NOTE: I CHANGED ALL THE PACKAGES TO BE CALLED `ger` INSTEAD OF `vcnc`.

This document is for people who want to understand how the program works, with an eye to contributing to it. The math part isn't too bad, but the other parts need work, and are out of date in any case.

This is a manual for developers. There are two aspects to the code: the user-interface and the G-code translator. The UI is basically how Swing is used, plus the plumbing that holds the entire program together, while the translator is the central task the program is intended to accomplish.

1 User-interface and Plumbing

The entry point for the program is found in[1]

`vcnc.Main`

There's not a lot to say about this class: it merely kicks off the entire program by creating a

`vcnc.MainWindow`

This is where all the plumbing comes together and is the primary window for user activity. Most of what happens in the class is standard stuff: menus, windows, *etc.*

Alternatively, the `main()` method can invoke a series of units tests. Which of the two is done (ordinary GUI or unit test) depends on which of two methods is called by `main()`.[2] The unit tests are discussed further, below, with the translator.

This window puts each file in a tabbed frame. By default, Java's `JTabbedPane` doesn't have all the functionality desired, so there's a fair amount of UI code in the `vcnc.ui.TabbedPaneDnD` package, which is discussed further, below.

In addition, each tab shown has a type that corresponds to the contents of that tab. These types are managed by the `vcnc.ui.TabMgmt` package. What is does is simple.

`vncc.ui.TabMgmt.TabbedType`

is an `enum` listing the possible types of tab contents and

`vncc.ui.TabMgmt.TypedDisplayItem`

is an interface that each tab `Component` must implement to associate a type with its contents. Currently, all of these extend `JScrollPane`, but any number of other `awt.Component` objects would probably work as the base class.

Currently, there are only three of these types:

[1] "vcnc" stands for "Virtual CNC."

[2] It wouldn't be hard to the choice of which to run into a command-line argument, but it seems like an unnecessary complication.

`vncc.ui.TabMgmt.GInputTab`
`vncc.ui.TabMgmt.LexerTab`
`vncc.ui.TabMgmt.ParserTab`

The `LexerTab` and `ParserTab` are essentially identical, [3] and are used to display G-code output, which is not editable. The `GInputTab` is a bit more complicated because this code is editable and due to toggling line numbers.

The final class found in this package is

`vncc.ui.TabMgmt.StaticWindow`

which is used to convert a tab to an independant window whose contents can no longer be edited.[4]

The remaining files in the `vcnc` package are

`vcnc.TextInputDialog`
`vcnc.TableMenu`

The `TextInputDialog` class is currently used to set the material billet, but it's not being done in the way that it ultimately should be done.

`vcnc.TableMain` is leftover junk – test code for `JTables`. Get rid of it.

TALK ABOUT HOW THE MAIN WINDOW INVOKES THE UNDERLYING TRANSLATOR.

1.1 `vcnc.ui.TabbedPaneDnD` Package

Two features beyond those provided by `JTabbedPane` are added: an "X" to close the tabs, and the ability to drag and drop the tabs within each window or from one window to another. The close-box is managed (mostly) through `ButtonTabComponent`, and the remaining classes manage drag & drop, with `TabbedpaneDnD` being the main class of interest externally. See[5]

`vcnc.ui.TabbedPaneDnD.ButtonTabComponent`
`vcnc.ui.TabbedPaneDnD.GhostGlassPane`
`vcnc.ui.TabbedPaneDnD.TabbedPaneDnD`
`vcnc.ui.TabbedPaneDnD.TabDragGestureListener`
`vcnc.ui.TabbedPaneDnD.TabDragSourceListener`
`vcnc.ui.TabbedPaneDnD.TabDropTargetListener`
`vcnc.ui.TabbedPaneDnD.TabTransferable`
`vcnc.ui.TabbedPaneDnD.TabTransferPacket`

[3] Clean that up.

[4] This is also similar to `LexerTab` and `ParserTab`.

[5] I wrote this ages ago, and it seems to work, but it's not pretty. I need to rewrite it, make it more of a stand-alone thing that is useful more generally and provide better documentation for it. It's messy Swing, so I'm putting it off.

1.2 Miscellany in the `vcnc.util` Package

`vcnc.util.ChoiceDialogRadio`
`vcnc.util.ClickListener`
`vcnc.util.EmptyReadException`
`vcnc.util.FileIOUtil`
`vcnc.util.LoadOrSaveDialog`
`vcnc.util.StringUtil`

Not much to say about these...`ClickListener` may be trash, and they could all be tidied up with unused stuff taken out. They're mostly older code pulled from other projects.

2 G-code Translator

GER is similar to a (very simple) compiler or interpreter. It converts an input file of G-code to a simpler from. This simplification happens as the code passes through a series of layers, where each layer handles a particular aspect of the simplification. The code related to this translation process is all in `vcnc.tpile.*`, either in that package or in a sub-package.

The lowest layer is the lexical analyizer (or "lexer"). Because G-code is so simple, with very little context-dependence, the lexer is equally simple. It converts the incoming text file to a stream of `Token` objects. Each token represents one of the letter codes (G, M, I, J, *etc.*) and any associated value. GER allows the user to extend ordinary G-code with user-defined functions, and these functions are also converted to `Token` objects.

The `Token` objects are passed to the next layer, which is the parser. The parser assembles the tokens into `Statement` objects. These statements correspond to the individual conceptual steps of the program. The remaining layers convert these statements into an increasingly stripped-down subset of the possible G-codes, ultimately producing nothing but `G00, G01, G02, G03` codes for moving the cutter, plus a few M-codes and other codes that pass through the process untouched.

The remaining layers are where the meat of the simplification occurs. By layering the simplification process in this way, each layer is made easier to understand, although it does lead to a certain amount of repetative boilerplate. These layers are given numbers, starting with `00`.

2.1 `Lexer`

The code used for lexing is found in `vcnc.tpile.lex`. Classes outside this package – the next layer of the translator – need access to the `Lexer` and `Token` classes. The other classes here have default visibility, so are not accessible outside the package: there's a token buffer and some internally used exception classes.

2.2 `Parser`

The parser takes a stream of `Token` objects, generated by `Lexer`, and produces a series of `Statement` objects. This code is found in `vcnc.tpile.parse`. Most of the classes in this package extend `StatmentData`. Each type of `Statement` may need different data, one class for ciruclar interpolation, one for linear moves, *etc.* The names for these classes start with "`Data`."

2.3 `Layer00`

This layer eliminates all calls to subroutines. In particular, `M98` (call subprogram) and `M99` (return from subprogram) are replaced by the code of the corresponding subprogram(s).

IN ADDITION, this also eliminates certain items that serve no purpose in a simulator, like `M07, M09` and `M09` for coolant control, together with `M40` and `M41` for spindle high/low and `M48` and `M49` for feed and speed overrides.

For several other commands, it's not clear what the appropriate action should be, so they are treated as a "halt." These codes are `M00, M01` and `M02` (various forms of "stop"), along with `M47` (repeat program).

BUG: I suspect that some of these should pass all the way through the program, and only be dropped at the very end, when rendering.

2.4 Unit Tests

As noted above, the unit tests can be run from the `main()` method. The basic idea is that each layer of the translator, starting with the lexer, is able to produce output as text (a `String`). Each test is specified by an input file, which is run through the translator up to a certain level. The output from this run is compared to a static text file to see if they match.

3 Geometry

There are a few geometric problems that come up.

3.1 Arc Centers

One problem is that we have two end-points of an arc and the radius, and we need to know the center. This arises in the `ArcCurve` class; see `calcCenter()`.

Work in the xy-plane. Let the two end-points be (x_0, y_0) and (x_1, y_1), with radius r, and the center be at (c_x, c_y). We want to determine the center given the other values.

We have two equations and two unknowns:

$$\begin{aligned} r^2 &= (x_0 - c_x)^2 + (y_0 - c_y)^2 \\ r^2 &= (x_1 - c_x)^2 + (y_1 - c_y)^2 \end{aligned}$$

Solve the first equation for c_x:

$$\begin{aligned} (x_0 - c_x)^2 &= (y_0 - c_y)^2 - r^2 \\ x_0 - c_x &= \pm\sqrt{(y_0 - c_y)^2 - r^2} \\ c_x &= x_0 \mp \sqrt{(y_0 - c_y)^2 - r^2} \end{aligned}$$

Substitute that into the second equation and solve for c_y:

$$\begin{aligned} r^2 &= (x_1 - c_x)^2 + (y_1 - c_y)^2 \\ r^2 &= \left(x_1 - \left(x_0 \mp \sqrt{(y_0 - c_y)^2 - r^2}\right)\right)^2 + (y_1 - c_y)^2 \\ r^2 &= \left(x_1 - x_0 \pm \sqrt{(y_0 - c_y)^2 - r^2}\right)^2 + (y_1 - c_y)^2 \\ r^2 &= (x_1 - x_0)^2 \pm 2(x_1 - x_0)\sqrt{(y_0 - c_y)^2 - r^2} + (y_0 - c_y)^2 - r^2 + (y_1 - c_y)^2 \end{aligned}$$

Solving that for c_y might work, but it's sort of horrifying. Instead, use the fact that a line perpendicular to a chord of the arc must pass through the center. That is, there is a chord passing through (x_0, y_0) and (x_1, y_1). Let (m_x, m_y) be the mid-point of that chord. The slope of the chord is given by

$$s = \frac{y_1 - y_0}{x_1 - x_0}$$

(and the order of the x_i and y_i doesn't matter). The slope of the perpendicular to the chord is $-1/s$ or

$$s = \frac{x_0 - x_1}{y_1 - y_0}$$

and the line perpendicular to the chord is given by

$$y - y_0 = \left(\frac{x_0 - x_1}{y_1 - y_0}\right)(x - x_0).$$

In particular, the center of the circle lies on this line, so we must have

$$c_y - y_0 = \left(\frac{x_0 - x_1}{y_1 - y_0}\right)(c_x - x_0).$$

Let's simplify this mess...This is closer to what appears in the code. Define

$$\begin{aligned} d_x &= x_0 - x_1 \\ d_y &= y_0 - y_1 \end{aligned}$$

and

$$\begin{aligned} m_x &= (x_0 + x_1)/2 \\ m_y &= (y_0 + y_1)/2. \end{aligned}$$

The slope of the chord is then $s = d_y/d_x$ and the slope of the perpendicular is $-1/s$ or $-d_x/d_y$. The mid-point of the chord is at (m_x, m_y) so that the equation for the perpendicular to the chord through the mid-point is given by

$$y - m_y = (-d_x/d_y)(x - m_x)$$

and we must have

$$c_y - m_y = (-d_x/d_y)(c_x - m_x)$$

or

$$c_y = m_y - (d_x/d_y)(c_x - m_x).$$

Substitute that into our first equation and get

$$\begin{aligned} r^2 &= (x_0 - c_x)^2 + (y_0 - c_y)^2 \\ r^2 &= (x_0 - c_x)^2 + (y_0 - [m_y - (d_x/d_y)(c_x - m_x)])^2 \\ r^2 &= (x_0 - c_x)^2 + [y_0 - m_y + (d_x/d_y)(c_x - m_x)]^2 . \end{aligned}$$

This is ugly, but it can be cleaned up. Substitute

$$u = c_x - m_x$$

and observe that $y_0 - m_y = d_y/2$ and $x_0 - m_x = d_x/2$. Then the above can be written as

$$\begin{aligned} r^2 &= (x_0 - c_x)^2 + [y_0 - m_y + (d_x/d_y)(c_x - m_x)]^2 \\ r^2 &= (x_0 - u - m_x)^2 + [d_y/2 + (d_x/d_y)u]^2 \\ r^2 &= (d_x/2 - u)^2 + [d_y/2 + (d_x/d_y)u]^2 \\ r^2 &= (d_x/2)^2 - d_x u + u^2 + (d_y/2)^2 + d_x u + (d_x/d_y)^2 u^2 \\ r^2 &= (d_x/2)^2 + (d_y/2)^2 + (1 + (d_x/d_y)^2)u^2. \end{aligned}$$

Set $q^2 = d_x^2 + d_y^2$ and the above becomes

$$\begin{aligned} r^2 &= (d_x/2)^2 + (d_y/2)^2 + (1 + (d_x/d_y)^2)u^2 \\ r^2 &= q^2/4 + (q^2/d_y^2)u^2. \end{aligned}$$

Solve this for u:

$$u = \pm \frac{d_y}{q} \sqrt{r^2 - q^2/4}$$

and then

$$c_x = m_x \pm \frac{d_y}{q}\sqrt{r^2 - q^2/4}$$

and

$$c_y = m_y - \frac{d_x}{d_y}(c_x - m_x) = m_y \mp \frac{d_x}{q}\sqrt{r^2 - q^2/4}.$$

3.2 Arc Extent

Given an arc move, we need to know the maximum extent of travel. That is, the maximum and minimum values for x, y and z. See `ArcCurve.noteMaxMin()`. Assuming an arc in the XY-plane, it is given in parametric terms as

$$(x(t), y(t)) = (c_x, c_y) + r(\cos t, \sin t),$$

where t ranges over some portion of the interval $[0, 2\pi]$, and this internval may "wrap around," so it may be best to think of t as being over an arbitrary interval. We want to know the maximum and minimum values for x and y; the value of t where this happens isn't important.

These extrema occur either at an end-point, or at a place where the first derivative is zero. We have

$$(x'(t), y'(t)) = r(-\sin t, \cos t),$$

So the extrema in x may be when t is a multiple of π, and the extrema in y may be when t is a an odd multiple of $\pi/2$ – when $t = (2k+1)\pi/2$, for $k \in \mathbb{Z}$.

3.3 Arc Approximation

Working with linear moves is easier than arcs, but if line segments are used to approximate arcs, then it's important to have some measure of the error. Consider the chord of a circle of radius r. Imagine this chord being vertical through $(r, 0)$.[6] Let x be the distance from the mid-point of the chord and e be the distance from the chord to the circle (so e is the error). You have two triangles around the x-axis, and let y be the height of one of these triangles. So the length of the chord is $2y$. We have $r = x + e$ and $r^2 = x^2 + y^2$. Take r as given, along with y as a particular length for the chord. We have

$$x = \sqrt{r^2 - y^2}$$

so that

$$e = r - \sqrt{r^2 - y^2}.$$

[6] Really need a diagram.

Rearrange:

$$\begin{aligned} e &= r - \sqrt{r^2 - y^2} \\ r - e &= \sqrt{r^2 - y^2} \\ (r-e)^2 &= r^2 - y^2 \\ r^2 - 2re + e^2 &= r^2 - y^2 \\ e^2 - 2re + y^2 &= 0, \end{aligned}$$

So that

$$e = \frac{2r \pm \sqrt{4r^2 - 4y^2}}{2}.$$

DUH, same as before. Think in terms of the angle. Let the angle subtended by the arc be 2α so that $y = r \sin \alpha$. Then

$$e = r - \sqrt{r^2 - r^2 \sin^2 \alpha} = r(1 - \cos \alpha).$$

If we want e to be no more than a certain size, then we must choose α so that

$$\cos \alpha \geq 1 - e/r.$$

For small α, $\cos \alpha \approx 1 - \alpha^2/2$ (Taylor series), so

$$\begin{aligned} 1 - \alpha^2/2 &\geq 1 - e/r \\ \alpha^2 &\leq 2e/r \\ \alpha &\leq \sqrt{2e/r} \end{aligned}$$

and α needs to be pretty darn small if e is anything reasonable. On the other hand, it's not that bad. If $r = 1$, then $2e/r$ around 0.010 is pretty tight, which would give $\alpha = 0.1$. So you'd need 31.4 chords around the circle. If $r = 10$, then the same estimates leads to 99 chords, which is still not crazy. If e is very tight, so that $2e/1$ is 0.0005 ($e = 0.001$), then a circle of radis 1 needs 140 chords and a circle of radius 10 needs 444 chords.

The question will be whether it's possible to save a significant more calculation by working with linear moves or with arc moves.

4 Rendering

4.1 Scale

The biggest reason to include a concept of scale is for rendering: the coarser the scale, the fewer positions to consider. In 3D, this works by the cube of the change in scale, so this is a big factor. For example, changing the scale from 0.001 in to 0.005 in will hardly matter in terms of what the user sees, but the number of voxels in a given volume goes down by a factor of 125. This can make the difference between an imperceptable wait for a result and an annoyingly long wait.

There is a tension between converting to a given scale earlier or later. Rendering is done by breaking "cuts" into convex volumes, and letting the coordinates that define these volumes remain at high resolution until the last step of rendering, where it's determined whether a voxel is inside or outside a given cut, is more accurate. However, if arc moves are converted to linear moves, then how that is done should be influenced by the scale.

As a compromise, non-linear moves are converted to a sequence of linear moves at high resolution, and any coarsening of the scale is delayed until the rendering step. In fact, there seems to be no downside to expressing all moves at maximum accuracy and only considering the scale when quantizing to voxels. In theory, it might be possible to come up with some case where the linear moves used to approximate an arc would be better chosen if the scale is taken into account, but these cases seem odd and unlikely.

4.2 Other Stuff

In a lot of ways, rendering an image is the hardest problem. Translation itself took time to implement because there were certain decisions to make about coordinate systems and edge cases to consider, but it was pretty clear and the algorithms aren't that complicated. For 3D rendering, we are a lot closer to the edge of what we have the horsepower to accomplish.

There are several ways to approach the problem if displaying some kind of image that represents the part being cut.

1. Plain 2D, in black and white.
2. 2D with depth.
3. 2D with depth and surface normals.
4. 3D of some kind.

Many of these could be done using either an ordinary array or a quadtree/octree.

The plain 2D case is just an array (or quadtree) where the tool either touches/cuts the material or it doesn't. Any time the tool descends below $z = 0$, there's a black pixel.

The "2D with depth" case is similar, but with something like a z-buffer. For each pixel (whether that pixel is in an array or a quadtree), there's a value for the maximum depth of cut seen over the entire G-code program. Surface normals can be added by having two arrays or quadtrees, one with the depth and one with these normals. That takes considerably more memory – at least four times as much.

I'm not sure what the best way is to do the fully 3D case, but it needs to be done if concave cutters or a 4th axis is to be dealt with. I'm leaning toward octrees. I want something that doesn't require a GPU to work, and I would prefer not to mess with a GPU at all. OTOH, using a GPU might not be that difficult.

4.3 Speed

The simplest and most obvious way to deal with the 2D cases is to think of the tool as cutting a certain circular pattern, and move this pattern one pixel at a time and update the array over the entire circle. Thus, if the cutter has radius 0.25 in, then you have a circle of area $0.125^2\pi$ or roughly 50,000 square thousandths. If the cutter moves an inch, then that is 1,000 individual steps and you'd have $50,000 \times 1,000$ or $50,000,000$ individual positions to check. More generally, if the tool has radius r and it moves distance d, both expressed in terms of the rendering resolution (like thousandths), then you have to consider $\pi r^2 d$ individual positions/voxels.

The above is too much. The cutter's total distance of travel will typically be large (many inches). Let u be the scale or rendering resolution, expressed in multiples of one thousandth (or whatever the basic unit of the machine may be). Let t be the total distance travelled by the cutter over an entire program, in thousandths. Then we're looking at $\pi(r/u)^2(t/u)$. Assume $r = 2^8$ (roughly a half-inch cutter) and t could be 2^{10} inches or 2^{20} total units travelled. So, overall, we have something like $2^{38}/u^3$ voxels to visit. Obviously, letting u be something like 4 or 8 would help a lot, but it's still too much.

This can be reduced by adding a slight complication. Instead of considering all the voxesl of the cutter at each position, consider only those voxels on the leading edge of travel. For an ordinary end-mill, this changes from requiring that the area of the cutter be considered to only the perimeter. The savings is less for something like a ball-mill or v-cutter; in those cases, the "leading edge" is really a leading surface, so the reduction is only by a factor of about one half. Then again, you could let the outer perimeter cut (like the semicircle in the vertical plane of a ball-mill), and at the two ends of the cut you would have to consider the hemispheres. So the savings is almost as large.