1 Overview

The goal is to write a program that takes G-code as input and converts it to an image of the resulting 3D object. There are many choices to be made in regard to the user-interface, but those issues are simply a matter of making choices; they don't determine whether the entire idea is viable. The more important, and more difficult, question is how to represent and render these 3D objects. Nevertheless, certain desired features of the program should be kept in mind.

- The user should be able to rotate and zoom in and out on the object interatively.
- The user should be able to click on a line of G-code and see the volume of the corresponding cut in the rendered object.
- Changes to the G-code should lead to an immediate update of the rendered image.
- Working with both inches and millimeters should be possible.
- Although a 3-axis machine is much simpler, four and five axes should be allowed.
- Arbitrary shapes should be allowed for the material to be cut.
- Any tool shape should be permitted.

There are several steps to the process of converting from G-code to an image.

1. Parse the code.
2. Generate an internal representation of the resulting object.
3. Render this object on the display.

Consider step (1). However this step is implemented, it will be the least time-consuming part of the process. The prototype that I've written shows that this step will take a tiny fraction of a second to execute.

Define a "G-cut" to be an action of the simulated CNC machine that results in some change to the workpiece; e.g., `G01X1.0Y2.0` cuts material and is a G-cut, while codes like `M03` and `G17` merely affect the internal state of the simulated CNC machine. Implementing step (1) could be done using at least two approaches. The first way is to work with each G-cut; the second is to work with the pulses that would go out to the motors' controllers.

For simple objects, working with G-cuts is likely to be faster. The internal representation used in step (2) would most naturally be a set of bounding polygons (and other curved facets). The problem with this approach is that for complex objects with many small moves, there would be a large number of bounding facets. I suspect that dealing with such a large number of facets would make step (2) slow. Allowing for four and five axis machines complicates this approach even more. In addition, I would like to write a CNC controller (to work with actual machines) that allows arbitrary cuts, not just arcs of circles,

and that makes working with facets even harder. Working with G-cuts may be possible, but it seems very unlikely to be as fast or as flexible as working with pulses. Moreover, the pulse approach is clean; it leads to a clear demarkation between steps (1) and (2); changes in either step should have no effect on how the other step is implemented.

I will assume below that the output of step (1) is a series of pulses. Equivalently, these pulses can be thought of as a set of (x, y, z) coordinates for the tool. The remainder of this document discuss the implementatin of steps (2) and (3) under this assumption. These two steps will take orders of magnitude more computation than the first step, and deserve the most attention.

2 Step (2): Internal Representation

The step includes the conversion from pulses to a representation of the resulting object. The contraints on memory and on time are both important, but the time constraint seems most severe. The choices made in this step are likely to affect the implemention of the next (rendering) step, so there is also some discussion of that problem in this section.

I have thought of many different approaches. Each is considered below. There are two broad categories of representation: volume-based and surface-based. My initial guess is that volume-based methods (e.g., octrees) will be more natural to implement, but that surface-based methods will be somewhat faster if I put in enough effort. However, the extra complexity required by the surface methods may make them slower when actually implemented by a mortal.

When thinking about volume methods, it's natural to imagine a two-step process. First, the volume swept out by the tool is determined, then this swept volume is subtraced from the workpiece. Handling things in this way will make it easier to allow the user to select and view the volume cut away by a particular line of G-code. However, it may be faster to eliminate this step. For example, to determine the swept volume on its own requires that we examine every bit of volume near the tool, whether or not that bit of volume has already been cut away from the workpiece. If the cuts are made directly to the workpiece, then we need only consider volume elements that still remain. This example isn't the best, since there's nothing to prevent the program from noting the volume cut away these cuts are observed.

Exactly where to start and end these individual swept volumes is another thing that is yet to be determined. It's natural to allow for each of these swept volume to be associated with a single line of code (a G-cut), but the algorithms may be more or less efficient if the number of tool positions considered at once is larger or smaller.

In comparing different methods, I will make example calcuations based on certain canonical assumptions.

- The volume is to be divided into 1/1024 inch cubes, or into cubes of 1/128 inch on a side. At one extreme the material is represented at roughly the

greatest precision that a machine can attain. The other extreme was chosen because, when rendering an object at normal levels of magnification, this is the greatest precision that can be shown on the display – any further subdivision of the material would not lead to a more detailed rendering. Even 1/128 may be more than necessary, depending on the size of the object. If the display window is 512×512, then there is no good reason to use more than 2^{10} subdivisions on each side of the object. For a square object 16 inches on a side, that works out to 2^{10} subdivisions over 2^4 inches or a resolution of 1/64.

- A typical program involves 2^{16} different tool positions. At 2^{10} samples per inch, this amounts to $2^6 = 64$ inches of tool motion. This is more than will be used for a simple object, but far less than might be used for something more complex.
- The tool is an endmill, 0.256 inches in diameter, and it makes a cut one inch deep. That's not to say that other tools couldn't be used; I just mean to use this as a base case for comparing different ideas.
- The surface area of the finished object is $2^6 = 64$ square inches. A cube that's 3 inches to a side has $6 \times 3^2 = 54$ square inches of surface area before any cuts have been made, and a flat surface 8 inches on a side has 64 square inches of surface area, so this is a reasonable value for hobbyist use. In practice, five of the six sides will often be featureless. The surface area is most important as a determinant of the amount of memory required.

Also, the algorithm chosen may depend on the particular tool and/or whether the machine has three or more axes. For example, with a simple end-mill on a 3-axis machine, even if I use an octree to represent the volume at the end of step (2), it may make sense to quadtree-oriented algorithms to reach the ultimate octree goal.

2.1 Building the Volume

The most natural volume-based method of representation that I know is octrees. CSG (constructive solid geometry) and other analytical methods seem too slow because there are too many primitive objects. Bounding surface ideas may also work. A variety of these methods are discussed below.

Recall that an octree is a tree where each node has eight sub-nodes. A cube of size s is divided into eight cubes of size $s/2$, and each cube is either solid, empty, or partially solid. If a cube is either solid or empty, then the tree terminates at that node; only nodes whose cubes that are only partially solid need to branch. This makes the representation fairly efficient memory-wise.

The number of nodes required is roughly equal to the surface area of the object. This is more in the nature of a rough upper bound since the exact number of nodes required depends on the shape of the object, but it's accurate to a first approximation. If the surface area is 64 square inches, and the resolution

is at 1/1024 inches, then the number of nodes required is on the order of

$$2^6 \times (2^{10})^2 = 2^{26} = 64 \text{ Meg};$$

if the resolution is only 1/128, then the number of nodes is

$$2^6 \times (2^7)^2 = 2^{20} = 1 \text{ Meg}.$$

This is only the number nodes, not the amount of memory required. Each of these nodes represents a $1 \times 1 \times 1$ cube. Call an $n \times n \times n$ cube an "n-cube." Suppose that for every two of these small cubes there is a single 2-cube so that, on average, each 2-cube points to two 1-cubes. Each 2-cube has a single pointer to an array of eight 1-cubes, and each of these pointers takes (suppose) four bytes. Therefore, each 2-cube uses 36 bytes. Each 1-cube requires an additional four bytes since solid cubes are to be represented by letting the array of eight sub-cubes be null. Then, for 1/1024 resolution, there will

$$64 \text{ Meg} \times (36/2 + 4) = 64 \text{ Meg} \times 22 \approx 1.4 \text{ Gig}.$$

If the resolution is only 1/128, then the number of bytes required is 1/64 of this amount, or

$$22 \text{ Meg}.$$

This is probably an underestimate of the amount of memory required. I am neglecting any cubes larger than 1-cubes, and Java probably requires more than four bytes for each pointer. However, see below for a method of reducing the memory requirements at some cost in speed.

The bottleneck of any method of volume representation is converting the series of tool positions to the representation of the resulting object. Most of the discussion below assumes that the volume will be represented by an octree. The simplest method to generate an octree is to step through each of the tool positions and add the interior of the tool to the octree of the swept volume. Implementation would be easy, but it would be about the slowest method imaginable since there would be so much redundant re-adding of the same volume.

2.2 Top-down Cube Checking

This is what I do now, though the current implementation is very inefficient. A set of tool positions, as (x, y, z) values, is given. Start at the largest cube, and work down recursively. For each cube, determine whether the cube is solid, empty or partially solid. If the cube is solid or empty, then the recursion stops; if the cube is partial, then descend to the next smaller cube size and continue checking. This strategy requires the status of multiple cubes to be determined, and these checks must be based on the set of tool positions. This method is too slow without some means of reducing the search time for tool positions that are near the cubes being checked.

Roughly speaking, once the tool position nearest to the cube in question has been determined, here are the steps involved.

- Determining whether or not a cube is solid is easy. A cube is solid iff all eight vertices are inside the tool. For an end-mill, this is an easy test. The vertices must be above the base of the tool, and they must be within the radius of the tool. Each of these tests requires 6 additions/comparisons and two multiplications. There are eight vertices, so that is 48 additions/comparisons and 16 multiplications.

- If the cube is not solid, then there are messy additional tests that must be made to determine whether the cube is empty or partial. This is much simpler in the case of 1-cubes since these must be either solid or empty. As an estimate assume that, if the cube is not solid, then 16 extra additions/comparisons must be made

This is an underestimate of the time involved since it does not include time for subroutine calls, memory allocation, moving about the octree, *etc.* Roughly speaking, the number of 1-cubes that are tested will be equal to two times the surface area – *two* times because all of the 1-cubes on the inside must be tested (and they will be solid), and all of the cubes that are along the outside must be tested too to determine that they are in fact outside the swept volume.

Note that a cube cannot be determined to be solid unless the cube fits entirely in the volume of a single tool position. Each of the eight vertices of the cube might lie in a different tool position, yet the cube as a whole might be only partially filled.

Using the numbers above, a surface area of 64 square inches, expressed as the number of 1-cubes, is

$$2^6 \times 2^{10} \times 2^{10} = 2^{26}.$$

This many solid 1-cubes, and this many empty 1-cubes must be visted, so that the total amount of computation required will be

$$\begin{aligned} \text{additions/comparisons} &= (2 \times 48 + 16) \times 2^{26} \approx 2^{33} \\ \text{multiplications} &= 2 \times 16 \times 2^{26} = 2^{31}. \end{aligned}$$

The $2\times$ terms arise from the empty 1-cubes. Roughly speaking, we can summarize this by saying that the total number of operations required is 2^{33}.

As a test, I had Java perform a loop 2^{30} times. I had two additions (one of which was for the loop counter), and one comparison for each pass. That took about 1.5 seconds. Interestingly, multiplications are not much slower than additions on my machine. So, this *might* be managable.

The other task that must be performed is determining which tool position to use when comparing a cube to the tool volume. Currently, I do this by examining the entire list of tool positions for each cube, which is about the most inefficient thing possible. Given a set of potential tool positions, to find the nearest one (for an endmill in a 3-axis machine), I must do about 12 adds/compares and 2 multiplications for each tool position. Since adds and multiplies seem to take about the same amount of time, for a round number say that there are 16

operations. The biggest example that I have done so far involves $3,065$ tool positions, $11,763$ 1-cubes, $4,595$ 2-cubes and a few thousand larger cubes. To see if what I have estimated so far is reasonable, I find that there are about

$$3,065 \times 16,000 \approx 48,000,000 \approx 48 \times 2^{20}$$

checks of tool position, each of which involves 16 operations, for a total of about 3×2^{28} operations. That's consistent with the amount of time that I observe that the process takes (1 or 2 seconds).

In any case, what I am doing now can be made much more efficient. As we descend from larger cubes to smaller cubes, the number of potential nearest tool positions should go down as well, provided that I keep track of which tool positions were nearby for the larger cube. As the program recurses to smaller cubes, the subroutine should be passed a list of all tool positions that are near this cube. Then, only those positions need to be examined (and they are examined as before). Roughly speaking, each time the cube gets smaller, the number of nearby tool positions should go down by a factor if $1/8$. Therefore, the total number of times that a tool position must be examined as we descend to each cube is roughly

$$n + (n/8) + (n/8^2) + \cdots ,$$

where n is the total number of tool positions. But each of these examinations must be made 8 times, so that the number of total examinations that must be made is

$$n + 8(n/8) + 8^2(n/8^2) + \cdots .$$

This sum bottoms out when we reach the level of 1-cubes. Therefore, the sum is on the order of

$$n \log_2 s,$$

where the largest cube in the octree is an s-cube.

This is certainly an underestimate of the number of tests that must be made since descending to the smaller cubes will not reduce the number of tool positions by quite as much as a factor of 8 (some tool positions will be on the boundary), but it's of the right order. A considerable reduction in this number could be achieved by dividing up the workpiece into 2^7-cubes (say), then pass though all of the tool positions a single time and divvy up the tool positions to these 2^7-cubes. Doing this chops off the begining of the series above so that $n \log_2 s$ might be reduced to something like $n(\log_2 s - 4)$. In any case, $n \log_2 s$ is a far cry from n times the number of 1-cubes.

In practice, we can probably assume that each 1-cube will require that 4 tool positions be examined. Each of these examinations requires 16 operations, for 64 operations per 1-cube devoted to sorting. This is roughly the amount of time per 1-cube required to test whether the cube is solid or empty. Referring to the test done above, if this scheme is implemented, it would take around 3 seconds to create the octree. This is barely livable.

Aside: the memory required to do this is manageable.

An interesting statistic is the total number of operations required for each tool position. Assume the $n \log_2 s$ implemention of the algorithm, and a tool path that is 64 inches long, using a 0.256 diameter end-mill cutting 1 inch deep. If the tool cuts a single non-overlapping groove, then the surface ares of the cut is roughly $2^{10} \times 2^{10} \times 64 \times 2 = 2^{27}$ (neglecting the ends of the path). This is not far from 64 square inches (which is 2^{26}). So assume that the surface area of the path, after taking overlaps into account, really is only 2^{26}. As already calculated, the number of operations that arise from visiting and checking each cube of the octree is about 2^{33}. The number of operations per tool position is then $2^{33}/2^{16}$, or 2^{17}. The number of operations required for sorting and searching is negligable by comparison. There are 2^{16} tool positions, and $s = 2^{16}$ (64 inches, a gross overestimate) implies that the total number of examinations of the tool position will be $16 \times 2^{16} = 2^{20}$. Each of these examinations takes about 16 operations, for 2^{24} operations. Divide by the number of tool positions to find that the number of sorting and searching operations per tool position is only 2^8. Therefore, we can summarize by estimating that the total number of operations per tool position to generate the octree is about 2^{17}.

One way that the cube checking could be speeded up is by avoiding rechecking the same vertices multiple times. If cube A is in the swept volume, and we want to check whether cube B is in the volume, where cube B is adjacent to cube A, then we already know that the four vertices of A in common with B are in the volume. Ideally, we only need to look at the four new vertices of A. The problem with this idea is that the four common vertices may have been tested relative to a different tool position than the tool position that must be used for the four new vertices. This idea has the flavor of a seed fill.

Another issue here is the need to reexamine the octree after it should be complete to eliminate any empty or solid cubes that are observed to be empty or solid because every sub-cube is observed to be so. This situation is likely to arise because a 2-cube may appear to be partial if a tool barely nicks it, while all 8 1-cubes are ultimately determined to be empty since the tool's nick isn't deep enough. A similar situation arises for solid cubes. Not only could a 2-cube end up being solid because each of its 1-cubes is barely solid, but larger cubes (like 2^9-cubes) can be observed to be solid by comparing to a single position of a relatively small tool. The largest single cube that can be immediately observed to be solid must fit in the tool volume.

2.3 Quadtree with Height

A quadtree is usually used to represent a region on the plane, just as an octree is used for a 3D volume. By associating a height value with each square of the quadtree, we can use such a quadtree to represent simple volumes – in particular, those with no undercutting. Another way to express this is to say that they can represent volumes defined by a *function* $z = f(x, y)$. These will work in only a limited number of cases (e.g., end-mills on a 3-axis machine), but they may be so much faster that they are worth implementing for those cases.

The simplest way to implement these objects is to allow a single height

value for each square. If the square is subdivided, then the height value is to be ignored. A potentially useful change is to make the height value for each square equal to the highest square among any of the sub-squares. That way, if a tool position is being compared to a large square (a 2-square, say), we don't have to descend to the 1-squares if the tool is higher than the height given by the 2-square.

The memory required for this representation should be far less than what is required for an octree. In the worst case, the two amounts are comparable, but this worst case can't easily arise for an endmill on a 3-axis machine. For instance, if the end-mill is used to carve out a very detailed 3D surface, with very few flat areas, then an octree should require roughly the same amount of memory as a quadtree with height. Also, if it's necessary, a quadtree could be compressed in much the same way that an octree can be. The associated height values could be stored in a seperate array.

In fact, it might make sense to dispense with the quadtree altogether, and use a simple 2D array of shorts. For a square object 8 inches on each side, this would require $(2^{10} \times 2^3)^2 = 64$ Meg of entries. If each entry is 2 bytes, then this requires 128 megabytes. Working under this framework, the cuts made by an end-mill are analogous to painting the screen with a circular brush. For tool moves that are bounded within a fairly small rectangle (and most will be), this could be sufficiently faster to make it worth handling as a special case.

So, consider the situation of a simple end-mill on a 3-axis machine. The simplest algorithm is to consider each tool position, and adjust all of the squares within that tool position to reflect the cutter's depth. The number of squares covered by the base of the end-mill is roughly equal to the perimeter of the cutter. For a 0.256 diameter end-mill, this is about $2^8\pi \approx 2^{10}$. To adjust the depth of each square, we must find that square, then adjust the height value. Adjusting the height value is only 4 operations (say); in any case, this is negligable. Finding the square will involve 8 (say) operations for each layer. Descending from the largest square, of size 2^{16} (a gross overestimate), down to the smallest 1-square involves 16 of these transitions. So, adjusting the height of each square will take about $16 \times 8 = 2^7$ operations. There are 2^{10} of these squares to be adjusted, so that the total number of operations for each tool position is 2^{17}. This is exactly the same number of operations per tool position that was estimated earlier for octrees, where that estimate was based on fairly efficient algorithms.

The real value of this method comes by considering the typical situation in which it would be applied: an end-mill cutting to only a limited number of depths. Effectively, the problem is reduced from a 3D problem to a 2D problem. Suppose that the same general strategy is used here as was used for octrees: start at the largest square and recusively check whether that square, or any sub-square, is cut by any of the tool positions. The number of squares that must be visited is equal to the 2D perimeter of the tool path rather than the 3D surface area of the swept volume. The perimeter of a 64 inch long cut, assuming no overlaps, is $2 \times 2^6 \times 2^{10} = 2^{17}$, neglecting the ends. If we assume that the same number of operations are required per square as were required

per cube in the octree case, then the total number of operations required for all of the square tests is

$$(2 \times 48 + 3 \times 16) \times 2^{17} \approx 2^{24},$$

where I lumped together adds, compares and multiplies. The number of operations required for sorting and searching the tool positions will be roughly the same as for an octree. Earlier, I determined this to be equal to 2^{24}. Together, the number of operations is 2^{25}. Compare to the similar example given for octrees. In that case, the number of operations was about 2^{33}, so the quadtree with height should be faster by a factor of about 2^8. In addition, I think that the modulo idea below would apply here more easily than to the general octree situation.

These quadtrees must be converted to octrees, or to some other representation that can be used to generate the 2D rendering. That may take some time, but this idea does appear to be worth implementing as a special case.

2.4 Modulo Building

The idea here is to determine which cubes are in the tool volume and shift them around as the tool moves. The advantage is that only those cubes that are inside the swept volume are ever considered. There is no sorting or searching either, but it turns out that this isn't such a huge advantage. The problem with this idea is that as the tool moves, the way in which it is cut by the grid of the octree changes. The cube positions change in a manner that is similar to arithmetic modulo n, so that's why I call it "modulo building."

This idea is complicated, so consider the analogous situation in one dimension. Rather than octrees or quadtrees, in one dimension we have bi-trees. The tool consists of a line segment, and its position is given by a single coordinate. Suppose thet the tool is defined by the segments $S_1, \ldots S_m$ of length $d_1, \ldots, d_m$, where these lengths are powers of 2. To make things concrete, suppose that there are two adjacent segements, of length 4 and of length 1. As the tool moves through the bitree grid, a segment of length 4 might not be possible. In fact, as the tool moves from right to left, there are four ways that the tool could be broken down:

$$\begin{array}{ll} (4,1) & (1,4) \\ (2,2,1) & (1,2,2), \end{array}$$

where the next step to the left brings the tool back to $(4,1)$.

The idea of the algorithm is to determine all line segments of length 4 that are swept by the tool, then all line segments of length 2, *etc.* Let x_0 be the position of the tool when it is in configuration $(4,1)$. Any tool position that contains a 4-segment must be of the form $x = x_0 + p + 4q$, where $p = 0$ or 1. That is, $x \equiv 0 \pmod 4$ or $x \equiv 1 \pmod 4$. If $x \equiv 0 \pmod 4$, then the 4-segment is at x, and if $x \equiv 1 \pmod 4$, then the 4-segment is at $x + 1$ (*i.e.*, the 4-segment is to the right of this coordinate). Break this down into a table:

x value	4-segment(s)	2-segment(s)	1-segment(s)
$x \equiv 0 \pmod 4$	x	none	$x+4$
$x \equiv 1 \pmod 4$	$x+1$	none	x
$x \equiv 2 \pmod 4$	none	$x, x+2$	$x+4$
$x \equiv 3 \pmod 4$	none	$x+1, x+3$	x

Work through the list of tool positions, making a list of all the 4-segments. This will be fast; if the tool positions are a continuous path (if p is a position, then the next position is either $p+1$ or $p-1$), then division is not needed to dermine the position modulo 4. Actually, division is always by a power of 2, so it's a bit shift and it might not matter.

Next, work through the list in the same way to make lists of 2-segments and 1-segments. There are two things that I would like to do but can't quite see how. First, after sweeping through to generate the list of 4-segments, it should be possible to eliminate many tool positions from further consideration since their entire length has already been accounted for. Second, even if the entire tool length can't be eliminated, how to quickly see whether a new smaller segment is worth considering.

A different perspective on this idea is to start with the largest object contained in the tool (a 4-segment in this example) and list all of the tool positions that contain the same 4-segment. Do the same for 2-segments and 1-segments. In two dimesions this will give an annulus of positions for the tool distributed around the square. This is the same thing as above, but in a different format which may be better.

Considering the 1-dimensional analogue in any further detail doesn't appear to be helpful, so consider the situation in two dimensions.... In fact, I don't see how this idea is an improvement. It's clever and there may be odd situations in which it's better, so it's worth continuing to think about, but I doubt that it will go anywhere. The problem is that there is no way around the requirement that the program visist every 1-cube on the surface.

One way that this might help is as a replacement for the test whether a cube is inside the tool volume. I've been thinking of that test in terms of comparing the vertices of the cube to the tool radius. Maybe it would be faster to do this kind of modulo arithmetic instead. Using the method above, this test requires about 64 operations, so the modulo idea may be faster – modulo arithmetic should be fast.

Consider the situation in 2 dimensions. The tool is defined by the set of squares associated with the tool when it is at $(0,0)$. Just as in 1 dimension, as the tool moves about, we have a list of r-squares for each position. Think of the test that we want to make. We are given a quadtree square and we want to test whether that square is contained in some tool position. We don't want to look at the tool position closest to the square because of the way that it may line up with the quadtree grid. Instead, we need to look at all of the nearby tool positions. For each of these tool positions, we do a bit of arithmetic and look in the resulting table of squares to see if the square in question appears on that list.

Exactly what do we have to do to get this list of the squares in a given tool position? In 2 dimensions, we need to find the remainder after dividing the x and y coordinates by some constant (which is roughly equal to the size of the largest square in the tool). That's only two operations. These point to a pre-made table listing all of the squares found in that tool. Now we need to see if the square in question is on the list. The number of operations to do this is roughly equal to the log base 2 of the number of squares in the table. It looks like we are trading 64 operations under the old way of doing the cube test for more like 10 operations.

My initial thought was to build a table of contained octree cubes for each tool position. That is, work out the octree for every tool position that gives a distinct octree modulo n. This doesn't seem feasible since the number of these distinct positions is so large. If the tool can move 0.25 inch in each of three axes, then there are $(2^8)^3 = 2^{24}$ distinct octrees to be stored, and it's much worse for a multi-axis machine. The overall idea might still work if I can find a way of doing the necessary testing arithmetic quickly.

This idea seems worth investigating. It's complicated, and I'm not convinced that it will be significantly better in practice, though it sure seems like it should be. As a start, it could be added to the quadtree with height algorithm that's used for end-mills on a 3-axis machine. If it works there, then it will work more generally with octrees. If the idea does work for quadtrees with height, then the number of operations required to digest 64 inches of tool path is reduced from around 2^{25} to about 2^{23}. That is downright speedy.

2.5 Seed Fills

The basic idea here is to start with a cube that is known to lie inside the swept volume and let it grow to fill the entire volume. The same general idea could be used with quadtrees too. Obviously, this idea only works if the volume is connected. Since the tool positions are generated by a continuous path, this isn't a problem.

Since the situation in 2D is easier to work with, consider the implementation for quadtrees. Suppose that we have some initial square that is known to lie in the swept area. This could be obtained as the largest square covered by an arbitrarily chosen tool position.

The easiest way to implement this idea is to look at the four neighbor squares of the seed square, then their four neighbors, *etc.* If a square is partial, then look at subsquares. In this way, we are forced to look at smaller squares when we must, but some method of choosing when to look at larger squares is also needed. One way would be to being looking at larger squares when it has been observed that all four subsquares of a larger square are solid. In any case, this implementation isn't going to be much faster (if it's any faster at all). Each square must still be tested against a tool position.

A slightly different idea is to extrude the squares. Extend each of the four sides of the seed square out to the edge of the swept area. This could be done in a series of steps, one step for each newly examined tool position. That is,

if the line extends from p_0 to p_1 and we want to extend p_1 further, then look for a tool position that contains p_1 and allows the line to be extended as far as possible to p_2; then look for another tool position that contains p_2, *etc.* Note that, initially, lines should be considered in parallel pairs so that entire squares of constant size are added at each step. Eventually, one line may stop while the other continues some distance. When this happens the smaller squares must be filled in. The advantage is that we get the maximum benefit from each tool position that's considered, and we will get some 1-squares almost for free along with the larger squares.

I think that this idea does have the potential to be faster than top-down cube checking, but it is complicated, and I think that there may be better ideas. One big advantage of this idea is that it generates a set of faces that bound the volume as a natural side-effect. This may save time later in the rendering step.

2.6 Layered Quadtrees

This is similar to the idea of a quadtree with height, but associated with each square of the quadtee is a list of ranges of heights. So, for a given square, these ranges indicate that the object is solid above that square in the range $[z_{i0}, z_{i1}]$, for $i = 1, \ldots, n$. The has the advantage that it allows totally arbitrary shapes. I haven't thought very hard about this, but I don't see how it would be any faster than an octree, and it would be more difficult to reduce the memory consumption.

2.7 Tool as Surface

The idea here is to work directly with the surfaces that bound the cutter. As the tool moves, it leaves it's bounding surface behind. An advantage of this is that there is no need for an octree; everything is done in terms of the surfaces needed for rendering.

To see how this compares speed-wise, consider an end-mill with diameter 0.256. It is bounded by (roughly) $2^9 = 512$ vertical strips, each 1 unit wide. In fact, there may be considerably more of these strips, (or perhaps less) since the tool volume is considered to be made up of cubes. I am also neglecting the 256 strips along the circular bottom of the cutter.

Each of these strips must be adjusted for the tool position. They are bounded by four corners. The z-coordinates at the top and bottom are the same for all strips, but the x and y coordinates are different from strip to strip. As an (under)estimate, assume that 4 operations must be made to adjust each strip. For 512 strips, that's 2^{11} operations for each tool position.

As the tool moves, for each position that the tool visits, these 512 strips are stored, along with information about which side of the strip is the inside (the cut away portion). The standard example assumes that there are 2^{16} tool positions. If each tool position requires 2^{11} operations, then the total number of operations over the entire tool path is 2^{27}.

This seems to compare favorably to the octree estimate, but we're only partway done. This massive list of rectangles (2^{25} entries) must be examined to remove rectangles that in the interior of the swept volume before the object can be rendered. I don't know how long that would take; moreover, a cutter that is not a simple endmill will have many more faces than 512. One way to reduce the number of faces that end up in the initial list is to consider various short moves that the cutter may take, and not include the faces that are clearly convered by a later or previous move of the cutter. This could reduce the number of faces enourmously.

Not only is this long list of rectangles a computational hurdle, but it takes a lot of memory. Each rectangle requires 6 coordinates for two vertices, or 12 bytes. Call it 16 bytes since there will be some pointer overhead. Storing 2^{25} of these rectangles will require 2^{29} bytes or 512 megabytes. However, that's a bit of an overestimate since we only need 3 coordinates, and a flag to indicate which of three planes that the rectangle is in, plus two shorts for the extent of the rectangle. Moreover, because the rectangles are orthogonal, they could be sorted into three seperate sets: one for each orthogonal axis. That brings us down to five shorts (instead of six). Within each of these lists, the rectangles can be sorted according to which parallel layer they lie in. The reduces the memory needed for one of the three coordinates to almost zero, and the rectangles can be sorted again on their left/right position. All together we can probably get it down to 8 bytes per rectangle, which means that the whole surface will require 256 megabytes. But that's still a lot.

A related idea is to try to detect the faces as the tool moves. This could be done by trying to detect straight lines in the tool's motion, but doing this would be difficult (or pointless) for complex paths. The important thing is that as the tool moves, the perimeter of the tool traces two lines parallel to the tool's motion (something like cutter comp). By following the tool's path, at an offset equal to the radius, you could get a 2D curve that bounds a cross-section of the volume. This also seems fast (if it works), though there is still the issue of depth and dealing with included sides. I'm not sure if there's any advantage (or if this is even workable) for complex cutters.

2.8 Analytic Methods

My instinct is that this is hopeless for anything other than simple objects, and that it would only work if the shapes are defined in terms of lines of G-code. For very simple objects, this would be super fast, but it's extremely complicated and would probably be ultra slow for more complicated objects. Because I want to work in terms of pulses and not G-cuts, this is unlikely to work.

One way that I could use analytic methods with pulses is, for each pulse train, build a spline-based surface as the tool moves. The control points would be the points visited by the tool as it moves. Obviously, this has many (!!) more control points than are necessary. Once the tool has completed its path, revisit the surface and try to remove control points without messing the surface up too much. This might be possible because (a) the effect of control points is

local; for each point that's removed, we only have to look at nearby points to see how much difference the removal makes; (b) it may be possible to determine the error analytically by integrating to obtain the volume between the surface with the test point and without the test point.

Another analytic method is to track the tool path with a bezier curve. As the tool moves, there are two curves, one on each side (like cutter comp). This curve could be stored very efficiently, and from the curve, the cutter's shape defines a surface that extends above the curve. Note that for multi-axis machines, this curve must have a vector defined at every point indicating the angle of the cutter. This vector function could also (?) be modeled as some sort of bezier curve.

It remains to see how these surfaces would be rendered.

2.9 Cross-sections

The idea here is to work with a stack of many parallel cross-sections of the object. These could be horizontal or vertical, but I suspect that horizontal cuts would be more complicated – somewhat more complicated to program, and the shapes of the cross-sections would be more complicated and hence slower. What this amounts to is a set of slices of the object, each only 0.001 inch thick. These slices can be thought of as regions of the plane, and they can be represented very efficiently.

This idea may make rendering the object more complicated. No octree is ever generated. In fact, generating an octree would (I think) be difficult. Determining whether an entire large cube is inside the volume requires that each slice of the cube be examined, and for each slice we would need to check whether the entire square formed by the cross-section with the cube is contained in the region. This is bound to be slow.

One way to represent an area of the plane is by following the path of the perimeter. At each step, there are only three choices: turn left, turn right, or go straight. This choice could be represented with two bits, and the fourth value could be used as an escape so that long runs of the same move (or other patterns) could be run-length encoded. There are some complications though. A cross-section may consist of several disjoint regions so that multiple perimeters are required per slice. Also, a region may have a hole, and the hole could even have a solid part inside of it.

Memory usage of this idea is managable. Even if an entire byte is used for each step, the number of steps is roughly equal to the surface area, and the standard assumption that I have been making is 64 Meg square units for the surface area. Run-length encoding would allow more typical objects to be encoded in much less space. For deep storage, I suspect that standard compression methods would work too.

Cutting an untouched billet of material with this method is super fast. The problem is how to deal with adding cuts to an object that's already been cut in complicated ways.

Suppose that we start with an uncut billet of material. Note that the assumption that the material is uncut implies that the tool path does not double back on itself. There are a fixed number of cross-sectional slices of the tool, and they have the same shape no matter where the tool is in the material – the shape does change with the depth of the tool, but in a trivial way. The number of these slices is equal to the diameter of the tool. For each tool position, we need to merely insert each of these slices into the path of the material's cross-section. If the tool is 2^8 thick, then this will involve about 2^{10} operations per tool position, assuming 4 operations per slice. Under the original octree method, there were 2^{17} operations per tool position, and I may be able to reduce this to something like 2^{14} with the modulo idea. Even so, this is at least 10 times faster, and may be 100 times faster. In fact, this may be faster than the quadtree with height, even in the case of an end-mill on a 3-axis machine. In that situation, the quadtree with height was estimated to take about $2^9 = 2^{25}/2^{16}$ operations per tool position, so the two methods are certainly comparable.

In fact, the number of operations required per tool position is generally going to be even smaller. By looking at the preceeding step(s) and the next step(s), we should be able to reduce the number of slices that need to be adjusted to only a few per tool. In that case, the number of operations per tool position is closer to 2^6 than to 2^{10}. At that speed, these calculations are nearly trivial to perform.

Note that these slices could also be represented with quadtrees. The amount of memory required to do this is comparable to the perimeter method, and may be much less if I use a compression method similar to that proposed (below) for octrees. Under the perimeter method, two bits are required for each step. Assuming 8 inches of perimeter per slice, this works out to $2^3 \times 2^{10} = 2^{14}$ bits or 2^6 bytes. Most likely, I would use an entire byte per step so that 2^{14} bytes per slice would be needed. Under the quadtree representation, a perimeter of 8 inches has 2^{13} 1-squares. If I used compressed quadtrees, then the list of 1-squares will require 2^5 bytes. With overhead, and the larger squares, this will probably require 2^6 bytes, which is the same as the perimeter representation.

Representing these slices as quadtrees at all times is not the way to go since that defeats the advantage of the perimeter representation in terms of the speed with which it allows cuts to be made. However, there may be times when converting to a quadtree representation is useful. There may be faster ways to do this, but here's one way. We want to include every quadtree square that's on the interior. Testing each square against the perimeter will be far too slow. Instead, add to the quadtree all of the 1-squares that make up the perimeter, along with some kind of notation as to which sides of these squares are on the interior. Then work inward. When considering whether to add a square, all that we need to know is whether this square crosses the perimeter. So, it may be useful to keep a copy of the quadtree for the perimeter only. Observing whether a potential square crosses the perimeter should be easy: if the square in the perimeter-only quadtree is non-null, then it contains *something* and that something must be a perimeter square.

Another conversion that could be useful is to convert a set of slices to a

single QTD (quadtree with depth). One way to do this is as follows. Imagine all of the slices lying side by side to form the volume. Start walking along the perimeter of all of the slices at the same time, observing if the height changes. If there is no undercutting, then you could convert the entire slice set to a single QTD; the problem is what to do if there is any undercutting. As the program walks the perimeters, not the first point at which any undercutting occurs, and terminate the walk (and the QTD specification) at that point. This idea could be used to convert a set of slices to a set of QTDs that fully specifies the workpiece, even with undercutting or other arbitrary features. The basic idea is this. Create a QTD from above the set of slices (depth relative to the z-axis) as described, terminating when undercutting occurs. This accounts for some portion of the perimeters of all of the slices. Now, "project" the slices to some other plane and create a new QTD from that side. Basically, this peels away layers of the workpiece so that any undercuts can be exposed. The trick is finding the right planes to which to project, and there's no reason that the plane as to be orthogonal to the coordinate axes. Maybe, by judicious choice of a series of these planes, everything could be done using QTDs, even multi-axis machines and weird cutters.

This *looks* like it is far and away the best method, but there are unexamined issues. Taking the intersection of two of these objects is not trivial; neither is converting this to a surface representation for rendering.

Consider the intersection question. This is the same as asking how to take the intersection of two plane regions. Once that question is answered, then we need only apply the solution to every slice of the object. It appears that this can be accomplished by walking along the two paths (i.e., the paths that the define the regions to be intersected) and finding the times when the two paths cross. The number of such positions that must be visited, over all the slices, is equal to the surface area of the object, or somewhat more since there are two objects. There shouldn't be more than a few ($c = 8$?) operations per step, so the number of operations to perform the intersection for the entire volume is $c \times 2^{26} \approx 2^{30}$. This is roughly equal to the number of operations to build an octree, after using the modulo trick. So this appears to be somewhat faster than octrees, but not dramatically faster. However, I suspect that for most typical objects, the intersection could be determined much more quickly since much of the region's perimeter will be straight lines. Moreover, there may be faster ways of taking the intersection. Also, a lot of the time I could take the intersection of several of these objects at one time, and that is bound to reduce the total time required.

If the slices are represented by quadtrees, then taking the intersection becomes much easier. Think of the intersection as a difference and visit each of the squares to be removed. But converting the perimeter representation to a quadtree representation looks (?) difficult and using quadtrees when determining the swept volume isn't much better than using an octree – the modulo idea might make it a bit better, but it's a far cry from working with perimeters.

It appears that the idea of walking around the polygon has been investigated by others. In particular, I found a reference to "Computational Geometry in C"

(2nd ed) by Joseph O'Rourke, Cambridge University Press, 1998, ISBN 0-521-64010-5 Pbk, ISBN 0-521-64976-5 Hbk. See also http://cs.smith.edu/ orourke/. I ordered this book.

There is another idea, which probably won't work for my extremly complicated polygons. The comp.graphics FAQ says "Unlike intersections of general polygons, which might produce a quadratic number of pieces, intersection of convex polygons of n and m vertices always produces a polygon of at most (n+m) vertices. There are a variety of algorithms whose time complexity is proportional to this size, i.e., linear. The first, due to Shamos and Hoey, is perhaps the easiest to understand. Let the two polygons be P and Q. Draw a horizontal line through every vertex of each. This partitions each into trapezoids (with at most two triangles, one at the top and one at the bottom). Now scan down the two polygons in concert, one trapezoid at a time, and intersect the trapezoids if they overlap vertically." The problem with this (for me) is that most of the time there will be too many of these trapezoids.

2.9.1 Scanline Cross-sections

A variation on the idea above is to encode the cross-sections in terms of scanlines rather than in terms of the perimeter of the region. Think of the cross-section as being in the (x, y)-plane, with y as the vertical axis. For each y, the intersection of the line $y = \text{const}$ is a set of line segments, and generally there will be only one or very few of these line segments. So, for each plane, we need a list of line segments, where this list is as long as the cross-section is tall. For instance, if the material is four inches high, then we need 2^{12} lists of line segments. These lists of line segments can be encoded as a set of shorts. The first short is the length of the line segment, then their meaning alternates: first the length of the gap, then the length of the next line segment. The list is terminated by a short equal to zero.

Memory usage is likely to make this idea unmanagable. If each scanline has a single line segment, then each scanline requires 4 bytes (2 shorts). So, the entire plane (four inches thick) requires $2^{12+2} = 2^{14}$ bytes. If the workpiece is eight inches thick, then we need 2^{15} of these slices, for a total of $2^{14+15} = 2^{29}$ bytes, or 512 megabytes. That's not really manageable. On the other hand, in most cases, the scanline could be more than 1 pixel wide. That is, if a stack of adjacent lines start and end in the same place, then they could be encoded as a single rectangle. This would save a lot of memory in thoses cases.

Taking the intersection of two regions defined in this way is conceptually simple. Work from top to bottom, looking at each pair of scanlines. Take the intersection of the two scanlines and store it as the intersection. The number of operations, per scanline, to do this will be about 8. That's 3 ops to increment the scanline index, 3 ops to pull out or store the data, and two ops for comparing. The real total is probably more like 20, once various overhead is taken into account. Assuming that 8 is the right number, and assuming 2^{12} scanlines per slice, that's 2^{15} ops per slice, for a total of 2^{30} ops for the entire volume. That's comparable to octrees.

My assumptions for memory consumption and, as a result, computation time may be overly pessimistic, so this idea is not out of the running, but it doesn't seem like it will pan out.

2.10 Compressed Octrees

There is a way to represent octrees very efficiently so that they only require roughly one bit for each cube. Using this method of representation, memory is no problem, but navigating the representation would be too difficult. This may be used as way of putting octrees into "deep storage", perhaps as a way of letting the user examine the effect of individual lines of G-code. This could be done by storing an octree for the swept volume of each G-cut. This may make it possible to adjust the rendered image on the fly as the user modifies the input G-code. To do this, the internal state of the machine must also be stored for each line of input code.

Here's how to reduce the storage to nearly one bit per cube. Group the cubes of the same size together into seperate lists. First, a list of all cubes that are 256 units (say) on a side, then all cubes that are 128 units on a side, 64 on a side, etc. The top (largest) cubes will need some assocated data about where theses cubes are in relation to each other, and what their coordinates are, but that's a modest amount of additional memory.

The first list is organized as an integer for the length of the list of cubes, then a series of bits, one bit for each sub-cube (not the parent cube). So, if the first list has n cubes, then there will be $8n$ bits in that list. If the bit is 0, then that sub-cube is entirely empty; if the bit is 1, then that sub-cube is at least partially full; and if all eight bits for a given parent cube are 0, then that parent cube is completely solid. This trick is confusing, but it works because it will never happen that all eight sub-cubes are empty, for in that case the parent cube would not have appeared on the list to begin with.

Based on the first list of parent cubes, we know how many subcubes there must be in the sub-list of smaller cubes, and we can assume that they are ordered in the same way that they appear in the parent list. A nice feature of this is that the data can be allocated in whole bytes. Each cube has eight sub-cubes.

This could be compressed somewhat further by running this data through Lempel-Ziv. I'm not sure how much it would compress, but it will compress some because there will be lots of strings of eight zeros (for every totally full cube). Also, my guess is that there will be more 1's than 0's for the rest of the data. Finally, the data is organized so that geometrically near cubes tend to appear together in the data. This should also make the data more compressible. I can't guess whether it would compress only 10% or by as much as 80%, but it's worth trying.

With some slight modification it may be possible to navigate octrees stored in this format relatively efficiently. Suppose that each layer is represented with

```
public class OctLayer {
  public byte[] value;
```

```
}
```

and that we have a series of layers given by an array of `OctLayer` objects in `layer`, and that we want to traverse the tree. If we are at the parent cube represented by `layer[i].value[j]`, then, to examine the sub-cubes, we would need to look at `layer[i+1].layer[k]`, for some `k`. The problem is determining `k`. With this form of representation, we would need to look at all `layer[i].value[r]`, for `r<i`, so that the number of 1-bits could be counted to determine `k`.

For an ordinary traversal, this is not a problem. A count of the number of bits seen as the traveral progresses could be saved. This is likely to be somewhat slower than following pointers in a more standard implementation of a tree, but it might be fast enough given the huge savings in memory.

As a compromise, I could store an integer with each of these bytes, where the integer is the offset in the next lower layer to the first of the eight sub-cubes. By doing this, rather than using only 1 bit for each cube, we will be using an average of 5 bits per cube, assuming 4 byte integers. This is still a huge memory savings (5 bits versus at least 4 bytes, and typically more like 36 bytes). Using a resolution of 1/1024, an object with 64 square inches of surface area could be represented with about

$$64 \text{ Meg cubes} \times (5 \text{ bits/cube}) = 40 \text{ Meg bytes},$$

which is very little. At a resolution of only 1/128, the representation would take less than a Meg.

This would still be somewhat slower than traversing a tree based on standard pointers, but my guess is that the slowdown would be small.

A variation on this idea that takes somewhat more memory, and should be faster is to allocate an integer of each sub-cube so that it's not necessary to count the individual bits in each parent cube. That would require 4 bytes for each cube. A strategy that wouldn't be quite as fast but would use less memory is to allocate only a short (2 bytes) for each sub-cube, which won't be enough to handle the situation where the next layer has more then 32K entries. To solve that problem, provide a list of indices at which the count "rolls over."

However, none of these schemes lends itself to modifying an existing octree. When a new cube is inserted into the list, the offsets for all of the cubes that follow must be updated – it's like trying to insert an element into an array rather than a linked list. These ideas might work very well for working with a fixed octree, but they're not well-suited to modifying the octree. On the other hand, because this method takes so little memory, it may be possible to rebuild the tree from the ground up whenever it must be modified; this representation takes so little memory that holding two (or more) copies at once is no big deal.

An octree is nothing more than a set of pointers, and the reason that an octree takes so much memory is the amount of space required by the pointers. It might be possible to work with some kind of pseudo-pointers, like indices into an array.

Finally, the tail end (small cubes) of the octree does not need to take as much memory as the nodes for the larger cubes. In particular, the 1-cubes pointed to by the 2-cubes do not need to be represented by full-fledged objects. Rather than having a 2-cube hold an array of eight pointers to seperate 1-cubes, they could hold a single byte, where each bit indicates whether the corresponding 1-cube is solid or empty. In fact, due to the way that Java allocates memory, it may make more sense to let the 4-cubes use a 64 bit value to indicate the state of all of the contained 1-cubes. The 1-cubes and 2-cubes will vastly outnumber all of the larger cubes, and this may save enough memory on its own. The downside is that they must be handled as a special case.

2.11 Miscellaneous Issues

I've mentioned this idea in passing, but it may help many of these algorithms to pre-build various short tool paths; e.g., a little octree for when the volume moves by one step in the x-direction. The longer tool paths can be built up of these shorter pre-calculated octrees (or other data structures). I'm not sure how long to make these mini-paths, but they could speed things up a lot. They shouldn't take much space either since they will (obviously) be very similar. The octrees (or quadtrees) for each of these mini-paths could share large parts of the octree data – I mean that octree pointers in different mini-paths could point to the same sub-tree. The issue with this idea is how to efficiently generate the octree for a shift of an octree by an arbitrary number of cubes at the finest resultion, and that's related to the modulo building idea.

I should bear in mind the quirks of the CPU. I don't want to write a seperate program for every machine, especially since that's hard to do with Java, but things like pipelining, local compilation and memory caching mean that counting ops is not the whole story. The operations need to be done near each other in the program to get maximum possible speed.

Many programs have multiple passes over the same material; e.g., cutting a pocket to depth in several steps. It may be worthwhile to detect these occasions since it's only the deepest cut that really matters.

2.12 Multi-axis Machines

Generally, in the discussion above, I have tacitly assumed a 3-axis machine. For some of these ideas above, there's no way that a multi-axis machine would work. For comparison, here's a list of the methods already discussed.

Method	**Memory**	**Ops**	**Note**
Octree, top-down	1.4 Gig	2^{33}	Basic algorithms
Quadtree w/height	$<$ 64 Meg	2^{25}	Only for 3-axis end-mill
Octree, modulo	1.4 Gig	2^{30}	Not fully fleshed out
Octree, seed fill	1.4 Gig	??	Complicated; rough idea only
Layered Quadree	??	??	Seems unlikely to pan out
Tool as Surface	512Meg?	2^{27}	Dealing with all of these surfaces is hard when rendering.
Analytic Methods	??	??	Totally out of reach?
Cross-sections	64 Meg	2^{24}	Ops value is an estimate.
–intersections		2^{30}	Rough guess.

These memory values are somewhat approximate, and well on the conservative side. In particular, if the compressed octrees can be made to work so that a pointer-based tree is not needed, then the octree memory values become almost trivial. The operations value is for the cut made by 64 inches of travel by the tool, and does not count time needed for performing intersections.

So there are three strategies that seem most likely to pan out for multi-axis machines: octree with some kind of modulo tweak, working with the tool as a surface and cross-sections. Of the three, cross-sections is the most easily amenable to multi-axis machines. It also has the advantage of modest memory consumption.

An idea that might help with any of the strategies is to work as though the tool is on a 3-axis machine, then rotate the resulting cut. This only works when the angle of the tool doesn't change over a long distance, but in those cases, which are probably the most common, it could help. In particular, working with an end-mill at constant angle would allow me to use a quadtree with height for most of the calculation. When the quadtree is converted to an octree, the angle of the tool could be taken into account.

For the octree idea on a multi-axis machine, both sorting to find the closest tool position and testing whether a cube is contained in the tool volume will be more complicated. As usual, the base case is an end-mill, but now it's at an angle. Considering this as a tilted cylinder isn't much more complicated, but it will take somewhat more operations. The modulo idea should still work on multi-axis machines, though the tool has more degrees of freedom, which means that any lookup tables will be longer.

The tool as surface idea might be preferable since it handles complex cutters at odd angles easily. The problem is that as the cutter becomes more complex and the angle changes, the number of faces that make up the surface of the tool is much larger. The estimate of 2^{27} for the number of operations is based on the assumption that only 512 faces are needed to bound the tool. That is a gross underestimate for oddly shaped tools at an angle. However, the 2^{27} estimate is based on a relatively crude (exhaustive) list of all faces generated by the tool. In any case, I suspect that the biggest hurdle here is the need to deal with all of these faces at the rendering step, and I have not yet considered that stage.

The cross-section idea can also handle oddly shaped cutters at arbitrary

angles without much difficulty. In fact, they should take no more or less time than an end-mill on a 3-axis machine. Provided that these can be rendered efficiently and that my estimate of the time required to perform intersections is accurate, this looks like the best approach for the general case.

2.13 Oddball Cutters

Most of the time I have been thinking of an ordinary end-mill, but the program must handle arbitrary cutters. The feature of an ideal end-mill that make it easy to deal with are that it is a simple cylinder that extends up to infinity.

First, we need to distinguish the shank from the cutter. Typically the shank will be smaller than the cutter, but allowing it to have arbitrary diameter is one way to handle the detection of collisions of the tool holder with the workpiece. Next, we need to break the tool into pieces (convex?) that can be managed. Some canonical shapes appear in the figure. The "dimpled" cutter is like an end-mill, but with a concave dimple in the bottom. Cutters exactly like this don't seem to exist, but there are cutters with similar features – as the cutter moves it acts like an ordinary end-mill, but over a (short) plunge, it leaves a raised dimple. Rosette cutters are certainly used in woodworking.

Unless I end up using the cross-section method, it's certain that different cutters will use different code. I've already discussed the fact the quadtrees work well with 3-axis end-mills. Even if the shank of the end-mill is taken into account, a quadtree could still be used (at least on a 3-axis machine). The maximum height of the cut would be implicit.

A modifed quadtree could be used for a slit saw too. Such a tool can not cut vertically (the program should check for this). The issue is that the saw could come into the side of the material at multiple heights. Therefore, the quadtree needs to allow multiple heights for each square, where each of these values represents the depth of the cutter when it enters into the material. As usual, this idea only works for a 3-axis machine. However, for a slit saw, changes to the angle of the cutter while engaged in the material are not allowed (just as changes to the height are not allowed). So, even on a multi-axis machine, the quadtree could be used, followed by rotation of the volume.

Breaking the tool into the shank part and the cutter part is one way to detect

collisions. Even with a simple end-mill, the shank could be defined to coincide with the tool holder. You now have two tool-like objects: the endmill and the shank. Treat the shank just like the cutter (it would be simplest to assume that it's either a cylinder or a cube) and allow it to sweep out a volume. If the resulting swept volume has any intersection with the remaining material, then there has been a collision. Even for shanks that are smaller than the cutter, it is possible to have a collision. If the end-mill in the figure with a small small shank is plunged into the work, then moved horizontally, then there will be a collision after moving somewhat less than the radius of the tool. Note that the order in which this testing is done matters. If we wait to test the swept volume of the shank until after the swept volume of the tool has been removed, then we could miss collisions if the colliding material was cut away. Likewise, we can't test the swept volume of the shank before the cutter acts because spurious collisions will be detected.

Unless I use the cross-section method, breaking the tool up in this way is how I will probably have to treat general cutters. This may not be necessary for a ball-mill, but even in that case it may be the fastest way. Treat the ball-mill as an ordinary end-mill, plus the hemispherical cap, and run through the cutting path twice. Treating the end-mill part should be fast if the quadtree method is used on a 3-axis machine, and even on a multi-axis machine a cylinder should be easier to manage. The second pass for the cap will be faster since the cap is short, so that most octree cubes will fail to intersect on a height test.

There's nothing especially tricky about either the sphere mill or the dove-tail cutter. They'll need somewhat unique code for maximum efficiency of the intersection test, but they can be treated in the same general way as any other cutter under the octree methods. Quadtree-type shortcuts won't work.

It is "complex" type cutters that are the biggest problem. What makes them tricky is that they are concave and they can cut in any direction. This means that any kind of shortcut to octrees, like the quadtree with height won't work. Moreover, testing whether cubes are contained in these volumes will be trickier. It's trickier because determining which tool position to test against is harder, and because the test test of a cube against a fixed tool volume is harder.

For some of these complex cutters, I should be able to break it down into well-understood shapes. For instance, if the profile is an arc of a circle or a straight line, then specially tuned code should work for that part. It's more arbitrary shapes that cause the most trouble.

With octrees, the modulo idea does seem to extend to arbitrary shapes and multi-axis machines. If that's the case, then the seriousness of some of these concerns goes down.

Again, if I use the cross-section method, then cutters of arbitrary shape are much easier to deal with – that's a huge advantage of the cross-section method. The shape of the cutter and whether or not it's a multi-axis machine makes very little difference.

Treating the tool as a surface has advantages similar to the cross-section method, although I doubt that this method will pan out.

An issue that I haven't given much thought to is detecting unsupported

material. If the workpiece is undercut, then you could end up with hunks of material that are unsupported and would be either chewed to bits or flung out. This is dangerous and you could damage the workpiece.

3 Rendering

There are three major ideas that are still in the running: octrees, cross-section and surface representation. Layered quadtrees and analytic methods seem unlikely to work out.

It seems that there are two major stages to the rendering process: (1) converting the volume representation to form that can be rendered in 2D; and (2) drawing and shading it to the display. An additional step that may be necessary is reducing the resolution, but it's not clear where that should occur.

3.1 Converting Octrees

I have already implemented a rough alogorithm for this case. Each cube of the octree is visited, and each of the six sides is added to a list of surface squares. If a square appears twice, then it is removed from the list since it must be interior to the volume.

The time required is proportional to the number of cubes. Each face of each cube requires about 4 operations to copy the face specification and there are 6 faces, so that's about 32 operations per cube. If the surface area is 2^{26}, then this is 2^{31} operations to convert from octree to a list of bounding squares. Comparing 32 as an estimate for the number of operations to my code shows that 32 is a gross underestimate. There are a lot of subroutine calls and tests for toggling, so that a more honest estimate seems to be 256 operations per cube, which gives 2^{34} operations total. That's too much, and I still think that 256 may be an underestimate. On the other hand, by traversing the cubes more cleverly and keeping track of common faces, I should be able to avoid a lot of the checking necessary for toggling.

3.2 Converting Cross-sections

This is trickier than converting octrees. Think of the cross-sections as being parallel to the (x, z)-plane, with z as the vertical coordinate. Individual slices provide 1-squares in the (x, y) and (y, z) planes, but the squares in the (x, z) plane must be determined by looking at the slices on each side. Moreover, nearly all of these squares are 1-squares. They must be aggregated into larger squares in an extra step.

One way to proceed is to convert the set of slices to an octree. However, this seems slow. The only way that I can see to do this is to convert each slice to a quadtree, then all of these quadtrees to an octree. *Then* use the same algorithm as above to obtain the bounding surface.

On the other hand, an advantage of these slices is that most of the surface 1-squares can be observed almost trivially. These small 1-squares are numerous, but listing them is easy. The squares (or strips) in the (x, z) plane can be obtained by checking whether, for each x-coordinate, there is a drop-off from one slice to the next. In theory, there should be only (say) 32 operations per step of the slice perimeter. These steps along the perimeter give the surface area so that converting from slices to the bounding surface should take about the same amount of time as converting from the octree representation.

I may still need to aggregate these 1-squares. I say "may" because it may be best if all of the squares in the bounding surface are 1-squares. As discussed elsewhere, working strictly with 1-squares that are smaller than a pixel may simplify and speed up the final rendering step.

3.3 Surface Representation

The issue here is not how to convert to a bounding surface, since that's implicit in the way that the volume is represented; rather, it is how to remove faces that are interior to the volume. The initial number of interior elements could be vastly larger than the number needed only for the surface.

Above, I estimated the list of faces to have 2^{25} elements. That's almost certainly an underestimate; 2^{29} or 2^{32} is probably closer to the truth. There may be fast ways of eliminating redundant faces, but it's hard to see how this could be fast enough. Moreover, the memory requirements for this idea are large.

3.4 Memory Requirements

All of the methods under consideration reduce the object to a set of bounding faces stored as polygons (squares or rectangles). If the polygons are orthogonal to the coordinates axes, then each rectangle requires three shorts for each of two corners, or about 16 bytes. I may be able to reduce this somewhat, though not dramatically. If there are 2^{26} faces then we would require 2^{30} bytes to store all of these faces. That is *far* too much memory. If the resolution is reduced to 1/128, then there are only 2^{20} faces, which would require about 16 megabytes to store. That's feasible, provided that this many faces can be rendered fast enough.

3.5 BSP Trees

The best way to manage the surface elements so that they are drawn back to front is with a BSP (binary space partition) tree. The algorithm to create the tree works like this:

1. Pick a reference plane (which should contain a surface element).
2. Split the remaining faces into two sets, depending on which side of the reference plane that they are on.

3. For each of these two sets, go back to step (1).

Assume that there are $N = 2^{26}$ faces. If each node gets only one surface element (i.e., no two surface elements are in the same plane), and the elements are perfectly distributed, then the total number of nodes required is n, where $N = 2^{n+1} - 1$, and the tree is n layers deep. I got this because, for a tree n layers deep, there is one node on the first layer, 2 on the second layer, 2^2 on the third layer, *etc.*, for a total of

$$1 + 2 + 2^2 + \cdots + 2^n = 2^{n+1} - 1$$

nodes. Thus, $n \approx \log_2 N$. For each layer of the tree, all of the surface elements must be examined (less those that have been taken out of the running as reference planes). So, each surface element must be examined $\log_2 N$; for N nodes, there are then $N \log_2 N$ total examinations, each of which involves a comparison and some copying. This count is an overestimate because, as the tree is built, surface elements will be put into nodes and no longer need to be considered, but it is a gross underestimate because the tree is unlikely to be perfectly balanced, and some faces will be split by the refence plane. Many branches will be empty. In any case, using $N \log_2 N$ with $N = 2^{26}$ means that there will be about 26×2^{26} examinations, which is roughly 2^{31} examinations. Each of these examinations will require roughly 16 operations, so that the total number of operations to build the BSP tree is about 2^{35}. This is too much.

Also, the resulting BSP tree will be highly dependent on how the reference planes are chosen. Choosing reference planes that are near the perimeter of the object will reduce the number of planes that are split, but will tend to make the tree unbalanced. On the other hand, choosing reference planes near the core of the object will make the tree more balanced, but more of the surface elements will be split.

The largest example I have tried on the existing code (hextest04) has $46,386$ 1-squares, $16,530$ 2-squares and $5,202$ 4-squares before the BSP is built. After the BSP is built, due to splitting there are a total of $135,645$ squares, which indicates that almost every one of the larger squares is split into 1-squares – however, I pick the reference planes more or less randomly. The number of nodes is $26,016$ so that each node has 4 or 5 squares. If we assume that there are $N = 2^{17}$ squares, then $N \log_2 N \approx 2^{21}$, and 16 operations per square implies that the total number of operations is 2^{25}. In fact, the building process seems to take close to 1 second, so my estimates must be low; however, the current implementation is poor.

3.6 Rendering a BSP

There are two steps to the rendering process: (1) traversing the BSP tree (or determining the back to front order in some other way); and (2) drawing the surface. Traversing the BSP should take about 4 operations per face. Each node will typically have multiple faces, but some branches will be null. This includes the operations needed to get each face out of the data structure.

When drawing each face, there are two situations: (1) only a single corner must be considered because the face is smaller than a pixel; and (2) all four corners must be considered. Something to consider is how large a face needs to be before it becomes worthwhile to allow four corners.

As an aside, an example program (hextest04) requires $135,645$ faces ($\approx 2^{17}$) over $26,016$ nodes. It seems to take about 1 second for each frame. Assuming 2^{30} operations per second, that's 2^{13} operations per face.

Consider the situation of only one corner and a single pixel. The corner must be (parallel) projected to the plane, taking into account the position of the observer, then some adjustments are made for zoom and to shift the coordinates within the window. Finally, the shade is determined and the pixel is set. Parallel projection requires two dot products, taking 8 operations for both. Adjustments for zoom and placement are another 8 operations. As things stand now, I am determining the shade intensity very inefficiently, and it takes upwards of 100 operations. In theory, I should be able to do this with a single lookup, provided that I precalculate the dot products, at a cost of only 4 operations (say). It's difficult to say how many operations are required to set the pixel since that's a Java call. It may be possible to do that more efficiently than what I am doing now.

All together, to set a single pixel based on a single corner takes about 2^7 operations (or even 2^8) now, but I think that I can get it down to more like 2^5, plus the time required by Java to set the pixel. However, as noted above, the current implementation is observed to take more like 2^{13} operations per face. Some of this is due to the extreme redundancy of the current implementation, but it's still a high number.

When there are four corners, we have about 16 operations for projection and position adjustment for each corner, or 64 operations total. I should be able to reduce this somewhat by tuning the dot product and using the fact the some of the corners have coordinates in common. Drawing a polygon instead of setting a single pixel will be more costly than setting a single pixel. So, the cost for a 4 corner square is roughly 64, plus cost of polygon drawing. My *guess* is that drawing a 2-square is considerably less efficient than drawing 4 1-squares; drawing 16 1-squares is probably comparble to drawing a single 4-square, and an 8-square is far more efficient than 64 1-squares.

Assume that each face/pixel requires 2^5 operations, and that we want 8 frames per second. If the machine can perform 2^{30} operations per second, then we should be able to handle an object with 2^{25} faces, which is about the same number of faces that I expect these objects to have in the worst case. Using the graphics card may change these conclusions.

3.7 Shading

A surface normal should be assocated with each face. The simplist thing to do is to use the actual normal, but this will lead to some oddities. Cuts made at an angle leave behind jagged edges and the faces that made up these edges will have two or three different shades when they should all be the same shade. One

way to fix the problem is to fit more general polygons to these faces. That would save memory, both because the polygon would be larger, and also because it would span jagged corners. Another way to fix the problem is to smooth the normals more directly. Let the pseudo-normal (p-normal) to a given face be defined as the (weighted?) average of the normals to the adjacent faces.

One of the features of these normals is that they should take on a limited number of values. This allows the shading step to proceed without taking a dot product. These normals would be represented as a single byte (say) that is an index into a lookup table of the actual vectors. As the viewer's point of view changes, the necessary dot products of all of these vectors could be calculated once. The shade of each polygon is then determined by a lookup into this table of dot products. This eliminates a very long calculation ($> 2^7$ operations?) to just a couple of operations.

Working with the surface-based representation when cutting would make smoothing these normals very difficult. Because there is no natural structure to the list of bounding faces, finding the faces that are adjacent to a given face is hard.

It's somewhat fussy with the octree representation, but doable. One way is to do what amounts to a seed fill around the surface. Identify a single exterior face, then look at adjacent cubes and determine which of these is part of the continuation of the surface. As you proceed, each face must be flagged as having been visited. This is because the entire tree must be traversed to verify that every face of every cube has been visited; if the object consists of disjoint pieces, then you could miss part of the surface otherwise. As a very rough estimate of the amount of time required, each cube will require 64 operations, for a total of 2^{32} operations over the entire object. The memory requirements could be substantial. To avoid visiting each face several times, we need a list of the faces already visited for each surface element. In theory, these should be allocated then released as all of the adjacent faces are visited, but all of this trickery will cost considerable time.

It appears to be somewhat easier with the perimeter representation. Think of the surface normal as the cross product of two vectors; one is in the plane of the perimeter (roughly the tangent to the perimeter) and the other is perpedicular to that plane. Finding the tangent is easy: just look forward and backward along the perimeter. The other vector can be observed as the surface is built. The adjacent slices must be considered as the surface faces are determined, and noting how far up/down the face jumps at these transitions is a simple addition to the algorithm.

There are a few conditions that these surface normals should meet. Sometimes they should be smoothed over a fairly wide area – say 16 units, which is about 1/64 of an inch. Otherwise, the jagged steps due to the way that the object is quantized will lead to odd variations in shade over cuts made at an angle. This could be done by observing whether a given area is flat over more than some fixed surface area. Shading of such large areas should not be allowed to interact with adjacent areas.

Once these surface normals are determined, they could be used to find larger

oblique polygons that could be used to replace existing faces. If the surface normal is constant over some region, then they implicitly define a single polygon underneath them.

Here's an added complication. Imagine a flat workpiece with several pockets. The way I am thinking of shading this, the flat bottom will have the same shade as the flat top. I suspect that that won't look right. For one thing, when viewed from directly above, you won't be able to see that there are pockets at all. The correct solution is ray-tracing to put the pockets in shadow, but that's too costly. A cheaper trick would be to use a darker color as surfaces move further into the interior of the workpiece. That description is to vague to implement, but it's the general idea. Here's one rough and ready idea. The normal vector of each polygonal facet has three components, and these components tell how much the polygon faces in each orthogonal direction. Uses these three values as weights. If the polygon is primarily upward facing, then use the distance of the center of the polygon from the top of the workpiece as a scaling factor for the shade. So, the deeper it is and the more upward-facing it is, the darker it will be.

3.8 Resolution

Reducing the resolution seems to be a necessity for the rendering step. Moreover, at normal screen resolution, there is no point in trying to render an object at a resolution of 0.001. If a window is 512×512 pixels, and each pixel is associated with four 1-squares, so that there are 2^9 subdivisions in each direction, then there is no need for more than about $6 \times 2^{18} \approx 2^{20}$ faces (the 6 is for six faces of a cube). For an object that is 8 inches square, dividing into 2^9 pieces makes each piece $2^{13}/2^9 = 2^4$ units, which corresponds to a resolution of 1/64.

For instance, at 0.001, building the BSP was estimated to take 2^{35} operations. If the resolution is reduced to 1/128, then, instead of 2^{26} surface elements, there are only 2^{20} surface elements. The number of operations to build the BSP tree is then on the order of

$$20 \times 2^{20} \times 16 \approx 2^{29}.$$

This is still a large number, but it's manageable. It's most likely an underestimate in the worst case of a highly irregular object, but for more typical objects it should be a gross overestimate.

Zooming in and out is a problem. I would prefer to allow the user to zoom all the way in on the object so that the ultimate resoultion of 0.001 is not lost. At this point, it's difficult to say whether this will be feasible. It is looking like the resolution must be reduced to 1/128 (say) at a fairly early stage in the process. I think that the work could be done at 0.001 up to the point where the rendering is to occur. In fact, the amount of memory required to store the surface faces at 0.001 is too much; above I estimated it at 2^{30}. This means that, from the point where the rendering process begins we must move to 1/128. The only way that I can see to allow deeper zooming is if the idea of large general polygons pans out and leads to a dramatic reduction in the number of faces. In

that case, I could work at the full resolution at all time. The capabilities of the graphics card may render this issue moot.

3.9 General Polygons

Above, I have assumed that all polygons are orthogonal. Would it make sense to try to fit polygons to the surface? This introduces two additional complexities. One is finding the right polygons; the other is that building the BSP tree will become messier. The advantage is that in most cases there will be far fewer polygons to consider.

Building the BSP with more general polygons will require more complicated code, but the basic idea is the same. The difference is that testing which side of a reference plane a particular polygon is on will take more work, as will splitting polygons into piece when they are intersected by the reference plane. However, if graphics cards are as sophisticated as I think they might be, then this is a non-issue.

Finding these polygons is the hardest part conceptually. This is feasible under the perimeter representation. It shouldn't be too hard to find linear sections of these perimeters. In fact, I could define the perimeter as a series of linear moves. One way to do that is to use the escape value in the perimeter (recall that only three values are needed and that leave a fourth unused value in two bits). Once the perimeter is broken into polygons, I could look at adjacent slices and test how well these polygons match up. That could be done in a relatively sophisticated way, with tests of the error inherent in different choice for the spanning polygons. At the other extreme, I could convert these linear faces into one unit wide strips, and forget about trying to find polygons that span multiple slices.

In fact, I could associate surface normals to each step of the perimeter as it is cut. These are determined by the tangent, which could be left implicit in the linear breakdown of the perimeter, along with the vector that's perpendicular to the slice. Considering the motion of the tool, perhaps by using the G-code that gives rise to the pulse, this could be done, although there would be some additional memory required.

Finding these polygons could be done under the octree representation too, but it wouldn't be nearly as easy. The easiest case appears to be quadtree with height. In fact, this appears to be so easy that it may make sense to do as much as possible in that framework. One way to grow these polygons is with a series of seed fills. Obviously, a large n-square, or a set of adjacent squares at the same height, should make up a single polygon. The trick is what to do about polygons that aren't flat. The program could start at a square and inch out to adjacent squares and try to fit a polygon that spans both of them, and continue out to larger and larger polygons, checking that the error isn't too great. In this way, the number of polygons that must be dealt with could be vastly reduced, even for the most complicated surfaces (like a 3D engraving of a photo).

3.10 Miscellaneous

Would it be extra fast to show the cutter's action as it moves? What I mean is to show a picture with no user interaction from a single point of view. Might be able to do something with analytic surfaces? This is mostly for the cool factor since the image would be just that: a static image from one perspective.

Machines today have graphics cards. Would it be better to just use that? My impression is that these cards are extremely fast and sophisticated. They seem (?) to be based exclusively on polygons, not splines, and I think that the polygons must be convex. Is there a unified Java front-end to these things so that I don't have to think about which card is in there? Knowing how these cards work may influence the implementation. This may influence whether (and how) I use BSPs. It appears that these cards use a z-buffer so that a BSP isn't needed. Recall that a z-buffer associates a depth with each pixel. As a polygon is drawn, for each pixel that is drawn the depth of the polygon is set in a seperate bitmap; the pixel is not set if the polygon is deeper than the one already drawn.

I have assumed that ray tracing is out of the question, and that does seem to be true. If I can draw each face at a cost of 2^5 operations and there are typically four faces for each pixel, then the cost per pixel of the implementation above is 2^9 operations per pixel. I find it hard to believe that I could examine a ray at each pixel and determine the first face that it intersects (which would require a search) in fewer than 2^9 operations. Maybe this would work, but it seems unlikely.

Using splines to define surfaces is another idea that just seems too messy. Drawing them to the screen is not easy. It appears that I would have to approximate these with polygons anyway. Rendering them directly from their definition is too hard.

I also considered a wire frame model, but this is really just the surface representation is another guise.

4 Notes

It turns out that other people have written G-code verifiers, and they look pretty good. MetaCut (nwdesigns.com or roneysoftware.com) costs $600 and looks downright professional. Vericut.com is another. Their's seems to cost more like $1500-2000. ncplot.com is another, but it seems to work only with wireframes (just the tool path, not really in 3D). CutViewer.com is another; it is 3D, but is more bare-bones. There's a video for CutViewer at grzsoftware.com, and it does appear that you can zoom in and rotate – it looks pretty good too. It costs $250. Another is at advameric.com. tkcnc.com is a G-code IDE, but it doesn't seem to work with solid models. editcnc.com is another like this.

GET THIS: See Lecture Notes in Computer Science, Volume 3482 (year 2005), pp. 1089-1098. ISBN: 978-3-540-25862-9. It's the proceedings of a Computer Graphics and Geometric Modeling Workshop. The abstract says that they "develop a new algorithm based on the octree model for geometric and

mechanistic milling operation at the same time. To achieve a high level of accuracy, fast computation time and less memory consumption, the advanced octree model is suggested. By adopting the supersampling technique of computer graphics, the accuracy can be significantly improved at approximately equal computation time. The proposed algorithm can verify the NC machining process and estimate the material removal volume at the same time." It's not a very long article, but it looks interesting. See also scholar.google.com.

5 Conclusion

Here's are points to consider as I proceed.

1. Consider implementing a lathe program first. The G-code interpreter would be very similar to the one used for mills, and the graphics part is much easier. Doing this first is a way to prove that I'm not crazy.

2. Investigate the features of graphics cards. I don't think that I want to assume that the user has a high-end card, but if these cards are powerful enough, then it may make sense to have two versions of the program, one that assumes the presence of a card and one that does not.

3. Implement quadtrees with height for 3-axis operations with simple cutters. Cuts can be made to them very quickly and generating the polygons defining the surface is also fast. These are well-suited to using polygonal facets that are not simple squares. Moreover, quadtrees with height mix more easily with the perimeter representation; it's not so hard to convert a quadtree with height to a set of slices (although going the other way would be hard). See if this can be applied to ball-mills too. Ball-mills and end-mills are probably the two most commonly used tools.

   The idea of using quadtree with height as an intermediate step when creating the polygons faces is good. It may be worth while to try to convert the slice representation to a quadtree with height, and generate the polygons from that. To do this for the entire process may require that multiple quadtrees be generated in cases where there is undercutting. Each of these quadtrees might be from a different perspective.

4. Ignore the shank of the tool (and collisions) for the first version of the program.

5. Couldn't I render the slice idea without generating any polygons at all using pseudo ray tracing? To do this, for each pixel there is a line of light and I must determine which slice that the line intersects. I'm not sure how hard this would be to determine, but it's worth considering.

6 Lathe Program

Many of the issues are the same for a lathe program, but others are much simpler. The geometry of the cutters is a tad trickier to deal with, but not bad.

The internal reprentation of a simple 2-axis lathe should be a simple array, where each entry is the radius of the material at that point along the z-axis. This defines a surface of revolution which could be rendered either as a collection of polygons or by pseudo ray tracing – something like I used for spheres in my astronomy book. Using polygons would require that I triangulate (say) the surface, while pseudo ray tracing is very easy.

Either one of these approaches is not compatible with more sophisticated lathes, nor does it allow for threading. If the lathe has live tooling or additional axes, then the workpiece will not be a surface of revolution and these ideas will not work.

I could handle threads with a cylindrical quadtree with height. That is, a quadtree with height, but where the surface is assumed to curve around a cylinder. One problem with this idea is that the circumference changes as cuts are made.

Another idea is to use perimeter slices. Each slice is taken over a half-plane that extends from the z-axis radially outward to infinity. The slice idea is more flexible and probably easier to implement.

Writing a simple G-code generator for surfaces of revolution would be pretty easy too. Let the user define the curve specifying the surface of revolution (by drawing it interactively), and provide some settings like depth of cut, and it would not be hard to spit out the G-code that creates this object.

7 Related Ideas

- Programs like SolidWorks and SolidEdge are combination CAD and CNC programs. The user draws a 3D object in the usual CAD way, and the program generates the G-code necessary to create this object. For some people, it would be more natural to specify the object directly in G-code, or something like G-code. I could design a language that extends G-code that allows the user to specify the object. The software would show the user a 3D representation of the object as the code is written. If the code can be converted to pulses for a milling machine, then this is the most direct way to make the part. The same code is used to design the part and to make it.

- I suspect that many of the problems with Mach3 stem from the fact that it's written in C (or C++) to run under Windows. A program written in Java could run under any machine, but getting the pulses out to the machine would (probably) remain tricky. One way around this would be to put a single-board computer into the CNC machine. The Java program could communicate with this board using internet sockets, and the board would simply buffer the pulses and send them out to the motors.

- The presence of a board like this opens up other possibilities. For instance, you could have one Java program that controls many different CNC machines. Each of these single board computers could replace something like the FANUC controller. This controller could communicate over the internet with higher-level software. To get really fancy, we could even have touch screens.

- Another advantage of Java (whether with a single-board computer in the CNC machine or not) is that it would be relatively easy to provide hooks for users to implement their own ideas. If users can modify the software easily, without giving away all of its internal secrets, then users will improve the software for us.