I am looking at writing a G-code simulator again. There are two other documents that have a lot more background, but I want a fresh start. These two documents are found in the `old vcnc` directory. One is an overview of differnt strategies that I have considered and the other is a more detailed look at the steps along the way as I tried different ideas. These are in `notes.tex` and `overview.tex`. There is also `manual.tex`.

1 April 18, 2013

I'm looking at this again for a couple of reasons. One is that I want to learn Qt and refresh my C++ fluency, and this is a relatively easy program to do that. Not only is this a good way to learn (or relearn) this stuff, but I think I could sell this program over the internet – not many copies, but enough to make it worth figuring out how to set up a website, PayPal, and so forth.

My goals are modest for the program. The resulting object may not have any undercutting, the mill can only be a three-axis machine, and the tools must be convex. This simplifies things a great deal.

1.1 The Dream Program

Someday I would like a version of this program without any restrictions at all, but it's too difficult for me to think about right now. Multi-axis machines, undercutting, concave tools, *e*tc. would all be nice, but they're not easy to deal with. If I do eventually go that route, I now think that working in polygons (triangulating the surface) is the way to go. Here are some thoughts about that.

Dividing the volume into voxels of 0.001 requires far too much memory, and I suspect that the amount of processing required would be too great in any case. Octrees are another route, but they take a lot of memory too if they have the potential to get down to 0.001. Also, things like shading the surface would be a royal pain. I looked at these issues in an earlier document. At the time, they seemed surmountable, but I'm less sure of that now.

Something like triangulating the surface now seems to me to be the way to go – memory becomes a non-issue and shading the surface is almost trivial – but I think that it would be too slow if done naively. This approach requires three basic operations: union, intersection and difference of two volumes. Actually, I think I could finesse the need for union and intersection, or at least they wouldn't be needed in their full generality, but the difference operation is the crux of the entire problem. See below: the interesection operation is needed, but it may not be so hard.

I haven't done any research yet, but it seems that everything will need to be based on convex volumes since my intuition is that a point-in-volume routine will always come down to whether a point is in a *c*onvex volume, and this would be done by checking whether the point is on the in-side of every surface polygon that bounds the volume.

The basic problem is how to take a given a convex volume, V, and subtract from it another convex volume, D, to obtain $V - D$. I would start by checking which pgons of D lie entirely within V, which are entirely outside, and which are partly inside and partly outside. From this you could build a surface that is equal to $V \cap D$, and we now want $A = V - (V \cap D) = V - D$. Now I need to know which pgons of V are inside, outside or partway in/out of $V \cap D$. The pgons that bound A will be those of V that do not touch D, plus those of D that are entirely inside V, plus a set of pgons needed to "stitch things together." The later set of pgons is obtained by looking at how D "cuts through" the pgons of V that are not entirely inside or outside of D.

Once we have the bounding pgons for $V - D$, as desribed above, the volume must be broken into pieces, where each piece is convex. This is needed becuase I think that working with convex volumes only is what makes the process above work in successive steps.

All of the above seems conceptually do-able – not trivial, but probably well-understood. However, I will need to be careful that the number of pgons doesn't explode. Suppose that we work with triangles alone (3gons). Each 3gon will take no more than 24 bytes – call it $2^5 = 32$ to make it easier. Then we could store 8 million 3gons in 256 meg of memory. That's enough (I think) for any reasonable surface, provided that the pgons are made as "large as possible." The problem is that it would be slow, both in terms of simple calculations, like point-in-volume, and in terms of rendering.

One way to speed this up would be a very coarse "voxeling" above the level of triangulation/p-gon-ification. If the volume is divided into voxels of 1/16 inch, then a 10 x 10 x 10 inch cube would require $10 \cdot 10 \cdot 10 \cdot 16 \cdot 16 \cdot 16$ $2^{10} \cdot 2^{12}$, or 4 million elements. Each element would be either solid, empty or partial. It would be simple (and fast) to determine whether a given point was in a solid, empty or partial voxel. The solid and empty cases are then finished and dealing with the partial voxel case would be easy too since these voxels are only 64 (1024/16) to a side, and could not involve very many pgons.

This has the additional advantage that all p-gon calculations would be in a very restricted space. This may not matter, but it's tempting to think that it would simplify the algorithms too.

1.2 Back to Reality

The dream program would be nice, but I want to focus on what I can do more easily. The best and simplest strategy in the situation where there is no undercutting seems to be what I called a DA (depth array) in earlier discussions. This is a 2-d array of depth values, with one entry for each 0.001 inch square. Each square inch of surface will then require 1M entries, and at least two bytes will be needed for each entry. Something like a 10 inch square sheet is then well within reason, using 200M (at two bytes per entry). For larger objects, I could implement the "quad tree with depth" (QTD) as I did earlier. For things that are highly detailed, like milling out a person's face, there is no advantage to a QTD since almost every array element will be at a different depth, but for very

large objects that are cut more simply, there is a huge memory savings.

For now, work strictly with a DA. Earlier, to save memory, I used shorts to specify the depth, but I will use floats now, which take four bytes rather than two. The reason is that it will be easier to determine the surface normal if the depth value is highly accurate to within a fraction of a voxel. On the other hand, with two bytes, I still have 64 inches of potential depth variation, but no actual object will every have more than a few inches of difference between the highest and lowest point. So, each whole number could represent a tenth of 0.001.

The reason I care about this is that I will want surface normals when rendering, and the more accruate the depth value, the less concerned I will need to be with weird artifacts near edges, or anywhere that the slope changes.

Another simplification compared to what I was trying to do before is that I will not work strictly from a pulse train. In earlier versions I converted the G-code to a series of pulses in each direction, and worked "pulse by pulse," without any knowledge of the G-code that generated these pulses. That is clean, but less efficient. Instead, I will now work in terms of lines of G-code.

For the current version, I will not worry about the niceties of the user interface, or anything close to "niceties." The goal is to get the algorithmic guts working, and then migrate the whole thing to Qt.

1.3 Version 01A

The "A" is because I'm in a "new series" now that I am leaving behind a lot of the older code.

One large difference is the fact that I am no longer working with pulses. Previously, the G-code was converted to individual pulses to the X, Y or Z axis, and each of these pulses was handled one at a time. The new version now starts the rendering process with the most elementary G-code statements possible: linear moves and arc moves.

The process of generating the DA has several steps. First, we obtain a line of G-code that specifies a cut. Next, this is converted to an intermediate form which will be used to adjust the global DA. There are various possibilities for this intermediate form – pulses is one of them, but these are not very efficient. A mini-DA based on scan lines seems like a good choice. The idea is to create a set of scan lines, one every 0.001 inch, where the depth of cut is given at every point along the line – i.e., a depth is given for every 0.001 inch. For a square-ended endmill, there is only a single depth, but a ballmills and drills are different.

The data for these scan lines should take the form of a count of the number of scan lines needed, the y-coordinate of the first line, and the x-position for the start of each line and a count of the number of steps to be taken in the x-direction, along with an array of depths for this many steps. One tricky thing here is that the scan lines may not have a "gap." For instance a semi-circle that extends "upward" will need to be broken into two parts.

Once we have the scan line-based DA, we can fold it into the global DA by simply stepping through each point.

After further thought, and looking at the original code, I see that rendering a DA is not so easy. In particular, see the various `paintComponent()` methods in `Cut3DPane`. Earlier, I tried two strategies, both of which occurred to me again, before I looked at the code. One strategy is to walk along horizontal or vertical lines of the DA and "connect the dots." Provided that these lines are close enough together, the rendering will be solid, with no gaps between the lines. This works, but it's difficult to avoid odd artifacts and methods of shading will always be somewhat ad hoc. The other method is to triangulate (or use other polygons) the surface.

Both methods require walking lines of the DA, including the need to walk up and down through depressions. Also, they both require a z-buffer to deal with hidden surface removal. Actually, I think I could use BSP trees (or some other method) with the polygons, but that would probably be slower than a z-buffer, and certainly more complicated.

2 April 20, 2013

I started implementing some of the ideas above, in particular the idea of encoding each cut as a collection of scanlines so that I could then copy these to the global Depth Array. This could be done relatively easily (although it's verbose) for endmills, but it becomes more difficult for things like ballmills and drills because the depth is not uniform.

I now think that a smarter version of the signal-based approach, which I was already using, is what is needed. There are three cases, one case for each of the possible tools: endmill (flat bottom), ballmill (hemispherical bottom) and drill (angular bottom).

For the endmill, start by "drawing" (i.e., setting the depth in the DA) a complete solid circle where the cut starts. Then, for each additional step, draw only the one pixel wide "edge" of the circle until the tool reaches the final position. This may or may not be faster than what I started to do with scanlines. The downside of the scanline method is that the depths must be set twice – once in the scanline representation and once in the global DA. However, the new method that I just outlined is slightly more complicated and may be a tad slower. It wouldn't be hard to change to one method or the other later for this single case.

For ballmills, the strategy starts by drawing only the first half of the cut at the tool's initial position. Then, with each additional step a one-pixel wide semicircular cut is made, taken through the center of the tool. At the final tool position, the other half of the ballmill's base is used to cut.

Drills and other angled cutters are handled similarly to the ballmills. In fact, there is a similarity among all three cases. Store a mini-DA for each tool representing the profile of the base of the tool. At the tool's initial position, we cut by copying half of this mini-DA to the global DA. This brings us up to a

"center line" of the mini-DA that represents the cuts of each additional step. Whatever the type of tool, we need to copy this center line repeatedly through the intermediate steps. At the final tool position, we copy the remaining half of the mini-DA.

As noted, the advantage of this method for endmills making linear cuts may be small or non-existent, but, even for endmills, this method is better for cuts along arcs – it's certainly simpler. In any case, the big advantage of this method is that every tool is handled the same way once you have the profile of the bottom of the tool.

Even so, cutting along arcs is not trivial. Linear cuts *a*re very easy since the same one pixel wide "slice" through the center of the tool simply needs to be replicaed as the tool steps along the line. For arcs, we need a small pie-slice of the tool to be replicated as the tool moves along the arc. Think of it this way: the outside edge of the tool must move faster than the inside edge...On the other hand, each step is in either the x-direction or the y-direction. Either way, we merely copy a single slice through the diameter. I think that the problem with these pie-shaped slices goes away.

On second thought, this won't work...With ballmills and drills, at each step forward, you must draw the entire half of the tool that's on the leading edge. So, you get some savings (about half) over copying the entire circle with each step, but not a tremendous amount. These ideas would work for endmills though.

The way to get efficiency seems to be to handle each cut (line of G-code) as a whole. A linear cut with a ball mill would be handled by cutting half of the circle at each end of the path, then a series of lines at various depths from one end of the cut to the other. Arcs would be handled similarly, but somewhat more complicated since the series of "furrows" are themselves arc of various radii.

So, it seems that what I have now, already written in Java, is not so different from what I am aiming for. It makes sense to move away from the Java version and start porting it to C++ and Qt. The next few steps should be as follows: (1) write a Qt program that allows me to open and view a text file; (2) put a splitter in the window and show something like a 3D cube in that window with the ability to maniuplate it with the keyboard; (3) port the G-code interpreter; (4) add various dialogs to the program that are needed, like to specify the tool turret; (5) move to the completed program.

3 April 23, 2013

I am working with Qt and liking it, although I haven't gotten very far yet.

I had what may be a GOOD IDEA for the problem of dealing with more general surfaces involving undercuts and the like. Triangulation is one solution, but it may be too cumbersome, with too many triangles to render in real time, and I'm not sure that it will look as nice as I would like. But trying to work with something more analytic, like NURBS, is difficult because of the problem of dealing with intersections. There's no (computationally reasonable) way to

determine the intersection of two volumes. Even something like a point-in-volume method is hard when the volume is bounded by NURBS.

A hybrid method may work. Use some voxel-based method to determine whether a particular cut has any effect; that way there's no need for point-in-volume calcuations involving NURBS. But, whenever a voxel is cut away, revealing some new surface voxels, the newly exposed surface voxels should have a pointer stored with them that points to some analytic/NURBS-like description of the larger surface of which that voxel is a part – in fact, each exposed face of the voxel would need such a pointer.

After all of the cuts have been made, look at every exposed voxel and make a list of the surface elements that appear on the surface. Now draw them, most likely using a z-buffer for hidden surface removal.

For example, a simple cut with an endmill leads to a nice surface that would be easy to define and draw. The problem is that once a particular suface is in place, further cuts could "mess it up." Suppose that a ballmill cuts a trough, then a smaller endmill cuts a groove down the middle of the trough. The surface due to the ball mill is nice, but the cut by the endmill means that some of that surface is no longer there; it's been cut away by the endmill. So, what you need are surfaces with holes and/or edges that have been cut back.

This might not be so hard to manage. It's clear how to deal with the data structures: each surface description needs an array of pointers to surfaces that are to be removed – areas of the surface that are holes. If we think of each patch as being a map $R^2 \rightarrow R^3$, then the hole could be a map $[0,1]^2 \rightarrow [0,1]^2$, where the range here is the same as the domain of the patch map. Put another way, the holes are simply regions of the plane.

One way to draw these "patches with holes" is to use a second buffer. Draw the complete (no holes) surface to the buffer, then draw the holes, but where the holes are "drawn" by marking the points of the original surface with a special code meaning "not to be transferred." Now copy from this buffer(but not the holes) to the z-buffer that will eventually be copied to the screen.

Another way, but this seems more complicated, is to draw only the surface and forget about trying to keep track of holes. The holes are made to appear as each surface is drawn by checking the voxels behind the surface. A particular pixel of the surface either "rests against" a voxel on the surface of the completed shape, or the voxel behind that point has been cut away. Don't draw pixels corresponding to voxels that have been cut away. This seems tricky, but might be doable. Each pixel of the surface to be tested corresponds to a point in 3D space, and this correspondence should be obvious since it's due to the map $R^2 \rightarrow R^3$. From this 3D point we need to check whether a particular voxel is solid or not. Conceptually that sounds easy, but it might involve a lot of calculation for every pixel.

It might be possible to speed up the "patches with holes" idea by precalculating the surfaces, although I'm not sure it would be necessary. The reason is might not be necessary is that each surface patch should cover a relatively large area of the object, and there shouldn't be that many patches. However, in a very complicated object, you could have more patches, and each of these patches

might have a lot of holes – think of something like cutting a surface that looks like a person's face. By the time each patch has had the holes removed, there won't be much of the surface left. There will be many (!) of these complicated hole-filled patches. If we think of each NURBS patch as a map $[0,1]^2 \rightarrow R^3$, then what we need is a way to specify the domain, D, after the holes have all been removed. That way, when we draw the path, we only need to consider $D \rightarrow R^3$. Hopefully, traversing D can be done more quickly than traversing all of $[0,1]^2$.

Something that must be considered with any analytic method is making sure that the edges of all of these patches line up correctly so that there are no holes in the surface where they meet. The coefficients of the Bezier curves must be floating point values, and I don't see how one ensures that rounding errors don't lead to small gaps. I don't know how that is handled, but it must be a common problem.

4 April 24, 2013

Some further throughts... I've been thinking in terms of only three tools: endmills, ballmills and drills, where "drill" includes any tool with a pointy angle on the bottom. In fact, I need to consider center drills and (maybe) reamers too. A reamer has a small angle cut off the bottom edge of the cutter so that the bottom is not entirely flat.

I see a way that it wouldn't be *too* hard to allow the user to specify any tool shape that does not allow undercutting. It's hard to explain without pictures, but such tool profile can be thought of as a series of simpler tools, "grown" one on top of the other. You "walk out and upward" (never inward) from the bottom center of the tool to trace the profile. Each of the steps is either linear or a portion of an ellipse. In both cases, all you need is the x and y distance of the step, and in the ellipse case, whether the arc is concave up or concave down. This wouldn't be that hard to implement, but I'm not sure if it's worth the trouble. In any case, it's the kind of thing that could be dropped in later without causing any serious re-engineering of the larger program.

Concerning how to render the image, I see two cases. There's the normal case of a typical part with a relatively simple surface, and there's the extreme case of machining something like a human face.

In the normal case, the number of triangles should be managable and the important issue is the accuracy of representation. If the edges of a cut, like the semicircular end of a groove cut by and endmill, are jagged, this will be visually obvious. To get accuracy of representation, run the DA through a filter that converts it to a series of scanlines. Thus, each line becomes a much shorter series of numbers, where the number only changes if the depth changes.[1] Now

[1] I think I tried something like this before, where there was no DA at all; only the scanlines. The problem was that adding a new cut to such a thing is complicated. The reason the idea is attractive is that it takes orders of magnitude less memory.

you have a much shorter representation that is just as accurate as the full DA. How to triangulate this is still open in my mind.

In the extreme case, the scanline idea has no advantage over the original DA; it won't be any shorter, and will probably be longer due to the extra data held with each change in depth. Here, the rendering strategy should probably be to simply draw each voxel as a three-sided cube. The problem is that rendering billions of cube faces isn't feasible. But the nature of the surface means that some averaging would be OK. Form a new depth array by averaging each five by five (say) area of the original DA, then render this new, much smaller DA, as a set of voxel cubes. The question is how to handle this averaging. For something like a human face, a simple average is probably fine. In other cases, you may want to throw out any outliers. Maybe I should measure the dispersion and act accordingly. For instance, if there are fewer than 5 (out of 25) outliers, then throw them away; otherwise, do a simple average.

5 April 30, 2013

I am liking Qt, and getting comfortable with C++ again. I think I made the right decision to go this route.

I am working on `vcnc06`. At this point, I have a main window that can open, save and edit text documents (G-code). I also implemented a dialog to input the machine geometry (PRZ, billet size and location of the tool turret). I have ported a good bit of the interpreter: the lexer, parser and layers 00, 01 and 02.

I've been reading about OpenGL. It looks like it may be the way to go for a lot of the rendering. I still have quite a bit to figure out, but it's worth pursuing. It's clear now that OpenGL works strictly by rendering triangles – no NURBS or anything like that – and it doesn't seem to include any sophisticated general purpose graphics routines either. It doesn't handle things like point-in-volume or BSP trees. Even if I don't use OpenGL for rendering as such, it may be possible to use the GPU to do many of the necessary matrix calculations that are used by just about any graphics algorithm.

6 May 2, 2013

Still on `v06`. I have the work offsets dialog done pretty much the way I want, and I am starting the tool turret dialog. Meanwhile, a bit of a rehash of an idea I think I wrote about before.

Think in terms of voxels, each of which is 1/1024 inches on a side. Now, represent the volume in two stages. First, as a set of super-voxels, each of which is 64/1024 on a side, so there are 16 to an inch. We can then represent a volume that is 16x16x16 inches (which is pretty large) with $16^3 \cdot 16^3 = 2^{24}$ (or 16 meg) of these super-voxels. That is reasonable. It will be easy and fast to go from a 3D coordinate to the corresponding voxel – almost as easy as simply indexing

the array element. Each of these super-voxels will contain one of three things: a flag indicating that it is empty, a flag indicating that it is solid, or a pointer to information about how it is partially full.

When a super-voxel is partially full, I can't just use a volume of the 1/1024-sized voxels to represent the partial fill. That takes too much space. There are two choices that might work. One is to bound the portion of the super-voxel that is solid with polygons (triangulate it). The other choice is something more analytic.

There are various ways to make use of the idea of super-voxels, all of which come down to what sort of information the partially full super-voxels point to. As I pointed out, except in very simple cases, subdividing the super-voxels into 1/1024 voxels won't work. Another choice relies on the fact that there are only a very limited number of ways that such a small volume could be cut – actually, this is more of an assumption than a fact, and dealing with the mismatch between the two could be more or less hairy, depending on how much accuracy and flexibility is desired.

If we assume that any super-voxel can only have one cut, made by a single tool, one time (the "1-1-1 assumption") then the number of possibilities is severely limited. Think about all of the ways that all of the possible tools could "extend into" a small cube. The number is large, but easily parameterizable. If a super-voxel is cut more than once, either by the same shape tool (perhaps at a different angle on a 4 axis machine, or on a different side of the cube), or by a differently shaped tool, the you could store both of these cuts with the super-voxel and sort out how the resulting surface should be rendered at the rendering step.

A different super-voxel strategy is as follows. For each cut (i.e., each G-code statement), store some kind of description of the volume removed – this might be splines, triangulations, or anything else. For each super-voxel that is cut (is not completely cut away or left untouched) by a statement, store a pointer in that super-voxel to the description of the entire cut. For most super-voxels, there will be a pointer to only a single one of these descriptions. In cases where two or more cuts touch the same super-voxel, determining which cut is "primary" or how they interact should be easier because the interaction takes place in such a restricted volume. With the description stored with each statement we can store a count of the number of super-voxels which that statement cuts. As we consume additional cuts, some of these cuts may be rendered irrelevant because some subsequent cut removes that entire super-voxel. In such a case, we reduce the count of super-voxels cut by the first description; when that count reaches zero, we can remove the description from memory; it has no relevance for the rendering. I could store a list of the exact super-voxels cut by each statement too, but I'm not sure what purpose that would serve, even if it sounds useful.

CLEVER IDEA (?). Suppose that I use the super-voxel idea above, where the partially full super-voxels hold pointers to the G-code statements that cut through them. Note that I will assume that these super-voxels really *are* partially full. That is, if a series of cuts gradually reduces a super-voxel to being empty, then it has been taken out of the process somehow. This could in fact

be a tricky thing. At any rate, make that assumption. The issue is how to render this. The problem is that the determination of which portions of the surface created by the lines of G-code that cut this super-voxel has not been made. You could have one of the two being totally irrelevant – that part of the surface was cut away by another line of G-code; this is probably the most common situation. Or, you could be in the trickier case where parts of both surfaces are "on the final shape" and must be rendered – but only a *part* of both (or all three, *etc.*). One way to approach this is analytically. I haven't looked into this, but I suspect that it's a mess: you need to know the curve where the two surfaces intersect, and you would like this curve to be expressed in each of the two parameter spaces; *i.e.*, in $[0,1]^2$ or whatever the domain is for the parameterization.

Another way to do this is with a kind of modified z-buffer. This doesn't involve any tricky analytic calculations; it's more brute force. Suppose there are only two surfaces in a given super-voxel and we need to figure out which one, or which part of each one, needs to be rendered. Each of the two surfaces came from a line of G-code, and suppose that we have a surface normal for each one, and that this normal is defined to point to the interior of the volume cut away by that line of G-code (so it points away from the object being created). The normal may vary over the different parts of the surface within the super-voxel, but imagine that it's more or less constant througout (just to make the argument clear). Now imagine looking at the surface against the direction of this normal (so you are looking toward the surface of the object being created). Your line of sight must intersect each of the two surfaces. The question is, "which one is in front at different portions of the super-voxel?" The one that is in front in a particular area should not be rendered in that area.

Conceptually, it feels like the idea above could be implemented by rendering each of the two surfaces into a z-buffer, but where the z-buffer is used backwards: the values that come from points that are *farther away* are the ones we want. In practice, I'm not sure how well this idea will work. There are too many possible perspectives on the situation – that is, the normals for each of the two surfaces may point in radically different directions, or they may vary a great deal over the super-voxel.

Above, I noted that having a parametrization of the curve along which the two surfaces intersect would (more or less) solve the problem of rendering. We might be able to find it by a kind of recusive subdivision, or what amounts to function minimization. Suppose you have two points on a single surface, with one of these points known to be in the finally cutway portion of the volume and the other in what remains. Aside: I need a name for what is ultimately cut away. Call the cut away stuff (what gets turned into chips) the *null volume* and what remains the *solid volume* so that the original billet is the disjoint union of the null volume and the solid volume. Anyway, you have two points on a surface, one in the null volume and one in the solid volume. Any path between the two must pass through the intersection of this surface with the other surface. You can find that point by any number of means. The simplest is probably to repeatedly subdivide the line connecting these two points in the parameter

space. So, now you have one point on the boundary. What you would like is to start at this point and draw a polygon (which might not be closed; in fact, it probably won't be closed) that approximates the curve of intersection. I don't see any clear way to do that.

Maybe a different idea would work. Start at an arbitrary point on one of the surfaces. Determine whether that point is in the null volume or the solid volume. Now use a "seed fill" type algorithm, where the seed is grown in parameter space (the domain). You could pixelate the domain, and check each square of the resulting grid, much like seed filling a polygon when drawing to the screen in 2D. You could tweak the resolution of the grid based on the level of magnification. This would work, but it seems slow...OTOH, it might not be so slow. The amount of "surface area" is comparable to what I'm doing now with a DA and no under-cutting, and it only has to be done once, prior to the real-time rendering. If it is too slow, there are ways to speed it up, with some sacrifice in accuracy. For instance, I could walk from that point out along each of the four directions (x and y, up and down). This would give me a quadrilateral. I could then try a couple of points near the middle of each of the four sides of the quadilateral, and I would have an eight-sided figure, which should be plenty since we're restricted to a 1/16 inch volume. If these points are used for a Bezier curve, then that gives a *lot* of fine control, probably more than I need.

Somehow, though, I don't think that's the "correct" way to solve this problem – the method used by others. OTOH, the software that I've seen for this problem may not handle this sort of situation at all well. I've never seen it used in cases that were tricky like this. What I've seen have been fairly boring examples, strictly using three-axis machines, and square-ended mills. The output in those cases is really nice, and my idea would work equally well, but they might not be able to handle the kinds of situations I envision.

What's important in any situation is the transition from one cut to another. I can render a single cut beautifully (it's just a swept volume); it's the way they interact when they "cross paths" that causes problems. Think about what you see with a program like SolidEdge as you zoom in on the arc of a circle. For just a flash, you see an approximation to the circle by line segments, then it gets overwritten by a better image of the arc, one that doesn't appear to be an approximation.

Clearly, the thing that's important is the curves that act as the boundaries between one swept volume and another. One one side of such a curve, you render one surface, and that can be done nicely; on the other side you use a different surface. But that's the wrong perspective; from the 3D point of view there is no "one side or the other." Everything happens from the point of view of being on the surface. That is, as you render a surface resulting from a particular statement, the question is how much of that surface to render. You want to render that surface right up to some boundary, where (I think) it makes the most sense to think of that boundary as being in parameter space. Also, although I've been focusing on intersections due to different G-code statements, the intersection between two faces from the same statement (say the 90° angle between the base of an endmill and the sides) is just as important (maybe more)

and will account for the large majority of these boundary curves. Now, as you zoom in/out, you can make these boundaries more/less accurate. That would explain the behavior in SolidEdge. From a distant perspective, it's enough to use a course approximation to the boundary curve. The user zooms in, the system draws a quick picture with the coarse approximation while it makes the boundary curve more accurate, then redraws.

Determining which parts of the total volume to focus on and make more detailed is not hard – just intersect the viewing frustum with the set of super-voxels, and examine those. In fact, I think that this kind of adjustment to the level of detail may be necessary to avoid weird pixelation artifacts (aliasing). The usual rule of thumb is that the "frequency" of the screen should be greater than the frequency of the sampling by a factor of two. That is, each screen pixel should correspond to a two-by-two area in object space.

7 May 13, 2013

Making good progress. I'm working on the `Layer05` class now, which handles cutter comp. I'm on version `v07`. Most of the code is pretty much the same as the Java version (with several bug fixes), but the `Layer05` class has some more significant reorganization. The Java version used various numerical methods along the lines of function minimization or root finding to deal with the intersections of curves that arise in cutter comp. For the C++ version, I'm doing everything algebraically.

8 May 20, 2013

`Layer05` is done in the sense that I think it's correct and it compiles, but I'm sure there are bugs. That is version `v07`. At this point I've begun reworking and validating the interpreter, from `Layer05` down to the `Lexer` by writing a test suite of input files. I am doing this with version `v08`. I've already found several small (but annoying) bugs in the `Lexer`.

9 May 24, 2013

I'm making progress on the validation of `v08`. So far, I have validated up through `Layer02`. I've found numerous small(ish) bugs. The biggest change I have made is avoiding throwing errors and doing a better job of allowing the interpreter to forge ahead, even in the presence of errors. I also added the ability to post a warning (rather than an error) for questionable statements.

10 May 30, 2013

I cut `v09` because I see now that one cannot mix cutter comp with `G18` or `G19`, or at least that there is no natural and intuitive way to do it. See the comment and example in `main.cpp` under `v08`. I think that what I've done "works," in the sense that it does what I intended, but there are too many situations where this will give things that the user will find weird or unexpected. The bottom line is that there is no situation that I can think of where what I have done would be useful. In fact, I don't see just what the user might want when mixing cutter comp and planes other than the XY-plane.

That's too bad because I spent a fair amount of time trying to iron all of that out, but it's a relief too because the code is much cleaner without having to worry about the entire issue.

11 June 6, 2013

I'm just about ready to release a version of the program to friends for testing, but I've been very busy with other stuff lately (the boat, Maine Hacker Club, *etc.*).

I had an interesting idea on copy-protection. I had been thinking of embedding each user's name, and other identifying information in a picture used by the program – maybe the splash screen, maybe some icons, whatever. I'm not sure how obvious that would be. It doesn't seem like it would be obvious to the average person, but someone might figure it out. Here's another strategy. In many situations, several different machine language ops could be used to do the same task. I could choose some block of code in the program and investigate which machine language ops have equivalent alternatives. By choosing among them, I can encode whatever I want.

Also, while I'm thinking of it, I starting making a change to `v09` that I thought better of. When an `M06` appears (tool change), the tool should move to the tool turret. For the translator, exactly what the tool does in the physical world is less important, but for the simulator, I really need to keep track of where the tool is at. I had thought of inserting extra comamnds to cancel things like cutter comp just before an `M06` appears. It seemed like this would make the move to the tool turret easier for the interpreter, but it really doesn't matter, so this isn't needed. The natural place to make these changes is `Layer 02` since that layer already keeps track of all of the different modes that I wanted to cancel. I thought that cancelling these modes would make it easier for me to move the tool to the turret, but it's really not necessary. However, I should insert an absolute move to the tool turret just before the `M06`. The place to insert that move is at the interpreter layer, not the translator.

12 June 8, 2013

I spent a crazy amount of time trying to get a version of the program that would run on machines without Qt. I sent it to Jim and Charlie. It can be found in the `release-6-7-2013` directory. What I sent them is the `Ger Compressed.zip` file. See my `QtNotes.txt` file for what I did to make a deployable version. I'm calling this release version 0.50.

So...it's time to make the hard decisions about the entire rendering process. The first stage of rendering is really the last stage of the interpreter: converting the simplified G-code that comes out of `Layer05` into some sort of signals, or preparing the G-code in some way to be ready for the next step. There are two broad choices here. The first choice, which is the one that I used in the Java version, is to convert each statement to a series of pulses to the axes. Another choice is to work with each statement as a whole. The pulse approach has the advantage of simplicity and flexibility, but it's slower because you loose the structure inherent in knowing how the pulses fit together into a larger move (linear, arc or helical). For now, I will go with the pulse strategy. I'm not convinced that the other strategy would be dramatically faster unless I go to a great deal of work.

I've done this calculation before (several times), but assume that the tool will travel 1,000 inches over the course of the program. That's quite possible – if each line of G-code moves the tool one inch, then that's only 1,000 lines of G-code. Assume a half-inch end-mill, so that the area of the mill is $0.25^2\pi$ in^2, which is about halfway between than 2^{17} and 2^{18} squares of 0.001 in^2 – call it 2^{18}. It will take this many copy (and check) operations to have the tool cut the DA at a single position. If the tool travels 2^{10} inches, then the tool needs to cut at 2^{20} positions, and each of these cuts requires 2^{18} copy operations. The total of 2^{38} operations is too large – I'm trying to keep major operations down to something like 2^{30}. However, this is an overestimate for several reasons. I should be able to look forward to the next pulse so that I can avoid cuts for many pulses (e.g., if the next pulse goes to greater z-depth, then there's no point in doing the cut at the current position). Many moves will be entirely above the part, and I can skip cutting those by a global check on the part height. If the next move is an x/y-pulse, then I only need to cut the leading edge (for endmills) or the leading half of the tool (for ballmills and drills). This only reduced by half for ballmills, but it's a huge reduction for endmills: from 2^{18} operations per cut to about 2^{10}.

If necessary, here's a trick to speed up the DA cutting process. Use something like a quad-tree, or it would probably be sufficient to just have two DAs: one at the full level of detail (0.001 in) and one at a resolution of something like 0.016 in. This extra DA would take only a fraction of the memory of the full DA (1/256 for the example I gave). In the set-up phase, divide the cutting base of the tool into a grid – put another way, have two versions of the cutting area: one version at 0.001 in and one at some coarser level of detail. Check the coarse cutting grid against the coarse DA before going to the finer grid. In fact, there would be some speed-up using this strategy, even without the extra

DA (or quad-tree). Divide the base of the tool into regions, and compare the depth of each of these regions to the DA. If any point in the region does cut the DA, then look at each pixel of the region. The best way of dividing the cutting area into regions is probably to have a small circle around the center of the tool (where a ballmill or drill cuts deepest), then an annular series of sort-of-rectangles around that.

An entirely different strategy is this. Don't cut the DA until you know what the user is looking at and the resolution. If he's looking at the part from such a zoom factor that each screen pixel is 0.1 in, then there's no point in cutting at a resolution of 0.001 in, and if he zooms in so that a resolution of 0.001 in is needed, then there's no reason to cut those tool positions (at 0.001 in) that he can't see (which will be most of the part). This is complicated, and all of this cutting happens before real-time rendering, making the time needed less of a concern, so I want to avoid anything like this if I can.

13 June 10, 2013

I found a free open-source program that does pretty much what I intend to do. See `openscam.com`. It looks pretty nice, and renders in 3D. There are links there to some other open source software, like HeeksCAD. Another program that does 3D rendering is at `cncsimulator.com`, but they charge (minimum) $99 per year. Yet another is at `cnccookbook.com`; he also charges $99 per year, but I don't think he does 3D rendering.

14 June 16, 2013

I have most of the code needed to render in 2D, although it only works with endmills, and there are bugs of some kind. That was in verion `v11`, but I think I found a strange bug in some combination of QtCreator, MinGW and/or gdb. I've spent the last couple of days trying to isolate it, and I ended up posting a bug report to the Qt website. For now, I will try to ignore the fact that this bug exists – I'm pretty sure it's a bug in the development environment, so I hope that I can get my code working even if this bug is never resolved. I've mved to version `v12` to try to get rid of whatever bugs are left in my code.

15 June 22, 2013

The 2D render code works, but only for endmills. There's not a lot I could do to make it faster, but it's still slower than I would like when working at full resolution (no adjustment to the "scale."). It's *very* fast when working at reduced resolution. One way around this might be to cut the DA every time the user changes the view. We only need the full resolution when the user is zoomed all the way in, and in that case we only need to cut a small portion of the entire DA. I might be able to do something like this to speed things up for

the user's experience when rendering in 2D, but I don't see how to do anything like this that would work in 3D, at least not very easily.

For 2D, I could do something like the above, but a bit more "fixed" in the approach. Always cut the object at a resolution of 0.010 in per step. That's fast, and it doesn't require a lot of memory. If the user zooms in, go ahead and cut only the part that's visible at the higher resolution. Since it's only for a limited area, it wouldn't take a lot of memory either, but it needs to be fast enough to be interactive in real-time. Obviously, we need to hold all of the signals in memory, and that will take some memory (but within reason).

For speed, there are several approaches. It might work to tag each signal with information about which region of the surface that signal cuts, but adding so much information to every signal is likely to take too much memory. I could just run through *all* of the signals every time, and pause to cut only for those signals that are within the tool radius of the visible region. That might be fast enough – the speed issue is not the number of signals, but looping over all of the pixels at the base of the cutter. Another strategy is to break the entire set of signals into groups of 100 (say) pulses. In 100 pulses, the tool can't move more then 0.100 in in any direction. Preface each set of 100 pulses with the x, y, z coordinate of the tool as you move into that set of pulses, along with a pointer to the set of 100 pulses. You could easily check whether those pulses could possibly be able to cut the area being viewed; if not, then jump to the next 100 pulses.

I like the idea above. I may find that doing it is not necessary, but the idea is clean if I need it.

Here's another general speed-up idea, similar to the idea of quadtrees. Put a grid over the main DA, say 16 by 16 pixels of the DA for every pixel of the grid. Let the grid contain the tallest (least deeply cut) position of the 256 it represents. Now, apply the cutter in a two step process. The cutter should have two local DAs – the one I ready have and a local grid. First, cut the grid with the cutter's local grid. If the cutter's grid doesn't cut the local grid, then it won't cut any of the pixels in the full DA for those 256 pixels. If it does cut the grid, then we have to visit all 256 pixels of the DA that belong to that grid-point. My guess is that this could lead to a dramatic speed up, and in the worst case it won't slow it down much at all – it could make it run in only 1/256-th of the time, and in the worst case, it will add that amount of time to the total.

Don't forget about OpenGL. A lot of these ideas may be moot if I use OpenGL. Cutting the DA seems like the ideal thing to send through a compute shader – it's easily parallelizable, all it does is trivial integer arithmetic, and it's working with a 2D array that's basically a bitmap. Memory remains an issue, but not speed, and memory isn't such an issue. At one pixel per 0.001 in, a square inch takes (just under) 1 meg and a square foot takes $144,000,000$ bytes. Just say that $1000 \cdot 1000$ is 1 meg – it's actually 5% less than that – so that a square foot takes 144 meg. A four by 8 foot sheet takes 4.6 gig, which is more memory than I can reasonably assume that a person has. If I work at "medium" resolution, then I can divide this by 16 to get 288 meg, and if I

work at "conservative" resolution, then I divide by 100 to get only 46 meg. So memory is not really an issue – highly accurate machinining will generally take place on small objects (less than 1 square foot) and for something as large as a sheet of plywood, the user is going to be cutting large shapes, not doing fine machining.

However, it appears that Qt 5.0 does not know about compute shaders. A quick check on the net shows that there are ways to use them from Qt, but they're not part of the normal Qt framework. However, they do have plans to add them to Qt in version 5.1. Maybe I don't need the full power of a compute shader anyway.