Filthy Engine Proposal

Contents

[2D Rendering 1](#_Toc25656143)

[Choices 1](#_Toc25656144)

# 2D Rendering

## Choice

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. Below is a table of pros/cons comparing SDL2 and SFML

|  |  |
| --- | --- |
| 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 |

After some consideration I came to the conclusion that I would stay with SDL for this GameEngine.

## Why?

The first and foremost reason I am using SDL2 over SFML is because I like the nitty-gritty of GameEngine development. While contradicting my previous statement about OpenGL, this is to less of an extreme. Given a lot more time, I would likely use OpenGL as my renderer. The nitty-gritty with SDL2 that I would be making is all concepts I am very familiar with from my time working with 2D engines; sprites and 2D vector math are something I have spent a lot of time using and I am eager to implement my own versions of these, rather than use the SFML library’s. Writing my own implementations also opens up more opportunities to fix them should something go wrong, or I want more functionality it is easily added to the class. This also makes these classes better suited to optimisation and profiling using my own tools.

So why would I ever use SFML? It is a great library and it seems suited towards quick prototypes of ideas and small projects since there is a solid framework for you to work off. As I have a few months to work on this, writing my own (and hopefully more flexible for my needs) implementations will be a better solution in this case. However, should I ever need a small graphics application written in C++, SFML will likely be my first call for prototyping.

# Input

## Choice

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 using a singleton class is the best way to do this. Obviously, some classes will have no need for user input but for those that do, it saves me some stress.

The singleton will poll the input once per frame and store this data. I plan on only storing the data that I care about; reducing memory usage. The data I care about will be player control inputs, mouse movement and input, and window events like closing and resizing. Next comes how I should store and pass about this data. My first thought was to have an enumeration with all of my possible inputs and use this to index into an array holding the data I want. This immediately posed an issue of naming as SDL also uses this method. However, after further development I came up with a name for my engine so I could easily prefix them with FILTHY\_ which would solve my issue and also allow for future use of this engine to easily help distinguish what is engine code and what is game code. My second thought of how to manage my inputs was with an input mask where I use a bitmask to mask the bits I want from the input and then grab that value from a dictionary.

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

I have not yet decided whether I want to use enumerators or a bitmask to handle my input. I am leaning towards BitMask but I definitely need to finish this section.

## Why?

# 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. This can be avoided by just loading normally on assets that we know will only be needed once. Also, this method adds a small amount of extra CPU time to loading resources for the first time, but subsequent loads are significantly faster.

All considered, the benefits far outweigh the pitfalls of this method. To keep this method as fast and lightweight as possible, I will be using a hashmap of <Key, asset\*> to store the loaded asset references. Other options are available like Dictionaries and arrays but hashmaps are better from every view: they are quicker and take up less room. Because of this, I will not be discussing the pros and cons of using each method.

# Entities