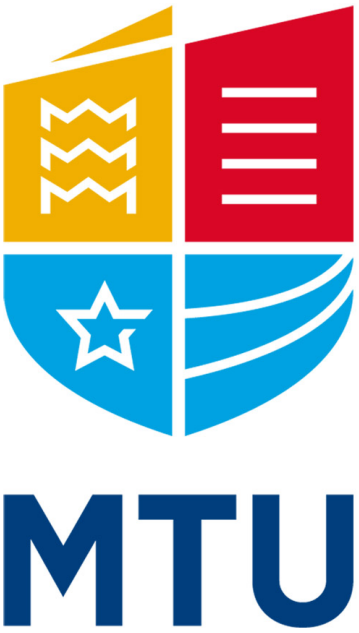

Streamlining The Design And Implementation Process of Dynamic Three-Dimensional Game Development

by

Lochlann O'Neill

This thesis has been submitted in partial fulfillment for the degree of Bachelor of Science in Software Development

in the
Faculty of Engineering and Science
Department of Computer Science

May 2023

Declaration of Authorship

I, Lochlann O'Neill, declare that this thesis titled, 'Streamlining The Design And Implementation Process of Dynamic Three-Dimensional Game Development' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for an undergraduate degree at Munster Technological University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at Munster Technological University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this project report is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

Munster Technological University

Abstract

Faculty of Engineering and Science
Department of Computer Science

Bachelor of Science

by Lochlann O'Neill

Since their conceptual foundation, video games have grown rapidly, and there is no indication that they will end in the near future. In the wake of the COVID-19 pandemic and the subsequent lockdowns, the gaming industry has experienced a surge in activity among all age groups. Various subgenres within each generalised genre will ensure that players from all backgrounds will be able to find unique experiences that cater to their specific tastes.

For the inexperienced developer, developing a fully-functional game with dynamic physics and interesting gameplay may seem overwhelming. Utilizing modern technologies such as Unity and Blender, I will demonstrate how to create a third-person perspective video game with three-dimensional navigation using a practically streamlined process for both design and implementation. It is intended that the reader will gain an understanding of the steps necessary to develop a game with similar functionality. Among the features of this game are a movement system, a combat system, an item management system, characters, character statistics, as well as both original and referenced 3D artwork.

Acknowledgements

I would like to thank my coordinator Larkin Cunningham for being an excellent research adviser, providing me with continuous invaluable feedback in order to reach my full potential.

It would not have been possible for me to complete my degree without my mother, who provided the majority of the financial aid.

The core concept of this research paper derives from my introduction to video games by my father at a young at a young age.

My passion for gaming would not have developed to its current level if my uncles had not helped me build my computer expose me to so many different games.

I learned a great deal about game development from my brother throughout the years, who graduated from TUS with a BSc (Honors) in Game Development.

I would like to thank my sister who created an excellent paper prototype based on my game description.

The last thing I would like to acknowledge is how much I appreciate the support I received from my friends during many sleepless nights spent doing assignments, without which I would not have been able to reach my potential.

Contents

List of Figures

List of Tables

Abbreviations

GUI	**G**raphical **U**ser **I**nterface
PLC	**P**ost **L**eaving **C**ertificate
QQI	**Q**uality and **Q**ualifications **I**reland
IDE	**I**ntegrated **D**evelopment **E**nvironment
BSc	**B**achelor of **Sc**ience
OOP	**O**bject **O**riented **P**rogramming
FSM	**F**inite **S**tate **M**achine
SWOT	**S**trengths **W**eaknesses **O**pportunities **T**hreats
MVP	**M**inimum, **V**iable, **P**roduct
MDA	**M**echanics, **D**ynamics, **A**esthetics
POV	**P**oint **O**f **V**iew
ARG	**A**lternate **R**eality **G**ame
VR	**V**irtual **R**eality
AR	**A**ugmented **R**eality
RPG	**R**ole **P**laying **G**ame
2D	**T**wo **D**imensional
3D	**T**hree **D**imensional
2.5D	**T**wo-and-a-half **D**imensional
COM	**C**enter **O**f **M**ass
AI	**A**dvanced **I**ntelligence
EAI	**E**nemy **A**dvanced **I**ntelligence

Dedicated to all my friends and family who supported me through all my restless nights throughout my degree....

Chapter 1

Introduction

The key objective of this project is to highlight some of the key concepts behind the creation of games, and intends to streamline the development process as much as possible. The game's functionality will be implemented using the Unity interface and the C# programming language, while the 3D artwork will be created using Blender. It is my intention to create a captivating and dynamic three-dimensional video game, after providing a general guideline on how to similarly do so. It will include characters, character statistics, a movement system, a combat system, a user interface, item management, as well as both original and referenced 3D art.

1.1 Motivation

I have always been passionate about gaming for as long as I can remember. I was introduced to my first video game by my father when I was a young child. Entitled 'Freedom Fighters', the game was a third-person shooter for the PlayStation 2. An alternate reality of the Cold War where the Soviet Union suddenly and successfully invades the United States. As the main character, one is placed at the forefront of a resistance to liberate their country from its occupying aggressors. At a young age, both my brother and I were fascinated at the ability to create a virtual story through such an interactive medium. In spite of the passing of time, this young passion of ours has never waned, but instead only grew stronger. Over time, we expanded to other games, other genres, and eventually other platforms. Our journey progressed from the PlayStation 2, to the Xbox 360, to sharing our first desktop computer, to ultimately building our own exclusive computers.

For me, gaming was a medium to learn many lessons, with each genre offering something different from the rest. Europa Universalis introduced me to the world of administration

and solidified my understanding of history, seeing the world of the past and present from different perspectives and ideologies. Counter Strike helped me learn how to work with a team in competitive environments, in order to win LAN tournaments around the country.

It was only a matter of time before I became fascinated by the technology behind gaming, followed soon by software in general. 'How does this work?', I would always ask myself, while exploring websites and GUIs. In hindsight, I consider myself to have been one of the fortunate few to have shown such a keen interest in a specialized subject at an early age, as it clearly indicated my future career path. In life there are many butterfly effects. If it were not for my father's early introduction to gaming at a young age, I may not be in my final year of my Bachelor of Software Development degree today.

1.2 Contribution

Having completed my Post-Leaving Certificate (PLC) in 2018, a Quality and Qualifications Ireland (QQI) level 6 award, I was introduced to the field of software development. As I gained knowledge of the principles and fundamentals of software, I had the opportunity to interact with Integrated Development Environments (IDEs) for the first time, developing my first relatively simple program in the C# programming language. Upon completion of this course, I continued my studies with the aspirations of acquiring a Bachelor of Science (BSc) Honours degree in Software Development. In doing so, I acquired the theoretical and practical knowledge and competencies necessary to pursue a career in the industry. However, my perception of the purpose of university has always been to teach students how to teach themselves.

The skills I gained during my BSc program proved invaluable in the completion of this study. My experience in modules such as Discrete Mathematics, Modular Programming, and Object-Oriented Programming (OOP) has provided me with an understanding of the importance of logic and modularity within an application's overall architecture. I learned about systematic approaches to performing tasks in a reliable manner through modules such as Embedded Systems and Linear/Non-Linear Data Structures, which introduced concepts such as Finite State Machines (FSM) and general algorithms. General software development relies heavily on programming, yet the method by which it is approached bears a heavy burden on in and of itself. It was through modules such as Agile Processes and Requirements Engineering that I learned the importance of the approach to development.

Game development is a complex and multifaceted process that requires a wide range of skills and expertise. It involves a variety of tasks, including designing game mechanics, creating art and audio assets, programming gameplay systems, and testing and debugging the game. This process can be especially challenging when developing three-dimensional games, as they often require a higher level of technical complexity and visual fidelity compared to two-dimensional games.

Despite these challenges, the goal of this study is to provide a comprehensive overview of the game development process for dynamic three-dimensional games. This includes teaching the reader the methodology required to design and implement the various aspects of the game, from concept to final product. The study aims to provide a clear and concise guide to the game development process, highlighting key considerations and best practices for successful game design and implementation.

1.3 Structure of This Document

Chapter 1 The introduction provides the reader with a clear and concise overview of the overall project, including its motivation, background, and expected contribution. A clear and compelling reason is provided for the reader to engage with the project, providing a clear and compelling context and purpose for the research. This introduction provides an explanation of the motivations behind the decision to conduct research or development on such a niche topic. Moreover, it outlines the primary sources of information or inspiration that contributed to the development of the project, highlighting their most important contributions and insights. Also, the introduction emphasizes the contribution that the project is expected to make to society, illustrating the potential impact the project may have on a particular field or discipline, as well as the broader implications of the project for society at large.

Chapter 2 A comprehensive overview of the various factors that go into creating video games is provided, serving as the basis for the rest of the research presented. By demonstrating how the new game fits within the broader context of the game development industry, it would help establish the importance and relevance of the game. An overview of the steps involved in the design process is provided, including idea generation, play context, the MDA framework, and prototyping, and how they may be used to analyze and design games. There is a distinction made between the conceptual and practical characteristics of a game in this chapter. A game's conceptual characteristics include genre, rules, perspective, space, reality, flow, and balance. The chapter also discusses the practical aspects of games, including

the gameloop, dimensions, physics, controls, graphics, and music. An overview of the context and motivation for the new game would be included in the background chapter in the context of game development, as well as setting the stage for the game's development and release.

Chapter 3 An essential step in the development process is defining the specific goals of the project. It involves identifying the problem or issue that the project aims to address, outlining the specific objectives or outcomes that the project aims to achieve, and specifying the functional and non-functional requirements that must be met for the project to be successful. The process ensures that the project is well-organized, focused, and aligned with the needs and expectations of the intended stakeholders, and provides a clear roadmap for the project team. Assuring meaningful and impactful results is essential for the success of a project.

Chapter 4 . The implementation approach discusses the process that will be followed, the risk assessment and management, and the development of a plan for the implementation. Identifying and classifying potential risks and developing strategies to mitigate them is crucial to the success of the project. Implementation plan schedules outline the tasks and milestones to be achieved throughout the development process and provide a timeline. Also discussed in the chapter is the importance of testing, debugging, and risk assessment.

Chapter 7 The conclusion summarizes the key findings and implications of the study, while also acknowledging any limitations or challenges encountered during the research process. Providing a balanced perspective on the strength and weaknesses of the study through this type of conclusion helps to contextualize the study's results. Furthermore, it provides an overview of potential next steps or areas of further exploration, emphasizing the need to continue to build on the knowledge gleaned from the study.

Chapter 2

Background

From its conceptual foundation, the video gaming industry has grown rapidly, and this trend is not expected to slow down in the near future. A significant increase int he recent participation in gaming culture has been attributed to the COVID-19 pandemic, and its subsequent lockdowns. The diverse range of sub-genres within each generalised genre assures that players of all backgrounds will find an experience tailored to their preferences.

However, despite its growing popularity, an aspiring independent developer does not appear to have a streamlined design and implementation process, easily at their disposal. The fundamental design concepts involved in game development must be emphasized before moving forward with implementation.

2.1 The Design Process

Good design is making something intelligible and memorable, great design is making something memorable and meaningful[**?**]. To coherently portray the intrinsic constituents associated with a game's conceptual design process, one must first understand the underlying concepts behind the deliberate invocation of specific emotional responses. It supports clearer design choices and analysis at all levels of study and development[2]. Thus, some sort of premeditated design methodology facilitates a streamlined process to create an intentionally calculated artifact. Thinking about games as designed artifacts helps frame them as systems that build behaviour via interaction[2], with the ultimate goal of upholding cognitive stimulation.

2.1.1 Idea Generation

Idea generation is the process of creating and developing new ideas or concepts. This can be a creative process that involves brainstorming, idea mapping, and other techniques to come up with original and innovative ideas. However, idea generation should be a continuous cycle[3], with constant revision[4], throughout all stages of development, resulting in a sort of intrinsic feedback loop.

> "Regardless of the exact steps in your process, it's important to keep track of the ideas within it, and the results from implementing them, systematically. This helps in communication and transfer of knowledge, which again helps people in your organization come up with more, and better, ideas. What's more, when you have that information, it can also be used to improve the process itself."[3]

Within this feedback loop, the process of generating ideas typically consists of five key steps:

1. **Identify the problem or challenge:** The first step in generating ideas is to identify the problem or challenge that needs to be addressed. This can include understanding the needs of the target audience, the goals of the project, and the constraints or limitations that need to be considered. Such problems may be uncovered upon conduction of an internal Strengths, Weaknesses, Opportunities, Threats (SWOT) analysis. The technique is used for challenging risky assumptions and identifying blind-spots in organization performance[5]. In the context of game design, a SWOT analysis may highlight certain areas within the game which may require some sort of attention or revision, seeing the problem through the perspective of either the player on the front-end or developer on the back-end.

2. **Brainstorm ideas:** Once the problem or challenge has been identified, the next step is to generate as many ideas as possible. This can involve holding a brainstorming session with a team of people, or working individually to come up with as many ideas as possible. For the most effective results, such brainstorming sessions should be adhere to certain guidelines:

   > "Although there are local variations in how brainstorming is conducted, it is founded on a number of basic ideals, including suspending critical judgement during the session, allowing one speaker the floor at a time, encouraging 'wild' ideas, and building on the ideas of others."[6]

3. **Evaluate and refine the ideas:** After generating a list of ideas, the next step is to evaluate and refine them. An idea that has been refined is a fully developed concept, or a concrete plan that is ready for implementation[3], hopefully eliminating some of its related uncertainty. This can involve looking at each idea in more detail and considering its feasibility, potential impact, and relevance to the problem or challenge at hand. Ideas that are not feasible or relevant can be discarded, while those that show potential can be refined and developed further.

   > "This modern conception has begun to be reflected in educational activities. Consider the popular "process" approach to writing. What mainly sets it apart from older "take up your pen and write" approaches is that a composition is developed over an extended period of time, with revisions and improvements all along the line—to plans, purposes, and to content as well as style of the composition itself."[4]

   Although in the context of creative writing, this core concept of idea generation remains analogous to the game design process. However, an idea is just a theory until it has been refined into something that could actually be implemented[3].

4. **Choose the best idea:** After evaluating and refining the ideas, the next step is to choose the best one. This can involve comparing the different ideas against each other and considering their strengths and weaknesses. The idea that is most likely to be successful in addressing the problem or challenge and meeting the goals of the project should be selected. Nonetheless, available resources must be considered when choosing such an idea, as the resources required for its practical implementation may not be aligned with those presently available. Such resources may include, time, capital, manpower, etc.

   > "The key to selecting the right projects is to focus on the capability that will be required to commercialize the new product or service, not on the product or service itself. That is, the focus should be on the business model that will be required to bring the offering to market, rather than on the technology or product features. When the business model is not yet clear, it is generally best to continue to test and learn, rather than to make a large investment in a specific product or service."[7]

5. **Develop the idea further:** Once the best idea has been chosen, the next step is to develop it further. This can involve creating a plan or roadmap for implementing the idea, as well as identifying the resources and support that will be needed to bring it to fruition. Building and testing a Minimum Viable Version (MVP) of an idea is more important than waiting for it to be perfected before it is launched is

important. Over time, a company can iterate and develop the idea by gathering market data and using it to learn and improve.

> "The key to maximizing the chances of success for a new product or service is to use the Build-Measure-Learn feedback loop. This means that the company should focus on building a minimum viable product (MVP) as quickly as possible, measuring the impact of that MVP on the market, and using that data to learn how to improve the product or service. By focusing on learning, rather than on trying to get everything right the first time, the company can iterate and improve its product or service over time."[8]

Overall, the process of idea generation is an iterative and ongoing process that involves generating, evaluating, refining, and developing ideas in order to find the best solution to a problem or challenge.

2.1.2 Play Context

Play context refers to the specific circumstances in which a game is intended to be played and experienced. How people feel and which devices they use is determined by the context of when and where they play.

> "Play context is determined by the when and the where, and how those factors influence how people feel and which devices they use."[9]
>
> - Jurie Horneman; experienced game developer at Ubisoft

The context of play includes factors such as the genre, setting, characters, objectives, and rules of the game, as well as the player's motivations, expectations, and previous experiences with the game itself or of similar titles. Play contexts can be implied through marketing, or rather people can be encouraged to play in a particular context by marketing.

It is paramount for game designers to create a game by considering the context in which it will be played. This is likely to be more enjoyable and satisfying for players, if it is suited to the real-world situation in which it will be played. A home computer and console game is designed to be played for long periods of time in which players pay close attention, and these sessions recur repeatedly until the game has been completed[9]. In such contexts, however, it may be difficult for a player who has not played for some time to understand what to do or how to proceed. Conversely, mobile and social games are

designed so that they can be played in contexts where real-world fatigue or distraction is expected, since they require less active cognitive attention, or have little to no context of play. If a game is played in a competitive multiplayer environment rather than in a casual single-player setting, players may experience the game differently. Their expectations and motivations may also affect the way they experience the game. Creating games that work in multiple contexts, either through adaptation or collaboration, is the most intriguing approach[9].

In our role as game developers, it is our responsibility to take into account the mindset and situation of players, as well as their general reasoning for playing the game. "A game designed for long sessions in the living room is very different from a game designed to be played during a bus ride."[9]. As platforms and devices have proliferated over the past ten years or so, it is worthwhile spending more time thinking about these questions. The purpose of this is both to create more effective games as well as to identify new market opportunities within specific niches.

2.1.3 Mechanics, Dynamics and Aesthetics (MDA) Framework

The MDA framework serves as the fundamental foundations from which serve as the implementation of the game's intended conceptual outcome, according to its respective play context. A descriptive framework that highlights various elements of meaningful enterprise gamification, and provides an overall synopsis of strategy, design, and user experience elements[10]. It aims to explicate the connections between end-user motivations, interactive gameplay elements,and technology features and functions[10]. It disassembles games into three distinct categories; mechanics, dynamics and aesthetics, where each category maintains close interrelational ties with its bidirectionally adjacent layer(s).

Systematic coherence is achieved upon satisfaction of conflicting constraints, where each of the game's parts can relate to each other as a whole[2]. Decomposing, understanding and creating this coherence requires travel between all levels of abstraction; fluent motion from systems and code, to content and play experience, and back[2].

The MDA framework also helps in conceptualizing the relationship of the designer and the player[10].Designers tend to see from Mechanics to Dynamics to Aesthetics, while players tend to see from Aesthetics to Dynamics to Mechanics[11], as seen in Figure 2.1. From the designer's perspective, the mechanics give rise to dynamic system behavior, which in turn leads to particular aesthetic experiences[2]. From the player's perspective, aesthetics set the tone, which is born out in observable dynamics and eventually, operable mechanics[2]. Hence, the designer's perspective links mechanics to dynamics

and subsequently aesthetics, whereas end users formulate their experiences based on the aesthetics and they engage in specific activities towards satisfying their favoured gratifications.[10]. Thus, even relatively minor updates within one layer may result in a considerable cascading domino-effect on the following layer(s).

For example, a developer may aspire to illicit a sense of dramatic tension upon an end-user, to create a more captivating experience. Dramatic tension comes from dynamics that encourage rising tension, a release, and a denouement[2]. These may in turn be supplemented my mechanics that encourage the requirement of time pressure, for example.

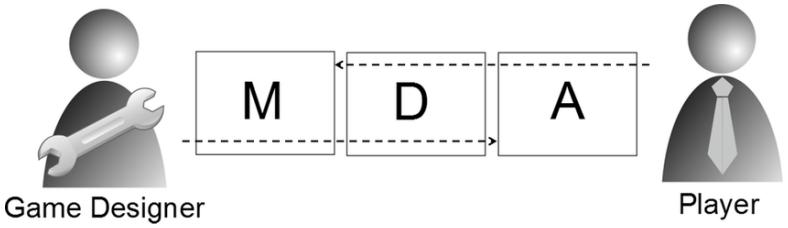

FIGURE 2.1: Graphical representation of the MDA framework perspectives[12]

Mechanics describe the particular components of the game[2] in terms of what actions players can undertake; the processes that drive user actions; and the conditions for progress and advancement[10] afforded to the player within a play context[2], at the level of data representation and algorithms. It includes the realizable actions taken within the limitations of the game engine to support dynamics in gameplay; the shooting of a weapon, for example.

Mechanics describe rules or components implemented in games[**?**], such as basic action, algorithm, game engine, game elements, etc.

Dynamics Dynamics describes the run-time behaviour of the mechanics acting on player input and each others' outputs over time[2], as well as interactions between players[13]. Together, the dynamics of consequences, completion, and continuation establish the basis for a feedback system in gamification to help drive changes in end user behaviour[10], based on their perception of thitherto experience. Randomness (chance) can also be introduced to make the gameplay more dynamic for end users, or to compel users to venture outside their comfort zones[10].

Prior to the end-user's opportunity for interaction, the implication of play context should provide an almost instinctual knowledge of possible activity. It is this play context that establishes a cognitive anchoring point for players to recognize what types of activities they can undertake[10]. However, physical constraints may also be placed upon the player in order to enforce compliance in accordance to the

limitations initially specified by the game's intended rule-set. Dynamics work to create aesthetic experiences[2].

Aesthetics Aesthetics describes the desirable emotional responses evoked in the player, when they interact with the game system[2]. End users formulate their experiences based on the aesthetics and they engage in specific activities towards satisfying their favoured gratifications[10]. The objective designation of "fun" requires a taxonomic breakdown into uniquely identifiable spectrum-based aspects.

1. Sensation - Sense of pleasure through realized actions or achievements
2. Fantasy - Immersion within a make-believe scenario within a virtual world
3. Narrative - Captivating drama through compelling story-telling
4. Challenge - Competitive trials to accomplish tasks
5. Fellowship - A median to engage with social networks
6. Discovery - Exploring uncharted territory
7. Expression - Formulation of decisions based on subjective opinions
8. Submission - Relative devotion through longevity of attention

2.1.4 Prototyping

Prototyping provides the developer with a logical understanding of the presentation of conceptualized information, to the perspective of the end-user. Arguably, usability testing is most effective when integrated into the user-centered design process[11]. Rather than occurring once, used as a quality-assurance tool to evaluate a "snapshot" of the product, it offers more positive feedback when employed as early and often as possible [11].

Paper prototypes (low-fidelity) offer a rough prototype of a game is constructed using paper and pencil, and then consulted with others to garner feedback and iterate on the conceptual design. Early in the design process, paper prototypes are often used prior to the development of any digital or high-fidelity prototypes. Using this approach, generalised concepts may be presented in an efficient and cost-effective manner.

Designers typically begin by sketching out the basic layout and interactions of a game on a piece of paper. To represent the different objects and characters in the game, they may use cut-out paper shapes or other simple materials, and create basic prototypes of the user interface and controls, as seen in Figure 2.2. User feedback is obtained by playing through the prototype and providing feedback on

the gameplay, mechanics, and overall experience. As a result of this feedback, the design of the game is refined and improved, and the process is repeated until the game is ready for digital development.

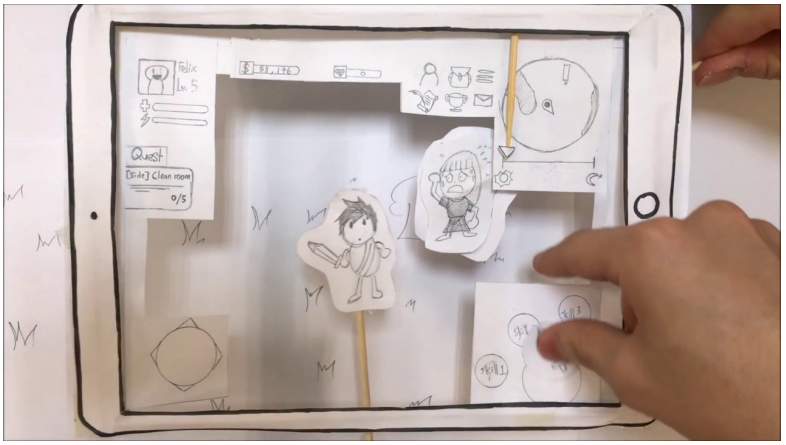

FIGURE 2.2: An example of a paper-prototype, showcasing the general UI and expected gameplay mechanics[14]

Wireframes (medium-fidelity) represent the skeletal structure of a user interface or design, as seen in Figure 2.3. A wireframe is normally comprised of basic shapes and lines that outline the layout, structure, and interactions of a design without including any visual design elements. They are typically created as black and white or grayscale designs, focusing on the structure and functionality of the design rather than its visual appearance.

Wireframes are commonly used in the early stages of design, before visual design and practical characteristics have been developed. Upon interaction with a wireframe component, this component may present additional information or may refer to another interconnected wireframe. They are useful for exploring and testing different interactable design ideas and layouts, as well as communicating to stakeholders and other members of the design team the basic structure and functionality of a design.

Practical Testing environments (high-fidelity) are detailed and functional prototypes of a product or system that closely resembles the final product or system in terms of features, design, and functionality. A high-fidelity prototype provides a realistic and accurate representation of the final product or system, and can be used to test and evaluate its design and functionality. They are impractical at lower stages in the design process due to resource limitations, only to be produced in the active implementation phase.

In the context of game development, a high-fidelity prototype is a detailed and functional simulation of the final game in terms of gameplay, mechanics, visual

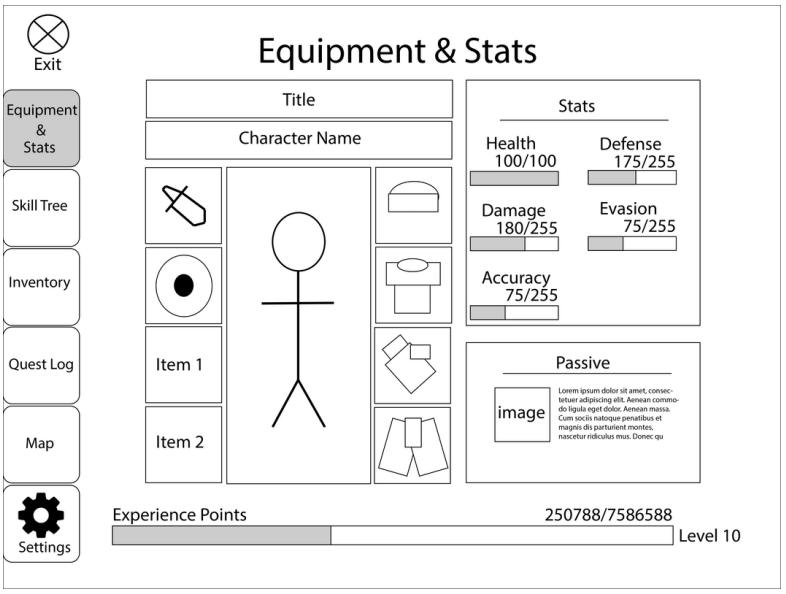

FIGURE 2.3: An example of a wireframe, showcasing a character's equipment GUI and expected component functionality[**?**]

design, and audio design. They serve as the medium for the real-time testing of practical game characteristics, to showcase working game dynamics, such as physics or user-control, as seen in Figure 2.4. Prototypes of high fidelity can be used to test and evaluate the gameplay, mechanics, and design of a game, and can provide a realistic and accurate representation of the final product.

FIGURE 2.4: An example of a high-fidelity prototype[12]

2.2 Conceptual Characteristics

Conceptual characteristics focus on ideas and concepts rather than the physical implementation of challenges or competitions. As a means of representing abstract ideas or concepts, they often use abstract or symbolic elements, such as icons, images, or words, to represent them.

Players are provided with a visual representation of a particular topic or concept as well as the process involved in stimulating their cognition or intellect. Using creative thinking to approach challenges in a creative way, or requiring players to think outside the box. Making decisions based on a limited amount of information, solving puzzles, or formulating a plan of action fall within this category.

Often, narrative or storytelling elements are among the most engaging aspects of games, which can help to engage players and make them more immersed in the experience. In order to create a sense of adventure in a story, there are a variety of ways to do so, such as using characters, dialogue, or twists. Ultimately, this results in a captivating sense of intrigue.

2.2.1 Genre

Genres are categories or types of games based on their specific characteristics, gameplay mechanics, and overall style. A game genre can help players and developers understand the essential elements and features of a particular game by categorizing it. Among the most common types of games are action, adventure, role-playing, simulation, strategy, and sports games. Among these genres, each has unique characteristics that set it apart from others. The gameplay in an action game, for example, focuses primarily on quick reflexes and intense combat, while an process-oriented game focuses primarily on exploration and mastery.

	Action	**Adventure**	**Strategy**	**Process-oriented**
Action	Battle	Mystery solving	Build nation	Exploration
Success	Fast reflexes	Logic ability	Analyzing variables	Varies widely

TABLE 2.1: The Four Genres; Understanding Video Games[1]

Action games are a type of video game that typically involve physical drama and require motor skills and hand-eye coordination to be successful. Examples of action games include Pac-Man, Half-Life 2, and MotorStorm. In some action games, the player only needs to coordinate the movement of the on-screen character and does not

need to think about the correct choices to make. In other, more complex action games, the player must also solve spatial puzzles and figure out how to complete challenges through a series of physical actions.

Adventure games are a type of video game that require deep thinking and patience to play. These games often involve a narrative based on detective stories, and the player is typically represented by a character who must solve puzzles and mysteries. Adventure games may lack fighting or action sequences and may not have any risk of the main character dying. To succeed at adventure games, players must use their logic and deduction skills. Examples of adventure games include Adventure, Maniac Mansion, and Dreamfall: The Longest Journey. Single-player role-playing games, such as Ultima, Wizardry, and Baldur's Gate, are also considered a subgenre of adventure games, although they may also have elements of strategy.

Strategy games are a genre of video game that occupy a space between action and adventure games. In these games, the player often takes on the role of a general, mayor, ruler, or deity, and the conflict is typically represented on a map. There are two main sub-genres of strategy games: real-time strategy and turn-based strategy. Real-time strategy games involve balancing a large number of variables and paying attention to other players' choices and strategies, and they are played in real-time or continuous-time. Examples of real-time strategy games include Dune II, Warcraft, and Dawn of War. Turn-based strategy games, on the other hand, involve players making choices in turns, similar to board games like chess or risk. Examples of turn-based strategy games include Balance of Power, Civilization, and Warlords.

Process-oriented games are a type of video game that do not have a clear goal or objective, but instead provide the player with a system to interact with and manipulate. These games are often labeled as "games" for entertainment purposes, but they could also be considered toys. There are two main approaches to designing process-oriented games: one type involves the player exploring and manipulating a dynamic world, while the other type puts the player in charge of more fundamental variables. Process-oriented games do not have a consistent criterion for success, but each game encourages certain types of play. Examples of process-oriented games include Elite, The Sims, and Zoo Tycoon.

Further sub-genres and hybrids can also be created by combining game genres, such as role-playing games with strategic aspects, or sports games that incorporate simulation elements. Having a basic understanding of game genres can be useful for players when

choosing a game to play, and for developers when designing and creating a new game for a particular target audience.

While some authors prefer to emphasize player activity (or game play, or interaction) over all other aspects of video games, this has limited the scope of what can be (or should be) included in a study of a single game or genre. [15]. With regard to interactive media, the seemingly minor or secondary nature of non-gameplay elements has also raised questions regarding interpretation and meaning. [15].

2.2.2 Rules

Having rules provides the structure, constraints, and challenges that define a game and make it enjoyable to play; a specific sort of immaterial support. A game's rules provide the framework within which players interact with the game and with one another. It is the rules that determine the boundaries and challenges that players must navigate and overcome in order to achieve success in a game, providing the game with meaning and purpose. The determination of what moves and actions are permissible, and what they will lead to the open-endedness of a game is not necessarily dependent on complexity[16]. They provide a set of expectations and guidelines that players can use to understand and interact with the game.

Rules can also serve as an important component of the game's aesthetic. In addition to contributing to the overall enjoyment and satisfaction of the game for the player, the specific rules and constraints of the game can influence its style, tone, and atmosphere.

In the second chapter of his book, Half Real: Video Games between Real Rules and Fictional Worlds[16], Jesper Juul compares multiple perspectives and definitions of game design and combines them into a formal definition he calls the "classic game model"[17].

According to this model, games contain six distinct features[16]:

1. Fixed Rules
2. Variable, quantifiable outcomes
3. Values associated with each outcome
4. Player effort
5. Player attachment
6. Negotiable Consequences

2.2.3 Perspective

Game perspective describes how the player perceives the game world and its elements, it is integral to the gameplay experience. Changing the game perspective can significantly impact a player's experience of a game, since it can impact his or her ability to perceive, interact with, and comprehend the game world. Excluding text-based and abstract puzzle games, all games employ the perspective of either first-person, second person, third person or top-down/isometric[1]. Each of these perspectives has its own unique characteristics and advantages, and can be used to create different types of gameplay and player experiences.

Games are generally classified according to the perspective they use, and as such include the following;

First-person games are presented from the Point-Of-View (POV) of the player's in-game character. Players can immerse themselves into the world of the virtual world by experiencing the game through the eyes of their character, allowing them to create games that emphasize exploration and discovery. It is anticipated that the player feels as if they are almost taking on the role of the character in the game.

Second-person games refer to the perspective of the player's character through the perspective of a non-playable character or entity. Despite this out-of-body perspective, the game's narrative and dialog will still refer to the character as "you". Using the second-person perspective in video games is relatively uncommon, as creating a compelling and engaging player experience can be challenging relative to its counterparts. However, it should be noted that there have been some games that have created unique and immersive gameplay experiences by using the second-person perspective, or at the very least temporary utilization of it.

Third-person games feature a positional camera that is independent of the player as a separate entity and positioned relative to the in-game character. The player watches whatever it is that they are controlling[1]. The camera typically tracks the in-game character directly behind and over their character's shoulder. It is possible for the player to not only control their characters, but also generally manipulate the camera's movements. Utilization of such camera movements may offer the player a greater sense of situational awareness. In a wide variety of game genres, such as action, adventure, and role-playing games, third-person perspective is a common type of game perspective. Players are able to see their character from a distance, enabling them to gain a broader perspective of the game world and to engage in more complex and dynamic movement and action.

Top-down/isometric games offer an elevated viewpoint above the game world, as if the camera was looking down from an aerial position, a "bird's eye view" per se. This type of perspective is common amongst strategy games, as it provides a general in-game context overview to facilitate micro-management.

Essentially, an isometric game employs isometric perspective, which is a form of 3D projection where the camera is positioned at an angle above the game world. Despite the fact that this type of perspective creates the illusion of three-dimensional space, objects in the game world are still represented with two-dimensional graphics. Isometric games are often used in role-playing games, real-time strategy games, and other types of games that require a more detailed view of the game world.

Top-down and isometric games are similar in that they both use a non-perspective view of the game world, but they differ in the way the game world is represented. Top-down games use a pure two-dimensional view, while isometric games employ a pseudo-3D view.

2.2.4 Space

The gamespace in a video game refers to the virtual world in which the game takes place.. It can include the environment, objects, characters, and other elements that make up the game world. The gamespace is the space within which the game's mechanics, dynamics and aesthetics operate. A virtual space in which the player's avatar moves and interacts with other objects and characters. In general, it sets the stage for the player's experience and defines the boundaries within which the player can interact with the game world.

The perspective from which a player perceives the gamespace in a video game should not be limited to just first- or third-person, but rather be considered as a point of perception that may vary and overlap in different ways. An example of this is the use of maps in real-time strategy games, which provide a wider view of the gamespace while the player can also choose their own standard view[1].

> "As we follow the historical evolution of video game design, we should increasingly not cling to a strict division between first- and third-person perspective; rather, we should discuss a game's point of perception, the point from which the player perceives the gamespace. Importantly, a game may offer varied, often overlapping, points of perception. Real-time strategy games, for instance, offer maps of the entire gamespace, which coexist with the standard view chosen by the player."[1]

It is important to note that the gamespace may vary in size and dimensions depending on the type of game and the platform on which it is played. Depending on the game, a player may be able to explore and interact with a wide variety of objects and characters in a large, open-world gamespace or in a smaller, more focussed space designed to be completed quickly.

Side-scrolling video game consists of a vertically scrolling screen on which the action takes place, a gradual unveiling of space. The game's environment scrolls along with the player's movement to create the illusion that the player is moving forward in a side-scrolling game. During the early days of video gaming, side-scrolling games were popular due to the limitations of the hardware. However, side-scrolling games remain popular, and many modern games use this perspective to create a nostalgic, retro atmosphere.

Linear gamespace is characterized by a more structured gamespace and a predetermined path through the game, typically utilizing a level-based progression system. For a player to progress through a linear game, they must complete a specifically designed roadmap of goals and objectives.

Non-linear (open-world) gamespace is vast and open, generally in three dimensional space. It allows players to explore and interact with the game world in a non-linear manner. A sandbox-style gameplay is often employed in open-world games, where the player is free to choose their own goals and objectives, as well as how to achieve them. The world may be open to discover, whether it be through forced or non-forced exploration. Regardless of whether the player explores the world forcibly or unforcedly, the world is available for discovery nonetheless.

2.2.5 Reality

Game reality refers to how a game represents and simulates the game world. This concept encompasses the visual, audio, and tactile elements that make up a game world, as well as the rules and mechanics that govern how players interact with it. Depending on the genre, style, and target audience of the game, different approaches to game reality are employed. Game reality consists of the elements that make up the game world, as well as how those elements are experienced by the player. It may be affected by the perspective of the player as well as the technology used to create the game. In contrast to traditional 2D games, virtual reality games will have a very different sense of reality.

Traditional Alternate Reality Games (ARG) depict the traditional way in which players have interacted with systems since the conceptual foundation of video

games. Physical displays such as monitors and televisions are generally positioned perpendicularly to the user, which serve as the output medium for in-game context.

Augmented Reality (AR) allows users to view and interact with digital information and content in the real world. It requires some sort of sensory medium which serves to bridge the gap between simulation and reality. It overlays digital information and content onto the user's view of the physical environment around them. A seamless integration of digital and physical elements is a technology which has the potential to change the way we see and interact with the world around us.

Virtual reality (VR) allows users to experience the game world in a fully immersive manner, moving and interacting as if they were physically present. It is currently necessary to wear VR headsets in order to experience virtual reality; however, future advances may enable this to become obsolete in a manner that is currently deemed unimaginable. A VR headset covers the user's eyes and displays a 3-dimensional environment. Through specialized controllers, users may interact with this digital environment, moving around, looking at objects, and interacting with them.

2.2.6 Flow

In the field of game design and psychology, "game flow" refers to a game's overall flow and pacing, as well as the player's experience of immersion and engagement. It is a state in which people are so involved in an activity that nothing else seems to matter; the experience is so enjoyable that people will continue to do it even at great cost, for the sheer sake of doing it.[18]. It is a feeling of complete and energized focus in an activity, with a high level of enjoyment and fulfilment"[18].

According to Csikszentmihalyi's research[18], it emerged that masters of a demanding skill, such as surgeons, were not motivated primarily by money or prestige, but by the exhilaration experienced when performing tasks that were just within their ability. These tasks are 'autoletic' (auto = self, telos = goal); task undertaken for their own sake.

Key components in the flow state:

1. Clear goals
2. No distractions
3. Direct feedback

4. Continuous challenge

According to Csikszentmihalyi, the narrow margin of a challenge lies between boredom and frustration[18]. To what he described as the "flow channel" this margin of challenge is the optimal experience for a user. As seen in Figure 2.6, the degree of challenge introduced to the end-user should be directly proportional with their growing level of skill, allowing for the existance of a "flow channel".

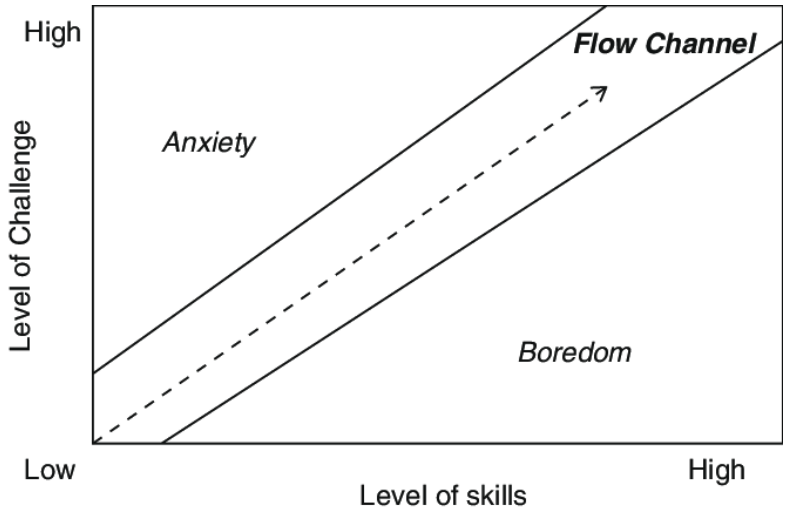

FIGURE 2.5: Csikszentmihalyi's 'Flow Channel'[12]

Csikszentmihalyi outlines claims the existence of 8 elements common to those who enter into a 'state of flow', or who have an optimal experience. Such elements include the following[18]:

1. A challenging but tractable (controllable) task to be completed.
2. One is fully immersed in the task, no other concerns intrude.
3. One feels fully in control.
4. One has complete freedom to concentrate on the task.
5. The task has clear unambiguous goals.
6. One receives immediate feedback on actions.
7. One becomes less conscious of the passage of time.
8. Sense of identity lessens, but is afterward reinforced.

2.2.7 Balance

A game's balance refers to the process of ensuring that all elements are equal in power to one another. This ensures that no one element is overpowered or underpowered relative to the other elements. The balance of strengths and weaknesses of various characters, items, or abilities in a game may be achieved by balancing the difficulty level of different levels or challenges within the game. As the target group's preferences dictate, the perceived difficulty should ideally stagnate or increase throughout the game[**?**]. During the development process, this may be accomplished through extensive official playtesting, or it may be accomplished through community feedback after the release of the product.

The following steps are commonly followed by game designers and developers in order to achieve game balance:

1. **Establish the game's goals and objectives:** In order to achieve game balance, it is crucial to identify the game's goals and objectives. Identifying the target audience, defining the intended gameplay experience, and stating the overall vision of the game can be included in this process.

2. **Develop and test game mechanics**: Once the goals and objectives of the game have been established, the next step is to develop and test the mechanics of the game. In order to determine how the game will work in practice, prototypes can be created and playtested. The game mechanics can then be refined and improved by collecting feedback from playtesters.

3. **Balance the different elements of the game:** After the game mechanics have been designed and tested, the next step is to balance the different elements of the game. It may involve adjusting the strengths and weaknesses of specific game characters, items, or abilities, as well as adjusting the difficulty level of specific game levels or challenges. Often, this process involves a lot of trial and error, as well as collecting feedback from playtesters and making adjustments accordingly.

4. **Continually monitor and adjust game balance:** Maintaining game balance is not a one-time event, but rather an ongoing process that needs to be monitored and corrected continuously. As the game is played by a growing number of people, it is vital to collect feedback in order to adjust the balance of the game accordingly. The game mechanics, game elements, or overall gameplay experience can then be modified accordingly to maintain fairness and balance for all players.

Having the right level of game balance is crucial for competitive games, as it ensures that every player has an equal chance of winning, according to their relative skill level or their

character or strategy selections. When a game is balanced, players are able to choose from a variety of different options and strategies and still have a fair chance of success, which can help to increase the overall enjoyment of the game. Conversely, if a game is not balanced, certain strategies and characters may have a significant advantage over others, resulting in frustration and lack of enjoyment. There is a particular importance to competitive balance in team games since it is influenced both by intra-team dynamics and dynamics between teams[19]. Unbalanced competitive games also tend to lead to a lack of diversity in competitive play, as players may feel compelled to adhere to an objectively superior 'meta' by using the most powerful strategies or characters.

Multiplayer games such as Overwatch, in which character abilities are carefully balanced so that no character is significantly overpowered or underpowered, when compared to other players.

Strategy games such as Civilization, requiring careful resource management and calculated decision making.

Fighting games such as Street Fighter, where the balance of different character moves and abilities is crucial to the competitive nature of the game.

Role-Playing Games (RPG) games such as World of Warcraft, where the balance of different character classes and abilities is important for creating a fair and enjoyable gameplay experience for all players.

Overall, achieving game balance is a crucial part of game design and development, as it ensures that all players have a fair, enjoyable and diverse gameplay experience. By following a systematic process and collecting ongoing feedback from playtesters, game designers and developers can create games that are well-balanced and enjoyable for all players.

2.3 Practical Characteristics

It is the pseudo-sensory elements within the game world that constitute its practical characteristics. Sensory pseudo-tangibility refers to something that, although not technically real, can still be perceived by the senses. They have a direct impact on the ability of players to comprehend the game, relative to the aspirations of the developer while adhering to the specified requirements.

2.3.1 Game Loop

The gameloop is a fundamental part of every video game[20], existing as the repeating sequence of information package transit at regular intervals to enhance real-time game-state accuracy. The game loop typically consists of three main steps: input, update, and render. In the input step, the game collects and processes any user input from the player. In the update step, the game uses the input to update the game state, such as moving characters or objects, checking for collisions, and so on. In the render step, the game uses the updated game state to generate and display the game visuals on the screen. This sequence of steps is then repeated continuously until the game ends. The game loop is an essential part of the game's architecture and is responsible for managing the flow of information and actions in the game.

2.3.1.1 Ticks

The gameloop consists of a collection of 'ticks'. A tick is a unit of simulation time in a networked game. It is a step in the game's simulation that is broadcast to all connected clients in order to help them synchronize with the server. A high tick rate, or a high number of ticks per second, gives players a more accurate representation of the game world. This is because the server sends updates more frequently, so players see changes in the game world as they happen, rather than a few steps behind. However, a high tick rate also requires more bandwidth and CPU power on both the client and the server[21], and may not necessarily improve the player experience if the client cannot handle the amount of data being transmitted. The most important networking variables for clients in games running on the Source engine, according to Valve developers, are their incoming bandwidth capacity and the snapshot rate (refresh rate) they prefer to play at[22].

2.3.1.2 Tickrate

The tick rate (or framerate/update rate), of a game refers to the number of times per second (frequency) that the server sends updates to connected clients. It is measured in Hertz (Hz) and is an important factor in the responsiveness and accuracy of the game. In the gaming industry, the tick rate of a game is typically 20Hz, 30Hz, 60Hz, or 120Hz, although some games offer adaptive tick rates that adjust based on the client's bandwidth capacity. An example of a game with a high tick rate is CounterStrike: Global Offensive, which offers servers with a tick rate of 128Hz for competitive events. However, there may be diminishing returns in terms of responsiveness at higher tick rates due to the limitations of human perception[23]. Some games also offer the option

for players to adjust the tick rate themselves or to limit it in order to reduce bandwidth consumption.

In order for a game to be smooth and responsive, the tickrate must be considered. Higher tickrates result in a smoother and more fluid game experience, since the game updates and renders the game state more frequently. Higher tick rates, however, require more processing power and place increased demands on the player's computer, which at a certain point may become relatively superflouos. Consequently, the ideal tickrate for a game varies depending on its specific characteristics and the level of smoothness and performance desired.

2.3.2 Dimensions

Physical game dimensions refer to the ways in which a game represents and simulates the tangible properties of the game world. In physics, dimensions refer to aspects of a physical quantity that can be measured; length, breadth, depth, etc.

Two-dimensional (2D) games represent the game world by two-dimensional graphics, where objects and characters are represented as flat images using tile-mapping. In 2D games, the player's movement is usually restricted to left, right, up, and down in a side-scrolling or top-down perspective. Side scrolling games are popular among this dimension type, due to the lack of three-dimensional movements.

Three-dimensional (3D) games depict the game world using three-dimensional graphics, with objects and characters being represented as three-dimensional models with depth, width, and height. In 3D games, the player's movement is usually more complex and dynamic, allowing for movement in all three dimensions.

Two-and-a-half dimensional (2.5D) games are also referred to as 'pseudo-3D', combining aspects of both 2D and 3D alike. 2.5D games present the background environment and constraining directional movements in 2D, while characters and objects are displayed in 3D. As a result, it is possible to create a more detailed and realistic visual style than what is possible with pure 2D graphics. This is while still retaining the simplicity and ease of development of a 2D game. These games typically feature a side-scrolling or top-down perspective, and movement is typically limited 2D. However, some 2.5D games may also allow for limited movement in 3D, allowing the player to move closer to or further away from the game world, relative to the player camera.

2.3.3 Physics

In game physics, mathematical simulations are used to produce realistic and believable physical behavior and interactions. The simulation may include the movement of objects in the game world, their interaction with each other, and the effects of various physical phenomena, such as mass. Game physics is integral to game development, as it facilitates the creation of more immersive and realistic game worlds and interactions. Additionally, it can provide players with challenges and obstacles to overcome, such as objects that can be moved or manipulated to solve puzzles or reach new areas.

The physics of a video game are usually implemented using physics engines, which are software libraries that provide tools and algorithms for simulating the behavior of physical objects. To produce more realistic and believable simulations, physics engines use mathematical equations and simulations to calculate the effects of forces and collisions on objects in the game. For example, a developer may aspire to calculate the mass of a rigid-body, along with its respective Center of Mass(COM), in order to determine how it collides with other bodies.

Rigid-bodies are solid objects that move separately from other elements within the game-world when physically acted upon. They have a fixed shape and do not deform or change shape when external forces are applied to them. They are like extensions of particles because they also have mass, position and velocity[24]. Furthermore, they can rotate due to their volume and shape.

Mass of a body is the measurement of matter within its volume. For a 3D object, the mass is therefore the integral over its volume along the three dimensions[25]. Using Figures 2.6 and 2.7, the mass and center of mass may be calculated respectively, where 'ρ' (rho) is the density at each point within the body, 'r' is the position vector of each point, and 'V' is continuous volume.

$$m = \int\int\int p(x, y, z)dV$$

FIGURE 2.6: Calculating mass of 3-dimensional object[**?**]

Center of Mass (COM) refers to the rotation point of a body. A rigid body naturally rotates around its Center Of Mass (COM), and the position of a rigid body is considered to be the position of its center of mass[24]. It is the mid-point of the mass distribution of it's respective body. A 3-dimensional rigid body is made up of continuous particles, thus its COM is calculated using an integral.

$$COM = \frac{1}{M} \int_V \rho(r) r . dV$$

FIGURE 2.7: Calculating COM of 3-dimensional object[12]

Collision detection consists of finding pairs of bodies that are colliding among a possibly large number of bodies scattered around a 2D or 3D world[26]. In the context of rigid body simulations, a collision happens when the shapes of two rigid bodies are intersecting, or when the distance between these shapes falls below a small tolerance[26]. Different approaches can be used to calculate collision detection in games, depending on the specific requirements and characteristics of the objects.

1. The shape of objects can be approximated by using bounding boxes or spheres, which are simple geometric shapes. It is possible to determine whether two objects are colliding by comparing the locations of their bounding boxes or spheres.
2. In more complex collision detection calculations, geometric algorithms such as the Separating Axis Theorem[27] or Minkowski Sums[28] may be used. Calculating the precise intersection of the shapes of two objects may be accomplished with these algorithms. The accuracy of these methods can be enhanced, however, they may also require more computational resources and may be inefficient when dealing with a large number of objects at once.

2.3.4 Controls

Controls in video games refer to the methods by which players interact with and manipulate the game. They are an integral part of game design, as they determine how players can perform actions within the game and directly affect the outcomes of the game. Players' experiences can be greatly affected by the design of controls, as poor controls can lead to frustration, difficulty learning how to play the game, and a lack of enjoyment. Conversely, well-designed controls that are easy to understand, intuitive, responsive, and accurate contribute to a smooth and enjoyable gameplay experience. Simple control schemes can become second nature to players[29]. It is imperative that the controls be designed taking into consideration the target audience, the genre and mechanics of the game, the input devices available, and the desired level of complexity and depth of the game. Various types of controls can be used to suit different game types and platforms, including keyboard and mouse controls, gamepad controls, and touch

screen controls. To ensure that the controls are effective and enjoyable for players, game designers need to carefully consider and test them during the game development process.

2.3.5 Graphics (Art)

Generally speaking, game graphics refer to the visual elements of a game, such as the game world, characters, objects, and other visual elements. In game design, graphics play a significant role in how the game world appears and how the player directly perceives and interacts with it. An immersive and realistic game world can be enhanced by high-quality graphics, making it feel more believable and engaging. Moreover, they can contribute to the depth and richness of the game's story, characters, and setting. It should be noted, however, that poor graphics do not ruin a great game, whereas outstanding graphics do not necessarily improve a mediocre game[30].

Various techniques can be used to create game graphics, such as 2D graphics, 3D graphics, vector graphics, and pixel art. Depending on the genre, style, and intended audience of the game, a particular technique will be employed. Retro platformers may use pixel art to create a nostalgic, retro feel, while realistic racing games use 3D graphics to create highly detailed and realistic game worlds. It is important to recognize that technology is an integral part of the creation of game graphics. The graphics capabilities of a game played on a high-end gaming PC are higher than those of a game played on a mobile device, for instance.

2.3.6 Music

Music plays an important role in the aesthetics of a game as well as enhancing the gameplay experience. It contributes to the sense of immersion and engagement in a game, as well as establishing the character and tone of the game. Depending on the genre and style of the game, music can be used to create different moods and atmospheres. A well-designed musical score can create a sense of tension and excitement in a fast-paced action game, as well as calm and tranquility in a more relaxed puzzle game. A game designer can create a unique and distinctive atmosphere for the game by carefully selecting and composing the music, and this can help establish the game's identity and distinguish it from other games in the same genre.

The manipulation of musical elements, such as melody, harmony, rhythm, and timbre, can elicit emotional responses from listeners. Several biochemical substances may be targeted, including cortisol, oxytocin, dopamine, and serotonin[31] by doing so. Listeners can experience a wide range of emotions based on a combination of these elements,

ranging from excitement and joy to sadness and melancholy. Fast and energetic melodies can create a sense of excitement and joy in listeners when combined with lively rhythms and upbeat harmonies. Meanwhile, slow, mournful melodies, coupled with somber harmonies and rhythms, can evoke sadness and melancholy.

Chapter 3

Streamlining The Design And Implementation Process of Dynamic Three-Dimensional Game Development

3.1 Problem Definition

Developing a 3D video game using Unity and C# can be a complex and challenging task, requiring a range of skills and knowledge in areas such as game design, programming, and asset creation. The Unity development platform is a popular choice for game development due to its powerful tools and resources, including a visual scripting system called "Unity Play Mode," which allows developers to easily test and iterate on their game designs.

However, simply having access to these tools and resources is not enough to ensure a successful game development project. It is essential for developers to have a clear understanding of their goals and objectives, and to follow a coherent and well-planned development process. This includes defining the game's concept and mechanics, creating a detailed game design document, prototyping and playtesting to refine the game's design, and implementing and debugging the final product.

This research paper aims to provide a comprehensive practical guideline for creating a 3D video game using Unity and C#, with a focus on the key considerations and best practices that should be followed throughout the development process. The accompanying playable game will serve as a practical example of the effectiveness of following

this methodology. A means to an end using Unity and C# together with a systematic approach to demonstrate their potential as powerful game development tools.

3.2 Objectives

This project aims to create a fully-functioning video game through a series of iterative high-fidelity playable builds, each introducing new dynamic features and incorporating unique 3D art created using Blender. The development process will begin with the creation of a basic "bare-bones" build that serves as a testing environment for player movement mechanics. Subsequent builds will introduce additional features such as combat and item interaction.

The game will start with the player character, introduced as a prisoner in a dungeon, and will involve a tutorial quest to escape. The tutorial will teach basic mechanics such as camera control, movement, combat, item interaction, and GUI navigation. Quest markers will guide the player towards relevant objectives, including targeting specific enemies or collecting required items.

Throughout the game, the player will encounter various items that can provide permanent stat bonuses, temporary buffs or debuffs, or are necessary to progress to new areas. The player will have the option to either fight or sneak past enemies, but may need to consider their capabilities and resources before engaging in combat.

The project will use "Semantic Versioning 2.0.0" within it's respective Github repository to clearly label the iterations of the game, with the first "major" release being the completion of the initial set of functional requirements, and subsequent "minor" releases representing functional deliverables. Work between these deliverables will be considered "patches." The final "major" release will be the completion of the tutorial quest.

3.3 Requirements Engineering

As the name implies, requirements engineering is the process of identifying, analyzing, and documenting the requirements for a system or product. In the context of software development, requirements engineering involves working with stakeholders to identify the needs, goals, and expectations for a software system, and then defining and documenting the requirements that the system must satisfy in order to accomplish these objectives.

An essential function of requirements engineering is to ensure that the requirements for a system or product are well-defined, comprehensive, and consistent, as well as to ensure that the system or product is designed and developed to meet those requirements. During the software development process, requirements engineering is an important component, as it helps ensure that the resulting product or system meets the stakeholders' expectations.

In the context of game development, requirements engineering involves working with game designers, artists, and other stakeholders to identify the game's requirements and then defining and documenting these requirements clearly and concisely. As a result, the game can be designed and developed to meet the needs and expectations of the stakeholders and players, ensuring that the player experience is satisfying and enjoyable.

3.3.1 Functional Requirements

Functional requirements define the essential core functions and capabilities of a game, with respect to to the overall design and proposed architecture, implementation, and operation. They may include specific gameplay mechanics, user interface elements, performance benchmarks, and other features and capabilities that are necessary for the game to function properly and achieve its design objectives.

Typically, functional requirements are documented as part of the game design process, and may come from a variety of sources, including market research, user feedback, and industry standards. They are used to determine the scope and constraints of a game development project, and to guide the design and implementation of the game.

Functional requirements are an integral part of the development process. As well as ensuring that the game meets the expectations and needs of both players and stakeholders alike, they provide a clear and measurable set of criteria and goals that can be used to evaluate the game's quality and success.

In the implementation phase of this research paper, the respective game will attempt to meet the following functional requirement criteria:

Character Objects There are many types of characters in a game, all of which inherit from the generic 'character' super-class; the player, an enemy, and a friend. Each individual character has a unit model, a visual representation of the character object. Each character object contains a list of dynamic statistics, affected by external sources such as combat, movement etc.

Character Statistics Each character object contains a multitude of dynamic statistics, affected by how its respective object interacts with the world around it; health points, stamina points, magic points, maximum carry capacity. These statistics determine the current state of the character, and whether or not it can perform certain actions, given its respective type-points available.

Movement Each character has the ability to move around on a 3-dimensional grid. Whether it be to move from one area to another, or to evade enemy attacks, it is possible to walk, run, sprint, jump, fall and roll. Utilization of a technique known as 'ray-casting' will aid in the physics of character/item falling.

Camera In a 3-dimensional game, the player camera is required to host the functionality of a dynamically-changing position upon multiple axes, responding to external variables such as player movement and environment collision.

Items There will exist an abundance of items in the game, each of a different type. There are weapons, armour, consumables, quest items, etc. Such items may appear around the world environment, able to be picked up by the player. Each item has its own individual stats, such as weight, damage, price, etc.

Combat An ability bar will provide a list of abilities available to the player. Each ability has a different function, to inflict damage, to heal, to apply a buff/debuff (a stat affecting passive ability to either the player or target character). The amount of damage inflicted on another character is determined on the statistic provided from the attacker's equipment, as well as the 'spell' used.

Inventory Items picked up by the player will be placed into their inventory. A Graphical User Interface (GUI) will display the contents of this inventory, along with the statistics of each item. The amount of items a player can carry is limited by their 'maximum-carry-weight' statistic.

Equipment As mentioned before, some items are equip-able, such as weapons and armour. An equipment GUI will display to the user the currently equipped items, the statistics for each individual item, along with the total statistics provided to the player by the entire equipped armour-set as a whole.

Item Containers Items may be located within a container, such as a chest. The player must open such a container to display its contents. Items within this container may be picked-up by the player and placed into their inventory.

Compass A compass aids the player in understanding the route in which they may need to take, in order to achieve an objective. This compass not only shows the direction in which the player is facing, but also offers a marker to the next objective

points. This marker may be set to specific individual objects (e.g to open a door, pickup an item, etc.), the compass will direct the player to the exact location of this object.

Animations Each action taken from a character must be supplemented by an animation. The concept known as 'animation blending' refers to the smooth transition between two or more animations enacted upon a singular skeletal mesh. Employment of 'animation masking' will allow certain parts of the skeletal mesh to override certain parts of the body with another animation. For example, in the case that a character equips a one-handed sword during a running animation, the newly equipped hand will operate an 'unsheathe' animation, while the opposite hand continues to run the sprinting animation.

Sneaking Rather than fighting an opponent, the player may wish to avoid combat altogether by avoiding detection from enemy units. Such a mechanic would require the player to enter the 'sneak' stance by crouching. At the expense of a movement penalty, the player can slowly move past a target. If the player is out of the enemy unit's 'line-of-sight', the effect of the sneak will improve.

Quests Throughout this game, the player will be faced with many quests. Once accepted, one must reach several checkpoints to reach quest completion. The game begins with a tutorial quest, teaching the player the basic mechanical fundamentals; how to move, how to interact with the User Interface (UI), etc.

Unique 3d art Using Blender, I will attempt to create unique models for characters and items. When imported into Unity, they will become both viewable and interactable to the player. There will, however, be a limitation on the amount of original artwork I can create, since I am limited by time, so I may also use referenced material from other artists.

Game State Persistence It should be possible for a player to save and restore their progress and actions within the game, allowing the player to continue from where they left off when they return to the game. Once the game is relaunched, their previous game state should be reloaded, rather than rebuilding a new game instance. For games which cannot be completed within a single session, this is particularly important.

Enemy Advanced Intelligence Using Finite-State Machines (FSM) - Creating an enemy character using Enemy Advanced Intelligence (EAI) and Finite-State Machines (FSMs) will undoubtedly prove to be a challenging task, however, it will certainly be rewarding as well. An FSM is a powerful technique for implementing

artificial intelligence (AI) in games, as they allow developers to specify the different states and transitions of an AI character, as well as the actions and behaviors associated with each state.

Implementing EAI for an enemy character using FSMs can be a complex process, as it involves designing and implementing the enemy character's different states and behaviors, as well as defining the conditions that trigger transitions between states. It may be necessary to have a good understanding of game AI techniques as well as a substantial amount of planning and testing if the AI character is to behave in a realistic and engaging manner, but the resulting AI character can be a compelling and engaging part of the game.

3.3.2 Non-Functional Requirements

Non-functional requirements in a game are those that determine the overall quality and performance of the game but have no direct relationship to its specific features or functions. They may relate to aspects such as the overall performance of the game, stability, reliability, usability, scalability, and maintainability. They serve as constraints or restrictions on the design of the system across the different backlogs. In addition, they may relate to more subjective factors such as aesthetics and user experience.

Generally, non-functional requirements are derived from a variety of sources, including market research, user feedback, and industry standards. Defining quality and performance expectations for a game can help ensure that it meets the needs and expectations of players and other stakeholders. With adherence to the non-functional requirements, a game can be made more sustainable and successful in the long run. Adherence to the non-functional requirement, ensures the long-term success and sustainability of a game, ultimately gaining an advantage over its adversaries.

Performance - The performance of a game determines how well it will run on a particular device. When performance is poor, players may experience slow or turbulant gameplay, which can be frustrating and negatively impact their experience. As a result of poor performance, the game may also crash more frequently and experience other issues, causing the game to become unstable and difficult to play. Alternatively, good game performance ensures that the game runs smoothly and consistently, providing players with an enjoyable and immersive gaming experience. The key to ensuring optimal overall performance in a game is optimizing its code and assets, using a reliable game engine, and testing and benchmarking it on a variety of hardware configurations.

Scalability - Scalability refers to a game's ability to handle increased numbers of players or other demands without significantly affecting its performance. Scalable games can support a large number of players or other demands without becoming sluggish or unresponsive, which can improve the overall enjoyment and satisfaction of the players. In online multiplayer games, where thousands or even millions of players can play simultaneously, scalability is particularly important. Good scalability can ensure that these games remain responsive and enjoyable regardless of the number of players or other demands placed upon the server and infrastructure.

Maintainability - In order for a game to remain maintainable in the long run, it must be capable of being updated, modified, and improved over time. In addition to remaining relevant, enjoyable, and satisfying for players over time, a game that is maintainable is easily and efficiently modified and updated. A game's maintainability is especially relevant for games with a long lifespan or with a large player base, since these games may need to be updated and improved frequently to remain engaging for players. By maintaining a game well, it can be updated and improved without significant interruptions or downtime, and the player can continue to enjoy the game. In the absence of proper maintenance, a game will eventually become outdated or unenjoyable for players over time, which can negatively affect the game's player base and profitability.

Aesthetics - Creating a deeper and more meaningful relationship between a player and a game is possible when a game is able to invoke emotional responses in the player. An engaging and immersive experience can be created for the player by a game that can evoke emotions such as excitement, joy, or fear in them. It contributes to the overall enjoyment and satisfaction of the game by keeping the player motivated and engaged, potentially distinguishing it from other games of the same genre.

User Experience: The overall enjoyment and success of a game are greatly influenced by the user experience. It contributes to the game's overall immersion and enjoyment when players feel engaged, motivated, and satisfied with the game experience. A successful game should be easy to use and learn, enjoyable to play, and satisfying for the player. Designers should focus on usability, feedback, and other elements that contribute to the user experience. Having a poor user experience may make a game difficult to learn and use, unenjoyable, or frustrating for the player. A well-designed user experience, on the other hand, is more likely to engage, satisfy, and engage the player. By focusing on usability, feedback, and other factors that affect the user experience, game designers can create a game that is easy to learn and use, enjoyable to play, and satisfying for the player.

In the context of this research paper's respective implemenation, the player is expected to be faced with the a fun and engaging combat experience to ensure the captivity of attention. A variety of options will be available to the player, such as choosing to fight their enemies or sneak past them, subject to the player's preference. In spite of this, it may not be beneficial for the player to engage in every encounter, as they may quickly become overwhelmed if they do so.

Chapter 4

Implementation Approach

4.1 Architecture

4.1.1 Unity

Unity is a cross-platform game engine developed by Unity Technologies, which is utilized in the creation of 2D and 3D games and interactive experiences for various platforms, including personal computers, mobile devices, gaming consoles, and virtual and augmented reality headsets. An engine is the software that provides the basic architecture, but not the concrete content, of a game[1].

In addition to providing a comprehensive set of tools and features for game development, such as a visual editor and a scripting engine, Unity also supports the development of games for multiple platforms. It boasts a large and active community, with ample resources and support available online.

One of the major advantages of Unity is its flexibility and versatility, which allows for the development of a wide range of games and interactive content, from basic 2D games to complex 3D simulations and virtual reality experiences. Its user-friendly interface and abundance of tutorials and resources make it relatively easy to learn and use.

4.1.1.1 Scenes

In Unity, a scene is a collection of game objects that represents a specific area within a game or application. A scene can be thought of as analogous to a stage in a theater production, where actors, props, and sets are arranged and presented to the audience. It

is where you work with content in Unity[32]. A game or application can contain multiple scenes, and the player can transition between them as the game progresses.

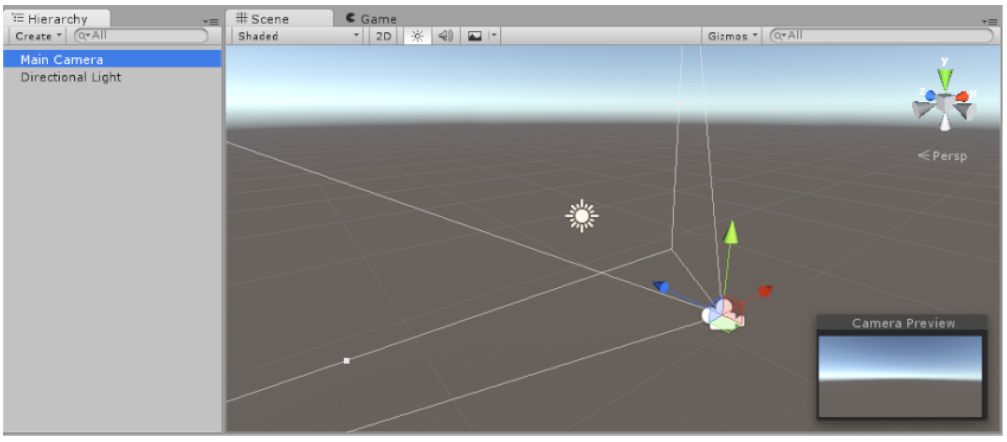

FIGURE 4.1: An empty Unity Scene[32]

To create a new scene in Unity, one can go to "File ¿ New Scene" in the top menu or right-click in the Hierarchy panel and select "Create ¿ Scene". Game objects can be added to a scene by dragging them from the Project panel into the Hierarchy panel, or by using the "Create" menu in the Hierarchy panel. Assets such as 3D models, audio files, and other resources can be imported into the scene by dragging them from the Project panel into the scene view.

To switch between scenes in a game, one can use the "SceneManager" class in C# scripts. Alternatively, UI elements such as buttons can be set up to allow the player to navigate between different scenes.

4.1.1.2 Objects

Objects in Unity are data structures that represent characters, props, or scenery in a game[33]. Prefabs define the properties and behavior of objects and allow them to be created easily. By creating and configuring objects and their components, you can create complex, interactive experiences for players.

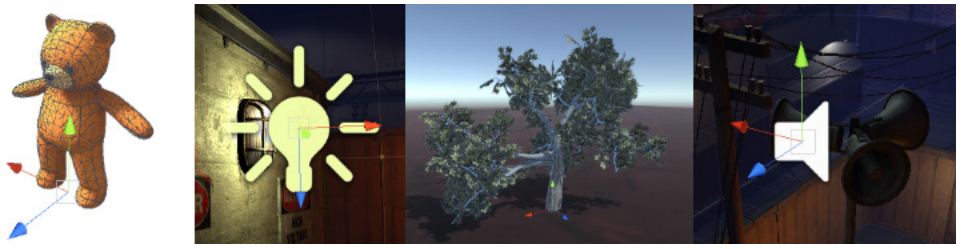

FIGURE 4.2: Collection of Unity Objects[33]

Each Unity object consists of several components, which can be assembled using building blocks. This can include components such as a transform, which determines the position, rotation, and scale of an object in the scene; a mesh filter, which stores a reference to a 3D mesh representing the object's shape; and a mesh renderer, which controls how the object is rendered in the scene, as well as its material, texture, and other visuals.

Primitive objects refer to 3D models representing fundamental shapes, such as spheres, cubes, cylinders, and cones. These objects are labeled as "primitive" due to their built-in status within the engine, which facilitates their creation and manipulation[34].

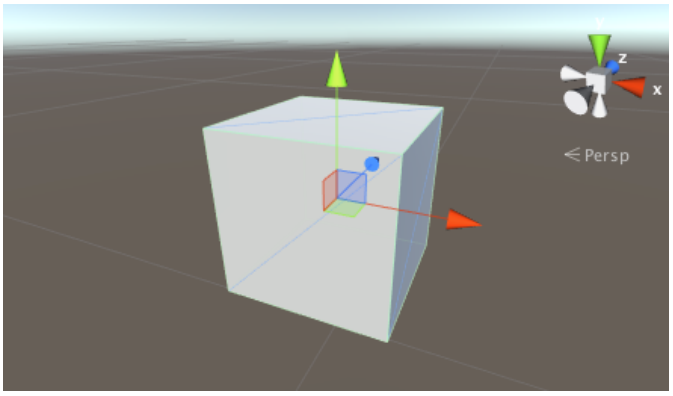

FIGURE 4.3: A primitive object in Unity[34]

To create a primitive object in Unity, one can go to "GameObject ¿ 3D Object" in the top menu, and then select the desired shape from the submenu. Alternatively, a primitive object can be created at runtime through code by calling the "GameObject.CreatePrimitive" method and specifying the desired shape as a parameter.

Once a primitive object has been created, its properties can be modified using the Inspector panel. For instance, the object's position, rotation, and scale can be changed, and materials can be added to alter its appearance. Additionally, scripts can be attached to the object to define custom behavior.

Primitive objects serve as a useful foundation for the creation of more complex models in Unity, and they can be utilized for various purposes, such as constructing level geometry, creating props and characters, and implementing simple physics simulations.

Static game objects are GameObjects that are not intended to move or change at runtime[35]. GameObjects of this type are usually used for objects that do not need to move or change, such as walls, floors, and buildings within an environment. Checking the "Static" checkbox in the Inspector window will mark a GameObject

as static in the Unity Editor. However, Unity optimizes the way objects are rendered and processed in order to improve performance. If a GameObject is marked as static, it is not prohibited from being modified or altered in other ways, such as by modifying its materials or textures. Nevertheless, it indicates to Unity that the object is not expected to move or change position, thereby enhancing performance.

Dynamic game objects are GameObject that are intended to move or change at run time[35]. Characters, vehicles, and projectiles are examples of these types of GameObjects, which are often controlled by the player or by artificial intelligence. In the Inspector window, dynamic GameObjects are not marked as static, but can be created and modified similarly to static GameObjects. In a game or application, this gives them the ability to move around and change as needed. It is possible to move and change dynamic GameObjects by using scripts, by modifying their transform properties, or by utilizing physics components to apply forces or impulses.

4.1.1.3 Scripts

Scripting is a fundamental aspect of Unity development, as it enables the creation of custom functionality in games and applications using code. Unity supports scripting in C#, UnityScript, and Boo. In Unity, scripts can be attached to objects to control their behavior. For instance, a script might be used to allow a character to move in response to player input, or to enable an enemy to fire projectiles when it detects the player. In this research paper, the game will be implemented using C# scripts. As well as being used for the creation of graphical effects, scripts may also be used to control the physical behavior of objects or to implement a custom AI system for the avatars within the game using FSMs. [36]

As seen in Figure 4.4. It is the MonoBehaviour class that functions as the base class for scripts that are attached to GameObjects. This framework defines the behavior of GameObjects in a scene and provides a number of overridable methods for implementing custom functionality. Derived from the Behaviour class, which is itself derived from the Component class, the MonoBehaviour class allows developers to create custom scripts that can be attached to GameObjects and that have access to useful methods and properties, such as Update, Start, and Awake. For GameObjects attached to the script, these methods can be overridden to specify custom behavior.

As soon as a script is enabled, the start() function is executed, while the update() function is called one time per frame. During the script's aforementioned gameloop (see Chapter 2.3.1), these functions allow the user to specify code that should be executed at

regular intervals (ticks). When initializing variables or performing setup tasks that only need to be performed once, the start() function is typically used, whereas the update() function is typically used for implementing continuous or recurring logic. By specifying which code should run once and which should run repeatedly, the start() and update() functions can help developers organize and structure their code, as well as optimize performance.

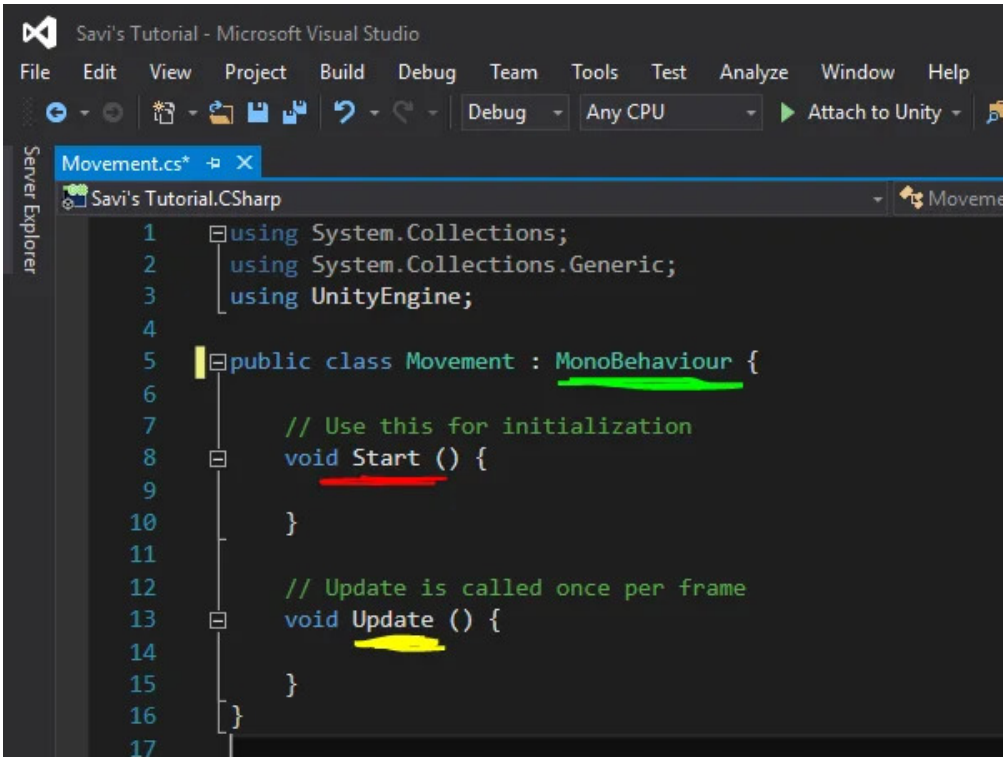

FIGURE 4.4: A newly created Unity script[37]

To create a script in Unity, one can go to "Assets ¿ Create ¿ C# Script" in the top menu or right-click in the Project panel and select "Create ¿ C# Script"[37]. This will create a new script asset in the project, which can be opened in a code editor to write the desired code. Once the script has been written, it can be attached to a game object by dragging the script asset from the Project panel onto the object in the Hierarchy panel. The Inspector panel can then be used to configure and set the script's properties.

4.1.1.4 Materials

Materials are assets that dictate the visual appearance of a 3D object[38]. They are utilized to apply textures and colors to an object, as well as to specify other visual characteristics, such as shininess, transparency, and reflectivity.

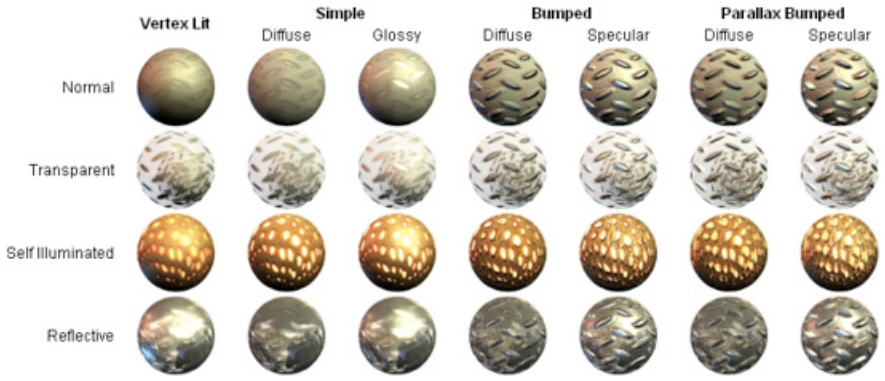

FIGURE 4.5: A list of materials in Unity[39]

To create a material in Unity, one can go to "Assets ¿ Create ¿ Material" in the top menu or right-click in the Project panel and select "Create ¿ Material". This action will create a new material asset in the project, which can be opened in the Inspector panel by double-clicking it. In the Inspector panel, the properties of the material can be set, including its color, texture, and other visual properties. A shader can also be assigned to the material, which is a program determining how the material is rendered in the scene.

To apply a material to an object in Unity, the material asset can be dragged from the Project panel onto the object in the Hierarchy panel. Alternatively, the object can be selected and the material can be assigned to its "Material" property in the Inspector panel.

4.1.1.5 Tutorials

Unity offers a vast array of lab-like tutorials and courses for beginners to understand the inner workings of the Unity Engine. It does so through many interactive and simple-to-understand tutorials where the student is expected to code alongside the tutorial to achieve a certain output, where independent tinkering is encouraged.

Roll-a-Ball[40] is an official Unity tutorial that introduced me to the basics of the Unity interface and engine. A simple game where the user is expected to 'roll-a-ball' around a plain to collect coins. It proved to lay the foundations to my initial understanding of Unity's interface navigation. The learning outcomes of this tutorial include how to set up a game environment, how to create objects and materials, how to write custom scripts to create game functionality and how to build the game so other people can play it.

Create with Code[41] official Unity course offers a more in-depth experience and teaches how how to design and animate game objects, create and trigger events, and write code to control the behavior of your game. The tutorial is designed for beginners, so no prior programming experience is required. However, it is helpful to have some familiarity with basic programming concepts. The course consists of many units, each with their own sub-tutorials and learning outcomes. Such units include 'Player Control', 'Basic Gameplay', 'Sound and Effect's, 'Gameplay' Mechanics and 'User Interfaces'.

4.1.2 Blender

A powerful and versatile 3D graphics software toolset, Blender is used to create a wide variety of visual content, including animated films, visual effects, 3D models, and interactive applications. As an open-source project, the source code is available to anyone for use, modification, and distribution. The versatility and quality of Blender make it a popular choice among artists and developers seeking a free, high-quality 3D graphics program. In addition to its high-quality rendering capabilities, it has the ability to produce high-quality visuals on a professional level. As well as generating 3D printed models and motion graphics, it is also widely used in the production of 3D printed models, an aspect which will prove beneficial for this implementation. It is a comprehensive toolkit that includes a wide range of features for 3D modeling, texturing, rendering, animation, and compositing. It features a user-friendly interface that is suitable for both beginners and experienced users, and a strong community of users and developers contribute to its development and provide support and resources to users.

The goal of this project is to create as much original art as possible, using my previous experience in trying Blender as seen in Figure 4.6 , while keeping within the constraints of the planned schedule. Original art refers to artwork that is created specifically for this project, rather than being sourced from external sources. This could include character designs, environments, and other assets that are unique to the project. The use of original art allows for greater control over the visual style and aesthetic of the project, and it ensures that the project has a unique and distinct look and feel. It can also be more cost-effective in the long run, as it avoids the need to license or purchase art from external sources.

However, it is also important to consider the time constraints of the project. If the schedule is tight and there is not enough time to create all of the necessary art assets from scratch, it may be necessary to use referenced art in order to meet the deadlines. Referenced art refers to artwork that is sourced from external sources, such as stock

FIGURE 4.6: A previous attempt to create an enemy character using Blender in the past.

images or pre-existing assets. While the use of referenced art may not be ideal, it can help to ensure that the project is completed on time and within budget.

4.2 Risk Assessment

An individual's ability to anticipate and respond to potential challenges during the course of a project is based on their ability to assess and manage risk. Inadequate risk identification and management can have serious consequences, such as project delays, budget overruns, and even project failure.

Identifying and addressing risks effectively requires a thorough risk assessment, in which all potential risks are identified and classified according to their likelihood and importance. The categorization of risks based on their impact and probability can be assisted by tools such as Table 4.1. Mitigation strategies can then be developed based on the identified and classified risks. A contingency plan may be developed to address high-impact, high-probability risks, as well as seeking additional resources or support for risks that have a lower likelihood of occurrence but still have significant impacts.

Throughout the course of a project, it is also crucial to continually monitor risks and adjust mitigation strategies as necessary. Keeping the risk assessment current can ensure

that developers are address any potential challenges as early as possible and remain on track to complete the project successfully.

TABLE 4.1: Initial risk matrix

Frequency/ Consequence	**1-Rare**	**2-Remote**	**3-Occasional**	**4-Probable**	**5-Frequent**
4-Fatal					
3-Critical					
2-Major					
1-Minor					

Several potential risks could prevent this implementation from being successful. These potential risks are listed below, along with their respective frequency and consequences, according to Fig 4.1:

1. **Time constraints:** In order to ensure a successful development of a game, careful planning and management are required to ensure its success. It is difficult, given time constraints, to allocate sufficient time to each task and to remain motivated and focused throughout this process.

   Frequency == Occasional

   Consequence == Critical

2. **Lack of resources:** In order to develop a game, various resources are required, including programming skills, art assets, and test resources, among others. Without these resources, it can be challenging to complete the game successfully, since they are essential for integrating mechanics and features, creating a visually appealing and immersive experience, and identifying and fixing problems before release.

   Frequency == Rare

   Consequence == Critical

3. **Technical issues:** It is common for game developers to encounter technical difficulties during the development process, such as bugs, compatibility problems, and performance problems. To ensure that the game performs optimally and provides a positive experience for players, it is imperative that developers identify and troubleshoot these issues as they appear. A debugging tool may be used, the game may be tested on different platforms or devices, and code changes may need to be made.

   Frequency == Major

   Consequence == Frequent

4. **Scope creep:** During the course of a project, the scope may change or expand, resulting in delays or strains on available resources. Scope creep can be prevented by setting clear goals and objectives, establishing a project plan, and tracking progress consistently. In this manner, a project can be kept on track and within its defined scope, while still allowing flexibility and adaptability if necessary.

   Frequency == Occasional

   Consequence == Critical

5. **Quality issues:** In order for a game to be successful, its quality is crucial. Negative reviews and player frustration can result from quality issues, such as bugs or performance issues. In order to increase the chances of a successful game, developers must prioritize quality during the development process and address any issues that may arise. Tests may be conducted extensively, debugging tools may be used, and player feedback may be incorporated.

   Frequency == Major

   Consequence == Frequent

6. **Intellectual property issues:** There is a risk of legal challenges or disagreements when using third-party assets or intellectual property in game development. To minimize this risk, developers need to carefully research and evaluate the legal implications of using these assets, as well as obtain the appropriate licenses or permissions when necessary. It is important for developers to take these precautions in order to reduce the risk of legal challenges and to ensure that their games are protected from any potential legal repercussions.

   Frequency == Rare

   Consequence == Minor

7. **Inadequate testing:**. It is essential to conduct thorough testing during the game development process in order to identify and address any issues that may arise before the game is released. Negative reviews, player frustration, and in some cases, even possible legal repercussions may result from failing to thoroughly test a game. It is therefore important for developers to thoroughly test their game during development, using testing tools and techniques, incorporating feedback from playtesters and players, and collaborating with a quality assurance team to ensure that the game performs well and is of high quality.

   Frequency == Remote

   Consequence == Major

4.3 Implementation Plan Schedule

The concept of software development methodology refers to a framework from which serves as the structure, planning, and control the development process. Agile, waterfall, and iterative methodologies are some of the most common software development methodologies. In addition to planning and managing their work, these methodologies enable teams to deliver software on time and within budget, and ensure that the end product meets their customers' requirements. Choosing the right methodology for your specific situation is essential, as it sets the pace and determines the longevity of the entire development process. Depending on the type of project and the type of team, different methodologies may be more effective.

> "Give me six hours to chop down a tree and I will spend the first four sharpening the axe." - Abraham Lincoln

The purpose of the methodologies are to ensure that the product or system being developed meets the requirements of the intended audience, progressively throughout all phases of the development process. The development process can be streamlined with the intention of preventing costly mistakes and rework, and ensuring end users are satisfied with the finished product. Developing a clear plan for how to proceed can be achieved by gathering and documenting requirements early in the development process. This will allow teams to better understand the goals and constraints of the project. By doing so, communication and collaboration between team members can be improved, and everyone can work towards the same goals. The use of effective requirements engineering can ultimately result in a reduction of both time and resource usage as well as an increase in system or product quality.

4.3.1 Semantic Versioning 2.0.0

In accordance with the agile methodology, this research paper's associated game will make extensive use of Semantic versioning 2.0.0 to track and document progressive development. Also known as SemVer, it is a versioning system for software projects. The policy is commonly accepted by open source package management systems to inform whether new releases of software packages introduce possibly backward incompatible changes[42]. Users can understand the impact of these changes on compatibility and stability by utilizing a clear and standardized means of communicating changes to a software project. A project's compatibility with the version of the project it is being used is particularly useful when it is utilized or reviewed by other developers.

The semantic versioning format consists of three digits separated by dots. Major version numbers are indicated by the first digit, minor version numbers by the second digit, and patch versions by the third digit. For instance, the version "2.0.7" comprises a major version number of "2", a minor version number of "0", and a patch version number of "0". Software projects are released with major and minor version numbers of 0 in accordance with the semantic versioning specification. The version numbers are incremented in accordance with the progress of the project[42]. When a new feature is added, the minor version number is incremented, and when a bug is fixed, the patch version number is incremented. Software developers commonly use semantic versioning to communicate changes to a project and provide users with information concerning the impact of these changes on compatibility and stability.

4.3.2 Agile Methodology

The Agile methodology is a project management approach that emphasizes flexibility and adaptability in the development of high-quality products. Iterative and incremental software development is the foundation of this methodology[43]. Its principles and values are derived from the Agile Manifesto, which outlines a set of principles and values for software development that prioritize customer satisfaction, the delivery of working software on a regular basis, and the ability to respond to changes. In Agile software development, requirements and solutions are evolved through a series of short development cycles, called sprints, during which the software is demonstrated to the customer and feedback is collected to guide future development. As part of the Agile methodology, various tools and techniques are encouraged to support effective collaboration and communication, including agile project management software, daily stand-up meetings, and retrospectives.

Iterative sprints of approximately 3 weeks will be undertaken in order to complete the associated implementation of this research paper, as seen in Figure 4.7. In accordance with the aforementioned semantic versioning, each sprint will be considered a "Major" release, relative to the scope of this project. Each sprint includes a collection of new features, each of which to be considered a "Minor Release". Any additional code pushes between these minor realeases will be considered "Patches".

4.3.3 Kanban Board

Kanban boards are used to organize and track work by teams or individuals as part of a project management process. A lean manufacturing process used to optimize work flow, it is based on the principles of the Kanban method[44]. First developed by Toyata

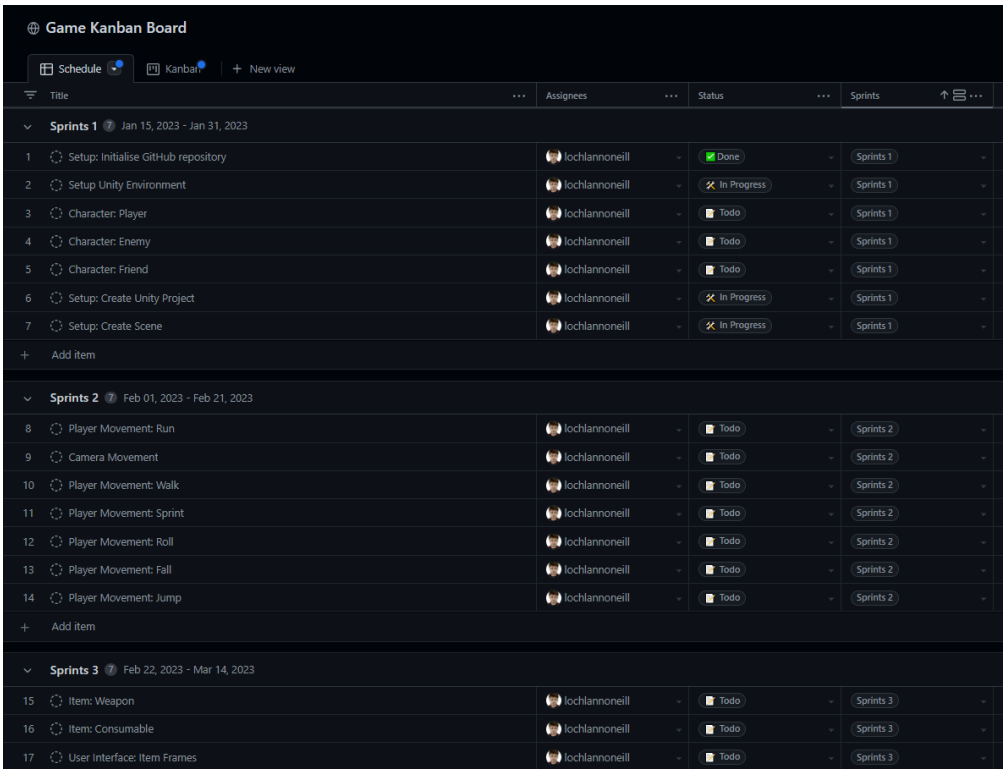

FIGURE 4.7: Implementation Plan Schedule adhering to the Agile Methodology

to minimize waste and improve efficiency, it is a flow control mechanism for pull-driven (actual demand) Just In-Time (JIT) production, in which the upstream processing activities (e.g raw material procurement, component production) are triggered by the downstream process demand signals (e.g customer order, sales forecasts).[45]. Many industries now utilize Kanban boards to organize their activities, track their progress, and identify bottlenecks and inefficiencies.

As seen in Figure 4.8, Kanban boards consist of a series of columns, each representing a different stage in the work process. In accordance with the stage of the work, cards are created to represent individual tasks or pieces of work. Cards are moved from column to column as work progresses until they reach the ”Done” column.

4.4 Evaluation

4.4.1 Playtesting

Testing plays a vital role in the development of a video game, since it provides developers with the opportunity to identify and fix problems, balance gameplay elements, and fine-tune the overall user experience. The process of testing ensures that the final product meets the expectations of the target audience while being enjoyable to use. The purpose

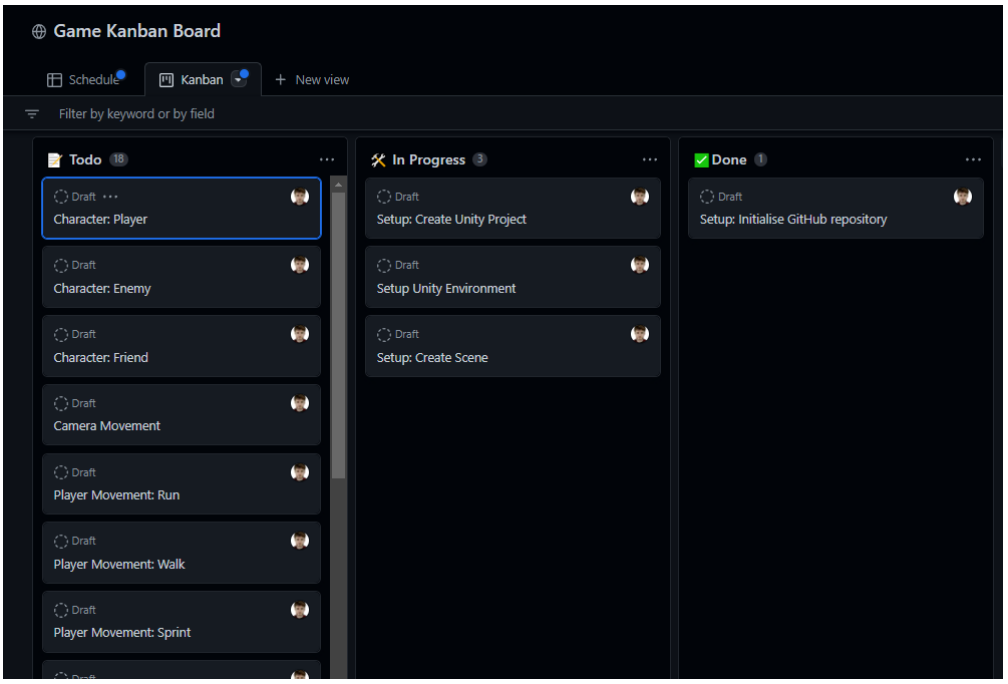

FIGURE 4.8: Implementation Kanban Board

of playtesting is to collect valuable feedback from a diverse group of players in order to identify issues which may not have been apparent immediately. Developers can also use playtesting to determine what aspects of the game are most enjoyable and engaging, and work on improving them. The test will also allow developers to identify any problems or bugs in the game, which can then be fixed before the game is publicly released.

Different approaches can be used to playtest a game, and the specific approach chosen by the developer will depend on its goals and needs. Alpha and beta testing, usability testing, and focus groups are some of the most common methods used. Playtesters should be provided with clear guidelines and instructions, and their feedback should be carefully analyzed and used to improve the game, regardless of the approach employed.

Throughout the implementation phase, regular playtesting will take place. Friends and family will hopefully participate and provide their subjective opinions on specific features, as they would they may be more forgiving of any rough edges and be more willing to provide constructive feedback. The data collected from this playtesting will be used to make informed decisions about the direction of the game. For example, if a feature is not well received by the chosen playtesters, I may choose to revise it or remove it entirely. Similarly, if a feature is particularly popular, the I may choose to expand on it or highlight it more prominently in the final version of the game.

4.4.2 Feedback

The feedback loop is the process of gathering and using feedback to improve a product or process in a consistent and reliable way, especially through regular playtesting. Maintaining the integrity of the feedback loop involves having a clear process for gathering and using feedback, and ensuring that this process is followed consistently in order to deliver a high-quality product that meets the needs and expectations of the intended audience. The both quantitative and qualitative data will be gathered by means of surveys, interviews, and observation.

Quantitative testing is a method of collecting and analyzing numerical data in order to understand and make decisions about a phenomenon, such as a game. Various methods can be used to gather this data, including surveys, analytics, and playtesting, so that you can measure a player's behavior, performance, engagement, and enjoyment. The objective of quantitative testing in the development of video games is to identify trends and patterns, test hypotheses, and evaluate the effectiveness of specific game design elements. Additionally, it can be used for optimizing the game and making informed decisions regarding how to improve it, especially when working with a large team.

Qualitative testing involves collecting and analyzing non-numerical data, such as through observation, interviews, or focus groups. Playtesting and player interviews are common methods for gathering subjective information. Players' experiences, thoughts, and feelings can be analyzed using this data, as well as their subjective perceptions of the game, their engagement with different components, and their preferences. Developers can gather and analyze data about players' motivations, preferences, and behaviors through qualitative testing in order to better understand how players experience their games and improve them accordingly.

The combination of qualitative and quantitative testing can provide complementary insights and provide a more coherent general understanding of a game when both methods are used in conjunction with one another. In this way, quantitative data can identify trends and patterns in player behavior, and then use qualitative data to understand why these trends are occurring and how subjective factors such as player preferences or motivations might influence them.

4.5 Prototype

4.5.1 Paper-Prototype

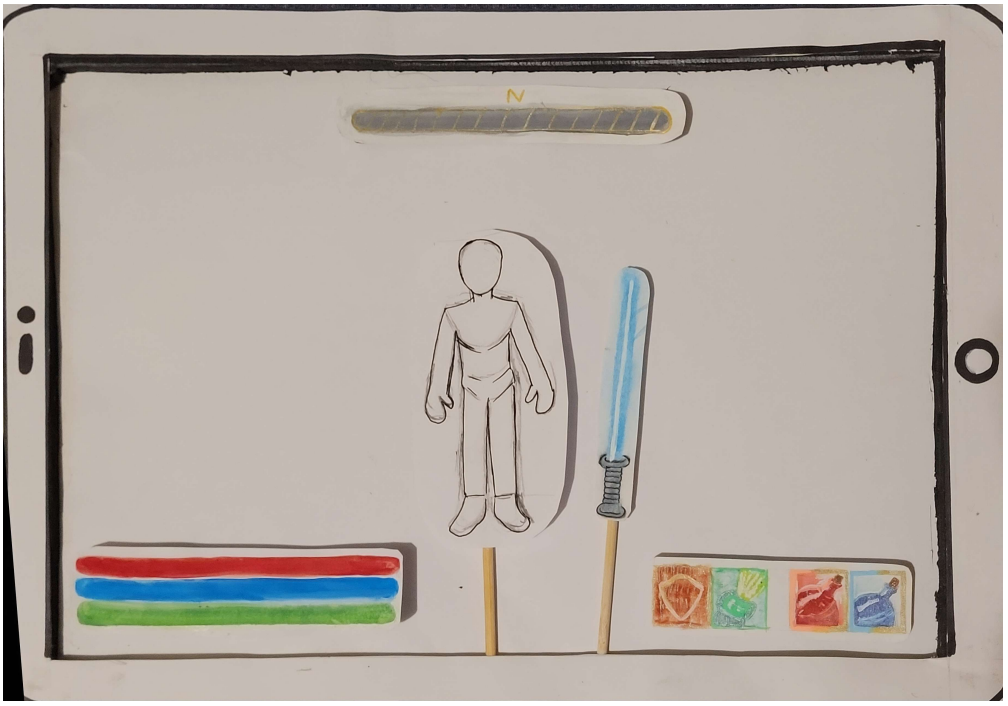

FIGURE 4.9: Paper-prototype showing default User Interface (UI)

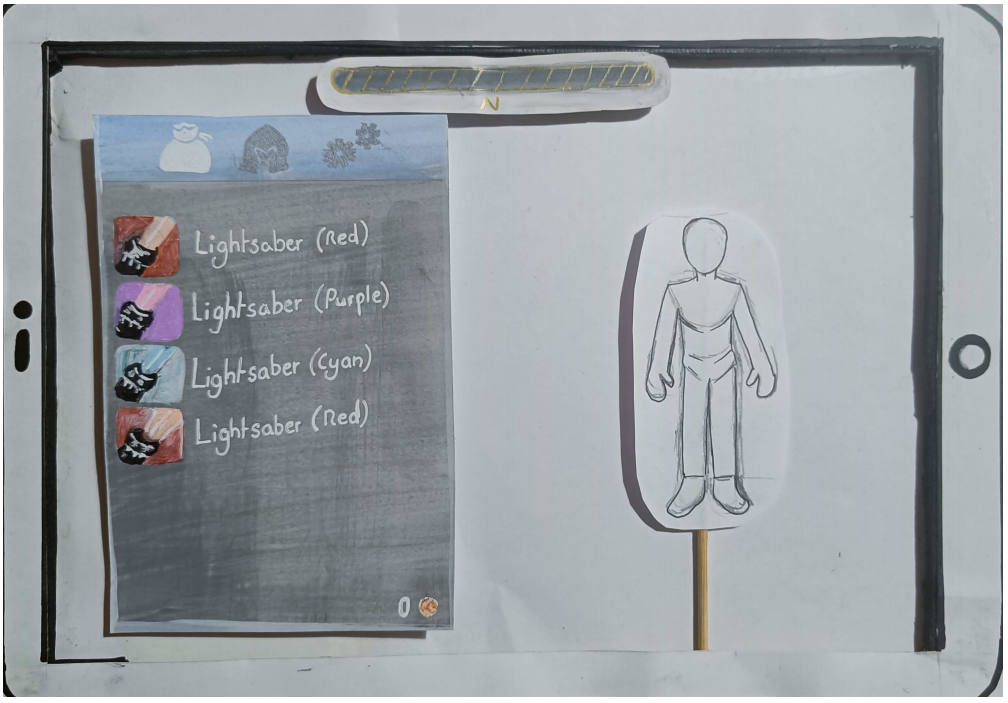

FIGURE 4.10: Paper-prototype showcasing open inventory UI

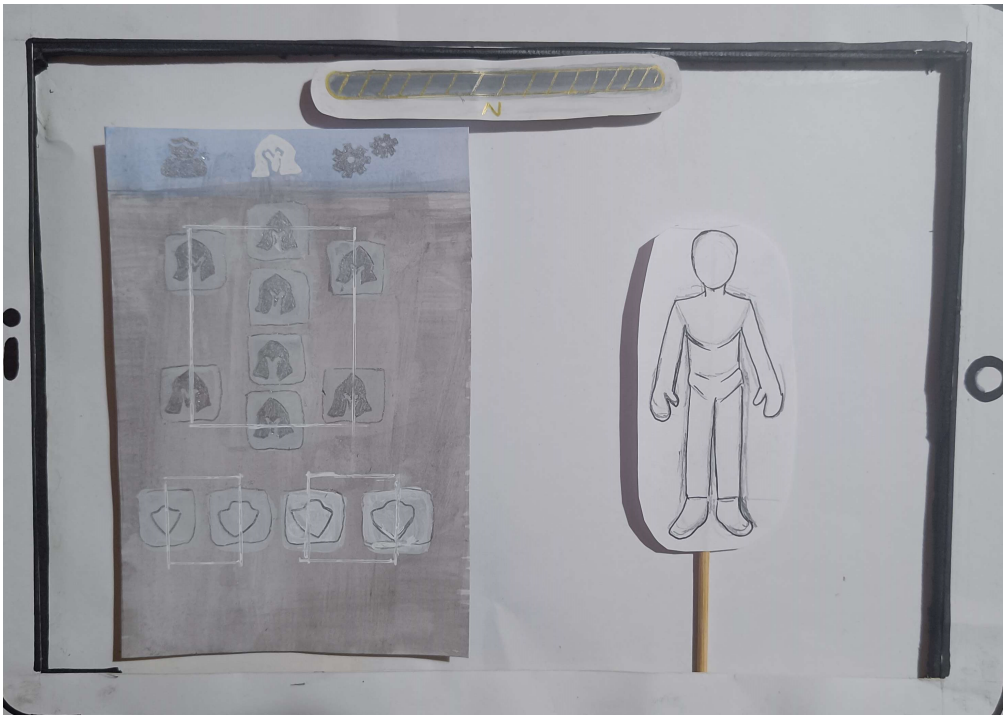

FIGURE 4.11: Paper-prototype showcasing open equipment UI

Chapter 5

Implementation

Developing a game in Unity proved to be a challenging yet exhilarating experience. In the following sections, I will delve into the architectural design of the solution, the difficulties encountered during implementation, and the approach used to overcome them. Doing so should provide the reader with a comprehencive guideline for developing a game with similar functionality.

In order to provide players with a fully engaging experience, I incorporated instinctive navigation and dynamic camera movement. My aim was to create an environment that made players feel like they were exploring a genuine world, which required me to be mindful of the character's reaction to user input, as well as the way the camera moved. After testing various camera angles and movements, I identified those that offered the most control and an authentic perspective.

I also recognized the importance of creating movement and camera controls that would be familiar to players of similar games. This allowed players to easily grasp the game mechanics and enjoy the gameplay without becoming frustrated. By examining how other popular games employed these controls, I was able to integrate comparable elements while adding my own innovative flair to create a unique and distinct gaming experience.

5.1 Solution Approach

To establish the game's foundation, I began by creating a new project in Unity and importing necessary assets, such as models, skybox art, graphical animations (of which still require actual functionality to operate), etc.

As evident from the generated project hierarchy, the scene initially consisted of only the static main camera and a directional light source. However, to begin with the game development, the immediate requirement was to create a player object that would hold all the player-related functionality and a plane which will act as the floor.

5.1.1 Project Architecture

The solution architecture consists of three primary components: the player, the camera, and the in-game environment. The player component includes various sub-components, such as the player object itself, input controls, movement, and falling behavior. Similarly, the camera component comprises camera handling, rotation, pivot, and collision detection. Lastly, the environment encompasses lighting and interactable/non-interactable items that can be found throughout the game scene via prefabs.

5.1.2 Player

5.1.2.1 Object

To incorporate an object and model into the Unity scene, first, launch the Unity software and either start a new project or access an existing one. Once you're inside the project, import your 3D model by right-clicking in the Project window and choosing "Import New Asset". Remember to ensure that the model is in a format that Unity can recognize, such as .fbx or .obj. Once you've imported the model, add it to your scene by selecting "3D Object" after right-clicking in the Hierarchy window. With the new object now in place, you can proceed to adjust its properties and behavior using the Inspector window.

To enhance the capabilities of an object, it can be equipped with a Rigidbody and Collision Collider. First, navigate to the Hierarchy window and select the object's, Inspector window and locate the "Add Component" button. Once clicked, add a Rigidbody component to allow the object to respond to the laws of physics in the scene. Next, add a Collision Collider component to establish a collision boundary around the object, enabling it to interact with other objects in the game environment. By tweaking the settings of these components, such as the mass and drag of the Rigidbody or the size and shape of the Collision Collider, you can fine-tune the object's behavior and optimize it for your specific game needs.

To further extend the custom-capabilities of objects to allow for inticate behaviour and interaction between objects, writing supplementary scripts becomes a necessity. To add a script to an object, go to the Inspector window, click the "Add Component" button,

and select the desired script. By doing so, the object becomes capable of executing new tasks and exhibiting new behaviors in your game.

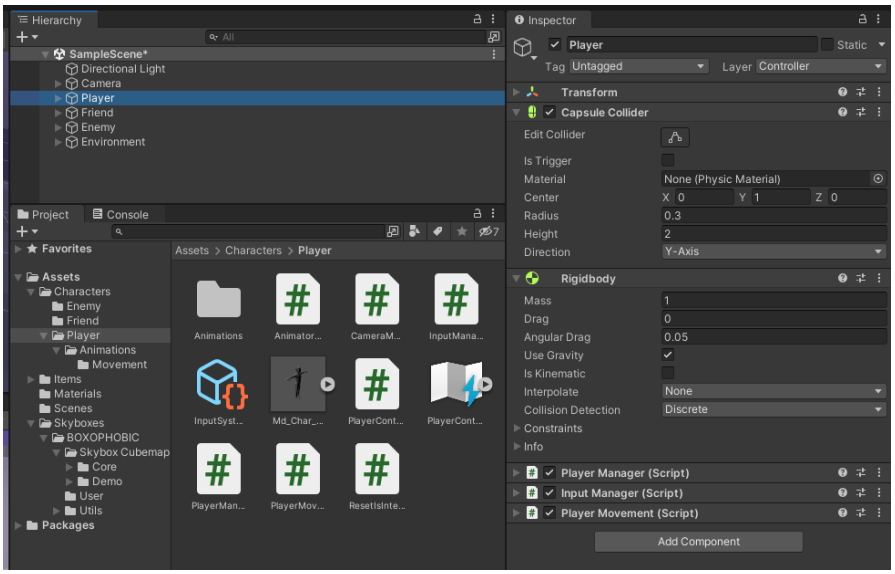

FIGURE 5.1: Adding functionality to objects in it's respective Inspector window

5.1.3 PlayerManager

As seen in Figure 5.2, the PlayerManager script serves as the central hub for managing the player object in the game, encompassing its movement, input, and camera handling. Acting as an entry point, the script references other crucial player scripts, namely "PlayerMovement", "InputManager", and "CameraManager". Boolean flags are utilized to keep track of the player's state, such as whether or not they are in the air.

```
namespace LON {
    4 references
    public class PlayerManager : MonoBehaviour
    {
        6 references
        PlayerMovement playerMovement;
        4 references
        InputManager inputManager;
        4 references
        CameraManager cameraManager;
        2 references
        Animator anim;
```

FIGURE 5.2: The player manager serves as the entry-point to the Player object's functionality

To optimize the performance, this is the only player-related script where the "Update" method is called, handling player movement and animation. Update() is called once

per frame and is used for most gameplay-related updates such as user input, animation, and physics updates. The time between each call to Update() varies depending on the hardware and can cause inconsistent behavior when using Time.deltaTime for physics calculations.

The "Update" method, as seen in Figure 5.3, is responsible for handling gameplay-related updates, such as user input, animation, and physics, and is called once per frame. However, the time between each call to Update() can vary depending on the hardware, which may lead to inconsistent behavior when using Time.deltaTime for physics calculations. To ensure optimal performance, the delta time is calculated within the method, and it's then used to call several methods related to the player's movement and falling, including inputManager.Input(), playerMovement.HandleMovement(), and playerMovement.HandleFalling().

```
0 references
void Update()
{
    float deltaTime = Time.deltaTime;

    isBusy = anim.GetBool("isBusy"); // halts input while falling

    inputManager.Input(deltaTime);
    playerMovement.HandleMovement(deltaTime);
    playerMovement.HandleFalling(deltaTime, playerMovement.moveDirection);
}
```

FIGURE 5.3: Implementation of Update logic

The "FixedUpdate" method, as seen in Figure 5.3, in the PlayerManager script is responsible for camera rotation and follow. This method is called at fixed intervals, which are determined by the Fixed Timestep setting in the Project Settings. The FixedUpdate method is specifically used for physics updates, including rigidbody movement and collisions. This approach ensures that physics calculations are consistent across different hardware and avoids any issues caused by the varying time between each Update call, as experienced with the use of Time.deltaTime.

The "LateUpdate" method is called after all other update functions have been processed. In this project, it is used to reset flags and update the player's state if they are in the air. This approach is suitable since LateUpdate ensures that all necessary changes have been made to the player's position and rotation before camera movement and other components are updated.

```
0 references
private void FixedUpdate() {
    float deltaTime = Time.fixedDeltaTime;

    if (cameraManager != null) {
        // Update the camera's position to follow the target
        cameraManager.HandleFollow(deltaTime);

        // Handle the rotation of the camera based on user input
        float mouseX = inputManager.mouseX;
        float mouseY = inputManager.mouseY;
        cameraManager.HandleRotation(deltaTime, mouseX, mouseY);
    }
}
```

FIGURE 5.4: Implementation of FixedUpdate logic

5.1.3.1 Controls (Input Actions)

Input actions in Unity provide the ability to create custom actions that can be triggered by a variety of input sources, including keyboard, mouse, or gamepad. By defining these actions, one can easily map them to specific keys or buttons and utilize their values within the game code. To set up input actions, the creation of an input action asset is required, which contains all the necessary actions and mappings. However, it is first required to import the official Unity 'Input System' package within the package manager.

As seen in Figure 5.5, input actions allow for both player movement and camera control. This can be achieved by mapping the movement to the industry-standard WASD format, while utilizing analog input from the mouse (delta) for camera control.

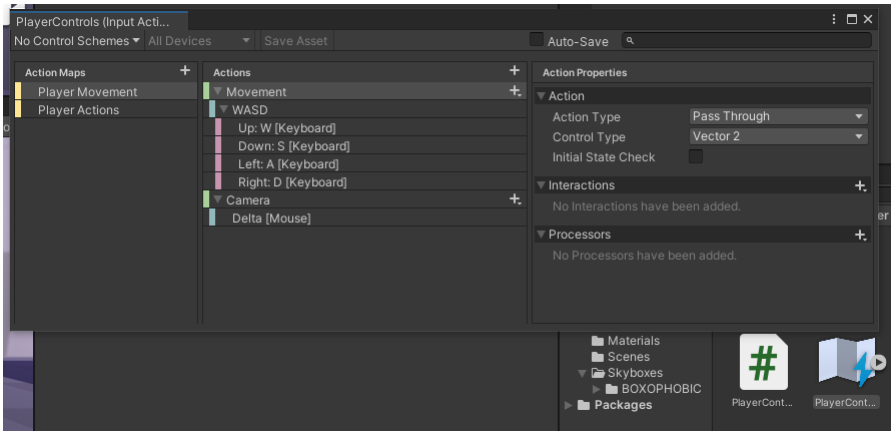

FIGURE 5.5: Input actions for player movement and camera control

It is then necessary to instantiate PlayerControls for player movement and camera control, as seen in 5.6 This allows for input values to be read and used within the game

logic. By utilizing the OnEnable method, this setup is performed automatically when the script is enabled, ensuring that the input actions are ready for use in the game.

```
0 references
public void OnEnable()
{
    if (inputActions == null)
    {
        inputActions = new PlayerControls();
        inputActions.PlayerMovement.Movement.performed += inputActions => movementInput = inputActions.ReadValue<Vector2>();
        inputActions.PlayerMovement.Camera.performed += i => cameraInput = i.ReadValue<Vector2>();
    }

inputActions.Enable();
}
```

FIGURE 5.6: Instantiating PlayerControls within the game logic

Next, I can read input values and translates them into movement and rotation values used in the game code, as seen in Figure 5.7. By updating these variables every frame, the player's movement and camera rotation can be updated smoothly and responsively based on user input.

```
1 reference
private void MoveInput(float delta)
{
    horizontal = movementInput.x;
    vertical = movementInput.y;
    moveAmount = Mathf.Clamp01(Mathf.Abs(horizontal) + Mathf.Abs(vertical));
    mouseX = cameraInput.x;
    mouseY = cameraInput.y;
}
```

FIGURE 5.7: Creating input variables from the input actions in PlayerControls

5.1.3.2 Movement

The "HandleMovement" function calculates the movement direction of the player based on input, calculated the velocitye, updates movement animator values, and calls "HandleRotation" if the player can rotate. The "HandleRotation" function calculates the target direction of the player based on input and sets the rotation of the player to the target rotation.

To visually supplement this functionality, a blend tree must be created that operates on imported animations (from Mixamo), as seen in Figure 5.9. The intensity of the animation is determined by the velocity value calculated from the movement functions mentioned earlier. As seen in Figure **??**, this velocity value is then translated into a value an acceptable value for the blend tree animation thresholds. Each movement animation is distinguished by a specific velocity threshold. This technique portrays a sense of weight and momentum in the character's movements,

```
public void HandleMovement(float delta) {
    // Guard Clause : skip function if player is interacting
    if (playerManager.isInteracting)
        return;

    moveDirection = cameraObject.forward * inputManager.vertical + cameraObject.right * inputManager.horizontal;
    moveDirection.y = 0f;
    moveDirection.Normalize();

    float speed = movementSpeed;
    Vector3 projectedVelocity = Vector3.ProjectOnPlane(moveDirection * speed, normalVector);
    rigidbody.velocity = projectedVelocity;

    animatorManager.UpdateAnimatorValues(inputManager.moveAmount, 0);

    if (animatorManager.canRotate)
    {
        HandleRotation(delta);
    }
}
```

FIGURE 5.8: Implementation of HandleMovement functionality

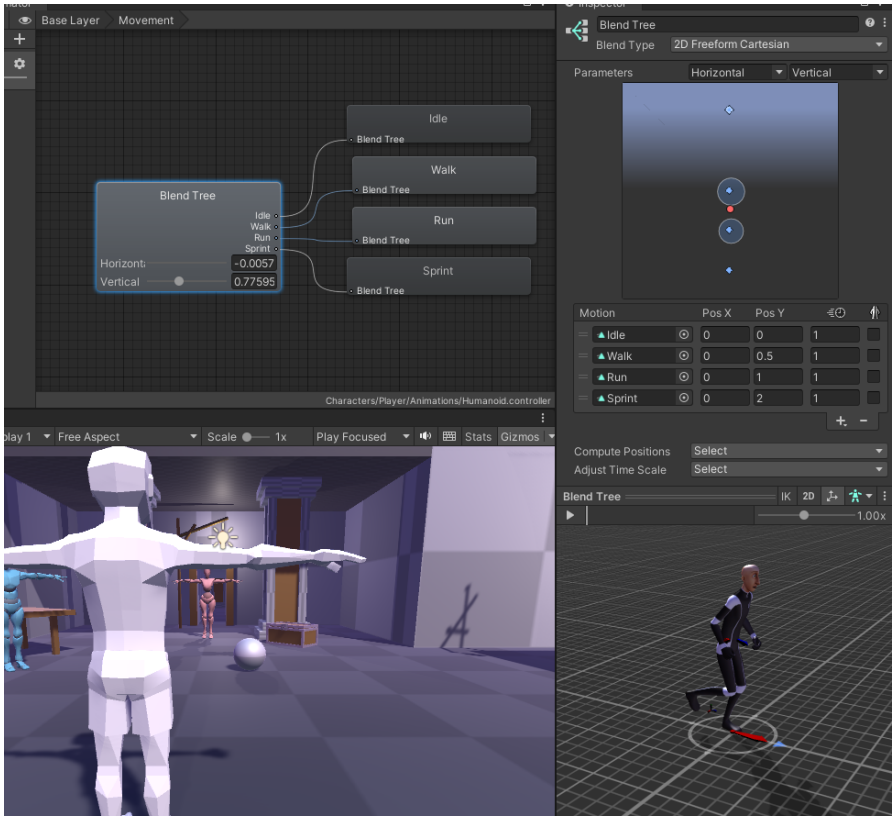

FIGURE 5.9: Blend tree animation thresholds

5.1.3.3 Falling

The "HandleFalling" function is responsible for handling the player's movement and interaction with the ground. In the beginning, the function sets the "isOnGround" boolean variable of the "playerManager" object to false, indicating that the player is not on the ground. With utilization of the raycast physics function, it then casts a ray (red debug line) from a point above the player's position, as seen in Figure 5.12, to detect if there is any obstacle in front of the player that would prevent them from moving forward.

```
1 reference
public void UpdateAnimatorValues(float verticalMovement, float horizontalMovement)
{
    #region Vertical
    float v = 0;
    if (Mathf.Abs(verticalMovement) > 0.55f)
    {
        v = Mathf.Sign(verticalMovement);
    }
    else if (Mathf.Abs(verticalMovement) > 0)
    {
        v = Mathf.Sign(verticalMovement) * 0.5f;
    }
    #endregion

    #region Horizontal
    float h = 0;
    if (Mathf.Abs(horizontalMovement) > 0.55f)
    {
        h = Mathf.Sign(horizontalMovement);
    }
    else if (Mathf.Abs(horizontalMovement) > 0)
    {
        h = Mathf.Sign(horizontalMovement) * 0.5f;
    }
    #endregion

    anim.SetFloat(vertical, v, 0.1f, Time.deltaTime);
    anim.SetFloat(horizontal, h, 0.1f, Time.deltaTime);
}
```

FIGURE 5.10: Implementation of blend tree animation functionality

If the player is in the air, the function applies a downward force and a forward force to the player's rigidbody, simulating the effect of gravity and air resistance. It then checks if the player has landed on the ground by casting a ray downwards from a point in the direction of the player's movement. If the ray hits the ground, it sets the player's position to the point where the ray hit the ground and updates the "isOnGround" boolean variable of the "playerManager" object to true. It also checks if the player has just landed from a fall, in which case it plays a landing animation.

If the ray cast does not hit the ground, it sets the player's "isInAir" boolean variable to true and applies a forward velocity to the player's rigidbody. If the player is on the ground, the function lerps the player's position towards the target position, which is the point where the ray hit the ground.

As shown in Figure 5.12, the implementation of the falling animation is less complex than the movement animation. This is because the falling animation does not depend on a dynamically changing movement speed variable, so it can simply play the 'Fall' or 'Land' animations without the need for a blend tree. However, implementing a blend tree could improve the overall aesthetic of this feature.

FIGURE 5.11: Red debug line demonstrating how raycasting is used to detect when the player character is falling, and whether they have landed on the ground

5.1.4 Camera Handling

The Camera Manager is responsible for smoothly following and rotating the camera based on user input while avoiding collisions with other objects in the scene, while maintaining a fixed distance and angle. It provides an essential aspect of gameplay by allowing the player to view the game world and interact with it. The script is divided into different functions that control the camera's behavior in different situations.

5.1.4.1 Rotation

The HandleRotation() function in the CameraManager script handles the rotation of the camera based on user input, as seen in 5.14. It updates the lookAngle and pivotAngle variables based on the mouse input control, as seen earlier. This value is multiplied by lookSpeed and pivotSpeed respectively. The pivotAngle is clamped between the minimumPivot and maximumPivot values to ensure that the camera does not rotate beyond certain angles, these values are Serialized Field ranges, changeable within the Unity editor. It then rotates the camera using Quaternion.Euler() to create a Quaternion rotation based on the calculated rotation angles. The rotation is applied to the CameraManager's transform component, which ultimately determines the camera's orientation in the scene. By updating the camera's rotation based on user input, the player can freely look around the game world and interact with it in a natural and intuitive way.

```
if (Physics.Raycast(origin, -Vector3.up, out hit, minimumDistanceNeededToBeginFall, ignoreForGroundCheck))
{
    // If the ground is detected, set the player on the ground
    normalVector = hit.normal;
    Vector3 tp = hit.point;
    playerManager.isOnGround = true;
    targetPosition.y = tp.y;

    if(playerManager.isInAir)
    {
        // Play the landing animation if the player was in the air for more than 0.5 seconds
        if(inAirTimer > 0.5f)
        {
            animatorManager.PlayTargetAnimation("Land", true);
            inAirTimer = 0;
        }
        else
        {
            animatorManager.PlayTargetAnimation("Movement", false);
            inAirTimer = 0;
        }
        playerManager.isInAir = false;
    }
}
else {
    // If the ground is not detected, set the player in the air
    if (playerManager.isOnGround)
    {
        playerManager.isOnGround = false;
    }

    if(playerManager.isInAir == false)
    {
        // Play the falling animation if the player is not busy
        if(playerManager.isBusy == false)
        {
            animatorManager.PlayTargetAnimation("Fall", true);
        }

        Vector3 vel = rigidbody.velocity;
        vel.Normalize();
        rigidbody.velocity = vel * (movementSpeed / 2);
        playerManager.isInAir = true;
    }
}
```

FIGURE 5.12: Partial implementation of HandleFalling functionality

5.1.4.2 Following

The HandleFollow() function moves the camera to follow the target using Vector3.SmoothDamp() function to interpolate between the camera's current position and the target position, gradually moving at a velocity determined by the followSpeed parameter. The camera's position is then set to the calculated target position, and the method calls another method called "HandleCollisions" to handle any collisions that may occur.

5.1.4.3 Collisions

The HandleCollisions() function is responsible for preventing the camera from clipping through objects in the gameworld and obstructing the player's view. As seen in Figure 5.16, it achieves this by using Physics.SphereCast() to check for collisions between the camera and other objects in the scene, essentially creating a sphere around the camera. The cameraCollisionOffset and minimumCollisionOffset variables are used to control the distance between the camera and other objects in the scene. If a collision is detected, the camera's target position is adjusted to avoid the collision.

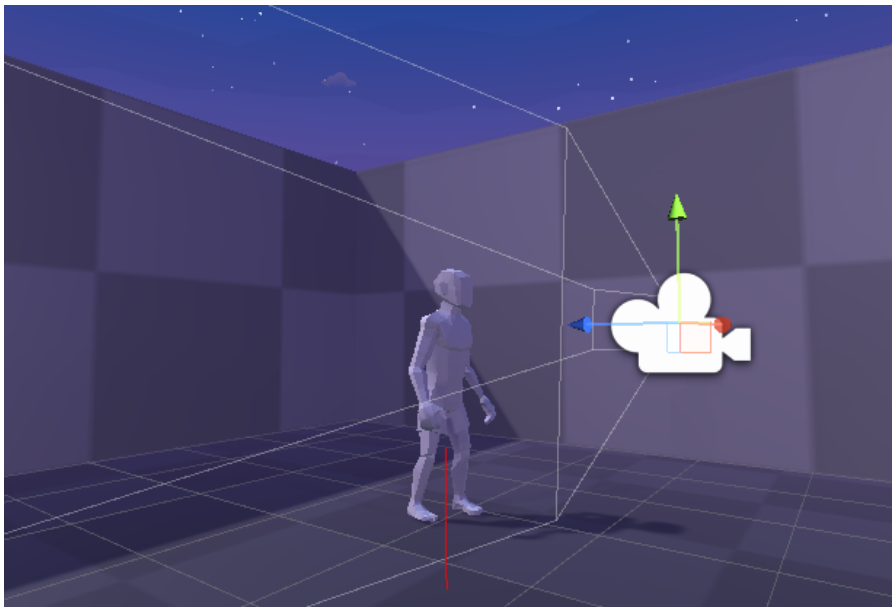

FIGURE 5.13: Camera object within the scene

```
1 reference
public void HandleRotation(float delta, float input_mouseX, float input_mouseY)
{
    // Calculate new look and pivot angles
    lookAngle += (input_mouseX * lookSpeed) / delta;
    pivotAngle -= (input_mouseY * pivotSpeed) / delta;

    // Clamp the pivot angle between minimum and maximum pivot angles
    pivotAngle = Mathf.Clamp(pivotAngle, minimumPivot, maximumPivot);

    // Set the look angle to the Y component of the rotation vector
    Vector3 rotation = new Vector3(0f, lookAngle, 0f);
    Quaternion targetRotation = Quaternion.Euler(rotation);
    myTransform.rotation = targetRotation;

    // Set the pivot angle to the X component of the rotation vector
    rotation = new Vector3(pivotAngle, 0f, 0f);
    targetRotation = Quaternion.Euler(rotation);
    cameraPivotTransform.localRotation = targetRotation;
}
```

FIGURE 5.14: Implementation of HandleRotation functionality

Specifically, the function calculates the direction of the camera from the pivot point, normalizes it, and checks for any object in the camera's path. If a hit is detected, it calculates the distance between the pivot point and the hit point and sets the new target position for the camera to avoid a collision with the object. If the new target position is too close to the pivot, the function sets the target position to the minimum collision offset to prevent the camera from getting too close to the character. Finally, it uses Mathf.Lerp to interpolate between the current camera position and the new target position and updates the camera position.

```
0 references
public void HandleFollow(float deltatime) {
    Vector3 targetPosition = Vector3.SmoothDamp(
        myTransform.position, targetTransform.position, ref cameraFollowVelocity, deltatime / followSpeed
    );
    myTransform.position = targetPosition;

    HandleCollisions(deltatime);
}
```

FIGURE 5.15: Implementation of source-code functionality

```
1 reference
private void HandleCollisions(float deltaTime)
{
    float newTargetPosition = defaultPosition;
    RaycastHit hit;

    // Calculate direction of camera from pivot and normalize it
    Vector3 cameraDirection = cameraTransform.position - cameraPivotTransform.position;
    cameraDirection.Normalize();

    // Check if there is an object in the camera's path and update target position accordingly
    if (Physics.SphereCast(cameraPivotTransform.position, cameraSphereRadius, cameraDirection, out hit, Mathf.Abs(newTargetPosition), ignoreLayers))
    {
        float distanceToHit = Vector3.Distance(cameraPivotTransform.position, hit.point);
        newTargetPosition = -(distanceToHit - cameraCollisionOffset);
    }

    // If the target position is too close to the pivot, update it to the minimum collision offset
    if (Mathf.Abs(newTargetPosition) < minimumCollisionOffset)
    {
        newTargetPosition = -minimumCollisionOffset;
    }

    // Interpolate between the current camera position and the target position and update camera position
    Vector3 cameraLocalPos = cameraTransform.localPosition;
    float cameraZPos = Mathf.Lerp(cameraLocalPos.z, newTargetPosition, deltaTime / 0.2f);
    cameraTransform.localPosition = new Vector3(cameraLocalPos.x, cameraLocalPos.y, cameraZPos);
}
```

FIGURE 5.16: Implementation of HandleCollisions functionality

5.1.5 Environment

Creating a well-designed environment is crucial in Unity because it can elevate the user's immersive experience, increasing their engagement with the game or application. It can also establish a sense of place and atmosphere, influencing the tone of the story or gameplay. Moreover, a thoughtfully crafted environment can serve as visual cues and facilitate player navigation, contributing to their comprehension of the game mechanics and objective completion. Finally, an aesthetically pleasing environment can enhance user satisfaction and create a lasting impression, ultimately improving the overall quality of the game or application.

5.1.5.1 Skybox

Incorporating a new skybox into your Unity project can substantially increase the visual aesthetics and ambiance of a scene, as seen in Figure 5.17. Essentially, a skybox is a cube that contains images on its inside faces and is utilized to replicate a background for the environment of the scene. These images can be of the sky, clouds, stars, and other environmental elements that can generate a more engaging experience for the user. By integrating a new skybox, you can transform the entire scene's appearance and atmosphere, creating a more captivating and visually striking environment for your game.

FIGURE 5.17: Addition of a skybox to the game scene

To add a new skybox from the Unity Asset Store in Unity, you can begin by opening the Unity Asset Store from the "Window" menu, followed by searching for a suitable skybox asset. After finding the right one, click on the asset's thumbnail to open its Asset Store page, and download it by clicking the "Download" button. Next, navigate to the "Assets" folder in your Unity project and drag the downloaded asset folder containing the skybox assets into it. Once the import process is completed, open the "Skybox" material in the "Materials" folder, and simply drag it onto the "Skybox Material" slot located in the Lighting window. This quick and straightforward process allows you to enhance the visual appearance of your scene with new and appealing skyboxes.

5.1.5.2 Prefabs

Prefabs are a useful tool for creating and reusing game objects with all their components, settings, and scripts intact. They are particularly useful for creating repeated instances of a specific object that requires customization. For example, the lamppost prefab in this project, as seen in Figure 5.18, can be customized with the desired lighting and design. Once the prefab is created, it can be easily duplicated and placed in the scene multiple times without the need to recreate the lighting and other settings for each instance. This not only saves time but also ensures consistency in the objects' appearance and behavior throughout the game. Prefabs can also be updated globally, meaning that any changes made to the prefab will be applied to all instances of that prefab in the scene.

To create a prefab in Unity, the user must select game objects from the Hierarchy or Scene view and drag and drop them into the Project window to create a new prefab asset. The prefab can then be used in the scene by dragging it from the Project window into the Hierarchy or Scene view, as seen in Figure 5.19. The Inspector window allows

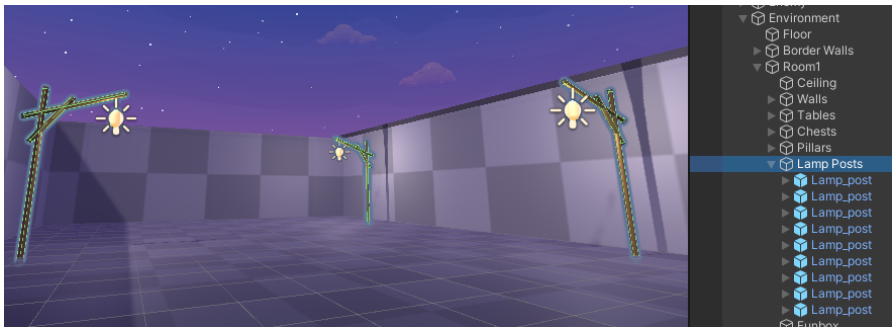

FIGURE 5.18: Reusing a prefab in the project scene

users to modify the properties of the prefab, and changes made to the prefab are reflected in all instances of it in the scene. Prefabs are useful for saving time and reusing objects across different scenes in Unity projects.

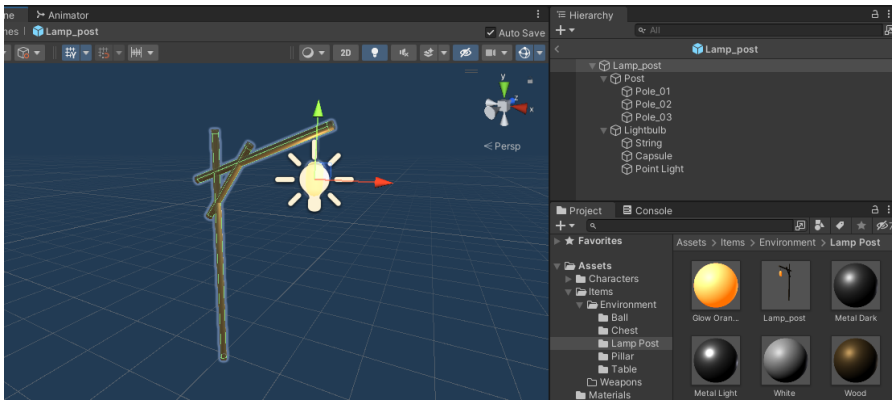

FIGURE 5.19: The creation of a prefab

5.1.5.3 Audio

Audio can enhance a game in Unity by setting the tone, adding depth and emphasizing key moments. It makes the game more engaging, immersive, and enjoyable for players, potentially providing direct feedback for player interaction.

In comparison to other elements in the project, adding a background track can be a relatively simple process. By creating an empty object in the project hierarchy, adding an audio source component, and specifying the required .mp3 file located in the assets folder, the process can be completed quickly and easily.

When it comes to spatial and directional audio, it's important to consider situations where certain sounds, like a person talking or a radio playing, should only be audible from a specific distance. This can add an extra layer of realism to the game environment. To implement this, you would need to use audio spatialization tools, such as Unity's 3D

sound settings, to specify the audio source's position, range, and directionality, and how it interacts with the game world.

5.1.5.4 Compass

To implement a compass in Blender that displays the cardinal directions, you can start by creating eight empty game objects, each located far away from the player in each direction: North, Northeast, East, Southeast, South, Southwest, West, and Northwest. These empty game objects will act as reference points for the compass. To make sure they are always located in the right direction relative to the player, they can be set to follow the player's movement. By doing this, the empty game objects will always stay a fixed distance away from the player in their respective directions. Once these reference points have been set up, you can use a simple algorithm to calculate the angle between the player's current direction/position and each of the eight reference points. If the angle is within a certain range, you can display the corresponding cardinal direction on the compass UI.

To make the compass point towards a specific target as seen by the green line in Figure 5.20, you can add a slot for an objective object. Whatever object is placed in this slot will be tracked on the compass. To make the compass point towards the objective, you can calculate the angle between the player's current direction/position and the objective object. If the angle is within the same bounds as the angle being checked for each cardinal direction, you can display a marker on the compass UI that points towards the objective. By following these steps, you can create a functional compass that provides the player with an easy-to-use interface for navigating the game world.

FIGURE 5.20: Compass UI component showcasing its functionality

5.1.5.5 Lighting

There are some examples of different lighting in this project, as seen in Figure 5.21. The DirectionalLight object acts as the sun, casting shadows on every object in a determined

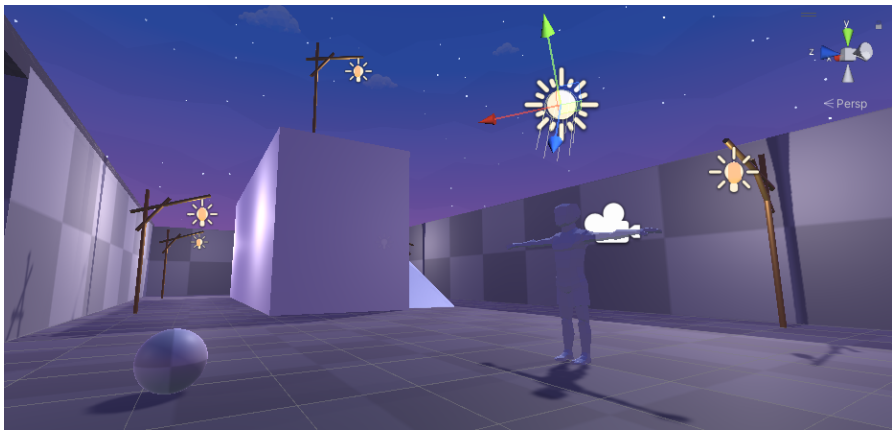

FIGURE 5.21: Lighting in the project scene

direction. It is possible to further simulate a light-build by creating an object with an emission, as seen in the lamppost prefab.

This project features different lighting examples, including the DirectionalLight object, which functions as the sun, casting shadows in a specific direction. Additionally, light-build simulation can be achieved by creating an object with emission, such as the lamp-post prefab.

For more realistic lighting, real-time global illumination can be set up by enabling it in the Lighting window and adjusting the settings. The color and brightness of the scene's lighting can be further customized by adjusting the environment lighting in the Lighting window.

Fine-tuning of lighting can be done by adding light sources, using tools like lightmap baking, light cookies, and light attenuation. However, this project does not cover those advanced techniques.

5.2 Difficulties Encountered

Throughout the project development, a significant number of functional difficulties surfaced, either through a lack of knowledge, or poor implementation logic. The issues varied in complexity and required different levels of effort to address. The following is a breakdown of the difficulties encountered, categorized as easy, medium, or hard, based on the complexity of resolving the issues.

- **Easy**: Managed to solve the problem with little difficulty.
- **Medium**: Not easy to solve, but managed to develop a workaround or solution and still achieve the functionality we originally had in mind.

- **Hard**: The difficulty was so complicated that solving it wasn't successful. As a result, some functional requirement / non-functional requirement or use case from the solution approach was not achieved.

The issues will be arranged in chronological order, beginning with the earliest and ending with the most recent.

1. **Hardware limitations:** Unity is a highly capable game engine that demands ample system resources to operate smoothly. Consequently, creating a project in Unity on a less potent device, like a laptop, can prove to be arduous and could result in a sluggish performance. Such a constraint can have adverse effects on productivity, making it challenging to work on the project expeditiously, necessitating a switch to a more potent desktop computer or restricting work to such a device exclusively.

   Difficulty == **Medium**

2. **Failing to push code to GitHub:** After starting development on the game, I realized that I couldn't push code to GitHub because the size of the files exceeded the 100MB limit. To solve this issue, it is essential to include a .gitignore file, which specifies what files should and should not be included when pushing code to the repository. The .gitignore file can be customized to exclude files like build artifacts, temporary files, and other non-essential files, which can bloat the repository and cause issues like slow downloads or failed pushes.

   After researching similar issues on Stack Overflow, I also learned that including the /Library folder in the repository is not recommended, as this folder contains generated assets that may differ on different computers. Instead, this folder should be excluded from version control and regenerated automatically upon importing and running the project on another computer. This can help to ensure that the project runs smoothly and consistently across different systems.

   Difficulty == **Easy**

3. **Non-stopping movement animation:** After successfully implementing the animation blend tree in my Unity project the movement animation continued to play at the highest velocity value, despite the actual physical movement ceasing. Further investigation revealed that the root of the problem was caused by the parameters in UpdateAnimatorValue() not resetting themselves when the input was stopped. I spent some time analyzing the code and found that the parameters were not being reset to their default values, causing the animation to continue playing at the highest velocity. To fix this issue, I had to update the code to ensure that

the parameters were reset to their initial values when the input stopped. This required careful attention to detail, but ultimately resolved the problem and allowed the movement animation to stop correctly when the physical movement stopped.

Difficulty == **Easy**

4. **Architectural Spaghetti Code:** Managing the game logic was increasingly difficult and time-consuming. Each function responsible for camera, movement, input, and more had its separate Update function and would reference each other randomly. To tackle this issue, I developed a PlayerManager script that acts as a central hub to manage the player object, including movement, input, and camera handling. This script is an entry point and references other critical player scripts, such as "PlayerMovement," "InputManager," and "CameraManager." As a result, the code is significantly streamlined, and its maintainability and organization are improved, making it easier to add new features to the game.

   Difficulty == **Medium**

5. **Stuck environment object corners while falling:** After successfully implementing the falling functionality, I encountered a new issue where the player model would frequently get stuck on object corners while attempting to run off. To address this issue, I added a small speed boost to the player's movement when the raycast detects that they are not on the ground. The speed boost allows the player to overcome the obstacle quickly, eliminating the problem entirely. To ensure that the speed boost doesn't look unnatural, I fine-tuned the boost so that it fits well with the game's overall gravity and physics mechanics. Specifically, I increased the fallingSpeed of the rigidbody by a factor of 1/5 in the direction of the moveDirection vector to give a slight speed boost to the player.

   Difficulty == **Easy**

6. **Combat:** At the outset of this project, I underestimated the complexity of the functional requirements, and as a result, could not implement combat functionality within the given timeline. The scope of the project would have to be significantly expanded to accommodate this feature, including the implementation of character statistics, attacking animations, and additional functionality. While it may be possible for readers to implement such features using the knowledge gained from this project, time remains the biggest constraint

   Difficulty == **Hard**

Chapter 6

Testing and Evaluation

It will be the focus of the following chapter to assess the project's achievements in light of its objectives identified during the research phase. Depending on the level of achievement, each objective will be rated as a Failure, Partial Success, or Success. The objectives are described in Chapter 4 and will be thoroughly examined. To determine the effectiveness of the system testing process, self-selected accuracy metrics will be utilized as part of the chapter.

6.1 Metrics

1. **To provide the reader with a comprehensive guideline 3D game development in Unity:** The reader should have sufficient technical knowledge to begin the game development process with feedback from friends, family, or classmates. A comprehensive explanation of Unity's core concepts and features has been provided, along with guidance on developing custom script functionality. Understanding these core concepts is crucial to the success of the project, and is one of the most important non-functional requirements for the game development process.

   Success == **True**

2. **Character Objects:** character objects have been successfully added to the project hierarchy, providing opportunities for custom functionality. For example, the player object showcases additional custom functionality which has also been implemented successfully.

   Success == **True**

3. **Movement:** The player object is now capable of moving around the game scene based on user input, using familiar controls. In addition, falling functionality has been implemented to enhance the gameplay experience.

   Success == **True**

4. **Character Statistics:** Currently, there is no implemented functionality to track character statistics within the game. This is an area for potential future development, as it allows for a much greater possibility for improvement in other functionality.

   Success == **False**

5. **Camera:** The camera position can now be adjusted based on user input in a familiar way. In addition, the handling of camera collisions has been improved to enhance the player's experience. With these features, the player may navigate throughout the game scene in run-time.

   Success == **True**

6. **Items** The game does not have any consumable or equip-able items. Implementing these features would first require further development of item objects, models, statistics, and UI.

   Success == **False**

7. **Combat** The game does not feature any sort of combat system. To introduce custom combat functionality, further development is first required for character statistics, item statistics, weapons, attack animations, etc.

   Success == **False**

8. **Inventory** No inventory system has been implemented. Such implementation would be required for the development of the functional requirement of item-containers. Further development is first required for item statistics, user interfaces etc.

   Success == **False**

9. **Compass** A compass UI component is located at the top of the screen, which features custom functionality. The compass points towards various parameters, including the cardinal directions of North, West, South, and East, as well as specific object markers. This provides players with an intuitive and easy-to-use navigation system, allowing them to orient themselves within the game world and locate points of interest more easily.

   Success == **True**

10. **Animations** The game incorporates numerous animations to give players feedback on their movements. One such animation, the movement animation, is modified using a blend tree that takes into account the player's current movement speed. As a result, the animation dynamically adjusts to the player's speed, resulting in a more interactive experience. By integrating visual cues through animations, players can better comprehend how their movements impact the game environment and feel more involved in their character's actions.

    Success == **True**

11. **3D Art** The game's models have been carefully crafted or sourced from external applications such as Blender and Mixamo. To ensure seamless integration with the game's mechanics, custom functionality has been developed to directly manipulate the skeletal mesh of these models, resulting in smooth movement animations. Furthermore, Unity materials have been utilized to enhance the appearance of 3D objects within the game world, creating a visually stunning and immersive experience for players.

    Success == **True**

12. **Enemy Advanced Intelligence Using Finite State Machines (FSM)** Other than the player object, characters in the game currently do not have any custom functionality and serve as static objects with collision colliders and rigidbodies. As a result, they do not have unique behavior or responses to the player's actions.

    Success == **False**

6.2 System Testing

1. **Character Objects:** The collision detection and physics simulation of character objects can be visualized using a sphere that represents their respective colliders and rigidbodies. When a character collides with the sphere, its position can be manipulated by applying forces to its rigidbody, which is initialized with a mass value of 1. The resulting movement of the sphere is determined by its assigned rigidbody mass value, and the strength and direction of the applied force by the player.

2. **Movement:** Within the game scene, a flat plane can be utilized to test the player's movement using the WASD keys. To add an element of challenge, a 3D object has been placed at a height to test the player's ability to navigate heights. The game utilizes raycasting, which is visualized through a red debug line gizmo, to detect collisions and ensure that the player doesn't fall through the object or the plane.

3. **Camera:** Collision-enabled objects have been strategically placed throughout the game to enable collision testing for the camera. These objects are designed to detect collisions with the camera, and if one occurs, the camera's position is adjusted to avoid the collision through custom functionality. This helps to ensure that the player has an unobstructed view of the game environment, and can move the camera around freely without worrying about colliding with objects.

4. **Compass:** Within the game scene, a flag has been set to a specific object to provide directional navigation towards its physical location. This feature is dynamic and adjusts in real-time based on the current location of the player object. As the player moves around the game environment, the flag's orientation and position are updated to continuously point towards the target object.

5. **Animations:** Unity's built-in blend tree functionality showcases the dynamic changes to animations based on specified parameter thresholds, which can be seen in real-time. Additionally, the 3D box and ramp located in the center of the game scene showcase the falling animation. However, a blend-tree is not utilized for the falling mechanic, instead the imported 'fall' or 'land' animations are played depending on the value of the isInAir flag.

Chapter 7

Discussion and Conclusions

7.1 Solution Review

I was able to successfully design a game template that the reader could easily build upon and improve. The template is structured in a way that allows for flexibility and customization, while still providing a solid foundation for future development. By using best practices and creating clear documentation, I aimed to ensure that the reader could quickly and efficiently modify the game to their desired specifications.

While I did not fully accomplish my initial goals, I must acknowledge that I underestimated the complexity of executing the functional requirements initially outlined. However, as the project progressed, I re-evaluated the necessary functional requirements for the reader's understanding and removed any that could, in hindsight, be considered superfluous. I consider the final implementation deliverable to be a significant and valuable contribution to the world of game development, specifically aimed at individuals who are new to the field, which was the primary objective of the project.

I successfully incorporated several key functionalities into my project, including player movement, camera control, user input, animations, and lighting. To enable movement, I crafted scripts to regulate the player's velocity and orientation based on user actions. I also designed a script that allowed for camera manipulation, with options for rotation and zooming. User input was facilitated via Unity's input system, which permitted the assignment of user activities to particular game functions. I employed the Animator component in Unity to add in intricate animations and seamless transitions. To create a more immersive environment, I integrated lighting through Unity's built-in lighting system. Overall, I'm extremely pleased with my progress and the knowledge I've gained from this endeavor.

In retrospect, I am very satisfied with the final outcome of my project and the proficiency I have gained through its execution. The project has provided me with invaluable insights into the world of game development, particularly in terms of designing and integrating diverse mechanics and features, which I am confident will be useful in my future endeavors.

7.2 Project Review

The importance of planning as a precursor to initiating a task was exemplified through personal experiences with time management and assignment completion. A plan was implemented and tasks were broken down into more manageable units to facilitate better organization and efficiency. A successful completion of tasks within a certain time frame depends on planning. To achieve both efficiency and effectiveness, the incorporation of planning into the starting stages of a project or assignment has become a priority for me in other modules also.

The completion of all assignments to the best of my ability is essential in order for me to achieve a high overall grade. The ability to manage time efficiently and prioritize tasks based on their importance and deadline is essential. It may be challenging to balance multiple assignments and commitments, but staying organized and focused is essential for me in order to succeed academically. I am confident that by consistently putting in the necessary effort and staying on top of my workload, I will be able to achieve the high grades that I am striving for. After all, it is my motto that "those who do more, out of worry of not doing enough, will always do the most".

Managing the scope of a project is crucial for its success, but it's not always easy to do. I've worked on Unity projects that were impacted by scope creep, leading to fewer features being implemented than planned. I discovered that Unity development can be challenging, even with careful planning and scope definition. Underestimating the difficulties can contribute to delays and make it harder to keep the project on track. In my experience, new requirements and features kept being added to the project, making it increasingly difficult to deliver the expected functionality. This led to some tough decisions, and we had to prioritize the essential features and scale back on the others to meet the deadline. I learned that it's essential to be flexible and adaptable to the changing requirements of a project, even when it's challenging.

The experience taught me valuable lessons about managing project scope and anticipating unexpected challenges, including the complexities of Unity development. Future Unity projects can be more successful by accounting for these factors, planning carefully,

and having realistic expectations. It's crucial to stay focused on the essential features, be prepared to adjust as needed, and deliver a functional project within the given timeline.

7.3 Conclusion

This research aims to streamline the process of creating games by highlighting key concepts and providing a general guideline for initial development. The motivation for the project comes from a lifelong passion for gaming and an interest in the technology and software behind it. The project aims to provide a practical application of the theoretical and practical knowledge gained during the completion of a Bachelor of Science in Software Development, facilitated by immense individual learning. As seen on GitHub, this project's contribution involves the creation of a dynamic three dimensional video game that can serve as a learning tool or template for those interested in advancing their knowledge in game development.

A game template was designed, providing a solid foundation for future development while allowing for flexibility and customization. Although the initial goals were not fully accomplished, the final implementation is subjectively considered a significant contribution to the world of game development, especially for individuals new to the field. Key functionalities were implemented and explained, such as player movement, camera control, user input, animations, environment, and lighting. The importance of planning, time management, and prioritization of tasks were highlighted in this project, which is crucial for academic success. Furthermore, effective project scope management and the ability to anticipate and mitigate unexpected challenges are crucial for the success of future software development projects, regardless of the specific platform or technology used.

7.4 Future Work

1. **Zoom In/Out** Adding the feature of zooming the camera in and out is a relatively straightforward addition in Unity. This feature would provide the player with more control over their perspective, allowing them to view the scene from various angles and distances. To implement this feature, additional input actions must be created to recognize mouse scroll input. The scroll input would then be used to directly control the targeted camera transform position, enabling the camera to zoom in and out. Overall, this feature could enhance the gameplay experience and give players more flexibility in how they interact with the game environment.

2. **Movement audio** Integrating movement audio should not differ significantly from adding background music, as previously discussed. However, it may require additional research to ensure proper synchronization of each footstep sound with each footstep taken. Careful attention to detail in timing and spacing may be necessary to achieve a more immersive and polished sound design.

3. **Statistics** Adding character statistics such as health and stamina would be a significant enhancement to this project, enabling a multitude of additional features like environmental damage, combat, and death mechanics. Such functionality would vastly expand the potential improvements to the project and enhance its overall gameplay experience. A Heads-Up Display (HUD) could be created to display such important information to the player in an intuitive manner.

4. **Inventory** Adding an inventory system that can store items and/or currency would be an amazing contribution for user experience. However, such functionality would further require additional custom scripting, such as item statistics. This feature could also be further supplemented with a user interface, where if properly designed, could give players more control over their in-game immersion.

5. **CombatManager** Due to the need to integrate various components, such as character statistics, animations, and sound effects, creating a combat system for the game would be a significant challenge. As a prerequisite to the implementation of a combat system, it is important to establish basic functionalities, such as character statistics, animations, sound effects, etc.

Bibliography

[1] S. E.-N. Jonas Heide Smith, Susana Pajares Tosca, "Understanding video games: The essential introduction." [Online]. Available: https://forskning.ruc.dk/en/publications/understanding-video-games-the-essential-introduction)

[2] R. Z. Robin Hunicke, Marc LeBlanc, "Mda: A formal approach to game design and game research." [Online]. Available: https://www.aaai.org/Papers/Workshops/2004/WS-04-04/WS04-04-001.pdf

[3] J. Nieminen, "How to refine and develop ideas – the doer's guide." [Online]. Available: https://www.viima.com/blog/developing-ideas)

[4] G. Scardamalia and C. Bereiter, "Beyond brainstorming: Sustained creative work with ideas." [Online]. Available: https://ikit.org/fulltext/2003_BeyondBrainstorming.pdf)

[5] M. T. C. Team, "Swot analysis." [Online]. Available: https://www.mindtools.com/amtbj63/swot-analysis)

[6] B. Matthews, "Intersections of brainstorming rules and social order." [Online]. Available: https://www.tandfonline.com/doi/epdf/10.1080/15710880802522403needAccess=true&role=button)

[7] C. Christensen, "The innovator's dilemma." [Online]. Available: https://books.google.ie/books?hl=en&lr=&id=K6FrJTWeUssC&oi=fnd&pg=PR4&dq=the+innovator%27s+dilemma&ots=3wySYzLBfw&sig=nJDyQBlBLBQW0IbQYGYOOyg49JQ&redir_esc=y#v=onepage&q=the%20innovator's%20dilemma&f=false)

[8] E. Ries, "The lean startup." [Online]. Available: https://mbrf.ae/storage/app/post/uploads/tUwXs3Vv70Ld8YTyj8PgbFenPyYzdgnegNRvl5XY.pdf)

[9] J. Horneman, "Play context." [Online]. Available: https://www.gamesindustry.biz/play-context#:~:text=Play%20context%20is%20determined%20by,played%20during%20a%20bus%20ride.)

[10] U. Ruhi, "Level up your strategy: Towards a descriptive framework formeaningful enterprise gamification." [Online]. Available: https://www.researchgate.net/publication/326311784_Level_Up_Your_Strategy_Towards_a_Descriptive_Framework_for_Meaningful_Enterprise_Gamification

[11] B. S. J. Morris, "The blank-page technique: Reinvigorating paper prototyping in usability testing." [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5467312

[12]

[13] V. Y. Syahrizal Dwi Putra, "Mda framework approach for gamification-basedelementary mathematics learning design." [Online]. Available: https://www.researchgate.net/publication/356064528_MDA_Framework_Approach_for_Gamification-Based_Elementary_Mathematics_Learning_Design)

[14] F. Leong, "Game paper prototype - the secret of the funfair." [Online]. Available: https://youtu.be/dt1bQsZ68iw?t=26)

[15] D. A. Clearwater, "What defines video game genre? thinking about genre study after the great divide." [Online]. Available: https://journals.sfu.ca/loading/index.php/loading/article/view/67/105

[16] J. Juul, "Half real: Video games between real rules and fictional worlds." [Online]. Available: https://is.cuni.cz/studium/predmety/index.php?do=download&did=27982&kod=JJM169

[17] G. Wiki, "Jesper juul's classic game model." [Online]. Available: https://books.google.ie/bookshl=en&lr=&id=6IyqCNBD6oIC&oi=fnd&pg=PA195&dq=Csikszentmihalyi%E2%80%99s+theory+of+flow&ots=INIdUDWbrA&sig=GHOyHna7nUrMq5g7Ir4PXyjBAbs&redir_esc=y#v=onepage&q=Csikszentmihalyi%E2%80%99s%20theory%20of%20flow&f=false)

[18] M. Csikszentmihalyi, "Flow and the foundations of positive psychology." [Online]. Available: https://link.springer.com/chapter/10.1007/978-94-017-9088-8_15)

[19] S. M. Nikolakak, "Competitive balance in team sports games." [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9231859)

[20] C. Pitt, "The game loop." [Online]. Available: https://link.springer.com/chapter/10.1007/978-1-4842-2493-9_2

[21] E. R. Tveterås, "Evaluating loss and latency mitigation techniques in a tick-based game server." [Online]. Available: https://www.duo.uio.no/bitstream/handle/10852/65783/1/Eirik_Tveteraas_Masteroppgave.pdf#page=90&zoom=100,90,830

[22] S. D. Community, "Source multiplayer networking." [Online]. Available: https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking

[23] N. Entertainment, "Tick rate: What is it and how can it affect your gaming experience?" [Online]. Available: https://www.nbnco.com.au/blog/entertainment/what-is-tick-rate-and-what-does-it-do

[24] N. Souto, "Video game physics tutorial - part i: An introduction to rigid body dynamics." [Online]. Available: https://www.toptal.com/game/video-game-physics-part-i-an-introduction-to-rigid-body-dynamics#:~:text=A%20rigid%20body%20is%20like,sounds%2C%20especially%20in%20three%20dimensions.

[25] D. N. Pronost, "Game physics - game and media technology." [Online]. Available: https://perso.liris.cnrs.fr/nicolas.pronost/UUCourses/GamePhysics/lectures/lecture%204%20Rigid%20Body%20Physics.pdf

[26] N. Souto, "Video game physics tutorial - part ii: Collision detection for solid objects." [Online]. Available: https://www.toptal.com/game/video-game-physics-part-ii-collision-detection-for-solid-objects

[27] J. Huynh, "Separating axis theorem for oriented bounding boxes." [Online]. Available: http://jkh.me/files/tutorials/Separating%20Axis%20Theorem%20for%20Oriented%20Bounding%20Boxes.pdf)

[28] G. V. D. Manocha, "Accurate minkowski sum approximation of polyhedral models." [Online]. Available: https://ieeexplore.ieee.org/abstract/document/1348370)

[29] A. H. Cummings, "The evolution of game controllers and control schemes and their effect on their games." [Online]. Available: https://d1wqtxts1xzle7.cloudfront.net/4966311/6-libre.pdf?1390839515=&response-content-disposition=inline%3B+filename%3DThe_evolution_of_game_controllers_and_co.pdf&Expires=1671736613&Signature=F520oxguFLKsajpFE6IQUp2YsWcPovz2BCqVyxcnPO20k~VM-DCyVaVuJN6Fz8rWRLUDj51LI_&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA

[30] M. Masuch and N. Röber, "Game graphics beyond realism: Then, now, and tomorrow." [Online]. Available: http://www.digra.org/wp-content/uploads/digital-library/05150.48223.pdf

[31] M. T. Susan Hallam, Ian Cross, "The oxford handbook of music psychology." [Online]. Available: https://books.google.ie/books?hl=en&lr=&id=d-2DYVjNVpQC&oi=fnd&pg=PT238&dq=how+does+music+evoke+

emotion&ots=aqTkljzULG&sig=r4sr75_sbxDCUsHOi9ZA_6BfzMg&redir_esc=y#v=onepage&q=how%20does%20music%20evoke%20emotion&f=false

[32] U. Documentation, "Unity manual: Scenes." [Online]. Available: https://docs.unity3d.com/Manual/CreatingScenes.html

[33] ——, "Unity manual: Gameobjects." [Online]. Available: https://docs.unity3d.com/Manual/CreatingScenes.html

[34] ——, "Unity manual: Primitive and placeholder objects." [Online]. Available: https://docs.unity3d.com/Manual/PrimitiveObjects.html

[35] ——, "Unity manual: Static gameobjects." [Online]. Available: https://docs.unity3d.com/Manual/StaticObjects.html

[36] ——, "Unity manual: Scripting." [Online]. Available: https://docs.unity3d.com/Manual/ScriptingSection.html

[37] S. Tonight, "Adding a c script to unity game project." [Online]. Available: https://www.studytonight.com/game-development-in-2D/basics-of-unity-script

[38] U. Documentation, "Unity manual: Materials." [Online]. Available: https://docs.unity3d.com/Manual/Materials.html

[39] ——, "Materials and shaders." [Online]. Available: https://docs.unity3d.com/430/Documentation/Manual/Materials.html

[40] Unity Technologies, *Roll-a-Ball*, Unity Technologies. [Online]. Available: https://learn.unity.com/project/roll-a-ball

[41] ——, *Create with Code*, Unity Technologies. [Online]. Available: https://learn.unity.com/course/create-with-code

[42] T. M. Alexandre Decan, "What do package dependencies tell us about semantic versioning?" [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8721084/authors#authors

[43] M. McCormick, "Waterfall vs. agile methodology." [Online]. Available: http://www.mccormickpcs.com/images/Waterfall_vs_Agile_Methodology.pdf

[44] J. M. Muhammad Ovais Ahmad and M. Ovio, "Kanban in software development: A systematic literature review." [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6619482

[45] J. K. Liker, "The toyota way: 14 management principles from the world's greatest manufacturer." [Online]. Available: https://www.accessengineeringlibrary.com/binary/mheaeworks/1efa283f2a882c2e/02831bfbff6f4a5387fa0989660278fec3d80c7ae35b19cd5bb68fbf4d61e0ad/book-summary.pdf