A 2D Game Engine for Beginners

Johan Boscher

June 20, 2020

Contents

1 Introduction

We talked about the possibility of making a game engine instead of a game. I really like this idea and I believe that even though we are far from experts in this domain, we can make something new and exciting. The goal would be to create the perfect game engine for beginners who are not planning on selling their game and just want to learn game development. For that we should develop three things. The first should be a C++ library that is the core of the engine, the code of the games made with the engine should call that library, the goal would be that the users of the engine can use ANY language that can call C++. The second is a GUI for the engine that is well integrated with the library. The last is the ability to easily extend the engine with plugins.

1.1 Is there a market for this?

Most game engines have one programming language for you to make the game with. So usually engines for beginners have a scripting language to make things easier on the learner, while big name engines use C++ or C#. The goal of this project is to have a very bare bone engine that nevertheless handles all things necessary for simple 2D game development. There are two big selling points for an engine like the one I have described. The biggest one is that a developer can use the language they want provided that language has a foreign function interface (FFI) with C++. The second one is that the engine being easily expendable with plugins, it is easy to transform the engine into something very powerful, and let advanced users share features that they make with the community.

2 Interfacing with other languages

What makes this engine extremely powerful is the possibility for the user ot code in the language that they want, and to mix up languages. This is the very complicated part of the engine, that I want to work on. I do not think this has ever been done and could make this be the number one choice for Indie game development, especially if a sizable community picks up the engine and starts developing plugins.

2.1 The library

For this to work I believe our engine should be implemented mostly as a C++ library. We can both work on that library, it should be the thing we complete first. It should have all the features of a basic game engine: (Add as we figure out all that needs to be added)

- Management of keyboard inputs.
- Management of content: Loading textures/soundfiles/...

- Management of gamestate and scenes. Transition from one scene to the next.
- Collision management.
- Rendering scenes using textures and light sources.

2.2 FFI

I'll be honest, I'm not sure how doable this is, or how many languages support this. This is not too bad though, because even if this turns out to be way too difficult to do, we can still make the game engine without the inter-language functionalities When the basics of the library are set up, I will let you continue to work on it, and we can start looking at how we will add support for languages that can interface with C++.

Case example: Python Fun fact: Python is the most used language in AI, but most Python AI libraries are written in.. C++! I do not know how easy it is to use C++ in other languages, but it is pretty trivial in Python. The user can call the library to build the objects that he needs, then he modifies them and scripts the game with python syntax, while calling the necessary functions from our library. We provide the building blocks, and the user makes the game!

3 The GUI

The GUI should be extremely simple and extendable. The main principle is that the GUI is used for quick modification that directly impact the user's code. Basically things work as follow:

1. Every GUI action is translatable into code: One button maps to one modification of the code file.
2. The GUI is here to quickly modify what is easier to visualize than to code. Examples of this are changing an objects size/color.
3. The GUI should also be here for actions that are a pain to code, like adding a texture to an object or modifying the hitbox size.
4. The GUI loads the code file and changes what needs to be changed, it's like a graphical code editor on steroids.

4 Expendability

I have no experience on how adding plugins/functionality to a project works, but this is the most important feature. If we can make a SOLID engine that implements all the features added below, then people are gonna be interested in our engine. It's gonna be super lightweight, giving great power to power users and programmers who are good coders and want to make a game for fun. If we make sure that the project is easily expendable, then we can let the project fly. Power users will be interested in adding features and we can continuously add optional features without touching the solid core of the engine: A multi-platform, multi-language 2d engines for beginners and control freaks who want to make video games.

4.1 How do we implement expendability

I think we should do three things to make the engine easily expendable:

A very generic core library with classes that can be extended If the core of the engine is super generic, then is is very easy to add things to the engine by extending the classes of the core library.

Example Let's say the only class in the core libraries for managing moving objects is something like movingObject. Then we ourselves code an optional plugin with a class "entity" that extend movingObject for things such as players and NPCs, and another class "inanimate" for things like bullets and projectiles. This plugin would be in a library that extends the core. It would be added to the engine by default, but power users could decide to install the bare-bone engine that does not implement this and only has "movingObject". That power user can make his own library and share it with others.

Easy library to add GUI functionalities to plugins The GUI should be a very dynamic structure that can easily be modified with extensions. Like this plugins can easily add GUI functionalities without too much added development type. There should be functions that modify the GUI so that we can modify it easily and without having to recompile the engine.

A engine package manager This is the most important thing to make plugins easy to use and widespread. If we can make a package manager that automatically downloads, installs and setups the plugins, we have user friendly plugins in the engine.

5 Planning

Here we will develop how we will make this game engine and what we start with.

5.1 Basics

We first need to setup multiple things that are necessary to test features and add stuff to the engine.

5.1.1 GUI

I feel like we need to setup very basic GUI functionality. This will enable us to test features easily by launching the GUI. Also integrating the core library and features with the GUI as soon as we start is very important otherwise things are going to break. Three things need to be done at the start.

Custom file type/Settings file The GUI is the core of the game engine, a bit like a game IDE. I imagine working on a game as working in a directory that will be the "Project Directory". That directory should have a file that has a lot of info for the GUI. I thing we will either go with a JSON file or a custom file type. That file will have very little at first:

- Where the source code for the game is.
- Mapping core library functionalities to the GUI and adding an easy way to add a GUI functionality from library.
- What object in the GUI "screen" maps to what in the user's code.

Translating GUI to code For example let's say we have a ball object that's loaded in the GUI and who's attributes can be modified, we need to map that modification to a change in the code. For example changing ball size changes it in the source code. This is HARD(I think). Note that these GUI actions do not change the library's source code, they change the user's code that uses the library.

Managing Graphics The center of the GUI needs to be a graphical view of the scene being currently edited, this is the hardest part.

What libraries? GUI library options:

- Qt, the default and most popular library.
- Kigs Framework, great GUI framework and really well integrated with Visual Studio: To use if we can get David on a Windows Machine somehow.

5.1.2 The core library

As soon as we have a basic GUI working, we need to setup the basics of the core library. Here are the things that we need to setup first:

Graphics This is the most important part of the engine, graphics are the basics of video games. We will need to use a graphics library to do this, because doing this ourselves is entirely out of our skill set. But this does not mean that we do not have any work to do. Graphics library are very low level and we still need to code a lot of stuff. I am not sure of the details but I think the goal would be to implement the following to start.

Item drawing Drawing basic shapes easily, squares, triangles...

Importing textures The ability to link a texture file to a specific object.

Library choice

SDL The SDL Library (Simple Direct Media Layer) is a cross platform library that provides "a hardware abstraction layer for computer multimedia hardware components". Basically instead of having to code separate stuff for the sound/user input/2d graphics for each OS we use this library and this makes the game cross platform. In terms of graphics, SDL only supports 2D graphics, this is a problem if eventually we want to extend the engine to have 3d graphics. But SDL can be extended with one of the two options below.

-OpenGL This is THE industry standard library for graphics, 2d and 3d, it is lower level than SDL and probably overkill for what we are doing. OpenGL is used in 90% of 3D engines, and is much faster than SDL. The other big drawback it is slowly being replaced by a new library called Vulkan. It will no longer be supported in further versions of MacOS, so it will likely not be as big in the future. it would be very hard to finish the engine by the summer if we were to use OpenGL, but not impossible. Note that we might have to use OpenGL for the GUI.

-Vulkan I don't think we are skilled enough for Vulkan, but if you absolutely want to go with it I would understand, but we WILL NOT finish the engine by the end of the Summer. So what is Vulkan? It's the future of graphics. It's the most low-level of the three libraries, so hardest to use, it's super recent (OpenGL and SDL are more than 20 years old), so it's adapted to modern hardware. It has the best support for multi-threading and CPU-GPU cooperation. Finally, we would become experts on how graphics work at the super low-level, and this is great because GPUs are now used everywhere, including in Machine Learning. I **really** wanted to use Vulkan, but that would be like taking Cal 3 before Cal 1.

I think for the graphics if we keep the engine 2d Open GL is not great since it is overkill and even though it will probably still be used a lot in the next 10 years, its starting to be outdated. I think Vulkan is way too complicated for us. So for the graphics I think we should use SDL and maybe extend it with OpenGl or Vulkan. There is one last possibility for the graphics though:

-V-EZ This is Vulkan Easy, **but it's really not that easy.** I still think this is way too hard, and we should go with SDL go look at the documentation to see for yourself. But if you wanted to use it, I'd be down. There is one big problem though, I am not sure Vulkan easy is compatible with SDL. If that was the case we would have to use another library for I/O and sound than SDL.

5.1.3 Input

Another big part of the core engine is management of user input. I think to start we should setup one basic input to be expended: Clicking with the mouse. To do this with SDL is very simple, I/O is part of SDL so we can use it. Otherwise, we can use another cross-platform library to manage this, there are quite a few out there. What we need to do for I/O is to link the operation with a specific object, so that the object knows it was acted upon.

5.1.4 Basic Scene/gamestate management

We should also at the start of the engine development setup a scene class. This scene class should be very simple at first and just be a simple class where we place all the objects of a scene. How we place them is the difficult matter but for now we should not worry about a final implementation. Instead we should just setup a scene class that can have any one object of any type using generics.

5.1.5 Generics and multi-threading

These are not features, instead they are important design philosophies that we should think about at the very beginning of making the game. The goal of this core library is to be as modular as possible. This is why I think using generics is a must when they are necessary. The best example of this is the scene class, we can have any type of objects in a scene, so using generics is the best way to do this.

Single core CPUs are dead. Single thread applications are not the future. Our core library needs to fully support multi-threading. This is super important.

5.1.6 Summary

To summarize here is what needs to be done first before we an start in depth work on any part of the engine:

- Set up a very basic GUI:

  A settings file for the GUI that is read and loaded in memory at app launch

  Code for translating GUI actions to code in the users Cpp files

  A graphical window at the center showing the current scene

- Set up very basic Graphics functionalities of the core library:

  Drawing basic polygons.

  Link a texture file to a specific object in a scene

- Add basic I/O functionality, namely clicking with the mouse which should affect the object linked to I/O action.

- A basic scene class which only has one object. This is necessary to debug what happens to that object.
- While doing this we should use generics when necessary and try and make the core library multi-threading capable.

5.2 Chosen libraries so far

The Goal here is to gain as much time as possible. So David, if we want to finish this, we are not going to reinvent the wheel. Here is a list of libraries that we are going to use in the Project:

- A GUI API (Qt, Kigs, or imgui)
- A Graphics API (V-Ez, OpenGL, or SDL)
- A general purpose C++ library (Boost)

6 The GUI

So far my choice for the GUI is ImGui. I believe it is the best tool for what we are trying to do. However, I also think this might be very complicated and I'm considering creating the GUI in a language that is not C++. An example of this is a JAVAFX UI that connects to a backend written in C++ using Apache Thrift. There is also the possibility of not using a cross platform API which would make our App run only on windows but would make our Job so much easier. Anyways I believe we should try and first write the basics of the graphics engine before the UI. The GUI is definitely going to be a challenge...

7 Notes on GUI

If we want this to actually be adopted by other people, which I believe is a great goal to have that could motivate us, we need to spend time on the GUI. It needs to be PRETTY, apps that look like they come from the 90s don't do well. We can consider asking other people for help in the non-code parts like GUI design, because a nice looking engine is a BIG plus.