TABLE OF CONTENTS

How to Port from Windows to IRIX 1
- Finding and Linking OpenGL libraries at CTC . . 1
- Porting a multiple-file C++ program . . . 2
- Removing Windows-specific routines . . . 3
- Porting BMP Textures 3
- Dealing With Machine Architecture . . . 4
- User Interaction in IRIX 4

The Evolution of My Game Engine 5
- The Benchmarking System 5
- A Description of the Graphics Engine . . . 6
- Single-Pass Texturing 6
- Multi-Pass Texturing 6
- Optimized Square Root 7
- Using a Dot Product 7
- Removing Water Perturbation 7
- Removing Sky 7
- Other Programming Improvements . . . 8
- OpenGL Performance Considerations . . . 9
  - Grouping Triangles 9
  - Local-viewer vs. Infinite viewer . . . 9
  - RGB vs. RGBA Window 9
- Useful Tricks 10
- Conclusions 10

HOW TO PORT FROM WINDOWS TO IRIX

(In the following discussion, I will assume that Microsoft Developer Studio was used on the Windows environment and that no special development environment was used on IRIX.)

It turns out that porting across platforms was not quite as easy as I had thought, or hoped. I had many different concerns initially that needed to be addressed. As time progressed, I found that there were many different issues to consider in the porting process. These included:

- Finding and linking OpenGL libraries (and making a Makefile) on the IRIX environment
- Porting a C++ program that was split into numerous files
- Removing windows specific routines
- Porting BMP textures to IRIX
- Dealing with machine architecture
- User Interaction in IRIX

Finding and Linking OpenGL Libraries on the IRIX Environment

The SGI machines belonging to the Visualization Group at the Cornell Theory Center differ remarkably from each other in terms of graphics capabilities and file locations. Initially I started my work on Hack, which is an SGI Indigo2 workstation, but I found that its color depth was not set to the maximum 24-bit color. So I worked on the Onyx instead, because, in addition to having better color capabilities, it also rendered much faster and I was able to achieve a better frame rate with it.

Here is a nice and simple Makefile which is appropriately compiles a program called simple and includes all the necessary OpenGL libraries. To use this file, you need only to type: *make simple* at the UNIX prompt and the CC compiler will take care of everything for you. CC will look for a file called simple.C (capital C means that the source file is a C++ file, lower case c means that the source is an ordinary C file) so make sure you rename your main source file as simple.C . All the libraries that are linked in are already on the system and it will find them without any problem except for the Auxiliary library. I had to find this one myself and to add it using a -L (capital L means that you get to specify the path to the library). Please contact me if you need a copy of this library.

```
all: simple

CFLAGS = -I. -O

simple :simple.o
        CC -o simple simple.o -L../libraries/libaux/ -laux  -lXmu -lXext -lX11 -lm -lGL -
lGLU -lGLUT

clean:
        -rm -f *.o
        -rm simple
```

I think the best thing to do instead of using the auxiliary library is to stick with the GLUT library, which is already installed on the SGI machines. The main reason why I was using

the Auxiliary library is that on Windows, and in the OpenGL development guide, all the examples are done using “aux.h”. Slightly mislead perhaps…

Porting a C++ Program That Was Split Into Many Files

This was actually nothing much to worry about at all. Even though Developer studio makes you do all kinds of “Add File to Project” commands to add more C++ files to your project, on IRIX this is not a problem at all. All you need is for your main file (in my case simple.C) to have all the #Includes of other files. (They can be named anything you want, not necessarily ending in .C, as long as you specify the full filename in your #Include statements). This will allow you to link together multiple source files into one huge program with the advantage that you can easily isolate problematic bite-sized sections and edit them conveniently.

Removing Windows-Specific Routines

When working with OpenGL on the Windows platform you need to include the “windows.h” header. The key thing to understand is that the OpenGL Auxiliary library implementation differs by platform. So the “aux.h” in Windows needs “windows.h” but the “aux.h” in IRIX doesn’t. Naturally, if you try to include the windows header in your IRIX compilation you will get a lot of nasty errors. Because of the way windows.h works, you have to place the keyword CALLBACK in front of many special routines (your init, display, idle function, and key-bound routines, to mention a few) in Windows. However, IRIX does not need this keyword and it will generate errors instead. So make sure to delete all these keywords before you do your port. Of course, this means that you will not longer be able to compile your program in Windows so make sure everything is final and ready to be ported first. Also make sure you are careful about backslashes versus frontslashes.

Porting BMP Textures to IRIX

In order to texture-map in OpenGL you first need to convert the texture into a form OpenGL can handle. It turns out that what OpenGL needs is a texture in which red, blue, and green components are specified with one byte each. This means that each pixel is represented by three bytes (the alpha component is not included). The easiest way to get access to textures in Windows is to use “bitmap.c” which is a little demo program (one of SGI’s sample programs, I think) that converts a bitmap into a texture RGB format, and shows you how to take care of the texture mapping process. However, you cannot simply port bitmap.c into the IRIX environment because bitmap.c uses windows.h! Naturally, because the BMP file format is essentially exclusively meant for Windows. To get around this, I wrote a function that converts a BMP file into a “raw” RGB format. Essentially it chops off the BMP file header and stores the resulting data in a file. I then wrote a different library function to read in these raw texture data files and use them as textures. There was an important subtlety, though, which is mentioned in the next section below.

Dealing With Machine Architecture

MIPS chips (used in Silicon Graphics workstations) and Intel chips (used on most Windows machines) differ in Endianness. This means that for multiple-byte quantities, such as words, one stores the most significant byte first out of the four bytes, while the other stores the least significant byte first. This means that a word written to a file on an Intel machine and then read back on a MIPS machine will be inverted and garbage values will be read off! Unfortunately, I overlooked this simple point and spent many days trying to figure out why my texture files were not porting correctly. It turned out that in the

header for my texture format (.TEX), I put in two words specifying the length and width of the texture in pixels. Unfortunately, the Silicon Graphics platform read off negative values when it read the file, and all I got were horrible and inexplicable white textures on the screen. The way to get around this problem is to have a simple library routine that will check what kind of architecture you are on. This can be done by creating a word-sized variable (call it A) and assigning the value 1 to it. Then create a pointer to a byte and point to A. The pointer will be pointing to the first byte (out of four) of A, and we can check what value is at that memory location. If the value is 1, then we know that the least significant byte is stored first. If the value is a 0, we know the most significant byte is stored first. The result of this test can be used to set a flag that all architecture-specific calls can use to interpret data correctly.

User Interaction With IRIX

I found that using the OpenGL auxiliary library for event trapping resulted in very sluggishly performing routines. For example, if a key was pressed, held for a few seconds and then released, the computer would continue to process the "held down" event long after the release. Despite the problems with the keyboard interactivity, the mouse event handlers worked just fine. I think the best way to get around these problems is to use the GLUT library. The reason why GLUT works well is that it is more machine-specific than the auxiliary library and includes specific routines to help interact with windows and the operating system, whether it be X-based or Windows.

The Evolution of My Project This Semester

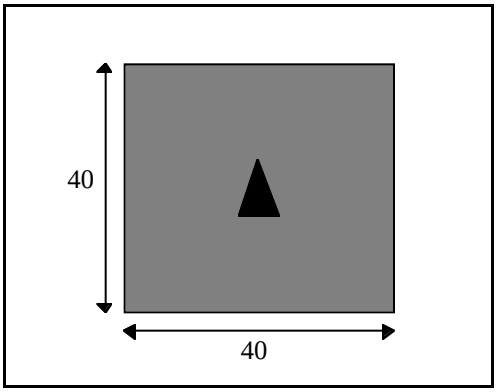
This semester my goal was to convert the OpenGL 3-D walkthrough program I wrote last semester into a tool to view and walk through landscapes. To do this, I added an array structure that holds the elevation at every point on the grid (map). In addition, I converted the original display( ) routine to draw a triangle mesh instead of a quadrilateral mesh. Having a triangle mesh allows total flexibility with respect to its shape since we can be assured that the triangles will always be able to share edges and maintain the desired shape. The first thing the program does is to read in the elevation file. Of course, the next thing I needed was a program to make the maps. So I wrote a program called MapMaker (described later). Once this was done, I modified the program to map colors and textures to the landscape according to elevation. After this conversion was complete, I had the facility to make maps and to effectively walk through them virtually. However, things were still slow on my home machine, as OpenGL is on machines without hardware acceleration. So I decided to focus in-depth on speeding up the program and learning about OpenGL's performance characteristics as well as general techniques to improve a "walk-through" engine.

The Benchmarking System

In order to compare performance characteristics I needed to establish a benchmark system of some sort. What I came up with was a self-timing loop in the program that told me how many frames were drawn in 60 seconds. As the program started (after all initialization steps), and system time was checked and stored. Then, each time a frame was drawn, a "number-of-frames-drawn" counter was incremented and the program checked to see what the system time was. If 60 seconds elapsed, the "number-of-frames-drawn" was output to the screen. By dividing this number by 60, I could find out how many frames per second

my program was generating. As I developed the program, I have kept track of the progress score-wise so that I can now analyze how it increased and what factors play a dominant role in OpenGL and walk-through graphics engine performance.

Description of the Graphics Engine



Since the map that I used was 200 by 200 squares, it would not have been feasible to draw the entire map each time since, clearly, the player could only see a certain number of squares ahead. So I ended up working with a region 40 squares by 40 squares with the player centered on it. I then modified this scheme to make it more and more efficient.

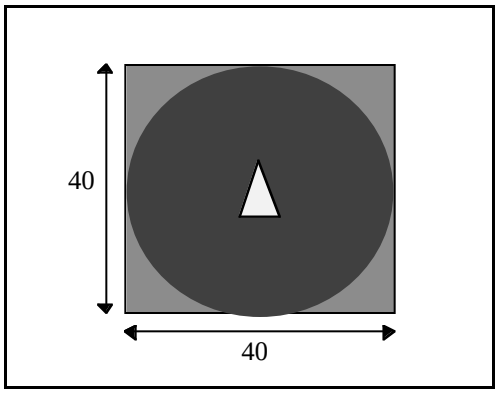
Single-Pass Texturing: 158

Because my scene used many different types of textures (water, beach, grass, rock, and snow), I was not sure whether to implement the main drawing loop as a single-pass or as multiple passes. Single-pass meant that I would go through each of the 400 squares to be drawn and switch textures for each square (or, more efficiently, switch if the current square had a different texture from the previous one). This technique turned out to be extremely slow, even though the textures were switched using display lists.

Multiple-Pass Texturing: 216

Instead of constantly swapping textures, I used multiple passes (each with the texture set to something) to render the scene. This way, I only had to swap textures five times (once each for water, beach, grass, rock, and snow) instead of hundred of times. The disadvantage of this technique, however, is that you actually have to loop through the squares to be drawn multiple times. But, if the innermost loop is optimized so that it checks if the current square is of the current texture, the loop becomes a very quick series of iterations which is quite efficient.

With Optimized Square Root: 246



At this point, the engine was drawing all 400 nearby squares, regardless of whether they were behind the viewer or further than the maximum visibility. I realized that if I did a quick distance test I could cut the number of squares to be checked significantly because now the area to be checked would be that of a circle instead of a square. Thus the number of squares to check would be reduced from 40*40 = 1600 to $\pi*(20^2) \approx 1257$. This resulted in a 14% performance increase. An important note: there is no need to actually take the square root for the distance, one can simple compare the squared distance with the square of the desired distance.

With Dot Product: 298

Now that I had restricted the number of squares to a circle, and noticed a significant performance increase, I further reduced the number of squares to draw using a simple dot product test. This meant that squares behind the viewer were no longer drawn. Also, squares outside of the viewer's field of view were no longer drawn. This increased the score by another 25% or so.

Without Water Perturbation: 333

Until this point I had put in a few extra lines to perturb the water surface's lighting to make it look like it was moving around. But I realized that it would be more useful to swap water textures instead because the calculation was taking too long. In addition, although the water perturbation looked good on slow computers, on faster ones it looked like the water was flickering, which was not a nice effect.

Sky Removed: 434

Removing the sky was the single largest improvement that I did on the engine (increased the number of frames rendered in 60 seconds by 143 – a huge improvement in frame rate over the same system with sky present). Originally I had a large texture mapped rectangle that served for the sky, but I realize that using something simpler makes a big difference. In particular, I was only showing a certain fraction of the rectangle depending on the player's view direction so that the sense of rotation was improved. I even put each segment of the rectangle into a display list to help improve performance but to no avail. For some reason, OpenGL does not handle the texture-mapping of a piece of a large rectangle well at all. So I took this out, and I will be replacing it later with a simpler "sky" pattern which will probably be a series of gradated non-textured rectangles. The bottom line is that the huge performance loss is simply not worth it just to have the sky texture map. Scores : 496 - no sky , 353 - sky

Other Programming Improvements: 510

A series of other programming tricks and algorithm optimizations increased the score to 510. These improvements were mainly to do with tightening up the inner loops as much as possible and reducing redundant code, as well as adding some pre-calculation stages to save time. One of the best things I did was to remove the GL_CLEAR call because for some reasons, this seems to execute quite slowly, at least with software OpenGL. Instead, it is easy just to draw the horizon and sky, followed by the scene. This way, you don't actually need to clear the screen because new data simply overwrites the old data, and the sky clears off the only region that could have provided evidence of this. By cutting out the GL_CLEAR call I was able to get a 2 point improvement in "still" benchmark score (346 vs 348) but the performance increase is substantially greater if the player is in motion throughout the duration of the benchmark (363 vs 381). I'm still not quite clear why a GL_CLEAR call seems to be so inefficient. The depth buffer needs to be cleared as well though. I have not experimented with them but I have thought of methods to bypass that clear as well. An easy solution would be to draw the further things first, starting with the sky, followed by the distant landscape in view and finally end with the closest squares. However, I think the increase in processor time needed to figure all that out would not outdo OpenGL's depth buffer clear.

OpenGL Performance Considerations

Here are some more interesting influences on OpenGL performance…

Grouping Triangles: First and foremost, when defining triangles, use triangle strips or triangle fans. The more triangles you are able to put together in one continuous strip or fan the faster your program will run. The reason for this is that the graphics pipeline has a lot less to deal with in terms of vertices when you use a strip or a fan because vertices are shared optimally. Each vertex entered into the graphics pipeline needs to be clipped, colored, lit, depth-buffered, and rendered on-screen, so minimizing this is probably the single biggest improvement you can find. I am still working on optimizing my innermost loop to optimally draw strips of triangles if they are all of the same texture type.

Local-viewer vs. Infinite-viewer: One was the choice of local-viewer versus infinite viewer for the lighting model. It turns out this did not have much of an effect at all. The scores were 445 (local-viewer) versus 456 (infinite viewer). This means that unless the extra 10 frames per 60 seconds are truly crucial, one can get the benefit of a better lighting model by using the local viewer model. The local viewer model would probably give better specular highlights, which would be important in a situation where metallic materials are used.

RGB vs. RGBA: Another minor increase (about 7 points) could be gotten from making the window into an RGB window instead of an RGBA window as it originally was. This would be an appropriate design decision in a case where no transparency or other use was made of the alpha values. However, I think it is well worth to keep the alpha capability.

Useful Tricks

The lighting array: setting up your own lighting array can help a great deal. The way I set it up in my engine, each square has additional an additional array entry associated with it that tells what extra lighting to add to its color. Since these additions are performed *before* the data is sent through the OpenGL graphics pipeline, you can save a huge amount of computation at the expense of coarser and more crude lighting. This is perfect, however, for things like shadows, lights along a race track, or other minor additions that can have a large effect on the viewer's experience. To improve the coarseness problem, some modifications could be made to the engine to refine areas around lights into more polygons. The amount of refinement would have to be chosen so that a suitable compromise between performance and appearance was established. I found that even the relatively large triangles my engine works with were adequate for things like shadows and lamp posts because OpenGL's lighting model naturally smoothed them out with respect to other polygons.

Conclusions

Even though OpenGL does a significant amount of clipping for you, *just sending* something to the pipeline for consideration is a slow process. I have found that it is a lot faster to clip off extraneous objects yourself before sending the scene to OpenGL to render.