MicroCosm™ Graphics

how we generate the displays and animation
by
Chip Morningstar

Lucasfilm Ltd. Games Division
November 10, 1985

Introduction

This document attempts to describe the preliminary design of the graphics internals for the **MicroCosm™** home system. This document describes the design as generically as is possible, since portability is a major concern. However, it also goes into some nitty-gritty Commodore 64 specific details because that machine will be the first target for this software.

General Concerns

We want to give the **MicroCosm** player a colorful, animated display that changes in real-time at a tolerable frame-rate (say, no worse than 6 frames per second). This display should show the player a plausible representation of the environment in which his avatar is acting, including both background and foreground details. It should also display his own avatar and the avatars of other players who are in that environment with him. Objects in the scene should move and change according to the actions of the various players.

To make the graphics rendering task tractable to a small processor, we wish to constrain the scene being rendered in as many ways as we can reasonably get away with. The more constraints we have on what must be displayed, the more we can use pre-computed imagery and other graphic tricks to speed things up.

Regions

The first set of display constraints arises out of the topology we are going to use for the universe itself. This topology is detailed in the companion document**Micr oCosm Coordinate Systems and Topology**. However, we will summarize briefly here.

The basic geometric unit of the universe is something called a *region*. A region is an independent, bounded section of the cartesian plane. Conceptually, the bounds of a region could be any sort of geometric figure. For simplicity, however, we decree that all regions will be bounded by rectangles. This allows us to describe any region with just two numbers: its `x` and `y` dimensions. Locations within a region can be specified using ordinary cartesian coordinates. Regions connect to other regions edge-to-edge. Depending on the topology of these connections, the resulting world geometry can be perfectly ordinary or quite bizarre. The topological specifics are discussed in the aforementioned topology document, however.

Any given object in the world can be located in one and only one region at a time. This includes avatars. Regions are always small enough, relative to the size of an avatar, that an avatar can ‘‘see’’ everything that is in his region with him.

The point of view is restricted. We will not build a general purpose rendering engine capable of displaying every possible situation from every possible point of view. Instead, each region will have a pre-assigned viewpoint associated with it. The home system displays just one region at a given time: the region in which the player’s avatar is currently located. The display generator renders this region as it looks from its pre-assigned viewpoint.

Objects

A region all by itself is blank. To render it all we need do is fill the screen with one color above the horizon line and another color below it, corresponding to the sky and the ground, respectively. A region acquires visual detail through the objects that are found inside it. Thus, rendering a region basically comes down to rendering a collection of objects in three dimensions.

We get another display constraint by classifying the objects to be rendered according to whether they are *background* or *foreground* objects. Background objects, as the term suggests, are located in the background and at the sides: at the periphery of the region where action never occurs. Foreground objects, on the other hand, occupy center stage and take an active part in the on-going events. Graphically, they are distinguished by whether anything can ever move behind them.

We can define the sets of foreground and background objects inductively, as follows. First, all avatars are, by definition, foreground objects. Furthermore, any object that could ever be displayed in front of a foreground object is itself a foreground object. Any object that is not a foreground object is a background object. However, this definition is somewhat impractical for a machine to use. It is instead a guideline for us human implementors. In practice, objects will be statically classified as background or foreground objects according to functional roles chosen by their creators.

Cel Animation

Objects are rendered using the *Lucasfilm Cel Animation Driver*. Each object possesses some number of *cels*. A typical object will have only one cel. Some objects that do not appear on the screen at all will have no cels. Avatars, being highly animated, will have six or more cels. Each cel consists of a state variable and a set of images. The value of the state variable determines what image from the set will be used to display the cel. To render an object, the Cel Animation Driver paints the appropriate image for each cel on the screen at the appropriate location. The various cels of an object have a predetermined spatial relationship to each other and to the screen location of the object itself, so that the Driver can tell where to paint them given a base location for the object. This base location is the projection of the object's three-dimensional location in the scene onto the two-dimensional display screen surface. This projection is computed by the Driver.

The Driver paints its objects onto a static background which it obtains from a buffer somewhere in memory. There will thus be two modes in which the Driver will be called. Once each time the point of view changes it will render the background objects on top of a blank background resulting in a detailed background image that we save away. This background image is in turn used as the background each frame as the Driver is called to render the animated foreground objects on top of it. We presume that the background may be fairly complex, thus this two-stage process allows us to boost our frame rate significantly while still retaining a single very generalized object-oriented rendering engine throughout the whole process.

The Driver also scales the size of the images it renders according to the distance from the viewpoint to the object being depicted, thus far away objects appear small and nearby objects appear larger. This is the way the world should be, of course. In addition, each cel can carry with it some additional information that is used to modify the rendering process. In particular, we can differentially scale the X and Y dimensions of the image and we can scale the various cels of an object independently from one another. This means that we can parametrically distort the object in various ways. This is one of the means by which we personalize avatars. We can adjust the aspect ratios of each body part, for example making the torso fatter or the legs longer. This way we can use a single set of avatar cels to render a wide variety of different avatars. We can also parameterize the colors — there's no need for objects of a given type to all be the same color. In other words, you can have a blue book and a red book and render them both from a single book image.

The display of any particular object is limited to the points of view provided for by the imagery that defines its appearance. Most objects will only have one point of view depicted in their imagery, and so will appear the same from all directions. This is the price we pay for saving both computation time and storage at once. Certain objects will be depicted from more than one direction. These will be ones for which the change in appearance from one view to another has some particular importance. The most notable example

of this will be avatars themselves, who will be viewable from any of eight directions. Thus, the avatar image set will be quite large. This is the reason we choose to customize avatars parametrically rather than by image substitution.

Closeups

Some complexity is introduced by the need to have occasional *closeups* of certain things in the world. This is so that the player can have the opportunity to choose among a number of small objects that would otherwise be packed into too small an area on the screen to make conventional indication with the cursor practical. For example, if the avatar draws three letters from his mailbox, the normal view of the region around the mailbox simply shows the avatar holding a letter-like object in its hands. However, a closeup on the hands shows the three letters as distinct items, so that the player can pick the one he wants to read.

The purpose of the closeup display is to smooth the player interaction, rather than to enhance the visual richness of the game. We simply do not have enough storage space to hold images of every object writ large. Instead, the closeup display is somewhat abstract, showing ordinary images of the objects in conventional states, arrayed geometrically. These images would act as icons. They would be shown against a stylized background indicating the context in which the objects are being viewed. Thus, to return to our example of the three letters, the player would not see a picture of a giant pair hands filling the screen and holding three letters. Instead he would see three letter images all in a row in the middle of the screen, with hands at the sides of the screen to indicate that these objects are in the avatar's hands. Situations in which closeups are possible will be dictated by the behavior of the certain objects. In other words, only those objects that know how to display their contents in closeup form will do so.

Screen Layout

The screen is divided into two parts: a graphics window and a text window (see the document **MicroCosm Player Interface** for information on how these affect player interaction). The graphics window fills the lower portion of the screen. It is as wide as the screen is and exactly half that high (i.e., it has a 2:1 aspect ratio). All the animation is rendered within this window. The upper part of the screen is where the text goes. The text window is used to display text typed by the player, messages from other players, and other messages from the system itself (the latter will be rare).

The text will be displayed with surrounding graphics or with a special character set that lets us show the text inside a cartoon style word balloon. Each time a new player speaks the balloons in the text area scroll upward. Successive speakers' balloons will have different background colors to distinguish them from one another. In addition, for the first few seconds a balloon is on the screen, we will draw a downward pointing "V" from the bottom of the balloon to the head of the avatar who is speaking. This will enhance the visual impression of the balloon as cartoon device. After a few seconds (a time computed on the basis of the length of the text in the balloon) the "V" disappears. On the Commodore 64 this "V" is easily implemented as a sprite.

Cursor

The cursor is controlled by the joystick. The player uses the cursor to designate his focus of attention for purposes of issuing commands to his avatar. The cursor is thus interpreted as pointing to a particular object on the screen at any given time. Sometimes this will be the blank background, which must function as a sort of pseudo-object for the designation process to work correctly. Not surprisingly, the cursor is probably most easily implemented on the Commodore 64 using a sprite.

The shape of the cursor is, as yet, undecided. Strong candidates include the arrow pointer (as found in the Macintosh and the Amiga), a crosshair, or a circle. The circle is an interesting idea: we could vary the diameter of the circle depending on the size of the object it is indicating, so that the circle always more or less encloses the object. Another possibility is to not really have a cursor but to simply highlight the selected object in some fashion. However this could be unsightly and also makes it hard to simply point at *locations* in the region, rather than at particular objects.

Sound Effects

Many of the objects in the world will be able to make sounds as part of their behavior. The home system will include a sound effects driver and a library of sound effects data. To make a sound, the behavior code for an object simply calls the sound effects driver with the number of the sound effect that it wants. Sounds will be implemented as a simple set of discrete noises of various sorts.