# MicroCosm™ Home System Architecture

*the design of the home-based portion of the MicroCosm system*
*by*
*Chip Morningstar*

Lucasfilm Ltd. Games Division
November 10, 1985

## Introduction

This document attempts to describe the preliminary **MicroCosm** home system architecture. The purpose of this document is to try to get all the design straight before getting bogged down in implementation.

The names of system components refered to in this document match those used in the document **MicroCosm Home System Architecture Block Diagram**. Please refer to this diagram to avoid hopeless confusion.

## Overview

The home system consists of five general parts: the *Communications Channel Controller*, the *Local Object Database*, the *Object Memory Handler*, the *Display Generator* and the *Action Interpreter*. Each of these will be explained in turn.

## Communications Channel Controller

The Communications Channel Controller is responsible for communications with the host. Communications are carried on via *messages* from one computer to another. Messages from the home system to the host consist exclusively of requests from the various home system components for action or information on their behalf. Messages from the host to the home system can be of two sorts: responses to requests sent out previously and notifications of external events that the home system should be aware of. The Communications Channel Controller consists of five components: a *Request Handler*, a *Send* unit, a *Receive* unit, a *Response Classifier* and an *Asynchronous Event Handler*.

### REQUEST HANDLER

The Request Handler is the interface through which various components of the home system issue requests to the host. The Request Handler consists of two routines and some data structures. The two routines are the ‘‘request’’ routine and the ‘‘response’’ routine. Home system components that wish to make a request call the ‘‘request’’ routine with parameters that reflect the request desired. The ‘‘request’’ routine takes these parameters and constructs a request message, which it then passes to the *Send* unit to be sent off to the host. Roughly speaking, requests can be divided into two groups: those that expect a response and those that don’t. Requests that do not require a response may simply be sent off and then forgotten about. Those that do require a response, however, require that the requester provide the address of a *response handler*, which is simply some code that can handle the response appropriately on the requester’s behalf. This address is logged in the Request Handler’s internal data structures. When a response arrives from the host, the *Request Classifier* calls the ‘‘response’’ routine to look this address up and branch to it.

### SEND

The Send unit is responsible for actually sending messages out over the packet switching network to the host. It controls the modem for transmission of messages. The Request Handler gives it a message to send. It attaches the appropriate header and trailer bytes and sends the message down the wire.

### RECEIVE

The Receive unit is responsible for actually receiving messages off the packet switching network from the host. It controls the modem for receipt of data. A message comes in off the wire. The receive unit strips off any header and trailer bytes (verifying that all is well as it does this), and passes the unprocessed remainder of the message to the *Response Classifier* to figure out what to do with it.

### RESPONSE CLASSIFIER

The Response Classifier handles incoming messages from the hosts. It divides them into those that are responses to requests and those that are asynchronous messages from the host. Messages that are responses to requests are handled by calling the ''response'' routine of the Request Handler. The others are handled by calling the *Asynchronous Event Handler* (n.b.: this means that the message protocol must be designed so that message headers tell whether the messages they are attached to are response messages or asynchronous notifications).

### ASYNCHRONOUS EVENT HANDLER

The Asynchronous Event Handler is responsible for processing messages from the host that are not responses to requests that were previously issued. Such messages are called ''asynchronous notifications'' because they are notifications from the host that some external event has taken place that the home system should be aware of. Such events are, by design, always relevant to particular objects in the world. The Asynchronous Event Handler uses information contained in the message to determine a specific action to execute, as if it were a directive from the player that was input through the conventional player interface.

## Local Object Database

The **MicroCosm** world model is object based. Each home system keeps an internal representation of the world around it in terms of the objects local to it. The Local Object Database is what contains all these objects. There is one entry in this database for each object. Each such entry contains (conceptually speaking) three parts: the *Object Imagery*, the *Object Behavior Code* and the *Object state*. Actually, each object that exists is a member of a *class* of objects that are all the same. All the objects of a given class can share their Object Imagery and Object Behavior Code in common. Nevertheless, it is sometimes useful to maintain the fiction that each object is a unique and distinct entity, which we shall do.

### OBJECT IMAGERY

An object's Object Imagery is a collection of one or more images that are used to display that object on the screen. Images are represented in the standard **MicroCosm** image format (see the document **MicroCosm Graphics** for more details). Most objects have a single image that represents them. Some have two or three representations corresponding to various states that the object can be in (for example a box object could have two images, one for when the box is open and one for when it is closed). A few special objects have many images. The most notable example of this is the object that represents the avatar itself, which has a wide range of images that are used to animate it.

### OBJECT BEHAVIOR CODE

The Object Behavior Code defines the object's behavior (I hope you are not surprised by this). It is a collection of executable code with seven entry points. There is one entry point for each of the six essential verbs *get*, *put*, *stop*, *go*, *do* and *talk*. These entry points are used by the Action Interpreter to invoke the essential behavior of the object. The seventh entry point is used by the Asynchronous Event Handler to handle asynchronous notification messages from the network host. There *may* also be additional entry points for internal use by other objects' Behavior Code.

Before calling one of the seven main entry points, the calling routine sets a register to point to the object's Object State information, so that the Object Behavior Code can distinguish which object of a particular class it is and manipulate its state information easily.

### OBJECT STATE

An object's Object State is just a section of memory that defines the particular state of the particular object. The size of this state information may vary depending on what type of object it is. The interpretation of the various bytes is entirely dependent on the definition of the Object Behavior Code.

## Object Memory Handler

The Object Memory handler is responsible for managing the storage of objects. It is a sort of three-tier virtual memory system. The first tier is in core, the second is on the home system's disk, and the third is in the host's database. As with conventional virtual memory systems, the tradeoff between the tiers is between capacity and access speed. Ideally, all objects would like to be kept in core. However, there will not be room for this because there are too many of them. We thus go to disk as a secondary storage medium. However, not all possible objects can fit on the disk either and so we use the host itself as a tertiary medium. The basic idea is that all the objects that a player is dealing with (or could conceivably deal with readily) at any given time are kept in core. All the objects that are part of a home system's particular world view are kept on disk. All the objects that exist in the universe are kept in the host.

The Object Memory Handler is different from conventional virtual memory systems in a couple of important ways. The first difference is that the first level (between core and disk) is not fault-driven. That is, it is not transparent to the higher level routines using the object database. Rather, the routines that work with objects explicitly call the Object Memory Handler to set things up for them. The second level (between disk and host) *is* fault-driven, in that a reference to an object that is not on disk automatically triggers the retrieval of the object from the host and the discarding of an old object from disk to make room.

The second difference between the Object Memory Handler and conventional virtual memory is that object state is not unique to a particular machine. In other words, multiple home systems can have a representation of the same object at the same time. Any of these systems is free to update its state information for this object as it wishes. This raises the question of which system's representation corresponds to the ''real'' object. The answer is that none of them do. Only the host knows what is ''real''. It is presumed that the home system will only alter the local state of an object as a result of requests to the host or asynchronous notifications from the host. This means that the second level of swapping (between disk and host) is entirely outbound — objects never travel from the home system back to the host (for security reasons the host cannot generally trust a home system to tell the truth about an object's state and therefore does not attempt it).

The Object Memory Handler module consists of three parts: the disk storage itself, the *Disk I/O* routines, and the *Cache Manager*.

### DISK I/O

There is not much to say about the Disk I/O routines. They ought to be fairly conventional. In the Commodore 64 they should incorporate somebody's fastload routines to get around the abysmal Commodore disk performance.

### CACHE MANAGER

The Cache Manager is the heart of the Object Memory Manager. Since the memory model is three-tiered, it has two jobs. The first is to move objects between core and disk. It does this as requested by various routines in the Action Interpreter. To do this it calls on the Disk I/O routines as needed. The second job is to retrieve objects from the host. It does this whenever it needs to get an object off disk but the object is not there. To do this it sends requests to the host via the Request Handler in the same manner that object behavior routines do.

## Display Generator

The Display Generator is responsible for graphics and sound. The Display Generator consists of a *Display Controller* module with associated *Display State* information, that oversees the operation of five subsidiary modules: the *Background Generator*, the *Object Cel Renderer*, the *Sound Effects Generator* and the *Cursor Display*.

### DISPLAY CONTROLLER

The Display Controller is the master of the Display Generator. It controls all the graphics except for the cursor. It keeps an internal representation of the state of the display in the *Display State* data structures. From these data structures it generates a new display each frame-time. Calls to various Display Controller routines update these data structures and thus change the state of the display the next time it is refreshed.

The **MicroCosm** display is rendered using a fixed point of view for any given location in the fantasy world. All action takes place inside fixed regions of the world. The display shows the particular region of action in which the player's avatar is located. When the avatar leaves the screen it also exits a region of action, entering a new region as it does this. The point of view then cuts to this new region. No ''camera motion'' takes place. All changes of view are executed by cuts. Each region of the world can be thought of as a stage, consisting of a floor, a backdrop, props and actors. The actors are the avatars involved in whatever actions are taking place at the moment. The props are the objects that the avatars may encounter and manipulate. The backdrop is made out of additional objects that the avatars interact with only indirectly. The stage floor itself is blank. For any given point of view, the backdrop is fixed and therefore only needs to be rendered once, when the point of view shifts. The actors and props, however, may move around. This can change both their positions on the screen and their depth priority with respect to each other.

Each type of display element has a set of routines that render it on the screen as needed. These rendering routines are called by the Display Controller as indicated by the Display State data structures.

### DISPLAY STATE

The Display State is a set of data structures that are used to drive the operation of the Display Controller. The basic operation of the Display Controller is to read through these data structures and render things on the screen according to what it finds in them. Object Behavior Code affects the display by updating the Display State data structures. Standard routines are provided to perform such updates easily.

### BACKGROUND GENERATOR

The Background Generator is called whenever the point of view changes. It generates the background image that lies behind each frame. This image is generated from a series of background object images that are resident in the local object database. These images may be large, thus requiring the rendering of the background to take more than a single standard frame-time. However, the background need only be generated once per point of view shift, so this slowdown is acceptable. The rendered background is then saved away in a buffer for reuse in successive frames.

### OBJECT CEL RENDERER

The Object Cel Renderer is called each frame to render the various avatar figures and foreground objects that appear on the screen. The Object Cel Renderer is an enhanced version of the Lucasfilm cel animation driver that was used in *The Eidolon* to render animated critters. It is enhanced in two significant ways. The first is that it incorporates algorithmic scaling so that the size of the image rendered can be adjusted according to the distance between the viewpoint and the object being displayed. The second is that the various cels are parameterized in different ways so that such things as their individual aspect ratios, proportions and coloration can be altered at will.

For character animation (i.e., the avatars) we work from a single set of basic images that animate a generic avatar figure through a wide variety of possible actions. This generic image is then adapted parametrically (during the rendering process itself) to render each avatar as a distinct individual. The choice of cels (i.e., how the figures animate) is determined by entries in the Display State data structures that are established by the object behavior code of the avatars.

Foreground objects are rendered similarly, but by a much simpler control process since they do not have to be customized on the fly. Which object images should be displayed and where they are to be place on the screen is a indicated by the Display State data structures. As with avatars, this is determined by the object behavior code for the objects in question.

#### SOUND GENERATOR

The Sound Generator emits various sounds corresponding to the behavior of objects and avatars. It too is driven from the display state data structures as set up by the object behavior code.

#### CURSOR DISPLAY

The Cursor Display is responsible for the cursor. The cursor is used in the player interface and so is controlled by the Action Interpreter routines which implement that part of the system. The cursor simply moves over the screen according to the player interface's calls. The Cursor Display routines must keep the cursor on target while making sure that the cursor update and display functions do not interfere with the regular graphics.

## Action Interpreter

The Action Interpreter is the heart of the home system. It encompasses the player interface and the object model interpreter. It consists of eight modules: the*Attention Selector*, the *Action Selector*, the *Cursor Controller*, the *Text Input Handler*, the *Action Generator*, the *Synchronous Action Selector*, the *Action Lookup* module, and the *Asynchronous Action Selector*.

#### ATTENTION SELECTOR

The Attention Selector is the primary input handler for the player interface. It operates in tandem with the *Action Selector*. Inputs to the Attention Selector come from the joystick and from the joystick button. Joystick inputs cause the Attention Selector to direct the *Cursor Controller* to move the cursor around on the screen. When the button is pressed, the Attention Selector attempts to determine which object the cursor is pointing at. This object becomes the *focus of attention*. Both the focus of attention and the raw screen coordinates indicated are noted and control is passed to the Action Selector.

The only hard part is determining which object the cursor is pointing at. The Attention Selector keeps a data structure that is based on the Display Generator's Display State data structure (actually, in the implementation these two data structures may become one). This data structure lets the Attention Selector do a lookup using the screen location as a key to retrieve the object under the cursor.

#### ACTION SELECTOR

The Action Selector forms the other half of the player interface. It operates in tandem with the Attention Selector. It too gets its input from the joystick and the joystick button. When control is handed to it by the Attention Selector, a focus of attention has already been determined. The Action Selector simply waits for the joystick button to be released. The state of the joystick when the button is released then determines a choice of action, as explained in the document **MicroCosm Player Interface**. Once the action is determined, control is passed to the *Action Generator* so that an action may be executed.

#### CURSOR CONTROLLER

The Cursor Controller directs the motion of the cursor over the screen. It is called by the Attention Selector with input from the joystick and in turn calls the Cursor Display to actually render the cursor. We may choose to highlight the specific object that the cursor is indicating, in which case the cursor shape and size may change as it roams over the screen. If we decide to follow this strategy, the Cursor Controller will also be responsible for doing this highlighting.

#### TEXT INPUT HANDLER

The Text Input Handler is responsible for dealing with the other primary channel of player input, the keyboard. As characters are typed it buffers them up. It handles the echoing of text in the text window of

the display. It also takes care of local editing functions (backspace and so on). When a *message termination* character (a carriage return or one of the gestural function keys) is typed, the string that has been input is passed to the *Action Generator* as a *talk* action.

**ACTION GENERATOR**

The Action Generator integrates input from the Text Input Handler and the Action/Attention Selectors. It takes the focus of attention, the selected action, and other state information (the input text or the contents of the avatar's hands, for example) and decides three things: an object to be the *actor* (the thing executing the action), the *action* (the thing to be done), and an object to be the *target* (the thing that the action is directed towards, if any). The target may be optional, depending on the action. These items are then passed on to the *Synchronous Action Selector* for use.

**SYNCHRONOUS ACTION SELECTOR**

The Synchronous Action Selector is responsible for the execution of actions that are the result of player input. It also handles actions that are generated internally by the Object Behavior Code — the routines defining the behavior of an object can themselves spawn a cascade of actions, essentially pretending to be the player and invoking actions recursively.

The Synchronous Action Selector calls the Action Lookup routines to retrieve the entry point to the appropriate Object Behavior routine from the Local Object Database. It uses the *actor* object and the *action* as the index keys for this lookup. The routine thus retrieved is then called with the *actor* object and the *target* object as parameters. The called routine then does whatever the action calls for.

**ACTION LOOKUP**

The Action Lookup routine finds the entry points of Object Behavior routines in the Local Object Database. It looks these up from object/action pairs. The database is organized to simplify this lookup operation. If necessary, the Action Lookup routine calls on the Object Memory Manager to retrieve the required object from disk so that it's Object Behavior Code will be memory resident and thus executable.

**ASYNCHRONOUS ACTION SELECTOR**

The Asynchronous Action Selector is responsible for the execution of actions that are the result of asynchronous notification messages from the host. It is called by the Asynchronous Event Handler. Asynchronous notification messages contain the three essential action selection components: the *actor*, the *action* and the *target*. The Asynchronous Action Selector must extract these from the message and then proceed as the Synchronous Action Selector would with similar input from the player.