Case Study: Building an In-House Parametric Name-Plate Generator

Project: Davenport Designs — custom name-plate/keychain product line **Tools:** OpenSCAD, Google Fonts, Terminal/Bash, VS Code, Claude **Outcome:** A fully owned, zero-royalty parametric design system — no MakerWorld license, no Customizer link, no per-file fees

---

The Problem

Davenport Designs' 3D-printed product lines had been built around remixing other creators' Customizer-style files on MakerWorld — clicking a link, adjusting sliders, exporting. That workflow has a hidden catch: even when a design looks "customizable," the license almost always still belongs to the original creator. Selling prints made from someone else's file — even a heavily modified one — usually requires a separate paid commercial license, sometimes recurring, sometimes per design.

For a product line built around personalization (customer names on keychains and plates), that meant a real, ongoing legal exposure: takedown risk, revoked licenses, and recurring fees eating into margin on every single order.

The goal: build the geometry from scratch, in-house, so every name plate sold is 100% original IP.

The Plan

Three pieces, all free and all clear of any licensing entanglement:

- **OpenSCAD** — free, script-based parametric CAD. The same engine that powers Thingiverse's Customizer, except writing the base script yourself means you own the output outright.
- **Google Fonts** — nearly all licensed under SIL Open Font License 1.1, which explicitly permits commercial use with no royalty and no attribution requirement.

- **A command-line generation script** — so producing a new order is “edit a name, run one command” instead of opening a GUI and clicking through menus each time.

Designing the Actual Model

The first working version was a simple rounded rectangle with extruded text on top and an optional keychain hole. Two rounds of design iteration got it to its current form:

1. **“Bubble” background** — instead of a plain rectangle plate, the background now follows the actual silhouette of the letters with a small margin, using OpenSCAD’s `offset()` function. The first attempt used `hull()` to guarantee the background stayed solid (no gaps, no holes from letters like “e” or “O”), but that flattened the shape into a smooth blob and lost the tight, letter-hugging look. The fix was a “closing” operation — grow the shape, then shrink it back by the same amount — which fills small holes and gaps while preserving the concave dips between letters.
2. **Two-layer construction** — a 0.1” solid background layer with a 0.1” raised letter layer on top, all specified in inches directly (with a single internal conversion constant to millimeters, since that’s what the slicer actually expects).
3. **An attached keychain loop** — positioned using a simple, directly-editable offset value rather than an auto-calculated one, after an early attempt to auto-estimate letter width consistently misjudged how wide certain fonts render and placed the hole on top of a letter instead of beside it.

None of this design work was the hard part.

Where It Actually Got Hard

The script itself came together in a handful of iterations. Getting a single file off the model’s disk and into a slicer took roughly ten times longer — and almost none of the obstacles had anything to do with the code. In order:

- **A hidden password prompt.** Installing Homebrew requires typing an admin password into Terminal, which shows zero visual feedback — no dots, no cursor movement. It looked frozen. It wasn’t.
- **Function keys mapped to hardware controls.** OpenSCAD’s keyboard shortcuts (F5 to preview, F6 to render) do nothing on a stock MacBook Pro, because F5/F6 are mapped to brightness controls by default. The fix was using the Design menu directly

instead of the shortcuts.

- **A GUI export that failed completely silently.** The OpenSCAD app's Export as 3MF and Export as STL dialogs would run through their full flow — options screen, filename, Save button — and then produce nothing. No error, no dialog, no file anywhere on the entire Mac (confirmed via direct filesystem search). This is a known class of macOS issue where an app lacks proper write permission or notarization, and the OS silently drops the write instead of surfacing an error.
- **The workaround: bypass the GUI entirely.** Since the OpenSCAD command-line binary writes files through a completely different code path than the GUI's save dialogs, moving the whole workflow into a Terminal script sidestepped the broken export mechanism rather than requiring a fix for it.
- **A chain of smaller environment mismatches** on the way to a working script: the `.sh` file losing its "allowed to execute" permission on download, the two working files (script + model) ending up in different folders, the `openscad` command not being linked into Terminal's known program paths, and finally — the actual root cause of that last one — the OpenSCAD application living in a Desktop folder instead of the standard Applications folder, which meant every standard lookup path missed it entirely.

Every one of these was invisible until it was hit. None were predictable from the code itself, and none were things a first-time Terminal user would have any framework for diagnosing alone.

The Resolution

Once the actual install location of the OpenSCAD binary was found — via dragging the file itself into the chat rather than trying to describe a file path in words — the fix was a one-line change: point the script directly at the real path instead of assuming a standard one. From there, the full loop worked end to end:

1. Edit one line in a text editor (the customer's name)
2. Run one Terminal command
3. Drag the resulting STL into the slicer

Takeaways

- **The hard part of "AI-assisted development" usually isn't the code.** The OpenSCAD script itself needed maybe three real revisions. Getting a working file off a specific Mac's specific configuration needed closer to fifteen back-and-forth exchanges,

almost all of it environment-specific troubleshooting that had nothing to do with the original design task.

- **Screenshots and dragged files beat descriptions.** Several rounds of “describe what you’re seeing” went nowhere; the moment an actual screenshot or the actual binary file got shared, the fix was immediate.
- **Silent failures are the worst kind.** A clear error message would have cut most of this debugging chain by more than half. The GUI export’s silent no-op was the single biggest time sink in the entire project.
- **Once solved, it’s solved for good.** The finished workflow has zero ongoing licensing cost, zero dependency on any third-party creator’s terms, and scales to any future product in the line — sliders, ornaments, whatever comes next — using the same pattern.

Extending It: Multi-Color Parts

Once the core workflow was solid, the next real requirement surfaced: the background and the raised letters needed to print in two different filament colors, cleanly, every time — not just as a visual nice-to-have, but as a repeatable production step.

The original script rendered everything as one fused solid, which is fine for a single-color print but gives a slicer nothing to select independently. The fix was adding a `part` variable to the script itself (`"background"`, `"letters"`, or `"both"`), and having the generation script call OpenSCAD twice — once per part — producing two separate STL files built from the same coordinate origin. Because both pieces share that same origin, importing them together into Bambu Studio (selecting both files in one Import action) drops them in already aligned, with no manual repositioning. From there, each part gets its own filament/color assignment directly in the slicer’s object list.

This is also where the value of writing the geometry from scratch showed up again: because the script controls the model at the level of individual modules — background, letters, hole — splitting the output into independently colorable parts was a script change, not a redesign. A remixed Customizer file wouldn’t have offered that kind of structural access.

Closing the Loop: Documentation

The last step wasn’t technical at all — it was writing down the workflow so it didn’t have to be relearned or re-explained each time. A one-page reference PDF was produced covering the full day-to-day loop (edit name, run script, import both files, assign colors), the two Terminal errors actually encountered along the way (permission denied, wrong working directory) with their one-line fixes, and a table of every tunable variable in the script for future sizing or style adjustments.

That last piece matters more than it might seem: a working script is only as useful as the business’s ability to operate it without re-deriving the process from scratch on the next order.

Where This Leaves the Business

Davenport Designs now has a repeatable, fully-owned production pipeline for personalized 3D-printed goods: pick a font, type a name, run a command, get two color-ready parts, print. No commercial license fees, no MakerWorld terms to track, no risk of a design’s license changing out from under an existing product line — and a one-page reference doc so the process doesn’t depend on remembering it.