This document details the various things I have tried and the ideas behind them. It's more of a blow-by-blow record than the `overview` document. That document is meant as a high-level look at various ideas that I have considered.

1 First Prototype

This was written in November & December of 2008. It takes simple G-code and parses it to pulses, then displays a 3D representation of the resulting object.

The G-code is limited to `G00` and `G01`, along with a few other codes that are unavoidable, like `F` for feed rate, `S` for spindle RPM, *etc.* The G-code is converted to an octree representation of the volume, then the surface of this volume is converted to a set of bounding squares. These squares are put into a BSP tree, and displayed.

The conversion to an octree is extremely inefficient. For one thing, there is no sorting of tool positions when testing for cube containment. The BSP tree is generated with equally inefficient algorithms, and the drawing is not very clean either.

The program works, but it can only handle tiny objects of perhaps 3/8 of an inch on a side and it only knows about end-mills.

2 January 1, 2009

Write a program based on the quadtree with depth (QTD) that works only with simple end-mills. A QTD could be based on a genuine quadtree, but an even faster method is to use a 2D array of shorts, where each short is the depth (or height). This takes a lot of memory, but it's managable for objects of modest size and it should be super fast. In fact, that's one reason to do this. This will serve as a benchmark and target for other methods. It's hard to imagine a faster framework.

1. Don't try to improve the G-code interpreter yet. That's a relatively easy thing, though it could be time consuming, and I should focus on the much harder problem of rendering the images. The very limited amount of G-code that the interpreter can handle now is sufficient to thoroughly exercise any graphics routines.

2. Implement a depth array (DA) instead of a true QTD for cutting with an end-mill. I estimate that this should require about 9 operations for each voxel along the perimeter of the tool position. This is 2 ops for the loop over the perimeter (loop increment and test), plus 2 ops to retreive the offsets that specify which array elements to set, plus 4 ops to find the correct array element, the 1 op to make the assigment. For a 0.256 diameter cutter, each tool position should require that the program examine 2^8 of these perimeter positions, so that the total number of operations per tool position is 9×2^8, which is a bit more than 2^{11}.

3. Consider how far ahead and behind in the pulse train to look. For instance, if the next pulse is an increase in z, then the current tool position can be skipped. By looking one pulse behind, we only need to look at half of the ring of points around the perimeter of the tool. Consider the way that a circle is discretized. By looking two or moves behind and ahead it may be that fewer elements of the DA must be set with each change in tool position.

4. Get some good timing information for conversion from pulses to the DA.

   Answer: Even with the crappiest method of filling the depths of circles possible, it's still fast. For a tool of radius 10, and filling circles to depth by checking the distance of every pixel from the center of the tool, the time taken to run `!longtest`, which has more than 3,000 lines of code and 157,898 tool path elements, was almost instant. It took 579 milliseconds, which isn't instant, but it felt like it. When I increased the tool radius to 50 (in `Cut2D.radius`), it took 11,485 milliseconds. That makes sense. Increasing the radius by a factor of 5 increases the number of pixels that must be considered for each tool position by a factor of 25, and $25 \times 579 = 14,475$.

   Incidentally, the time taken to parse the entire file and generate all of the pulses as a single array was only about 100 milliseconds (it varies).

   The speed of even this cruddy method of marking the depth of cut indicates that working with ballmills, drills and any other reasonable tool that does not permit undercutting should be possible.

5. Try doing this with a genuine Java 2D array (i.e., `short[][]`), then try doing it with a 1D array. For a 1D array, the program will have to work out offsets to the individual entries (by multiplying by the line length). I want to see which is faster: multiplying myself or allowing Java to unravel the pointers.

   Answer: it turns out that a large 1D array is faster than a 2D array. It's only about 10%, but it's a definite improvement.

6. Initially, display the DA as a shaded 2D object. That is, for each entry of the array, shade it based on the depth. This avoids the need for tricky 3D graphics when testing the code that creates the DA. There will be some issues with how to shade these points; the shades must be based on the range of depths that appear in the DA.

7. Consider implementing code that converts the DA to a set of polygons. To do this, start at the corner of the DA. Look at the adjacent array elements. This gives the slope of a polygon to a first approximation. Now step a bit further away from the corner and consider these array elements. As the program moves away from the starting point, the range of permitted slopes for the polygon becomes tighter and tighter. If there is a global maximum permitted error for the polygon (like one or two units), then

we should be able to determine fairly quickly whether the polygon can be extended outward. It probably makes sense to restrict the permitted polygons to some combination of squares, triangles and rectangles, where the triangles and squares have a limited aspect ratio. I must be careful not to paint myself into a corner when seeking these polygons. One way to avoid this is to proceed along one entire edge of the DA so that the set of polygons marches across the surface in a wide line.

8. If I can create the polygons in this way, then the BSP (and rendering) code must be adjusted to allow arbitrary reference planes. As a first cut, I could implement the conversion to surface polygons in such a way that the polygons are all squares which are orthogonal to the coordinate axes. Then I could use my existing BSP and rendering code.

9. Investigate graphics cards. Do generic computers these days (like a cheap Dell) have graphics cards? If not, then it makes sense to postpone implementing anything the relies on the presence of a graphics card.

10. Try extending the DA to work with ball-mills, and perhaps to other tools.

11. If I can get this to work for end-mills and ball-mills, and to parse more general G-code, then this is releasable as a beta. It might be worth doing to see if there's interest.

12. For a depth array (DA), as a way of getting an interactive 3D image, it may be fastest (and simplest) to scan every pixel of the output window, and determine which 1x1 column that a ray from this pixel intersects. This is something like ray-tracing; in fact, it's the same basic idea that I use throughout the astronomy book. The alternative is something based on polygons. My suspicion is that polygons will be faster, but that's more complicated.

    As it turns out, this isn't such a good idea; it's slow. For each pixel, there is a line from the eye. All of these lines are parallel, but that doesn't help. The problem is that for each line, we need to walk through the (x, y)-coordinates (in 3-space) traversed by the line, and compare the z-coordinate of the line at each of these points to the height of the workpiece at that (x, y). The first time that the height of the workpiece is greater than the z-coordinate of the line is where the line intersects the workpiece. So, I need to do a Bresenham for the line associated with each pixel, and do some arthimetic at each step of the line. There are ways that I could speed this up, but there's no way to avoid thousands (millions?) of operations per pixel.

13. Another idea about how to get a 3D display is what could be called a "smart copybits." Another way to look at this is as a wireframe, but where the wires consist of all of the rows and columns of the DA. There are several ways to do this. The simplest is to take every n-th row and column of the DA and copy it to the display at the correct (x, y) coordinates on the

display. As we proceed along a scanline of the DA, we get a varying curve on the display. This alone has two problems: (1) it is only a wireframe; if there are gaps from one line to the next, then those gaps will remain at the background color; and (2) hidden line/surface removal.

The obvious (and probably quite effective) way to handle hidden line removal is with a z-buffer. Create two $m \times m$ arrays, one for the resulting `BufferedImage` and the other for the depth of the most recently drawn pixel. If a potentially new pixel isn't on top of what was drawn earlier, then don't draw the new pixel.

There are a couple of ways to handle the gaps. One way is to assume that the gap consists of a surface from the upper edge to the lower edge. Another way is to be somewhat adaptive in choosing the scanlines of the DA. In places where there is a gap between two scanlines, go ahead and look at the intermediate scanlines. This won't fill in every gap, but it could improve the appearance. Another way is to simply choose every n-th row and column, and treat this grid of samples (roughly) like the corners of polygons.

This brings up the issue of shading. The shade of each pixel/polygon could be determined on the fly or be precalculated. Precalculating the surface normals would take a lot of memory – at least one byte per DA entry, and that assumes that we somehow restrict the permited angles of the surface normals. Another way is to assume that the surface normals do not vary very much, and calculate these normals over a grid. This reduces the number of samples.

I think that this basic idea can be the basis for a fast rendering method. The result may not be as pretty as some others, but it should be quite fast.

Answer: This works very well. I think that this could be used with the idea of cross-sectional slices too, when I get around to allowing undercutting and multi-axis machines.

One tweak that this needs is a way to draw attention to potentially ambiguous regions of the rendering. For instance, I could note all local extrema of the surface, and make sure that a denser set of samples in taken in those areas. This will improve things in two ways. First, if there are small bumps that might peek up above the horizon of the surface in the foreground, then those bumps are guaranteed to appear. Second, the rendering will be more detailed near edges and sudden transitions. Note that by "local extrema", I don't just mean extrema relative to the z-coordinate; I mean any kind of "pointiness."

2.1 CopyBits

It's not called copybits in Java (copybits is Macintosh lingo), but the question is the same: how to copy a raster from one rectangle to another. It

takes surprisingly long. For one example, based on a 3200 by 6200 raster, it takes about 3900 milliseconds for the single line of code that does the copybits (`Graphics.drawImage()` is the Java method).

I'm surprised how long this takes. Here's what I did (roughly):

```
DataBufferUShort db = new DataBufferUShort(theCut.surface,theCut.surface.length);
WritableRaster rast = Raster.createPackedRaster(db,width,height,16,null);

// Create color model, consisting of arrays of red, green and blue levels,
// 65536 of each.

IndexColorModel icm = new IndexColorModel(16,65536,red,green,blue);
BufferedImage image = new BufferedImage(icm,rast,false,null);
g.drawImage(image,0,0,destWidth,destHeight,0,0,width,height,Color.white,null);
```

It's the `g.drawImage()` line that takes so long. What I've done is very efficient, memory-wise, since it uses the 1D array that's already stored in the Cut2D object, but it's slow. I'm not sure why, but here's a guess. First, drawImage() may be more clever than it needs to be. The source rectangle is larger than the destination and it may try averaging over pixels of the src to create the dest. There may also be more overhead than I though in converting a raster of 2 byte shorts to the destination, which is probably a 4 byte RGB value. Each pixel may require a table lookup.

When I implemented by own copy bits by first allocating an 8 bit per pixel buffer that's equal in size to the destination rectangle, and copying (and translating) the bits that I want out of Cut2D, I found that the conversion takes less than 50 milliseconds. That's a ratio of more than 78 to 1. Some of this is because I am skipping scanlines of the source, and we only take about 1 line in 12 from the source. Since we only take about 1 column out of 24, this could account for it, but there's no way to know.

Very Odd. The `Cut3DPane` class uses a two step process to render a DA. The DA is scanned to create a offscreen pixmap based on 8 bits per pixel. Then this double buffer is copied to the display window with `Graphics.drawImage()`. What surprises me is that this takes much longer than generating the buffer. In one example, blowing up the window to the entire screen, generating the offscreen pixmap takes only 31 milliseconds, but `Graphics.drawImage()` takes 234 milliseconds. I'm not sure why, and these number are based on code without a z-buffer or a lot of other features, but it's a huge difference. One guess is that if the buffer uses a different number of bits per pixel than the screen (say the screen is RGB), then translation takes a lot of time. It could be that calling `Graphics.drawIamge()` is not the most efficient thing to call in Java. I need to get my hands on the data underneath the window. Is there a way to do that in Java?

3 January 10, 2009

The ideas above work. If I want to make this into something useful, then I need to change from tinkering around to building something more stable, permanent

and reusable. Here's a to-do list.

1. Polygon filling code. I need this to use in conjunction with a z-buffer.

2. Bresenham type code.

3. Here's a way to render the object. I've already implemented code that connects the dots from one sample of the DA to the next, as the code moves along lines of constant y. This gives a kind of wireframe rendering. The problem with this is that when zoomed in there are gaps from one line to the next. One way to fill these gaps is to treat the space between these lines as polygons. As we move down one line, move down the line immediately above at the same time. That is, consider four points (two from each line) at once, and consider these four points to define a quadrilateral. They don't in fact define a quadrilateral (or they need not) since the four points may not be co-planar, but it should be OK. Let these four points be a, b, c and d, where a, c are on one line of the DA and c, d are on the other line, with a above c and b above d. Although these four points may be be co-planer, their projection certainly does form a quadrilateral (or perhaps a triangle?). This should look OK. Shade it based on the normal defined by the cross-product of the vectors pointing from one vertex to the opposite vertex. If I've overlooked something and this doesn't work, then triangles could be used, and there would be no approximation if we did this. First draw the triangle defined by a, b, c, then draw the triangle defined by b, c, d.

   The idea just outlined would probably work, but it's more complicated (hence slower) than it needs to be. A modest addition to what is already done should do just as well, or better. As the wireframe-ish rendering is made, just as it is now, store the values drawn to the output buffer due to the previous line. This should include all the values, whether they end up being filtered out by the z-buffer or not. As we proceed along the new line, drawing lines from one sampled point to the next, the program should also connect to the previous line along the direction perpendicular to travel. The program will have drawn lines from a to b and from c to d (their projections, actually). Now, for each point of the buffer that was drawn to connect c to d, draw another line that goes from that point to a point along the line from a to b. Let the shading be given by a linear transition from the shade of the point along segment ab to the shade of the destination point between c and d.

Some observations.

1. I had a bit of code with the statement

   ```
   Graphics gr = image.getGraphics()
   ```

   inside a loop so that it was called a lot. One would think that the cost of this call is small, but it's not. The drawing code with this statement inside the loop took about 200ms to run; by moving this line out of the

loop, and leaving everything else unchanged, the time required dropped to 90ms, or about half. There were a fair amount of other things done in the loop so that the call must amount to some 50 or 100 operations.

2. This kind of slowness occurs elsewhere. The code that renders the image involves drawing many short line segments. I compared my own implementation of Bresenham's algorithm to calling `Graphics.drawLine()`, and my routine is much faster. With my routine, each frame took 30 to 50 milliseconds, while the Java call took 60 to 80 milliseconds. Considering how much other stuff happens in the rendering loop (rotation, projection, etc.), this is a big difference and my implementation of Bresenham's algorithm is not even very good.

4 January 24, 2009

Here are two thoughts related to the ultimate program for multi-axis machines and arbitrary tools. They assume that some form of slice representation will be used instead of octrees or something else.

First, I have been thinking in terms of letting the slices represent the perimeter of the postive material at all times. The usefulness of this idea is more apparent if you think of the slice as being represented with a quadtree. The essence of the idea is that taking unions of quadtrees is faster than intersections or differences – at least I think that's true. When a tool cuts, adjusting the quadtree to reflect removed material is somewhat harder than adding the removed area to a quadtree that represents all of the material that has been removed so far. Small quadtrees for each slice of each tool could be stored and it might be faster to add these quadtrees than to subtract them. I'm not so sure that this is such a useful idea, but it's worth thinking about.

Next, there are difficulties with undercutting and odd shapes. Each slice is just that: a slice and nothing more. It will not always be obvious how these slices touch. If we think of the slices as true slices with non-zero thickness, then it is clear how the slices touch one another, but rendering these slices may still be tricky.

I am making progress rendering, but there are still some issues. Currently, I render a crude image by traversing each line of the DA and drawing it to the screen. The question is what to do with the space between the lines. I see a couple of choices. Right now, the rendering is based on a grid of samples at values of (x_i, y_i), and I draw lines connecting along constant y values. One way to fill in the space between these constant y values is to treat the image as a set of square patches (where I use "square" as short-hand since the actual shape merely has four corners and may not be a square at all). There is a shade at each of the four corners and we can fill the interior with something like Gouraud shading.

At first glance, it appears that a much simpler method is to simply repeat each line until the space is filled. If there is a gap between the previous line and

the current line, then repeat the previous line until the gap is filled. This might work for relatively tame surfaces, but where large peaks and valleys occur, it is bound to show weird behavior. A similar strategy is to generated interpolated slices where they are needed, and that might work, but, conceptually, it's not much simpler than Gouraud shading, and it's computationally more demanding.

5 January 25, 2009

I implemented Gouraud shading. The surface is broken up into strips, each of which consists of a series of quadrilaterals, and these quadrilaterals are drawn with Gouraud shading. This works, but it's too slow. I need to speed up by a factor of at least 10, and a factor of 50 would be plenty fast.

Visually, it seems to be fine, although it is not theoretically correct. The program takes four corners from the DA, rotates them and projects them to the plane, then considers this projection to be a quadrilateral. The problem with this is that the four points (in 3-space) need not be in the same plane. To be correct, I should handle this as two triangles. In practice, it seems to look fine, although I wonder whether two triangles might be faster to shade than one quadrilateral.

I think that the slowness can be fixed. There are several things I can do. Most obviously, the algorithm used to shade the quadrilateral is horrible. I could also be smarter about which rows and columns of the DA are considered – some of them may not project to the display at all. I should be smarter about the frequency of sampling too. In fact, until the user zooms in very closely, I might be able to use the original connect-the-dots method and not consider polygons at all. Provided that the lines are close together, this will be fine.

In fact, this kind of check is almost certainly something that I should do. After building up the two sets of (x, y) coordinates for each strip of the DA, make these checks. If the four points are adjacent or coincident, then just set those points. If they differ by only 1 pixel in one of the two , then it suffices to draw these two lines. Only if these tests fail should the program draw a polygon. In fact, I could probably test to see if there is only a single pixel in the interior of the polygon and handle that as a special case too. Obviously these tests take additional time. Whether to perform any given test depends on how often we end up in each case.

The shading is currently a crude cheat. Pixels are shaded by their depth in the object, rather than according to the surface normal. This often looks fine, but I can see that it won't always work.

Because I am skating so close to the edge of what is computationally feasible, I may want to allow the user considerable control over the methods used to render.

I am pretty much on the home stretch now. Before starting the final implementation, I need to consider where I am going with this. Which features should be in the first beta version, and which features are expected to be in the first released version? Which features am I aiming to include in the ultimate killer

app? At the moment, I think that the first released version should be based on DAs, pretty much as I am doing now. I need to add more tools and a better user interface, but it would be better to release something that has obvious short-comings (no undercutting, small workpieces and a limited tool selection) than something that seems like it should do more, but doesn't. Moreover, by tweaking what I have now and making it as efficient as possible, I will have a better idea of what is attainable down the road.

5.1 Speed Issues

I wanted to see whether the use of Java's `ArrayList` classes was slowing things down at all, and the results of my test are odd. The method `Fund.polyShade()` does Gouraud shading with depth and a z-buffer. The original implementation used four `ArrayList` objects, allocated inside the method. The method calls the `line()` method to make a list of perimeter pixels, and associated depth and shade values, then sorts these lists, then removes redundant information, and finally does a scanline fill. For a particular example, this took 375-425 milliseconds per frame.

For comparison, I first replaced the four `ArrayList` objects with four ordinary Java arrays. Because of the way that the redundant information in the arrays is removed, it was easiest to allocate a second set of four arrays and copy the non-redundant information into these arrays. So, I eliminated the four `ArrayList` objects, along with their allocation, but I replaced it with two allocation of four 1D arrays. I found that the program took about 1,200 milliseconds per frame, which means that the new method took three times as long.

I then removed the array allocations from the method and made them static arrays of the class, and changed nothing else about the method. It then took only 250-300 milliseconds per frame, which is meaningfully faster than the original `ArrayList`-based code. This means that allocating the arrays take a lot (!) of time. I'm not sure if this is due to the actual allocation or due to the fact that Java zeros the array before I get my hands on it.

As an additional test, I changed the original method (that uses `ArrayList`) so that the `ArrayList` objects are allocated as static fields of the class. It's no faster – if anything, it's a bit slower, probably because I had to replace the `new` with an `ArrayList.clear()` at the start of the method.

The bottom line is that I should avoid `ArrayList`, and the range of numbers above doesn't give the full difference. The real speed improvement is more like 275 milliseconds versus 400 milliseconds. I'm not sure exactly why the difference is so great. Some is probably due to the way that `ArrayList` itself works, but I'll bet that a lot of it has to do with converting to/from `int` and `Integer` with every access of `ArrayList`.

6 May 18, 2009 – Version `01` Freeze

I took a long hiatus. To get back into this, I need to rethink where I am going. I moved most of the code to the `v02` directory. Going forward, I will use `v02`. I only moved what is necessary to pick things up more or less where I left off. There's a lot of code for octrees, binary space partitions, quadtrees, etc., in the `graphics` package of `v01`. Look there if I ever want to go back to any of these ideas.

There are lots of ways to store the 3D representation of the object. The best one seems to be some type of slicing method. I haven't re-examined the other methods in great detail, but that was the conclusion that I came to a few months ago. Looking, with a fresh perspective, at the "overview" document that I made, this really does seem to be the case. The other ideas are more complicated, and they don't seem to have any speed advantages; in fact, they seem slower even if they were well implemented.

There are two slicing methods. The basic idea is to represent each cross-sectional slice (0.001 inch thick) of the object. There are two potential methods. The first is based on walking around the perimeter (or perimeters if the slice is made of disjoint parts). The other is based on scanlines. It appears that either method could be made to work in terms of the memory required. The bigger issue is going from G-code to the 3D representation. In fact, even this might not be such a big concern. It should be relatively easy to convert from one representation to the other if one has a big advantage in the G-code to representation step, and the other has a big advantage in the rendering step.

6.1 Slicing Methods: obtaining the representation

First consider the slices given by walking around the perimeter. Each step would be represented by a single byte, whose values depends on whether that step is up, down, left or right (so this really only needs two bits, and run-length encoding could make this even shorter).

One of the disadvantages of this method is that slices which consist of disjoint regions will be trickier since such slices would require two or more perimeters. Another disadvantage is that unions and intersections of these slices would be harder to do. I think that there are known algorithms to do this, but they're non-trivial.

If a tool cuts into a slice, then the perimeter walk must be adjusted accordingly. This seems easy – all you need to do is find the first and last points of the perimeter where the tool touches the slice, and replace the intervening part of the existing perimeter with the path given by the cross-section of the tool. Obviously, you have to find these two points. Another issue is the need to test whether removing this part of the slice serves to cut the perimeter into two or more disjoint pieces.

Don't forget that the tool might be an oddball shape, and that it might cut at a weird angle. This doesn't complicate the basic idea, but it makes it clear how tricky it could be to find the portions of the exisiting perimeter that need to

be replaced. In fact, there might be two or more disjoint parts of the perimeter that need to be replaced – imagine an ogee bit that just touches along the edge of the material and cuts two parallel groves.

Now consider the slices given by scanlines. Each slice is made up of the scanlines of the slice. There are different ways to represent these line segments. One way is as a list of shorts. The first short is the position where the slice first becomes solid, then comes the length of solidness, then the next position where it becomes solid, etc.

The basic idea of dealing with cuts is the same as with the perimeter idea. The tool cuts into a certain set of scanlines, and each of these scanlines must be adjusted. It should be easier to figure out which scanlines are cut, but I think that it would be more time consuming to make the adjustments since every scanline touched by the tool must be considered, and there will often be hundreds of these – maybe more than 1,000. Another advantage of scanlines is the fact that holes would be easier to represent. A perimeter does have a natural direction of walk, and that can be used to define inside versus outside, but the idea of a hole is represented more directly with scanlines.

So, to see which of these two methods is better for the cutting part of the job requires looking in more detail at how it would be done. By looking one tool step forward in time, I think that we can assume that only one intersection of the tool outline with a single slice will be needed for each tool position. There are exceptions or odd cases to this observation, like plunge cuts with a concave bit and cuts at certain angles to the slices, but it will be true for the most common situation of an endmill moving in the horitontal plane (although if the endmill moves in a direction nearly parallel to the plane of the slices, then it will be less true).

The steps required for tool intersection with one slice under the perimeter representation are

1. Find the first and last points where the tool outline intersects the slice. As noted, it is possible that there will be several disjoint areas of intersection; whether this is the case must always be checked, but it should be uncommon.

2. Insert the new path elements into the perimeter of the slice. The cut could also break the perimeter into two disjoint parts. Not only would this require extra work in this step, but once it has occured, the previous step becomes more complicated.

The steps required for tool intersection with one slice under the scanline representation are

1. Determine the maximum depth of the tool. This tells us the number of scanlines that must be considered.

2. Cycle through the scanlines, adjusting each one.

Under either strategy, it may be more efficient to aggregate the cuts into a larger negative volume, then remove that volume by taking the difference with

the slices of postive volume. If that's the case, then the steps above are not correct. Nevertheless, here's how these could be handled in each case, and the number of operations it would take.

The first step under the perimeter representation is the trickiest. Under the worst case, there is no way to avoid traversing every step of the perimeter – we have to know if any bit of the existing positive volume has been cut. If we build up the negative cut-away volume, then it could be faster (because the perimeter of the cut-away volume is shorter) but I don't think that it would be dramatically different, and you would still have the problem of subtracting the negative volume from the positive volume, and that is non-trivial.

One way to speed this up is to represent the perimeter as a polygon. Most shapes will be fairly simple, but others will not be. Using the coordinates of the vertices of the polygon will work only for these simple shapes – there are too many vertices for complex shapes. So we need some kind of hybrid of polygon plus perimeter steps. If we use an entire byte for each step, then there are lots (254) of escape codes available. I was already thinking of using these for run-length encoding, and it's only a short additional step to add the ability to encode polygons. These could be given either as

escape code, x-coord, ycoord,

or as

escape code, rise,run,distance,

or maybe some other scheme. However this is done, we can determine every point of intersection of the perimeter with the tool volume with many fewer steps along the perimeter. For most typical shapes that consist primarily of polygons, we only need to visit each vertex and we can use analytic methods to see whether the lines connecting the vertices intersect the tool.

As a rough (!) estimate, suppose that there are $2^5 = 32$ vertices. Most of the time I think there would be only 6 or 8. Determining whether each line intersects the tool would take (guess!) $2^6 = 64$ operations, for a total of 2^{11} operations.

Once the points of intersection have been found, then the perimeter must be adjusted. This should take very few operations, *except for the fact* that it requires memory allocation. Leaving aside memory allocation, the number of operations should be proportional to the length of the new perimeter (or the number of polygon vertices) which must be copied. Say there are still 32 vertices and that 2 operattions are needed for each one; that's only 2^6 operations, which is much less thant the 2^{11} required above. So let's say that 2^{12} operations are required for everything except memory allocation. Earlier tests show that memory allocation is relatively costly. I should only have to do this once, but it will often be a large object (a few kilobytes). If this is a bottleneck, then I could build my own memory manager. The idea is that perimeters will rarely become smaller. If I don't return "bad" perimeters to the GC, then I can reuse them in a situation where a shorter perimeter is needed. Wait – in fact, using polygons, the perimeters should not typically be very large objects; they might be only

100 bytes, and probably less. Suppose that there are 32 vertices and that each one takes 8 bytes (five is probably a better estimate). That's only 256 bytes per slice. So it seems less likely that I will need my own memory manager and if I do, then it shouldn't be so hard to design. A tweak that I could make is some kind of "roughness guage" to indicate that using polygons as an approximation is OK provided that the sides of the polygon are within a given amount of the exact values.

Now consider the worst case where the perimeter is so jagged that I can't use polygons. If the object is $8 \times 8 \times 4$ inches thick, then each vertical slice has a perimeter on the order of

$$2 \times 8 \times 1024 + 2 \times 4 \times 1024 = 2^{1+3+10} + 2^{1+2+10} \approx 2^{15}.$$

It will be rare that every step of the perimeter is so jagged that neither polygons nor run-length encoding can be used anywhere, but this is meant to be the worst case. Testing the intersection of each step of the perimeter will take (guess!) about 2^4 operations, so that 2^{19} operations will be required to let the tool cut each slice. In reality, I hope that it will be more like 2^{16} operations once run-length encoding is used and I am smart about taking intersections (for instance, if a given point of the perimeter is 100 units away from the tool, then we don't have to test any nearby steps of the perimeter). In this worst case of a jagged perimeter, making adjustments will be more of a burden in terms of memory allocation as well. Once the memory has been allocated, the data must be copied and that will take on the order of 4 operations per step, or about 2^{17} operations. So, 2^{17} or 2^{18} operations per slice, plus memory allocation costs, is the estimate for the worst case of extreme jaggedness. That's in comparision to 2^{12} operations (plus memory allocation) for the more typical case of polygons.

Now consider the scanline representation. It must be determined which scanlines intersect the tool and each of these scanlines must be examined for possible adjustment. If the tool extends 1 inch below the top of the work, then 1024 scanlines must be considered. Adjusting each scanline will take something like 8 operations, for a total of 2^{13} operations (plus memory allocation). Note that memory allocation will not happen as frequently as it might appear. The length of the list of line lengths needs to be adjusted only when the tool cuts a new groove in the material (new depth-wise). If the tool is just widening an existing groove, then the lengths of the segments are adjusted, but the number of segments stays the same. That's a very large advantage for scanlines. Moreover, these lists will generally be short; it would be rare for there to be more than a 10 entries, and 50 entries is about the maximum that I can imagine. On the other hand, the polygon-type perimeter won't need such frequent memory allocations either; new objects are needed only when the number of vertices changes, not when they are merely shifted. Highly jagged objects are a much different story; for those, the advantage of scanlines seems clear. In many cases, the scanlines will repeat and the cost stated above will be an overestimate. If there's an escape code meaning that "this line is the same as the previous line", then, frequently, it wouldn't be necessary to examine 1024 lines.

Overall, it seems like scanlines are almost as efficient as polygon perimeters, and perhaps just as efficient. Also, I haven't said much about the problem of slices made of disjoint areas. This is a non-issue for scanlines, but it gums things up a bit for perimeters, but not a lot. The larger problem is the amount of memory required for scanlines.

Assuming again a block of $8 \times 8 \times 4$ inches thick, the number of slices needed is $8 \times 1024 = 2^{13}$. The number of scanlines in each slice is $4 \times 1024 = 2^{12}$, so there are a total of 2^{25}, or 32 Meg scanlines. Each scanline requires *at least* 4 bytes, for a minimum of 128 Mbytes. A reasonable worst case scenerio is one where each slice has 10 entries of two bytes each (something like a weird grid pattern could require a lot more though). If 20 bytes per scanline are needed instead of only 4, then you need 5×128 Mbytes, or more than one gigabyte. However, this doesn't make any use of repeating scanlines. In the vast majority of cases, you would have repeating scanlines, and cutting more than 2 inches into the material (so that the billet could be considered only 2 inches thick) is also unusual.

Assume that the scanlines are encoded in such a way that each line requires at least two bytes, even if its value just indicates a repeat, and that each slice has 32 different scanlines, and that each of these scanlines takes 64 bytes (which is pretty extreme). The total memory required per slice is then about

$$2 \times 2^{12} + 32 \times 64 = 2^{14} + 2^{11},$$

which we can just call 2^{14}. With 2^{13} slices of 2^{14} bytes each, the total memory required is 2^{27} bytes or 128 Mbytes. This is just manageable, and it would be less if the scanline repeats were encoded in such a way that the number of times that the line repeats is included.

The memory requirements of scanlines are large, but they should be about as fast as perimeters for the G-code to volume step, and they will be much easier to code. As long as this step occurs in a few seconds (not minutes), then whatever method we choose should be fine. It's the rendering step that's *must* be fast.

Aside: I've been thinking in terms of doing the entire object at maximum resolution right from the start. Unless the user is zoomed in, this is overkill, and when he is zoomed in, he can only see a small part of the object. It might be possible to generate the complete object at a resolution of only 1/64 (say), then generate a more detailed version on the fly if the user zooms in. By using the existing rough model, and focusing only on those parts of the object that are visible, it might be possible to speed up this final fine-tuning step enough to make it part of the rendering process. That's a complicated idea that I would rather not get into unless I must.

6.2 Slicing Methods: rendering

I really can't think of a better idea that what I was doing before I took a long break. The basic idea is to walk along the perimeter and project each point

visited to screen coordinates. There are issues of shading, but the idea given is what drives the process.

One idea that I had about the rendering step is to try to make new slices by intersecting the existing slices with new slices that are at an oblique angle to the original slices, where this oblique angle is chosen so that the new slices can be most simply projected to the screen. If these slices are chosen well, then they can be projected to the plane trivially using parallel projection. The problem is that building these new slices is not easy. Suppose that we have one point on an old slice and we know where it is on the new slice. If we walk to the adjacent old slice along the direction of the plane of the new slices, then this would tell you what the next step is on the new slice. That makes it sound easy. The problem is finding the point along the perimeter of the old slide, and being sure that it's the highest point on that vertical line through the slice. The obvious way to be sure of this is to visit every point of the perimeter. Visiting each element of surface area once (on average) would take a while since, for an $8 \times 8 \times 4$ billet, there are

$$2 \times 8^2 \times 2^{20} + 4 \times 4 \times 8 \times 2^{20} = 2^{27} + 2^{27} = 2^{28}$$

surface elements. I might be able to make this work, and the important/hard check would be faster and easier with polygon perimeters, but it seems logical to first pursue the simpler idea that I already had.

Another idea is to make the drawings iteratively. This doesn't speed up the overall process, but it would make any method feel better to the user. First make the drawing crudely, sampling the data only every 1/64th of an inch, or even every 1/8th of an inch. Then, if the user pauses at that frame, start a thread that will calculate that image in full detail. If the user continues to rotate away from that frame, then kill that thread, but if it finishes while the user is still looking at it, then display it. This is a bit of fooling around, but if speed is an issue, then it's one way to give the user a real-time feel, but still make the full detail available.

Finally, the image should be rendered on something like a light blue background, not black. The point is that if there is a through-hole, then the background color needs to be different than the black used for the bottom of a blind hole.

For the time being forget about these complicated ideas, and just think about the basic idea that I already had. Walk along each perimeter, and project the points of the perimeter to the screen. Use a z-buffer to ensure that the front-most surface is what appears on the screen. If these projections are dense enough, then that's enough. If they are not dense enough, then some kind of shading, like Gouraud shading, will be necessary between samples.

Fortunately, visiting every surface element won't be necessary. When the user is zoomed out, only one slice every 1/64th or 1/128th of an inch (say) needs to be visited. That cuts down on the number of samples by a factor of 2^6, assuming sample every 1/128th of an inch, or sampling every eighth slice and every eighth element of the slice. Since there are (roughly) 2^{28} surface elements, this means that we must visit $2^{28-6} = 2^{22}$ points of the surface.

Here's how to determine the frequency of sampling. Project the corners of the bounding box for the material. From this you can determine the number of slices per pixel (or pixels per slice). It may be best to assume one slice per pixel, or maybe two pixels per slice, but allow the user some control over this. If the image is jagged, then he can increase the number of samples per pixel, and if rendering is slow, then he can reduce the number of samples.

In the case of the perimeter given purely by steps, this really is the number of times that each calculation must be done. As a very crude estimate, each of these points will take something like 2^7 operations, for a total of 2^{29} operations. I can probably speed this up a bit because all of these points are in the same plane, which means that they can be projected more efficiently, but it does seem to be only border-line feasible.

If the perimeter is given as a polygon, then the sampling will be faster because, for one thing, if we project the two endpoints of a side of the polygon, then the projection of the line between them (in 3D) is just the line between the points after they've been projected to 2D. To a first estimate, the number of operations is only the number of slices taken (say one out of 8 as above) times the number of vertices of the polygon. The total number of slices in the surface is (we are assuming) $8 \times 1024 = 2^{13}$. If we only visit one out of eight, then we are visiting only 2^{10} slices. If each slice has $64 = 2^5$ vertices (which is a lot), then the total number of points visited and projected is 2^{15}. Assuming that 2^7 is still a decent estimate of the operations required for each point projected, the total operations required is then 2^{22}, which is quite reasonable.

If the slice is given by scanlines, then the situation is similar to the one for polygons. The first step is to determine the vertices of the polygon that's equivalent to the scanline representation. This isn't trivial, but it's not especially hard. The end of a scanline (where a gap starts) is one vertex. The question is where to find the following vertex/corner. I'm sure that the problem of converting a set of scanlines to a polygon has been well-studied. My instinct is that, although there are probably good algorithms for this, it will take a significant amount of time.

It's looking more like the polygon method is the way to go. The disadvantage of this method is that it's more complicated to implement the G-code to 3D representation step. Memory storage is a huge advantage of polygon perimeters. Basing things on simple geometric shapes also seem intuitively preferable. When unforeseen problems arise, solving them in a world of polygons should be easier than in a world of scanlines.

A final problem with any of the slice methods is what to do about rendering when the observer is facing the slices head-on instead of edge-on. In particular, consider what to do if there is a hole on the side that pases through a slice. Thinking about this situation makes in clear that the "interior" of every slice must be drawn. If we render every slice as a thin slab, then the result will look right and it would solve the problem at hand, but it would take far (!) too long.

One solution to this problem is to build DAs for the two sides. Recall that a DA is a 2D array of depths. In fact, these could be implemented as a series of line-segments – in other words, a part of a polygon just like the polygonal

perimeter. The only difference is that the "polygon" is not closed. There is the question of how to "stitch" this face to the four faces that it touches.

Given that these "trucated polygons" will be needed, their similarity to DAs, and the fact that I was well on the way to a complete DA-based rendered, one way to proceed is to build a simple system based on one flat surface specified by a set of slices made of truncated polygons. For simplicity, assume that only endmills are used. In fact, it may make sense to start with a simple DA (just a 2D array of depths) and work on showing that first. In fact, I think I had already done that, or had been well on the way to getting this to work.

For comparison with the speed of the actual implementation, determine an estimate of how fast I expect it to be and the amount of memory required. For this purpose, it makes sense to think of the speed per square inch. In one square inch, there are 2^{10} slices (they're not really slices since we are thinking purely of a surface).

When using the polygon method, there is a choice to be made: we can assume that polygon vertices are the norm, or that individual steps are the norm. Assume that vertices are the norm; unless the surface is very rough that should be the case, and even when it is rough, I'll bet that the perimeter can be effectively approximated by line segments most of the time. So, each slice is given by a series of (y, z)-coordinates, and each of the two coordinates will be a 2-byte short. Using the y and z step distance from the previous vertex is another option, but my guess is that the additional complication of tallying these up to get the actual coordinate makes that unattractive. The only advantage of that choice is that we could restrict to only one byte values, which might be better memory-wise if single steps are more common since using more than a byte for each step is wasteful. Instead, as the escape code for move from vertex coordinates to steps, using a negative value. If the y-coordinate is negative the the absolute value of this negative number is the number of steps that will be taken. I can either let each step use two full bytes (wasteful), or pack the steps in at two bits each, or one byte each. In any case, assume that at most 64 vertices will appear per slice. This requires $2^6 \times 4 = 2^8$ bytes per slice. Since there are 2^{10} slices per inch, the total memory required is 2^{18} per square inch. Aside: an $8 \times 8 \times 4$ block has 2^8 square inches of surface area, so that this would require 2^{26} bytes total, or 64 Megabytes.

Ignore the time required for the G-code to rendering step for the time being, and look at the speed of rendering. Remember that this is being rendered as a surface; only one face of the object is drawn. If we visit every 8th slice, then we must visit 2^7 slices. There are 64 vertices, so there are 2^6 line segments to project and draw per slice, or 2^{13} line segments per square inch of surface area. If each line segment requires 2^7 operations to render, then we are looking at 2^{20} operations per square inch.

7 May 20, 2009

After spending a couple of days getting back up to speed, the thing to do is to pick this up almost where (as I recall) I left this off. Implement a verifier based on a simple DA of shorts, focusing on how to display this smoothly and quickly. Then define the surface with a polygon – really, this would be a series of line segments.

I notice that one of the mistakes I was making was in how I move from one pixel/voxel to the next. I shouldn't simply connect the dots. A voxel ends at one edge, and the next step along the perimeter starts at the same edge. Visually, what I was doing meant that every groove appeared to have sloped sides when it really has vertical sides. I fixed that – see `Cut3DPane.paintComponent11()`.

The problem of how to shade realistically is difficult. Here's a thought on that. Build the final image in a two step process. In the first step, render an "image" (though this image will never be displayed), where each pixel has several values associated with it: the distance of the surface point to the observer's eye (as in the z-buffer), and the x, y and z-coordinates of the surface point (the coordinates in the 3-space of the object). In fact, I want something a bit more complicated, which I will describe shortly.

Drawing the final image is the second step. Determining the correct shade for any pixel is a matter of knowing what the tangent plane is at that point. Using the data from the previous step, we know the coordinates of each visible surface element. Wait...this won't work as I've described it.

Instead, the first step must be based on looking at three slices at a time. We need to know the state of the two slices on each side of the middle slice that we are interested in. If I want to get fancy, then I could look at more than one slice on each side, and do some kind of a convolution, but the basic idea is the same. Construct an "image" as usual (the thing that we care about is the z-buffer), and note the slope in each of the two directions for every front-most point. These slopes are what is needed to determine the shade. Now go back and make the final image, translating these slopes to shades.

In fact, doing this in two steps doesn't seem to have any advantages. Just fill each surface polygon with the correct shade to begin with. As a first step, do Gouraud shading with a single shade, where the "Gouraud shading" part of what I'm doing applies to the z-buffer only.

I'm about at the same point where I was when I left this back in January or February. There are problems with the Gouraud shading routines. It's probably just a dumb bug (or two), but it's tangled enough that it's hard to figure out what the problem is. Because I'm going to need polygon-based perimeters at some point, it makes sense to convert the slices of the DA to polygon slices before attempting to render. Theoretically, this doesn't make the problem of how to shade any harder or easier, but it does recast the problem is a more pure form, and that should help me to see more clearly how to approach it.

I also think that part of the problem I am having is due to the fact the the Gouraud shading routines that I've written are for an arbitrary n-gon. I think that by assuming that the polygon must be a triangle, the code will be a bit

A D C B

simpler. It's also true that there are various degenerate cases that occur when I assume that the projection of four points in 3-space is a quadrilateral in the plane. The only way that three points can be a degenerate case is if they are colinear, and that's not hard to deal with. To be strictly correct, I have to use triangles anyway – unless, that is, if I render each square of only 1/1000th of an inch. If I do that, then the four points in 3-space are co-planar, and their projection must be a quadrilateral (perhaps with some acceptable form of degeneracy, like having a colinear projection).

8 May 21, 2009

Shading Gouraud triangles can certainly be done more efficiently. There are only four basic cases, shown in the figure. When specifying a triangle, the vertices can also be ordered in three different ways, and there's the degenerate case where the three points are colinear, or even all the same.

It probably makes sense to have two versions of each method: one that assumes that all vertices are visible in the raster, and one that checks the location of each pixel before it's set. There should be one high-level method that checks this, orders the vertices in a standard way and determines which of the four cases we are in, and checks for degeneracy.

It would be possible to write a polygon shading method strictly for quadrilaterals in the same way. There are more cases and more ways in which degeneracy could occur. If I work with polygon perimeters, then (I think) I could assume that the quadrilaterals being projected really *are* quadrilaterals because their four points lie in a plane. I do wonder whether it which would be faster, even if I put a lot of work into quadrilateral code: calling a triangle shader twice or a quadrilateral shader once. It's certainly the case that there is a tremendous amount of overhead in the code to Gouraud shade arbitrary n-gons (even if there may be a bug or two remaining in my code).

Exactly what to shade is a different question. Think of each slice as a slab. I've tended to try to think of each slice as 1-dimensional, but that gets me into trouble when one slice sticks up above an adjacent slab. It's similar to the problem that I've had with sloped sides of grooves. Imagine that we have two of adjacent slabs, A and B. As we move from A to B, we need to look at the B-side of A to see if any of it is exposed, then at the A-side of B to see if any of that is exposed, then at the top of B. The next step is to move from B to C, etc.

9 May 22, 2009

I realize now that the four cases that I did for filling a triangle could be collapsed into only two cases. There's the case where one of the two sides is a horizontal line (cases A and B combine) and there's the case where that's not true (cases C and D). If I did that, then each of the two implementations would be considerably more complicated, so it looks like a wash.

I wrote code to do all four cases of triangle fill yesterday. It's in the various forms of `graphics.Fund.triFill`. Originally, I wrote version `01`, but it was buggy. To find bugs, I modified `graphics.Tester`; `paintComponent05()` can be used to generate random triangles. Once I realized how hard these bugs are to track down, I stepped backwards and wrote `00` versions of `triFill`. These are based on always calling the same two routines to get the right and left edges. See `getLE()` and `getRE()`. Once these are right, it should be easier to drop them into the `01` versions of `triFill`.

10 May 23, 2009

Concerning the polygon fill code, I've decided to more or less abandon version `01` of the various `triFill` cases, at least for the time being. The `00` version is a lot less code to maintain (and get right to begin with) and the efficiency cost is not large. Whatever improvement I can make there is not going to make or break the entire idea. The real test is going to be whether I can reduce the number of polygons that must be considered.

There are some ways that I might be able to save a meaningful amount of time drawing triangles though. For instance, if the left and right edges of the triangle are drawn as they are observed (for the purpose of determining scanlines), then only the points in between these lines needs to be drawn. This could save a fair amount, especially for thin triangles. Another tweak for long thin triangles is to check whether the two edges have met. Once they meet, the remainder of the triangle consists entirely of edge, and that's already been drawn.

Assuming that there are 64 vertices in each polygonal perimeter, this is 64 rectangles or $2^7 = 128$ triangles. If the block is 8 inches wide, then there are 2^{13} slices and 2^{20} triangles. That's far more than I can draw interactively. If only every 8th slice is sampled, then the number of triangles falls to 2^{17}, which is still too large, although it's getting closer. If 64 vertices along each slice is a valid estimate, then the same estimate should hold for the number of vertices if I cut in the perpendicular direction. Thus, there should be only around $64 \times 64 \times 2 = 2^{13}$ triangles if I can find some way to collapse and combine them, and that's at perfect resolution. However, this would only be the case for non-rough surfaces.

Ultimately, I think that will need some method of triangulating the entire surface. One way to so this is to start with an arbitrary triangle, then try to find one or more new triangles that "cover" an edge of that seed triangle. Don't try to be overly clever about tweaking a particular choice of triangle in an attempt

to get improvements at a later step; just do it greedily. For this to work, I need a couple of things. First, a stack that has all of the "uncovered" edges that remain. Second, some way of keeping track of which elements of the surface have been triangulated, and which remain. One of the harder problems is going to be closing the surface, and a record of which surface elements remain to be triangulated will be necessary for that.

Triangulation could be done under some tolerance value. If a test triangle agrees with the actual suface to within this tolerance, then the test triangle is accepted. If I do this, then one way to speed up the drawing would be to triangulate at several tolerances. Triangulate at a high tolerance to get large but crude triangles, then break these crude triangles into a sub-triangulation at a tighter tolerance, etc. Draw a crude image using large tolerance triangles. See which of these triangles are visible (or are partly visible), then draw tighter tolerance triangles that are inside the crude triangle. If this works, then the total number of triangles to be drawn will be modest, even with a rough surface, and we can zoom in and out quickly. I also wonder whether, by properly choosing the surface normal at each triagle, whether we could get away with drawing fewer triangles. That's the primary point of Gouraud shading – you get smooth looking objects from a rather jagged specification (think of the typical example of a face). In any case, memory is likely to be the bottleneck, not speed (although setting these triangles up in preparation may take some time).

Assuming that the idea above works, then in the absolute worst case, the number of triangles drawn should be of the same order as the number of pixels on the screen. That's a lot, but (see below) it should just be within reach. In practice, my guess is that, even in the worst case, the number of triangles drawn should be only about a quarter of the number of pixels since, realistically, I expect that many triangles will span more than a single pixel.

An assumption/restriction that I think will simplify things is to require that every vertex of a triangle coincide with a point of the actual surface (even for very rough surfaces) . In other words, each of the x and y coordinates must be a multiple of 0.001 inches, and the z coordinate must be chosen so that (x, y, z) is on the surface. This will make breaking large triangles into sub-triangles easier and more natural.

I'm pretty sure that the triangle drawing code works. See `graphics.Tester` for some test routines. The `paintComponent` methods, versions `04`, `05` and `06` were all used for testing.

Next up: write code that draws the surface using triangles, where each triangle has a single shade based on its depth. It took from 2 to 4 seconds to draw the same triangle 1,000,000 times using my current code (`triFill00()`), depending on the size of the triangle. Even the smallest triangle, that took 2 seconds to draw 1,000,000 times, was much larger than I think will be typical. So, drawing 100,000 triangles should be within reach and I hope that I can manage with a lot less than this. I do wonder whether these numbers are unreasonably good since the hotspot compiler and/or CPU might be super efficient when executing the same block of code repeatedly. Moreover, adding depth and shade to the scanline fills will slow them down by a factor of at least two. I still need to write

versions of the code that don't check whether each pixel is on the raster. That would save a fair amount of time – maybe as much as will be required to handle variation of shade and depth.

Before I can embark on this goal, I need a z-buffer with my shaded triangles. This shouldn't be too hard. I just need to add depth values to the two edges and modify the scanline filling code slightly.

11 May 24, 2009

Implemented `addToLE` and `addToRE`, in the `Fund` class, that includes the depth of each pixel. Then I wrote `triFill02()`, and the sub-cases, to fill triangles in the context of a z-buffer and I started to write version `13` of `paintComponent` in the `Cut3DPane` class of the `da3ddisplay` package. There's still some ways to go, but it shouldn't be too bad.

12 May 25, 2009

I finished the code to draw the figure in 3D, but there must be a problem with the depth code. It hangs – my guess is that it goes into an infinite loop somewhere when I do the `while`s to adjust the value of `derr`. If I draw the triangles without worrying about the z-buffer, then it does work, though with a few visually obvious mistakes (aside from the fact that there's no z-buffer) in my implementation that appear to be easy to fix.

I'm having trouble tracking down all of the bugs in `Cut3DPane`. I found some, but the one that I can't find is the tendency of the renderer not to draw triangles that are vertical boundaries between slices. I thought that it might be the underlying triangle drawer, so I wrote version `14` of `paintComponent()`. There are some minor problems with the triangle filling code – twist the triangle around and you'll see them, though they are uncommon – but I don't think that's the real problem with the rendering.

13 May 26, 2009

As I look at the (buggy) output that I have so far, I think that adding smooth shading will make the output look worse. There needs to be some contrast between faces that are at an angle to each other. If I implement true Gouraud shading, then these faces will be smoothed together. Gouraud shading will only work if I also take into account the angle of the face relative to a light source.

Most of the problems with triangles have been fixed (leaving aside the fact that depth does not work). There might still be occasional triangles that don't quite meet up at the seams, but I'm not sure.

Getting the z-buffer code to work was not hard after all. However, there are weird problems with triangle drawing. It's a lot like what I had without the

z-buffer, before I fixed the issues in `triFill00`. Other than that, the z-buffer seems to work.

14 May 27, 2009

Under `da3ddisplay.Cut3DPane.paintComponent13()` (version `v02`), almost everything is fixed, and the program works with a z-buffer. There are still some minor and occasional problems with the triangle filling routines, but they are visually minor. It seems like the edges of triangles don't always meet, and this leaves a bit of a dotted line between them. I'm not sure whether this is due to the triangle filling code itself, or whether it's some kind of rounding error that introduces error in the rotated vertices, but I will leave it on the back burner for now.

Drawing 50,000 triangles seems to be no problem (although 100,000 would be too many), and I should be able to speed things up a quite bit (15%?) by changing the test as to whether or not a pixel is on the raster so that it's not made at every pixel.

Where to next? An intermediate goal is to write a program that can handle endmills, ballmills and drill bits on a three-axis machine with no undercutting. This accounts for almost all situations that arise, so it would be genuinely useful and would be a way to force me to work out the kinks in a less complex system before going any further. Problems that still remain are:

1. The shading doesn't look good. Right now, it's based on the depth of the face. There are lots of reasons that this doesn't look good. We need high contrast between faces that are at an angle. In fact, shading is likely to be a big problem. The best method of shading will depend on the nature of the object. A face-type surface should use relatively smooth shading (though contrast may be an issue), while a more typical object that consists of straight sides might look best if green was used (say) for the vertical parts and white for the horizontal parts. I'll probably need to provide the user with a menu of shading methods, and let him choose the one that looks best for the given project. Shading should depend on the degree of zoom too. If the user zooms in so that only 0.020 is visible, then show him the blocks.
2. Use something like light blue for the background. Otherwise through-holes are hard to see.
3. Investigate why copying the rendered image takes so long. I mean the call to `Graphics.drawImage()`. This often takes longer than generating the image itself. Getting this down to a speed that's in line with what I would expect would free up enough time to draw another 25,000 triangles.
4. Think about through-holes. The big complexity that this introduces is the fact that slices may consist of several disjoint perimeters. However, if I only allow cuts on one side of the material and the "perimeter" is restricted

to only the top of the material, then this is a non-issue. Through-holes can be noted by setting the point to be at "height" equal to $-\infty$.

5. Make the frequency of vertex sampling (or the size and accuracy of the triangles) depend on the degree of zoom. This will speed things up a lot.
6. Give the user some control over sampling. He might want to work entirely in multiples of 0.010 for very large objects, and in rare cases, they may want to go down to 0.0001, although memory will be such a huge issue that the latter seems impossible to allow. Also, at the rendering step, he may see a need for greater or less detail than the default.
7. Improve the implementation of triangle shading to be faster. The biggest change here is checking whether a triangle is entirely on the raster before begining to shade it.
8. The rotation of vertices could be made faster, but I'm not sure if this is worth the trouble since it's not the bottleneck.
9. Add Gouraud shading to the triangles. Right now a triangle is a solid shade.
10. Fix up the renderer to eliminate the small errors that lead to seams and jittering pixels. This might be a z-buffer issue (at least in part).
11. Render from a list of triangles rather than from the slices. This might make it much faster, but generating these triangles will not be easy to do. The difficult part is generating these triangles at increasing levels of accuracy.
12. Do I want to work with a DA, and convert that to polygons, or do I want to work directly with polygons? If I use polygons, then I need some means of inserting "rough patches" when the surface is too rough to use polygons efficiently.
13. The slab of material needs sides and a bottom.
14. Some way to specify the size of the slab of raw material. Allow the material to have shapes other than a simple rectangle. Any easy way to allow a fair amount of generality is to allow the user to save objects and use them as the starting point. They could write a program that makes a hexagon (say) from a rectangle.
15. A way to indicate the PRZ for the material. Most of the time it will be the corner of a rectangle, but not always.
16. A way to indicate the tool radius. Really, a tool turret is needed to handle the three possible tools: endmills, ballmills and drills. This should handle tool length offsets too.
17. Complete the implementation of the G-code interpreter so that it understands more than `G00` and `G01`.
18. Allow sliding the object in addition to rotating it.

19. Allow the user to move the eye? This is complicated but it would allow him to get down in holes to see what's in there. Another way to accomplish this is to let him slice off parts of the object so that he can see cross-sections. That would be much (!) easier to do.
20. Allow using the mouse to rotate and slide? Using the mouse seems to be common, though I don't like it.
21. Allow the user to specify which keys are used for rotation.
22. Think about metric verus imperial issues. In theory, a single G-code program could use both metric and imperial. My system has to choose one or the other before it parses the G-code.
23. Ability to click on a line of G-code and see what it cuts. Rendering these cuts with alpha (i.e., transparency) is one way to do this visually. I would like to allow the user to view the formation of the object as the tool cuts (like a movie), but that seems hard to make fast enough.
24. I have been focused on the rendering step, and converting the G-code to an object is likely to require some efficiency improvements, though I'm not sure what. I suspect that once I use realistic tool sizes (like 0.250 inches), this is going to be a bigger issue. Looking some steps forward may be an easy thing that makes a big difference.
25. Implement Macro-B to G-code compiler. In some ways this is a seperate issue, but doing this meshes nicely with the problem of fleshing out the G-code interpreter. I do wonder whether there might be legal problems with doing this. Macro-B is a product of GE and there might be a way for them to prevent me from doing this.
26. Do some simple collision-detection type things. For instance, does the tool touch the material while the spindle is turned off?

Letting the rendering issues stew in my head for a while seemed to be a good idea. So focusing on the lexer/parser/interpreter is what I'll do for the next few days.

Certain codes are the most important, and these are the ones that introduce the greatest complexity. These are: `G02/03` (circular interpolation), `G40/41/42` (radius offset), `G43/49` (tool length offset), `G80...89` (various canned cycles), `G90/91` (absolute and incremental moves), `I, J, K, R` (for circular interpolation), `H, D` (for TLO and radius compensation), `L, P, Q` (for subroutines or canned cycles), `M98/99` (call/return subprogram).

Others that I may as well allow are: `G04` (dwell) `G17/18/19` (to select a pair of axes; I don't see why this matters), `G20/21` (inches or mm), `G28` (return to home), `G54/55/etc.` (though not sure how to handle them), `U` (used with dwell time), `M01` (program stop), `M02` (end of program), `M03/04/05` (spindle start/stop), `M06` (tool change), `M07/08/09` (coolant), `M30` (end of program), `M40/41` (spindle low/high), `M47` (repeat program), `M48/49` (enable/disable overrides).

15 May 29, 2009

I've made progress on extending the interpreter to handle more G-code commands, but I still have a ways to go. This is mostly drudge work without any great complexities.

What I've realized is that some of my analysis of what's involved in going from G-code to some form of 3D representation leaves out important details. One of the advantages of working with a DA is that the correspondence between the coordinates of the tool and which data needs to be adjusted is clear. The 2D array of the DA is in one-to-one correspondence with the coordinates. For the perimeter representations, that is not true. Given a tool position, it's easy to determine which slices are affected, but it's not so easy to determine which entries of each slice must be adjusted. Whether the slice is based on walking the perimeter or on polygons, to determine which entries are affected the program must walk the entire perimeter. This is especially true given that there could be undercutting and/or disjoint regions of the slice. There are two broad ways to handle this difficulty: for each tool position, search the perimeter; or, for each slice, search the tool positions.

If the program walks through the tool positions more or less in the order that they are generated by the G-code, then I don't see any way to avoid scaning the entirety of the perimeter of every slice that the tool intersects. There are some things that could be done to speed this up a bit, like looking a few moves ahead. Another thing that might help is a lookup table that maps from 3D coordinates to points in each slice. The latter idea would speed things up a lot, but updating this table might take enough time to cancel any benefit. This map could be looked at as continuously "sorting" the elements that make up the slices.

The other option is to work through the slices and, for each slice, see if there are any tool positions which touch that slice. It might be possible to speed this up by sorting the tool positions somehow. My instinct is that this will be faster than the earlier approach (sorting elements of the slices) and it should be simpler to program.

Consider a set of tool positions, all of which apply to the same tool (so every time that the program changes tool, I would need to go through the following process). Sort these tool positions and eliminate any tool position (x, y, z), where there is another tool position of the form (x, y, w), where $w < z$. The point is that for the three tools to which I am restricting the program (endmills, ballmills and drills), only the lowest position for any given (x, y) matters. Another obvious thing to do is throw out any tool position that clearly does not intersect the material (any one of the coordinates too large or small). These things are so easy to do that they make sense, but it won't usually elminate many tool positions. After this is done, for each (x, y) there will be at most one tool position, at the deepest z for that tool position. Sort these positions on y, then x. In some ways, the result of this sort is very much like a DA.

Move down from one slice to the next. For each slice, walk along its perime-

ter. Let r be the radius of the tool. For each point of the perimeter (whether the perimeter is given by polygons or steps), the program must consider all tool positions within distance r of the given point.

An improvement may be to sort the tool positions on z, then on y, then on x. So, for each z, we have a seperate set of sorted (x, y). If a given slice has already been cut down to depth z, then we don't have to consider any of the tool positions higher than this z. The question then becomes, for each point of a slice, does any tool position within distance r touch that point. If so, then we don't have to look any further. Moreover, I expect that most of the time the program will be able to make steps of size r along the perimeter. If the tool crosses a slice at a given point, then it adds r to the length of the new "furrow". The tool may merely graze the slice, and the step will be for a distance of less than r, but it should frequently be as large as r. In any case, one the program finds a tool position that touches the slice, then it doesn't have to consider any more distant tool positions.

I'm not sure, but I might be able to walk down all slices and all of the sorted tool positions at the same time, at least for the current version of the program that allows only simple cuts. To do this we need a counter for each slice to keep track of how far (x) we've travelled down that slice, along with an x-counter for each y-value of the sorted tool positions. I'm not positive that this would work though. I'm concerned that I might really need a different y-tool position counter for every slice – in fact, I think this is the case.

For the more general program, I still think that these ideas will work, they just can't be relied upon to save as much time. There are two primary differences. First, the perimeter really is a loop, and it may be a set of disjoint loops; each loop has to be considered seperately. Secondly, for oddly shaped tools and/or multi-axis machines, I would need to consider some kind of bounding box (or bounding cylinder) for the tool and look at any tool position in that bounding box. Most likely, each general type of tool will require its own set of code to handle the way in which it cuts. One way to handle oddly shaped tools is to convert the single tool position (which I think of as the base of the tool) to a set of z-values. The idea is that the intersection of the tool with the (x, y)-plane is a circle...I'm not sure where this is going, but I think it's worth thinking about. For tools at at angle on multi-axis machines, things are not so simple. The intersection of the tool with the (x, y)-plane will typically be an ellipse, not a circle.

This idea of sorting the tool positions really seems to be the fastest way to approach the issue, whether working with the restricted set of tools and only three axes, or the more general setup. I will need to walk down each perimeter, taking relatively small steps, and searching for nearby tool positions as each step, but most of the time there will be no nearby tool positions (which will be fast to observe); the tool position will be right over the line much of the time, which is also fast. Considering that I have to visit every step of the perimeter under the current method of rendering, and how much computation is done at each position, this strategy should work fine.

16 May 20, 2009

Implemented `TextBuffer` and `TextGetter` in the `ui` package. These allow buffering of the contents of a `JTextArea` object. To use these, I had to modify a lot of code, all the way from the `MaineWindow` down to the `Lexer`. None of this was complicated, but it touched a lot of stuff from the interpreter (and how it's called) on down.

A similar change is pre-parsing the G-code to identify subprograms. Allowing the ability to call sub-programs (and sub-sub-programs, etc.) adds a real complication. The best way that I can think of to handle this is with a stack of interpreters. Most aspects of the machine's state, like the absolute coordinates of the tool, are global to all interpreters, and these values will need to be shared. They can probably be static. Really, this is the same as a stack of program counters in assembler.

Another moderately large change that I must make is to go from "pulses" (see `OnePulse` in the `interpreter` package) to "signals". There are a couple of reasons to do this. Under the old scheme, every third pulse indicated whether the x-axis (say) should move. If the x-axis never moves, then zero-valued pulses are still generated. So, using signals saves some memory. Another reason is that some signals, like tool changes, aren't changes to an axis at all, and adding them to the existing framework would be messy.

These two changes, allowing subroutines and going from pulses to signals, are more involved than they seem. They'll take another day or two.

I don't want to maintain the ability to display wireframes, so I got rid of everything to do with that, including the entire `wireframe` package. It was nice for testing at earlier stages, but it wouldn't be easy to allow curves.

I realize now that the way that I organized the lexer/parser/interpreter was not the best choice. The problem is the buffers that I use at each layer. They make the code cleaner, but they present other problems. The problem is that there's no easy way to move around in the code to call subroutines – actually, it's relatively easy to program; the problem is that there's a tremendous amount of overhead required to fill the buffers. If the subroutines are themselves long programs, then this overhead pays for itself, but for very short subroutines, the buffers are much larger than necessary.

Any situation involving subprograms where this would lead to a significant speed problem seems like it would be a contrived example, but it's a different story with Macro-B. That's closer to a real programming language and jumping around the code will be very common. So, before I implement Macro-B (but only if I implement it), I need to rethink this framework. Part of the problem is that it's not easy to get your hands on the text inside of a `JTextArea` and I wanted to avoid reading the entire contents as a single string or char array. Maybe loading it all like this isn't such a big deal. One megabyte amounts to more than 20,000 lines of code, which is a mighty long program based on Macro-B (shorter code is one of the points of Macro-B). Also, there might be a way to get my hands on the actual underlying data, especially if I roll my own text interface. I think that the `Document` class is the one that I need to look at.

However, even if I go ahead and read the entire program into a char array, that doesn't solve all of my problems. The lexer and parser have buffers of their own. Some degree of buffering is needed for cutter comp. One way to handle this is to perform peeks on the fly rather than simply pulling them out of the next buffer entry. One problem with this is the need to peek ahead for an entire statement (or several of them) which will typically be made of several tokens. This will make peeking slower, but it shouldn't be that much of a problem in terms of speed. It seems that parsers often rely on code that puts a character (or whatever) back into the buffer after its been read, rather than thinking in terms of peeking. I'm not sure what makes sense yet.

17 May 30, 2009

Writing the compiler/interpreter is generating all kinds of problems that I really hadn't thought through. I need to go back and refresh my memory about how compilers work before I jump any further into this. It may be that I should be using lex and yacc (or flex and bison) after all. In the meanwhile, there's still a lot that I can do. There are user-interface issues that I could tackle, like implementing some of the G/M-code features, changing the pulses to signals, issues of the PRZ and metric/imperial and scaling, and graphics issues.

I've begun changing from pulses to signals. Using pulses does make more sense if you are thinking of hooking this program up to an actual machine. In that case, every motor is getting a signal at all times, even if it's usually a zero. Also, using signals means that feed rates are totally ignored. The code that generates signals was changed a lot compared to using pulses. In some ways, it's simpler now that I've changed it. It may be that even if I go back to pulses, the cleanest way to do this would be to convert the signals to pulses (however, this doesn't help with issues of feed rate). I finished this in only a few hours, and it wasn't that bad.

18 June 1, 2009

After reading various compiler books (not very carefully), I think that using lex and yacc is not needed to interpret ordinary G-code, but I will probably want to use it to deal with Macro-B. The hairiest part of compilers/interpreters is dealing with variable declarations and types, but there are no types in Macro-B, and all variables are of the form #[number]. So, I could probably write the interpreter by hand, but there's one messy thing that I can't avoid: complicated expressions. Those pretty much require a parse tree. I'm sure that I could write such a thing by hand, but I'm not sure if it's worth the trouble. One reason not to use lex and yacc is that they output a C program, and I would have to convert it to Java. That sounds easy, but there's a lot of glue code involved in getting the source code into the program, and it looks like the generated code uses a lot of C tricks (pointer arithmetic). All of this means that going from a lex/yacc

specification to Java will be a multi-step process so that debugging, testing and making changes will be frustrating. There are lex/yacc-like tools written in and for Java, but learning how to use them would take time. The are really only two major sources of complication when interpreting Macro-B: parsing expressions and calling subroutines (and things like goto, if/then or loops that jump and branch). Moving around in the code can be handled with a stack, and I will have to deal with that issue whether I use lex & yacc or not. Because there is no symbol table (or a very simple one), parsing expressions is doable. One reason to go ahead and use a parser-generator is learn how to use some of the more sophisticated tools. That way, if I ever want to extend Macro-B to be more like a real programming language, then I'll have a better idea how to do it. See `catalog.compilertools.net`. In paricular, there's a list of Java tools. There are several lex & yacc ports. There's also something called "ANTLR", which seems sophisticated, well-documented and widely used. It has an Eclipse plug-in too. Note that there's an older version 2 of ANTLR, which may be easier to use for a novice (or not, user's comments make it sound like version 2 was a nightmare by comparison).

For the time being, I will concentrate on the interpreter for G-code without Macro-B and without using any compiler generation tools. The only real complication when interpreting ordinary G-code is how to call subprograms. To do this I need a call stack and a program counter. When a subprogram is called, do the following. Let n be the number of times that the subprogram is to be called (the `L`-value). Push a pointer to the line after the call to the subroutines onto the call stack (for the return), then push $n-1$ copies of the first character of the subprogram onto the call stack. Then jump to the subprogram. This handles the `L`-value very cleanly; one disadvantage is the that stack could be very large if thousands of calls are made. If this is a problem, then I could give each stack entry two values: the address to which to jump, and the number of times that the jump should be made. Each time we jump to the given address, decrement the number of jumps left. When the count reaches zero, pop the stack.

I still want to read up on G-code and Macro-B to make sure that I'm not unaware of something important, so I will focus on the more ordinary G-code commands like circular interpolation for now.

Important Observation. To handle cutter comp I have been thinking in terms of looking ahead to the next move so that I know how to terminate the current move. That would work, but it makes the parser/lexer much more of a problem since the number of characters forward that I must be able to peek is unlimited. Moreover, when subprograms may be called (or under Macro-B), a given move may be the next move in a lexical sense, but not in a logical sense. Peeking ahead to the next move will be almost impossible in that situation. A much better way to handle this is to hold onto any moves when cutter comp is in effect, and not execute them until the next move comes along. If I do that, then I don't think that the lexer (or parser) ever needs to peek ahead by more than one character. Keeping track of these buffered moves will be some work, and I will have to be careful that they are always flushed, but it's really the only way to handle this problem.

19 June 5, 2009

I've implemented more of a front-end for this thing. It's not fancy, but it has all (?) of the inputs that the program can use: metric versus imperial, a tool table (endmills, ballmills and drills), a way to specify the size of the starting material, and a table for work offsets (`G54`,..., `G59`). None of it is used when actually generating the solid (yet), but it's all there.

Dealing with all of the various modes that the machine can be in is complicated. Some of these machine states are easy to deal with. Things like feed rate and spindle RPM don't really matter for in a simulation, but others are more complicated and interrelated. Here's a list.

- Whether currently in rapid traverse, the feed rate and whether the spindle is on or off, along with the spindle RPM. These I can safely ignore. The only reason to even keep track of them is in case I want to check for problems in the code, like attempting to cut when the spindle is off.
- Choice of plane, as in `G17`. This should be relatively easy. The only situation in which this matters is when doing helical cuts and polar coordinates. Handling this will be one more special case when dealing with circular interpolation.
- Inch versus metric. This doesn't have a lot of side-effects, but it is a nuisance. Every time that a coordinate comes through the parser, it must be converted to the units of the machine. Once that's done, nothing else should be affected.
- Work offsets, either with `G52` or the more common use of `G54`,...,`G59`. I think I can handle this by adding some standard offset value to every coordinate, something like inch versus metric.
- Tool length offset on or off. Remember that `G43` means that a positive H-value implies that the tool will be raised when entering TLO mode. When the user enters TLO mode, there shouldn't be any problem crashwise with immediately raising the tool by the H-value. When a program reaches `G49` to cancel TLO, some machines might try to be smart and not lower the tool until it can do so in the context of the next move, thus avoiding potential crashes. Other machines don't do this (like the one at school that I crashed in exactly this way). Moving immediately on `G49` makes sense in that it's more conservative (will detect problems on more machines). It's easier to implement too.

  I should (?) be able to handle this with an offset similar to what's done for work offsets (`G55` and the like). The main difference is that `G55` (say) doesn't move the tool; it means that all coordinates from that point are to be taken relative to the new origin. TLO does move the tool, and only the horizontal coordinate changes, but otherwise it's the same idea.
- Absolute versus incremental mode. I don't think this will be very complicated, but it does mean that almost every move will have special cases.
- Cutter comp on or off. This is the one that's a royal pain. Note that the plane is not relevant (basically the offset is alway considered to be in the

(x, y)-plane, although I suppose that it's more complicated for multi-axis machines). According to Smid (p. 258), the older "type B" machines don't have look-ahead. Programming that way would be easier, but my guess is that nearly all machines today are "type C", which do have look-ahead. In any case, "look-ahead" is the wrong idea; see my comments a page or two above. A better way to do this is to buffer moves under cutter comp, and only output the pulses for a given move until the next move has been seen. It's this buffering that makes things complicated.

So, when a program enters cutter comp mode, it must begin buffering, and the first move is the one in which cutter comp gradually takes effect. For lines, that's no problem, but for arcs, it will be messier. Just as for TLO, the conservative thing to do is make the tool "jump" the instant that cutter comp is cancelled.

- Polar coordinates on or off. To some degree this can be handled in a way similar to inch versus metric. The x and y-coordinates are really r and θ under `G17`. I do have to see which plane has been chosen and I assume that the tool is meant to move in arcs of circles so that, under polar coordinate mode, changes to y are really changes to θ and the move must be treated as a case of circular interpolation. If both r and θ change, then the tool must move in a spiral. My guess is that most real machines treat this in a variety of ways. This will add multiple special cases.

  There's no obvious answer as to how polar coordinates should be implemented. I've chosen to follow Smid's description (see pp. 233-4). This means that absolute versus incremental mode is irrelevant. When `G16` is called to invoke polar coordinates, the tool position at that point becomes a new origin – what Smid calls the "pivot point." When the user enters polar coordinate mode, both cutter comp and absolute/incremental are temporarily suspended. In fact, my program won't allow a `G16` when the machine is in either incremental mode or when cutter comp is in effect.

Handling all of these issues is going to be a pain. The best way to proceed seems to be to implement cutter comp last. That's the one that will cause the most grief. Do circular interpolation next-to-last.

20 June 8, 2009

I'm making good progress. I still have to handle circular interpolation, cutter comp and calling subroutines. Circular interpolation has one tricky thing. It's possible to specify an arc by giving the two end-points of the arc and the radius. To draw the arc, I need to know the coordinates of the center of the arc. Let $(x1, x2)$ and $(x2, y2)$ be the end-points of the arc, and let r be the radius. We

must have

$$\begin{aligned}(x_1 - c_x)^2 + (y_1 - c_y)^2 &= r^2 \\ (x_2 - c_x)^2 + (y_2 - c_y)^2 &= r^2,\end{aligned}$$

where (c_x, c_y) is the center of the circle. We have two equations in two unknowns, so there is a solution. I fussed with that a bit, and I found someone on the internet who solved it (at least I think that's what he did), but it's hideous. Much better is to work in parametric coordinates. In this case, we have

$$\begin{aligned}x_1 &= c_x + r\cos a_1 \\ y_1 &= c_y + r\sin a_1 \\ x_2 &= c_x + r\cos a_2 \\ y_2 &= c_y + r\sin a_2,\end{aligned}$$

where a_1 and a_2 are the angles of the end-points. This is four equations in four unknowns, and it has the advantage of finding a_1 and a_2 directly, and I will need those two values.

Subtract to eliminate the center coordinates to get

$$\begin{aligned}x_2 - x_1 &= r(\cos a_2 - \cos a_1) \\ y_2 - y_1 &= r(\sin a_2 - \sin a_1).\end{aligned}$$

Now we have two equations in two unknowns, and we need some trig tricks. Recall the trig identities (the product-to-sum identities):

$$\begin{aligned}\cos x \cos y &= \frac{1}{2}[\cos(x+y) + \cos(x-y)] \\ \sin x \sin y &= \frac{1}{2}[\cos(x-y) - \cos(x+y)] \\ \sin x \cos y &= \frac{1}{2}[\sin(x+y) + \sin(x-y)].\end{aligned}$$

Actually, a better form to use is the sum-to-product version. These are

$$\begin{aligned}\sin x + \sin y &= 2\sin\left(\frac{x+y}{2}\right)\cos\left(\frac{x-y}{2}\right) \\ \sin x - \sin y &= 2\cos\left(\frac{x+y}{2}\right)\sin\left(\frac{x-y}{2}\right) \\ \cos x + \cos y &= 2\cos\left(\frac{x+y}{2}\right)\cos\left(\frac{x-y}{2}\right) \\ \cos x - \cos y &= -2\sin\left(\frac{x+y}{2}\right)\sin\left(\frac{x-y}{2}\right)\end{aligned}$$

Using these identities, we now have

$$\begin{aligned}\frac{x_2 - x_1}{r} &= -2\sin\left(\frac{a_2 + a_1}{2}\right)\sin\left(\frac{a_2 - a_1}{2}\right) \\ \frac{y_2 - y_1}{r} &= 2\cos\left(\frac{a_2 + a_1}{2}\right)\sin\left(\frac{a_2 - a_1}{2}\right)\end{aligned}$$

Set $A = (a_2 + a_1)/2$ and $B = (a_2 - a_1)/2$. Then these two equations become

$$\begin{aligned}\frac{x_1 - x_2}{2r} &= \sin A \sin B \\ \frac{y_2 - y_1}{2r} &= \cos A \sin B,\end{aligned}$$

and solving these for A and B suffices to find a_1 and a_2.

A little algebra shows that

$$A = \arctan\left(\frac{x_1 - x_2}{y_2 - y_1}\right)$$

and plugging this into $\frac{y_2 - y_1}{2r} = \cos A \sin B$ gives

$$\begin{aligned}\frac{y_2 - y_1}{2r} &= \cos\left[\arctan\left(\frac{x_1 - x_2}{y_2 - y_1}\right)\right]\sin B \\ &= \left(\sqrt{1 + \left(\frac{x_1 - x_2}{y_2 - y_1}\right)^2}\right)^{-1}\sin B\end{aligned}$$

or

$$\begin{aligned}\sin B &= \frac{y_2 - y_1}{2r}\sqrt{1 + \left(\frac{x_1 - x_2}{y_2 - y_1}\right)^2} \\ &= \frac{1}{2r}\sqrt{(y_2 - y_1)^2 + (x_1 - x_2)^2}\end{aligned}$$

and

$$B = \arcsin\left(\frac{1}{2r}\sqrt{(y_2 - y_1)^2 + (x_1 - x_2)^2}\right)$$

From the definitions of A and B, we must have

$$\begin{aligned}a_1 &= A - B \\ a_2 &= A + B.\end{aligned}$$

This isn't quite the whole story. Given two end-points and a radius, there are two possible locations for the center. Draw a picture and let α_1 and α_2 be

the angles to the two points using one choice for the center, and let β_1 and β_2 be the angles using the other choice for the center. We must have

$$\begin{aligned} \beta_1 &= \pi + \alpha_2 \\ \beta_2 &= \pi + \alpha_1. \end{aligned}$$

You can see this from a picuture using horizontal and vertical parallel lines though end-points and centers and the fact that the lines from the centers to the end-points form a parallelogram. NOTE: on re-thinking this, I'm not sure that this formula is true. In any case, I must know the coordinates of the center, so I'll use the method outlined below.

Another way to look at this problem is as follows. Recall that if U and V are two points, then $V - U$ is the vector pointing from U to V. Also, if (r, s) is a direction vector, then $(s, -r)$ and $(-s, r)$ are perpendicular to (r, s). Let

$$q = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

be the distance between the end-points of the arc. Let

$$m = (1/2)(x_1 + x_2, y_1 + y_2)$$

be the mid-point of the line connecting the two end-points. The two possible centers of the circle are along a line perpendicular to the line connecting the two end-points which passes through m. The direction vector

$$v = (x_1 - x_2, y_1 - y_2)$$

points from (x_2, y_2) to (x_1, y_1), and

$$v_\perp = (y_2 - y_1, x_1 - x_2)$$

is perpendicular to v. The distance from m to either center is

$$\sqrt{r^2 - (q/2)^2}$$

(draw a picture). Dividing $v_\perp$ by q normalizes it to have length 1 so that the two centers are given by

$$m \pm (v_\perp/q)\sqrt{r^2 - (q/2)^2}.$$

The issue of determing CW versus CCW (and which of the two centers to use) still remains. Given three points on a circle, this question is equivalent to asking whether the three points are given in CW or CCW order. Let P_1, P_2 and P_2 be three points (they don't really have to be on a circle for this to work though). Define

$$E_1 = P_1 - P_2 \qquad \text{and} \qquad E_2 = P_3 - P_2.$$

Then E_1 points from P_2 to P_1 and E_2 points from P_2 to P_3. Moving from P_1 to P_2 to P_3 is CW iff the angle between E_1 and E_2 is less than $180°$. Unfortunately,

I can't simply use the dot product to determine this angle since the dot product always gives a value in $[0, 180°]$. Also, the ideas of CW and CCW only make sense relative to some center of rotation.

Just to be fussy, I want to check that the expression above really does act like the center of the circle. In particular, we should have

$$r^2 = (c_x - x_1)^2 + (c_y - y_1)^2.$$

Plug in the expressions for c_x and c_y, and we should have

$$\begin{aligned}
r^2 &= (c_x - x_1)^2 + (c_y - y_1)^2 \\
&= \left(m_x \pm (v_\perp/q)_x \sqrt{r^2 - (q/2)^2} - x_1\right)^2 + \left(m_y \pm (v_\perp/q)_y \sqrt{r^2 - (q/2)^2} - y_1\right)^2 \\
&= \left((x_1 + x_2)/2 \pm ((y_2 - y_1)/q)\sqrt{r^2 - (q/2)^2} - x_1\right)^2 + \left((y_1 + y_2)/2 \pm ((x_1 - x_2)/q)\sqrt{r^2 - (q/2)^2} - y_1\right)^2
\end{aligned}$$

The $\pm$ is confusing so, let σ be $+1$ or -1 so that we can write this as

$$\begin{aligned}
r^2 &= \left((x_1 + x_2)/2 + \sigma((y_2 - y_1)/q)\sqrt{r^2 - (q/2)^2} - x_1\right)^2 + \left((y_1 + y_2)/2 + \sigma((x_1 - x_2)/q)\sqrt{r^2 - (q/2)^2} - y_1\right)^2 \\
&= \left((x_1 + x_2)/2 + \sigma((y_2 - y_1)/q)\sqrt{r^2 - (q/2)^2} - x_1\right)^2 + \left((y_1 + y_2)/2 + \sigma((x_1 - x_2)/q)\sqrt{r^2 - (q/2)^2} - y_1\right)^2 \\
&= (1/q^2)\left(q(x_1 + x_2)/2 + \sigma(y_2 - y_1)\sqrt{r^2 - (q/2)^2} - qx_1\right)^2 + (1/q^2)\left(q(y_1 + y_2)/2 + \sigma(x_1 - x_2)\sqrt{r^2 - (q/2)^2} - qy_1\right)^2 \\
&= (1/q^2)\left(q(x_2 - x_1)/2 + \sigma(y_2 - y_1)\sqrt{r^2 - (q/2)^2}\right)^2 + (1/q^2)\left(q(y_2 - y_1)/2 + \sigma(x_1 - x_2)\sqrt{r^2 - (q/2)^2}\right)^2 \\
&= \left(\frac{1}{q^2}\right)\Big[(q^2/4)(x_2 - x_1)^2 + \sigma q(x_2 - x_1)(y_2 - y_1)\sqrt{r^2 - (q/2)^2} + (y_2 - y_1)^2(r^2 - (q/2)^2) \\
&\quad + (q^2/4)(y_2 - y_1)^2 + \sigma q(y_2 - y_1)(x_1 - x_2)\sqrt{r^2 - (q/2)^2} + (x_1 - x_2)^2(r^2 - (q/2)^2)\Big] \\
&= \left(\frac{1}{q^2}\right)\left[(q^2/4)\left((x_2 - x_1)^2 + (y_2 - y_1)^2\right) + \left((y_2 - y_1)^2 + (x_1 - x_2)^2\right)(r^2 - (q/2)^2)\right. \\
&\quad \left. + \sigma q\left((x_2 - x_1)(y_2 - y_1) + (y_2 - y_1)(x_1 - x_2)\right)\sqrt{r^2 - (q/2)^2}\right] \\
&= \left(\frac{1}{q^2}\right)\left[(q^2/4)\left((x_2 - x_1)^2 + (y_2 - y_1)^2\right) + \left((y_2 - y_1)^2 + (x_1 - x_2)^2\right)(r^2 - (q/2)^2)\right] \\
&= \left(\frac{1}{q^2}\right)\left[(q^2/4)q^2 + q^2(r^2 - (q/2)^2)\right] \\
&= q^2/4 + \left(r^2 - (q/2)^2\right) \\
&= r^2
\end{aligned}$$

That was (sort of) a waste of time, but it does verify the equation for the center, or that it's valid for (x_1, y_1) anyway.

21 June 10, 2009

I finished implementing circular interpolation. There are probably bugs in it, and it assumes `G17` and doesn't do helical milling, but it's basically there.

Cutter comp is the next thing. I'm sure that the idea of buffering each move and not writing out the signals until the next move is seen is the right way to do cutter comp, but there are a lot of cases. First is the split between cutter comp on the left and on the right, but symmetry will make that easy to deal with. In each case, the issue is how to cut path A, given that path B is the one that follows A. Assume that path A is a line (`G01`). B could be a line, where the angle along with the tool travels is acute or obtuse; or B could be an arc of a circle. If B is an arc, then there are four possible cases: B could travel CW or CCW, and the angle between A and B could be acute or obtuse (it's not really an angle, but the idea is clear). Assume that A is an arc. If B is a line segment, then we are in a situation similar to one already considered; it's just that the order is reversed. If B is an arc, then there are four possible cases. Pictures would be good here, but I don't feel like fooling with MetaPost.

Geometrically, there are two important cases. Adjacent curves meet at either an acute angle or an obtuse angle. In both cases, we need the "offset curve" that the tool follows for cutter comp. If the angle is acute, then the tool goes to the point where these lines intersect. If the angle is obtuse, then the tool goes to the last point of the offset curve for the previous path, and it follows an arc to get to the first point of the offset curve of the new path. I suspect that I could apply this logical framework to any kind of curve, not just line segments and arcs of circles; the key datum is the vector tangent to the paths at the point where the paths meet. In fact, it may make sense to change the way that the code is organized (especially if I extend G-code with new commands for additional curves) so that each curve is represented by an object. As it is, the code explicitly branches to handle different combinations of adjacent curves. What matters is these tangent vectors and the intersection of paths. If each class of curve provides these, then the code would be the same in every case. This change might make sense even as things stand now.

In fact, this makes a lot of sense. Define line and arc objects that specify the path in machine coordinates, and store these in the buffer when doing circular interpolation. This might simplify the code quite a bit.

See below. The distinction made above between the case where the curves meet at an obtuse angle or an acute angle is not important. All that matters is whether the offset curves intersect or not.

22 June 11, 2009 – Version `03` Freeze

Because I decided to reorganize things around the idea given above of using classes for different types of motion, I created version `04`. The code was getting ratty anyway with a lot of stuff commented out that I wasn't quite ready to erase. Version `03` has everything except cutter comp and subprograms, although

it's probably buggy.

This was a very good idea. It made the code much neater, and I could probably have made it even neater. Nearly all moves, both linear and circular interpolation moves, could be treated with the same code. The fly in the ointment is polar coordinates. I could probably find a way to fold that case into one of the others, but it's not worth the trouble, and it might come back to bite me.

The big savings is the way that I can now treat cutter comp. Given two curves which have been adjusted for cutter comp, ask whether these curves intersect (at a point near the end-points of the curves). If they do intersect, then cut to the intersection point and that's it. If they do not intersect, then cut to the end of the first curve and "pivot" to move the tool to the start of the next curve.

I have chosen to test for intersection using analytical methods. If I ever extend to allow arbitrary curves, then one way to do this would be to find intersections using function minimization. To do that, each curve class needs to have some parametric function of the form $f(t)$, $t \in [0,1]$.

22.1 Line-Line

The intersection of two line segments can be found as follows. Let the two lines by given by (x_1, y_1, z_1) to (x_2, y_2, z_2) and (x_3, y_3, z_3) to (x_4, y_4, z_4). The two lines can then be parameterized by

$$\begin{aligned} L_{12} &: (x,y,z) = (x_1,y_1,z_1) + s(x_2-x_1, y_2-y_1, z_2-z_1) \\ L_{34} &: (x,y,z) = (x_3,y_3,z_3) + t(x_4-x_3, y_4-y_3, z_4-z_3), \end{aligned}$$

where $s, t \in [0,1]$. We must find the values s and t at which these equation are equal. There are three equations in three unknowns, so there might not be a solution. Ignore the z-coordinate for the moment since the third equation may be "extra." Then the x-equation gives

$$s = [x_3 - x_1 + t(x_4 - x_3)]/(x_2 - x_1).$$

Plugging this into the y-equation then gives

$$\begin{aligned} y_1 + s(y_2 - y_1) &= y_3 + t(y_4 - y_3) \\ y_1 + [x_3 - x_1 + t(x_4 - x_3)][(y_2 - y_1)/(x_2 - x_1) &= y_3 + t(y_4 - y_3) \\ y_1(x_2 - x_1) + [x_3 - x_1 + t(x_4 - x_3)](y_2 - y_1) &= y_3(x_2 - x_1) + t(y_4 - y_3)(x_2 - x_1) \\ y_1(x_2 - x_1) + (x_3 - x_1)(y_2 - y_1) + t(x_4 - x_3)(y_2 - y_1) &= y_3(x_2 - x_1) + t(y_4 - y_3)(x_2 - x_1) \\ y_1(x_2 - x_1) + (x_3 - x_1)(y_2 - y_1) - y_3(x_2 - x_1) &= t(y_4 - y_3)(x_2 - x_1) - t(x_4 - x_3)(y_2 - y_1) \\ \frac{y_1(x_2 - x_1) + (x_3 - x_1)(y_2 - y_1) - y_3(x_2 - x_1)}{(y_4 - y_3)(x_2 - x_1) - (x_4 - x_3)(y_2 - y_1)} &= t \\ \frac{(y_1 - y_3)(x_2 - x_1) + (x_3 - x_1)(y_2 - y_1)}{(y_4 - y_3)(x_2 - x_1) - (x_4 - x_3)(y_2 - y_1)} &= t \end{aligned}$$

Stick this back into the equation for s to get

$$\begin{aligned} s &= [x_3 - x_1 + t(x_4 - x_3)]/(x_2 - x_1) \\ &= \frac{x_3 - x_1}{x_2 - x_1} + t\frac{x_4 - x_3}{x_2 - x_1} \\ &= \frac{x_3 - x_1}{x_2 - x_1} + \frac{(y_1 - y_3)(x_2 - x_1) + (x_3 - x_1)(y_2 - y_1)}{(y_4 - y_3)(x_2 - x_1) - (x_4 - x_3)(y_2 - y_1)} \times \frac{x_4 - x_3}{x_2 - x_1} \end{aligned}$$

I could grind through this to find s, but it would take forever. Another way to get s is to restart the solution process with

$$t = [x_1 - x_3 + s(x_2 - x_1)]/(x_4 - x_3).$$

Now plug this into the first equation, just as before, to get

$$s = \frac{(y_1 - y_3)(x_4 - x_3) + (x_3 - x_1)(y_4 - y_3)}{(y_4 - y_3)(x_2 - x_1) - (x_4 - x_3)(y_2 - y_1)}$$

(the point is that we have symmetry). It seems like there should be a way to do this with determinants too, but I haven't thought very hard about it.

For these values of s and t to give an intersection point, they must also satisfy the third equation in z. In addition, for this solution to be on the line segment, we must have $s, t \in [0, 1]$.

Consider the degenerate case where

$$(y_4 - y_3)(x_2 - x_1) - (x_4 - x_3)(y_2 - y_1) = 0.$$

This is equivalent to

$$\frac{y_4 - y_3}{x_4 - x_3} = \frac{y_2 - y_1}{x_2 - x_1},$$

which means that the two line segments have the same slope; they're either parallel or the two segements are part of the same line. Adding the z-coordinate really doens't change much about this degenerate case.

22.2 Line-Arc

Next up is calculating the intersection of a line segment and an arc. I might be able to do it as above by looking directly at the parameterizations, but that's messy. First, consider the problem of finding the intersection of an infinite line and a circle.

To make it a bit simpler, let the circle be centered at $(0, 0)$, and have radius r, and let the line be defined by (x_1, y_1) and (x_2, y_2). I got a solution from the internet, but I haven't verified it. Set

$$\begin{aligned} d_x &= x_2 - x_1 \\ d_y &= y_2 - y_1 \\ d_r &= sqrt(d_x^2 + d_y^2) \\ D &= |x_1 x_2; y_1 y_2| = x_1 y_2 - x_2 y_1. \end{aligned}$$

Then the points of intersection are given by

$$\begin{aligned} x &= \frac{Dd_y \pm sgn(d_y)d_x\sqrt{r^2d_r^2 - D^2}}{d_r^2} \\ y &= \frac{-Dd_x \pm |d_y|\sqrt{r^2d_r^2 - D^2}}{d_r^2}, \end{aligned}$$

where $sgn(x)$ is defined to be -1 for $x < 0$ and $+1$ otherwise. The discriminant $\Delta = r^2d_r^2 - D^2$ determines the incidence of the line and circle, as follows. If $\Delta < 0$, then there is no intersection; if $\Delta = 0$, then the line and circle are tangent at a single point; if $\Delta > 0$, then there are two points of intersection.

I have not verified any of this because it doesn't quite get me what I want. I need to know whether a line *segment* and an *arc* of a circle intersect. I think that I might be able to tackle this thinking parametrically using the following identity:

$$a\sin x + b\cos x = \sqrt{a^2 + b^2}\sin(x + \phi),$$

where

$$\phi = \begin{cases} \arcsin(\frac{b}{\sqrt{a^2+b^2}}), & \text{if } a \geq 0 \\ \pi - \arcsin(\frac{b}{\sqrt{a^2+b^2}}), & \text{if } a < 0 \end{cases}$$

I haven't verified this identity; I got it off of Wikipedia under "trig identity." Wikipedia also says that

$$a\cos x + b\sin x = \sqrt{a^2 + b^2}\cos(x - \text{atan2}(b, a)).$$

I did plug this one into Excel and play and it seems to work, but I haven't proved it. Apply it to the parametric equations. Neglecting the z-coordinate and assuming that the circle is centered at $(0, 0)$, the two equations are

$$\begin{aligned} (x, y) &= r[\cos(a_1 + \delta s), \sin(a_1 + \delta s)] \\ (x, y) &= (x_1, y_1) + t(x_2 - x_1, y_2 - y_1), \end{aligned}$$

where δ is a constant that depends on a_1, a_2 and whether the arc is CW or CCW. From the first equation in x, we must have

$$t = \frac{r\cos(a_1 + \delta s) - x_1}{x_2 - x_1}.$$

Plug this into the equation for y to get

$$\begin{aligned} r\sin(a_1 + \delta s) &= y_1 + t(y_2 - y_1) \\ r\sin(a_1 + \delta s) &= y_1 + \frac{r\cos(a_1 + \delta s) - x_1}{x_2 - x_1}(y_2 - y_1) \\ r(x_2 - x_1)\sin(a_1 + \delta s) &= y_1(x_2 - x_1) + (r\cos(a_1 + \delta s) - x_1)(y_2 - y_1) \\ r(x_2 - x_1)\sin(a_1 + \delta s) &= x_2y_1 - x_1y_1 + r(y_2 - y_1)\cos(a_1 + \delta s) - x_1y_2 + x_1y_1 \\ r(x_2 - x_1)\sin(a_1 + \delta s) &= x_2y_1 - x_1y_2 + r(y_2 - y_1)\cos(a_1 + \delta s) \\ x_2y_1 - x_1y_2 &= r[(x_2 - x_1)\sin(a_1 + \delta s) + (y_1 - y_2)\cos(a_1 + \delta s)] \\ x_2y_1 - x_1y_2 &= r\sqrt{(x_2 - x_1)^2 + (y_1 - y_2)^2}\sin(a_1 + \delta s + \phi), \end{aligned}$$

where ϕ is defined as above. Now

$$\sin(a_1 + \delta s + \phi) = \frac{x_2 y_1 - x_1 y_2}{r\sqrt{(x_2 - x_1)^2 + (y_1 - y_2)^2}},$$

or

$$a_1 + \delta s + \phi = \arcsin\left(\frac{x_2 y_1 - x_1 y_2}{r\sqrt{(x_2 - x_1)^2 + (y_1 - y_2)^2}}\right).$$

I can solve this for s, then substitute back and get t (or I could get an expression for t directly using symmetry).

The second trig identity that uses atan2() is nicer. Going back to the derivation above and using this other identity, we have

$$\begin{aligned} x_2 y_1 - x_1 y_2 &= r[(x_2 - x_1)\sin(a_1 + \delta s) + (y_1 - y_2)\cos(a_1 + \delta s)] \\ x_2 y_1 - x_1 y_2 &= r\sqrt{(x_2 - x_1)^2 + (y_1 - y_2)^2}\cos(a_1 + \delta s - \text{atan2}(x_2 - x_1, y_1 - y_2)), \end{aligned}$$

which implies that

$$a_1 + \delta s - \text{atan2}(x_2 - x_1, y_1 - y_2) = \arccos\left(\frac{x_2 y_1 - x_1 y_2}{r\sqrt{(x_2 - x_1)^2 + (y_1 - y_2)^2}}\right).$$

This isn't exactly a thing of beauty, but it can be solved for s, which leads to a value for t, and it's a straightforward formula without the special cases that arise when it's handled using the other trig identity.

Above, I assumed that the circle is centered at $(0, 0)$. Relax that assumption, and assume that the circle is centered at (c_x, c_y). Then, by rehashing the derivation above, we get

$$t = \frac{c_x + r\cos(a_1 + \delta s) - x_1}{x_2 - x_1},$$

and

$$a_1 + \delta s - \text{atan2}(x_2 - x_1, y_1 - y_2) = \arccos\left(\frac{x_2 y_1 - x_1 y_2 + c_x(y_2 - y_1) - c_y(x_2 - x_1)}{r\sqrt{(x_2 - x_1)^2 + (y_1 - y_2)^2}}\right).$$

22.3 Arc-Arc

The last case is arc, then arc. This one is horrible to do analytically. I could find the intersection of two circles using rectangular coordinates, then check whether these two points are on the actual arcs. Working directly with the parametric equations seems like a nightmare. Even using rectangular coordinates is a real mess.

Go back to the idea of trying to determine intersection by numerical approximation.

22.4 Working Numerically

Suppose the we have two curves specified by parametric equations:

$$f(s) = (f_x(s), f_y(s)) \qquad \text{and} \qquad g(t) = (g_x(t), g_y(t)).$$

Given points on the two curves, the distance between them is

$$d(s,t) = \left[(f_x(s) - g_x(t))^2 + (f_y(s) - g_y(t))^2\right]^{1/2}$$

and the squared distance is

$$D(s,t) = (f_x(s) - g_x(t))^2 + (f_y(s) - g_y(t))^2.$$

We want to find s and t such that $D(s,t) = 0$. Moreover, this function is everywhere non-negative, so $D(s,t) = 0$ implies that dD/dt and dD/ds are both zero at (s,t).

For relatively tame curves, it should be easy to find a solution. If the curves are "tame," then there should be at most one point of intersection near either end. In fact, for the problem at hand, I only care about intersections that are within one tool diameter (or radius?) of the end of the curves.

Here's one way to do this. Fix t_0 to be the end-point of one of the curves. Find the point on the other curve that's as close as possible to t_0 and call it s_0. Now go the other way and find t_1 on the first curve that's closest to s_0, etc. I think that this will walk to the intersection in very short order. If there is no intersection, then the algorithm will stall.

Finding the closest point should be feasible, especially if I have the derivative of the parameterizations, although it is another numerical loop. In some cases, I could find the closest point analytically. In fact, I think that for the two cases I care about (line segments and arcs) I could do it analytically. The key fact is that the curves are tame and don't have too many bends. The reality is that it's probably not worth the trouble to do this analytically. The numerical method will be fast enough.

23 June 13, 2009

I finished the cutter comp code. I haven't tested it at all, and there are probably some serious bugs – and I don't deal with the z-coordinate either. But it's basically done and ready to test. Assuming that the numerical method of finding intersections works, it probably makes sense to shift to that method entirely and avoid the analytic methods. Doing this should allow me to combine things and make the code tidier.

There are two aspects to testing this code. Obviously, it shouldn't crash and the output should look reasonable. Much fussier is checking to be sure that the cutter comp offset works *exactly* right. I'm not sure how to do that. Ideally, I need some tricky G-code that makes objects that will tell me immediately, on visual inspection, whether my program is right or not.

Cutter comp is a royal pain. A better way to have organized this may have been to use two layers of filters. The first layer takes the G-code and translates it to new G-code by taking out any cutter comp command, but adjusting the coordinates for any G-code to take cutter comp into account. The second layer is pretty much what I have now, except that there's no such thing as cutter comp. Let $L0$ be the lowest layer and $L1$ be the final layer. Certain commands could be filtered out of the lowest layer, like turning coolant on and off, which is nice, but very minor.

I like this idea, but it might cause other problems. On problem is calling subroutines. The difficulty is that if the program doesn't see a subroutine call until the final layer, then any statement points in lower layers will have to be reset. If $L0$ keeps track of this so that higher layers just keep eating statements without regard to what line we are on, then this problem goes away. In fact, I like the idea of making the lowest layer keep track of subroutine calls and nothing else. This makes sense because it's closely tied to what the lexer is doing.

Other issues or modes are inches/mm, cutter comp, TLO, absolute *vs.* incremental, polar coordinates, `G17/18/19`, work offsets, tool changes and (of course) the moves themselves. I think that we can convert to standard units (inches or mm) at the lowest level. I should be able to handle TLO and work offsets at a low level too. Next (I think) is eliminating cutter comp. It makes sense to leave absolue/incremental mode in the pipeline, and I don't see any way to eliminate polar coordinates (or benefit) if I want to allow funny spiral cuts. With polar coordinates, there is a choice: allow spiral cuts or convert everything to arcs. I don't see any reason to pull `G17/18/19` out of the pipeline early; in fact, I don't see how I could. So, here are the proposed layers:

1. Filter out subroutine calls.
2. Convert all units to the standard: inches or mm.
3. Eliminate TLO and work offsets.
4. Eliminate incremental coordinates so that everything is absolute.
5. Translate cutter comp coordinates.

Some of these might be combined. For instance, eliminating TLO, work offsets and incremental mode all involve translating the coordinates from one system to another. I also think that I was wrong to make polar coordinates draw in spirals. If it's true that all moves in polar coordinates are straight lines, then eliminating polar coordinates is another step. As a side benefit it should be easier to determine whether cutter comp works right. I could ouput the G-code at each layer to see whether it's being translated correctly.

I will also take this opportunity to convert the machine's internal coordinates to integer multiples of the scale. I need to do this to be absolutely sure that no rounding error builds up from things like arcs of circles. The downside to doing this is that if the user uses a scale of 10 steps to the inch, then rounding error could build up, especially if there's a lot of incremental code. I suppose that

the solution is to have two "scales"; one scale determines the internal accuracy of the machine (the number of pulses per inch) and the other scale determines the resolution of the material being cut. If I do that, then there's very little reason to reduce the scale in the machine. It might speed up the conversion from G-code to solid a bit, but that's minor.

On further reflection, I don't think that the previous idea is good. A `double` object has around 15 digits of accuracy. The chances of any rounding error cropping up are tiny, and I don't see how using whole number multiples of the machine resolution would really improve matters. In theory there is some value to rounding the final point of every arc to the nearest 0.001 inch to nip the accumulation of rounding error in the bud, but the only way this could be an issue is if the user input millions of arcs at weird angles, and the error would still be only 0.001 inch or so.

Organizing the interpreter as a series of layers was a great idea. The code is longer and more repetative, but it almost wrote itself. Cutter comp is still a pain, but by striping out as much as possible before cutter comp is tackled, the job is much easier.

Something else that I probably should have done is organized the `Statement` class to include the features of `ToolCurve`. When I get to the point of handling cutter comp, there's a certain amount of translating back and forth between `Statement` objects and `ToolCurve` objects. I'm not convinced that combining the two is the right thing to do (especially at this stage of the game), but it's worth thinking about. Moreover, linear moves and arcs should both be considered a single type in `Statement`. The feature that distinguishes this type is that the cutter actually moves. Most other types don't have that effect, or have it only as a by-product (e.g., tool length offset).

A minor improvement to the interpreter that I could make is the way that I handle errors. Right now, there are lots of lines like

```
throw new Exception(formError(cmd.lineNumber,"error msg"));
```

and that works, but it would be better if `formError` just threw the exception. If the `Interpreter` class also had a field for the current line number, then errors could be handled with a much simpler call of the form

```
error("error msg");
```

24 June 17, 2009

I finished coding *all* of the G-code interpreter (or all that I want to do at this point), including handling `G17/18/19` correctly. No doubt there are bugs galore, but it's all there. This is version `05`. The only major thing that I left out are the canned cycles. I ought to be able to deal with those by translating them to standard statements.

I also added the ability to see what's happening at each layer of the interpreter. This is useful for debugging, but it would also be helpful when teaching. Most things seem to work. It's cutter comp which seems to have the bulk of

the bugs, and that's no surprise.

The system is currently set up so that the tool radius is hard-coded (at a radius of 10 units) so that there's a disconnect between the cutter comp radius offset and the actual tool radius. This turns out to be a great thing for debugging. If the cut is only 20 units wide while the cutter comp is done as though the tool is 250 units in diameter, then I can really see the path of the tool.

25 June 19, 2009

Most things seem to work, but I've been focused on the cutter comp routines, which definitely do not work. The crux is finding the intersection of two curves. I'm sure that this can be done numerically, and in the world of numerical analysis it's not a very hard problem, but it's still tricky. In fact, this is the sort of problem that makes for a good case-study. See `ToolCurve.intersect`.

I tried bouncing between the two curves, much as a ball would do. The ball does converge to the intersection point, but it happens far too slowly when the curves intersect at anything other than a large angle. You can see what happens by using the `IntTester`. This problem is obvious in hindsight.

I discuss various ways to deal with this problem in a long comment at the start of `ToolCurve.intersect()`. The real fly in the ointment is the fact that either of the two curves (or both) might not lie in a plane. A helix does not lie in a plane. I'm sure that I could solve the intersection problem using some brute force method involving tables, but most promising idea that's not brutal is discussed below.

The ball bounce method does walk toward the intersection (or the closest point of approach), but it does so too slowly. Let $f(s)$ and $g(t)$ be parameterizations of the two curves. Assume that the bouncing ball gives points $f(s_1), f(s_2), g(t_1)$ and $g(t_2)$, along with distances d_{s1}, d_{s2}, d_{t1} and d_{t2}, where d_{xi} is the distance from a given point on the curve to the closest point on the other curve. This gives two functions, D_f and D_g, where $D_f(s)$ is a linear approximation of the distance from $f(s)$ to the nearest point on g, and $D_g(t)$ is a linear approximation of the distance from $g(t)$ to the nearest point on f. Solve $D_f(s) = 0$ and $D_g(t) = 0$ to find s_0 and t_0. To the extent that the approximations are accurate, the intersection point should occur at $f(s_0) = g(t_0)$. Start the bouncing ball again at these two points and repeat the process. As a way of taming the algorithm, it might make sense to start the bouncing ball at a point not quite all the way to s_0 or t_0.

A problem with this is how it will handle curves that don't intersect. As you near the point where the curves are close, large moves in s and t will give very small changes in the distance from one curve to the other and solving D_f and D_g for zero will give parameters that are way (!!) out of the ballpark since these functions have no zero point.

Here's another idea. My difficulty is that in three dimensions there's nothing like the mean value theorem. I can't say for sure that the distance between the

curves ever crosses a zero value. Put another way, there is no mean value theorem for functions that take two variables, like $D(s,t)$. The bouncing ball definitely walks toward the intersection, but it does so too slowly. I could walk a bit from one side of the intersection, just to be sure that I'm walking in the right direction. Then try to walk from the other side of the intersection point. I'll know that I really am on the "other side" because, if s and t were increasing/decreasing on the first side of the intersection, then they should do the opposite on the other side.

Maybe I should just bite the bullet and use something like Powell's method or the downhill simplex method on the distance as a function of the two curves' parameters: minimize $D(s,t)$. The chief reason that I have not is that these methods do not naturally lend themselves to bounding the area over which to search for a solution. I implemented the simplex method (amoeba from NR), which has the same geneneral flavor as the idea discussed above, and it works.

I implemented the downhill simplex method (amoeba) and it works (kind of), but I think that it's too sensitive to the characteristic length. I could play around and hope that I've found a way to use this method that won't behave oddly, but I need to be absolutely certain that it will *never* behave oddly in *any* case and I don't think that I'll ever have that level of confidence with the downhill simplex method.

Another thing that I realized is that I don't really want to take the intersection of the two tool curves in three-dimensional space. If there is a helical curve with varying Z followed by a line segment, then the curves we want to intersect are the projetion of the helix to an arc of a circle and the projected line segment. This makes the problem slightly more tractable, but not much.

I ought to be able to do this as two one-dimensional problems. Define $D(s,t)$ to be the distance between $f(s)$ and $g(t)$, where f and g are parameterizations of curves F and G. Define $C(s)$ to give the value t for which $g(t)$ is closest to $f(s)$; C finds the point on G that is closest to $f(s)$ on F. Calculating C is an easy one-dimensional problem that I've already solved in `ToolCurve.findClosest()`. I don't care about most of the domain of $D(s,t)$; for any s, the real test is whether $C(s)$ gives a point t so that $g(t)$ is close to $f(s)$. In other words, it suffices to minimize $\tilde{D}(s) = D(s, C(s))$. For any point $f(s)$, on F, $\tilde{D}(s)$ is the distance from $f(s)$ to the curve G.

I care more about the certainty of finding a solution than about the speed. Numerical Recipes (NR) has lots of clever methods, but I need to think long and hard about whether I should trust them.

I implemented the method of minimizing $\tilde{D}(s)$. It's still very buggy, but it made me realize that I haven't fully understood the problem. It's possible to come up with all sort of weird situations. Imagine two arcs with a common endpoint, but whose radii are only slightly different. If things like bezier curves are permitted, then you can get even more complicated situations. Any algorithm that searchs for an intersection must be careful to take the correct intersection point. I could *require* that there be only one point of intersection, but I would rather not.

There are two main cases. Consider the point where the original (not offset

for cutter comp) curves meet. The angle at which they meet is either less than π or more than π. If the angle is more than π, then the offset curves are guaranteed not to have an intersection; if the angle is less than π, then they definitely will intersect. Determining which of these two cases applies is easy if we know the tangent vector of the curve at the end-points.

Suppose that the angle is less than π. The offset curves will intersect, and I want the intersection that is "closest" to the end-points. One problem is figuring out what it means to be closest.

Actually, even if the angle is less than π, there is no guarantee that the offset curves will intersect. Think of a line followed by an arc, where the arc doubles back on the line and continues some distance. There is a small slice off the side of a circle between the line and the arc. Imagine that `G41/2` is set so that the tool is on the same side as this slice and that the tool has a large diameter. Then the two offset curves do not intersect.

Moreover, if the angle is more than π, then there no guarantee that the curves will *not* intersect. If the tool is large enough then it could cut into some portion of the following curve that's some distance from the place where the two non-offset curves meet. This example is kind of bogus, but it could happen with things like bezier curves. Even so, I think it's reasonable to *assume* that curves that meet at an angle of more than π do not intersect. In very rare cases, you might get odd behavior, but I don't think you'd get anything totally haywire. Some (all?) of these odd situations would disappear if cutter comp were permitted only when the radius of curvature of the curve is larger than the cutter diameter.

If I want to take a really brute force approach, then I could literally trace the curves. Unlike the Bresenham algorithm, the curves that the tool follows are "fat"; under Bresenham, pixel squares may meet at the corners, but the pixel squares of a tool's path must always meet along sides. Under Bresenham, you can have two lines that intersect in an abstract sense, but have no pixel in common; that's not possible with the way that I generate tool curves. The reason it's not possible is that only one axis of the machine can move at a time. The brute force line-tracing method is to draw each curve to a 2D array of booleans, then check for intersection. In practice, I would draw the first curve to the array, then follow the path of the second curve, checking whether any of the positions of the second curve were visited by the first curve. The answer is the first such point that is found. This method isn't elegant, but it's not that bad. The biggest issue is the memory required since booleans in Java don't take a single bit. I could use an array of `BitSet` objects, where each `BitSet` is a row of the raster. Even that isn't the greatest solution. If the two curves require a square 16 inches on a side, then I need $(2^{14})^2 = 2^{28}$ bits, or 2^{25} bytes, which is 32 megabytes. That's doable, but merely allocating that much memory would be slow. The other strategy is to list the coordinates as an array of pairs. The problem there is that searching to determine whether a particular pair appears on the list could be slow. A quadtree might be a compromise. Another idea that has a similar flavor as quadtrees is to draw the curves with at a low resolution (say that the tool moves in steps of 0.10 inches), find the intersection,

then zoom in on the square in question. Or, the curves could be broken into segements; determine a bounding box for each segment, then zoom in on any pair of bounding boxes that intersect. All of these seem like a lot of trouble.

Since the curves whose intersection to be found are projected to a plane (due to `G17/18/19`), I might be able to define a signed distance. The problem is that you need to know the intersection point first, so that we know the tangent vector to use to define the sign of the distance. Provided that the curves are tame enough, it might be OK to let this tangent vector converge to more accurate values as the intersection point is more tightly bracketed, but that seems unlikely to work.

Another observation that may be useful is that we can use the arc-length to avoid searching at too fine a level. If a given point on curve A is more than the tool diameter away from curve B, then the two curves can't intersect until we move at least the tool radius away from that point on A.

This question is much more complicated than it seems. I need a battery of test curves and a program tailored to looking at how my algorithm(s) handle each of these tests.

For reasons of precision, I think that the *best* solution might be to convert the statements into a set of tool signals (pulses) while they are still being buffered, after they have been offset, but before any intersection has been found. Then use the basic idea of a direct search for a point of intersection by comparing coordinates. If such a point is found, then trim from each set of tool pulses. The direct search doesn't have to be painfully slow if I use the idea of working my way to higher and higher resolutions. That is, look for two large squares (say at a resolution of 0.25 inches) that the curves have in common; if such is found, then look inside for a common square at a resolution of 0.050 inches, then for a common square at 0.001 inches. This would require some changes to `Layer05`. Ideally this would be done in such a way that the set of pulses would not need to be recalculated at the final layer.

26 June 23, 2009

I've wasted a lot of time on the cutter comp problem. I just want something that works dependably, even if it's not the most elegant. I need something that works so that I can test the system as a whole.

The idea of looking for an intersection point by tracing along the tool paths is crude, but easy to understand and effective. I can be a bit more sophisticated than that (and avoid the need for so much memory). Break the curves into a series of boxes that bound a particular range. Let $A(s_i)$ be a series of points along the curve A. Define $R(A, i)$ to be a box that bounds all of the curve between $A(s_i)$ and $A(s_{i+1})$. If we have two curves, A and B, and a set of bounding boxes for each curve, then it's simple matter to see if there is any pair of boxes $R(A, i)$ and $R(B, j)$ which intersect. Zoom in on any such box, subdivide more finely and do it again. This is naive and inelegant, but it works.

First, note that the degree of subdivision at each step should not be too

large. If the two parameterizations are each divided into n sub-parts, then, to determine which pairs of boxes require further attention, I need to do n^2 box intersection tests, but I only get a $1/n$ improvement in accuracy. For example, if I use binary subdivision ($n = 2$), then I get one more binary digit for every four box comparisons (plus bookkeeping overhead). If I use $n = 4$, then I must do 16 comparisons, but I only get two more binary digits. There's some balancing point due to the overhead, but I don't want to do anything like $n = 1,000$.

Simply evaluating $A(s_1) = (x_1, y_1)$ and $A(s_2) = (x_2, y_2)$ is not enough to determine a bounding box; the curve could do anything between s_1 and s_2. One way to handle this would be to *assume* that the curve can be accurately approximated by a third degree polynomial. Consider the x-coordinate. We know the tangent vector at each end of the curve; let these be $(u_1, v_1) = A'(s_1)$ and $(u_2, v_2) = A'(s_2)$. Assume that $x'(t)$ takes the form

$$x'(t) = at^2 + bt + c.$$

Integrate this equation and apply the conditions $x(s_1) = x_1$, $x(s_2) = x_2$, $x'(s_1) = u_1$ and $x'(s_2) = u_2$ to determine a, b, c and the constant of integration. Now we can find the largest and smallest x-values that are possible between s_1 and s_2 to ensure that the bounding box is large enough. This would work, provided that the curve conforms to these assumptions. The resulting equation is ugly, and there's a better way below.

An important observation is that the radius of curvature of the curves must be greater than the radius of the tool. This can be used to determine the bounding boxes. Recall that the radius of curvature at parameter t is defined to be $r = 1/k$, where $k(t) = |A''(t)|$. So, to determine the radius of curvature, we need the second derivative of the parameterization, but we can easily get that. Note, however, this this definition requires that the parameterization be given by arc-length. This is equivalent to choosing the parameterization so that $A'(s) = 1$ for all s. I should be able to arrange that.

Suppose that $s_2 > s_1$. Because the radius of curvature of A must be more than r, the extremes of A between s_1 and s_2 should be limited. See DoCarmo's *Differential Geometry*, p. 24, problem #9. It says that given a plane curve (which is what I have) with radius of curvature given by $k(s)$, then the curve must be given by the equation

$$\alpha(s) = \left(\int \cos\theta(s) ds + a, \int \sin\theta(s) ds + b \right),$$

where

$$\theta(s) = \int k(s) ds + \phi,$$

for some a, b, ϕ; (a, b) allows for translation of the curve and ϕ is a rotation angle. The way this is stated is strange; the definition of $\theta(s)$ doesn't really make sense; s can't be both an argument to θ and an argument to k. The intended statement is $\theta(s) = \int_a^s k(t) dt + \phi$ (where a in this context refers to the

domain of the parameterization; $\alpha : (a, b) \to \mathbb{R}^2$), and similarly for $\alpha(s)$. In any case, this isn't too hard to prove.

I need to determine the possible extremes of $\alpha(s)$ (which is the same as $A(s)$) in the range from s_1 to s_2. Because I am interested in the worst case (the wildest possible curve), I would like to assume that $k(s) = 1/r$ everywhere. I can't do that because a curve with a constant radius of curvature is a circle (or a line). However, I do think that the curve has to be inside the "lens" formed by two arcs of radius r. Put another way, if the curve goes outside this lens, then it must have some point at which the radius of curvature is less than r; otherwise, how could it get back to the known end-point?

So, the bounding box for the curve between $A(s_1) = (x_1, y_1)$ and $A(s_2) = (x_2, y_2)$ is the bounding box of the lens with these end-points. Finding these arcs is the same as a previous previous problem (see discussion for June 8): we have two end-points of an arc and the radius. The gist is that

$$q = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

is the distance between the end-points of the arc,

$$m = (1/2)(x_1 + x_2, y_1 + y_2)$$

is the mid-point of the line connecting the two end-points. Set

$$v = (x_1 - x_2, y_1 - y_2)$$

and

$$v_\perp = (y_2 - y_1, x_1 - x_2).$$

The distance from m to either center is

$$\sqrt{r^2 - (q/2)^2}$$

and the two centers are given by

$$m \pm (v_\perp/q)\sqrt{r^2 - (q/2)^2}.$$

What I need is the width of the fattest part of the lens. This is

$$2\left(r - \sqrt{r^2 - (q/2)^2}\right).$$

Any bounding box must be at least as wide and as tall as this value. If the box is already squarish, so that the lens is on a diagonal, then no adjustment is needed. If the box is nearly horizontal (vertical), then it needs to be made taller (wider).

The argument above only applies in situations where the arc-length of the curve in the bounding box is less than the radius of the tool. When the box is larger, then the curve could have a greater extreme. I think that I could use DoCarmo's statement to handle the case when the boxes are larger than the tool diameter, but it's not worth the trouble for now.

I implemented this bounding box algorithm and it seems to work, but there are other bugs. I think (?) that they they come down to the fact that once you apply cutter comp and you need to trim paths where they intersect, you may change the paths in a way that it inconsistent.

27 June 25, 2009

The tricky parts seem to work. No doubt there are bugs, but they aren't obvious at this point. It's time to bring this to a form where people can try it out.

28 June 27, 2009 – Version `05` Freeze

The ability to do tool changes was the last significant improvement. Then I made a few small changes to make the program more user-friendly, and jared it up. **This is frozen as version `05`.** Charlie Whorton has a copy, and I'm going to have lunch with Brian Barker (of Mach3) to talk about it. From now on, I will work in version `06`.

From here, there are several directions that I could take, listed below. See also my comments of May 27 for a detailed list of improvements.

- Spend time really thinking through the possibility of built-up rounding error, especially in the context of cutter comp and incremental coordinates.
- Macro-B to G-code converter. In many ways this is entirely independent of everything else. I think that the way to do this is to use ANTLR (or something similar) and replace the existing lexer and parser with the result. This might be a good thing to do, even if I never implement Macro-B.
- User interface stuff. There's a host of things I could do here to make the interface look and behave nicer, along with additional functionality to allow the user greater control over what's happening internally.
- Allow drills and ballmills in the tool table. This wouldn't be very hard, but it might make sense to wait until I've fixed on a framework for converting pulses to a 3D object.
- Allow the user to choose a single line of code and see what it cuts. Animating the cutting process is similar. With threading, I could allow the user to start and stop the cutting, like a movie.
- Improve the drawing method. I'm not sure exactly what to do here, but both the shading and z-buffer need to be improved (at least I think it's lack of resolution in the z-buffer that causes some quirks). Speed could be improved, but the larger issue is making it look nicer.
- Speed up conversion of the pulses to a 3D object. This would make a very noticable difference to the user, and it would be natural to do this when I move away from using a DA to represent the object.
- Allow undercutting. This absolutely requires that I move away from using a DA, and it must be done before I can allow the user to use tools other than endmills, ballmills and drills.
- Although the G-code to pulses part of the program has no serious structural problems, it could be re-organized to be more elegant. For one thing, the distinction between `Statement` objects and `ToolCurve` objects could be eliminated.

Now that I've released the program on a limited basis (only to Charlie so far), I need to worry about versioning. I really don't like cvs, even with the various scripts that I wrote for Overlook. Right now, it's not such a big deal. The only errors that I will try to fix are ones found in the G-code interpreter, and I don't intend to change that in any significant way for a while. Versions `06` and `05` can share that code. This is every package in the project, except for `da3ddisplay` and `flatimage`. I should be able to fix bugs in the other packages and the result should be valid in both versions `05` and `06`. I can avoid this whole issue by creating two new menu choices under "Render" and leave the old ones alone. That way I can work on new object representation and rendering code without making any changes to the older stuff.

The existing drawing code converts each row of the DA to a vertex-defined slice before drawing it. So, converting the pulse-to-object part of the code to use slices based on vertices wouldn't have too many external implications. It makes sense to do that next.

I've decided that only two approaches of slicing are still in the running: polygon slices, with explicit vertices, and walking slices, where there are three possible choice for each step. These two methods are almost identical in the abstract; the difference is in their implementation. I discussed the pros and cons above, particularly on May 29 and May 18.

As noted above, determining which slices may be affected by a particular tool position is easy – the slices are in one-to-one correspondence with the coordinates. The problem is figuring out where along each curve the tool might change the perimeter. Due to things like undercutting, each tool (or category of tool) will have to be handled differently. E.g., endmills can't undercut on a 3-axis machine, but an ogee bit can. On a four axis machine, all bets are off, but for endmills, drills and ballmills on a 3-axis machine, I think that this could be made quite fast using the idea of sorting tool coordinates (May 29).

29 To-do List

Things that need to be done. I will modify this as things move along.

- How to deal with tool length and setting the value of H given that this is an imaginary machine.
- Make the tool turret dialog fancier. To begin with, I want a popup menu to let the user choose the tool, and I'd like a picture of the tool. Eventually, I want to specify the actual length of the tool. It should be possible to specify a different tool diameter and cutter comp value too. Make this savable. I'll want the ability to define tools with an odd outline and this requires a simple drawing program.
- Allow drills and ballmills. These would be realtively easy to add.
- Use a different method of representing the object internally. It needs to be faster and take less memory.

- Dialog to let the user choose the size and shape of the starting material. The dialog exists, but it's crude and it doesn't work. I want the ability to save an object and load it as the starting point.
- Ability to run the tool path as an animation.
- Macro-B to G-code translator.
- Let the user use the mouse to rotate the image.
- Triangulate the surface of the object assuming different scales, and use these triangles when drawing. That should be faster. When the user is not zoomed in, there's no reason to use such small triangles. This might be a way to speed up the rendering. Draw the object using the largest triangles, then progressively smaller ones. In fact, it might make sense to delete the slice representation from memory and just hold the triangle representation. Most of the time (?) it will take less memory anyway, and rendering would certainly be faster since it works directly from triangles. A simpler strategy that I could implement more quicily is to do what I'm doing now, but choose the frequency of sampling dynamically.
- Figure out why there are jittery pixels. Is it low resolution of the z-buffer?
- Come up with a better shading method. There are lots of methods, and different methods may look better under different circumstances. The user should be able to choose. For instance using different colors (say green and red) for horizontal and vertical surfaces makes some sense for simple objects, but something like a 3D engraving should use smooth grayscale shading. If the user zooms way in, then he should see blocks.
- Give the user some control over the level of detail used when rendering.
- Change the background to light blue (say) so that the user can see the difference between through-holes and deep holes.
- Think about the problem of holes in the side. The obvious thing to do is to draw every slice as a solid slab, but that would be too slow. One solution is to encode the side faces too, but as "open polygons". Through-holes are similar. Their big complexity is that slices may consist of disjoint perimeters.
- Come up with a way to insert rough patches in the polygon representation.
- Improve the polygon drawing code. In particular, don't check whether ever pixel is on the raster. Check that before-hand. Also, fix the bug that occurs when you zoom in too far (the triangle gets too large to deal with).
- Investigate why copying the rendered image takes so long. I mean the call to `Graphics.drawImage()`. This often takes longer than generating the image itself. Getting this down to a speed that's in line with what I would expect would free up enough time to draw another 25,000 triangles.

- Consider making vector rotation faster. I'm not sure how much it will help though; that's not the bottleneck.
- Add sides and bottom to the visual object.
- Allow sliding the object in addition to rotating it. The ability to cut through an object to see a cross-section would be nice. Really nice would be the ability to descend into an object to look around.
- Let the user define the keys to use for rotation (instead of I, J, L, M).
- Allow the user to click on a line of G-code and see what it cuts. Rendering these cuts with alpha (i.e., transparency) is one way to do this visually. To do this, I need to keep the polygon representation in memory (I can't ditch it and have only a list of triangles). I might be able to compress it somehow, though, and put it in deep storage.
- Add collision detection-type things, like whether the spindle is on when the user cuts. Consider the actual dimensions of the tool. If a drill is only three inches long, the user can't try to plunge four inches. I could test the feed rate too if I know the nature of the material.
- Improve the text editor. Add line numbers on the left. Syntax coloring would be nice. Pretty printing?
- Although the G-code to pulses end of the program is done (even if there may be bugs), it could be cleaned up. For instance, I combining the notion of `Statement` and `ToolCurve` makes sense. Errors could be handled better too.
- The ability to see what the differen layers of the G-code translator are doing is nice. Clean that up for use as a teaching tool.
- Spend time really thinking through the possibility of built-up rounding error, especially in the context of cutter comp and incremental coordinates.
- I had lunch with Brian Barker and Scott Nichols of Mach3 on June 29, 2009. They said that a simulator really must have more than three axes. I'm not sure that I agree with them. To some extent they may just want compatibility with their stuff. They don't think that there would be any legal difficulty with writing a Macro-B translator. They agree that Mach3 is a mess internally. Users of Mach3 are *not* at all sophisticated, and are only interested in being able to push the button. They really didn't like the idea of me talking about a card, and tried to tell me that it's already been done, don't go there, etc. To some extent, I think this is because they are coming out with a card of their own, though they are doing it secretly so that competitors don't know. I'm not sure, but I think that Java might be able to talk to the serial port after all. If so, then I could write a program to replace Mach3 pretty easily. The biggest hurdle is

understanding what signals are required and just getting to know how the Java interface works. Other than that, the only technical problem is the fact that motors must be accelerated and decelerated smoothly. Overall, they seemed interested in what I'm doing, but wary.

RANDOM THOUGHTS

These are things that I thought of at random times. They might just repeat what I've already thought of and dismissed.

4/13/2011 A 3D object is a union of cubes and one could quickly determine whether a line intersects the object, especially if the cubes are in a binary tree. You may need to break the total volume up into sub-cubes (conceptually; each of these sub-cubes might be solid, empty or partially full; the point is just to speed up the search). This idea would take a lot of memory.

You could describe the surface by breaking the volume up into slices and consider each slide to be a polygon. This could be stored as a set of coordinates for the vertices, oras a list of vectors (direction and distance). This would take a lot less memory, but I'm not sure about finding intersections. And the cutting would be complicated.

What's needed is something like the use of edges in two dimensions.

30 Notes

- There's a third edition of Numerical Recipes out. It's better than the old edition, though it's mostly the same. The most interesting thing to me is the addition of a chapter on geometric algorithms.
- Write a program that "compiles" Macro-B code into G-code. Macro-B doesn't take any user input so this is possible.