop will allow an actor to continue watching someone as they move across the screen.

Example:

```
do {
        do-animation max face-towards sam
        break-here
}
```

- Wait-for-actor.i.wait-for-actor:see also;can be used with a turn-animation to wait until the turn is completed

3.1.5 put-actor [at-object] [at x-coord, y-coord] [in-room]

This will put an actor at a desired location. The put-actor at x-coord, y-coord command will put the actor at the coordinates given in whatever room they are in at that moment. Be careful, if the co-ordinates you select are not within a walk box, then the actor will be placed at the edge of the nearest walkbox. (Of course there is an exception to this. See ignore-boxes-on..i.class-of:actor related:ignore-boxes-onloff;)

Adding the parameter in-room will move the actor to the new room, and then place him at the indicated co-ordinates. However using in-room unnecessarily (when the actor is already in the room), especially in a loop, will slow the system down.

Adding the parameter at-object will put an actor at a desired location. The put-actor at-object statement lets you position an actor at the "use-position".i.Use position:see also; of an object. Objects are not valid unless they are in the .i.current-room:see also;. Therefore the actor must already be in the room with the object.

Syntax:

```
put-actor actor-name at x-coord,y-coord
put-actor actor-name at x-coord,y-coord in-room room-name
put-actor actor-name at-object object-name
```

Example:

```
put-actor indy at 438,113
put-actor grail-knight at 192,110 in-room grail-chamber
put-actor radioman at-object radio
```

Examples:
put-actor head in-the-void
put-actor indy at 575,112
do-animation indy face-right
put-actor henry at 540,118
do-animation henry face-right
put-actor brody at 518,111
do-animation brody face-right

put-actor selected-actor at 430,45
do-animation selected-actor face-back
put-actor stan at-object sea-monkey-ship
do-animation stan face-front

Things to remember:
- When you want to move an actor out of a .i.current-room:see also;, but don't want to put it in any other room yet, you can put it into a room called in-the-void. If the actor is performing an animation when you used put-actor, it will put the actor at the new location and continue the animation, unless the actor was moved to a new room.

- The actor will automatically face in the direction the object's .i.use position:see also; indicates.

- If the camera is already following an actor that you then move to a new room, the camera will remain in the current-room but will pan to the actor's new x-coord. You must issue a new camera command to avoid this.

3.1.6 stop-actor;

Used to stop an actor from walking. This automatically triggers do-animation.i.do-animation [face-towards]:see also; stand. Most useful when overriding a .i.cut-scene:see also;where characters may be walking around. Usually put-actor.i.put-actor [at-object] [at x-coord, y-coord] [in-room]:see also; is used to place the actor into place. However put-actor does not effect animation. If the actor was walking when the .i.override :see also;was hit, the actor would continue the walking animation in place. Stop-actor will make the actor stop walking.

Syntax:
stop-actor actor-name

Example:
stop-actor indy

Example:
walk indy to 190,120
do-animation brody turn-right
break-here 2
say-line brody "Indy, wait!"
break-here 3
stop-actor indy

Things to remember:
- Stops actor at the next frame.

3.1.7 wait-for-actor

This command will wait before running the next line of code until the actor gets to its destination or is stopped by the programmer. This works with walking and turn animation only , not other animations.

Syntax:
wait-for-actor actor-name

Example:
wait-for-actor butler

Example:

```
walk indy to 286,112
wait-for-actor indy
do-animation grail-knight turn-right
say-line grail-knight "I knew you would come."
wait-for-message
walk grail-knight to 250,112
wait-for-actor grail-knight
```

Things to remember:
- Wait-for-actor is a shorthand method (both in keystrokes and in heap space) of writing the following:

break-until (not actor-moving(indy))

The .i.actor-moving:see also;function checks the exact state of the actor's motion. It takes up quite a bit more space—12 bytes instead of 2, for wait-for-

actor.

3.1.8 walk to [x-coord, y-coord] [actor [within]] [to-object]

This instructs an actor to walk to a specific location in the room. This will turn the actor to face the direction of the walk destination and then walk the actor to that location.

Adding the actor command walks an actor to the current x and y, coord position of another actor. If used in conjunction with within, this command tells the actor to walk within a certain number of pixels of the target actor.

Adding the to-object command walks an actor to the use-position.i.Use position:see also; of a touchable object, and faces him in the direction indicated by the use-position. The actor must be in the same room as the object.

Syntax:
walk actor-name to x-coord,y-coord
walk actor-name to actor-name within number
walk actor-name to-object object-name

Example:
walk donovan to 473,107
walk indy to vogel within 24
walk butler to-object cas-entryway-front-door

Examples:
walk indy to 293,108
sleep-for 30 jiffies
walk guard to 325,108
wait-for-actor guard

walk henry to-object temple-exit
break-here 4
walk indy to-object temple-exit
wait-for-actor henry

Things to remember:
- Walk to actor doesn't keep checking the current position of the target actor, it walks the actor to the target actor's coordinates as they were at the time the command was executed. So if a chase is required, this command will have to be placed inside a loop.

Example:

```
do {
        walk weird-ed to eds-temp within 2
        break-here
} until (proximity(weird-ed, eds-temp) <= 2)
```

- Remember that objects are only valid in the room they are defined.

Camera Related (3.2)

3.2.1 camera-at

Sets the horizontal camera position in a room. The value indicates the midpoint of the screen.

Syntax:

```
camera-at x-coord
```

Example:

```
camera-at 160
```

Example:

```
if (actor-x selected-actor > 320) {
        camera-at 480
} else {
        camera-at 160
}
```

Things to remember:

- 160 is the center of a 320 pixel wide screen so, camera-at 160 will position the camera at the far left side of a multi-screen room.

3.2.2 camera-follow

Tells the camera to follow a specific actor around the room. Once set, it will continue to follow the actor within that room. If this actor is not the .i.selected-actor:see also ;and goes into another room, you must change to that room, using either .i.current-room:see also; or .i.come-out-door:see also; The camera will then continue to follow the actor in the new room. Any other camera command will disable camera-follow. If the actor used as the parameter is in another room

then the system will fade out of the current-room, and fade into the room in which the actor resides.

Syntax:
camera-follow actor-name

Example:
camera-follow selected-actor

Here is an edited .i.cut-scene:see also; from Indy 3 that illustrates the use of camera-follow within a room, in this instance Vogel's office. Note that the camera-follow dog is deactivated by the .i.camera-pan-to:see also; statement in this example and not by the camera-follow .i.selected-actor:see also; statement that follows.

Example:

```
cut-scene (no-verbs) {
        camera-follow dog                               ;camera-follow initiated
        wait-for-actor dog
        walk dog to 370,111
        wait-for-actor dog
        sleep-for 1 second
        put-actor dog at 390,111
        do-animation dog face-left
        walk dog to 340,111
        wait-for-actor dog
        break-here 3
        start-script bak dog-behavior
        sleep-for 5 seconds
        camera-pan-to 160                               ;This first deactivates
        break-until (camera-x = 160)        ;camera-follow dog
        camera-follow selected-actor        ;Not this!
}
```

Things to remember:

- Camera-follow is important when a room is bigger than one screen or when there are multiple actors that can be used by the player (i.e. Indy and Henry in Indy 3 game)

- Sometimes camera-follow won't show all of the picture because the .i.selected-actor:see also;can't walk far enough to the left or right to trigger the automatic scrolling. For example the left side of the village in Monkey 1 would never be seen normally so there's a script that artificially forces the camera all

the way over if the player gets close enough. The script is called auto-scroll and is within village.scu in Monkey 1.

Example:

```
script auto-scroll {
        break-until (actor-x selected-actor > 268)
        do {
                break-until (actor-x selected-actor < 260)
                camera-pan-to 160
                break-until (actor-x selected-actor > 268)
                camera-follow selected-actor
        }
}
```

3.2.3 camera-pan-to

Tells the camera to smoothly pan to a new position in the room. The current position can be watched using the .i.camera-x:see also; system variable.

Syntax:
camera-pan-to x-coord

Example:
camera-pan-to 100

Example:
camera-pan-to 300
break-until (camera-x >= 216)

Things to remember:

- This statement can cause problems. The .i.wait-for-camera:see also; statement should be used in conjunction with camera-pan-to, as the wait-for-camera statement waits until the camera system is stable. You don't want the following code to run before the camera gets to its pan position.

- Make sure that when the player enters into a dialog in a scrolling room that you either use the .i.say-line:see also; overhead statement or that the camera is repositioned so that the actors are more or less centered on the screen. This looks better and prevents ugly word wrapping. When the dialog is over, do a .i.camera-follow:see also;.i.selected-actor:see also;.

- Be aware of possible lock-ups due to a .i.break-until:see also; .i.camera-x:see also; never becoming true. Make sure you are checking for a value that the

camera-pan-to will hit. Camera-x increments by 8 so make sure you check a value which can be returned.

3.2.4 fades

The fades statement controls the screen turns transition during the next room change. There are 4 basic fades, none, iris, pixel and snap, which have both and up and a down version. These are used in combination to generate the 6 fades that the scumm programmer uses. The fades statement does not run a fade. The statement sets the fade and the system remembers this fade until a new one is set.

Syntax:
fades fade name

Example:
fades pixel-down

While they are pretty self explanitory, here are the 6 fade choices and what they contain:

cross-pixel.i.fades:cross-pixel; (none-down + pixel-up)
This fade allows you to pixel directly into a new room.

cross-iris.i.fades:cross-iris; (none-down + iris-up)
This fade irises down and then jumps you directly into the new room.

cut-to.i.fades:cut-to; (none-down + snap-up)
This fade jumps straight to the new room.

iris-to-black.i.fades:iris-to-black; (iris-down + iris-up)
This fade is the standard. The screen irises down in the old room and irises up in the new room.

cut-to-black.i.fades:cut-to-black; (snap-down + snap-up)
This fade jumps to black and then jumps to the new room.

pixel-to-black.i.fades:pixel-to-black; (pixel-down + pixel-up)
This fade pixels down to black and then pixels up in the new room. A nice effect.

3.2.5 wait-for-camera

Similar to .i.wait-for-actor:see also; in that a wait-for-camera command will wait before processing the next line of code until the camera has stopped moving.

Syntax:
wait-for-camera

Example:
wait-for-camera

Example:
walk selected-actor to 280,130
camera-pan-to 320
wait-for-camera

Things to remember:
- This statement is nearly always used with the .i.camera-pan-to:see also; statement to make sure that the camera has reached its destination before the resumption of code processing.

Flow Control (3.3)

3.3.1 case

The case construct is provided to make code more readable. It is actually converted to a series of if statements by Scumm. The case statement must have at least one "of" <value> statement.
Default and otherwise are the same. Either one can be placed as the last choice in the case statement and will be used if all the other options are negative.

Syntax:

```
case case-name {
        of number {
                [statements]
        }
        [of number {
                [statements]
        }]
        [defaultlotherwise    {
                [statements]
```

```
        }]
}
```

Example:

```
case word-message {
        of 0 {
                say-line "In Olde English"
        }
        of 1 {
                say-line "In Latin"
        }
        of 2 {
                say-line "In Etaskrit"
        }
        of 3 {
                say-line "In Porcinum-Atinlay"
        }
        of 4 {
                say-line "In Tuskin"
        }
        of 5 {
                say-line "In Vaachi"
        }
}
```

Here is an example of the use of "default" within the case statement. This is taken from line 429 of Indy 3 - fighting.scu.

Example:

```
case indy-action {
        of block-high {
                if (punch-angle) {
                        punch-angle = punch-low
                } else {
                        punch-angle = punch-med
                }
        }
        of block-med {
                if (punch-angle) {
                        punch-angle = punch-low
                } else {
                        punch-angle = punch-high
                }
```

```
        }
        of block-low {
                if (punch-angle) {
                        punch-angle = punch-high
                } else {
                        punch-angle = punch-med
                }
        }
        default {
                punch-angle = punch-high
        }
}
```

Things to remember:

- If you are using a "compile iflendif" (example: #if MAC) statement within a case statement then it cannot be placed between the case and of portion of the code. Similarly the #endif to terminate is also placed within the "of" structure.

```
case (case-name) {
        of (# choice) {
                ; compile if (#if) goes here.
                ; compile endif (#endif) goes here
        }
}
```

- It is also possible to do nested compile if's.

3.3.2 do [until]

The do construct is used to repeat a section of code. It will repeat indefinitely until the script it is in is stopped, either by a .i.stop-script:see also; or, if it is in a local script, by a room transition.

The do until command is used to repeat a section of code until a condition is satisfied.

Syntax:

```
do {
        [statements]
}
```

```
do {
```

```
        [statements]
} until condition met
```

Example:

```
do {
        for i = start-frame to end-frame ++ {
                draw-object i
                break-here
        }
}

do {
        [statements]
}until (proximity indy henry <= 26)
```

Here is a good example of a constantly running do loop.

Example:

```
        do {
                sleep-for 1 second
                if (nazi-energy < nazi-health) {
                        nazi-energy += nazi-recovery-rate
                }
                if (nazi-energy > nazi-health) {
                        nazi-energy = nazi-health
                }
                if (indy-energy < indy-health) {
                        indy-energy += indy-recovery-rate
                }
                if (indy-energy > indy-health) {
                        indy-energy = indy-health
                }
                start-script print-energy
        }
```

Things to remember:

- The following symbols and expressions are available =, !=, <, >, <=, >=, ==, is, are, and is-not.

- There should be an interrupt, .i(.break-here:see also;, .i.sleep-for:see also;, or .i.wait-for:see also;, somewhere inside the bracketed code or the computer will lock up

This is correct:

```
do {
        draw-object which-lights
        sleep-for delay jiffies
        state-of which-lights is R-GONE
        sleep-for delay jiffies
}
```

This would cause a lock-up:

```
do {
        draw-object which-lights
        state-of which-lights is R-GONE
}
```

Things to remember continued:

- An exception to the above is if you intentionally want to wait until a condition is met and you are sure this will happen soon. Here is an example of an exception from line 28 of Indy 3 - bookburn.scu.

```
script fire-glow {
        local variable glow, rnd, last-rnd
        do {
                do {
                        rnd = random 2
                }       until (rnd is-not last-rnd)
                last-rnd = rnd
                case rnd {
                        of 0 {
                                glow = red
                        }
                        of 1 {
                                        glow = light-red
                        }
                        of 2 {
                                        glow = yellow
                        }
                }
                palette glow in-slot light-purple
                break-here
        }
```

}

Note that while the do loop has a .i).break-here:see also; within it, the do until loop within the do loop does not. This can be done as long as the time spent within the do until loop is limited.

• Do until loops can be embedded within other do until loops or within a do loop.

```
do {
        do {
                [statements]
        } until
} until

do {
        do {
                [statements]
        } until
}
```

3.3.3 do-sentence

Normally, when a player clicks on a .i.verb:see also; and an object on the screen, a sentence is constructed within the system. The do-sentence command lets the programmer artificially construct a sentence which will execute the sentence as if the player had controlled it. Do-sentence is most useful when a command must execute code already defined in another command.

Syntax:
do-sentence verb-name object-name

Example:
do-sentence open alarm-exit

Examples:
do-sentence pick-up envelope
do-sentence use video-game1 with quarter

Things to remember:
• Do-sentence can also be used with current-noun1 and current-noun2 to save having to carry around a lot of code that's attached to a specific object.

• It's possible to string a series of do-sentences together. Each one is placed on the stack, and then they are all executed, one-by-one, in reverse order.

• Note that the use of a preposition in do-sentence is an optional defined value which Scumm throws away. It is included for increased readability. Other prepositions which are defined are in and on.

3.3.4 for

This command is similar to for loops in other languages. The first variable (var-1) is initialized with the second variable (var-2). The system continues through the loop and on each pass, variable one is incremented (++) or decremented (--) until it equals the third variable (var-3), at which point it cancels the loop.

Syntax:

```
for var-1 = var-2 to var-3 ++ | -- {
        [statements]
}
```

Example:

```
for foo = dialog-1 to dialog-9 ++ {
        verb foo off
}
```

Nested for loops are permitted in Scumm. The following example is taken from Bar.scu in Monkey 1.

```
for k = 0 to 1 ++ {
        for j = 0 to 2 ++ {
                obj = ((spin-guy-1 + j) + (k * 3))
                draw-object obj
                obj = (chain-1 + j)
                draw-object obj
                break-here
        }
}
```

Things to remember:

• There must be a space between the step increment (++, --) and the following brace.

This is correct:

```
for k = 0 to 1 ++ {

This is incorrect:
for k = 0 to 1 ++{
```

3.3.5 if [else]

This is pretty self-explanatory to anyone who knows any other structured language. Complex conditions using OR or AND can not yet be constructed. The condition within the parenthesis can be a single parameter such as "if (met-sheriff) etc or a more involved condition such as the examples below.

Syntax:

```
if condition met {
        [statements]
}
if condition met {
        [statements]
} else {
        [statements]
        }
```

Example:

```
if (to-whom is bill) {
        jump give-to-bro
}
if (sank-my-own-ship) {
        walk selected-actor to 87,98
        wait-for-actor selected-actor
        start-script cliff-hermit-dialog
} else {
        walk selected-actor to 124,98
        wait-for-actor selected-actor
        start-script cliff-crew-dialog
}
```

The if condition can take many different forms.

```
if (not second-time-on-melee) {
if (noun2 is root-beer) {
if (where-am-i < 320) {
if (actor-x cook > 310) {
```

```
if (button == 2) {
```

Things to remember:
- Nested if statements are allowable.

```
if (item >= '0') {
        if (item <= '9') {
                return-value is (item - '0')
        }
}
```

3.3.6 jump

Used to jump from one section of code to another within a script or an object's code.

Syntax:
```
jump label-name
```

Example:
```
jump open-mein-kampf
```

Example:
```
wait-for-awhile:
        break-until (!script-running leader-dialog)
        start-script three-second-egg-timer
        delay = (((random 20) + 30) * 60)
        sleep-for delay jiffies
        if (script-running leader-dialog) {
                jump wait-for-awhile
        }
        if (script-running kitchen-door-clicked) {
                jump wait-for-awhile
        }
```

Things to remember:
- A label must have a colon after it, but don't include the colon in the actual jump statement.

- Jump is used extensively in dialogs, and is used to simulate AND and OR statements. Currently the following instruction is not legal in Scumm.

break-until ((foo=10) OR (hell-freezes-over))

but it may be simulated as follows.

```
do {
        if (foo = 10) {
                jump escape
        }
        if (hell-freezes-over) {
                jump escape
        }
        break-here
}
escape:
```

• Never jump out of a .i.cut-scene:see also; where you bypass the cut-scene close brace. This is because a cut-scene launches a number of scripts that control the .i.cursor:see also ;and .i.userput:see also; and a cut-scene must exit properly for the cursor and userput to be reset correctly.

• Scumm convention is for label names to be placed flush left.

• Labels are global within a room and therefore each must have a unique name in that room.

3.3.7 override

Used to jump over long sections of a game, especially cut-scenes when a special key (usually escape) is pressed by the player. This lets the player skip long dialogs or long non-interactive sections of the game.

Syntax:
override label-name

Example:
override skip-ghost-story

The override-hit system variable can be checked before exiting a .i(.cut-scene:see also; to see if override was used to get to that spot. It will be true if an override was hit.

Example:

```
cut-scene     {
        current-room logo
        override skip-opening                   ; Can hit override
                                                                        ; after logo co
        break-until(return-value)
        break-until(music-flag > 22)
        return-value is 0
        fades cross-pixel
        current-room train
        break-here
        break-until(return-value)
        break-until (!sound-running indy-theme)
skip-opening:
        if (override-hit) {
                stop-music indy-theme
        }
}
```

Things to remember:

- If the override key is hit the override is turned off, and the override-hit flag is set. If you reach the override label without the key being hit the override is still on. This can be dangerous! If the player hits override now, sputm will jump back up to the label. Overrides are cleared when the cut-scene in which the override is located ends, and with override off. Therefore don't put any commands that may cause a break after the override label but before you either end the cut-scene or say override off.

- Note that if an override is used in a cut-scene its label must be located within the same cut-scene. Don't use the override command to jump out of a cut-scene, bypassing its closing brace.

Things to remember continued:

- When the escape is hit the override will immediately take over and jump to the label. This is dangerous because the escape key may be hit at any time after the override command. It is possible that scripts, sounds and animations will have started that need to be terminated. Make sure that all conditions that get set in the .i).cut-scene:see also ;are properly reset after the override is hit. Use an .i.if (override-hit):see also; to check and turn off as appropriate.

```
if (override-hit) {
        print-line " " ;erases text
        actor selected-actor costume guybrush-skin
        do-animation selected-actor stand
} else {
```

```
        override off
}
```

• It is possible to have several overrides in a cut-scene. Only the last one executed will be used.

3.3.8 quit

The quit command exits the player back out to DOS.

Syntax:
quit

Example:
quit

Things to remember:
• The quit statement is only used for copy protection and exiting a game.

```
        if (copy-tries >= 3) {
                jump bail-out-of-the-game
        }
bail-out-of-the-game:
        quit
```

3.3.9 restart

Used to restart the game from the beginning. It clears all variables, terminates all running scripts, empties the heap, and starts the boot-script over again.

Syntax:
restart

Example:
restart

Things to remember:
• This may be used after the end credits while in a loop waiting for a key press.

3.3.10 stop-sentence

Terminates the sentence script and clears the .i.do-sentence:see also; stack.

Syntax:
stop-sentence

Example:
stop-sentence

Example:

```
script move-and-get-clobbered {
        break-until (actor-moving selected-actor)
        stop-actor selected-actor
        stop-sentence
        hit-guybrush-again
}
```

Things to remember:

- If stop-sentence is used in an object before a break.i.break-here:see also; .i.break-until:see also;it may cause a quick-verb to remain highlighted. The solution is to set the variable last-obj = -1 after you do a stop-sentence. See the shimmer-effect script in nav-dialog.scu in Monkey 1 for an example of this.

```
cut-scene quick-cut {
        stop-actor selected-actor
        stop-sentence
        last-obj = 65535 ; -1 now that negatives can be used
```

3.3.11 wait-for-sentence

Waits until all do-sentence's.i.do-sentence:see also; in the stack have been executed.

Syntax:
wait-for-sentence

Example:
wait-for-sentence

Example:

```
if (actor-box selected-actor < near-platform-box) {
        do-sentence walk-to crossing-near-post
```

```
        wait-for-sentence
} else {
        if (actor-box selected-actor >= 7) {
                if (actor-box selected-actor <= 8) {
                do-sentence walk-to crossing-far-post
                wait-for-sentence
                }
        }
}
```

Heap Management (3.4)

All the commands in this section move things onto or off of the heap. This is all done automatically during normal game play. So why do we need these commands? Mainly to pre-load something in preparation for a smooth running .i.cut-scene:see also;. We don't want the disk to be accessed in the middle of someone's speech or an exciting animation.

The contents of the heap can be listed using Windex.

3.4.1 clear-heap

Clear-heap removes all UNLOCKED items that are not currently in use such as a room, script, or sound. It does not UNLOCK or remove UNLOCKED items. Its use is to clear the heap of old outdated items so you'll have more room to pre-load upcoming items.

Syntax:
clear-heap

Example:
clear-heap

Example:

```
script final-showdown {
        local variable s,j,x
        unlock-script final-showdown
        cut-scene {
                current-room melee
                clear-heap
                load-lock-script fly-from-a-to-b
```

```
        load-script spin-the-bottle
        load-costume spin-skin
        load-costume up-chuck
        load-costume bottle-skin
}
}
```

3.4.2 load-

Moves the appropriate parameter from a disk to the heap. For example, load-costume, loads an actor's costume (all his/her animation frames and sequences) onto the heap.

Syntax:

load-costume	costume-name
load-room	room-name
load-script	script-name
load-sound	sound-effect
load-music	music-score
load-charset	charset-name

Example:

load-costume	henry-dead-skin
load-room	zep-hall
load-script	butler-dialog
load-sound	crash-sound
load-music	lechuck-theme
load-charset	fat-font

Things to remember:

- If the item you are loading is not locked onto the heap before, you run the risk of it being thrown off the heap as soon as the next item is loaded.

3.4.3 lock-

Sets a flag which prevents an item from being removed from the heap once it's loaded in. Without this command, it would be possible to .i.load:see also; something, and then have it tossed out as soon as something else was loaded. Make sure you unlock the item when it doesn't need to be in memory any longer.

Syntax:

lock-costume	costume-name

lock-room room-name
lock-script script-name
lock-sound sound-effect
lock-music music-score

Example:
lock-costume hole-indy-skin
lock-room beach
lock-script edna-chases-kid
lock-sound kid-screams
lock-music lechuck-theme

Things to remember:
- It is still possible to have a locked item thrown off the heap if the heap is full of locked items. This will generate a run-time warning, if Windex is running. However locking it means that all unlocked items will be thrown off first. So remember to unlock items when they are no longer needed.

- If something is already loaded, it will not be loaded again. If something is locked but not loaded (yes, it is possible) it will become locked on the heap, as soon as it is loaded.

3.4.4 load-lock-

The load-lock command is the primary command for placing items from disk onto the heap. Even loading and locking an item does not guarantee that it will stay on the heap, but it will only be removed after all unlocked items are removed.

Syntax:
load-lock-costume costume-name
load-lock-room room-name
load-lock-script script-name
load-lock-sound sound-effect
load-lock-music music-effect

Example:
load-lock-costume sheriff-skin
load-lock-sound door-open
load-lock-room cu-brush
load-lock-script see-parrot

Things to remember

- Remember to unlock items when you don't needed them.

3.4.5 nuke-charset

A charset is the only scumm item that will not be removed by the system, even if it is unlocked and no longer in use. Nuke-charset removes a .i.charset:see also; from the heap.

Syntax:
nuke-charset charset-name

Example:
nuke-charset fat-font

3.4.6 unlock-

Unlocks the item on the heap but doesn't remove it. The next time space is needed on the heap for something, the item might then be removed, as long as it's not being used in the .i.current-room:see also;.

Syntax:

```
unlock-costume      costume-name
unlock-room         room-name
unlock-script  script-name
unlock-sound              sound-effect
unlock-music              music-score
```

Example:

```
unlock-costume      nurse-edna
unlock-room               ednas-room
unlock-script       edna-chases-kid
unlock-sound              kid-screams
unlock-music              lechuck-theme
```

Things to remember:

- The statement .i.clear-heap:see also; will remove all unlocked items not in use.

Interface and Screen (3.5)

3.5.1 cursor

The cursor statement allows you to remove and disable the cursor during cut-scenes. The soft-on/soft-off are for embedded cut-scenes.

Syntax:
cursor on|off|soft-on|soft-off

Example:
cursor on
cursor off
cursor soft-on
cursor soft-off

When a .i.cut-scene:see also; starts, the system decrements the .i.cursor:see also; flag and when the cut-scene ends, the system increments the cursor flag. Any time that the cursor is a positive number the player can use the cursor. The number 0 or a negative number will deactivate the cursor.

In this example the cursor does not become active after the end of cut-scene two but waits until cut-scene one has finished.

Example:
```
cut-scene one {              ;cursor set to 0
        cut-scene two {      ;cursor set to -1
        }                                          ;cursor set to 0
}                                                  ;cursor set to +1 and is
"on"
```

3.5.2 delete-verb

Removes the .i.verb:see also; from the screen and the heap.

Syntax:
delete-verb verb-name

Example:
delete-verb travel-1

Things to remember:
- Look in main-scripts.scu in Indy 3 for a detailed use of delete-verb.

3.5.3 draw-box

Draw-box paints a colored box on the screen. This can be used for bar graphics such as the fighting levels in Indy 3, or the lines on the travel maps, made by drawing a series of small boxes in a line. The first x, y coord is the upper left-hand corner of the box and the second (x), (y) coord is the lower right-hand corner.

Syntax:
draw-box x-coord,y-coord to x-coord,y-coord color color-name

Example:
draw-box 0,155 to 320,185 color black

Things to remember:

- This statement is frequently used to remove a .i.print-text:see also; statement from the screen.

- There is a problem with the draw-box statement during saving and loading. If a game is saved after a box is drawn, when it is reloaded, the drawn box will not appear on the screen since the state of the screen is not saved to disk. We kludged around this in Indy 3 by disabling save/load during fight scenes.

3.5.4 palette

The palette statement allows the programmer to change colors within the 16 color EGA palette. This is needed for different machines' graphic capabilities and for fast color cycling animation such as the rotating propellor on the Zepplin in Indy 3.

Syntax:
palette color-name in-slot color-slot-name

Example:
palette light-red in-slot light-magenta

Here are the names and slot locations of the EGA palette.

black	= 0
blue	= 1
green	= 2
cyan	= 3
red	= 4

magenta = 5
purple = 5
brown = 6
light-grey = 7
dark-grey = 8
light-blue = 9
light-green = 10
light-cyan = 11
light-red = 12
peach = 12
orange = 12
light-magenta = 13
light-purple = 13
yellow = 14
white = 15

Things to remember:

- It is possible to do "palette shifting" to get such special effects as the lava flow in Monkey 1. Here is the way we "palette shifted" the neon sign at Hook Island in Monkey 1.

```
do {
        palette yellow in-slot light-magenta
        palette light-red in-slot light-green
        break-here 2
        palette light-red in-slot light-magenta
        palette yellow in-slot light-green
        break-here 2
}
```

3.5.5 set-screen

The screen size of a room is initially set in the rooms.ifo file to extract the proper screen size from the dpaint file. The set-screen command allows the programmer to set the size of the room during the game.

Syntax:
set-screen number to number

Example:
set-screen 0 to 144

Here for example, the enter code is used to set the screen for the one room and then the exit code returns the screen back to the regular size.

Example:

```
enter {
        set-verbs save-normal-verbs
        set-screen 0 to 200
        start-script react-to-parrot
}
exit {
        set-screen 0 to 144
        set-verbs normal-verbs
}
```

3.5.6 shake on/off

The shake on statement simulates an earthquake (or explosion etc) by having the screen shake. This may not be implemented on all platforms since it must take advantage of the computer's special hardware.

Syntax:
shake on|off

Example:
shake on
shake off

Things to remember:

- This is not limited by any time frame and will continue indefinitely until deactivated by the shake off statement.

3.5.7 userput

The userput statement allows the programmer to activate and deactivate the keyboard and mouse.

Syntax:
userput on|off|soft-on|soft-off

Example:
userput on

```
userput     off
userput     soft-on
userput     soft-off
```

Things to remember:

- This command is used in the system level scripts, cut-scenes.i.cut-scene:see also;,enter and exit scripts, dialogs and to set up unusual interfaces. It is rarely used in "everyday" scripting.

3.5.8 verb

The verb statement controls the look of the interface. The verbs are initialized in the main-scripts.scu and can be updated during game play.

Syntax:

```
verb verb-name      [at x-coord,y-coord]
                                        [new]
                                        [color color-name]
                                        [dimcolor color-name]
                                        [hicolor color]
                                        [key key name]
                                        [on]
                                        [off]
                                        [dim]
                                        [name "string"]
```

Example:

```
verb open                   at 100,171
                                        new
                                        color green
                                        hicolor yellow
                                        dimcolor dark-grey
                                        key key-f
                                        on
                                        off
                                        dim
                                        name "turn on"
```

Here are the individual components of the verb statement:

at
Allows you to place a verb at an x and y-coord or at a macro position, such as

vpos11, which has been previously defined, as the x and y position of the verb.

new
This needs to be used when a verb is first initialized.

color
This is the regular color of the verb. The default color is green. Sometimes when palette shifting is involved (see .i.palette:see also; (3.5.4)) such as in Monkey 1 Hell scenes, then the color of all the verbs will be changed, along with the rest of the palette.

dimcolor
Dimcolor is used to show a verb on the screen even though it is untouchable. The default dimcolor in is dark grey.

hicolor
Activated when the cursor moves over a "touchable" verb. The default hicolor in is yellow. The verb automatically returns to its normal color when the cursor is off the verb.

key
Sets the keyboard key that is equivalent to a mouse click on the verb. will

on/off/dim
These set the visual state of the verb. Off removes it from the screen, dim leaves it on the screen, in the dimcolor, but "untouchable", and On activates the verb.

name
This is the name the user will see on the screen when the verb is visible. It also allows you to change the verb's name.

Things to remember:

- A maximum of 255 verbs can be associated with any given object.

- The usable colors names for color, dimcolor and hicolor are defines. The equivalent numbers may be used instead of the color name.

- Multiple verbs can be used within a single verb definition

Message Handling (3.6)

3.6.1 charset

The charset command sets the current character set being used. The system will load the charset onto the heap if it is not already there.

Syntax:
charset charset-name

Example:
charset fat-font

Here is an example of a character set being loaded in for a scene and then being removed from the heap.

Example:
```
load-charset upside-down-font
charset upside-down-font
start-dialog
verb dialog-1   new color green hicolor yellow
verb dialog-2   new color green hicolor yellow
dont-say-dialog-lines is true
set-dialog 1 "?rehtom ym uoy erA  .nibboB m'I"
set-dialog 2 "            ?temleh ym s'erehW"
charset fat-font
nuke-charset upside-down-font
```

3.6.2 print-line

The print-line statement is used to print out a message to the screen at an x and y coord or at a default location. The message will normally stay on the screen for a short amount of time and then erase. An actor's.i.actor:see also; talk animation is not activated (as it is with .i(.say-line:see also;). This is used for system messages or messages from non-actors (e.g., PA system or when there are no actors present.).

Syntax:
```
print-line      [left]
                                [overhead]
                                [color color-name]
                                [center]
                                [at x-coord, y-coord]
                                [clipped number]
                                ["string"[,]|[+]|[:]|[-]]
```

Example:

```
print-line      left
print-line      "       Har Har Har"+
print-line      color light-blue "Har Har"
print-line      overhead
print-line      at 160,40
print-line      center
print-line      clipped
```

The left, color, overhead and center parts of the statement set the print-line default in preparation for a string to print to the screen. The system will apply the default to all subsequent print-lines until the deafaults are changed.

The clipped part of the statement prints only that number of pixels. Print-line clipped 74 "This is a test" will print only 74 pixels.

There are a few characters that can be embedded in messages:

^ will produce ...
@ is a non printing character
` will produce "

Placing a "+" at the end of a print-line will keep the message on the screen awaiting a further print-line, which is printed next to the first.

Placing a "-" at the end of a print-line will add the next print-line directly after. There is no space generated by this, so you must place one in the print-line to avoid words runningtogether! This is used in conjunction with autowrapping text.

Placing a ":" at the end of a print-line will cause the lines to be printed on top of each other with a timed pause.

Placing a "," at the end of the print-line will cause the 2nd line to be printed below the first.

Things to remember:

- This will erase any previous print-lines or say-lines.i).say-line:see also; but not print-texts.i.print-text:see also;. For this to happen, you need at least one space in the quotes. Also print-line " " will shut everybody's mouth. This should be done before special .i.case:see also; animations, and after .i.cut-scene:see also; overrides.

• It is possible to string print-lines together by using the "+" character. The following is taken from line 353 within Bardialog.scu in the game Monkey 1. The plus signs allow the 3 pirates to all laugh at the same time with the "Har Har's" all on one line.

• print-line will wait for the camera to stop moving before printing.

Example:

```
for j = 1 to 3 ++ {
        print-line left at 1,30 color light-blue "Har Har Har"+
        print-line left color yellow "       Har Har Har"+
        print-line left color light-purple "       Har Har Har"
        break-here 3
        print-line " "
        break-here
}
```

• The color of the message can be optionally specified. The default color is white.

3.6.3 print-text

The print-text statement is used to print out a message to the screen at an x and y coord or at a default location. Print-text has nothing to do with actors. The text stays on the screen until the room is exited.

Syntax:

```
print-text      [left]
                                [overhead]
                                [color color-name]
                                [center]
                                [at x-coord, y-coord]
                                [clipped number]
                                ["string"[,]|[+]|[:]|[-]]
```

Example:

```
print-text      left
print-text      "       Har Har Har"+
print-text      color light-blue "Har Har"
print-text      overhead
print-text      at 160,40
print-text      center
```

print-text clipped

The left, color, overhead and center parts of the statement set the print-text default in preparation for a string to print to the screen. The system will apply the default to all subsequent print-texts until the deafaults are changed.

The clipped part of the statement prints only that number of pixels. Print-text clipped 74 "This is a test" will print only 74 pixels.

There are a few characters that can be embedded in messages:

^ will produce ...
@ is a non printing character
` will produce "

Things to remember:
• The color of the message can be optionally specified. The default color is white.

3.6.4 say-line

This command is used exactly like .i.print-line:see also; except it causes the speaker's mouth to animate. If an .i.actor:see also; name is omitted, then the .i.selected-actor:see also; speaks. Each actor is assigned a text color using the actor command.

Syntax:
say-line [actor-name] [center]
[at x-coord,y-coord]
[clipped number]
["string"[[,]l[+]][:]

Example:
say-line dr-fred "Heh, heh, heh!"

Things to remember:
• .i.print-line:see also; " " will shut everybody's mouth. This should be done before special .i.case:see also; animations, and after .i.cut-scene:see also; overrides.

• Say-line will wait for the camera to stop moving before printing.

3.6.5 wait-for-message

This command causes a .i.break-until:see also; the current message has finished being displayed.

Syntax:
wait-for-message

Example:
wait-for-message

Example:
say-line sandy "My friends will save me!"
wait-for-message
walk dr-fred to 30,70
wait-for-actor dr-fred
say-line dr-fred "That's what she thinks!"

Things to remember:
- Wait-for-message is equivalent to:

break-until (message-going == false)
    (* message-going is a scumm system variable)

Object Related (3.7)

3.7.1 class-of

This statement is used to allow the classification of objects. Each object can be a member of any (or none) of the twenty-four possible classes, with class 0 being the default (not a member of any class). Both objects and actors (see 3.1.2) have classes associated with them. Object classes are set up within the object code with the expression .i.class is:see also;. They are changed throughout the game with the class-of statement..

Syntax:
class-of object is object-class

Example:

class-of head-bighead-door is UNTOUCHABLE

Things to remember:

- All class names should be capitalized.
- The list of 24 classes available can be found in Appendix F

3.7.2 dependent-on

Used when the appearance of an object is dependent on the state of another. For example, a can of Pepsi inside a refrigerator. If the door is closed, the Pepsi would be invisible.

Synatx:
dependent-on object-name being object-state

Example:
dependent-on refrigerator-door being OPEN

3.7.3 draw-object [at x-coord, y-coord]

Draw-object allows us to incorporate screen animations using multiple objects. The object can either be drawn at its original position (defined in flem), or a new x,y location can be specified. Note that the object's state 0, the one drawn into the background, should not be used in animation cycles since changing its state will cause an entire screen redraw (too slow for animation).

Syntax:
draw-object object-name
draw-object object-name at x-coord,y-coord

Example:
draw-object gh-hull-flame1-1
draw-object hell-obj at 100,0

Here is an example of cycle animation of a ghost's hand in Monkey 1.

Example:

```
do {
        draw-object ghost-hand1
        break-here 4
```

```
        draw-object ghost-hand2
        break-here 4
        draw-object ghost-hand3
        break-here 4
        draw-object ghost-hand2
        break-here 4
}
```

Here is the way we did a random draw-object for the students in Indy's office.

Example:

```
do {
        which-frame = (random 2 + start-frame)
        draw-object which-frame
        delay = ((random 7 + 3) * delay-constant)   ; 3-10 units
        sleep-for delay jiffies
}
```

Things to remember:

- Draw-object just blasts the image onto the screen background. It does not matter what was there before. Animation sequences that are constantly moving are ideal for draw-object and not .i.state-of:see also;. State-of is a very sophisticated object image manipulator which takes time.

- This command is also useful for making up new versions of rooms which have different features (pseudo rooms.i.pseudo room:see also;). For example, the same door object could appear in a different place on the screen in each version of the room. You would then have to position these objects every time you enter the room (using the enter code) or they will revert back to their originally defined location.

- Remember that the x-coord and y-coord are byte boundaries, so 40 is the right hand edge.

3.7.4 name is

The name is command is used at the beginning of every object definition to provide the name printed on the sentence line, when the object is touched. If the name is command is to be changed during the game, be sure to pad the name with enough @'s. This is so the name's definition contains the maximum number of characters that it will ever need

Syntax:

name is "object-name"

Example:
name is "cooking pot"

Things to remember:
If the object is never to be touched, the name string can be null.

3.7.5 new-name-of

Used to change the name of an object. The "string" name is only the name the player sees on the screen. When you refer to the object in the program, use the unquoted object name (object-name).

Syntax:
new-name-of object-name is "string"

Example:
new-name-of full-river is "river"

The name of an object is only successfully changed if it is in the current room or on the heap as an object the user has picked up. If the room with the object is re-loaded from disk, the name will revert to the original name. The solution is to use enter code to set the names of all objects with multiple names. This way, every time the room is visited, it will have the correct names.

Example:

```
script name-microwave-objects {
        case glass-jar-filled {
                of empty {
                        new-name-of jar-in-microwave is "empty jar"
                }
                of with-water {
                        new-name-of jar-in-microwave is "jar of water"
                }
                of with-developer {
                        new-name-of jar-in-microwave is "jar of developer"
                }
        }
}

enter {                         ; code run on room entry
```

```
        start-script name-microwave-objects
}
```

Things to remember:

- The length of the new name can not exceed the length of the original object name. To get around this, just pad the original names with @'s to the length desired. This sets up the appropriate storage to allow the new name later. There is no error checking for this, so be very careful. Exceeding the alloted size will cause part of the object's code to be overwritten, yielding unpredictable crashes. You can evoke the brief macro to count the characters in a string with escape-C.

3.7.6 owner-of

Used to assign ownership of an object to an actor. This only works on objects that have been "picked up" and are on the heap.

Syntax:
owner-of object-name is actor-name

Example:
owner-of coins is henry

Ownership may also be assigned to one of the following:

nobody.i.owner-of:nobody;
It was picked up and is still on the heap but nobody currently has it in their inventory. Nobody is defined as 14.

nuked.i.owner-of:nuked;
Thrown off the heap. Nuked is defined as 0.

in-the-room.i.owner-of:in-the-room;
The object is in the room. Not-being-held can also be used instead of in-the-room. They accomplish the same and both are defined as 15.

Things to remember:

- If you wish to take something out of the inventory, to be picked up again later, you first have to nuke it to remove the object code from the heap, and then set owner-of object to in-the-room. If the object is not nuked, it will still leave your inventory but when you pick it up again, you now have two, (and then three, then

four...). This was the way Stan's business cards were done in Monkey 1. See the example in .i.pick-up-object:see also; (3.7.5)

- Only actors one through thirteen (1-13) can own an object.

3.7.7 pick-up-object

This command picks up an object, puts it into the selected-actor's inventory (sets ownership), automatically sets the state of the object on the screen to GONE, and makes it UNTOUCHABLE. The most common usage of this command is within the pick-up verb:

Syntax:
pick-up-object object-name

Example:
pick-up-object phone

Example:

```
object toothbrush {
        name is "toothbrush"
        verb pick-up {
                pick-up-object me
        }
}
```

Note the use of the name ".i.me:see also;" instead of the object's name. "Me" is the same as .i.current-noun1:see also;. The command could have also been written,

pick-up-object toothbrush

and it would have the same result.

Things to remember:

- As a rule it is best not to use "me" in scripts. The use of "me" can be valuable when the same script is called as a subroutine from within several different object definitions.

- If you wish to take something out of the inventory, to be picked up again later, you first have to nuke it to remove the object code from the heap, and then set .i.owner-of:see also; object to .i.in-the-room:see also;. If the object is not nuked, it will still leave your inventory but when you pick it up again, you now

have two, (and then three, then four...). This was the way Stan's business cards were done in Monkey 1.

```
if (business-card-counter < 5) {
        ++business-card-counter
        owner-of stans-business-card is in-the-room
        pick-up-object stans-business-card
        state-of stans-business-card is HERE
}
```

- If you wish to pick up an object for an .i.actor:see also; other than the .i.selected-actor:see also; then use the following:

```
pick-up-object cheese
owner-of cheese is mouse
```

3.7.8 start-object

Launches the code within an object/verb immediately, rather than having the .i.actor:see also; walk over to the object first as part of a .i.do-sentence:see also; execution.

Syntax:
start-object object-name verb verb-name

Example:
start-object gen-jungle-exit-l verb walk-to

Things to remember:
- This statement is particularly useful when there are no actors in a room or when you want to have an actor do the same thing from different places on the screen.

3.7.9 state-of

Sets the state of an object. There are only two states, 0 (the background) and 1 (the object's second state from the object's screen).Only one state can be set within each state-of command. Use multiple commands to set statesof additional states. State-of can set the state of an object regardless of the object's location. The object does not have to be in the current room.

Syntax:

state-of object-name is state

Example:
state-of kitchen-entryway-door is CLOSED

There are a number of different state expressions used. They all accomplish the same thing but are there for ease of understanding.

The following are the settings of state 0.

HERE
CLOSED
R-OPEN (reverse-open which is same as CLOSED)
OFF
R-GONE (reverse-gone which is same as HERE)

The following are examples of setting the state to 1.

OPEN
GONE
R-CLOSED (reverse-closed which is same as OPEN)
ON
R-HERE (reverse-here which is same as GONE)

Things to remember:

- State-of is very slow, because state-of deals with the proper rules of displaying objects on the screen. They may be completely hidden by another object or they could be overlapping another object. For example, you open a cabinet with a flashlight inside. Pick up the flashlight and when you close the cabinet the system will not try to draw the flashlight on top of the cabinet.

Draw-object just blasts the image onto the screen background. It does not matter what was there before. Animation sequences that are constantly moving are ideal for .i.draw object:see also; and not state-of. Use draw-object when you don't care about the rules and just want the image drawn as fast as possible.

Room Related (3.8)

3.8.1 current-room

Changes the camera to a new room, and irises it in. In the process it causes the exit code from the last current room to be run and then the enter code in the new current room. The new room is automatically loaded off the disk if it isn't already on the heap. The costume of any .i.actor:see also; already placed in the room is also loaded in.

Syntax:
current-room room-name

Example:
current-room driveway

Things to remember:
- Use with caution. This command closes the room that you are presently in. All local scripts and code in objects are stopped (one of the reasons you don't use current-room in a local .i.cut-scene:see also;).

3.8.2 lights [are]]beam-size]

Used to set the lights in the room or a flashlight. The three values used for the room are dark, lights-dim, and bright. When lights are set to lights-dim, the values set by lights beam-size is used to create a flashlight.

Lights beam-size is used to set the size of the flashlight beam. The first value is for its width (in pixels), and the second its height. If the second number is omitted, then the beam is a square.

Syntax:
lights are light-status
lights beam-size is width [,height]

Example:
lights are dark
lights are lights-dim
lights are bright
lights beam-size is 2
lights beam-size is 5,6

Things to remember:

• Note that due to the fact that "dim" (in the .i.verb:see also; statement) is a reserved word, lights-dim must be used.

• Lights are independent of an actor, so if an .i.actor:see also; using a flashlight scales, the flashlight will not.

• The lights beam-size command has not been used since Zak McKracken, although it is still active. Lights, and the flashlight in particular, will be undergoing revision real soon now.

3.8.3 pseudo-room

The pseudo-room statement allows a number of similar rooms to be built off of one room rather than using many actual rooms. The pseudo-rooms are declared in the boot.scu.

Syntax:

```
pseudo-room        base-pseudo-room-name is pseudo-room-name\
                                            [pseudo-room-name]
```

Example:

```
pseudo-room beach is beach-2 beach-3 beach-4
```

Within the enter code of the base pseudo room is a .i.case:see also; statement that sets up the room depending on which pseudo room was entered. In the example below, an actor entering the beach-2 room would launch the "of beach-2" portion of the case statement in the beach.scu room (the base room).

```
case selected-room {
        of beach-2 {
                set-box 2 box-invisible
                for foo = rocks-cliff-1 to rocks-cliff-3 ++ {
                        state-of foo is R-HERE
                }
                if (!got-beach-2-memo) {
                        state-of beach-note is R-HERE
                        class-of beach-note is TOUCHABLE
                        load-lock-script add-memo
                        load-lock-script get-next-memo
                }
        }
```

And this would then continue for "of beach-3" and "of beach-4"

3.8.4 room-color

The room-color statement changes one color within a room to another color. This is used to accommodate different graphic modes, and setting up pseudo rooms, so the same art can be made to look different.

Syntax:
room-color color-name in-slot slot-name

Example:
room-color is white in-slot light-grey

Example:

```
room "diary-window" diary-window {
        enter {
#if IBM
                if (graphics-mode is CGA-MODE) {
                        room-color is white in-slot light-grey
                }
        }
#endif
}
```

Things to remember:

- An actor's.i.actor:see also; colors are not effected by this statement.
- This statement is usually found in the enter code.
- A useful statement to convey "mood", such as a sunset or sunrise.
- When changing colors, the original slot name never changes.

room-color is white in-slot light-grey

Would have to be followed by this to change it back to light-grey,

room-color is light-grey in-slot light-grey

- The next time the actor enters the room, the rooms colors will revert to its default (original) colors.
- Do not use for color cycling animation effects. Room-color causes the entire screen to be redrawn and it is too slow for animation. Use .i.palette:see

also; instead

3.8.5 room-scroll

Used to restrict the scroll area, where the camera will be able to scroll in, of a room. Values given are the minimum, and maximum x-coord of the camera.

Syntax:
room-scroll is minimum-x maximum-x

Example:
room-scroll is 480 488

Things to remember:

- The following is a kludge to fix a scumm bug involving negative actor-x positions, and .i.come-out-door:see also;. The system gets confused when a door is situated in the negative range (i.e.off the screen) and the system starts the new room around camera position 300 and then pans across the screen to the point that the actor just entered. It's actually two kludges, because we don't have OR statements right now. This is at line 736 of the sword-master.scu in Monkey 1.

```
        if (last-room is melee) {
                jump the-kludge
        }
        if (last-room is damnforest-9) {
the-kludge:
                room-scroll is 160 160      ;prevents the room scrolling
                break-here
                room-scroll is 160 304      ;ok to scroll now
        }
```

3.8.6 set-box;

The set-box statement allows the programmer to change the box path that an .i.actor:see also; can follow. You would normally use this where a fence (or other such obstacle) initially blocked an actors path. When the obstacle is removed, you would change the setting of the appropriate walk box.

Syntax:
set-box box-number box-status

Example:
set-box 5 box-normal

There are a number of different types of box status. These are:

box-normal:.i.set-box:box-normal;
The normal state of a box. You would use this class to return a changed box to the norm.

box-flip-x:.i.set-box:box-flip-x;
Normally when an .i.actor:see also; is walking to the right it is facing right but if you say flip-walk-x the actor will actually walk to the left and do a little moon walk

box-flip-y:.i.set-box:box-flip-y;
Normally the camera looks down on a screen and the actor coming toward the screen is forward and going away from it is back. However in some instances it is desirable to change this. For example, there is a scene in loom, at the dock, where the user is actually looking at the dock from below. In this case coming toward the screen is up and going away from the screen is down, therefore we have to flip the y orientation of the actors.

Individual boxes can be flip x and flip y. In the loom example, we didn't set bobbin to be flip x or flip y, we set the box that is on the dock to be a flip y box. Whenever bobbin stepped into that box he would always face the opposite direction that the system thought he should and then it looked right.

box-player-only.i.set-box:box-player-only; [invisible] [locked]:
box-player-only is used as a prefix for the box classes box-player-only-invisible and box-player-only-locked.

In Monkey we used box-player-only in the store. Guybrush is set to the class "player-only" when he enters the store, and then the boxes behind the counter are set to box-player-only-invisible, which indicates that this box is invisible only to actors who are class "player-only."

box-locked: .i.set-box:box-locked;
The box-locked class sets the box so that if the character is outside a locked box, it can't get in, and if the character is within the locked box, it can't get out.

box-invisible: .i.set-box:box-invisible;
Probably the most useful class, this pretends that a box doesn't exist. For example on the overhead map views of Monkey Island, boxes 1-8 could be on land, and boxes 9-15 could be in the water. If the character enters on foot, you would set all the water boxes to box-invisible so you can't walk on the water, but

if the character is in the row boat the land boxes are all turned box-invisible to prevent the character rowing over land.

Setting a box invisible doesn't remove it from the connectivity matrix. It is possible to walk through an invisible box on your way to a visible box, but you cannot stop in the invisible box, nor get to the invisible box by clicking directly on it.

Things to remember:
- The box number can be an actual number or a variable.

Script Related (3.9)

3.9.1 chain-script

Loads and executes a new script, placing it in the slot of the calling script, thus stopping the calling script as well.

Syntax:
chain-script [bakIrec] script-name [(] [<value16>[,]…] [)]

Example:
chain-script ed-to-front-door

Things to remember:
- Chain-scripts can be background or recursive

3.9.2 cut-scene

A cut-scene is a computer controlled sequence which is embedded inside a script or object. During a cut-scene, the player has no control of the game (other than, usually being able to terminate the cut-scene prematurely by pressing a special key), and the user interface is removed from the screen. All other scripts running during a cut-scene are temporarily halted. (Yes there are exceptions. See .i.bak:see also;..i.start-script:bak:see also;)

Syntax:
cut-scene [type]{
    [statements]
}

Example:

```
cut-scene {
        do-animation selected-actor get-up
        break-here 20
        actor selected-actor costume guybrush-skin
        build-sentence-script = build-sentence
}
```

Other than this basic type of cut-scene, there are three kinds of cut-scene that have been used in our recent games. The parameters are just flags passed to the enter/exit cut-scene scripts. New flags can be defined by the Scumm programmer as neccessary.
Here are the enter and exit scripts.

```
script start-cut-scene level {
        local variable i
        cut-scene-level += 1
        cursor soft-off
        userput soft-off
        if (level is-not quick-cut) {
                stop-sentence
                revert-sentence
                display-sentence
                last-obj = -1
                if (last-double-verb) {
                        verb last-double-verb color green
                        if (interface-color-sceme is demonic) {
                                verb last-double-verb color red
                        }
                        last-double-verb is 0
                }
        }
        if (level is no-verbs) {
                save-verbs start-normal-verbs to last-verb set\
                                                        cut-scene-set
        }
        if (level is no-verbs-no-follow) {
                save-verbs start-normal-verbs to last-verb set\
                                                        cut-scene-set
        }
        ;if (level is quick-cut) {
        ;do nothing
```

```
        ;}
        freeze-scripts
}
script end-cut-scene(level) {
        cut-scene-level -= 1
        cursor soft-on
        userput soft-on
        if (level is no-verbs) {
                restore-verbs start-normal-verbs to last-verb set\
                                                        cut-scene-set
        } else {
                if (level is no-verbs-no-follow) {
                        restore-verbs start-normal-verbs to last-verb set\
                                                        cut-scene-set
                }
        }
        unfreeze-scripts                                        ;This was changed
to fix a bug
                                                                                ;in Mor
        if (level is-not quick-cut) {
                stop-sentence
                revert-sentence
                display-sentence
                ;update-special-verbs
                if (level is-not no-verbs-no-follow) {
                        camera-follow selected-actor        ;WARNING!!!
                                                                                ;This c
                }                                                               ;last st
        }                                                                       ;the en
}                                                                               ;room s
```

The 3 basic kinds of cut-scene are:

no-verbs.i.cut-scene:no-verbs;
This removes the interface during the cut-scene. On exiting the cut-scene the camera is following the selected-actor.

```
cut-scene no-verbs {
        walk selected-actor to-object lookout-cliffside-door
        wait-for-actor selected-actor
}
```

quick-cut.i.cut-scene:quick-cut;

This leaves the verbs on the screen but still turns off the .i.cursor:see also; There is no .i.camera-follow:see also; .i.selected-actor:see also; at the end.

```
cut-scene quick-cut {
        do-animation selected-actor reach-low
        stop-script bubble-soup
        start-sound running-water
        sleep-for 1 second
        stop-sound running-water
        start-script bak bubble-soup
        do-animation selected-actor stand
}
```

no-verbs-no-follow.i.cut-scene:no-verbs-no-follow;
This component of cut-scene was put in place because of a problem in Monkey 1. It is the same as the no-verbs section with the exception that .i.camera-follow:see also; .i.selected-actor:see also; is not implemented.

```
cut-scene no-verbs-no-follow {
        override skip-cliff-via
        camera-at 160
        current-room the-void
        charset big-font
        sleep-for 30 jiffies
        print-line color light-red left at 5,5 \
        "After some more furious paddling^"
        wait-for-message
        skip-cliff-via:
}
```

Things to remember:

- Do not use the statements .i.come-out-door:see also; or .i.current-room:see also; in a local cut-scene.

- Parenthesis around the cut-scene identifiers are optional.

- A cut-scenes must always exit through its end (closing brace).

3.9.3 freeze-scripts

The statement is called by the .i.cut-scene:see also; statement and freezes all scripts except background scripts (see .i.bak:see also; .i.start-script:bak:see also;within .i.start-script:see also;).This command is used in the system level

scripts, cut-scenes,enter and exit scripts, dialogs and to set up unusual interfaces. It is rarely used in "everyday" scripting.

Syntax:
freeze-scripts

Example:
freeze-scripts

Things to remember:
- Check with the Scumm God before using this.

- If you must use it, don't forget to eventually issue an .i.unfreeze-scripts:see also ;command.

3.9.4 start-script

Starts the execution of a new script, automatically loading it onto the heap if it's not already there.

Syntax:
[start-script] [bak][rec] script-name [(] [<value16>[,]…] [)]

Example:
start-script cook-microwave-dinner

Besides the normal way, a script can use two special commands:

bak:.i.start-script:bak;
This is a background script which is unaffected by the freezing of scripts. So these scripts will be running while cut-scenes.i.cut-scene:see also; are occurring. Use for background animation (fire flickering, twirling pirates or special "watchdog" scripts, like waiting for an actor to arrive at a specific coordinate.

rec:.i.start-script:rec;
This is a recursive script which can have multiple copies of itself running at the same time. If you have copies of a script running with the rec command, and then launch yet another copy but without rec, all copies will stop and just the one will run.

Things to remember:

- Remember that the system can only have 20 scripts running at one time.

- The term start-script is optional. The line of code cook-microwave-dinner will start that script.

3.9.5 stop-script

Stops the execution of a script, but doesn't remove it from the heap.

Syntax:
stop-script [script-name]

Example:
stop-script tentacle-chases-kid

Things to remember:

- If the name of the script is omitted, then the current script is stopped.

  Example:

```
walk hermit to-object head-island
do {
    break-here
    foo = (camera-x - (actor-x hermit))
} until (foo > 180)
put-actor hermit in-the-void
stop-script
```

- Scripts can be stopped at any point by this command. Therefore it may be necessary to reset graphic and other states in conjunction with this.

3.9.6 unfreeze-scripts

The reverse of the .i.freeze-scripts:see also; command. The statement is called by the .i.cut-scene:see also; statement and unfreezes all scripts except background scripts (see .i.bak:see also;.i.start-script:bak:see also; within .i.start-script:see also;).This command is used in the system level scripts, cut-scenes,enter and exit scripts, dialogs and to set up unusual interfaces. It is rarely used in "everyday" scripting.

Syntax:

unfreeze-scripts

Example:
unfreeze-scripts

Things to remember:
- Check with the Scumm God before using this.

Sound/Music Related (3.10)

The Sound/Music system is under revision even as I type. The basic concept of starting and stopping music will remain but features such as improved priority and volume control are being added.

3.10.1 start-music

Starts the execution of a song, automatically loading it onto the heap if it's not already there. The function, sound-running can be checked to see if the music has finished.

Syntax:
start-music music-score

Example:
start-music part-4-music

Example:
start-music main-title-theme
break-until (not sound-running(main-title-theme))

Things to remember:
- This may kill other pieces of music. It always will kill music and other sounds in scumm 3.0 or lower.

- The Scumm system supports the internal PC speaker, Adlib, Soundblaster, and Roland. Since the Roland is on a separate disk (on games prior to 1991), we always make the Roland sound separate sounds. For this reason when we start a Roland song, we use a different number. When we start shipping Roland on the same disks as the rest of the game, there will be no difference.

3.10.2 start-sound

Starts the execution of a sound, automatically loading it onto the heap if it's not already there.

Syntax:
start-sound sound-effect

Example:
start-sound clock-tick

Things to remember:

- This may kill other sounds, depending on the priority of the sounds involved and how many sounds the system can handle at once. Some sounds are looping that play until they are stopped while others are background sounds that will return after the present sound finishes.

- In Scumm 4.0 and lower, sound priorities, and whether it is a background sound, are set out in a program prior to compilation. These can not be set at run time.

3.10.3 stop-music

Stops music from playing, but doesn't remove it from the heap.

Syntax:
stop-music [music-score]

Example:
stop-music main-title-theme

Things to remember:

- The parameter is optional since at this time only one piece of music can be playing at a time.

3.10.4 stop-sound

Turns off a sound, but doesn't remove it from the heap.

Syntax:
stop-sound sound-effect

Example:
stop-sound running-water

Wait Related (3.11)

Scumm uses a multitasking system in which you, the script writer, determine when to break away from the current task and go on to another task. This is done with the .i.break-here:see also; command family. The procedure is to write a sequence of commands that you want executed together, then insert some sort of a break. All the stacked up commands will be completed, and the next script in the queue will get its turn to continue. Forgetting to include a break might cause certain commands to be executed on top of each other, and you won't see some of them happen. In the following example, only Message 2 will appear on the screen.

say-line "Message 1"
say-line "Message 2"

3.11.1 break-here

Simple break away command. Can be used to time animation sequences.

Syntax:
break-here [number]

Example:
break-here
break-here 4

Here is some typical code

do-animation brody turn-right
break-here
break-here
say-line brody "Indy, wait!"
break-here
break-here
break-here
stop-actor indy

```
break-here
break-here
do-animation indy turn-left
break-here
break-here
wait-for-message
walk indy to brody within 20
```

This could be written as follows

```
do-animation brody turn-right
break-here 2
say-line brody "Indy, wait!"
break-here 3
stop-actor indy
break-here 2
do-animation indy turn-left
break-here 2
wait-for-message
walk indy to brody within 20
```

Things to remember:

- Breaks are frame driven. That makes them useful in timing animation. Sleep-for.i.sleep-for:see also; uses real time and is not recommended for animation, because of the wide range of computer power on our target machines.

- It is possible to condense the break-here's into one statement when they follow each other. This is accomplished by placing a number after the break-here.

3.11.2 break-until

A combination of a break-here and a do-until.

Syntax:

```
break-until condition-met
```

Example:

```
break-until (! script-running done-with-head)
```

The following two examples are identical.

```
do {
```

```
        break-here
}       until (music-flag)

break-until (music-flag)
```

Things to remember:
- Both of the above are guaranteed to have at least one break executed.

3.11.3 sleep-for

Causes a timed break. Units can be any of the ones below.

Syntax:
sleep-for number [jiffy|jiffies] [seconds|second]\

Example:
sleep-for 3 seconds
sleep-for 30 jiffies

Example:

```
display-message:
        print-line "The game is over."
        sleep-for 5 seconds
        print-line "Please insert another quarter."
        sleep-for 5 seconds
        jump display-message
```

Things to remember:
- Sleeping time is based on real time not frames. So a slow machine might not be done with the animation when the sleep-for times out. Break-here's (3.11.1) should be used when exact timing is necessary because .i.break-here:see also; always waits exactly one animation frame.

- It is possible to say sleep-for .i.random:see also; jiffies. It is not possible to say sleep-for random seconds.

- Scumm internally converts all units to jiffies.

Chapter 4 Functions

Functions are used in Scumm to return a value which can then be stored in a variable, or used in a comparison with an if statement.

At this time, functions can not appear as parameters inside other functions.

Example:
walk wendy to-actor (closest-actor(wendy))

This is illegal. You have to write it in two steps:

temp = closest-actor(wendy)
walk wendy to-actor temp

Actor Related (4.1)

4.1.1 actor-box

The actor-box function returns the value of the walk box that the .i.actor:see also; currently occupies.

Syntax:
var=actor-box actor-name

Examples:
last-box = actor-box indy
if (actor-box selected-actor > 7) {
break-until (actor-box selected-actor > 7)

4.1.2 actor-costume

The actor-costume function returns the value of the costume the actor is wearing.

Syntax:
var=actor-costume actor-name

Examples:
if (actor-costume lechuck is-not chuck-punching) {
if (actor-costume jojo is monkey-skin) {

4.1.3 actor-elevation

The actor-elevation function returns the actor's current elevation, either positive or negative.

Syntax:

var=actor-elevation actor-name

Examples:
height = actor-elevation seagull
foo = ((actor-elevation special-effect) + 1)

4.1.4 actor-facing

The actor-facing function returns the actor's.i.actor:see also; current directional facing. This can be one of four defined facing's. These are actor-back, actor-front, actor-left, actor-right.

Syntax:
var=actor-facing actor-name

Examples:
head-facing = actor-facing imaginary-actor3
temp = (actor-facing selected-actor + 0xf8)
if (actor-facing selected-actor is-not actor-left) {
if (actor-facing dock-keeper is actor-back) {

4.1.5 actor-moving

The actor-moving function is used to return the status value of the actor's journey.

Syntax:
var=actor-moving actor-name

Examples:
foo = actor-moving felon1
if (!actor-moving cook) {

The function returns one of three values.

walking.i.actor-moving:walking;
The actor is still on his way to a destination.

stopped.i.actor-moving:stopped;
The actor isn't currently walking and he never reached his destination.

arrived.i.actor-moving:arrived;
The actor isn't walking and he did reach his destination.

Things to remember:
• Remember that .i.wait-for-actor:see also; (3.1.8) uses less memory than actor-moving.

4.1.6 actor-room

The actor-room function returns the number of the room the indicated .i.actor:see also ;is in.

Syntax:
var=actor-room actor-name

Examples:
if (actor-room cook is-not bar) {
if (actor-room pirate-num is the-void) {
leader-room = actor-room selected-actor

4.1.7 actor-width

The actor-width function returns the width of the the indicated actor.

Syntax:
var=actor-width actor-name

Examples:
dist = (actor-width jojo / 2)
give-prox = ((actor-width current-noun2 / 2) + 4)

4.1.8 actor-x

The actor-x function returns the x-coord of the indicated actor.

Syntax:
var=actor-x actor-name

Examples:
if = (((actor-x fishy) - x) / 30)
where-am-i = actor-x selected-actor

4.1.9 actor-y

The actor-y function returns the y-coord of the indicated actor.

Syntax:
var=actor-y actor-name

Examples:
foo = ((actor-y selected-actor) + step-size)
return-y = ((actor-y selected-actor) - 30)

4.1.10 closest-actor

The closest-actor function is used to find which .i.actor:see also; is closest to any other actor within the range specified by the system variables, .i.actor-range-min:see also ;and .i.actor-range-max:see also;. Closest-actor will return the value no-one (255) if there aren't any specified actors in the room.

Syntax:
var=closest-actor actor-name

Example:
temp = closest-actor(wendy)

4.1.11 find-actor

The find-actor function examines an x and y position and returns a value for the actor that it finds there. If no actor is present at the x and y position, the function returns a 0.

Syntax:
var=find-actor x-coord, y-coord

Example:
to-whom = find-actor cursor-x,cursor-y

4.1.12 proximity

The proximity function returns the distance between two actors.

Syntax:
var=proximity actor-name, actor-name

Example:
break-until (proximity stan selected-actor < 100)
turn-threshhold = ((proximity act special-effect) / 2)

Interface and Screen (4.2)

4.2.1 find-inventory

The find-inventory function returns the number of the inventory object at a specific slot of the named actor.

Syntax:
var=find-inventory actor-name, slot-number

Example:
obj = find-inventory selected-actor,sslot

4.2.2 inventory-size

The inventory-size function returns the number of items in the named actor's inventory.

Syntax:
var=inventory-size actor-name

Example:
inv-size = inventory-size selected-actor

4.2.3 valid-verb

The valid-verb function returns a true value if the .i.verb:see also; listed is within the object's definition, otherwise it returns a false value.

Syntax:
var=valid-verb object-name, verb-name

Example:
if (valid-verb current-noun1,give) {

Object Related (4.3)

The first two functions in this section are different from the rest in that they can only be used inside of if statements. You can not use them to return a value to a variable.

4.3.1 if (class-of

The if (class-of function is used to return the class of an object.

Syntax:

```
if (class-of object-name [==,!=] class-name) {
        [statements]
}
```

Example:

```
if (class-of meat == food) {
        say-line "I'm not hungry."
}
```

Things to remember:

- This statement can only be used inside of if statements. You can not use it to return a value to a variable.

4.3.2 if (state-of

The if (state-of function is used to return the state of an object.

Syntax:

```
if (state-of object-name [==,!=] state-name) {
        [statements]
}
```

Example:

```
if (state-of door == OPEN) {
        say-line "I should close this."
}
```

Things to remember:

- This statement can only be used inside of if statements. You can not use them to return a value to a variable.

4.3.3 find-object

The find-object function examines an x and y position and returns the number of the object that it finds there. If no object is present at the x and y position, the value returns a 0.

Syntax:

```
var=find-object x-coord, y-coord
```

Example:
var=find-object 12, 140

4.3.4 object-x

The object-x function returns the value of the x-coordinate of the use-position of an object.

Syntax:
var=object-x object-name

Example:
dest-x = object-x obj

4.3.5 object-y

The object-y function returns the value of the y-coordinate of the use-positionof an object.

Syntax:
var=object-y object-name

Example:
var=object-y scroll

4.3.6 random

The random function returns a random number from 0 to the number indicated, to a maximum of 255.

Syntax:
var=random number

Examples:
timer = random 60
fly-x = (random 160 * 2) ; even number from 0 to 320
fly-y = (random 100 + 40); number from 40 to 100

Things to remember:

- The sequence of random numbers will always remain the same unless the

programmer maintains the random number generator by calling it over and over while waiting for the player to do something.

Script Related (4.4)

4.4.1 script-running

The script-running function is used to tell if the specified script is currently running:

Syntax:
script-running script-name

Examples:

```
if (not script-running practice-boxing) {
        start-script bak boxer-in-ring
}
if (script-running walk-dock-keeper) {
        load-lock-script meeting-of-the-masters
}
```

Sound/Music (4.5)

4.5.1 sound-running

The sound-running function is used to tell if the specified sound is currently running:

Syntax:
sound-running sound-name

Examples:

```
if (not sound-running machine-gun) {
start-sound machine-gun
}

if (sound-running chase-music) {
```

Chapter 6 System Variables

6.1.1 actor-range-min

Resides at slot 15 actor-range Contains the lowest numbered actor which will be searched for using the closest-actor() function.

6.1.2 actor-range-max

Resides at slot 16 actor-range+1 Contains the highest numbered actor which will be searched for using the closest-actor() function.

6.1.3 actor-talking

Resides at slot 25 ; last actor who spoke

6.1.4 build-sentence-script

Resides at slot 32 ; script to run when button hit

6.1.5 camera-max

Resides at slot 18 Contains the maximum X position the camera can move to in the current room.Set by room-scroll.

6.1.6 camera-min

Resides at slot 17 Contains the minimum X position the camera can move to in the current room.Set by room-scroll.

6.1.7 camera-script

Resides at slot 27 ; called when camera moves -- out of sync with other scripts -- use for cool paralax effect

6.1.8 camera-x

Resides at slot 2 Contains the X position of the camera.

6.1.9 complex-temp

Resides at slot 0; used for complex expressions

6.1.10 current-lights

Resides at slot 9 ; state of lights in current room

6.1.11 current-disk-side
Resides at slot 10 ; side of disk current in drive

6.1.12 cursor-x
Resides at slot 20 ; x position of the cursor in the room

6.1.13 cursor-y
Resides at slot 21 ; y position of the cursor in the room

6.1.14 cut-scene1-script
Resides at slot 35 ; script to run on start of cut-scene

6.1.15 cut-scene2-script
Resides at slot 36 ; script to run on end of cut-scene

6.1.16 cursor-state
Resides at slot 52

6.1.17 entered-door
Resides at slot 38 ; contains object that actor came out of

6.1.18 enter-room1-script
Resides at slot 28 ; script to run before enter code

6.1.19 enter-room2-script
Resides at slot 29 ; script to run after enter code

6.1.20 exit-room1-script
Resides at slot 30 ; script to run before exit code

6.1.21 exit-room2-script
Resides at slot 31 ; script to run after exit code

6.1.22 frame-jiffies
Resides at slot 46 ; jiffies last frame took

6.1.23 graphics-mode
Resides at slot 49 ;

6.1.24 hard-disk
Resides at slot 51 ; 1 = playing from hard disk

6.1.25 jiffy1
Resides at slot 11

6.1.26 jiffy2
Resides at slot 12

6.1.27 jiffy3
Resides at slot 13

6.1.28 K-of-heap
Resides at slot 40

6.1.29 last-sound
Resides at slot 23 ; last sound started

6.1.30 machine-speed
Resides at slot 6 For IBM's and compatibles, value stored representing the CPU's speed.Can be used to time sequences differently for slow and fast machines.We arbitrarily set the cut-off point at 40 (anything greater than 40 is a "fast" machine).

6.1.31 message-going
Resides at slot 3 ; Contains 0 if there's no message on the screen, a 1 if amessage is still being displayed

6.1.32 me
Resides at slot 7 Same as current-noun1.

6.1.33 min-jiffies

Resides at slot 19 Contains the minimum number of jiffies between frames.

6.1.34 music-flag

Resides at slot 14 A flag that gets set by the score itself.It is usually incremented at the beginning of each measure.Used for synchronizing music with action

6.1.35 number-of-actors

Resides at slot 8 Maximum number of actors allowed.This must be set at the beginning of the game

6.1.36 override-hit

Resides at slot 5 ; Tells whether the override command was executed in the current cut-scene.0=not hit, 1=hit

6.1.37 override-key

Resides at slot 24 ; which key to use for override

6.1.38 pause-key

Resides at slot 43 ; store value of pause key

6.1.39 real-selected-room

Resides at slot 22 ; base room of pseudo room

6.1.40 restart-key

Resides at slot 42 ; store value of restart key

6.1.41 save-load-key

Resides at slot 50 ; Must be set for save/load to work

6.1.42 screen-x

Resides at slot 44 ; screen cursor coordinates

6.1.43 screen-y

Resides at slot 45

6.1.44 selected-actor
Resides at slot 1 Contains the value of the currently selected actor

6.1.45 selected-room
Resides at slot 4 Contains the currently displayed room's number.

6.1.46 sentence-script
Resides at slot 33 ; script to run from do-sentence

6.1.47 snap-scroll
Resides at slot 26 ; snap scroll flag

6.1.48 sound-mode
Resides at slot 48 ; 0=PC, 1=TANDY (don't use), 2=ADLIB

6.1.49 sputm-debug
Resides at slot 39 ; SPUTMDEBUG env set

6.1.50 sputm-version
Resides at slot 41

6.1.51 text-offset
Resides at slot 54 ; Height above actors feet (if negative) or abs position.

6.1.52 text-speed
Resides at slot 37 ; value from 0 to 255, determines delay
; on text messages

6.1.53 total-jiffies
Resides at slot 47 ; total jiffies of all frames

6.1.54 update-inven-script
Resides at slot 34 ; script to update inventory

6.1.55 userput-state
Resides at slot 53

Chapter 7 Errors

Chapter 10 Byle

10.1 Introduction

Byle is a versatile and powerful graphics tool that creates the actor animation for our Scumm Adventure Games. The Chapter is organized so you can use it as a guide or a reference handbook.

As with all the Scumm system tools, the animation editor is named after a disgusting bodily fluid. In this case Byle.

Byle allows the artist to prepare all the animation sequences used in a Scumm game, from the simplest walk animation, to a complex shot such as a character being fired out of a canon.

The basic component of byle is the cel. Everything that happens in byle revolves around this concept, usually by stringing these cels together to form a choreography.

10.1.1 Cel Definition

A cel is a single image or object within an entire sequence or animation that is intrinsically related to the image that comes before it and the one that follows to create a flow of movement.

10.1.2 Choreography Definition

A choreography is an action (walk, talk, reach etc) performed by an actor. This is formed by stringing cels together.

10.1.3 Choreography, Cel Interaction

Now we could spend a whole week describing the way these interact. In its simplest form there is a choreography, (a more descriptive term is animation sequence), that an actor can perform. The artist will set up this animation sequence by placing cels of the actor in the Choreography Cel list.

For example, in a Scumm game,the act of walking by an actor is a choreography

which is just repetitive. So when the system says walk it starts doing the walk choreography.

10.1.4 Choreography Directions

Actors in Scumm games can walk in any direction and therefore the walk choreography must contain four different representations of the actor's walking animation sequence. Not unreasonably these directions are front, back, left and right.

Other choreographies will have anywhere from 1 to 4 of these directions depending on their use in the game.

The choreographies can be fairly involved and complicated such as a guy exploding and his head blowing off.

10.1.5 Level Definition (formerly known as limbs!)

Byle has 16 different levels (or limbs) for cels. These levels are identified as A through P. The different levels allow an artist to superimpose one image over the top of another. Each subsequent level has a higher priority over the previous level. Most of the time you will only use the first couple of levels.

For those of you familiar with the technique, levels could also be described as like the different separations in a 4 color separation.

10.2. The Interface: An Overview

The byle screen is similar in many ways to dpaint. The menu bar, tool box, color indicator and palette have all been retained in a similar fashion.

The major changes from dpaint are the cels table, choreography cel list table, the lower screen buttons and the divided painting area.

10.2.1 Menu Bar

Byle has numerous features and functions that are available through a series of pull-down menus in the Menu Bar at the top of the screen. The menu bar is divided into 6 options.

The File option contains the following choices;
Load costume, save costume, save costume as, make costume, load (not implemented), load lbm palette, load lbm backdrop, write chores.def and quit.

Section 10.5.1 looks at these in more depth.

The Edit option contains the following choices;
Copy level (not implemented), copy cel, copy chore, copy list and paste.
Section 10.5.2 looks at these in more depth.

The Brush option contains the following choices;
Flip x, flip y, normal scale (not implemented), half scale (not implemented), scale up (not implemented), scale down (not implemented), set scale (not implemented), replace (not implemented), and color (not implemented).
Section 10.5.3 looks at these in more depth.

The Chores option contains the following choices;
Make a chore, next set of chores (which include the untitled choreographies 16 through 30), the standard first 6 choreograhies (Null, Init, Walk, Stand, Talk, Stop Talking) and the untitled choreographies 6 through 15.
Section 10.5.4 looks at these in more depth.

The Special option contains the following choices;
Insert (which contains the selections Cel, Sound, Empty and Flip), Palette, No flip on left, Verbose Output (part of Brad's debugging, ignore), Debug Cel States (also part of Brad's debugging, ignore), Remove unused cels (not implemented) and Onion Skin (not implemented).
Section 10.5.5 looks at these in more depth.

The Backdrop option contains the following choices;
Load lbm backdrop, show backdrop, save lbm backdrop and load wak file.
Section 10.5.6 looks at these in more depth.

10.2.2 Toolbox

The dpaint Toolbox has been modified to add more useful tools and delete tools that are not needed. When you start Byle the Freehand Brush tool is active by default. We will get to examine each of the components in section 10.6.

10.2.3 Palette and Color Indicator

The Palette contains the colors representing your current color spectrum. Directly above the Palette is the Color Indicator. The two rectangles display the colors that you are currently using to paint. The inner rectangle shows the brush color, while the outer rectangle shows the current background color. Unlike dpaint's defaults of black and white, byle's defaults are black and blue.
Section 10.6.22 looks at these in more depth.

10.2.4 The cel table

The cel table contains 16 levels labelled A to P which are accessible using the scroll bar on the left (see fig XXX). Each level has 64 available cels (numbered 00 to 63).
Section 10.3 looks at these in more depth.

10.2.5 The choreography cel list table

The choreography cel list table is similar to the cel table in that it has 16 levels available. These are also numbered A to P. However the choreography cel list table has only space for 32 cel placements.
Section 10.3 looks at these in more depth.

10.2.6 Painting Area/Animation Area

In byle, the screen is divided into two sections. The left screen is the painting area and the right screen is the animation area. It is possible to place backdrops in the right area, for example, to check that an animation flows correctly.

10.2.7 Lower Screen Buttons.

The lower screen buttons are chore, chore name, animation start/stop, animation cel by cel, xrel, yrel, directional and cel addition.
Section 10.5 looks at these in more depth.

Lastly, just as in dpaint, when a menu option has a Ë symbol to the right, it means that the option has a sub menu.

10.2. File requestor window

The file requestor window (see fig XXX) is activated by the load costume, save costume as, load lbm palette, load lbm backdrop, save lbm backdrop and load .wak file choices.

The File and Directories window
The File and Directories window displays (not surprisingly) the directories and files that are in the current directory.

To open a directory or load a file you must click on the file or directory name and then click on the load button. It is possible to open these by double clicking on the file or directory name, but that is a little temperamental.

Once you have clicked on a directory or file in the Files and Directories window, that file or directory name will appear in the edit button box. If a directory is opened then the path is displayed in the current directory window.

Load Button
The load button is case sensitive. The button will either load a file or open a directory depending on what type of item is activated.

Cancel Button
The cancel button closes the file requestor window.

Current Drive Button
The current drive button allows you to switch drives by left clicking on the button and moving down to the drive you want. Activating another drive will generate a new set of files and directories in the File and Directories window

Current Directory Window
The current directory window shows the current directory path (for example c:\scummu\costumes). If a directory is opened then the path is displayed in the current directory window.

Edit Button Box
The edit button box (for want of a better name!) displays the current file or directory highlighted in the Files and Directories Window. It also can be used to edit a file or directory name for the saving of a new file or copy.

While in the file requestor window, hitting a letter on the keyboard will scroll you through the files with that letter as the first letter in the file name. If no file exists then the system assumes you wish to be in the edit button box.

10.3 The Cel Table and Choreography Cel list Table

Section 10.7, will go through a costume in depth, but here is a brief look at the cel/choreography/cel list relation.

Cels
In figure 1 you will see where the cel section of byle is contained. As noted earlier, there are 16 levels or limbs for the cels. These run from A to P. As a convienence we cross reference the levels and cel numbers. Therefore the first cel in the 4th level would be D/00. Rarely will more than a couple of levels of cels be used.

Generally all the actors standard movement is drawn in the A level of cels. This would be the different cels that go to form walking for example. The other levels are used for cel drawings of reaching, talking and further special case animation.

Probably the easiest way to understand the relationship of the cel list table and cel choreography cel list table is to regard the cel list table as your cel palette.

Just as you can choose colors in a palette to use for painting, you use the cels in the cel list table to produce your animations in the cel choreography cel list table. Similarly with a palette, a cel may be used in more than one choreography.

Choreography
Again as noted earlier a choreography is a group of cels. That choreography must contain the cels for all four directions (front, back, left and right).

So with a Choreography of Walk you must set up a string (or list) of cels for Walk/back Walk/front Walk/left and Walk/right

For arguments sake let us assume that level A cel 00 through 03 were the walking front animation cels, 04 through 07 was the walking cels, etc. Then the artist would set up the choreography by activating the front button and then inserting the cels 00 through 03 on the level A of the choreography cel list table.

Setting up some simple cels
Let's now do that. Go to the choreography name button and click using the left mouse button on the button. Hold down the mouse button and move up to the 02 Walk choreography. Once that is highlighted then release the mouse button. The choreography button will change to say 02 Walk.

Activate the filled rectangle brush by clicking on it with the left mouse button. Next click on first cel in the A level, where the light grey 00 is located. Select a color from the palette by left clicking on any color that appeals and then go to the left hand display window and draw a rectangle of that color.

Next left click on the next blank cel to the right of the 00 cel and a 01 should appear and the rectangle that you just drew should disappear from the left hand display window. Select another color in the palette and draw a rectangle of that color, again in the left hand window display.

Repeat this process for cells 02 through 11. (Use the scroll bar to move over as needed) Each time pick a new color. Only use the left button to click on the cels.

Setting up a simple choreography
Now lets set up an animation sequence. Make sure that the directional button "front" is the active button. If not, then left click on that button first. Left Click on the A/00 cel (Level A, cel 00) in the cel table. Now left click on the first space in the choreography/cel listing area. You should see the 00 cel appear in that space.

You now need to add cels 01 and 02 to this list in the blank spaces to the right side of the 00 cel over in the choreography cel list table. Follow the instruction for

the 00 cel above.

Once you have the cels 00, 01 and 02 in the list, press the animate start/stop button and the animation should appear on the right hand window. In a very simple way this is all there is to byle.

Adding an additional choreography direction
Now lets add the back animation. Left click on the front button and while holding down the button move down to the back listing and once the back is highlighted then release the button. The back button should appear instead of the front button and the list of cels in the choreography cel list table should disappear.

Now insert cels 03-05 in order just as you did for the cels 00-02 when you were doing the front listing. Remember that the first cel in the list must be done by by clicking over in the choreography cel list area, but after that you can use the shift key and click to pass cels over. Click on the animation button to see these cels animate.

In a real costume, obviously, the 00-02 cels and 03-05 would have the appropriate walk front and walk back cels. The colored boxes are just for a quick example. Add cels 06-08 to the right direction and 09-11 to the left. Move back and forth between the different directional buttons and you will see the different cels that you have assigned to each.

Adding a new Choreography
Now let's add another choreography. Go to the choreography button and left click as you did before. Move up to the stand choreography and release the button. The choreography button should change to 03 stand and again you should see the list in the choreography cel list table, on the right hand side, will disappear. (The directional button, however will not change). Experiment with the 4 directions and setting up different sequences of animation cels. (you do not have to go in numerical order). If you feel confident, create some new cels with the ellipse fill brush.

Levels of animation
There is only one further basic component to understand. That is the concept of levels and levels of animation. The different levels are numbered A through P with each successive level having a priority over the previous one.

Let's show an example of this. Go back to the walk choreography and add an ellipse filled in a new color in the B/00 cel. Activate the front directional facing and start a new choreography list for level B. This should be below the 00-02 listing that you already have for level A. The 00 from the B level in the cel table should be placed on the B level in the choreography cel list table (you are unable to place it in another level anyway!), directly under the 00 on the A level in the choreography cel list table. Place this same 00 cel in the next two blank areas too.

The choreography cel list table should look like this.

00 01 02
00 00 00

Run the animation by activating the animation start/stop button and assuming that the ellipse is over the filled rectangle, then, you will get the ellipse shown fully with the changing filled rectangles behind it.

10.4 The Lower Screen Buttons.

10.4.1 chore

The "chore" button is a time saving button placed there for convenience. Left clicking the mouse button on this button will move the "chore name" button to the next numbered chore . Right clicking the mouse button on this button will decrement the "chore name" by one.

10.4.2 chore name

The chore name contains the number and name of the choreography that is currently active. Clicking either mouse button, pulls up a window from which a new choreography can be selected. The "chore" button will also move to the next chore for convenience.

10.4.3 animation start/stop button

The animation start/stop button activates the current choreography. Depending on the choreography cel list table, this will either be a one time animation or a cyclic animation.

10.4.4 animation cel by cel button

The animation cel by cel button moves through the current choreography cel list one cel at a time. This is useful, for example, for finding the one cel in an

animation that is out of sequence.

10.4.5 xrel button

Probably the most difficult aspect of Byle to understand is the use of the relative offset on the x and y axis. Reading the explanation below, the first time, will probably stump you. If it doesn't, could you please explain xrel to the rest of us? You will, however, understand it more once you have played with xrel when we look at the Indy costume in section 10.7.

Xrel effects all levels below it in the choreography cel list table not the cel list table. The fact that B/01 is xrel -1 means absolutely nothing to cel C/01 which is below it in the cel list table. For example you could designate cel 06 as xrel -1 and then all the levels below would also be affected at the time that 06 is run. If you have uneven chains i.e.

04 05 06
01 02

with 06 at -1 then alternate cels below would be effected by the xrel.

It is possible to have multiple x and y. For example one level could be set to -1, while two levels down a cel is set to -2. All the cels below that at the time of animation would be a cumulative -3.

10.4.6 yrel button

Probably the most difficult aspect of Byle to understand is the use of the relative offset on the x and y axis. Reading the explanation below, the first time, will probably stump you. You will, however, understand it more once you have played with yrel when we look at the Indy costume in section 10.7.

Yrel, just like xrel above, effects all levels below it in the choreography cel list table not the cel list table. The fact that B/01 is yrel 1 means absolutely nothing to cel C/01 which is below it in the cel list table. For example you could designate cel 06 as yrel 1 and then all the levels below would also be affected at the time that 06 is run. If you have uneven chains i.e.

04 05 06
01 02

with 06 at 1 then alternate cels below would be effected by the xrel.

It is possible to have multiple x and y. For example one level could be set to 1, while two levels down a cel is set to 2. All the cels below that at the time of

animation would be a cumulative 3.

10.4.7 directional buttons

A choreography can have up to four directions. A Choreography such as walking would usually have all four directions (front, back, right and left), while a special case animation might have only one direction.

In Byle, this direction can be changed one of two ways. The directional button can be activated by left clicking on the button and dragging down to the direction you want. Alternatively you can left click on one of the four directional buttons to the right of the choreography cel list table.

10.4.8 Cel addition button

The default setting for the cel addition button is cel. The cel addition button allows the adding of four different special case cels into the choreography cel list table. These special case cels are, sound, counter, empty and flip.

sound

The sound cel allows a sound to be inserted by the scumm programmer. The most common use of the sound cel is in the walk choreography where it is placed to allow for the sound of the actor's footsteps to be added. This is done by the actor sound component of the actor command (see page xxx).

counter

Not implemented at this time.

empty

Added by Brad after he tried to explain xrel to me. This provides you with your own padded cel.

flip

Not implemented at this time.

10.5 The Menu bar options

10.5.1 Pull down Menu: File

Figure X shows the different options available under this menu.

10.5.1.1 load costume

The load costume menu choice activates the file requestor window (see interface 10.2). By clicking on the filename and then clicking on the load costume button, costumes can be loaded. Costumes may also be loaded by double clicking on the costume file name.

10.5.1.2 save costume

This will save the costume that you have currently loaded. If you attempt to save a costume and have no costume loaded then the system will inform you with an error message.

10.5.1.3 save costumes as

Click on the save costume as choice and then click on the edit button box and type in the new name. If you do not add the .byl extension, the system will do it for you.

10.5.1.4 make costume

Makes a cos file so that the scumm programmer can use it. See scumm manual.

10.5.1.5 load

Not implemented at this time.

10.5.1.6 load lbm palette

The load lbm palette choice loads an lbm palette (palette only) and replaces the current palette.

10.5.1.7 load lbm back drop

The load lbm backdrop choice loads a picture into the backdrop (backdrop is the right hand window) to see if the animation works in relation to the background.

Note: The right hand window is also used for importing .lbm files from dpaint animator.

10.5.1.8 write chores def.

Not implemented at this time.

10.5.1.9 quit

The quit option returns the artist to the DOS prompt. You are given the option of

whether "you really want to quit."

10.5.2 Pull down Menu: Copy

Figure X shows the different options available under this menu.

10.5.2.1 copy level

This allows you to copy the entire level (with chore list and directions) down to another level. This choice is not implemented at this time.

10.5.2.2 copy cel

The copy cel choice allows you to copy a single cel. First click on a cel that you wish to copy and then activate the copy cel choice. Nothing will seem to happen. However click in the cel space where you wish to place the cel copy and then go up to the edit menu option again and select paste cel. The cel will then be placed in the space you designated.

10.5.2.3 copy chore

The copy chore choice allows the artist to copy the list (in 4 directions) of cels in the choreography cel list table but not the cels in the cel table. The copy chore choice does not copy the chore name either.

10.5.2.4 copy list

The copy list choice allows the artist to copy the one current level and direction that is active.

10.5.2.5 paste

The paste choice is context sensitive and changes as each copy is activated. For example, if the copy cel choice has been activated, then the paste choice will be paste cel.

10.5.3 Pull down Menu: Brush

Figure X shows the different options available under this menu.

10.5.3.1 flip x

This works just like dpaint. Using the grab brush, you take part of the costume (or all if you wish) and flip x will rotate it on the x axis.

10.5.3.2 flip y

This works just like dpaint. Using the grab brush, you take part of the costume (or all if you wish) and flip y will rotate it on the y axis.

10.5.3.3 normal scale

Not implemented at this time.

10.5.3.4 half scale

Not implemented at this time.

10.5.3.5 scale up

Not implemented at this time.

10.5.3.6 scale down

Not implemented at this time.

10.5.3.7 set scale

Not implemented at this time.

10.5.3.8 replace

Not implemented at this time.

10.5.3.9 color

Not implemented at this time.

10.5.4 Pull down Menu: Chores

Figure X shows the different options available under this menu.

10.5.4.1 name a chore

The name a chore menu choice allows you to name the choreography that you are currently within. An edit box will pop up in which you can type the new name.

10.5.4.2 Choreographies 00-15

Clicking on one of these will place you in that choreography and update the chore name button.

10.5.4.3 next set of chores

Clicking either mouse button, pulls down a window from which a new choreography can be selected.

10.5.5 Pull down Menu: Special

Figure X shows the different options available under this menu.

10.5.5.1 insert

Insert is just a pull down way of setting the cel addition button.The cel addition button allows the adding of four different special case cels into the choreography cel list table. These special case cels are, sound, counter, empty and flip.

10.5.5.2 palette

The palette editor and dk are currently undergoing revision to make them compatible.

10.5.5.3 Remap palette

The palette remap menu option determines how many colors have been used in the animation including the background color. One of the limits of the system is that actors and animation can only use 16 colors out of the 256 available. (recommended that this is out of the upper 48) to build a costume.

On activating the remap palette choice the screen will post a message saying "please wait, finding used colors in all cels."

The palette remap window contains two palettes. The upper palette (source) shows the number of colors in the current costume. The lower palette (destination) is where you designate the colors you wish to keep.

To identify the colors used, there is a black mark against the colors. To get rid off the colors that you don't want click on them in the lower palette.

The counter keeps track of the number of colors in the source palette and destination palette.

The revert button will take undo your changes and return you to the colors originally identified.

Once you are happy with the colors chosen, click on "ok" and then the source

palette will be remapped to the destination palette by changing the offending colors to the nearest match in the destination palette.

10.5.5.4 no flip left

One of the nice (yes there are a couple) items in byle is a time saving feature called flip left. By default the system automatically flips the image of whatever cels are in the choreography cel table listing in the left facing. This allows you to draw the right facing cels and then the system will take care of the left facing automatically. However please note you must still place those right facing cels in the left facing chore listing.

The menu choice No flip left just turns this off.

10.5.5.5 verbose output!

This menu option is used by Brad for debugging.

10.5.5.6 debug cel states

This menu option is used by Brad for debugging.

10.5.5.7 remove unused cels

Not implemented at this time.

10.5.5.8 onion skin

Not implemented at this time.

10.5.6 Pull down Menu: Backdrop

Figure X shows the different options available under this menu.

10.5.6.1 load lbm backdrop

This menu choice is available both here and under the file menu option. The load lbm backdrop choice loads a picture into the backdrop (backdrop is the right hand window) to see if the animation works in relation to the background.

Note: The right hand window is also used for importing .pict files from dpaint animator.

10.5.6.2 show backdrop

The show backdrop menu choice toggles the backdrop on and off. It does not

however immediately disappear from the screen. You must play animation, by clicking on the animation start/stop button to remove the backdrop.

10.5.6.3 save lbm backdrop

The save lbm backdrop menu choice saves the frames in a lbm format, so you can use them in dpaint or dk. This is being reworked.

10.5.6.4 Load .wak file

This is being reworked.

10.6 Toolbox explanation

The Toolbox, located on the right of the screen, is similar to the standard Toolbox used by paint programs. While Byle adds a couple of new tools, most remain constant to their dpaint equivalents.

10.6.1 Tool 1: Freehand Brush

The standard Byle brush is the dpaint Freehand brush tool default. This paints a single pixel at a time.

10.6.2 Tool 2: Straight Line Tool

Use the straight line tool to paint straight lines at any angle. Holding down the shift key will draw a straight vertical or horizontal line. (Dpaint page 39).

10.6.3 Tool 3: Rectangle Tool (Filled)

The Rectangle tool (filled) lets you paint rectangles and squares in a filled mode. Left click on the rectangle tool, move the cursor to the painting area and the pointer turns into cross hairs. Hold down the left mouse button. This is the first corner of the rectangle. With the mouse button still pressed down, drag the rectangle in any direction to the size you want. Release the mouse button. (DPaint page 41).

10.6.4 Tool 4: Rectangle Tool (Unfilled)

Painting an unfilled Rectangle is no different from painting a filled Rectangle. (Dpaint page 42)

The Rectangle tool (unfilled) lets you paint rectangles and squares in a unfilled mode. Left click on the rectangle tool, move the cursor to the painting area and the pointer turns into cross hairs. Hold down the left mouse button. This is the first corner of the rectangle. With the mouse button still pressed down, drag the

rectangle in any direction to the size you want. Release the mouse button. (DPaint page 41).

10.6.5 Tool 5: The Fill Tool

The fill tool is not implemented yet.

10.6.6 Tool 6: The Brush Pickup Tool

Click the brush pickup tool and move the cursor to the painting area. The pointer turns into a large crosshair. Position the crosshair at the upperleft corner of the area you wish to capture. Hold down the left mouse button and drag the cursor to the lower right hand corner of the image. Release the mouse button. Your new custom brush now has a copy of the image attached to it. Try stamping the brush on the screen by left clicking anywhere on the screen.

Unlike dpaint you may not pick up the image by pressing the right mouse button instead of the left mouse button in the first part of the above explanation.

10.6.7 Tools 7 (and 17): Grid Tools

Tools 7 and 17 work together to place the actor position. This is generally the foot position of the actor and is used for the calculation of elevation etc in Scumm.

Clicking on tool 17 will display the currently defined position for the grid in relation to the actor. This is shown in the animation (right) window. Once, and only once, tool 17 is active you can activate tool 7 to set the grid point to another position. This is done in the left window. Once in the left window the cursor will change to a large white cross hair. Left clicking within the left hand window will redefine the position of the grid and the change will be shown in the right window immediately.

Note that the tool 7 cross hair can only be seen when tool 17 has been toggled.

10.6.8 Tool 8: Magnify Tool

Hold down the left mouse button on the Magnify Tool. When the pop-up menu appears, the highlight the magnification power that you want. Hot key is M (Dpaint page 53).

10.6.9 Tool 9: Color Pickup Tool

Clicking colors in the palette isn't the only way to select foreground and background colors. You can also select colors directly from your artwork. This is especially useful if you are working with many shades of the same color.

Click the color pickup tool and move the cursor to the painting area. The pointer turns into a small target cursor. Place the cursor on the shade you want to pick up. Click the left mouse button to select this color as your new foreground color. Click the right mouse button to select the color as your new background color. Clicking automatically deactivates the color pickup tool.(Dpaint page 38)

10.6.10 Tool 10: Undo

This tool lets you undo your last painting action as long as you haven't clicked the mouse button in the meantime. Unlike dpaint, toggling the magnify tool is not the same as clicking (Dpaint page 36).

10.6.11 Tool 11: Stamp Tool

The Stamp tool works with the brush pickup tool. Use the brush pickup tool to capture the image that you require. Next click in the cel (usually blank) that you wish to stamp the image. Finally click on the stamp tool and the image will be imported into the cel you designated.

10.6.12 Tool 12: Blank

No tool has yet been assigned to this button.

10.6.13 Tool 13: Ellipse Tool (filled)

Move the mouse cursor to the painting area. The pointer turns into a large crosshair. Hold down the mouse button and drag the ellipse until it's the size you want. Release the mouse button.

10.6.14 Tool 14: Ellipse Tool (Unfilled)

To paint an unfilled ellipse is exactly the same as a filled ellipse.
Move the mouse cursor to the painting area. The pointer turns into a large crosshair. Hold down the mouse button and drag the ellipse until it's the size you want. Release the mouse button.

10.6.15 Tool 15: The Grid Tool I

This tool is usually used with a dpaint file setup with the standard costume box grid. Load the file in as a backdrop and click on tool 15.

By holding down the left mouse button, while in the right hand window, the artist defines a grab brush size. When the left mouse button is released, it will appear that nothing has happened. However the grab brush has been set.

To see this, left click again and the grab brush will reappear. Notice the effect that this has in the left hand window. Whatever is within the grab brush is portrayed at the center of the left hand window.

Move the grab brush around, while holding down the mouse button, to position the grab brush as you require. Releasing the mouse button will place the image that you captured with the grab brush into the first available cel within the current level (limb).

Doing this again will generate the image over the first image etc. An easy way to do animation from a dpaint input.

10.6.16 Tool 16: The Grid Tool II (costume box grid)

Tool 16 provides an easy way to input animation cels from a dpaint file, as long as the animation has been drawn using a costume box grid.

Load the .lbm file that contains the backdrop. Make sure that the boxes are the same color and that they are connected properly.

Click on any box line and the system will automatically place the contents of each box in the cel table. The system will take the top left drawing in the costume box grid and place it in the current "hot" cel within the cel table.

The system sets up the cel list from top left to top right if the images in the right hand window are each surrounded by a box and the left hand side of the box is touching the previous.. (These boxes do not have to be the same size).

The next drawing is then placed in the next cel after the "hot" cel. For example if the first cel is B/00 then the system will add the drawings to BB/01 then B/02 etc.

10.6.17 Tools 17 (and 7): Grid Tools

As noted above, tools 7 and 17 work together to place the actor position. This is generally the foot position of the actor and is used for the calculation of elevation etc in Scumm.

Clicking on tool 17 will display the currently defined position for the grid in relation to the actor. This is shown in the animation (right) window. Once, and only once, tool 17 is active you can activate tool 7 to set the grid point to another position. This is done in the left window. Once in the left window the cursor will change to a large white cross hair. Left clicking within the left hand window will redefine the position of the grid and the change will be shown in the right window immediately.

Note that the tool 7 cross hair can only be seen when tool 17 has been toggled.

10.6.18 Tool 18: Hand Tool

The hand tool has not been implemented yet.

10.6.19 Tool 19: Blank

No tool has yet been assigned to this button.

10.6.20 Tool 20: Clear Tool

When you want to erase the entire screen, use the clear tool. Selecting CLR erases everything on the current screen (or document if it's larger than the screen) and replaces it with the new background color. This is also useful for deleting a cel you no longer require.

10.6.21 Color Indicator

The two rectangles display the colors that you are currently using to paint. The inner rectangle shows the brush color, while the outer rectangle shows the current background color. Unlike dpaint's defaults of black and white, byle's defaults are black and blue.

10.6.22 Palette

Right clicking in the palette changes the background color. A window will pop up and ask the question "what to do? Rebuild cels to reflect backgroung color change? Yes or No"

Yes changes the color throughout the entire animation including within the costume.

No does change the background color but not within the frames. This is a good way to tell how much space the animation cels take up.

10.6 Question section.

Question 1:
In pixels, what is the maximum size for an animation box?
Answer 1:
The MAX size is 144 pixels wide by 132 high

Question 2:

When moving xrel and yrel what is the movement
Answer 2:
One (1) pixel. Holding down the shift key activates this by five.

Question 3:
How do you move around an image?
Answer 3:
The arrow keys will help the positioning except when in magnify mode. In that instance the arrow keys move the zoom position.

Question 4:
How many types of animation are there?
Answer 4:
There are two types of animation, ones that loop and one shots. Walking is a looping animation.

Question 5:
I'm running an animation but I don't see the cels highlighting in the cel table?
Answer 5:
Make sure that you are on the correct level. The cels will only show up if the level is highlighted.

Question 6:
When I load one costume it opens on the 02 (walk) choreography. When I open another costume, it opens on the 00 (null) choreography. Why?
Answer 6:
The system remembers where you were in a costume when you last saved the costume.

Question 7:
I lost part of my image, why?
Answer 7:
Whenever an image in a cel is moved out of the boundaries of the (left) window, that part of the image gets erased when it is moved back into the window.

10.7 A look at a specific costume

As Byle is rather complicated and I've probably confuse the heck out of you, let's look at the Indy costume (Indy.byl) in a little detail.

Load the Indy costume (Indy.byl) into newbyle. You will be in the init choreography facing right.

10.7.1 Walk choreography

Click on the B/06 cel and you will see a cel, in the left hand window, with Indy walking forward. By clicking on cels B/06 through B/11 (use the scroll bar as appropriate) you will see the 6 cels that are used for Indy's walking forward animation.

To get to the actual walk choreography from the init choreography, left click on the chore button once. Click on the forward button and you will see the cels 06-11 listed in the choreography cel list table.

Click the animation start/stop button and Indy's walking forward animation will run. You will also see the cels in the cel list table cycling through the cels numbers 06-11 by highlighting in yellow. Click the animation start/stop button to stop.

Now as you did with cels B/06-11, look at B/12 through B/19. Here you will see the 8 cels that make up the right direction walking.

Click on the right facing button and click the animation start/stop button. The right facing walk animation will start cycling just as the forward walking animation did above. Click the animation start/stop button to stop the animation.

A nice thing about byle is that it will save you time if the left directional animation is the mirror image of the right directional animation. The system will automatically "flip" the cels placed in the left direction choreography cel list.

Click on the left directional button and you will see that the same cels 12-19 are in the choreography cel list table. However when you click on the animation start/stop button, the animation window will show Indy walking toward the left.

Note you still need to place the 12-19 cels in the left facing choreography cel list table. As noted, the system will flip whatever you put in there. It is possible to turn off the feature if you do not need it or if it is a special animation that requires a separate and special left facing (see no flip left). Click the animation start/stop button to stop the animation.

Now check out cels B/00 through B/05 for the 6 cels that are used for the walk back direction. Click on the back button to see the list of cels in the choreography cel list table. Again click the animation start/stop button to see the walk back animation cycle.

Nearly all costumes should be set up this way for their walk animation.

Explanation

One of the more difficult aspects of byle to comprehend is the effect that a previous choreography can have on the current choreography. Very simply, if a previous choreography had five levels and the current one has 4 then that 5th level will run with the current animation unless you prevent it. This can be useful for example with walking and talking at the same time (no they can't chew gum too!). By placing the talk animation in a seperate level, that animation is added to the walk animation until deactivated.

We activate and deactivate these levels with the special cels invisible, visible and kill.

Invisible/Visible
The invisible cel makes the level invisible. It is important to note that this is not only in the current choreography, but on that level in every choreography within the costume.

The invisible cel does not stop the cel list on the level it is placed. Instead it "hides" it from your view.

The visible cel returns the level to a visible state in every choreography. When you use the visible cel to turn the level back on the list will become visible to the user again and it is possible for a choreography cel list to become active immediately.

Kill
The kill cel stops the animation in that level of the costume/choreography. Running a chore with cels in that level is the only way to deactivate the kill cel.

10.7.2 Init choreography

You may have noticed the in (which stands for invisible), within the C level of the choreography cel list table. This is because of the init choreography that was set for the costume.

Click on the chore name button and choose the (01) init choreography. Then click the front direction button. You will see that levels 1 and 3 in the choreography cel list table contain 00 cels. (i.e. A/00 and C/00). Click on cel C/00 in the cel list table and you will see that it is just a head, while A/00, as you saw earlier, is the torso and legs of Herman.

If you look at the different directions of the init choreography you will see that the left/right direction has cels A/07, B/00 and C/04 depicting the legs, torso and head. While the back direction has cels on only the A and C levels similar to the front direction. This may also explain a question you had having to do with the cels on level A that were not used in the walking animation (i.e. cels A/07 and

A/14.)

The init animation is run every time the costume (actor) enters a room and each time you change the choreographies. This can occasionally be inconvenient, hence the use of "invisible" and "visible" cels. (system uses visible as the default and therefore the "visible" is only needed to clear a previous, or potentially previous "invisible" cel.) The system still thinks an "invisible" cel is running, however the cel is unseen.

If the init has a level that is not subsequently over written by the next choreography (i.e. walk etc) then a level of animation will remain active with the next choreography unless a cel such as "invisible" is inserted. Similarly a visible cel will allow the init level to show through, this is the case with the stand choreography.

10.7.3 Stand choreography

To get to the stand choreography from the init choreography, left click on the chore button twice. If you are not in the front facing direction, click that button.

As you can see from the choreography cel list table, this choreography is a little different. Cel A/00 is in the A level of the choreography cel list table. As you saw earlier this is only the torso and legs of Herman. Level B has the "kill" cel inserted (which we will get to later) and level C has the "visible" cel allowing us to see the init choreography for the C level facing front.

Just to check that, go back to the init choreography (right click twice on the chore button) and you will see the init choreography facing front has level C cel 00 in the choreography cel list table. Click on level C/00 in the cel list table and you will see the head that was on top of cel A/00 back in the stand choreography.

Return to the stand choreography. Click on the right directional button and you will see level A in the choreography cel list table contains A/07 (a leg cel) and level B contains B/00 (a torso cel). Level C has a "visible" cel because similar to above the init chore (in this case facing right) is used.

As before the left directional button is just the mirror reflection of the right. The standing back choreography is similar in concept to the standing front choreography.

10.7.4 Talk choreography

Move to the talk choreography by left clicking once on the chore button. Click on the front direction button.

There is no cel in the A level of the choreography cel list table. The system uses the default and makes the init choreography, for this level and facing, visible. The B level of the choreography cel list table contains a "kill" cel.

Kill cels stop all animation including inits on that level. This is a good way of making sure that an init is not in the way.

Level C of the choreography cel list table contains the front talk animation. Cels C/00-C/03 are arranged in a 16 cel looping string to generate a realistic talk pattern. The talking back chore is similarly set up. Click on the right facing direction button and then click on the animation start/stop button. You should see Herman's arms move at the start of the animation but then stay still as Herman keeps on talking.

This effect is created on the B level of the choreography cel list table in this costume. A non looping cel { ( ) } is placed at the end of the 3 cels (B/00, B/03 and B/04) in the B level of the choreography cel list table.

Meanwhile the mouth talking animation on level C continues in a loop.

10.7.5 Stop Talking choreography

Move to the stop talking choreography. You would think that the init would accomplish this but most stop talking choreographies require a little more work.

The front and back facings are easy enough. Level A in the choreography cel list table is left open allowing the init to take place and level B contains a "kill" cel. Meanwhile level C contains the appropriate stop talking cel.

However it is the right facing that is of interest here. The B level of the choreography cel list table in this direction (and also left) returns the arms of Herman to their original position before he started talking.

Again as with the action in talking, this is a one time animation.

Basic Choreography conclusion.
These five choreographies, init, walk, stand, talk and stop talking are standard for all costumes. Most costumes have special case choreographies. Herman is no exception.

10.7.6 One hand point choreography.

One hand point is a good example of only using two directions in a costume. If you click on the front facing (or back) you will see that there are no cels in the

choreography cel list table.

The right direction is the area of interest. The A level of the choreography cel list table contains cels A/07 and A/21. This simulates Herman's legs swaying while the B level contains cels B/01-04 which do the specific one hand point maneuver of Herman's body and arms.

Click on the animation start/stop button and watch the head of Herman. Even though the C level of the choreography cel list table contains just a "visible" cel signalling that the init chore facing should be used, you will see the head move as the body sways. This is because of relative offset. Click on the animation start/stop button to stop the animation.

Take a look at cels B/01 and B/02 on the cel list table. Notice that the xrel button is set to -1 and -2 respectively on these. Xrel effects all levels below it in the choreography cel list table not the cel list table. The fact that B/01 is xrel -1 means absolutely nothing to cel C/01 which is below it in the cel list table.

Look at the choreography cel list table on the B level. The 4th cel in the cel list is 01. Xrel effects whatever is below that cel when that cel is run in the animation. In this case the "visible" cel in level C means that the init choreography is being used on level C. Therefore the xrel -1 effects that cel by -1.

In the animation you see this causes the head of Herman to move 1 pixel to the left (-1). When the B/02 cel is active, it causes the cel below it (still the init chore cel) to move 2 pixels to the left (xrel = -2).

Hence the illusion of the head moving without the need for further cel drawing.

Special case conclusion.
The choreographies Point (07), Shrug (08) and Talk to Camera (09), continue the concept of relative offset. The shrug choreography is particularly worth looking at as it combines both positive and negative x offset for a nice effect.

Appendix A Definitions

2.1.1 Frame

One update cycle. During a frame the following occurs in this order: execute all scripts in the order they appear in the script slot table;, move all actors one step, redraw the screen and objects, redraw all actors with next animation frame.

2.1.2 Heap

A large contiguous block of memory where the various rooms, sounds, objects, costumes, and scripts reside while they are being used. The heap manager discards items when they are no longer being used.

2.1.3 Use direction

The facing direction the actor automatically stands when he arrives at an object. The use direction is assigned to an object within the object editor, flem.

2.1.4 Use position

The X,Y coordinates attached to an object to which the actor will go when he is told to walk to that object. The use position is assigned to an object within the object editor, flem.

sputm script presentation utility tm

windex	scumm debugger
byle	animation tool
flem	room building utility
mmucas	maniac mansion universal compresion utility
spit	font and character editor

Appendix B Structure

3.0.1 Basic Structure

The structure of a typical room file. It includes a number of scripts followed by a room definition:

```
[include "filename"]
[script script-name {
        [statements]
}]
define variable-name=value
room "room-name" room-name {
[define first-script-in-room = max-scripts]
define script
[script script-name {
        [statements]
}]
        sounds {
```

```
        "SFX/sound-name" sound-name
    }
    costumes {
        "costumes\actor-name" actor-costume-name
    }
    enter {
        [statements]
    }
    exit {
        [statements]
    }
    [order {
        [object-names]
    }]

    object real-object-name {
        name is "[visual-object-name]"
        [dependent-on object-name being object-state]
        [class is class-state [class-state]]


        [verb verb-nameldefault{
            [statements]
        }]
    }
}
```

Structure Room Definitions (3.1)

3.1.1 include

Syntax: [include "filename"]

These are including other files, the header, system, boot, and main-scripts. The four includes are the scripts that control the user interface. The entire user interface, the verbs (see xyz), clicking on, the sentence construction is all written in Scumm. That's why Loom has a radically different user interface than Indiana Jones. We just wrote a different interface for it. These are all at the system level and we will get into that later.

3.1.2 script

Syntax: [script script-name {
[statements]
}]

A script is made up of a series of statements. Scripts are run as multitasking elements—there are slots available for up to 20 scripts to be running at any one time (though this could be increased if necessary). Scripts are stored with room files, along with costumes, background graphics, and sounds. This is to keep various items together on the disk to reduce seek time overhead.

When you start up a new script from within a script, the new one is executed immediately, like a subroutine. When the second script hits its first break, it is placed in the slot table after the calling script. Control is then returned to the calling script. From then on, the scripts are executed, one after another, in the order they appear in the slot table. It is guaranteed that no script will be run twice in any given frame.

When a script is going to be around in memory for a long time, it's best to keep it relatively small. For example, let's say you had a script which needed to execute some code, then wait around for 15 minutes, and then execute some more code. Rather than putting this all into one script, it might be a better idea to split it into three scripts:

```
script nuclear-meltdown {
        cut-scene {
                current-room ready-room
                [a bunch of code]
        }
        chain-script wait-for-fifteen-minutes
}

script wait-for-fifteen-minutes {
        sleep-for 15 minutes
        chain-script big-explosion
}

script big-explosion {
        [a huge amount of code]
}
```

3.1.3 room

Syntax: room "room-name" room-name {

The word in quotes is its file name. "Rooms" are not always traditional rooms such as the office. The Zeppelin maze, the College Exterior (both in Indiana Jones), and the Lucasfilm logo are examples of other forms of "rooms".A room is actually just a bunch of object definitions. Stored along with these definitions are a number of scripts, sounds, costumes, and the graphic room data. The string in quotes is the source file name. The next symbol is the room's assigned number.

```
room "kitchen" kitchen {
        [room stuff]
}
```

3.1.4 sounds

Syntax:
```
sounds {
        "SFX/sound-name" sound-name
}
```

This is another thing that trips people up. The doorclos sound effect is enclosed in the office but you can use the sound effect anywhere. If the actor is in another room, for example, the police station, you can still use the close door sound effect. If the sound is not in memory, the Scumm system will retrieve it from disk no matter what room is showing on the screen.These are the sounds that are attached to the room. They are not automatically loaded with the room, they're just stored with it. The string in quotes is the source file name. Then comes the numeric value assigned to the sound. Musical pieces are also treated as sound effects.

```
sounds {
        "Sfx/GarageDoorOpen" garage-door-opening
        "Sfx/EdselIdle" edsel-idle
        "Sfx/MeteorSfx3" meteor-sound
}
```

3.1.5 costumes

Syntax:
```
costumes {
        "costumes\actor-name" actor-costume-name
        }
```

This lists the costumes which are attached to the room. It's made up of the actor's animation frame data and their animation sequences. The string in quotes is the source file name. Then comes the numeric value assigned to the costume. Let's explain the difference between actors and costumes.

An actor is a person who walks around the room. The actor can hold things in an inventory. A costume is what the actor looks like. An actor can look like anything by changing the costume to sam, max, indy, a nazi, anything you want just by changing the actor's costume. Think of it as the actor is a skeleton and a costume is the skin. An actor can have no costume in which case the actor is invisible. However the actor still walks around and can have an inventory etc.

```
costumes {
        "Costumes/zak" normal-zak
        "Costumes/small-zak" small-zak
        "Costumes/newscaster" newscaster-skin
}
```

3.1.6 enter

Syntax:

```
enter {
        [statements]
}
```

The statements in the enter construct are executed as soon as the room becomes the current-room, but before it irises in. Things serviced here could be background sounds that might have been left on (a shower, waterfall), repositioning objects with draw-object, setting object names with new-name, starting scripts for actors in the room, and turning on the lights.

```
enter {
        start-script check-lights
        if (state-of shower-water == ON) {
                start-sound shower-sound
        }
        if (state-of bathroom-sink == R-ON) {
                start-sound running-water
        }
}
```

3.1.7 exit

Syntax: exit {
[statements]
}

The statements in the exit construct are the last things to be executed before we leave the current-room. Things that might be serviced here are turning off background sounds, stopping scripts, and setting flags.

```
exit {
        stop-sound shower-sound
        stop-sound running-water
        stop-script dead-ted-dancing
}
```

3.1.8 order

Syntax: [order {
[object-name]
[object-names]
}]

This contains a list of all the objects in the room in back to front order. This construct is optional—it's not needed if the order of the objects isn't important, i.e., no object covers another object.

```
order {
        attic1-ednas-room-hatch envelope
        wall-safe-combination wall-safe
        wall-safe-painting attic1-light quarter
}
```

Structure Object Definitions (3.2)

3.2.1 object

Syntax: object real-object-name {

Each object in a room needs its own definition.

3.2.2 name is

Syntax: name is "string"
name is "[visual-object-name]"

Used at the beginning of every object definition to provide the name printed on the sentence line when the object is touched. If the object is never to be touched, the name string can be null. If the name is to be changed during the game, be sure to pad the name with enough @'s so its definition contains the maximum number characters it will ever need.

name is "plant"
name is "flashlight@@@@"

(Then we can use new-name-of flashlight is "lit flashlight")

3.2.3 dependent-on

Syntax: [dependent-on object-name being object-state]

Used when the appearance of an object is dependent on the state of another. For example, a can of Pepsi inside a refrigerator. If the door is closed, the Pepsi would be invisible:

dependent-on refrigerator-door being OPEN

3.2.4 class is

Syntax: [class is class-state [class-state]]

This is used to allow the classification of objects. Each object can be a member of any (or none) of the twenty-four possible classes, with class 0 being the default (not a member of any class). Examples of classes could be
The class of an object is tested with the special function, .i.class-of:see also;().

3.2.5 verb

Syntax: [verb verb-name|default{
[statements]
}]

Each object will most likely have several verb definitions. Multiple verbs can be used within a single verb definition. The verbs used in Indy 3 were:

open turn-off push walk-to
close pull pick-up

give use default
turn-on look talk-to

Default is used for any verb not explicitly defined. If default isn't defined, then a special script is run which has its own set of defaults for each verb.All objects are made up of verbs. The verbs can be literally anything the Scumm programmer wants. A maximum of 255 verbs can be associated with any given object. The verbs open, close, push, pull, etc. are what we decided to call the verbs for Scumm-U and Indiana Jones.

```
verb open pull unlock {
        start-script open-door
}

verb read {
        say-line "It's too tiny to read!"
}
```

Appendix F System Variables - Slot Order

variable complex-temp	@ 0
variable selected-actor	@ 1
variable camera-x	@ 2
variable message-going	@ 3
variable selected-room	@ 4
variable override-hit	@ 5
variable machine-speed	@ 6
variable me	@ 7
variable number-of-actors	@ 8
variable current-lights	@ 9
variable current-disk-side	@ 10
variable jiffy1	@ 11
variable jiffy2	@ 12
variable jiffy3	@ 13
variable music-flag	@ 14
variable actor-range[2]	@ 15
variable actor-range-min	@ 15 (actor-range)
variable actor-range-max	@ 16 (actor-range+1)
variable camera-min	@ 17
variable camera-max	@ 18
variable min-jiffies	@ 19
variable cursor-x	@ 20

variable cursor-y @ 21
variable real-selected-room @ 22
variable last-sound @ 23
variable override-key @ 24
variable actor-talking @ 25
variable snap-scroll @ 26
variable camera-script @ 27
variable enter-room1-script @ 28
variable enter-room2-script @ 29
variable exit-room1-script @ 30
variable exit-room2-script @ 31
variable build-sentence-script @ 32
variable sentence-script @ 33
variable update-inven-script @ 34
variable cut-scene1-script @ 35
variable cut-scene2-script @ 36
variable text-speed @ 37
variable entered-door @ 38
variable sputm-debug @ 39
variable K-of-heap @ 40
variable sputm-version @ 41
variable restart-key @ 42
variable pause-key @ 43
variable screen-x @ 44
variable screen-y @ 45
variable frame-jiffies @ 46
variable total-jiffies @ 47
variable sound-mode @ 48
variable graphics-mode @ 49
variable save-load-key @ 50
variable hard-disk @ 51
variable cursor-state @ 52
variable userput-state @ 53
variable text-offset @ 54

INDEX (from imported doc)