Project 1: Software rendering system

Introduction

In this project, you will develop a real-time, software-based rendering system from scratch. This system will be a simplified version of the graphics pipeline that is implemented on conventional graphics hardware. You will be responsible for writing all the principal blocks: vertex transformation, clipping, rasterization, texturing, and framebuffer operations, so at the end of the project you will have built a system that takes in the list of triangles and renders them into an image to display to the screen. There are two purposes to doing this implementation in software. First, by writing the code yourself, you will gain an understanding of the graphics pipeline that can only come from dealing with the special cases that come up during implementation. In addition, this project will help give you an appreciation of the power of the graphics hardware, which we will start using after this project!

History

One of the first consumer applications to feature "3-D" textured polygons was Wolfenstein3D by id. This was back in 1992, before any graphics hardware was available at the consumer level. For this reason, lead programmer John Carmack had to write a 3D rendering engine that ran completely in software and did so in real-time on the machines at the time (Intel's pre-Pentium 486 DX2-66).

Figure 1. *Wolfenstein3D* by id (1992)

The Wolfenstein3D rendering engine did not implement the real-time rendering pipeline as is available on graphics hardware today. Instead, Carmack developed a system based on the ray-casting algorithm for computing visibility. Ray casting had the advantage over scanline rendering in that it required little memory overhead and could be accelerated

with clever data structures. However, since we now have computers with ample memory and powerful CPUs today, it is possible to simulate the conventional real-time rendering pipeline completely in software. This is your task for this project.

Real-time Rendering System
As we discussed in class, the real-time rendering pipeline is composed of various stages. You will be implementing the following stages:

Transformation – You will need to implement the modelview and projection matrices given the eye position, view direction, field-of-view, etc, just as is done in OpenGL. In particular, to create the modelview matrix you will be given the view direction (expressed by `eye_theta` and `eye_phi`) and the distance from the viewer to the world-space origin (`eye_pos[3]`). To compute the projection matrix you will be provided the same parameters as for `gluPerspective()`, which are the field of view `eye_fov`, the aspect ratio of the frustum, and the near and far planes. To calculate these matrices you might want to create some sort of matrix class and implement some common operations such as matrix-matrix and matrix-vector multiplications.

Clipping – Once the vertices have been transformed by the modelview and projection matrices, you need to clip them to the canonical view volume using the algorithm described in class. You must clip against all six sides of the bounding volume. In my implementation, I maintained a list of the triangles that needed to be rendered and added the new ones created from the clipping process to the end of the list. After I had clipped against each of the sides of the view volume, the triangle list was sent to the next stage of the pipeline.

Triangle Setup – After the triangle has been clipped, you can set it up by performing the normalization (homogenous division) and then by transforming it by a viewport matrix so that the vertices are now in screen-space coordinates and ready for rasterization.

Rasterization – This is one of the main parts of your program. The rasterizer will take the triangle and convert it to an array of pixels on the screen. Your rasterizer should interpolate color, depth, and a single pair of 2-D texture coordinates across the triangle, making sure that the interpolation has been corrected for perspective as discussed in class. Note that you do not need to interpolate a particular interpolant if it will not be used in the final output. For example, when drawing a texture mapped square you do not need to interpolate the color, because in our application we will never blend colors and textures together. Likewise, when drawing a color-interpolated triangle you will not need to interpolate the texture coordinates. When rasterizing the triangle, only rasterize the pixels whose centers lie on or inside the triangle boundary. If the boundary goes through the pixel sample, develop a rule to render the triangle as presented in class so that pixels touched by adjacent triangles are not rasterized twice.

Texture mapping – The scenes to be rendered contain textures which have been bound to the appropriate triangles by my scene code. You should take the texture coordinates interpolated from the rasterizer and use them to fetch color samples from these textures.

You need to implement two kinds of texture sampling modes: nearest neighbor and bilinear interpolation. You do not need to worry about mipmapping the texture. The fragment will be automatically shaded by the value fetched from the texture (i.e. you do not need to modulate this by the color or any other value, as is commonly done).

Framebuffer operations – In graphics hardware there are typically a number of operations that can happen at the framebuffer. For this project, all you need to implement is standard "less than" depth buffering.

As I mentioned earlier, this pipeline is considerably simplified from what is currently implemented on graphics hardware. For example,

Skeleton code
To get you started on this project, I am providing some skeleton code that you should use to build your project. This code allows you to interactively navigate a maze, like you would in a simple first-person shooter. To render the maze, the program first reads in a file (passed in as the command-line argument) which specifies the geometry and textures which will be used in the maze. Once loaded, the viewer allows you to render the scene in two modes: OpenGL (which runs on the hardware) and software mode (which does not). I have written the OpenGL version for you – it was fairly trivial because the GPU essentially does all the work. Your task will be to write the software portion of the code. To toggle between the modes, hit the space bar. You can look around by moving the mouse around the view window and you can move by using the 'w', 'a', 'd', and 'z' keys.

The code is written in C++ and built using a Visual Studio project. I use OpenGL Utility Toolkit (GLUT) to handle the mouse and keyboard events as well as draw to the screen. Note that this means that GLUT will be used to handle these tasks when you are running in software mode as well. To help you get going, I will now briefly describe the code.

Upon starting, the program initializes GLUT which opens the window and establishes the appropriate callback functions for handling the display, keyboard and mouse. Of particular importance is the `display()` function, which is called when GLUT is ready to refresh the window. This function first updates the modelview and projection matrices by calling `computeModelViewMatrix()` and `computeProjectionMatrix()` respectively, then proceeds to render the scene. If `opengl_test` has been set, it draws the scene using the graphics hardware by calling `scene.renderSceneOpenGL()`. If it has not, it calls `scene.renderSceneSoftware()` (which is the function you need to fill in) and then calls `fb.dumpToScreen()` to dump the contents of the framebuffer onto the screen.

The mouse functions found in mouse.cpp implement the mouse movement to control the position of the viewer and change the view direction. Here you will find calls to `setPerspectiveProjection()` and `setModelviewMatrix()`. Fill in these functions to set the projection and modelview matrices you will use in your code. Remember that you

will have to compute these two on your own, you are not allowed to use any existing graphics libraries to do this.

This project has many associated classes which will be useful to you as you implement the rendering system. Let me go over briefly what they do.

Framebuffer – This class implements a basic framebuffer. When the constructor is invoked, it allocates an array of unsigned bytes of size `width x height x 3` to store the color data. The method `dumpToScreen()` from this class uses the glDrawPixels() call to dump the array of color data to the screen. Do not modify this call, and remember that you are not allowed to make any OpenGL calls in your code! To do this project, you might consider modifying the Framebuffer class to include things such as a depth buffer which you will need to perform the hidden surface determination.

Scene – This class contains a linked list of triangles that need to be rendered. When the scene description file is parsed, the method `addTriangle()` is called which adds the triangles to the list. The list is of type `TriangleList` and is pointed to by `*original_head` with the last element pointed to by `*original_tail`. The structure `TriangleList` is also defined in Scene.h, it essentially has a pointer to the Triangle structure and one to the next element of the list.

Triangle – This class implements the triangle primitive which will be rendered by our engine. It contains the three vertices, the colors and the texture coordinates at each vertex, and a pointer to the texture that is bound to the triangle.

Texture – This class reads an image from a file and stores it in an array for texture mapping. You will need to add code that does the nearest-neighbor and bilinear interpolation of the texture.

Vertex – A simple class that handles the implementation of a vertex. You might want to write a similar matrix class and then create operation methods that allow you to multiply matrices by vectors, etc.

You can download the skeleton code from the course website after class at *http://www.ece.unm.edu/course/ece595/hw/Project1.zip*

Getting started
This is a large, complex project and it is easy to get overwhelmed. The best approach is to compartmentalize it and work on a small portion at a time. Make sure that portion works before moving on.

To start off, just try to get the code compiling and then modify it so that you can write something to your framebuffer which should appear on the screen. Once you have this, I would recommend implementing the rasterizer first. Assume that you are given a pre-projected triangle that maps to the screen. Write the code that will fill in the appropriate

pixels with a solid color. You can debug your output by drawing the same triangle in the OpenGL side and verifying that the two match. It might be a good idea to start with a simple rasterization algorithm before moving on to a more aggressive algorithm.

Once you have the rasterizer generating pixels of a solid color, then try interpolating the color. Then work on the texture coordinates, and implement the texture fetch mechanism. By this point you will have textured triangles appearing on the screen.

Next implement the vertex transformations. At the beginning set up a super conservative culling algorithm that will reject any triangle that goes off the screen. This will help you get the functionality of the vertex and fragment sides working together before you worry about clipping. Again, take advantage of the OpenGL code to ensure that your code is rendering properly. Once this is done, all you have left is to implement clipping and then optimize your code as much as you can!

Tips & Advice
This is a big project, so get started early. If you wait until the end you will panic and will not be able to finish. I will not allow late days on this one. Functionality is the main part of the grade, so make sure your code works properly. Write a bunch of test cases and make sure you are rendering things properly. Focus on that first, and then worry about making it fast if you have the time.

Some tips to accelerate the rendering: The rasterizer is essentially the inner-loop of the code. Try to make it as fast as possible! Also, you are free to use assembly, multithreaded programs, resorting the triangle list or other data structures to try to accelerate the rendering as much as possible.

Warnings
You are not to use the graphics hardware, OpenGL, DirectX, or any other graphics API. The only allowed calls are the ones I made in the skeleton code. The purpose of this project is to learn by doing. If you cheat and use the graphics hardware, you will get a zero for the assignment. Of course, you can use the graphics hardware for debugging through the OpenGL code path, as I described before.

Grading
The following breakdown will be used to grade the project:

Functionality	75
Performance	15
Write up	7
Documentation	3
TOTAL	100

Functionality

In this project, functionality is the most important. Your code should work correctly, for all my test cases. In particular, I will assign the grade for this section in the following manner:

Rasterization	30
Correct pixels are identified for each triangle	10
No double-counting of pixels from adjacent triangles	5
Colors are interpolated correctly	5
Depth is interpolated correctly	5
Depth buffering is performed correctly	5

Vertex transformation	20
Modelview and projection matrices are correct	10
Vertices end up in correct positions before rasterization	10

Texture mapping	15
Texture coordinate interpolated correctly	5
Nearest Neighbor texture access works correctly	5
Bilinear interpolation texture access works correctly	5

Clipping	10
Triangles are properly clipped to the view volume	10

Performance

The second portion of the grade will be based on performance, which is also important for a real-time application. To grade this, I will compare your implementation to my own and use the following formula: `grade = (my_time / your_time) x 15`. So if my system renders in half the time of yours, you will get 7.5 points. If yours beats mine, you will get extra credit. Note that performance will only be graded if your functionality is completely correct. After all, it is easy to write extremely fast code that doesn't work. If you do not have everything working properly, you will get a zero for performance.

Project write up

I want to you prepare a short report (approx. 3 pages) that discusses your rendering system. You should talk about the choices you made regarding the different portions of the pipeline. For example, what rasterization algorithm did you implement and why? What did you find easy or difficult? What unexpected issues came about during your implementation? On the day the project is due, I might ask some of you to comment on some of these issues.

In this write-up, I also want you to address a few things. Undoubtedly the first time one implements texture coordinate interpolation they do not account for perspective and get the wrong answer. Include a couple of images of a checkerboard-textured quadrangle that shows the difference of perspective-correct and incorrect interpolation.

Documentation
Be sure to comment your code so that it is readable! Someone else should be able to look at your code, understand what you are doing and follow your implementation. I will spend some time looking at your submitted code and try to follow your algorithm. If I can't understand what is going on, I will deduct points from here but possibly also from the algorithm!

Final words
Get started soon, and feel free to ask me questions if you get stuck. Good luck!