 Filthy Engine Proposal

Contents

[Intro 1](#_Toc26709145)

[2D Rendering 1](#_Toc26709146)

[Input 2](#_Toc26709147)

[Resource Management 2](#_Toc26709148)

[Physics 2](#_Toc26709149)

[AI 3](#_Toc26709150)

[Scene Management 3](#_Toc26709151)

[Tool Development 3](#_Toc26709152)

[Conclusion 3](#_Toc26709153)

[Bibliography 3](#_Toc26709154)

# Intro

This 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 is to make the engine ready to be used as a library, rather than a framework. This route has been chosen because as it gives a more user-friendly feel to the engine. This influences my engine design as it will need to make everything as loosely-coupled as possible, 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, there is a plethora of options and I have narrowed these down to a few:

* SFML
* SDL
* OpenGL

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

This leaves 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 to implement custom solutions to problems.

|  |  |
| --- | --- |
| SDL2 | SFML |
| 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 |

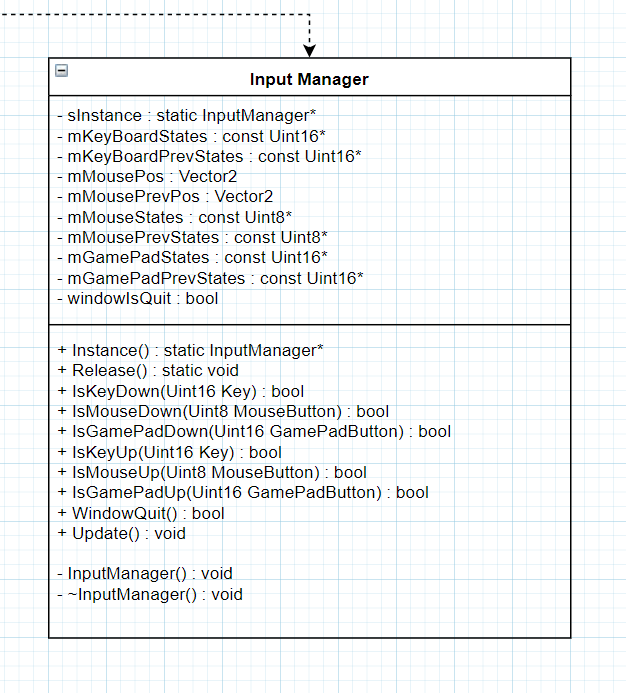
Information in table extracted from (SDL), and (Gomila)

SDL2 will be the rendering engine emplyed as it allows for custom implements in a lot more places. This allows more control over how the engine works, and tailor it specifically towards top-down games.

Another problem that crops up with 2D rendering is sprite render order. Each scene in the Engine will have a list of “Layers”, which the developer can add objects to, to control the order in which they are rendered. New objects will default to the lowest layer.

# 

# Input

Games heavily rely on user input, and the Filthy Engine will use the built-in system that SDL uses as it is simple to work with and comes with GamePad support out of the box. The only downside with the SDL input system is that it is Polled, instead of using input callbacks. However, this won’t be an issue since polling is simple to work with.

Input through SDL is done once-per-frame. This is via the Poll\_Input() function. This leaves one big issue: if player input is needed more than once per frame, it won’t work as the information has already been polled and cannot be re-polled. Hence, the need for an input manager. The inputs will be polled once in the InputManager, and then stored in integers (right) and the developer will bitmask these to determine which inputs have been received in a single frame.

The InputManager will use the singleton pattern as it saves hassle passing pointers about just in order to access it. 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.

Extract from the Filthy Engine Class Diagram, showing the Input Manager

# Resource Management

Game Engines 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 the resource manager will come into play. Any loaded asset will be loaded through the resource manager, which will check if it has already loaded an asset, and pass a reference to the loaded asset using it’s HashMap; doing the full ‘load, texture, material’ process only once per asset. This will save the most space when working with tile-sets and groups of enemies. When only loading one of a certain asset, more memory will be taken up than if we had loaded this one asset normally. However, this is a minimal gain in memory and will be far outweighed by the memory savings.

# Physics

Physics can make or break a game, and with a top-down game like Zelda, there isn’t much going on in the way of Physics. Therefore, the Filthy Engine will not use any pre-built physics engine. It will instead use its own implementation of a physics engine; including only what is necessary for a game of this genre: collisions, and momentum. There is no need for anything else, meaning the footprint can be very small and not need to jump through loops of other systems to do basic tasks.

This will link to the rest of the engine as a series of components that can be added to GameObjects, which adds it to the physics world, letting it collide, gain momentum, be a trigger volume; allowing each object to only use the part of the system it needs to.

# AI

The Filthy Engine will incorporate AI, albeit very simple. Since the engine is to be for games like the Legend of Zelda series, it would be reasonable to mimic the AI found in those games, with simple paths to follow, seeking behaviour, and finite state machines. The FSM will be a component that can be added to GameObjects and control them directly.

# Scene Management

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

# Tool Development

The best engines come with tools to help the developer. These are any programs, built-in or external, that help in the creation of games. One is a map editor. Games can often have large maps or dungeons and having a tool to edit maps visually will drastically increase the effectiveness of this engine. These tools will be internal and make full use of the imGUI library to supply the user with the most friendly tools. This library was chosen as it is very minimal with few dependencies. The tool will be internal to the engine as it will output file formats specific to how this engine will read them.

Another invaluable tool for developers is a profiler. The filthy engine will incorporate one of these to allow developers to see how long their functions are taking, and where their code inefficiencies are. This tool will include the profiler and a logging system with multiple verbosity levels, all built-in to the engine.

# 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.

# Bibliography

Gomila, L., n.d. *Home.* [Online]   
Available at: https://www.sfml-dev.org/

SDL, n.d. *About SDL.* [Online]   
Available at: https://www.libsdl.org/index.php  
[Accessed 2019].