Jon Baker
CS4250 Graphics Final Project

Preface: You need some pretty serious memory bandwidth on your GPU to handle the compute shader portion of the program (many, many texture references). The machines in 107 will crash if you try to do more than 20-ish steps, I have not had a chance to try it on the third floor machines.

I'll start off by saying that the techniques that I have access to that I did not before the final project began have huge potential and are going to take quite a while to take significant advantage of in any sort of efficient way. By having access to a huge read/write buffer like I do here, a lot of potential is further opened up, the simplest of which is making my program that constructs voxel models take advantage of the GPU parallelism in order to compute values for each cell, in the way that my earlier project did when it computed whether a huge cloud of points (millions) were each individually inside of a set of parametric objects defined by uniform arrays. Making that better, you would just have one thread per texel and you could just set up your uniforms the way you wanted and draw things into it in a hardware accelerated fashion, then set up a different set of uniforms, call the shader again, etc, etc, ad nauseum - many times faster than triple nested for loops to iterate through a three dimensional array. I will figure out compute shaders, as that's the correct (and easier) way to do what I did for this project with a vertex shader, but that's really just a detail.

Ok - here's how things work:

Initially, I just set up some geometry to get fragments that cover the screen. I was experimenting with dishing this geometry out to get a lens effect but I kind of scrapped that in favor of pretty-much-orthogonal. I use the position of the fragment in the clip space to offset the vector that represents that pixel so it is closer to perspective projection. When I am in the fragment shader, I have the position of the camera, the rotation variable to control things in two directions and the position offset and the ray representing the pixel. All these together allow me to construct a ray with a given position and direction, which I can test against an AABB for the 3D texture. Since switching to images, looping at the edges no longer works as part of the reference through OpenGL's access methods, but I could manually do it in the shader with some kind of mod type of logic. So, if I see that this ray hits this AABB, I begin to trace out a bunch of samples along that ray, with a sort-of-adaptive step size - notably, the order that I trace them in is from back to front so that I can do my own alpha blending. As I'm not using OpenGL's geometry, I don't have the option to use their alpha blending.

Once that process is finished, I have a final color for the pixel - I do not set depth, because I'm not sure how I want to handle transparent things, to have a threshold alpha value that you would consider 'solid', and set the depth at a sample that was above that threshold maybe. That remains to be seen. That's the bulk of the rendering code - now the compute shader portion of the project.

I'm not actually using a compute shader, rather I'm using something I saw called attributeless rendering of 256*256*512 verticies and using the resulting glVertexID to compute an ivec3 to reference unique cells in the image3d grid, computing the result of the cellular

automata, and then writing it back to the same cell that we started with. The attributeless rendering process is done by setting up a shader, then not specifying any geometry data, and then just invoking an arbitrary number of threads on the GPU by using glDrawArrays(points,0,n) - this runs your vertex shader a huge number of times (256*256*512 is approx 33.5 million shader invocations), and since my fragment shader was empty, I won't see any visual result.

The classic cellular automata wireworld generalizes perfectly to 3d, with no changes except that it considers 10 more neighbors. The rules are these: conductor with either 1 or 2 (NOT 3+ neighbors) electron head neighbors becomes an electron head; electron head becomes electron tail, and electron tail becomes conductor again. From these simple rules arises surprisingly complex behavior that I've seen explored extensively in 2d. People have schemes for 4 tick, 6 tick, n tick logic and they have gates figured out and encoders, decoders, adders, on and on. In the vertex shader that logic is applied, by doing some more texture references in the neighboring cells, and the value of the next state of the buffer is computed.

I use two buffers for avoiding coherency issues - by using one as read and one as write, per frame, I can make sure I don't step on my own toes and create unpredictable behavior. I can simply swap the uniform integer which identifies them, in both the display shader and the compute shader, to swap which buffer to look at - so that is a very cheap process, too.