Ubiquitous Reality. A 3D Game Engine for Stylisation and Performance.

Group Report for COMP5531

ALEXANDER CLARIDGE*, TEJASWA RIZYAL*, ABHIGYAN GANDHI*, SAHIL PURI*, and MICHAEL ASIAMAH*, University of Leeds, United Kingdom
MARKUS BILLETER†, University of Leeds, United Kingdom

Fig. 1. A scene from the game *Axiom Swerve* using different visual effects provided by the Ubiquitous Reality Engine.

As game engines become more accessible, they increasingly focus on providing general purpose tools, introducing unnecessary overheads and slowing down rapid development for genre specific games. We introduce a game engine that is focused on vehicle physics, artistic rendering and large scale entity handling. This enables rapid development for different styles of vehicular-based games, be that arcade or simulation based. Games developed with our engine are not confined to the racing genre, using vehicle movement to explore a variety of gameplay styles and mechanics. We do this by leveraging a powerful and modular entity-component system, leaving no lack of ability on the developers side. A wide range of rendering techniques enables highly stylized visuals and an expansive event handling system allowing the user custom control over the inner workings of the engine all while not alienating default features and use cases. We discuss the reasoning behind and simultaneous development of the game engine while exploring out-of-the-box features and genre-specific implementations offered by the Ubiquitous Reality Game Engine.

*Sharing first authorship
†Also with the role of Supervisor.

Authors' addresses: Alexander Claridge, sc20ac@leeds.ac.uk; Tejaswa Rizyal, sc20tar@leeds.ac.uk; Abhigyan Gandhi, tcrz4884@leeds.ac.uk; Sahil Puri, sc21sp2@leeds.ac.uk; Michael Asiamah, ed20m4a@leeds.ac.uk, University of Leeds, Woodhouse, Leeds, West Yorkshire, United Kingdom, LS2 9BW; Markus Billeter, m.billeter@leeds.ac.uk, University of Leeds, United Kingdom.

CCS Concepts: • **Computing methodologies** → **Computer graphics**; **Texturing**; • **Applied computing** → *Computer games*.

Additional Key Words and Phrases: Game Engine, Rendering, Vulkan

1 INTRODUCTION

The video games industry has always been a market for creativity, fun and art through an interactive media. Games often utilise artistic means of communication, incorporating a large variety of effects into the gameplay to create a sense of immersion or provide cues for the player. If a developer wanted to incorporate these effects, they could use a general purpose, publicly available game engine, and attempt to leverage the available features into their specific desired effects. The downside of this is that the engine can come with unnecessary bloat for features that the developer might not want to include. Additionally, the engine might not support all desired effects. The next option would be to modify an existing engine, however, this can come with the same issues if the engine isn't built for easy modification, or relies on certain features to not be removed. There are even some documented cases of engine's designed for one genre being forced into building a game of a very different genre, which emphasises many of the established issues.

To solve these issues, the final solution would be to build a custom game engine.

The main advantage of a custom engine is the ability to forgo generalization. There is no reason to appeal to the average game developer, so features are exclusively developed with the resulting game in mind. This is a significant advantage, not only because it completely eliminates obsolete code and avoids unnecessary complexity, but also because features can be optimised specifically towards the desired engine direction without worry of maintaining generalised features. Clearly, this is an optimal scenario, where only necessary components are designed to maximize efficiency.

The Ubiquitous Reality Game Engine is designed to help with these issues and provide an easy to use, configurable and adaptable development process for stylistic vehicular games. The engine provides interfaces for physics object control, an easy to use yet performative entity component system, and many visually distinctive effects. We intend to give the developer access to complex, out-of-the-box stylization effects to achieve easy to use artistically enhanced gameplay.

2 BACKGROUND RESEARCH

Game Engines encapsulate the foundation of games, providing much of the technical base that goes into game development. Often, game engines are crafted to satisfy the needs of a particular genre, for example *Supergiant* spent time rewriting their engine for the creation of *Hades* [The Forge 2025]. In other scenarios, targeted decisions, like those made by *Rocket League* creators *Psyonix*, resulted in their game having heavy modifications on the existing engine *Unreal Engine 3* [Bankhurst 2021]. Generalised game engines have been popular in the game development community for some time, however, this does not negate the argument that game development benefits most from engines tailored to a specific genre [Gregory 2018].

Designing genre specific game engines presents unique challenges. Compared to generalised engines, specialised engines require a more detailed functionalities list. An understanding of what exactly is required needs to be present from the start, as specialised features may have specialised dependencies and unique learning curves. Due to these reasons, we need to look into the genre and styles we aim to achieve and define both engine functionality and style goals.

2.1 Vehicle Action Games

The racing genre is generally either considered an all encompassing genre for all vehicular games, or pin pointed to only the elements of racing. This discrepancy is understandably due to the fundamental features that a game engine must provide to effectively support a specific genre. From the engine's perspective, there is little difference in features needed for a purely racing game, compared to an action game that uses a player vehicle. Although from a gameplay and game design perspective the creation is completely different. Certain games cross the threshold even in gameplay and utilise both sets of features, for example *Need for Speed Heat* [Ghost Games 2019] and *Wreckfest* [Bugbear Entertainment 2018]. These games emphasise action either in the racing itself, or promote it more in other parts of the game. One can argue that *Grand Theft Auto V* [Rockstar North 2013] and it's vehicle racing dynamics utilise the same mix of features.

Considering a comprehensive vehicle-based game engine, many features stand out. Racing games are often credited with some of the most impressive visuals. Games like *Gran Turismo* and *Forza Horizon* present incredibly impactful graphics and highly detailed world and environment. The lack of animations and having largely static assets contribute to this aesthetic, as you often see the character models and cutscene models do not present the same amount of detail as the environment or the cars [Minotti 2021].

At the same time, a gameplay largely involving just vehicles removes any issues like 'Uncanny Valley'. The uncanny valley is an idea that characters appear *almost* human-like but not quite real create a negative emotion in humans [Ratajczyk 2019]. Although the public opinion of this phenomenon is quite visibly available on the internet [tvtropes 2025b], the racing game genre is less affected by it. Stylistically, this gives more freedom to the racing genre's visual expression.

The genre is also known to explore more fantastical or cartoony approaches to much style and gameplay. Namely, *Mario Kart* uses item boxes which provide power-ups to the player and have features like drift to boost [Nintendo 2017]. Nintendo, the creators of Mario Kart are not only known for the so-called 'Nintendo Polish' implying the publicly noted ability to provide critically acclaimed games[Scott 2018], they also use internal game engines that are often designed specifically for a particular game. Nintendo opted to design a custom engine particularly for *The Legend of Zelda: Breath of the Wild* allowing programmers of the game to have custom control over every functionality [Main Leaf Games 2024].

Another notable game that explores unique avenues in the genre is *Rocket League*, arguably the most popular vehicle game that does not involve racing [CardinalSpawn 2020]. The success story of the game shows that the genre can be pushed even more, with flying cars playing football and an excessive particle system. Despite the rendering style being more 'cartoony', the gameplay emphasizes different graphical effects and highlights specialised graphics for vehicles [Mejia 2015].

2.2 Bullet Hell

Another genre which is heavily stylised and presents an idea on level design that focuses on populating the player's screen is the Bullet Hell genre [tvtropes 2025a]. Specialising in constantly having tasks at hand, and often exploring themes of *sci-fi* and fantasy genres, bullet hell games from an engine perspective, do not require too many unique features. Although this genre does not have much overlap with vehicular games, the idea of screen overpopulation, non photo-re stylisations and constantly having something to do, result in a similar mechanic as to what our engine aims to accomplish.

NieR:Automata is a 2017 action-adventure game with aspects of many genres incorporated within the gameplay, including bullet hell, where they focus on heavily stylised bullets [Platinum Games 2025; Wolf 2017]. The developers of the game used a proprietary engine the integrated Silicon Studio's 'Enlighten' rendering engine to achieve specialised lighting and effects beyond the capabilities of

standard game engines. [Silicon Studio 2025]. The developers use 'Enlighten' to "change lights in real time and finely tune global illumination" giving them a wider range of expression [Silicon Studio 2021]. The sub-genre is now heavily characterised by this uniqueness and the requirements out of an engine vary stylistically to provide more control and freedom over effects especially if the developer's intention is to link effects with gameplay and combat.

Popular games in the genre follow similar choices, providing aesthetically pleasing music, ambience, and unique graphics. *Hades* is another extremely popular game in the sub-genre and emphasises the art and style in every frame. The developers actively plan the gameplay and art, lobbying for specialised artistic choices, making it one of the most awarded game in recent history [MCV Develop 2021]. This success is also associated to *The Forge Interactive* [The Forge 2025] who have been been working with *Supergiant*, the developer of *Hades* since 2014. Prior to the development of the game the team at Forge help building a new cross-platform game engine in C/C++ which allowed them more rendering freedom in the development of the game [The Forge 2020].

The (sub-)genre is seen in many games associated with different major genre, but does not appear to have a large catalogue of games involving vehicles. Considering both genres provide stylistic rendering and effects, and rely on static models for environment or bullets, the avenue of incorporating both these genres in a game engine and game style is viable. The engine would require to be able to cover grounds in providing features, but as the research suggests, most of these games focus heavily on stylisation and generally use specialised engines specifically for rendering. This implies that an engine covering both bases and allowing a customizable and powerful rendering solution should be able to allow the development of games in either of these genres and even combining them into one game.

2.3 Stylisation

As the prior sections suggest, emphasis on user controlled effects and a powerful rendering system are crucial for an engine to support the stylised games mentioned. One of the reasons generalised engines fall short is that features like these require custom pipeline structures and configurability that generalised game engines often do not plan for [ter Elst et al. 2025].

Need for Speed Unbound [Criterion Games 2022] is a game that presents a visually unique style. The game uses a combination of cell-shaded smoke and, graffiti-inspired post processing effects on top of realistic PBR shading. In terms of performance the game has been known to drop frame rate below 55 fps in Xbox Series S and Series X consoles, which gets more noticeable in high intense races while rendering these effects [Mackenzie 2022]. The game was built using the *Frostbite* engine, which was originally meant for multiplayer First Person Shooters. Previous games built on the engine often criticised the engine for its rigidity and missing features such as inventory, items or even saving game data [Williams 2019]. Clearly there are flaws in implementation and as Frostbite is certainly not a specialised engine for either racing games or any effects that the game needed, improvements can be made.

Another stylistic inclusion with the research so far can be the cyberpunk aesthetic, including different effects that the genre is known for. An example of the style would be *Observer* [Bloober Team 2017], a psychological horror that uses screen space effects as a gameplay and narrative mechanic. These include using a 'glitch' post processing effect indicating low health, different levels of screen space blurring indicating psychosis, and a power up 'electro vision' that incorporates object specific effects. This incorporation of different rendering effects with mechanics makes the game more immersive and contribute to it's success [Inglis 2025; Vincent 2017].

Considering the already proven success of *Observer* and its stylistic choices, it is important to understand that the game was built using the *Unreal Engine* [Epic Games 2025c] and post processing pipelines in the engine are prone to stuttering. This has been acknowledged by the *Unreal* Engineers, but the issues have not been completely fixed, especially for games built in earlier versions [ter Elst et al. 2025]. The developers discuss the difficulty in updating the codebase to improve this for multiple post processing pipelines, as the engine was not originally designed for that. This is another example of the necessity for specialised game engines for these features as opposed to modifying the engine and battling rigidities in the example of Frostbite as well.

2.4 Game Engines

The engine we aim to develop is intended to specialise in the vehicle based genre but we aim to encompass more stylistic choices out of the box, encouraging hyper-stylised screen space and object specific rendering. Effects from *Rocket League* and *NFS: Unbound* as mentioned in Sections 2.1 and 2.3 require emphasis on performance and providing the ability to control different aspects of these effects from the game developers point-of-view.

Generalised game engines do not provide the modularity and support that these games need, as can be seen in issues relating to *Unreal Engine*'s performance in shader pipelines and 'stuttering issues'. Previous sections highlight the lack of success of generalised game engines in the genres and style-specific choices that these games make; only *Observer* is a game made using modern, and unmodified generalised engine, without significant overheads upon release. Rocket League uses a highly modified version of UE3, making them struggle against updating to UE5; NFS Unbound uses Frostbite, and issues while using that are highly documented. Every other game mentioned is built on a proprietary and highly specialised engine. We believe this highlights the need for, and the aims of 'UBQR'.

The architecture of the engine should support the systems that these games require and benefit from. Attention obviously falls on a performative and modular rendering system, a highly competent ECS and event processing to handle the overpopulated screens that Bullet Hells are famed for and a physics system capable of tackling vehicular physics and collisions.

2.4.1 Architectural Design Patterns. Game engine's in the past used deep inheritance techniques for system architecture. There are a lot of problems associated with this pattern, such as "The Deadly Diamond" [Gregory 2018], making it a highly unideal pattern to employ in our engine. In 1998, *Thief: The Dark Project* became the

first game to employ an entity-component system (ECS) [Ovid 2021], to great success. Since then, games and game engine's have employed this architectural design pattern an increasing amount, favouring its flexibility for the many different types of objects that could be in a scene. Looking at the popular, publicly available engines *Unreal* Engine and *Godot*, both of them use this composition styled architecture. Although they still use hierarchy in the entities and components themselves. In *Unreal* Engine, entities inherit the `AActor` class, which goes up the inheritance tree of `UObject`, `UObjectBaseUtility` and finally `UObjectBase` [Epic Games 2025a]. Components are assigned to the entity, which are also hierarchically based. For example, all components that are assigned to the actor inherit from `UActorComponent` [Games 2025]. In Godot, this is similarly done, for example to attach a 3D directional light node to an entity (known as a *scene* in *Godot*) [Godot 2025], you'd need to inherit from the ordering of `Light3D`, `VisualInstance3D`, `Node3D` and finally `Node` [Juan Linietsky 2025]. Even in these ideas there is still inheritance based problems, such as "The Bubble-Up Effect", where common functionality between different inherited classes tends to travel up the hierarchy until everything is forced to inherit from it [Gregory 2018]. So going further into this idea of composition, we can look towards ideas that use much less hierarchy, such as the popular publicly available game engine Unity. Unity uses an entity component architecture, which is different from an entity-component system architecture. In Unity, the systems (such as physics or rendering) are completed within the components themselves. *Unity* does offer *Unity DOTs*, which updates the systems to use a true entity-component system architecture, advertised to be a much better system for handling large groups of entities [Härkönen 2019]. Using an ECS gives a lot more flexibility to work with, even allowing for better accessibility within games [Muratet and Garbarini 2020]. With this in mind, we ultimately decided to go for an entity-component architecture using the "pure components model" [Games 2025], a method which avoids inheritance completely, opting for a purely data based system. This method promotes both scale and manageability [Saroufim 2020], allowing us to create a good system that encourages the use of mass entities, which is clearly the case in a bullet hell.

2.4.2 Renderer. The requirements from the renderer entail a few different things, namely, different post processing effects, customizability during each frame to allow the developer to use the effects as parts of gameplay, and performance. Other games that utilize these techniques generally have a bottleneck at multiple pipelines as is the case in *Unreal Engine*. Consideration to minimizing performance overheads must be taken to allow for maximum effects, as these include data transfer to the GPU in forms of images that can be a cause of decrease in frame rate.

Additionally, to allow for certain visuals like combining object specific rendering, like have certain object on screen be non-photorealistic while everything else is physically based, we need multiple pipelines that have only certain objects. This involves not only a larger setup, but also can often have issues with depth testing. Combating this is ideally done using Depth pre-pass, or a more naive shader logic to render *black* on unneeded objects. As the later is easier, but introduces overhead, the realistic solution is to start with the naive version and refactor into the former when it becomes a bottleneck.

In term of effects, the engine has to take the responsibility of all synchronisations and encapsulate the Vulkan API so that the user is able to use our ECS to inform the engine about all model rendering data, and have access to specific rendering arguments that the shader can use.

2.4.3 Physics System. From the start, physics was deemed to be an important aspect of this project. Although not every game requires physics, most game engines have an integrated physics system. *Unity* has *NVIDIA PhysX* engine integrated for 3D physics and *Box2D* engine integrated for 2D physics[Unity Technologies 2024]. *Epic Games* uses an in-house physics engine dubbed *Chaos Physics* for *Unreal Engine 5* [Epic Games 2025b]. With the goal of creating a game with vehicular physics and collisions, it was important that the chosen physics engine for this project could fulfil those needs.

PhysX is a physics engine developed by *NVIDIA*. It became prominent for its GPU-accelerated capabilities, offering high-fidelity simulations for rigid bodies, fluids, cloth, and particles [NVIDIA Corporation 2025]. It remains widely used in applications that benefit from hardware acceleration and realistic physics.

Bullet Physics is a real-time physics engine, widely used in games, robotics, and VR simulations. It supports rigid and soft body dynamics, collision detection, and constraints [Coumans 2025]. Bullet is known for its flexibility, modularity, and integration into systems like *Blender* and the robotics simulator *PyBullet*.

Jolt Physics is a high-performance open-source physics engine for real-time applications, especially games. It prioritizes speed, multithreaded scalability, and deterministic behaviour. *Jolt* focuses solely on rigid body simulation, omitting soft bodies and fluids to maintain a lightweight and efficient core [Rouwe 2025]. It's ideal for gameplay-focused simulations requiring precise control and low overhead. It also includes a built-in vehicle system that simulates wheeled vehicle dynamics using configurable suspension, steering, and drivetrain behaviour, optimized for stable and responsive gameplay rather than full physical realism.

3 METHOD

3.1 Entity Component System

Part of the advantage of making a pure entity component system is that the setup for data storage is very lightweight and easy to implement. Firstly, we setup a list. Then, an associative list is created for each component, this list stores the entity ID of each component. Finally, we a create a single 2D list for indexing, which is populated during engine initialisation. With these three list types, the engine can gather information for any component given either its index in its component array or its Entity ID at a complexity of $O(1)$;

It's traditional for game engine's to give a name to their entities; (`GameObjects` in *Unity*; Scenes in *Godot*) in the UBQR game engine, entities are refereed to as *GameWares*. To give a better user experience, developers are met with GameWares in the style of an unique interfacing system as shown in Figure 2. This system is presented to the developer in a way where it gives the appearance of a tangible entity. This makes the system easily comprehensible. Then, when

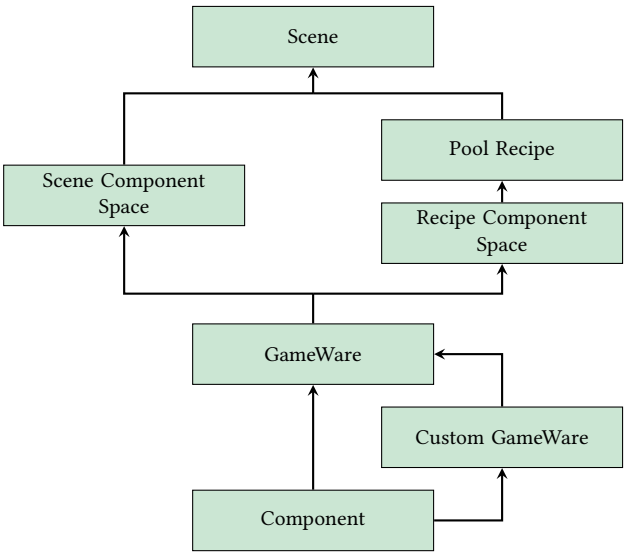


Fig. 2. A graph representing the *GameWare* interfacing system from the developer's perspective.

developers use GameWares, they conceptually give them components, however, the engine takes this action and repeatedly stores all components in the scene, where it will be handled by the engine. Following the pure component system of thinking, GameWares at their core are nothing more than a single ID.

With this basic ECS system in place, it becomes easy to create what we refer to as a component space. Component space is essentially where all components are stored. By separating the idea of a component spaces from a scene, we give way to the idea of storing information that we want to associate with a scene, but not initially use with it. In UBQR, this concept is known as *recipes*, similar to the idea of a *prefab* in Unity. Building upon these recipes we develop the engine's pooling system. Recipes are pooled through the idea that a scene and a recipe are both a component space, and so adding a recipe into the scene is as simple as adding one component space into the other

Finally, to give the developer full control over the entity component system, custom GameWares are established into the engine. GameWares are presented to developer as a class, by inheriting from this class, the user can override specific functions. In addition, these custom GameWares can be used in the pooling system.

3.2 Core Architecture

Since the engine is based upon and ECS architecture, all systems take advantage of functionality offered by it. Specifically with transform information. By default, GameWares are assigned a transform, this information is vital to many of the engine's subsystem such as the physics system or renderer.

The physics system works in a "ping pong" system with the GameWare transforms such that custom functions on the game dev side can update positions of objects in the physics system, which will then use physics computations to update GameWare positions and correctly sync them back up.

Functionality is developed to only capture systems that are needed. In the renderer `getModelTransforms()` allows quick access for the renderer to access the positions of each model to be rendered. A parent child system is utilized to allow the local space of GameWares to be used as the world space of another GameWare. Using `getWorldTransform()` then iteratively follows the hierarchy of matrix transformations upwards until it gets the world position.

The engine clock offers two time scales: engine delta time, which remains consistent, and game delta time, which can be adjusted by the developer to control gameplay speed.

A built-in save system with scoreboard and checkpoint support also allows developers to set up checkpoints, record times, and submit them.

Engine systems are divided into two loading phases: initialization loading and scene loading, to maintain an even amount of work across load screens. All non-GameWare setup is handled during engine initialization. When a scene is set to be loaded, each system then initialises based on the new component space scene data. By following this style, new systems are easily added to then engine.

For true customization, developers can create custom GameWares, inheriting form virtual functions called by the engine. Unlike the standard ECS model, each custom GameWare stays in memory while the scene runs. The engine keeps references to all custom GameWares in the current scene and calls their functions only if instances of that type exist. This type-based search supports the engine's pooling system, which allows multiple instances of the same custom GameWare. The engine manages these pooled GameWares so that minimal setup is needed, making them behave identically to a non-pooled GameWare to the developer.

3.3 Renderer

The engine internally uses two steps to initialize the renderer, one responsible for creating Vulkan resources and allocating memory pools that are required throughout the runtime; and a second step that assigns and creates scene-specific data, such as buffers and textures.

The per frame loop of the renderer is designed more linearly to accommodate synchronisation between multiple render targets in different render passes. The system flow is expressed in Figure 3. Object specific rendering and shadow mapping are the first stages in the pipeline as they use world geometry. The `RenderManager` class holds instances of all unique render pass classes and populates all world data and pipeline data that are then used in the renderer during initialisation.

We provide 2 potential light types, a point-light component via the GameWare system, and a directional light that can be set for a scene. The directional light is responsible for casting shadows and holds its colour information and orthographic projection details.

A set of renderer arguments are also provided that allow the developer to configure each effect during runtime along with a user interface that allows visualisation and testing of these effects. This include emission levels, kernel sizes for object specific stylisations, banding levels in 'posterise' filter, and more. As the renderer

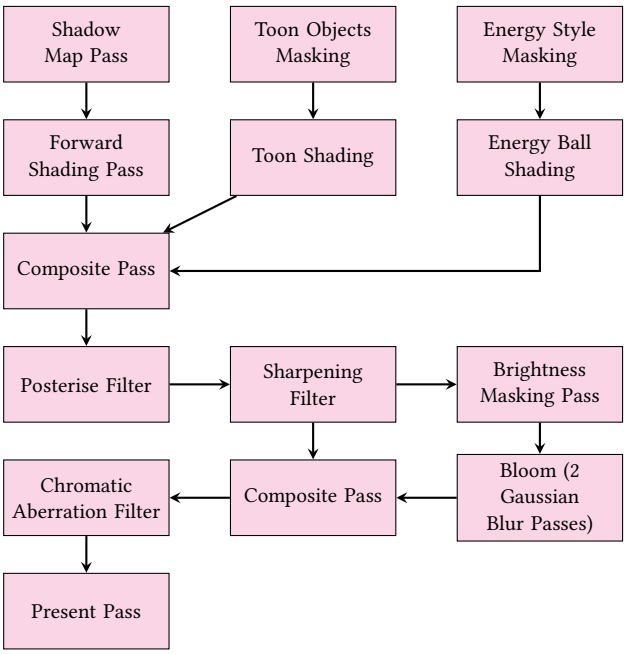


Fig. 3. The renderpass pipeline in the rendering system.

(a) Shadow Map (b) Shadows Rendered

Fig. 4. Shadow maps and shadows

provides two specialised object specific shaders, the developer has access to set them as `style` for a model component, which if unset, assumes default PBR rendering. Setting this to a particular style adds the masking and styling renderpasses to the scene and objects respectively. Some of the specifications of the renderer's features are mentioned further.

3.3.1 Shadow Maps. Shadows add depth to a scene and provide spatial relations between objects; and in rendering a highly stylised scene, perception is key [Eisemann et al. 2016]. We implement shadows for directional lights and this is done using a shadow pass that uses an orthographic projection from the direction of light and store only the depth information. This depth image is known as a shadow map as seen in Figure 4a. As we intend to make vehicular game engine, and projecting the entire scene in a singular image is infeasible, we use the player character as the centre of the projection, ensuring that shadows are always visible in the player's view. Using an orthographic projection helps with making sure that shadows from static object do not appear to be moving when the player moves.

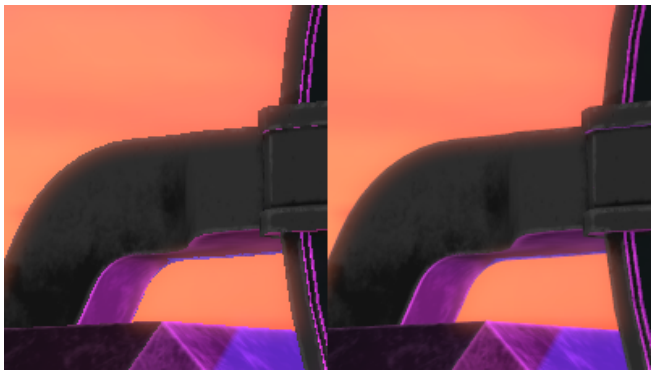


Fig. 5. Left: Aliased object. Right: MSAA enabled.

In the subsequent renderpass, we utilise built-in GLSL features to check whether a fragment is in shadow or not and utilize percent closer filtering (PCF) for softer shadows as seen in Figure 4b, although a design decision was mode to not implement additional PCF to suit the cyberpunk aesthetic.

3.3.2 Physically Based Rendering. A *Bidirectional Reflectance Distribution Function (BRDF)* describes how much light is reflected to the viewer when light makes contact with a certain material. In general, starting from the light equation:

$$\mathbf{L}_o = \mathbf{L}_e + \mathbf{L}_{\text{ambient}} + \sum_{n=0}^{N-1} f_r \mathbf{c}_{\text{light}} (\mathbf{n} \cdot \mathbf{l}_n)_+ \tag{1}$$

The equation describes the lighting in 3 components, the emissive light by $\mathbf{L}_e$, the ambient world light by $\mathbf{L}_{\text{ambient}}$ and f_r is the BRDF component. $\mathbf{n}$ and $\mathbf{l}$ denote the surface normal and direction to the light source respectively. There are many BRDF models each with their advantages and disadvantages, but we chose the Cook-Torrance model [Cook and Torrance 1981] primarily due to available documentation and known results for comparative analysis.

The engine uses the Beckmann NDF for the normal distribution function ($\mathbf{D}$) and the masking term from Cook-Torrence model for the Geometric Attenuation Function ($\mathbf{G}$) [Cook and Torrance 1981]. The specular reflection is given by the Fresnel term ($\mathbf{F}$), which is approximated using Shlick's Approximation [Schlick 1994]. We also use the Fresnel term to calculate the Lambertian diffuse ($\mathbf{L}_{\text{diffuse}}$) [Shirley 1992]. The resulting BRDF equation is:

$$f_r(\mathbf{l}, \mathbf{v}) = \mathbf{L}_{\text{diffuse}} + \frac{\mathbf{D}(\mathbf{n}, \mathbf{h})\mathbf{F}(\mathbf{l}, \mathbf{h}), \mathbf{G}(\mathbf{n}, \mathbf{l}, \mathbf{v})}{4(\mathbf{n} \cdot \mathbf{v})_+ (\mathbf{n} \cdot \mathbf{l})_+} \tag{2}$$

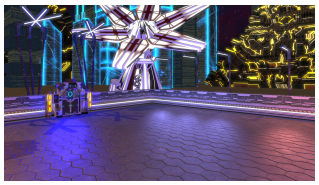
We also implement normal mapping at this stage by calculating the normal, tangent and bi-tangent (TBN) coordinate frame. We utilize this with the normal map texture to generate per-fragment normals for detailed shading with very little overhead.

3.3.3 Anti Aliasing. Aliasing is a result of limited number of pixels available for rendering at any given resolution, and produces jagged lines. We use the *Multi-Sampling Anti Aliasing* functionalities provided by Vulkan to correct this artifact [Khronos 2025]. We query the GPU for the maximum number of samples available, and use

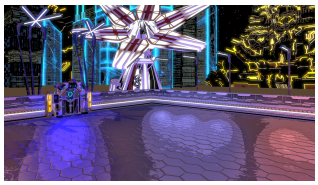
(a) Toon shaded smoke

(b) Energy ball bullets

Fig. 6. Object specific stylisations.



(a) Bloom with PBR



(b) Posterise Filter

Fig. 7. Bloom and Posterise post processing filters.

that in our *forward shading pass*, and resolve the multi-sampled images into a colour attachment. The result of this process are clearly visible in Figure 5.

3.3.4 Toon Shading. The engine provides a stylised single colour toon shader that can be used for specific objects like smoke. The effect requires two stages: firstly, in a geometry pass we colour each object to be shaded a unique identifiable colour. To ensure object occlusion we pass all object in the scene and discard the ones that are not to be shaded. In a second renderpass, we apply a custom filter using two square kernels κ_1 and κ_2 on the uniquely coloured objects, discarding the rest. The first stage of this filter applies the Gaussian function $G(x, y)$ of kernel κ_1 over all objects without distinguishing uniqueness.

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{3}$$

In the second stage we apply a user defined weight with offsets on each uniquely coloured object using kernel κ_2 and shade them with the desired colours. This step ensure a second stage of outlines amongst the objects themselves. The last step of this renderpass applies custom banding to the colours creating the toon shaded look as seen in Figure 6a.

3.3.5 Energy Ball Shading. Another stylised shading technique that the engine provides is the Energy Ball shading as it is inspired from blender assets of the same name [Asset 2024] and bullets from *Nier:Automata*. The first step for this effect is the same as for the toon shading. We mask and uniquely colour each object to be shaded. In the second step we use a kernel κ with radius $r = 5$ to evaluate object edges and calculating the number n of unique object in the kernel. We shade these objects with a custom model using a pseudo-random noise function $N(x, y)$ which is a deterministic random function in 2D [Vivo and Lowe 2025], and module the colour channels using time t.

$$\mathbf{L}_o = \begin{bmatrix} \frac{n}{4r^2} \cdot \sin(5t) \cdot N(x, y) \\ \frac{n}{4r^2} - N(y, x) \\ 0.1 \end{bmatrix} \tag{4}$$

$$N(x, y) = \mathbf{frac}(\sin(k_1 \cdot x + k_2 \cdot y) \cdot A) \tag{5}$$

where, k_1 and k_2 are arbitrary constants and A is an amplifier. The function **frac** provides the decimal part of the resulting value. The resulting objects can be seen in Figure 6b.

3.3.6 Posterize Filter. A posterize filter is a screen space filter that we provide the developer with, that reduces the number of unique colours in the screen [Lettier 2021]. It results in a comic book style look, and often looks similar to a cell shaded image. We use a banding levels parameter that the user can provide to calculate the number of unique values for colours in each channel. We quantize the colour value in a single channel greyscale space, and the resulting banding is later mapped back to RGB space. The resulting image is shown in Figure 7b

3.3.7 Bloom. Bloom is common technique that is used to convey the viewer of bright objects by making them glow. This is done via a simple Gaussian blur that results in the colours of the light source to appear to be 'bleeding out' [de Vries 2025]. We do this by extracting out all bright parts in the offscreen buffer using a threshold provided by the developer. Applying a single 2D Gaussian function is inefficient in this case, and we perform two Gaussian blurs with linear sampling [RasterGrid 2010].

The 2D Gaussian function as seen in Equation 3 can be split into two 1D Gaussian blurs:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}} \tag{6}$$

This blur is then applied to the buffer in two stages, once horizontally, and then the resulting images is further blurred vertically. The process is described by the equation:

$$B(x, y) = \sum_{i=0}^{r} G(i) \left(\sum_{j=0}^{r} I(x + i, y + j) \cdot G(j) \right) \tag{7}$$

where $I(x, y)$ is the input image pixel at (x, y). This image is then merged back with the original off screen buffer image to produce the intended effect (Figure 7a).

3.3.8 Chromatic Aberration Filter. Chromatic aberration is a screen space separates the colour channels at varying offsets. This is often used to simulate lens distortion or give an analog feel [Lettier 2021]. The engine provides the user with configurable offset multipliers and direction modes. The filter then simple offsets the position of each colour channel in the direction provided by the user.

3.4 Physics System

3.4.1 Jolt Physics Engine. Jolt's built-in vehicle constraint system supports raycast-based suspension and tuning of car dynamics such as friction, steering and wheel geometry. Allowing accurate and responsive vehicle behaviour while maintaining control over any needed customization. We organize the simulation jolt's central

PhysicsSystem, which manages all bodies, shapes, constraints, and simulation steps. Physical entities are represented as Body objects, created using BodyCreationSettings, where properties like shape, mass, motion type, and collision layers are specified. These bodies are controlled via the BodyInterface, allowing for runtime modifications and state queries. Shapes used for collision detection, such as BoxShape' or SphereShape, are attached to bodies at creation. Global settings like gravity and collision filtering are configured within the system, and multithreaded updates are handled via a user-defined JobSystem. Vehicles are supported through the VehicleConstraint system, which simulates suspension, wheel contact, friction, and torque using raycast wheels. Each wheel can be individually tuned for suspension stiffness, damping, friction, and steering.

3.4.2 Integration. The Jolt Physics engine is integrated into this project using a component-based architecture, where entities are defined by Transform, Rigidbody, Collider, and Model components. A central PhysicsSystem manages Jolt's internals, including the PhysicsSystem, BodyInterface, job system, and memory allocator. Entities are mapped to Jolt bodies using BodyCreationSettings, with physical properties such as mass, friction, and motion type derived from component data.

3.4.3 Collision Handling. For collisions, we used Jolt's BroadPhase interface which performs coarse detections against the world. GameWares are put into object layers which get mapped to their respective BroadPhase layers (Quadtrees) and events are triggered using our ContactListener. We also use Jolt's NarrowPhase collision interface which performs raycast to detect collisions for our camera.

3.4.4 Vehicle. Vehicles use Jolt's VehicleConstraint system, with wheel parameters suspension, damping, torque, and steering configured via WheelSettingsWV based on tagged wheel components. A WheeledVehicleController governs vehicle movement and responds to player inputs like acceleration, braking, and drifting. The physics system updates in fixed steps, syncing body states back to entity transforms and triggering gameplay events on collision. This design enables modular and high-performance simulation.

Wheels are set up using model data from tagged wheel GameWares in the scene. Their bounding boxes are used to calculate radius, width, and relative position to the chassis. This data is stored in WheelData and used to configure WheelSettingsWV, defining suspension, steering, and placement in Jolt's VehicleConstraint. This allows wheel setups to align with model geometry and be positioned visually in the editor. Ensuring that the rendered wheels remain in complete sync with the wheels simulated by Jolt's vehicle constraint system, both in position and rotation.

3.4.5 Camera. The camera system follows the active player using a dynamic offset and has two modes. In Follow mode, the camera smoothly tracks the player's position and orientation, interpolating its rotation to stay behind the car while avoiding sudden flips. In top-down mode, it orbits above the player, adjusting its yaw gradually if the car turns away. The camera uses shape casting to detect walls and prevent clipping, adjusting its position with a spring-damped system for smooth collision avoidance. Changing between modes is interpolated to ensure seamless transitions.

3.5 Event Handler

The event system uses c++ function wrapping, based on the observer design pattern. Listeners are loosely coupled with the event callers, being able to change local information with listener functions. Passing a class along with the listener function and event type allows for this. Templating is then used with class inheritance to designate any new event with a unique ID for correct event handling. The event system is then based on a backlog queuing structure. Events are stored over the course of a frame within the main event queue. While events dequeue, subsequent event calls are placed into the backlog queue. The backlog is then dequeued once the main queue has been cleaned of events. Events are able to chain queues as a result of this, and so the two queues are called subsequently and once per frame. This means event dequeuing is better spread out across multiple frames during runtime.

The Event system is paired with the engine upon initialization to prepare for event calls from custom GameWares. Custom GameWares can be associated to a pool, such classes require additional information for their event in order to allow for unity styled programming. Events are used engine-wide to greatly reduce the use of per frame calls. Additionally, the event handler system is made public to the game developer, such that they can register listeners to engine defined events (such as onCollisionEnter), or establish custom events.

3.6 Input and Action Handling

UBQR uses an input system based on polling and event-handling, using GLFW for keyboard and mouse support and SDL3 for game controller input, including haptic feedback(rumble). Each frame, input events collected from GLFW and SDL are queued and processed through the engine's input manager. To ensure consistency across input types, all input signals are normalised to a 0–1 scale, where values represent the magnitude of an input (e.g., Analog stick position or key press state).

3.6.1 InputActions. The input system is designed to support a wide range of interactions, including core gameplay functions such as vehicle control, as well as developer-oriented debug actions. Input is structured around input actions. Each input action can be bound to multiple input sources—including keyboard keys, mouse buttons, or controller inputs—allowing for high accessibility.

3.6.2 Custom Key-mapping. Key bindings/Mappings are defined using lightweight InputAction components. These mappings can be modified at runtime, allowing for dynamic key remapping without restarting the application. This flexibility ensures that both players and developers can tailor control schemes to fit their preferences or hardware constraints.

3.7 User Interfaces

The engine uses the Dear ImGui library [Cornut 2025] for user interfaces, including text and image overlays. The menu creation and scoreboard systems leverage the library to provide an easy-to-use interface for creating custom UI. Image overlays are rendered similarly, with developers able to set positions and interaction details via the engine's UI sub-system.

Support for more detailed UI is provided via custom functions to place elements like text, buttons and sliders via customisable grids and tables or directly positioning on screen.

3.8 Audio Manager

UBQR integrates the SoLoud audio library [Komppa 2020], which supports both 2D and 3D spatial sound. As a result, the audio system enables 3D audio, allowing sounds to be positioned within the game world based on the listener's position, orientation, and distance. Additionally, the engine allows for the creation and management of audio playlists, enabling background music to be layered contextually during different gameplay states.

The audio system also includes light post-processing capabilities, allowing developers to fine-tune the sound of their assets. UBQR's audio system supports audio filters, volume adjustments, and panning—providing flexibility when implementing audio within a game environment.

3.9 Debugging Tools and Interface

The engine provides tools for debugging and scene inspection, which are essential for efficient development and iteration. The UBQR debug UI allows developers to perform scene transitions in real time, enabling switching between levels and test environments without restarting the engine. GameWare Entities are fully accessible via the debug interface. Developers can inspect a wide range of attributes, including unique identifiers, component states, and other metadata. This visibility into the internal state of the engine facilitates the rapid tuning and validation of system behaviours.

Developer tools to tweak different rendering effects are also provided in debug builds. The user can adjust parameters such as emission intensities, kernel sizes, and colour values for applicable effects. The engine also provides debugging renderers to visualise normals, base colours, material roughness, depth map, and masking pass results.

4 EVALUATION

We evaluate the engine across three major aspects: usability from a game developer's perspective; features commonly required by the genre and style; and the CPU and GPU performance associated with utilising the engine at its full potential. We conduct benchmarks using our game *Axiom Swerve*, and use the development process itself as a means of testing the engine's usability. The game is designed to provide a complete experience, including controller support, a HUD, audio effects, background music, and gameplay features. Benchmarking a game of this nature is ideal, as it utilises all the features the engine provides and is intentionally designed to satisfy multiple criteria mentioned in Section 2.

4.1 Axiom Swerve

The game we developed using the engine is an action-based car game dubbed *Axiom Swerve*. The game specifically takes advantage of the engine's rendering system, built-in pooling, and vehicle physics features. The engine's custom UI and menu system is utilised for all gameplay and menu interfaces, allowing for the display of text, images, bars, and menu buttons. The vehicle-centred gameplay uses the engine's built-in vehicle controller with customisable settings. Easy scene setup and switching are enabled by the engine's entity-component interfacing system, *GameWares*. Specific gameplay functionality and GameWare spawning are implemented using custom GameWares, coupled with the engine's pooling system—this allows easy control over pool sizes for any number and variation of custom-scripted GameWares. Stylisation is achieved through numerous rendering feature options, all accessible via custom GameWare scripting. The engine's scoreboard system is also utilised and inherently benefits from the engine's save system.

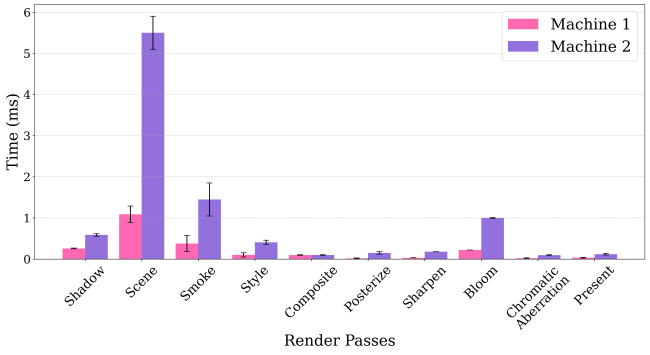


Fig. 8. Average time taken by each renderpass in the engine pipeline.

4.2 Performance

Profiled with two machines using the following hardware and software specifications:

Machine 1:

- OS: Red Hat Enterprise Linux 9.5 x64
- CPU: Intel® Core™ i7-12700 20 Cores
- Memory: 32GB
- GPU: NVIDIA GeForce RTX 4070

Machine 2:

- OS: Windows 10.0.19045 x64
- CPU: Intel® Core™ i5-10300H 4 Cores
- Memory: 8GB
- GPU: NVIDIA GeForce GTX 1650 Ti

As seen in Figure 8, our GPU performance evaluation measures the average render time for each render pass (or group of passes). The render times were averaged over multiple gameplay loops playing Level 1 of *Axiom Swerve*, with all render effects enabled on both Machine 1 and Machine 2. The smoke pass (responsible for tyre smoke) and the style pass (used for enemy bullet visuals) exhibit significantly greater variation during gameplay. These passes fluctuate by approximately 0.1 ms and 0.02 ms, respectively, on Machine 1, and by 0.2 ms and 0.02 ms, respectively, on Machine 2.

Figure 9 shows 1,000 samples of recorded data during the most intense section of the engine's showcase game. Different colours represent the various render passes, which together occupy the majority of each frame's time. One of these colours, however, represents the time taken by the CPU. Further analysis of the CPU

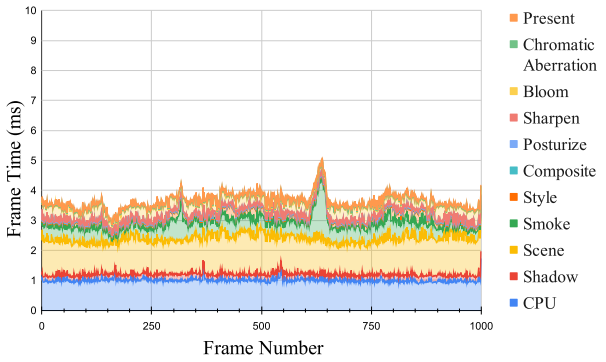


Fig. 9. A detailed analysis into per-frame data during play-testing for the final level in the game *Axiom Swerve*.

section shows that over 95% of this time is spent within the physics update function, and as such, we chose not to subdivide this section further. The final level features several types of GameWares, including roughly 200 physics bodies and over 300 models. This performance data was collected using Machine 1. As shown, the frame time remains under 5 ms for the entire duration, except for a single spike. This spike results from gameplay activity during the capture window—specifically, when the player hits an enemy and 20 smoke particles are spawned, increasing the GPU load. Despite this, the frame time consistently supports a target frame rate of over 100 frames per second.

5 DISCUSSION

Throughout the development of the *UBQR* engine, a major goal was to build a game engine tailored for vehicular gameplay and stylised rendering. Our results reflect a largely successful execution of this vision, though there were several trade-offs, limitations, and learning points along the way. By developing our own game, *Axiom Swerve*, we were able to experience the strengths and weaknesses of the *UBQR* engine from the perspective of a game developer. However, a limitation of this evaluation is our inherent familiarity with the engine, having developed it ourselves. As a result, we are unable to objectively assess its usability from the perspective of a developer with no prior exposure to the system. Regardless, our deep involvement in the engine's development allows us to provide meaningful insight into its effectiveness and capabilities.

5.1 Performance Analysis

The performance benchmarking results discussed in section 4 give us some insight into the engine and its rendering pipeline. From the benchmarking result in Figure 9, we see that the engine performs significantly better than a frame budget of 100fps (10ms/frame). This profiling was done on a relatively powerful machine, and running the game on machine 2 still provided a consistent 60fps gameplay. This was further optimised by turning certain effects such as smoke to a smaller kernel size.

Figure 8 shows individual render times for different renderpasses, and we can see that the scene, smoke, and style renderpasses vary the most. This is expected for a few reasons. Starting with the scene pass, the renderer currently does not perform a depth pre-pass or deferred shading and hence over-shading calls impact the performance there. Hence the fluctuation occurs due to the number of occluded objects in the view frustum. During development, this was observed but de-prioritised as we are well within our frame budget.

As for smoke and style passes, their variations are a result of the gameplay loop. Smoke rendering is triggered when enemies die, spawning multiple new particles each time. This increases the computational load for smoke shading, which is already handled by a 2D kernel-based shader. This can be optimised by separating the Gaussian function into two linear sampling Gaussians making it more efficient. The style renderpass is less expensive so variations are comparatively low, but as it is used for the bullets that enemies shoot, more bullets on screen increases calculations for these. The depth pre-pass should optimise overall render time, as ideally, when a bullet is occluding part of the scene, calculations for the occluded areas would be skipped.

The CPU performance is considerably faster, physics updates and collision detection being the only significant contributors to the frame time.

5.2 Gameplay Analysis

The development of Axiom Swerve was highly successful. Built around a cyberpunk/sci-fi aesthetic, the game effectively leveraged the engine's rendering capabilities through the custom GameWare interface. Development proved efficient and intuitive. Within just two hours, we progressed from an empty level to implementing a basic enemy that fires projectiles at the player. Within two hours after that, multiple enemies were spawning into the level, each with a death particle explosion trigger. This was easily achieved using the engine's effect pooling system, which provided a simple implementation process. On top of this, access to engine features generally required minimal effort, with the majority of features needing only one line of code to make changes. The UI options, including buttons, images, and text, were sufficiently versatile for the gameplay experience, and at no point did we feel limited by missing features. Finally, the engine's scene-switching capabilities enabled smooth level progression between the main menu and the opening level. The integrated scoreboard, tied to the save system, allowed scores to be conveniently viewed between levels. During a project poster session, where dozens of play-testers attempted the game, we received very positive feedback, specifically towards the game's overall style, which further affirmed its success.

5.3 Feature Evaluation

UBQR's architecture was built around a modular entity-component model, with the GameWare system at its core. By treating all entities as configurable GameWares composed of components, the engine promoted clean separation of logic and easy reuse across scenes. This design allowed new behaviours to be added without modifying existing systems, enabling scalable development. GameWare recipes further improved workflow by letting developers define and reuse common setups. Although this approach required upfront

planning, it resulted in a highly flexible foundation that supported rapid iteration.

A key benefit of this modular structure was seen in the physics system. Rigidbody and Collider components allowed developers to assign physical properties and collision shapes to GameWares with minimal friction. Collider types such as Sphere, Box, and Mesh, combined with layer-based filtering, gave developers control over how objects interacted. While the system worked well overall, the lack of a real-time physics visualiser made debugging edge cases—like vehicles on uneven terrain—more difficult. Integration of Jolt's debugger was considered, but performance limitations made it impractical for this version of the engine.

Entity reuse and performance were further supported by the pooling system, which paired naturally with the GameWare architecture. It was especially useful in gameplay scenarios involving frequent object spawning, such as enemies firing projectiles. However, the system's limited flexibility for customising pooled objects occasionally required workarounds, suggesting room for refinement in future iterations. To support interaction between systems without tight coupling, an event system was implemented. It allowed GameWares and subsystems to respond to collisions, input events, and other triggers without direct dependencies. This reduced polling overhead and improved maintainability. The system remained reliable throughout development and was a simple but effective solution for cross-system communication.

The engine's development experience was enhanced by a GUI system combining Vulkan 2D rendering with Dear ImGui. This hybrid approach enabled both polished in-game UI and a powerful developer interface. Gameplay elements such as health bars and scoreboards were clearly rendered, while the ImGui debug tools allowed real-time inspection and editing of GameWare data. Developers could view component values, adjust parameters, and switch scenes without restarting, which significantly improved iteration speed. Though ImGui's appearance is functional rather than stylised, it was well suited to its debugging role and proved essential during development.

Supporting tools also included a JSON-based save system designed for lap timing and racing scores. It served its intended purpose effectively, providing feedback for competitive gameplay. However, its limited scope meant it could not handle broader data types like settings or player state. For future versions of the engine, a more generalised save system would offer greater flexibility.

User input was handled through a unified system that supported keyboard, mouse, and gamepad devices. This allowed developers to write input logic once and apply it across platforms, simplifying both implementation and testing. Features such as analog input and runtime remapping made it easy to experiment with different control schemes, especially during gameplay tuning. The system also integrated smoothly with physics and UI, reinforcing its reliability and versatility across the engine.

Audio support, provided via the SoLoud library, added to the sense of immersion with spatial 3D sound, panning, and attenuation. Sounds could be positioned relative to in-game entities, and music was managed with playlist functionality for smooth transitions. While the system lacked advanced audio mixing features, it was lightweight and effective, fulfilling the project's audio requirements without issue.

At the core of the engine's visual output was a Vulkan-based renderer designed for flexibility and stylisation. It supported a modular render pass pipeline with features like shadow mapping, physically based rendering, and a range of post-processing effects—including toon shading, bloom, and chromatic aberration. These effects were adjustable at runtime, giving developers fine-grained control over the game's look and feel. One of the renderer's standout features was object-specific masking, which enabled hybrid styles by allowing different shaders to be applied to different GameWares. This was particularly effective for combining realistic and stylised assets within the same scene. Although Vulkan's low-level nature required careful management of resources and state, the performance and visual fidelity it enabled made the trade-off worthwhile.

6 CONCLUSION

6.1 Limitations and Future Work

The engine currently only uses model meshes in the `.obj` format, which is clearly a limitation. Due to this, the engine relies on a baking step to ensure all data from the files is sanitised, ensuring assets are consistent. It also reduces the chances of missing textures or incorrect materials at runtime. We intend to add further support to this by incorporating other file formats and adding a game file feature that links all assets together, directly being able to integrate with the GameWare system.

The method by which our physics system wraps Jolt functionality presents various challenges to a developer. Scene physics initialisation lacks complete customisability in an easy-to-use fashion. Combining two separate components for physics creates a situation where attempting to create the exact physics properties needed relies on understanding which combination of components results in which effect.

In its current state, the engine does not provide a customisable render graph. In order for a developer to add a new shader pipeline, they would require comprehensive knowledge of the engine's rendering system. We wish to improve this by first adding a customisable post-processing stage where the user has access to a commonly used set of `push constants` to perform custom effects. Secondly, a larger feature would be to develop and integrate a fully customisable render graph.

Since the engine's original aim was to target the development of bullet hell vehicular games, an animation system was not considered to be a priority. As a result, the engine does not support animations. However, a common feature in these genres of games is transform-based movement and patterns to give the appearance of animation, which the engine fully supports.

The engine's save system was created for the sole purpose of storing scoreboard information. As a result of this, the engine was never designed to save any other form of information. This is a clear limitation of the system, as the feature provides no customisability in this regard. Improvements in this feature could include providing the user with an interface to inform the engine exactly what to save.

6.2 Summary

UBQR was intended to be a highly stylized engine with the ability to produce gameplay that builds on the idea of using style to enhance the gameplay experience. We intended to provide the user with an easy to use interface for all our engine's subsystems, giving them the ability to have more flexibility when programming games within the engine. Design for the engine was inspired by a unique multi-genre experience intended to provide a bloat-free and feature full development process. Research in the target genres showed an existence of inefficient processes, complicating the goal of achieving a specific game direction. Our engine presents an attempt to provide style specific features on a modular and performant architecture. UBQR successfully combines genres as its foundation for game development, affording the developer a heavily stylized engine with vehicular based gameplay which can handle large-scale complex and dynamic environments.

ACKNOWLEDGMENTS

We would like to thank our supervisor Dr. Markus Billeter for his invaluable support throughout this project.

REFERENCES

Artistic Asset. 2024. Procedural Energy Ball – Blender material. https://artisticasset.com/downloads/procedural-energy-ball-blender-material/

Adam Bankhurst. 2021. Rocket League Will Move to Unreal Engine 5, But It's a 'Long-Term Project'. *IGN* (2021). https://www.ign.com/articles/rocket-league-unreal-engine-5-long-term-project-ue5

Bloober Team. 2017. *Observer*. Aspyr Media. https://www.blooberteam.com/observer-redux

Bugbear Entertainment. 2018. *Wreckfest*. THQ Nordic. https://wreckfest.thqnordic.com/

CardinalSpawn. 2020. Gaming Spotlight: Rocket League – The Success Story. *ESports Wales* (2020). https://esportswales.org/rocket-league-success/

Robert L Cook and Kenneth E Torrance. 1981. A reflectance model for computer graphics. *ACM Siggraph Computer Graphics* 15, 3 (1981), 307–316.

Omar Cornut. 2025. Dear Imgui. https://github.com/ocornut/imgui

Erwin Coumans. 2025. *Bullet Physics SDK User Manual*. Accessed: 2025-05-21.

Criterion Games. 2022. *Need for Speed Unbound*. Electronic Arts. https://www.ea.com/games/need-for-speed/need-for-speed-unbound

Joey de Vries. 2025. Learn OpenGL. https://learnopengl.com/

Elmar Eisemann, Michael Schwarz, Ulf Assarsson, and Michael Wimmer. 2016. *Real-Time Shadows*. CRC Press.

Epic Games. 2025a. Components. *Unreal Engine* (2025). https://dev.epicgames.com/documentation/en-us/unreal-engine/components-in-unreal-engine

Epic Games. 2025b. Physics in Unreal Engine. https://dev.epicgames.com/documentation/en-us/unreal-engine/physics-in-unreal-engine. Accessed: 2025-05-21.

Epic Games. 2025c. Unreal Engine - We make the engine. You make it Unreal. https://www.unrealengine.com/

Epic Games. 2025. AActor. https://dev.epicgames.com/documentation/en-us/unreal-engine/API/Runtime/Engine/GameFramework/AActor

Ghost Games. 2019. *Need for Speed Heat*. Electronic Arts. https://www.ea.com/games/need-for-speed/need-for-speed-heat

Godot. 2025. Node - Godot docs. https://docs.godotengine.org/en/stable/classes/class_node.html

Jason Gregory. 2018. *Game Engine Architecture, Third Edition* (0 ed.). A K Peters/CRC Press. https://doi.org/10.1201/9781315267845

Toni Härkönen. 2019. Advantages and Implementation of Entity-Component-Systems. *Tampere University, Tampere, Finland* (2019), 6.

Chris Inglis. 2025. Observer Review – An Intense and Engaging Sci-Fi Horror. *Roundtable Co-op* (2025). https://roundtablecoop.com/reviews/observer-review-an-intense-and-engaging-sci-fi-horror/

Ariel Manzur Juan Linietsky. 2025. DirectionalLight3D. https://dev.epicgames.com/documentation/en-us/unreal-engine/API/Runtime/Engine/GameFramework/AActor

Khronos. 2025. Vulkan Documentation. https://docs.vulkan.org/spec/latest/index.html

Jari Komppa. 2020. SoLoud. https://solhsa.com/soloud/

David Lettier. 2021. 3D Game Shader for Beginners. https://lettier.github.io/3d-game-shaders-for-beginners/index.html

Oliver Mackenzie. 2022. Need for Speed Unbound tech review - profound improvements, bold artistic touches. *EuroGamer* (2022). https://www.eurogamer.net/digitalfoundry-2022-need-for-speed-unbound-tech-review-profound-graphical-improvements-with-bold-artistic-touches

Main Leaf Games. 2024. What game engine does Nintendo use? https://mainleaf.com/what-game-engine-does-nintendo-use/

MCV Develop. 2021. Behind the art of Hades: "We value artistic integrity and excellence in artistic craft at Supergiant, however we're first and foremost a game design-led team.". https://mcvuk.com/business-news/behind-the-art-of-hades-we-value-artistic-integrity-and-excellence-in-artistic-craft-at-supergiant-however-were-first-and-foremost-a-game-design-lead-team/

Ozzie Mejia. 2015. Rocket League preview: gas-powered goals. *ShackNews* (2015). https://www.shacknews.com/article/89592/rocket-league-preview-gas-powered-goals

Mike Minotti. 2021. Forza Horizon 5 review — Graphics matter. *VentureBeat* (Nov. 2021). https://venturebeat.com/games/forza-horizon-5-review-graphics-matter/

Mathieu Muratet and Délia Garbarini. 2020. *Accessibility and Serious Games: What About Entity-Component-System Software Architecture?* Springer International Publishing, 3–12. https://doi.org/10.1007/978-3-030-63464-3_1

Nintendo. 2017. *Mario Kart 8*. https://mariokart8.nintendo.com/

NVIDIA Corporation. 2025. PhysX 5.1.0 Documentation. https://nvidia-omniverse.github.io/PhysX/physx/5.1.0/index.html. Accessed: 2025-05-21.

Ovid. 2021. The Unknown Design Pattern — dev.to. https://dev.to/ovid/the-unknown-design-pattern-1l64. [Accessed 21-05-2025].

Platinum Games. 2025. NieR:Automata. https://www.square-enix-games.com/en_GB/games/nier-automata

RasterGrid. 2010. *Efficient Gaussian blur with linear sampling*. https://www.rastergrid.com/blog/2010/09/efficient-gaussian-blur-with-linear-sampling/

Dawid Ratajczyk. 2019. Uncanny Valley in Video Games: An Overview. *Homo Ludens* 1 (12) (Dec. 2019), 135–148. https://doi.org/10.14746/hl.2019.12.7

Rockstar North. 2013. *Grand Theft Auto V*. Rockstar Games. https://www.rockstargames.com/gta-v

Jorrit Rouwe. 2025. Jolt Physics. https://github.com/jrouwe/JoltPhysics. Accessed: 2025-05-21.

Mark Saroufim. 2020. Entity component system is all you need? https://marksaroufim.medium.com/entity-component-system-is-all-you-need-884f972bd867

Christophe Schlick. 1994. An Inexpensive BRDF Model for Physically-based Rendering. *Computer Graphics Forum* 13, 3 (Aug. 1994), 233–246. https://doi.org/10.1111/1467-8659.1330233

Scott. 2018. Defining the Illusive Nintendo Polish. *Two Button Crew* (Jan. 2018). https://twobuttoncrew.com/2018/01/07/defining-the-illusive-nintendo-polish/

Peter Shirley. 1992. *Physically Based Lighting Calculations for Computer Graphics: A Modern Perspective*. Springer Berlin Heidelberg, Berlin, Heidelberg, 73–83. https://doi.org/10.1007/978-3-662-09287-3_5

Silicon Studio. 2021. Silicon Studio's "Enlighten" global illumination adopted by "NieR Replicant" from Square Enix. https://www.siliconstudio.co.jp/en/news/pressreleases/2021/en2104nier/en2104nier.html

Silicon Studio. 2025. *Enlighten: Real-time global illumination for all platforms*. https://www.siliconstudio.co.jp/middleware/enlighten/en/

Kenzo ter Elst, Daniele Vettorel, Allan Bentham, and Mihnea Balta. 2025. Game engines and shader stuttering: Unreal Engine's solution to the problem. *Epic Games* (2025). https://www.unrealengine.com/en-US/tech-blog/game-engines-and-shader-stuttering-unreal-engines-solution-to-the-problem

The Forge. 2020. 2020 - A Retrospective. https://theforge.dev/2020-a-retrospective/

The Forge. 2025. The Forge. Creation. Evaluation. Expansion. Optimization. https://theforge.dev/

tvtropes. 2025a. Bullet Hell. https://tvtropes.org/pmwiki/pmwiki.php/Main/BulletHell

tvtropes. 2025b. Unintentional Uncanny Valley. *tvtropes* (2025). https://tvtropes.org/pmwiki/pmwiki.php/UnintentionalUncannyValley/VideoGames

Unity Technologies. 2024. Unity Manual: Physics. https://docs.unity3d.com/Manual/PhysicsSection.html. Accessed: 2025-05-21.

Brittany Vincent. 2017. Observer Review – A Mindbending Mixture of Cyberpunk and Horror. *Game Revolution* (2017). https://www.gamerevolution.com/review/344911-observer-review-mindbending-mixture-cyberpunk-horror

Sergio J. Viudes-Carbonell, Francisco J. Gallego-Durán, Faraón Llorens-Largo, and Rafael Molina-Carmona. 2021. Towards an Iterative Design for Serious Games. *Sustainability* 13, 6 (2021). https://doi.org/10.3390/su13063290

Patricio Gonzalez Vivo and Jen Lowe. 2025. *Generative designs*. The Book of Shaders. https://thebookofshaders.com/10/

Mike Williams. 2019. How the Frostbite Engine Became a Nightmare for EA in General, and BioWare in Particular. *VG247* (2019). https://www.vg247.com/how-the-frostbite-engine-became-a-nightmare-for-ea-in-general-and-bioware-in-particular

Benjamin Wolf. 2017. How NieR: Automata masters two genres. *Medium* (Nov. 2017). https://medium.com/@BWolf1989/how-nier-automata-masters-two-genres-cdcb556e0fb5

A PROJECT MANAGEMENT

A.1 Supervisor and Theme

At the beginning of the project we were assigned Dr. Marks Billeter as our supervisor. At this point we had already been floating some ideas around as to the direction we wanted to go in, but Dr. Billeter's large amount of experience with Vulkan assisted us in the final decision that we should focus our efforts in the direction of interesting visualisations using Vulkan. At the top of the list of games that emphasize visualization are vehicular-based games, which often focus on enhancing the appearance of inanimate objects through advanced rendering techniques. A vehicle based game engine suited the members of the groups ideals very well, for three main reasons: Firstly, as aforementioned, it is a great game genre to focus on visualisation; secondly, a large amount of the group had a particular interest in physics, which tends to be one of the other main focuses of vehicular based games; and thirdly, vehicle games often only include limited animation, since most of the movement is in the cars wheels which can be programmatically animated, which keeps the scope of the overall project down.

Meetings throughout the project were kept relatively structured and were held every Friday afternoon. Since every group member was only timetabled for one other module that semester, it was easy to lock down this schedule as there were no timetabling conflicts. We felt once a week was a good frequency to have these meetings since the open time in our timetables allowed for a lot of potential progress even within a single week. Additionally, to prepare for these meetings, we would compile a power point presentation to efficiently convey information of the progress that was made for that week. We found this power point to be highly useful since the quick transfer of information gave us more time to ask questions and even discuss highly specific problems we were solving within the engine. As we neared the end of the project, these meetings became less frequent, but overall proved invaluable, as having a supervisor with a large amount of experience in the area helped guide us in the right direction.

A.2 Group organization

A.2.1 Sub-teams and Work Distribution. A project of this scope needs good methodologies, the approach we took to this project mainly relied upon agile methodology with emphasis on the use of Kanban for progress tracking. Given the simplicity of the overall racing game design, we aimed to have a minimum viable product of a car that could drive through checkpoints and record a players time. The thought behind this was to subsequently employ a more iterative approach as it is a better approach for game development [Viudes-Carbonell et al. 2021]. Once the initial ideas were consolidated upon and the group was ready to begin development, we split ourselves in to multiple teams to efficiently setup the project without conflicting overlap between teams:

Tejaswa	Build-system & Renderer
Alex	Core architecture/ entity-component system
Michael, Sahil	.obj file reading & assets manager
Abhigyan, Sahil	Physics system implementation/ integration

Reflecting upon this initial distribution, we believe we were incredibly successful. The main benefits were that we were able to setup a guideline for all future engine implementations through a rigid architecture system while separately setting up integral engine resources and features that were later integrated into the core engine design. If these individual areas were completed all within the same project, the rearranging of core structures for the engine's architecture would have caused large issues for consistent programming among the other members.

However the flaw of this approach was attempting to prepare for the systems fitting into the architecture before having the chance to research the requirements of them. This would ultimately lead to some unnecessary programming. But despite this error, the approach proved well made in the end, as our combination of entity component system and *GameWares* enabled quick and easy engine and game development down the line. After these initial setup goals had been mostly established, we then began assigning group members dynamically as needed. The reason we didn't continue with sub-teams was due to the large number of different aspects that the engine would require, and so many we wouldn't have the man power to have more than one person working on the same section. However, whenever a group member was assigned to a task, an emphasis was placed on that member being assigned to an area that was both close to their previous working area, and related to engine topics with which they felt most interested in undertaking. Using this method of task assignment, group members were able to make progress in areas where they would be able to make the most impact. This technique was efficient for the most part. Group members of different skill level were usually able to work where they felt most comfortable. But it is worth noting the downside of this method, which is that when task options were limited, some group members were forced to work on tasks with which they weren't completely comfortable. Discovering this issue as we progressed the project, we encouraged an environment where more confident members of the group would pair-program with members who felt stuck in the area they were assigned so that they could learn from and finish in that area as efficiently as possible.

A.2.2 Management. On top of the split of initial teams, we also decided upon two members to generally take charge of the group throughout the project (Alexander Claridge & Tejaswa Rizyal). For the majority of the project, these two members worked on opposite sides of the engine. They assigned tasks to group members who worked close to their respective sides of the engine while keeping constant contact with each other, keeping the sides of the engine working well with each other. Having this relationship of two pseudo-managers allowed the group to cope with the large amount of work that would come with the project while not placing too much responsibility on a single person such that they might be overwhelmed in such a position.

A.2.3 Means of Communication. Our team was initially setup through teams to gather contact information, but very quickly we migrated to Discord which remained our soul means of online communication throughout the project. Choosing discord was a good decision for the group since it enabled us to split all discussions in various different channels to keep communication streamlined. The separation of

multiple different work areas in channels also proved effective as the group encouraged progress updates within these channels. So whenever the group needed help or information regarding any specific area of the engine, their was always an point of access for enquiries, reading past decisions and accessing notes/power points from all in person meetings. It also proved very convenient, as everyone was already very familiar with the application and interface.

A.2.4 Meetings. Throughout the project, we had in person standups, these meetings would happen consistently on Mondays and often again before the Friday supervisor meeting. The Monday standups would set the group up for the week, giving time to communicate progress from the previous week. Additionally, due to different work styles, some members preferred completing large amounts of work over the weekend, and the Monday meeting allowed those members to discuss the changes they had made to members who would work at more conventional working hours. The meetings that would happen before the supervisor meeting would give the team a good chance to discuss the weeks accomplishments, and compile the weekly presentation for the supervisor meeting along with prepared questions regarding specific challenges we wanted our supervisors' advice on. During these meetings, a member of the group took minute recordings of the key notes. These minutes were especially useful during the supervisor meetings, as we were able to write down the questions we wanted to ask the supervisor beforehand so that we could efficiently note down the solutions we would receive during the meeting.

A.2.5 Version Control. We used GitHub for version control and to support our group methodologies through the Kanban board and project roadmap. Having all management resources within the same application provided a convenient single point of access for all organizational needs. Throughout the project, our group members consistently took advantage of the different areas of version control. New branches were created each time a member needed to add or adjust the codebase so broken changes would not impact other members of the group. Commit histories could be viewed to understand any changes that any member committed to the project, allowing members to build upon new changes without needing to enquire upon the specifics of the code. One member of the team created a Kanban board every weekend to plan for the upcoming week's sprint ensuring backlog tasks are being completed in time and every member has tasks assigned to them. With our methodology tools centralized in GitHub, the Kanban board was integrated with the Gantt chart, giving each group member a clear overview of the tasks due that week, as well as visibility into the broader goals and deadlines those tasks contributed to. The inherent difference in skill level and work rates for different members of the group presented a workload issue that was hard to avoid, however our constant usage of version control gave crutch to all members so that progress could constantly tracked and used as a point of discussion when discussing the weekly distribution of tasks.

A.3 Planning and execution

A.3.1 Initial Plan. Firstly, we set out with an initial planning of work as seen from the first Gantt chart in Figure 10. For our initial planning, we used members' individual interests to guide the split of work as well as the initial sub-teams that had been created. During this early stage, further planning was made for distribution for the rest of the project as members became more familiar with the assignment and the responsibilities that it would incur. Additionally, since two of the group members had specific learning disabilities, they knew that the group would be entitled to 2 weeks additional time for this project, and so this was accounted for with the Gantt chart.

A.3.2 Progression to the MVP. The second Gantt chart in Figure 10 represents the real progress of each member with respect to the different areas of the engine. Upon starting this project, we began working at a satisfactory rate in relation to our planned distribution. The aforementioned success of the starting sub-teams worked very well, we were able to immediately cover a lot of ground without much overlap. During this process, we ran into three main issues.

Initially our group consisted of a sixth group member who had joined slightly later into the project. We factored this member into our initial planning to do the audio manager. However, a few weeks later, this group member would leave the project, affecting our initial structure for workload distribution.

Over the first few weeks, one member of the team who had taken the responsibility to set up the build system became continuously responsible for integrating new libraries for each subsystem required. This was not ideal as this not only hampered their progress in other features but also started affecting their productivity and mental health. Due to the lack of interest from anyone else, the member was worried that the team would not be able to take over their workload. Due to intensity of the project at this time, this issue wasn't properly communicated within the team until around week 5/6 of the project. That member was then relieved from that area of work so that they could focus on areas that were their originally planned tasks. This was not an ideal situation, however, the problem was resolved before it lead to any detrimental issues.

For the last main issue, it became apparent that the physics system was taking longer to integrate into the engine than anticipated. On top of learning the intricate workings of the library system as a whole, the Jolt Physics library presented many decisions regarding our engine's ECS, requiring multiple consultations with the team lead in charge of architecture. However, we had already planned for this, our MVP deadline was much further ahead in the project, and physics wasn't a blocking task at this point in the project. So we added more time to the expected finish date of the physics, established an expectation for a higher volume of communication between the physics team and the rest of the team, and continued with progression of engine features that weren't blocked by physics integration. Unfortunately, the physics team ran into more problems that halted the production of the physics, namely vehicle physics. By this point, it was around week 8, and so the deadline for the MVP had become more pressing. The two team leads had been made aware of this situation, so more effort was transferred towards physics development. With the extra effort put towards accomplishing this task, our group was ultimately able to solve the fundamental issues blocking physics integration. There were improvements that could have made to the handling of this issue, specifically the time that

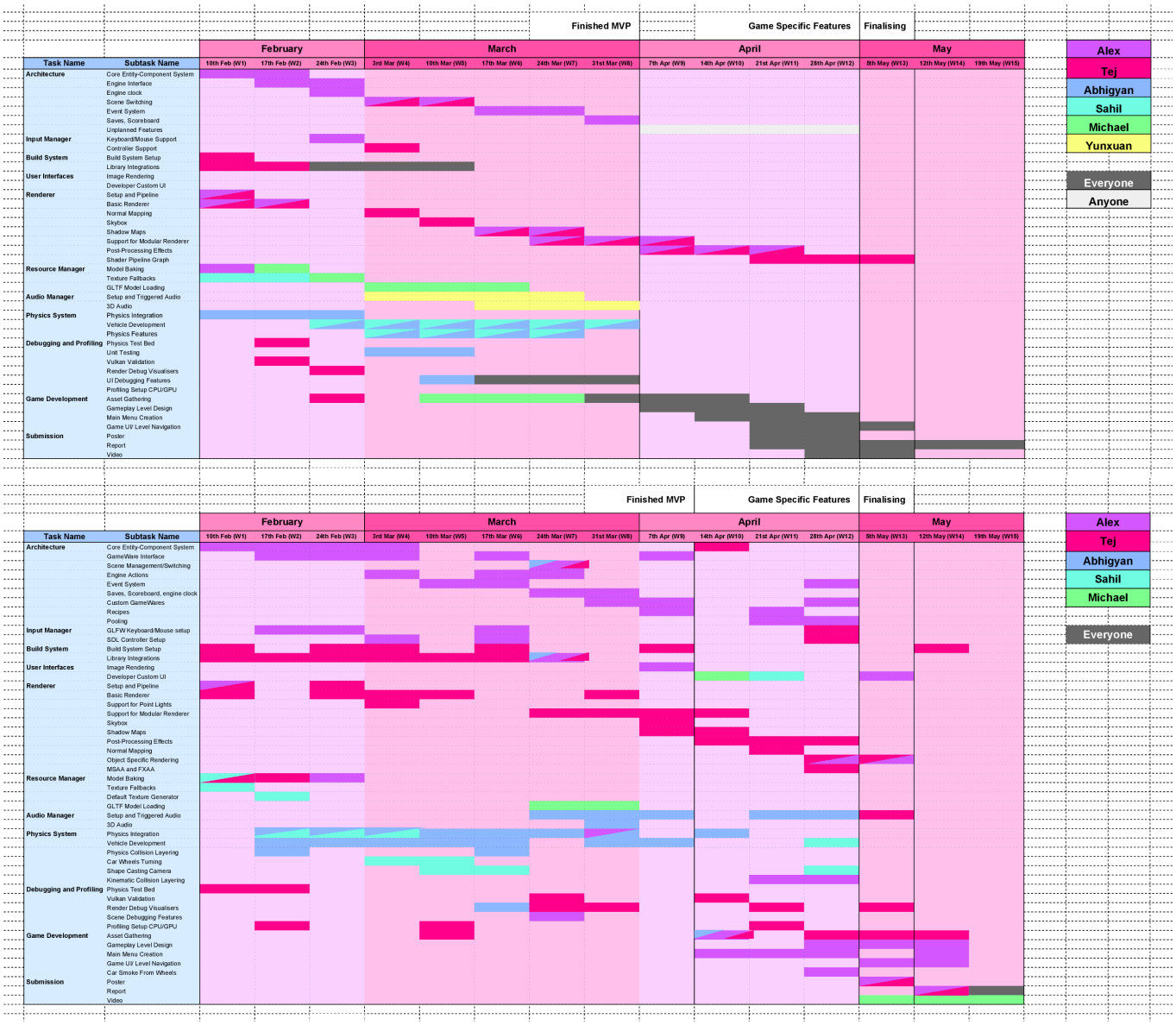


Fig. 10. Top: Original Gantt Chart and Bottom: Gantt Chart of the actual development process.

it was left unresolved for. The deadline for the MVP was set for week 9, and as mentioned, additional efforts were only contributed near the beginning of the week before. Regardless, the issue was resolved and the MVP was reached just before the set deadline. By this time, renderer modularity was nearing completion; we had fully controllable vehicle capabilities; and the player could drive through checkpoints to save their time to the engine's score-boarding system.

A.3.3 Completion for the Poster Session. In the latter half of the project, the group experienced a halt of communication, especially as April rolled around and different group members left at different points to return to their families for the Easter. This led to a large conflict of schedules, and unfortunately, meant that a large portion of the work load was ultimately placed upon the two team leads. This strain of work made it harder for the team leads to distribute workloads amongst the team and certain aspects of the engine suffered because of it. Most notably this is reflected in the GLTF Model Loading deliverable which the team was not able to complete for the final engine. While this was not ideal, the groups initial planning helped in this case once again, since the MVP was based upon a very achievable and simple idea of a car driving through checkpoints, GLTF was not a required feature so much as a good feature for convenience. Especially since we were not planning

to take advantage of any of the animation properties that GLTFs provide.

B SELF-REFLECTION AND SOFT SKILLS ACQUIRED.

A three month group project is an intense experience to undergo as a student given it contradicts the usual student experience where a student learns a subject, then completes a project in a short scale time. This was especially the case considering the little amount of other work to focus on outside the project. So due to its intensity, there was a large quantity of soft skills acquired by members of the team.

It is a very common piece of programming advice to make comments on your code, this way it is quickly understandable by members of the team that need to look through your code. But one lesser known reasoning as to why comments are useful, is so that one might understand their own code. Given this projects large scale, the code base that forms can be quite extensive, to the point where it is hard to keep track of everything all at once. And being able to read not just other group members' comments, but your own comments, proves invaluable while reciting different sections of the engine when needs justify expanding upon it.

In addition to writing comments, being able to read other members code and comments because a necessary skill to learn with the use of version control on the project. Once again, the large scope of the project meant that there was not enough time to discuss every single change that is added to the engine. But with version control, viewing commit histories becomes incredibly valuable. Reading the latest commits to understand specifics to features that could provide a member with additional functionality when programming their own code became a regular occurrence within the group. In addition, if an unforeseen bug arose in the current build, being able to look through commits to determine the source of the problem allowed for much increased rate at which bugs could be resolved. To further this skill, some members even learnt different version control functionalities. GitHub offers a useful command called git bisect, which proved invaluable when one team member noticed issues with the car handling after a period during which no one had intentionally modified related code. This command saved a significant amount of time that would have otherwise been spent reviewing each individual line change in potentially problematic commits.

The two team leads were able to reinforce their managerial skills, at different points throughout the project, multiple systems needed an overseer for its integration into the system. This was particularly true in the example of merging multiple large systems with the entity-component system architecture. During this merging process, efficient communication was essential between the team lead, who had implemented the architecture, and the other members who needed to understand it in order to integrate their systems with it. On top of this, the overall group learnt plenty about group communication, being able to efficiently convey information during group meetings would have a cascading effect, as goals would be able to formed quicker as a result, which lead to problem solving and efficient gathering of information such that we would almost always have new features and points of discussion ready for our supervisor during weekly meetings.

Maintaining the weekly deadlines was good practice for what would be standard workplace discipline. To add to this, by spending time every Monday discussing our aims for the week, we became highly adaptable to different situations. This was definitely the best course of action, given that none of the group members had created a game engine before, it allowed us flexibility in our goals as we discovered issues that we would not have been able to predict with our current knowledge, while also keeping the group to tight deadlines ensuring the constant output of work. It was especially useful to follow this methodology, given that in the gaming industry, it is often the case that the base product is constantly iterated upon rather than built up to a final goal. The only time our group was less successful was during the end of the project, the deadline was closer so we had all stopped having meetings and instead focused on getting more work done. Initially, we were able to produce more content, but it soon became clear that this came at a cost, the Kanban board was no longer being consistently updated, making it harder for the group to track the overall goal and the steps needed to achieve it. This reinforces how important it is to stick to established methodologies, even though the deadline was close and it felt as though we needed to get on with more work rather than update the board or have constant meetings.

Making good decisions in preparation for a project such as this early in the development progress is incredibly important. While there were plenty of systems that appeared to have no programming overlap, there were also plenty of systems that closely affected each other. By taking time to research game engine architecture and deciding upon a design for the entity component system as one of the first design choices for the project, we were able to prevent our group from needing to rewrite many of their systems. The decision to unify under a researched ECS early on also gave us confidence in future system design. We knew exactly how each new system should be integrated into our engine by clearly following this design structure. Many problems could have arisen if we did not unify as early as we did, and we believe part of our success has been due to making critical decisions like this as early as possible for many of the different engine features. This type of early decision making can be seen in many parts of our engine, such as deciding against animations. There are plenty of pitfalls in creating good animations which could have been detrimental to the projects polished outcome. This methodology extended even to the engine's visual style, where selecting a specific colour scheme made it easier to make consistent artistic design choices wherever needed.

While there were plenty of beneficial techniques that we learnt during the process of employing them, there were some points that we learnt through some of the pitfalls we fell into. For example, throughout the project, there was difficulty in some of the communication, the two team leads were not experienced managers, and often had a lot of work that needed completing for their section. This meant that often there was little time to discuss work in between meetings, and so keeping up with progress was limited to those meetings specifically when there could have been a benefit to more discussions to start the day. Then later in the project, when time got closer to April, different members would be away from

Leeds at different times. This inconsistent attendance, along with the slower rate of meetings set by the supervisor, partially broke the flow of the group. The group would meet up less and discuss different areas of progress less. Since the group had began to rely heavily on in-person meetings, switching to online communication proved difficult. The team leads struggled to consistently hand out the next work pieces that should have been completed, and because there were often group members who were away or busy from the project at the time, this created a less efficient environment where group members work rate was slower unless prompted by the leads. If we were to do this situation again, we would likely look at being better prepared for this sudden remoteness of the group. Better establishing online meetings where the team leads could have pointed out some current areas of the engine that were lacking, and taking time to explain during these meetings the newer areas that might be more difficult to understand so that less information was needed for a member at the moment of attempting to add to this system.

Finally, one of the most important soft skills the group learnt was the importance of listening to experienced supervision. The information provided to the group by the supervisor was incredibly beneficial in solving problems which are easier to understand only with experience in the area. For example, when the group was researching an event system, the supervisor provided valuable C++-specific insights, highlighting the advantages and disadvantages of different event system patterns and explaining how a simple event system could be implemented. By engaging in this discussion, the group was able to accelerate the process of event system integration, finishing the Event Handler within a week.

Overall, the techniques, methodologies, and management techniques employed in this project were clearly a success, as the final deliverable achieves almost exactly what the group set out to accomplish, roughly following the guidelines of the initial Gantt chart. There were certainly some failings as a group. Significantly, this was due to the breakdown of communication over the course of the project, caused by the rush to get more work done, rather than following the chosen methodologies all the way to the end of the project. Additionally, the slow decline of in person meetings caused members to not achieve the amount of work that they perhaps would have if they had been prompted within an in-person group setting. Despite this, at all times, there were at least a couple of members who were able to keep the flow of work and maintain the communication of the group. And so, while some initial goals set for the MVP were not achieved, additional goals which were not initially scoped were implemented as a result of the constant iterative approach to long-term goals and dedication to short-term deliverables. And at the end of it all, the group managed to produce what each member considers an impressive piece of work to be proud of.

C SUMMARY OF CONTRIBUTIONS

Alexander Claridge - Meeting Notes, Core Architecture, Entity Component System, Pooling System, Event Handling, Input Handling, External Libraries, Debugging Tools, In-Game User Interface, Game Design and Development (Axiom Swerve), Shaders (Smoke and Energy Ball), Art and HUD Design, Poster.

Tejaswa Rizyal - Build System and External Libraries, Project Management, Rendering System (Vulkan Pipelines, Scene Rendering, Object Specific Rendering, Post Processing Effects), Debugging Visualisations and Renderer Profiler, Audio Manager, Sound Design (Axiom Swerve), Poster, Report.

Abhigyan Gandhi - Physics System implementation and integration, Vehicle Setup, Initial Audio Manager, Report.

Sahil Puri - Physics System, Camera System, Health System, Wheel Sync System, Jolt Integration, Vehicle Handling, Model Baking, Report.

Michael Asiamah - Model Importing, Retopology, User Interface Customisation, Motion Graphic Design, Meeting Presentations, Concept Art, Engine Showcase Video, Report

Received 23 May 2025