Filthy Engine Proposal

Contents

[Intro 1](#_Toc26125486)

[2D Rendering 1](#_Toc26125487)

[Input 2](#_Toc26125488)

[Resource Management 2](#_Toc26125489)

[Physics 3](#_Toc26125490)

[AI 3](#_Toc26125491)

[Scene Management 3](#_Toc26125492)

[Tool Development 3](#_Toc26125493)

[Conclusion 4](#_Toc26125494)

# Intro

My Game Engine has been named the Filthy Engine and will be built to accommodate top-down 2D games, like the Legend of Zelda series. There will be the possibility to use this engine for other genres and if time allows, for the engine to support side scrolling games via an enhanced physics system.

Another consideration for me is to make my engine ready to be used as a library, rather than a framework. I have chosen this route because I feel it gives a more user-friendly feel to the engine, and I have enjoyed using libraries more then frameworks from my personal experience. This does limit an end-user being able to change the engine code for their own benefit, but this is not a top priority for me and this engine. This influences my engine design as I will need to make everything as loosely-coupled as I can, in order to allow the user to run their own Main() function where they can Init() the engine, just like SDL or SMFL.

# 2D Rendering

For 2D Rendering, I have a few options open to me:

* SDL2
* SFML
* OpenGL

OpenGL is immediately off the table as I feel I would be spending far too much time getting objects to draw on screen than I would like to admit. Also, this effort could be redirected elsewhere in the engine development which I feel would be a better use of time.

This leaves me with SDL2 and SFML. These both do 2D rendering very easily with most of the hard-work already done for you. SFML does a fair bit more work for you with the implementation of a 2D math library and more game-specific classes like Sprite and pre-built classes like their transformable and renderable classes but overall there is a lot of room for me to implement my own solutions to problems.

|  |  |
| --- | --- |
| SDL2 | SFML |
| Class-based | Object Oriented |
| Limited Language Support | Large Language Support |
| Build support for PC + Mobile | Only build support for PC |
| No base classes provided | Useful base classes provided |
| Usable API | Very readable API |

I have enjoyed making low-level systems so far and will be using SDL2 to make a lot of my own. Time constraints prevent me from using OpenGL, which would be my choice otherwise.

# Input

Input through SDL is done once-per-frame. This is via the Poll\_Input() function. This leaves me with one big issue: if I want to use player input more than once per frame, it won’t work as the information has already been polled and I cannot re-poll it. Hence, the need for an input manager.

The InputManager will use the singleton pattern as it saves me hassle passing pointers about just in order to access it. I feel that it should be accessible from everywhere so that any class can take user input and the singleton pattern is the best way to do this.

|  |  |
| --- | --- |
| Enumerator | BitMask |
| More growth potential | Max 32 different Inputs per mask |
| Fiddly | Makes more sense |
| Complicated naming | More enjoyable to work with |

I do not believe that choosing one or the other will make any significant difference to the end product, so I will go with the most scalable approach and use Enumerators to handle my input.

# Resource Management

Game Engine have many resources that they need to manage in order to make the game run as well as possible. These include sound effects, music, and sprites to name a few. When you have multiple of the same sprite on the screen at once, this would usually entail loading the same sprite multiple times, creating multiple textures and materials. It is clear to see that there is a lot of useless data floating about.

This is where my resource manager will come into play. Any loaded asset will be loaded through my resource manager, which will check if it has already loaded an asset, and pass a reference to the loaded asset; doing the full load, texture, material process only once per asset. This will save the most space when working with tilesets and groups of enemies. Cons with this method do exist but are limited with minimal impact. When only loading one of a certain asset, more memory will be taken up then if we had loaded this one asset normally.

# Physics

Physics engines are a whole beast of their own, so I will definitely be using a library for this part of my engine. There is an extensive list of 2D physics engines compatible with SDL2. However, two stuck out immediately: Chipmunk2D and box2D. Chipmunk is a higher-budget engine with an available premium option on top of the free option, boasting better performance. The box2D engine, however, is an open source engine free to use for anybody and work with any openGL renderer.

Since box2D hides no features or capabilities behind pay wall, I will be using box2D as my physics engine. As mentioned, it is also open source which means that editing / adding features to the physics engine is a possibility; one that I like the idea of.

# AI

I plan to have some form of AI in my game, albeit very simple. Since I am planning this engine to be for games like the Legend of Zelda series, I thought it would be reasonable to mimic the AI found in those games, with simple paths to follow, seeking behaviour, and definitely state machines.

# Scene Management

Scene management will be done all from one class, much like the resource management. I would like to have a hierarchy of { Scene > World > Entities } which will help me and end-users structure and handle scenes in a logical way. This also helps me to handle the render-order of my sprites, mentioned in the 2D rendering section, as well as move code out of my Main() function which should be occupied by nothing; making this as loosely-couple as possible and ready to be used as a library.

Box2D has it’s own world class so I will likely have these as sperate entities and let scene management handle the drawing of every instance and let the physics engine handle all the physics in their own worlds. This has it’s cons like having to clean up objects in both worlds but without making my own physics engine, or the program becoming more closely coupled, this is what will have to be.

# Tool Development

The best engines come with amazing tools. These are any programs that help the end-user in creation of their games or myself in creation of the engine.

I plan to have at least a one tool for my finished engine which is suited to my chosen genre and target game. One is a map editor. These games can often have large maps or dungeons, and having a tool to edit maps/scenes visually will drastically increase the effectiveness of my engine. Another tool I wish to make is a keybinder, where a user can bind keys to common actions, like MoveUp, MoveLeft, MoveRight, MoveDown, Shoot, etc. These tools are not necessary for the engine to function but are a stretch goal if development goes well. Without these tools, the engine’s usability will suffer as people will be less inclined to use it.

These tools will be integrated into the engine and will make full use of the imGUI library to supply the user with the most friendly tools. I have chosen this library as it seems to be the only GUI library written for c++ applications, along with the library being very minimal with dependencies.

# Conclusion

The Filthy Engine will make use of SDL2 and it’s limited features to have it’s own implementations of many base classes to build up loosely-couple systems. Everything touched upon in this document is explained with intent to be finished, with the exception of the tools which remain a stretch goal as they are not necessary for the engine. imGUI will be used for any Graphical User Interface in the engine. This includes the tool development, as they will be integrated straight into the engine.