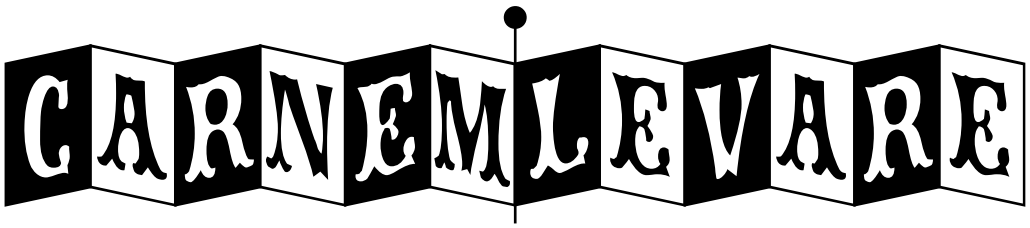
UNIVERSIDAD
DE GRANADA

ETSIIT
Escuela Técnica Superior
de Ingenierías Informática
y de Telecomunicación

Bachelor's Degree in
Computer Science and Engineering

Undergraduate Dissertation



Design and implementation of a Player-NPC interaction system

Academic course: 2022/2023

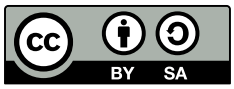
AUTHOR:

Guillermo García Arredondo

TUTOR:

Pablo García Sánchez

Department of Computer Engineering, Automation and Robotics



Guillermo García Arredondo

Computer engineer, game designer, amateur writer and soon to be founder of the game studio Parry Mechanics, if Fate wills it. He completed University of Granada's computer science degree in 2023, starting his journey as an indie game developer immediately after. His joy in learning has driven him to develop a diverse range of interests, such as calisthenics, martial arts, playing guitar, drawing or hiking. Nonetheless, games sit among its top hobbies, as both powerful tools to tell meaningful stories and to shape deeper connections with other people.

Pablo García Sánchez

Associate Professor at the Department of Computer Engineering, Automation and Robotics of the University of Granada and the current director of the Free Software Office of the Vice-rectorate for Digital Transformation. He has been one of the organisers of the Evostar joint conference on Evolutionary Computing from 2014 to 2019. His interests include service-oriented computing, evolutionary computing, computational intelligence in video games, distributed algorithms, free software and open science.

Cite this work:

```bibtex
@mastersthesis{citeKey,
 author = "Guillermo García Arredondo and Pablo García Sánchez",
 title = "Carnem-Levare: Design and implementation of a Player-NPC interaction system",
 school = "Escuela Ténica Superior de Ingeniería Informática y Telecomunicaciones",
 year = "2023",
 month = "September",
 address = "Periodista Daniel Saucedo Aranda, s/n, 18014 Granada, Granada, Spain",
 type = "Undergraduate Dissertation"
 }
```

Carnem-Levare: Design and implementation of a Player-NPC interaction system

Keywords:

Videogame, Indie, GDD, Frame Data, Software Design, SOLID, MVC, Observer, AI, Game Development, Unity

Abstract:

This dissertation describes the development of a yet to-be-released indie action game titled Carnem-Levare. It aims to showcase the design and implementation required to craft its core gameplay, a Player-NPC interaction system in the form of combat. This dissertation's work should result in a playable prototype where the player is able to challenge an AI-controlled opponent over and over until they defeat them.

The process in which this system will be implemented follows a pseudo-typical game development procedure, starting by establishing a game design document to define what the game is about and how is it played. This GDD will go into detail about all aspects pertaining to the combat system, including the environment, player's available actions and the enemy they face.

From this specification we will then draw software requirements from which to start designing a generic, engine-independent system's architecture. We will research and put to use software patterns and principles to ensure a clean and robust implementation, minimizing interdependence between modules while maximizing cohesion within said modules.

Once the software architecture is successfully designed, it will be put to the test by implementing it in a competitive game engine, more specifically Unity. We will research and explain all features relevant to implementing the architecture, along with possible improvements to the design resulting from the engine or, on the contrary, workarounds because of engine limitations. At any case, this should result in the prototype we set out to develop and therefore conclude this dissertation.

Carnem-Levare: Diseño e implementación de un sistema de interacción entre jugador y NPC

Palabras clave:

Videogame, Indie, GDD, Frame Data, Software Design, SOLID, MVC, Observer, AI, Game Development, Unity

Resumen:

Este trabajo de fin de grado detalla el desarrollo de un videojuego indie de acción aún no estrenado titulado Carnem-Levare. Se pretende presentar el diseño e implementación necesario para hacer el núcleo del gameplay, que consiste en un sistema de interacción entre jugador y NPC simulando un combate. El trabajo realizado en este proyecto debería resultar en un prototipo donde el jugador puede desafiar a un oponente controlado por IA una y otra vez hasta derrotarlo.

Se seguirá un pseudotípico proceso de desarrollo de videojuegos para implementar este sistema, partiendo de un documento de diseño de videojuego que define de qué trata el juego y cómo se juega. Este documento entrará en detalle sobre todos los aspectos relativos al sistema de combate, incluyendo el entorno, las acciones disponibles al jugador y el enemigo al que se enfrenta.

De esta especificación extrapolaremos una serie de requisitos de software desde los cuales empezaremos a diseñar la arquitectura del sistema, genérica e independiente al motor. Investigaremos y pondremos en uso patrones y principios de software para asegurar que la implementación es limpia y robusta, minimizando la interdependencia entre módulos a la vez que maximizamos la cohesión entre dichos módulos.

Una vez la arquitectura software esté diseñada, será puesta a prueba implementándola en un motor de videojuegos competitivo, más específicamente Unity. Investigaremos todas las características y herramientas relevantes para implementar la arquitectura, además de posibles mejoras al diseño dadas por el motor o, al contrario, soluciones alternativas que sean necesarias por limitaciones de dicho motor. En cualquier caso, esto debería resultar en el prototipo que nos propusimos desarrollar y por tanto concluirá este trabajo de fin de grado.

"Now that I am Undead,
I have come to this great land,
the birthplace of Lord Gwyn,
to seek my very own sun!
Do you find that strange?
Well, you should!"

Solaire of Astora
Dark Souls, 2011

Contents

List of Figures

Glossary

A | C | F | G | H | I | J | M | N | P | R | S | T | V

A

Action Videogame genre that emphasizes physical, real-time challenges that demand precise hand-eye coordination, quick reaction time and keen spatial perception, usually in the form of combat between characters. 1, 9

Ad-hoc behaviour authoring method Artificial intelligence technique which refers to the process of defining specific behaviour for an AI system on a case-by-case or as-needed basis. It involves manual or semi-automated design of behaviours to handle a specific set of situations the AI may encounter. 57

C

Coupling Degree of interdependence between software modules. 4, 39

F

Fighting Videogame genre that involves combat between two players or more. 11, 12

G

Game Feel The tactile sensation and joy experienced from interacting in a videogame, depending on response, context, aesthetic and rules [1]. 18

Gameplay The specific way in which players interacts with a game, defined by its rules, challenges and rewards, and the emotions this interaction triggers [2] [3]. 2, 9, 15

H

Hook Type of punch in martial arts where the fist is swung in an horizontal arc to the opponent. 27

I

Indie Independent game created by individuals or small development team without the financial and technical support of a large game publisher [4], that often focus on innovation and experimental gameplay thanks to their intrinsic development freedom [5]. However, it's important to note that this is a broad definition and many games considered "Indie" may lack some of the requirements mentioned above, blurrying the line between independent and superproduction [6]. 1

Input Buffering In the context of game design, processing the player's input for a move or action before the last execution is finished, so that the new action is performed in the earliest frame possible after the previous action ends. 53

J

Jab Type of punch in martial arts where the lead fist is thrown straight ahead, fully extending the arm from the side of the torso. 24

M

Mixup Situation where the offensive player has several ways to attack that each require a different defensive action to stop. Most mixups contain several fast options that are extremely difficult, or impossible, to avoid on reaction, and thus the defensive player must make a read to escape taking damage [7]. 30

Model-View-Controller Software design pattern that divides the program logic into three interconnected components: Model (application's data structure, logic and rules), View (visual representation of the model), Controller (converts input to commands for the model or view). 3

Moveset Complete set of actions, attacks or abilities a character can perform in a videogame. 23

N

NPC Non-playable character. 2

P

Percept The input an intelligent agent is perceiving at any given moment. 54

Platformer Videogame genre where the core objective is to move the player character to a predetermined goal through uneven terrain, suspended platforms and obstacles. 11

R

Recovery Fighting videogame term that refers to the ending period of an animation where the player must wait for a specific *recovery time* to regain control, during which they are vulnerable [8]. 15

Responsiveness In the context of videogames, the software's capacity to react quickly and accurately to player's inputs. 53

Rig A hierarchical structure of interconnected joints used to identify the virtual bones which a 3D model can move. 71

RPG Abbreviation for *role-playing game*. Videogame genre where the player controls a self-created character immersed in a fictional world, usually involving some sort of character development in the form of upgrading statistics and equipment. 18

S

Scene Distinct software state of a videogame which holds its own assortment of data, objects and behaviour. 49

Side-scroller Refers to a specific videogame perspective where the game is viewed from a side-view camera that follows the player left or right, usually 2D but not necessarily. 11

SOLID Acronym for five design principles intended to make object-oriented architecture more understandable, maintainable and flexible. 3

Spamming Spamming in a videogame refers to the repeated use of the same action in a videogame, usually to the detriment of the game's enjoyment. 27

Sprite Two-dimensional image that represents a character, object or element within the game world. 47

T

Third-person Graphical perspective in a 3D videogame where the camera sits behind and slightly above the player character, putting them into view [9]. 9

V

VoIP Acronym for Voice over Internet Protocol. It refers to the group of technologies intended for the delivery of voice communication sessions over IP networks such as the internet. 4

Chapter 1

Introduction

This dissertation follows the development of an indie singleplayer action game titled Carnem-Levare, intended to be released as a commercial product a few years after this paper is issued. It focuses on the framework that make up the core gameplay of the game, leaving other fundamental aspects —such as menu or sound integration— for my co-worker's dissertation: *Carnem-Levare: Basic systems and structures of a videogame*, by Alejandro Cruz Lemos.

This two dissertations together should result in a prototype that aims to showcase the main mechanics through a simple game loop that allows the player to retry a specific challenge over and over until they overcome it, while also being able to customize their experience through in-game menus and interfaces.

1.1 Motivation

Videogames are currently the highest grossing entertainment industry in the world. Estimated global revenue in 2021 was 192.7 billion US dollars, in contrast to films' 99.7 billion or music's 25.9 billion (see Figure 1.1). Most of this yearly revenue is generated by mobile games, which made 92.2 billion US dollars in 2022 alone. However, console and PC games —the intended platform of this dissertation' work— are not far behind, totalling to 90 billion instead in the same year [10].

Such a lucrative industry is necessarily supported by the increasing number of people that buy and play these games. By 2023 —the year when this paper is written—, there are currently 3.24 billion players worldwide, making for a 32% increase in the last seven years (that is, over a billion increase) [11]. And while, again, most of them are mobile players, around 50% do play on PC (including those who play on both platforms).

Now then, when we mention PC, we usually refer to Steam, the leading digital videogame web store [12]. The steam store has a community-based classifying system where users may apply tags to a game's store page defining the genre and other relevant features. The most popular tags —that is, those most frequently applied— establishes which browsing pages the game will appear on and what other games does it resemble —for recommendation purposes— [13].

The three most popular tags are "Indie", "Singleplayer" and "Action", in terms of both game count and revenue [14]. Steam's top selling games of all time by tag are also "Singleplayer" and "Action" [15]. From this data we can conclude that, while there is significant competition, due to the rapidly growing nature of the videogame industry and current gaming trends, singleplayer action PC games are a niche market worth exploring.

In a world where super productions focus on risk-free remakes and sequels of previously successful videogames, there is an increasing demand for innovative singleplayer experiences that is only ever expected from independent developers. The purpose of this dissertation is thus contributing to a singleplayer videogame worth of the previously mentioned niche, establishing the foundations of an eventually marketable product.

Moreover, financial prospect is not the only reason for this project's consummation. In the last decade, videogames have attained greater public recognition as a medium capable of

1

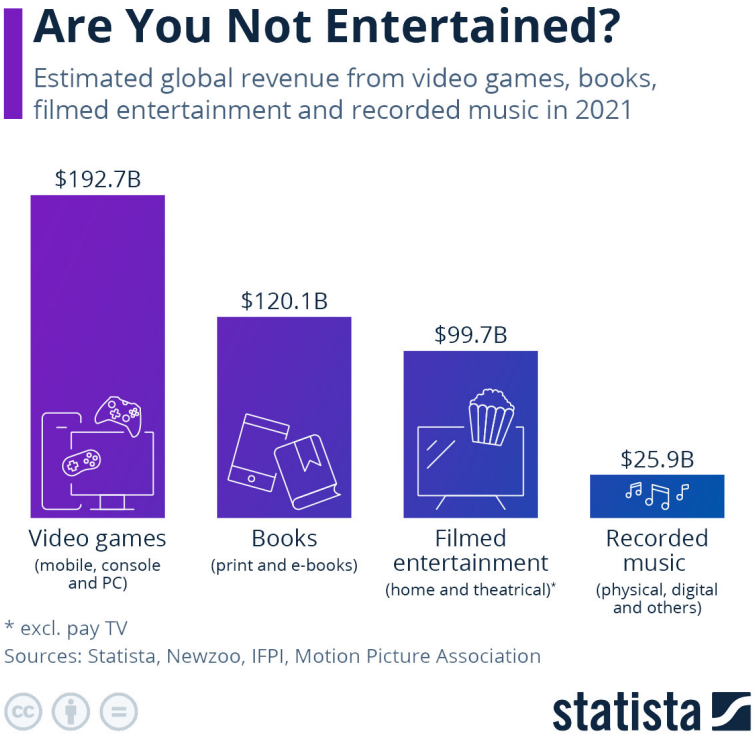


Figure 1.1 – Estimated global revenue from entertainment industry in 2021. Statista.

artistic expression, and not just as a potentially profitable software product. This is evidenced by museums dedicating exhibitions to the medium such as the Smithsonian American Art Museum's The Art of Video Games [16], or critically acclaimed mainstream titles such as Journey, Disco Elysium or Papers Please [17] [18] [19], that either stray from common videogame formulas or use them to their advantage to tell a meaningful story.

It is in this dissertation' best interest to create a product where engineering and art merge to offer a emotionally valuable and fulfilling experience capable of matching the aforemetioned works, and even other mediums more commonly referred to as "art".

1.2 Objectives

This dissertation's main objective is to develop a Player-NPC interaction system that will constitute the product's core gameplay, showcasing the software design and implementation that such a system may require.

Said software design will follow a pre-established game design document that defines the gameplay to be implemented, while allowing further design to be made directly into the game in the form of balancing to parameters. It is yet another objective of this project to provide an accessible infrastructure from which game designers can continue designing the game without coding.

The interaction system to be implemented is what is commonly denoted as a "combat system". That is, a type of interaction where the player must confront and defeat an opponent —either NPC or another player—, by acquiring the mechanical skill or power advantage neces-

sary beforehand. Since this game is singleplayer, the opponent will always be an NPC controlled by artificial intelligence.

Now, this so-called combat system is actually a combination of many tasks and processes that must play out in the right order and harmony to achieve the desired effect. Following the game design document, we will identify said tasks and divide it in subsystems through the use of software design patterns such as Model-View-Controller and SOLID for better performance, readability and scalability.

The resulting software architecture will be implemented in Unity, a free[1] and currently most popular game engine in the market due to its large community and comprehensive documentation.

We may summarize this entire process in the following sub-objectives:

1. Establishing a game design document that will describe in detail the intended gameplay experience, following a typical GDD structure to define all areas relevant to the implementation, such as: target, game world, actions, enemies, etc.
2. Inferring software requirements and systems from the GDD's ambiguous gameplay specifications.
3. Investigating the software design patterns that will allow us to design an efficient and maintainable architecture to support the requirements previously inferred.
4. Designing the software architecture in such a way that it will allow game designers to balance the game without resorting to code.
5. Implementing the resulting software design in a competitive game engine, complementing the rest of systems already implemented in the game beyond this dissertation.

1.3 Dissertation Structure

This dissertation is consequently divided in the following chapters, which aim to tackle and resolve the several objectives and sub-objectives previously mentioned:

- **Introduction** (chapter 1). The current chapter, which intends to outline the motivation behind this project and the process that will be followed to achieve its objectives.
- **Game Design** (chapter 2). The description of the "problem" this dissertation aims to solve. A detailed game design document featuring the areas relevant to the creation of a player-NPC interaction system.
- **Software Design** (chapter 3). The design of the software architecture resulting from analysing the game design document, composed of the software patterns that said architecture will require and how they should be implemented.
- **Implementation** (chapter 4). A specific case of implementation in Unity, describing the differences and nuances from the software design that result from programming in a specific game engine.

[1] Free for personal use or smaller companies generating less than $200,000 anually.

1

- **Conclusion** (chapter 5). A summary of the previous chapters and achieved objectives. Thoughts about the complexity of designing and implementing gameplay, along with the necessity of software design patterns. Future works and expectations about Carnem-Levare.

1.4 Planning

Planning was a key aspect of this dissertation to ensure that the final prototype was completed in a realistic amount of time, being able to stick to a specific deadline without rushing the project. Not only it had to be broken down into manageable pieces to design and develop, but it also had to be coordinated with my co-worker's project.

As it was stated at the beginning of this chapter, this dissertation focuses on the core gameplay of a to-be-released action game known as Carnem-Levare. Other fundamental systems such as menus or sound integration are left for to be realized by my co-worker Alejandro Cruz Lemos. While these areas are not necessarily related, we had to coordinate to ensure that we both designed our parts to easily complement each other in a way that was both efficient and easy to use. To ensure that the whole architecture was maintainable and had the least possible amount of coupling.

For that reason, most of this prototype was developed in constant communication (following a typical reduced labor schedule) and organised through the use of methodologies and tools that facilitated said development.

1.4.1 Schedule and methodology

To begin with, we followed a flexible 20-hour work week for 9 months of development. It was intended to be arranged from Monday to Friday in days of four hours, so that we would be working at the same time. However, we allowed days to be taken off or reduced with the idea of fulfilling the missing hours later in the week.

This schedule was adopted through remote work and the use of Discord, an instant messaging and voice call software which facilitated communication between each other [20]. The distinctive feature about Discord over other VoIP applications is the inclusion of *servers*. Servers are communities consisting of collections of chat rooms and voice channels. This collections allows servers to be organized and repurposed for any given task, such as having different chat rooms depending on topics.

Using a Discord server as a work environment, we were able to communicate in real time and organise notes and resources through different text channels. While it was not the main tool to coordinate tasks, it served as a fundamental space for the development of the prototype.

Furthermore, along with Discord we used GitHub for version control and task management. GitHub is a cloud-based platform for software development that uses Git to allow developers to store and manage their code [21]. It allowed us to safely back up the in-engine project in the internet with an extensive version history, while also keeping each other up-to-date with every new feature. The reader may take a look at the repository at github.com/ggarredondo/Carnem-Levare.

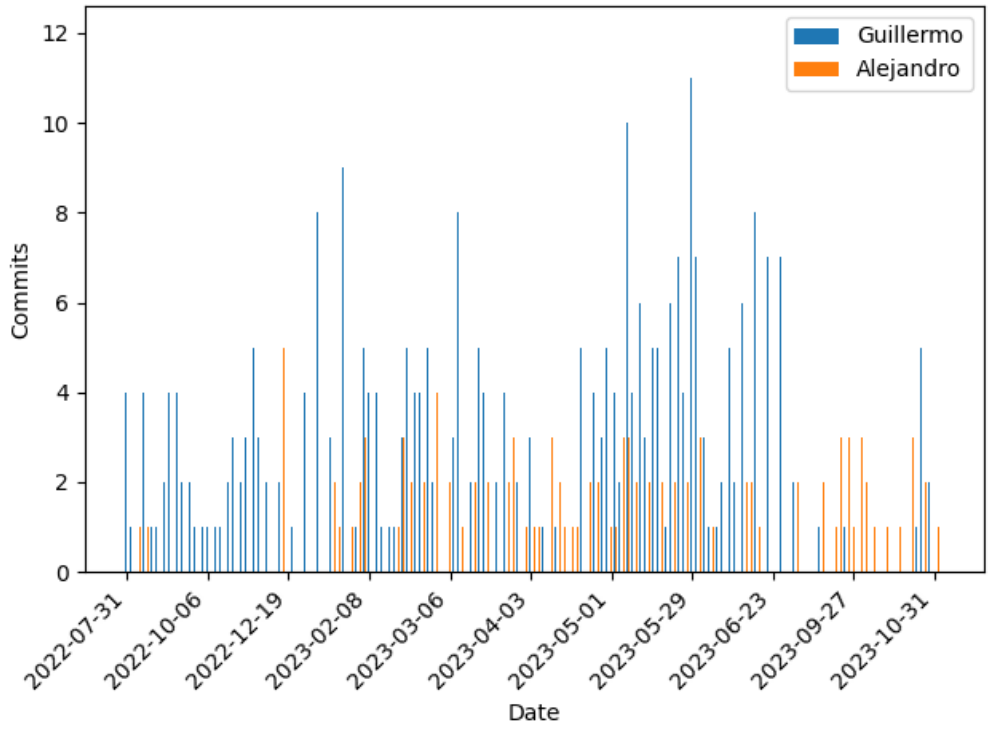


Figure 1.2 – Commits made to Carnem-Levare by date and contributor.

Task management came in the form of *issues*. Issues in GitHub allows users to report bugs or problems in a given project, suggest enhancements, new features, etc. They serve as notes which can be added into a project to keep track of tasks. Moreover, these issues can be distributed in a spreadsheet, similar to a Kanban board (see Figure 1.3).

In the case of Carnem-Levare, we distributed our work in the form of issues spread between five different columns: features, improvements, bugs, invalid and backlog.

- *Features* involves all new, previously planned additions and systems to be implemented into the game. Tasks such as implementing knockback to attacks, adding a dash mechanic or implementing hitstun decay may be included and tracked through this column.
- *Improvements* generally refer to smaller changes to current features that might improve the game's functionality and feel. For example, adjusting the speed of an attack, adding new methods to a class, etc.
- The *Bugs* column is self-explanatory. All behavior which is not intended or leads to software errors are tracked there, each with a detailed description of the context in which the bug is taking place.
- *Invalid* refers to details or features that, while they do not break the program and might even be intended, they are considered undesirable after testing or further inspection. They are not bugs as such but must be corrected still.
- *Backlog* include all tasks of lesser priority in regards to the current objectives of the project, and are left there to be conducted where no other task of higher priority is left.

This configuration allowed us to categorize the different tasks that the project required and establish a simple priority system through the use of column height (which issue is higher in each column) and a specific backlog column.

1

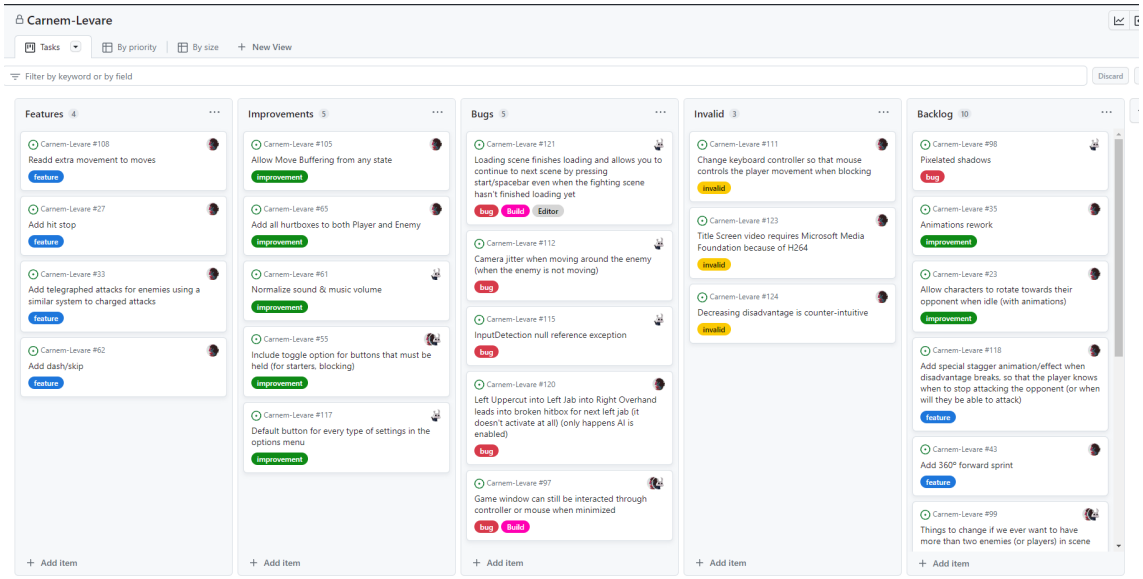


Figure 1.3 – GitHub's project planning for Carnem-Levare.

However, neither the schedule nor the issue classification were effective tools in planning the tasks long-term. For that purpose, we used Scrum methodology.

Scrum is an agile team collaboration framework which breaks work into goals to be completed within timed iterations known as *sprints*. Sprints are meant to be short periods of time, from one to a maximum of four weeks, and they may be split in the following stages: planning, design, development, testing, deployment and review (see Figure 1.4) [22].

Planning is the stage in which the team discusses and agrees on the goal that will be fulfilled by the end of the sprint. The product owner presents a set of prioritized items from the project's backlog and the team selects the items they believe they can complete during this sprint, thus setting a goal.

Design, development and testing are all phases in which the team works towards the stated goal. They are usually accompanied by a daily scrum meeting where developers gather to discuss the current progress made in the sprint, issues that may have arisen during development and plans for the day. A meeting of about 15 minutes for the purpose of team communication.

Finally, when these three stages finish, the resulting product is deployed and consequently reviewed. At the end of the sprint, the team holds a final meeting for each member to share the work they have completed with stakeholders, receive feedback and discuss expectations and upcoming plans for the project. Afterwards, a new sprint begins.

In the case of Carnem-Levare, since our work was independent, we would set personal sprints of one to two weeks to work on a set amount of features, drawn from game requirements or backlog. Still, daily scrums were held to inform each other about current progress and new implementations in the project, keeping each other updated.

First few days of the week usually involved designing the software architecture that said features would require. Development consisted of implementing said design, and testing usually resulted in bug fixes and improvements to the implementation.

For the shake of conciseness, the specific sprints will not be described in this paper. One week sprints in the course of 9 months resulted in around 36 sprints which, if reported and

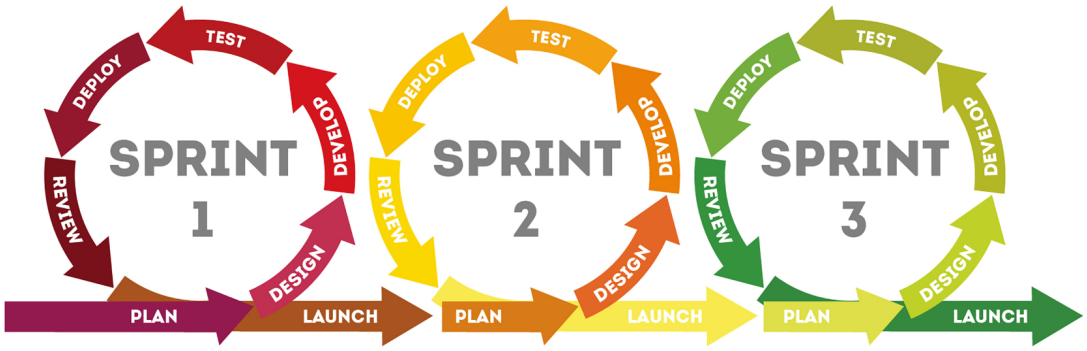


Figure 1.4 – Scrum's sprint cycle.

detailed individually, would only obscure this dissertation's findings and hinder its readability.

Nonetheless, this does not diminish the significance and utility that said methodology had in the project's development. It allowed us to implement the prototype's expected features in time without issues or setbacks. We took advantage of the Scrum process for development, while avoiding a granular breakdown of each sprint for a more streamlined narrative.

1.4.2 Costs

In this subsection, we will make a project cost estimation. It does not only take into account the contribution presented in this dissertation, but the overall development of the prototype that resulted from my coworker's and own work.

Considering prototypes are an early-stage production made to test game mechanics and design, the reader may extrapolate this costs over years of development to get a general idea of the budget necessary to produce an indie action videogame.

- 20 hours a week made by two software engineers during 9 months of work, for an average of $36.40 an hour for junior game developers in United States -> $52,416
- Unity ART Humble Bundle, which included some assets and tools that we used for this project -> $25
- Fighting Animset Pro, collection of animations used for the combat system -> $79.84
- Shapeforms Hit and Punch Essentials, a sound effect packs used for attacks -> $6.53

Totalling to a cost of $52,527 for the entire prototype, omitting other fundamental expenses such as electricity. It becomes clear how expensive game development can be, even for an indie production, which makes proper planning and methodology key for an efficient time-cost approach.

Chapter 2

Game Design

Before any software design takes place, there must be a gameplay specification from which programmers can begin designing and implementing. This chapter will describe said gameplay following the structure of a typical game design document, with a few exceptions.

A GDD (game design document) is a software design document used to describe how a videogame will play and feel [23]. During pre-production stage, a basic GDD might be developed to pitch the game to the publisher [24]. Its primary purpose is to excite the reader, encouraging them to read more and think of the potential success of the game [25], but it is mostly conceptual and likely incomplete.

A conceptual, incomplete GDD is not going to suffice for this dissertation' purpose. We are far from pre-production stage, and we need something concrete that can be implemented with code. Therefore, this dissertation' design chapter should resemble something more like a mid-production GDD.

When the project is approved, the previously basic, pre-production GDD is expanded upon by the designers, with the objective of providing a guide with which to organise team work between programmers, artists, and everybody else involved in the development of that videogame [26]. For this reason, it must be detailed enough so that every team member knows exactly what and how it needs to be done, and that detail is the concreteness we need for implementation.

A typical mid-production GDD may include sections for concept, marketing, story, characters, menu, level design, gameplay, music, multiplayer, etc [25]. However, some of these are not applicable —such as level design or multiplayer—, and others are out of the scope of this dissertation —story, music or menu—. Thus, we will focus on the general concept, target and the intended gameplay experience that draws from it.

2.1 Basic Concept

Carnem-Levare is a fast-paced third-person Action game set in a obscure carnival-like festival of the same name. This carnival soon has a dark twist when the player finds out they hold a yearly boxing tournament between the best costumes of the event. No gloves, no round limit. As long as they may last.

The player will be drawn to it, finding themselves fighting in duel after duel. That is, a boss rush, where every confrontation is against a significant opponent whose abilities and moves the player will have to learn and overcome with their own reflexes and precision. Contrary to other

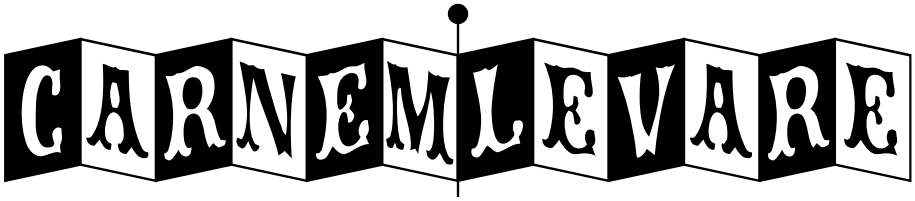


Figure 2.1 – Provisional title logo.

games, no advantages such as bonuses or protection can be expected from their clothing and armor; after all, they are only costumes.

Every punch must be dealt in the right time and place. Every dodge must be performed precisely in the right direction to avoid defeat and the horror that may lie within. Each opponent is tougher than the last, but the game will make no effort to stop the player from trying. It's up to them to decide whether to continue or give up.

2.2 Target

We can expect from Carnem-Levare a punishing, skill-based gameplay of the likes of Punch Out [27] or Sekiro [28], where the player main route of progress is improving at the game itself. This kind of demanding experience might seem like a niche within a niche, considering we were already targeting PC gamers —a niche market within the videogame industry—.

However, this niche of "hardcore games" —a rather ambiguous term that often refers to challenging but profound games gameplay-wise— has been on the rise recently. For example, with the emergence of genres such as *souls-like*, known for their deep but punishing combat systems that reward the player with a nearly euphoric satisfaction upon victory [29]. We can also take a look at 2022's best-selling games in United States, where games like *Elden Ring* —a *souls-like* game from the company that invented the *souls-like* genre— and *God of War: Ragnarok* stand at the top, among other more casual titles such as *Pokemon: Scarlet and Violet* or *FIFA 23* [30] [31].

It's a niche that is not only on the rise but also finds justification within this dissertation' intended demographic. If we consider the circumstances around PC gaming, we could assume that all PC gamers are, by definition, "hardcore" gamers.

Gaming in general is an expensive hobby; a recent survey by All Home Connections revealed that the average American gamer spends about 76 US dollars on videogames a month, or 912 dollars a year [32]. A great investment, considering it is also taking into account the lower financial end of the spectrum, with platforms such as mobile or Nintendo Switch that do not reach the 200$ price mark with their budget options [33] [34].

PC as a gaming platform lies at the opposite end of that spectrum, costing at least four to five times the price of the budget platforms previously mentioned [35] [36]. The GPU shortage that took place around 2021 didn't make it any easier for PC users to invest in their hobby [37].

And it's not only expensive, but also requires research and specialized knowledge about hardware (which components to buy, from brand to specifications) and software (which operating system, how to buy and install games, technical issues that might arise during gaming, etc).

All of these reasons pile up to make PC one of the least accessible and most demanding —in terms of both time and money— platforms, and yet people still play on PC. We can safely assume then that those people consider videogames their main hobby, and have been playing long enough to consider PC gaming a worth investment, which makes them skilled and eager to be challenged. The perfect target for Carnem-Levare.

While these conclusions may not apply to all PC gamers, it sheds light into the type of game Carnem-Levare should be. How it should be designed to appeal to this niche, following current

trends while also providing new ideas that makes it stand out in such a competitive market.

2.3 Gameplay

To stick to the basic concept and appeal to our intended target, Carnem-Levare's gameplay must be challenging but intuitive. For new players, it must provide an experience that feels familiar to other games of the genre, but that soon enough grows its own depth after enough practice and knowledge has been acquired.

To achieve this purpose, we will describe the four most fundamental aspects of the game, which are:

- **World.** Where the fight is going to to take place.
- **Actions.** How the player acts and moves around that space. The tools they have to fight their opponent.
- **Enemy.** The player's opponent, and thus their goal (to defeat them).

2.3.1 World

The world should be the simplest in terms of functionality. In future prototypes, each stage may display complex artistic environments that support the atmosphere and aesthetic of the duel. However, when it comes to character's interaction with said world, it is in the game's best interest to keep it as simple as possible.

Action games with deep combat systems already provide significant interaction for the player to understand and excel at. Adding another layer of complexity to that interaction through the stage does not only require more work by the development team, but also may prove to be a distraction to the player at best, and a hindrance at worst.

Features such as hazards that may harm a character when coming into contact with them (e.g. spikes, see Figure 2.2), destructible environments that dynamically change the space where the fight is taking place (see Figure 2.3) or even something as simple as moving obstacles soon become a challenge in both design and implementation to build something that is necessarily not as profound or fun as the combat system itself. It becomes counter-productive and a waste of resources.

This is not mere speculation. Players from competitive scenes have been voluntarily neglecting game content for the shake of the "pure" combat experience for many years now. One clear example of this is the meme from the Super Smash Bros. Melee community known as "No Items, Fox Only, Final Destination" [38].

Super Smash Bros. Melee is a side-scroller platformer fighting game where, instead of depleting the opponent's health points, the player's objective is to knock them out of the fighting stage. The usual health bar is instead replaced with a percentage that increases with each consecutive hit. The greater that percentage, the further the player gets knocked back. The game provides all sort of content in the form of random spawning items such as guns or bombs, dynamic stages that change as the fight progresses and about 26 distinct characters to play with for a fun, diverse experience [39] [40].

Figure 2.2 – Dark Messiah of Might and Magic's spikes.

Figure 2.3 – Tom Clancy's Rainbow Six Siege environmental destruction.

A game that was initially intended as a party game [41] soon grew a rather fierce competitive scene due to its deep combat system. This competitive scene produced rulesets that aimed to reduce random chance, abusive strategies and environmental exploitation so that "skill" became the sole focus of the match. From turning off random spawning items, to setting specific winning conditions (4 lives, 8 minutes) and only allowing the few stages that remain unchanged throughout the fight [42].

And, of course, not all characters are equal (see Figure 2.4). Add the unbalanced potential of characters from a party game to the already restrictive rulesets of competitive Super Smash Bros. Melee and we get to the landscape that the aforementioned meme aimed to mock. No items, Fox only, Final Destination (see Figure 2.5).

It was a joke that emerged as a way to parody the stale nature of smash competitive play, but it doesn't fail to illustrate how there is a trend in action players to avoid features that do not relate directly to the combat system itself, for the shake of excelling in said combat without distractions or unfair situations.

Another example of this is For Honor, a medieval 3D fighting game that, aside for its innovative and profound combat system, was famous for its environmental exploitation. Matches, regardless of gamemode, take place in random maps that may or may not have environmental hazards for the players to take advantage of, be it spikes, ledges, geysers, etc.

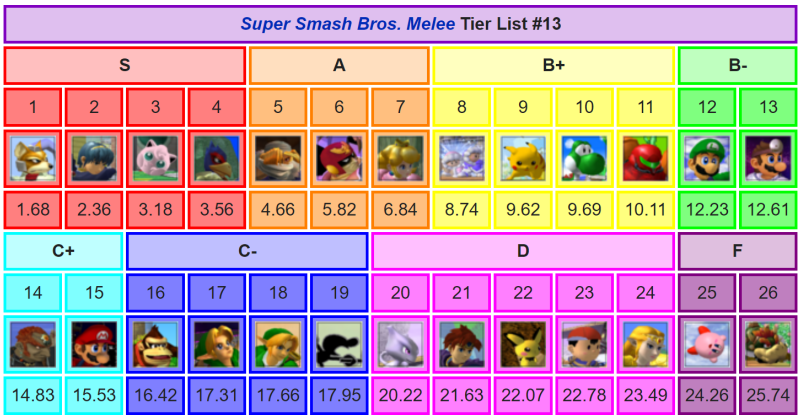


Figure 2.4 – Super Smash Bros Melee. community's tier latest tier list. Fox is the top ranked character. SmashWiki.

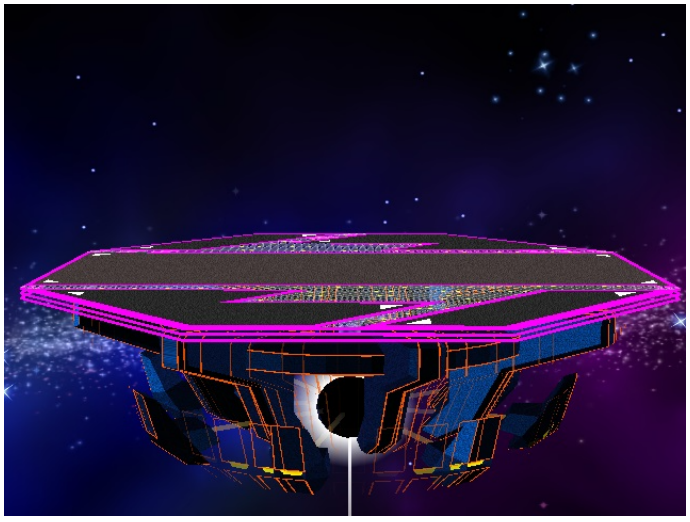


Figure 2.5 – Super Smash Bros. Melee's Final Destination stage.

The game is built around this, as grappling and manipulating the opponent is a fundamental feature known as "guard-break" (see Figure 2.6), and it provides an advantage significant enough not to be overlooked. This same guard-break can be used to push opponents —if they allow it—, often leading to what players refer to as "insta-kill" (self-explanatory).

Despite the clear integration with the combat system and the skill that may be required to take advantage of them, environmental hazards have been a rather controversial and widely discussed topic in the game's community [43] [44], as it often trivialises the fight altogether —that is, if the player or the opponent immediately dies to the environment instead of after a prolonged duel—.

Whether environmental hazards in For Honor are unfair is irrelevant to this dissertation, but then again it further confirms that there is a trend for action players to neglect and even reproach features external to the combat system itself.

It is for this reason that Carnem-Levare's stages will be as simple as possible; a decorative environment that contains the player and the enemy inside a enclosed space, without providing any kind of overlaying interaction so as not to interfere with the fight. This way we may also stick to the challenging but intuitive principle that was previously mentioned, establishing a focus for the rest of the design.

(a) Stance is broken. **(b)** Opponent is pushed.

Figure 2.6 – For Honor's Guard-Break mechanic.

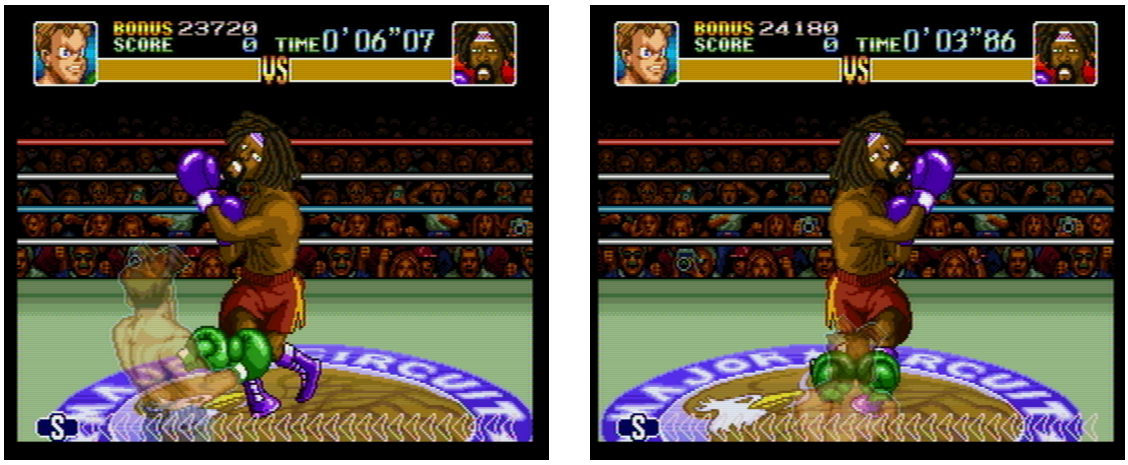
2.3.2 Actions

Any videogame cannot be a game without *actions*. Actions are the essential pieces of interaction, the tools the player has available to cause a change in the world, such as shooting, punching, jumping, playing a card, casting a spell, etc [45]. Anything that the player may perform in a videogame of any kind to progress or cause a change in the state of the game.

The player's list of possible actions is where we merge familiarity with innovation. The task is to recognise the common actions between games of the genre, what purpose do they serve and how can they be integrated, modified or improved upon.

2.3.2.1 Movement

The most basic action is **movement**, understood as the capacity to move freely around the world. An action game may include a very limited form of movement or none at all. An example of this would be Super Punch Out (see Figure 2.7), where the player is pinned to the center of the screen and is only able to dodge and duck out of harm's way [46].



(a) Dodge action. **(b)** Duck action.

Figure 2.7 – Super Punch Out's movement.

It is a simple yet effective design where movement is reduced into two defensive actions,

allowing for precise gameplay while being easy to understand and pick up. However, it leaves out a competence integral to most action games, which is *spacing*.

Spacing is a popular concept that emerges from fighting and action games communities and, as such, it does not have an exact definition. However, the meaning remains mostly the same between these gaming circles. We may describe *spacing* as the player's ability to gauge the distance between them and the opponent, taking into account each other's possible actions and how much distance do they cover with them [47] [48].

A skilled player is aware of their opponent's options —be it attacking or any kind of offensive action that may put the player in a disadvantage— and whether they are in range to be affected by them. They are also aware of the range of their own actions and the distance necessary for them to be effective.

This intuitive knowledge, once applied, becomes key to mastering the game. Every single interaction in an average action videogame involves some sort of space management, and being proficient at it allows the player not only to avoid the opponent's actions but also to counter them optimally.

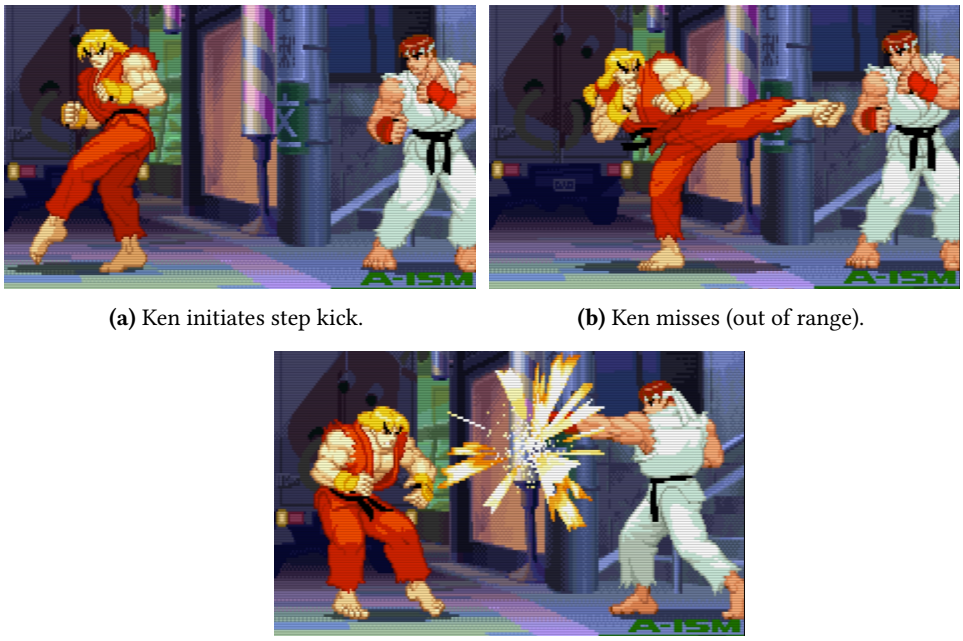
In this genre, all actions have a cost proportional to its usefulness, usually in the form of time spent performing said action. For example, Ken's step kick in Street Fighter Alpha 3 (see Figure 2.8) takes 24 frames to perform, deals damage for 2 frames, and takes 19 frames to recover [49]. If it misses, Ken must wait 19 frames to regain control, making him vulnerable.

Figure 2.8 – Ken's step kick animation key frames in Street Fighter Alpha 3.

A skilled Street Fighter player has the intuition to tell when they are in range for Ken's attacks, and if they expect the opponent to perform a step kick they can use that intuition to avoid it and counter with an attack of their own during those 19 frames of recovery (see Figure 2.9).

This competence grows in importance when both the player and the opponent are aware of space management. If the opponent is skilled enough not to attack the player when they are out of range, the player could lure them into attacking by stepping into their range and then stepping out at the last second for a counter. This is often referred to as *baiting* [7].

The reader may now see how a really basic action such as movement can become such an integral part of the game's depth. And while it may seem like this depth does only really apply to player versus player videogames, in reality it functions just as well against artificial intelligence. As long as the AI opponent has an understanding of distance, the player can take advantage of that understanding to avoid, bait and counter. Something as simple as making sure that the AI only attempts to attack when they are in range already allows this way of playing.



(a) Ken initiates step kick.

(b) Ken misses (out of range).

(c) Ryu counters.

Figure 2.9 – Spacing practical example in Street Fighter Alpha 3.

It is for this reason that limited movement such as Punch Out does not fit Carnem-Levare's design. The objective is to produce a profound combat system intended for *hardcore* players willing to master it, and removing one of the main sources of mechanical depth goes directly against it.

However, making the game overly complicated is not the solution either. We are already appealing to a niche of the videogame market, it is not in our interest to further reduce the target by making it less accessible. We must remain true to our original purpose of challenging but intuitive gameplay, and for something as fundamental as movement it is best to stick to the usual.

The player will be able to walk in any direction using the gamepad's left stick or keyboard's WASD (see Figure 2.10). A fast enough walk to be able to maintain distance, but not fast enough to run away from attacks.

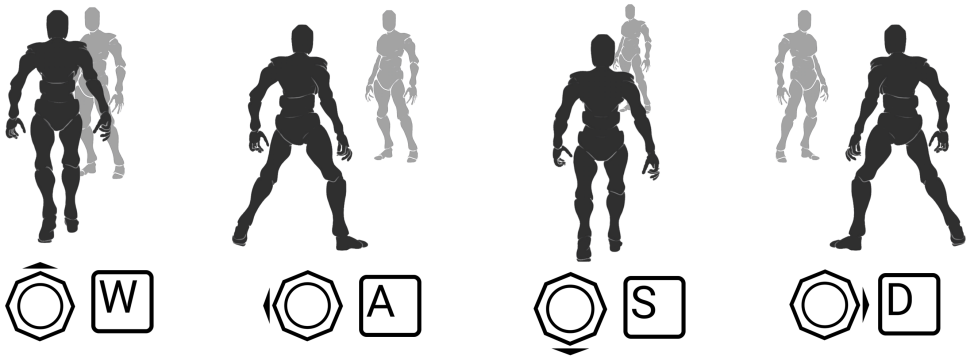


Figure 2.10 – Carnem-Levare's basic movement.

2.3.2.1.1 Targeting

Unrestrained movement in a 3D environment for an action videogame is rarely a good design decision. If the character is able to move and rotate in any direction without any sort of guidance, orienting them towards the enemy becomes difficult and counter-intuitive. It forces the player to align the character's position with the camera, which makes landing attacks awkward and unnecessarily challenging.

Fortunately, this is an old design problem which was solved as early as 1998, with the release of The Legend of Zelda: Ocarina of Time. Ocarina of Time is an action-adventure game developed and published by Nintendo, the first in the Zelda series to be set in a 3D environment. As such, they developed a system to facilitate fighting enemies one-on-one known as *Z-targeting*.

When pressing Z in a Nintendo 64 controller (hence the name), the player targets an opponent, orienting all of their attacks towards them while keeping both characters in view (see Figure 2.11). Movement is also oriented towards the enemy, making the player walk around the opponent when pressing left or right, approach when pressing forward, and flee when pressing backwards.

Figure 2.11 – Ocarina of Time's Z-targeting.

This system makes combat in a 3D space viable and engaging. The player no longer has to deal with controlling the camera at the same time as they fight, and can instead focus on the nuances of combat, such as spacing, timing, attacking and defense.

Since the release of Ocarina of Time, it has become an standard in the industry to include some sort of targeting system to all action videogames. In fact, some games in the genre make targeting forced when enemies are present. For example, Sifu [50] automatically targets nearby opponents when they approach the player. If there is a single enemy, the player's attacks are oriented towards them. If there are multiple, the player is oriented towards the nearest enemy [51].

From this background we may conclude that Carnem-Levare needs a targeting system. Since the game is centered around fights against single opponents, there is no point in making targeting

an optional tool to be enabled through the press of a button. Instead, from the moment the duel starts, both the player and the enemy will be automatically oriented towards each other.

2.3.2.2 Defense

While proper movement may already provide ways for the player to defend against any kind of offensive action, a game without a specific defensive action would prove to be lacking in terms of game feel and too punishing to be enjoyable.

All action games have a defensive action of sorts, allowing the in-game characters to protect themselves from incoming attacks. As it was previously mentioned, Super Punch Out has two defensive actions to dodge the opponent's punches [46]. They may dodge to either side or crouch to avoid an attack. In either case, the player moves out of the way for less than half a second, then comes back to the center of the screen. Since the player can be hit during the start and the end of the animation, they have to time it correctly for it to be effective. However, if performed successfully, it provides a window of opportunity that allows the player to strike back at the opponent.

This window of opportunity results from the same mechanics that take place during effective spacing in games with general movement. The opponent attacks but they miss (in this case, because the player used their defensive action successfully), leaving them vulnerable for the remaining frames of the attack animation.

Most defensive actions in the genre function similarly to Super Punch Out's dodges. A sudden action that protects the player for a short window of time, while also providing opportunity to strike back at the opponent if timed correctly. In essence, it encapsulates defense in one or few buttons, simplifying the technique required to do so.

This goes in contrast to concepts such as spacing, which is an unintended mechanic resulting from the precise execution of movement. This kind of skill require the player to have extensive knowledge about the game and general dexterity with the controls. Therefore, it becomes evident why relying exclusively in the unintended defensive nature of movement would be too punishing for new players.

As we stated previously, it would also be lacking in terms of game feel. Avoiding an attack by stepping away might not be as aesthetically pleasing as having your character visibly turning away from it, or blocking it at the last second. There must be room for both to allow for accessibility and aesthetics while also providing depth for high-level play.

But how do defensive actions and spacing complement each other? If they both provide a way to deal with the opponent's offense, it would seem logical that eventually one of them would render the other useless. Let's analyse a game that allow the player to take advantage of both: *Elden Ring*.

Elden Ring is an third-person action RPG where the player explores a hostile open world with a character of their own making. Their objective is to defeat progressively challenging opponents and grow stronger by upgrading said character, weapons and spells. Even though there is more to the game than fighting (such as talking to NPCs or crafting), the gameplay is clearly focused on combat.

Elden Ring has one main defensive action, which is *rolling*. Pressing B on gamepad or spacebar on keyboard allows the character to dodge roll in any direction (see figure Figure 2.12) [52].

This rolling action makes the character invulnerable to attacks for 13 frames (when playing at 30 fps, and with less than 70% equip load), then it makes them vulnerable as they must wait 8 frames to recover from said roll [53]. Those 13 frames of invulnerability (known as i-frames) makes rolling a powerful defensive tool, but it demands timing from the player so as not to be hit during those 8 frames of recovery.

Figure 2.12 – Elden Ring's dodge roll.

Rolling demands commitment and timing for it to be effective. Furthermore, it displaces the character significantly, which might put them in a unfavorable position to counter afterwards. Considering these downsides, one might assume that spacing, once mastered, might seem like the better choice to deal with attacks. Since it only involves slightly moving the character, it has no recovery frames to take into account and allows the player to counter the opponent instantly. Maybe even getting to hit them more times than they would have been able to otherwise after rolling.

All of these reasons pile up to make spacing the go-to defensive tool for skilled players. However, it is not always possible to outspace the opponent, and that is where having a specific defensive action truly matters.

Consider the encounter in Figure 2.13. Spacing is all about positioning, but outspacing a single attack does not guarantee being properly positioned for the next attack. The player successfully reads the enemy's range and counters with a single attack of their own. Had they reacted faster, they might have been able to attack twice.

However, as we can see in Figure 2.14, the enemy's next attack is a thrust, and the player does not have enough time to avoid it by walking. They have no choice but to roll away from it, taking advantage of the invulnerability frames rolling provides.

It's all about context, and the tools available for that context. Elden Ring without rolling would encourage the player to fight too defensively, making the game slow. Elden Ring without movement would eliminate a great chunk of the combat's depth and make exploration impossible. Not only the game needs both but they also complement each other as tools for the player to use and master.

Therefore, Carnem-Levare needs a defensive action. The obvious solution would be to add a single button action (such as Elden Ring's rolling) that allows the player to instantly protect themselves with a cost in recovery time. However, most action games have a dodge button integrated, and doing anything similar would result in a bland, derivative combat experience.

(a) Player steps away from the enemy's attack.

(b) Player counters.

Figure 2.13 – Player outspaces opponent in Elden Ring.

Instead, here is the opportunity to innovate. Considering how fundamental defense is in an action game, and how frequently the player must defend in a fast-paced game, innovating in the tools the player has to do so would do for a significantly different game experience.

Defensive actions in the genre are often rigid. An action encapsulated in a single button which, once pressed, the player has little to no influence in its execution. Games like Elden Ring might allow the player to choose the direction where the defensive action takes place (which direction they are rolling to), but once performed the player is subject to the consequences of said action and must wait for it to finish before regaining control.

In contrast, movement (when used for spacing) offers much more agency. The player is in complete control during the entire action, allowing them to correct missteps up to the last second before they are attacked.

An innovative defensive action should provide the instantaneous protection that typical defensive actions in the genre offer, along with the fluidity and agency of free movement. Considering the game is set in a boxing tournament, we can take a look at professional boxing for inspiration. More specifically, former heavy weight world champion Mike Tyson.

Range and distance is just as important in real life fighting as in videogames, and Mike Tyson did not have the advantage. Measuring 178 cm [54], he was short compared to the average heavyweight during his boxing years (187 cm [55]). As such, he had to be evasive to close the gap between him and the opponent, which led him to adopt Cus D'Amato's Peek-a-boo style.

Peek-a-boo refers to the defensive hand position where the hands are held in front of the face (like in the baby's game of the same name, see Figure 2.15), along with the footwork and technique that such a position demanded.

(a) Enemy starts thrust.

(b) Player rolls away.

Figure 2.14 – Player dodge rolls an opponent in Elden Ring.

One of these techniques is head movement, which is not exclusive to Peek-a-boo style but it is key in mastering it. Mike Tyson would avoid punches thrown at him by moving his head and upper torso side to side, bending at the waist (see Figure 2.16). At the same time, he would walk forward, dodging nonstop until he was in range to throw a punch himself [56] [57].

We can draw a clear parallel between Mike Tyson and the player in an action game, where they each must find ways to avoid the opponent's offense when in range disadvantage. We can use this parallel as a source of inspiration and design a defensive action that imitates Mike Tyson's Peek-a-boo style and head movement.

This results in the ability to *block*. When the player presses A in gamepad or spacebar in keyboard, the character raises their arms in front of their face. While this already protects them from hits (at the cost of stamina and recovery time), the interesting feature about this action is that it also allows the player to duck and bend side-to-side. Similar to Super Punch Out but instead of rigid, timed actions, we allow the player to precisely dodge in any direction (vertical movement, see Figure 2.17). If they do it correctly, they avoid the stamina cost and gain a window of opportunity to strike back. This way, we effectively give them a tool to avoid attacks even when inside the opponent's range.

Since the player is able to dodge indefinitely as long as they keep blocking, there are several design elements we need to keep in mind so as not to trivialize the fight with an unbeatable defensive action.

To begin with, it cannot have invulnerability frames. It would make no sense to have an

Figure 2.15 – Mike Tyson in Peek-a-boo stance.

Figure 2.16 – Mike Tyson dodges punch.

action that makes the player invincible for as long as they press the button, nor it would make sense to allow them that much range of head movement if they are not going to be affected anyway. Instead, the player must precisely move away from the attack, ducking or bending in the right direction to dodge it.

There must be some sort of penalty for missing the dodge. As we stated previously, if the player is hit while blocking, they will still be protected from the impact. This way, the game is not as punishing when the player misses the dodge and encourages players to fight close to the opponent by weaving through attacks as Mike Tyson would. However, there must be a cost and a risk to being hit when blocking, so that there is a benefit to dodging precisely. This cost is in the form of stamina.

Stamina is a resource that is spent when receiving a hit, regardless it was blocked or not, similar to Sekiro's posture [58]. Sure, blocking stops incoming damage, but it still drains stamina. When stamina runs out, the player is stunned and vulnerable for a long period of time. Even if they are not stunned, every time they block an attack they must wait a brief recovery time to regain control, missing the window of opportunity.

Stamina regenerates over time, allowing the player to mend previous mistakes if they dodge properly afterwards. However, consecutive hits will stun the player, as stamina does not regenerate fast enough to withstand several attacks in a row.

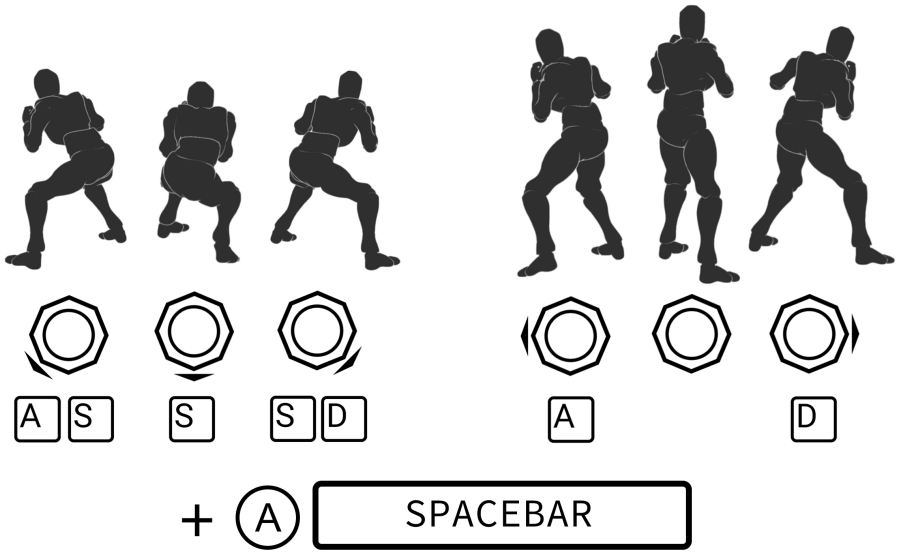


Figure 2.17 – Carnem-Levare's blocking movement. Press any direction (stick or WASD keys) while holding the block button (A or spacebar) to dodge.

2.3.2.3 Attacks

Few action games are complete without attacks. Even though there are some games that focus exclusively on defense (e.g. Way of the Passive Fist **[ref]**, Sekiro **[ref]**), we will overlook those examples for the shake of conciseness, as they are far from the boxing experience Carnem-Levare aims to offer.

Defeating the opponent is the main objective of all combat action games, and attacking is the main tool to achieve it, as it is the action that depletes the opponent's health. While the player may also block or dodge, these are just secondary elements intended to keep them alive so that they can continue attacking. Therefore, it is the ability to attack that makes a combat system, and that makes it essential for this game.

2.3.2.3.1 Moveset

To begin with, we will define the number of attacks the player has available. Even though the game is perfectly playable with keyboard, the intended controller is an standard gamepad. Since both thumbs are already occupied with other actions (left thumb over left stick for movement, right thumb over A button for blocking), the remaining fingers we have are the index and middle fingers, which are most comfortably placed on the shoulder buttons (see Figure 2.18). That leave us with four possible buttons: left and right bumpers, left and right triggers.

Four attacks is not the profound moveset one might expect from a videogame focused on combat. For example, Street Fighter has six distinct attack buttons which allow for dozens of different moves when taking into account stick-button combinations, character position and state [59]. Each of those moves become options for the player to use depending on the situation (e.g. A quick, light punch to catch the opponent off guard, then a slow, heavy kick while the opponent is recovering). Such a diverse moveset may stray from our "challenging but intuitive" goal, but it goes to show that four attacks might be insufficient for a combat action videogame.

There are two simple solutions we can consider to deal with the four-button limit: directional

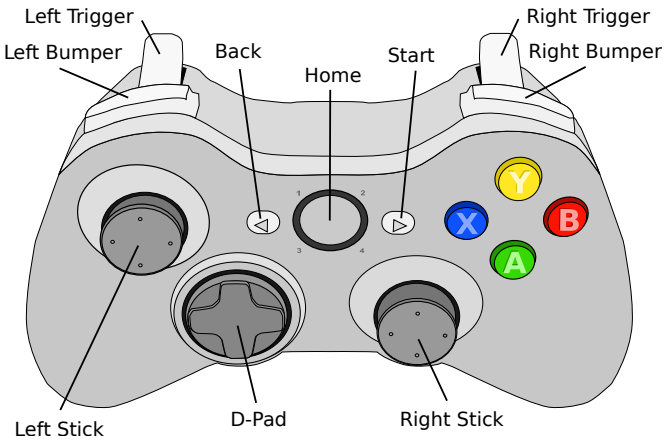


Figure 2.18 – Xbox 360 controller.

input and interaction types.

Similar to Street Fighter's stick-button combinations, we can take advantage of the previously described blocking movement to allow attacks to be performed from different angles. For example, a single jab might be executed while standing or while crouching, effectively becoming two different attacks in one (see Figure 2.19). The direction of the stick influences the specific attack the player performs, without making use of any new button.

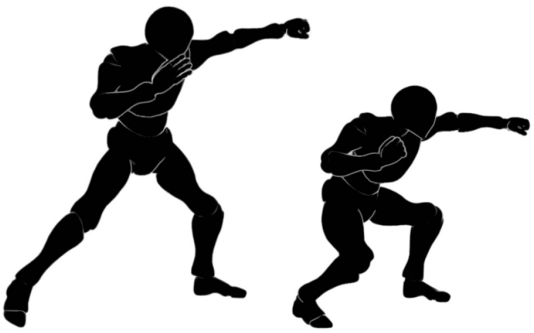


Figure 2.19 – Carnem-Levare's jab punch, standing and crouching.

We can also take advantage of the different types of interaction that a single button in a controller allows: tapping, multitapping and holding [60]. Tapping is to press and release, multitapping is to tap several times in a row, and holding is simply holding down the button. These interactions can be customised by defining variables such as how quickly the player must release the button in a tap interaction, or how long they must hold a button for it to be recognised.

Carnem-Levare must have a single tap interaction for each attack button, since it is the quickest type of button press. That results in four attacks, but we can raise the number by adding secondary, different interactions on top. Theoretically, we could have an infinite number of attacks assigned to a single button. In reality, we are not going to assign something like a 12-presses multitap to a single button. By adding a simple, hold interaction on top of the default tapping for attacks, we raise the number of available attacks from four to eight. Plenty for this game's purpose.

Now that we have established the number of available attacks, we must define what each of those attacks will perform.

The problem with most videogames is that, once the gameplay it offers is mastered, the

experience becomes stale. In the case of action games, one way of delaying that staleness is making a massive, convoluted moveset. Nonetheless, that makes the game less accessible and counter-intuitive.

Another way is to strive for diversity. In fighting games like Street Fighter, this diversity comes in the form of characters (see Figure 2.20). While two characters may have the same number of attacks, executed by the same buttons, the attacks themselves are completely different. They provide distinct play styles to master and therefore a diverse gameplay experience. Nonetheless, this is not the only way of providing diversity and, in fact, there are a few problems with it.

Figure 2.20 – Street Fighter 6's character roster.

Each new character integrated into the game requires work in all fields. The character must be drawn or modeled, animated, designed for a balanced experience, coded, etc. Then, usually, one or two character are not enough. If the game is centered around offering different characters for a diverse experience, it will need a numerous enough character roster to provide it.

Instead, we could draw inspiration from a RPG-fighting game known as Absolver. In this game, instead of choosing a predetermined character, the player builds a character of their own as they would in an standard RPG (armor, stats, weapons), but they also customise their moveset with attacks their character learns throughout the game [61] [62]. This way, instead of providing diversity with a few, pre-established characters, Absolver offers the endless combinations of moves and gear that can be applied to a single character.

Absolver character's moveset is known as the *combat deck*. It consists of four sequences of three moves, plus four alternative moves (see Figure 2.21). The player has two attack actions on two separate buttons: sequence attack and alternative attack. Pressing the former performs the sequences of attacks established previously by the player, while the latter breaks the sequence with an alternative move [63].

Since Carnem-Levare already has eight attack actions, we can simplify this process by allowing the player to assign a move to each of those actions. It results in half the amount of attacks available (Absolver's combat deck totals to 16 moves) but it gives the player more agency, as they

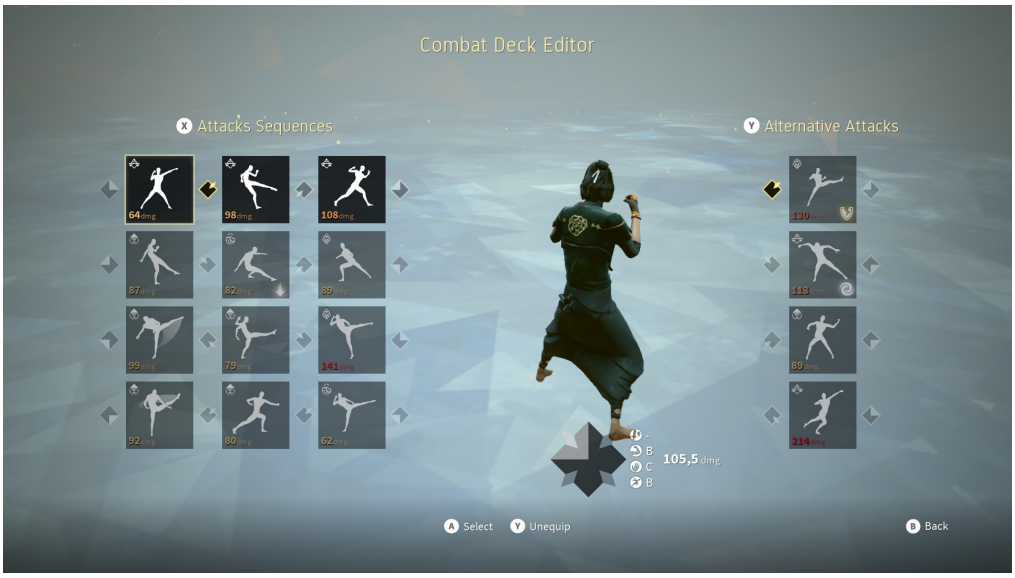


Figure 2.21 – Absolver's combat deck.

can now choose which attack to perform next *in the heat of battle* instead of having to follow predetermined sequences.

In conclusion, the player in Carnem-Levare will have a customizable moveset of eight attacks, distributed between four tapping and four holding interactions of a standard gamepad's shoulder buttons.

2.3.2.3.2 Frame Data & other specifications

A customizable moveset requires attacks to be all the more balanced. Since any combination of moves is possible, we must ensure that each attack has a potential usage without providing a game-breaking strategy.

Designing an attack is no trivial task. There are several factors to consider and each of those factors demand extensive study and testing. While we will not go into detail about concrete examples and design decisions (balancing and testing is out of the scope of this dissertation, as it could be a dissertation on its own), we will describe those factors for a better understanding and to allow for a proper implementation in the next chapters.

The most obvious factor is damage output. How much damage should each attack deal when hitting the opponent? How much damage *to stamina* should it deal when that attack is blocked? These questions not only affect balancing but also the pace of the game. High damage numbers generally result in a faster, frenetic combat, where every successful attack and dodge heavily influence the outcome. Lower attack damages result in a slow test of endurance where the player must remain patient and make sure that, overall, they maintain the upperhand throughout the entire fight.

Balancing damage must take into account the weak and strong points of an attack, and the purpose they are intended to serve in a fight. Slow attacks need higher damage to compensate for their speed, as they are not going to be performed as frequently throughout the fight. Attacks intended to break the opponent's guard when they are blocking might deal a lot of damage to stamina, but not so much to their health. Fast attacks are usually meant to catch the opponent

off guard and start combos, therefore they must deal low damage so that spamming isn't a viable option.

Then there is the factor of speed, which is actually just one part of a bigger picture known as *Frame Data*. Frame data describes three distinct phases that take place during the execution of a move in a fighting or action game, measured in frames. These phases are: Start-up, Active and Recovery [64].

- Start-up defines the number of frames it takes for an attack to deal damage. This is the phase we would ordinarily refer to as "speed" [1].
- Active is the time during which an attack deals damage. Usually the shortest of the three phases, around two or three frames in an standard fighting game.
- Recovery states the number of frames the player must wait to regain control after performing an attack. Usually the longest phase, in which a character is unresponsive and vulnerable.

These three phases go hand-in-hand with two other frame specifications known as *hitstun* and *blockstun* [65] [64]. When a character receives a hit, they must also wait a certain amount of frames before they regain control. If that amount of frames is greater than the attack's recovery, that attack is said to have *advantage on hit*. For example, if a kick has a recovery of 10 frames and a hitstun of 15 frames, then the difference between the two results in an advantage on hit of 5 frames. If the next attack performed has an startup of 5 frames or less, it is guaranteed to hit (this is known as a combo).

Blockstun functions in exactly the same manner, but when an attack is blocked instead of received. If the aforementioned kick has a blockstun of 5 frames, the difference between the two results in an *advantage on block* of -5. That is, the attacker is vulnerable for 5 frames after the opponent has successfully blocked, being open for a counter. If that advantage is positive, the attacker can continue chaining attacks that also have positive advantage until the opponent lowers their guard (or runs out of stamina, if such a system is implemented).

With this information we can begin balancing moves around their purpose. A jab may be intended to catch the opponent off guard and begin combos. Therefore it must have low damage so that it is not useful on its own, low startup to hurt the opponent as quickly as possible, and low recovery to be able to initiate the next attack as soon as possible. It needs to have a decent hitstun to allow for a slower attack to hit afterwards, but a short blockstun so that if the opponent expects it, they can simply block it and suffer no disadvantage.

A hook may be a mid combo move, the attack the player deals once the opponent is in disadvantage. To begin with, it needs high damage, otherwise there would be no point in using it over faster moves. Then it needs a long startup and recovery to justify its damage, and so that attempting it outside its intended purpose becomes risky. Hitstun must also be long, another reward for the player upon landing it beside the damage. Lasty, blockstun may either be long or short. If long, this hook becomes a safer option beyond combos, as the player is not left vulnerable when the opponent blocks. If it is short, it allows the opponent to counter after blocking, making it all the riskier.

[1] Start-up is also the phase during which targeting is enabled. In order to make attacks avoidable by both the player and the enemy, characters must not be tracking their opponent's position during the entire length of the attack. Instead, attacks stop targeting after start-up, leaving active and recovery as a time window for the defender to evade the attack.

But when do these combos end? If the player makes sure to attack with moves that combo into each other, abusing that advantage on hit or block, when do they stop? There are several solutions to this.

Standard fighting games usually have a pushback value for every attack, where the attacking character is *pushed back* slightly after landing an attack (see Figure 2.22). This is coupled with a *combo decay* mechanic, which increases pushback after every consecutive hit. This way, every attack becomes harder to combo, as the attacker is pushed increasingly further from the opponent, eventually out of range.

Figure 2.22 – Street Fighter 2's pushback.

Singleplayer action games conceive hitstun as an occasional reward. Instead of stunning the opponent every time the player strikes them, the opponent is only ever stunned after receiving enough consecutive hits (also known as stagger), or after performing certain actions intended to stun them [66] [67]. This way, the player cannot spam attacks and must instead consider very carefully every course action before being able to combo the opponent, otherwise the enemy might strike back while they are attacking or recovering. Since enemies in this kind of games are usually AI-controlled NPCs with predictable patterns of slow moves designed to be dodged, being able to stun them mid-animation as one would another player in a fighting game would render them useless.

2.3.3 Enemy

Considering Carnem-Levare is meant to be singleplayer, the opponent (or enemy) is something else to be designed with deliberation to fulfill the intended experience.

In a combat-focused game, the enemy the player fights against impacts the gameplay even more so than how they attack or defend. How aggressive or passive the enemy is, how often do they leave themselves open for an attack, etc. These are nuances that directly influence how the player behaves, as they have to adapt to said opponent in order to achieve victory.

One example of this is the development of Doom 2016. id Software (American videogame company of first-person shooters Doom and Wolfenstein series) initially opted for aggressive enemies that would chase down the player. However, during playtests they found out that this made players act defensively, backing up and fleeing, which was the opposite to the experience of being the "slayer of demons" [68]. Examples like this evinces how enemy design indirectly establishes the optimal strategies and play styles to triumph in a videogame, which sets the mood and game feel for the overall experience.

2.3.3.1 Difference between videogame and software AI

Proper enemy design is necessary for enjoyable gameplay, and designing an enemy results in, of course, designing its artificial intelligence. While this may seem closer to software design, truth is AI for videogames is quite different from AI intended for standard software.

The basic goal of any standard AI is to solve a specific problem as efficiently as possible. In this case, the problem would be to defeat the player, and one may easily guess how that would be detrimental to the experience. Not only it would trivial to give the enemy perfect reaction time and precision, it just would not do for a "fun" or "fair" videogame. Another thing entirely would be to have an AI simulate a player, producing specific control outputs given the same input a human player would have (video, sound, etc). However, the main purpose of simulating a player is to facilitate research of game-player interaction and player behaviour, rather than serving as videogame content [69].

Instead, the objective is to create a consistent, flawed AI that allows the player to plan around its predictability and exploit its weaknesses [70]. A properly designed videogame should encourage the player to learn the behaviour of their opponents and take advantage of that knowledge to formulate effective, satisfying plans. Therefore, the AI does not need to perform optimally in its given task and it is often discouraged to do so.

This "predictability" does not necessarily mean "easy". Every enemy in a videogame will display patterns of behaviour. Each enemy with a different pattern is yet another enemy to learn and defeat. The more complex those patterns, the harder it is to learn and exploit the opponent. This complexity may come in many forms, such as number of moves the enemy may perform, speed of those moves, how much the game punishes the player for every mistake, etc.

Also, they do not need to be intelligent, but rather *appear* intelligent. For example, Bungie (American videogame company creator of the first-person shooter Halo series), during the development of the first Halo, set up a playtest with two versions of the game that had the exact same enemy AI. Nonetheless, one version had enemies that dealt low damage and died in few hits, while the other presented enemies that could quickly defeat the player and resisted many of their shots. The number of people who agreed that the AI was "very intelligent" jumped from an 8% in the group that played the weak version, to 43% in the other group [71].

All of these reasons make videogame AI a significantly different problem from the ones academic AI intends to solve, and as such it is often debated whether videogame AI is actually AI. We will not contribute to this discussion as it is not the objective of this dissertation, but it is an important distinction to make from the start.

2.3.3.2 Type of enemy

To design the enemy's AI we first must determine which kind of enemy Carnem-Levare's opponents will be. To do so, we have to analyse the context in which the player will encounter that opponent and the specific challenge it will provide. The AI cannot behave the same if they are fighting alone or in a group. Same thing if they are fighting beside a cliff instead of an enclosed space. They must be designed to adapt to their environment and situation in order to provide a proper game experience.

To begin with, we could follow a set of specific criteria to categorize said opponents. An usual

criteria for enemies in melee-based combat is *gameplay purpose*. That is, which purpose do they serve in the game, in regards to how they behave and how the player should respond to them. This splits enemies into four possible roles: emphasizers, enforcers, smashers and challengers [72].

- *Emphasizers* are enemies that *emphasize* a specific combat mechanic. They are designed to be particularly weak to certain types of attacks or abilities in order to encourage the player to perform said abilities, but they are not necessary to defeat them.
- *Enforcers*, on the other hand, are enemies that force the player to perform a specific combat mechanic. They are designed to be specially resistant or invulnerable to all means except for a specific attack or ability, requiring the player to perform said ability in order to defeat them. Along with emphasizers, they are often presented after a new mechanic is introduced or when said mechanic is introduced in a new context.
- *Smashers* are enemies which present little to no challenge and are easily defeated. They are usually the enemies first introduced to the player as a way to safely learn the basics of the game, though they may reappear in groups accompanied by tougher enemies or as a form of rest from previous challenges.
- *Challengers* are difficult enemies designed to test the skills of the player. They are usually introduced by the end of a specific section of the game as a climactic way to test the knowledge and technique the player has acquired so far throughout the game.

This criteria is interesting to design enemies given their purpose at a given point in the game, but does not provide any behaviour specifications by itself. Since Carnem-Levare revolves around fighting enemies one-on-one in enclosed, flat spaces, distinct enemy behaviour is necessary for a non-monotonous gameplay experience. And for that we need specific criteria that allows us to differentiate between possible, deeper enemy behaviours.

Since our goal is to create complex enemy AI to complement the profound combat system and compensate for the lack of content beyond fights, an interesting criteria to follow could be player playstyles in multiplayer fighting games. While player behaviour is usually not a good model from which to design predictable, exploitable AI, we can use it as a source of inspiration and innovate in the process. A game focused on individual encounters as Carnem-Levare requires diverse, smarter than usual enemies, and player behaviour —necessarily deeper and more nuanced than standard videogame AI— is a valuable reference to achieve it.

Playstyles in fighting games varies greatly from player to player, but they can be categorized into three general archetypes: rushdown, keep-away and patient [73].

- *Rushdown* is the aggressive archetype, which aims to overwhelm the opponent with fast attacks and mixups. Players that adopt this playstyle seek to be at close range to ensure all their moves land, and their defense consist in not letting the opponent attack.
- *Keep-away* is the defensive archetype, the opposite to rushdown. Their goal is to stay out of the opponent's range while staying in range for their own attacks, taking advantage of good spacing and long-ranged moves. They only attack when it is safe to do so, moving away or blocking otherwise.
- *Patient* is the counter-aggressive archetype. They wait for the opponent to make mistakes and counter-attack with strong moves that would not land otherwise. This may be

achieved by blocking attacks that have disadvantage on block, baiting the opponent into performing an attack out of range, etc.

These three archetypes, even though they are a major generalization of player playstyles, already provide specific distinct behaviour from which to design AI. Together with the previous criteria, we can design enemies that provide a diverse set of challenges by assigning a specific archetype to the purpose the enemy intends to fulfill.

For example, we could design a enforcer enemy with a rushdown archetype, which forces the player to dodge and counter-attack to be able to defeat them. We could also design a patient challenger, that tests the player's skills and punishes the mistakes they should not be committing at that point in the game.

Smashers enemies can be of any type, as they are easily defeatable enemies. Though rush-down might suit them better, so that they walk into the player's range and facilitate their victory, while also providing a safe opportunity to learn the basics of defense and counter-attacking.

Keep-away can be interesting emphasizers. While the easiest way to defeat them would be to close the distance and pin them against the corner, they can also be defeated by being patient and counter-attacking their ranged moves.

2.3.3.3 AI design

Now that we have criteria to establish the types of enemies the player will encounter throughout the game, we have to determine how those types will be defined in the artificial intelligence's design.

Most of the types established by gameplay purpose can be reduced to parameter tweaking that need not affect the AI's design. For example, a smasher enemy can be constructed by having an enemy with low health deal low damage, regardless of the specific AI behaviour they may display. A challenger may have high health and damage values compared to the average enemy, along with a extensive moveset which complicates learning how to defeat them.

It is the fighting playstyles that require actual AI design. They define explicit behaviour where the enemy reacts differently to several kinds of stimulus and have distinct ideal status quo they attempt to maintain throughout the fight (*keep-away* tries to stay out of the opponent's range, *rushdown* tries to close the distance and attack nonstop, etc).

While these playstyles may seem convoluted to design consistent AI behaviours, they can be better understood and simplified if we take into account the concept of *turns* in combat-action videogames.

Even though this term is usually applied to multiplayer fighting games, it can be extended to most games that involve any kind of real-time combat in their gameplay. Turns are a concept that naturally emerge from the dynamics of attack-defense and decision making. Decision making is a basic skill that grows all the more important in action videogames because of the real-time factor. Optimal decisions must be made in fractions of a second, otherwise the player is overwhelmed and defeated. And those decisions can be reduced to attack and defend.

When the player is attacked, they are forced to defend. If they block an attack that has advantage on block, that means they have to continue defending. This would be considered the *opponent's turn*. Now, if the opponent were to deal an attack that has disadvantage on block,

the player would be able to counter-attack and *steal* the turn, beginning the player's turn (see Figure 2.23). If the opponent were to lose the turn because of any sort of combo decay mechanic (for example, being pushed back by their own attack) or give up the turn on their own (by not attacking), the game would return to an state where no one has the advantage, known as *neutral* [74] [75].

Figure 2.23 – Ken steals turn from Ryu in Street Fighter 2.

Action videogames where hitstun is delayed as an occasional reward may seem to stray from this concept, as frame advantage is not as relevant when the enemy is not interrupted after every hit. Nonetheless, it can be analyzed just as well with this perspective, the only difference being that the enemy orchestrates the turns.

For example, the enemy might begin a sequence of attacks where the player has no other option but to defend. Even though the attacks are slow, the enemy continues attacking and the player might expose themselves by trying to counter-attack during this sequence. This would be the *enemy's turn.* When the enemy finishes this sequence, they remain passive for a brief time window where the player can attack them safely. That would be the player's turn.

In general, all combat can be described as a back and forth between player and opponent's turns, and how they behave during each of those three states (player's turn, opponent's turn and neutral) defines their playstyle.

Therefore, we can design each of the three fighting archetypes AI by defining their behaviour during each of the three possible states, and which of those states they seek to bring about and maintain.

- *Rushdown.*
  - **Ideal state:** own turn.
  - **Own turn behaviour:** continue attacking and corner the opponent, mixing up moves that might make the opponent commit a mistake and extend their own turn.
  - **Neutral behaviour:** force their own turn. Close the distance and attack the opponent before they attack first.
  - **Opponent's turn behaviour:** defend until stealing the turn is possible. Find an opening and counter-attack.
- *Keep-away*
  - **Ideal state:** neutral.
  - **Own turn behaviour:** continue attacking as long as they can before the opponent is out of reach.

  - **Neutral behaviour:** stay out of the opponent's range, attack when the opponent cannot hit back either because they are in recovery or out of range.
  - **Opponent's turn behaviour:** defend and escape until the game returns to neutral state.

- *Counter-aggressive*
  - **Ideal state:** opponent's turn.
  - **Own turn behaviour:** continue attacking as long as they have the advantage. Maybe even give up the turn if unsure of the remaining frame advantage.
  - **Neutral behaviour:** step into the opponent's range in order to bait them into attacking.
  - **Opponent's turn behaviour:** defend and counter when the opponent is at a disadvantage, either because they missed an attack or dealt a move that had disadvantage on block.

The reader may now see how the concept of turns sheds light on the three fighting archetypes, and how well they fit into each of the three possible game states. These behaviours can also be parameterized for even more customizable enemies, such as establishing the attack frequency of a *rushdown* enemy or the spacing precision (how well they judge the distance) of a *keep-away* type. Paired with general character parameters such as health, stamina, damage and moveset, this allows us to create a varied range of enemies that provide distinctive experiences with each fight.

Chapter 3

Software Design

Given the game design document, we may now begin transforming gameplay specifications into an actual software system. However, before the implementation into code takes place, it is important to design the architecture that will guide said implementation.

Videogames are complex software applications, where each gameplay specification actually presents ambiguous problems that must be broken down into concrete pieces that can be made into code. A proper design of the architecture and the underlying systems is key for the implementation to be robust against bugs and allow for new features to be added easily in the future. Videogames are also pieces of software that tend to demand high resources from the system they are running on, which makes optimization and performance all the more important in contrast to other types of software.

In the case of Carnem-Levare, it must also be constructed to allow for further game design to be made directly into the game without coding. Specific gameplay values such as character health, stamina, move framedata, etc, must be parameterized and accessible to game designers to read and adjust without affecting the software architecture.

In this chapter, we will describe the programming patterns that will compose the software design and how they will be applied to the final interaction system.

3.1 SOLID

SOLID is an acronym that describes five design principles for object-oriented software applications. The purpose of these principles is to make the implementation more understandable, maintainable and flexible [76]. Each letter of the acronym refers to a single principle, being:

- ***S**ingle-responsibility principle.* There should never be more than one reason for a class to change. Therefore, every class should only have one responsibility [77].
- ***O**pen-closed principle.* Software entities (such as classes, modules, functions, etc) should be open for extension, but closed for modification. An entity should only allow its behaviour to be extended without modifying its previous code [78].
- ***L**iskov substitution principle.* Functions that use references to base classes must be able to use objects from derived classes without knowing it. An object should be able to be replaced by a sub-object without breaking the program (otherwise, it should not be a sub-object) [79].
- ***I**nterface segregation principle.* No code should depend on methods it does not use. ISP splits large interfaces into smaller more specific ones so that classes will only know about the methods they need [80].
- ***D**ependency inversion principle.* Modules should depend on abstractions (such as interfaces), not concretions. High-level modules should not reference low-level modules directly. Instead, they should both depend on abstractions [81].

These principles will be applied throughout the entire design along with the following patterns to ensure a scalable and readable architecture.

3.2 Model-View-Controller

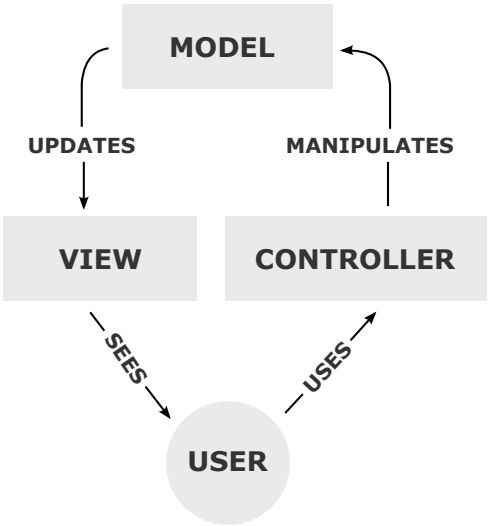


Figure 3.1 – Model-View-Controller diagram of interactions.

MVC is a software design pattern, commonly used for web applications and other user interfaces, that divides the program logic into three independent but interconnected components (see Figure 3.1) [82].

- *Model.* Manages the application's data structure, logic and rules. Updates the view with said data. Has no direct interaction with the user.
- *View.* The graphical representation of the model, whether a chart, a diagram or 3D graphics. The component the user is able to see.
- *Controller.* Accepts input from the user and converts it to commands to manipulate the model (also the view in some cases).

MVC is a common pattern in videogames as they can be similarly broken down in the same three pieces [83]. As such, it's an standard in the industry to implement the general architecture of the game. View is the interface and graphics they see in the game, along with the audio that complements said graphics. The controller is the system that accepts input from a controller or keyboard/mouse and converts it to commands, while the model consist of the inner workings of the game (the rules, the physics, the data, etc) that is being represented by the view [84].

Likewise, MVC can be applied more specifically to the design of a player-NPC interaction system. This interaction is no more than a model of information and rules that must be represented visually in the game, and change according to the actions of the player and the AI.

To simplify this process, we defined player and NPC not as separate entities but as *characters*, the difference being that one is controlled by input while the other is controlled by AI. Characters are composed of all the data necessary for them to function (health, stamina, movement speed,

etc) along with the objects that make interaction between each other possible (hitbox, hurtbox, state machine), which makes a model that can be represented through animation, audio and visual effects. This way, designing the character architecture results in designing the interaction system as a whole.

The character software design can be represented by the following diagram (see Figure 3.6). The classes *CharacterStats*, *CharacterStateMachine* and *CharacterMovement*, along with the classes they depend on, form the model. The data the character holds along with the rules and logic that it follows.

The view consists of the classes *CharacterAnimator*, *CharacterAudio* and *CharacterVisualEffects*, which manages the representation of the model. Lastly, *InputController* and *AIController*, both concretions of the interface *Controller*, transforms the input made by players or AI to cause changes in the model, forming the controller component.

Following the intended interactions of the MVC pattern, the controller transforms input given by an user (either a player through a gamepad/keyboard or an AI) and transforms it into commands to manipulate the model, which may lead to the state machine switching to another state (in the case of blocking or performing a move), the character changing position by moving, etc. This change is then adequately reflected by the view through a 3d object positioned in a 3d environment, with animation, sound and visual effects. However, while this process does follow MVC's expected interactions, there is an exception which must be taken into account. And this exception takes place during the interaction between two characters.

The combat interaction we are engineering is in the form of changes to the model. When a character gets hit, their health is lowered, their state machine switches to a hurt state during a given hitstun time, etc. Nonetheless, this interaction is not the direct result of a controller's transformed input. Instead, it is the result of a character's model (a character performs an attack, switches to move state, enables hitbox) connecting with the opposing character's model (their hurtbox). It is a model-to-model interaction that only takes places between two separate MVC's systems, and it would not be useful for it to work otherwise (e.g. the player being able to substract ten points of health from the opponent instantaneously through a button press).

A character's model, when given a command by the controller, is manipulated with the objective of causing a change in the environment. In this case, the environment is another character, which necessarily results in this model-to-model interaction (see Figure 3.2).

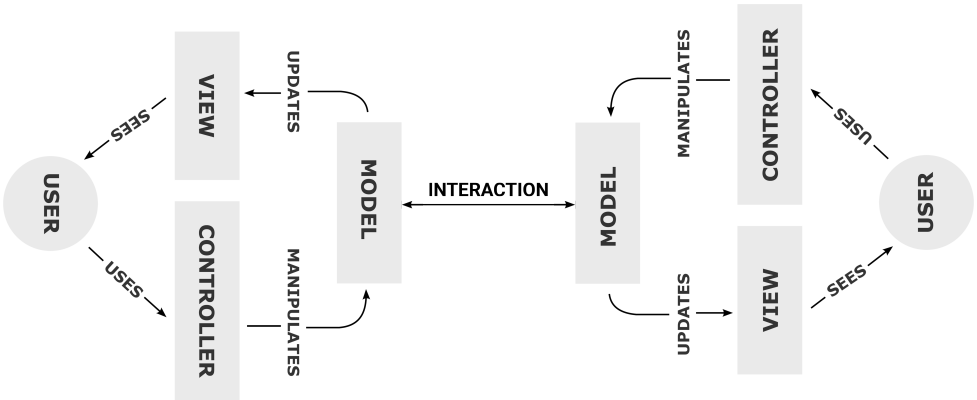


Figure 3.2 – Model to model interaction.

This concludes MVC pattern's application to this design. However, each of the classes that

compose the three components implement other design patterns on their own, which will be explained in the next sections.

3.3 Facade & Dependency Injector

To begin with, we have to describe a class which is not part of the MVC architecture, but instead acts as a mediator between the three components. That is, the Character class, which utilizes two software patterns in order to achieve this purpose.

Firstly, it serves as a *facade*, a software design pattern where an object acts as a front-facing interface masking a more complex subsystem beneath [85], providing access to said subsystem through a single module (see Figure 3.3).

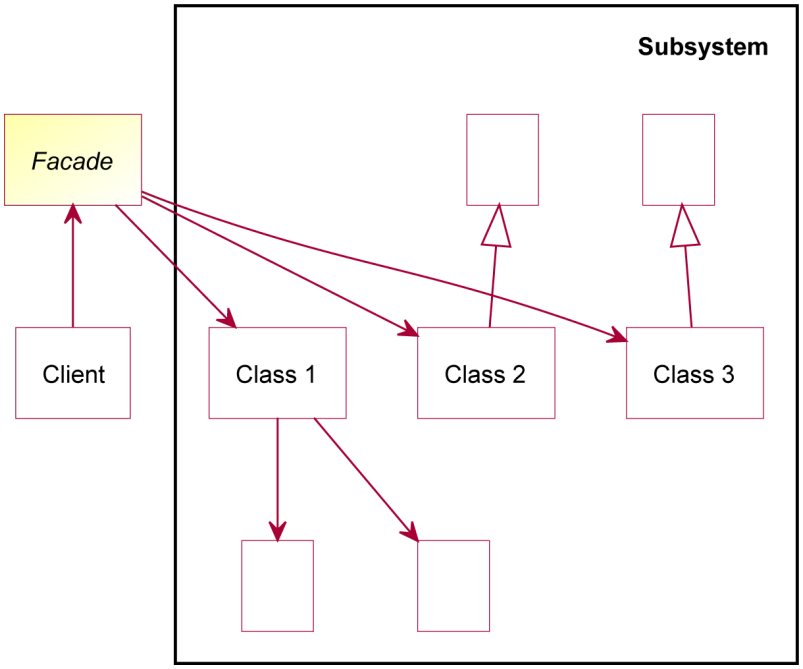


Figure 3.3 – Sample facade class diagram.

In the case of this design, Character acts as a facade to the game engine, providing access to the MVC architecture that compose the character behind a single, unified class (see Figure 3.4). The specific way it provides access will depend on the game engine used in the implementation, but it should allow game designers to balance and modify characters without coding.

Character also acts as an injector, making use of a programming technique known as *dependency injection*. The purpose of this technique is to separate the construction of objects from the entities using said objects. Instead of having an class or a function construct a given service they depend on, they receive such a service by an external actor known as *injector* [86].

Character construct the objects it serves as facade and injects the dependencies they need. Since said objects may depend on each other (for example, the state machine needs a reference to the movement class), it also ensures no race conditions take place as they are initialized in the right order.

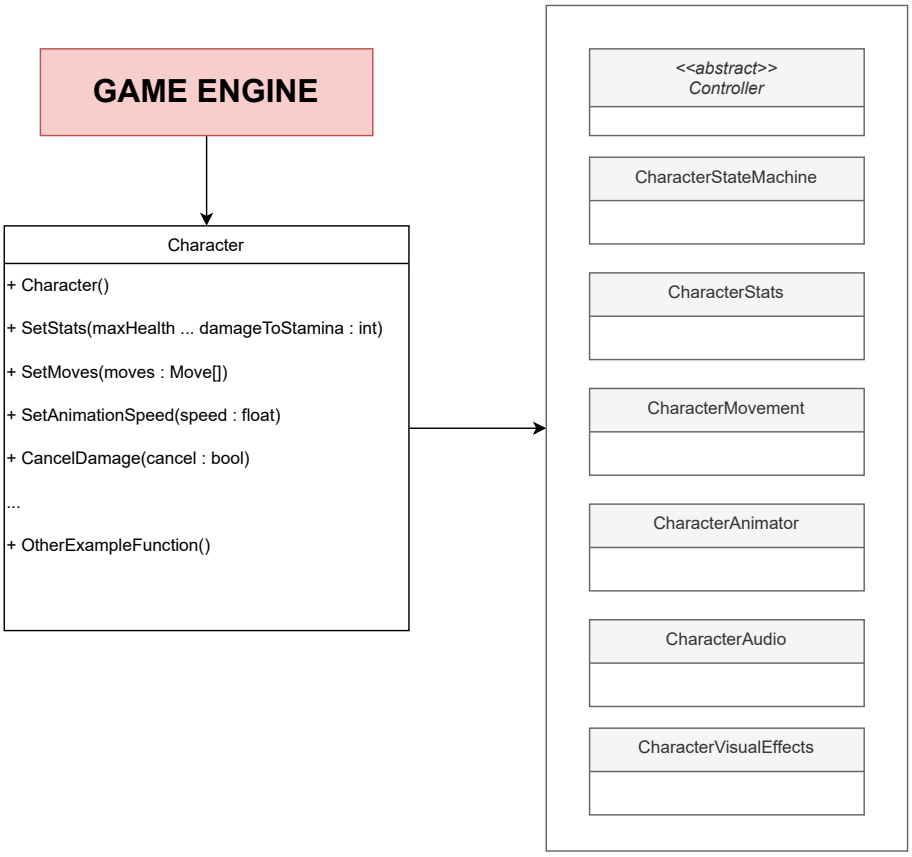


Figure 3.4 – Character class diagram.

3.4 Observer

Another pattern that will be used several times throughout the entire software design is the Observer, and it is not without reason. The Observer is a software design pattern where an object known as the *subject* notifies a list of *observer* objects when certain events take place [87]. It is a common underlying pattern in Model-View-Controller architecture as it facilitates the interaction between the three components without unnecessary coupling.

This is usually achieved by having an Observer interface with a virtual "OnNotify" public method, and an Subject class which holds a list of observers and implements the two public methods "AddObserver" and "RemoveObserver", along with a protected "Notify" function which notifies all subject's observers about a specific event (see Figure 3.5).

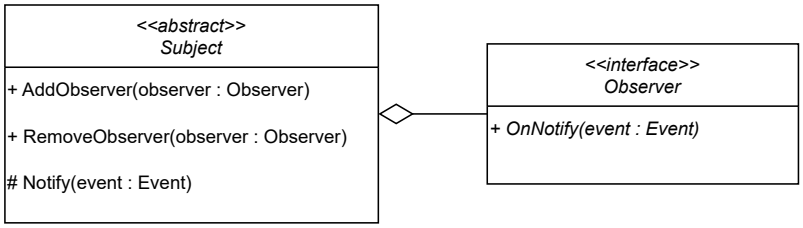


Figure 3.5 – Observer pattern class diagram.

"AddObserver" and "RemoveObserver" are made public so that they allow outside control of who receives notifications. The subject communicates with the observers but it is not coupled to them. It does not need to know the details of any observer (the classes that implement

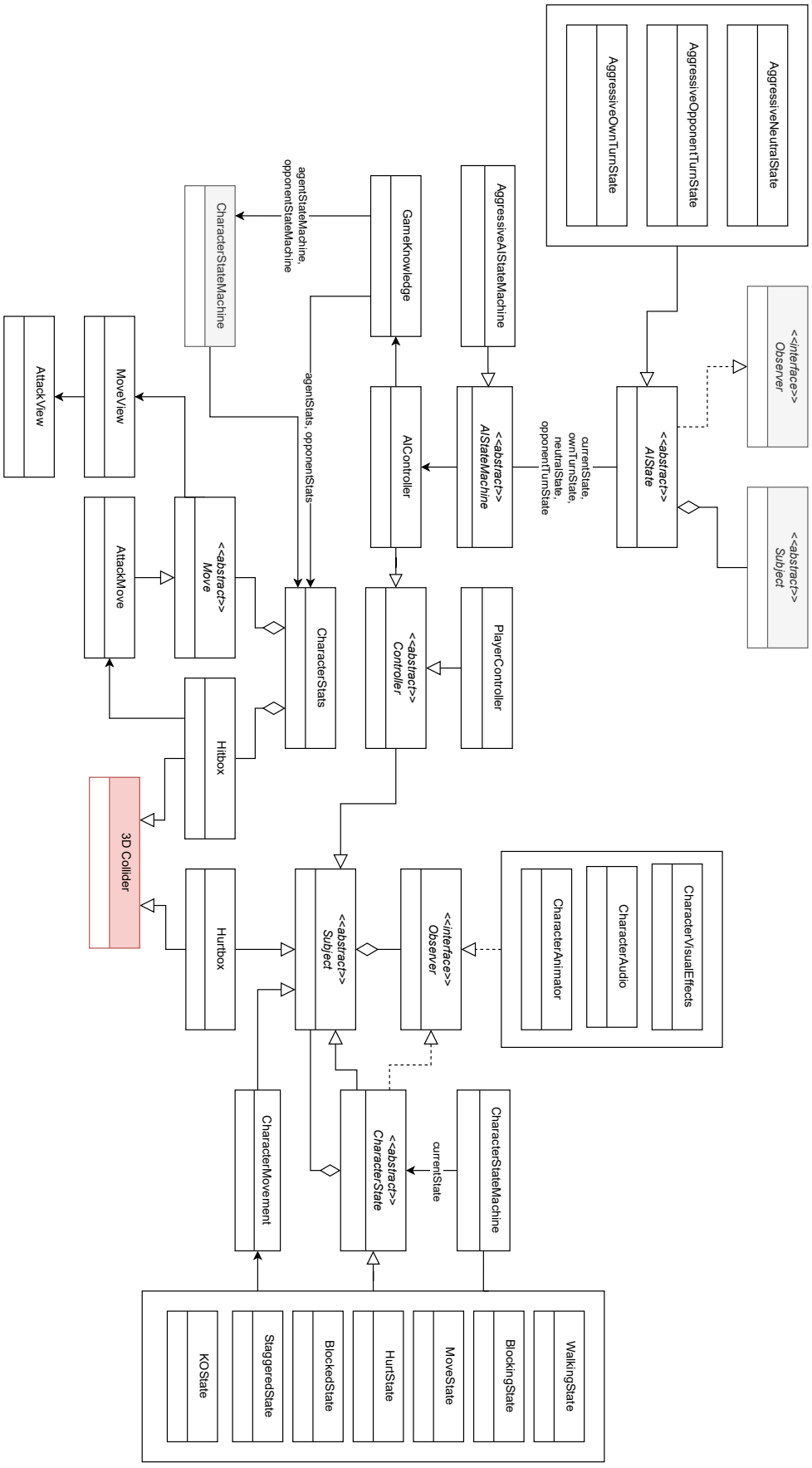


Figure 3.6 – Carnem-Levare's character full class diagram (simplified).

said interface, their methods, etc) nor have any specific line in their code to satisfy a particular observer's goal. It is only tasked to notify a list of generic observers when a certain event takes place in their own methods, whichever those observers may be.

In the case of this software's design, the main observer of the architecture is the character's state machine. Its behaviour will be explained in the following sections —as finite state machines are a design pattern of their own—, but we can describe it as an abstract machine that can only be in one state at a time, and transitions from one state to another after given a particular input. In the case of the character's state machine in Carnem-Levare, it implements the Observer pattern so that it is notified by a subject to change state.

Also, the three classes from the View component (*CharacterAnimator*, *CharacterAudio* and *CharacterVisualEffects*) are observers of the state machine, representing changes in the model by displaying the right animation, audio and effects given by each particular character state.

3.5 Model

Following the MVC pattern, we will now describe the classes that form the model, along with any software patterns they might include in their design. See Figure 3.7 for a complete class diagram.

3.5.1 CharacterStats

CharacterStats is the class that stores the model's main data, the set of attributes and objects established by the game design document necessary for a character to function. Not only it gives access to said information but also makes calculations given external objects and its own data.

Current health and maximum health are the most fundamental values for the interaction to take place, as it is the main measure of the player's goal (to defeat the opponent, not to be defeated). Stamina was described in subsubsection 2.3.2.2 as the resource that is spent everytime the player receives hit. Nonetheless, it is applied to characters in general so that the enemy is also able to block with a cost.

Just as there is health and stamina, there must be attributes that state the intrinsic strength of each character beyond the base damage each attack might deal. These two attributes are *damageToHealth* and *damageToStamina*, which are self-explanatory. *CharacterStats* then provides functions that calculates the final damage dealt given the base attack damage of the move performed and the character's own damage values.

This calculated damage may be substracted to *CharacterStats*'s current health and stamina through the method *HurtDamage* after every connecting hit. This method not only modifies these values but also forces the state machine to transition to the corresponding hurt state: a simple stun state if there is health and stamina remaining, a staggered state (long stun) if stamina reaches zero, and a knockout state if health reaches zero. The method *BlockedDamage* functions the same as HurtDamage but does not modify health.

Furthermore, we may implement delayed hitstun through the use of a *hyperarmor* flag. Hyperarmor is a fighting game term which refers to the state in which a character may absorb a hit without entering hitstun while performing an attack or move [7]. By having a hyperarmor

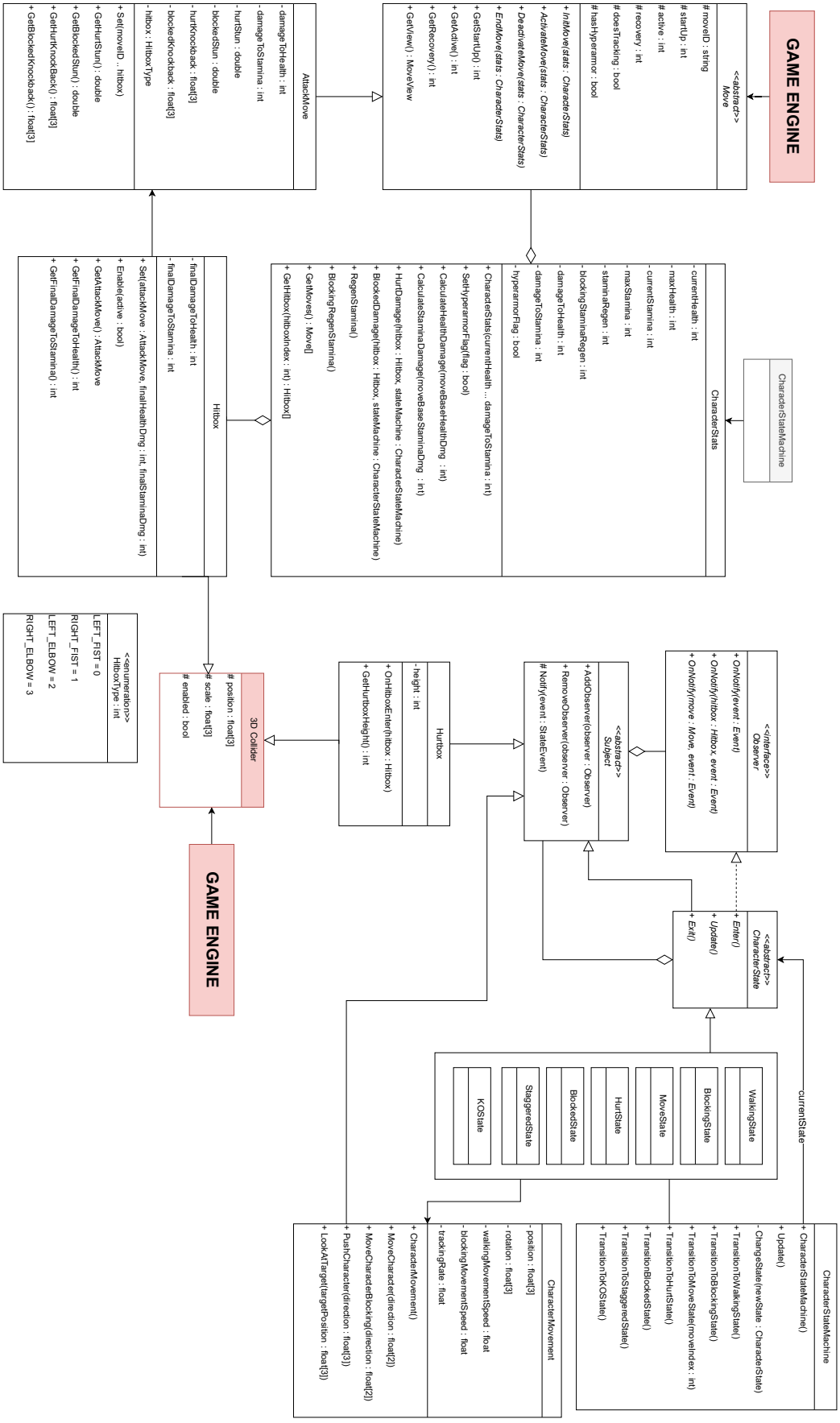


Figure 3.7 – Character's model class diagram.

flag, we allow moves to have hyperarmor by disabling the transition to a stun state if said flag is enabled. However, if the character's stamina reaches zero, they will still be staggered. This allows game designers to craft typical singleplayer enemies —whose moveset consists of slow, hyperarmor attacks— that do not get stunned before receiving enough consecutive hits.

Lastly, *CharacterStats* contains the customizable moveset mentioned in paragraph 2.3.2.3.1 in the form of a list of *moves*. Instead of making a specific attack class, we decided to make an abstract *Move* class that represents any fleeting action that may be performed by a character. We may define fleeting actions as any ability that takes places during an specific window of time, such as an attack or a dash. All real time action games have actions such as this, and making a generic move class gives us a framework from which to make concrete implementations in the form of subclasses.

This design decision fulfills SOLID's dependency inversion principle, which states that it is preferable to depend upon abstractions instead of concretions. It gives greater reusability value to the higher-modules that use the Move class as they do not depend on the details and subclasses that may be implemented beyond the abstraction.

This also fulfills Liskov substitution principle, which states that modules referencing base classes should be able to use objects of derived classes without knowing it. Any reference to Move should be replaceable by any of its concretions without breaking the program.

3.5.2 Move & AttackMove

Move contains the basic data all fleeting actions need to be performed and, similarly to Character's class, it allows game designers to modify said data directly through the game engine instead of code.

This data includes, most importantly, three numerical values that define the move's frame data, which was described in paragraph 2.3.2.3.2 yet in the specific context of attack moves. In this case, we use frame data for all kind of moves as it is an useful tool to design and balance any fleeting action. For example, a dodge move (such as Elden Ring's rolling action) could be divided into the usual three periods (start up, active and recovery), the difference being that the active period, instead of stating for how long an attack is dealing damage, it states the dodge move's invulnerability frames.

All fleeting actions have an effect in the game world or the character performing it. Frame data design allows us to state how long does it take for such an effect to take place, how long said effect lasts, and to establish a drawback in terms of recovery time.

However, there is an important distinction to be made in regards to how frame data will be applied to Carnem-Levare's design. The problem with balancing moves' timings around frames is that frames per second are rarely constant in a videogame. Moves are designed around an FPS standard (usually, 60 FPS), so that 20 frames of start up actually mean said move takes one third of a second to be active. Though a simple calculation could translate those 20 frames of startup to the equivalent period given by current running FPS (e.g. 20 frames of startup at 60 FPS is 40 frames at 120 FPS), multiplayer fighting games tend to lock the game at 60 FPS, which has some advantages to consider. Aside from facilitating frame data's balancing and implementation, it also makes framerate consistent between tested gaming platforms that may not have the same capabilities. If the game were to run at the highest framerate possible, players on more powerful

systems would have an advantage as animations are smoother and easier to react to [88].

Since Carnem-Levare is meant to be a singleplayer experience, framerate will not be limited as there is no reason to do so. Furthermore, to avoid unnecessary calculations and make balancing more intuitively, frame data will instead be measured in milliseconds.

Move also includes four abstract methods which define what the move performs in each frame data phase. It receives *CharacterStats* as a parameter as it includes all the character's relevant information and methods to modify said information.

- *InitMove*. Triggers at the very beginning of a move. Might be used to initialize data the move needs to be performed, set tracking or hyperarmor flags, etc.
- *ActivateMove*. Triggers after the start up phase, "activating" the move's effect. In the case of an attack, it would enable its hitbox. If it were a healing move, it would add a given numerical value to the character's current health.
- *DeactivateMove*. Triggers after the active phase, "deactivating" the move's effect. If it's an attack, it disables its hitbox.
- *EndMove*. Triggers after recovery, when the move ends. It is not necessary for all moves, but it can be useful to reset changes made to a character's state during previous phases, such as disabling hypearmor or invulnerability.

Move does not implement any of this methods but instead leaves them open for extension for any move concretion to use and specify its behaviour, therefore fulfilling SOLID's *Open-closed* principle.

One of these concretions is *AttackMove*, which adds all the necessary attack data stated in paragraph 2.3.2.3.2 along with the underlying behaviour it needs for the combat interaction to take place. This include the damage that it deals to both health and stamina, and hitstun values to state the advantage it provides to the attacker. Furthermore, *AttackMove* extends *Move*'s frame data methods to enable its intended hitbox during the active phase and disable it prior to recovery.

3.5.3 Hitbox & Hurtbox

Hitbox refers to an invisible predefined area or space within a game's graphical environment which, when colliding with another invisible area known as *hurtbox*, triggers the effect of an attack (lowering health, stunning the opponent, etc) [7].

Hitboxes are known to be an integral part of the fighting multiplayer genre (see Figure 3.8), but they are essential to any game that includes real-time combat. They not only make the interaction possible, but also take a huge role in balancing the game with matters such as how large a hitbox should be for a given attack or for how long should it be enabled. When an attack is said to be *active*, this refers to the window of time when its hitbox is enabled and is capable of interacting with a character's hurtbox.

In the case of Carnem-Levare, hitboxes are designed as collision boxes that exchange information between models. Usually, game engines —such as Unity— include colliders as part of the physics system, along with overrideable methods that only trigger on collision [89]. Using a layer-based collision detection approach, hitboxes may be implemented as colliders that only interact with other colliders of the same layer (hurtboxes). Once this interaction takes place, an

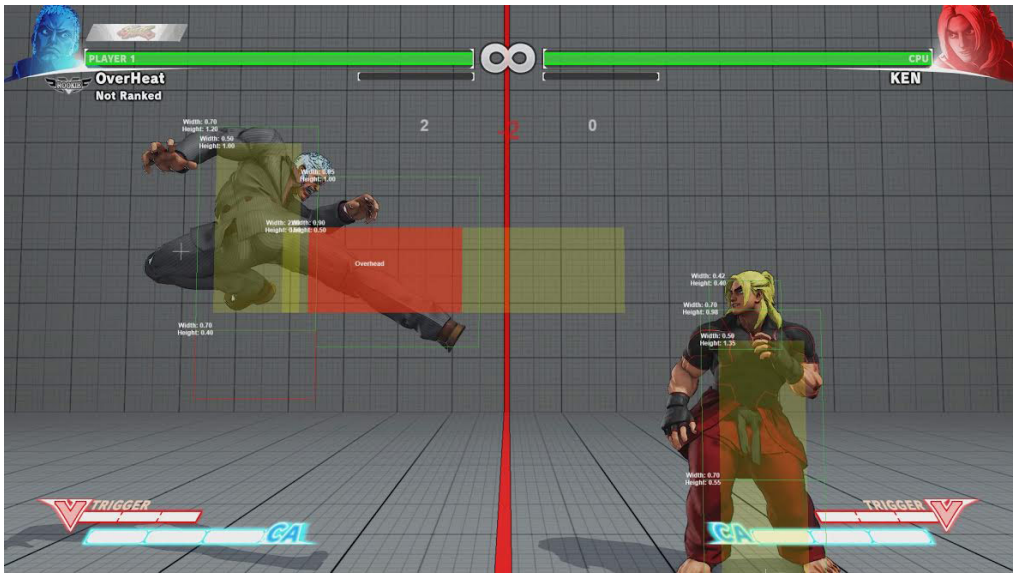


Figure 3.8 – Street Fighter V's hitboxes (red) and hurtboxes (green).

on collision method is triggered to send all relevant information from the attack to the receiving character's hurtbox.

Following the Observer pattern, the hurtbox then notifies its observers and passes the hitbox's information through the notification. The state machine receives said notification and handles the information accordingly, by reducing the character's health, stamina, and entering the corresponding hit state for a specific amount of hitstun.

3.5.4 CharacterMovement

Following SOLID's single purpose principle, we designed a class entirely dedicated to handling the character's movement. More specifically, a class which provides access to methods that manipulate the character's position and rotation in the environment, along with parameters which define how this manipulation takes place (such as walking speed and tracking rate).

We may classify character's movement into the three types which this class handles: tracking, walking and knockback. Tracking consists of the targeting system that was deemed necessary in paragraph 2.3.2.1.1 for intuitive free movement. When using the *LookAtTarget* method, the character is rotated towards their opponent up to a specific amount defined by a *tracking rate* parameter. If this method is called frame by frame (using an update or game loop call), said tracking rate becomes the character's rotation speed, which might be subsequently adjusted by designers to smooth out the animation and facilitate dodging attacks.

Walking movement is self-explanatory. Using the *MoveCharacter* method, the character is displaced towards a given direction by a specific amount, established by a walking speed parameter (how far does it move in a single frame). Since the character is also able to walk while blocking, though slower, the class implements a secondary *MoveCharacterBlocking* method which functions exactly as *MoveCharacter* but uses instead a different speed parameter. Both speed parameters may be tweaked by game designers in order to balance a character's capacity to approach and outspace the opponent.

Finally, knockback is a type of movement intended for two purposes: complementing the

impact of an attack by pushing the character in a particular direction (e.g. pushed upwards after being hit with an uppercut), as if they were shoved by the force of the strike; and serving as a pushback mechanic, where the opponent is pushed slightly further after every hit in order to eventually break the attacker's combo. The method *PushCharacter*, when used, immediately applies a directional force to the receiving character. Through consecutive hits, this directional force parameter may be multiplied proportionally to the combo as a way to eventually push the character out of the attacker's range.

3.5.5 CharacterStateMachine & CharacterState

The remaining classes of the model are designed around a mathematical model of computation known as *finite-state machine*. This model describes an abstract machine that can be in exactly one of a finite number of states at a time and, in response to input, changes from one state to another [90]. It is composed of a fixed set of states and a set of transitions, which define how the machine changes state given a specific input.

This model is useful to encapsulate varying behaviour for the same object. Resorting to conditional statements to change an object's behaviour in runtime may lead to error-prone code, as you have to account for each condition separately and in combination with each other [85].

An example of this could be programming the character's movement in an standard 2D platformer. When you press B in a gamepad, the character jumps. However, there's no accounting for when the character is in the air, allowing the player to jump indefinitely. Therefore, another condition needs to be added, which prevents the player from jumping when the character is already jumping.

Furthermore, the game design document states that the character should be able to crouch when pressing down in the D-pad, which allows the character to pass through tight spaces but prevents them from jumping. Now, there has to be a statement that allows the player to crouch if they press down *and* they are not jumping (otherwise, they would be able to crouch mid-air). Similarly, the jumping statement now needs to make sure that the player is both not jumping nor crouching. It becomes evident how this approach is flawed.

By simulating a finite-state machine, we establish the possible states the character might find themselves in, along with the actions they are able to perform in each of those states. Following the previous example, we establish three states: standing, crouching and jumping. When standing, the player is able to move, duck or jump. When crouching the player is only able to move or stand back again. Lastly, when jumping, the player is only able to move, in order to change its direction mid-air. There is no need to account for situations such as the player jumping again while in the air, as the jumping state inherently does not allow jumping.

These actions also define transitions between states. For example, when the player presses down, the character transitions from standing state to crouching state. Nonetheless, external factors also apply, such as the character landing on ground after jumping (see Figure 3.9).

Figure 3.9 – Platformer movement example state diagram.

We may draw up to seven possible states from Carnem-Levare's game design. To begin with,

we know the character is able to switch between walking normally and blocking, which already results in two states (walking and blocking). From either of these two states, a character is able to perform a move, defining another state (move state).

Then, we have all states which involve the character receiving a hit. If they receive a hit without blocking nor reaching zero stamina or health, the character is simply hurt (hurt state). If they receive a hit when blocking and they have stamina left, the character blocks the impact (blocked state).

Now then, if they run out of stamina (either when hurt or blocking), they are staggered, which results in a prolonged hitstun and a new stagger state. Lastly, if they run out of health, they are knocked out, resulting in a KO state. Take a look at the full state diagram present in Figure 3.11.

3.5.5.1 State pattern

Finite-state machines describe an useful abstraction which simplifies a typical character's behaviour from the mess of conditions and exceptions that may be inferred from the game design document. Nonetheless, this model on its own does not provide a software specification that can be implemented into code.

Here is where Gang of Four's *state pattern* comes into play. Implementing state behaviour into a single class is undesirable as it makes it impossible to add new states or change behaviour of an existing state without modifying said class (therefore, breaking SOLID's open-closed principle).

Instead, the state pattern contemplates each state as its own class, which extend an abstract class or interface. Then, the class tasked to run the object's behaviour (that would be *StateMachine* in Figure 3.10) delegates this task to its current state object instead of implementing state-specific behaviour directly [85]. Carnem-Levare's design follows this pattern by establishing a *CharacterState* abstract class which each concrete state extends, along with a *CharacterStateMachine* class which handles transitions and delegates behaviour to its current state.

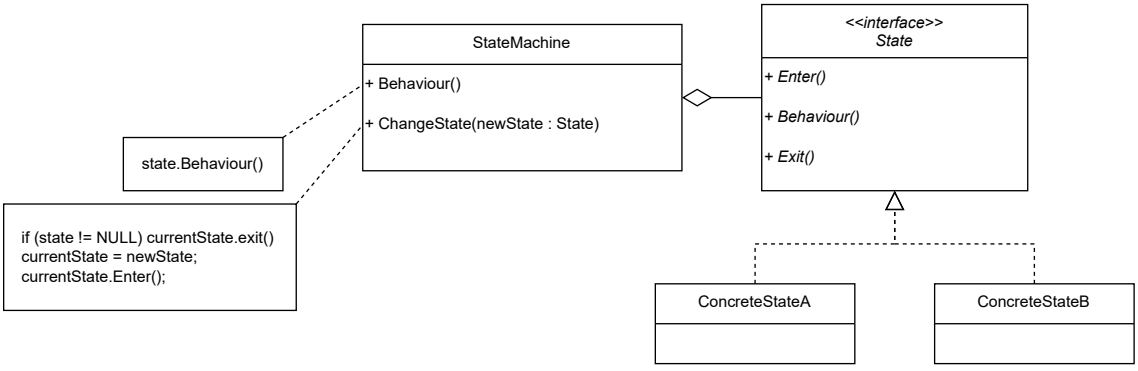


Figure 3.10 – State pattern sample class diagram.

Furthermore, state objects not only implement a *behaviour* method but also enter and exit calls which allows us to define more state-specific behaviour only meant to trigger during transitions between states.

Returning to the platformer movement example, when the player jumps the character should switch to a jumping sprite for effect. Jumping state's *behaviour* method is not an appropriate

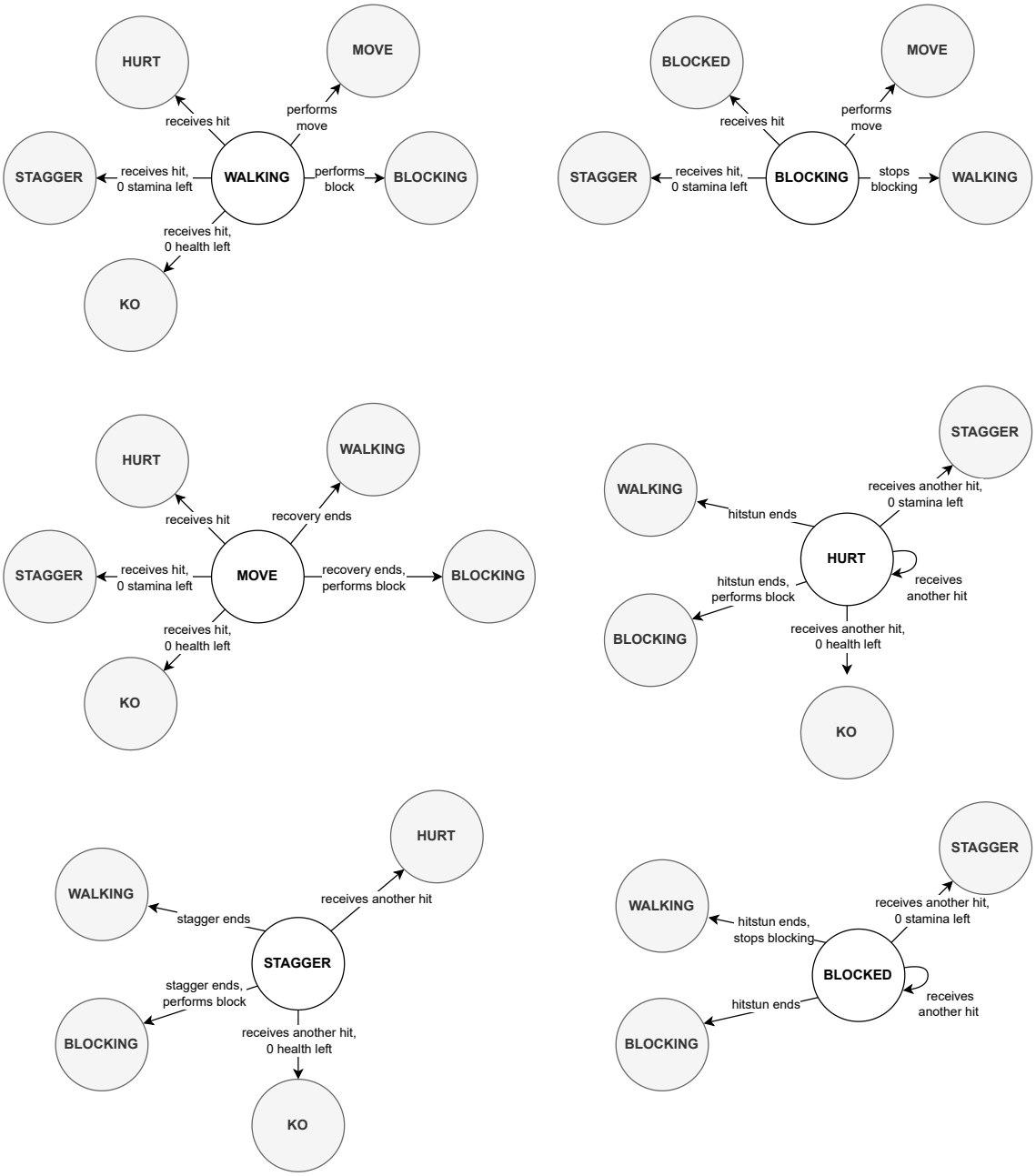


Figure 3.11 – Carnem-Levare character state diagram.

function from which to perform this task as it would switch the sprite every time the state machine runs its behaviour, adding unnecessary overhead for a task that should only be realized once during a state's lifetime.

Instead, the character's sprite is switched during an *enter* method call. When the state machine class transitions to a new state, it calls the previous state's *exit* method, assigns the new state to its current state object, and calls *enter* for this new state. This way, character switches to a jumping sprite only when entering jumping state, and the task to return to a standing sprite may be performed on jumping state's *exit* or standing state's *enter*.

This two methods further encapsulate state-specific behaviour within each state, as now states are not only responsible to implement their own behaviour but are also capable to initialize and prepare data once on entering the state, while also being able to destruct or undo this data on exit. However, we have not yet discussed what this state's behaviour consists of in the context of videogames. Here is yet another pattern which perfectly complements state machines in this regard: the Update Method.

3.5.5.2 Update Method

The Update Method is a basic game programming pattern that comes already implemented in most game engines of the market, and it is one of the main approaches programmers have to define an object's behaviour in a game environment.

When using this pattern, the game maintains a collection of objects for a given scene or game world (see Figure 3.12). Each of these objects implement an *update* method which simulates one frame of the object's behaviour [91]. Every frame, the game updates every object in the scene's collection by calling their corresponding *update* method.

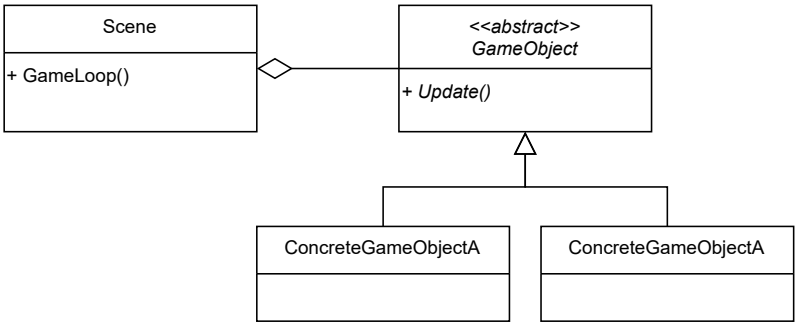


Figure 3.12 – Update pattern sample class diagram.

This pattern is useful to implement real time behaviour which needs to run constantly during the lifetime of a scene, such as character's movement, health bars that regenerate over time, etc. It is not as useful for objects whose behaviour only need to take place after specific events or inputs, such as chess pieces or characters in a turn-based combat game.

In Carnem-Levare's design, the update method is merely another tool for states to define behaviour, in conjunction with the previously described enter and exit methods. More specifically, behaviour intended to run constantly during a state's lifetime. The game world updates the state machine by calling its update method, and the state machine delegates this updating to its current state object.

The main beneficiaries of this pattern are the classes *CharacterMovement* and *CharacterStats*.

CharacterMovement updates a character's position and rotation every frame through the use of *MoveCharacter* and *LookAtTarget* methods, in the states where said movement is relevant (walking only takes place in walking and blocking states, targeting takes place in all but knockout state). *CharacterStats* regenerates a character's stamina every frame through the use of *RegenStamina* methods, then again, in the states where this regeneration should take place by design (walking and blocking states).

However, it's important to note that frame by frame behaviour is, by default, frame dependent. Operations such as stamina regeneration must take into account variable framerates to provide a constant, unified experience. Most game engines include variables in their public API from which to read the current running FPS or the time elapsed from the previous frame to the current one, such as Unity's delta time [92]. Ultimately, it depends on the game engine how this will be implemented, but it must be considered regardless.

3.5.5.3 State as Observer

As it was mentioned in the Observer pattern section, *CharacterState* implements the Observer interface in order to be notified by other character's classes to change state.

More specifically, each concrete state holds a list of subjects. When entering the state, it adds itself to every subject's list of observers using the public *AddObserver* method. Each concrete state then handles its subjects' possible notifications by implementing the *OnNotify* method.

For example, walking state adds itself to *Controller*'s list of observers. When the controller commands the character to block it notifies all its subjects. Walking state then handles said notification and transitions to a blocking state.

However, when exiting, each concrete state removes itself from its subjects' list of observers so that it is not notified when it is no longer active. This way, only the current state object handles notifications from other character's classes.

This methodology allows us to contemplate specific notifications for each state and make them handle differently if necessary. For example, walking state handles hurtbox's hurt notification by transitioning to a hurt state, while blocking state handles the same notification by transitioning to a blocked state. Nonetheless, knockout state does not observe the character's hurtbox at all, as characters cannot be hurt when they are already knocked out.

3.5.5.4 State as Subject

Moreover, *CharacterState* also extends *Subject*. While it may seem counter-intuitive to inherit from both observer and subject, they serve quite different purposes.

CharacterState inherits from *Subject* in order to implement two specific notifications, which are *on enter* and *on exit*. Each state notifies its list of observers when entering the state, then again when exiting, allowing the state machine to be decoupled from all operations that do not involve the model, such as animations, camera effects, etc.

External classes that may need to coordinate with the state machine only have to observe and handle the corresponding state's enter or exit notifications. States themselves have no knowledge of these classes which makes the software more flexible and maintainable.

3.6 View

The view in Carnem-Levare's character design aims to represent the model in real time. The most basic representations of character information in an action videogame are animation, audio and visual effects, though other areas could include interface such as health bars or menus to switch between abilities. For this dissertation's purpose, we will focus on the previous three basic areas.

As evinced in the diagram presented in Figure 3.13, the view results in a rather simple and straightforward design by virtue of having states extend the subject class. By implementing Observer's *OnNotify* method, classes in the view are able to handle each state's on enter and on exit notifications in order to switch to the corresponding animation, play a specific sound or display any visual effects such as particles.

CharacterAnimator does not play the character's animation by itself. Instead, it holds a reference to the engine's go-to utility to render and play animations, and uses it to queue the corresponding animation when notified by a state subject on entering. It is also notified by *CharacterMovement* to play a movement animation in the appropriate direction when using its *MoveCharacter* method.

CharacterAudio functions similarly to *CharacterAnimator*, holding a reference to the engine's audio utility which it later uses to play the appropriate sound when notified by a state.

Lastly, *CharacterVisualEffects*, by the time of this dissertation, only handles predefined particles effects with specific animations, position in the 3D space in regards to the character, and duration. When entering a state, it loads said particles using the engine's particle generator.

There are two main events the View needs to represent for a fulfilling gameplay experience: performing a move and receiving an attack. When the controller commands the model to perform a move, the model registers said command and changes to a move state. On entering the move state, it notifies its observers and sends the move's information through the notification (using *OnNotify(move : Move, event : Event)* method). The view, as subjects to the state machine, receives said move when handling the notification and uses it to trigger animation, sound and visual effects.

However, *Move* class does not have any reference to view's elements by itself. Instead, it holds a reference to a new class known as *MoveView* [1], which stores all animation, sound and particle data. When the view receives a move through a subject's notification, they may access its *MoveView* object for all data necessary to represent the model.

Furthermore, *MoveView* is extended by *AttackView* to add all remaining View's data necessary for an attack. More specifically, the animations, sounds and effects when a character receives an attack, which is the other main View event.

When a character is hit, its hurtbox notifies the state machine, which later switches to the corresponding hurt state depending on calculations made by *CharacterStats*. When entering any of the possible hurt states, the state notifies its subjects and sends the hitbox's information through the notification (using *OnNotify(hitbox : Hitbox, event : Event) method)*. Hitbox has a reference to *AttackMove*, which the view can access to read its *AttackView* object and play the

[1] The reason for *MoveView* to be coupled in such a way to the model's *Move* class is to facilitate access to game designers and artists. Instead of splitting each move into two separate, independent objects, *Move* holds a reference to its view so that all move's informaton is contained in a single object accessible through the game engine without code.

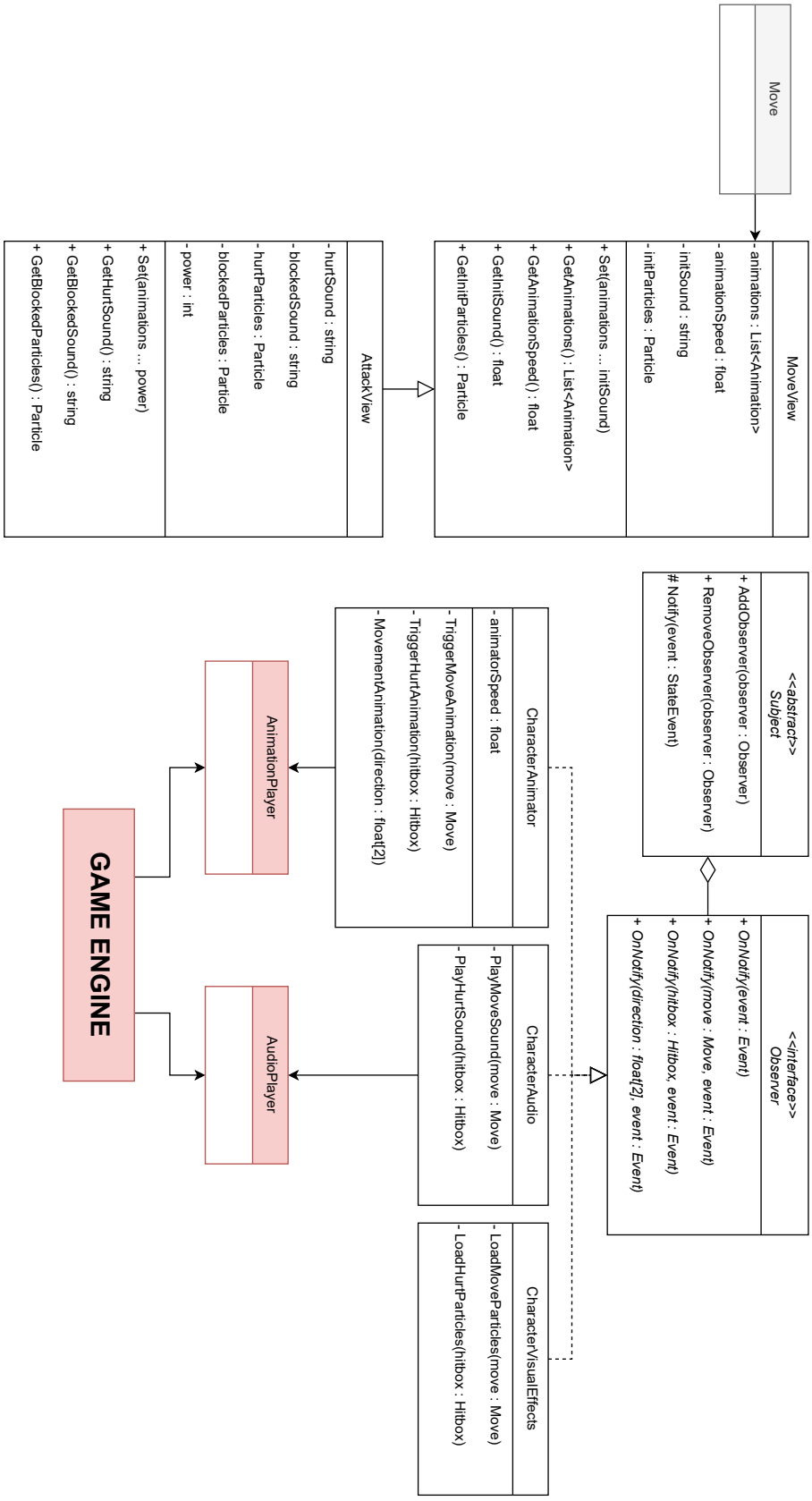


Figure 3.13 – Character's view class diagram.

corresponding hurt animation, sound and particles.

3.7 Controller

The controller is the one component responsible to accept input from an user external to the architecture and convert it to commands to manipulate the model. In the case of Carnem-Levare's character design (see Figure 3.17), these commands are notifications made by a subject class when receiving input from two types of users: player and artificial intelligence.

An abstract *Controller* class implements a protected method for every character's action established in the game design chapter: movement, blocking and moves. Each of these methods, when called, notify the state machine along with any information necessary for the action to be performed (movement direction, move number).

However, *Controller* only provides access to said methods. It is the classes that extend *Controller* which are responsible to call them, and those classes are *PlayerController* and *AIController*.

3.7.1 Player Controller

The player presents the simplest design from the two types of users. A single class contemplates a set of valid actions the player might press. These actions, implemented as public methods, are assigned to gamepad buttons and keyboard keys by the game engine, so that pressing them results in calling the corresponding method.

It further delegates the responsibility to command the model to a tertiary, external entity which is the game engine's input framework. Nonetheless, it is necessary in order to provide a facade, independent to the controller, which said input framework can understand and utilize. It also allows us to encapsulate input behaviour inside this facade, which may change depending on player's necessities (such as implementing input buffering for better responsiveness).

3.7.2 AI Controller

Contrary to the player, the AI user involves a complex system beneath its immediate *Controller* subclass, which is the artificial intelligence itself. Instead of receiving external input from another device such as a gamepad, *AIController* handles decisions made by a system functioning inside the game itself.

Similarly to *PlayerController*, *AIController* provides a facade to the artificial intelligence with methods that allow them to command the model in a format which the AI can utilize. It also provides access to the class *GameKnowledge*, which contains all the game's information relevant for an user to make fruitful decisions. More specifically: character and opponent's stats, character and opponent's state, and the distance between the two.

In the context of intelligent agents, this information would constitute the agent's observable environment. Following Russell and Norvig's classification [93], we may consider the AI to be a *reflex agent*. Reflex agents are a type of intelligent agent which makes decisions and takes actions in response to the current state of the environment. In the case of Carnem-Levare, the agent controlling a given character makes use of *GameKnowledge* to establish courses of action

depending on factors such as the opponent's remaining stamina, their own remaining health, the opponent's current state, their own state, etc.

The most basic type of reflex agent is the *simple reflex agent*. These kind of agents only act according to the current state of the environment, following basic *condition-action* rules (see Figure 3.14). Their entire behaviour revolves around perceiving and responding to stimuli, having no conception of the previous state of the world nor following a long-term strategy.

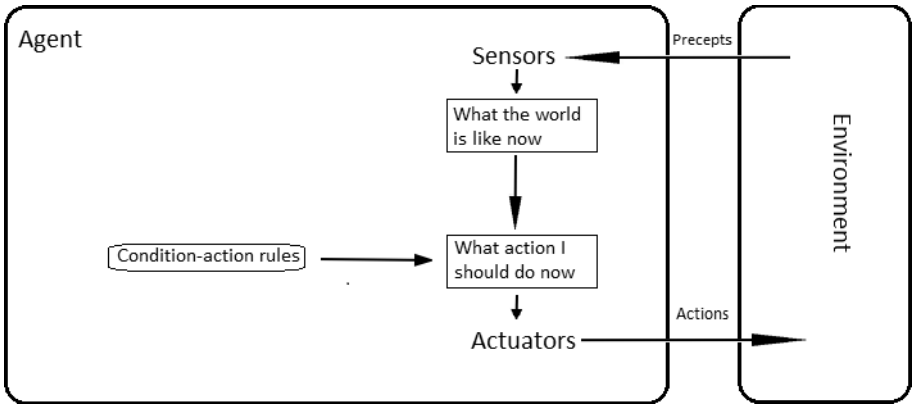


Figure 3.14 – Schematic diagram of a simple reflex agent.

A direct improvement to the simple reflex agent is the *model-based reflex agent*. This type of agent maintains an internal model which includes not only the current state of the environment but also previous percepts and possible predictions of future states, given by the observed impact of its actions (see Figure 3.15). This model of the world allows the agent to make more informed decisions when responding to current stimuli.

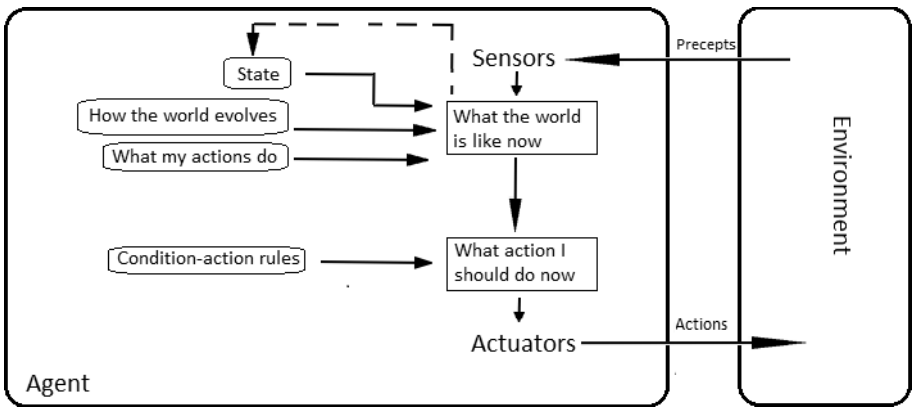


Figure 3.15 – Schematic diagram of a model-based reflex agent.

Lastly, *goal-based agents* expand on the capabilities of the model-based agents by introducing explicit goal information to the system. Goal-based agents have specific objectives they aim to achieve, which they use to further inform their decisions in combination to the current percept and internal model (see Figure 3.16). This allows them to plan about how to achieve said objectives and establish a course of actions.

Any of these three types of agents could be implemented for the AI user, because *AIController* does not specify behaviour by itself. *AIController* is detached from the AI architecture beyond its design, having no knowledge to the behaviour using its facade. Depending on the requirements established by the game design, this concretion could be in the form of a simple reflex agent

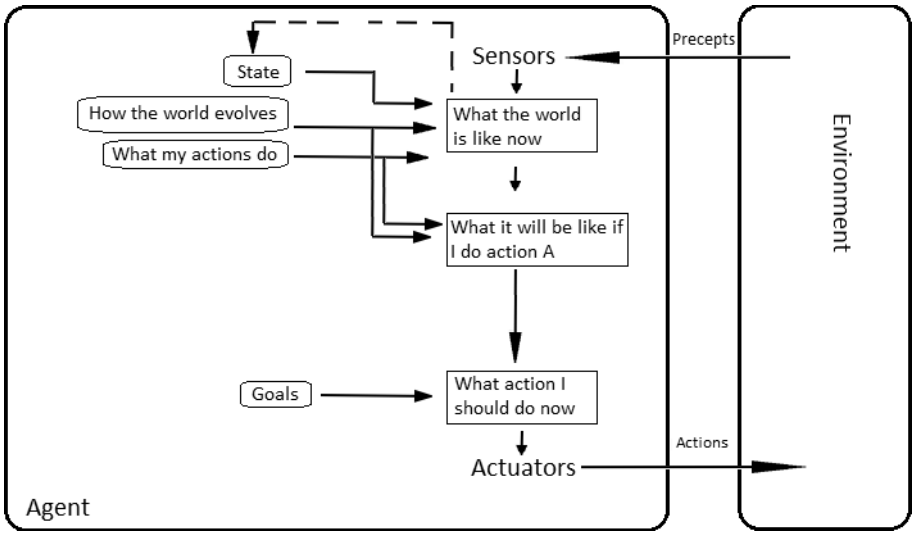


Figure 3.16 – Schematic diagram of a goal-based reflex agent.

whose only behaviour is to react to the opponent's actions, or a more sophisticated goal-based agent who develops a plan to defeat their opponent as efficiently as possible during the fight.

3.7.2.1 Reaction Time

The problem with *GameKnowledge* is that it provides perfect information about the agent's character and their opponent at all times. It is not a partial observation received through sensors but a direct reference to the true state of the environment (in this case, the fight), which could go against the objective of *flawed but consistent* artificial intelligence discussed in subsubsection 2.3.3.1.

Consider an enemy designed to simulate a keep-away fighter archetype. Their goal —beside defeating the player— would be to remain out of the player's range and counter their missed attacks with spacing. If we were to implement that behaviour directly it would result in an enemy impossible to defeat, which perfectly maintains the distance at all times and waits for the player to make a mistake to punish them. Not really a fair or interesting gameplay experience.

One solution could be to include explicit flaws to this behaviour. For example, adding error to the distance measure given by the model, or implementing probability to the enemy's actions (will they punish after the player attacks out of range?). However, this complicates the behaviour's design and adds unnecessary overhead to the AI system.

Instead, *AIController* implements a method known as *GetResponseTime*, which returns a random integer in an interval given by *AIController*'s minimum and a maximum reaction time variables. This response time can be used by AI to degrade their behaviour by establishing delays between their perception of the environment and the resulting course of action. It provides the error necessary to make the perfect information provided by *GameKnowledge* no longer perfect, since the AI user now has to act over outdated information (outdated by fractions of a second, which is more than enough in a real time action game). Since it is in the form of a delay, it can be implemented without affecting the behaviour's design.

This error can be accentuated or narrowed by adjusting the AI's reaction times. For example, a game designer might create an easy enemy with high reaction times, which the player can

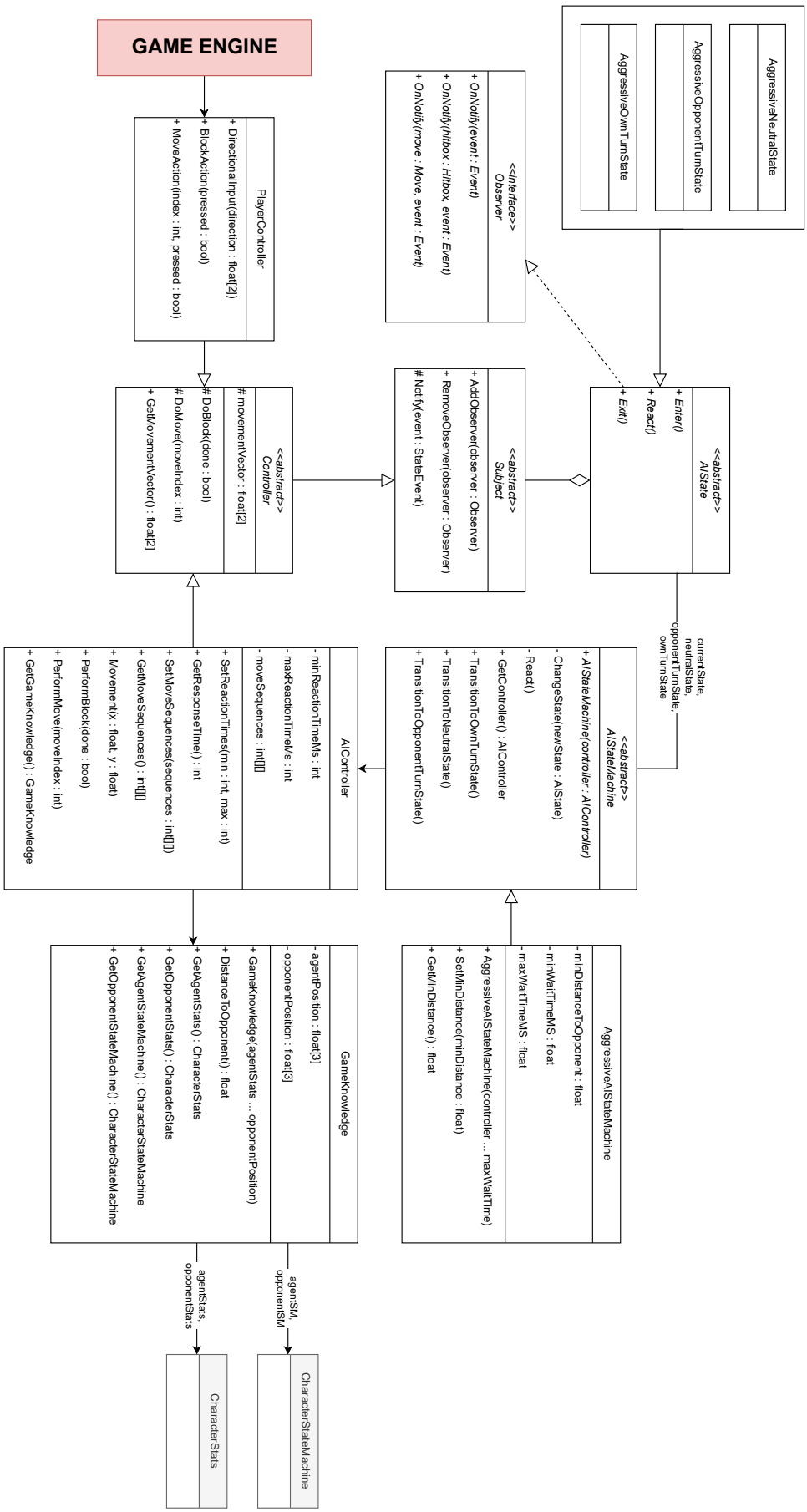


Figure 3.17 – Character's controller class diagram.

exploit by baiting and punishing their response. They could also design a harder enemy which instead punishes the player by reacting quickly to all their mistakes.

Enemies can also be made erratic by increasing the difference between the minimum and maximum reaction times. Enemies with greater reaction time difference could lead to exciting fights whose winner can hardly be predicted before the end, while enemies with smaller reaction difference could result in the kind of opponent which presents a consistent challenge for the player to study and overcome.

There are two main ways in which response time can be utilized. The first method is to use it in a loop; all information in constant change —such as the distance between the character and the opponent— may be accessed or updated every given milliseconds to establish an continuous error in the form of outdated information.

The second method is to delay the response intended to a given stimulus. Instead of immediately reacting to a percept —such as being attacked—, the agent waits for a set amount of milliseconds before performing said reaction.

3.7.2.2 Behaviour

As we stated previously, *AIController* does not specify behaviour. We now need to design the system that will command the enemy's character along with the technique to implement it most efficiently.

To know the kind of technique we need to design the artificial intelligence we first have to consider the game's requirements; what does the AI need to do? An AI user has the same actions available to any character: movement [2], attacking and defending. Their objective is to be able to decide which of these actions to perform at any given moment and how. Therefore, we need a technique designed for *decision making*.

The most popular methods for decision making in videogame AI are finite state machines, behaviour trees and utility-based AI [94]. These ad-hoc behaviour authoring methods have dominated the control of non-player characters for most of the industry's history, to the point of making the term *game AI* synonymous with the use of these methods.

Videogame AI has the specific requirement of being predictable, consistent and exploitable. Using more sophisticated techniques (such as machine learning) usually becomes counter-productive as the computational cost skyrockets only to produce behaviour that does not adapt well with the intended enemy's design. Ad-hoc behaviour methods turn out to be the most efficient options when an enemy needs to have specific and consistent behaviour for a set of given situations.

Therefore, using any of these three methods seems to be the right approach. However, we still need to choose between them. Each offer different advantages and are better suited for some games than others. In the case of Carnem-Levare, we may review the enemy's design which was established in chapter 2 to see how the AI is expected to act in each situation and which technique can better replicate it.

In subsubsection 2.3.3.2, the enemy was broken down into two classifications, one of which directly specified three possible modes of behaviour known as fighter archetypes. Continuing to subsubsection 2.3.3.3, we described the concept of turns in action games, where fights are broken

[2] Since the game world was described to be an empty space without obstacles in subsection 2.3.1, pathfinding is not considered nor needed for the AI's design.

down into three distinct states known as neutral, own turn and opponent's turn. Following the fighter archetypes, we designed three types of enemy AI by defining their behaviour in each of these states, along with their ideal state to be in.

From this description it becomes obvious which technique is better suited for Carnem-Levare's AI. We have three different states, each with their own set of actions to perform, and the transitions between said states when either character is hurt or escapes being hurt. Finite-state machine are the method which adapts best to the enemy's base game design.

3.7.2.2.1 Abstract AI State Machine

We can now begin designing the AI's system which will command the enemy's character. To begin with, the state machine will be designed following the state pattern in a similar fashion to the model's state machine, including the state as an observer. This leads to the creation of the classes *AIStateMachine* and *AIState*.

AIStateMachine has four references to *AIState*: current state, own turn state, neutral state and opponent's turn state. It implements a *react* method for all information that must be checked in a loop (mainly distance), accessing said information every given milliseconds defined by *AIController*'s *GetResponseTime*.

All remaining types of percepts are received through subject's notifications. When entering a state, said state is responsible to check for notifications relevant to its behaviour and to handle them appropriately. For example, the neutral state may transition to opponent's turn state when notified about the character being hit (the model's state machine enters hurt state).

It is important to note that, for the shake of modularity, the state machine is designed as an abstract class. *AIStateMachine*, instead of specifying behaviour directly, provides a framework from which to implement a character's AI using finite-state machines. The only concretion designed by the time of this dissertation's publication is *AggressiveStateMachine*, which implements a customizable rush-down archetype as a basic but functional type of AI.

3.7.2.2.2 Aggressive State Machine

As it was described in subsubsection 2.3.3.2, the rush-down playstyle is the aggressive archetype. A rush-down enemy seeks to be at close range, overwhelming the opponent with fast attacks and mixups while not letting them attack as a way of defense.

This is simple enough to implement following the design made in subsubsection 2.3.3.3. We just need to develop the set of operations it will perform to accomplish their intended objective in each state.

During their own turn, an aggressive enemy seeks to continue attacking and corner the opponent. However, this behaviour might be overwhelming for the player to deal with. So, instead, we defined a sequence of attacks the enemy can perform. When the enemy finishes a sequence, it gives up the turn for a random amount of time between *minWaitTime* and *maxWaitTime*. This allows game designers to set how aggressive the enemy should be by adjusting the length of the sequences or reducing wait time values.

There are another two ways the enemy might lose their own turn: getting hit or the opponent

going out of range. When the enemy enters hitstun the AI should transition to the opponent's turn state. The way this is achieved is by handling notifications from the model's state machine: when entering any hurt state, the current AI state receives said notification and immediately transitions to opponent's turn state.

The player going out of range is instead handled in the react loop. Every given milliseconds the state checks if the distance between the enemy and the player is below or equal to a certain threshold provided by *minDistanceToOpponent*. If said distance is greater, the AI transitions to neutral state.

During neutral, an aggressive enemy tries to close the distance and attack the opponent as soon as possible to force their own turn. This state can be implemented by simply moving the enemy in the direction of the player while they are out of range, checking the current distance in react loop. Once in range, it transitions to own turn state.

During the opponent's turn, the enemy defends until they find an opening to counter-attack and steal the turn. Said opening comes in the form of frame advantage: when the enemy is hit, they are in hurt disadvantage for a given amount of time. After said hitstun, they might have time to block but not to attack, putting them in block disadvantage. Once all this disadvantage dissipates, the enemy is safe to counter-attack.

Implementing this is as simple as handling walking and blocking state's on enter notifications. If the model's state machine manages to transition from any of the hurt states back to the default walking/blocking states, it means they are no longer in disadvantage.

Chapter 4

Implementation

In the previous chapter, we successfully designed the software architecture necessary for the Player-NPC interaction, going into detail as to how each of the subsystems should be implemented to accommodate this architecture.

The in-engine implementation will mostly follow said design. Therefore, for the shake of conciseness, this chapter will not be a point for point showcase of the implementation nor will it feature any of the project's code (the reader may visit the project's repository at github.com/ggarredondo/Carnem-Levare).

Instead, we will go over the differences between design and implementation. We will first describe the engine in which this architecture will be coded, its structure and its general behaviour. Then we will detail the features that will facilitate said implementation while also discussing its limitations and possible workarounds.

4.1 The Engine

Unity is a cross-platform game engine released in 2005 which allows game developers to create 2D o 3D videogames for PC, mobile, console and virtual reality [95]. It is currently the most popular game engine for indie development and one of the most accessible for beginner developers. [96].

The reason to use Unity over other game engines is, in fact, its popularity. Unity first launched with the objective of "democratizing" game development by making it accessible to more developers [97]. As such, it has one of the most intuitive application design in the market, along with an exhaustive online manual with information about most of the engine's functionality [98]. This goes hand in hand with its community, one of the largest and most active in the world of game development, which results in a wealth of online resources, tutorials and forums for developers to find help and solutions to common problems.

Other popular free engines may offer features or improvements over areas that Unity does not excel at, such as Unreal's multiplayer framework or photorealistic graphics rendering [99]. However, they simply cannot rival Unity in terms of accessibility and usability.

In this chapter we will go over all of Unity's characteristics and nuances that were relevant to the implementing the prototype's gameplay, briefly describing their function along with their specific use in this project.

4.2 Scene

Scenes are the containers that holds and organizes all the elements (such as objects, camera, lights, interface, etc) that make up a part or the entirety of a game. The final product that results from a Unity project will be composed of the scenes specified in its build settings.

A typical scene may contain the camera that will render said scene, lighting, interface in

the form of HUD and menus, objects ranging from environmental non-interactable pieces to physics objects and animated characters, etc. It is the engine's most fundamental unit and it allows developers to organize their game in meaningful portions.

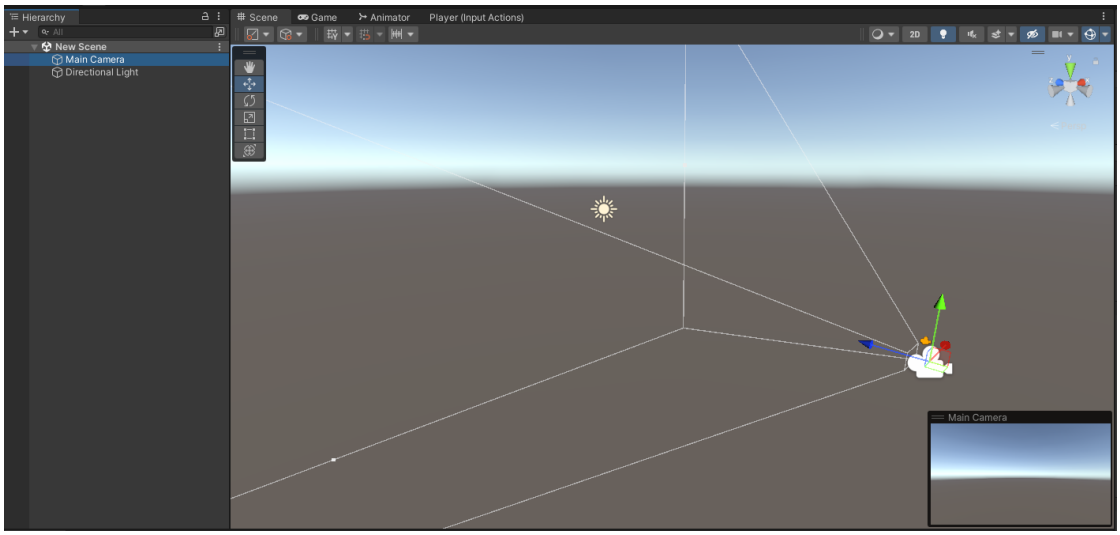


Figure 4.1 – Unity's default sample scene. Contains only camera and basic lighting.

In the case of this prototype, we implemented two different scenes to showcase the player-NPC interaction system. The first one is the *training* scene. When the player first starts the game, after going through a brief loading screen, a preliminary scene will load before the fight for them to test the character and its actions in a safe environment. The player may switch between different attacks and try them out against a sample, passive enemy.

This scene (see Figure 4.2) features a simple boxed environment, containing a camera, lighting, the player, an enemy with its AI disabled, and the menu that allows the player to switch between moves. It also includes a pause menu for the player to stop the game at any time or to return to the main menu.

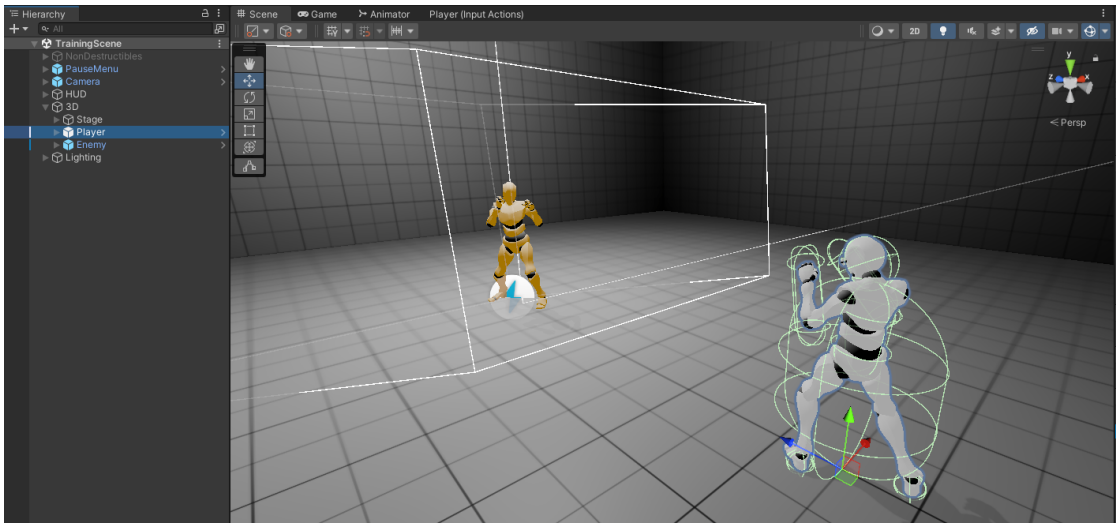
Once they are ready, they may hold the start button or Enter to begin the duel, effectively transitioning to the *fight* scene. The fight scene features a more elaborate environment and a fully functional enemy. The player can no longer switch between moves and is instead forced to fight in order to survive.

Aside from the environment, this scene (see Figure 4.3) contains nearly all same elements from the previous scene. However, the enemy is not present by default and is instead spawned from a list of pre-designed opponents when the scene is loaded. Furthermore, there is no menu to switch between moves. Instead, the HUD is occupied by animated bars which represent both characters' current health and stamina.

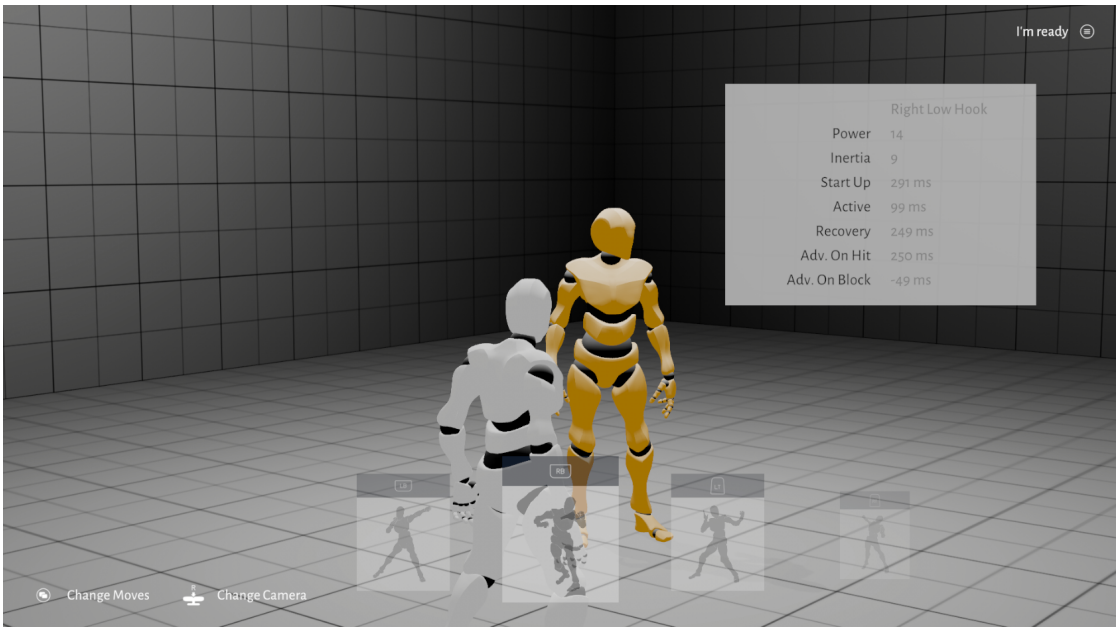
Further information about scene management, menu and other systems surrounding the gameplay may be found in my co-worker's dissertation: *Carnem- Levare: Basic systems and structures of a videogame.*

4.3 GameObject

GameObject is the base class for all entities in Unity scenes [100]. All objects that can be placed in a given scene inherit from this class, giving access to a set of properties and methods useful



(a) Scene view.



(b) Game view.

Figure 4.2 – Carnem-Levare prototype's training scene.

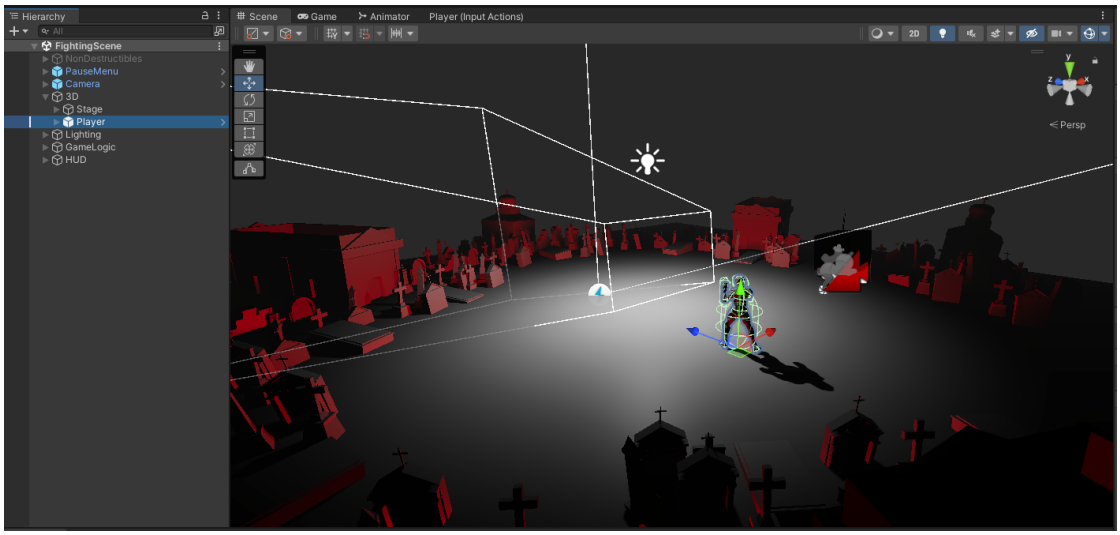
for scripting.

They can also be organized in a hierarchy, defining child objects which inherit the position, scale and rotation of the parent. This is useful to manage large scenes and create complex objects with profound behaviour by distributing components between child objects.

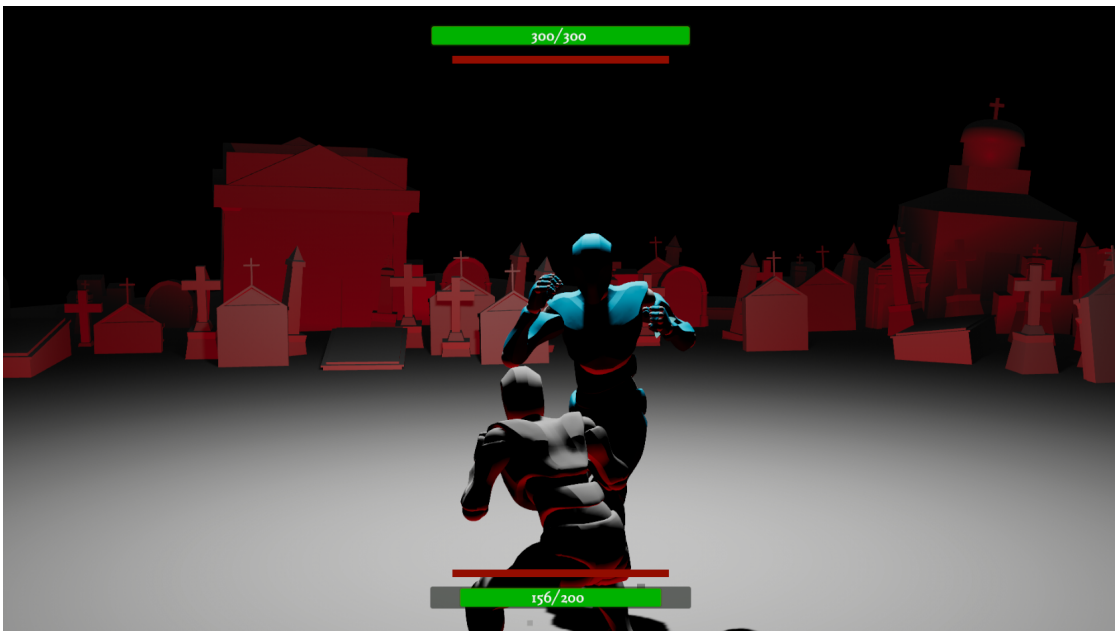
4.3.1 Components

Components are the base class for everything attached to a GameObject, defining its behaviour and containing properties which the user can edit to modify said behaviour [101]. Colliders, physics bodies, animators and MonoBehaviour scripts are all components which define a GameObject's function and can therefore be attached to one.

The list of components a given GameObject has can be viewed in the *Inspector*, a Unity win-



(a) Scene view.



(b) Game view.

Figure 4.3 – Carnem-Levare prototype's fight scene.

dow that displays information about the currently selected object or project asset (see Figure 4.4). This window allows developers to edit the properties of each component directly in the project (even in game mode) without coding.

All game objects have a unique Transform component which is automatically attached when the object is created. This component holds all basic information about the game object's position, rotation and scale in the environment.

The rest of components may be added through a list in the Inspector window or by dragging assets such as MonoBehaviour scripts to the desired game object.

Carnem-Levare's interaction system operates through the use of a character object hierarchy which contains the character's model, audio sources and hitboxes, along with all components which defines the behaviour necessary for the interaction to take place. From fundamental func-

tionality such as a transform, animator, collider, rigidbody, to the scripts designed in the software architecture such as the Character facade, player and AI controllers, etc.

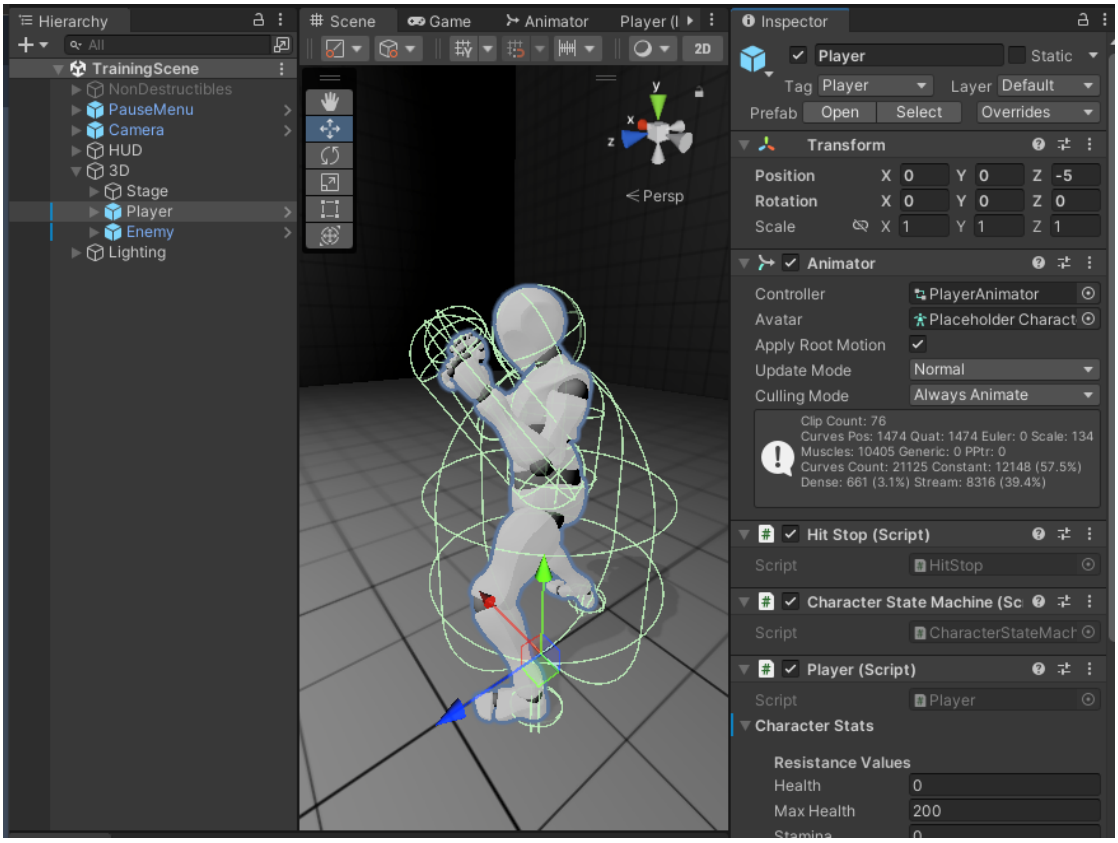


Figure 4.4 – Player GameObject selected in Training Scene.

4.3.1.1 MonoBehaviour

MonoBehaviour is the base class in Unity which allows programmers to create scripts that can be attached to game objects as components. It provides access to life cycle functions which programmers can use to define specific behaviour for the game object the script is attached to [102]. The most relevant are:

- **Awake().** Awake is called when an enabled MonoBehaviour script is being loaded, such as when the scene loads or a previously inactive GameObject is set to active [103]. It is useful to initialize variables or object states before the application starts.
- **Start().** Start functions similarly to Awake but is instead called on the first frame a script is enabled, before any of the Update methods are called for the first time [104]. Awake is called on all objects in the Scene before any object's Start function is called, which is useful to solve dependencies between objects. If a given object A needs an object B to be initialized, object B may be safely loaded in Awake() while A initializes in Start().
- **Update().** Update implements the pattern described in subsubsection 3.5.5.2. This function is called every frame if MonoBehaviour is enabled, allowing programmers to specify frame-by-frame behaviour for game objects [105]. *Time.deltatime* may be used to access the time elapsed between frames and perform frame-independent calculations.

- **FixedUpdate().** Fixed update implements a similar pattern to Update, but instead of being called frame by frame, it follows the frequency of the physics system [106]. Physics calculations in Unity are performed in a fixed time interval, a default of 0.02 seconds between calls, which can be accessed through *Time.fixedDeltaTime*. All behaviour involving the physics system should be executed here instead of the standard Update method.

Figure 4.5 presents an extensive flowchart featuring MonoBehaviour's order of execution functions during the lifecycle of a script. The reader may also examine said flowchart at docs.unity3d.com/Manual/ExecutionOrder.html.

MonoBehaviour was a fundamental structure in building the character in Unity. To begin with, it is used for the facade Character class described in section 3.3. Making Character inherit from MonoBehaviour allows us to attach it to a character's game object as a component and start up its whole behaviour when the scene is loaded, whether or not it runs frame-by-frame.

The purpose of Character is in fact to initialize each class of the architecture, first separately in Awake(), then solve dependencies between each other in Start(). Furthermore, by having a reference to each class, we provide access to said classes and their properties through the game engine, allowing game designers to modify characters' parameters without resorting to any code.

These properties become accessible due to Unity's serialization process. Serialization automatically transforms data structures into a format that Unity can store and reconstruct later [107]. All public variables, along with private properties followed by a *SerializeField* attribute (among other conditions), are automatically serialized, which makes them accessible directly through Unity's inspector window. Serialization is not limited to MonoBehaviour components, but also works with scripted assets such as scriptable objects —described in section 4.4—.

Frame-by-frame operations are kept to a minimum for the shake of computational cost. All scripts that need not be MonoBehaviour were implemented as basic C# classes to avoid unnecessary update calls, as even when empty they entail a great cost in performance [108]. Besides Character, only controller scripts and CharacterStateMachine are MonoBehaviour. The controller classes were made MonoBehaviour to facilitate attaching a specific type of controller to a given character. CharacterStateMachine, on the other hand, actually needs to run frame-by-frame calls for movement and stamina regeneration. Nevertheless, its update behaviour is always disabled when entering states that do not use frame-by-frame calls, such as the knockout state.

4.3.1.2 Animator

Mecanim is Unity's fundamental animation system. Using imported assets from third party tools such as Max or Maya, it allows users to define an object's animation behaviour. That is, to define the sequence of animations that will play at any given time and the process from which one animation will transition to another [109].

This system is, in fact, a finite state machine. Mecanim defines a set of states known as *motions* and a set of transitions between these states. Using an Animator Controller (an interface to the Mecanim system, see Figure 4.6), the user may create, modify or delete any of these states. They may also create transitions between said states using conditions given by numerical or boolean parameters, or by establishing a specific exit time after which a state must transition [110].

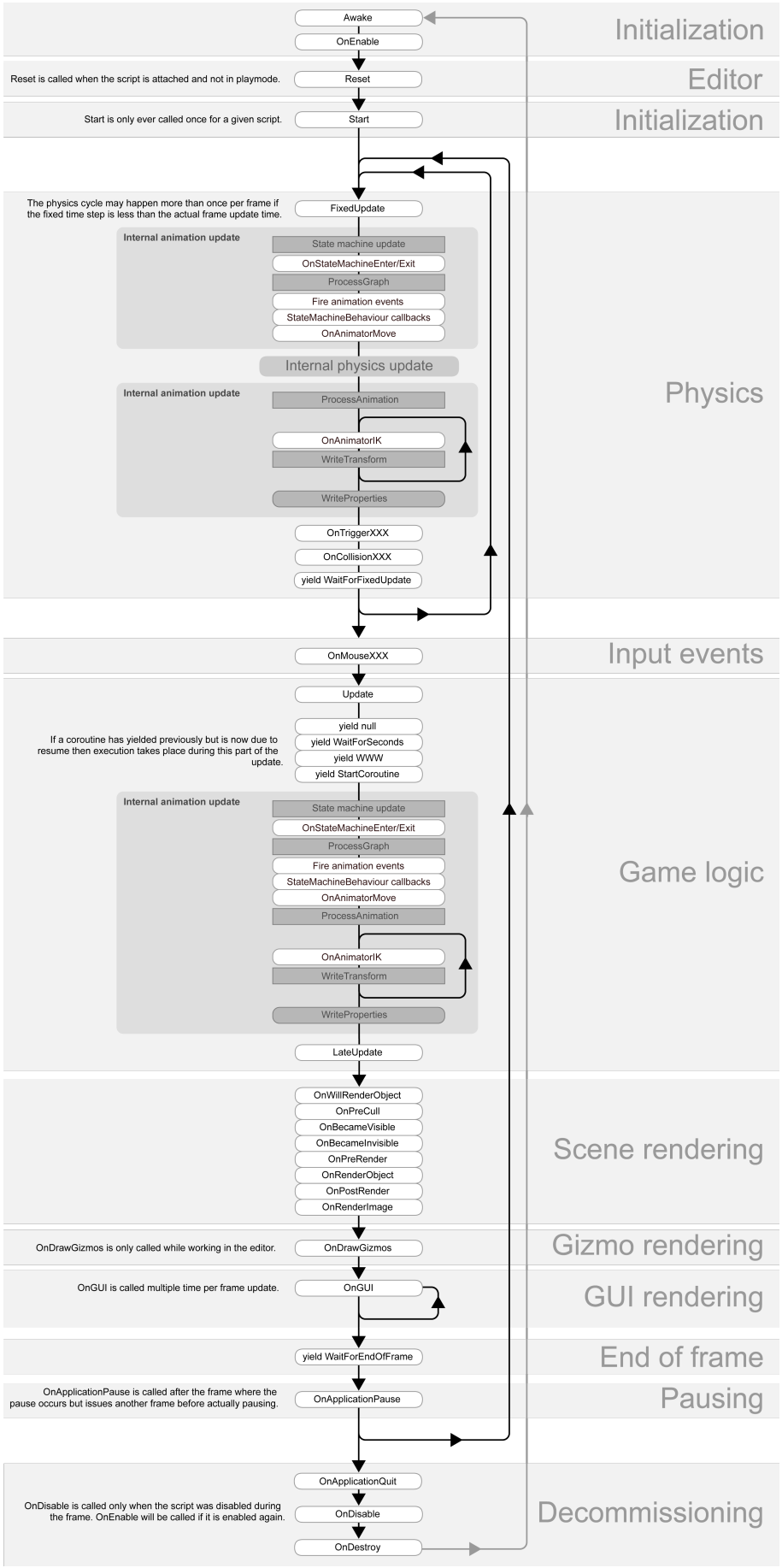


Figure 4.5 – MonoBehaviour's order of execution.

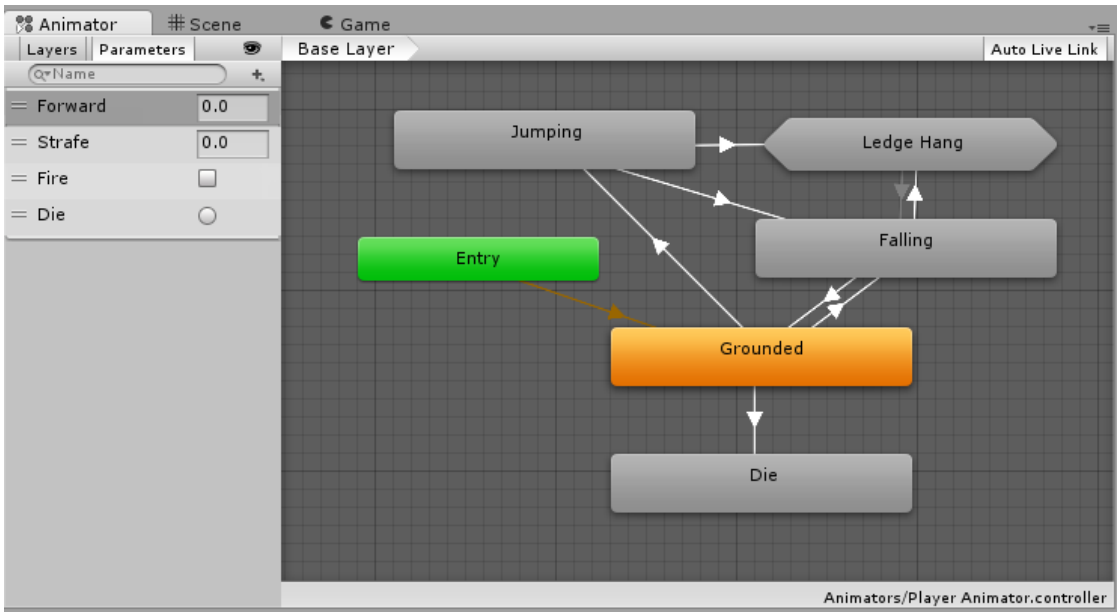


Figure 4.6 – Sample animator controller.

Motions define two types of animation states: animation clips and blend trees [111]. Animation clips are self-explanatory: animations previously made in third party software, which are converted into animation clips when imported into Unity. Blend trees, on the hand, present a framework with which to *blend* animations given numerical parameters [112]. When a developer creates a blend tree, they define a set of motions (animation clips or other blend trees) which will be blended given one or more parameters. These motions are then assigned a value for each of the blend tree's parameters.

The most common example of this is blending walking and running animations according to the character's speed (see Figure 4.7). Consider a blend tree where a walking motion is played at 0 speed, whereas the full running animation takes place at 1 speed. As the character's speed changes, the walking animation gradually blends with the running animation, going from a slow run at the beginning to a sprint when at full speed.

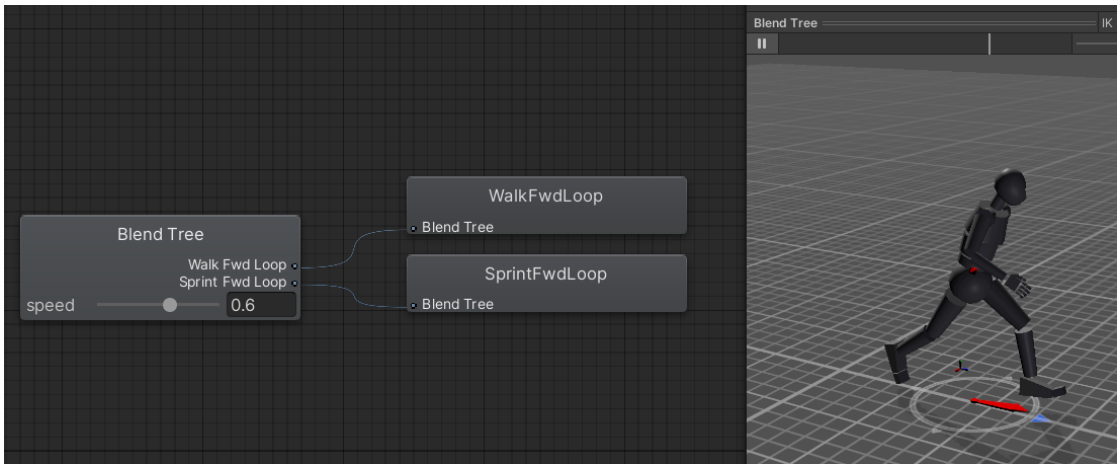


Figure 4.7 – Walking and running blend tree.

Inserting a Mecanim system to a scene is a simple as attaching an Animator component to the desired game object, then assigning the avatar to be animated along with an animator controller (see Figure 4.8). Once added, this component can be accessed through scripts to modify

its parameters in runtime and thus switch between its states.

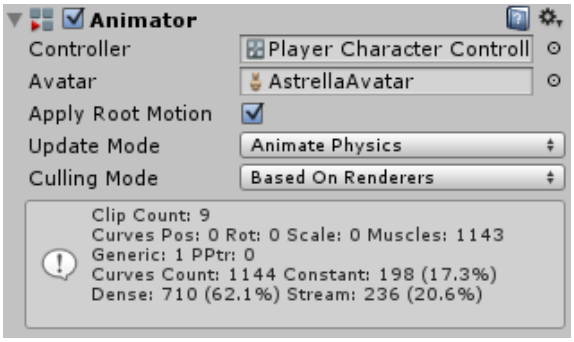


Figure 4.8 – Sample animator component.

Carnem-Levare's mecanim is designed so that all states smoothly transition directly into each other using the generic state known as *AnyState* (see Figure 4.9). As the name suggests, transitions defined from *AnyState* effectively function as transitions from all states to the entering motion, allowing animator designers to avoid a spider webs of transitions. We used booleans to specify the current state in which a character finds itself in (therefore, only one can be true at a time), along with triggers for transitioning into specific moves, which are booleans that turn false after being read.

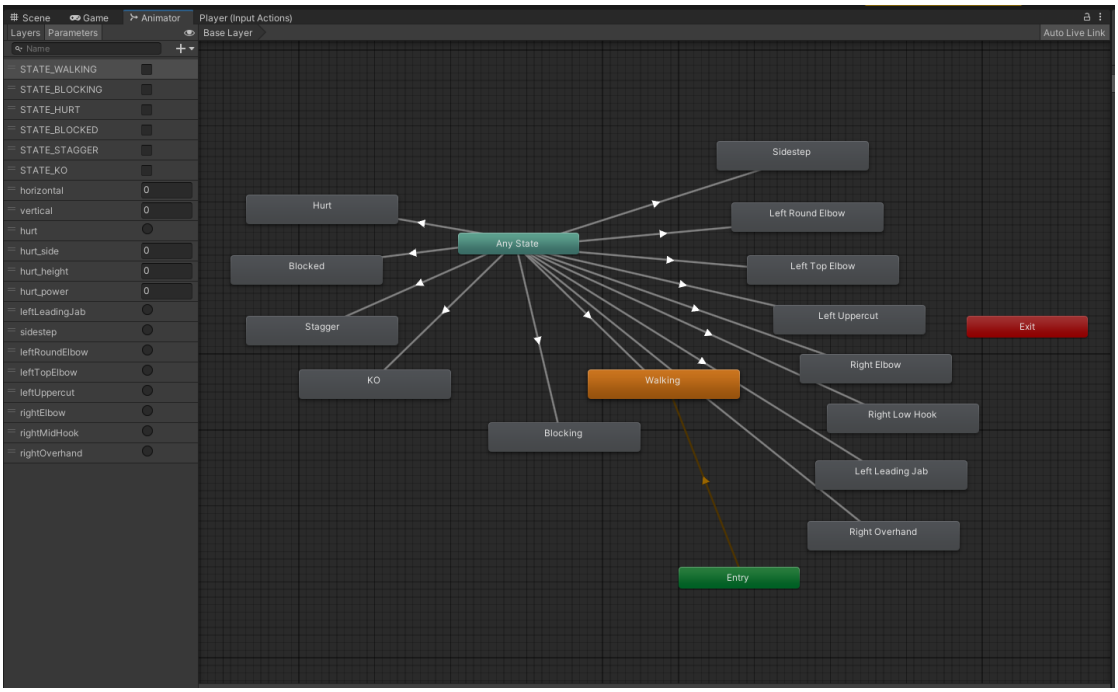
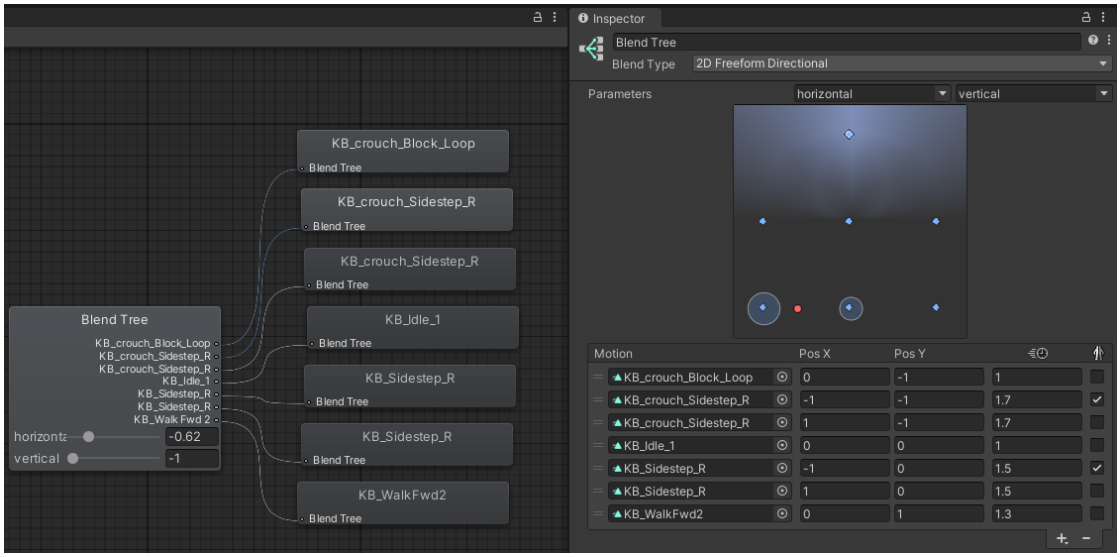


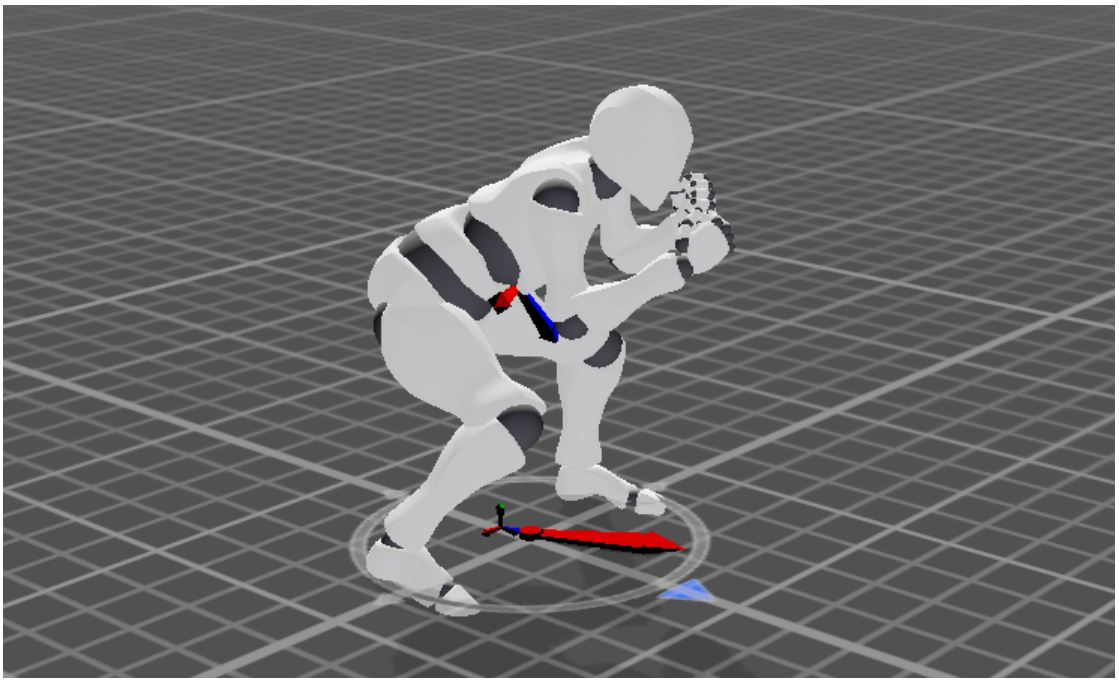
Figure 4.9 – Carnem-Levare's character animator controller.

Furthermore, we used numerical parameters for seven blend trees: walking, blocking, hurt, blocked, stagger and knockout. Walking and blocking blend their different movement animations by tracking the user's directional input (be it a player's stick or an AI command). The most notable example of this is the blocking blend tree (see Figure 4.10) which tracks vertical position to blend crouching and standing stances, along with an horizontal parameter to blend between strafing and idle animations.

The hurt blend trees, on the other hand, uses the information given by the hitbox to establish the side and height from which the character has received an attack. That involves three



(a) Blend Tree view, given random horizontal and vertical values.



(b) Animation resulting from blend tree.

Figure 4.10 – Character's blocking blend tree.

possibilities for the side (left, middle or right), and two possibilites for height (low or high).

These parameters are automatically adjusted by the class CharacterAnimator —now known as CharacterAnimation— previously described in section 3.6. Using the reference to the animator component injected by the Character class during its initialization, CharacterAnimation modifies the animator controller's parameters to the corresponding value when notified by the state machine of having entered a specific state. It is also notified by CharacterMovement to update walking and blocking blend trees with the current movement values.

4.3.1.2.1 Root Motion

Root motion in 3D animation refers to the movement of a character's root or base joint in its animation rig. The root joint is typically located at the base of the spine or pelvis and serves as the point from which all other joints and bones in the character's skeleton are connected. When a character moves, the root joint is responsible for carrying the overall motion of the character in the 3D space.

Animations imported into Unity with defined root motion will already displace the selected rig throughout the environment. Therefore, it is not necessary to implement movement via scripting and it is in fact preferable to avoid it, as otherwise the animation may not match its defined movement speed (for example, a humanoid character's steps not matching with the distance traveled).

Therefore, CharacterMovement no longer handles the speed at which a character walks. Instead, its MoveCharacter methods use directional input as a target to which its internal direction is interpolated towards. This way, changes in walking and blocking blend trees are smoothed out, and we can define factors such as how quickly the player can crouch or move to the sides when blocking.

4.3.1.2.2 Animation Events

Though not directly related with Mecanim, another feature which was fundamental to this implementation were *animation events*. Animation events are used to call functions at specific points of time in a given animation clip, when said animation is played by the Mecanim system [113]. Through Unity's animation window[1], a developer may assign to an animation event the object implementing the function to call and one parameter in the form of a float, string, int or object reference.

The main use of this feature is, of course, frame data. Animation events were added for Move's InitMove, ActivateMove, DeactivateMove and FinishMove methods. This allowed us to trigger each of these functions at precisely the timings specified by the start up, active and recovery phases.

[1] Or via scripting, as it was done for this prototype. Check github.com/ggarredondo/Carnem-Levare for more information.

4.3.1.3 Collider

Colliders are components that define the shape of a game object's physical boundaries for the purpose of collision detection during gameplay. If paired with a Rigidbody, a game object with both components will put its motion under the control of Unity's physics engine and react to collisions with other colliders [114] [115]. Using the inspector window, developers may define a collider's position in relation to the game object it is attached to, along with its size and height.

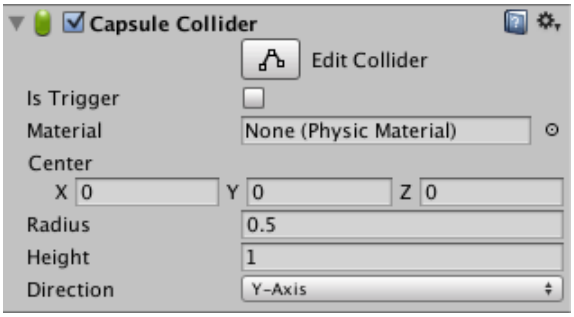


Figure 4.11 – Sample capsule collider component in inspector.

Colliders in Carnem-Levare served three different purposes: to be used as pushbox, hitbox and hurtbox (see Figure 4.12). Pushboxes involve the typical collider behaviour: to stop the object from passing through another collider. By adding a collider and a rigidbody component to each character game object, we made sure that neither the player nor the enemy could walk through each other.

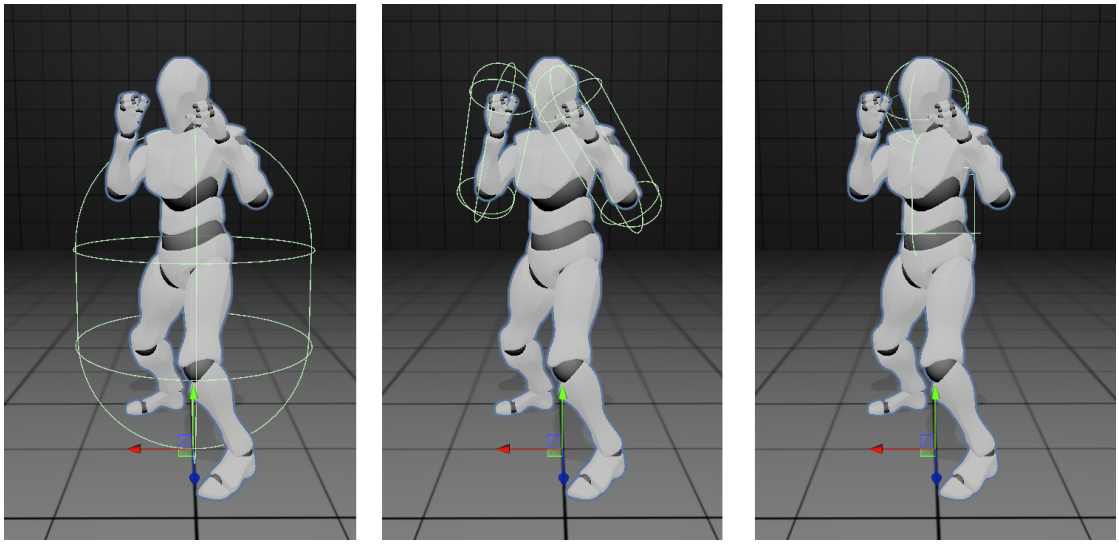


Figure 4.12 – Carnem-Levare character's pushbox, hitboxes and hurtboxes, respectively.

Hitboxes and hurtboxes were explained in detail in subsection 3.5.3. For this prototype, they were implemented as their own game object inside the character's model hierarchy, with a singular collider component (no rigidbody). Using Unity's layer-based collision detection [116], we were able to define specific layers in the physics' collision matrix (see Figure 4.13) to ensure that hitboxes only interacted with hurtboxes and viceversa.

Furthermore, we used collider's Is Trigger property. When enabled, collider's OnTrigger-Enter method is called everytime the object interacts with another collider. We overrided hurtbox's OnTriggerEnter to grab the hitbox's information as it was designed in the architecture.

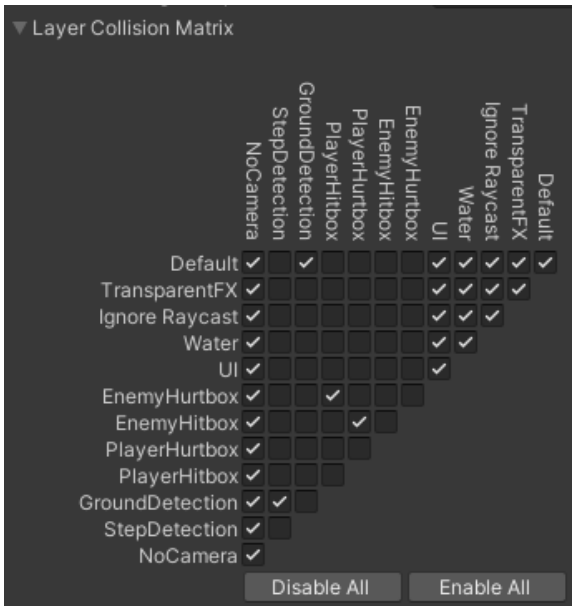


Figure 4.13 – Carnem-Levare's layer collision matrix.

4.3.1.4 Rigidbody

As it was previously stated, adding a Rigidbody component to a game object will put its motion under the control of Unity's physics engine, causing it to be affected by forces such as gravity and react to collisions if paired with a collider [115].

Moreover, Rigidbody has an scripting API which allows developers to control the character's physics and apply forces by code. The main use for this API was in CharacterMovement class.

To begin with, instead of rotating the character by adjusting its Transform component, targeting was implemented by modifying its Rigidbody's own rotation values. If the method LookAtTarget is then adequately called in FixedUpdate —for every state that involves targeting—, we ensure that the tracking movement does not skip physics simulation.

Additionally, attack knockback was implemented using Rigidbody's AddForce function. CharacterMovement's PushCharacter receives the attack's directional force through a hitbox and applies it using said method as an impulse to its physics body, pushing them back in relation to the attacker.

4.4 Scriptable Object

Scriptable objects are data containers that, instead of being placed in the scene, are saved as assets in the project to be eventually used by one or several game objects in the application [117]. The purpose of this class is to save memory by creating containers of information, independent of class instances, that would otherwise be duplicated between game objects. Instead, all objects requiring said information may access it by reference using these containers.

Developers may create scripts inheriting from ScriptableObject to define their own data structures to be saved in the project. Similarly to components, scriptable objects allow serialized properties to be modified through the inspector window, no coding involved. Which makes scriptable objects the fundamental structure for moves.

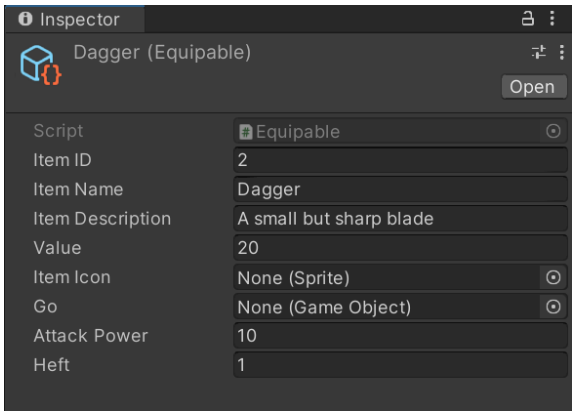


Figure 4.14 – Sample scriptable object in inspector.

Moves inherit from scriptable object and, beside implementing frame data related methods, it serializes all meaningful variables for game designers to comfortably adjust from Unity's inspector window. The scriptable objects instantiated however will always be subclasses of Move —such as AttackMove—, as Move remains an abstract class.

Moreover, AIStateMachine is also implemented as a subclass of ScriptableObject. Besides fulfilling the same purpose of allowing game designers to easily balance the game's features (in this case, by serializing variables related to the AI's behaviour), we facilitate assigning a specific behaviour to an AI controller, making it as simple as dragging the scriptable object to the AI controller's component in the scene.

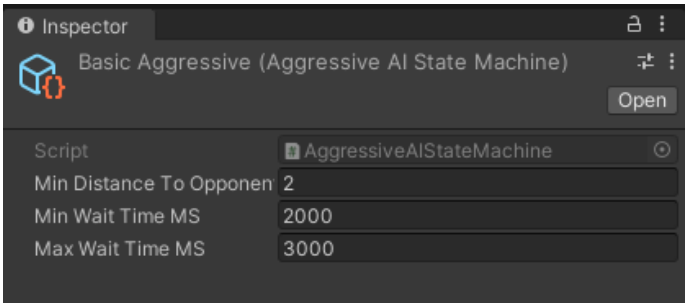


Figure 4.15 – Sample aggressive AI behaviour in inspector.

4.5 C# Events

In section 3.4, we described the Observer pattern as the fundamental design element to implement a Model-View-Controller architecture with minimal coupling. While this pattern was indeed used for implementation, instead of creating specific observer and subject classes we took advantage of a C# feature known as *events*.

Similarly to the Observer base design, events allows classes to notify other classes when a certain event takes place, in such a way that the notificator has no knowledge of the receiver. However, instead of implementing a subject interface with notify methods, the subject creates an event using *delegates* as event handlers.

Delegates are a type of variable that represents references to methods with a particular parameter list and return type [118]. They are often used to pass methods as arguments to other methods, and event handling is no more than invoking methods through delegates.

Once a delegate is created, the subject class may define an event by declaring a variable using said delegate and the *event* keyword. When something of interest happens, the subject may call the event's invoke method. All observers subscribing to this event will provide a function to be called when said event is invoked, successfully handling the notification [119].

Here is an example in code:

```
public delegate void EventHandler(object parameter);

public class Subject
{
    private Object myObject;
    public event EventHandler myEvent;

    public void Function() {
        // ... do something important
        myEvent?.Invoke(myObject);
    }
}

public class Observer
{
    private Subject mySubject;

    public Observer() {
        // initialization
        mySubject.myEvent += HandleNotification();
    }

    public void HandleNotification() {
        // notification is handled
    }
}
```

Besides the use of events, the implementation is no different to the software architecture designed in the previous chapter. Controller and Hurtbox implement events which all states subscribe to in their Enter function, providing methods to transition to another state or to perform a certain action. States then implement their own events for the view to subscribe and use to represent the model with animation, sound and visual effects.

Same architecture but with the advantage of avoiding the monumental indirection that an observer and subject interfaces would entail, considering half the classes in the design needed to extend them. Along with cleaner and more maintainable code overall.

Chapter 5

Conclusion

This chapter brings an end to the overarching process of research, design and implementation that involved this dissertation. Its purpose is to review its development and reflect on the results, whether they were ill or prolific. We may then establish a future perspective for ourselves, Parry Mechanics, founders and developers of Carnem-Levare.

5.1 Summary

Impelled by the promise of an ever-growing industry and the emergence of potentially marketable niches within, we set out with the objective of starting development and documenting the first steps of an indie singleplayer game titled Carnem-Levare. Though it was not only the prospect of profit which drove us, but a deeper, creative inquisitiveness to make our ideas a reality and set an example of how engineering and art can merge to offer an invaluable, fulfilling experience.

Considering gameplay as the primary provider of that experience, this dissertation's intent was to research the software design and implementation that a particular gameplay specification would demand, showcasing the patterns and tools that the reader could make use of for their own project. For that purpose, we established the sub-objectives of defining a game design document, typical in the production of a videogame, from which to derive the software requirements and systems that, once designed and implemented, would compose Carnem-Levare's first prototype. And we achieved these objectives.

We established a focused but detailed game design document which defined the core gameplay to be implemented, describing the environment in which the game was going to take place, the actions the player would have available and the opponents they would have to face.

The environment described was simple enough, an empty enclosed space in order to avoid distractions from the fight. Actions, however, were the gameplay's core, the tool with which the player would express themselves and feel the game. They had to be designed with deliberation and methodology, following industry design standards —such as attacks' frame data— while also providing an innovative experience from the conventional action game. Enemies were not far behind in their significance to gameplay, as their behaviour and characteristics directly impact the viable strategies to defeat them, delimiting player expression and thus also affecting the experience.

From this game design document, we were able to extrapolate the many processes and tasks involved in the interaction system we intended to develop. A character needed attributes to state its health, its stamina, its strength. It needed animations to move, and to attack. It needed to keep track of its current state and handle its behaviour when attacking, blocking or hurting. It needed to respond to input and artificial intelligence.

It soon became apparent that a clean, sustainable architecture was essential for such a complex system to be implemented efficiently and without bugs. To begin with, the whole architecture was designed following SOLID's five principles for readable, flexible and maintainable

architecture. Were it not for these principles, the final result would have been lacking in terms of usability and scalability. Furthermore, software design patterns such as Model-View-Controller and Observer were used to minimize coupling and organize the final architecture in sets of related classes. All of this resulted in a robust, efficient design, easy to both understand and implement while accounting for all requirements established in the game design document.

Consequently, implementation proved simple enough to execute. Using a popular, powerful yet intuitive game engine such as Unity, allowed us to easily learn the ropes of its features and code the architecture without setbacks. Moreover, we were able to further improve the design in a few areas thanks to Unity and C#'s particular quirks.

Overall, the project was a success and we achieved what we set out to do. We managed to design and implement a software system on the basis of a gameplay specification, documenting the whole process for future game developers to use as reference. Nevertheless, the commercial aspect was regrettably left behind. Researching and engineering proved overwhelming enough to eclipse further investigating any business opportunity which, frankly, would have not been specially relevant considering the nature of the project —to make a prototype—.

Therefore, we can be proud and hope that this dissertation will shed some light on the process of game development and the self-obscured, parsimonious industry of secrets and non-disclosure agreements that videogames tend to be.

5.2 Future work

Following a typical game development process, we could consider this dissertation's work to be an early stage of production for Carnem-Levare. Even though the resulting software was a prototype, it was designed and implemented in such a way that it could be easily expanded upon, providing a framework from which game designers and artists could begin introducing their work directly into a functional game without coding.

The objective is thus to continue development in order to eventually release Carnem-Levare as a competitive, fully-fledged commercial videogame. We will have to go deeper into the current production stage and expand on what we already built to craft the content the game is still missing.

To achieve this purpose, we will first have to expand the team and integrate the artists which our team of only engineers desperately needs. Following our vision, we will need concept artists to define the game's art direction and its overall aesthetics. We will also need modelers capable of creating 3D characters, environment and props using concept art as a basis. Lastly, we will need animators to bring said characters to life, creating attacks and moves that support the combat design.

Furthermore, we will continue improving and expanding on the current software architecture. Even though this dissertation's design proved to be successful overall, there is always room for improvement and any new mechanic, if elaborate enough, will probably need new systems to be implemented.

Expanding the team and further improving the software, however, are not realistic goals on their own from a business perspective. We will need financial support to employ the team members we lack and to make of our work a paid job. For that purpose, we will resume the

commercial research which was halted during the development of this prototype and investigate likely sources of income with which to fund Carnem-Levare's development, such as crowdfunding campaigns or contacting publishers.

With proper planning, hard-work, and the right amount of luck, we hope Carnem-Levare will see the light of day and be released to become the commercial success we envision. This way, we may embrace our dream job and continue developing games on our own terms as an indie studio.

Chapter 6

Epilogue

Ere the making of this text,
we embarked on a unlikely quest.
Unlikely as it was unexpected,
we trudged through early morn.
Long and lone was the road,
where town and cities torn.
Yet from swirling dirt and dust,
beneath bashful shine and sun,
unlikely company was formed.
Long and curled were his horns,
deep and knowing his eyes.
Ill and wicked, fateful devil,
eager to curse our pretty sky.
"There is but one chance,
for me not to feast on your souls.
The deal is sealed, your contract is due,
to make a game is your goal.
Any kind will suffice,
any style will do.
There is but one condition,
to make the greatest game in the world!"
His voice made knots of our throats,
his words made strife of our stroll.
Yet our hearts were unmoved,
yet our wit powered through.
Our choice was final,
our fate undone.
The devil we answered:
"Together we shall build our throne."
An idea struck,
like lightning to our minds.
Designed and crafted it soon was,
undoing our binds.
The devil was stunned,
his tongue tied to the tune of his taunt.
"Who would have known,
you were the mechanics of our time?"
Those were his words,
and soon the beast was gone.
No lies were spoken,
in the writing of this song.
But if we were to tell you,
we hold no memory of our work.
Consider this piece,
as a tribute to our feat.
A most wonderful tale,
this dissertation's final treat.

Bibliography

[1] S. Swink, "Game feel: A game designer's guide to virtual sensation," in CRC Press, 2008, ISBN: 978-0123743282.

[2] C. Lindley, L. Nacke, and C. Sennersten, "Dissecting play – investigating the cognitive and emotional motivations and affects of computer gameplay," in UNIV WOLVERHAMPTON, 2008, *"The experience of gameplay is one of interacting with a game design in the performance of cognitive tasks, with a variety of emotions arising from or associated with different elements of motivation, task performance and completion."*, ISBN: 978-0-9549016-6-0.

[3] S. Björk and J. Holopainen, "Patterns in game design," in Charles River Media, 2005, ISBN: 978-1-58450-354-5.

[4] F. Dutton, *What is indie?* Apr. 2012. [Online]. Available: https://www.eurogamer.net/what-is-indie.

[5] R. Carroll, *Indie innovation?* Jun. 2004. [Online]. Available: https://web.archive.org/web/20090615003836/http://www.gametunnel.com/indie-innovation-article.php.

[6] M. Gnade, *What exactly is an indie game?* Jul. 2010. [Online]. Available: https://web.archive.org/web/20130927182457/http://www.indiegamemag.com/what-is-an-indie-game/.

[7] Wiktionary, *Glossary of fighting games*, May 2023. [Online]. Available: https://en.wiktionary.org/wiki/Appendix:Glossary_of_fighting_games.

[8] Capcom, *Hour 9: The basics of attacking: The basics of attack composition.* [Online]. Available: https://game.capcom.com/cfn/sfv/column/131432.

[9] E. Adams and A. Rollings, "Fundamentals of game design," in Prentince Hall, 2006, ISBN: 9780131687479.

[10] HelpLama, *Game industry usage and revenue statistics 2023*, Jul. 2023. [Online]. Available: https://helplama.com/game-industry-usage-revenue-statistics/.

[11] B. Jovanovic, *Gamer demographics: Facts about the most popular hobby*, May 2023. [Online]. Available: https://dataprot.net/statistics/gamer-demographics/.

[12] Steamworks-Development, *2021 year in review*, Mar. 2022. [Online]. Available: https://store.steampowered.com/news/group/4145017/view/3133946090937137590.

[13] Steamworks-Development, *Steam tags.* [Online]. Available: https://partner.steamgames.com/doc/store/tags.

[14] Games-Stats, *Steam tags stats.* [Online]. Available: https://games-stats.com/steam/tags/.

[15] Steam, *Top sellers.* [Online]. Available: https://store.steampowered.com/search/?filter=topsellers.

[16] S. A. A. Museum, *Art of videogames*, https://americanart.si.edu/exhibitions/games.

[17] Metacritic, *Journey*, Mar. 2012. [Online]. Available: https://www.metacritic.com/game/playstation-3/journey.

[18] Metacritic, *Disco elysium*, Oct. 2019. [Online]. Available: https://www.metacritic.com/game/pc/disco-elysium.

[19] Metacritic, *Papers, please*, Aug. 2013. [Online]. Available: https://metacritic.com/game/pc/papers-please.

[20] Discord-Inc, *Discord official webpage.* [Online]. Available: https://discord.com/.

[21] Github-Inc, *Github official webpage.* [Online]. Available: https://github.com/.

[22] K. Schwaber, "Agile project management with scrum," in Microsoft Press, Feb. 2004, ISBN: 978-0-7356-1993-7.

[23] B. Bates, "Game design (2nd ed.)," in Thomas Course Technology, 2009, ISBN: 978-1-58450-580-8.

[24] M. E. Moore and J. Novak, "Game industry career guide," in Cengage Learning, 2010, ISBN: 978-1-4283-7647-2.

[25] T. Slopper, *Sample outline for a game design document*, Aug. 2015. [Online]. Available: https://www.sloperama.com/advice/specs.html.

[26] K. Oxland, "Gameplay and design," in 2004, ISBN: 0-321-20467-0.

[27] C. Harris, *Punch-out!! review, "It's also one of the most difficult action games to master because of its dependence on pattern recognition and almost split-second response on the controller."*, May 2012. [Online]. Available: https://www.ign.com/articles/2009/05/15/punch-out-review.

[28] B. Tyrrel, *Sekiro: Shadows die twice review, "Sekiro places emphasis on getting better with what you can do rather than looking for another weapon or piece of armor."*, Mar. 2019. [Online]. Available: https://www.ign.com/articles/2019/03/21/sekiro-shadows-die-twice-review.

[29] A. Wood, *What the hell is a souls-like?* Feb. 2022. [Online]. Available: https://www.gamesradar.com/what-is-a-souls-like-developers-explain/.

[30] E. Makuch, *Top 10 best-selling games of 2022 in the us*, Jan. 2023. [Online]. Available: https://www.gamespot.com/articles/top-10-best-selling-games-of-2022-in-the-us/1100-6510556/.

[31] T. Ivan, *Npd has revealed the best-selling games of 2022 in the united states*, Jan. 2023. [Online]. Available: https://www.videogameschronicle.com/news/npd-has-revealed-the-best-selling-games-of-2022-in-the-united-states/.

[32] N. Spector, *This is how much playing videogames will cost you over your lifetime*, Aug. 2022. [Online]. Available: https://www.gobankingrates.com/money/wealth/this-is-how-much-playing-video-games-will-cost-you-over-your-lifetime/.

[33] A. Barovic, *7 best budget gaming phones in 2023*, Apr. 2023. [Online]. Available: https://leaguefeed.net/best-budget-gaming-phones-2023/.

[34] Amazon, *Nintendo switch lite - blue*. [Online]. Available: https://www.amazon.com/Nintendo-Switch-Lite-Blue/dp/B092VT1JGD/ref=sr_1_1?crid=2BE0HC40XD453&keywords=switch+lite&qid=1689076351&sprefix=switch+l%2Caps%2C224&sr=8-1.

[35] F. Locknear, *Average cost of a gaming pc*, 2023. [Online]. Available: https://thecostguys.com/gaming/average-cost-of-a-gaming-pc.

[36] M. Pawlik, *How much does a decent gaming pc cost?* Feb. 2023. [Online]. Available: http://pcgamingadvice.com/how-much-does-a-decent-gaming-pc-cost/.

[37] J. Roach, *A complete, chronological history of the catastrophic gpu shortage*, Jul. 2021. [Online]. Available: https://www.digitaltrends.com/computing/catastrophic-gpu-shortage-a-chronological-history/.

[38] Firespikez, *No items, fox only, final destination*, Aug. 2009. [Online]. Available: https://knowyourmeme.com/memes/no-items-fox-only-final-destination.

[39] Fran and Peer, *Super smash bros. melee guide*, Dec. 2001. [Online]. Available: https://web.archive.org/web/20120629220208/http://guides.ign.com/guides/16387/index.html.

[40] Martin, *Super smash bros. melee review*, Dec. 2001. [Online]. Available: https://web.archive.org/web/20080506214334/http://www.gamefreaks365.com/review.php?artid=127.

[41] C. Stark, *7 questions with the creator of 'super smash bros.' What I'm more going for is something like a party game, something you can play on a whim and have fun as all sorts of things take place onscreen.*, Nov. 2014. [Online]. Available: https://mashable.com/archive/super-smash-bros-sakurai-interview.

[42] SmashWiki, *Tournament rulesets (ssbm)*, Jul. 2010. [Online]. Available: https://www.ssbwiki.com/Tournament_rulesets_(SSBM).

[43] u/OkRing7470, *Opinions on environmental kills?* 2021. [Online]. Available: https://www.reddit.com/r/forhonor/comments/rujs6p/opinions_on_environmental_kills/.

[44] u/pls-dont-judge-me, *Environmental kills are lame.* 2017. [Online]. Available: https://www.reddit.com/r/forhonor/comments/5u5kq9/environmental_kills_are_lame/.

[45] T. Kelly, *Action - what games are*, 2014. [Online]. Available: https://www.whatgamesare.com/action.html.

[46] Nintendo, *Super punch-out!!* Oct. 1994. [Online]. Available: https://www.nintendo.co.jp/clvs/manuals/common/pdf/CLV-P-SAAXE.pdf.

[47] SmashWiki, *Spacing*, Jan. 2009. [Online]. Available: https://www.ssbwiki.com/Spacing.

[48] saviour4ever, *What is "spacing"?* 2011. [Online]. Available: https://gamefaqs.gamespot.com/boards/975212-super-street-fighter-iv/58901660.

[49] SuperCombo-Wiki, *Street fighter alpha 3/ken*, Aug. 2022. [Online]. Available: https://wiki.supercombo.gg/w/Street_Fighter_Alpha_3/Ken.

[50] Sloclap, *Sifu game official website.* [Online]. Available: https://www.sifugame.com/.

[51] ONGBAL, *[sifu] 'the club' no death playthrough*, Feb. 2022. [Online]. Available: https://youtu.be/QpdkTAG0Cng?si=JqCFSWKKiesJxSfQ.

[52] Fextralife, *Elden ring's controls*, Apr. 2023. [Online]. Available: https://eldenring.wiki.fextralife.com/Controls.

[53] Fextralife, *Elden ring's dodge*, Mar. 2023. [Online]. Available: https://eldenring.wiki.fextralife.com/Dodging.

[54] J. Jay, *Mike tyson interview*, Aug. 2013. [Online]. Available: http://www.doghouseboxing.com/On-The-Ropes-Boxing-Radio/OTR-New-0829ii13-Mike-Tyson.htm.

[55] jezzamundo, *Average heavyweight height*, Mar. 2016. [Online]. Available: https://boxrec.com/forum/viewtopic.php?t=199245.

[56] TheModernMartialArtist, *Tyson's aggressive peekaboo head movement explained.* [Online]. Available: https://www.youtube.com/shorts/jIlqNLVGbHY.

[57] SweetScientist, *The science of mike tyson and elements of peeek-a-boo*, Feb. 2014. [Online]. Available: https://web.archive.org/web/20150925144552/http://www.sugarboxing.com/the-science-of-mike-tyson-and-elements-of-peek-a-boo-part-ii/.

[58] Fextralife, *Sekiro's posture*, Jul. 2023. [Online]. Available: https://sekiroshadowsdietwice.wiki.fextralife.com/Posture.

[59] Capcom, *How to play street fighter 5.* [Online]. Available: https://www.streetfighter.com/5/en-us/how-to-play/index.html.

[60] Unity-Technologies, *Interactions.* [Online]. Available: https://docs.unity3d.com/Packages/com.unity.inputsystem@1.0/manual/Interactions.html.

[61] Sloclap, *Absolver store page.* [Online]. Available: https://store.steampowered.com/app/473690/Absolver/.

[62] Sloclap, *Absolver official faq.* [Online]. Available: https://steamcommunity.com/app/473690/discussions/0/2579854400733819593/.

[63] Fextralife, *Absolver combat deck*, Aug. 2018. [Online]. Available: https://absolver.wiki.fextralife.com/Combat+Deck.

[64] Sea-Leaf-Dojo, *Frame data basics*, Apr. 2019. [Online]. Available: https://youtu.be/sbyUM5aWKpk?si=NVrHPfJ0X7Oen7dz.

[65] MrBug, *Hour 11: The basics of attacking: Damage part 2.* [Online]. Available: https://game.capcom.com/cfn/sfv/column/131545.

[66] Fextralife, *Elden ring's poise*, Sep. 2023. [Online]. Available: https://eldenring.wiki.fextralife.com/Poise.

[67] Wikidot, *Dark souls' stagger.* [Online]. Available: http://darksouls.wikidot.com/stagger.

[68] K. Graft, *New doom's deceptively simple design*, Apr. 2017. [Online]. Available: https://www.gamedeveloper.com/design/-make-me-think-make-me-move-new-i-doom-i-s-deceptively-simple-design.

[69] G. N. Yannakakis, "Artificial intelligence and games," in Springer, 2018, ch. 5, *"The primary goal of player modeling is to understand how the interaction with a game is experienced by individual players."*, ISBN: 978-3-319-63518-7.

[70] G. N. Yannakakis, "Artificial intelligence and games," in Springer, 2018, ch. 3, *"In other cases, the most important feature of an NPC is its predictability. In a typical stealth game, a large part of the challenge is for the player to memorize and predict the regularities of guards and other characters that should be avoided. In such cases, it makes sense that the patrols are entirely regular, so that their schedule can be gleaned by the player. Similarly, the boss monsters in many games are designed to repeat certain movements in a sequence, and are only vulnerable to the player's attack when in certain phases of the animation cycle."*, ISBN: 978-3-319-63518-7.

[71] Bungie, *The illusion of intelligence*, 2002. [Online]. Available: http://halo.bungie.org/misc/gdc.2002.haloai/talk.html.

[72] B. Vossen, *Enemy design and enemy ai for melee combat systems*, May 2015. [Online]. Available: https://www.gamedeveloper.com/design/enemy-design-and-enemy-ai-for-melee-combat-systems.

[73] BipolarShango, *Playstyles in fighting games*, Jul. 2023. [Online]. Available: https://medium.com/master-of-the-game/how-to-counter-different-playstyles-in-fighting-games-81d128295767.

[74] BipolarShango, *Decision making in fighting games*, Apr. 2022. [Online]. Available: https://medium.com/master-of-the-game/decision-making-in-fighting-games-3a88c8382fa8.

[75] SuperCombo-Wiki, *Pressure and turns*, Sep. 2018. [Online]. Available: https://wiki.supercombo.gg/w/Pressure_and_Turns.

[76] R. Martin, "Clean architecture: A craftman's guide to software structure and design," in Prentince Hall, 2018, ISBN: 9780134494166.

[77] R. Martin, *The single responsability principle*, 2015. [Online]. Available: https://web.archive.org/web/20150202200348/http://www.objectmentor.com/resources/articles/srp.pdf.

[78] R. Martin, *The open-closed principle*, 2015. [Online]. Available: https://web.archive.org/web/20150905081105/http://www.objectmentor.com/resources/articles/ocp.pdf.

[79] R. Martin, *The liskov substitution principle*, 2015. [Online]. Available: https://web.archive.org/web/20150905081111/http://www.objectmentor.com/resources/articles/lsp.pdf.

[80] R. Martin, *The interface segregation principle*, 2015. [Online]. Available: https://web.archive.org/web/20150905081110/http://www.objectmentor.com/resources/articles/isp.pdf.

[81] R. Martin, *The dependency inversion principle*, 2015. [Online]. Available: https://web.archive.org/web/20150905081103/http://www.objectmentor.com/resources/articles/dip.pdf.

[82] geeksforgeeks, *Mvc framework introduction*, 2015. [Online]. Available: https://www.geeksforgeeks.org/mvc-framework-introduction/.

[83] H. Hod, *Mvc in game development*, May 2014. [Online]. Available: https://www.gamedeveloper.com/programming/mvc-in-game-development.

[84] W. Werner, *Mvc game design pattern*, 2020. [Online]. Available: https://github.com/wesleywerner/mvc-game-design.

[85] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Elements of reusable object-oriented software," in Addison-Wesley, 1994, ISBN: 0-201-63361-2.

[86] WikiWikiWeb, *Hollywood principle*, Aug. 2013. [Online]. Available: https://wiki.c2.com/?HollywoodPrinciple.

[87] R. Nystrom, "Game programming patterns," in Genever Benning, 2011, ch. 2, ISBN: 0990582906.

[88] T. Tamasi, *Why does high fps matter for esports?* Dec. 2019. [Online]. Available: https://www.nvidia.com/en-us/geforce/news/what-is-fps-and-how-it-helps-you-win-games/.

[89] Unity-Technologies, *Introduction to collision*, Oct. 2023. [Online]. Available: https://docs.unity3d.com/Manual/CollidersOverview.html.

[90] J. Wang, "Formal methods in computer science," in CRC Press, 2019, ISBN: 978-1-4987-7532-8.

[91] R. Nystrom, "Game programming patterns," in Genever Benning, 2011, ch. 3, ISBN: 0990582906.

[92] Unity-Technologies, *Time.deltatime*, Jan. 2023. [Online]. Available: https://docs.unity3d.com/ScriptReference/Time-deltaTime.html.

[93] S. Russell and P. Norvig, "Artificial intelligence: A modern approach," in Prentice Hall, 2003, ISBN: 0-13-790395-2.

[94] G. N. Yannakakis, "Artificial intelligence and games," in Springer, 2018, ch. 2, *"In this section we discuss the first, and arguably the most popular, class of AI methods for game development. Finite state machines, behavior trees and utilitybased AI are ad-hoc behavior authoring methods that have traditionally dominated the control of non-player characters in games."*, ISBN: 978-3-319-63518-7.

[95] Unity-Technologies, *Unity official webpage*. [Online]. Available: https://unity.com/.

[96] M. Dealessandri, *What is the best game engine: Is unity right for you?* Jan. 2020. [Online]. Available: https://www.gamesindustry.biz/what-is-the-best-game-engine-is-unity-the-right-game-engine-for-you.

[97] S. Axon, *Unity at 10: For better —or worse— game development has never been easier*, Sep. 2016. [Online]. Available: https://arstechnica.com/gaming/2016/09/unity-at-10-for-better-or-worse-game-development-has-never-been-easier/.

[98] Unity-Technologies, *Unity user manual*. [Online]. Available: https://docs.unity3d.com/Manual/index.html.

[99] S. K. Arora and R. Johns, *Unity vs unreal: Which game engine should you choose?* Mar. 2023. [Online]. Available: https://hackr.io/blog/unity-vs-unreal-engine.

[100] Unity-Technologies, *Gameobject*, Nov. 2023. [Online]. Available: https://docs.unity3d.com/ScriptReference/GameObject.html.

[101] Unity-Technologies, *Introduction to components*, Nov. 2023. [Online]. Available: https://docs.unity3d.com/Manual/Components.html.

[102] Unity-Technologies, *Monobehaviour*, Nov. 2023. [Online]. Available: https://docs.unity3d.com/ScriptReference/MonoBehaviour.html.

[103] Unity-Technologies, *Monobehaviour's awake function*, Nov. 2023. [Online]. Available: https://docs.unity3d.com/ScriptReference/MonoBehaviour.Awake.html.

[104] Unity-Technologies, *Monobehaviour's start function*, Nov. 2023. [Online]. Available: https://docs.unity3d.com/ScriptReference/MonoBehaviour.Start.html.

[105] Unity-Technologies, *Monobehaviour's update function*, Nov. 2023. [Online]. Available: https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html.

[106] Unity-Technologies, *Monobehaviour's fixed update function*, Nov. 2023. [Online]. Available: https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html.

[107] Unity-Technologies, *Script serialization*, Nov. 2023. [Online]. Available: https://docs.unity3d.com/Manual/script-Serialization.html.

[108] Microsoft, *Performance recommendations for unity*, Nov. 2022. [Online]. Available: https://learn.microsoft.com/en-us/windows/mixed-reality/develop/unity/performance-recommendations-for-unity.

[109] Unity-Technologies, *Mecanim animation system*, Nov. 2023. [Online]. Available: https://docs.unity3d.com/462/Documentation/Manual/MecanimAnimationSystem.html.

[110] Unity-Technologies, *Animator controller*, Nov. 2023. [Online]. Available: https://docs.unity3d.com/Manual/class-AnimatorController.html.

[111] Unity-Technologies, *Motion*, Nov. 2023. [Online]. Available: https://docs.unity3d.com/ScriptReference/Motion.html.

[112] Unity-Technologies, *Blend trees*, Nov. 2023. [Online]. Available: https://docs.unity3d.com/Manual/class-BlendTree.html.

[113] Unity-Technologies, *Using animation events*, Nov. 2023. [Online]. Available: https://docs.unity3d.com/Manual/script-AnimationWindowEvent.html.

[114] Unity-Technologies, *Collider*, Nov. 2023. [Online]. Available: https://docs.unity3d.com/ScriptReference/Collider.html.

[115] Unity-Technologies, *Rigidbody*, Nov. 2023. [Online]. Available: https://docs.unity3d.com/ScriptReference/Rigidbody.html.

[116] Unity-Technologies, *Layer-based collision detection*, Nov. 2023. [Online]. Available: https://docs.unity3d.com/Manual/LayerBasedCollision.html.

[117] Unity-Technologies, *Scriptableobject*, Nov. 2023. [Online]. Available: https://docs.unity3d.com/Manual/class-ScriptableObject.html.

[118] Microsoft, *C# delegates*, Sep. 2022. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/.

[119] Microsoft, *Events in c#*, May 2022. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/event.

Thank you for reading this Undergraduate Dissertation.