The lab notebook for *vcnc*, aka *Ger*. There are three important documents: this one, `manual.tex` and `overview.tex`. The `overview` document has higher level thoughts on the problem that are too long and involved for the kind of blow-by-blow stuff that appears here.

# 1 July 22, 2022

This is a revival of the vcnc program that I worked on years ago. John Dunlap from the Bangor Makerspace is part of the reason I've reopened this. Enough time has passed that certain things should be technically easier, and my intent is to dial back on the goals too.

I spent most of today going through the old stuff. I pared and combined the various old versions without getting too obsessed with eliminating every last redundancy. That stuff is in the `ancient` directory. I pulled the documentation out of there, and renamed many of these files, to be available here (outside of `ancient`) for reference since certain aspects of the documentation remain relevant. There were essentially three versions.

- *G-code Interpreter* was (roughly) the first attempt. See `labA.tex`. I started this in November, 2008 and the lab notebook runs through the end of June, 2009. It looks like I let Charlie Whorton play with it and had a meeting with Brian Barker. That meeting went nowhere and I think that was part of the reason the idea fizzled out.
- *revived* appears to have been a brief attempt at building on the previous version. It looks like I made a fair number of fairly extensive changes, but it doesn't look like I left a lot of notes about it.
- *qt dev* was the last significant thing I did. I think (?) that I started on revived, in Java, then decided to port it all to C++ and Qt. I think I was hoping that C++ would be faster and allow me to access the GPU more easily. It looks like I worked on this from April to June of 2013. Based on date stamps, I think (?) I may have worked on revived a bit in 2011, then decided to go the C++ route in 2013.

  Based on comments found in this version, it looks like I did fix some non-trival bugs that were in the Java – nothing earth-shattering, but they were important improvments.
- *johns CNC* was a quick revision of revived so that John Dunlap could look at it. This can be ignored.

For all the shortcomings of Java, and even after all these years, I still don't see a better choice than Java/Swing. That's kind of shocking, but seems to be true. One reason not to use fully compiled language, like C++, is that I want to allow the user to define new CNC commands. Doing that with C++ would be very difficult for anyone who's not completely familiar with C++ and all it entails. JavaScript might work too, but complex data structures are difficult to work with in JS. OTOH, it may be that there are already tools to do certain rendering tasks in that framework.

Anyway, the goals for this attempt are as follows

- Allow the user to input standard G-code, pretty much as in every previous version.
- Display some kind of 3D (or 3D-ish) rendering of the result. I hope that the tech (outside libraries) has reached a point where I can do this with a reasonable amount of effort.
- Allow the user to define "wizards" to extend standard G-code. I've considered the idea of extending G-code to include various programming abilities, sort of like Macro-B, but it seems like a sink-hole. If a user knows enough to "program," then he knows enough to figure out how to do it via Java. This leaves the G-code as something relatively clean.
- Output "minimal" G-code, meaning G-code that only uses the most basic commands, like `G00`, `G01` and so forth. In earlier versions, a goal that I had in my mind was sending pulses to steppers, but that problem has been (more or less) solved by things like GRBL.
- Don't worry about multi-axis machines. The average hobbyist doesn't have that. At the same time, tools that undercut raise many of the same issues and I do want to consider such tools.
- It should be possible to simulate (*i.e.*, watch) the motion of the tool as it cuts. Ideally, there should be a way to indicate where tool crashes might occur against fixtures or if an attempt is made to run the tool through the material with rapid travel.
- John Dunlap wants to be able to output STL files. Since these are essentially just a list of the triangles defining the surface of the object, doing this should be a small extension to the ability to redner in 3D – assuming that kind of rendering is possible.

Aside from improving the rendering and adding the use of wizards, this is pretty much what already exists.

Here's a summary of important points from `labA`.

- From the very beginning, I was thinking of rendering with a QTD (quad-tree with depth). In fact, I started with what I called a DA (depth array), which I think was just a 2D array of shorts – hoggish of memory, but fast.
- I messed around implementing a BSP (binary search partition) and Gouraud shading, but I think it was too slow or something. Another idea was a kind of z-buffer with check for hidden pixels.
- It looks like I must have been working with some kind of simple G-code interpreter while I was working out some of the kinks with rendering; then I improved the renderer. In fact, long after I had (or thought I had) more or less settled various rendering issues, I was implementing things like cutter comp.
- There's some useful algrebra in `labA` about determining the intersection of simple curves.

- I had various difficulties dealing with cutter comp, and I need to make sure that what I did is correct.

From `labB`, I see the following.

- I discarded any attempt to deal with undercutting.
- I found some way (?) to deal with cutter comp that is analytical. See the entry for May 13, 2013.

My goal should be to produce a triangulation, and do it in a way that allows for undercutting. This requires (for practical reasons) that I use some outside library to render the triangles. I do still want a version that uses something like a DA, both for testing and because it will often be exactly what the user needs. If you're cutting shapes out of plywood, then you don't need a proper 3D rendering.

Note that OpenSCAM is now CAMotics.

## 2 July 23, 2022

Continue reading old documentation, and see how the software landscape has changed.

I mentioned above that CAMotics is the new OpenSCAM. The author of CAMotics posted an interesting overview of his journey:
`medium.com/buildbotics-blog/is-camotics-alive-3be83275493`
Most of the above is a kind of teaser for a project he is being purposely vague about, but that he says will allow him to make a bit of money off the whole thing, while keeping it open-source. The above was written in 2016, and I don't see anything on the current CAMotics website that could be this project. The interesting part of the above is that people do donate on his page, but not very much; he says that it works out to about $1.33 per hour of his time. Another interesting thing is that he released this in 2012, and started working on it in 2011. CAMotics is available on github. See
`github.com/CauldronDevelopmentLLC/CAMotics`
or his original version (called OpenSCAM, haha) is at
`github.com/jwatte/OpenSCAM`.

Somebody put together a list of open-source CNC software:
`www.reddit.com/r/CNC/comments/aizatc/free_and_open_source_camcnc_software/`
although most of what's on there isn't so interesting. F-Engrave is one exception. It's limited, but I've used it and it does it's thing well.

Something I found seperately is Open Cascade Technology (OCCT), which I think was/is used by HeeksCAD. I see from the Wikipedia page, that several well-known programs use this, like FreeCAD and KiCad. There was a fork of this, back in 2011, called Open Cascade Community Edition. The community edition is at `github.com/tpaviot/oce` and the non-community edition is at `github.com/gullibility/OpenCASCADE`, although I'm not sure how "official" that is. These are in C++, although there seems to be some kind of Java wrapper. This might be a way to go, but it's awfully heavy.

Better would be something designed from the ground up with Java in mind, like Java OpenGL (JOGL). There's something called Java3D, but I think that's more about rendering entire scenes and is higher-level in some sense and is more game-oriented. Although it still exists, I think it is dead, practically speaking.

Another choice is libGDX, which John Dunlap was advocating. This might work, but it's a huge thing intended for games. That said, it might be possible to use only the lower-level stuff that is needed. If you look on their website, under "features," their "low-level OpenGL helpers" might be all I need.

It seems that libGDX is based on LWJGL (Light-Weight Java Game Library). Wikipedia says that LWJGL is the "backend" used by libGDX for "the desktop" (as opposed to web apps).

Ten years ago, there was something called "Project Sumatra" that was supposed to get the JVM running on GPU, but it seems to have died.

`blogs.oracle.com/javamagazine/post/programming-the-gpu-in-java` isn't all that informative, but worth a quick look.

Another interesting thing is JCSG (Java Constructive Solid Geometry), at `github.com/miho/JCSG`. It might (?) be possible to use this to go from individual tool paths (like from a single line of G-code) to a triangulation. It claims to work using BSPs.

LWJGL is probably the place to start; either that or JOGL, but LWJGL seems to be more active, particularly since libGDX is based on it and libGDX is extremely active. And consider JCSG.

At the momemnt, I am thinking that the following might be a good way to approach the problem of triangulation and sweeping out the tool path. The basic idea is to use an octree, but, instead of going down to the level of tiny $0.001 \times 0.001$ voxels, if a cube of the octree is partially filled, let the cut through the cube be represented by one or more triangles. Of course, you could let the entire thing be represented by a single partially filled cube with a huge number of triangles – that is sort of the goal after all. The point of this idea is that the set of triangles in a give cube would divide the cube into what's inside and what's outside the solid. By using an octree this way, the sweeping should be faster. Or maybe JCSG could do this, and I won't have to think about it. If JCSG uses BSPs, and does it well, then I don't think I can really do things any faster.

I downloaded the JCSG stuff from github, just to remind myself that it's on the table. LWJGL (version 3) too. ligGDX is more of a monster, so I didn't download that.

I'm starting on `v01`.

How to set up the UI? Clearly, I want tabs so that the user can open multiple G-code files, but what about the output, bearing in mind that there could be a variety of outputs? Before, I think that any output opened to a different window, with one window for each instance of "output." There could be a variety of outputs: the transpiled G-code, which could be transpiled to varying degrees; different kinds of rendering (2.5D, 3D, etc.). The user might want to look at two or more of these at the same time, and that leads me to want to put things in individual windows. OTOH, having a zillion windows open is annoying. One way might be to put everything into its own tab, but allow the user to "windowize" any individual tab. This gives the user control over how messy things get, while allowing him to compare things as he sees fit.

I think I could do this using the same basic code, whether the thing being displayed is in a tab or a window. I also like this because it keeps things neat. So I need a tabbed area that can display arbitrary things in the tabs. I don't think that's hard, since additions to a `JTabbedPane` consists of `Component` objects.

G-code needs to be contained in a particular way. Every G-code file can have various settings associated with it, like a tool table, choice for metric/inch, and the like. Given how this will be used most of the time, each G-code file needs to point to the window that contains it because that window has the settings for these items. It's tempting to put these settings directly in the G-code somehow, within specially formatted comments, but some of these items would be difficult to express in ordinary text done in such a way that it would make sense to the average user – and so that it wouldn't be a big pain to maintain. What could be done is have the main program save a set of various named settings as "persistent data," and let the text of the G-code refer to these settings.

So, if a particular G-code file is opened in a given window, and that G-code does not refer to a particular set of settings, then it will use the settings for that window. This requires that it be possible to set all these values "together" so that each collection of settings can be named for future reference.

## 3 July 24, 2022

Have the rough plan in place for the UI. Start bringing over the transpiler. The Lexer part came over with almost no changes. I did make use of some improvements to how comments are read in the C++ code versus how it was done in Java in revived.

Something I debated was how to handle the situation where the user has a given bit of G-code and creates various outputs from it, like transpiled code or a rendering. It is tempting to let them keep old versions in tabs so that they can compare changes, but that starts to get confusing (for me to program and for the user). Instead, only allow one "kind" of output from every input. If they have (say) lexer output from their g-code and they regenerate the lexer output, then replace the contents of the existing lexer window.

I would like the ability to drag tabs around to reorder them, but it's painful.

## 4 July 25, 2022

Working on how to make tabs draggable, etc. If I put this off until I have more of the G-code infrastructure done, I will probably never do it. I found a couple of examples – MIT License, so free to use, though by the time I'm done, there will be nothing original left.

After some thought, I am not going to use scrollable tabs. If the user has lots of tabs, then it will be easier for him to see them if they're all visible, even if they end up being "stacked." There are pros and cons, but this will be better for the user and better for me, as a programmer. The only downside I see is that you lose some vertical drawing area if you have a lot of tabs stacking up.

I spent a fair amount of time unravelling the drag & drop code I found on github for tabbed panes. I could do a lot more to clean it up.

# 5 July 27, 2022

I've made some progress on the problem of getting the tabbed area to work the way I want, although I haven't settled on exactly what it is that I want. Something I've given up on is detecting double-clicks in the tabs as a way to bring tabs to their own window. No doubt, there's a way to do it, but as things stand, the drag & drop stuff interferes with detecting clicks.

What I'm leaning toward now is allowing multiple windows, each with all the menus and so forth. In practice, this would be treating the various tabs as though they all belong to the same window. I would like to be able to DnD tabs from one window to another, but that may be tricky.

Can I do DnD between windows? How about allowing you to make a tab "dead?"

# 6 August 8, 2022

Had a bit of a hiatus, and spent a few days without updating this document. I've moved to v02 because the drag & drop stuff is working well enough for the time being, and might be fully done. I deleted a bunch of versions of the intermediate D&D code going forward, but those versions are saved with v01.

Next: set up the lexer to accept externally defined Java functions. The big issue is how to distinguish ordinary G-code from these new functions. There are several ways this could be done, but I will say that these new functions have to start with a lower-case letter. Then, open parenthesis, comma-delimited arguments, close parenthesis. Another way would be to say that these must all start with something like `@`, but that's painful for the user to type all the time.

# 7 August 9, 2022

After some discussion with John Dunlap, I've revised my opinion about external functions. Allow names for an external to start with any letter. All I need to do is peek ahead one letter to see whether we have a single-letter G-code or a wizard function.

One problem is that everything in strict G-code is of the form letter+something. For example, you can't have a bare number, like `2.0` without a letter in front of it. This makes arguments to wizard functions trickier, and it also means that the tokenizer can no longer be context-free (or not quite as context-free).

# 8 August 11, 2022

I've settled on making a few significant changes. One of them is allowing codes (G or M) that are given using floating-point numbers. Most codes, use a whole number, like `G01`, but there are some oddballs like `G84.2` for tap drilling. I am trying to be open-ended about what the system will accept, so I need to accept these, even if I don't do anything with them. It's up to the user to define a wizard to interpret it.

# 9 November 2, 2022

I got sucked into getting proper bylaws for the makerspace, and then being president. Hopefully that stuff is on a more even keel and I can get back to this.

One of the things I see now is that things like tool crashes and the geometry of the (simulated) physical machine are too fiddly to deal with. The relates to things like `G28` (machine zero return) and the like.

# 10 November 6, 2022

I got the line numbers to display, but there are (minor) bugs. And I got the first (zero-th) layer in place to eliminate subroutine calls. Next is to get some of the preperatory commands in place – for things like setting up the tool table and the like.

The commands I need to handle before the G-code proper are:

- Something for tool table, although this is tricky for unusual tools. I need this for cutter comp at a minimum, but also for the shape of the tools.
- Work offsets table.
- Billet dimensions, with margin

Some of this data (work offsets and tool table) should be persistent between runs of the program. There is a `Preferences` class to help with this, but it is really intended for much shorter "settings," not entire data structures. This could be used here, but it probably makes more sense to let the user save a settings file (or more than one) with the settings he wants. On one hand, it will usually be the case that he only wants one such settings file; but he might want different files for different cases. The best solution might be for there to be a default location for the settings file, but allow the user to choose another file.

## 11 November 21, 2022

All the layers are in place, including the basic wizard framework. Most of the machine setup stuff hasn't been done (e.e., tool table) and none of the rendering.

Now that the bulk of the code is there, I've started going back and cleaning things up. One UI thing is toggling line numbers. I have something that sort of works, but it's buggy. What I want is something that acts almost identically to `JTextArea`, but with the ability to toggle line numbers.

## 12 Feb 10, 2023

I have been working on this, but somehow not really updating this document. There was a bit of a break while I tinkered with Haskell, but progress is pretty good. Now up to version `v14`. Part of the reason I'm not using this – I think – is that what I need to do is pretty well understood since I've been down this road before. There's not a lot of planning or options to explore.

## 13 Feb 14, 2023

Now I do have some planning...how to provide a graphical representation of what is being cut. A mistake I made in the "ancient" versions was throwing away old things when I tried new things. Old code wasn't erased from storage, but the various methods weren't usually available simultaneously from the program. What I want to do this time is "sneak up" on various approaches, while each of them remains accessible to the program.

This is likely to change, but here are some methods to implement.

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

Look at the "ancient" code, where I made various stabs at these. The one in `G-code Interpreter` seems to be the one with the most variety, with a focus on the 3D case, inclusing things that probably aren't the right approach, like BSPs and Gouraud shading. I took what I then thought were the best ideas and used them in `qt dev` version, and then (I think, maybe) I went to the `revived` version (in Java).

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. I could do this once, where you simply follow the tool path (the cutter is one voxel) and again, where the cutter has some extent.

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, though maybe less if I restrict the surface normals to being given by three bytes.

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.

The crucial thing in the 3D case (and maybe in the 2D case too) is that the volumes being cut away must be convex. Obviously, not all cutters are convex, but you can express their volume as the union (maybe overlapping) of convex volumes. I suspect that restricting to certain basic shapes will allow the code to be tuned and made faster. These basic shapes would be something like cylinders and spheres. It might make sense to including things like hemispheres and oblate spheroids. The question is whether it is faster to consider fewer shapes that are harder to deal with – the advantage of something like a sphere is how easy it is to tell whether a point is inside the sphere.

## 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.

Instead of getting distracted by all the things I might do, implement the simplest possible case first and work up from there.

## 14 August 8, 2022

## 15 August 8, 2022

## 16 August 8, 2022

## 17 August 8, 2022

## 18 August 8, 2022