Slicing Algorithm

To begin the discussion on the slicing algorithm, we introduce a sample 3D model used during development. Much of the discussion will revolve around this test model. A relatively complex shape was selected for testing as it was thought most likely to reveal flaws in the algorithm as it was developed. The shape and model asset were discovered during an Internet search for interesting shapes suitable for demonstrating the advantages of 3D printing versus traditional production methods. The model comes from George Hart's rapid prototyping web page at http://www.georgehart.com and is informally known as a "rhomball".

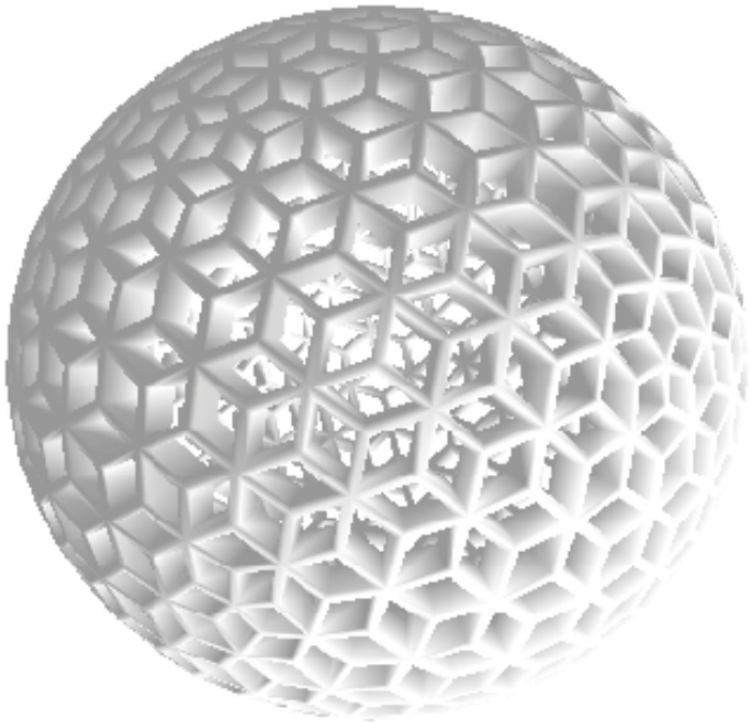

Figure 1 - Rendering of the Rhomball

The rhomball image in Figure 1 was sourced from a screenshot made during development of the model-loading module.

Normal production of a 3D scene involves placing 3D objects in "world" space and defining a viewing volume as illustrated in Figure 2. Objects (or portions of objects) that reside within the viewing volume are rendered; anything outside is omitted from the scene.

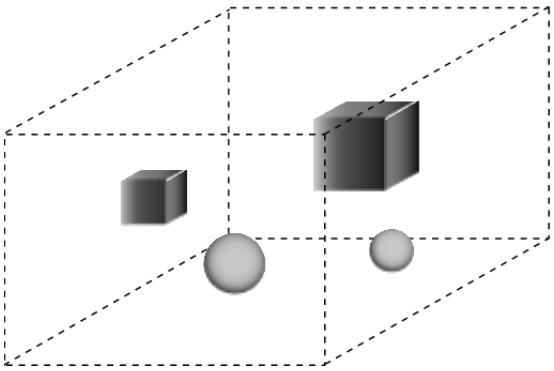

Figure 2 - Objects in a Viewing Volume

It was theorized that if the viewing area was reduced to the depth of a single print slice – say 0.1mm – passing an object through that viewing area would produce the series of slices we were after for printing. This would free us from the complex geometric calculations involved in vertex processing by letting the GPU do it all for us – extremely quickly. The concept is shown in Figure 3.

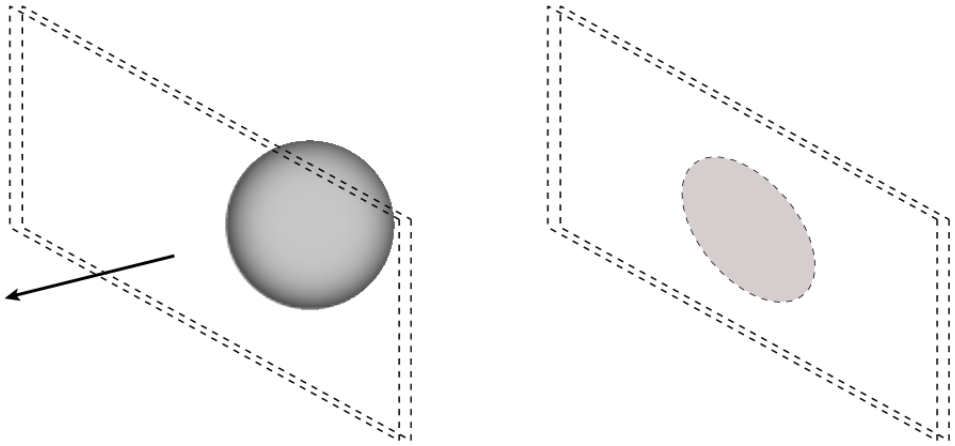

Figure 3 - Passing an Object through a Narrow Viewing Area Should Produce Slices

This idea was relatively straightforward to implement once we knew how. The first attempt, however, was not successful. Significant distortion of the image occurred as it moved through the viewing area. This distortion was due to the manner in which the image was projected onto the viewing plane.

Perspective Projection

Prior to rendering, the 3D space needs to be projected onto a 2D viewing plane – the screen – on which the scene is to be viewed. Perspective projection is normally used, giving the illusion of depth to the viewer. The concept of perspective projection is illustrated in Figure 4.

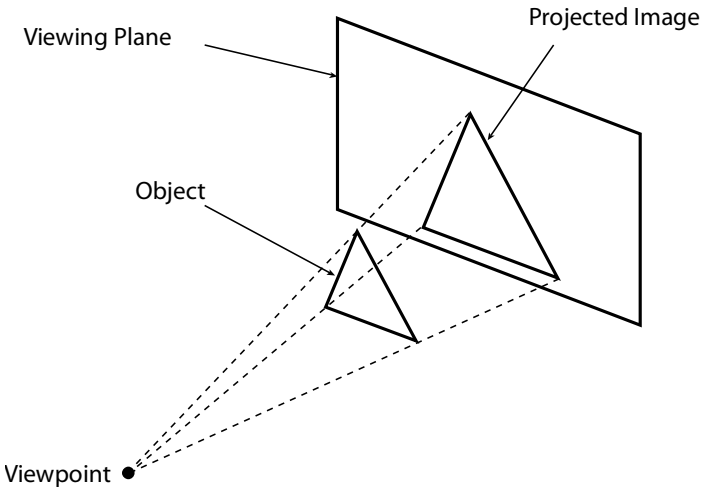

Figure 4 - Perspective Projection

As can be seen in Figure 4, the projected image of the object is distorted based on the position of the object in space, the position of the viewing plane, and the position of the viewpoint. Figure 4 shows that the projected image appears larger than the object under view. This was the cause of the distortion seen in the first attempt at slicing.

Fortunately, Direct3D supports a few different projection schemes; orthographic projection was the scheme we were looking for.

Orthographic Projection

Figure 4 also shows that in perspective projection, each point on the viewing plane converges to the viewpoint. In orthographic projection, each point on an object is projected on a vector perpendicular to the viewing plane, as in Figure 5.

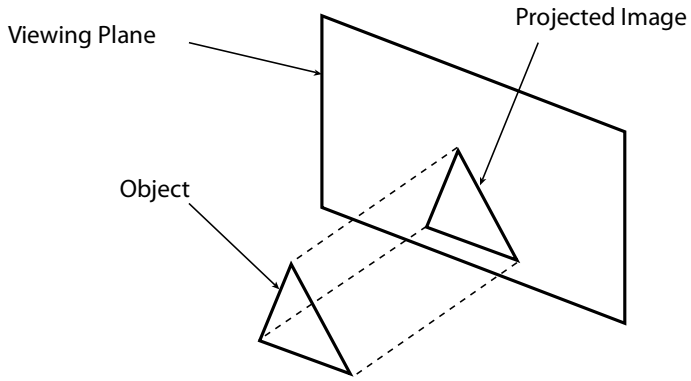

Figure 5 - Orthographic Projection

Orthographic projection produces no distortion in the viewing plane. A normal scene rendered using this projection mechanism looks extremely unusual to a viewer as it does not coincide with how humans see the real world. It is useful, however, in CAD applications where it is important to maintain visibility of parallel relationships between structural components. For our slicing application, it was also exactly what was needed.

Slicing the Rhomball

Once orthographic projection was set up, the rhomball was loaded for slicing. Figure 6 shows one of the slices. This particular slice was selected as an example because it nicely shows that there is a serious problem.

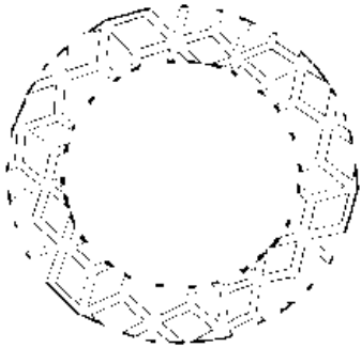

Figure 6 - One Slice of the Rhomball

There are two issues in Figure 6. First: it is hollow. Second: it is not a fully closed solid. This problem did not make itself apparent until many days later in development, during the time we were working on discerning the correct dimensions of the viewing area – or slice thickness. As the slice thickness went down, vertices not parallel to the viewing plane began to appear as small dots and lines; a solid outline of the object was no longer being rendered. This can clearly be seen in the figure.

We now fully understood the problem image slicing posed and it was clear why the 3rd party slicing packages had difficulty working at this resolution. The problem is not trivial.

The issue is that 3D models are represented by triangle meshes, and the surfaces applied to them by graphics software have no real thickness. All models are hollow. To create slices suitable for printing, we needed to fill them in somehow.

Figure 7 - Hollow Slices Are Not Sufficient

More thought revealed a sure-fire solution that would absolutely work if only we could figure out how to manipulate frame buffers in Direct3D, which turned out to be a bit of a struggle. Several days later, we had gained a broader knowledge of the API and were able to start exploring our perfect solution that was guaranteed to work.

Filling the Hollow Slices

During the slicing process, an object starts off behind the viewing area. For each slice, it moves forward through the viewing volume one slice-depth at a time. As it enters, we first see the front-facing facets of

the shape. Moving further, we enter the internal hollow volume of the object. As the object continues through the viewing area, we will then see the back-facing facets[1] as we transition from the internal volume back into outer space.

The solution was to track when we encountered a front-facing outer facet at each pixel in the view plane. That pixel would be kept on until we reached the back-facing facet, at which point we know we are exiting the internal volume of the object. Keeping a pixel on across slices is easy as it merely involves performing a logical OR on the frame buffer contents for each progressive slice. Turning it off is also easy – a logical XOR.

Figure 8 and Figure 9 illustrate the concept of activating and deactivating pixels while entering and exiting the model volume, respectively. Implementation of the algorithm was completed and resulted in everything we needed, plus yet another show-stopping issue.

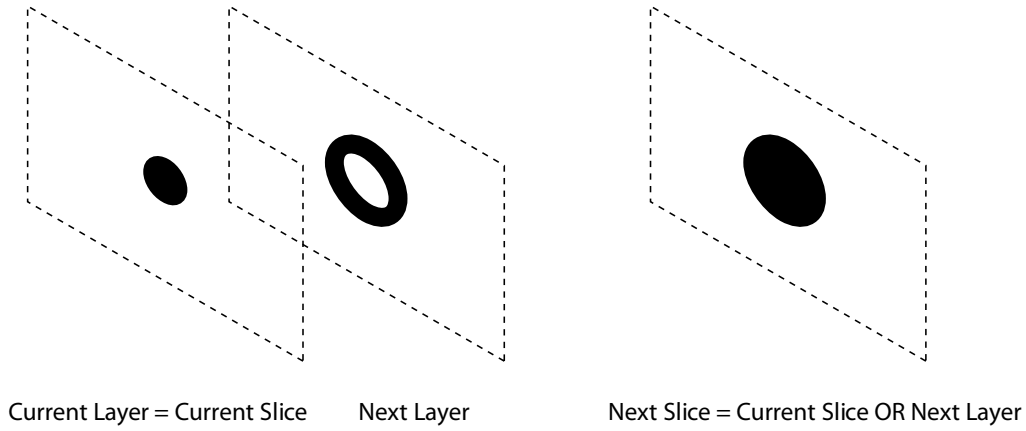

Figure 8 - Slice Production - Entering a Solid Progressively Adds Material to the Slice

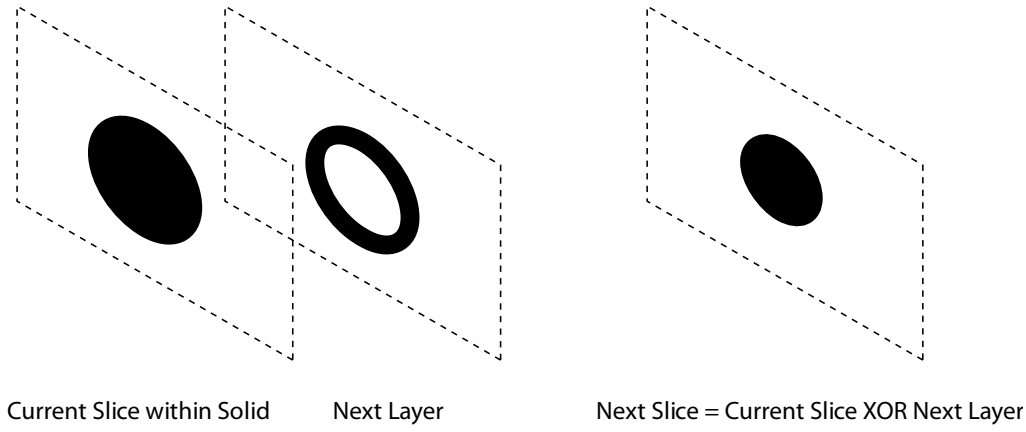

Figure 9 - Slice Production - Exiting a Solid Progressively Removes Material from the Slice

[1] Provided Back-Face Culling is disabled. Back-Face Culling is a 3D graphics rendering optimization technique in which surfaces on the back side of an object are optimized out of the rendering pipeline, as they would be obscured by the front-facing surfaces anyway. For our slicing algorithm, this optimization is undesirable.

Noise

Figure 10 shows a test of the modified slicing algorithm with the rhomball. The filling algorithm worked as anticipated, except a massive amount of noise accumulated as the slicer moved through the object.

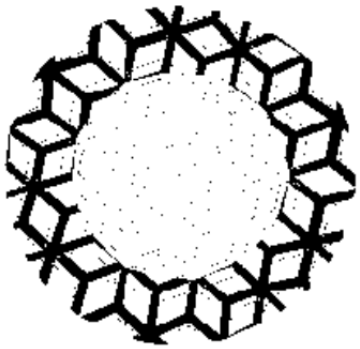

Figure 10 - Solid Good. Noise Bad

This was extremely problematic for the print process, as the noise would cause specks of the resin to harden in areas where it should not, perhaps accumulating to the point where the printer was no longer capable of forming an object. The algorithm at this point was unacceptable.

Why was the noise appearing? It was conjectured that the noise was due to edges of the model – the very edges where some vertices meet have no internal volume. For our algorithm, this would cause a pixel to be turned on when it was first encountered and never be turned off. These pixels did not represent the front or the back of the solid – they were just points on an edge. Moving through the solid, more of these edges were encountered and by the time the last few slices of the model were being rendered, an enormous amount of pollution existed.

A Solution to Noise

A few ideas were entertained. Development of an edge-detection algorithm would enable handling the special case of edges. Slicing the object from different directions and intersecting the results may also be effective. Each solution would require additional processing time. XORing the frame buffers had already produced a significant performance hit where it now took about half a second to produce a slice. The printing process required that slices be produced about every 3 seconds, as that was the estimated cure time for each layer of resin. The performance hit was due to the transfer of each frame out of video memory to system memory, where it was operated upon, and then back to video memory where it was displayed. Some reading about Direct3D and GPUs revealed that there is now a programming language – High Level Shading Language – for GPUs that programs can use to manipulate pixel and vertex data directly on the video card where video memory is in close proximity. This seemed like the best place to explore solutions. Using this approach potentially offered a 100 times performance boost but we were already pushing the limits of development time for this problem. Devoting resources to learning a new programming language and paradigm did not seem to be a prudent endeavor at the time. Although it was believed that moving some of the slicing logic to the GPU and experimenting with different solutions in HLSL would eventually lead to an effective and elegant solution, it was decided to take a different path – one that could get us from noisy to “good enough” slices in less time.

Denoising

Rather than addressing the root of the problem – noise creation – focus was shifted to filtering the noise created. In most cases, the noise was characterized by individual pixels accumulating across slices. Individual pixels should be easy to detect and remove.

First Attempt

The first attempt at denoising involved the inspection of each pixel in the frame buffer and its surrounding area. With the exception of the screen edges, each pixel has eight adjacent pixels. If a pixel was on, but surrounded by no other pixels, it was assumed to be noise. Moreover, if a block of nine pixels had a total pixel count less than a certain threshold – say three pixels – we could also assume that the area contains noise. If either of these cases were true, the center pixel of the block was turned off.

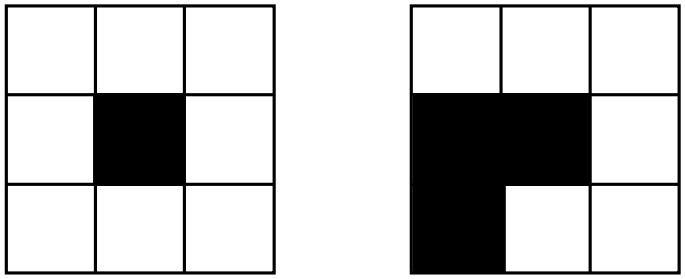

Figure 11 - Identifying Noise in a 3x3 Pixel Grid

This approach had mixed results. Indeed, some of the noise was removed from each slice, but not enough. The application demanded that noise be almost completely absent from the slices. The investigation of each pixel in the frame buffer also brought another performance hit; it now required just shy of one second of processing per slice, which was pushing the limits but still within acceptable range. This was definitely a process that would have benefited from an HLSL implementation.
While not quite sufficient, the imperfect solution to the noise issue still showed some promise, and it was decided to move slightly further in the same direction.

Second Attempt

As examining areas adjacent to each pixel was somewhat effective, we decided to broaden the search. This time, the basic approach remained the same, but instead of looking at a block of nine pixels, we upped to the search area to twenty-five.

Through trial and error, a frequency threshold of nine pixels was identified as the sweet spot. That is, if a twenty-five pixel block contained less than ten active pixels, the center pixel was assumed to be noise and was turned off.

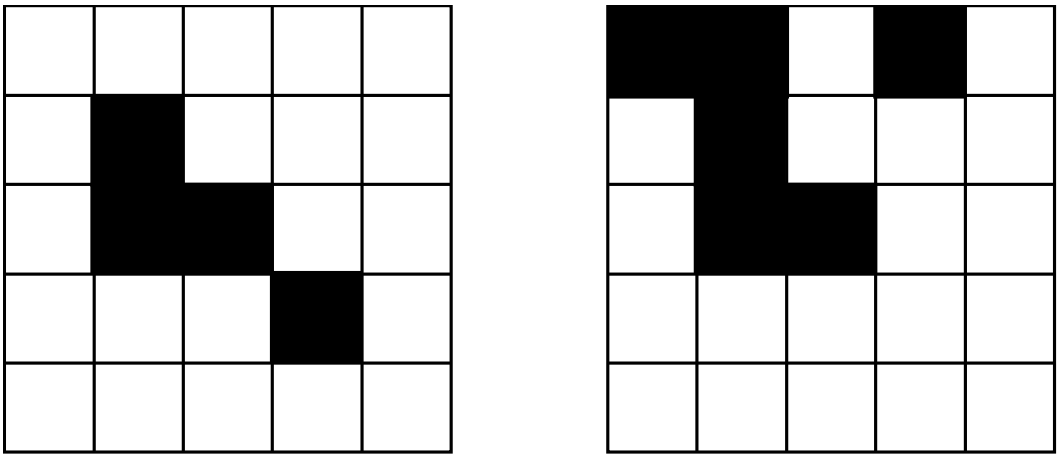

Figure 12 - Identifying Noise in a 5x5 Pixel Grid

Figure 13 shows a typical slice produced by the algorithm at this stage of development. While not an elegant solution, it produced a practical result that would work for our application. The slices were not perfect – some showed an extra pixel here or there protruding from an edge – but the overall quality of the print job would likely not be affected. Expanding the grid size was met with another performance hit; now it was just over one second per slice, but that was still within acceptable range.

Figure 13 - A Workable Slice

Figure 14 and Figure 15 show a comparison of the original rhomball model and the final printed object. The slicing algorithm is clearly up to the task. Close inspection of Figure 15 reveals the individual slices stacked upon one another, an interesting indication of the process by which the object was created. The printed object consists of 768 slices, each about 0.088 millimeters thick.

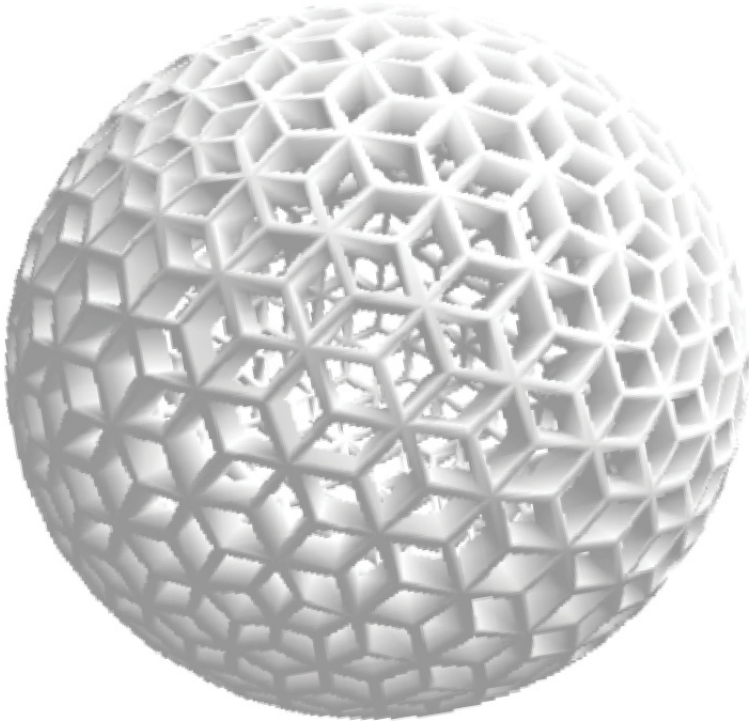

Figure 14 - Original Model of the Rhomball

Figure 15 – Actual Printed Rhomball