**Analysis**

Research

Graphics Abstractions

Graphics abstractions is a necessity when it comes to writing a game engine. The basic concept is that all the main graphics objects (such as Vertex Buffers) need to be hidden by a layer of abstraction, simply because it is going to be easier to use and you won’t have to worry what the engine uses as it’s graphics API, since the object will do all the necessary functions that will depend on the graphics API. The purpose is that you can make a graphics object and it will sort out the generation and manipulation for you, so you can write some generic code that will work for all graphics APIs.

So, the way that I found out that you should do this is:

* Make the object a pure virtual interface (meaning that the object will contain virtual functions only)
* Then make the specialisation objects that will override the virtual functions with defined functions that will be specific to the graphics API that the engine will decide.

Overall, this is very simple to do but will have to be done multiple times because there is lots of different types of graphics object that you need to abstract:

* Vertex Buffers
* Index Buffers
* Shaders
* Vertex Arrays
* Textures
* + a lot more!

These findings did not take a long time to search for but despite this, these findings are absolutely crucial for a game engine since this is how all modern engines manage their graphics objects.

Layer System

A Layer system in a game engine is not that necessary, but when you consider the versatility of it – it then becomes very useful. A Layer system is a system which contains a “Layer Stack” which (in itself) contains multiple layers. Layers can be seen as surfaces to render on, but they are more than that. Layers can represent one part of an application which can or cannot be seen on screen – they can contain functions and other objects to call. A Layer Stack is stack-like object which contains all the layers for a particular application. The first image represents this concept, imagine that on top of layer 1 is the screen – you will be able to see all of what layer 1 has to offer but all of the other layers will still be running but you will not be able to see it. If for example, the application decides to pop the first layer then Layer 2 will be visible, if Layer 2 was popped then Layer 3 will be visible and so on. The second image is a sorting layer system and is very different to a normal layer system. This system will be used in rendering on one Application layer since it will be able to render game objects behind others. Layers in this system can be seen as transparent and the camera will be able to see all the layers.

A picture containing text, knife, weapon

Description automatically generatedChart, diagram

Description automatically generated

My research has led me to a solution which can be improved using UI. Basically, when the engine draws all the geometry on the screen it needs to go through a list of all the user-created layers and draw all of the objects that the layer contains on the screen. This works because the layers that get drawn first will be behind all the objects that will be drawn last. This can be visually represented using UI. The image below is from the Unity Engine UI and illustrates the layers in a list. So as the engine draws the layers, the layers at the beginning of the list are drawn first meaning they are drawn at the back (hence why the first layer is called “Background”) and the layers at the back of the list are drawn in front of the all the other layers (This is usually when the player is drawn for example).

Graphical user interface, application

Description automatically generated

As for the application layers, they are very much all code based and are separate objects. As part of the application’s construction these layers will be pushed into a dynamic list (called the Layer Stack). All Layers will contain certain same functions such as OnUpdate(), these functions will be called at runtime and executed (this applies for every layer on the layer stack).

Overall, these findings are very beneficial for the project and will be implemented (even when I decide to design a UI system).

Input System

An Input system is vital to a game engine as it lets the user query the state of any key or mouse button. The difficulty in implementing this is that (like most of the engine) the code needs to generic enough so that it works on all graphics APIs. The implementation must include:

- Key Presses

- Key Releases

- Mouse Button Presses

- Mouse Button Releases

- Mouse Scrolling

After research on how to do this using OpenGL, there seems to be a GetKey() function which returns an integer based on what state the key is in.

GLFW\_KEY\_PRESS = when key is pressed for the first time

GLFW\_KEY\_RELEASE = when key is released

GLFW\_KEY\_REPEAT = when the key is repeated (when the key has been pressed more than once)

There also seems to be callback functions which calls a specified function when a certain event has happened. These callback functions allow the use of customised functions which can do whatever you like.

These findings are going to be very beneficial to the engine since it will let me design how the input system will handle certain events without having to create a whole other system that works on just one API. This also can lead a creation of an event system which is an input system but handles other events like window moves and resizes.

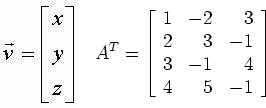
SIMD Maths

SIMD stands for single instruction multiple data. Just from that alone you can understand why this is a necessity for high performance, the ability to do multiple calculations in the time it takes to do one is excellent for speeding up calculations that will happen every frame. Every CPU will have different SIMD instruction sets that it can perform which is a hinderance when it comes to writing the code because somehow the engine needs to know what instruction sets the host processor can perform. SIMD instructions will translate to one machine code instruction which is one of the reasons why it will be so fast.

After research, there seems to be compiler intrinsic functions that will check the processor. These functions are called CPUID. These functions will query the processor for a hardcoded bit pattern in the processor which is universal for all modern processors. The bits will represent whether the processor has access to certain instruction sets. These bits need to be extracted and analysed as part of the engine's initialisation phase. Part of these bits is the hardcoded string which signifies the name of the processor for example:



As part of the mathematics module for the engine, it needs to contain mathematical functions that will have different definitions depending on what is the highest level of SIMD that the processor can perform. There needs to be a class that contains all the vector and matrix functions, so the user can just call the function and will know that it will do the function as fast as possible. The way that I found out that you need to do this is to have all the functions pure virtual and make an overriding class for every SIMD instruction set. Inside the base class you will need an instance of the class since there is only going to be one of them (this approach is called Instancing and the instances are sometimes called singletons). Then once all the overriding classes are implemented - in the initialisation of the engine - the instance will be set to the overriding class that equals the highest level of SIMD the processor can handle. There will also be Vector classes which represent mathematical objects that has more than one component. I'm planning to have a Vector2, Vector3 and a Vector4 class since that is what all commercial game engines have access to. Every Vector will have float, double and integer versions because it will add the versatility of precision. Matrices on the other hand, is less important since the engine is only going to be 2d. But i will still have a 4x4 Matrix since that is used in orthographic projection and a 2x2 Matrix since that is used in rotation.



The research done for this has been crucial and cannot be ignored because the sheer improvement in performance that this will cause is massive.

Pseudo-Random Number Generation

To have a random number generator in a game engine is not that necessary but if the user wants to add random maze generation (or random anything since this sort of thing is very common in games) then the engine will be able to accommodate that. All I know is that computers can not generate truly random numbers and hence why the random number generators are referred to as pseudo-random.

As the result of my research, I have learnt that every PRNG (pseudo-random number generator) needs an initial "seed" value which then it will generate all other random values from. The seemingly best random number generator is the Mersenne Twister which is named accordingly because it's period (number of numbers generated before the same sequence of numbers are generated again) is a mersenne prime. The algorithm is very hard to explain because a lot of it is random, but the numbers generated from them have an even distribution. The time taken to research the best random number generation algorithm has been rewarding and will be implemented into the engine towards the end of the development cycle.

2D Physics System

Static objects being rendered on the screen is not that exciting but if those objects are colliding with each other and reacting to other forces such as gravity then it gives a scene more realistic flair and is used by a lot of different 2d games. A 2d physics system is vital for a game engine to handle otherwise the user will have to implement their own physics and that is not what a game developer should be concerned with doing. Each object in the world has to be accounted for and collisions need to be resolved when one such a thing happens.

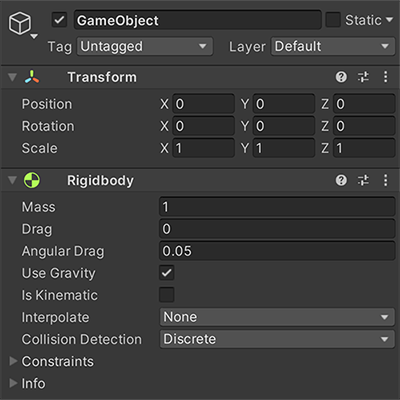
After some research, …

ECS (Entity Component System)

Most successful commercial game engines are built on an ECS which specifies that every game object can have a number of different components that make up that object, for example the objects position. Components can be added and taken away from objects during runtime to make it behave differently.

After some research, the most important component seems to be the Transform. This component holds the position, rotation and scale of the object. Since this component is vital it cannot be added or taken away at runtime – every object will have this component no matter what.

Other components include rigidbodies, sprite renderers and tilemaps. There are so many different types of components that I cannot list them all here and because of their specific nature I cannot give details on how they work, but in other sections of the documentation I will explain them.

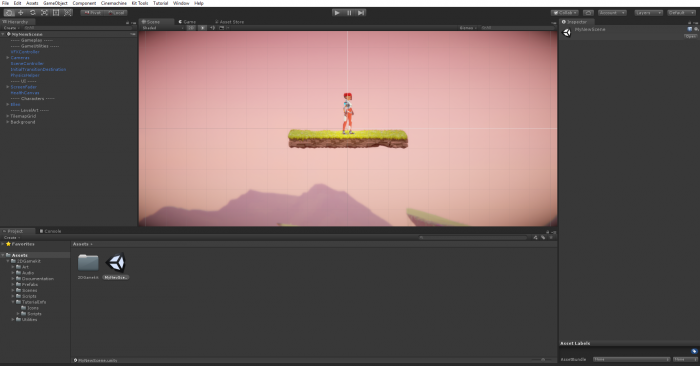


The other type of components are the user-created scripts, these have their own behaviour since it is defined by what the code inside entails. It could make a character jump when a certain key is pressed, it could make lots of objects spawn everywhere, it could contain any other logic that the user needs for their game. The variables inside the script will be settable parameters when it is added to a game object at runtime.

Overall, this research has been very useful and critical to understanding how a game engine is designed and how the idea of game objects having certain behaviours and not having all of them, but some are inactive.

Scene Runtime

Scenes are the objects that hold a game world and everything inside. Scenes should be loaded in and played using a play button and should be stopped by pressing a stop button. Scenes are not game objects but they hold all the game objects the specific world has.



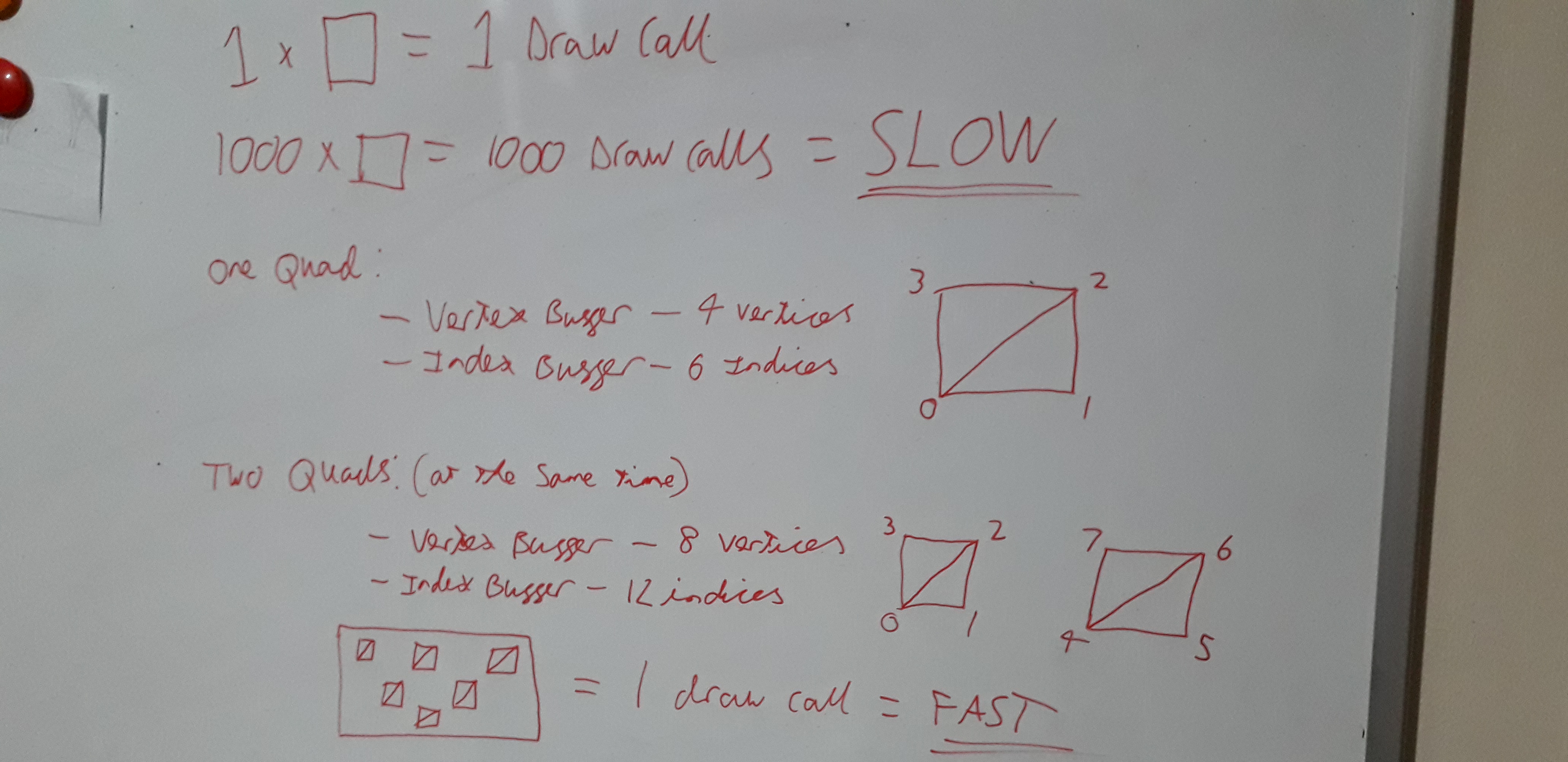
After research, a scene can be seen as a dynamic list of game objects that are rendered when the scene is selected. Only one scene can be loaded at one time. At first, this seems to be a perfect use for the layer system but the only problem with that is that all layers will be active all the time. A better solution will be to have a hash map of dynamic lists. The hash map is used because it is very fast to retrieve the dynamic lists. The dynamic list that is retrieved is then rendered instead of the current scene that is loaded. The addition of a play buttton is a lot harder to implement. This is where the layer system does come into play since every layer can have an active and inactive state and on it's activation it will call a method called OnStart() and then it will commence the start of the scene. In many ways the layers in the layer system can be seen as a list of game objects, so now we have a hash map full of layers. Although we did research of the use of a layer stack, this can be used instead of the hash map.

So to recap: every scene is a dynamic list of game objects, when you load in a new scene - the layer's game objects will be rendered, once you press the play button - the current scene will call an OnStart() method which will commence the beginning of the scene. This research has been very useful and it will be one of the core aspects of how the engine will run.

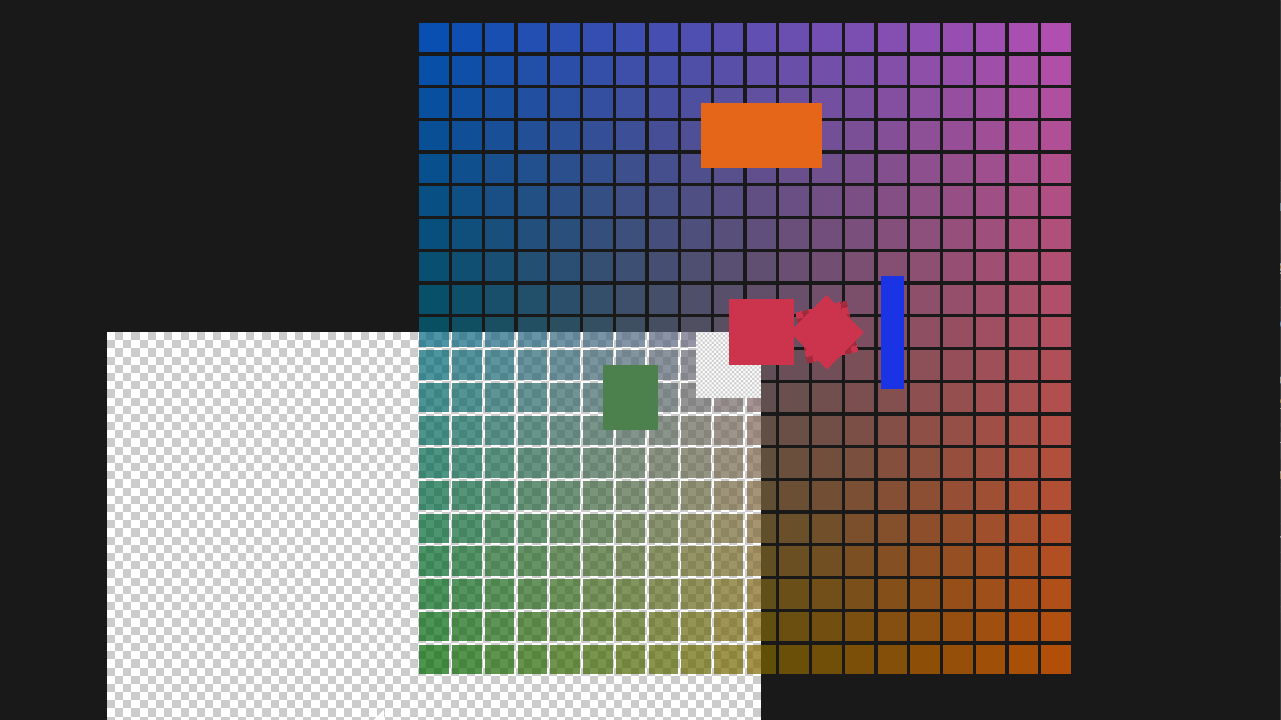
Batch Rendering

Batch rendering is the concept of rendering multiple primitives in one draw call. This is a very needed optimisation for the engine since all the primitives that need to be drawn would otherwise be drawn using one draw call each and that can quickly add up. Many draw calls are classed as slow and action must be taken in order to draw the required primitives in as few draw calls as possible.

After research, the way to batch together different primitives is to have the positions of all the vertices inside the vertex buffer. After that you need to add the extra indices to the index buffer to accommodate the other primitives and then you can draw all of the indices in one draw call.



The way to batch different colours together is to have more than one attribute inside one vertex. A Vertex isn't just a position - it can hold more than just a position - it can hold the RGBA colour values for the vertex. In order to shade a primitive a solid colour, the colours for each vertex needs to be the same. In order for the change to take effect, the shader needs to be changed in order to input the colour and then output it as the colour for the vertex.



Textures can be dealt with in the same way as colours - by adding more vertex arrtibutes in the vertex buffer. The addition of two more attributes to the vertex buffer with determine the texture coordinates and the texture index. The texture coordinates will be two floats (a x-value and a y-value) and they determine the position of the texture on the vertex. The texture index will be just one float and it will signify which texture to use. All the textures will be bound to a specific index in which the GPU will factor because the number of texture slots is specific to the host GPU. This does mean that the number of batches will be determined by the number of texture slots that the host GPU can handle. Before every batch, the maximum number of textures are bound and then get drawn in one draw call and it will repeat for the number of textures that need to get drawn. So unfortunately, the texture batching means that not all of the primitives can be drawn on the screen in one draw call but can be separated into a low number of draw calls.

So all of this research is good so far but how do you draw primitives dynamically so that each primitive has the ability to move. Well instead of hardcoding the vertices and giving them to the graphics API, you could instead just allocate enough memory to specify a certain number of vertices and dynamically stream data into the buffer. In OpenGL this is done by instead stating that the vertex buffer is set to GL\_STATIC\_DRAW, state it to be GL\_DYNAMIC\_DRAW. So now each frame, data can be streamed into the buffer which can change the vertices. So, not only can you change the positions of each of the vertices you could also change the colour, texture and texture coordinates and it will still be batched together and drawn in as few draw calls as possible. Although there is a problem with is and that is that the index buffer does not change, so you can only draw the number of vertices that the index buffer states upfront. The solution to this problem is to make an index buffer that contains the same pattern every 6 indices. The pattern is { 0, 1, 2, 2, 3, 0 } and then you would add 4 to each recurrence of this to produce an index buffer that contains all the indices for all the potential quads that need to be drawn.

Overall, this is a lot of research that was put into this and I can't stress enough how much of an optimisation this is. This will speed up rendering so the user can render a large scene without the engine slowing down that much.

UI & Text Rendering

Ease of use partly comes from how good the UI is and how well it is suited to the application. So, this part of the engine needs to be done professionally and efficiently to result in the best product. Part of the UI will be text and rendering text isn't the easiest thing in the world because you could think of text as a texture but the text can be and will be modified during runtime so that idea is not the right answer. I hope to find a way of rendering text so it can be modified during runtime

After research, ...

Synthesiser

The implementation of a synthesiser is not neccessary for a game engine but i have decided to have one in mine because it will result in making game music easier for the developer. A basic synthesiser must include: to be able to make sound waves of different pitch values, to be able to make different instrument sounds and a nice UI to piece it all together.

After research, ...

Art Drawer

This is also a not needed piece of mini software that my engine will have. All that it will have is the ability to view and edit very small images to be used as pixel art. It will also be able to write to a PNG file and then it can be used for use within scenes.

After research, the best way to do this is to render a grid with coloured quads representing the pixels, the quads that are clicked will change colour to a specified colour beforehand and if you right click you will be able to delete the quads (which will just turn the alpha to 0). The beauty of this is that with the batch renderer the whole scene can be drawn in a single draw call. There will be an array of all the colours that are on the grid for use when exporting to a PNG file.

Writing to a PNG is very specific because of the way the file type was made:

- To start off a PNG file it needs to have the PNG file signature which is this particular array of eight decimal values: { 137, 80, 78, 71, 13, 10, 26, 10 } (these values need to be encoded as one byte each so that it fills the first eight bytes of the file.

- Next needs to be an IHDR Chunk which contains the following chunk data

- Width: 4 bytes

- Height: 4 bytes

- Bit depth: 1 byte

- Colour type: 1 byte

- Compression method: 1 byte

- Filter method: 1 byte

- Interlace method: 1 byte

- Then an IDAT Chunk needs to be encoded which is the actual image data

- Then an IEND Chunk needs to end the file (this chunk contains no data)

Links

[An OpenGL library | GLFW](https://www.glfw.org/)

[https://glad.dav1d.de](https://glad.dav1d.de/)

[How to Create a Custom Physics Engine - Envato Tuts+ Game Development Tutorials (tutsplus.com)](https://gamedevelopment.tutsplus.com/series/how-to-create-a-custom-physics-engine--gamedev-12715)

[Intel® Intrinsics Guide](https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#ig_expand=6934,6954)

[Batch Rendering - YouTube](https://www.youtube.com/playlist?list=PLlrATfBNZ98f5vZ8nJ6UengEkZUMC4fy5)