DOOM-STYLE 3D GRAPHICS RENDERING ENGINE (srqEngine)

Sharjeel R. Qaiser

Reg No. 1804071

Dr. Michael Fairbank - Dr. Rassouli Borzoo

University of Essex, Computer Science Bsc.

ACKNOWLEDGEMENTS

My deepest gratitude to my supervisor Dr. Michael Fairbank for guiding me throughout my project, providing constructive feedback, being kind, supportive and giving me advice on personal issues as well. A warm thank you to the CSEE department for aiding with any technical issues despite the current climate. Thank you to Dr. Vishwanathan Mohan for solving all feedback related issues I had. Thank you to my friends for making my time at university a pleasurable experience and thank you to my family for instilling me with my work ethic and values which has brought me this far.

ABSTRACT

The field of computer graphics has gone through countless cycles of improvement as the technology that powers it, Graphics Processing Units (GPUs) continue to increase in computational performance. Today, we are able to render hyper-realistic models, particle effects, emulated physics and much more in real-time using GPUs.

DOOM is a video game series created in 1993 by John Carmack et al. This video game was one of the earliest implementations of a First-Person Shooter game, in which the point of view is from the player’s eyes. The game involves navigating through rooms that are contained within a virtual world which the player is confined in, shooting monsters, collecting points and more.

In this project, both elements from modern graphics rendering techniques and the DOOM video game are applied to create a 3D graphics rendering engine. The engine is implemented with C++, and modern OpenGL to interact with the graphics API. It contains several features which showcase the power of graphics rendering, and a world-based implementation with a first-person camera, basic physics such as collisions, jumping with gravity.

This report describes the literature surrounding the field, building up each individual concept that was implemented from the ground up, a complete overview of the architecture of the engine, descriptions of features with their implementation details and information about the project management involved since the beginning of the project.

TABLE OF CONTENTS

DECLARATION OF AUTHORSHIP

I hereby declare that this dissertation titled "DOOM-Style 3D graphics rendering engine (srqEngine)" is my own original work. The information in the literature has been dutifully acknowledged in the text and list of references.

All figures, diagrams, graphs and illustrations are my own and have been created using a variety of tools such as Photoshop, Tinkercad, draw.io and GNUplot. All screenshots are my own and have been captured from the implementation of my graphics rendering engine.

NOMENCLATURE

CG	Computer Graphics
FPS	Frames Per Second
GPU	Graphics Processing Unit
CPU	Central Processing Unit
NDC	Normalized Device Coordinate
FOV	Field of View
RGB	Red, Green and Blue
AABB	Axis-aligned bounding box
VBO	Vertex Buffer Object
VAO	Vertex Array Object

LITERATURE SURVEY

I. Context

This project implements a 3D graphics rendering engine, using game design features from the early DOOM game (1993).

Graphics rendering game engines are vital to the multi-billion dollar video games industry, as they empower developers to produce fully implemented video games in a much shorter amount of time than normally required, as a plethora of prebuilt features and components (relevant to their work) are available to them out of the box. The engines typically abstract away a lot of the low-level implementation details necessary to render graphics onto their display.

Video game developers may come from a range of different educational backgrounds, many of whom may not be well-versed in computer science and technology, making it harder for them to get started. Other developers may be experienced but are looking for a program to quickly prototype their video game ideas without having to spend time concerning themselves with low-level details and extensive setups. Addressing this problem is important in order to grow the industry, by providing a user-friendly and accessible tool that simplifies the underlying computer graphics concepts and potentially speeds up development time as a result.

This project directly addresses these needs by providing a rendering engine, named *srqEngine*, that not only abstracts away low-level implementation details, but also provides a set of prebuilt game design features that developers may or may not use in their own video games.

This is an open-source project under the Apache License Version 2.0, which permits commercial use, modification, distribution, patent use and private use. Trademark rights are not granted. The license includes a limitation of liability and does not provide any warranty.

II. Introduction

This section details how 3D computer graphics (CG) are created, how they are rendered to the screen, how we can view CG from different perspectives, how certain concepts from physics are modeled in the engine and how CG rendering works in real-time.

Computer graphics refers to an array of colored pixels on the display which are arranged in such a manner so that they form an image, or any type of geometry in general that can be viewed. They are rendered in two dimensions, across the width and height of the display

III. Graphics Rendering Pipeline

To render graphics onto the display, the machine has to go through a series of steps known as *the graphics rendering pipeline*, but prior to that, we need to supply the pipeline with a group of inputs for it to perform the tasks necessary to render the CG. This is commonly referred to as the application stage, as explained by [1, pp. 13-14]. Thinking in terms of implementation, this would be the actual development phase where we pass data through the pipeline to be rendered. The most basic form of input to the pipeline is the vertex, which represents a corner in a geometric shape. Inputting an array of vertices could be interpreted as defining the geometric shape of the composite graphic, eventually down the pipeline.

The graphics rendering pipeline is divided into two main sections. (1) Geometry Processing and (2) Rasterization, which are both further divided into tasks that run functional stages, one after the other (hence the name pipeline).

Geometry processing (see *Fig. 1.1*) entails making sense of the input data (primarily categorized vertices) and transforming the input under certain constraints to pass onto the rasterization stage.

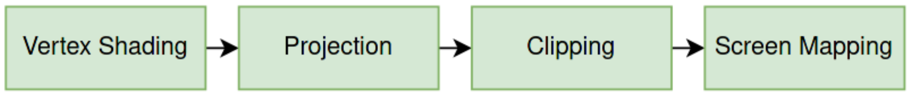


Figure 1.1 Geometry processing in the graphics rendering pipeline

A. *Vertex Shading*

The purpose of this task is to determine the position of the vertices given, and the role that they will serve when the graphics are rendered (a vertex can represent multiple things).

The position is dependent on the coordinate system which is being used. In *srqEngine*, there are several coordinate systems that a vertex can be represented in, namely the *model*, *view* and *projection* coordinate systems. The model system can be thought of as a single object or entity, such as a cube model that we are loading in. This would have 8 vertices (connecting each edge of the cube), and the position of these vertices would be in respect to the model coordinate system's origin. These vertices' positions would be different if we translated it into another system (as the origin would change). We might be moving the cube itself, as a standalone entity in a greater encompassing system, commonly called the *world coordinate system*, as inferred by [2]. This would mean that the origin of the cube is represented as a single point, so the details of the cube's vertices are hidden away in a more local coordinate system (the model).

There are various roles for vertices. The most common one can be thought of as connecting a model's edges together (as mentioned with the cube example). However, there are others such *texture* and *normal* vertices, as discussed later.

B. Projection

This stage involves a transformation applied (with a matrix) to the coordinate positions calculated in the previous stage. This transformation is known as *view-projection* and determines how we are viewing those vertices in regard to perspective. Since we want to emulate a 3D world, we need to consider a 3rd axis (*z*-axis), to achieve depth. Applying this transformation will reposition these vertices, so that we see it from the perspective we want.

C. Clipping

This builds on the previous stage, projection. Perhaps we are viewing our model from a specific perspective and then we decide to render another model which does not enter the field of view of our current perspective. To save GPU processing time, these vertices don't get rendered, even though our application may know of their existence.

D. Screen Mapping

In previous stages, I described the use of a 3rd axis and even multiple coordinate systems that we can represent our vertices in. However, up until this point, we are still viewing the graphic on a flat 2D display. This stage involves calculating the normalized device coordinates (NDC) for any given vertex. Screens may come in many different sizes and resolutions, so to ensure that the coordinate gets placed in the same location on every device, we need to normalize it - which is called NDC after the normalization calculation. For any given coordinate, OpenGL accepts it between the range [-1.0,+1.0] for both axes of the display. Therefore, we introduce a 4th component for a three-dimensional vertex, *w* – called the homogenous coordinate. Which is a factor that normalizes the other coordinates, in order to be rendered [3].

Rasterization (see *Fig 1.2*) involves using the vertices provided from the geometry processing stage to actualize the geometry onto the screen and represent them with pixels.

An in-depth elaboration on this section of the pipeline isn't too critical because it isn't necessary to understand the implementation details of *srqEngine*, however it is important to know that this stage will get the vertices from the previous one, form triangles out of them and create the shape you wish to draw (a cube wouldn't be made up of 4 quads, rather 8 triangles), and draw pixels on the screen according to any shading parameters that were passed in the application stage through a shading language, as discussed in the technical documentation.

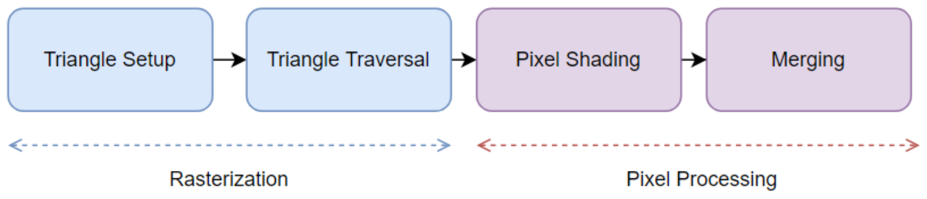


Figure 1.2 Rasterization in the graphics rendering pipeline

IV. Coordinate Spaces

As indicated before, *srqEngine's* model vertices go through several coordinate systems before being rendered in the correct positions. This is to simplify the understanding of how a vertex is positioned in space during implementation. For example, a *local* vertex position in respect to the model would be different to its *global* position, which would be represented within another coordinate system where the entire model itself is considered to have a single position (not the individual vertices). Likewise, there are other coordinate systems each with a different level of abstraction. I will outline the ones in *srqEngine* and simultaneously explain how a vertex moves from one coordinate system to another.

A. Local Space

This is where each individual vertex of a model is located with respect to an origin. For simplicity, the origin is set to (0,0,0) in the *x*, *y* and *z* axes in the engine.

B. World Space

In this space, we consider the model as a whole, to have a single coordinate representing it.

In order to transform the model so that it moves as a whole, with all its vertices, a transformation matrix must be applied to each individual vertex. This matrix could be for translation, rotation, shearing etc. This transformation matrix is called the *model matrix* because it only affects the individual model that all the vertices were a part of in the local space coordinate system. When we apply this matrix, the model is considered to be transformed into *world space*, because the matrix is applied universally to all vertices in that model. We aren't specifically editing each individual vertex (which would be a local-space transformation).

C. Camera Space

This is the final coordinate system that the model is transformed into using another transformation matrix, the v*iew-projection matrix*, before it is converted into NDCs and displayed onto the screen. The *view-projection matrix* is a combination of two separate matrices pertaining to the perspective that the user is viewing the model in, and the direction that the user is looking at. The key point here is that the model in the *world space* gets this matrix applied to it, and only after that, can we consider it to be in *camera space*, because all the existing models (and new ones if we were to create any) are in respect to a new origin point – the perspective from which the user is viewing them.

To summarize, transforming coordinates from one system to another requires us to have a transformation matrix. From a mathematics perspective, this is just moving a point around in a single coordinate system by applying one transformation after the other. However, abstracting this concept and labeling the different coordinate systems by the nature of the transformations that create them makes implementing the game design features much simpler and modular. [4]

V. Virtual Cameras

Graphical (3D) projection is a method to view a 3D object on a 2D surface. This definition fits perfectly with what we are trying to achieve – we have 3D models such as a cube and we wish to view them on a 2D surface, which would be our display screen [5, p. 141]. In CG, the two main types of projections are *orthographic* and *perspective*.

The *camera* in a sense, is the point from which we wish to view the objects from. This could be a point on our coordinate system. In fact, in *srqEngine* the camera is modeled as a C++ object which has a location in the scene (*x*, *y*, *z* coordinates).

A. Orthographic Projection

This is a type of parallel projection, where we are viewing 3D objects. The camera object in this case would just be a viewport, rather than a single point. It could be a rectangular plane from which we are viewing the scene from. Objects with coordinates that lie outside the width or height of this plane get clipped from rendering. We can also define a maximum depth (*z*-value) which would act as another clipping plane, so that objects that are further than this value get clipped too.

Despite the fact that we are working with 3D objects, this type of projection doesn't give us a perception of depth. In reality, objects that are further away appear smaller than they actually are. However, with this type of projection we are only dealing with depth as a numerical value rather than actually decreasing the size of the objects, as their *z*-value strays further from the viewport (see *Fig 2.1*). It is also to be noted that even though there is no perception of depth in orthographic projection, objects that overlap or are behind one another still get partially or fully hidden depending on their size and position, as explained by Shirley in [5, pp. 144-145].

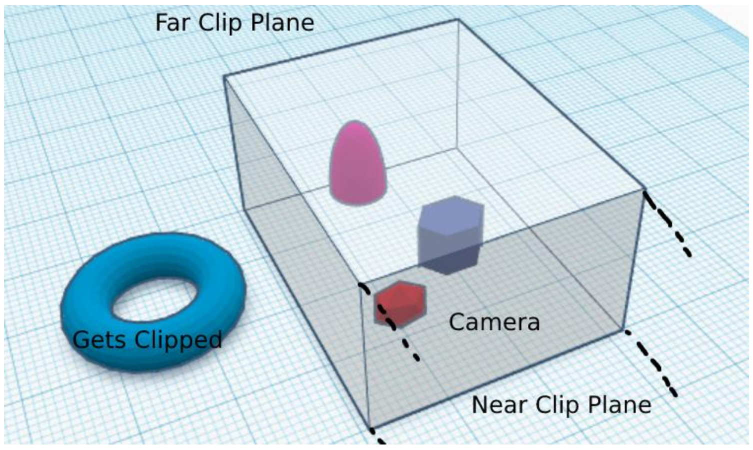


Figure 2.1 Orthographic Projection

B. Perspective Projection

This is a type of linear projection where we model the perception of depth of 3D objects. Like in reality, objects that are further away (*z*-value) appear smaller than they actually are. Unlike orthographic projection where we can just define a rectangular plane to be our window into which we see the scene, in perspective projection we define a point of view from which we will look at the scene from, and a field of view (FOV) which is an angle that determines how wide we can see. The larger the FOV is, the more of the scene is visible. Like in orthographic projection, we can define a *far plane* which sets the maximum distance at which objects don't get clipped.

To actually make objects appear smaller than they are depending on the distance, we use a matrix transformation on the objects. This is called the *perspective projection matrix*. In *srqEngine*, this is coupled together within the *view projection matrix* mentioned earlier.

Because we are using a FOV which starts from a single point and extends out, the *far plane* is guaranteed to be larger than the *near plane*, meaning that the shape formed is a frustum (as shown in *Fig 2.2*).

For the *perspective projection matrix*, we need to use the homogeneous coordinate which we couple together with the vertex coordinates. The homogeneous coordinate, often referred to as the *w*-value is affected by this transformation. The further away the object is from the *near plane*, the higher this value becomes. At the end, the (*x*, *y*, *z*) coordinates are divided by this *w-value* which are then converted to NDCs and displayed as seen in [5, pp. 148-151].

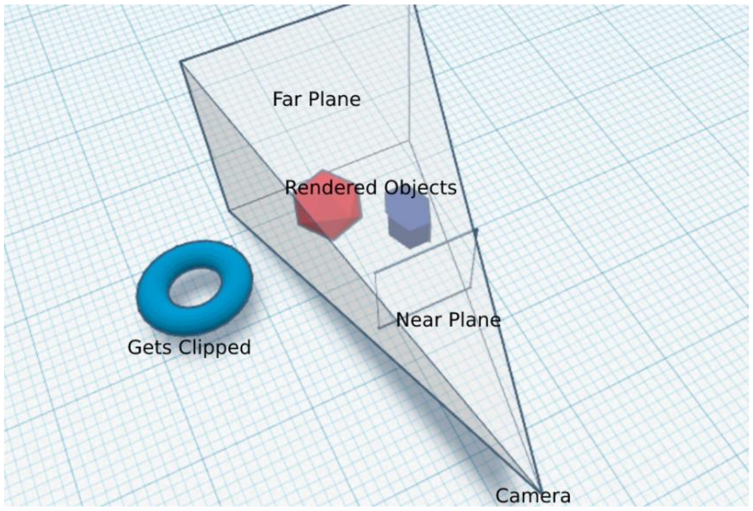


Figure 2.2   Perspective Projection

The derivation of the OpenGL-compatible *perspective projection matrix* involves using components of the projection frustum such as the near/far planes, aspect ratio and FOV as described by Ahn [6]. The OpenGL documentation provides a predefined matrix to apply to vertices in order to achieve perspective [7] as shown below.

$$M = \begin{bmatrix} S & 0 & 0 & 0 \\ 0 & S & 0 & 0 \\ 0 & 0 & -\frac{f}{(f-n)} & -1 \\ 0 & 0 & -\frac{f*n}{(f-n)} & 0 \end{bmatrix}$$

Where *M* is the *perspective projection matrix*, *f* is the far-plane location, *n* is the near-plane location and

$$S = \frac{1}{\tan\left(\frac{FOV}{2} * \frac{\pi}{180}\right)}$$

VI. Color and Light

On modern displays, all perceivable colors are mapped to a combination of red, blue and green (RGB) values, as there are an infinite number of color combinations in reality due to the nature of light, which we wouldn’t be able to store due to memory limitations. With RGB, we can represent any color that we could optically perceive in reality. A neat way to map RGB components is from

0.0 to 1.0 in floating decimal points. For example, for the red component, a value of 1.0 would be the absolute presence of red whereas 0.0 would be pure black (the complete lack of that color).

Light from a light emitting source (such as the sun) travels and collides with an object. Depending on the object's material, it absorbs a portion of the light and reflects some as well. The reflected light from the object is what we perceive the object's color to be when it enters the retina as surmised by Clark [8], so color is defined by the portion of the light source that gets reflected by an object rather than the object having a color by itself. For example, the material of wood is not actually brown, sunlight (which is a combination of all colors that form a near-perfect white light) hits the wood, which absorbs some of the sunlight and reflects some as well. The reflected light hits our eyes and that's what we perceive to be a brown color (see *Fig 3.1*).

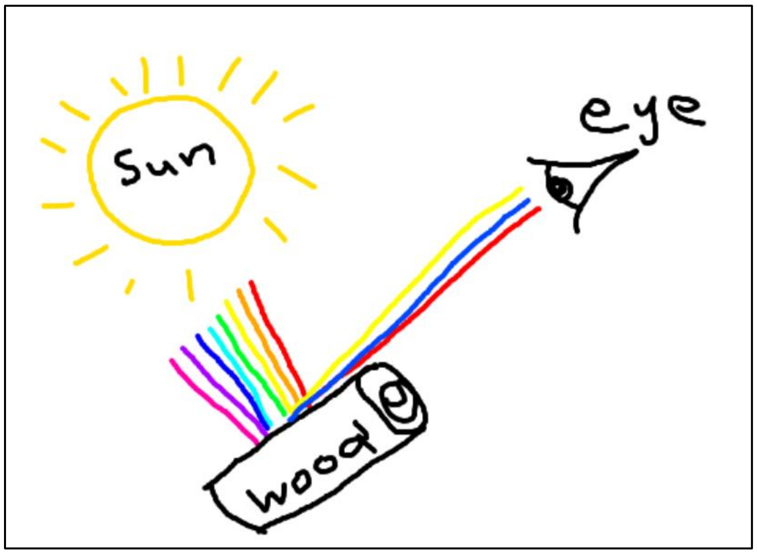


Figure 3.1 Light absorption and reflection from a light source, off of an object

The shading model that *srqEngine* uses is called *Phong shading*. This is a combination of three different kinds of lighting components which aid in realism when combined.

A. Ambient Light

Ambient light in *Phong shading* simulates leftover light photons from a light source which is no longer present. It could also be reflected light from an extremely weak light source. The idea is that even without a concrete light source, we can still see objects and their colors even though they are overwhelmingly dim. The best example I can give is looking at objects at night; we can still make out their colors and shapes because they are dimly lit (i.e. from moonlight) instead of being in complete darkness.

B. Diffuse Light

This is the simulation of the directional reflection of an object from a light emitting source, and its intensity. It takes into consideration the angle at which the light source is to the object; the more orthogonal it is to the object face, the higher the intensity. To measure the angle between the light source and the object, we use the normal, which is the vector that is perpendicular to the object's face (see *Fig 3.2*). To get the angle between the directional light and the normal, we can simply apply the dot product (see ***Appendix II***).

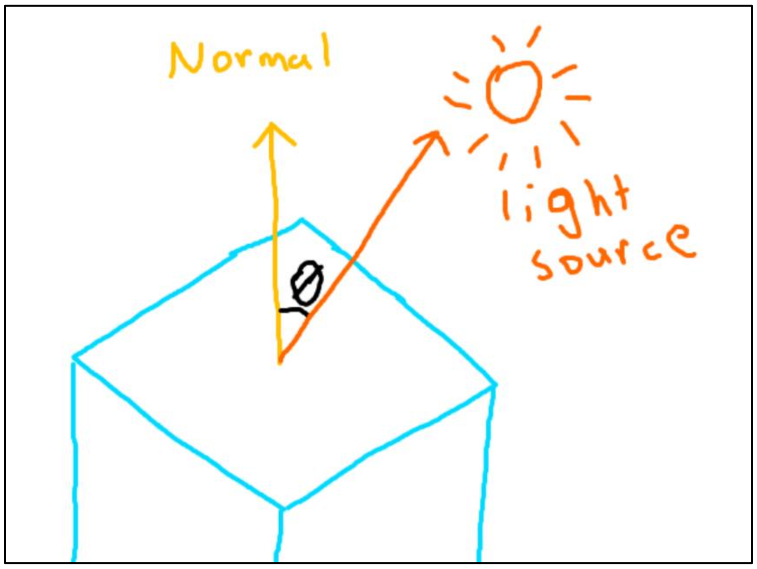


Figure 3.2 Diffuse component construction for Phong light in srqEngine

To calculate the normal of a face, we can use the cross-product (see ***Appendix II***) on the face's vertices to get a vector orthogonal to them. The light emitting object can be represented as a single point in the coordinate system, and since we already calculated the normal of the face, we can use the normal vector's origin to construct a directional vector for the light source - from the origin to the light emitting object position. Now we can proceed by using the dot product between these two vectors to get the angle and hence, determine the location of the face which should be lit.

C. Specular Light

Specular highlights are used to simulate real materials that have a degree of shininess. We are placing a bright spot on the object depending on the angle at which we are looking at it, to give the illusion that the object is shiny. A matte material would barely have any specular highlights whereas a metallic one could have a very bright one. To calculate the intensity of the highlight, the reflected light's direction vector (from the diffuse step) is used alongside another vector from the surface normal's origin to the camera position, which is the point of view (see *Fig 3.3*). The angle between these two vectors determines the specular highlight brightness. Shirely talks about adding a specular component to another shading model (*Lambertian Shading*), which is relevant here [5, pp. 82-83].

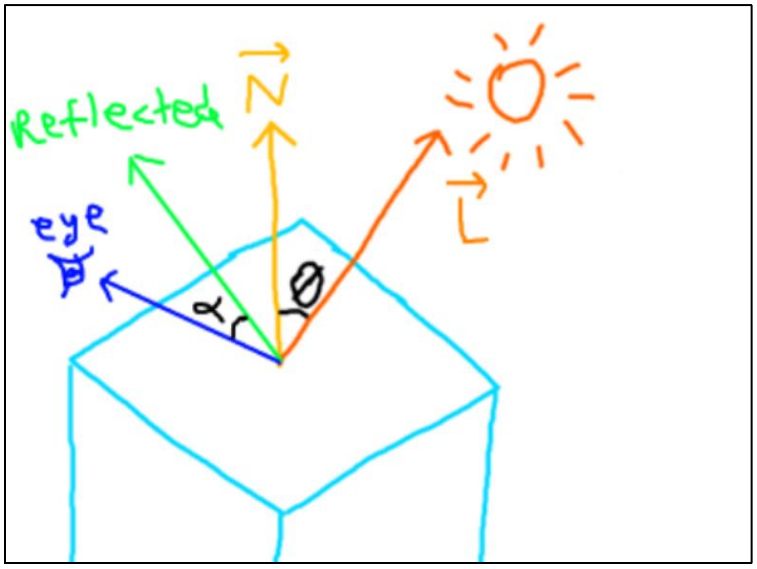


Figure 3.3 Specular highlight component construction for Phong light in srqEngine

VII. Collisions

srqEngine uses an axis-aligned bounding boxes (AABB) collision detection algorithm to resolve collisions between objects when they are intersecting with each other. AABB collision involves drawing an invisible bounding cube (in 2D you would use a rectangle) around an object you wish to make a collidable and using that as a mask to detect intersections between other collidable objects. A collidable object can be mobile or completely static. If a mobile object's bounding box intersects with a static one, the mobile object is pushed in the opposite direction of the face that collided with the static object. We can get the opposite direction using the normal vector of the bounding box.

The engine uses a variation of this collision detection called Point vs. AABB – this is where we are checking if a single coordinate is intersecting with a bounding box in any axis, and returning the axis with the *most* intersection; pushing the point out in the opposite direction. This is used in the implementation in form of a camera colliding with world objects (models), because as discussed in depth later, the camera is represented as a single point which is able to move depending on the user input and the objects are static.

An advantage of this type of collision detection is that it is quite memory efficient and doesn't take up a lot of processing power because we are representing the bounding boxes as simple cubes and checking for intersections with cubes is computationally inexpensive, as [9, p. 77] notes "it involves direct comparison of individual coordinate values". A disadvantage however, is that a model could be a complex shape with many vertices and a cube might not fix it correctly, so it is not a pixel perfect collision which would give far more realistic results, but using a complex bounding shape would also come with the cost of being computationally expensive as we are checking a huge amount of vertices for intersection with just two objects.

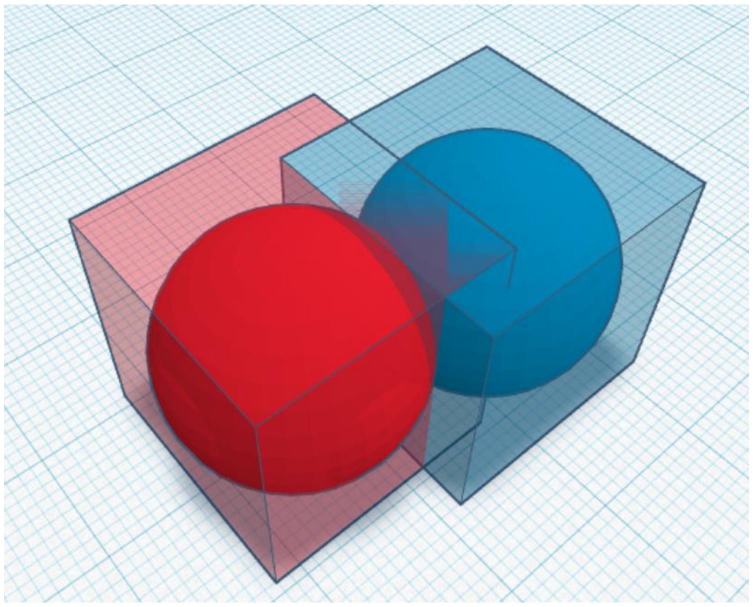


Figure 4.1 AABB Collision between two round objects

In *Fig 4.1*, two round objects (the models) have bounding boxes attached to them, which are updated along with their positions. The collision is detected when the AABBs intersect (not when the spheres themselves intersect).

The bounding boxes for the objects in the engine are initialized in local space, as [9, p. 81] argues that "One benefit of transforming into local space is that it results in having to perform half the work of transformation into world space. It also often results in a tighter bounding volume than does transformation into world space". I agree with their deduction, and it applies in the engine's scenario because the *model matrix* to transform the model from *local space* to *world space* would need to be applied once more for its bounding box if it wasn't already defined together with the model.

VIII. Real-time Rendering

Rendering CG in real-time refers to the change of the geometry over a defined period of time, which is affected by the refresh rate of the viewing device. In this context, *still* geometry (not subjected to change yet) is called a *frame*. The rate of the frames changing is usually measured in seconds, and commonly referred to as *frames per second* (FPS) [1].

In order to implement this concept, we introduce a *game loop* which produces a desired amount of FPS, in which we can manipulate the geometry through transformations. This also introduces interactivity with the graphics, as we can transform them by providing live input, which leads us to implementing game design elements from the DOOM game. We can also update model positions, perform collision checks and resolve collisions here.

Because we are concerned with interactivity, we wish to maintain a high FPS rate to increase immersion. This comes with a high computational performance cost as we are performing many multiplication operations (through matrix transformations) every second. With highly detailed (and numerous) geometry, complex collision detection (and other physics), particle effects, lighting and so on, it quickly becomes unfeasible to calculate on the CPU. Advancements in graphics hardware (GPUs) allows us to send over a huge amount of data and process it much quicker, giving us the ability to achieve a high frame rate even with hyper-realistic models made from a vast amount of geometric shapes [1, p. 1].

TECHNICAL DOCUMENTATION

I. Overview

The objective of this project was to create a graphics rendering engine with DOOM game design elements, which I felt was achieved, as the final implementation contains (1) a rendering API which abstracts away low-level details in order to render graphics in a concise and efficient manner, (2) a model importer which allows high-resolution models to be accessed and drawn with a direct export from a 3D modeling software, (3) a first-person camera which is analogous (and improved) to the one in DOOM, (4) player controller motion in each axis with a smooth jumping mechanism, (5) collision detection and resolution between the player and the models, (6) a virtual world simulation enclosed within a skybox, (7) particle effects to simulate atmosphere, (8) a realistic lighting model with an adjustable light source, (9) ray projection from the player into the world and, (10) an entity-component system for developers to make the renderer API more accessible, and their code more organized.

The technical documentation explores these features in detail and provides hyperlinks to the relevant sections of the GitLab repository's ***README*** file, pertaining to the discussed feature in the footnotes. The README file contains code usage examples, or extra information about the implementation details.

Alongside this, the footnotes also contain hyperlinks to the *source code* files where the feature is implemented. Any figure may be accompanied with a hyperlink to a *video demo* for optional viewing, which will also be put in the footnotes of that feature.

II. Libraries

To accommodate the OpenGL API, I have used the following libraries.[1]

A. GLFW

This is an abstraction layer for the Win32 API for the Windows operating system to provide context creation (a GUI to display the graphics that were created using the engine), along with keyboard/mouse input handling as it maps button clicks, mouse movements or key presses (and releases) to callbacks which you can define, or poll for device input in general. Since this project

[1] README - Libraries Used • Source Code – Entry.cpp (and throughout the API)

is about graphics rendering and not application programming, I felt like this would be an appropriate supplementation.

B. GLAD

The OpenGL specification only describes an abstract API for drawing the graphics, but a loading library such as GLAD is needed to map the thousands of function calls to actual function names that can be used in programming. OpenGL is almost always paired with a loading library such as GLEW or GLAD.

C. GLM

OpenGL header-only Mathematics library. This is a widely used library to perform computationally efficient operations on matrices and vectors, and provides output in neatly organized data structures, whereas working with native arrays for matrices, their operations and organizing the amount of data needed could become quickly unmanageable.

D. STB Image Loader

This is a simple header-only library used to import images into C++ (for example in PNG format). I only use this to load in some textures that map onto certain models.

III. Camera Implementation

The camera in *srqEngine* is implemented as a first-person camera – meaning that the way objects are perceived in the *world space* is from the player's eyes. To achieve this effect, I used the *perspective projection* transformation described earlier and applied it to all objects in the scene, so that the illusion of perspective is being created. I provide a camera class in the engine to output the *view-projection matrix*, which will be used to linearly transform all the existing model coordinates into the desired perspective view.[2]

First, the camera is defined as a 3D coordinate to represent its location in the scene. Next, we wish to give it a target to look at, which would be the direction that the camera is pointing in. This can be represented as a vector. We can find the other two axes that are orthogonal to this direction-vector (the right and up vectors) using the *Gram-Schmidt* process (see ***Appendix III***). We wish to create a coordinate space for the camera because the objects in our scene will be in respect to its origin (the position of the camera). The resultant vectors are shown in *Fig. 5.1*, which depicts the constructed camera pointing at a model (the red sphere).

[2] README – Camera • Source Code – Camera.h , Camera.cpp • Video Demo – Camera Demo

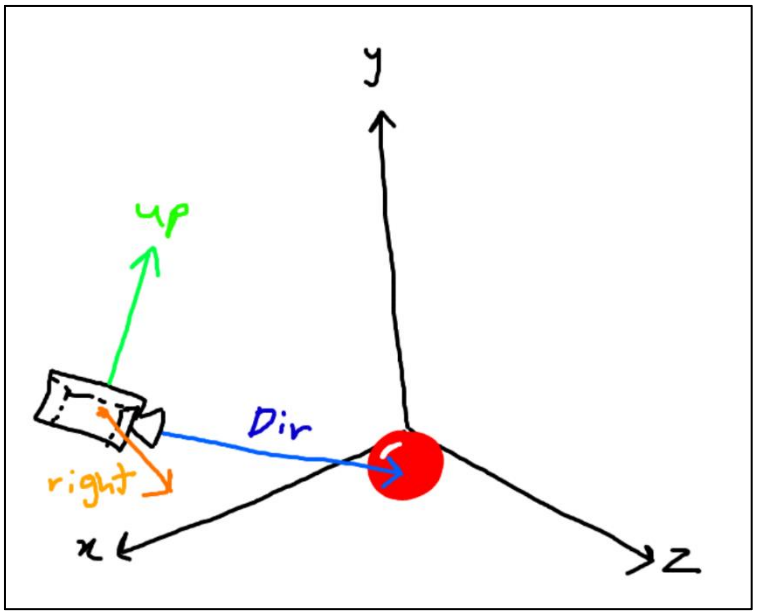


Figure 5.1 Constructed camera looking at a model

Using the camera vectors that we found through the *Gram-Schmidt* process, we can construct a matrix that can map any vector to the camera's coordinate space. The matrix is defined as follows

$$LookAt = \begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & P_x \\ 0 & 1 & 0 & P_y \\ 0 & 0 & 1 & P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Where *R*, *U*, *D* are the camera's right, up and direction vectors respectively and *P* is the camera's position coordinates. When we multiply the *LookAt* matrix with any model's vertices, we are essentially transforming them to the *camera space*. GLM provides a handy function to construct this matrix by inputting the vector variables. This is also the *view* part in the *view-projection matrix*. In order to calculate the perspective now, we have to define a FOV and the near/far planes (essentially the matrix derived by [6] as mentioned earlier). When we multiply the view and perspective matrices, we get the *view-projection matrix*. This is what the camera class outputs.

Since this engine is based on the DOOM game, I wanted to introduce some movement of the camera to emulate the player moving around in the scene. This is done in three different ways, first we have the movement of the camera itself, which would be done through changing the camera position's *z/x* coordinates to simulate the forward/backward and left/right motion. Next, we have a smooth jumping mechanism which uses gravity and jumping velocity variables, and applies that to the camera position's *y* coordinate and finally there is a *looking around* function which allows the user to use the mouse to translate the camera's yaw and pitch so that we can simulate a head-turning motion.

The calculations for motion are done in the game loop of the application, in confines of a frame rate limit. The goal is to update the camera's position and rotation depending on the input. Using GLFW, we can poll for key presses such as the arrow keys or WASD. When a key press is detected, the camera position is updated accordingly. In *srqEngine*, W is for moving the camera position forwards, towards the look-direction, A/D is for strafing left and right of the look-direction and S is negating the forward motion to move in the opposite direction of the direction vector. Pressing the Spacebar results in the jump function being executed. The mouse's *x* and *y* positions on the screen also gets polled. Moving the mouse left and right results in the yaw of the camera being adjusted, you can look around the full 360 degrees in the yaw-plane. To make it more realistic, the pitch-plane is capped at 90 degrees in both up and down directions.

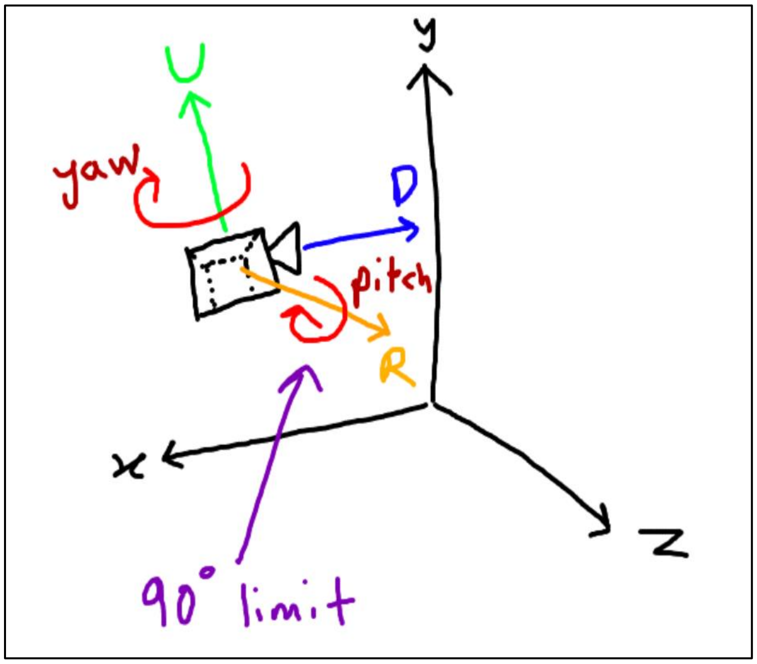


Figure 5.2 Camera rotation around its vectors in the yaw and pitch planes. There is a 90 degrees limit on the pitch plane (which turns the camera up and down)

Note that there is no rotation in the roll-plane, which would be a rotation around the direction-vector. This would result in a head-tilting effect rather than the head-turning one which we have. I also avoided implementing it to prevent running into the *Gimbal lock* problem where we lock two axes together and lose dimensionality as pointed out by [10, pp. 237-242].

Since the camera position and rotation gets updated through input polling, we have to recalculate the *view-projection matrix* on each update for smooth movement. This matrix gets passed into the graphics rendering pipeline alongside our model vertices and applied to them so that they are visible from the perspective of our camera.

As for jumping, a *y* coordinate is defined as the *ground point,* which is where the camera has to be positioned in order for the jump to be activated – this is to prevent jumping again while mid-air. Gravity and jump-velocity variables are defined and can be tinkered with to adjust the power of the jumps. The jumping mechanism uses the following logic, which results in a smooth jump movement over time.

$$\begin{aligned}
&\Delta\, speed \ = \ -gravity * \Delta\, time \\
&velocity = \ velocity + \ \Delta\, speed \\
&y = \ y + (velocity * \Delta\, time)
\end{aligned}$$

Where Δ *time* is the elapsed time between the current frame on display and the previous frame; *y* is the camera's *y* coordinate. *Fig* 5.3 shows the result of constructing a camera and viewing a model with it in *srqEngine*.

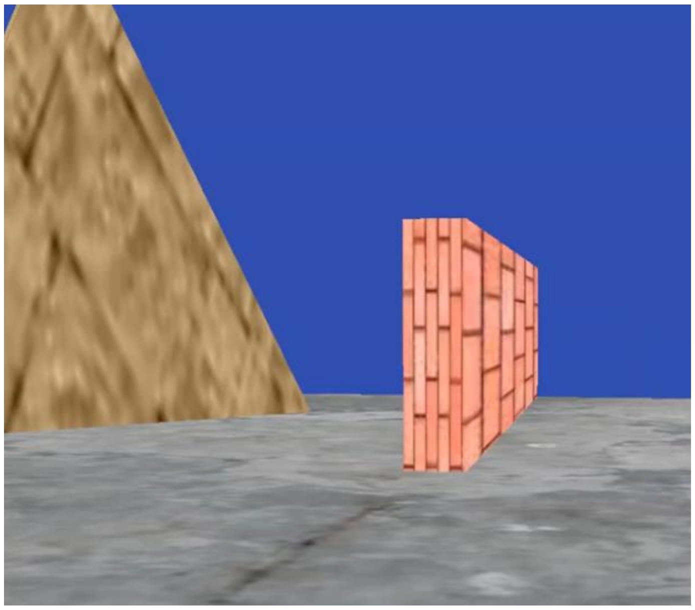


Figure 5.3 Implemented first-person camera in srqEngine

IV. Renderer API

The renderer API in *srqEngine* is a set of classes that abstracts OpenGL functions to make it much more modular and provide an overall *renderable* concept in a single object. These classes instantiate said objects and can be used in another class called the renderer, which is responsible for calling OpenGL's draw function. The API makes up the bulk of the project and is used to create models, textures, skyboxes, particles and rays. Each of these major components is comprised of several sub-components that belong to the renderer API as well. The overall architecture of the engine is best explained with a diagram first (see *Fig 6.1*).

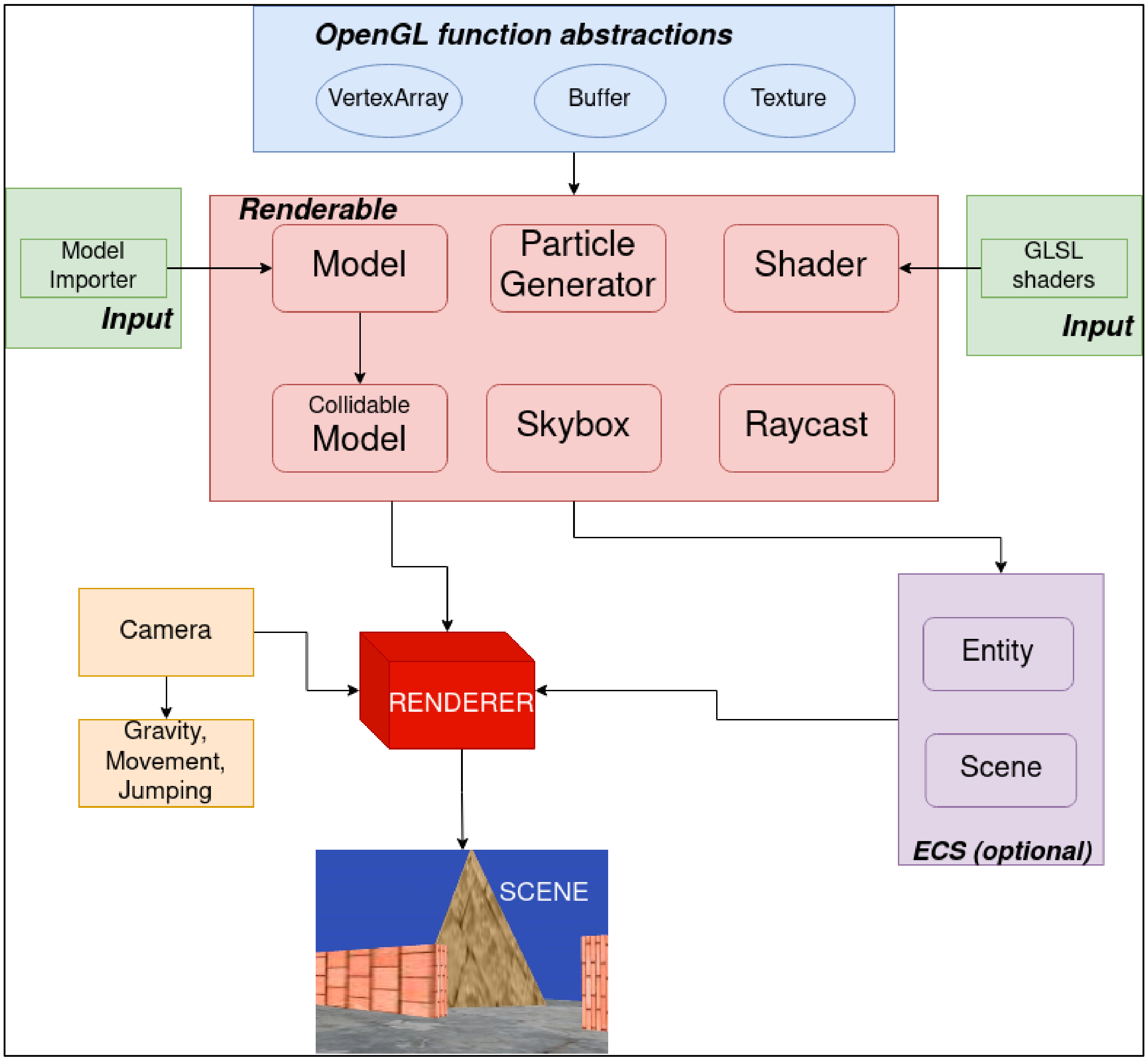


Figure 6.1 srqEngine's Architecture

A. Vertex Arrays and Buffers

In order to send vertex data to the graphics rendering pipeline to be processed and rasterized, we need to place them in OpenGL's memory management objects called the *Vertex Buffer Object* (VBO) and *Vertex Array Object* (VAO). They allow us to store a huge amount of vertex data in the GPU memory. The VBO is used to retrieve and store vertex data from a typical primitive data structure such as an array, using several arguments like its size in bytes, the data types and providing predefined flags that tells it how we wish to use the data.

The purpose of VAOs is to bind VBOs with a certain specification for the data. If we are just passing an array of floating-point numbers to the VBO, we need to define its layout to give it any meaning. Another OpenGL object called *vertex attribute pointer* allows us to do that. We give it information such as what a vertex is – i.e. the number of coordinates (3 in our case for each axis), the stride (separation amount of each vertex within the array) in bytes and a few more attributes.

To make it clearer, we could want an array of 9 floats to be interpreted as 3 vertices with 3 coordinates each. The *vertex attribute pointer* would be the method used to provide this information as shown in *Fig 6.2*.

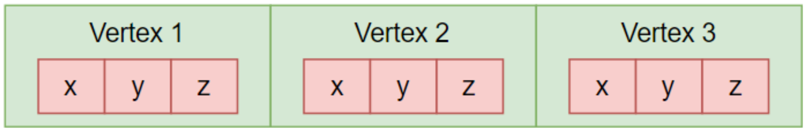


Figure 6.2 Separation of data from the VBO using the vertex attribute pointer

It is possible that we could have multiple different attributes. For example, some of the data could be for positions and others could be for texture coordinates. In order to save these different configurations and layouts, we need a VAO to bind them to. The latest bound VAO (in combination with the latest bound shader discussed later) is what is used to make a functional draw call in the renderer. Unfortunately, to do all this requires a vast amount of OpenGL calls to functions and is not maintainable with a project of this scope, so the renderer API provides a compact method to automate this process by only giving vertex data (which doesn't need to be manually inputted as we can extract it from a 3D model – as shown later), and figuring out the layouts/stride calculations by itself.[3]

[3] README – Renderer API – VAO/VBO • Source Code – Buffer.h , Buffer.cpp , VertexArray.h, VertexArray.cpp

B. Shaders

A shader is a program that gets compiled, linked and runs on the GPU, which is passed to it through the central processing unit (CPU), just like any other data that we send to it (i.e. VAOs). OpenGL uses GLSL (Graphics Library Shading Language) which is a language with C-like syntax, for its shader programs. The shader instructs the GPU on how to draw the vertices passed to it through the VAO. The two most common programmable properties, and the ones that are used in *srqEngine*, are the position and color of the models that the vertices make up. We write shaders to evaluate these properties and apply them on the GPU. To configure the position and color/shade properties, we use shaders called the *vertex shader* and *fragment shader*, respectively. The shaders get executed along the graphics rendering pipeline, with the vertex shader preceding the fragment shader. Shaders can also pass along data that they have received to the next shader in the pipeline.

The *vertex shader* gets called for each vertex that we are trying to render from the buffer objects. The primary purpose of this shader is to tell the GPU where on our screens we want the vertex to be positioned when it is rendered. This gives us room to apply all sorts of linear transformations to the vertex to determine its position. Earlier in the VBO/VAO section I briefly glossed over the *vertex attribute pointer* which defines the layout of our data. An argument of this function is an index that uniquely identifies the layout object (as shown in the README file). The shader uses this index to associate itself with the object and access the position from this layout, which it can then perform transformations on. This is where the *model* and *view-projection* matrices are applied as discussed earlier, so that the vertices are positioned in accordance to our view.

The *fragment shader* gets called once for each pixel that needs to get rasterized. The shape defined by the vertices need to get filled in with color for rasterization to happen, and the fragment shader is responsible for determining this color. We can do all sorts of color manipulation such as painting gradients, applying different opacities and even using data passed to it from the *vertex shader* to determine where on the model to draw certain highlights. This is where the *Phong shading model* is created as discussed earlier, by using the vertex data in the *fragment shader*.

In order to draw geometry to the screen, we need to have a VAO, a *vertex shader* and a *fragment shader* bound before the draw call. Using the vertex layout information from the VAO, transforming the positions with the vertex shader and coloring in the geometry with the fragment shader, a model can be drawn to the screen.

A *uniform* is a global shader variable that can be set through the application layer. In *srqEngine*, the camera's calculations are done separately in another class, and applied to each vertex in the *vertex shader*. In order for the vertex shader to access the transformations (such as the *view-projection matrix*) outputted by the camera class, and apply them to the vertices, a *uniform* matrix variable can be declared within the shader and set in the application layer.

To do this in OpenGL requires a lot of work and manual tinkering (like with native VBO/VAOs). First we have to define a string for each shader that contains the GLSL code, initialize each shader, compile them, store them in an all-encompassing shader program object, link them, remove individual shader programs from memory, set any uniforms we wish to and then bind every shader that we need before drawing something. This process takes numerous OpenGL function calls.

Therefore, I created a shader class in the renderer API that does all that work. The only things that you need to provide it are the paths to the shader scripts that contains the GLSL code. I also made a bunch of handy methods to make setting uniforms a piece of cake, as common data types have been hard-coded (such as a 4x4 matrix *uniform* or a 3D vector). *Fig 6.*3 makes this portion of the architecture clearer.[4]

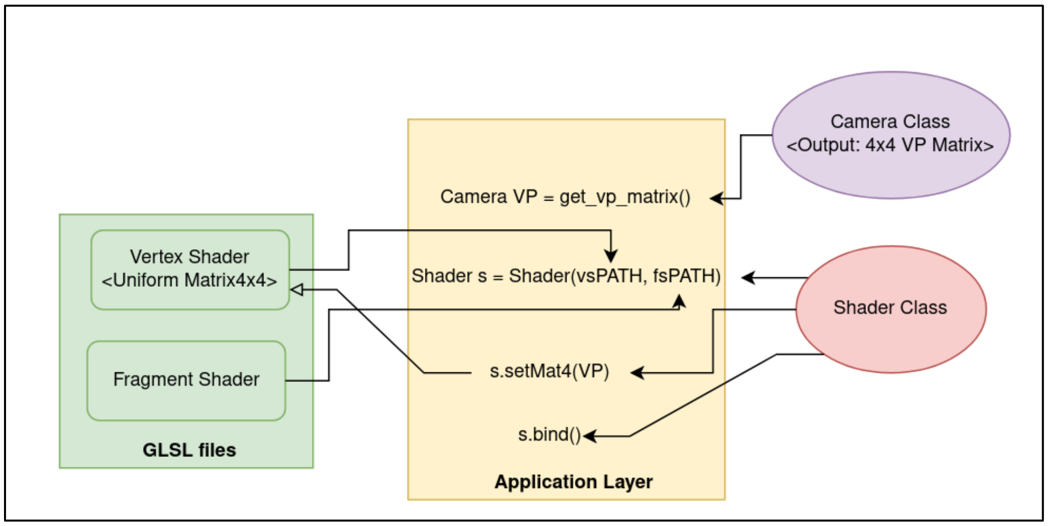


Figure 6.3 Separation of data from the VBO using the vertex attribute pointer

C. Textures

Textures in CG are images that can be plastered onto the geometry that we render [5, p. 243].

The images are in 2D and often need to be wrapped onto 3D objects, so OpenGL provides several flags to set the texture wrapping methods, clamping options, interpolation methods and more. To know what pixel on the texture to place on the geometry, the texture coordinates are contained within the model data, which is identified by the vertex attribute pointer.

[4] README – Renderer API – Shaders • Source Code – Shader.h , Shader.cpp

My renderer API also has a class for setting up textures automatically so we can easily pass a texture object around without having to set up all the manual flags ourselves. To load in the images into C++, I used the *STB Image Loader* library as mentioned previously. The textures have to be bound, and image-buffer memory has to be freed after using it as well, which is all handled by the texture class too. The *fragment shader* has to know about the texture so that it can apply it to a model before rasterization, so we have to provide a uniform with a GLSL primitive called the *sampler2D*, which uses the currently bound and active OpenGL texture.[5]

D. Models

The engine supports a diverse model class which abstracts all the previous concepts into one, to pass into the renderer for drawing. The idea is that we have a bunch of vertices that make up a 3D model which is handled by the model class, including the entire VAO/VBO setup, texture binding if it is a textured model (or not if it is a plain one) and appropriate shader association. Manually inputting coordinates for positions, normals and textures would not be feasible so I built a model importer which parses a 3D model file containing these coordinates, which can be created in a 3D modeling software (such as Blender) and exported. It then filters the parsed data into auxiliary data structures from which they can be passed into our VAO for automated layout configuration.

The model importer parses a Wavefront object file which is a simple-data format that represents 3D geometry. It is designed to be human readable and different types of coordinates are marked with identifiers. *Fig 6.4* shows an example of a Wavefront object file exported from Blender after creating a 3D model with it.

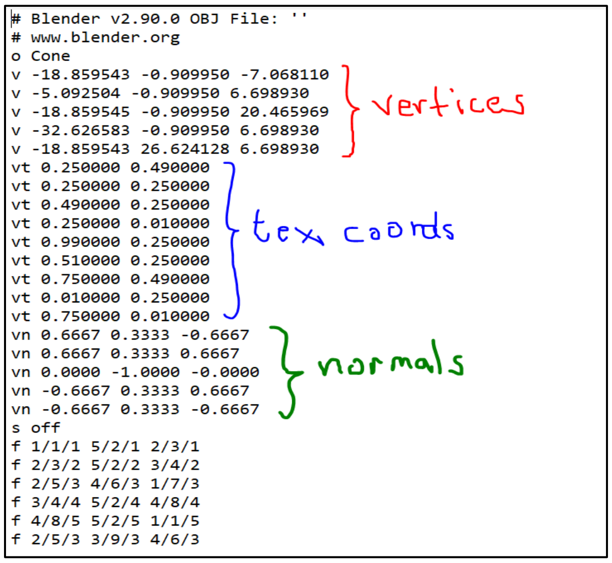

```
# Blender v2.90.0 OBJ File: ''
# www.blender.org
o Cone
v -18.859543 -0.909950 -7.068110
v -5.092504 -0.909950 6.698930
v -18.859545 -0.909950 20.465969
v -32.626583 -0.909950 6.698930
v -18.859543 26.624128 6.698930
vt 0.250000 0.490000
vt 0.250000 0.250000
vt 0.490000 0.250000
vt 0.250000 0.010000
vt 0.990000 0.250000
vt 0.510000 0.250000
vt 0.750000 0.490000
vt 0.010000 0.250000
vt 0.750000 0.010000
vn 0.6667 0.3333 -0.6667
vn 0.6667 0.3333 0.6667
vn 0.0000 -1.0000 -0.0000
vn -0.6667 0.3333 0.6667
vn -0.6667 0.3333 -0.6667
s off
f 1/1/1 5/2/1 2/3/1
f 2/3/2 5/2/2 3/4/2
f 2/5/3 4/6/3 1/7/3
f 3/4/4 5/2/4 4/8/4
f 4/8/5 5/2/5 1/1/5
f 2/5/3 3/9/3 4/6/3
```

Figure 6.4 Wavefront object file

[5] README – Renderer API – Textures • Source Code – Texture.h , Texture.cpp , Example textures in srqEngine/res/textures/ directory

We can observe that each line that contains useful data is prefixed with an identifier (*v* for vertex, *vt* for vertex texture, *vn* for vertex normal). Using this information, I parsed the lines into dynamic data structures (vector objects in C++), through comparisons and delimiters for each attribute respectively. After importing the model, we have the data to pass into the VAOs for automatic setup and we can pair it with a texture if we wish to create a textured model. We can optionally leave it plain and bind it together with the correct shader depending on our choices. We are left with an object that has all the items needed for a valid draw call. We can also set a flag for collisions upon initialization, indicating that the model is intended to be collidable. Through implementing this feature, the engine now supports high resolution/poly models. This means that the model that you wish to load can contain as many vertices as needed. The risk, however, is longer loading times as one of the limitations of the model loader is that it reads line by line, thus we have *O(n)* (linear) time complexity as the load time is directly proportional to number of vertices.[6]

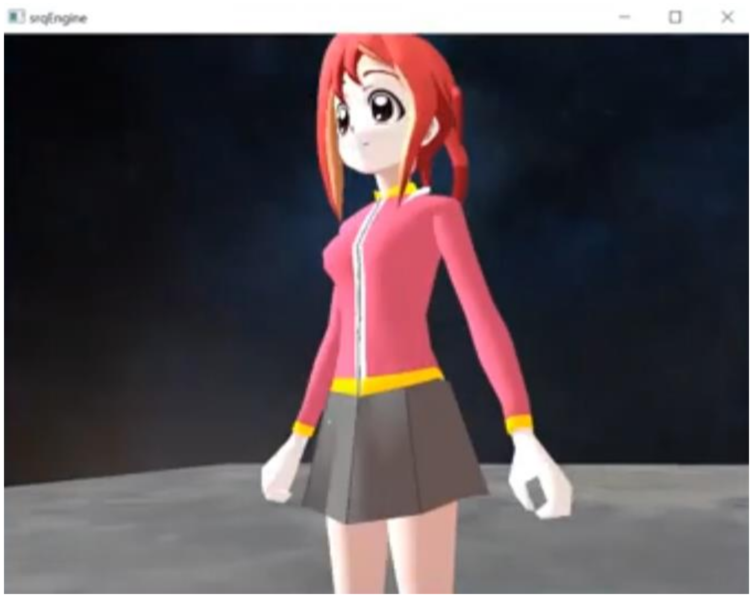


Figure 6.5 A high-detail model imported into the engine using the model importer

E. Collision Models

As mentioned earlier, models can be optionally flagged as *collidables* when initialized, which would translate to the camera not being able to pass through them. In the literature survey, I

[6] README – Renderer API – Models • Source Code – Model.h , Model.cpp • Video Demo – Model Demo

explained how *srqEngine* uses AABB vs Point collision detection. An AABB can be set on the models, which the camera (the point) is not able to pass through. Since the camera is represented as a coordinate that is moving through space, it makes sense to see it as the *player* which cannot pass through solids such as *walls* (DOOM terminology).

Once a model gets flagged as a collidable, its AABB gets generated according to the size definition provided to it on initialization. Then it gets added to a dynamic data structure with all other collidable objects, which gets queried with the camera position, to check if it is intersecting with any of them. If it does intersect, the camera gets pushed in the opposite direction of the normal vector for the face of the AABB that the model is assigned to.

This results in smooth collision detection and resolution, with no clipping. The intersection checks happen on each frame in the game loop. The camera cannot penetrate the wall as seen in *Fig 6.7* (this is clearer in the video demo)[7].

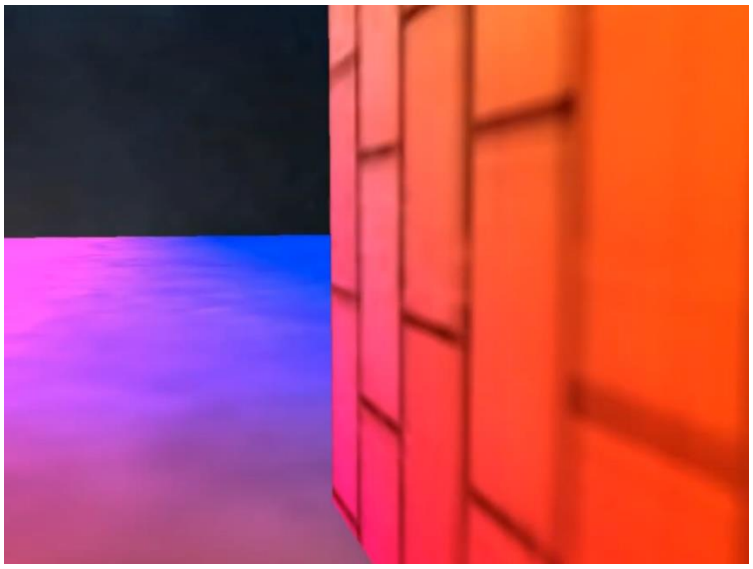


Figure 6.7 Camera collision with a wall

F. Skybox

The skybox is essentially a 3D version of a background image. Gordon [11] explains that the steps to make a skybox are to first instantiate a cube model, texture it with an environment image of our choice and then position the cube on the camera so that it surrounds it. This feature is implemented to give the perception of being enclosed within an atmosphere. Without the skybox we would have a plain color, which is not as realistic, as we are trying to emulate being present in a virtual world

[7] README – Renderer API – Collision Model • Source Code – Collision.h , Collision.cpp • Video Demo – Collision Demo

(which is an element of the DOOM game). The skybox is designed in a way that makes it seem endless (we can never go beyond it). This is of course an illusion, which happens to be quite a computationally efficient one, as I have not hard-coded it with a maximum distance, and instead exploited the *view-projection matrix* itself to achieve this effect. The way I implemented the skybox is typical in 3D games[8]. First, I created a cube model using the model class from my renderer API (just 1 unit long in each axis) and textured it using a seamless cube map as shown in *Fig 6.7*.

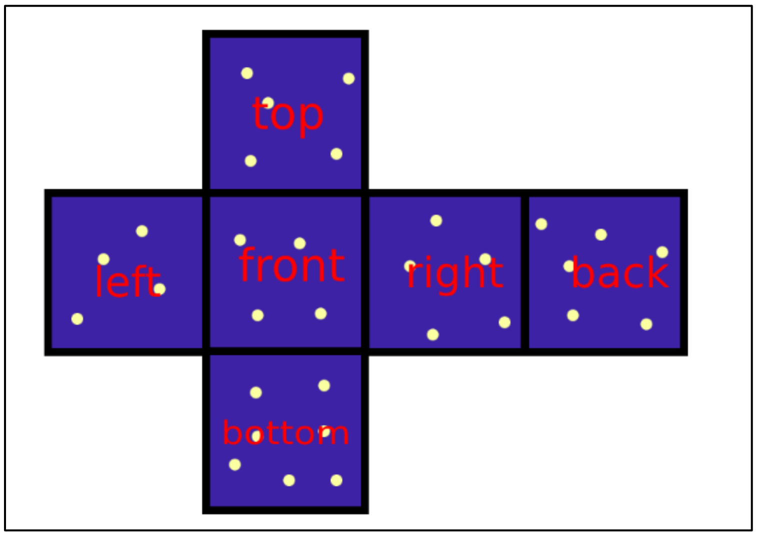


Figure 6.7 A cubemap image for a skybox

There are 6 different textures for each face of the cube; once they are plastered onto the cube model, the cube will appear seamless, as if it had a plain color, but now with a texture.

The next step is to place the cube model right at the camera position and update it so that its position stays on the camera for every frame. The trick here is that the translation area of the *view matrix* (view component of the *view-projection matrix*) is cleared as shown below, which causes the effect of seeing the cube wherever we turn (it takes up the screen). To be able to render other 3D objects in front of the skybox, I altered the depth buffer options for it, causing it to always render behind other objects.

$$Skybox\ LookAt = \begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

[8] README – Renderer API – Skybox • Source Code – Skybox.h , Skybox.cpp • Video Demo – Skybox Demo

The *view matrix* for the skybox has the position coordinates removed. In the engine, the class provided for the skybox can be bound before the draw call through a method call which would make it stay active throughout the program. When initializing it, you can pass in the file path to the directory that contains the faces' textures and it will map it automatically.

The result is an aesthetically pleasing, never-ending backdrop which makes us appear to be within an atmosphere or universe of some sort (see *Fig 6.7*). I have used a copyright-free cubemap for this demonstration.

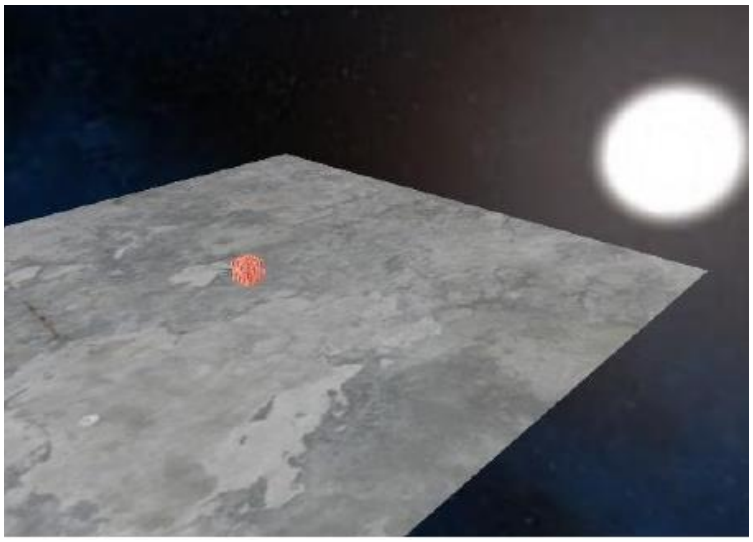


Figure 6.7 Models viewed with a surrounding skybox in srqEngine

G. Particle Generator

In the CG context, particles are tiny fragments that are intended to create some sort of special effects or atmosphere. A very concrete example would be rain droplets or snowflakes, which are the sort of particles that *srqEngine* can create through its particle generation [1, p. 586].

In the particle generator implementation, a dynamic data structure is used to add vertices that make up a single particle. In this case, I was trying to create a rain droplet particle, so I used vertices that make up a rectangular shape. Alongside the vertex data, a lifetime variable and position coordinate (within the 3D scene) is added. The position is randomized on the *z/x*-plane and given a large *y* value, as I am trying to simulate the raindrops falling towards the ground from above. The lifetime variable can be adjusted and decreased by the update delta time in the game loop – it determines how long the particle will be rendered before being deconstructed. Its position also decreases in the *y*-axis over time to give it the rainfall effect. The particle generator is updated in the game loop so that the particles move and are rendered in the renderer class. The color/lighting of the particle can be adjusted in the *fragment shader* before rendering[9].

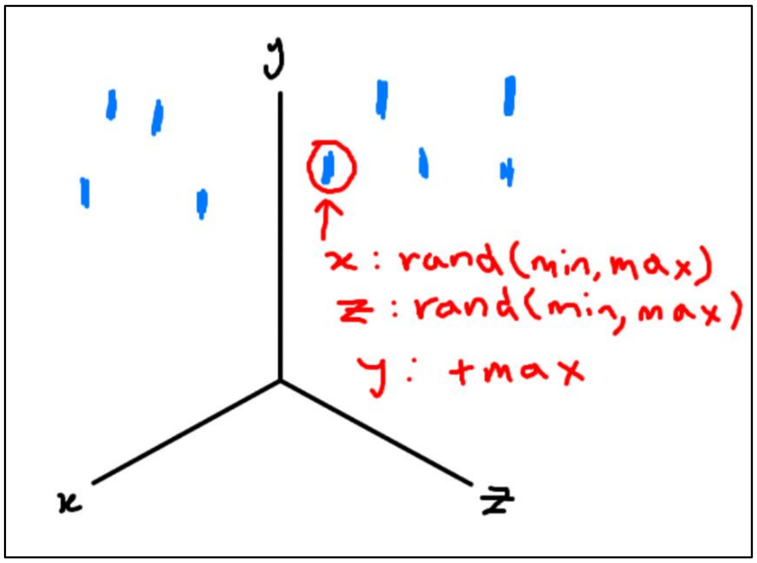


Figure 6.8 Particle spawning in a scene

Figure 6.9 Particles generated in srqEngine

[9] README – Renderer API – Particle Generator • Source Code – ParticleGen.h , ParticleGen.cpp

H. Phong Shading

Lighting in *srqEngine* is handled through shaders that are specifically applied to models that we decide are affected by light[10]. Another model can be created and designated as a light source, which acts as a point of reference for how the light-affected models are illuminated. This is done through simulating the *Phong shading model*, which consists of ambient, diffuse and specular illumination as discussed previously.

We define each of these three components individually and then add them up, applying the combined color to the object in the *fragment shader*.

For ambient light, we simply define an ambient intensity and multiply that with the model's color to simulate it being dimly lit. Since it is ambience, it is wise to use a low intensity for realistic results, *Fig 6.10* shows an object from the engine that has ambient light applied to it.

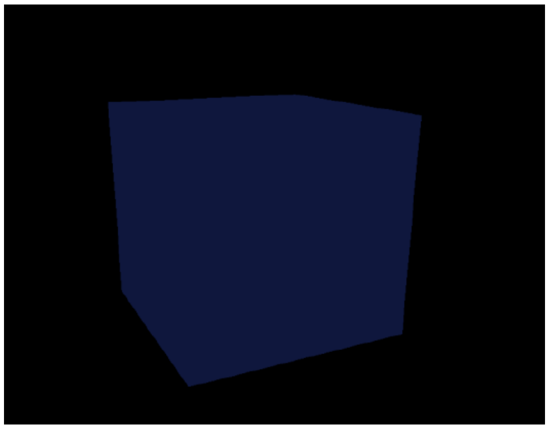


Figure 6.10 Ambient lighting on a blue cube. It appears dimly lit

In order to get diffuse lighting working, first we pass the normal vectors of the faces of the model from the *vertex shader* to the *fragment shader*. In order to fix potential scaling issues, we normalize the normal vectors of the faces. Once it's in the *fragment shader*, the light direction vector can be calculated using the difference between the designated light source position, and the current fragment's position. Like with ambient light, we can also assign a diffuse light intensity which would determine how bright the light source appears to be. It is then coupled with the ambient light and applied to the object color (see *Fig 6.11*).

[10] README – Renderer API – Lighting • Source Code – lit_vert.shader , lit_frag.shader • Video Demo – Light Demo

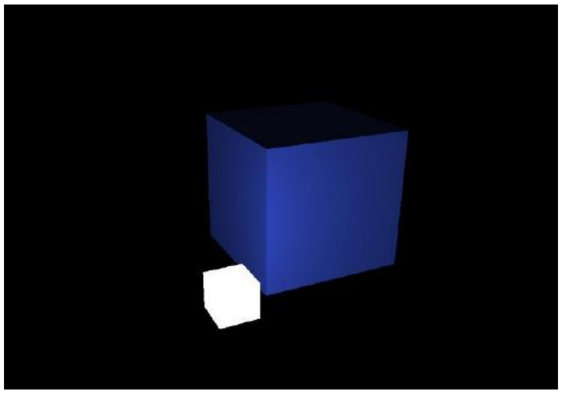


Figure 6.11 Diffuse lighting using a light source

For specular light, we can first define a specular intensity which would determine the overall shininess of the material, which would be visible from the bright spot. The highlight moves on the object depending on the camera position, so we need to calculate the view direction vector by using the difference between the camera and fragment positions. In order to get the reflection vector from the surface, GLSL provides a *reflect* function that we can use by inputting the light source direction vector and the normal at the surface. Combining all three components described above to the static model color, gives us lighting that looks like *Fig 6.12*.

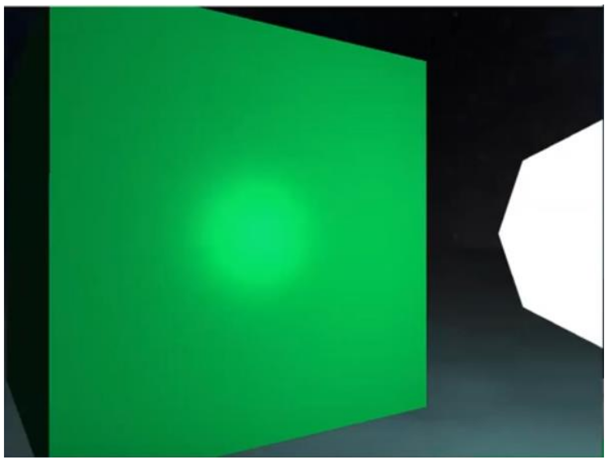


Figure 6.12 Complete Phong shading model in srqEngine. Notice the bright spot on the cube – that is the specular highlight

I. Raycasting

Raycasting is drawing a line from the near-plane of the camera to the far-plane, in the view-direction. This has potential utilities such as checking for intersection of the ray with any objects that you may wish to manipulate individually [1, p. 437].

Like the *Phong Shading* model, the raycast is implemented through shaders and the renderer[11]. To draw the line, we need the start and end positions; the function to draw the line contains an array of points which can hold the camera's updated coordinates as the start point. The end point can be defined as a point along the view-direction vector, where the magnitude is the far-plane's value as shown in *Fig 6.13*.

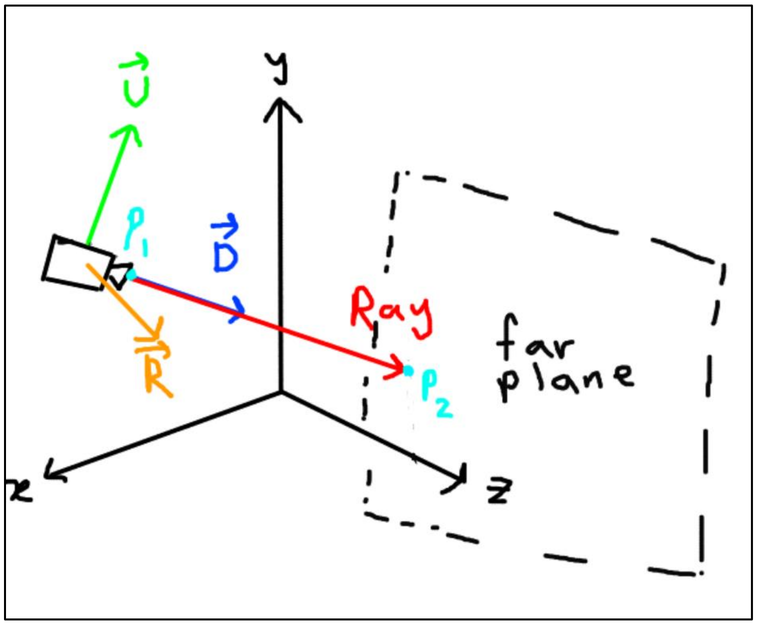


Figure 6.13 Raycast creation

In the implementation, static rays can be drawn by using the *I* key as input; the drawn ray can be seen in the *Fig. 6.14*.

[11] README – Renderer API – Raycast • Source Code – line_vert.shader , line_frag.shader • Video Demo – Raycast_Demo

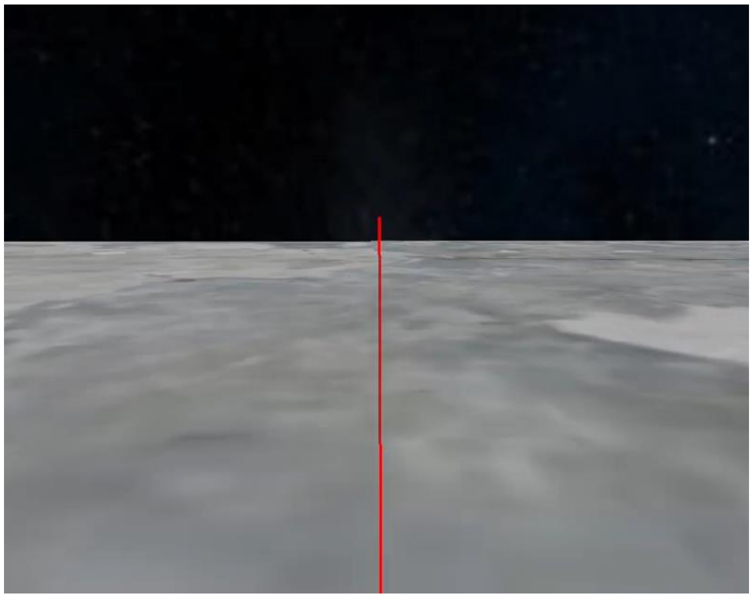


Figure 6.14 Casted ray in srqEngine

V. Renderer

In order to send VBO/VAO objects to the graphics rendering pipeline, we need a draw call. In OpenGL, this is simply a function with certain flags and modes that determines how the graphic is rendered. However, as mentioned consistently before, we need to bind a bunch of things for the draw call to even work; this is where the renderer simplifies it by providing a set of methods for drawing the different *renderables*[12]. For example, a textured model would have a different method to draw it than a plain model (as an extra texture parameter is required).

The renderer also takes the updated *view-projection matrix* from the camera class, and a custom *model matrix* as a parameter (for local editing) and sets these as uniforms for the *renderables* so that they can be viewed in the desired perspective, after binding a provided shader. Some *renderables* require specific OpenGL calls to setup them up properly, but a common theme among the methods is to bind VAO/VBOs, bind shaders, set the uniforms and issue the draw call (see *Fig 7.1*).

The *model matrix* for any given *renderable* is defined after initialization; by using GLM's transformation functions we can construct the matrices required to perform translations, rotations, shears etc.

[12] README – Renderer • Source Code – Renderer.h , Renderer.cpp

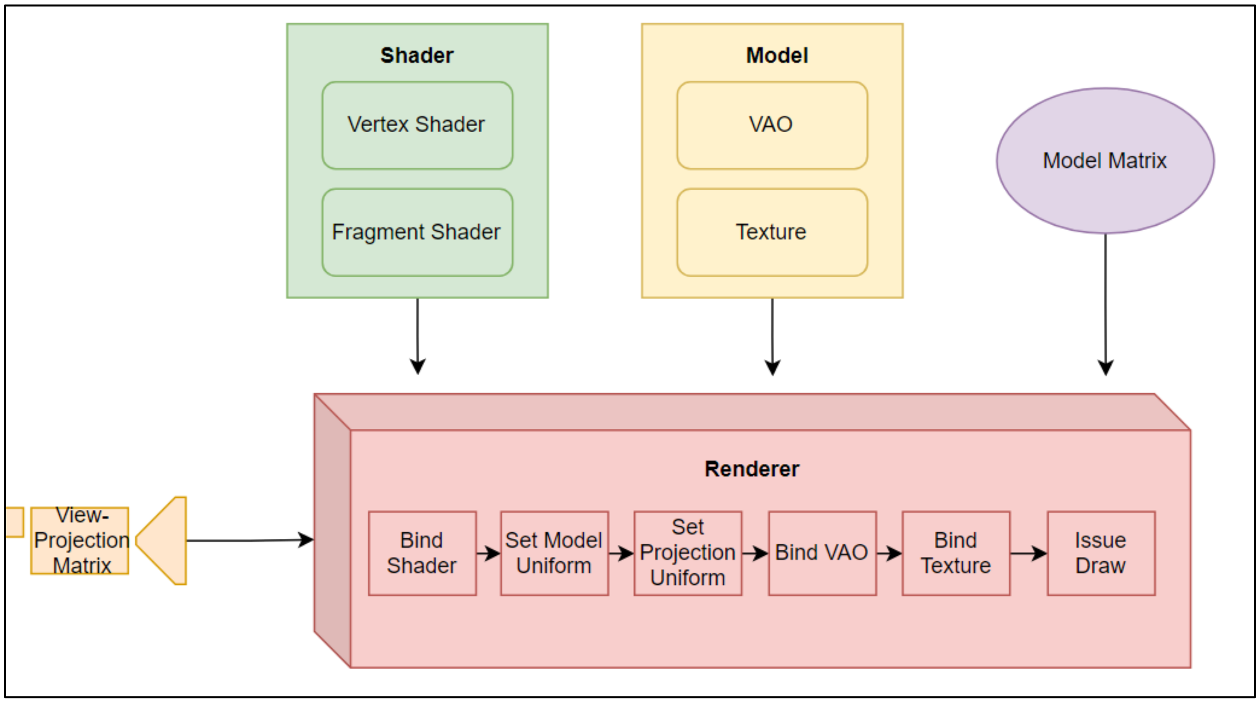


Figure 7.1 A closer look at the renderer architecture

VI. Entity-Component System

The entity-component system is a data-driven architectural design pattern used in game engines to organize game objects and their properties to be computationally efficient and human-readable. It is also used to increase modularity as entities (objects) can have components (properties) that are shared across multiple different ones [12]. For example, a model in *srqEngine* can be considered an entity as it is a game object which exists in the scene; it can have a collidable component attached to it which enables its AABB and adds it to the collidables list[13].

In the engine, this is implemented in the form of a scene manager, where a scene can have multiple entities attached to it so that we can render a single scene, abstracting away all the prior model initializations, particle generators, shader setups and so on. The idea is that we are associating a set of object configurations to a single scene object, so that we can draw different scenes in one go.

The scene management API contains two classes, the *Entity* and *Scene*. The scene class initializes an empty scene object which holds data structures of existing components – so far, only

[13] README – ECS • Source Code – Entity.h , Entity.cpp , Scene.h, Scene.cpp , Renderable.h , Renderable.cpp , Signature.h , Signature.cpp

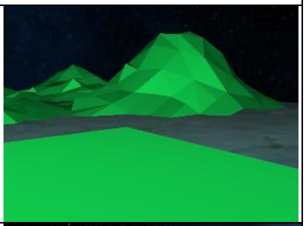
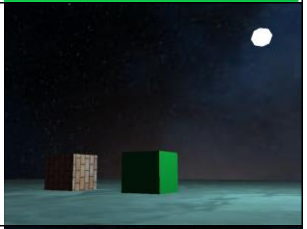
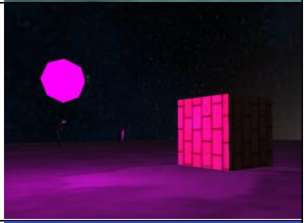
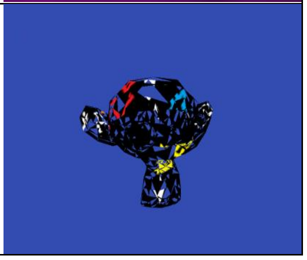
a *renderable* component which indicates that we are able to represent it with geometry. The scene class can also set an active scene, which will be the one that gets rendered.

The *Entity* class initializes after attaching itself to an existing scene object; the entity will be associated with it. The entity can then attach different modular components (such as the *renderable*) to itself through a C++ template. The scene object gets a copy of the unique components attached to each entity, in its data. Helper methods are used to generate unique, reusable signatures for entities to identify them.

VII. Testing

A. Rendering Tests

In these tests, unique rendering scenarios that I encountered during development are presented in the table below.

Component	**Test**	**Results**	**Screenshot**
Collision System	Should be able to jump and stand on collidable objects for vertical collision resolution	Player collides with top face and does not fall through; vertical collision works as intended	
Light	Diffuse light from a single light source should affect multiple objects	A single light source will act on multiple objects and even with different types of objects i.e. plain and textured	
Light	Color changes in light source should affect objects lighting	Different light colors effect the lighting of the models	
Model Importer	Model importer should not be able to handle complex primitives such as quads. Only triangles should be able to work.	Models exported with quads gets rendered in fragments.	

B. Stress Tests

Stress tests are intended to analyze the performance of the hardware and the graphics engine. The target frame rate is 60 FPS for ideal interactivity and immersion. In order to get accurate results, I calculated the milliseconds it would take in order to render the current frame.

$$Frames\ per\ second\ (FPS) = \frac{1}{Seconds\ per\ frame}$$

$$Let\ FPS = 60$$

$$\therefore 60 = \frac{1}{Seconds\ per\ frame}$$

$$\therefore Seconds\ per\ frame = \frac{1}{60} = 0.0167$$

So, we are aiming for around 16.67ms per frame in order to achieve 60 FPS. The hardware on which the tests were conducted is as follows

Device	Name	Manufacturer	Information
GPU	Intel® UHD Graphics 620	Intel Corporation	4128 MB total memory, 128 MB VRAM (display memory)
CPU	Intel® Core™ i5-8250U	Intel Corporation	@ 1.60GHz, Max 3.40GHz
RAM	Vengeance® LPX DDR4	Corsair	8 GB Memory @ 3200 MHz

The following tests describe the scene being rendered, a description of the test and statistics about the frame rate.

(i) Regular Scene

The scene consists of a couple of high-resolution models, lighting and a few of particles. The idea is that the scene is what a typical implementation of a video game would contain. This is to give us a point of reference on what to expect when performing the stress tests for exaggerated amounts of data transfer and processing to the GPU. *Fig. 8.1* shows what a typical scene would look like when rendered.

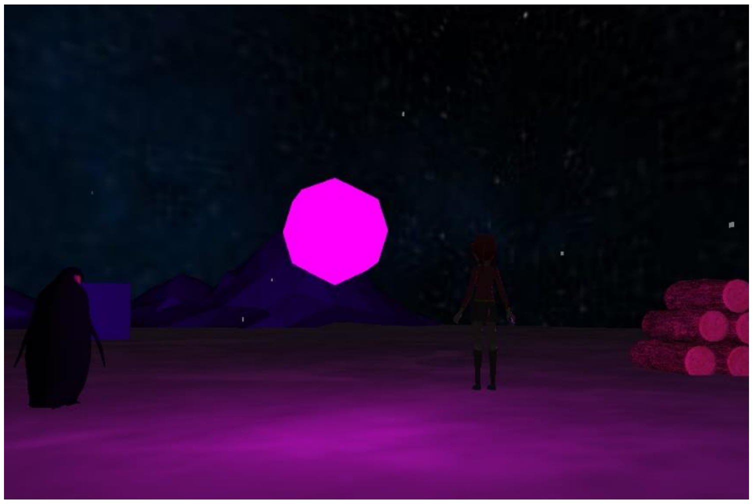


Figure 8.1 A typical rendered scene in srqEngine

The test consists of moving the camera around the scene, interacting with collidable objects and casting rays. The results are shown in *Fig. 8.2*, which indicates that in a regular scene, our frame rate remains within the desired amount of time (16.67ms).

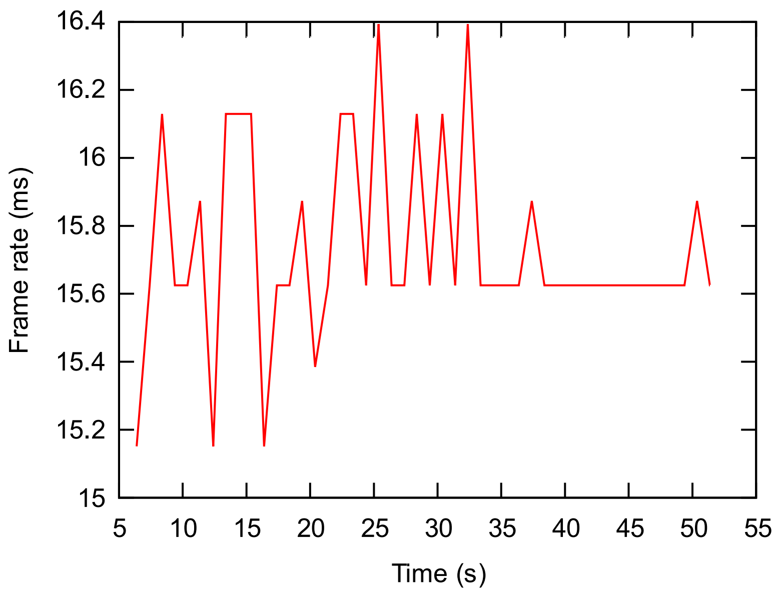


Figure 8.2 Frame rate in a regular scene over time

(ii) Particle Stress

In this test, the scene only contains our camera and an instance of a particle generator, which will be used to generate rectangular particles. Particles will be generated in increments; the goal is to see how this affects the frame rate. The results are shown in *Fig. 8.3*, where the frame rate was averaged during each increment. As you can see, over the course of 2.5 million particles, it takes a toll on the frame rate until it reaches a 1000ms, where it then tapers off and begins to skip frames entirely.

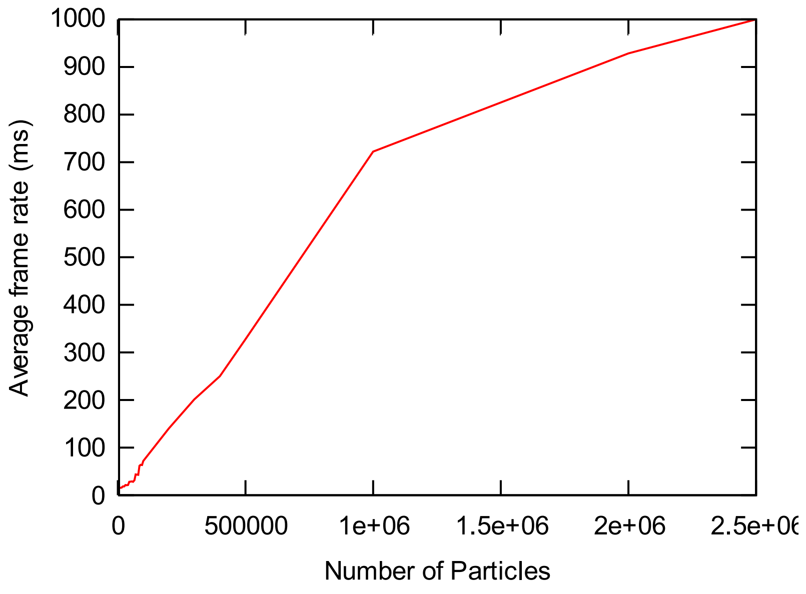


Figure 8.3 Frame rate with increments of particles

Fig 8.4 shows a screenshot of generating a vast number of particles. The camera barely moves due to how long it takes for a single frame to process.

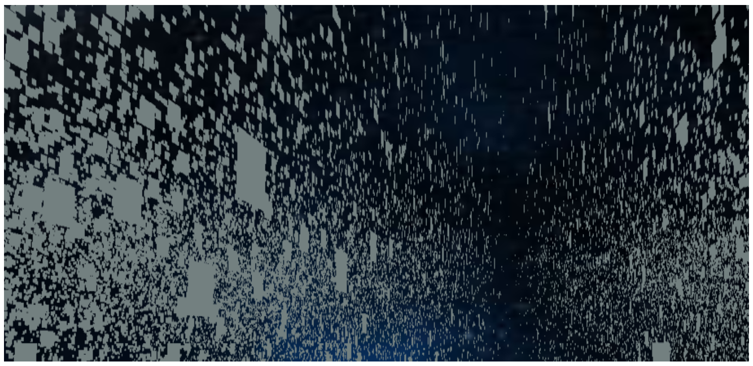
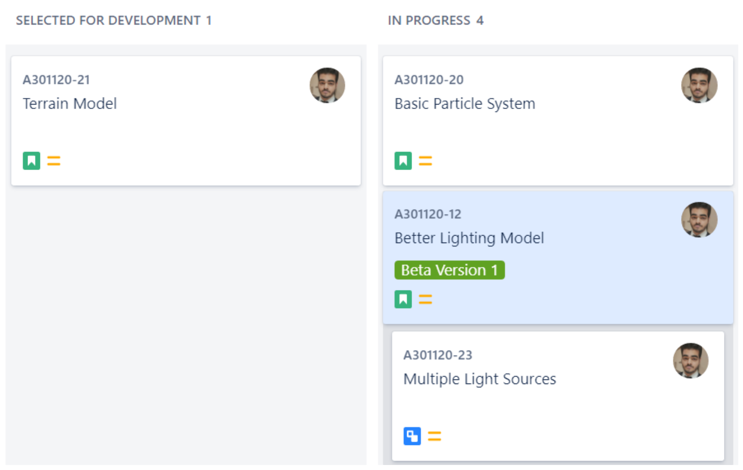


Figure 8.4 100,000 particles in srqEngine

PROJECT PLANNING

The project planning began in September 2020 with a prototype of the engine. I utilized Kanban boards which contained cards for selected issues, issues in progress and completed issues (see *Fig 9.1*). This helped me keep track of the implementation as I could reference the board whilst programming.


SELECTED FOR DEVELOPMENT 1

A301120-21
Terrain Model

IN PROGRESS 4

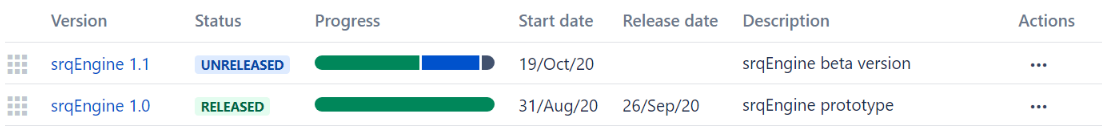
A301120-20
Basic Particle System

A301120-12
Better Lighting Model
Beta Version 1

A301120-23
Multiple Light Sources

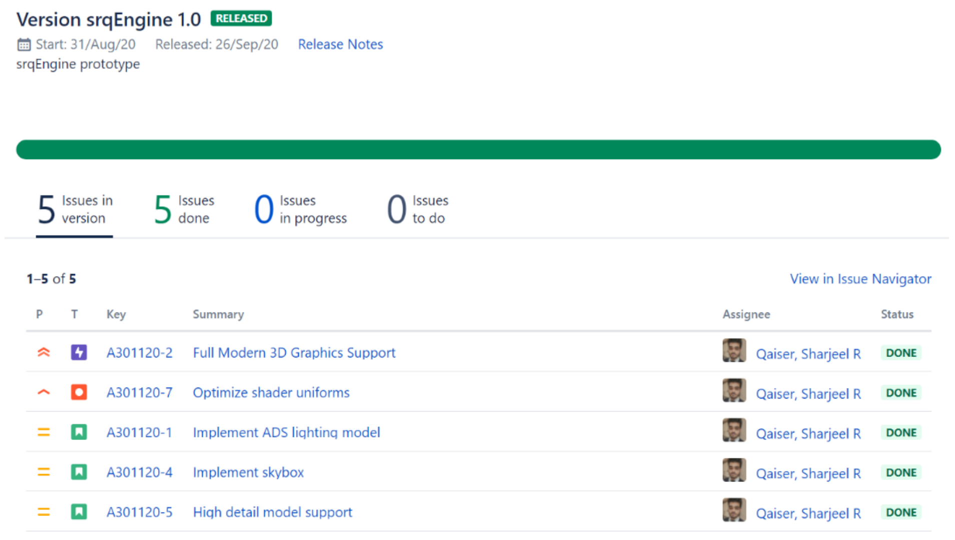
Figure 9.1 Section of Kanban board

I utilized Jira's Release system in order to keep issues organized within a release update. The prototype was released within a month, with a couple of issues attached. Each issue featured a video demo, a mini report so that I could reread it in case I needed to revise anything, and a breakdown of the sub-tasks within that issue. I also ensured to place the GitLab SHA commit code in case I needed to reference the git commit for the implementation details pertaining to that issue.



Version	Status	Progress	Start date	Release date	Description	Actions
srqEngine 1.1	UNRELEASED		19/Oct/20		srqEngine beta version	•••
srqEngine 1.0	RELEASED		31/Aug/20	26/Sep/20	srqEngine prototype	•••

Figure 9.1 Section of Kanban board



Version srqEngine 1.0 RELEASED

Start: 31/Aug/20 Released: 26/Sep/20 Release Notes

srqEngine prototype

5 Issues in version | 5 Issues done | 0 Issues in progress | 0 Issues to do

1–5 of 5 View in Issue Navigator

P	T	Key	Summary	Assignee	Status
⌃⌃	⚡	A301120-2	Full Modern 3D Graphics Support	Qaiser, Sharjeel R	DONE
⌃	●	A301120-7	Optimize shader uniforms	Qaiser, Sharjeel R	DONE
=	▣	A301120-1	Implement ADS lighting model	Qaiser, Sharjeel R	DONE
=	▣	A301120-4	Implement skybox	Qaiser, Sharjeel R	DONE
=	▣	A301120-5	High detail model support	Qaiser, Sharjeel R	DONE

Figure 9.2 Version release in JIRA

I also used various features within an issue to label my work correctly so that I could quickly identify the details of it. For example, I used epics to link a group of issues under a single category, and I made use of the bug issue when it was appropriate. The priorities were also set to give me a clear picture of what I needed to work on first as shown in *Fig. 9.2*.

I was quite successful at maintaining momentum up until February 2021, where it started to decrease a bit, due to pressure from other commitments, but I bounced back after that and made a final effort to get another release out.

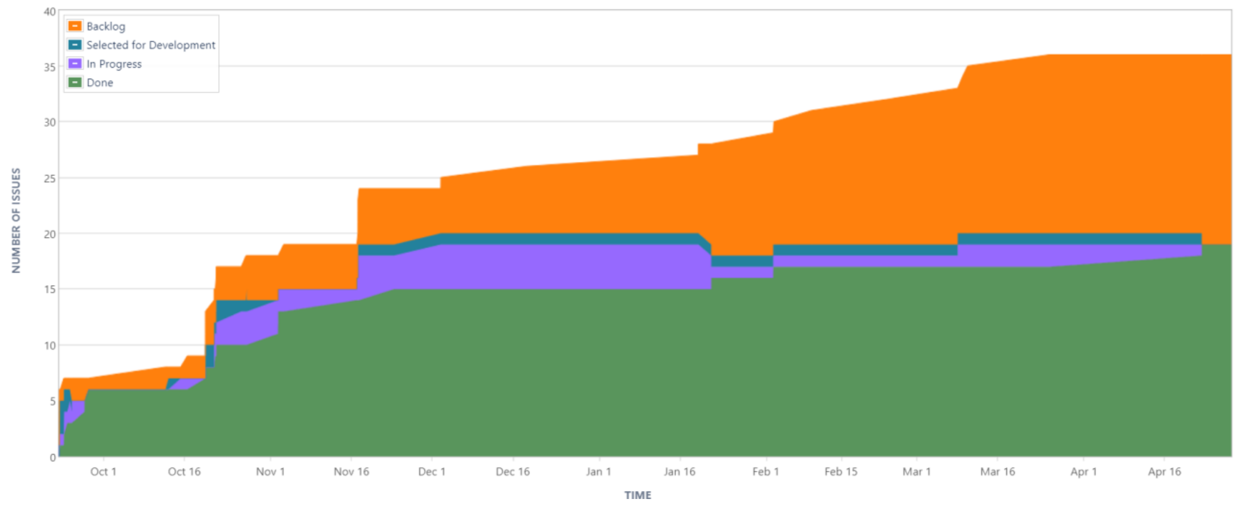


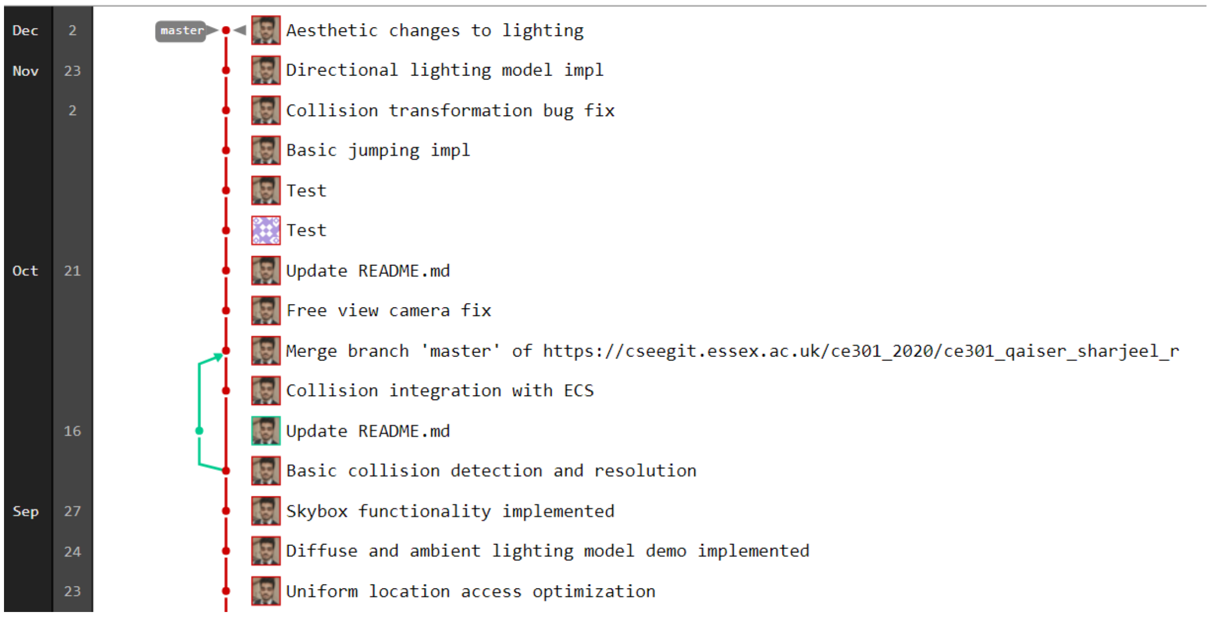
Figure 9.3 Issue cumulative flow diagram

I used GitLab to host the code and technical documentation .

One risk that I encountered was an optimization bug which decreased my engine's performance quite a bit, this was due a shader uniform being bound in the game loop every iteration when it could have been accessed from a predefined shader configuration instead (which was the solution). In order to overcome this, I set up an issue for the bug and worked around it, being aware of its consequences in the program until I figured out the solution.

I used the agile methodology to keep improving my project in iterative steps, which proved to be useful as I had time to reflect on the potential changes and upgrades, as the requirements changed. It proved to be suitable as I managed achieved most of my initial goals.

I am happy with my overall achievements and performance as I set the bar really high from the start, in order to motivate me to finish feature implementations.



```
Dec  2   master  Aesthetic changes to lighting
Nov 23           Directional lighting model impl
     2           Collision transformation bug fix
                 Basic jumping impl
                 Test
                 Test
Oct 21           Update README.md
                 Free view camera fix
                 Merge branch 'master' of https://cseegit.essex.ac.uk/ce301_2020/ce301_qaiser_sharjeel_r
                 Collision integration with ECS
    16           Update README.md
                 Basic collision detection and resolution
Sep 27           Skybox functionality implemented
    24           Diffuse and ambient lighting model demo implemented
    23           Uniform location access optimization
```

Figure 9.4   GitLab repository graph

CONCLUSIONS

In conclusion, the engine produced provides many features that developers can use in order to make prototypes for their games easily, without having to experience the difficult aspects of graphics rendering. Graphics modeling artists can also use the engine to quickly import their art into a scene using the accessible model importer to see how it looks within a video game. The engine showcases the power of modern graphics technology by providing real-time rendering with high definition models, textures, particle effects, global illumination and physics.

The final result is a successful product that can be used to create different scenes with elements from the DOOM-game, using the tools provided by the renderer API. The engine is highly customizable because it is written in native C++, so developers can use memory-level functions in their implementations. They also have an assurance of high performance because of this.

There are many ways to improve the engine to make it more accessible, for starters, the entity component system could use a lot of work as it currently features a single component. Engine contributors can write custom components to be attached to entities in order to steadily build a library of attachable components that can be quickly accessed and used through method calls by developers. Because I abstracted the shaders in the renderer API, developers can also write their own shader files with GLSL, which the engine will handle automatically. They may want to play around with custom lighting models or ray-casts, which can be done through the shaders. The particle system could also be improved by creating different particle models to emulate a larger variety of special or atmospheric effects.

In order to make this an engine on the level of commercially available ones such as Unity or Unreal Engine, it would need some revamping and vast amounts of improvements. Some planned features that didn’t make it into the final implementation include terrain composition/decomposition, a sound system with RTX reverb, advanced collision detection algorithms (quadtree/octree) and day/night cycles (animated skybox). These can all be worked on in order to improve the engine.

REFERENCES

[1] T. Möller, N. Hoffman and E. Haines, Real-Time Rendering, 1999.

[2] D. J. Eck, "Projection and Viewing," in *Introduction To Computer Graphics*, 2018.

[3] J. F. Blinn, "A trip down the graphics pipeline: pixel coordinates," *IEEE Computer Graphics and Applications,* vol. 11, no. 4, pp. 81-85, July 1991.

[4] Coding Labs, "World, View and Projection Transformation Matrices," [Online]. Available: http://www.codinglabs.net/article_world_view_projection_matrix.aspx. [Accessed April 2021].

[5] P. Shirely, Fundamentals of Computer Graphics, 2002.

[6] S. H. Ahn, "OpenGL Projection Matrix," 2008. [Online]. Available: http://www.songho.ca/opengl/gl_projectionmatrix.html. [Accessed April 2021].

[7] Khronos Group, "Official OpenGL Documentation," 2006. [Online]. Available: https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/gluPerspective.xml. [Accessed April 2021].

[8] A. F. Clark, "Computer Vision" lecture notes for module CE316, CSEE , University of Essex, 2020.

[9] C. Ericson, Real-Time Collision Detection, 2004.

[10] F. Dunn, 3D Math Primer for Graphics and Game Development, 2002.

[11] S. V. Gordon and J. L. Clevenger, Computer Graphics Programming in OpenGL with C++, 2018.

[12] A. Morlan, "A SIMPLE ENTITY COMPONENT SYSTEM (ECS) [C++]," 25 June 2019. [Online]. Available: https://austinmorlan.com/posts/entity_component_system/#what-is-an-ecs. [Accessed April 2021].

[13] J. Vince, Mathematics for Computer Graphics, 2005.

[14] J. Miller, "Vector geometry for computer graphics," *IEEE Computer Graphics and Applications,* vol. 19, no. 3, pp. 66-73, 1999.

[15] W. Cheney and D. Kincaid, Linear Algebra: Theory and Applications, 2007.

APPENDIX

I. Introduction

We have to represent our graphics with mathematical models because the machine only understands binary. Certain concepts from linear algebra and geometry play a key role in forming this representation. We may need to represent vertices that make up a geometric shape such as a cube; any particular vertex on this cube is typically represented as a vector.

A. Vector

A vector is simply a point in space with a magnitude and a direction, where the magnitude is the length of the vector and the direction is the graphical area it points towards with respect to its origin (normally at coordinates *x*: 0, *y*: 0, *z*: 0 for most practical purposes in CG). In the cube example, the vertices can be represented as vectors as shown in *Fig A.1*.

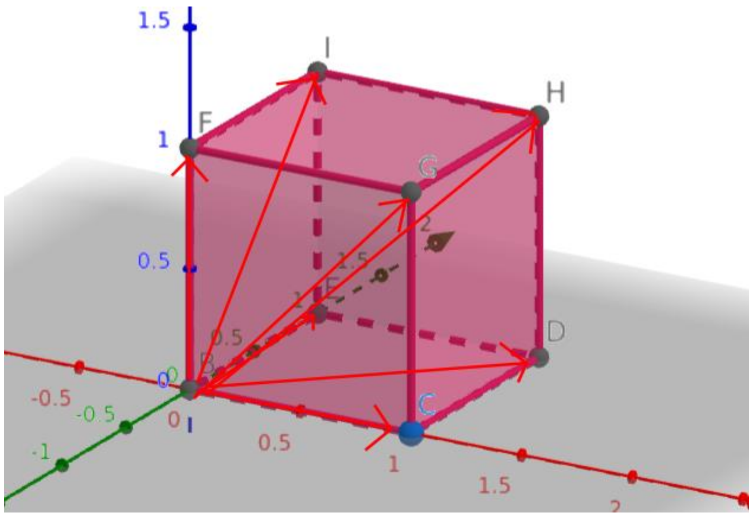


Figure A.1 Vectors representing a cube

The vectors are represented with red arrows. They start at the origin and point towards the vertices marked with alphabets. They have a magnitude and a direction [13, pp. 30-32].

Another important vector concept is the *unit vector*. This is the vector with a magnitude of 1, in the direction of each axis. So, for our 3D coordinate plane where we represent the cube, the unit vectors would point towards (1,0,0), (0,1,0) and (0,0,1) from the origin. This is helpful when understanding operations on the vectors that make up an object. This is also known as the *direction vector*.

B. Matrix

A matrix is a rectangular array of numbers with a certain number of rows and columns. In CG, they are a numerical representation of transformations and are crucial for performing said (affine) transformations on *renderable* objects such as the cube. Some of the common ones are translation, scaling and rotation, but there are other ones such as reflection, shearing and even custom transformations without any predefined name. We can also combine a series of transformations into a single matrix that represents them all (in the order that they are performed) through matrix multiplication.

Typically, in the context of three dimensions, we use 4x4 matrices for all linear transformations (as opposed to 3x3 matrices) since we need an extra dimension due to handling perspective projection (homogeneous division). This extra dimension of coordinates is said to be homogeneous [10, p. 113].

The transformation matrix is applied to each vertex of the geometric shape (which is represented by vectors). If we wish to move our cube around by some coordinates, we will need a translation-matrix. We already have the vertices of the cube in form of a vector representation, which is essentially just a 4x1 matrix with the rows being *x*,*y*,*z* coordinates and a homogeneous coordinate 1 at the end, therefore matrix multiplication is possible here as it would be 4x4. The translation transformation would look like the following

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{pmatrix}$$

Where T_x , T_y , T_z are how much we wish to move the coordinates *x, y, z* by respectively. The resultant vector is the new location of the vertices.

II. Operations on Vectors

The vector multiplication operation appears repeatedly in certain areas of CG, especially when dealing with viewports, perspectives and cameras. There are two types of vector multiplication techniques that we use, namely, the dot product and the cross product (although there are other types such as the Hadamard product). These operations would be performed between vertices in vector-form for various applications [14].

A. Dot Product

The dot product is a type of vector multiplication where two vectors are *dotted*, which results in a *scalar* - the product of their magnitudes times the cosine of the angle between these two vectors as shown below.

$$x \cdot y = \|x\| \cdot \|y\| \cdot \cos(\theta)$$

Where x and y are two distinct vectors and θ is the angle between them. This has a very interesting property. If x and y were unit vectors, that would mean their magnitudes would be 1. Therefore, the formula could be evaluated as

$$x \cdot y = 1 \cdot 1 \cdot \cos(\theta) = \cos(\theta)$$

Now we have an efficient way of getting the angle between these two vectors and determining if they are orthogonal (at right angles) or parallel; the result would be 0 and 1 respectively. If they aren't unit vectors, you can simply divide the magnitudes from both matrices [10, p. 56].

B. Cross Product

The cross product is another type of vector multiplication where two vectors (that are non-parallel) are *crossed*. It is only possible to calculate it in 3D space. Unlike the dot product, this results in another vector as the output (as opposed to a scalar). The key feature of this is that the output vector is always orthogonal to the two vectors that were crossed. Note that if the two input vectors were already orthogonal to each other, the end result would be three vectors that are fully perpendicular. This is powerful for constructing coordinate systems. The formula for calculating the cross product between two vectors involves a lot of matrix multiplication so I'm omitting the details here, the important part to know what it outputs (for our purposes) [10, p. 66].

III. Gram-Schmidt Process

The Gram-Schmidt process is used for finding a set of orthogonal vectors that are normalized to unit vectors (orthonormalization) within an inner product space - Euclidean space for our purposes. The set of non-zero vectors that belong to the inner product space, is the orthogonal set if they are orthogonal to each other, so given any two vectors in this set v_a and v_b , $v_a \cdot v_b = 0$ (dot product) where $a \neq b$. If on top of this orthogonality property, they are also unit vectors (meaning their magnitudes are 1), it is said to be the orthonormal set (as opposed to orthogonal). To get the *orthogonal projection,* p, of a given vector x , onto the orthogonal set of vectors, *V,* in that inner product space, the following formula is used

$$p = \left(\frac{x \cdot v_1}{v_1 \cdot v_1} * v_1\right) + \left(\frac{x \cdot v_2}{v_2 \cdot v_2} * v_2\right) + \cdots + \left(\frac{x \cdot v_n}{v_n \cdot v_n} * v_n\right)$$

Where $\langle v_1, v_2 \ldots v_n \rangle \in V$

The Gram-Schmidt process uses the orthogonal projection concept described above to get the orthogonal basis vectors for a given inner product with any set of basis vectors. Suppose we have an inner product space *V* with the set of *basis* vectors $\langle x_1, x_2 \ldots x_n \rangle$. To get the *orthogonal basis* vectors $\langle v_1, v_2 \ldots v_n \rangle$, we must apply the *Gram-Schmidt* process steps which are as follows

Let

$$v_1 = x_1$$

$$v_2 = x_2 - \frac{x_2 \cdot v_1}{v_1 \cdot v_1} * v_1$$

$$v_n = x_n - \frac{x_n \cdot v_1}{v_1 \cdot v_1} * v_1 - \cdots - \frac{x_n \cdot v_{n-1}}{v_{n-1} \cdot v_{n-1}} * v_{n-1}$$

[15, pp. 544,558]. And so, out of the inner produce space's basis vectors, we were able to find the set of orthogonal basis vectors. This is especially useful when we are constructing the camera in *srqEngine* to view the scene in.