

Abstract

This thesis aims to perform a comparative analysis of Unreal Engine and Unity, focusing specifically on their effectiveness in creating multimedia applications. The differences between the two platforms, particularly regarding their capacity to import, manage, implement, and combine media types into interactive elements, are analyzed. The analysis also covers the integration of ChatGPT 4o into development workflows and concludes that the AI chatbot has limited usefulness when working with blueprint scripts. Overall, the findings highlight that Unreal Engine is a promising platform, and its tools like Lumen, Niagara, and Sequencer allow for crafting immersive and visually impressive experiences. Unreal Engine's Blueprints system also makes it beginner-friendly and accessible for rapid prototyping. However, the engine's extensive feature set requires beginners to invest time in understanding its workflows, exploring its tools, and seeking out tutorials to fully take advantage of its potential.

Contents

Abstract	i
Contents	iii
1 Introduction	1
2 Related Work	3
2.1 Unity Engine	4
2.2 Unreal Engine	4
2.3 Comparison of Unreal and Unity	5
2.4 Meltdown Mayhem in Unity	5
3 Methodology	9
3.1 Basic Setup	9
3.2 Networking	12
3.3 Feature Library	14
3.4 Final Game	21
4 Results and Analysis	31
4.1 Challenges and Feature Comparison	31
4.2 AI Tools: ChatGPT's Role in Development	37
5 Conclusion	41
5.1 Implications for Students of the Multimedia Course	41
5.2 Final Thoughts	43
Bibliography	45
Ludography	47

CHAPTER

1

Introduction

Digital media has become an essential component of daily life, incorporating a wide range of media types such as text, audio, images, video, animation, and interactive content. The combination of these various forms has led to the creation of more advanced digital experiences, often merging and overlapping traditional media formats. The domain of developing applications that incorporate a variety of media types has come to be known as *multimedia development*. A leading example of this is video games, which make use of all types of media to deliver immersive and entertaining experiences. Interdisciplinary fields such as video game development and cinematography are also beginning to show some overlap. Some newer video games (e.g. *Hellblade 2* [GThe24] and *Death Stranding* [GPro19]) are starting to lean heavily into cinematic visuals and motion-captured acting performances, resulting in a conceptual merge of games and films.

Creating such complex digital experiences would be incredibly challenging without the use of game engines. Game engines are software frameworks that provide the tools and capabilities necessary to develop interactive applications, particularly video games. They allow developers to integrate and manipulate assets such as 3D models, textures, animations, audio, and physics systems in a cohesive environment. By offering features like real-time rendering, lighting systems, and scripting tools, game engines enable developers to focus on creative design rather than building foundational technology from scratch. Among the most widely used game engines today are Unreal Engine and Unity.

This thesis seeks to perform a comparative analysis of Unreal Engine and Unity, focusing specifically on their effectiveness in creating a multimedia application (video game). Even though both engines are versatile and support a wide range of media types and interactive content, major differences in their architecture, tools, workflows, and user experience may make one engine more suited for specific multimedia projects than the other. Thus, the aim of this study is to identify key differences between the two platforms, particularly regarding their capacity to import, manage, implement, and combine media types into interactive elements. The following research questions will direct the study:

1. INTRODUCTION

- What are the important differences between Unreal Engine and Unity concerning multimedia content creation?
- What are the implications of these differences, specifically in the context of students using Unreal Engine instead of Unity to implement game logic in future iterations of the Multimedia (188.912, UE) course at Vienna University of Technology?
- How do the engines manage the integration of different media types (such as video, audio, images, 3D models, etc.)?
- What are the specific features, advantages, and limitations of each engine in multimedia development?

Alongside this technical analysis, the thesis will also investigate the potential impact and helpfulness of AI tools, specifically *ChatGPT 4o*, on enhancing multimedia content creation within these engines. Recent developments in AI have shown their capability to support activities like code generation, problem-solving, and content enhancement. A secondary goal of this thesis is to assess whether AI systems can effectively aid in optimizing workflows, resolving engine-related challenges, and even participating in the creative aspects of multimedia development.

Another goal of this project and thesis is to aid in the conversion of the course Multimedia (188.912, UE) at TU Wien from Unity-based assignments to Unreal Engine. In order to find differences in time and effort for students completing future assignments, this thesis will be based on the migration of a completed Unity student project that was created during a past iteration of the course to Unreal Engine. This migration process consisted of multiple phases working inside Unreal Engine 5:

- Basic multiplayer setup
- Networked multiplayer setup
- Creation of a feature library
- Combination of the above to create the final game

2

CHAPTER

Related Work

In the field of real-time multimedia development, Unreal Engine and Unity stand as two of the most prominent platforms, offering robust solutions for a wide range of 2D and 3D applications including game development, virtual reality (VR), architectural visualization, and film production. Both engines provide comprehensive toolsets and features, enabling developers to create high-quality interactive applications. They are designed to support multiple platforms, from desktop computers and consoles to mobile devices, and have commercial licensing models that cater to both indie developers and large-scale studios.

In their thesis from 2016 [GL16] the authors Gora and Leibetseder describe differences in the content pipeline, differences in features and how to create a basic project in Unity 5 and Unreal Engine 4. According to their findings, Unreal Engine 4 is better suited for creating high fidelity AAA-titles, but can overwhelm beginners due to the large amount of available features. Unreal Engine's visual scripting was praised for being innovative. The greater amount of file types supported by the Unity engine for importing was described as unnecessary. Finally, the authors preferred Unreal Engine 4 and found it more intuitive to work with, after an initial period of familiarization.

It is important to note that both engines have changed greatly during the last eight years. And, while the thesis from 2016 [GL16] focuses more on editor functionality, content pipeline, and general comparison between older versions of the two engines, this thesis covers different aspects (specifically the creation of multimedia applications) in more detail and in a different context. The following sections roughly outline general facts and some basic differences between Unity and Unreal Engine while briefly mentioning each engine's history and use cases. Finally, the project which was migrated from the Unity engine to Unreal Engine 5.4 is described.

2.1 Unity Engine

Unity Technologies [UT2] released Unity in 2005 as an affordable, flexible game engine, initially focused on supporting indie developers and smaller studios. Unity's appeal lies in its versatility and ease of use, offering a more lightweight and user-friendly experience compared to Unreal. Unity's component-based architecture and the ability to script in C# make it accessible to many developers, from beginners to professionals.

Unity supports a large number of platforms, including Windows, macOS, Linux, iOS, Android, and consoles like PlayStation, Xbox, and Nintendo Switch. Its adaptability also extends to newer technologies such as AR, VR, and wearable devices, making it a great choice for cross-platform development. Unity's licensing model includes a free tier for individuals and small teams, with paid options like Unity Enterprise and Unity Pro providing additional features and services for larger projects and enterprises.

Unity has been embraced for its versatility, particularly in mobile game development and interactive 2D and 3D applications. It is widely used in the indie game scene, as well as in educational and research settings for prototyping and simulation. Unity's growth in VR and AR applications has made it a key player in industries ranging from healthcare to education, where interactive and immersive environments are increasingly in demand.

2.2 Unreal Engine

Unreal Engine [UE2], originally developed by Epic Games in 1998, first gained recognition with the release of the first-person shooter *Unreal*. It is widely praised for its high-end graphics capabilities. With the release of Unreal Engine 4 in 2014 and Unreal Engine 5 in 2021, the engine has continued to introduce new features such as real-time ray tracing and Nanite, a virtualized micropolygon geometry system that allows for highly detailed models with optimized performance.

The engine is written in C++ and is fully open-source, providing developers with the flexibility to modify its core systems to suit specific project needs. Unreal Engine's blueprint visual scripting system allows for fast prototyping and development, also making it accessible to developers or designers without advanced programming expertise.

Unreal Engine supports a wide range of platforms, including Windows, macOS, iOS, Android, Linux, and leading gaming consoles like PlayStation and Xbox, as well as emerging platforms such as VR and AR devices. Its licensing model offers a royalty-based commercial option, where developers are required to pay 5% of gross revenue after the first \$1 million USD earned per product.

With its focus on high-end graphics, Unreal Engine has also found usage in the film and television industries. The engine has been used for virtual production in shows such as *The Mandalorian* [GLE19], where real-time rendering allowed for dynamic and immersive set environments for the actors using large LED screens. Its use in architectural

visualization, automotive design, and realistic digital avatars (*MetaHuman*) combined with motion capture has continued to expand in recent years.

2.3 Comparison of Unreal and Unity

A substantial body of research has examined the strengths and weaknesses of Unreal Engine and Unity, offering various perspectives on their capabilities and suitability for various development scenarios. These studies collectively provide valuable insights into the engines' performance, usability, and applicability across different project types.

Dickson et al. (2017) [DBEK17] provide an education-focused comparison, emphasizing Unity's accessibility for beginners in a classroom setting. They conclude that both engines fit the criteria necessary for students to learn easily and to use them to build games in a limited time. The authors find that while Unity is a little less complicated to learn, Unreal can create more professional-looking results more easily.

Ciekanowska et al. (2021) [CKGD21] analyze the engines' efficiency in creating virtual exhibitions. Their findings reveal that Unreal's high-end rendering capabilities produce visually superior results with less work put into it. The authors also praised Unreal's blueprint system and variety of tools. However, Unity had the advantage in terms of performance, especially in RAM usage and CPU and GPU resource consumption.

Alexandrini et al. (2022) [AdFI22] evaluate the engines using usability heuristics, finding that Unity excels at preventing the user from having to use their memory for actions and error prevention, while Unreal performs better in user control and freedom, as well as flexibility and efficiency in use. However, the authors conclude that both engines are very similar overall while Unity is aimed at lay users and Unreal at professionals.

Soni and Kaur (2024) [SK24] compare Unity and Unreal Engine in terms of their learning curve, feature sets, and target projects. Unity is praised for its cross-platform capabilities, user-friendly interface, and extensive asset store, while Unreal is noted for its superior graphics performance, customization options, and Blueprint visual scripting system. However, Unity's performance limitations and Unreal's steep learning curve are recurring themes.

Together, these studies highlight how the choice of an engine depends on project goals, hardware constraints, and user expertise, aligning with this thesis' exploration of the engines' multimedia application capabilities.

2.4 Meltdown Mayhem in Unity

Meltdown Mayhem is a third-person two-player split-screen jump and run racing game for Windows machines. Players try to reach the goal before the other player while using power-ups to hinder the other player or to enhance their own avatars speed. The controlled player avatars are ice-cream cones that can walk, run and jump. The in-game

2. RELATED WORK

world players traverse consists of an indoor scene depicting a coffee shop with furniture and appliances that can be walked on, as can be seen in figure 2.1. One of the main focus points during development was to include interactive elements that can trigger multimedia events. Supported methods of controlling the player avatars are mouse and keyboard as well as controller.

Input Actions and Controls

- Movement (W, A, S, D keys or left control stick)
- Looking (Mouse movement or right control stick)
- Sprinting (Shift key or left trigger)
- Jumping (Spacebar or south button)
- Using power-ups (E key or right shoulder button)
- Joining the race (Enter key or north button)
- Pausing the game (Escape key or Start button)

Core Features

- Split-screen functionality for two player simultaneous interactive gameplay. Each half of the screen is filled with a players camera viewport.
- Collaborative interactions: both players can navigate the world and control their third person camera and their avatar at the same time. They can interact with each other.
- Player Avatar: each player is represented by a 3D model of an ice-cream cone and an attached third-person camera that can be controlled freely.
- Environment: the world which players are able to navigate is made up of an ice-cream shop / coffee shop. The world outside the house (Unity terrain with hills and trees) can be seen through the windows of the building. The sky is represented as a cubemap. The 3D models used to represent the environment are imported from the Unity store, or custom made for the project.
- Lighting: the world is lit by a directional light as well as multiple point lights and baked spotlights.
- Basic user interface (UI): players can navigate the application using a main menu, options menu and pause menu. Buttons for ‘Play’, ‘Settings’ and ‘Quit’ can be found in the main menu. The volume of the game can be controlled in the settings menu. Player names, the background image and the flavor of the ice-cream can be

changed in the settings menu. Players can freeze the game by opening the pause menu during the race by pressing ‘ESC’ or the ‘Start’ button on their controller. In the pause menu, buttons exist for continuing the race, returning to the main menu and resetting the game. A heads-up-display (HUD) based on the Unity canvas system tells players which power-up / item they possess and what lap they are on.

- Multimedia types:
 - Text
 - Audio
 - Video
 - Images
 - Particle effects
 - Animations

Additional Features

- Checkpoint system: a system to keep track of the players’ progress in the race.
- Out-of-bounds zone: when a player character touches this zone, the player is teleported back onto the racetrack.
- Item effects: players can pick up items during the race and use their effects.
- Improved character motion: by holding down the Sprint key or button players can control their character’s speed.

2. RELATED WORK

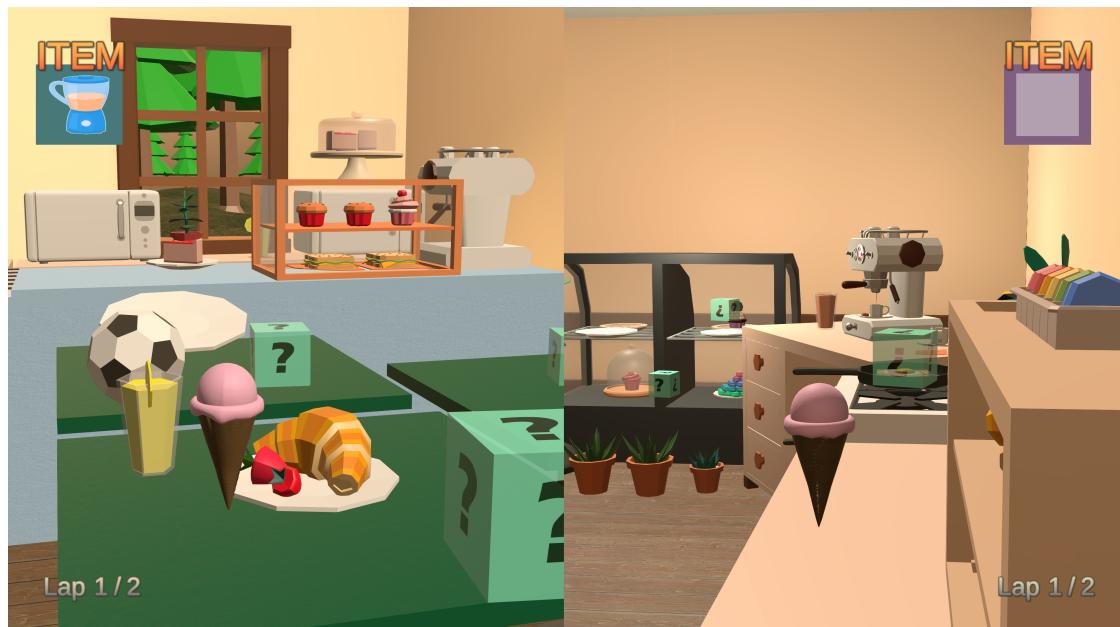


Figure 2.1: Screenshot of an ongoing race in Meltdown Mayhem developed with the Unity Engine.

CHAPTER 3

Methodology

This chapter presents the methodological approach employed to fulfill the objectives of this thesis, which is centered on a comparative analysis of Unreal Engine and Unity, with a particular emphasis on their capacity to handle multimedia content in video game development. The approach entails transferring a previously completed Unity-based student project from the Multimedia course at TU Wien to Unreal Engine, along with an evaluation of the challenges, time, and effort associated with this migration. The following sections document the migration process by providing a detailed description of the step-by-step process used to implement/migrate the project in/to Unreal Engine.

As mentioned previously, the migration process includes multiple stages. First, a basic multiplayer project was created in Unreal Engine 5. After that, networking features were tested. Next, the features of the original Unity game were implemented one by one in Unreal Engine, resulting in a feature library. Finally, the features were combined to create a completely migrated game. The following subsections describe and document the migration process in detail.

3.1 Basic Setup

Starting from a fresh Unreal Engine 5.4 installation, a new project was created using the *Third Person Template* that comes with the engine. The goal while creating the basic setup was to have a simple two-player split-screen prototype with keyboard and controller support. The gameplay of this basic prototype consists of three dimensional movement and controlling the player cameras.

Creating a player character blueprint class often functions as the starting point when implementing a video game in Unreal Engine from scratch. Unreal provides a working player input management system and templates for third-person player camera logic as well as others (first-person, top-down, etc.). Importing the 3D model for the player

3. METHODOLOGY

avatar in Unreal Engine is done by dragging the FBX-file into the content browser in the editor. Right-clicking the mesh inside the content browser and clicking *Convert to skeletal mesh* enables it to be used as the mesh for a character blueprint as well as to create bones and animations for the character. By using the provided third-person player character template and replacing the character skeletal mesh (figure 3.1), it was now possible to move around the default level as an ice cream cone. Texturing was done by importing visually suitable textures from the built-in *Quixel Bridge* [Qui] library and assigning a basic material with texture samples for albedo, roughness, and normals to the skeletal mesh.

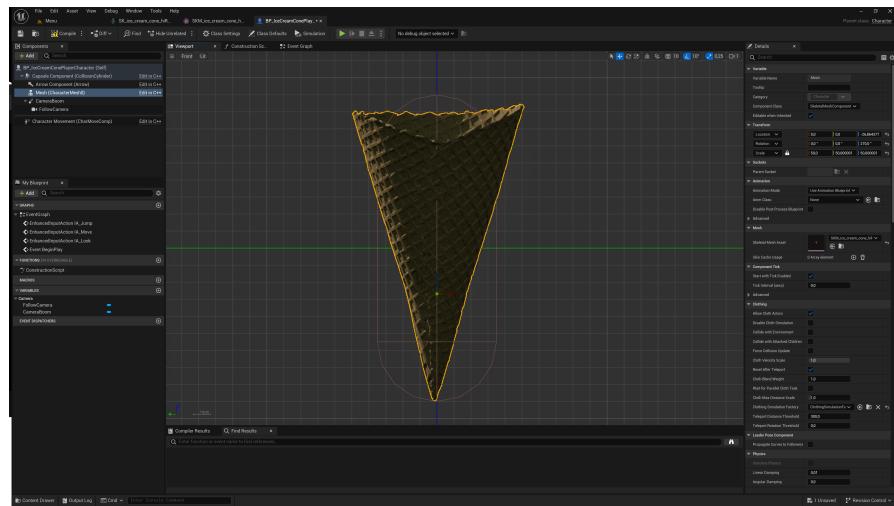


Figure 3.1: Frontal view of the Ice Cream Cone Player Character blueprint using an imported 3D model of an ice-cream cone.

Creating a walking animation for the player character was done in-engine using the skeleton created for the skeletal mesh, which in this case only has a single bone (root). Unreal Engine allows animating bone transforms using sequencer curves with key frames as seen in figure 3.2. The created curves and key frames were baked as an *Animation Sequence* asset. Next, *Blend Space* and *Animation Blueprint* assets were used to dynamically adjust the walking animation to the player's speed.

Since *Meltdown Mayhem* is a split-screen multiplayer game, enabling the built-in split-screen feature in the project settings was necessary. When using splitscreen the *Number of Players* setting in the *Play Mode and Play Settings* menu should be set to 1. In order for the engine to detect two separate players a *DualShock 4 controller* was connected to the PC as a second input device. To enable proper input detection from a keyboard and mouse, as well as from the controller *Input Action* assets and *Input Mapping Context* assets were created. By overriding the world settings of the level, custom game mode logic was used by creating a *Game Mode* blueprint class. Here, two players are created when the game is started. This process entails the following:

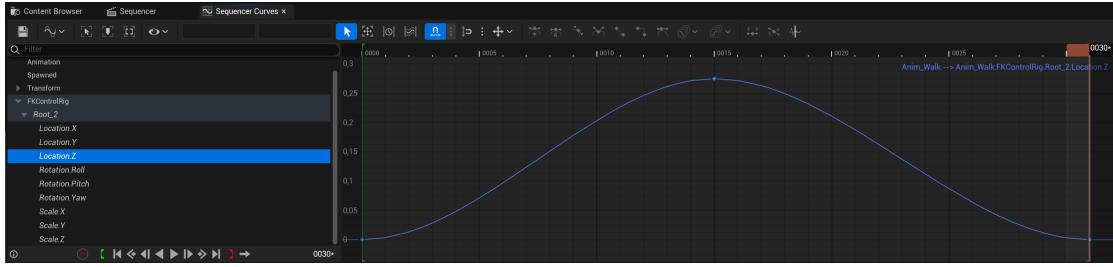


Figure 3.2: Unreal Engine’s built-in *Sequencer Curves* feature for creating animations using key frames.

- Creating two local players using the *Create Local Player* node
- adding the corresponding input mapping contexts to each of the created player’s *Enhanced Input Local Player Subsystem* using the *Add Mapping Context* node
- Spawning one instance of the Ice Cream Cone Player Character blueprint per player
- Possessing each character with the correct player controller

Finally, a static mesh representing a scoop of ice cream was imported and attached to the *Skeletal Mesh Component* inside the player character blueprint. The result is a simple split-screen prototype with keyboard and controller support where two players can look around, run, and jump simultaneously in the same world, as seen in figure 3.3.

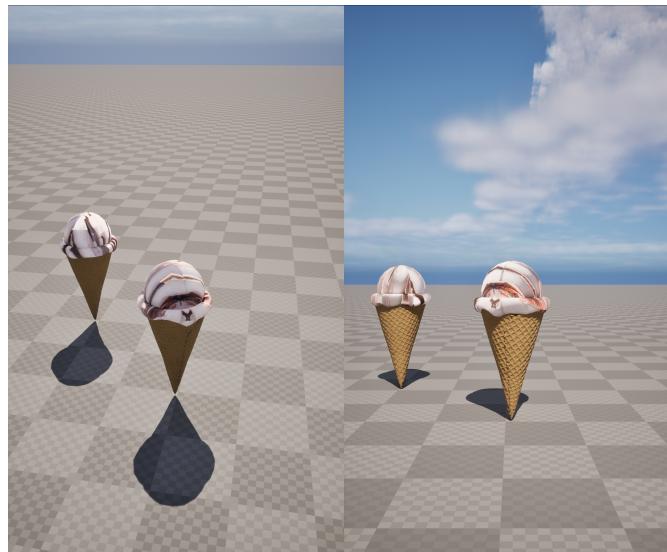


Figure 3.3: Screenshot of the working two-player split-screen prototype.

3.2 Networking

Since split-screen multiplayer is not always the best solution, for example, for students wanting to test their game frequently without having to physically be in the same place, this section covers the option of using networked multiplayer in Unreal Engine 5. The goal was to create a simple lobby-system using the built-in session functionality (*Online Subsystem*) of Unreal Engine. This way one player acts as the host by creating a session and the other player joins the existing session over the network.

First, a simple user interface functioning as a menu was created using *Widget Blueprint* assets including buttons and options for creating and joining sessions. Creating a session and a listening server in Unreal Engine is done by calling the *Create Session* node inside of a blueprint and opening a level with the *listen* option, as seen in figure 3.4. Respectively, sessions can be found with the *Find Sessions* node and joined with the *Join Session* node. Joining a session this way automatically opens the level of the server if the server is listening 3.5.

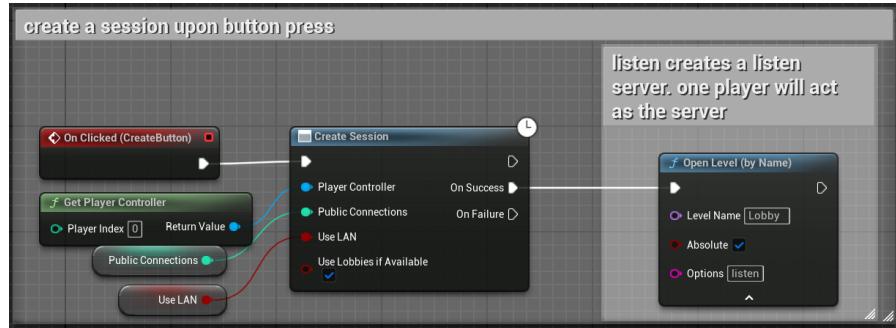


Figure 3.4: Blueprint script to create a server and a session in Unreal Engine.

To make sure the game states of the connected host and client stay synchronized Unreal Engine offers a variety of *Replication* settings. This allows, for instance, moving objects to be in the same location on both players' screens. The following settings were enabled inside the player character blueprint to ensure replicated player movement:

- Replicate Physics to Autonomous Proxy (skeletal mesh, static mesh and capsule components)
- Replicate Movement
- Replicates
- Component Replicates (static mesh component)

Replicated variables and game events are also supported in Unreal Engine sessions. Due to the host-client-relationship between the players it is necessary to handle events that

should happen for both players by using *Custom Event* nodes. This is because these types of nodes allow for being called either on the server or the client or on both (*Multicast*). Combining this functionality with the *Switch Has Authority* node enables one to execute code on the desired game instance, be it host or client. Variables that use the *Replication* setting stay synchronized between players after changing values.

Playtesting the networked multiplayer application inside of the Unreal Engine 5 editor is possible by setting the *Number of Players* to *2* in the *Play Mode and Play Settings* and choosing *Play In Editor (PIE)* mode. This way, two instances of the game are created and networking can be tested by creating a session with one of the instances and joining the session with the other instance.



Figure 3.5: Screenshot of the lobby with two players connected via the network.

When building and packaging a networking application that uses built-in sessions in Unreal Engine 5 it is important to include the *Online Subsystem* module in the build and to set the application to use this system as the default networking system. To achieve this, the following lines are needed inside the engine's configuration file *DefaultEngine.ini*:

```
[OnlineSubsystem]
DefaultPlatformService=NULL
[OnlineSubsystemNull]
bIsUsingNullSubSystem=true
```

The constructor inside the *<ProjectName>.Build.cs* configuration file was adjusted like so:

3. METHODOLOGY

```
PublicDependencyModuleNames.AddRange( new string[] {
    "Core",
    "CoreObject",
    "Engine",
    "OnlineSubsystem",
    "OnlineSubsystemNull",
    "OnlineSubsystemUtils" });

PrivateDependencyModuleNames.AddRange( new string[] {
    "OnlineSubsystem",
    "OnlineSubsystemNull" } );
```

3.3 Feature Library

After ensuring that the multiplayer functionality works as intended, the next step was to migrate the multimedia features of the Unity game to the Unreal prototype. This was done feature by feature, resulting in a project where each feature was independently implemented and tested. The goal was to be able to then use this library of features to create the final game without much difficulty in the next step. In this section, the implementation of each feature of the original Unity game in Unreal Engine 5 is documented.

3.3.1 Environment and Lighting

Creating an environment in Unreal Engine 5 generally consists of:

- Adding a skybox
- Generating a *Landscape* actor and adjusting its height values and material
- Adding foliage, meshes and blueprints to the level

When creating a new level in Unreal Engine from one of the templates a simple skybox is created automatically. It is possible to replace or adjust the skybox material to adjust the look of the sky. Generating a landscape and assigning a ground material took few clicks. Unreal Engine also offers brush tools for adjusting the landscape height values and adding trees and other plants by painting over the landscape actor. When creating a level from scratch the *Environment Light Mixer* window is useful, as it allows one to easily create all the necessary actors and manage them in one place.

After creating a new level from one of the provided templates, a directional light, a sky atmosphere, a skylight and atmospheric height fog were automatically set up. Before making changes to light intensity *Auto Exposure* was disabled inside a *Post Processing Volume* actor with unbound extent. This way, changes applied could be seen in real time

while adjusting the settings. Transforming the directional light resulted in changes to the location of the sun on the sky and realistic light colors due to atmospheric effects. By adjusting light scattering coefficients, fog density, light intensity and color settings the lighting and thereby the look of the previously created environment was modified.

3.3.2 Text

In the creation of multimedia applications using Unreal Engine 5, text-based elements serve as critical tools to enhance accessibility and to convey information. The engine provides a robust framework for managing text within 3D environments, user interfaces, and interactive scenes. The following key features were used to implement text-based elements:

- Text Render actors
- 3D Text Mesh plugin
- Text animations
- Captions
- Text elements in Widget Blueprints
- Custom fonts

Text Render actors allow one to place text within 3D space. This feature is optimized for scenarios that do not require complex mesh structures. Instead, *Text Render* actors produce flat 2D text that is often used for labels, user instructions, or other in-world annotations that need to maintain legibility regardless of scene complexity. *Text Render* actors can be customized with various parameters, including font selection, size, and color, as well as transformations like location and rotation.

Activating the *Text 3D* plugin allows the generation and manipulation of text as 3D meshes within the editor. This feature is particularly advantageous for multimedia applications that require immersive, contextually embedded text (e.g., signs, labels, or interactive information panels within 3D spaces). Text Meshes are constructed as polygonal models, which enables them to interact naturally with environmental lighting, shaders, and other physical effects present within UE5. Moreover, 3D Text Meshes support materials and are compatible with UE5's high-fidelity rendering technologies such as Lumen, enabling realistic and detailed representations of text that integrate seamlessly into complex scenes. Figure 3.6 shows 3D text, a 2D *Text Render* actor and text on a canvas as a heads-up-display.

Unreal Engine 5 makes it possible to assign materials to text elements inside widget blueprints. Doing this makes it possible to animate text (and other UI elements). This is because the engine's material editor offers nodes that support values that change over

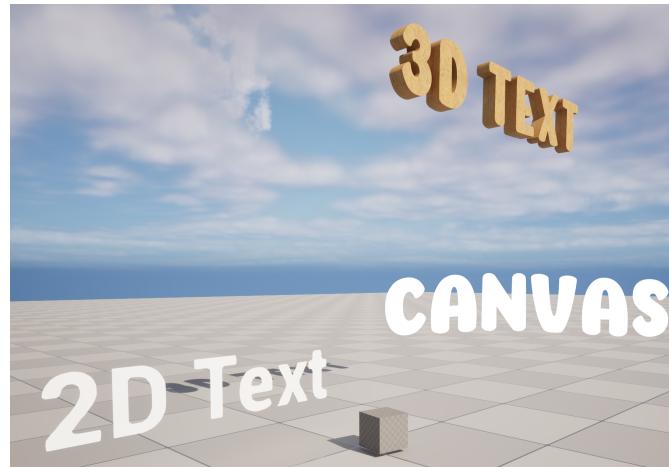


Figure 3.6: Demonstration of a 3D text mesh, a 2D Text Render actor and a text element on a canvas.

time. For example, by using a sine wave to translate the UV coordinates in a material (figure 3.8) and then assigning this material to an element in a widget, the element will appear to move in a sine wave pattern. This effect was achieved by

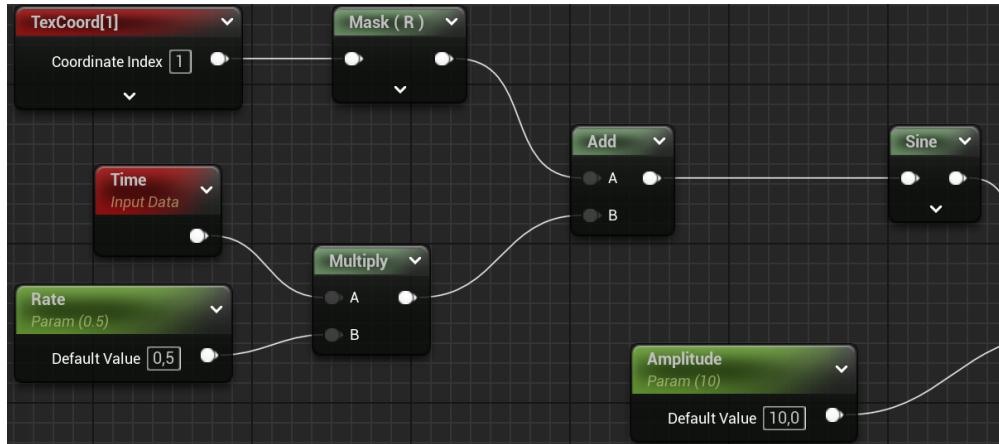
- creating a *Data Table (Rich Text Style Row)* asset for a font and assigning the sine wave material as its font material in the *Default* row,
- using the same font in a text element inside of a widget blueprint and assigning the sine wave material as its font material.

Captions can be used for improved accessibility by enabling captions in the project settings in the Unreal Engine editor. Enabling this setting tells the engine to access the caption text of audio assets used in the project. Thus, it is necessary to enter the text that should appear as the caption into the *Subtitles* text field in the specific *Sound Wave* asset. By doing this the engine will automatically display the entered text at the bottom of the screen (or between the players' view ports if split-screen is enabled) whenever the audio is played. At this time, there are no options to customize the appearance of the captions when using this built-in system.

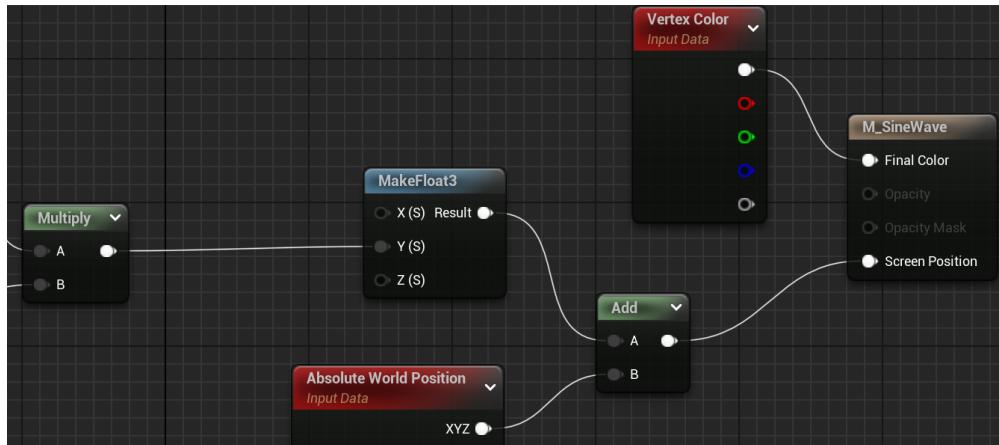
However, other types of text media such as 3D Text Meshes and text elements in widget blueprints can be customized by creating custom text style assets. Custom fonts can be imported by dragging TFF or OTF files into the media browser.

3.3.3 Audio

Sound is one of the most important aspects of multimedia applications. Luckily, Unreal Engine provides many useful tools to incorporate audio into the project, such as spatial



(a) Part 1 of the scripted material for creating a sine wave effect.



(b) Part 2 of the scripted material for creating a sine wave effect.

Figure 3.7: Scripted material for creating a sine wave effect.

audio, audio visualization, audio capture and dynamic sounds based on factors like ground material.

After importing a WAV audio file there are multiple steps that were completed to achieve spatial audio effects:

- Creating an *Audio Cue* asset based on the WAV file
- Creating a *Sound Attenuation* asset
- Assigning the attenuation to the cue

When creating the Attenuation settings there are a multitude of options such as settings for:

3. METHODOLOGY

- Attenuation function and shape
- Radius and falloff distance
- Spatialization method
- Air absorption
- Reverb support

For demonstration purposes *Natural Sound* was selected as the attenuation function and *Spatialization* and *Reverb* were enabled while leaving default values for the other options. Next, a *Reverb Effect* asset was created and assigned to an *Audio Volume* actor in the scene. The result was an *Audio Cue* asset that could be attached to any actor or spawned independently in the world to be played at specific locations while making use of spatial audio effects. For example, by adding it as a component inside of a blueprint actor and enabling auto-play and looping, the actor now always produced a sound following its location that gets louder with proximity relative to the camera. When inside the *Audio Volume* the reverb effect was audible.

To achieve adaptive audio based on the ground-material the player is standing on two *Physical Material* assets were created and assigned to two different parts of the floor mesh in the scene. Now a line trace downwards could be performed to fetch the physical material of the surface hit by the ray.

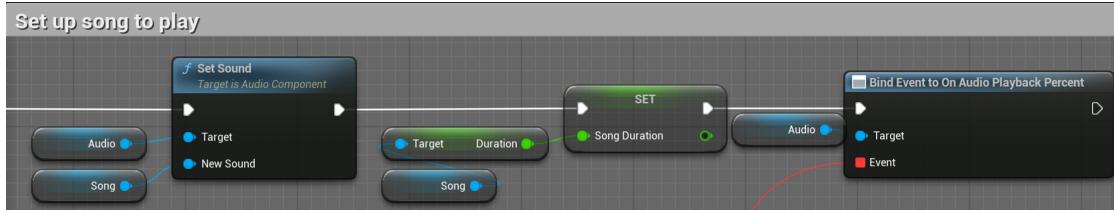
Audio visualization was achieved by binding an event to changes in the audio playback of a WAV file. The bound event calculates the percentage of a percentage bar which thereby visualizes the amplitude of the audio at any given point in time ???. A *Synesthesia NRT (Constant)* asset with default settings and a reference to the WAV file was needed to use the blueprint script node *Get Normalized Channel Constant QAt Time*.

To detect the volume level of the player's microphone a *Audio Capture Component* was used. This allows the usage of a blueprint script node called *On Audio Envelope Audio* which returns a floating point value between zero and one. By routing the value into a progress bar widget the microphone input level was visualized.

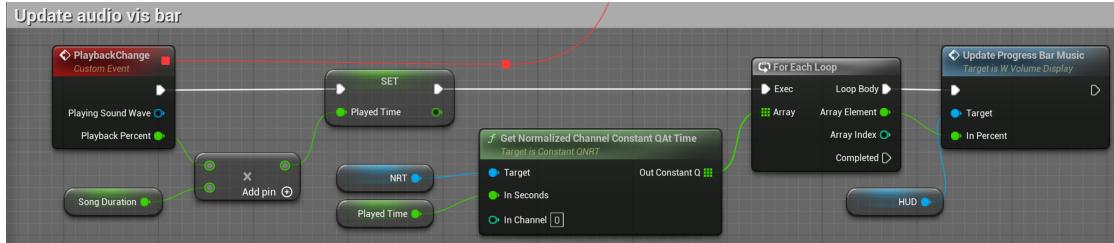
3.3.4 Images and Textures

Image files can be imported and used as textures in various ways. To display textures as user interface elements a Widget blueprint with image widgets that use the respective texture as brush was created. Also, widgets and thereby images were placed into the world by creating a blueprint class with a *Widget Component*. The *Space* setting was set to *World*, the previously created widget was selected as *Widget Class* and the *Draw Size* was adjusted to fit the desired image resolution. The created blueprint actor was then placed into the scene resulting in a floating canvas as seen in figure 3.9.

Unreal Engine provides functionality to create scripted materials using the material editor. Nodes such as *Noise*, *Time*, *Rotator* and *SmoothStep* were used to create the speed line



(a) Part 1 of the script for visualizing the audio amplitude of a WAV file.



(b) Part 2 of the script for visualizing the audio amplitude of a WAV file.

Figure 3.8: Script for visualizing the audio amplitude of a WAV file. The *PlaybackChange* event is bound to the *On Audio Playback Percent* trigger.

effect seen in figure 3.9. The material's domain was set to *Post Process*, allowing the material to be used in post processing settings.



Figure 3.9: An image used as a widget inside of a blueprint actor to place it in the scene and a scripted material used a post processing material to simulate a speed effect on the player's screen.

3.3.5 Video and Cutscenes

To play imported video files the following assets were created:

- *File Media Source*
- *Media Player*
- *Media Texture*
- A material using the *Media Texture*
- A *Widget Blueprint* that uses the material

In Unreal Engine 5 it is necessary to place all movie files that should be included in the packaged build in a folder called *Movies* inside of the *Content* folder. After importing an MP4 video file a *Media Player* asset was created. The corresponding *Media Texture* asset was created automatically. After using the texture to create a material, the material was used in a *Widget* and placed in the scene using a blueprint actor. The result was similar to the screen in figure 3.9 but with a video instead of an image. This material can also be directly applied to meshes. Finally, the blueprint script node named *Open Source* was used to open the media player. For the audio of the video to be audible, a *Media Sound Component* can be added to the blueprint actor which calls the *Open Source* node.

Creating and playing a cutscene was done using the *Sequencer*. It allows the creation of keyframes for transform values, animations, sounds, events and more. The sequence can then be played using a blueprint script. By default, the changes imposed by playing the sequence will be reset to the starting point (e.g., the positions of actors will be reset to how they were before the cutscene). Though, this can be disabled by making use of the *Keep State* option. The simple cutscene which plays at the start of the game was made by transforming a *Cine Camera* actor using keyframes to simulate a camera flying through the scene. The movement of the camera in the main menu was also done this way.

3.3.6 Effects and Programming

The built-in *Niagara* system was used to create a particle system that will be used to simulate the destruction of item boxes in the final game. The particle system consists of two emitters:

- A confetti burst
- A burst of small triangular meshes with collision

Niagara particle systems offer templates and detailed options to create effects, such as scaling particle size and color over time, forces like drag and gravity as well as collision

with other actors in the scene. By making use of these to set up the described particle system an effect that simulates the destruction of an item box from the Meltdown Mayhem game described in section 2.4 was created.

A top-down view of the player’s environment (*minimap*) was created by placing a *Scene Capture 2D* actor above the player, which continually renders a top-down view onto a *Render Target* actor. This render target was then sampled by a material which was assigned to an image widget. Displaying the widget results in a live view of what the *Scene Capture 2D* actor sees.

3.3.7 User Interface

The user interface (UI) in Unreal Engine 5 was developed using the engine’s *Widget Blueprint* system, which allows for the creation of interactive and dynamic elements. All of the menus’ and HUD’s items were implemented using this system. The foundation of the UI was built on widgets, which are modular components designed for creating interactive elements such as buttons, sliders, and text fields. Widgets were also extensively used in features like displaying images (see Section 3.3.4), text animations (see Section 3.3.2), and live data visualization (e.g., progress bars for audio input levels, as discussed in Section 3.3.3).

3.4 Final Game

After implementing the core features of the Meltdown Mayhem game the next step was to use them to create a version of the game in Unreal Engine 5 (figure ??). During this process, which is described in this section, additional features which are not present in the original Unity game were also implemented. The goal was to create a video game as similar as possible to Meltdown Mayhem while implementing everything on the feature list of the Multimedia course at the Vienna University of Technology. This section describes how the feature library was used, the implementation of the features listed in section 2.4 as well as the implementation of additional features such as fluid simulation.

3.4.1 Core Blueprint Classes

The *Game Instance* blueprint class handles a single instance of the game and keeps variable values when loading new levels. In Meltdown Mayhem’s Unreal Engine version it stores the players’ nicknames, ice-cream colors and volume settings.

A *Game Mode* class is used to handle starting, ending and pausing the game. It also manages the HUD, the pause menu and result screen, and it enables or disables player input when needed. For these purposes it keeps a reference of all existing players at all times.

As the class containing the most code the *Ice Cream Cone Player Character* class inherits from the built-in *Character* class and handles player input, provides important logic for



Figure 3.10: A screenshot taken during an ongoing race in the Unreal Engine version of *Meltdown Mayhem*.

item effects and keeps track of a player's lap count. Its Character Movement component controls available movement modes, movement speeds and how gravity acts upon the player characters.

The *Options Menu* class inherits from the *User Widget* class and provides the logic needed to adjust settings. It applies or stores the selected user settings in the *Game Instance* class. The layout and design of the options menu itself are also defined in this class. An instance of this class is created when a player opens the options menu from the main menu.

3.4.2 Racetrack

The main level of the game consists of a racetrack where players race each other to complete two laps and reach the finish line. The racetrack is designed with a whimsical and creative theme, utilizing everyday objects such as furniture, tools, appliances, and

office items to construct the racing environment. For example, desks serve as elevated platforms, chairs and books create obstacles, and kitchen utensils are repurposed as dynamic elements that players must navigate around or over as seen in figure 3.11. This design transforms ordinary household items into gameplay elements, encouraging competitive gameplay through the racetrack's structure, which includes shortcuts and alternative paths. These options allow players to gain an edge if they can successfully execute more complex maneuvers, adding a strategic element to the race. Creating the indoor racetrack was done by importing free asset packs from the Unreal Marketplace copying the assets used in the original Unity project and adding meshes to the scene. It is important to note that some static meshes use simple box collisions by default, which can lead to imprecise and unwanted collisions for the players. To mitigate this the default collision mode for specific meshes can be set to *complex* inside of the *Static Mesh* asset. This setting results in the engine using the object's trimesh for collision detection.

The racetrack scene features a combination of directional light, a skylight, and multiple point lights, complemented by baked spotlights. To achieve global illumination, Lumen was utilized and it is enabled by default. While the baked lights are not visible with Lumen active, they serve as a fallback for the lowest graphic setting where Lumen is disabled. A *Post Process Volume* actor was used to disable auto exposure by setting the values for *Min EV100* and *Max EV100* to -2.0 . To convey the scale of the world to players, who take control of ice-cream cones, depth of field was added, emphasizing the miniature perspective of the environment.

3.4.3 User Interface

All user interface elements were implemented using widgets. Built-in button-press events were used to detect player input in the menu. Players are greeted with a main menu upon launching the game, providing them with buttons to (1) play, (2) open the settings menu, or (3) exit the game.

The settings menu, which can be seen in figure 3.12, gives the players the option to:

- adjust the game audio volume
- change their nicknames
- select their ice-cream colors
- change the graphics preset (Medium, High, Epic)
- change the background image
- enable or disable captions.

Two sliders were used to make the game music volume and game sound effect volume adjustable separately. To achieve this, two different *Sound Class* assets were created, one

3. METHODOLOGY



Figure 3.11: Image of the racetrack scene. Players race each other by navigating this environment. The assets used were imported from the Epic Games Marketplace and the Unity Store.

for each sound category. During development every sound asset was assigned one of the created classes. This way the volume of sounds in the same category can be controlled collectively. After pressing the Play button players can skip the introduction cutscene by using the Skip button. Next, the race is started after a visible countdown.

During gameplay, a heads-up display provides players with information such as (1) the lap count, (2) which item they possess, (3) a minimap and (4) captions for sound effects (if they are enabled). By opening the Pause Menu players freeze the action and gain access to a simplified settings menu, which offers:

- sliders for volume adjustment
- a dropdown menu for graphics setting
- a toggle for displaying or hiding the minimap

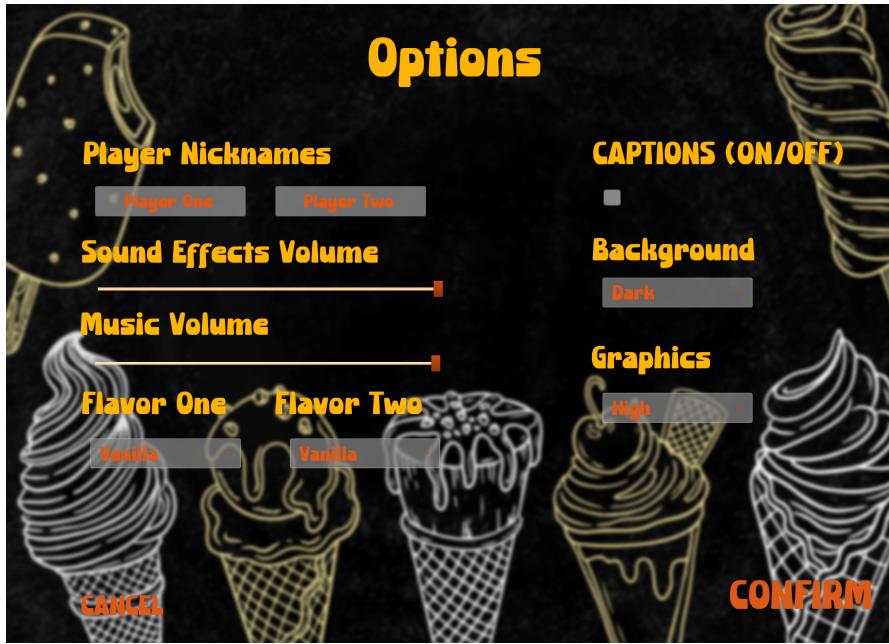


Figure 3.12: Image of the options menu.

- buttons to continue or exit the race. After a race has finished a result screen is shown, congratulating the winner. Players are then returned to the main menu which can be seen in figure 3.13.

3.4.4 Checkpoint System

The game features a checkpoint system that ensures fair competition and accurate progress tracking, functioning similarly to the mechanics found in popular racing games such as *Mario Kart 8* [GNin14]. The racetrack is divided into multiple segments, with checkpoints placed strategically along the course. Players must pass through these checkpoints in the correct order to validate their progress and complete a lap successfully.

The checkpoint system serves two purposes within the game:

- Lap Validation: A lap is counted if the player has crossed all checkpoints in sequential order. This prevents players from skipping sections of the track or taking unintended shortcuts that might disrupt the integrity of the race.
- Respawn Points: If a player leaves the playable area, the game moves the character to the last checkpoint they crossed. This minimizes frustration while maintaining the challenge of overcoming the same section again.

To detect when a player leaves the racetrack, e.g., by falling off a table, blueprint actors with *Box Collision* components were strategically placed in the level. When a player



Figure 3.13: Image of the main menu. A looping cutscene runs in the background slowly moving the camera around the table. Cupcakes rain down from above with their speed influenced by the microphone input volume level. The ice-cream cones dynamically stretch to the beat of the background music.

begins to overlap such a collision box the player’s ice-cream cone character is moved to the last checkpoint they passed through after a short delay.

Each checkpoint has an index $i \geq 1$. Passing through a checkpoint A with the index i_a only counts as progress if the index i_b of the checkpoint B which was passed immediately before A equals $i_a - 1$. The finish line is its own class and keeps a reference of every checkpoint. If a player passes the finish line and the index of the checkpoint they passed immediately before passing the finish line is equal to the length of the list of checkpoints that player’s lap count is increased by one.

3.4.5 Item Effects

The game incorporates a dynamic item system to enhance the competitive and chaotic nature of gameplay. Players can collect items by passing through item boxes, which are scattered at key points along the racetrack. These item boxes are designed as rotating cubes (see figure 3.14) that reappear after a short delay once collected, ensuring that all players have opportunities to obtain items during the race. The inclusion of reappearing item boxes promotes risk-taking, as players must decide whether to detour slightly to collect an item or maintain their current route for a faster lap time. After collecting an

item box the player receives a random item if they are currently not already in possession of an item. Items are consumed after a single use. The available items are designed to introduce a mix of advantages for the user and obstacles for their opponents:

- Speed Boost: Upon using this item, the player's ice cream cone gains a temporary surge in speed, allowing them to close gaps, overtake opponents, or navigate challenging sections of the track more efficiently. This effect lasts for a brief period, balancing its power while giving players a significant advantage in the moment.
- Distracting Video: This item disrupts the gameplay of the opposing player by displaying a semi-transparent video on their screen, partially obscuring their view of the racetrack.
- Camera Shake: When triggered, this item causes the camera perspective of the opponent to shake violently, making it difficult for them to focus on precise movements or judge distances. This effect is particularly disruptive during platforming sections, where accuracy and timing are critical.
- Homing Projectile: This item launches a rock that automatically targets the opposing player. The hit player's movement is temporarily disabled.

Depending on the type of item a player possesses the *Use Item* button or key activates the corresponding effect. Item-types were implemented using an *Enumeration* asset and using the *Switch on Enumeration* blueprint node. The homing projectile was implemented using the *Projectile Movement* component. It enables the ability to tune aspects like speed, acceleration, the influence of gravity on the projectile. More importantly, it offers a built-in option called *Is Homing Projectile*. Combined with the *Homing Acceleration Magnitude* setting and the variable *Homing Target Component* a simple rock projectile could be spawned and set to seek the other player.

By creating a blueprint class inheriting from the *Default Camera Shake Base* class the shaking of the camera that occurs when the *Start Camera Shake* node is called on a *Camera Component* can be tuned. It allows for choosing a noise pattern and a duration as well as per-axis amplitude and frequency settings for location- and rotation-based camera shake.

The distracting video makes use of the method described in section 3.3.5. Upon using this item, the *Start Media Player* node is called, and the opacity of the video canvas corresponding to the other player's half of the screen is increased for a set duration.

When using the *Speed Boost* item the user's Max Walk Speed and Max Acceleration variables of the Character Movement component are increased for a fixed duration. Also, the speed lines effect (see figure 3.9) is made visible by setting the *Post Process Blend Weight* to 1.0 for the same duration in the *Camera* component of the player character blueprint before returning its value back to 0.0. This works because the speed lines material was chosen as a *Post Process Material* in the *Rendering Features* settings of the *Camera* component in the player character blueprint.

3. METHODOLOGY

3.4.6 Audio Visualization

Audio visualization was implemented in two ways:

- visualizing audio volume by dynamically changing the scale of static meshes to reflect volume
- visualizing microphone input by making objects rain down faster when microphone input is detected in the main menu.

Dynamic mesh scaling was achieved by making use of the code from the feature library described in section 3.3.3. Instead of passing the calculated value to a progress bar it was used as input for the *Set World Scale 3D* node to scale the *Static Mesh Component* of actors in the scene.

The physics simulation of *Static Mesh* components offers adjustments such as the *Linear Damping* variable. The fall speed of objects in the main menu was increased dynamically based on microphone input by setting this variable to smaller values if the *Envelope Value* of the *Audio Capture* component gets larger.

3.4.7 Fluid Simulation

Unreal Engine 5 provides a free plugin called *Niagara Fluids*. Enabling it enhances the capabilities of the Niagara particle system editor by supporting fluid and smoke simulations. Using this plugin a particle system for simulating water coming out of the faucet and flowing into the sink below was created which can be seen in figure 3.14. The flowing water collides with players when they move through it, causing it to flow over the player's character model.

The starting point was a *Grid3D FLIP FluidCo Emitter* in a Niagara particle system. It was adjusted to fit into the environment and be more performant by tuning variables such as

- Num Cells Max Axis
- Particles Per Cell
- Pressure Iterations
- Water Height
- World Grid Extents.



Figure 3.14: Simulated water exiting a faucet. Fluid simulation was implemented using the *Niagara Fluids* plugin. An item box can be seen in the bottom left corner of the image.

3.4.8 Sound Design

The sound design of *Meltdown Mayhem* aimed at enhancing player immersion and complementing the visual and gameplay elements. A combination of original compositions and sound effects was utilized to create the unique atmosphere of the game. The music is comprised of upbeat and whimsical tracks that match the lighthearted and playful nature of the game. Additionally, custom sound effects were created to accentuate player actions and environmental interactions. These effects included sounds for item pickups, power-up usage, and collision impacts, adding depth and feedback to the gameplay. The game incorporates both 2D and 3D sound elements to create an immersive auditory landscape. 2D sounds, such as background music and UI interactions, ensure a consistent auditory presence. In contrast, 3D sounds were used to localize specific audio cues within the game world, such as environmental ambiance and item-related effects. To further enhance the immersive quality of the game, audio effects like reverb and attenuation were applied to simulate environmental characteristics. For instance, the indoor racetrack features a slightly reverberant audio profile, reflecting the acoustics of a confined space.

Results and Analysis

This chapter presents the results of the comparative analysis between Unreal Engine and Unity, highlighting the challenges encountered, the ease of use of various features, and their potential impact on development workflows. In addition, the chapter explores the role of AI tools, particularly ChatGPT, in assisting with these workflows and resolving engine-specific challenges.

4.1 Challenges and Feature Comparison

This section outlines the challenges encountered during development in both engines, focusing on the general implementation process and the development of specific features. Comparison criteria include the ease of use of engine-specific tools, the time spent on implementation, and the availability and helpfulness of online resources and AI tools.

4.1.1 General Comparison

Unreal Engine's third-person template provided a comprehensive starting point, including a fully implemented camera system, character movement, animation system, and replaceable player avatars. Unity templates, while lightweight and flexible, required more initial setup, especially for integrating features such as third-person controls.

Unreal Engine's visual scripting (Blueprints) allowed fast prototyping of features, making it beginner-friendly. Unity's scripting offered more control and modularity but at the cost of higher implementation times for similar tasks. For example, creating the blueprint for item boxes with player collision, particle system, and sound was faster in Unreal Engine due to prebuilt functionality, whereas Unity required more manual scripting.

Both engines provided extensive documentation and tutorials. However, while using Unreal Engine's blueprints, the built-in tooltips and descriptions often answered any

4. RESULTS AND ANALYSIS

questions that had come up. In cases where that was not the case, searching for a solution online almost always resulted in finding a multitude of detailed tutorial videos such as [YT1] [YT2] [YT3] [YT4] [Vim]. Working with Unity and its code documentation seemed to be less beginner-friendly overall.

The development time for creating *Meltdown Mayhem* varied significantly between Unity and Unreal Engine, influenced by differences in workflows, prior experience, and tasks involved. The Unity version of the game was developed by two developers, each contributing approximately 45 hours, totaling around 90 hours of combined effort. This version of the game was created entirely from scratch, which required building all systems, assets, and features from the ground up. Furthermore, since neither developer had prior experience with Unity, a considerable portion of the time was spent familiarizing ourselves with the interface of the engine, scripting in C#, and understanding its workflows. In contrast, the Unreal Engine version of the game was developed as a migration of the existing Unity project. The migration process took approximately 50 hours and was completed by a single developer (myself). While this was a solo effort, my prior experience with Unreal Engine played a key role in streamlining the process. The migration made use of Unreal Engine's built-in templates, tools like Blueprints for rapid feature implementation, and intuitive systems for tasks such as lighting, particle effects, and spatial audio. The project can be found on GitLab [MM2].

4.1.2 Feature-Specific Analysis

Lighting

Unreal Engine's lighting system, powered by Lumen, provided dynamic global illumination out of the box, making it straightforward to achieve realistic lighting effects. The baked lighting setup in Unity required manual configuration, which was more time-consuming. Unreal's environment light mixer made enhancing the visual fidelity of the game quick and simple.

User Interface

Unity's canvas-based UI system offered flexibility and integration with its scripting model, but setting up menus, HUDs, and interactive buttons required careful anchoring and manual adjustments. Unreal Engine's widget blueprint system was more intuitive for creating interactive elements, with drag-and-drop functionality that minimized scripting. For instance, displaying power-ups and lap progress was faster to implement in Unreal due to the ease of integrating Blueprints with UI elements.

Terrain and Foliage

Unity's terrain tool allowed the creation of an outdoor environment, complete with hills and trees. While it offered basic sculpting and painting features, adding foliage required manually placing imported tree assets in the scene. Unreal Engine's foliage tool provided

more advanced functionality, including automatic placement and density control, which reduced development time for the outdoor scene.

Animations and Cutscenes

Creating cutscenes and animations in Unreal Engine was streamlined by the Sequencer tool, which provided an integrated timeline editor for animating actors and cameras. The version of Meltdown Mayhem made with the Unity engine does not include cutscenes. However, animating the text for the main menu took more time with Unreal's node-based animation system compared to Unity's script-based approach. This is because ChatGPT was able to directly generate the code needed for the script, but was unable to generate blueprint code. However, ChatGPT was able to provide a list of steps needed to create the material in Unreal Engine (see section 4.2.1).

In Unreal Engine, character animations were enhanced using blend spaces, a feature that enables smooth transitions between different animations based on player movement. For instance, the player's movement speed was linked to the animation state, ensuring that the character naturally transitioned between idle, walking, and running animations. The blendspace visual editor allowed for the mapping of speed values to corresponding animations. This made it possible to easily fine-tune the system for fluid motion. In Unity, transitions between animations were manually triggered based on predefined thresholds for player movement speed.

Split-Screen Multiplayer

Setting up the split-screen functionality in Unity required custom scripts and camera management, adding to the development overhead. Specifically, an issue that arose during the development was that both halves of the screen were showing the same viewport (that of player one), instead of two different viewports. Multiple hours were spent focusing on this problem, which was solved by creating the following custom script which was added to each of the player prefabs. It enables shared rendering of global elements like UI while rendering each player's camera's viewport on the correct half of the screen.

```
public class PlayerManager : MonoBehaviour {

    public Camera cam;
    public GameObject virtualCam;

    public void Setup(int layer) {
        virtualCam.layer = layer;
        var bitMask = (1 << layer)
        | (1 << 0)
        | (1 << 1)
        | (1 << 2)
        | (1 << 4)
        | (1 << 5)
        | (1 << 8);
```

4. RESULTS AND ANALYSIS

```
    cam.cullingMask = bitMask;
    cam.gameObject.layer = layer;
}
}
```

Character and Camera Systems

Unreal Engine's third-person template included a pre-configured character controller with movement, jumping, and sprinting mechanics, along with a smooth third-person camera system and a physics system with gravity and other forces working out of the box. These features were ready to use and required minimal adjustments. In Unity, creating the player avatar and camera system demanded scripting, including handling movement input, camera rotation, and gravity. Unreal's Blueprint system also made customizing character motion and interactions more accessible to beginners by providing adjustable parameters with tooltips and descriptions listed by category inside the player character blueprint.

Multimedia and Effects

Both engines support a wide range of multimedia types, but the implementation process differed:

- **Audio and Spatial Effects:** Unreal Engine's spatial audio tools were easy to set up and integrate with environmental elements while allowing detailed fine-tuning. The engine provides detailed controls for fine-tuning effects such as sound attenuation, occlusion, and reverberation, allowing for realistic and immersive soundscapes. These features were leveraged to implement 3D audio in the Unreal version of the game, where sounds dynamically changed based on player position and environment. Spatial audio was not implemented in the Unity version of the game.
- **Imagery:** Unreal Engine and Unity both handled the inclusion of images and textures with ease, but their workflows diverged. In Unreal Engine, materials and textures were managed via the Material Editor, which offers a highly visual, node-based interface. This made it easier to create dynamic and layered materials. Unity, on the other hand, lacked some of the visual intuitiveness of Unreal's editor, though it still provided sufficient functionality for static imagery and texture manipulation.
- **Video:** Integrating video content in Unity relied on the built-in VideoPlayer component, which was attached to the player prefabs and controlled via a custom script. This straightforward setup worked well for playing and stopping videos as well as changing the video's opacity. Unreal Engine, by contrast, required a more complex pipeline, including creating a *Media Player*, *Media Texture*, and corresponding materials or widgets to display the video. While this approach allowed for greater customization, such as integrating video seamlessly into 3D environments, it also required more effort and familiarity with Unreal's asset system.

- Particle Systems: Unreal’s Niagara particle system provided a rich set of templates and editing tools, enabling fast creation of effects such as item pick-up animations. The ability to control particle behavior using customizable modules allowed for more dynamic and detailed results. Unity’s built-in particle system, while functional, felt more limited in comparison. This was because of the comparatively smaller number of options and because there were no templates to choose from in contrast to Unreal Engine.
- Text: In Unreal Engine, text elements were implemented through a combination of 3D text meshes (*Text 3D* plugin), widgets for HUD elements, and support for rich text styling and animations via material assignments. This allowed for high customizability, but required familiarity with Unreal’s UI systems.

Overall, while Unreal Engine offered more advanced multimedia capabilities and customization options, these features often came at the cost of a steeper learning curve and more complex workflows compared to Unity’s code-centric approach.

Physics and Simulation

Unreal Engine’s physics system allowed for realistic interactions with minimal configuration. Features like fluid simulation and object collisions were integrated into the engine and ready to use. Adding simulated fluids to the scene was easy after enabling the *Niagara Fluids* plugin in Unreal Engine because the plugin simply extended the Niagara particle system editor. Unity did not offer a comparable solution, so fluid simulation was not implemented in the Unity version of the game.

Additional Features

- Checkpoint System and out-of-bounds Zone: The implementation of this system (described in section 3.4.4) was very similar in both engines. The only notable difference was the way the code was written (C# script and blueprints). In Unity the *OnTriggerEnter* event provided the same functionality as the *Begin Overlap* event in Unreal Engine.
- Item Effects:
 - Distracting Video: Unity utilized the *VideoPlayer* component to overlay a semi-transparent video on the opponent’s screen. A script controlled the playback. In Unreal Engine, the process required creating a Media Player and associating it with a *Media Texture*, which was then rendered to a UI widget. Unreal’s approach required more setup and familiarity with the engine’s media tools.
 - Camera Shake: In Unreal Engine, the camera shake was achieved using a *CameraShake* blueprint class, which provided built-in parameters to control

4. RESULTS AND ANALYSIS

the intensity and duration of the effect. In Unity, a custom script was needed to implement the camera shake. The script used the amplitude and frequency controls of the *CinemachineMultiChannelPerlin* component to change the values over time. Camera shake was easier to implement in Unreal Engine because of the ready-to-use *CameraShake* base blueprint.

- Homing Projectile: In Unreal Engine the projectile’s behavior was controlled using a *Homing Projectile Component*, which simplified the targeting logic but required configuration in the blueprint editor. In Unity, the homing logic had to be coded manually using a script. Both engines made collision detection equally easy by using the built-in collision volumes. Unreal’s built-in features made it easier to implement the homing mechanics than it was in Unity.
- Speed Boost: This item effect was not implemented in Unity. In Unreal, the speed boost was implemented by momentarily changing the walk speed variable of the *Player Movement Component*.

4.1.3 Performance

Another aspect of evaluating the Unity and Unreal Engine versions of the game was performance. The two engines demonstrated distinct trade-offs in terms of visual quality and runtime efficiency, influenced by their respective rendering technologies and implementation choices.

Visual Quality vs. Performance

The Unreal Engine version of the game utilizes Lumen, a real-time global illumination and reflection system introduced in Unreal Engine 5. Lumen significantly enhances the visual quality by providing dynamic lighting that reacts in real time to environmental changes. However, this feature comes at a high computational cost, resulting in increased system resource usage and lower frame rates. On the other side, the Unity version relies on baked lighting for its environment, which, while less dynamic, is much more performant. The use of baked lighting in Unity allowed for a consistent frame rate, especially on lower-end systems, but at the expense of less visually compelling real-time effects.

The Unity version of the game used low-poly assets, which are optimized for performance and contribute to maintaining a consistent frame rate on lower-end systems. In contrast, the Unreal Engine version utilized higher-poly assets to take full advantage of the advanced rendering capabilities of Unreal Engine 5, delivering a more detailed and visually impressive visual experience.

To accommodate a broader range of hardware capabilities, graphics options were added to the Unreal Engine version. The lower graphics presets disable Lumen and instead use baked lights included in the scene. These baked lights are automatically hidden when Lumen is enabled (higher graphics presets), so that they do not interfere with the real-time lighting effects. This approach balances visual quality and performance,

allowing the game to run smoothly on both high-end and low-end systems while providing visually beautiful effects on high-end hardware.

Fluid Simulation

Unreal Engine's Niagara particle system allowed for the creation of visually complex effects. While versatile, complex effects such as fluid simulation demanded careful parameter adjustments to maintain acceptable performance. Fluid simulation was not implemented in Unity, so, due to the comparatively simple nature of the Unity particle systems, there were no noticeable drops in performance when spawning the particles.

Minimap Implementation

A notable performance bottleneck was the implementation of the minimap. In Unreal Engine, a live-view minimap using a secondary camera rendered the entire scene in real time. While this approach provided accurate and dynamic visual feedback to players, it incurred a significant performance hit due to the additional rendering workload. The Unity version does not include a minimap. To address the performance cost of the live-view minimap, an alternative method could be explored, using a pre-drawn map texture with dynamically updated player positions represented as sprites.

4.2 AI Tools: ChatGPT's Role in Development

This section examines the role of ChatGPT in assisting the development process in both Unity and Unreal Engine. ChatGPT's strengths and limitations as a generative AI tool are evaluated to provide insights into its impact on multimedia game development workflows.

4.2.1 Strengths

ChatGPT demonstrated significant utility in various stages of the development process, particularly in Unity:

- **Code Generation:** Having the ability to generate and modify C# code proved very useful when working with the Unity engine. For example, the AI was able to generate the complete code needed for animating text vertices in a sine wave pattern.
- **Configuration Assistance:** In Unreal Engine, ChatGPT provided valuable support in generating and editing *.ini* configuration files. Specifically, during the development of the network template an issue arose in which no sessions were found in the packaged version of the game. ChatGPT suggested modifying the *DefaultEngine.ini* configuration file as described in section 3.2, which solved the problem.

4. RESULTS AND ANALYSIS

- Step-by-Step Guide: For example, even though the AI could not generate blueprint code directly, it was still helpful in creating the scripted material for the sine wave text animation in Unreal Engine, though, some adjustments were needed to create the final material seen in figure 3.8.

The prompt and response for creating the sine wave material were as follows:

User Prompt 1

I want to animate text in a widget in my Unreal Engine 5.4 project using a sine wave without using the timeline. Generate the blueprint code for the material.

ChatGPT Response (paraphrased) 1

Right-click in the Content Browser and select Material. Name it something like *M_SineWaveText*. Next, open the material editor and configure the nodes as follows:

1. Add a ‘Time‘ node.
2. Connect the ‘Time‘ node to a ‘Multiply‘ node (adjust speed with a constant).
3. Connect the ‘Multiply‘ node to a ‘Sine‘ node.
4. Connect the ‘Sine‘ node to another ‘Multiply‘ node (adjust amplitude with a constant).
5. Optionally, add an ‘Add‘ node after the ‘Sine‘ node to shift its baseline.
6. For position animation: Connect the final value to ‘World Position Offset‘.
7. For opacity animation: Connect the final value to ‘Opacity Mask‘.
8. Apply a ‘Font Parameter‘ node for text materials if needed.

4.2.2 Limitations

Despite its strengths, ChatGPT faced various limitations that affected its utility, especially in Unreal Engine development:

- Blueprint Support: The biggest limitation proved to be ChatGPT’s inability to read, modify, or generate blueprint code. Because of this, when working with Unreal Engine, the AI was used to complement the process of looking up specific tutorials online.

- Over-complicated Solutions: Another limitation was that ChatGPT sometimes seemed to suggest overly complicated solutions. For example, when the AI was asked how to animate text in a sine-wave pattern in a widget, it initially suggested using the sequencer instead of a text material. Only after asking for a way to do it without using the sequencer did it suggest using the material system.
- Hallucinations: When ChatGPT was unable to find a solution, it sometimes started to make up solutions that did not exist. For example, the AI hallucinated a function to set the input device of a player directly using blueprints (seen below). While some of the things mentioned by the AI in its response exist, the final step it suggested to solve the problem was impossible to follow because the mentioned function does not exist, and the alternative it suggested also does not solve the problem.

User Prompt 2

How do I assign a specific input device to a specific player?

ChatGPT Response (paraphrased) 2

1. Set Up the Input Subsystem: Each Player Controller has an associated *EnhancedInputLocalPlayerSubsystem*. This subsystem is used to manage input mappings for the specific player.
2. Set up Input Mapping Contexts: You should have separate *Input Mapping Contexts* for each player, especially if they're using different types of devices (e.g., Player 1 on a keyboard, Player 2 on a gamepad).
3. Assign Devices Based on Input Events: Listen for specific input events (like a button press) from each device, and when an input is detected, assign that device to the corresponding player.
4. Example Blueprint Logic:
 - a) For Player 1 (assuming they use a keyboard/mouse): Use the Enhanced Input system to bind actions like movement or jump to the keyboard/mouse. This binding will naturally be assigned to Player 1 by default.
 - b) For Player 2 (using a gamepad): Set up a Blueprint logic that detects a specific gamepad input (like pressing a specific button). When that button is pressed, assign the gamepad to Player 2 using the *Set Input Device* function or by directly managing it through the *EnhancedInputSubsystem*.

CHAPTER 5

Conclusion

This thesis explored the development of the game *Meltdown Mayhem* in Unity and Unreal Engine, highlighting key differences, advantages, and disadvantages between the two engines. The analysis covered various aspects of game creation, including feature implementation, performance, and the integration of AI tools like ChatGPT into development workflows.

ChatGPT played a valuable role in supporting development tasks, particularly in Unity where it excelled at generating C# code and debugging advice. However, its limitations, especially in Unreal Engine's Blueprint system and its partial knowledge of replication settings, required manual research and corrections. Despite these shortcomings, ChatGPT proved to be a useful assistant for brainstorming, prototyping, and resolving technical challenges, though with some limitations.

5.1 Implications for Students of the Multimedia Course

For students in the Multimedia course, I recommend that the following aspects be prioritized to enhance their learning experience and support them during the development phase. The list is based on the results of the analysis, the ease of use and beginner friendliness of available tools and features, as well as the importance of the points with regards to developing a functional game in Unreal Engine 5.

5.1.1 Understanding the Blueprint System

Students should focus on using classes and nodes correctly to effectively manage the logic of the game. This includes understanding components, references, casting, and inheritance, as well as learning how to link blueprints to create interactions and functionality within the game. Learning about these concepts will allow students to prototype gameplay mechanics and integrate multimedia elements efficiently.

5. CONCLUSION

5.1.2 Integration of Multimedia Types

A key focus should be the integration of multimedia assets. Students should learn to embed video, text, spatial audio, and other media into the game scene. Additionally, they should learn about creating particle effects using tools like Unreal's Niagara system to add visual flair and enhance interactivity.

5.1.3 Lighting and Rendering Fundamentals

Understanding lighting and rendering is critical for creating visually compelling and immersive scenes. Students should learn the differences between static and dynamic lighting, explore the use of Lumen for global illumination, and evaluate the performance implications of different lighting techniques. These skills are essential to achieve both visual quality and technical efficiency in their projects.

5.1.4 User Interface Creation with Widgets

Students should develop skills in the design of effective and reusable user interfaces. This includes creating widgets, nesting widgets within other widgets, and ensuring responsive layouts with proper use of anchor points. In addition, they should learn to implement button events and animate user interface elements to deliver a polished and engaging user experience.

5.1.5 Optimization Techniques

To ensure their projects run smoothly, students must learn optimization techniques. This involves using profiling tools to identify and address performance bottlenecks. By understanding how to streamline multimedia assets and how to use lights without forfeiting too much performance, students can achieve a balance between quality and performance.

5.1.6 Project Organization and Workflow

Effective project organization is essential for collaborative development. Students should avoid version control merge conflicts, especially with binary `.uasset` files, by working on separate test levels and blueprints. Communication within the team is critical, and they may benefit from using plugins to lock files during editing. Consistent naming conventions should also be emphasized to maintain clarity and efficiency throughout the development process. An overview of the process of packaging a game for Windows and encouraging students to regularly build their game could help them avoid long debugging sessions towards the end of the semester.

5.2 Final Thoughts

This thesis provided an in-depth exploration of the development process for collaborative multimedia applications, highlighting the comparative advantages and challenges of working with Unity and Unreal Engine. The findings underscore the importance of understanding the tools and workflows of each engine to align development priorities with project goals effectively. Although the analysis covered a wide range of topics, recent advancements, such as the texture graph and compute shaders introduced in Unreal Engine 5.5, were not explored due to their absence in the version used during development. Similarly, while the alternative pre-drawn minimap method was suggested as a solution to performance issues, its implementation and potential benefits remain an area for further investigation.

The results suggest that Unreal Engine is a promising platform for future projects, particularly for students in multimedia courses. Its toolset, including Lumen for real-time global illumination, Niagara for particle systems, and Sequencer for cutscene creation, allows for crafting immersive and visually impressive experiences. While Unreal Engine's Blueprints system makes it beginner-friendly and accessible for rapid prototyping, the engine's extensive feature set requires students to invest time in understanding its workflows, exploring its tools, and seeking out quality tutorials to fully take advantage of its potential.

There are several areas that could be explored further to enhance the outcomes of this project. For example, implementing and comparing the newly introduced texture graph and compute shaders in Unreal Engine 5.5 could provide insights into their impact on performance and open up new creative possibilities. Additionally, refining the minimap system with a pre-drawn map and dynamic player markers could address the performance issues encountered with live-view minimaps. Other potential improvements include experimenting with advanced networking solutions to expand the multiplayer functionality, incorporating procedural generation techniques for creating more dynamic environments, and exploring the use of behavior trees to add intelligent non-player characters.

Bibliography

- [AdFI22] Fabio Alexandrini, Adriano Gomes de Freitas, and Bruno Inacio. Comparative evaluation unity and unreal, using nielsen's 10 heuristics as an evaluation parameter. *IEOM Society International*, 12, 2022.
- [CKGD21] Agata Ciekanowska, Adam Kiszczał-Gliński, and Krzysztof Dziedzic. Comparative analysis of unity and unreal engine efficiency in creating virtual exhibitions of 3d scanned models. *Journal of Computer Sciences Institute*, 20:247–253, 2021.
- [DBEK17] Paul E Dickson, Jeremy E Block, Gina N Echevarria, and Kristina C Keenan. An experience-based comparison of unity and unreal for a stand-alone 3d game development course. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, pages 70–75, 2017.
- [GL16] Przemysław Gora and Lukas Leibetseder. Unreal vs unity - ein vergleich zwischen zwei modernen spiele-engines, 2016.
- [MM2] Meltdown mayhem gitlab. https://gitlab.tuwien.ac.at/e193-03-virtual-and-augmented-reality/multimedia_unreal. Accessed: 23.1.2025.
- [Qui] Quixel bridge. <https://quixel.com/bridge>. Accessed: 8.1.2025.
- [SK24] Lav Soni and Amanpreet Kaur. Merits and demerits of unreal and unity: A comprehensive comparison. In *2024 International Conference on Computational Intelligence for Green and Sustainable Technologies (ICCIGST)*, pages 1–5. IEEE, 2024.
- [UE2] Unreal engine. <https://www.unrealengine.com>. Accessed: 7.1.2025.
- [UT2] Unity technologies. <https://www.unity.com>. Accessed: 7.1.2025.
- [Vim] Beginner tutorial series. <https://vimeo.com/showcase/ue5>. Accessed: 23.1.2025.

- [YT1] Split-screen tutorial. <https://www.youtube.com/watch?v=QSnUqHF6X0M>. Accessed: 23.1.2025.
- [YT2] Playing video files tutorial. <https://www.youtube.com/watch?v=UwJ1Ty2crxM>. Accessed: 23.1.2025.
- [YT3] Audio visualization tutorial. <https://www.youtube.com/watch?v=3JYyNucyMBk>. Accessed: 23.1.2025.
- [YT4] Animated text tutorial. <https://www.youtube.com/watch?v=hRBw-sqqdZk>. Accessed: 23.1.2025.

Ludography

[GLE19] Lucasfilm and Fairview Entertainment. *The mandalorian*, 2019. TV Series.

[GNin14] Nintendo. *Mario kart 8*, 2014. Video Game.

[GPro19] Kojima Productions. *Death stranding*, 2019. Video Game.

[GThe24] Ninja Theory. *Senua's saga: Hellblade ii*, 2024. Video Game.