# Jon Baker - Final Project Proposal

First off, I appologize ahead of time for the poor formatting of images, Google Docs has become pretty limiting in the options it provides for images.

I had experimented this summer with volumes, creating them, as a solid model consisting of voxels the way that images consist of pixels, and a way to visualize them. Part of what appeals to me about this representation is the fact that through that space, at all points, what is 'inside' that point, or what exists at that point, is defined. When dealing with polygons, you're dealing with a surface representation, but if you wanted to look in space to see if you were inside or outside of a polygonal model, you'd have to use a different representation of your object or get more clever with the math you do in order to determine that.

There were some interesting results from the project that I worked on this summer, called v07 - part of a series of projects over the past year that I've kind of just generally been calling Voraldo. There are a wide number of considerations, when trying to come up with an implementation for this kind of a graphical representation - I will describe some past approaches below.

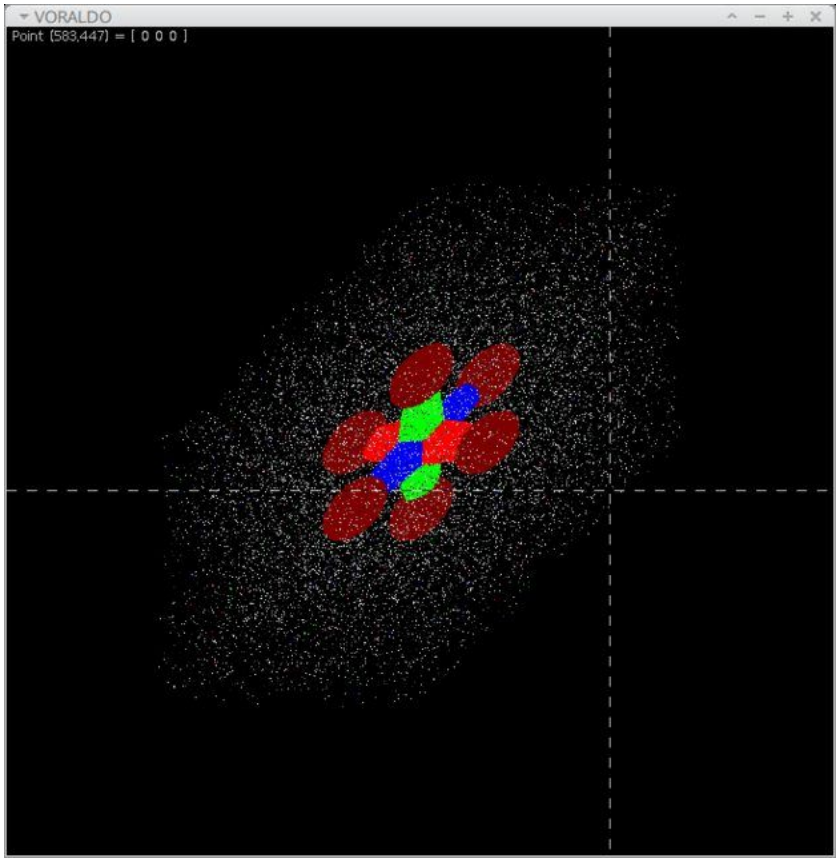
## past approaches:

This started from some experiments with heightmaps - I like that representation for similar reasons, because it partitions space in a way that's more complex than a plane, but without getting into a complex mathematical representation - it goes from 2d data to a 3d representation of a shape when you consider it the right way.

A couple years ago now, I found some code for how to write a windows BMP image, all the header information, how to order the bits when you write it all out. So I had some code that would maintain a 2d array representing the RGB values, and let me draw a picture, and save it as a BMP. I was drawing a heightmap isometrically by picking a point to be the middle, then drawing as many pixels up from that point as the heightmap indicated. I'm not recalling right now how I was loading the input image, but I may have been generating it with something like the diamond square algorithm. In any event, once that column of pixels is drawn, I would return to the pixel that was down and to the left one pixel from the starting point for that column I had just finished. Then, again the same process, the column, then move your base pixel, the column, etc, till you finished one strip across the image. The next row then would be drawn, by moving down and to the right of your original base pixel, and then drawing that column.

This was pretty rudimentary, and I didn't do much to extend it, beyond trying to see what would happen if instead of drawing the whole column, I just drew the pixel at the very top, or what would happen if I used different schemes to color it based on how high it was. I was introduced to some python bindings for OpenGL around the same time, and messed with cubes made of a bunch of points sort of representing voxels - that didn't really go anywhere either.

The first iteration of Voraldo came a while after that - I have the repositories on my Github account, for all the versions going back to this, mostly just for if I want to go back and mess with different ideas, or just for reference on what I was trying to do. Version 1 of Voraldo used a very similar scheme in terms of placement of the volume elements, in terms of constant, usually single pixel offsets. I don't have the code from earlier versions, that's from before I learned to use git at all.

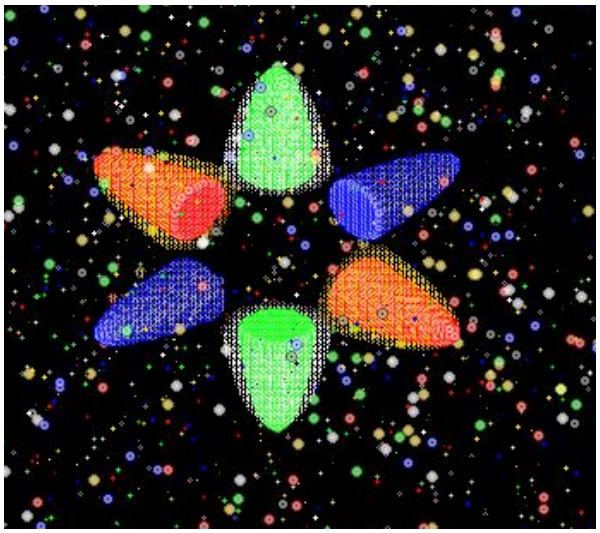


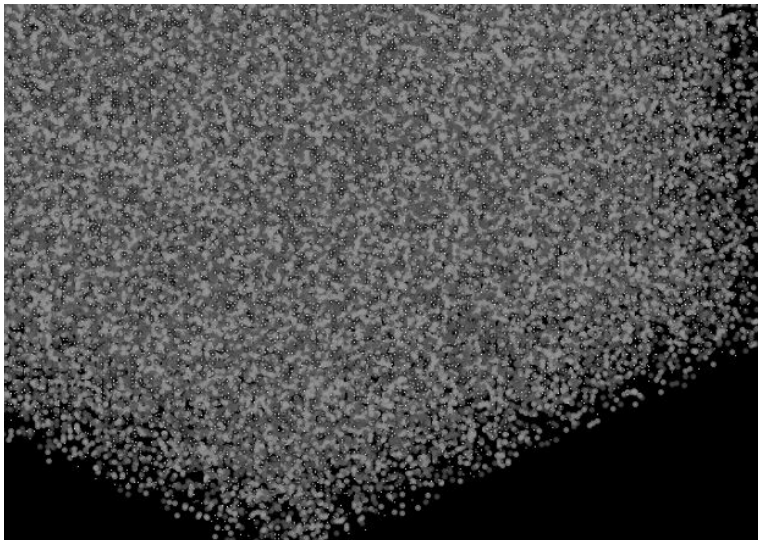

Version 1, using the Cimg display() window.

Version 2 took a slightly different approach in that it was using something other than pixels to represent the individual voxels - I was using a framework called Cimg, which gives a whole lot of image processing capabilities in addition to saving and displaying images, but among the things provided was a routine for drawing a circle, allowing you to specify a color and size. This came with the benefit of some amount of depth, because the order in which you drew the circles really did start to leave some artifacts, in terms of how they would draw over each other in an order dependent manner. I liked the look of it, there's some good examples of outputs from it, but it was pretty limited and like the first version, it was something that ran offline and generated an image. The Cimg circle routine took an opacity factor, which was one of the first times I had messed with alpha blending in this context. It keeps the same reliance on order, as what we are doing with OpenGL, but it made sense, because you were doing these as separate calls - it makes sense it would change the state.

As of yet, I'm still figuring out alpha in OpenGL, because while most things OpenGL are not order-dependent, this is one thing that remains as a holdover from immediate-mode stuff with a changing state that kind of breaks the parallel nature of things. I will describe some thoughts I have had about how to overcome this later in this document (multiple sets of slices, discarding backs of slices).

V3 was focused more on doing some animation, and went back to the rendering method used by the first version. There really wasn't much to this, because it was when I started messing with animation a little more, it made multiple views, like what you'd think of when looking at engineering drawings.



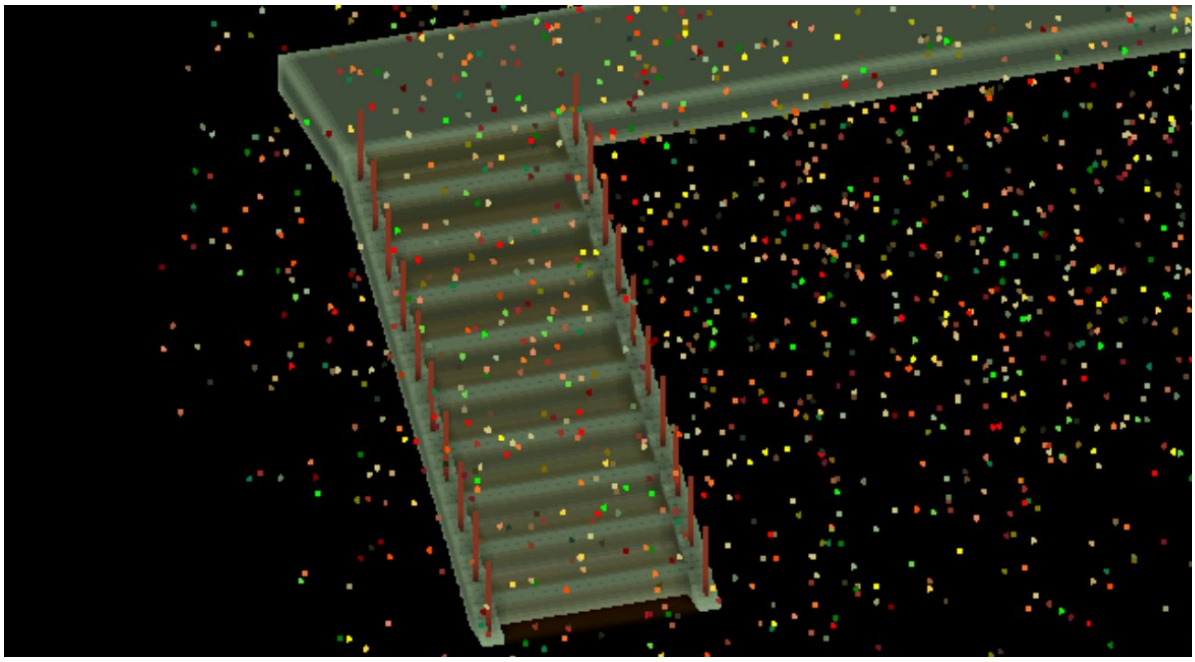


Version 2, drawn with circles - starting to mess with alpha.

V4 implemented a renderer that I had thought a lot about - in essence, it took some ideas from ray racing, without getting into much detail on lighting. This first version simply created a single ray per pixel, traced them into space where these voxels were defined, and

returned a color when it hit something with a nonzero value. At this point the voxel values were simple integers that I would map to colors in different ways. This was really cool, to me, maybe more at the time than it is now, because of how closely it mimics the behavior of light, as rays.

V5 got more structured, I did some documents that spelled out a system where different pieces of functionality were in different classes, so that I could kind of scale that up. It defined a palette so that I could go straight from integer to vec3 color, simplifying the code that I was using to get the color. It also included a number of optimizations, most important of which was a bit of ray-box intersection code that allowed me to stop evaluating some pixels before tracing through empty space, as well as giving a near and far intersection point, so that you could trace even less empty space. Over the way I had been doing it before, it sped it up almost 10 times. This was still completely independent of OpenGL, only outputting the images as files, or stitching them together with ffmpeg to make short animations.



V6's ambient occlusion, applied to sort of like a staircase.

V6 implemented a few cool things that I had wanted to mess with - still, not touching anything hardware accelerated. First among them, ambient occlusion, the component of the lighting which comes from the ambient light, and how it is blocked by neighboring objects. This was a simple computation (if very slow) done by looking at a neighborhood around each voxel, and taking a ratio of how many of the cells you touched were occupied vs how many were not. Ideally, you might figure out weights for each of these so that it was less uniform. I talked to someone giving a paper at SIGGRAPH who took this same approach but they were calling it 'shells', essentially doing the same computation I was, but with a much more refined approach, GPU based instead of this one, which is CPU based and single threaded, to boot.

The next - transparency - this is achieved in a way that I've considered for other things - basically if you imagine you took a bunch of samples along that ray, like a core sample, and then just drew them from most distant to closest, using the Cimg library's functions that took an

opacity parameter. I have thought about this for volume raycasting of signed distance fields, but instead of using Cimg, I would be just manually doing the alpha compositing in the shader, averaging the closer, nonzero samples with those further back till arriving at the final pixel color - at the time, I was doing sort of a stack-based thing where samples got pushed onto a stack as you traced the ray through the volume, and popped back off as the compositing happened.

The third, was interesting, and kind of didn't work as well as I wanted, but it involved directional lighting computed in a very similar way to how the display function was working. In essence, there was another value kept per voxel, representing the intensity of the lighting computed. This had some artifacts, due to the cubic nature of the voxels - there were often crenelations on the edges of shadows, where rays would have been intercepted by the sharp edges of voxels. Again, very slow, some animations would take nearly a day - the lighting was essentially the same algorithm as the display function, but at a much higher resolution. The one that I thought turned out best is linked here - https://www.youtube.com/watch?v=ABLM5HDZmJs - this took approximately 3 minutes per frame, because the model is generated, the lighting is computed, the ambient occlusion is computed, and then the display function runs, per frame - initially I didn't have to recompute the lighting per frame, but in this case, the geometry changed so I had to. This process takes a good portion of a day for a 30 second video - not super efficient (just more of a margin to be amazed by when you go to hardware acceleration, I guess). This is the only version which has had this lighting scheme implemented - I think it would be cool to mess with this more in the future, or something similar.

V07 was the first time I used hardware accelerated 3d graphics to represent the voxels. I found out how to define a 3d texture, and how to output files which would play well with that representation. The way that ended up is very tall PNG images, which take slices along the z axis and put them out like if you imagine someone sliding out a deck of cards into a bunch of cards side to side. The actual rendering was achieved with a bunch of polygonal slices through this volume,  at each fragment on each triangle, referencing the appropriate location in the 3d texture. By use of alpha blending, you can support transparent objects inside the volume, that is, as long as those polygonal slices don't violate the back-to-front assumption of the alpha blending - if the order changes, for example, when the shape rotates 180 degrees, you will get significant artifacts that prevent you from seeing into the volume. There was also a limitation in that if the volume was viewed exactly from the side, you would be able to see through between the slices. In addition to that, the lighting wasn't implemented in this version. I did some interesting stuff, though, with sort of a planar cursor, which can be varied in thickness and offset, and is used to consider only the slices which are within some variable distance of a slice that you can move through the volume. This allows for some CAT-scan/MRI-like interaction with models. This really speaks to me, it's a really, really neat way to see solid 3d information on a 2d monitor.

V08 is the latest iteration, and as of yet, has no visual output - it was an extension of a client-server program I did for Operating Systems, changed so that instead of just one client connecting to a server using fork(), an arbitrary amount of clients could connect to the server program and send it commands via pipe to draw shapes into the volume, using poll() or select() to read the messages from clients. I think this might make for some interesting potential related

to collaborative editing or something like that. I haven't had much time to put into this during the semester, but in its current state it can handle a pretty significant subset of the primitives which I have had Voraldo draw in the past (noise, spheres, some other geometric shapes, I'd have to check though, because I haven't messed with it for a few months). The server program holds the voxel model, draws to it when it receives a message, and outputs the model as a very tall PNG image when told to do so. This is the same format which I have used to represent these models in the past - I find it pretty favorable because you get to take advantage of PNG compression, while employing a lossless algorithm which ensures you're going to get all your pixel values back the way you expect. I have yet to encounter a limitation on the PNG format, at 65k+ pixels along an axis - in addition to that, the filesize is pretty minimal, a few, single digit megs at most (vs the 15+ megabytes you should be dealing with, uncompressed). This is also kind of nice since I've got an SSD in the laptop I'm using and they rate SSD lifespans in 'total bytes written' now, it's just generally nice to avoid having more data than you need flowing around.

# the final project:

I had a thought in this last project, while dealing with the front and back faces of triangles differently, using the gl_frontfacing built-in variable in GLSL - in v07, I had the issue with ordering, only when the faces of the triangles are facing away from me. So long as they are facing towards me, I know that they have valid contributions to the output pixel - so why not discard those fragments which are on a polygon with gl_frontfacing set to false? If you simply have two sets of geometry, in opposite order, and apply this logic, you can be looking at a system where you can look at it, front and back. I thought also, why not put things in the yz-plane and the xz-plane rather than just the xy plane, so as to avoid the issue present when viewing from the side. I have yet to experiment with this as a real solution, so I do not know if you would end up with more significant artifacts from composing the slices along different axes, if they're being drawn in different orders, but I do get the sense this might be viable. In addition to that, you could look at putting other geometry inside of that space, and you could be drawing polygonal shapes over the geometry which is defined inside the sliced volume.

I want to elaborate on the texture part of the volume, as well. Using part of the core spec which is called 'arbitrary image load/store', which uses image objects in the shader rather than sampler objects, and allows for writing to those textures. I want to represent a 3d version of wireworld, which is a classic cellular automata that does a good job of representing a rudimentary model of electricity through a wire, allowing for logic gates and branching and generalizes well from two dimensions to three. In essence, the rules are these: there are four states: conductor, electron head, electron tail, and a catchall state which represents everything else as 'emtpy'; if a conductor has either one or two electron head states in its neighborhood (not considering diagonals) it becomes an electron head - less than that, or greater than that, and it remains a conductor; electron heads become electron tails after an update, and electron tails go back to the conductor state after an update. That's all there is to it - it is just as simple as it sounds, and requires only knowledge of the states of the cell being considered and that of the

neighbors at the last update. The diagram of it as a finite state machine fits in the margin of a page.

The appeal here, to me, is the fact that this can be unified with a model of an object - in the same way you could talk about 3d printing a circuit into a plastic object, here, we're talking about wiring which could be included within a volume - assuming you consider all the non-wire, non-electron cells in the volume as the 'empty' state. If you came up with a model for damage to this object, for example, as a space ship, you could be looking at a situation where the wireworld simulation's parameters change, as a consequence of the fact that the image changed (damage to the wiring, electrons no longer are able to pass). By bringing these two representations together, I can do a lot with compute shaders on the GPU to evaluate the cellular automata, and to manipulate the texture/image such that it will change the parameters of the cellular automata, in the event of damage (perhaps the simulation of a laser coming through and destroying all the cells along a ray, using some line drawing algorithm). Then, as part of a realtime demo, I could show these updates, maybe running one update per couple frames, so that you could show off the fact that when undamaged, the electron state propagated through the wire, and if it was damaged it was no longer able to propagate through that wire. I would think to represent the volume as a 3d texture (or image, I'm still not 100% clear on the distinction between the two besides how they are referred to in the shader), applied to the geometry representing a number of slices through that volume.

I will put together something this weekend as a demonstration, based on code from v07, to see whether or not this is a viable scheme, to use multiple slices along each of the axes as a way to represent the shape with a bit less artifacts - I will also look into discarding the fragments on the back of the shape. At minimum, I can drop the fragments which are going to be drawn incorrectly, and add some geometry which will represent slices from the other direction. From talking to people who know a lot more about this than I do, if you are discarding a lot of fragments from a warp, you do actually get some speedup from that - I know in some cases it's a situation where all the shader invocations have to run both branches and then pick at the end which branch was correct, but if they're all dispatched as one draw call, I think you'd be getting rid of half the fragments in any case, unless you're directly side on to the slice, but even then it's going to make a call, as far as setting gl_frontfacing to true or false for the fragments on either side. I think it's a viable approach.