
# GAME ENGINE PROGRAMMING REPORT

**Leon Sen (S5222204@bournemouth.ac.uk)**

## Overview

A game engine using C++, OpenGL, GLEW and SDL libraries, with many features and components that are necessary for game development. Computer graphics via OpenGL, using a shader program that takes into any vertex and fragment shader file and renderer to draw models to the screen using VAO class with a model loading library. The game engine core which the game revolves around; contains the game loop, adds entities, as well as allowing other components to access core features such as keyboard or the game environment, store essential objects such as colliders and a camera as well as delete those objects once redundant and initialise SDL (for creating window, event handling) and ALC (for audio). Resources such as object files or sound files are handled by the resource manager in which saves up on memory by saving all resources into itself and calls back to that resource whenever required. Collision is supported by the engine via sphere colliders and a trigger to set off any behaviour once something has collided and an audio system which plays music with the wanted audio file.

My project has a small tech demo showcasing all the features of the game engine. The player can move a cat using WASD, there is music playing and other cats around the scene, the player can move and collide with the player, triggering their model with texture change and a sound effect. The camera is at a top-down view, so the player can see around the scene.

For documentation, there is a doxygen html that showcases information about my program such as the header files and .cpp files in detail with comments. As well as a git log with every version of the engine; showing the progression throughout the engine's creation.

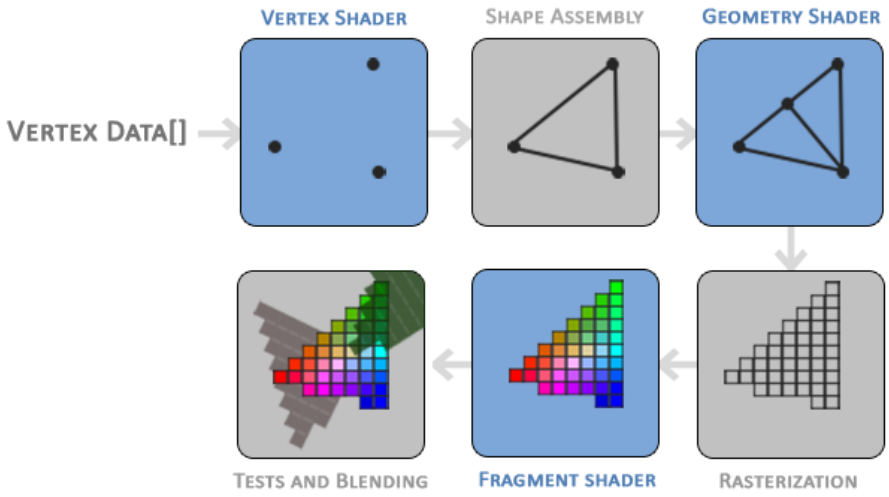
**Initial Specification**

| | |
|---|---|
| Component and Entity System | This architecture allows the user to create any number of entities and allow them to attach game components to them; these components provide the desired behaviour. |
| Resource Manager | The resource manager helps the game engine save resources by keeping all resources within and re-using it whenever that resource is called again. |
| Shader Program | The Shader Program allows the user to set uniform locations and send it to the vertex and fragment shaders. As well as read the vertex and fragment shader files. |
| VAOs & VBOs | Vertex Array Objects store the Vertex Buffer Objects, "designed to store information to complete a rendered object". Vertex Buffer Objects hold information such as vertices, normal, texture cords. Etc. |
| Model and Textures | Our engine allows the user to locally add an .obj file and .png textures and store it within the resource manager. |
| Model/Triangle Renderer | The model renderer takes in the models and texture files and renders it with the shader program. |
| Audio Player and Audio Clips | The audio clip is set up by the resource manager and the audio players plays any audio clips (.ogg) |
| Camera | Camera is a game object which position, and rotation is put into the shader program via uniform location. |
| Input | Game engine has keyboard input. |
| Collision | Our game engine has collision and trigger class, when a game object collides with another game object with a trigger, onTrigger() will be called. |
| Debug | Our engine has an exception class which tells the user the error. Custom error handling. |
| Game Engine Core | The entire game engine uses the core, it plays an important role. It's duties are to: initialise SDL and AL, run the main game loop, register and unregister colliders and cameras, allow entities to access important components: keyboard, environment and to add the entities into our game. |
| Transform | Allows developers to set position, rotation and scale. As well as control the player. |
| SDL | SDL for events and the window. |

## Research

## OpenGL

One of many reasons why developers would choose a game engine rather than to start from scratch is to avoid learning/utilising computer graphics from scratch. My game engine has several OpenGL features.




(Representation of all stages of the graphics pipeline)

This image is essentially how all the graphics are being shown within our engine, in our engine we use a triangle renderer which does all these steps minus the geometry shader. This triangle renderer was initially used to figure out how to implement OpenGL within our engine but is now used for debugging purposes.

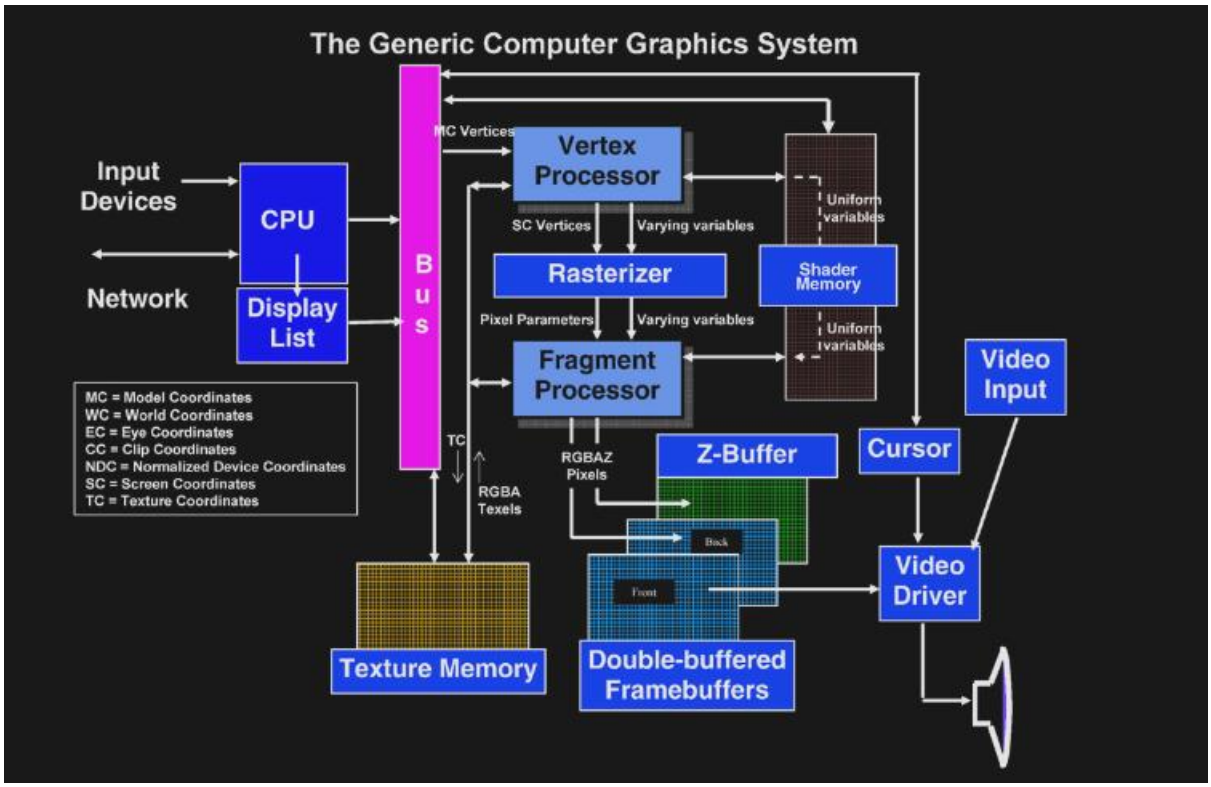
Firstly, to start drawing, we need to give some input vertex data into OpenGL, this data needs to be 3D (x, y and z coordinates) since OpenGL utilises 3D graphics, using a triangle as an example; we need to send in a vertices array with each vertex having a 3D position.

```
//Create Position Buffer
std::shared_ptr<VertexBuffer> positionsVbo = std::make_shared<VertexBuffer>();
positionsVbo->add(glm::vec3(0.0f, 0.5f, 0.0f));
positionsVbo->add(glm::vec3(-0.5f, -0.5f, 0.0f));
positionsVbo->add(glm::vec3(0.5f, -0.5f, 0.0f));
```

(Example code from the engine)

Furthermore, we send this input to the vertex shader, this input is stored as vertex data as memory within the GPU. This memory is then managed through the Vertex Buffer Object which can stored many vertices within the GPU's memory. Since sending data to the graphics card from the CPU is slow, using VBO's is a great method of doing so as we send it all in one batch at once. Once that data is inside the GPU's memory the vertex shader has instant access to the vertices. Above shows an image sending in the positions to the Vertex Buffer.

## Shaders




(Generic Computer Graphics System)

From the figure and mentioned above, there are two locations in which a developer can input their custom shader code into which is the vertex shader and the fragment shader. The vertex shader takes in the vertices from the Vertex Buffer Object, then it needs to translate them into the world space via world coordinates then transform them into eye-space coordinates using a viewing transform. In my engine, my vertex shader has multiple variables: u_Projection which is the perspective projection matrix. Using glm::perspective creates a large frustum shape that defines the visible space to the player, anything outside of the frustum won't show up in clip space volume and will be clipped. Moving away from triangles, going 3D requires a model matrix (u_Model), the model matrix(glm::mat4) consists of translations, scaling and/or rotations. Lastly u_View, our view matrix, which is essentially our camera location, we control our camera location via transforming it.

## Resource Manager

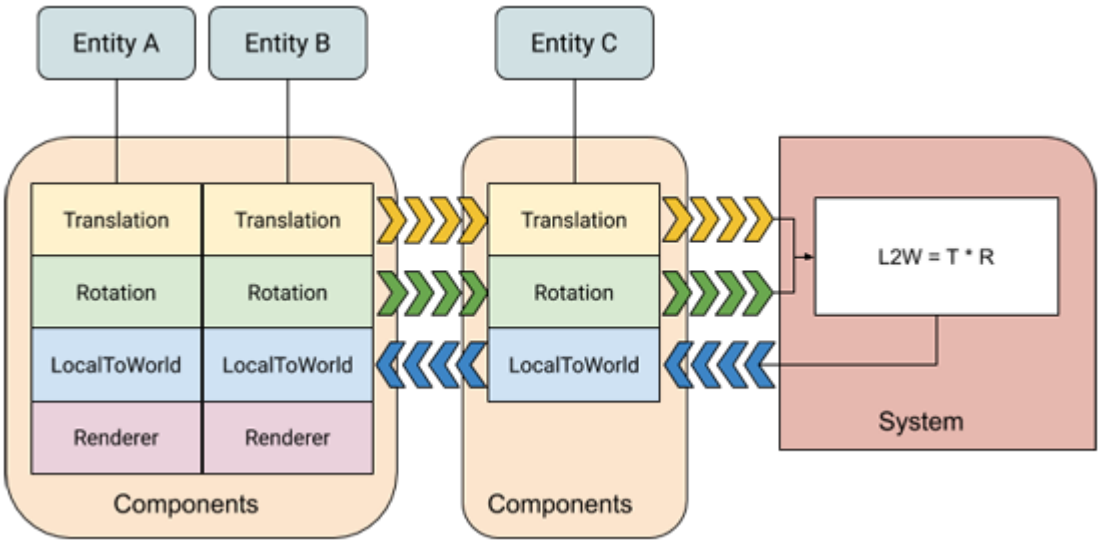
According to Scott B, "All computer applications are databases" "Games typically contain multiple types of databases." These statements are agreeable within game engines as lots of files such as models, audio, assets, scripts, etc. all need to be loaded within game engines and some sort of resource manager is needed to keep the integrity of a game engine in terms of memory and resources, if a game engine didn't have an efficient method of doing so it will slow down and make a game experience less enjoyable.

**Entity Component System Model**

The ECS model is an "Architectural pattern designed to structure code and data in a program", this article also mentions that "It appeared in the field of Video Game development where teams are typically separated between programmers designing logic and tools and designers producing scripts and multimedia content" This supports the theory that ECS is a competent system for game development, as it allows different people to work on a game despite having different roles. The interaction between programmers and designers are critical to producing a video game and having a game engine allow this type of interaction makes game production faster and resourceful.

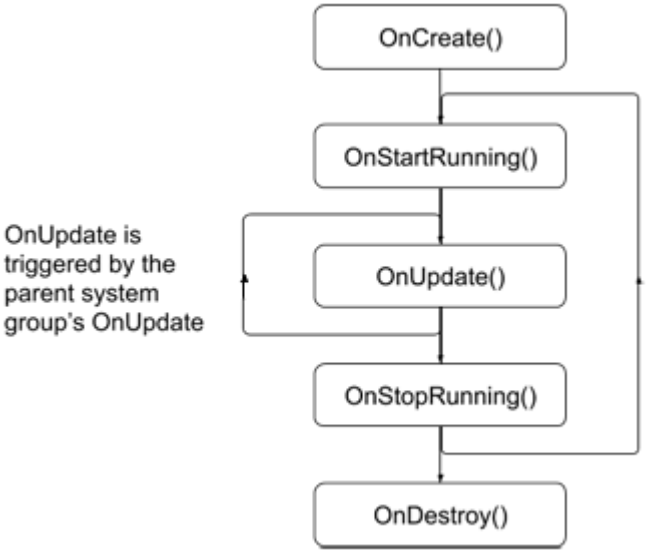
The main elements of Entity Component System Model are:

- Entities represent the elements of the program, similar to objects in Object-Orientated Programming but have no data or methods
- Components represents the data (e.g. Model Renderer or Sphere Collider) and are dynamically associated with entities.
- Systems/Core runs continuously, the main game loop essentially; the core also holds some of the important functions for the game such as event handling. Entities have access to core to gain access to some of these functions.




(ECS demonstration from Unity)

"To create a system, you program the necessary system event callbacks", this is referred from Unity's Creating a system; system callback functions allow entities to interact and behave the way developers intend to.




My engine has similar system event callback functions which include: onInitialise(), onBegin(), onTick(), onDisplay(), onCollision().

The Entity Component System approach felt favourable personally compared to the inheritance approach and data orientated design is the usability. This is evident with the commercially successful game engine Unity. Unity is used by not only game developers but other forms of media and business such as work simulations. ECS has a simple to understand design, thanks to this it is easier for prosumers to use. Our engine can be more powerful compared to Unity, this is because we can use C++ which is a powerful language, we do not have to worry about editor limitations and we can handle resources better. In addition, since smart pointers are favourable in our game engine they also benefit CES as there are a lot of objects that can be hard to manage.

## OpenAL

OpenAL is similar to OpenGL's API, the similarities are that they are both state based meaning you have to interact with handles and identifiers. When interacting with OpenAL API, there are three main objects: these are the listeners who are the player's ears are and are positioned in 3D space. The source (Audio Source) are the speakers that emit the desired sound in an 3D space. These ears and speakers can be moved around in the world space and hear changes according to your position. The final object are the buffers that hold the samples of the audio that the developer wants the player to hear. OpenAL has similar data types to OpenGL and function similarly.

| OpenAL Type | OpenALC Type | OpenGL Type | C++ Typedef | Description |
|---|---|---|---|---|
| ALboolean | ALCboolean | GLboolean | std::int8_t | 8-bit boolean value |
| ALbyte | ALCbyte | GLbyte | std::int8_t | signed 8-bit 2's-complement integer |
| ALubyte | ALCubyte | GLubyte | std::uint8_t | unsigned 8-bit integer |
| ALchar | ALCchar | GLchar | char | character |
| ALshort | ALCshort | GLshort | std::int16_t | signed 16-bit 2's-complement integer |
| ALushort | ALCushort | GLushort | std::uint16_t | unsigned 16-bit integer |
| ALint | ALCint | GLint | std::int32_t | signed 32-bit 2's-complement integer |
| ALuint | ALCuint | GLuint | std::uint32_t | unsigned 32-bit integer |
| ALsizei | ALCsizei | GLsizei | std::int32_t | non-negative 32-bit binary integer size |
| ALenum | ALCenum | GLenum | std::uint32_t | enumerated 32-bit value |
| ALfloat | ALCfloat | GLfloat | float | 32-bit IEEE 754 floating-point |
| ALdouble | ALCdouble | GLdouble | double | 64-bit IEEE 754 floating-point |
| ALvoid | ALCvoid | GLvoid | void | void value |

**C++ Smart Pointers and Performance**

It is very easy for programmers to write code with memory leaks due to compilers typically not support their work with error or warning diagnostic to leaks. “Smart pointers play an important role in avoidance of memory leaks in C++ because they enable the programmer to unfocus on memory allocation” from this it is in a programmer’s interest to use smart pointers during the development of an C++ game engine. As we do not want any memory leaks that may slow down or even crash the game engine.

Smart pointers in C++ take advantage of temple construct meaning they are independent of the type of managed memory; these templates are important due to their performance on an program. Smart pointers are based on the RAII (Resource Acquisition is Initialization), the advantages of using RAII as a resource management is that it can provide encapsulation, exception safety and locality. “The RAII principle: constructor and destructor are automatically executed in a well-defined moment. Invocation of these operation is based on the smart pointers objects’ lifetime.” The standard smart pointers are: std::auto_ptr<T>, std::unique_ptr<T>, std::shared_ptr<T> and std::weak_ptr<T>. In my engine I will mostly be using shared pointers and weak pointers.

According to this article based on the performance analysis C++ smart pointers performance are slower compared to their raw pointer counter part in terms of runtime. With this we must optimise the use of our smart pointers throughout our engine to avoid having a slower runtime. “Without optimisation, smart pointers have very large overhead construction, destruction, copy and assignment against raw pointers.”

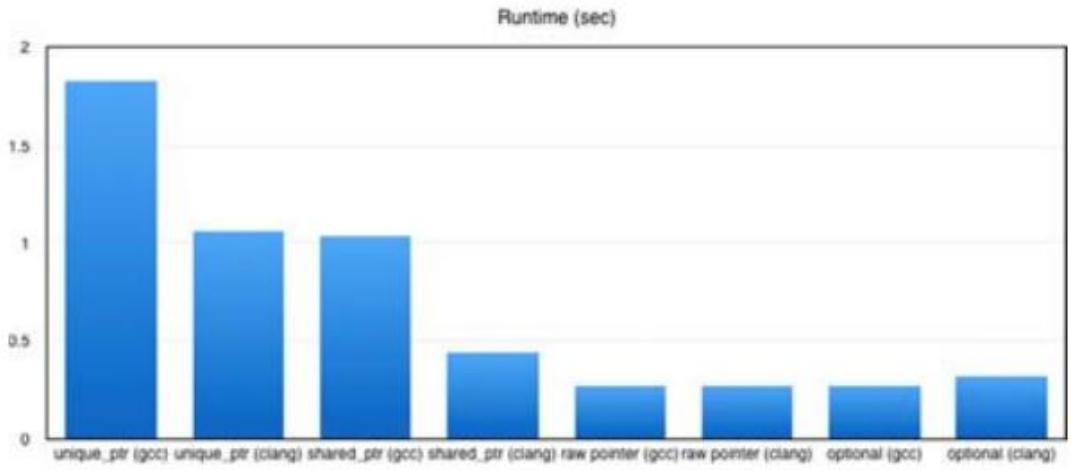
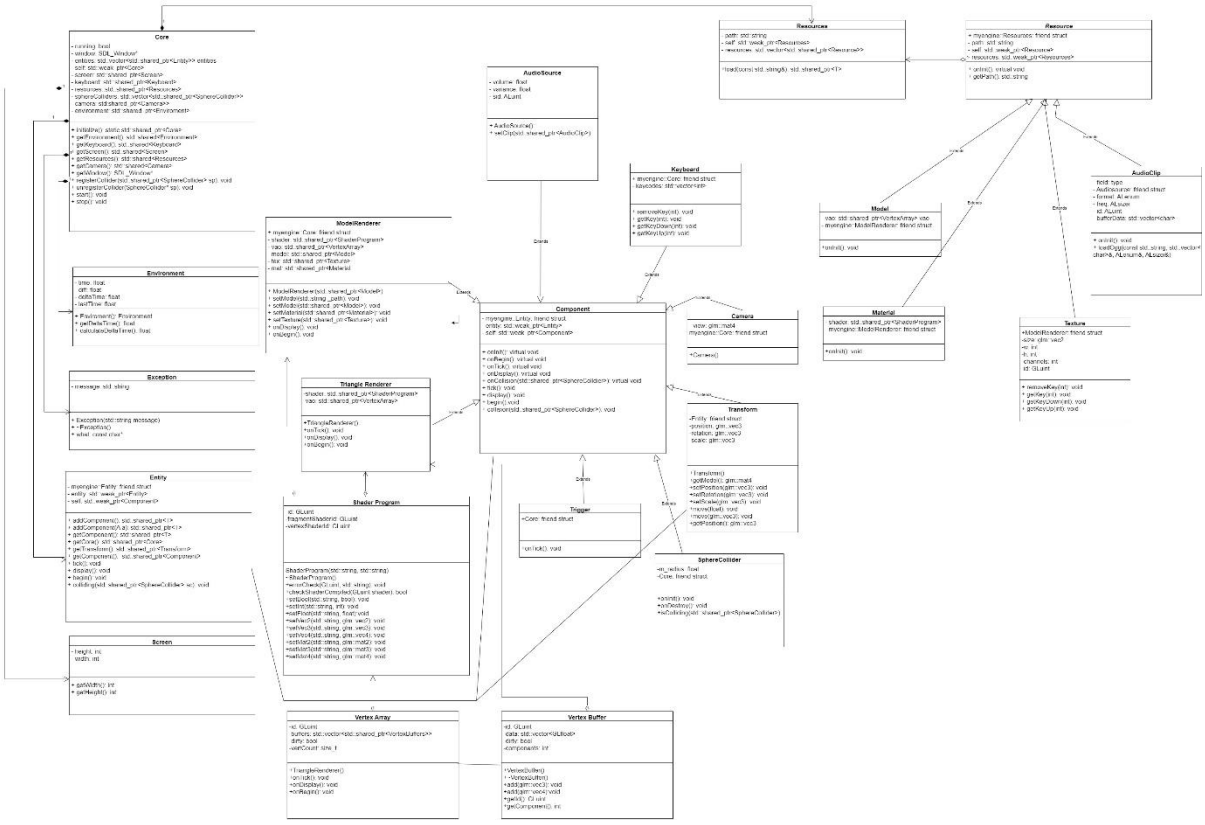



*Fig. 1.* The runtime of construction with - O0

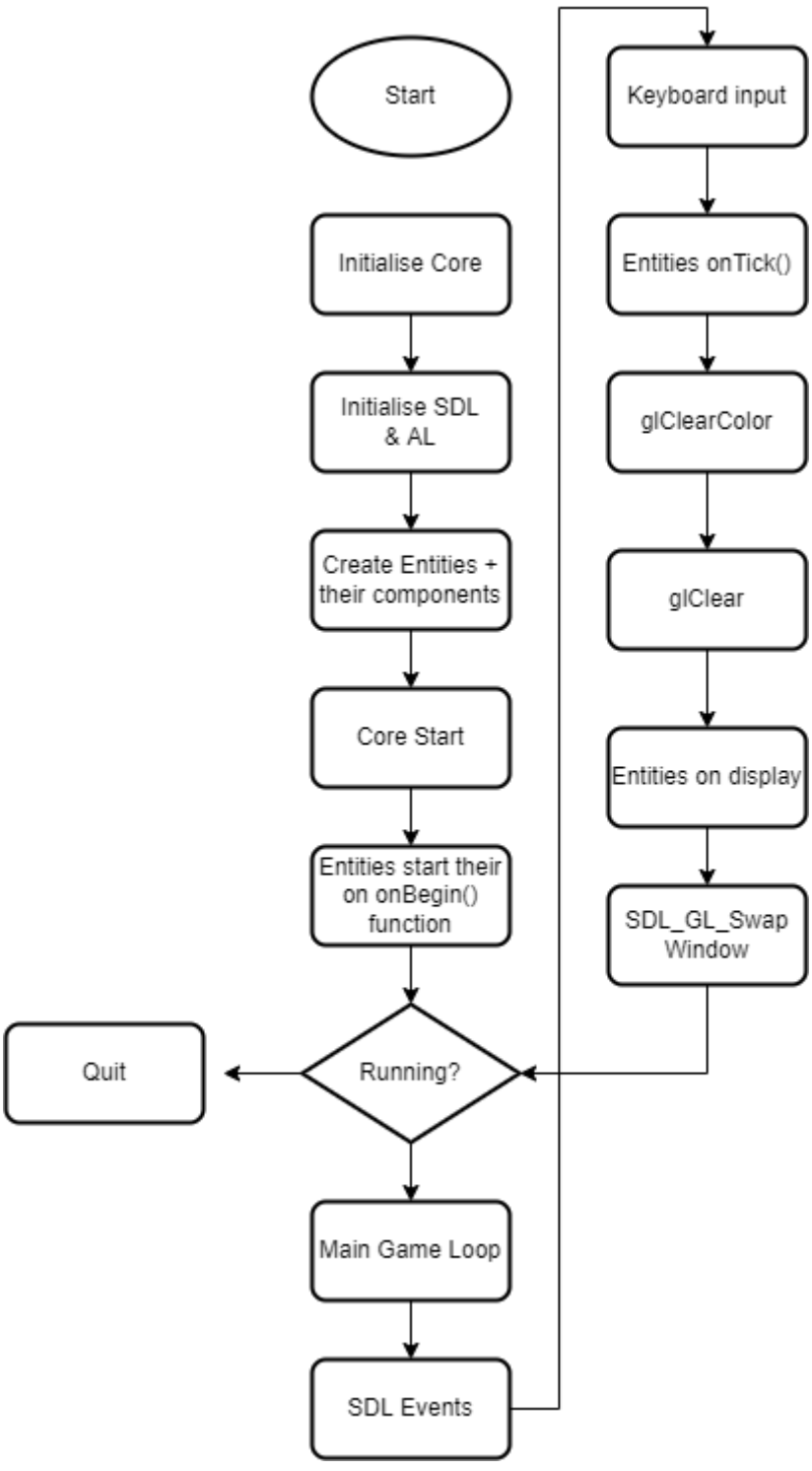
Overall, personally the use of smart pointer outweighs the use of raw pointer as raw points doesn’t provide the same amount of utility that smart pointers have. Raw pointers may be faster than smart pointer however the RAII system, resources management and the safety of memory leaks make smart pointers more desirable within my engine. The runtimes maybe slower but as hardware grows; these runtimes won’t be noticeable especially with the uses of SSDs and faster CPUs and GPUs.

## Diagrams

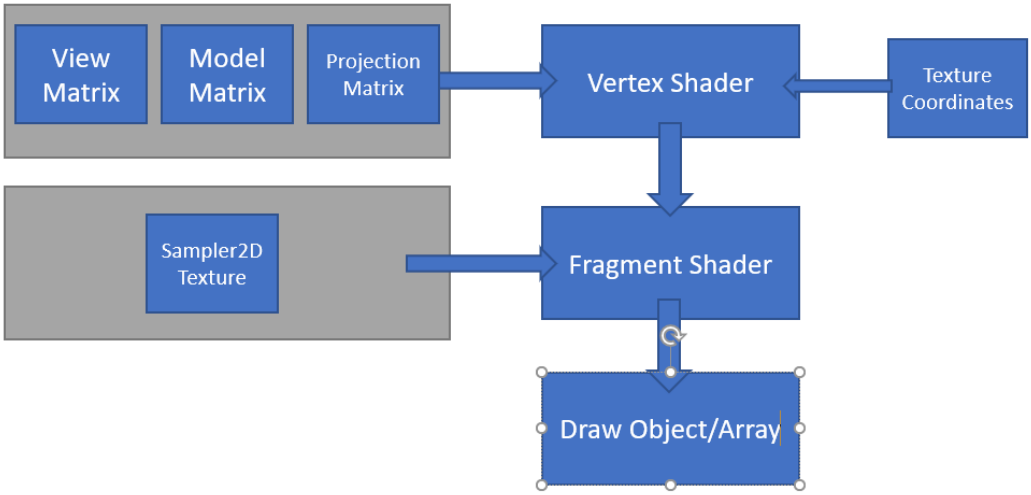


My game engine's UML Diagram

This UML diagram shows every single structure and how they interact with each other using UML class arrows and symbols.

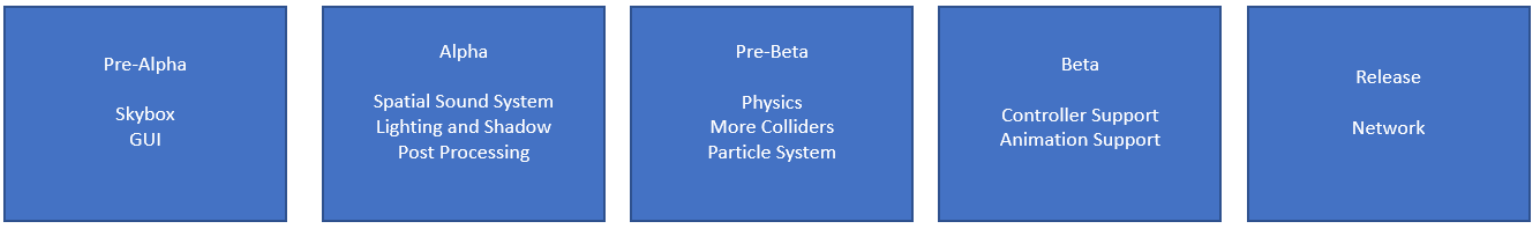



Flow chart of the main game loop




A simple diagram showing the steps for rendering within my engine

| Strength | Weaknesses |
|---|---|
| Model / Triangle Renderer – the ability to render models and shapes is extremely important for game engines, as they represent something on the screen. | No Lighting or Shadows – this is required for the game to be more realistic and visibility appealing |
| Shader Program – allows the developer to change how the models are being drawn. As well as having textures on models gives personality and shows more detail on models. | No GUI/UI – Developer need this to display important information to the player |
| Collision & on Trigger – the ability to collide and trigger is a key part of gameplay. For example, collecting coins. | Only one collider, which is the sphere collider. With other colliders – collision will be more accurate. |
| Camera -The ability to manipulate the camera location can change the type of game the developer wants to have. E.g. first person or third person games. | No Post Processing – no special effects (bloom, blur, etc) |
| Keyboard Input – move or control the player using the keyboard. | No animation manager – without animation, the models and overall feel of the game will feel less lively. |
| Transform & Movement – the ability to move and set positions, rotations and scale can help design an level or simply placing objects in desired locations | Limited Debugging system – if the developer has no experience with C++, they may get confused with some of the errors. We need to provide them with some debugging messages. |
| Audio – having audio and music will make the game feel more immersive and more enjoyable. | No environment or spatial audio system – the developer cannot have sound effects play at a specific location without spatial audio. |
| VAOs and VBO classes – can draw other shapes and meshes using these classes for future use. | No other forms of input such as controller – developers may want to develop a console game that requires controller support. |
| ECS System – great for usability | No Physics – physics is required for again the realism or even the gameplay of the game. |
| Resource Manager – Allows the game engine to store resources, so the developer doesn't have to keep adding the same ones. This also saves memory on our game engine. | Skyboxes / Cubed texture maps – game engines can benefit from skyboxes, instead of a solid colour having a sky of some sort adds realism. |




Roadmap to further improve my game engine.

## Conclusion

The Game Engine I've created utilising the Entity Component System and C++ is a great start to a game engine covering many of the main and important features that are required for a developer to only create a simple game, as there are limitations. The use of the ECS allows for developers to easily understand and learn my engine efficiently especially if they have experience with Unity. Since there are a lot of objects created through using ECS, it can be quite cluttered; using smart pointers, my engine can be easier to manage. RAII allows objects that are out of scope to call their destructors, cleaning up the engine. Along with the resource manager, my engine is quite resourceful and efficient with memory. OpenGL allows computer graphics to be used on my engine; the usage of my renderer and shader program produces graphics for my engine, however I have not utilised OpenGL to it's full potential; there are many features that need to be implemented – same goes for OpenAL; producing sounds for the player, however a spatial sound system would benefit my game engine. Whilst my game engine has the means to produce graphics and sound, the features such as the collider, transform, keyboard and camera are the means for gameplay. However, this is where I believe my game engine lacks in, gameplay features. Adding more features such as physics, more collider, rigidbody and controller support – it could add so much depth to gameplay within my engine allowing for more opportunity of gameplay. With Doxygen, developers can see the documentation of my engine through the comments I've put in my engine. And the Github log provided developers with the versions and updates during the development of this engine.

**References**


Khronos Group, 2018. *Tutorial2: VAOs, VBOs, Vertex and Fragment Shaders (C / SDL)* [online]. Portland: Khronos Group. Available from: https://www.khronos.org/opengl/wiki/Tutorial2:_VAOs,_VBOs,_Vertex_and_Fragment_Shaders_(C_/_SDL) [Accessed 16 January 2022].

Raffaillac, T. Huot, S., 2019. Polyphony: Programming Interfaces and Interactions with the Entity-Component-System Model. *HAL Open Science* [online], 2.1, 8:2. Available from: https://hal.inria.fr/hal-02147180/document [Accessed 16 January 2022]

Unity, 2021. *Creating a System* [online]. San Francisco: Unity. Available from: https://docs.unity3d.com/Packages/com.unity.entities@0.17/manual/ecs_creating_systems.html [Accessed 16 January 2022].

Bailey, M., 2007. Teaching OpenGL shaders: Hands-on, interactive, and immediate feedback. *Computers & Graphics* [online], 2,. Available from: https://www.sciencedirect.com/science/article/pii/S0097849307000647 [Accessed 16 January 2022].

Vries de, J., 2021. *Coordinate Systems* [online]. Available from: https://learnopengl.com/Getting-started/Coordinate-Systems [Accessed 16 January 2022].

Bilas, S., 2009. *Gem: A Generic Handle-Based Resource Manager* [online]. Available from: https://web.archive.org/web/20090426074540/http://www.drizzle.com/~scottb/publish/gpgems1_resourcemgr.htm [Accessed 16 January 2022].

IndieGameDev, 2020. *The Complete Guide to OpenAL with C++ - Part 1: Playing a Sound* [online]. Available from: https://indiegamedev.net/2020/02/15/the-complete-guide-to-openal-with-c-part-1-playing-a-sound/ [Accessed 16 January 2022].

Babati, B. Pataki, N., 2017. Comprehensive Performance Analysis of C++ Smart Pointers. *Pollack Periodica* [online]. Available from: http://real.mtak.hu/66602/1/606.2017.12.3.14.pdf [Accessed 16 January 2022].