Curv and the art of Programming Language Design

Doug Moen

curv3d.org
github.com/curv3d/curv

In The Beginning

- Idea: a language for 3D graphics/3D printing, inspired by OpenSCAD and ShaderToy.org
- First github commit: April 2016.
  It was a 4 function calculator.
- I had no clear plan, just a vision of the end result.
- So I used TDD to evolve the calculator into Curv.

By 2019

- Mar 2019: release 0.4.
  The original 2016 vision is achieved.
- Jan 2019: StarCON presentation

  https://www.youtube.com/watch?v=H3Q-Sbxhrj8

  2:12-6:10 (intro), 6:10-11:48 (live demo)
- May 2019: KWLUG presentation

2019 Goals

- Core language: more expressive & powerful
- More shape primitives, shape libraries
- Faster rendering, using new shape reps & GPU compute pipelines
- Shadertoy parity: animated interactive 3D art
- Curv running in a web browser

2019 Challenges

- Inadequate core language: blocking the graphical goals
- GPU compiler is crap: wrong design, code is tarpit
- MacOS + OpenGL: no compute shaders
- Vulkan: hyper complex, 1000 LOC to render a triangle (on MacOS via MoltenVK). No web version. Nope?
- No compute shaders on Web (WebGPU???)
- Early stage WASM exists, but it sucks

Limits of Incremental Design

- Tried to proceed by evolving the code, picking low hanging fruit.
- Significant redesign of core language: more consistent and powerful. But: some churn, due to lack of clear goals + master plan.
- Some improvement of GPU compiler. But failed to rewrite from scratch via incremental changes.

A New Approach

- Time to rethink the project from scratch.
- Need clear goals. Then, a design bible.
- Then, a comprehensive top down design.
- Need to manage complexity (because: language design + optimizing GPU compiler + advanced 3D graphics + research into new shape reps)

Curv is an Art Project

- It is a programming language for making art.
- It is a language for creative coding, for expressing ideas, where the program itself is part of the art.
- It is my personal art project.

Advice to aspiring artists

What nobody tells people who are beginners — and I really wish someone had told this to me . . . is that all of us who do creative work, we get into it because we have good taste.
But there is this gap. For the first couple years you make stuff, and it's just not that good.
It's trying to be good, it has potential, but it's not. But your taste, the thing that got you into the game, is still killer. And your taste is why your work disappoints you. A lot of people never get past this phase. They quit. Most people I know who do interesting, creative work went through years of this. We know our work doesn't have this special thing that we want it to have. We all go through this. And if you are just starting out or you are still in this phase, you gotta know it's normal and the most important thing you can do is do a lot of work. Put yourself on a deadline so that every week you will finish one story. It is only by going through a volume of work that you will close that gap, and your work will be as good as your ambitions. And I took longer to figure out how to do this than anyone I've ever met.
It's gonna take awhile. It's normal to take awhile. You've just gotta fight your way through.

Ira Glass, 2011

Why Am I Doing This?

- I have a taste for programming languages. They all suck, even my own. I’m trying to create a language that doesn’t suck.
- In 2021, Curv still sucks. Although I have 860 github stars, and some people recognize its potential.
- If I work through the current challenges and perservere, I still have a chance to make some extraordinary.

So now what?

- Curv is a research project.
- I learned a lot from the current version.
- Time to rethink everything from scratch. This will be the 3rd major design.

A 2021 Roadmap

1) Language Design Bible – what are the goals?
2) New Language Design
3) New Compiler
4) New Runtime
5) Port from OpenGL to WebGPU

Next year, redesign everything else on this foundation.

Language Design Process

- Don’t blindly copy other languages, because they all suck.
- Instead, fully understand the requirements. Write a Language Design Bible.
- Then design a language from scratch that meets those requirements.

Determining requirements

Curv is two languages in one:

1) A graphics/modelling language

2) A library implementation language

And these have different, conflicting requirements.

The Modelling Language

- Simple & learnable (fits in your head)
- Expressive
- Good feedback (error messages) when things go wrong
- Live programming (edit program while running)

Simplicity: Orthogonal, Composable

- Orthogonal: don’t make me choose between multiple features that are almost the same but subtly different.
- Composable: everything composes with everything else with no stupid restrictions.
- A small set of orthogonal, composable features can be highly expressive, without language bloat.
- A small language is easy to fit in your head.

Simplicity: Local Reasoning

- Simple, clear semantics. No “spooky action at a distance”.
- All things have a hierarchical structure and compositional semantics.
- To understand a thing, you understand each part, then understand the composition rule that joins the parts.
- Nonlocal: goto, exception handling, continuations, side effects, shared mutable state, tracing garbage collectors

The Implementation Language

Lets you define library APIs that are:

- Simple, learnable
- Expressive
- Provide good feedback on errors
- High performance

Requirements Conflict!

Modelling language:

- Simple, zero bureaucracy, dynamically typed

Implementation language:

- More complex, bureaucratic
- Static types, data abstraction mechanism, GPU programming

Progressive Disclosure

Curv is a tower of self contained subsets,

with no complexity leaking from lower levels.

1) Declarative modelling language
2) Parametric modelling language
3) Imperative modelling language
4) Library Implementation language
5) Metaprogramming goes here, if I get that far...

(Fixes conflict; improves learnability)

1. Declarative Modelling Language

- A data description language, like SVG & JSON
- Minimal syntax, no bureaucracy|boilerplate. Eg,
  - [1, 2, 3]
  - circle 10
- Simple, declarative semantics
- Safety: no side effects, no malware

2. Parametric Modelling Language

- Adds user-defined variables and functions
- Permits parametric design:
  - Abstract constants into named parameters.
  - Vary the parameters to vary the design.
- A simple, pure functional language.

3. Imperative Modelling Language

- We add mutable local variables, assignment statements, data structure updates, loops.
- For coding algorithms, imperative code (mutable variables and loops) is easier than recursion & functional style programming. (Lesson from the OpenSCAD community.)

Pure Functional & Imperative

- All data is represented by immutable values.
- All operations are represented by pure functions.
  No side effects, all arguments are explicit.
- All expressions are referentially transparent.
- Mutable state is represented by variables, not mutable objects.
- Mutable variables are local, or they are reference parameters.
  All variables are disjoint (no aliasing).
- There are no mutable global variables.
  There is no shared mutable state.
  Local reasoning: No “spooky action at a distance”.

4. Library Implementation Language

- Complexity and bureaucracy is introduced
- Data abstraction (simpler library APIs)
- Types, parameter checking (good error msgs)
- GPU programming (using static types)
- Typed arrays (efficient huge data structures)