jamk.fi

Object-oriented programming with Unity

Inheritance versus composition

Timo Holopainen

Bachelor’s thesis
May 2016
Technology, communication and transport
Degree Programme in Software Engineering

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Description

Author(s) Holopainen, Timo	Type of publication Bachelor's thesis	Date May 2016
		Language of publication: English
	Number of pages 82	Permission for web publi-cation: x
Title of publication **Object-oriented programming with Unity** Inheritance versus composition		
Degree programme Software Engineering		
Supervisor(s) Nelimarkka, Paavo; Hämäläinen, Raija		
Assigned by		

Abstract

The main objective of the thesis was to study different object-oriented solutions for game development with Unity game engine using traditional object inheritance together with entity-component-system, a principle that concentrates on the creation of components from the object's attributes and functionalities.

Different solutions were implemented and tested in a game prototype called Quantum Knight that was built using the Unity game engine and C# programming language. The solutions focus on the design and actualization of the prototype's characters and game mechanics, especially on the prototype's turn based battle system. The thesis also covers other parts of the prototype's implementation such as dialogue and event systems.

The game prototype provides two distinct solutions for the character objects. The first solution utilizes the entity-component-system completely by distinguishing the player character from the non-player characters simply with the use of different components. The second solution is featured in the characters used in the battle system, in which they are all similar entities controlled by a greater manager entity. A third solution implemented for the skills that characters use in the battle combines inheritance with composition by deriving new specialized component scripts from existing custom scripts while utilizing them in the same way.

The results were examined by the terms of usability and versatility. The thesis does not provide any data on the performance of the solutions as it would need another research on its own.

Keywords/tags (subjects)

Unity, object-oriented programming, inheritance, entity-component-system, composition

Miscellaneous

Appendix: Screenshots and scripts from the prototype, 30 pages

jamk.fi

Kuvailulehti

<table>
<tr><td rowspan="3">Tekijä(t)
Holopainen, Timo</td><td rowspan="2">Julkaisun laji
Opinnäytetyö, AMK
Sivumäärä
82</td><td>Päivämäärä
Toukokuu 2016</td></tr>
<tr><td>Julkaisun kieli
Englanti</td></tr>
<tr><td></td><td>Verkkojulkaisulupa
myönnetty: x</td></tr>
<tr><td colspan="3">Työn nimi
Object-oriented programming with unity
Inheritance versus composition</td></tr>
<tr><td colspan="3">Tutkinto-ohjelma
Ohjelmistotekniikan koulutusohjelma</td></tr>
<tr><td colspan="3">Työn ohjaaja(t)
Nelimarkka, Paavo; Hämäläinen, Raija</td></tr>
<tr><td colspan="3">Toimeksiantaja(t)</td></tr>
</table>

Tiivistelmä

Opinnäytetyön päämäärä oli tutkia erilaisia oliokeskeisiä menetelmiä peliohjelmoinnissa Unity pelimoottorilla käyttäen apuna olioiden perinnöllisyyttä ja entiteetti-komponentti-systeemiä, joka perustuu olioiden ominaisuuksista ja toiminnallisuudesta rakennettuihin komponentteihin.

Ratkaisut toteutettiin ja testattiin Unityllä ja C# ohjelmointikilellä rakennetun Quantum Knight peliprototyypin kehittämisen yhteydessä. Ratkaisuissa keskityttiin prototyypin pelihahmojen ja –mekaniikkojen suunnitteluun ja toteuttamiseen, erityisesti prototyypin vuoropohjaisen taistelusysteemin toimintaan. Opinnäytetyö käsittelee myös muiden pelimekaniikkojen, kuten dialogien ja tapahtumien, toteutuksen.

Peliprototyyppiin toteutettiin kaksi erilaista ratkaisua pelin hahmoja varten, joista ensimmäinen käyttää entiteetti-komponentti-systeemiä erottamalla pelaajahahmon muista pelin hahmoista yksinkertaisesti käyttämällä erilaisia komponentteja niiden luomisessa. Toista ratkaisua käytetään pelin taistelusysteemissä, jossa samanlaisia hahmoentiteettejä ohjaa erillinen hallintaentiteetti. Taistelussa käytettäviä hahmojen kykyjä varten käytettiin kolmantena tapana luokkien perinnöllisyyttä komponenttien kanssa perimällä uusia komponentteja omatekoisista komponenttiskripteistä.

Ratkaisuja tarkasteltiin niiden käytettävyyden ja monipuolisuuden osalta. Opinnäytetyö ei siltä osin ota kantaa ratkaisujen tehokkuuteen, joka vaatisi oman tutkimuksensa.

Avainsanat (asiasanat)

Unity, olio-ohjelmointi, perinnöllisyys, entiteetti-komponentti-systeemi, kompositio

Muut tiedot

Liitteenä kuvankaappauksia ja ohjelmakoodia prototyypistä, 30 sivua

Contents

1 Introduction 6

2 Object-oriented programming 7

2.1 Classes and objects 7

2.2 Attributes and methods 7

2.3 Most important features of object-oriented programming 8

2.3.1 Encapsulation 8

2.3.2 Abstraction and generalization 9

2.3.3 Inheritance 9

2.3.4 Polymorphism 10

3 Entity-component-system 11

3.1 Entity 11

3.2 Component 12

3.3 System 12

3.4 ECS in game development 12

4 Unity 14

4.1 The graphical environment 15

4.1.1 The Game View 15

4.1.2 The Hierarchy Window 16

4.1.3 The Inspector Window 17

4.1.4 The Project Window 18

4.1.5 The Scene View 18

4.2 Programming with Unity 19

4.2.1 GameObject and components 19

4.2.2 Prefabs 20

4.2.3 Scripts 21

5 Project Quantum Knight 23
5.1 Story 23
5.2 Gameplay 24
5.3 Dialogue 24
5.4 Battle mechanics 25
5.4.1 Battle modes 25
5.4.2 Turn sequence 26
5.4.3 Combatants 26
5.4.4 Actions 26
6 Implementation 28
6.1 Player character 28
6.1.1 Player controls 29
6.1.2 Animations 31
6.1.3 Non-player characters 33
6.2 Game progression 33
6.2.1 Dialogue 33
6.2.2 Events 34
6.2.3 Scenes 35
6.3 Battle system 36
6.3.1 Objects in battle 38
6.3.2 Turn sequence 39
6.3.3 Movement 40
6.3.4 Attacking 41
6.3.5 Artifial intelligence 43
6.3.6 Skills 44
7 Discussion 48
7.1 Results 48

7.2 Credibility 49

7.3 Final thoughts 49

References 50

Appendices 52

Appendix 1. The player GameObject 52

Appendix 2. Non-player character 53

Appendix 3. Turn sequence 54

Appendix 4. Player Controls script 55

Appendix 5. Event Manager script 58

Appendix 6. Dialogue script 60

Appendix 7. Scene Changer script 62

Appendix 8. Battle Manager script 63

Appendix 9. Enemy AI script 65

Appendix 10. Base Skill script 74

Appendix 11. Damage Skill script 75

Appendix 12. Absorb Damage Skill script 78

Appendix 13. Skillbook script 79

Figures

Figure 1. Class diagram with inheritance 10
Figure 2. Class diagram for a platformer game 13
Figure 3. Class diagram for a platformer game with composition 14
Figure 4. The Game View 16
Figure 5. The Hierarchy Window 16
Figure 6. The Inspector Window 17
Figure 7. The Project Window 18
Figure 8. The Scene View 19
Figure 9. Example of a light object 20
Figure 10. Scripts as components 23
Figure 11. Dialogue window 25
Figure 12. Movement 27
Figure 13. Attacking 28
Figure 14. Colliders in the Scene View 30
Figure 15. Animation states for the player in the Animator Window 32
Figure 16. Sprite sheet of the main character 33
Figure 17. Dialogue script 34
Figure 18. BattleManager 37
Figure 19. Character Stats 38
Figure 20. Action panel 40
Figure 21. Selector 41
Figure 22. Enemy AI activity diagram 43
Figure 23. Skill class diagram 45
Figure 24. Skillbook and skill scripts 47

Tables

Table 1. Controls..31

1 Introduction

The main goal of this thesis was to find effective solutions using the entity-component-system ideology and to experiment the system together with object-oriented inheritance in game programming as well as improve the author's programming and object design skills in that field. Video game industry is one of the fastest growing sections in the entertainment industry in the 21st century. The functionality and efficiency of a game is an essential part of its success as it is for any program, and effective programming solutions in that field can also be for the benefit of software engineering field in general.

Unity game engine was chosen as a tool of choice for experimenting object-oriented game programming for this thesis because the author has previous experience with it, and the engine provides a free version for independent developers. Different object-oriented solutions were studied and implemented in a small project during the development of a game prototype using the Unity game engine and C# programming language.

Chapters 2, 3 and 4 of this thesis cover the basics of object-oriented programming, entity-component-system and Unity game engine, respectively. Chapters 2 and 4 use web-based and textbook sources for the background information while chapter 3 relies solely on material found on the web. The sources are reliable as most of them are official releases in their field. Especially the information from the Unity manual used for chapter 4, as it is published by the same author as the product that the manual instructs.

The fifth chapter defines the game design and mechanics of the prototype while the sixth chapter explains how the game mechanics were implemented in the project. Some of the scripts of the prototype can be found in appendices. Their basic functionalities are examined and most important code excerpts are shown and explained in chapter 6. Chapter 6 also offers alternative suggestions for the mechanics and the final chapter provides analyses for the solutions.

2 Object-oriented programming

Object-oriented programming, or OOP for short, is a programming paradigm based on the concept of objects. Object-oriented approach is used for depicting real world concepts in programming.

The most significant object-oriented programming languages include, but are not limited to, C#, C++, Java, Perl, PHP and Python.

2.1 Classes and objects

Two most important keywords in object-oriented programming are *class* and *object*. An object is a representation of a single individual of a greater group. A class represents the group and it is a blueprint from which objects are created. For example, a video game can have multiple characters of the same type, built from the same blueprints, but each of them eventually are different individuals. In object-oriented programming terms, one unique character would be an instance of a character class, an object in other words. (Lesson: Object-Oriented Programming Concepts 2015.)

2.2 Attributes and methods

Each class has *attributes* that represent the state of the objects derived from it. Attributes are also known as variables. The character class in the example above could have many attributes, such as a name, a height, a mathematical value for movement speed and so on. (Lesson: Object-Oriented Programming Concepts 2015.)

Methods, also called functions, determine an object's interaction with other objects and its environment. They operate on the object's attributes creating the desired functionality and enabling communication between the objects and the environment.

2.3 Most important features of object-oriented programming

2.3.1 Encapsulation

An object stores its state in its attributes. As the internal state of an object is critical to its functionality, it is protected with a restricted access to it by other objects. All interaction with the object is required to be done through the object's methods. This restriction is known as *data encapsulation*. An object's attributes are protected by giving them access specifiers. The specifiers determine the type of access and visibility to given attributes. A *public* attribute can be accessed by any other object or part of the program, while a *private* attribute is commonly used for encapsulation as it restricts the access and hides the visibility from all others than members of the same class. (4 major principles of Object-Oriented Programming 2005; Harwani 2015.)

Two basic methods that can access object's private state are called *accessor* and *mutator* methods. An accessor method is a method that asks the object about its state. Usually it gives information about a particular attribute or attributes of the object; the method in the below example returns the name of the object that has been created from the class. (4 major principles of Object-Oriented Programming 2005.)

```
public string GetName()
{
    return this.name;
}
```

A mutator method can change the objects state, while hiding the implementation of how it is done. The method below overwrites the name of the object after it has been initialized. (4 major principles of Object-Oriented Programming 2005.)

```
public void SetName(string n)
{
    this.name = n;
}
```

The accessors and mutators are commonly called *get* and *set methods*, respectively, and are often named after that fashion as the examples above demonstrate.

2.3.2 Abstraction and generalization

Abstraction is a philosophy that places the emphasis on hiding unnecessary details and using names to refer to objects. Abstraction reduces program's complexity by prioritizing the idea, qualities and properties rather than the particulars of a class, therefore it is essential in the construction of programs. *Generalization*, on the other hand, reduces complexity by integrating multiple similarly functioning concepts within a single class. (Introduction to Object Oriented Programming Concepts (OOP) and More 2015.)

For example, creating a generalized vehicle class would be a more effective way in contrast to creating a car class and a motorcycle class, because both classes share abstract characteristics such as a set number of wheels, maximum speed and a name of the manufacturer.

2.3.3 Inheritance

Sometimes two or more classes share some characteristics, however, are distinct enough so they cannot be generalized into a single class. *Inheritance* solves this problem. In this case one could create a *parent class* (also called a super-class) that is composed of the characteristics that are shared between the two classes, and then create two different *child classes* (also called subclasses) that inherit the parent class. This way the two child classes will share the parent class attributes and methods and they can be distinguished from each other by creating their own unique attributes and methods in them. (4 major principles of Object-Oriented Programming 2005.)

The character class in the previous example could be difficult to generalize if there are many specialized characters with different behaviors. This problem can be solved by creating a parent class with the attributes and methods that all the character objects would use. Then the specialized child character classes can be inherited from it. This solution is illustrated in the class diagram in Figure 1.

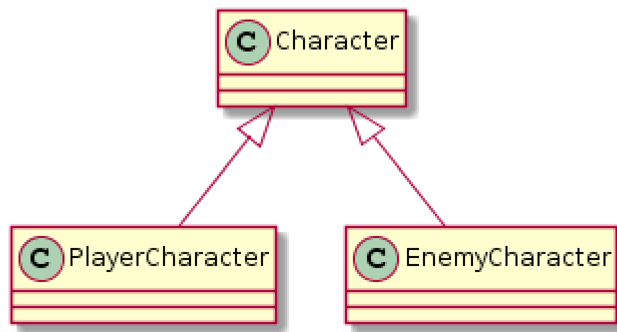


Figure 1. Class diagram with inheritance

Some object-oriented programming languages, such as C# and other .NET based languages, support only single inheritance, which means that a child class can only inherit one parent class. There are, however, languages that support multiple inheritance, such as C++. As the name implies, the multiple inheritance permits a child class to inherit one or more parent classes. (C# Inheritance 2008.)

2.3.4 Polymorphism

Polymorphism means the ability to have multiple methods with the same name, while they behave differently. Two basic principles that achieve polymorphism in object-oriented programming are *overloading* and *overriding*. (Introduction to Object Oriented Programming Concepts (OOP) and More 2015.)

A single class can have various methods of the same name when overloaded. This way there can be many implementations of the same method that function differently depending on the types of parameters they receive. The number of parameters and their types are used to distinguish the methods. For example, classes often have multiple constructor methods (methods that are called when an object is created from the class); one without the parameters and one with them as the example code below demonstrates. (Harwani 2015.)

```
public Character()
{
    this.name = "";
}

public Character(string n)
{
```

```
            this.name = n;
        }
```

This way, when a character object is to be created, one can use the method with the parameter and name the object as needed. Using the default constructor without the parameter will create an object with a default name value that has been initialized in the code.

Overriding methods is closely tied to inheritance. A child class can override a method that is provided by one of its parent classes. It means that the child can have a different implementation of the parent class' functionality, but with the restriction of having the same name and return types for the overridden methods. (Harwani 2015; Introduction to Object Oriented Programming Concepts (OOP) and More 2015.)

3 Entity-component-system

Entity-component-system (ECS), also called entity composition, is an architectural pattern commonly used in game programming. ECS is based on a principle called *composition over inheritance*, in which complex inheritance hierarchies that can cause design and implementation problems in large software projects, are completely forgotten and the objects are split into three pieces; *entities*, *components* and *systems*. (Entities and Components 2015; Understanding Component-Entity-Systems 2013; What's an Entity System? 2014.)

3.1 Entity

Entities represents real world concepts in programming just like objects, however, the idea behind them differs significantly from objects. As a complete object would include state and behavior, an entity contains only one unique ID that is used for reference. That said, entities do not have any functionality of their own other than being containers for components and systems. (Entities and Components 2015; Understanding Component-Entity-Systems 2013; What's an Entity System? 2014.)

3.2 Component

A component is a small part of a larger entity that stores the data of the entity and represents certain aspects of it. Most components are made to be universally applicable so they can be reused with different entities, only the single instances of a component are tagged with the ID of the entity they are in. Components do not provide any functionality for the entity, just the raw data although some architectures have coupled them with functions. (Entities and Components 2015; Understanding Component-Entity-Systems 2013; Unity: Now You're Thinking With Components 2013; What's an Entity System? 2014.)

3.3 System

A system generates an entity's functionality and behavior using the data stored in the components. Systems clue the pieces of the entity together, making the entity a functional structure. Ideologically they are separated from the data components, but they are sometimes merged together so the component-system hybrids operate on the data stored within itself and not in other parts of the entity. (Entities and Components 2015; Understanding Component-Entity-Systems 2013.)

3.4 ECS in game development

Most games have a wide variety of objects; from simple moving cubes to complex characters with artificial intelligence and animations. Creating unique classes of every single one of them requires a great deal of work, and can cause some of the classes to have same kind of functionality, possibly same code, in them. That is why entity-component-system pattern is highly regarded approach in game development, because it helps the game design and programming by getting rid of complicated inheritance hierarchies and avoiding repetitive code as the components and systems can be reused. (Avoiding the Blob Antipattern: A Pragmatic Approach to Entity Composition 2012.)

The example illustrates how the ECS works in game programming compared to traditional object-oriented programming with inheritance. In a basic two dimensional

platformer game the player is running and jumping across different platforms while dodging various enemies and traps. Four classes can be derived from these specifications: A player class, a platform class, an enemy class and a trap class.

There are two distinct types of objects; the static ones like platforms, and the dynamic ones like the player character and enemies. Platforms have no behavior, but they contain colliders that enable the player to jump and move on them. The player object has its own behavior that is linked to player input and the enemy object has predetermined behavior and movement patterns, including attacks that cause damage to the player. Now the problematic part is the trap object, as it should be both stationary and cause damage to the player when triggered, as depicted in Figure 2. As the trap object has features of the platform object and the enemy object, it can be difficult to determine how the inheritance hierarchy should be constructed without creating duplicate code within the classes. Furthermore, as the game itself expands, the hierarchy of the objects will become more and more complex and problematic.

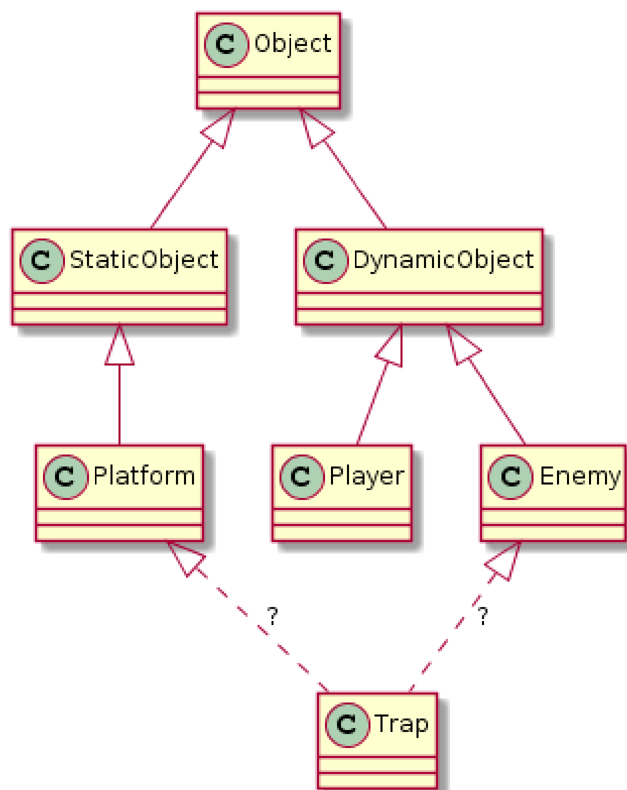


Figure 2. Class diagram for a platformer game

Using the entity-component-system pattern in designing the objects would provide much more flexibility. For this case, the four types of entities would be created based on the classes in Figure 2; a player entity, an enemy entity, a platform entity and a trap entity. Then, all the characteristics of the objects are split into components and systems. As for the components, the entities will need a component to indicate their position in the game. In a two dimensional game, the component can be as simple as a two dimensional vector containing the entity's x- and y-coordinates. The player's and the enemy's positions will change according to their input movement system and behavior system, respectively, while the platform and the trap will be stationary. Finally, all of the entities will need a solution for collisions. This is usually dealt with a collider component that holds the boundaries of the entity and collision check system that registers all collision happening to the collider it is linked to. In this case only the player and enemy entities need collision check, as they are the only dynamic entities. The class diagram with composition can be seen in Figure 3.

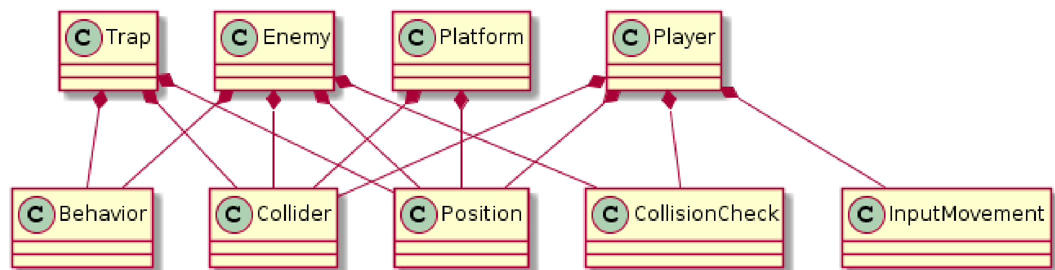


Figure 3. Class diagram for a platformer game with composition

Because the components and systems can be linked to any entity, composition grants great extensibility to object design. For example, one could create a moving platform simply by adding a movement system to the platform entity. Because of the easy-to-understand approach to object design and implementation, the entity-component-system is featured by default on some game engines. One of them is *Unity*.

4 Unity

> *Unity is a flexible and powerful development platform for creating multiplatform 3D and 2D games and interactive experiences. (The best development platform for creating games 2016.)*

Unity is a cross-platform game engine developed by Unity Technologies for the development of three and two dimensional applications. Its main feature is its ability to deploy a once built application to 23 platforms, including consoles and mobile devices. (Build once deploy anywhere 2016; The best development platform for creating games 2016.)

Unity supports different licenses, varying from the free Personal Edition for independent developers to Professional Edition with monthly fees and solutions for bigger enterprises, education and other industries. (Get Unity 2016; Unity, Source 2, Unreal Engine 4, or CryENGINE - Which Game Engine Should I Choose? 2015.)

4.1 The graphical environment

The Unity game engine is integrated with the Unity Editor, a desktop application for Windows and MAC OS X operating systems. The editor is visually driven and has tools for 2D and 3D physics, animation, audio, graphics and optimization. (A feature-rich and highly flexible editor 2016; Menard & Wagstaff 2015.)

4.1.1 The Game View

The Game View renders the game as it would look like in the final build of the project. This view is connected to a camera object that can be moved and rotated in the Scene View. Every change the user makes in the scene can be seen in this window if the main camera object is directed correctly. The game can be tested and played using this screen without the need of building it separately. The Game View can be maximized to full screen width to show the game in more detail. It can also show optional statistics when running the game, such as frame rate and processor load, as shown in Figure 4. (Menard & Wagstaff 2015; The Game View 2016.)

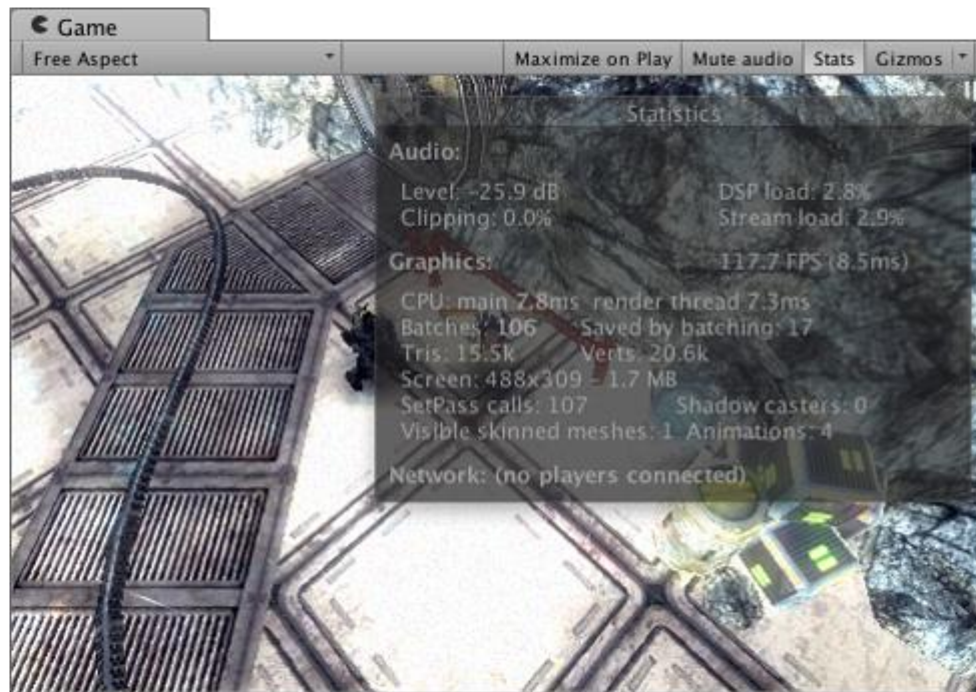


Figure 4. The Game View (The Game View 2016.)

4.1.2 The Hierarchy Window

The Hierarchy Window lists all objects in the current or multiple scenes. The objects can be dragged onto another to create a parental relationship between the two as seen in Figure 5. Selecting objects will show their information in the Inspector Window and they will be highlighted in the Scene View. New objects can be created by using the Create dropdown menu or right-clicking the list's area and selecting the type of an object wanted. (Menard & Wagstaff 2015; The Hierarchy Window 2016.)

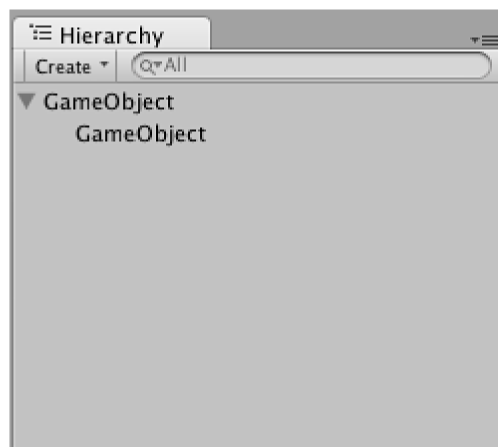


Figure 5. The Hierarchy Window (The Hierarchy Window 2016.)

4.1.3 The Inspector Window

The Inspector Window shows the properties of selected game objects and other settings. The selected object's name is located at the top of this window, followed by a list of the properties of the object which can be edited. In Figure 6 for example, the user could change the Main Camera object's three dimensional position, rotation and scale by entering numeric values in the applicable places. (Menard & Wagstaff 2015; The Inspector Window 2016.)

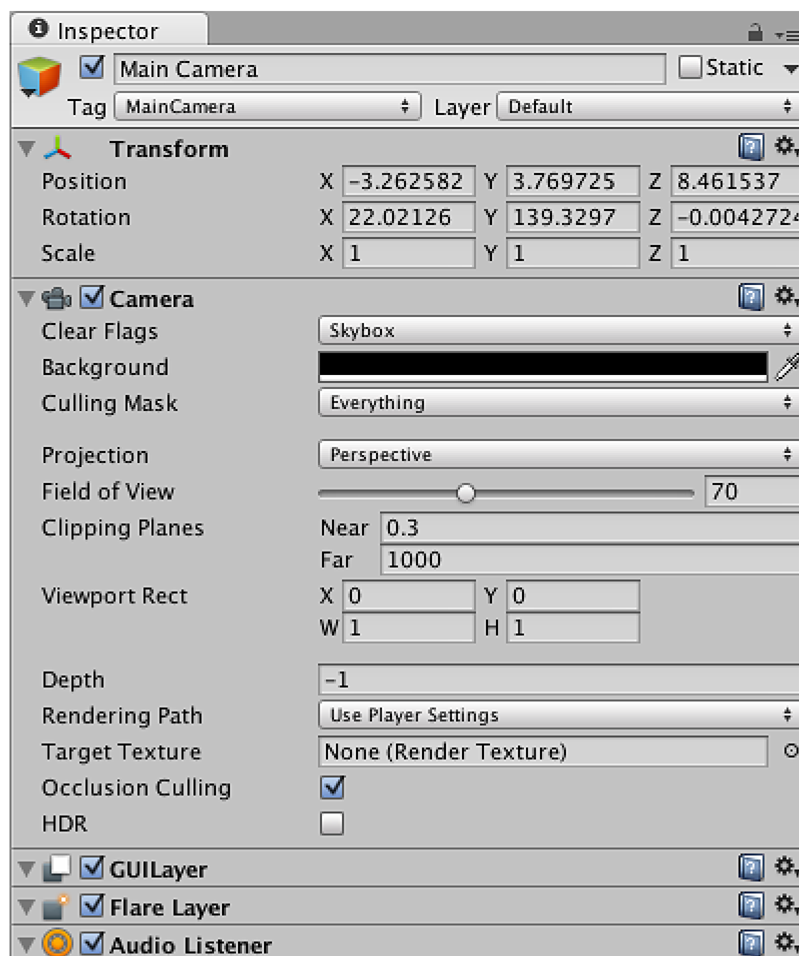


Figure 6. The Inspector Window (The Inspector Window 2016.)

4.1.4 The Project Window

The Project Window organizes all the project files in an Assets folder. The left panel in Figure 7 lists the folder structure of the project while the individual files, called assets in Unity, in the currently selected folder are shown on the right. All files are arranged in the same way the computer's hard drive in the Unity's project folder. The assets can be dragged from this window and dropped into the Scene View or the Hierarchy Window, while the selected assets' information is shown in the Inspector Window. The Project Window also has a search bar that helps finding assets by name, type or label. Additionally, the user can create new files and folders in the Project Window using the Create dropdown menu or right-clicking and selecting the type of a file needed. (Menard & Wagstaff 2015; The Project Window 2016.)

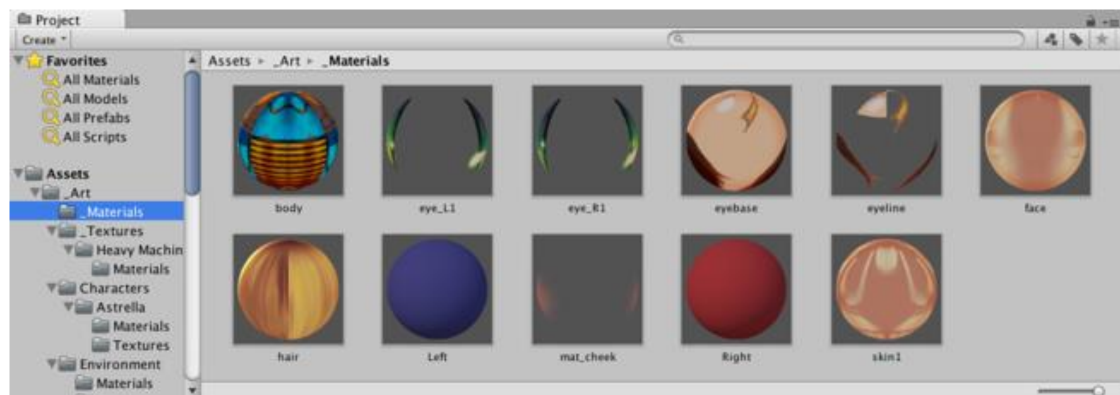


Figure 7. The Project Window (The Project Window 2016.)

4.1.5 The Scene View

The Scene View shows the game scene, where all objects can be seen. The objects can be moved, rotated and scaled by hand as opposed to the Inspector Window's numerical approach. The view can be changed from three dimensional to two dimensional and the viewpoint can be moved and rotated freely in the scene. (The Scene View 2016.)

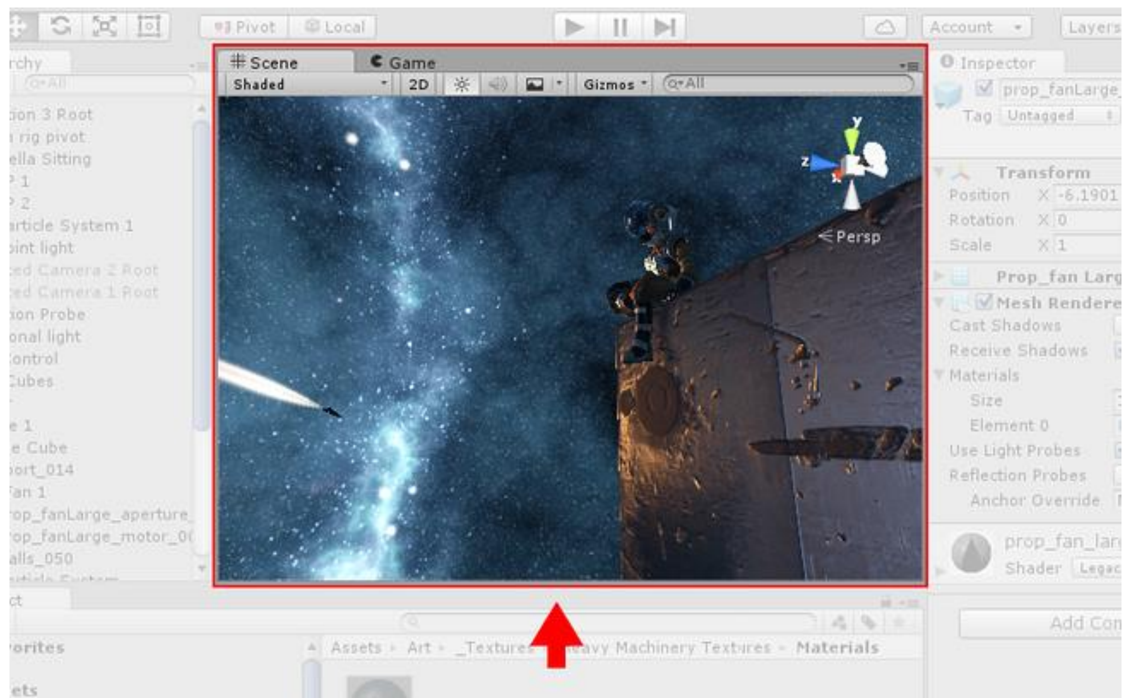


Figure 8. The Scene View (The Scene View 2016.)

4.2 Programming with Unity

4.2.1 GameObject and components

Every object that is created in the Unity Editor is a *GameObject*, Unity's native type of object that represents characters, scenery and props. GameObjects themselves have no functionality of their own other than being a container for components; they are just like entities in the entity-component-system pattern. (GameObject 2016; GameObjects 2016; Introduction to Components 2016.)

Components provide the functionality of the GameObject that contains them. In entity-component-system terms, they are component-system hybrids rather than just plain components, as many of them are specialized in particular functions. For example, a premade Light Component produces a light in the game environment. A light object can be easily made by creating an empty GameObject and attaching a Light component in it from the Inspector Window, as Figure 9 demonstrates. (Introduction to Components 2016.)

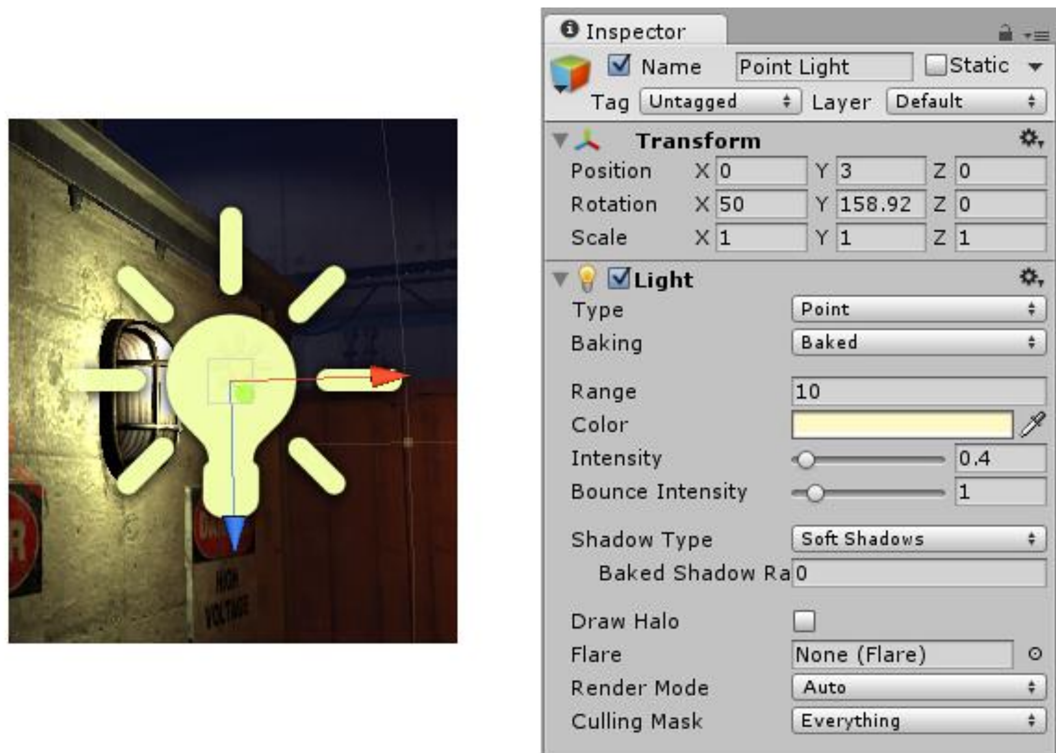


Figure 9. Example of a light object (GameObject 2016.)

Every GameObject has a Transform component by default and it cannot be removed. The Transform component defines the objects position, rotation and scale in the game scene. The component is propably the most important component in Unity, because without it a GameObject would not exist in the game world. It also enables the parental relationship between objects that can be seen in the Hierarchy Window and accessed through it in scripts. (Introduction to Components 2016.)

4.2.2 Prefabs

The same instance of a GameObject built in the game scene cannot be used in other scenes without creating a *prefab* out of the object first. A prefab is an asset type that allows one to store a GameObject with all of its properties and components. It works as a template from which similar instances of the GameObject can be created. (Prefabs 2016.)

Prefabs also come in handy when multiple instances of the same type of a GameObject would be needed. All changes made to the prefab are reflected in the

instances created from it. Likewise, all changes to an intances of a prefab in the scene can be applied to the prefab. (Prefabs 2016.)

A good example would be a shooter game, where the player shoots bullets. As there would be multiple bullets of the same type, they should be prefabs so they could be edited simply by editing one single prefab.

One can create a prefab simply by dragging a GameObject from the Hierarchy Window to a folder in the Project Window. The prefab can then be modified by selecting it and making the changes in the Inspector Window. New instances of the prefab can be created by dragging it back to the Hierarchy Window or the Scene View. (Prefabs 2016.)

4.2.3 Scripts

Unity provides an optional integrated development environment (IDE) named MonoDevelop that is used for *scripting*. Users can also use any other IDEs for creating scripts, such as Microsoft's Visual Studio. Unity natively supports two programming languages: C# and UnityScript that is modeled after JavaScript and specifically designed for Unity. (Creating and Using Scripts 2016.)

Scripts are considered as assets and therefore they are saved in the asset folders that can be seen in the Project Window. When selecting a script from the Project Window, a preview of its code can be seen in the Inspector Window. If double-clicked, the script will be opened in the IDE the user has chosen for editing. (Creating and Using Scripts 2016.)

When creating a new script for a project, Unity generates a preformatted script file. A new C#-script looks like this:

```
using UnityEngine;
using System.Collections;

public class NewBehaviourScript : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
```

```
        void Update () {

        }
    }
```

By default, every script contains a class named after the script's name that inherits *MonoBehaviour* class. MonoBehaviour has many useful methods, such as the event functions *Start* and *Update*. The Start method is called once when the script is enabled and is commonly used for the initialization of variables. Update is called on every frame and is used for handling over time actions, such as triggers and user input during gameplay. MonoBehaviour has also other event functions that are not predefined in new scripts. One of them is *FixedUpdate*, similar to Update, but it is called on a fixed framerate. It is ideally used for handling physics and other functions that need to be handled regardless of current framerate that can fluctuate by time. A script's class does not have to inherit MonoBehaviour however, but creating one without it excludes the usage of MonoBehaviour's methods and it cannot be used as a component in the editor. Developers can therefore create scripts that are not used directly inside the objects of the game, but with public mehodss that can be called by onjects instead. For example, a custom class for writing and reading data from files outside the game folders does not need to inherit the MonoBehaviour class; the GameObjects that need to write or read data can just use the public methods of the custom class. (Event Functions 2016; MonoBehaviour 2016.)

Scripts behave like Unity's other components. They can be added to a GameObject by dragging them from the Project Window onto the Inspector Window when the object is selected, or they can be added from the 'Add Component'-button in the Inspector Window. If a script has public variables and is attached to a GameObject, the public variables and their values can be seen in the Inspector Window, as seen in the Figure 10. This way the values can be edited in the Unity Editor without opening and editing the script itself. It is very useful when testing scripts, as the values of the public variables can also be edited while the game is running in the Game View. (Creating and Using Scripts 2016; The Inspector Window 2016; Variables and the Inspector 2016.)

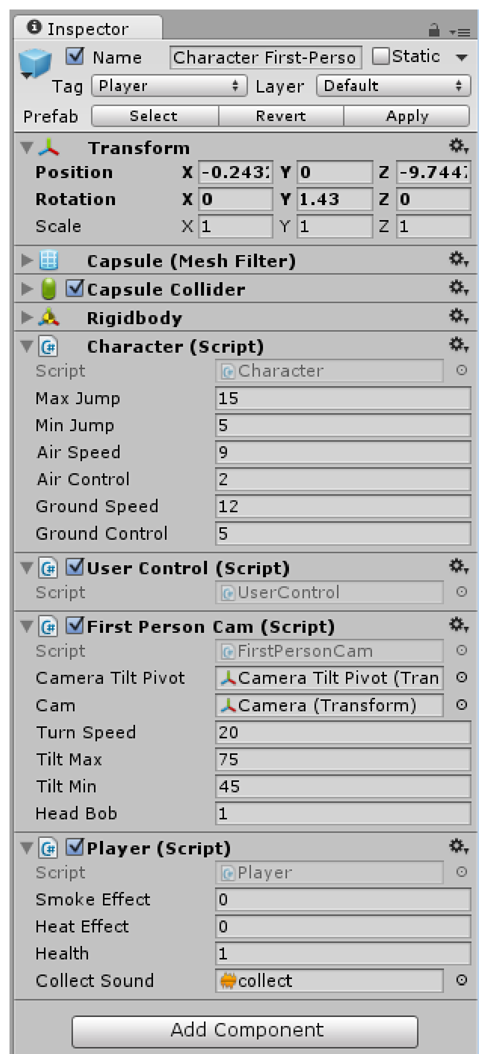


Figure 10. Scripts as components (The Inspector Window 2016.)

5 Project Quantum Knight

Project Quantum Knight's objective is the creation of a game prototype using efficient object-oriented programming solutions. This chapter concentrates on the design and gameplay of the game. The implementation of the project is reviewed in chapter 6.

5.1 Story

The game is set in the near future, where a single corporation called Yú Corporation rules the dystopian city of Kyoto with its own security and military forces. The human rights have been diminished by Yú Corporation in order to ensure the city's peace

and the safety of the citizens. The main character of the story is known only by the name of Void, who is an infamous hacker. The main character does not know their true nature; Void is one of the synthetic lifeforms created by the Yú Corporation's Quantum Knight project. The Quantum Knights have an ability to enter in the cyber world of the city's digital infrastructure and exit to an entirely different place in the real world via terminals.

At the beginning of the game the main character's personal domain in the cyber world is attacked by a tracker virus dispatched by Cybernatural Affairs Department, a section of the Yú Corporation that specializes in cyber security. The virus causes a lockdown in the main character's apartment and the main character has to defeat the virus and escape the city district before agents of the Yú Corporation arrive.

5.2 Gameplay

Quantum Knight has single and multiplayer modes. The single player mode is the main part of the game; it tells the story of the game and the main features are all utilized in it. The player controls the main character in the game world that has been divided into two parts; the city of Kyoto and the cyber world. The players can interact with different objects and characters in Kyoto and they have the ability to enter the cyber world and battle viruses that cause chaos in the game world. By defeating the viruses, the player can advance in the story and reach more places both in Kyoto and in the cyber world.

The multiplayer mode allows two players to play against each other in a multiplayer battle arena. The multiplayer mode has mainly the same battle mechanics as the single player and therefore it is not further discussed in this thesis.

5.3 Dialogue

The story of the game is told through a series of dialogues. When the player interacts with a character in the game, a dialogue window is shown on top of the screen with the character's lines of speech written in it. An example of a dialogue window can be seen in Figure 11. The player can choose the main character's answer or response in

the conversation which will have an impact on the outcome of the conversation and eventually the whole story.

Figure 11. Dialogue window

Interaction with objects other than characters will bring up the dialogue window as well. In that case the dialogue window will tell what the main character is observing and thinking about the object.

5.4 Battle mechanics

The largest part of the game is the battle system. The battle is turn based which means that the two sides of the battle take turns moving one character and making an action with that character. All the battles are set in an enclosed battle arena. These limitations place the emphasis of the battle on strategy and timing one's actions.

5.4.1 Battle modes

Quantum Knight features two battle modes. The first mode is Deathmatch the objective of which is to defeat all opponent's characters. The second mode is called Node

Conquest, where both sides of the conflict have a node, a stationary character that cannot be controlled. The objective of this mode is to destroy the opponent's node.

5.4.2 Turn sequence

At the start of the battle, the participating characters are arranged in order by their speed attribute. The resulting order is used to handle the turn order of the characters to take their actions. The higher the character's speed, the sooner it will act. If all the characters have taken their actions, the order starts from the beginning.

If the character in turn belongs to the player, the player can choose the character's actions as desired. The artificial intelligence chooses the actions for enemy characters and completes the turn automatically. The turn sequence is depicted as an activity diagram in Appendix 3.

5.4.3 Combatants

The main character of the game participates in all battles. The main character is statistically stronger than the other characters, however, as a drawback, the game is over when the main character is defeated. Other characters in the battle are network programs, such as viruses, antiviruses and firewalls, which can be controlled by the player. All enemies are programs controlled by the game's artificial intelligence.

5.4.4 Actions

During their turn in battle, a character can move once. The movement is limited to maximum of five spaces in the arena that is cut into square shaped tiles. The character cannot move outside the arena or onto a space containing an obstacle or a space already occupied by another character. The movable spaces are shown with a blue tint for the player's convenience as illustrated in Figure 12.

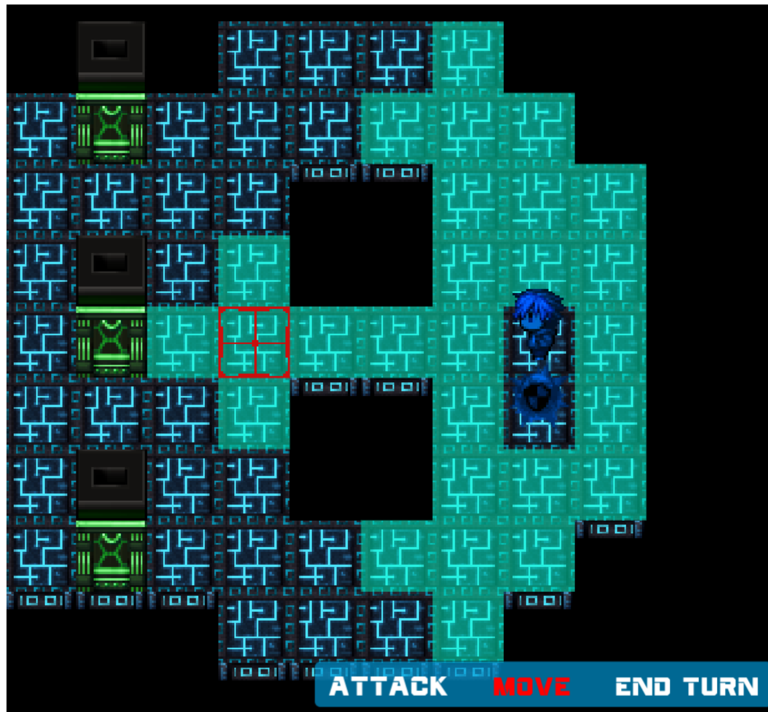


Figure 12. Movement

The character can make one attack against an enemy character. The attack can be made before or immediately after moving. The character can attack an adjacent enemy if it is a melee attacker, or up to three spaces if it uses ranged attacks. If an enemy character is adjacent to the attacker or an obstacle stands in the way of the attack, the enemy cannot be targeted. The attack range is shown in a similar fashion as the movement, however, this time with an orange tint as shown in Figure 13.

Figure 13. Attacking

Instead of attacking normally, a character can use a skill. In this context, skills refer to special actions. Skills create various effects; they can cause significant damage to enemies, heal the user and their allies or improve character's abilities. Skills have different target types and ranges depending on their purpose.

6 Implementation

The Quantum Knight prototype is implemented using the Unity game engine 5.3.2f1 version Personal edition. The focus in this chapter lies on the battle system that features solutions for generalized object design and using inheritance together with composition.

6.1 Player character

The player is embodied as a controllable character in many games, especially in role-playing games, and usually the player acts as the main character in the story. The player character in Quantum Knight is no exception. The player character is considered a unique entity as there is only one player character in the single player game.

The player character is a GameObject which is brought to life with different components and scripts. The main functionalities of the entity are the controls that register player's input and handle the character's actions and animations. The player character GameObject is featured in Appendix 1 as it can be seen in the Inspector Window of Unity editor.

As there are no other characters in the prototype that need to be controlled, be they other players or non-player characters, there is no need to create traditional class hierarchies. Even if that was there, the composition over inheritance principle would still be a better solution. For example, if the characters' controls need to be disabled that can simply be done by disabling the component handling the controls. For the player character that would be the Player Controls script.

6.1.1 Player controls

The player's movement and actions are handled in a script called Player Controls, which is featured in Appendix 4. The script listens to the user's input from keyboard and gamepad controller. For example, if the player presses a movement key or button, the script moves the player GameObject in the proper direction. The following part of the script listens to the user's input from keyboard and gamepad controller and composes a movement vector based on the input. This is done on every frame as it is inside the Update event function.

```
void Update ()
{
    inputX = Input.GetAxis ("Horizontal");
    inputY = Input.GetAxis ("Vertical");

    movement = new Vector2 (speed * in-
putX, speed * inputY);
}
```

The player object's movement itself is implemented in FixedUpdate event function in the code below because it has to be separated from the current framerate. The movement is carried out by the GameObject's Transform component's Translate method that receives the movement vector as a parameter.

```
void FixedUpdate()
{
```

```
        transform.Translate (movement * speed * Time.del-
taTime);

}
```

Player's movement and interaction with the game environment in Quantum Knight relies heavily on collisions. The player GameObject, interactable objects and obstacles have collider components attached to them. The colliders stop objects' movement if they collide with others by default if the colliders are not configured as triggers in the game engine. The colliders of one scene in the game are displayed as green bordered rectangles in Figure 14.

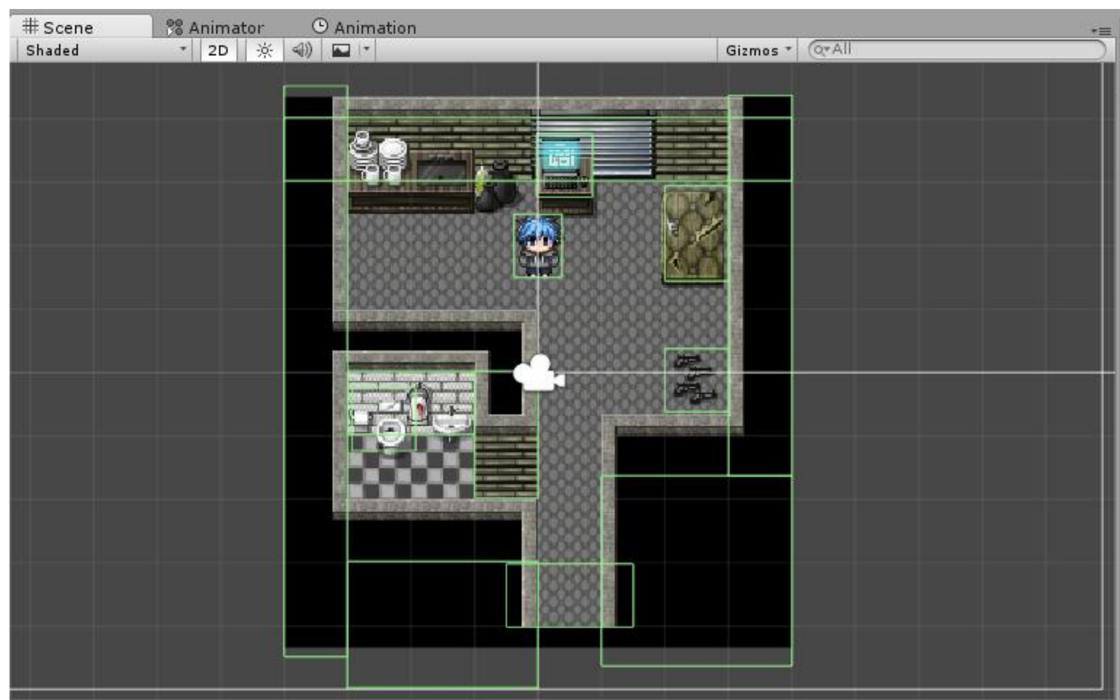


Figure 14. Colliders in the Scene View

The colliders are tagged with keywords, so the collisions can be identified. The next code fragment from the Player Controls script specifically checks collisions between interactable objects and an invisible ray in front of the player. If the ray hits a GameObject that can be interacted with, the player is notified when they can interact with an object in front of them. Additionally, if the player presses the proper button, the Dialogue script in the interacted GameObject is triggered.

```
if (hit.collider != null && hit.collider.tag == "In-
teractable")
{
```

```
    if (panelBool == false && hit.collider.GetCompo-
nent<Dialogue>().enabled == true)
    {
        InteractionPanel.SetActive (true);

        panelBool = true;
    }

    if (Input.GetButtonDown ("Submit") && hit.col-
lider.GetComponent<Dialogue>().enabled == true)
    {
        hit.collider.GetComponent<Dialogue> ().inter-
acted = true;

        InteractionPanel.SetActive (false);

        panelBool = false;
    }
}
```

The different actions the player can take and the controls for keyboard and gamepad are listed below in Table 1.

Table 1. Controls

Action	Keyboard key	Gamepad button
Move left	Left key / A	Left directional button / stick
Move right	Right key / D	Right directional button / stick
Move up	Up Key / W	Up directional / stick
Move down	Down Key / S	Down directional / stick
Activate	Enter / E	A

6.1.2 Animations

The player GameObject has an Animator component attached to it. The Animator is a controller that holds different animation states and handles transitions between them, which are created in the Animator Window that is featured in Figure 15. The states are illustrated by grey boxes while the default state, in which the animation is directed from the start, is coloured yellow. The transitions between the states are

drawn as white lines with an arrow in the middle pointing to the direction of the change in the states.

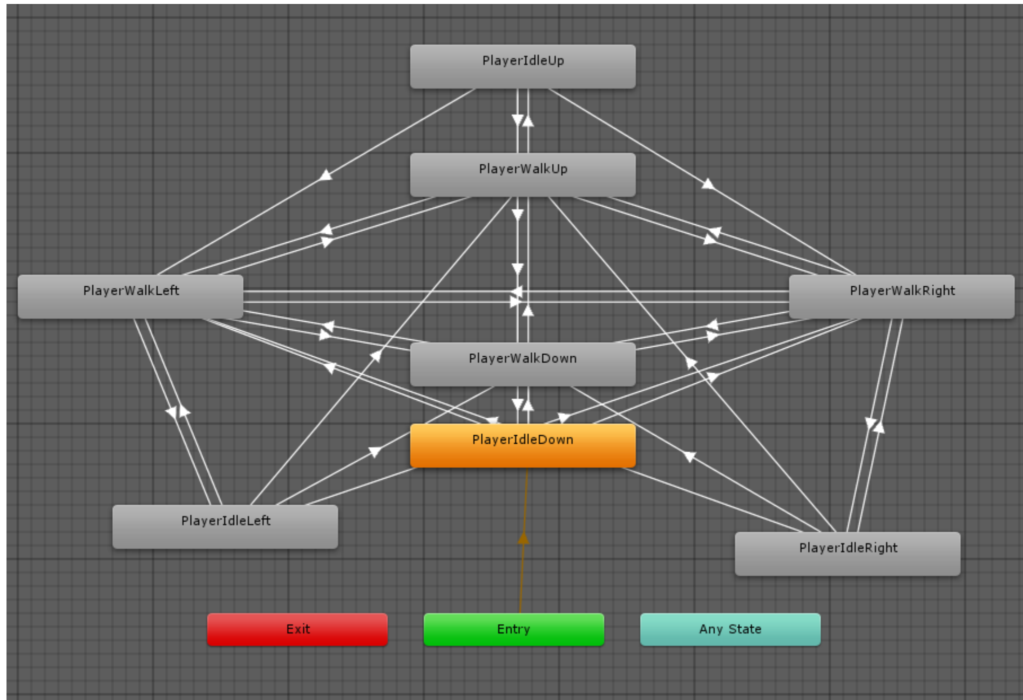


Figure 15. Animation states for the player in the Animator Window

The Player Controls script sends signals to the controller when a change in the states is required. In this case, when the player is moved, the Animator changes its animation to a corresponding one; if the player moves left, the animation for walking left is triggered. The animation state is changed simply by assigning a value of a Boolean variable for the Animator with the SetBool method in the code below.

```
playerAnimator.SetBool ("WalkLeft", true);
```

The animations for player movement consist of three frames per one of four directions. When the animation frame changes from one to another, the Animator controller switches the sprite of the player character's Sprite Renderer component. The different sprites are drawn in a sheet in Figure 16.

Figure 16. Sprite sheet of the main character

6.1.3 Non-player characters

Non-player characters are easily done by removing few components from the player GameObject and adding other components in their place. The character in Appendix 2, for instance, has been created by removing the player only components and scripts, such as the Player Controls script, and adding the components needed. In this case, the character is stationary, so any movement scripts or Animator components are not included. The character is then given a functionality with a Dialogue script that is further discussed in the next chapter.

The character can be modified simply by adding new components and scripts. For example, the non-player character can be made dynamic with a movement script and by animating it.

6.2 Game progression

The progression of Quantum Knight is managed mostly by a single GameObject called the GameManager. It stores important game data at runtime, such as which events have taken place and which areas of the game are available for the player to explore.

6.2.1 Dialogue

Dialogue is triggered when the player interacts with an object in the game that is tagged as interactable with the Dialogue script attached, which is featured in Appendix 6. The script disables the player's movement and opens the dialogue window and

presents the first lines of the dialogue. The part of the script below changes the dialogue window's text from line to line when the player presses the appropriate button and if there are more lines of dialogue in the conversation.

```
if (Input.GetButtonDown ("Submit") && firstCon-
tact == true && dialogIndex < dialog-
Lines.Length - 1)
{
    dialogIndex++;

    dialogueText.text = dialogLines [dialogIndex];
}
```

Figure 17 demonstrates in Unity editor the Dialogue script that is attached to an object. The lines of the conversation are public variables in an array, and can be edited in the Inspector Window without writing all possible dialogue lines in the game in the script itself. The dialogue can also be checked off to have an event; when the dialogue is over, the Dialogue script calls the Event Manager script and triggers an event with the index that is sent as a parameter.

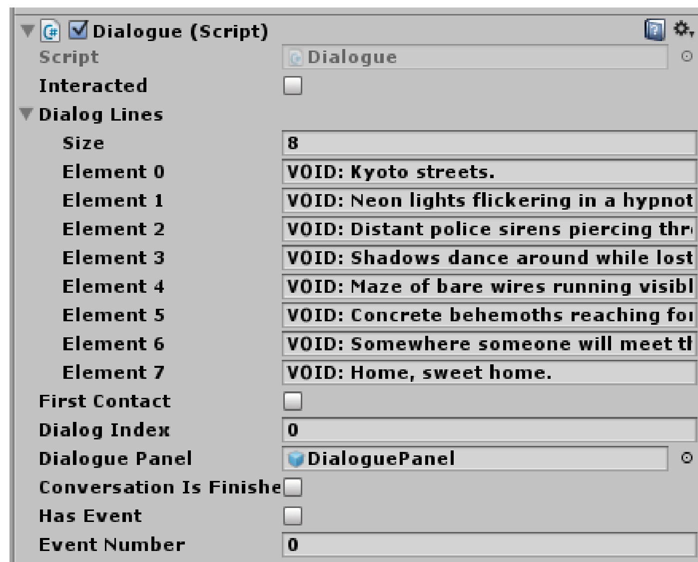


Figure 17. Dialogue script

6.2.2 Events

The most important events in the game are recorded in one of the GameManager's scripts called the Event Manager, when they are triggered. The script contains one method called the EventStater that starts a certain event depending on the index

number it receives as a parameter. The method has a switch-statement that divides the different events. For example, the part of the script below will take effect if the EventStater is called with an integer of 3 as a parameter. It finds a GameObject called ‘Drone’, sets on a specifically engineered event script in it and finally saves the information that the event has been triggered so it cannot be triggered again.

```
case 3: // drone event

    GameObject.Find("Drone").GetCompo-
nent<DroneEvent> ().enabled = true;

    droneEventHasHappened = true;

    break;
```

The whole Event Manager script is featured in Appendix 5. Most events are triggered by dialogue between the player and objects in the game.

Most of the events in the prototype are located in the Event Manager script, or they are special scripts triggered by the manager. This solution is adequate with the small number of events in the prototype, however, should the project grow bigger, more efficient solution would then be needed. Events could be generalized into a single class, similar to the Dialogue script. For more advanced events, the solution could also use inheritance in the same way the skills use it, as explained in chapter 6.3.6.

6.2.3 Scenes

Quantum Knight is divided into multiple areas, all of which make up the scenes of the game. The scene is changed from one to another by events and by walking out of the area exits that have the Scene Changer script attached. The script has ChangeScene method that is featured below. When it is called, it fades the game screen black, changes the background music, moves the player object to correct position for the next scene and finally changes the scene to the next one. The whole script is found in Appendix 7.

```
IEnumerator ChangeScene()
{
    faderScript.fadeToBlack = true;

    playerMovementScript.enabled = false;
```

```
    GameObject.Find ("SoundManager").GetCompo-
nent<SoundManager> ().ChangeClip (newMusicClipIndex);

    yield return new WaitForSeconds (waitTime);

    GameObject.Find ("GameManager").GetCompo-
nent<PlayerPositionHandler>().playerStartPosi-
tion = playerPositionInNextScene;

    SceneManager.LoadScene (sceneName);

    faderScript.fadeToBlack = false;

    playerMovementScript.enabled = true;
}
```

If the solution with event classes presented in the last chapter were to be used, the scene change could be one of the advanced event classes. However, the Scene Changer script being rather simple and effective, the solution could be more complicated than the current one.

When the game engine changes the game scene, all the objects in the current scene are destroyed to clear the memory. It would create problems with the management objects that need to be available at all times during the game's runtime. Fortunately, GameObjects can be protected by a simple script with one method called DontDestroyOnLoad. As the name implies, the method prevents the game engine from destroying the object that has been given as the parameter for the method. The code below calls the method when the corresponding GameObject is initialized by the Awake event function.

```
using UnityEngine;
using System.Collections;

public class DontDestroyOnLoad : MonoBehaviour
{
    void Awake()
    {
        DontDestroyOnLoad (gameObject);
    }
}
```

6.3 Battle system

Quantum Knight's battles are overseen by a single GameObject called BattleManager. It handles the turn sequence, actions available for the current character,

characters' movement and attacking, artificial intelligence for the enemies and ending the battles when certain conditions are met. The BattleManager can be seen in Figure 18 as it is seen in the Inspector Window.

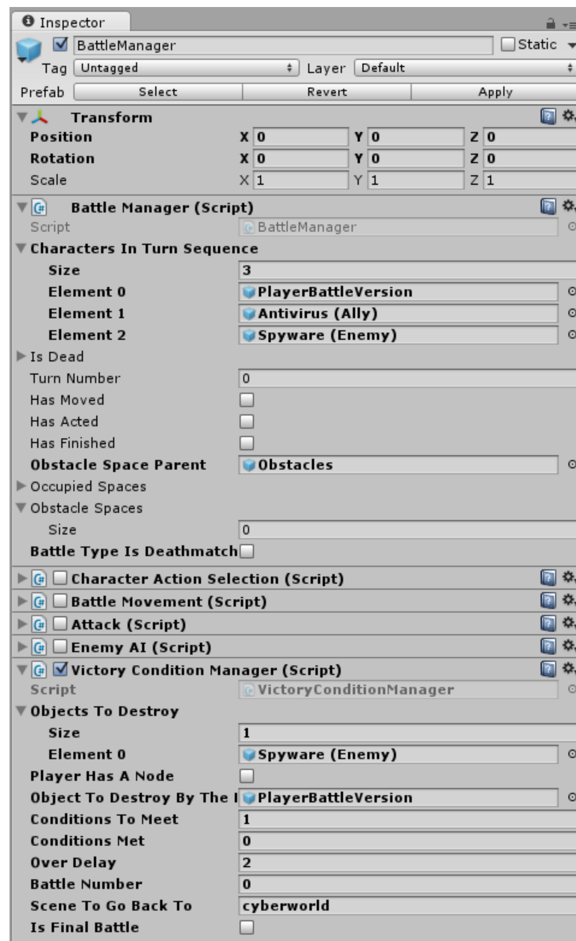


Figure 18. BattleManager

The BattleManager has numerous scripts linked together creating a complex system. Most of the scripts are disabled at the start of the battle, only to be enabled when necessary. The script named after the manager, the Battle Manager script, launches the battle by sorting the characters that are dragged from the scene into the turn sequence list. The characters are sorted by their speed values so the character with the highest speed value is first. Additionally, the script inserts all the obstacles and spaces occupied by characters in an array that is used by the enemy artificial intelligence. Finally, it starts the battle by giving the turn to the first character in the turn sequence.

The Battle Manager monitors the battle event with the Victory Conditions Manager script. It keeps track of the enemies that the player needs to defeat in order to win the battle and vice versa. When the winning conditions are met, the script ends the battle and changes the game scene.

6.3.1 Objects in battle

Instead of creating entities with control scripts of their own, the characters in battle are merely simple pawns controlled by the BattleManager. This way the objects are simplified in contrast to them being complex entities of their own. Their functionality is limited to having only statistics relevant to the battle mechanics. The statistics are public variables stored in Character Stats script that is featured in Figure 19 as a script component in the Inspector Window.

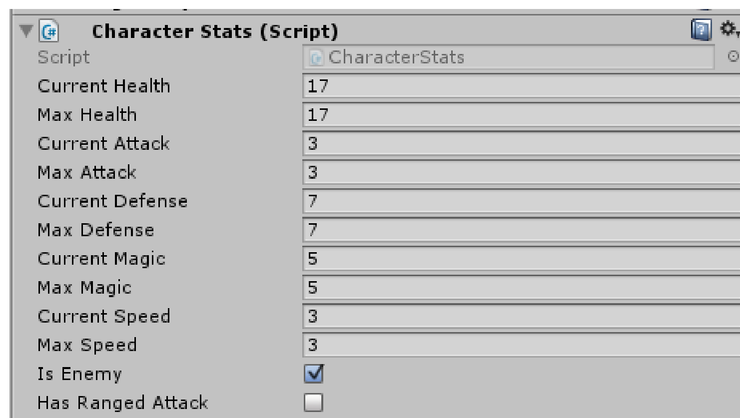


Figure 19. Character Stats

All the statistics have values for the current status of the character that can change in the course of the battle and the maximum values that are the standard values for the character at the beginning of the battle.

The more traditional solution for creating the battling characters is to design a class hierarchy similar to Figure 1 in chapter 2, however, a solution with the composition over inheritance principle in mind brings more flexibility to the design. For example, the player characters and enemy characters could be differentiated easily by adding different controllers for them, the player character having a controller component that listens to user input and the enemy character having an artificial intelligence

component. Additionally, the characters could have components for specialized attacks and movement but as all the characters currently behave in the same fashion in the battle, the solution discussed in this chapter is used instead.

6.3.2 Turn sequence

The Battle Manager script has a public array into which all the participants of the battle are dragged from the game scene. It also has a public integer that shows the current turn number. The manager uses the array, turn number and Character Action Selection script together to create the turn sequence of the battle. The Character Action Selection script is always enabled at the beginning of a new turn. The first thing it does in the OnEnable event function, which is called when the script is enabled, is checking whether the current character in turn is already at zero health, as seen in the code below.

```
if (managerScript.CharactersInTurnSequence [manager-
Script.turnNumber].GetComponent<Character-
Stats> ().currentHealth <= 0 )
{
    StartCoroutine(SkipTurn ());
}
```

If the current character has already fallen in battle, it is forced to skip its turn which is carried out by the SkipTurn method featured below.

```
IEnumerator SkipTurn()
{
    if (managerScript.turnNumber < manager-
Script.CharactersInTurnSequence.Length - 1)
    {
        managerScript.turnNumber++;
    }

    else
    {
        managerScript.turnNumber = 0;
    }

    this.enabled = false;

    yield return new WaitForSeconds (0.5f);

    this.enabled = true;
}
```

SkipTurn method increases the turn number by one, or if it has reached the maximum of the array indexes it will start from the beginning. Finally the script turns itself off and on, effectively starting a new turn with the new character based on the turn number and the array.

If the current character can act on its turn, the next step for the script is to check whether the character belongs to the player or the enemy. The script opens the panel featured in Figure 20 on the player's turn from which the player can choose to move the character or to attack with it. On an enemy's turn, the artificial intelligence is turned on, which performs the actions automatically for the enemy. The whole turn sequence can be viewed as an activity diagram in Appendix 3.

Figure 20. Action panel

6.3.3 Movement

When the player chooses to move a character on their turn, the Battle Movement script in the BattleManager is enabled. As the script is enabled, it spawns blue tiles around the character that show the movement range for the character. All the blue tiles have a script attached to them with the following method:

```
void OnTriggerStay2D(Collider2D other)
{
    if (other.tag == "Obstacle" ||
other.tag == "Hole" ||
other.tag == "Boundary" ||
other.tag == "Enemy" ||
other.tag == "Player")
    {
        Destroy (gameObject);
    }
}
```

The OnTriggerStay2D method destroys the corresponding tile that collides with any collider tagged with one of the five tags above; as all obstacles and such are tagged in the battle arena, the script effectively removes the blue tiles from the movement

range in those spaces the character cannot move into. The movement range can be seen in Figure 12 in chapter 5.

The player chooses the space to move with a selector; a red cursor that has its own movement script similar to the Player Controls script. If the selector collides with a blue movement range tile, its movement script sends information to the Battle Movement script that the character can move there, otherwise the selector changes its sprite to feature a cross in the middle as a sign to the player that the space cannot be moved into. The left screenshot in Figure 21 shows the selector in a space that can be moved into while the screenshot on the right shows the opposite.

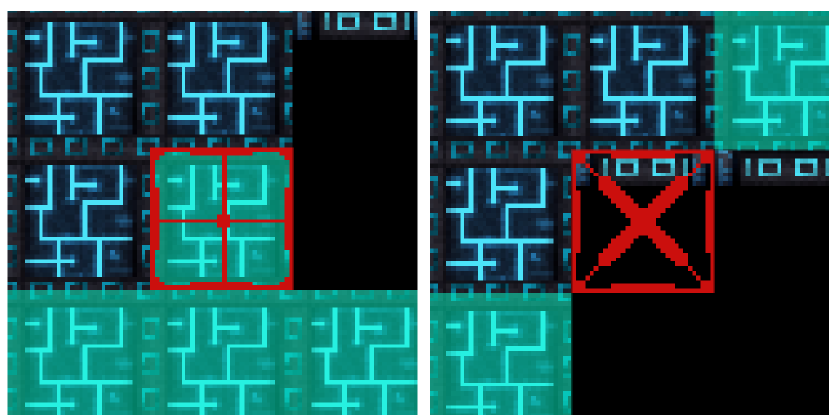


Figure 21. Selector

When the selector is in a proper space, the character can be moved there by pressing the submit button.

6.3.4 Attacking

Attacking works in the same way as moving; only instead of moving a character as the end result, the character deals damage to another. When the player chooses to attack, the Attack script in the BattleManager is enabled. The range of the attack is displayed in the same way as the movement, with orange tiles in this case as Figure 13 in chapter 5 illustrates.

Like the Battle Movement script, the Attack script uses the selector. The Attack script needs two pieces of information from the selector; whether the selector is on the or-

ange tile and whether it is on a space containing an enemy character. If both conditions are true, the player can press the submit button to deal damage to the targeted enemy character.

The damage each character takes when they are attacked is calculated with the attack and defense values in the Character Stats script as shown in the following excerpt from the Attack script that handles such calculations.

```
int damage = CharacterAttacking.GetComponent<Charac-
terStats> ().currentAttack - CharacterTakingTh-
eHit.GetComponent<CharacterStats> ().currentDefense;

    if (damage > 0)
    {
        CharacterTakingTheHit.GetComponent<Character-
Stats> ().currentHealth -= damage;
    }

    else
    {
        CharacterTakingTheHit.GetComponent<Character-
Stats> ().currentHealth -= 1;
    }
```

The damage equals the target’s defense value subtracted from the attacker’s attack value. The final damage value can be less or equal to zero if the target’s defense value exceeds the attacker’s attack value resulting in erroneous cases where the attacker actually heals the target, which is why the target is forced to take one point of damage in that case.

Character’s speed value is used for the turn sequence as mentioned before. Additionally, it serves as a variable for the hit chance calculations for the basic attacks. The calculation is featured in the Attack script excerpt below.

```
int hitChance = 90 - (CharacterTakingTheHit.GetCompo-
nent<CharacterStats> ().currentSpeed - CharacterAt-
tacking.GetComponent<CharacterStats> ().cur-
rentSpeed);
```

After the chance is determined, a random integer is generated between zero and one hundred. If the random number equals or is lower than the hit chance value, the attack hits.

6.3.5 Artifial intelligence

All enemy characters are controlled by an artificial intelligence or AI for short. Again, instead of creating repetitive AI scripts for all enemies, the battle system uses a single script that is located in the BattleManager. The Enemy AI script shares many similarities with the Battle Movement script and the Attack script, however, the AI operates independently. It uses the Battle Manager script and selector in the same way as the player would.

The AI is fundamentally very simple. It locates the closest player character in battle and attacks or moves towards it and then attacks if possible. But as can be seen in Figure 22 that features all the possible paths the AI can take to carry out the enemy's turn, the whole system is rather complex with multiple calculations and analyses.

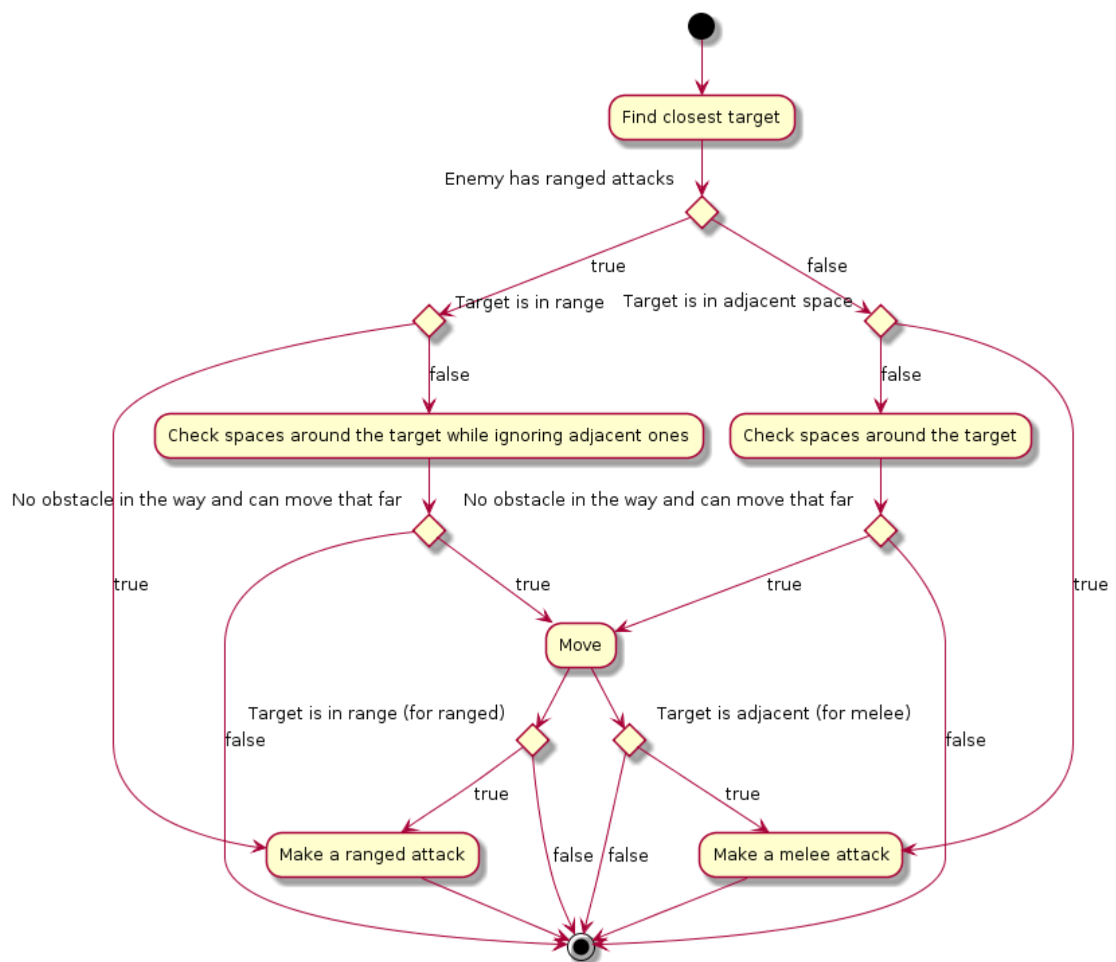


Figure 22. Enemy AI activity diagram

The AI script contains multiple methods for different actions the enemy can take during its turn. A method called EnemyTurnSequence assembles the enemy's turn sequence and it contains series of if-statements which follow the activity diagram in Figure 22. For example, the first statement in the code below checks if the enemy uses ranged attacks and if that condition is true, the next statement checks if the enemy can attack the player character. If the second condition is true, the enemy attacks and finishes its turn, otherwise the AI continues to check the next possible actions.

```
if (hasRangedAttack) // ranged enemy
{
    CheckDistanceToTarget ();

    PrepareForRangedAttack ();

    yield return new WaitForSeconds (waitTime);

    if (distanceToAttack > 1 && dis-
tanceToAttack < 4 && selectorScript.isInAttacka-
bleTile) // the target is in range
    {
        DealDamage ();

        FinishRangedAttack ();

        Debug.Log (CurrentEn-
emy.name + " made a ranged attack");

        EndTurn();
    }
```

The entire Enemy AI script is featured in Appendix 9.

6.3.6 Skills

The skills used by the characters in a battle are comparable to the basic attacks, however, they create various effects in addition to or instead of dealing damage to targets. The implementation of the skill system uses both inheritance and composition. All instances of the skills are component scripts, nevertheless, they are inherited from a parent class that features characteristics shared among all of them.

Two types of skill classes along with the base class were experimented on in the project. The two types feature a basic damage dealing skill and a skill that absorb health

from the target and heals the user. The class diagram for this solution is illustrated in Figure 23.

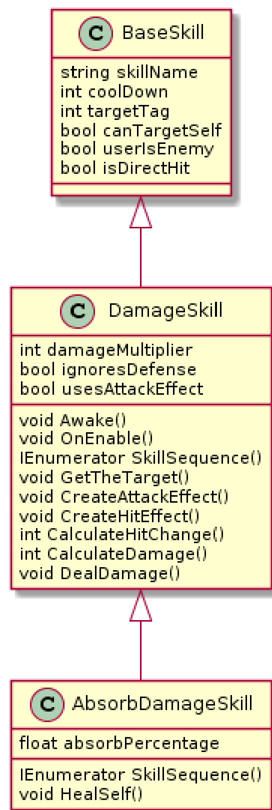


Figure 23. Skill class diagram

The Base Skill script is a simple collection of public variables that all skills inherited from the Base Skill are meant to use. It has no methods as it is not used as such in the game. The base class inherits the MonoBehaviour class so the child classes can then be used as components.

The Damage Skill class inherits the Base Skill class, therefore it features all the variables its parent has and some of its own. The script is meant to be used directly as a component, and thus it contains methods that provide the visual effects and calculations for hit chance and damage. The calculations are more advanced than the methods in the basic Attack script; for example, the skill can be programmed to hit with 100 percent accuracy if one of the public Booleans is made true as the CalculateHitChance method in the following excerpt from the script demonstrates.

```
public virtual int CalculateHitChance()
{
```

```
        int hitChance;

        if (isDirectHit)
        {
            hitChance = 100;
        }

        else // if target's speed is higher -
    > the change is lower, if at-
    tacker's speed is higher -> change is higher
        {
            hitChance = 90 - (Target.GetComponent<Charac-
    terStats> ().currentSpeed - gameObject.GetCompo-
    nent<CharacterStats> ().currentSpeed);
        }

        return hitChance;
    }
```

All of the methods in the Damage Skill script are declared virtual so they can be overridden in the classes inherited from it. The Absorb Damage Skill script takes advantage of this by overriding the SkillSequence method and creating its own version of it. The skill itself is very similar to the Damage Skill, but it creates a healing effect for the user after damaging the target. The overridden method is featured below with the changes compared to the Damage Skill script's method emphasized with strikethrough and red font color.

```
    public virtual override IEnumerator SkillSequence()
    {
        GetTheTarget ();

        yield return new WaitForSeconds (0.1f);

        if (usesAttackEffect)
        {
            CreateAttackEffect ();

            yield return new WaitForSeconds (0.1f);
        }

        CreateHitEffect ();

        yield return new WaitForSeconds (0.5f);

        HealSelf ();

        yield return new WaitForSeconds (0.5f);

        this.enabled = false;
    }
```

The skills are assembled in a script called the Skillbook. The script contains a public array for objects, specifically for Base Skill objects. As the Damage Skill and Absorb Damage Skill both inherit the base class, the components can be dragged into the array in the Inspector Window as Figure 24 shows. The figure also illustrates how the skills themselves can be configured in the editor by inserting values, checking Booleans and dragging effect GameObjects in them.

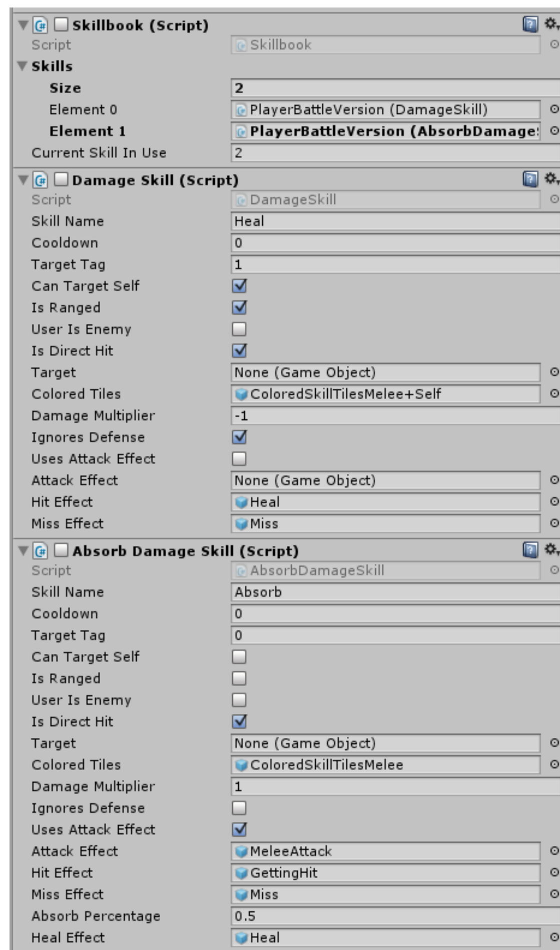


Figure 24. Skillbook and skill scripts

The Skillbook script behaves just like the Attack script does; it spawns the orange tiles to show the range of the skill and turn on the selector. When the player has chosen the target and pressed the proper button, the Skillbook script enables the skill script in question by using a public integer as the array index with the statement below.

```
Skills [currentSkillInUse].enabled = true;
```

The Base Skill, Damage Skill, Absorb Damage Skill and Skillbook scripts are featured in Appendix 10, 11, 12 and 13, respectively.

7 Discussion

The main objective of this thesis was to study different solutions for object-oriented game programming with the Unity game engine. It was done by developing a game prototype called Quantum Knight that utilizes some of the ways to create characters and other objects in Unity.

7.1 Results

The two solutions for the game's characters feature the composition over inheritance principle completely. The main character and non-player characters are differentiated simply by the specific components in them. The characters in the battle system have no functionality of their own and are essentially identical in design. The use of the BattleManager as a controller for them is an extremely generalized and rather complex concept. The same functionality could have been achieved effectively by creating similar objects as the main character including the scripts for moving in the battle arena, for attacking and for skills.

The skills utilize inheritance together with the composition successfully. The Absorb Damage Skill class inherits the Damage Skill's functionality in the desired way while adding new methods in the skill sequence. The two classes can most probably be generalized into a single class, however, the main purpose of the solution is to prove that it can be done with Unity script components and that can be utilized in the Skillbook script with the array of the base parent class typed objects. Furthermore, inheriting the skills reduces the complexity of the code as there is no need for distinguishing the effects they have with several if statements and other checks.

Traditional object-oriented programming with inheritance hierarchies is not used as such in the prototype mostly because of the Unity engine's, and specifically, the Editor's built-in support for components and their editing. The visual editing with the Editor holds no small part when developing games with Unity, and it enhances and speeds up the programming work when used as it is meant to.

7.2 Credibility

The results and solutions in this thesis are evaluated by the terms of usability and versatility from a single programmer’s point of view. They do not offer any opinions or data on the performance of the programming. On that note, a further study on the comparison between composition and inheritance by those terms is needed. As for the credibility of the results here, they are examples of solutions for object design and implementation rather than definitive means for game development.

7.3 Final thoughts

Using components provides many advantages compared to complete classes for objects. Without good design, a large inheritance hierarchy can become overwhelming in big projects. That is of course true of composition also, however, the reusability of components and systems more than makes up for it. Together with the visual programming the Unity Editor provides, the entity-component-system principle outweights traditional object-oriented approaches in game programming. For example, the animation of a character is far easier and faster with the help of the Animator than writing just pure code to do it. In addition, the entity can be remodeled by adding and removing components, saving time and effort while doing so.

The use of public variables in the script components helps the testing of objects. Instead of hardcoding and rewriting the values in the scripts, they can be smoothly changed in the Inspector Window, which is not possible with traditional classes with restricted accesses and private variables. It is possible to create such scripts, however, and they should be used for undisclosed functions, such as passwords, network connections and file management.

Composition can replace inheritance in many cases, however, inheritance still has its uses. Along with the premade Unity components derived from premade base classes, the scripts created by the user can be derived from not only the MonoBehaviour class, which enables the scripts to be used as components but also other custom scripts. That way it is possible to create a vast amount of scripts that work efficiently without overgeneralizing them.

References

4 major principles of Object-Oriented Programming. 2005. Accessed on 25 February 2016. Retrieved from http://codebetter.com/raymondlewallen/2005/07/19/4-major-principles-of-object-oriented-programming/

A feature-rich and highly flexible editor. 2016. Accessed on 29 February 2016. Retrieved from https://unity3d.com/unity/editor

Avoiding the Blob Antipattern: A Pragmatic Approach to Entity Composition. 2012. Accessed on 27 March 2016. Retrieved from http://gamedevelopment.tutsplus.com/tutorials/avoiding-the-blob-antipattern-a-pragmatic-approach-to-entity-composition--gamedev-1113

Build once deploy anywhere. 2016. Accessed on 29 February. Retrieved from https://unity3d.com/unity/multiplatform

C# Inheritance. 2008. Accessed on 30 March 2016. Retrieved from http://www.blackwasp.co.uk/Inheritance.aspx

Creating and Using Scripts. 2016. Accessed on 16 March 2016. Retrieved from http://docs.unity3d.com/Manual/CreatingAndUsingScripts.html

Entities and Components. 2015. Accessed on 19 March 2016. Retrieved from https://developer.apple.com/library/ios/documentation/General/Conceptual/GameplayKit_Guide/EntityComponent.html

Event Functions. 2016. Accessed on 16 March 2016. Retrieved from http://docs.unity3d.com/Manual/EventFunctions.html

GameObject. 2016. Accessed on 16 March 2016. Retrieved from http://docs.unity3d.com/Manual/class-GameObject.html

GameObjects. 2016. Accessed on 16 March 2016. Retrieved from http://docs.unity3d.com/Manual/GameObjects.html

Get Unity. 2016. Accessed on 29 February 2016. Retrieved from https://unity3d.com/get-unity

Harwani, B. 2015. Learning Object-Oriented Programming in C# 5.0. Boston: Cengage Learning PTR.

Introduction to Components. 2016. Accessed on 4 March. Retrieved from http://docs.unity3d.com/Manual/Components.html

Introduction to Object Oriented Programming Concepts (OOP) and More. 2015. Accessed on 25 February. Retrieved from http://www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep#OOP

Lesson: Object-Oriented Programming Concepts. 2015. Accessed on 25 February 2016. Retrieved from https://docs.oracle.com/javase/tutorial/java/concepts/

Menard, M. & Wagstaff, B. 2015. Game Development with Unity, Second Edition. Boston: Cengage Learning PTR.

MonoBehaviour. 2016. Accessed on 16 March 2016. Retrieved from http://docs.unity3d.com/ScriptReference/MonoBehaviour.html

Prefabs. 2016. Accessed on 16 March 2016. Retrieved from http://docs.unity3d.com/Manual/Prefabs.html

The best development platform for creating games. 2016. Accessed on 29 February 2016. Retrieved from http://unity3d.com/unity

The Game View. 2016. Accessed on 16 March 2016. Retrieved from http://docs.unity3d.com/Manual/GameView.html

The Hierarchy Window. 2016. Accessed on 16 March 2016. Retrieved from http://docs.unity3d.com/Manual/Hierarchy.html

The Inspector Window. 2016. Accessed on 16 March 2016. Retrieved from http://docs.unity3d.com/Manual/UsingTheInspector.html

The Project Window. 2016. Accessed on 16 March 2016. Retrieved from http://docs.unity3d.com/Manual/ProjectView.html

The Scene View. 2016. Accessed on 16 March 2016. Retrieved from http://docs.unity3d.com/Manual/UsingTheSceneView.html

Understanding Component-Entity-Systems. 2013. Accessed on 19 March 2016. Retrieved from http://www.gamedev.net/page/resources/_/technical/game-programming/understanding-component-entity-systems-r3013

Unity: Now You're Thinking With Components. 2013. Accessed on 20 March 2016. Retrieved from http://gamedevelopment.tutsplus.com/articles/unity-now-youre-thinking-with-components--gamedev-12492

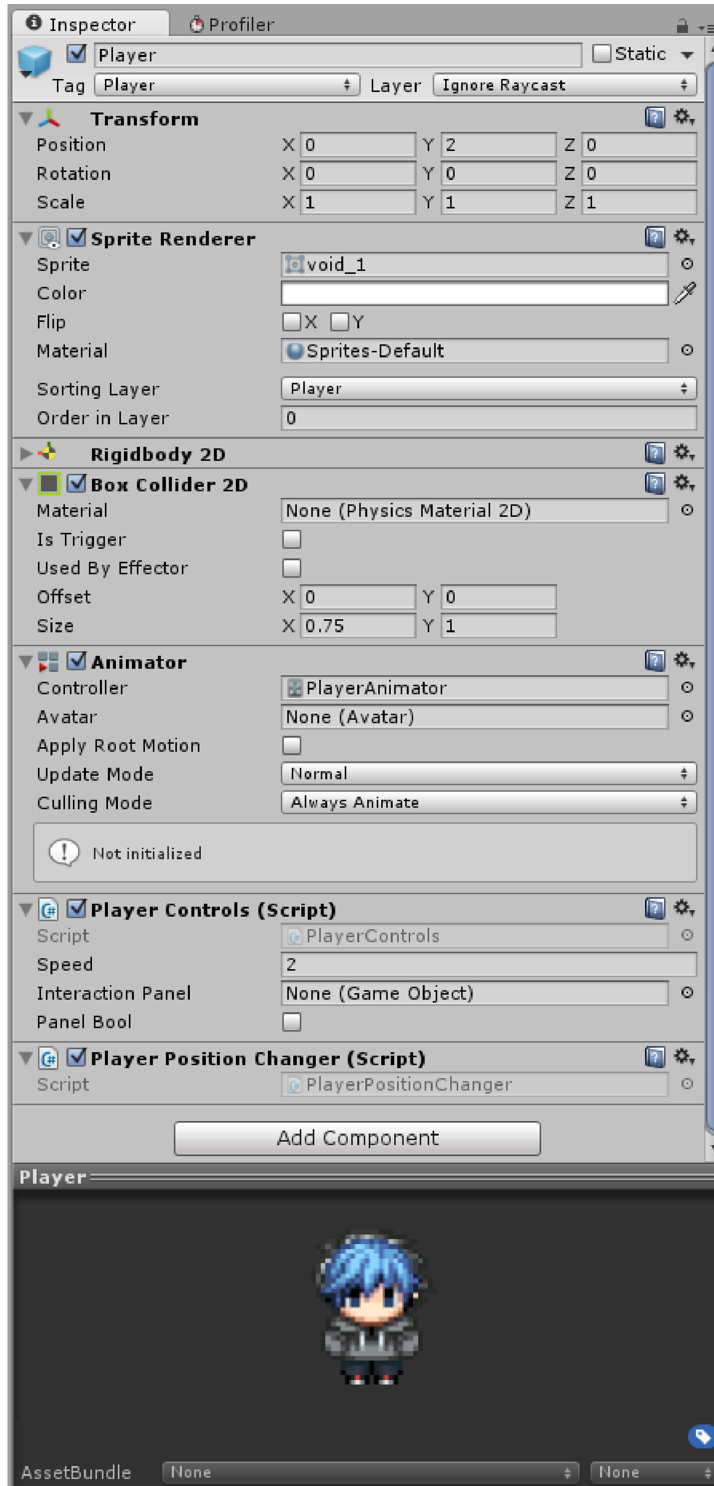
Unity, Source 2, Unreal Engine 4, or CryENGINE - Which Game Engine Should I Choose? 2015. Accessed on 29 February 2016. Retrieved from http://blog.digitaltutors.com/unity-udk-cryengine-game-engine-choose/

Variables and the Inspector. 2016. Accessed on 16 March 2016. Retrieved from http://docs.unity3d.com/Manual/VariablesAndTheInspector.html

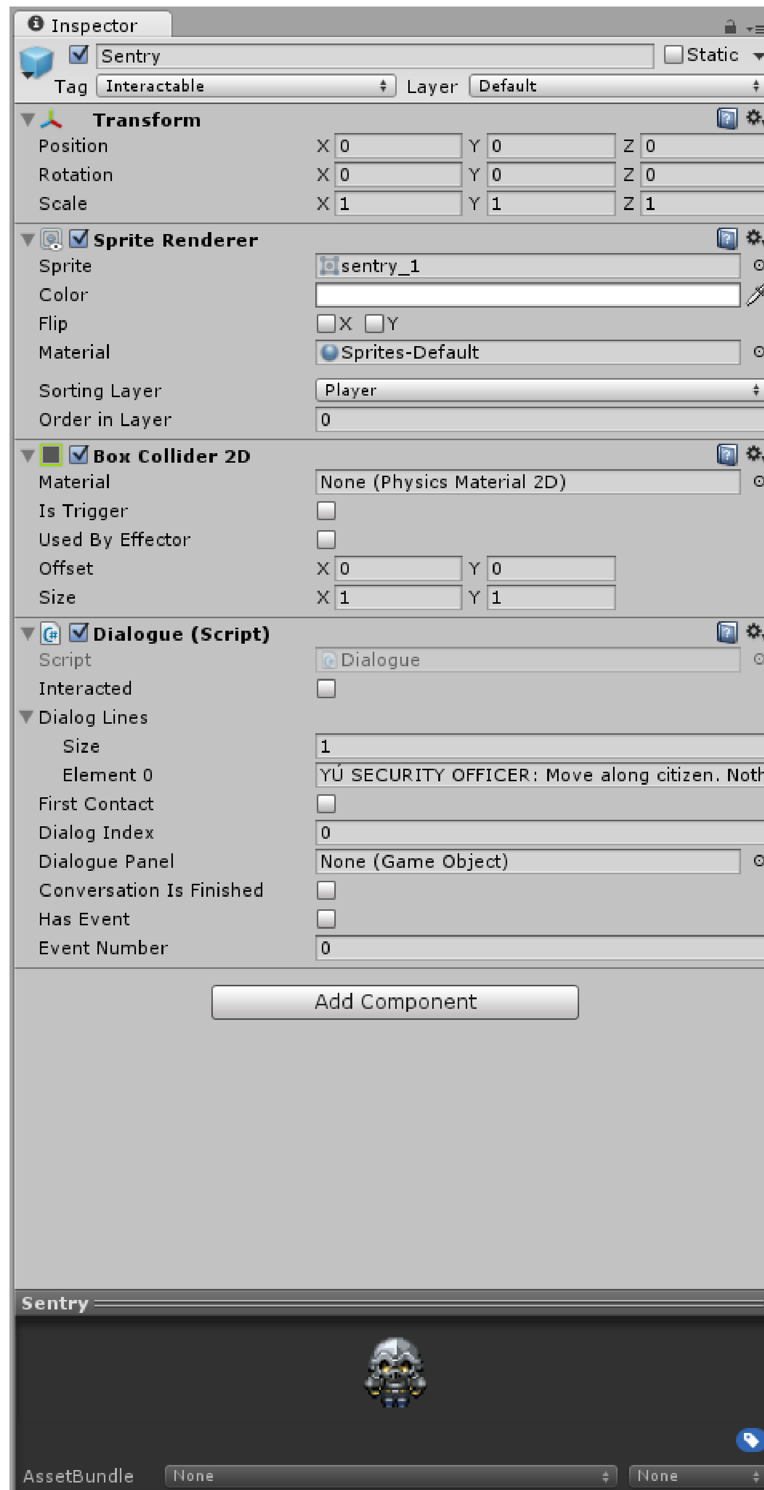
What's an Entity System? 2014. Accessed on 16 March 2016. Retrieved from http://entity-systems-wiki.t-machine.org/

Appendices

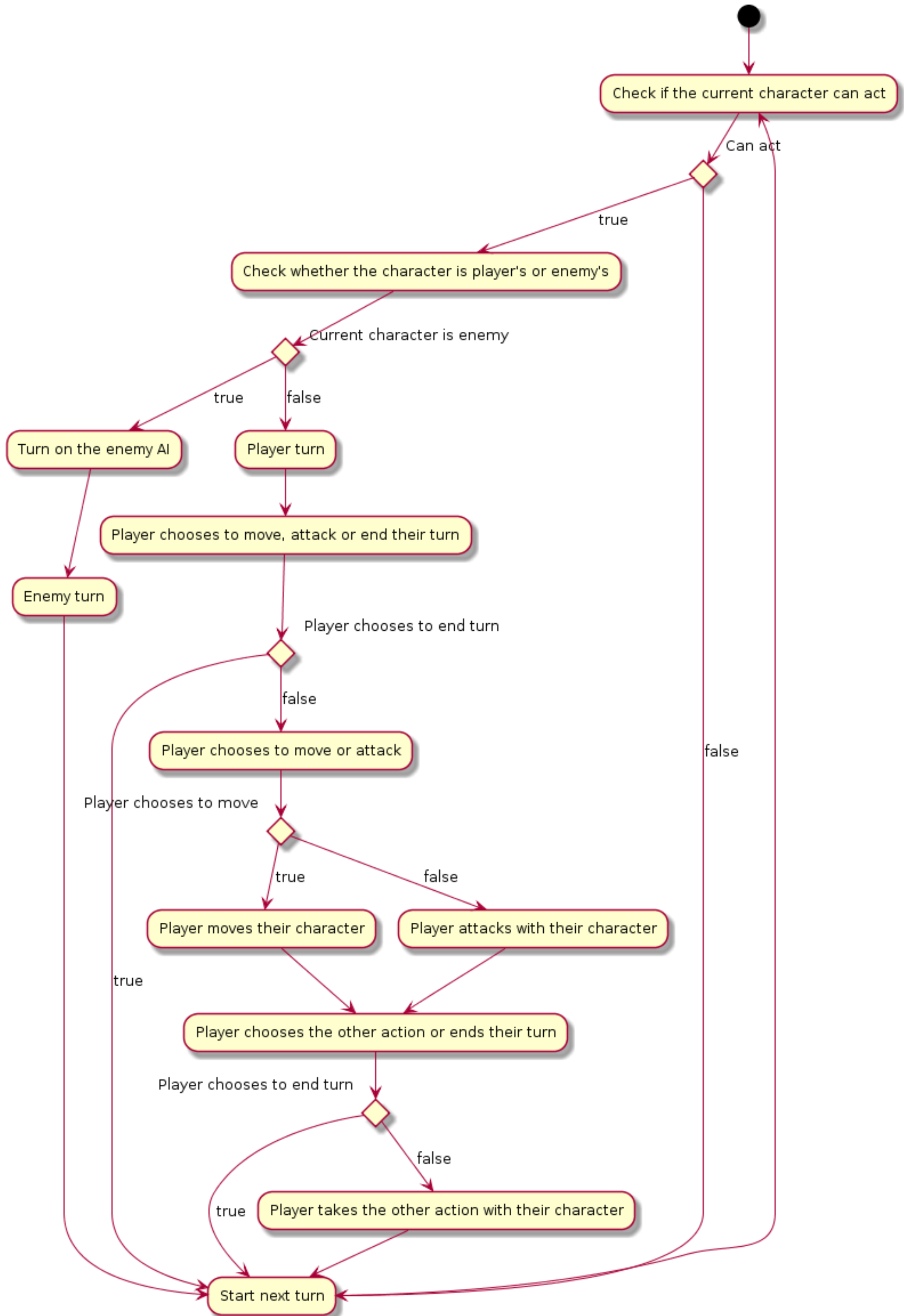
Appendix 1. The player GameObject



Appendix 2. Non-player character



Appendix 3. Turn sequence



Appendix 4. Player Controls script

```
using UnityEngine;
using System.Collections;

public class PlayerControls : MonoBehaviour
{
    float inputX, inputY;
    public float speed;
    Vector2 movement;
    Animator playerAnimator;
    Vector2 previous = Vector2.zero;
    public GameObject InteractionPanel;
    public bool panelBool = false;
    EventManager eventScript;

    // Use this for initialization
    void Start ()
    {
        playerAnimator = gameObject.GetComponent<Animator> ();

        eventScript = GameObject.Find ("EventManager").GetComponent<EventManager> ();
    }

    // Update is called once per frame
    void Update ()
    {
        inputX = Input.GetAxis ("Horizontal");
        inputY = Input.GetAxis ("Vertical");

        movement = new Vector2 (speed * inputX, speed * inputY);

        //Normalizing speed for diagonal movement
        if (movement.magnitude > speed)
        {
            movement = movement.normalized * speed;
        }
    }

    void FixedUpdate()
    {
        transform.Translate (movement * speed * Time.deltaTime);

        AnimationControl (inputX, inputY);

        InteractRaycast (inputX, inputY);
    }

    void AnimationControl(float inputX , float inputY) // controls the character's an-
imation states
    {
        if (inputY > 0.02f && playerAnimator.GetBool("WalkRight") == false && play-
erAnimator.GetBool("WalkLeft") == false)
        {
            playerAnimator.SetBool ("WalkUp", true);

            if (playerAnimator.GetBool ("WalkUp") == true)
            {
                playerAnimator.SetBool ("WalkRight", false);
                playerAnimator.SetBool ("WalkLeft", false);
                playerAnimator.SetBool ("WalkDown", false);
            }
        }

        else if (inputY == 0.0f)
        {
            playerAnimator.SetBool ("WalkUp", false);
        }

        if (inputY < -0.02f && playerAnimator.GetBool("WalkRight") == false && play-
erAnimator.GetBool("WalkLeft") == false)
        {
            playerAnimator.SetBool ("WalkDown", true);
```

```
            if (playerAnimator.GetBool ("WalkDown") == true )
            {
                playerAnimator.SetBool ("WalkRight", false);
                playerAnimator.SetBool ("WalkLeft", false);
                playerAnimator.SetBool ("WalkUp", false);
            }
        }

        else if (inputY == 0.0f)
        {
            playerAnimator.SetBool ("WalkDown", false);
        }

        if (inputX > 0.02f && playerAnimator.GetBool("WalkUp") == false && playerAni-
mator.GetBool("WalkDown") == false)
        {
            playerAnimator.SetBool ("WalkRight", true);
            playerAnimator.SetBool ("WalkLeft", false);

            if (playerAnimator.GetBool("WalkRight") == true)
            {
                playerAnimator.SetBool ("WalkUp", false);
                playerAnimator.SetBool ("WalkDown", false);
            }
        }

        else if (inputX == 0.0f)
        {
            playerAnimator.SetBool ("WalkRight", false);
        }

        if (inputX < -0.02f && playerAnimator.GetBool("WalkUp") == false && playerAni-
mator.GetBool("WalkDown") == false)
        {
            playerAnimator.SetBool ("WalkLeft", true);
            playerAnimator.SetBool ("WalkRight", false);

            if (playerAnimator.GetBool("WalkLeft") == true)
            {
                playerAnimator.SetBool ("WalkUp", false);
                playerAnimator.SetBool ("WalkDown", false);
            }
        }

        else if (inputX == 0.0f)
        {
            playerAnimator.SetBool ("WalkLeft", false);
        }
    }

    void InteractRaycast(float x, float y) // raycasting for interactions with objects
    {
        RaycastHit2D hit = Physics2D.Raycast(transform.position, -Vector2.up);

        Vector2 dir = new Vector2(x,y);

        if (dir == Vector2.zero)
        {
            dir = previous;
        }

        else
        {
            previous = dir;
        }

        hit = Physics2D.Raycast(transform.position, dir, 1);

        Debug.DrawRay(transform.posi-
tion, dir, Color.green); // draws the ray in the scene view

        if (hit.collider != null && hit.collider.tag == "Interactable")
        {
            if (panelBool == false && hit.collider.GetComponent<Dialogue>().ena-
bled == true)
            {
                InteractionPanel.SetActive (true);

                panelBool = true;
```

```
            }

            if (Input.GetButtonDown ("Submit") && hit.collider.GetComponent<Dia-
logue>().enabled == true)
            {
                hit.collider.GetComponent<Dialogue> ().interacted = true;

                InteractionPanel.SetActive (false);

                panelBool = false;
            }
        }

        if (hit.collider == null && panelBool == true)
        {
            InteractionPanel.SetActive (false);

            panelBool = false;
        }

    }
}
```

Appendix 5. Event Manager script

```
using UnityEngine;
using System.Collections;

public class EventManager : MonoBehaviour
{
    public GameObject ToCyberWorld;
    public bool playerIsInCyberWorld;
    public bool droneEventHasHappened;

    public void EventStater(int index, GameObject sender)
    {
        switch (index)
        {
        default:
            break;

        case 0: // prologue part 1

            GameOb-
ject.Find ("PROLOGUE").GetComponent<PrologueEvent> ().pleaseContinue = true;

            break;

        case 1: // prologue part 2

            GameObject.FindWithTag ("Player").GetComponent<PlayerMovement> ().ena-
bled = true;

            GameOb-
ject.Find ("PROLOGUE").GetComponent<PrologueEvent> ().Bunk.SetActive (true);

            Destroy (GameObject.Find ("PROLOGUE"));

            break;

        case 2: // To cyber world and back

            if (playerIsInCyberWorld == true)
            {
                playerIsInCyberWorld = false;
            }

            else
            {
                playerIsInCyberWorld = true;
            }

            Destroy (Instantiate(ToCyberWorld, Camera.main.ScreenToWorldPoint(new Vec-
tor3(Screen.width/2, Screen.height/2, Camera.main.nearClipPlane)), Quaternion.Eu-
ler(0, 0, -90)), 1f);

            GameObject.Find (sender.name).GetComponent<SceneChanger> ().Acti-
vateSceneChange ();

            break;

        case 3: // drone event

            GameObject.Find("Drone").GetComponent<DroneEvent> ().enabled = true;

            droneEventHasHappened = true;

            break;

        case 4: // battle start

            gameObject.GetComponentInParent<PlayerPositionHandler> ().playerStartPosi-
tion = GameObject.FindWithTag ("Player").transform.position;

            GameObject.Find (sender.name).GetComponent<SceneChanger> ().Acti-
vateSceneChange ();

            break;
```

```
            }

        }
    }
```

Appendix 6. Dialogue script

```
using UnityEngine;
using System.Collections;
using UnityEngine.UI;

public class Dialogue : MonoBehaviour
{
    public bool interacted;
    public string[] dialogLines;
    Animator playerAnimator;
    public bool firstContact;
    public int dialogIndex;
    public GameObject DialoguePanel;
    Text dialogueText;
    bool extraBool;
    public bool conversationIsFinished, hasEvent;
    public int eventNumber;

    void Awake()
    {
        playerAnimator = GameObject.FindWithTag("Player").GetComponent<Animator>();

        dialogueText = DialoguePanel.GetComponentInChildren<Text> ();
    }

    void Update()
    {
        if (interacted == true)
        {
            GameObject.FindWithTag ("Player").GetComponent<PlayerControls> ().ena-
bled = false;

            playerAnimator.SetBool ("WalkUp", false);
            playerAnimator.SetBool ("WalkDown", false);
            playerAnimator.SetBool ("WalkLeft", false);
            playerAnimator.SetBool ("WalkRight", false);

            DialoguePanel.SetActive (true);

            if (firstContact == false)
            {
                dialogueText.text = dialogLines [dialogIndex];

                extraBool = true;
            }

            if (Input.GetButtonDown("Submit") && firstContact == true && di-
alogIndex >= dialogLines.Length - 1)
            {
                interacted = false;
                firstContact = false;
                extraBool = false;

                dialogIndex = 0;

                DialoguePanel.SetActive (false);

                GameObject.FindWithTag ("Player").GetComponent<PlayerControls> ().ena-
bled = true;

                conversationIsFinished = true;

                if (hasEvent)
                {
                    GameObject.Find ("EventManager").GetCompo-
nent<EventHandler> ().EventStater(eventNumber, gameObject);
                }
            }

            if (Input.GetButtonDown ("Submit") && firstContact == true && di-
alogIndex < dialogLines.Length - 1)
            {
                dialogIndex++;
```

```
                dialogueText.text = dialogLines [dialogIndex];
            }

            if(extraBool == true)
            {
                firstContact = true;
            }
        }
    }
}
```

Appendix 7. Scene Changer script

```
using UnityEngine;
using System.Collections;
using UnityEngine.SceneManagement;

public class SceneChanger : MonoBehaviour
{
    GameObject Player;
    public string sceneName;
    public Vector2 playerPositionInNextScene;
    SceneLoadFader faderScript;
    public float waitTime;
    PlayerMovement playerMovementScript;
    public bool isEventful, changeMusic;
    public int newMusicClipIndex;

    void Awake()
    {
        Player = GameObject.FindWithTag ("Player");

        playerMovementScript = Player.GetComponent<PlayerControls> ();

        faderScript = GameObject.FindWithTag ("Fader").GetComponent<SceneLoadFader>();
    }

    void OnTriggerEnter2D(Collider2D other)
    {
        if (other.tag == "Player" && isEventful == false)
        {
            StartCoroutine (ChangeScene());
        }
    }

    public void ActivateSceneChange()
    {
        StartCoroutine (ChangeScene());
    }

    IEnumerator ChangeScene()
    {
        faderScript.fadeToBlack = true;

        playerMovementScript.enabled = false;

        GameObject.Find ("SoundManager").GetComponent<SoundMan-
ager> ().ChangeClip (newMusicClipIndex);

        yield return new WaitForSeconds (waitTime);

        GameObject.Find ("GameManager").GetComponent<PlayerPositionHandler>().play-
erStartPosition = playerPositionInNextScene;

        SceneManager.LoadScene (sceneName);

        faderScript.fadeToBlack = false;

        playerMovementScript.enabled = true;
    }
}
```

Appendix 8. Battle Manager script

```
using UnityEngine;
using System.Collections;

public class BattleManager : MonoBehaviour
{
    public GameObject[] CharactersInTurnSequence;
    public bool[] isDead;
    int[] speeds;
    public int turnNumber;
    public bool hasMoved, hasActed, hasFinished;
    CharacterActionSelection actionScript;
    public GameObject ObstacleSpaceParent;
    GameObject[] Obstacles;
    public Vector2[] occupiedSpaces, obstacleSpaces;
    int count;
    public bool battleTypeIsDeathmatch;

    // Use this for initialization
    void Awake ()
    {
        actionScript = gameObject.GetComponent<CharacterActionSelection> ();

        isDead = new bool[CharactersInTurnSequence.Length];

        SortCharacters ();

        CheckOccupiedSpaces ();

        CheckObstacles ();

        Debug.Log ("Please enjoy your imminent destruction");

        actionScript.enabled = true;
    }

    public void SortCharacters()
    {
        Debug.Log ("Checking speeds");

        speeds = new int[CharactersInTurnSequence.Length];

        for (int i = 0; i < speeds.Length; i++) // get speed of the characters
        {
            speeds [i] = CharactersInTurnSequence [i].GetComponent<Character-
Stats> ().currentSpeed;
        }

        Debug.Log ("Speed check complete");

        Debug.Log ("Sorting characters");

        for (int i = 0; i < speeds.Length; i++) // sorting
        {
            for (int j = i + 1; j < speeds.Length; j++)
            {
                if (speeds[i] < speeds[j])
                {
                    GameObject Temp = CharactersInTurnSequence [i];
                    CharactersInTurnSequence [i] = CharactersInTurnSequence [j];
                    CharactersInTurnSequence [j] = Temp;
                }
            }
        }

        Debug.Log ("Character sorting complete");
    }

    public void CheckOccupiedSpaces()
    {
        Debug.Log ("Checking occupied spaces");

        occupiedSpaces = new Vector2[CharactersInTurnSequence.Length];
```

```
        for (int i = 0; i < occupiedSpaces.Length; i++) // check all characters posi-
tions
        {
            occupiedSpaces [i] = new Vector2 (CharactersInTurnSequence[i].trans-
form.position.x, CharactersInTurnSequence[i].transform.position.y);
        }

        Debug.Log ("Space check complete");
    }

    public void CheckObstacles()
    {
        Debug.Log ("Checking obstacles");

        Obstacles = new GameObject[ObstacleSpaceParent.GetComponentsInChildren<Col-
lider2D>().Length];

        for (int i = 0; i < Obstacles.Length; i++)
        {
            Obstacles [i] = ObstacleSpaceParent.GetComponentsInChildren<Col-
lider2D> () [i].gameObject;
        }

        obstacleSpaces = new Vector2[Obstacles.Length];

        for (int i = 0; i < obstacleSpaces.Length; i++)
        {
            obstacleSpaces [i] = new Vector2 (Obstacles[i].transform.position.x, Ob-
stacles[i].transform.position.y + 0.25f);
        }

        Debug.Log ("Obctacle check complete");
    }
}
```

Appendix 9. Enemy AI script

```
using UnityEngine;
using System.Collections;

public class EnemyAI : MonoBehaviour
{
    public float waitTime;
    BattleManager managerScript;
    GameObject[] CharactersInTurnSequence;
    int turnNumber;
    public float[] distancesToPlayerCharacters;
    float distanceToAttack;
    public GameObject PlayerCharacterToAttack;
    GameObject CurrentEnemy;
    public GameObject RangedAttackTiles;
    public bool hasRangedAttack;
    bool hasActed, hasMoved;
    public bool canAttack = false;
    GameObject Selector;
    SelectorForEnemyPurposes selectorScript;
    SelectorMovement selectorMovementScript;
    public Vector3 direction, nextPosition;
    public GameObject ColoredTiles;
    GameObject MovementTiles, AttackTiles;
    public int counter;
    public bool xIsBigger;
    public Vector3[] MoveRange;
    CharacterActionSelection actionScript;
    public GameObject[] Characters;
    public GameObject EnemyTurnIndicator;
    public float minX, maxX, minY, maxY;
    public bool canMove, movementIsRestricted;
    public Vector2 positionToMove;
    public Vector2[] occupiedSpaces;
    public GameObject teleportEffect, MeleeEffect, RangedEffect, HitEffect, MissEf-
fect;

    // Use this for initialization
    void Awake()
    {
        managerScript = gameObject.GetComponent<BattleManager> ();

        CharactersInTurnSequence = managerScript.CharactersInTurnSequence;

        distancesToPlayerCharacters = new float[CharactersInTurnSequence.Length];

        Selector = GameObject.FindWithTag ("Selector");

        selectorScript = Selector.GetComponent<SelectorForEnemyPurposes> ();

        selectorMovementScript = Selector.GetComponent<SelectorMovement> ();

        actionScript = gameObject.GetComponent<CharacterActionSelection> ();
    }

    void OnEnable ()
    {
        turnNumber = managerScript.turnNumber;

        CurrentEnemy = CharactersInTurnSequence [turnNumber];

        hasRangedAttack = CurrentEnemy.GetComponent<Character-
Stats> ().hasRangedAttack;

        selectorMovementScript.forEnemyPurposes = true;

        Selector.GetComponent<SpriteRenderer>().color = new Color (1, 1, 1, 0);

        EnemyTurnIndicator.SetActive (true);

        StartCoroutine(EnemyTurnSequence());
    }

    // Update is called once per frame
```

```
    void Update ()
    {

    }

    IEnumerator EnemyTurnSequence()
    {
        Debug.Log ("Start");

        FindClosestTarget ();

        yield return new WaitForSeconds (waitTime);

        if (hasRangedAttack) // ranged enemy
        {
            CheckDistanceToTarget ();

            PrepareForRangedAttack ();

            yield return new WaitForSeconds (waitTime);

            if (distanceToAttack > 1 && distanceToAttack < 4 && selectorScript.isInAt-
tackableTile) // the target is in range
            {
                DealDamage ();

                FinishRangedAttack ();

                Debug.Log (CurrentEnemy.name + " made a ranged attack");

                EndTurn();
            }

            else
            {
                FinishRangedAttack ();

                PrepareToMove ();

                if (canMove)
                {
                    MoveCharacterToPlaceTheBeginning ();

                    yield return new WaitForSeconds (1f);

                    MoveCharacterToPlaceTheEnd ();
                }

                yield return new WaitForSeconds (waitTime);

                CheckDistanceToTarget ();

                PrepareForRangedAttack ();

                yield return new WaitForSeconds (waitTime);

                if (distanceToAttack > 1 && distanceToAttack < 4 && selector-
Script.isInAttackableTile) // the target is in range
                {
                    DealDamage ();

                    FinishRangedAttack ();

                    Debug.Log (CurrentEnemy.name + " moved and made a ranged attack");

                    EndTurn();
                }

                else
                {
                    FinishRangedAttack ();

                    Debug.Log (CurrentEnemy.name + "only moved");

                    EndTurn();
                }
            }
        }
```

```
        else // melee enemy
        {
            yield return new WaitForSeconds (waitTime);

            CheckDistanceToTarget ();

            if (distanceToAttack <= 1) // the target is in adjacent space
            {
                DealDamage ();

                Debug.Log (CurrentEnemy.name + " made a melee attack");

                EndTurn();
            }

            else
            {
                PrepareToMove ();

                if (canMove)
                {
                    MoveCharacterToPlaceTheBeginning ();

                    yield return new WaitForSeconds (1f);

                    MoveCharacterToPlaceTheEnd ();
                }

                yield return new WaitForSeconds (waitTime);

                CheckDistanceToTarget ();

                if (distanceToAttack <= 1) // the target is in adjacent space
                {
                    DealDamage ();

                    Debug.Log (CurrentEnemy.name + " moved and made a melee attack");

                    EndTurn();
                }

                else
                {
                    Debug.Log (CurrentEnemy.name + "only moved");

                    EndTurn();
                }
            }

        }
    }

    public void EndTurn()
    {
        Debug.Log ("Ending turn");

        if (managerScript.turnNumber < managerScript.CharactersInTurnSe-
quence.Length - 1)
        {
            managerScript.turnNumber++;
        }

        else
        {
            managerScript.turnNumber = 0;
        }

        selectorMovementScript.isMoving = false;
        selectorMovementScript.isAttacking = false;
        selectorMovementScript.forEnemyPurposes = false;
        selectorMovementScript.enabled = false;

        actionScript.enabled = true;

        EnemyTurnIndicator.SetActive (false);

        this.enabled = false;
    }
```

```
    public void PrepareForRangedAttack()
    {
        PlayerCharacterToAttack.GetComponent<Collider2D> ().enabled = false; // disa-
ble target collider

        AttackTiles = (GameObject)Instantiate (RangedAttackTiles, CurrentEnemy.trans-
form.position, Quaternion.identity); // spawn ranged attack tiles

        // disable Selector's ShowStats and movement

        Selector.GetComponent<ShowStats> ().enabled = false;
        Selector.GetComponent<SelectorMovement>().enabled = false;

        // Enable Selector's SelectorForEnemyPurposes and the selector itself

        selectorScript.enabled = true;

        // move the Selector on the target

        Selector.transform.position = PlayerCharacterToAttack.transform.posi-
tion - new Vector3(0, 0.25f, 0);
    }

    public void FinishRangedAttack()
    {
        // Disable Selector's SelectorForEnemyPurposes and the selector itself

        selectorScript.enabled = false;

        // enable Selector's ShowStats

        Selector.GetComponent<ShowStats> ().enabled = true;
        Selector.GetComponent<SelectorMovement>().enabled = true;

        Destroy (AttackTiles); // destroy the attack tiles

        PlayerCharacterToAttack.GetComponent<Collider2D> ().enabled = true; // ena-
ble target collider
    }

    public void CheckDistanceToTarget()
    {
        if (PlayerCharacterToAttack != null)
        {
            distanceToAttack = Vector3.Distance (CurrentEnemy.transform.posi-
tion, PlayerCharacterToAttack.transform.position);
        }
    }

    public void FindClosestTarget()
    {
        Debug.Log ("Finding closest target...");

        Characters = new GameObject[CharactersInTurnSequence.Length];

        for (int i = 0; i < CharactersInTurnSequence.Length; i++)
        {
            distancesToPlayerCharacters [i] = Vector3.Distance (CurrentEnemy.trans-
form.position, CharactersInTurnSequence [i].transform.position);

            Characters[i] = CharactersInTurnSequence[i];
        }

        for (int i = 0; i < distancesToPlayerCharacters.Length; i++)
        {
            for (int j = i + 1; j < distancesToPlayerCharacters.Length; j++)
            {
                if (distancesToPlayerCharacters[i] > distancesToPlayerCharac-
ters[j])
                {
                    float temp = distancesToPlayerCharacters[i];
                    distancesToPlayerCharacters[i] = distancesToPlayerCharacters[j];
                    distancesToPlayerCharacters[j] = temp;

                    GameObject Temp = Characters [i];
                    Characters [i] = Characters [j];
                    Characters [j] = Temp;
                }
            }
```

```
        }

        for (int i = 0; i < distancesToPlayerCharacters.Length; i++)
        {
            if (distancesToPlayerCharacters[i] > 0 && Charac-
ters[i].tag == "Player" && Characters[i].GetComponent<CharacterDeath>().is-
Dead == false
                && Characters[i].GetComponent<CharacterDeath>().isRegenerat-
ing == false)
            {
                PlayerCharacterToAttack = Characters [i];

                Debug.Log ("Closest target: " + PlayerCharacterToAttack.name);

                distanceToAttack = distancesToPlayerCharacters [i];

                break;
            }
        }
    }

    public void DealDamage()
    {
        if (hasRangedAttack)
        {
            // calculate angle from enemy to player

            float angle = Mathf.Atan2(PlayerCharacterToAttack.transform.posi-
tion.y - CurrentEnemy.transform.position.y,
                                        PlayerCharacterToAttack.transform.posi-
tion.x - CurrentEnemy.transform.position.x) * 180 / Mathf.PI;

            Destroy(Instantiate (RangedEffect, CurrentEnemy.transform.position, Qua-
ternion.Euler(0, 0, angle)), 1f);
        }

        else
        {
            // check from which direction the attack is coming

            Vector3 attackDirection = (PlayerCharacterToAttack.transform.posi-
tion - CurrentEnemy.transform.position).normalized;

            GameObject Melee = (GameObject)Instantiate (MeleeEffect, PlayerCharacter-
ToAttack.transform.position, Quaternion.identity);

            if (attackDirection.x >= 0)
            {
                Melee.GetComponent<SpriteRenderer> ().flipX = true;
            }

            else
            {
                Melee.GetComponent<SpriteRenderer> ().flipX = false;
            }

            Destroy(Melee, 1f);
        }

        // hit chance calculation, if target's speed is higher -
> the chance is lower, if attacker's speed is higher -> chance is higher

        int hitChance = 90 - (PlayerCharacterToAttack.GetComponent<Character-
Stats> ().currentSpeed
                                    - CurrentEnemy.GetComponent<Character-
Stats> ().currentSpeed);

        if (Random.Range(0, 100) <= hitChance)
        {
            // It's a hit!

            Destroy(Instantiate (HitEffect, PlayerCharacterToAttack.transform.posi-
tion, Quaternion.identity), 1f);

            int damage = CurrentEnemy.GetComponent<CharacterStats> ().currentAttack
                - PlayerCharacterToAttack.GetComponent<CharacterStats> ().cur-
rentDefense;

            if (damage > 0)
```

```
                {
                    PlayerCharacterToAttack.GetComponent<CharacterStats> ().curren-
tHealth -= damage;
                }

                else
                {
                    PlayerCharacterToAttack.GetComponent<CharacterStats> ().curren-
tHealth -= 1;
                }

                // show / update hp bar here maybe?

                Debug.Log (CurrentEnemy.name + " attacks " + PlayerCharacter-
ToAttack.name + " for " + damage + " damage!");
            }

            else
            {
                // miss

                Destroy(Instantiate (MissEffect, PlayerCharacterToAttack.transform.posi-
tion, Quaternion.identity), 1f);
            }
        }

        public void PrepareToMove()
        {
            occupiedSpaces = new Vector2[managerScript.occupiedSpaces.Length + manager-
Script.obstacleSpaces.Length];

            for (int i = 0; i < managerScript.occupiedSpaces.Length; i++)
            {
                occupiedSpaces [i] = managerScript.occupiedSpaces [i];
            }

            for (int i = 0, j = managerScript.occupiedSpaces.Length; i < managerScript.ob-
stacleSpaces.Length; i++, j++)
            {
                occupiedSpaces [j] = managerScript.obstacleSpaces [i];
            }

            if (hasRangedAttack) // ranged enemy movement preparations
            {
                for (int i = 4; i < 52; i++) // check the positions near the tar-
get, but ignore the adjacent ones
                {
                    Vector2 spaceToCompare = new Vector2 (PlayerCharacterToAttack.trans-
form.position.x + MoveRange[i].x,
                                                          PlayerCharacterToAttack.trans-
form.position.y + MoveRange[i].y);

                    Debug.Log (spaceToCompare.ToString());

                    float distanceToMove = Vector2.Distance (CurrentEnemy.transform.posi-
tion, spaceToCompare);

                    for (int j = 0; j < occupiedSpaces.Length; j++)
                    {
                        Debug.Log ("Comparing potential movable space #" + i + " to occu-
pied space #" + j);

                        if (spaceToCompare == occupiedSpaces[j] || distanceToMove > 5f
                            || spaceToCompare.x > maxX || spaceToCompare.x < minX
                            || spaceToCompare.y > maxY || spaceToCompare.y < minY)
                        {
                            movementIsRestricted = true;

                            positionToMove = Vector2.zero;

                            break;
                        }

                        else
                        {
                            movementIsRestricted = false;

                            positionToMove = spaceToCompare;
                        }
```

```
                    }

                    if (movementIsRestricted == false)
                    {
                        break;
                    }
                }

                if (movementIsRestricted) // move forward if able
                {
                    for (int i = 5; i > 0; i--)
                    {
                        direction = (PlayerCharacterToAttack.transform.position - Curren-
tEnemy.transform.position).normalized;

                        Vector2 spaceToCompare = new Vector2(CurrentEnemy.transform.posi-
tion.x + Mathf.CeilToInt(direction.x * i),
                                                                        CurrentEnemy.transform.po-
sition.y + Mathf.CeilToInt(direction.y * i));

                        float distanceToMove = Vector2.Distance (CurrentEnemy.trans-
form.position, spaceToCompare);

                        for (int j = 0; j < occupiedSpaces.Length; j++)
                        {
                            Debug.Log ("Comparing potential mova-
ble space #" + i + " to occupied space #" + j);

                            if (spaceToCompare == occupiedSpaces[j] || distanceToMove > 5f
                                || spaceToCompare.x > maxX || spaceToCompare.x < minX
                                || spaceToCompare.y > maxY || spaceToCompare.y < minY)
                            {
                                movementIsRestricted = true;

                                positionToMove = Vector2.zero;

                                break;
                            }

                            else
                            {
                                movementIsRestricted = false;

                                positionToMove = spaceToCompare;
                            }
                        }

                        if (movementIsRestricted == false)
                        {
                            break;
                        }
                    }
                }

                if (movementIsRestricted == false)
                {
                    canMove = true;

                    Debug.Log ("Movable space found: " + positionToMove.ToString());
                }

                else
                {
                    positionToMove = Vector2.zero;

                    Debug.Log ("No movable space found");
                }
            }

            else // melee enemy movement preparations
            {
                for (int i = 0; i < 52; i++) // check the positions near the target, in-
cluding the adjacent ones
                {
                    Vector2 spaceToCompare = new Vector2 (PlayerCharacterToAttack.trans-
form.position.x + MoveRange[i].x,
                        PlayerCharacterToAttack.transform.position.y + MoveRange[i].y);

                    Debug.Log (spaceToCompare.ToString());
```

```
                float distanceToMove = Vector2.Distance (CurrentEnemy.transform.posi-
tion, spaceToCompare);

                for (int j = 0; j < occupiedSpaces.Length; j++)
                {
                    Debug.Log ("Comparing potential movable space #" + i + " to occu-
pied space #" + j);

                    if (spaceToCompare == occupiedSpaces[j] || distanceToMove > 5f
                        || spaceToCompare.x > maxX || spaceToCompare.x < minX
                        || spaceToCompare.y > maxY || spaceToCompare.y < minY)
                    {
                        movementIsRestricted = true;

                        positionToMove = Vector2.zero;

                        break;
                    }

                    else
                    {
                        movementIsRestricted = false;

                        positionToMove = spaceToCompare;
                    }
                }

                if (movementIsRestricted == false)
                {
                    break;
                }
            }

            if (movementIsRestricted) // move forward if able
            {
                for (int i = 5; i > 0; i--)
                {
                    direction = (PlayerCharacterToAttack.transform.position - Curren-
tEnemy.transform.position).normalized;

                    Vector2 spaceToCompare = new Vector2(CurrentEnemy.transform.posi-
tion.x + Mathf.CeilToInt(direction.x * i),
                        CurrentEnemy.transform.position.y + Mathf.CeilToInt(direc-
tion.y * i));

                    float distanceToMove = Vector2.Distance (CurrentEnemy.trans-
form.position, spaceToCompare);

                    for (int j = 0; j < occupiedSpaces.Length; j++)
                    {
                        Debug.Log ("Comparing potential mova-
ble space #" + i + " to occupied space #" + j);

                        if (spaceToCompare == occupiedSpaces[j] || distanceToMove > 5f
                            || spaceToCompare.x > maxX || spaceToCompare.x < minX
                            || spaceToCompare.y > maxY || spaceToCompare.y < minY)
                        {
                            movementIsRestricted = true;

                            positionToMove = Vector2.zero;

                            break;
                        }

                        else
                        {
                            movementIsRestricted = false;

                            positionToMove = spaceToCompare;
                        }
                    }

                    if (movementIsRestricted == false)
                    {
                        break;
                    }
                }
            }
```

```
            if (movementIsRestricted == false)
            {
                canMove = true;

                Debug.Log ("Movable space found: " + positionToMove.ToString());
            }

            else
            {
                positionToMove = Vector2.zero;

                Debug.Log ("No movable space found");
            }
        }
    }

    public void MoveCharacterToPlaceTheBeginning()
    {
        GameObject teleportEffect2 = (GameObject)Instantiate (teleportEffect, Curren-
tEnemy.transform.position + new Vector3(0, 0.75f, 0), Quaternion.identity);

        Destroy (teleportEffect2, 2.0f);

        CurrentEnemy.GetComponent<SpriteRenderer> ().color = new Color (1, 1, 1, 0);

        // Update the occupied spaces array in the manager script here

        Vector2 oldPosition = new Vector2 (CurrentEnemy.transform.position.x, Curren-
tEnemy.transform.position.y);

        for (int i = 0; i < managerScript.occupiedSpaces.Length; i++)
        {
            if (managerScript.occupiedSpaces[i] == oldPosition)
            {
                managerScript.occupiedSpaces [i] = positionToMove;

                Debug.Log ("Old vector " + oldPosition.ToString() + " in the occupied-
Spaces array was replaced by" + positionToMove.ToString());

                break;
            }
        }

        CurrentEnemy.transform.position = positionToMove;
    }

    public void MoveCharacterToPlaceTheEnd()
    {
        GameObject teleportEffect3 = (GameObject)Instantiate (teleportEffect, Curren-
tEnemy.transform.position + new Vector3(0, 0.75f, 0), Quaternion.identity);

        Destroy (teleportEffect3, 2.0f);

        CurrentEnemy.GetComponent<SpriteRen-
derer> ().color = new Color (1, 25f/255f, 25f/255f, 1);
    }
}
```

Appendix 10. Base Skill script

```
using UnityEngine;
using System.Collections;

public class BaseSkill : MonoBehaviour
{
    public string skillName;
    public int cooldown;
    public int targetTag; // 0 = enemy, 1 = ally
    public bool canTargetSelf;
    public bool isRanged;
    public bool userIsEnemy;
    public bool isDirectHit;
    public GameObject Target, ColoredTiles;
}
```

Appendix 11. Damage Skill script

```
using UnityEngine;
using System.Collections;

public class DamageSkill : BaseSkill
{
    public int damageMultiplier;
    public bool ignoresDefense;
    SelectorForSkills selectorScript;
    public bool usesAttackEffect;
    public GameObject AttackEffect, HitEffect, MissEffect;

    public virtual void Awake()
    {
        selectorScript = GameObject.Find ("Selector").GetComponent<SelectorFor-
Skills> ();
    }

    public virtual void OnEnable()
    {
        StartCoroutine (SkillSequence());
    }

    public virtual IEnumerator SkillSequence()
    {
        GetTheTarget ();

        yield return new WaitForSeconds (0.1f);

        if (usesAttackEffect)
        {
            CreateAttackEffect ();

            yield return new WaitForSeconds (0.1f);
        }

        CreateHitEffect ();

        yield return new WaitForSeconds (0.5f);

        this.enabled = false;
    }

    public virtual void GetTheTarget()
    {
        if (userIsEnemy)
        {

        }

        else
        {
            Target = selectorScript.Target;
        }
    }

    public virtual void CreateAttackEffect()
    {
        if (isRanged)
        {
            // calculate angle from enemy to player

            float angle = Mathf.Atan2(Target.transform.position.y - gameObject.trans-
form.position.y,
                Target.transform.position.x - gameObject.transform.posi-
tion.x) * 180 / Mathf.PI;

            Destroy(Instantiate (AttackEffect, gameObject.transform.position, Quater-
nion.Euler(0, 0, angle)), 1f);
        }

        else
        {
            // check the direction from which the attack is coming
```

```
            Vector3 attackDirection = (Target.transform.position - gameObject.trans-
form.position).normalized;

            GameObject Melee = (GameObject)Instantiate (AttackEffect, Target.trans-
form.position, Quaternion.identity);

            if (attackDirection.x >= 0)
            {
                Melee.GetComponent<SpriteRenderer> ().flipX = true;
            }

            else
            {
                Melee.GetComponent<SpriteRenderer> ().flipX = false;
            }

            Destroy(Melee, 1f);
        }
    }

    public virtual void CreateHitEffect()
    {
        // the random number is randomized between 0 and 99 (100 is excluded with in-
teger parameters)
        // if it is lower or the same as hit chance, the attack hits

        if (Random.Range(0, 100) <= CalculateHitChance())
        {
            // It's a hit!

            Destroy(Instantiate (HitEffect, Target.transform.position, Quater-
nion.identity), 1f);

            DealDamage (CalculateDamage ());
        }

        else
        {
            // miss

            Debug.Log ("You missed!");

            Destroy(Instantiate (MissEffect, Target.transform.position, Quater-
nion.identity), 1f);
        }
    }

    public virtual int CalculateHitChance()
    {
        int hitChance;

        if (isDirectHit)
        {
            hitChance = 100;
        }

        else // if target's speed is higher -> the change is lower, if at-
tacker's speed is higher -> change is higher
        {
            hitChance = 90 - (Target.GetComponent<CharacterStats> ().cur-
rentSpeed - gameObject.GetComponent<CharacterStats> ().currentSpeed);
        }

        return hitChance;
    }

    public virtual int CalculateDamage()
    {
        int damage;

        if (ignoresDefense) // if the skill deals defense ignoring damage
        {
            damage = gameObject.GetComponent<CharacterStats> ().currentMagic * dam-
ageMultiplier;
        }

        else
        {
```

```
            damage = gameObject.GetComponent<CharacterStats> ().currentMagic * dam-
ageMultiplier - Target.GetComponent<CharacterStats> ().currentDefense;
        }

        return damage;
    }

    public virtual void DealDamage(int damage)
    {
        int finalDamage = damage;

        if (finalDamage > 0 && damageMultiplier > 0) // deals normal damage
        {
            Target.GetComponent<CharacterStats> ().currentHealth -= finalDamage;

            Debug.Log (gameObject.name + " uses " + skillName + "! " + Tar-
get.name + " takes " + finalDamage + " damage!");
        }

        else if (damageMultiplier < 0) // deals negative dam-
age, in other words, heals the target
        {
            Target.GetComponent<CharacterStats> ().currentHealth -= finalDamage;

            if (Target.GetComponent<CharacterStats> ().currentHealth > Target.GetCom-
ponent<CharacterStats> ().maxHealth)
            {
                Target.GetComponent<CharacterStats> ().currentHealth = Target.GetCom-
ponent<CharacterStats> ().maxHealth;
            }

            Debug.Log (gameObject.name + " uses " + skillName + "! " + Tar-
get.name + " heals " + -finalDamage + " damage!");
        }

        else // if the defense of the target is too high, the dam-
age could be zero or negative
        {
            Target.GetComponent<CharacterStats> ().currentHealth -
= 1; // so the skill should make at least one point of damage

            Debug.Log (gameObject.name + " uses " + skillName + "! " + Tar-
get.name + " takes 1 damage!");
        }
    }


}
```

Appendix 12. Absorb Damage Skill script

```
using UnityEngine;
using System.Collections;

public class AbsorbDamageSkill : DamageSkill
{
    public float absorbPercentage;
    public GameObject HealEffect;

    public override IEnumerator SkillSequence()
    {
        GetTheTarget ();

        yield return new WaitForSeconds (0.1f);

        if (usesAttackEffect)
        {
            CreateAttackEffect ();

            yield return new WaitForSeconds (0.1f);
        }

        CreateHitEffect ();

        yield return new WaitForSeconds (0.5f);

        HealSelf ();

        yield return new WaitForSeconds (0.5f);

        this.enabled = false;
    }

    public void HealSelf()
    {
        int healAmount = Mathf.CeilToInt(absorbPercentage * CalculateDamage ());

        gameObject.GetComponent<CharacterStats> ().currentHealth += healAmount;

        if (gameObject.GetComponent<CharacterStats> ().currentHealth > gameObject.Get-
Component<CharacterStats> ().maxHealth)
        {
            gameObject.GetComponent<CharacterStats> ().currentHealth = gameObject.Get-
Component<CharacterStats> ().maxHealth;
        }

        Debug.Log (gameObject.name + " heals " + healAmount + " damage!");

        Destroy(Instantiate (HealEffect, gameObject.transform.position, Quater-
nion.identity), 1f);
    }
}
```

Appendix 13. Skillbook script

```
using UnityEngine;
using System.Collections;

public class Skillbook : MonoBehaviour
{
    public BaseSkill[] Skills = new BaseSkill[2];
    public int currentSkillInUse;
    SelectorForSkills selectorScript;
    GameObject AttackTiles;

    void Awake()
    {
        selectorScript = GameObject.Find ("Selector").GetComponent<SelectorFor-
Skills> ();
    }

    void OnEnable()
    {
        selectorScript.enabled = true;

        if (gameObject.GetComponent<CharacterStats>().isEnemy == false)
        {
            AttackTiles = (GameObject) Instantiate (Skills [currentSkillInUse].Col-
oredTiles, gameObject.transform.position, Quaternion.identity);
        }
    }

    void Update()
    {
        if (selectorScript.canAttack && Input.GetButton("Submit"))
        {
            Skills [currentSkillInUse].enabled = true;

            selectorScript.enabled = false;

            Destroy (AttackTiles);

            this.enabled = false;
        }
    }
}
```