Torrengine

Tuesday, January 12, 2021 4:32 PM

Definition: Making an engine through making games

Goals

- **Main goal:** The process of engineering
  - Working on this project is its own reward, i.e. any length of time spent here is considered to be "success" itself
- Subgoal: Developing an engine to make old-school FPS games
  - Developing an engine to make *any* game distracts the effort too much. We don't have to go in that direction from day-1, but eventually it needs to face towards some predetermined destination
- Subgoal: Handmade codebase/tools/build process. Reduce external help to a sane minimum (sane: not doing any OS-specific stuff)
  - Supports the main goal. Discovering how usual stuff works under the hood,
  - Clarification: tools means game tools, not language tools (We won't develop any vim plugins etc.)
- Non-goal: Shipping games
  - Not building games for other people to play, or to feel the satisfaction of clicking the release button. The effort necessary for that is beyond the scope of this challenge.
- Non-goal: Building something for other people to use
  - We won't be solving the problems we don't have at the time of implementation, and this certainly includes the potential problems of *other people*. We won't be responsible to *anybody* but to ourselves.
- *Absolutely haram*: Any kind of time constraints
  - Time isn't a resource: This challenge doesn't have an end. It's fine if things take too much time and we don't have enough visible progress
  - Being able to not work on this challenge on a daily/weekly basis is a hard requirement. Sanity is the only resource we have. We must use it wisely
- Subgoal-maybe-in-1-year-*maybe*: Rewrite it in Rust
  - We decided against using Rust (see below) but we'll reevaluate it in the future

Process steps

- Start with the bare minimum development environment: nvim/msvc (syntax highlight plugin)
  - Set the sane defaults, hotkeys etc. but don't go too far in language tools, or overly customizing things. Just go write stuff.
- Pick up a project
  - Criteria: First few entries 2D. Move to 3D when confident
    - Reason: Simplicity. There's enough challenge to tackle in the beginning. Starting directly with 3D might cause too much sanity expense
    - The resultant renderer wouldn't be redundant for 3D games. We're still gonna implement UI/minimap/2D math
      - Even if it would, throwing code away is fine (as long as it's a sane amount) since it wouldn't clash with any of the goals
    - TBD: more criteria to be added here, when it's time to turn towards FPS genre
  - Do some preparation work from the list (see below), if necessary
- Implement it, until we can call it done
  - Definition of "done": Playable, stable, fast, juicy, fun *(in this order)*
  - No consideration in code beauty whatsoever. Just write something that works but looks ugly
    - Soft constraint: everything goes in the main file, unless we have a *really* good reason to do otherwise. This would force us to go on with the logic, when we're tempted to refactor
    - Refactoring just for the sake of it costs sanity, because it takes more and more effort as the time passes and the code grows, and at some point the lack of productivity starts to hurt motivation. We need this wave-like structure
  - If some improvement to the dev environment is *absolutely* needed (i.e. it would reduce sanity expense), go for it.
    - Think twice before doing that, though. Do we really need it in the middle of development? Or just slacking off?
- After it's done, refactor the hell out of it. Beautify the code and divide it into reusable modules
  - Think of the pain points in the development process and improve the tools. Bigger infrastructure changes are introduced here
- *TBD*: Make sure the previous entries are working after the refactor
  - Need to figure out: How to organize the previous entries' code. Can't think of anything now, gonna be decided on the way, i.e. after the second entry. Could very well dump the previous entries if the effort appears to drain the sanity too much.
- Go back to project selection, this time start with the code at hand

Development

- Going minimal with C
  - Little dependencies: We implement almost everything ourselves.
  - C standard headers is fine. Talking to OS is beyond the scope of this (C headers docs)
  - Allowed external stuff: GLFW
    - Image and sound asset loading: start with stb headers. If it causes inconvenience or we think it's a good exercise to write parsers for those, then replace with the own solution. Parsing a .png or .ttf doesn't feel interesting at this point
- Reason why not C++: There are some ideas on the internet explaining why "writing C in .cpp files" style is more comfortable than plain C, but we want to experience it first hand
  - If lack of stuff like operator overloading, bool etc. turns out to be costing sanity, then we just change the file extensions and remove the excess handmade-ness.

    - Idiomatic C++ is out of question anyway
    - "C feels clunky today, but writing more code in exchange for not having to curate the language can be helpful for learning." (HN link)
    - Differences between C and C++ (long list)
- Rust?
    - Downside: There is a lot of stuff in the Rust std (list of modules), which renders 'going handmade' subgoal invalid. We have the option to disable this with [no_std] attribute which leaves only the core library. This is something minimal that doesn't provide anything beyond stack-allocated memory manipulation. Not having I/O is a deal breaker in this case. There are ways to provide heap-allocation or calling libc functions (printf etc.) from unsafe code, but, meh, doesn't sound like a problem we'd want to tackle.
    - Downside: introducing potential subgoal "100% safe". People see it as an achievement to avoid using "unsafe" keyword when writing Rust, since the thing with Rust is the safety. Dealing with OpenGL introduces a lot of unsafe, so it's unlikely that we'd chase after this subgoal.
    - Upside: Safety. Less time spent with debugging, which saves sanity
    - Downside: Hard to compile. Need to cover the unlikely cases in order to get things running, which costs sanity
    - Upside: Toolchain comes out of the box more or less, which means less time spent on that.
    - Downside related to ^this: it clashes a bit with 'handmade' subgoal, since it's not practical for us to learn what Rust does under the hood
    - Upside: Learning a new language that has a pretty good future.
    - ***BOTTOM LINE***: Let's not start with Rust. We can *consider* it if/when:
        - Being handmade is no longer a goal (which can happen if progress on the games themselves starts to feel more satisfying)
        - We have enough technical foundation to move on with, i.e. enough stuff is set in stone that we can be fine with Rust's rigidity
        - We have enough motivation to carry out the transition (should *not* go without saying. It's important. Perhaps *the* most important)
- Using OpenGL for rendering (tutorial playlist)
    - Reason why not Vulkan: Learning this sort of thing is difficult as it is. We draw the complexity line here, i.e. we need to stay productive to a degree, to keep the sanity expenses in check
    - Reason why not DirectX: Possibility of moving to Linux someday, however small. Also, the same reason as Vulkan

Projects

(Non-exhaustive list)

- Preparation:
    - Strings
    - Containers: vector, map (hash functions list)
    - 2D
        - Vector2: rotate, angle,
        - Shapes: rect, circle, ray
        - Intersection queries between ^these shapes
        - Tilemap (square & hex)
        - Blockmap/Quadtree
    - 3D
        - Vector3, Mat4, AABB
        - Triangle
        - capsule/triangle intersection queries
    - Memory allocator (articles on different kinds of allocators)
    - Obj importer
    - Wav loader/player
    - Project infrastructure / dev environment
        - Unit tests
        - Logging
        - Measuring execution time
        - Vim ctags
        - Build script
            - Compiling only the modified files, by checking the last modification time of .c and corresponding .obj files (link)
- Game of life
- Pong
- Hasirt *(see backlog page)*
- Picross
- Visual novel
- Jetpack
- 2D rope physics (Reference: Worms ninja-rope)
- Shopping cart pushing: *(still not solid in mind, requires brainstorm)*
- Slingshot *(see backlog page)*
- Pixlander
- C-Dogs
- 12:27 *(see backlog page)* & Hellish twin-stick shooter
- TD: *(currently a generic one. Do brainstorm for a sophisticated one)*
- Sokofabric *(see backlog page)*
- Wolf3D-clone
- Horror game
- Doom-clone
- Lots of ideas

Archived challenge idea

- (This is the first idea, conceived before Torreng One Game A Month (TOGAM). Because of TOGAM's mental legacy, we don't think this is sustainable anymore, due to it's time constraint)
- Goal: The process of engineering
    - Subgoal: To end up with a gamedev tech stack that suits one's needs specifically
    - Non-goal: Shipping an end product
- Start with a language/framework:
    - Examples: C/SDL, C++/SFML, C#/OpenTK/Monogame, Java/libgdx, Rust/whatever, haxe/openfl, typescript/pixi, lua/SDL bindings, python/pygame, (no js please)
    - *NO ENGINES*. The middleware needs to provide the window/low-level rendering/input stuff *AT MOST*. The lower level the better
        - Exception could be the backend (for potential leaderboards). Doing things the manual way could leave security holes which cause permanent damage.
- Build a *finished* game in a month. Start on 1st by picking a theme, release on 30th *at the latest*
    - Hard condition: It must feel sufficiently polished
    - Corollary: It must be playable, finished and published
    - Corollary: The scope should be *very* small, even smaller than a jam game
        - Like, just a jumping rectangle. But it must be a *good* jumping rectangle. It must *feel good*
        - TOGAM scale XS *at most*
    - Corollary: The code will be horrible because of the time limit
    - Option: C/SDL (and maybe C++) could have 2 months instead of one because it involves a lot more to do
    - Option: 3D games could have 2 months instead of one, since OpenGL stuff is tough to learn and get working
- On the next day, the refactor begins. This is the heart of the challenge: Rewrite the entire thing *without breaking the game* and divide it into their own modules. Don't implement anything new. Do this for a month.
    - Probably it's gonna finish early, there'll be time for rest
    - So the point is to maintain this wave-like schedule, not constant pushing for release
- On the next day the 1st, pick another theme and start from the code at hand
    - Q: How to identify the theme? How can we differentiate stuff when there's so little functionality? From here it sounds like these are milestones of a bigger thing than separate projects
        - A: The general direction will be an engine feature, and the theme will make us of this feature heavily: rendering, UI, physics etc. Examples: Physics-based game, UI-heavy game, game where shadows are important, game that allows (or encourages) modding etc.
    - Q: Would the new iteration be backwards compatible? i.e. Should the old games' code be updated, or just left behind after their release?
        - A: ?
- Prediction: After ~10 cycles, we end up with our own middleware which solves common gamedev problems and speeds up the development sufficiently
    - It satisfies the rule of "Make games, not engines". Instead of shooting for something generic, we just focus on making the game itself and after shipping it, we refine its code