Reno™

a realm editor
by
Aric Wilmunder

Lucasfilm Ltd. Games Division
November 1, 1986

Introduction

Reno will be a stand alone software product for the Commodore 64 family that will allow independent developers to create both regions and realms for the *Habitat* universe. The product will be written around our already existing Region Editor which will give us a solid base for further development. Upgrading of the current system will only be necessary in six different areas. Listed in order of development, these are:

- Realm map editor
- Disk load/save routines
- Object editor
- Region editor polishing
- Text editor
- Final polishing

Realm Map Editor — 1 month

This portion of the tool will allow the designer to create a map of the realm under development. These maps will resemble the Macintosh maps used for current realms, and will have the added feature of enabling the designer to move a cursor over any particular region and have that region loaded from disk. The interface will allow a designer to create new regions, change region connectivity, name regions, move existing regions, and change region orientation.

Other features will allow a designer to adjust specific region parameters. These include region light level, horizon line, and style. All of these will have common default settings, but an interface for changing them will be needed.

Disk Routines — 2 weeks

This time will be necessary for implementing disk routines for loading and saving both the realm map as well as individual regions. These will be saved on the same disk and the user will need to swap to a *Habitat* imagery disk when loading requested regions or creating objects currently not in the heap.

Object Editor — 1 week

There is currently no utility (other than *Twiddle*) that allows a designer to change specific parameters of objects in the *Habitat* world. These parameter adjustments would include locking/unlocking doors, setting token denominations, and changing state values of all other objects. Unfortunately, most objects are different, and special case code will be needed for many objects.

Region Editor Polishing — 2 weeks

This will involve cutting the umbilical cord from the Sun workstation and moving all of the Sun-based commands to the Commodore 64. Initially this will include creation of objects from the Commodore

64 keyboard, and rejection of incorrect requests made by the user. It will also include additional modes for editing patterns in trapezoids, readouts for mode and object data, as well as simplification of the user interface.

Text Editor — 2 weeks

We wish to include a simple text editor designed for editing the books, magazines, and newspapers used in *Habitat*. It may be possible to modify the editor that we already have. It will also be necessary to load and save the materials onto the disk.

Final Polish — 1 week

This time will be used for final debugging as well as for completing documentation for the realm designer.

Command Interface for the Region Editor

Since the base program for Reno is the current Region Editor design, an explanation of both the current and proposed command set for the Region Editor will aid understanding of this proposal. Following this explanation will be a list of the current and proposed commands for the realm editor.

Realm/Region Editor Capabilities

As described earlier, the current Region Editor is attached to our Sun Workstations. The principal reason that the Commodore and the Sun are attached is for storage of region data. In the current version, the Sun also controls the creation of objects in a region. In the stand-alone version of Reno, object creation will be done at the Commodore 64 keyboard. This will simply involve entering an object class and style and pressing a key to create the object. For example, in order to create a box, I look onto my list of objects, see that the `box` object is class #13 and if I wish to specify a different style (e.g., crate vs. treasure chest) I can specify a style number as well. If a style is not entered, the default style is zero.

After an object has been created, next you will want to position it within the region. Part of this is accomplished by creating the object at the cursor location. This allows you to roughly position the object while it is being created. Following initial positioning, you can now use the keyboard to move an object around the screen. There are both fine and coarse control keys to speed up movement. Next the designer will want to decide what orientation to use for the object (i.e., whether to face left or right). After positioning is complete, patterning or coloring the object is possible. The designer has a choice of 15 different patterns (solid blue, black, and pink are among these) and the full range of 16 colors on the C64. Objects that may be carried by an Avatar are not capable of being colored.

Finally, the designer will need to decide if the object is to be in the foreground or the background of the region. Background objects include walls, doors, ground, some trees, pictures on walls, and other objects that the avatar will always walk in front of. Foreground objects include all non-opaque containers (tables, chairs), objects that the avatar can pick up, and objects that the avatar can walk behind. The designer's choice of foreground vs. background objects is important since the number of foreground objects will affect the frame rate of the screen. The faster the frame rate, the faster the interaction in the region.

Once this object is complete, the designer now has the option of creating a different object or making a duplicate of the current object (''twinning''). This will create an exact duplicate of the current object (class, style, pattern, position, etc) that the designer can now move around as an independent object. If you want two chairs in a room, create one chair the way you want it, twin it to make another, and now position the new chair to the location you want it. If you decide you didn't want the new chair, you can delete it from the screen, or if you want to change the first chair, simply move the cursor over to it, press a key to ''touch'' it, and now use all of the earlier commands to change its colors or position.

There are a series of special commands for control of region editing. These include a key that will redraw the entire screen (if you move an object that is in the background, it will leave a ''shadow'' image in the background buffer). There is also a key that will erase the background objects. This allows the designer to see just which objects are in the foreground in each region. Another set of special commands control the graphic images of certain objects. Some of the images have more than one ''state''. For example,

the image for the `house` object has some 10 to 12 states. These include the house with door, walkway, roof, chimney, and windows. Different states of `house` include versions without windows, chimneys, chimneys alone, just the walkway, and more. This allows the region designer to create different types of houses by adding these pieces together to make new houses. You can add another chimney, remove the windows, make a house two stories high, and so on. Similar variations of other objects exist, including pieces of trees, sidewalks in different shapes, and doors that open and close. These parts can be mixed and matched to create more interesting environments for the *Habitat* user.

There are also special commands for containers. If you have a bottle that you want to place onto a table, after the table and bottle are created, you simply ''touch'' the bottle, and point the cursor at the table and press a key that says the bottle is contained by the table. Objects are placed in the first available slot in a container, and objects can only be placed in valid containers. There is also a special command to ''look'' into closed containers. Say you have created a bag with tokens inside, you cannot see the tokens inside the bag, so you place the cursor on the bag and ''look'' inside it. The screen will be replaced with a screen displaying the contents. If there is another container inside the first, you can repeat the process to look inside the next level. This is useful if you place bags of tokens inside holes in the ground.

Next I will discuss special commands for signs, trapezoids, and flats. All three of these classes are quite powerful and as the world has evolved they have become even more important.

There are two classes of signs, short (up to 10 characters), and long (up to 40 characters). Commands for both classes act the same so I will describe them together. When a sign is created, it appears full of black spaces. The first command is one that erases the entire sign (this may be done automatically in the future). Next, you will want to enter sign editing mode. This mode allows you to enter text into the specified sign as well as special sign commands that alter the way text is displayed. The designer can enter every character in the C64 character set as well as commands for inverse video, half character space, increment width/height, decrement width/height, change between 4 and 8 pixels per letter, as well as move the cursor around. When the user reaches the maximum number of characters, the Commodore 64 beeps. Like a simple editor, the delete key will remove the previous character and replace it with a space. Also since signs tend to be slow to draw, it is recommended that they be placed in the background.

Like signs, there are two classes of trapezoid: `trap`, and `super_trap`. These classes allow us to draw a simple trapezoid by controlling the positioning of the four corners as well as the height. Triangles can also be drawn by moving two of the corners together. Like other objects, the designer can control positioning, colors, and patterns of the trapezoid. The trapezoids also have bordered and non-bordered states. This allows us to put a black border around all four edges. The `super_trap` is controlled in the same manner as the normal `trap`, except instead of only being able to choose between 15 patterns and 16 colors, you can design a pattern that gets drawn in the trapezoid. With this, you can create new wall patterns including bamboo, stone, wood, as well as other objects such as curtains, or buildings. This is a fairly new capability, and many of the new regions that are now being developed take advantage of this object.

Finally, there is class `flat`. This is an object that can act like ground, sky, wall, or ''impassable''. When creating a region, such as the beaches on D'nalsi Island, it was often desired to create a half beach, half water region where the avatar could walk on the beach but not on the water. With this class, you can create areas that avatars can travel on, as well as areas that they can't. Fairly simple in explanation, sometimes it is more difficult to use. Control is simply a key that changes the way that the object sees itself (from the four possible choices). The mode of the object is displayed on the top of the screen when the object is touched.

This is the basic command set for the region editor, but some changes will be made as the tool is completed. Most notable will be the addition of readouts on the top of the screen and a pattern generator. In order to speed development of the realm editing mode, we will take advantage of many of the already existing region editor commands and create a few new ones specific to realm creation.

When in the realm editing mode, the Commodore 64 will display a map. The cursor system will be changed to allow the cursor to appear only on grid coordinates. This will keep the region positions on the screen locked to an even grid for easier development. The creation of a region will be done through a single key stroke. Regions can be attached to one another via a method similar to putting objects into containers, but instead you will touch the first region and then point to the second region and press a key to connect

them. Since many regions will be created in simple grid fashion, there will also be a single key that will connect the region to the others surrounding it. Each region also has a 10 character name, and when a region is ‘‘touched’’, its name and the names of the four surrounding regions are displayed. This will make it easier to move around the map.

Special commands will exist for changing the horizon level of the region (currently this is set to allow the Avatar to only travel in the bottom 32 scan lines of the screen, but the value is adjustable). An illumination level for the region is also adjustable. All of these levels will be set on region creation to the standard default positions, but they need to be adjustable by experienced realm creators. At any point when the designer is moving around the map screen, he will be able to press a key that will save the current map, and load the graphic image for the specified region. Using the editor, we will be able to take ‘‘walking’’ tours around realms both developed and under development.

The realm editor will bring all of the power of the region editor together with the ability to tie regions together into an easily understood mapping and control system.

Programming Resources

Reno is currently scheduled at a total of 12 man-weeks. The schedule will involve one programmer working for 4 weeks doing initial work and preparation, 2 weeks of parallel development between two programmers followed by 4 more weeks of a single programmer for completion. All totaled, 10 weeks from start to finish.

There is also a need for Quantum to have software that will read the files created on a Commodore 64 disk and convert them into realms. We will be using standard Commodore 64 file formats when saving realms. This way, standard file transfer programs (including the one used by Q-Link) will be capable of transmitting the files from the developer up to Quantum. What is needed then is the software at Quantum that translates these files.

Both region and realm files are written in ‘‘contents vector’’ form. Contents vectors are the messages that the Stratus normally sends to the *Habitat* user when the user changes from one region to another. Reno has the capability of both reading and creating contents vectors and the files it creates will consist of these. What Quantum will have to write is a program that translates contents vectors (both realm and region) into *Habitat* database entries. Having both the region and realm files in the same format will simplify the programming effort. We will also be able to supply the programmer with sample contents vectors for actual realms within the first two weeks of programming on our end. This will allow us to test and debug both ends of the system with greater ease.