UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Erbium – Third Person Character Creator

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Mikhail Albershtein

Tutor: Albert Albiol, Manuela

2º Tutor: María Victoria Torres Bosch

2019-2020

Abstract

Video game development has always been an issue that a lot of programmers are interested in. With the technologies growing at an incredible speed it became easier to dive into creating video games for everyone. However, with the increased demand for new mechanics and more patronizable games, programmers must make sure that their code supports these new features. There is a lot of educational resources for creating games, however, only a few are teaching how to write quality code.

This thesis is centered on developing a framework that helps programmers to create 3rd person characters for Unity by applying the best practices for coding. To develop the framework, design patterns have been applied, which allows us to build quality software.

Keywords: Video game development, Unity, design patterns, clean code

Resumen

El desarrollo de videojuegos siempre ha sido un área sobre la que mucha gente se ha interesado. Con las tecnologías creciendo a gran velocidad, se ha hecho más fácil la creación de videojuegos. Sin embargo, con el aumento de la demanda de nuevas mecánicas y videojuegos más personalizables, la programación de videojuegos se ha hecho más exigente. Aunque existen una gran cantidad de recursos educativos para ayudar en programación de videojuegos, sólo unos pocos enseñan cómo escribir código de calidad.

Este trabajo final de grado se centra en el desarrollo de un framework que facilita a los progamadores la creación de personajes de 3a persona para Unity aplicando las mejores prácticas para el código. En el desarrollo del framework, se han aplicado patrones de diseño que permiten la obtención de un código de calidad.

Palabras claves: Desarrolo de videojuegos, Unity, patrones de diseño, clean code

Resum

El desenvolupament de videojocs sempre ha sigut un àrea sobre la que molts programadors han mostrat interès. Gràcies al ràpid creixement de les tecnologies, ha sigut cada vegada més fàcil la creació de videojocs. Però, amb l'augment de la demanda de noves mecàniques per a videojocs i més personalització d'aquests, la programació de

videojocs s'ha fet més exigent. Encara que hi ha una gran quantitat de recursos educatius per a ajudar a la creació de videojocs, sols uns pocs d'aquests recursos ensenyen a crear codi de qualitat per a videojocs.

El present treball final de grau se centra en el desenvolupament d'un framework que facilita la creació de personatges de 3ra persona per a Unity aplicant les millors pràctiques per al codi. Per a la construcció del framework s'han aplicat patrons de disseny que aconsegueixen l'obtenció d'un codi de qualitat.

Paraules clau: Videojocs, Unity, patrons de disseny, clean code

Table of Contents

List of Figures

Chapter 1

Introduction

A lot of programmers start learning how to code because of video games. Thankfully nowadays it is much easier to learn how to develop one. We have access not only to free game engines but also to a lot of literature, videos, courses, and other educational resources. However, video games are becoming more and more complex and the information on the internet quickly becomes irrelevant. So it is important not only to create a cutting edge solution but one that lasts longer and is customizable, because at the end that is what matters – developers want to create their video games and share their experience with the public.

1 Motivation

When the technologies became easier to obtain and learn, game developers did not stand in the way. They allowed people to get into the world of creating a video game by realizing their game engines to the public. This, of course, led to a lot of people being interested in creating their video games and the impact was huge. The amount of new people coming to this area is still growing. Nowadays, the game engines have become so powerful that they are being used not only for video games but also in movies, simulations and even in mobile apps. It seems good from the outside, but when you try to use them you quickly understand that they are overwhelming. They have grown so much and have so many things that it is frustrating for the beginners to get started.

Big companies that are developing the most popular game engines are trying to help beginners by not only providing documentation, guides, tutorials and so on but also making the development workflow easier. For example, Unity with their DOT system or Unreal Engine with the Blueprint System is meant to make it easier and faster for a developer to get into creating stuff rather than learning all the details or how to code. Although they achieved their goal of reducing the complexity of the first steps in the learning of the engine, we still have to understand that the game engine is fundamentally a big and complex system and a developer has to know all the tools that the engine is providing to create his or her project.

I started being interested in game development two years ago when I introduced myself to Unity, which is by far the most popular free game engine. I was motivated to learn but I was even more motivated to create my projects. It led me to the biggest problem I faced – everyone had their way to structure the code and the project. I had a lot of problems maintaining and expanding the previous ideas that were implemented by other people.

That is my main motivation – to create a framework for a Unity third person character that is easy to maintain, expand, and personalize.

2 Objectives

The main objective behind this project is to create a modular framework for Unity third-person characters that are easy to implement, maintain, expand, change, and add new features to. The goal is not to create a framework that would fit every situation, but to create a solid structure that would allow the developer to create prototypes of the mechanics. The main aim is a solid structure and design of the overall project.

Moreover, the testing of the code in game engines is not that simple as it is in normal software. For this reason, there is a need to investigate the available technologies for testing the code.

Next, because of the nature of being open source, it is necessary to a have solid documentation that describes not only the main structure of the code but also goes in-depth on how the code is integrated with the Unity components.

Finally, when the project is done, the final objective is to maintain the project by keeping it up to date with the latest Unity versions. In the future, I will create modules for the main project that will provide new features.

3 Structure of Document

This document is composed of 5 sections:

1. Introduction. The motivation and objectives of the project are presented. This section describes the purpose and the goals of this project
2. State of the art. This section introduces some approaches in the area of for game engines that are related to the main goal of this work. We focus the analysis on Unity in particular, and the third person character controller's solution in the popular web sites for beginners
3. Analysis. The analysis of the problem and the technologies that are going to help to reach the goal are introduced in this section. We are going to discuss unity, the design of the project, technologies, and the main flow of data
4. Implementation. In this section, the main aspects of the implementation of the framework and the problems that occurred during the implementation are presented.
5. Conclusions. This section is about the achievement of objectives that are defined in the introduction section and future work.

Chapter 2

State of the Art

Creating video games has always been difficult in terms of technologies. If 30 years ago people were limited by the amount of memory they had and the technologies they were using, today we are facing the same problems but on a much bigger scale. Luckily, we have a lot of high professional tools available to everyone.

To create a video game, we have a lot of technologies, and not the only the game engine itself, among others: audio editor, 3D model editor, animation editor, image editor, material editor and the list goes on. But game engines became so powerful that they could have all of these functions, not on the same level though, but if we would need to correct, for example, an animation we could do that in the editor itself.

1 Game Engines

Game development is a hard, time-consuming, and slow process that requires a high knowledge of the subject and a lot of professional from different areas. Programming the video game is a big field in this area and a programmer needs to have some knowledge about math, physics, algebra, and other related subjects.

Game engines are the tools that allow the developer to transform their ideas into reality. The companies that created them did not release it for the public audience, it has been used strictly in their own company and sold to the other big companies. That led to almost non-existing of the small indie games and it was harder to become a specialist. But that changed. In 2005 Over the Edge (which is now called Unity Technologies [14]) released Unity – a game engine that anybody can download for free. It was not growing fast by the begging, but with the release of iPhone and the raised popularity of mobile games, Unity quickly added IOS support for its game engine. It was a big step for Unity becoming the most popular free game engine, as Samuel Axon mentions that in his article Unity at 10: For better—or worse—game development has never been easier"[1]. Nowadays Unity is not the only one in the market. There is also Unreal Engine 4, freshly released open-source Godot and CryEngine. While Unreal Engine 4 is being more popular for big companies, but it has a higher difficult learning curve and Godot is still fresh and CryEngine doesn't have a big establish community, Unity still is a good choice with constant updates and the largest community.

2 Unity

As we discussed previously, Unity is one of the most popular game engines now that are open for the public. For all these years being developing by professional teams, it became a huge tool for developers with a lot of features. The engines itself is suited for almost every game genre and not only video games. Unity is also being used for movies, cars [4] and in the virtual reality projects. All that means that you can create a unique video game for a huge variety of platforms: starting from small web games and finishing with the newest consoles and phones.

Unity is not only a code framework or a variety of libraries, it provides visual tools which meant to be integrated with the code and enable simpler, faster, and easier development process. It is not only for programmers but also for other people that are not familiar with coding: artists, game designers, level designers, UI/UX artists and so on. The big part of the development is to integrate the code with the provided features of the engine and finding the balance between the code and the manual manipulation of the assets.

Unity provides a working environment for both 2D and 3D games[15] (That includes semi 2.5D and isometric) with the variety of tools and already build in features: graphics, audio, physics, AI and other functions that the developer does not have to worry about implementing those, which in fact, what makes a game engine a game engine. But Unity is not as simple as that, by its age, it became necessary for new features that not only adding new graphics support but also the features that make development easier: debug, asset view, visual scripting, new input system[3] and so on.

2.1 Unity Editor

Unity Editor is the main application that a developer will use. It is the base that connects all the features with all the code and provides that for manipulation for the user. Even for a programmer that essential to understanding how to use the editor and integrate his code with the unity components by a provided scripting API.

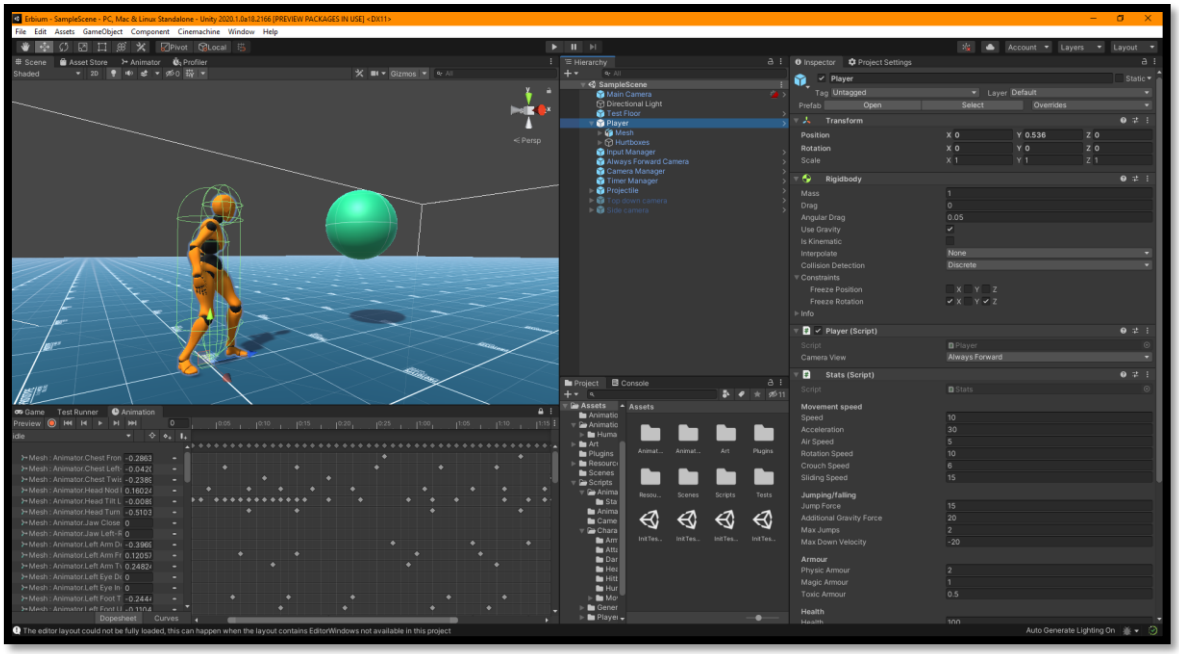

Figure 1 Unity Editor

We can see at figure 1 main windows of the editor:

- **Scene** – here the developer can move around the world and interact with the objects that are in this world (move, scale and rotate them)
- **Hierarchy** – manager of the Game Objects (every object that is in the game) and the hierarchy among them
- **Project** – file manager of the current project
- **Game** – the main game window where a developer can play the game in the current state
- **Inspector** – displays details about the current selected Gameobject including its components and properties.
- **Console** – a normal console that is showing the errors and other logs in the runtime and compiles errors

There are also a lot of more other windows that serve for different purposes: Animation for modifying animation, Test Runner for running test, Profiler for analyzing performance and so on. For custom solutions, Unity provides developers to create their editor windows and extensions with tools.

2.2 Game Object

Game Object is a fundamental block of any game in Unity. Everything that you can see in the game, and even what you do not see (any kind of managers) is a game object.

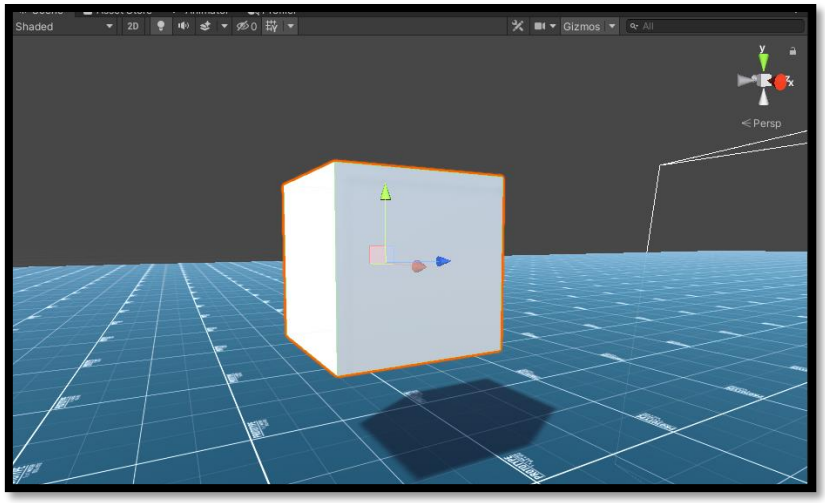

Figure 2 Game Object

In this picture, you can see a game object in the scene window. As we can see, that's a cube which is one of the defaults meshes in Unity. You can rotate, scale, and position the mesh with your mouse selecting one or multiple axes. In the scene view, unity displays axis and other gismos (as you can see in figure 2, in the center of the cube there are 3 arrows).

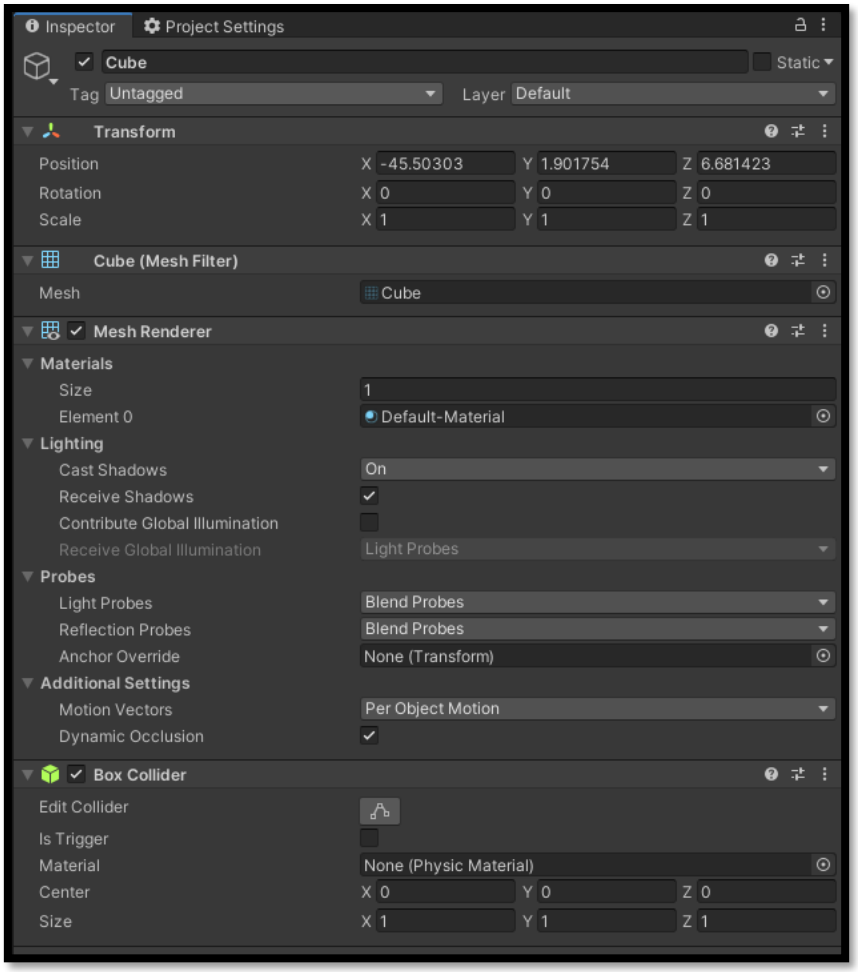

Figure 3 Inspector of the cube

Figure 3 shows us the inspector of a game object with components. Each game object consists of components that describe its behaviour and functionality. For managing components, we have a special window – inspector. Beyond components, it provides organization manipulations for game objects – changing its name, tag, layer and more which is being used to identify and group game objects for special logic(for example, by the layer we can determine what game object is considered as a floor)

As we can see in figure 3, this game object has different components:

- **Transform** – Describes where and how the object is being placed in the world by 3 vectors – position, rotation, and scale. It is the most important component for any object and by default, any object has it (except UI elements which are being placed in a special canvas)
- **Cube** – component that tells what mesh needs to be rendered
- **Mesh Render** - the component that allows attaching model to a game object. It allows add materials, allows lighting, probes, and other different settings.
- **Box Collider** - Collider is a component that generates borders for the component to interact with other objects (physically or just to check if other components are overlapping). Colliders can be different shapes: box, sphere, capsule, and complex ones, such as mesh collider. There are two types of colliders: physical and a trigger. Physical one does not allow other objects to go through it, on the other hand, a trigger is being used for logic, for example, activate weather when a player exits a building.

2.3 Unity Scripting API

At the begging Unity was supporting two programming languages for the coding: JavaScript (UnityScript) and C#. Starting from 2017.2 Unity Script is depreciated [6] which led to the only one available option – C#. Usually, in game development, the most popular programming language is C++ for one good reason – performance. That is extremely important that a video game is well optimized and run with a high stable frame rate. Optimization in video games is not only optimizing the code, there is a lot of techniques that can help achieve that. But still, Unity is being punished by the community for picking C# instead of C++. However, Unity Technologies and Unity Community provides good solutions for optimizing C# code – from basics like Object Pooling, caching references, special scripting API usages to things like ECS(Entity component system)[9], which is a complete turn-around from a typical OOP to a Data-Oriented that can performance heavy logics much better. However, ECS is still developing and the basic way of writing code in Unity is still at the top of popularity, and because of this reason for this project I used normal OOP behaviour.

2.3.1 Mono Behaviour

To give the Game Object functionality we must attach components to it. There are built-in components, like Rigidbody for physics calculation, Mesh Render to attach a model to the game object and so on. We can create our components through code using the available API.

Mono Behaviour is a class that a developer can expand. By expanding this class to the script, this script is becoming a component and available to be attached to a game object.

This class provides methods which include life-cycle methods which you can customize. There are around 40 life-cycle steps, but the most important ones are:

- **Awake/OnEnable/Start** – method is called when the game object is created or became real in the world (depending on the chosen method)
- **Update/FixedUpdate** – method being called every frame or every 0.02 seconds depending on the method
- **OnDestroy/OnDisable** – method is called before the game object destroys

2.3.2 Using Other Components in Code

Already prebuilt components not only providing functionality to a game object in the world, but they also provide API for using them in code. In the custom script all the components that are attached to the game object, could be used getting the references to those by special Mono Behaviour methods. By that, it is possible to apply, for example, physics forces in a custom script.

3 Third-Person Character Controller

Character controller is not only one component but with other features that allow building a basic structure for basic mechanics (taking damage, camera movement, animations and so on). It can be applied not only for the third person but rather for any object that is being controlled by the player or AI.

Unity provides one Character Controller [12] which is just a component for game objects. It provides some basic movement functionality and some minor features. But by the nature of being a prebuilt component in unity, it is difficult to change or even override default functionality. It could be used for a normal character only, like humanoid characters. And the other inconvenience about this component is that it is not physics-based, and if a developer would want to add physics to the character, it will lead to removing the Character Controller component.

Unreal Engine 4, on the other hand, has a lot of character controllers out of the box: third-person, first-person, vehicle, air-vehicle and so on. It also provides integration with other prebuilt functions of the engine such as damage system, navigation, multiplayer

and many more. They could be modified or extended but there is a problem with it. Extending a controller to add new functionality in UE4 is easy, thanks to a big functional API and blueprints, but modifying the existing functionality requires a lot of knowledge of the engine. A character controller is one of the core systems of Unreal, so to modify it, a developer must go deeply in the source code and carefully change it.

Because Unity does not provide controllers, developers should come up with their ideas and structures for logics for the characters. It has its advantages and disadvantages. If the developer has a lot of experience and knowledge, he can create the structure with the features he needs. But if the developer does not have those quantities, he needs to spend a lot of time of refactoring the code later and researching some implementations.

Even though you can find some character controllers created by other developers in the asset store provided by Unity[13], there are somethings that you have to consider: mainly they all are not free and because of that, they tend to have with a lot of functionality which leads to the complexity of the code. All that leads to the problem that we face with Unreal Engine 4 – difficulties with changing the source code.

4 Public Implementations

As we discussed previously, beginners do not have experience or a good customizable default character controller in Unity. The character tempts to be the first thing that developers try to create. Usually, beginners tend to search for the tutorials on the popular video-course websites, such as YouTube or Udemy. Those platforms provide a bunch of quality tutorials, free and paid as well.

I was learning by them as well such. By my experience, I found some problems with the tutorials online, even though the quality was good.

The problems I encouraged were:

- **Non-generic solution** – mainly all these courses/tutorials are around building a final video game, where the controller is meant to be done the only way
- **Not customizable** – that is the main problem that I faced. By the end of the course, you will have an almost identical project. But video games were meant to be unique and people will try to add new features. But there is a problem with it – the code was not ready to be expanded or modified. You will most likely find yourself rebuilding a lot of systems that it would not break and would function with a new feature.
- **Undocumented project** – the other problem is being that there is no documentation for those projects. If you want to find the purpose of a feature of why it was implemented the way it is, you mostly fill try to find it in a video which is not the quickest and easier way.

There is a good chance that you will encourage a lot of bad practices in terms of the writing of clean code, SOLID principles [10], untestable code, files with thousands of lines of code and so on. That was my main motivation - develop a clean solution for quick prototyping mechanics.

Chapter 3

Analysis

This chapter presents the requirements that have been identified for the framework together with the plan of the project to develop the framework. Finally, the chapter introduces the technologies used in the project.

1 Requirements

Here we will discuss the requirements that were established from the analysis of the encouraging problems and the features that need to be in the default version of the project.

- **Movement** – Movement is one of the fundamental parts of the character. It is essential to create a framework that allows developers to easily create/manipulate different types of movement.
    - **Jumping** – In almost every action video game character can jump, so it is important to include this feature in the project.
    - **Crouching** – Crouching, like jumping, exists in most video games, for that reason, it is supposed to be one of the default movements.
    - **Sliding** – Sliding is another type of movement popular in video games, where the character slides on the ground for a few seconds
- **Camera** – The camera is one of the most important tools to transfer the experience to the player. It Is essential to provide a separated structure that allows a developer to connect and adapt to the camera.
- **Damage system** – It is likely that video games have some sort of damage/health system. The project must provide a basic implementation of the health system.
- **Combo attacks** – Combo attack is an important part of action games. Because of being one of the fundamental parts of game design, the combo system must be customizable both for a programmer and a game designer. Because of that, it is important to create a customizable system for attacks.
- **Clean code** – It is a high priority to create code that is easy to read with established clean code principles
- **Uniform code style** – Uniform standards provide good quality of the code for the entire project
- **Design patterns that allow expanding** – There is a lot of design patterns out there, but we can divide them into 3 categories: creational, structural and behavioural. To reach the established goals, there is a must of choosing the best-fit design pattern for different problems in the project.

- **Testing** – Unity provides a testing environment that is a good choice to create tests for the default features of the project.
- **Documentation** – It is essential to create quality documentation for the basic and overall structure of the code.

2 Development Plan

Before starting to develop the project, I decided to divide the development into two main parts: the actual development and the documentation. At the begging, I estimated that those two parts would consume the same amount of time, but in the end, my rough estimations were not correct, and the documentation and testing required more time.

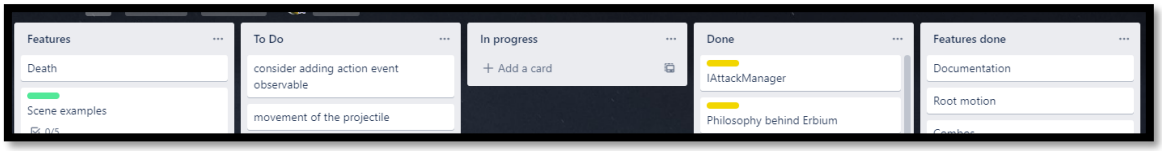

Figure 4 Kanban board

For task management, I used Kanban board workflow. As figure 4 shows, I had 5 columns:

1. Features – This column has all the features of the project.
2. To-Do – This column represents all the tasks that need to be done. They are closely related to the actual work. They are labels with colour to distinguish them: red for bugs, yellow for documentation, purple for tests and so on.
3. In Progress – This column has all the tasks that are in progress.
4. Done – This column has all the tasks that are done.
5. Feature Done – This column represents all the feature from the first column that has been done.

To create this project, I established the following steps:

- Establish the technology stack that will be used
- Create basic UML diagram of overall project structure
- Establish design patterns that would fit the project
- Find assets to use
- Start developing process.

3 Technologies

Apart from Unity Engine, to create a framework we would need:

- Code editor
- Version Control
- Task Manager
- Additional libraries

3.1 Unity

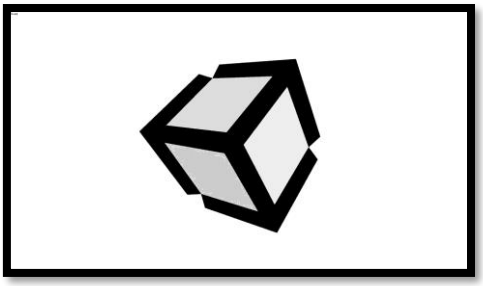

Figure 5 Unity Logo

Unity is the engines that I will be using with provided default features and scripting API. I will not use other assets available from the asset store, only the model, animations, and textures from Mixmao.

3.2 Rider

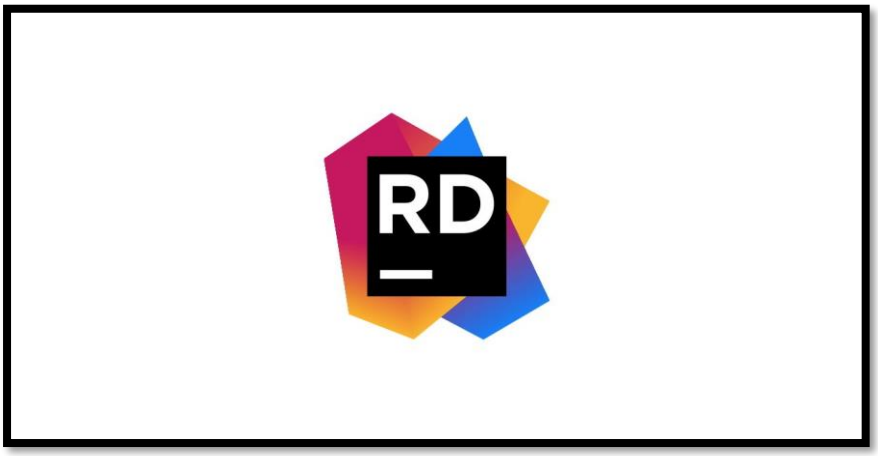

Figure 6 Rider Logo

Rider is an IDE for .NET from JetBrains that has all the typical JetBrains IDE features refactoring, indexed search, autocomplete and unity API integration. Although Unity by default suggests using Visual Studio, I found Ridder much more powerful and easy to use. JetBrains provides integration for Unity by default in Rider with plenty of features to help us: showing performance-critical contexts, static analysis of unity API with suggestions of replacing high costly methods and so on. Besides that, Rider provides, in my opinion, the best coding experience in the market due to overall JetBrains established ideas.

3.3 GitKraken & GitHub

Figure 7 GitKraken and GitHub Logos

I want to publish the project to the public, and what is the better place for developers than GitHub. The documentation will be also there in the wiki page of GitHub. GitKraken is a GUI for git that helps using git.

GitHub is the most popular site for open source projects. It does not provide only version control, but also a wiki editor for each repository found issues by other users and other things that makes maintaining the project easier. GitKraken, on the other hand, is just a graphical user interface for GIT version control. It integrates without problems with GitHub. It allows us to keep track of our commits and branches with the visual representation of the changes we make.

3.4 Trello

Figure 8 Trello Logo

For planning and managing the tasks and the whole development process, I used Trello, which is a Kanban-style website. Although it is usually being used in group projects, it is still very useful even for a solo project. Not only it helps you have a plan of the development process, but it can also help you estimate the time, track bugs and issues and think about the structure of the project before starting.

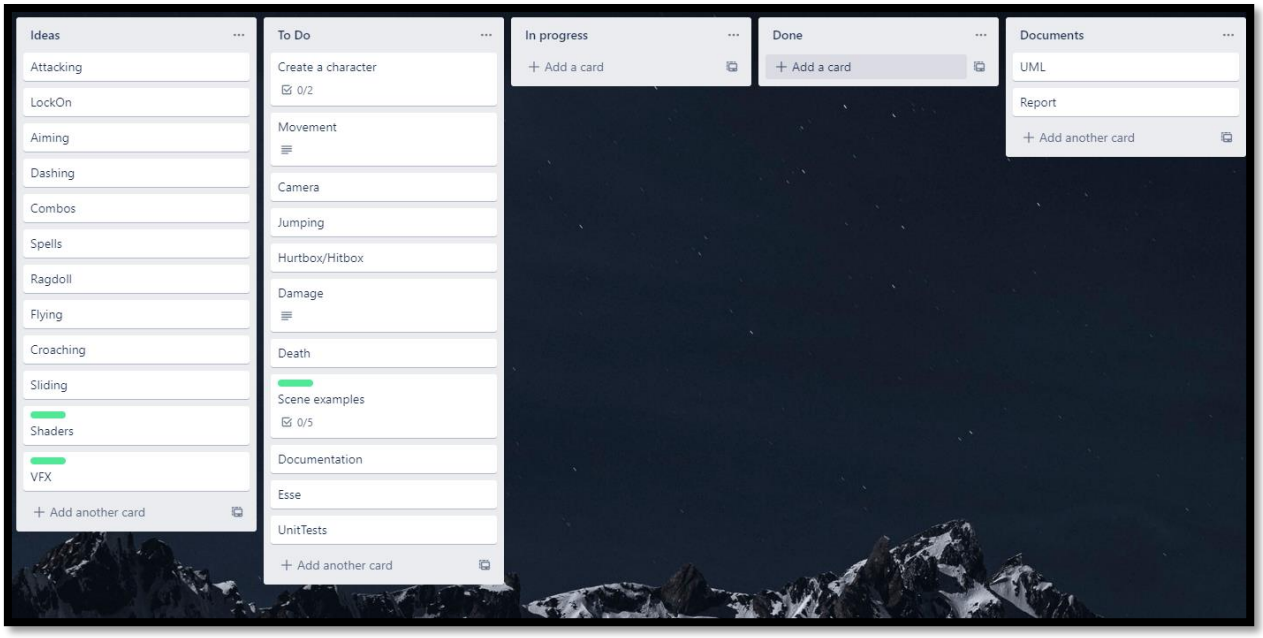

Figure 9 Trello at the begging

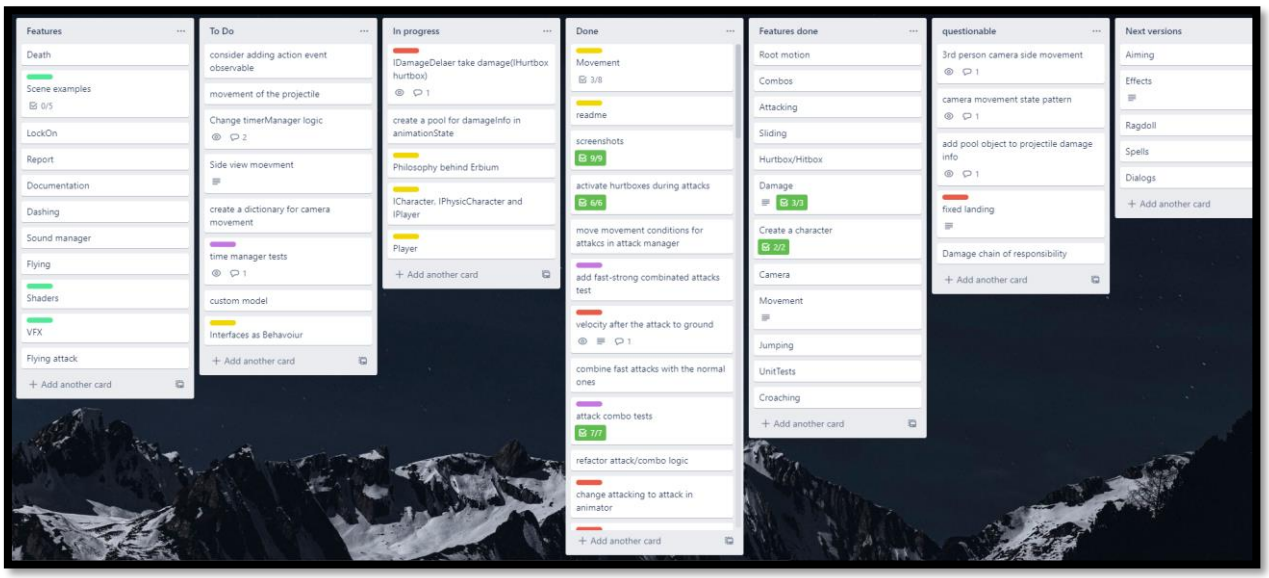

Figure 10 Trello near the end

3.5 NSubstitute

Figure 11 NSubstitute

To create tests, I need a library for mocking the objects. I decided to use NSubstitue because it does not have problems with unity integration. This library allows us to mock classes and methods for testing purposes.

In Unity testing environment, it is very helpful for mocking the inputs of the player. When we must test the player character, we need to make sure the inputs of a player control the character. But during tests, we cannot use the inputs. NSubstitute allows us to mock the input Unity API to test the player properly.

Chapter 4

Implementation

In this section, I will discuss the overall design of the framework and the problems that occurred during the development of the framework.

1 Structure of the Project

Before explaining each component, we should take an overall look at the structure of the framework. To achieve expansion, we should consider a modular approach. By reducing the amount of logic every system can do, it gives us a lot of potentials to reuse and replace the components.

The framework is made of 4 main components:

- Character
- Movement
- Animations
- Combat

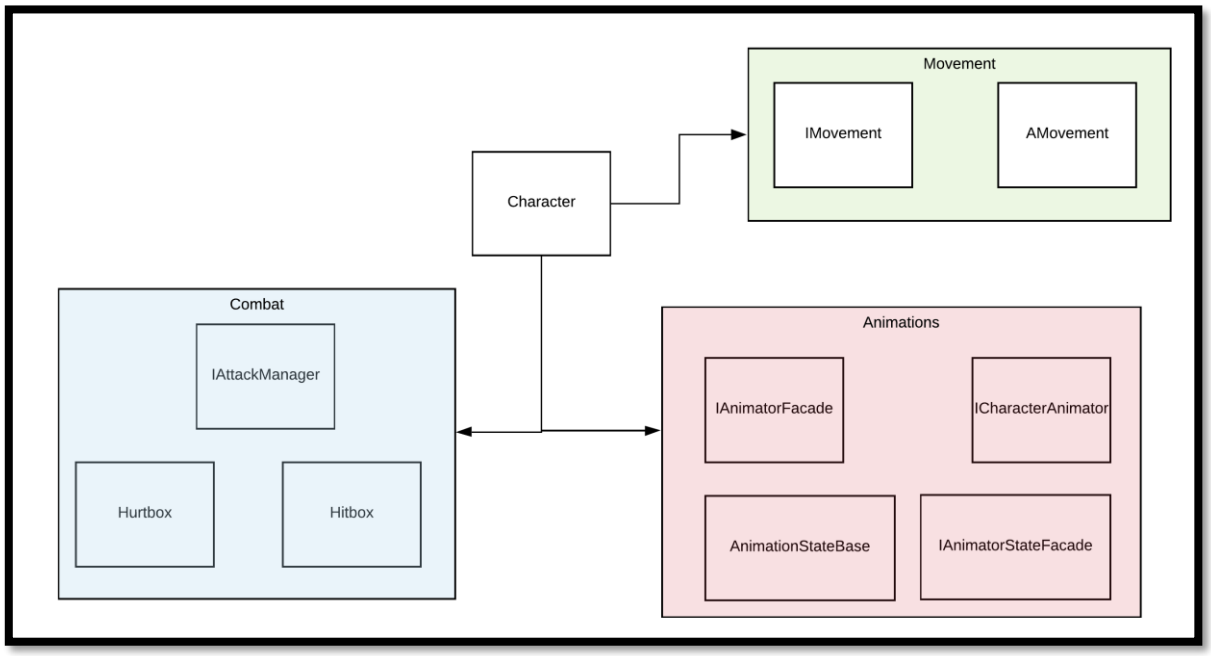

Figure 9: Structure of the project

As we can see at figure 9 Combat, Animations, Movement components are connected to the Character component. It is the base of the whole system and it is the script that needs to be on the game object of the character. We are going to dive deeper into each one of them.

Each component has different structure behind it and has a specific role. The movement system oversees moving the character, it has different states for each kind of movement. The animation system translates the actions of the character to the animator system in Unity to play the correct animation and to connect animation with the other systems. The combat system allows the character to deal and receive damage, it is working closely with the movement component and the animation component.

2 Character

Character is a base class that is attached to the main game object of the character. The script can be attached to any character – the player or the AI. Depending on the character the input should be specified – camera and normal input for the player and behaviour trees for AI.

Character script does not have heavy logic, but it manages all the subsystems of the character and keeping the general state. The class does not move or animate the character directly, it is passing that responsibility to the correspondent modules. That is a perfect fit for a facade – the structural design pattern. Introduced in the popular book of Erich Gamma [5], the author of many books about design and design patterns in software. He defines facade as an object that provides a single and simplified interface to facilitate access to a subsystem.

2.1 The Facade Pattern Applied to Character

Imagine running a cloth store, but you must deliver the cloth from the warehouse to the store, selling the products, managing the marketing strategy, everything by yourself. It would be easier to create teams that are managing a specific area. That is what facade is doing - managing all the modules.

Facade provides basic interface solution for the user without complicating it with the logic behind each interaction. In place of doing the logic all by itself, it calls the subsystem. It might be more proficient to work with the module directly, but a facade does not only simplify the talk with a module, but the main purpose of the facade is also to control a bunch of modules.

You can see a game engine as a facade that managing a lot of facades. The top layer facade – the editor, is a very simplified version of a facade. Game Engine does not need to know how to render a polygon, how to increase the pitch of the audio track or configure the gravitation scale. It manages that all the component work well with each other and providing the functionality of each to the user.

A good smell of implementing the facade is when in one class you have different functionality and it gets complex. The idea is to separate each functionality by the modules. The complexity of each module gets more and more complex and applying a

facade typically leads to dividing classes into more classes and you might be interested in combing the facade with other design patterns, such as factory, singleton, adapter and others. The risk of the facade is that the class can become too big and you might consider dividing the facade into other facades.

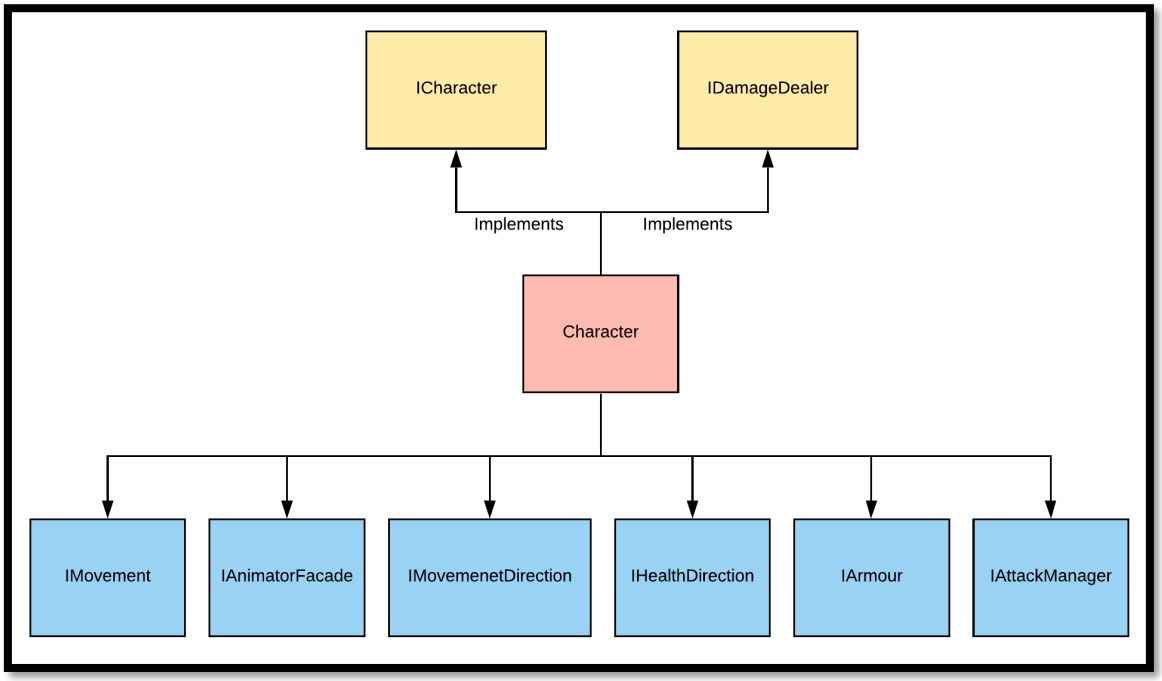

Figure 10: Character class diagram

As we can observe from the image above, the character implements two interfaces: ICharacter and IDamageDealear. IDamageDealear allows the character to deal with damage (more on that later) and ICharacater has methods to access all the modules that a character must have:

```
1. public interface ICharacter {
2.          IMovement getMovement();
3.          IHealthComponent getHealthComponent();
4.          IAnimatorFacade getAnimatorFacade();
5.          IArmour getArmour();
6.          IAttackManager getAttackManager();
7.          void changeMovement(MovementEnum movementEnum);
8.          void die();
9.          Stats getStats();
10.      }
```

The character does not have a lot of methods for the functional purpose, more so, almost every method is a GETTER for other subsystems to have the correct reference to other modules.

By having a character class as a facade, we can simply add and remove the components and modules of the character and customize it without changing a lot of code. Although by changing the subsystems we must make sure that it will not break other components.

That is a risk we should take by moving the logic under the modules. We cannot remove the communication between the modules at all, but module design makes it much easier to change, add, remove, and test the code then have a major class that does everything. Even though sometimes it seems like adding a new module for one small thing is overkill but it can be beneficial in a long run, for example, make the sound when a character lands – it is much easier add the method for that in a class that's in charge of making a character jumping but it would be a better solution creating a sound module for the character, so later we could add and modify more sounds and control the sound behaviour much easier.

The character has the following modules:

- **IMovement** – the module that allows the character to move in the world
- **IHealthComponent** – the module that allows the character to receive damage
- **IAnimatorFacade** – the module that connects the code to the animator system of Unity
- **IArmour** – the module that calculates the received damaged depending on the damaging nature and the character's amour
- **IAttackManager** – the modules that allow the character performing attacks and combo attacks
- **Stats** – the modules that contain all the information about the current state of the character, like current health, speed, jumping force and so on

We are going to discuss every module individually in the next topics, but now we should focus on that Stats module.

```
[Header("Movement speed")] public float speed = 10f;
public float acceleration = 30f;
public float airSpeed = 5f;
public float rotationSpeed = 0.1f;
public float crouchSpeed = 6f;
public float slidingSpeed = 15f;
[Header("Jumping/falling")] public float jumpForce = 10f;
public float additionalGravityForce = 20f;
public int maxJumps = 2;
public float maxDownVelocity = -20f;
[Header("Armour")] public float physicArmour = 2f;
public float magicArmour = 1f;
public float toxicArmour = 0.5f;
[Header("Health")] public float health = 100f;
public float invincibilityTime = 2f;
public float headDamageMultiplier = 2f;
public float bodyDamageMultiplier = 1f;
```

Figure 12 Stats

As we can see from figure 12, this class does not have a single method, all it does is contains the data that could be changed in runtime. The purpose of this class is to unify

all the data of the character so when the value of an attribute changes it will affect all the systems that are using this variable. But even though this class does not have functionality beyond that, it extends the unity API class – MonoBehavior, so the developer can see and change the value in the editor as shown in figure 13 in Stats.

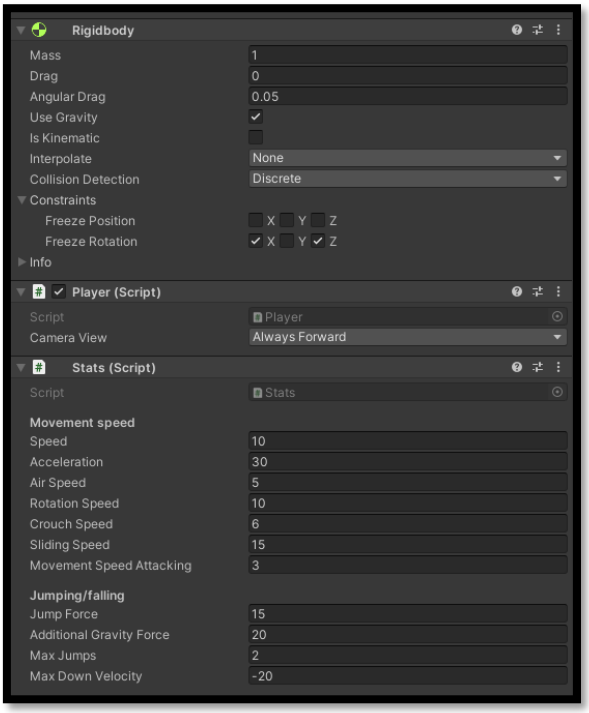

Figure 13 Character Game Object Components

Because the character is using a rigidbody for moving, the character's game object must have one. But it also needs more components to properly function. For the game object hierarchy, I decided to divide the main logic of the main functionality with the art part of the character (model, animations, sounds and other components). We can see it as a facade but with hierarchy model: we have the main object which has the general functionality and all the main components, then under this game object, we have different systems: model game object, colliders game objects and other as shown in figure 14. Depending on their role they communicate with the main game object through the code or the Unity systems.

Figure 14 Player Game Object

Parent game object (Player) has 4 important components: Player script, Stats Script, rigidbody and physical collider. Mesh game object has the mesh component as well as Animator Facade and Character Animator scripts (More on that later). Hurtbox game object is defining the area of where the character can receive a hit and registers it. Currently, a character can be damaged to the head and the body.

Now let us dive deeper into the character interfaces. As a base interface, we have *ICharacter* which has all the getters, that we discussed previously, along with *move* and *die* methods. The child of these interfaces is *IPhyscialCharacter* that obligates the character to work with Rigidbody. The purpose of these interfaces is that it allows the character to use CharacterController instead of working with physics. Then, the children of PhysicalCharacter is *IPlayer*, that allows the character to use the camera as an input. If we are working with an AI character, we would have to need to use an AI interface with the functionality for deciding the direction for the movement and other needed behaviour. As we saw previously on the UML diagram, the character class also implements *IDamageDealer* interface which allows the character to deal with the damage.

3 Movement

Movement is one of the most important parts of this project. It deals with moving a character in a world with the ability to quickly change the state of that movement: a player can jump, slide, run, walk, crouch and so on.

Normally, movement is integrated deeply with the basic functionality in the character class. Although this approach makes everything much easier at the begging and sometimes it's beneficial to keep it that way for integrating with other systems (for example: starting the animation of a jump when a player pressed the space bar, making a landing sound effect, emitting the dust particles when a character is running and so on). However when you would want to implement a new way to move your character (wall-running, flying, etc..) you would likely to find yourself in a situation where you would rewrite a bunch of code or put a lot of *ifs*, which would affect the code readability, maintenance and overall the code quality.

By the idea of giving developer freedom to build his code yet by providing a base structure for implementing the movement module, I decided to search up design patterns for this problem. Although they are a bunch of patterns that allow expansion, I decided that the best fit would be a state design pattern.

3.1 State Design Pattern Applied to Movement

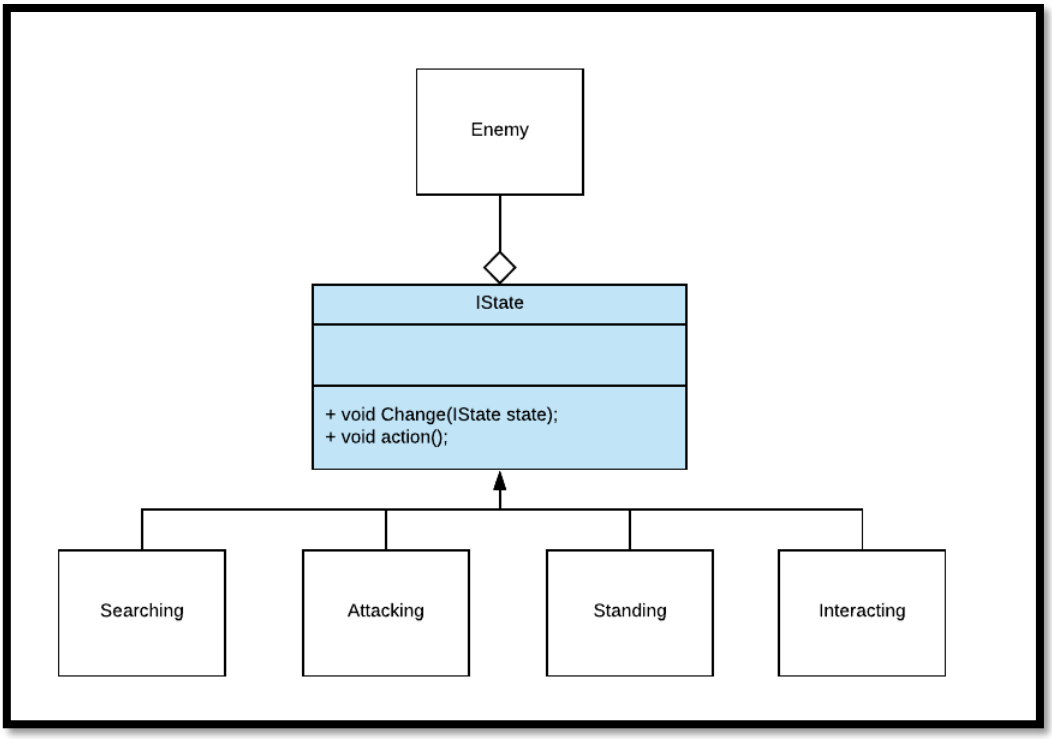

Figure 15 State design pattern UML enemy example

For the sick of the example let us imagine an enemy in a video game. It can stand still, chase the player, attack him, search for him, interact with other enemies and so on. We can see the pattern there; the enemy has different states and we could separate the states from the main system. That is the main purpose of the state design pattern: a system has states that are constantly changing between each and one independently. Each state does not depend on the others, it interacts with the main system and changes depending on the behaviour. The change of the state could be done internally (in the state machine) or it could be triggered by the external systems. Imagine an enemy standing still (standing state) and it hears a player, the state now is searching, and the change was triggered by the external system – sense system, as shown in figure 16. But when an enemy is close enough, he will change his state to the attacking (internal change).

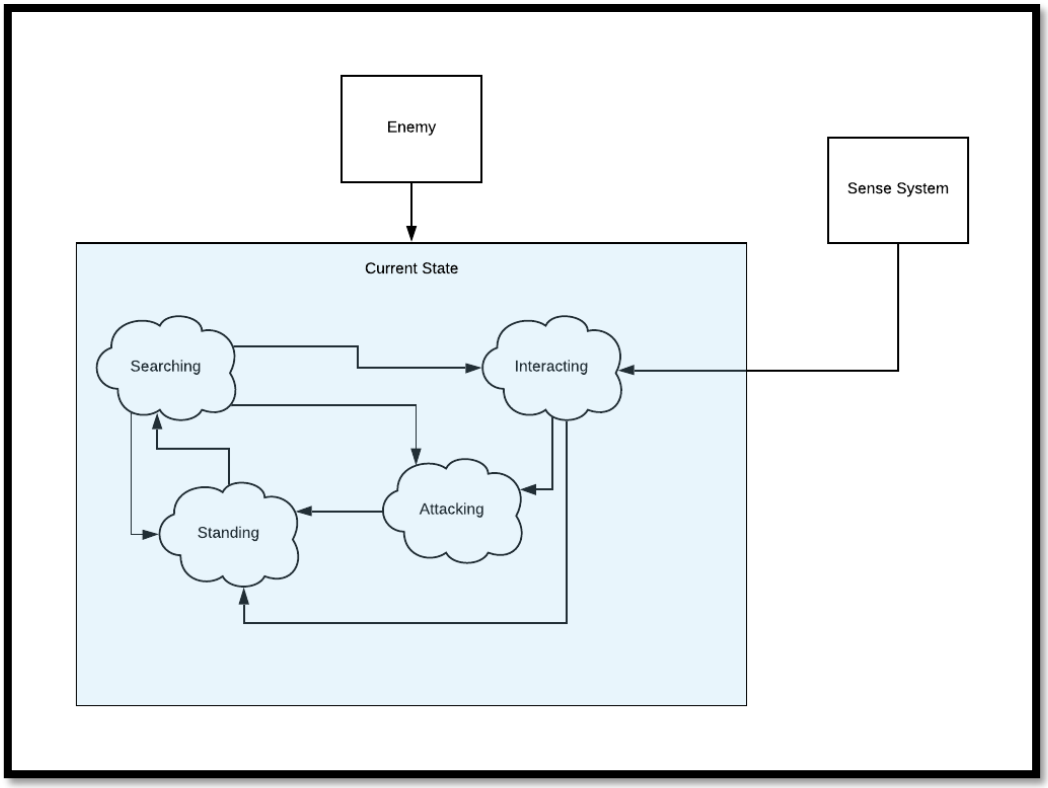

Figure 16 State design patter flow diagram enemy example

By implementing this design pattern, we could achieve a lot of advantages:

- **Single Responsibility Principle** – a principle from SOLID that obligates classes to do only one thing. That goes perfectly with the state design pattern – all the states are describing how the system would act in a different state, and each state has its responsibility.
- **Open-close Principle** – a second principle from SOLID – a class should be open for extensions and close for modifications. This pattern is a perfect match for this principle – it is easy to create new extensions without modifying the previous states.
- **Eliminating conditions** – by implementing this design pattern we eliminate a lot of conditions and our code becomes simpler.
- **Eliminating repeated code** – we can create an abstract class that implements the state interface and has methods and attributes that could be useful for every state.
- **Modular approach** – this design patterns allows us not only to encapsulate the entire state module, but we can easily add and remove states.

State design patterns are not only a solid option by its own but it is a great choice for video games because it would fit a lot of situations, not only the general state of a character but also subsystems that could benefit from this design pattern.

Movement in video games could be just a basic walk-run cycle but in a lot of modern video games we could see a lot of different types of movements: wall-running, ladders, flying, swimming and the list goes on. In each type the character could have different behaviours: disabling camera movement, disabling attacks, increasing the gravity and so on. It is not only hard to keep track of everything that is going on, but we also must pay a lot of attention to the changes between the states and how they would change the character parameters.

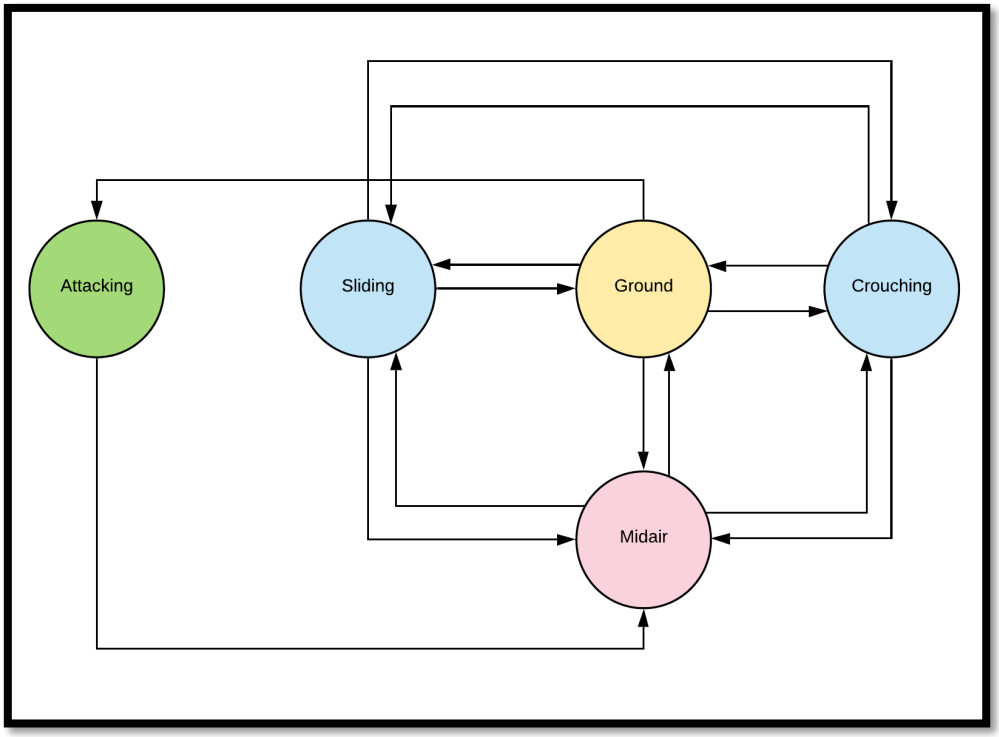

Figure 17 Movement state flow

As we can see in figure 17 each state has different behaviour for changing. However even with that many change possibilities state design pattern allow us to reduce ifs statements a lot, by polymorphism and sharing the same behaviour in the abstract state (we can see that every "ground" movement can transit to the midair – to fall from the ground, we can encapsulate this logic to the abstract base class).

IMovementState is the main interface for movements. It describes the lifecycle of each state:

SetUp

This method is happening right after the current state has been changed. In this method, we usually configure a character and all related systems for this specific state. For example: telling the animator to play correct animation, increasing the gravity, disabling camera rotation and so on. This typically is done in a constructor but for the performance reasons (which will be explained later) it is being used here.

Move

The main method which is being called every frame. Here is we execute the main logic of the state.

ChangeState

This is the method for external and internal changes of the state.

CleanUp

Before the current state changes, this method is being called. The main purpose of this is to set all the attributes that were changed in *SetUp* to the default values and do some finishing actions if the state requires those. It could be done in a destructor but for the performance reasons, it was expanded to a new method.

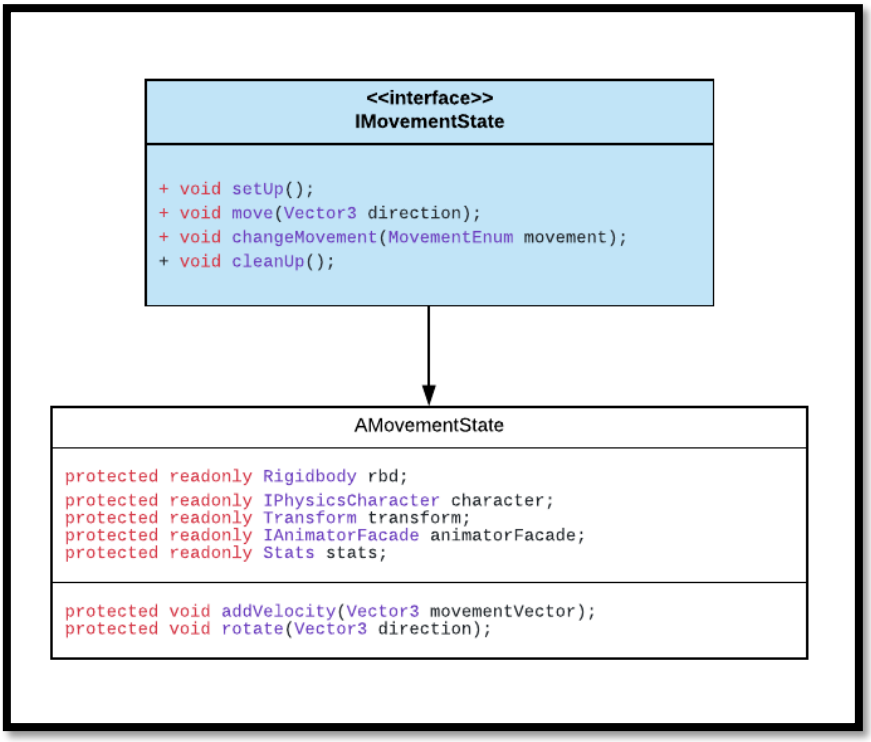

Figure 18 IMovementState UML

Every state extends the *AMovementState* class, which is abstract. The idea behind it was to share all attributes for every state to work: rigidbody, character, transform of the character, animator facade and stats. It provides two methods that are being used in almost every state but with the option to override them.

Internal change of the state is being called in the states. Let us imagine that the character is walking off the cliff and starts falling. We must check every frame if the player is on the ground. If he is not, then we must change the state to midairMovement. But it is

important to remember to call the cleanup and setup methods. That is why we invoke changeMovement in the ICharacter class.

```
1. public override void move(Vector3 direction) {
2.     if (isFalling()) {
3.          changeMovement(MovementEnum.Midair);
4.          return;
5.     }
6.     var velocity = accelerateAndMove(direction);
7.     rotate(direction);
8.     updateAnimParameters(velocity);
9. }
```

We can see that every frame in the ground movement we check if the character is falling, if so, then it changes the current state to MidAiarMovement, if not then move the character on the ground. But changeMovement calls the method of a character because before we need to execute cleanUp and setup methods:

```
1. public void changeMovement(MovementEnum movementEnum) {
2.     movement.cleanUp();
3.     movement = movements[movementEnum];
4.     movement.setUp();
5. }
```

External change of state works similarly, the only difference is that it directly calls the character's method.

3.2 Performance

Performance in the video game is crucial and states have problem with it. In a usual representation, we create a new instance of a state each time we change it. In video games, we always want to create and load all the instances before entering the game loop so in a run-time we would not have to spend the computation power to create instances. And the second reason is that C# uses garbage collector and we do not want to create a lot of garbage because that will increment the computation needed for the game to deal with garbage spikes (cleaning the garbage) which is critical for the performance.

For that reason, I created a MovementEnum - enumerator with all the possible states:

```
1. public enum MovementEnum {
2.      Ground,
3.      Midair,
4.      Crouch,
5.      Slide,
6.      Attack
7. }
```

The character has a dictionary that has an enumerator value as a key, and a movement state instance as a value. In the constructor, every state is initializing and being stored in the dictionary. In the change state method, all we must do is change the current movement to one from the dictionary. In the end, we have all movement instances in the memory and changing the current movement by simply accessing it from the dictionary.

However, that leads to another problem with the garbage collector. In C#, a dictionary with the enumerator as a key generates a lot of garbage due to boxing [7] of the enumerator while comparing the values. To solve that we need to implement our comparator, that avoids boxing.

```
1. struct FastEnumIntEqualityComparer<TEnum> :
   IEqualityComparer<TEnum>
2.         where TEnum : struct {
3.         static class BoxAvoidance {
4.             static readonly Func<TEnum, int> _wrapper;
5.
6.             public static int ToInt(TEnum enu) {
7.                 return _wrapper(enu);
8.             }
9.
10.              static BoxAvoidance() {
11.                  var p =
   Expression.Parameter(typeof(TEnum),null);
12.                  var c =
   Expression.ConvertChecked(p,typeof(int));
13.
14.                  _wrapper =
   Expression.Lambda<Func<TEnum,int>>(c, p).Compile();
15.              }
16.          }
17.
18.          public bool Equals(TEnum firstEnum, TEnum secondEnum) {
19.              return BoxAvoidance.ToInt(firstEnum) ==
20.                     BoxAvoidance.ToInt(secondEnum);
21.          }
22.
23.          public int GetHashCode(TEnum firstEnum) {
24.              return BoxAvoidance.ToInt(firstEnum);
25.          }
26.      }
```

By doing this optimization we achieve decrement in the garbage allocation per frame around 200b in average (even more when we rapidly change states), which is not much but as the system gets more complex that could be a significant issue. It is not only the code; it is also all the sounds/particles/models and more things that could be created in a state.

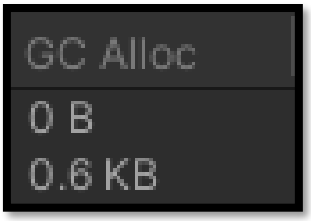

Figure 19 GC allocation before optimization

Figure 20 GC allocation after optimization

3.3 Interfaces as Behaviours

When I was working on the movement states I reached the point where I needed to implement some generic behaviour (is the character able to jump in this state? is the character able to fall in this state? and so on...). I did not want to implement those behaviours in AbstractMovement because I want this class to be light and be well-rounded, but those behaviours are meant to me be customizable for every state. I was inspired by the Haskell naming conventions[11], where you can easily guess what this method could do just by looking at the function's declaration. I wanted something similar but for the states. I found a possible solution - to express the capability of a state through interfaces. That way we can see what the state can do just by looking at the class definition. It would not say us how the certain behaviour is being used though, but we would know that this state has this behaviour in some form or shape. It allows us to be more organized and easily understand the code.

Let us take *CrouchinMovement* as an example:

```
1. public class CrouchingMovement : AbstractMovement, IFallable,
   IJumpable
```

Just by looking at the definition when can tell what this movement can do. Let us have a deeper look into that:

- *AbstractMovemet* – that says that it is a movement class, not a subject of interest here
- *IFallable* - we can see that from that movement we can fall. We are not obligated to create the same behaviour for every class that implements *IFallable*, it just helps us understand the possible behaviours that class can do.

- *IJumpable* – that says that a character can jump from this state.

Those interfaces are simple and usually have one or two methods to obligate us to create the functionality of this behaviour.

```
1. public interface IFallable {
2.         bool isFalling();
3. }
```

4 Animations

Animations are another big part of videogames and Unity provides not only an API but also a special editor for tech artists and animators. If we take for example a rigidbody, that is a component designed to be interacted with only in code. But animator, on the other hand, is designed for both types of work: the editor and through coding. Because of that we must keep a good balance between automation with our code and giving the freedom for artists to animate characters.

4.1 Animations in Unity

One of the most fundamental things in the animations is a transition from one animation to another. Unity takes all the hard work of calculating the mesh movement, bone rotation and other things for itself. But we are in charge to create an animation state machine. The way we do it is not that hard (for a simple state machine) – create parameters and put conditions into the transitions between the animations. All of that we can do in the animator editor. However, to actualize those parameters we must connect animator with the code. They are many debates of how we should organize our character: we can have a general state machine of a character in Unity animator editor, that connects a; the actions and the animations very closely or we could divide our state machine – one for animation and one for the character. That way we have more control over the character and be independent of the animations, but it could lead to bugs with the animations and overall players' feedback of their input.

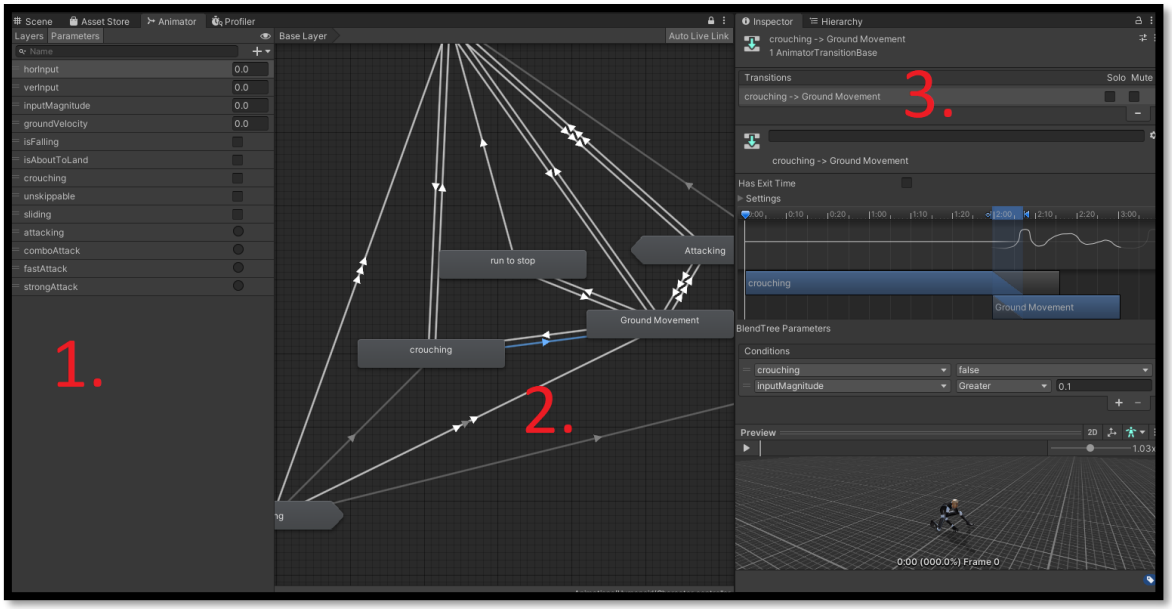

Figure 21 Animator editor

In the image above we can see the animator editor window. In window 2 we can see all the animation states and the transitions between them. To add conditions to a transition we should pick one and, as shown in window 3, add conditions and configure the transition time. To make a condition we need parameters that could be Boolean, float, trigger, or integer. We could set them up in the window 1 from the image above in the animator editor. To update those parameters, we need to use them in code through the animator API.

4.2 Animation Module

Unity provides us with a good API for an animator to update the parameters, but often happens that we must have some logic with our animations: disable animation transition for a jump animation, enabling root motion, activating particles in the middle of animation and so on. Also, the animation could be triggered for any class of a character, so it is important to have it organized and try to separate it from the logic part as much as possible.

For that reason, I decided to create two classes: one class for directly connect the animator parameters with the code, and one facade that connects this class with the rest of the systems and have some animation logic if that is needed. In figure 22 we can see it as a simplified version of a layered architecture: we have the persistence layer and database layer, but instead of database it is working with the animator.

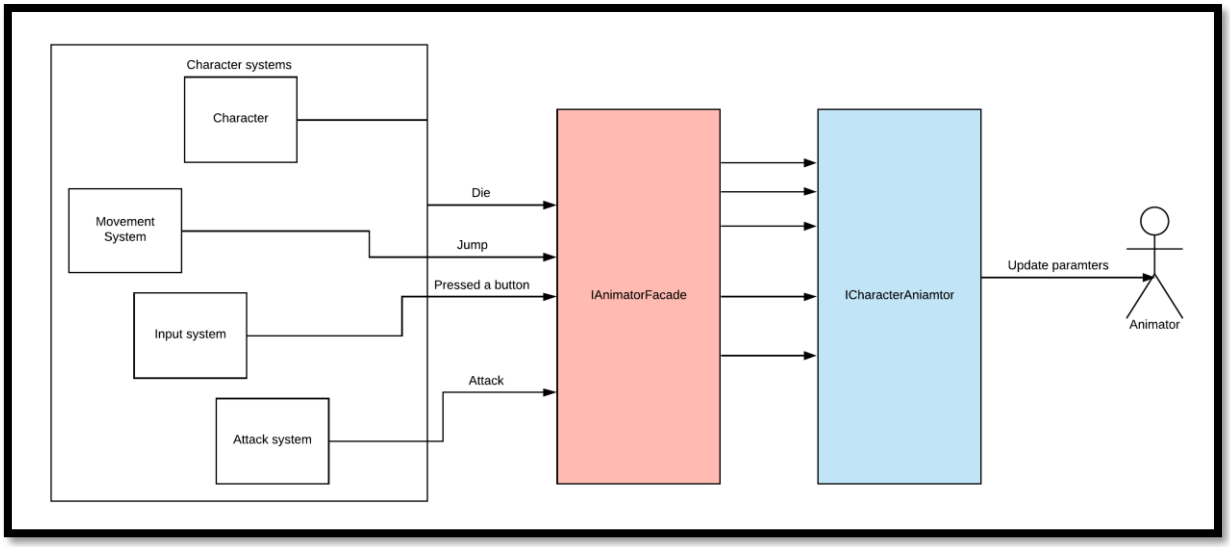

Figure 22 Animator architecture

That structure has many advantages:

- **Easy to test -** it is much easier to create tests for independent systems and we can be surer about each functionality
- **Consistency –** each layer is doing only what that layer is supposed to do, it helps us be more organized and be more consistent across all the layers.
- **Easy to change –** when everything is separated by layers it is much easier to change something in one layer and be sure that it would not affect the layer below it. But we must be sure when we are changing the bottom layer it will affect the topper lever, that is why we need to get usage of integration tests.
- **Easy to expand –** when we will need to add new animation logic it will be clear of how to implement it without major side effects.

IAnimatorFacade could be a normal class but ICharacterAnimator should extend the unity object API class – MonoBehavior because it is getting the animator directly through the game object.

However, sometimes there is a need to offer the heavy-based functionality to the animators so they could make the logic in the editor or get some data from the animator. There are two options to do it:

1. **Extend ICharacterAnimator** – If we already are working with the animator, we could extend the basic functionality of this class so it could have some logic.
2. **Created a second system** – create a system that would work directly with the animator offering heavy functionality for both ends: animator and the rest of the systems.

Each option has its advantages: extending ICharacterAnimation would not only be better in terms of optimization - we would have fewer monobehaviours and code in general, and structural -we would have only one class that is working with the animator so we will always know where the problem is. Or the second approach - to create a second system which has no advantages in terms of optimization but, in my opinion, it is a better solution for the overall structure of the project. I decided to sacrifice a little bit of performance for a cleaner solution. ICharacterAnimation would continue working only with parameters of the animator and a new system would oversee the persistence layer that has to be close to the animator itself.

IAnimatorStateFacade is this system, it is working not only with the animator but also with the rest of the systems. That was the main reason to take this solution over the other.

That allows us to separate working with the animator. To demonstrate it I am going to discuss the first problem I faced that led to reorganizing the animation system.

Typically, the character's movement is done in the code because that way we have much more freedom to customize it and change every detail, while the animation is being played on the model. This way we decide how, where and when the character moves. That is a good solution for generic walking, running and other primitive types of movements. However, sometimes we need our animation to move the character. That is what is called RootMotion. What it does is moving the mesh along with the animation. Every parameter, such as scale, speed, timing and so on, is importing from the animation. That is a good option for short, barely interactable animations – cut scenes, attacks, interacting with items and so on. Although, with the structure of game objects of a character there is a little problem with it.

```
1. public class ThirdPersonCameraDirection : IMovementDirection {
2.         public Vector3 getDirection() {
3.             Vector3 forward =
   CameraManager.getCameraForwardDirectionNormalized();
4.             Vector3 right =
   CameraManager.getCameraRightDirectionNormalized();
5.             return forward * InputManager.getVerInput() + right
   * InputManager.getHorInput();
6.         }
7.     }
```

In a normal movement state, we determine directional vector by inputs and the camera rotation and then we calculate direction based on these parameters. However, in RootMotion movement state we cannot do that because we cannot control the speed and position of the character, we must check every frame for the position of a character in the animation.

It means that now an animator controls the movement state during RootMotion animation. We could create that functionality in ICharacterAnimator and connect it to the

movement system but, in the end, for the sake of more separated logic I decided to sacrifice the performance in this area.

```
1. private void OnAnimatorMove() {
2.       makeSureCharacterIsNotNull();
3.       var movement = character.getMovement();
4.       if (!animator || !(movement is AttackingMovement) ||
   !(movement is IRootMotion))
5.             return;
   ((IRootMotion)movement).setRootMotionAdditionalPosition(animator
   .deltaPositio);
6. }
```

5 Combat

Combat is very important for a majority type of games; it is not only important to implement the system by its own but also to provide good feedback to the player of the actions that he is doing. They are many ways we could do the combat system, but I implemented the simpler one yet covering a lot of combat styles. The character could do fast and strong attacks, mixing them in between. With only 6 animations there will be a total of 8 different combos with a maximum of 3 attacks in a row:

- FAST-FAST-FAST
- FAST-STRONG-STRONG
- FAS-STRONG-FAST
- FAST-FAST-STRONG
- STRONG-FAST-FAST
- STRONG-FAST-STRONG
- STRONG-STRONG-STRONG
- STRONG-STRONG-FAST

5.1 Animations in Combo Attacks

To activate a combo attack we use InputManager which calls the combat system – AttackManager. It checks wherever the character is in a state he could perform his attack (GroundMovement or AttackingMovement) and changes it to AttackingMovement if needs to. Then tells IAnimatorFacade to trigger the animation depending on the attack (fast or strong). And because the attack animation is a number one feedback for the player to feel free in the combat situation we have to not hard-code all the attack animation behaviour, but rather give freedom to the animator to modify the animation itself and the combo system.

Combo system works the following way:

1. Character starts his first attack
2. Combo is started

3. Each animation has its limit of when the combo could be continued (Combo gap)
4. If the player performs another attack while the animation has not reached its end of combo gap, the combo continues with a different attack
5. If the player did not press a button in time or he did nothing at all, combo resets and the next attack will be the first one.

As we can tell it is important to change the combo gap without changing the code because we want to make sure that each attack is unique and works as a player wants it to work.

We can put on each animation a trigger which will call a method of IAnimatorStateFacade to reset the combo, as shown in figure 23. That is how we can determine on what frame of the animation the combo gap is finished.

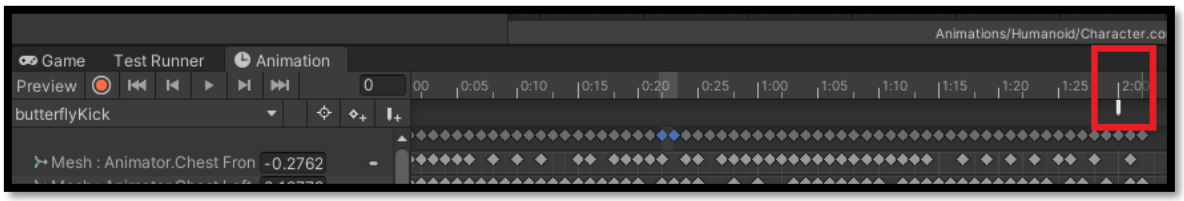

Figure 23 Trigger of the End of the Combo Gap

In the end, we have three major frames:

- The first frame – where the characters start his attack and enter the combo gap. This starts when a transition to this animation starts, not the animation itself.
- The frame of the end of the combo gap – triggers the method to resets a combo score. We must be careful with the transition to the next combo attack while a character is in the combo gap – the transition must end before the trigger.
- The last frame – the combo was already reset and the animator transitions to the upper level (standard non-attack animations).

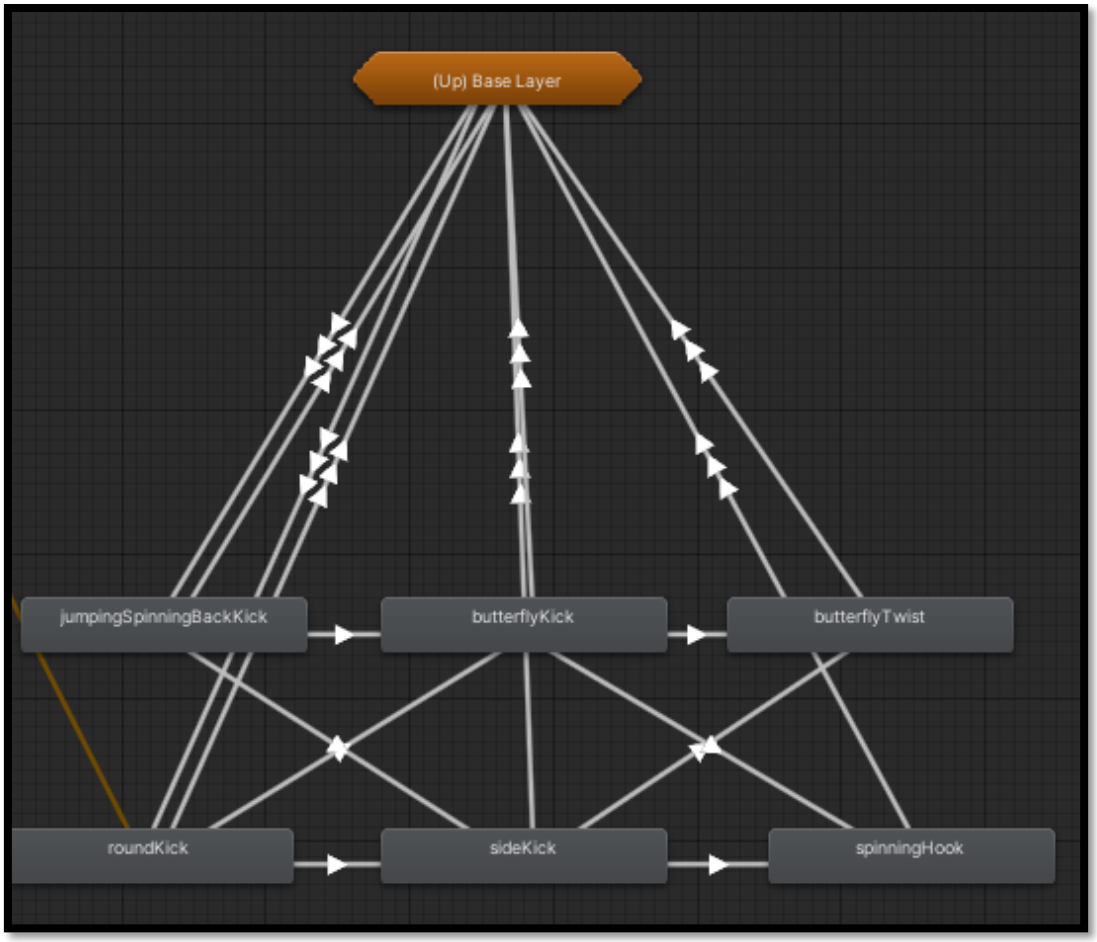

Figure 24 Combo state machine

Inside of our animation state machine, we can define other state machines. A combo system is a great example of that. We can encapsulate all the states into another state machine and use it as a normal state in the main state machine, as shown in the image above. As we can see in the combo state machine, we allow the animator to define the combo by simply creating the transition between the animations. However, the animator still must put the end gap trigger and create a DamageInfo – a structure that only contains the amount of damage and the damage type and assigns to the animation.

To assign the damage to the animation I created an AnimationStateBase script that extends StateMachineBehavior from Unity API. This script is being placed directly on each animation and has three lifecycle states:

1. **OnStateEnter** – executed on the first frame
2. **OnStateUpdate** – executed each frame of the game
3. **OnStateExit** – executed on the last frame

That gives us a lot of options to create different behaviours for different animations. And more so we can explore it more and be able to add as many animation states to a single animation as we want. To give the animator more freedom another class was

created - AnimationStateData the has all those life cycles and is ScriptableObject. That means that we can create it as an asset and give the animator possibility to change the values without touching the code and still be able to put it to the animation.

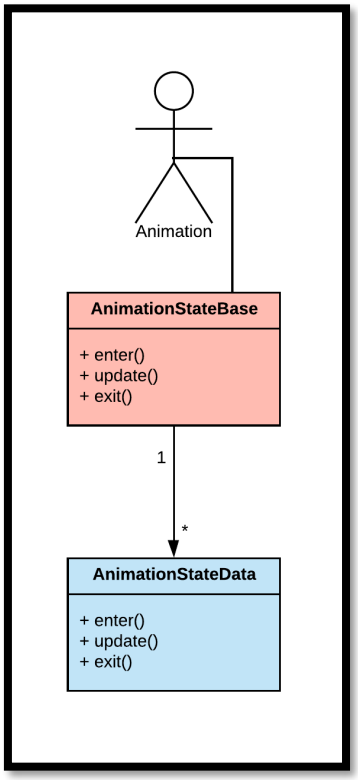

Figure 25 Animation State Data

AnimationStateBase has a list of AnimationStateData and executes every state for all of them. By creating AnimationStateData we can define behaviours for animations and an animator can put those behaviours to the animation and configure parameters without touching the code. DamageInfo is one of the examples – in the first frame, it creates damage info with the values that animator provided. Another example is Unskippable – some of the animations could not be skipped until they are finished. By putting this state on the animation, it makes it unskippable automatically.

5.2 Registering a Hit

They are two types of colliders in video games: physical and a trigger. Physical allows an object being placed in the world. It defines the edges of the object and would not allow it to be overlapped by another object. A trigger collider registers that another object has overlapped it. There are many cases for triggers, but we are going to focus on a very important one – hurtbox.

Hurtbox and hitbox are essential parts of registering a hit. Hurtbox is trigger colliders that register a hit from the hitbox. Let us imagine a sword and head of an enemy. A sword

would have a hitbox collider that will determine if it touched the head of the enemy – hurtbox.

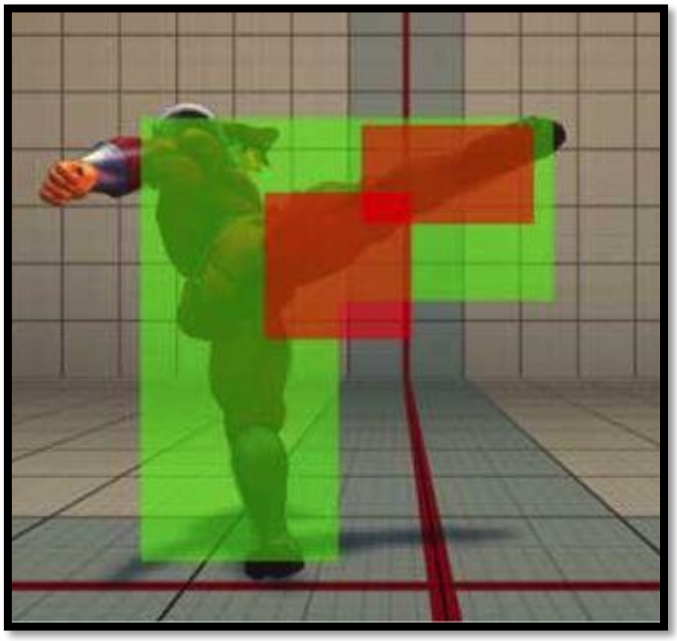

Figure 26 Hurtbox and hitbox in Street Fighter

In this image from Street Fighter, we can see red rectangles that represent hitboxes and green rectangles that represent hurtboxes. As Nahuel Gladstein explains in blog Gamasutra: "A Hitbox is an invisible box (or sphere) that determines where an attack hits. A Hurtbox on the other side is also an invisible box (or sphere), but it determines where a player or object can be hit by a Hitbox"[8]. If a hitbox overlaps with the hurtbox of the enemy that means, there is a hit, and vice versa for the player. It is one of the things that define balance in the video game, so it is important not only to create them properly but also to allows game designers to modify it without touching the code.

Hurtbox is straightforward to implement. We need to create a hurtbox game object with the trigger collider and a hurtbox script as a child of the main game object. This way hurtbox would always be in the correct place but still could be modified for a specific animation. Having many hurtboxes for a single character not only helps us to be more precise with the registering hits but also it allows us to create different behaviours for every hurtbox. Usually, in video games, a hit to the head deals more damage than to the body.

Hitboxes, on the other hand, is a little bit complex because they must be disabled while the character is not performing the attack. And during the animation, it should be enabled and could be modified. On the surface, the implementation looks like hurtbox – a game object under the character game object with a trigger collider and a script. But to move it along with the attacks we should put it under the bone of body part which should be dealing with damage (swords, fits, legs, etc..) depending on the animation.

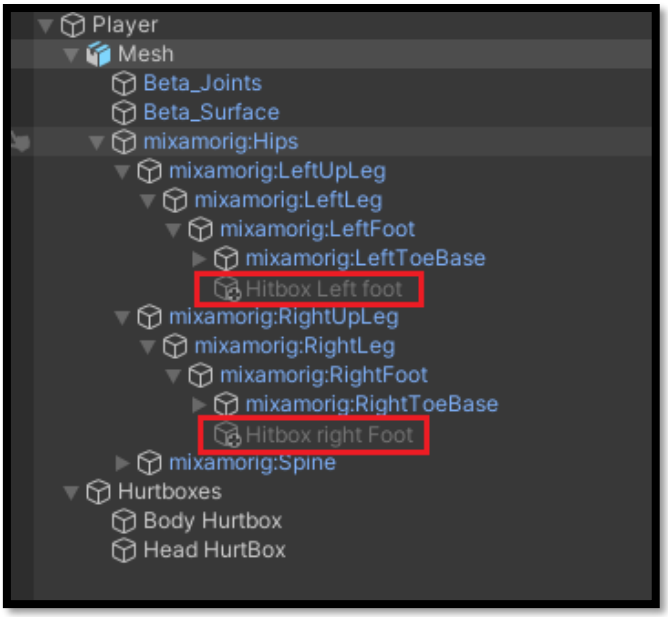

Figure 27 Leg hitboxes

In the image above we can see that under each foot we have a hitbox game object, so it moves always along with the feet. We can see that they are disabled because they must activate only in the animation.

Jonathan Cooper, the animator in Naughty Dog, breaks down the attack animation into three main stages of the attack animation: anticipation, attack, and recovery[2]. Anticipation phase is where a character is preparing for the attack. The attack phase is when the actual contact could happen, and the recovery phase is when a character recovers from the attack to a normal state. It is important not only to follow those principles while creating the animation but also enable/disable hitboxes according to those stages – hitbox should be enabled only during the attack stage. Thankfully to Unity, the editor of animation allows not only move/rotate bones, but we can change every parameter of the character during the animation. That is being said, the animator can not only enable and disable the hitbox but also manipulate the trigger collider of it. It allows us to make hitbox bigger during a specific frame to create a special effect and change the in game balance.

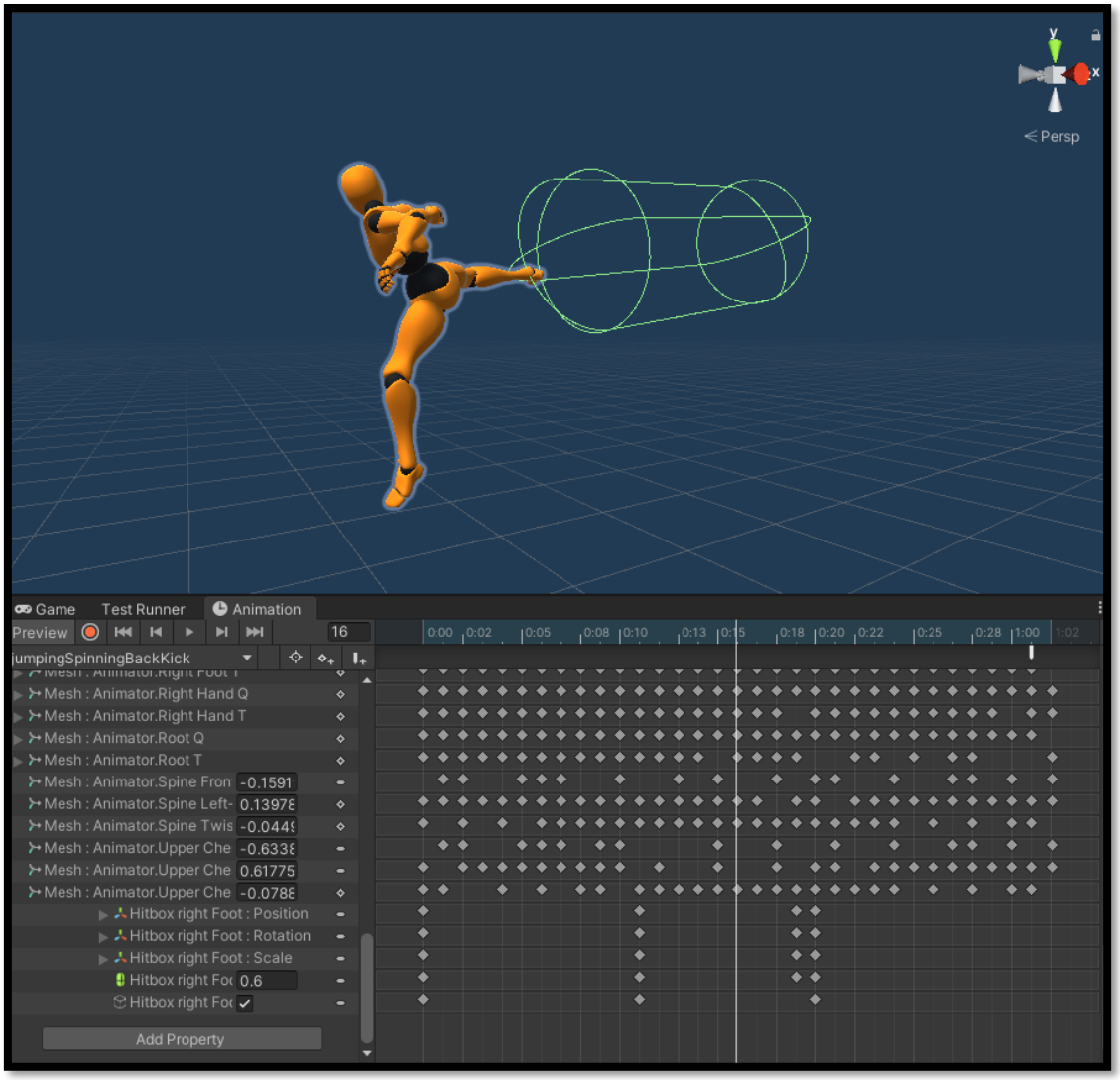

Figure 28 Hitbox during the animation

In the image above we can see a scene view and an animation view above it. In the animation view on each frame, we manipulate the bones to move the character and below all the bones we could see the hitbox parameters. On the first frame we disable the hitbox game object and it the attack phase of the animation we enable it back and making it bigger in specific frames. In the scene view, we can see a character during the animation and a big green cylinder – a trigger collider for the hitbox, that way we can see how the collider behaves on each frame.

When a hit is registered, hitbox passes the hurtbox to the AttackManager which is responsible to deal damage, depending on the current animation, to the hurtbox.

```
1. public void dealDamage(IHurtbox hurtbox) {
2.       hurtbox.takeDamage(currentDamageInfo);
3. }
```

Chapter 5

Testing and Documentation

This chapter presents the development of the tests for the framework together with the documentation in Github.

1 Testing

Testing is important for every software and video games are no exception, but they are not that easy to test. We must test not only the code but the game world itself.

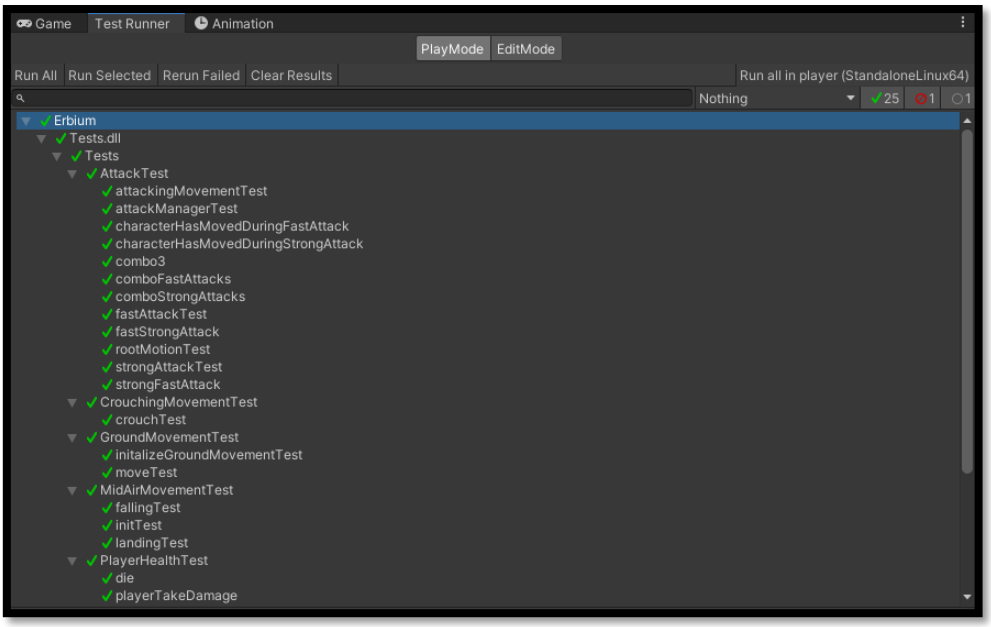

Figure 29 Unity test window

Unity provides two testing environments: play mode and editor mode. In editor mode, we can test only the code. In play mode for each test, unity creates a world and game objects. That is being said in editor mode we can create unit tests for small methods, but it is difficult to create integration tests because all the code that we write mainly manipulates the game objects. Almost all the methods used in some way or form use game object or in game components which leads to shifting to more test being placed in the play mode environment.

In play mode, we have a setup method that is executed before every test and a teardown method that is executed after the test is done. In the setup method, we must instantiate everything a world needs to perform the test: character, managers, floor, enemy, etc.... And in the teardown, we must eliminate all of them because if we do not do, it would lead to unpredictable behaviours when the tests change, especially all the singletons and destroyable objects.

```
1. [SetUp]
2. public void setUpTestScene() {
3.     gameObjects = init();
4.     GameObject inputManagerGo = initInputManager();
5.     InputManager  inputManager =
   inputManagerGo.GetComponent<InputManager>();
6.     playerGo = initPlayer(inputManager);
7.     gameObjects.Add(inputManagerGo);
8.    gameObjects.Add(playerGo);
9.    player = playerGo.GetComponent<IPlayer>();
10.  }
```

In play mode, we can check not only our code but also, which is most important, how the game objects and its components behave in certain conditions. For example, to check the landing of a character we must:

- Put our character above the ground.
- Check if his movement state is MidAir.
- Wait half of a second.
- Check if the character is about to land (for that it uses a special ray cast above the player to check if there is a ground above him).
- Check if the state is still falling.
- Wait for another second.
- Finally, check if the character is on the ground and the movement state is now GroundMovement.

As we can observe, the tests in the play mode could get difficult to write pretty fast but that is the most proficient way to test the mechanics and even help to find adequate parameters for the game.

In the testing mode, we cannot use players or camera input, but the players have some of the movement functionality that heavily depend on the input. For that and other cases we need to mock some of the code. Sadly, unity does not provide a library for mocking in tests but Nsubstitute works well with Unity and has no issues with adding it to the project. This library allows us to mock the classes. For example, if a method A and returns a random number, we can use the library for mocking this method and it will always return 5. It allows us to mimic methods and test only the part of the code that we need.

```
1. [UnityTest]
2. public IEnumerator moveTest() {
3.    IMovementDirection moveDirection = For<IMovementDirection>();
4.    var direction = new Vector3(0, 0, 1);
5.    player.changeMovementDirection(moveDirection);
6.    moveDirection.getDirection().Returns(direction);
7.    Vector3 playerInitPosition = playerGo.transform.position;
8.    Assert.True(player.getMovement() is GroundMovement);
9.    yield return new WaitForSeconds(1f);
10.   Vector3 expectedPosition = playerInitPosition + direction *
   (player.getStats().speed * 1f);
```

```
11.   Assert.True(Vector3.Distance(expectedPosition,
   playerGo.transform.position) <= 2f);
12.    Assert.True(player.getMovement() is GroundMovement);
13.  }
```

One of the advantages of dividing the framework into a different system is that it is easy to test every system independently. In cases where systems are depending on each other, using NSubstitute help us to achieve that by mocking the functionality that we need.

For each movement type there are at least for tests, one for every lifecycle:

- Test for Setup
- Test for movement
- Test for CleanUp
- Test for Changing

Some of the movement types need additional tests: attacking movement for disabling input, mid-air movement for landing and others.

Combat has a lot of functionality and is not easy to test due to combo gaps. There are tests for every combo and single fast/strong attack. While a character is attacking, as we have seen previously, it uses Root Motion for moving. We must test if the character has moved during the attack and if the hitboxes were activated.

Animations are tough to test because we can't test the animation that is being played at the moment, but we can test all the animation parameter, attached animation systems and when the transition is happening. However, the most important tests are combat-related because it is not only using animations with logic in the code, but also the most important animations for the game. There are tests for each movement to check whenever the transition has happened and root motion tests for the attacks.

The character system is easy to test, however it has one of the important tests. We have to make sure that by putting it in the game world, it would have all the subsystems attached and working.

2 Documentation

The purpose of this project was to create a framework for providing characters and their mechanics that helps other developers to easily build their video games. But the code does not mean anything without documentation and user manuals. It is an important part of the project because writing good documentation is what can separate a good framework from a bad one.

I had three ideas where create documentation: a pdf file, a website, or a GitHub wiki. While a pdf file could be accessed online and offline, I think that it did not provide the best experience for that. A website could be beneficial for the design purpose and

organization, but it requires resources to host it and time for building it. Github wiki seemed like a perfect solution- it is fast to do due to markdown markup; it is close to the source code and we can add a side menu along with the footer.

The structure of the documentation is not like java-doc where every method is explained. It describes the overall structure and the principle flows of the framework. Each system has a related block with pages that explain the main part of the corresponding systems. There is not only the code explanation but also the ideas behind creating this system in this specific way.

For the expandable modules, they are guides on how to expand this system with guidelines to match the same style and ideas of the framework.

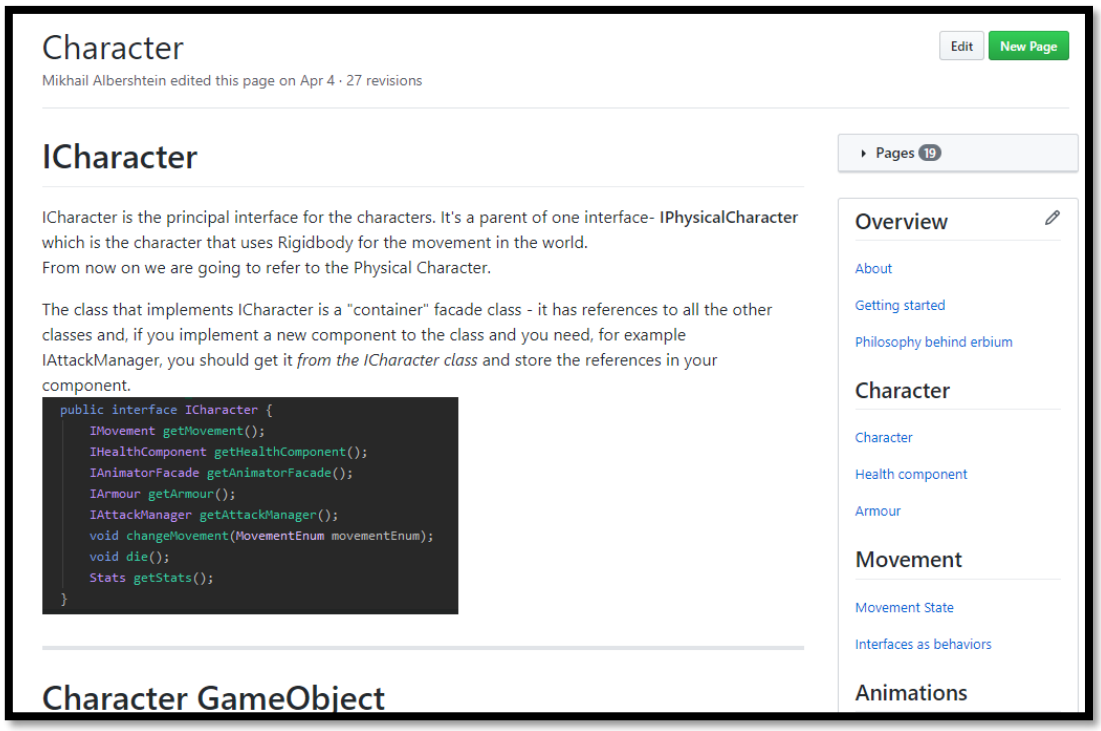

Figure 30 Github wiki page

The general rule was to create an easy to follow guide about the project. For that, I decided the best approach would be put a lot of images that show the small parts of the code describing the purpose and where and how it is being used, not what it does. That helped me reduce the number of words which made it easier to read, without sacrificing much of the content. If the code or method has a weird look then I would go deeper into the explanation why it was written like that.

Chapter 6

Conclusions

The main objective of the framework was to provide a structure that would allow developers to quickly create and test their new mechanics. The principal focus was on the connection of movement animations and attacking to the character because in general, this is the most important area and the most demanding area to be customized.

It was a time-consuming project, and, on many steps, I encouraged a lot of problems with many ways to solve them. The hardest part was to pick one solution over the another because it is relatively easy to implement as many design patterns as you know, but as always in architecture and clean code - we have to find a balance between hardcoded solutions and overwhelming structures.

1 Achievement of Objectives

At the beginning of this document I defined three main objectives, besides future work, for this project:

- **Create a framework for creating 3rd person character for unity** – Although I did not release the framework in Unity Asset Store yet, so I do not have other people's experience feedback, I performed my experiment. I had an assignment for Animation and Design of Videogames by Muñoz García, Adolfo, where I needed to create a 2D game and I used Erbium. I adjusted easily, without consuming relevant time, the code to my needs. Specifically, I spent an hour of work to transform it from 3D to 2D and then another hour to create unique mechanics for the character. It worked just fine. Besides translating it into the 2D world, I built my code around the codebase.
- **Testing** – I have never used tests before in video games, so I had to investigate what is the difference between the normal software tests and video games tests. It turned out that in video games testing are a little different from what we see on the regular base. We still have normal unit and integration tests for our code but on the other hand, we still need to test the game by its by with creating test worlds. Testing allows us to develop the software much faster and much more secure and games are not exceptions. In total there were 27 integration tests created for all the systems. Every time there is a new feature, tests are going to make sure that each system is working properly. By implementing tests, the development process got easier because it told me if I had a bug early when it is still easy to fix.
- **Documentation** – Every framework or library needs documentation. Especially when structures are becoming more and more complex. Even though design

patterns are made for more stable and easier code, without the proper knowledge and documentation it is difficult to understand code. Erbium does not have a super complicated system although it still needs good documentation. Github has provided us with a good platform for creating documentation for our project, which has not been only easy to make but it is one click away from the codebase

2 Development Process

Thanks to the knowledge about the agile methodology learned at the degree in Informatics Engineering at the Polytechnic University of Valencia, the development process was not frustrating or estimated badly, even though I did not have a team, many agile principals could be beneficial for solo projects. Mainly I used Trello for sticking to the plan of the development. It did not only allow me to control my time limits but also to track down the bugs. In the end, 17 bugs were tracked and fixed. However, the estimation on the writing code and creating the documentation was 80%/20% but because I had no experience writing in markdown style the time it took to write documentation was larger, around 70%/30%.
Tests are another great thing that helped the process. Once I finished implementing a feature, I could be sure that it did not break anything.

3 Future Work

Erbium is the base framework and I want to expand it with different common modules such as the ability for wall-running, aim system, swimming and so on. I will not add them to the main framework, instead, the idea is to create a bunch of compatible modules with the framework itself and with each other. I have already the module ideas on the Trello page of this project with some rough estimations. They will not violate the principles established at the begging: being expandable, customizable, and clean overall. They are principles that would be the main priority for those modules. However, the first thing, before developing modules, is releasing it on the Unity Asset Store, receiving feedback, and fixing the problems that would be encouraged.

Glossary

- **Animator** – Unity editor for animations
- **API** – Application Public Interface
- **Cryengine** – Game engine by crytech
- **ECS** – Entity Component System
- **GameObject** – Any object in the Unity world
- **GC** – The Garbage Collector
- **Getter** – A method for getting a reference to an object
- **Godot** – Game engine by Juan Linietsky, Ariel Manzur
- **MonoBehavior** – A class from unity API that could attach to the game object
- **Unity** – Game engine by Unity technologies
- **Unreal Engine 4** – Game engine by Epic Games

Bibliography

[1] Ars Staff. (2016, September 27). Unity at 10: For better—or worse—game development has never been easier. Retrieved June 17, 2020, from Ars Technica website: https://arstechnica.com/gaming/2016/09/unity-at-10-for-better-or-worse-game-development-has-never-been-easier/

[2] Cooper, J. (2019). GAME ANIM. ISBN 1138094870

[3] Damm, R. (2019, October 14). Introducing the new input system. Retrieved June 17, 2020, from Unity Technologies Blog website: https://blogs.unity3d.com/2019/10/14/introducing-the-new-input-system/

[4] Edelstein, S. (2018, May 17). How gaming company Unity is driving automakers toward virtual reality. Retrieved from Digitaltrends.com website: https://www.digitaltrends.com/cars/unity-automotive-virtual-reality-and-hmi/

[5] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Boston, MA: Addison Wesley Professional. ISBN 0201633612

[6] Fine, R. (2017, August 11). UnityScript's long ride off into the sunset - Unity Technologies Blog. Retrieved June 17, 2020, from Unity Technologies Blog website: https://blogs.unity3d.com/2017/08/11/unityscripts-long-ride-off-into-the-sunset/?_ga=2.53872556.506042004.1530922883-2145738851.1524228284

[7] Hargreaves-MSFT, S. (n.d.). Twin paths to garbage collector nirvana. Retrieved June 21, 2020, from Shawn Hargreaves Blog website: https://web.archive.org/web/20190216001802/https://blogs.msdn.microsoft.com/shawnhar/2007/07/02/twin-paths-to-garbage-collector-nirvana/

[8] Gladstein, N. (2018, May 14). Hitboxes and Hurtboxes in Unity. Retrieved June 17, 2020, from Gamasutra.com website: https://www.gamasutra.com/blogs/NahuelGladstein/20180514/318031/Hitboxes_and_Hurtboxes_in_Unity.php

[9] Meijer, L. (2019, March 8). On DOTS: Entity Component System. Retrieved June 17, 2020, from Unity Technologies Blog website: https://blogs.unity3d.com/2019/03/08/on-dots-entity-component-system/

[10] Martin., R. C. (2003). The Principles of OOD. Retrieved June 17, 2020, from Butunclebob.com website: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

[11] Programming guidelines - HaskellWiki. (n.d.). Retrieved June 21, 2020, from Haskell.org website: https://wiki.haskell.org/Programming_guidelines

[12] Unity Technologies. (n.d.-a). Unity - Scripting API: CharacterController. Retrieved June 17, 2020, from Unity3d.com website: https://docs.unity3d.com/ScriptReference/CharacterController.html

[13] Unity Technologies. (n.d.-b). Unity Asset Store Character Controllers. Retrieved June 17, 2020, from Unity.com website: https://assetstore.unity.com/packages/templates/systems/ultimate-character-controller-99962?q=character%20controller&orderBy=0

[14] Wikipedia contributors. (2020a, April 29). Unity Technologies. Retrieved June 17, 2020, from Wikipedia, The Free Encyclopedia website: https://en.wikipedia.org/w/index.php?title=Unity_Technologies&oldid=953866746

[15] Wikipedia contributors. (2020b, June 10). Unity (game engine). Retrieved June 17, 2020, from Wikipedia, The Free Encyclopedia website: https://en.wikipedia.org/w/index.php?title=Unity_(game_engine)&oldid=961750756