

Jari Tervo

Development of supplementary material for teaching game development

Bachelor's thesis
Information Technology

2019



South-Eastern Finland
University of Applied Sciences

Tekijä/Tekijät	Tutkinto	Aika
Jari Tervo	Insinööri (AMK)	Marraskuu 2019
Opinnäytetyön nimi		
Opetusmateriaalin kehittäminen peliohjelmoinnin koulutukseen		37 sivua 5 liitesivua
Toimeksiantaja		
Kaakkois-Suomen ammattikorkeakoulu Oy		
Ohjaaja		
Niina Mässeli		
Tiivistelmä		
<p>Tämän opinnäytetyön aiheena oli kehittää opetusmateriaalia peliohjelmoinnin koulutusta varten Kaakkois-Suomen ammattikorkeakoulun GameLabille. Opetusmateriaali kehitettiin pelikokoelman muodossa, jossa kokoelman pelit koostuivat samoista modulaarisista osista. Kyseisiä osia voitiin käyttää myös omien pelien luomiseen, jopa aloittelijan toimesta.</p> <p>Kehityksen aikana siirryttiin kuitenkin pois modulaarisuudesta kohti yksilökeskeisyyttä siksi, että sisältö alkoi keskittyä yhä enemmän koodaamiseen. Pelit erotettiin ja niitä lähestyttiin yksilöinä, ja jokaiselle pelille luotiin ohjeistus pelin luomiseen sekä yleistä dokumentaatiota pelin komponenteista. Lopullinen tuote muodostui siis kokoelmasta pelejä sekä niiden dokumentaatiosta.</p> <p>Pelikokoelman pelit tuotettiin Unity-pelimoottorilla käyttäen Visual Studiota koodin kirjoittamiseen ja Blender-mallinnusohjelmaa grafiikkaa varten. Opinnäytetyö käy läpi yhden pelin luomisen kokoelman peleistä. Kyseinen peli oli ensimmäisen persoonan ammuntopeli, joka otti inspiraatiota villistä lännestä sekä ampumaradoista. Opinnäytetyössä käydään myös läpi dokumentaatio, joka kehitettiin pelin pohjalta. Siihen kuuluvat Doxygen-työkalulla koodista generoitu tietokanta sekä yksityiskohtainen ohje pelin luomiseen.</p> <p>Opinnäytetyön seurauksena toteutettu pelikokoelma on otettu jo käyttöön. Paria poikkeamaa lukuun ottamatta kokoelman vastaanotto on ollut positiivista ja sitä on käytetty peliohjelmoinnin opettamiseen uusille oppilaille. Ilman lisätestaamista ei voida kuitenkaan sanoa, kuinka hyvin tuote on lopulta onnistunut tehtävässään.</p>		
Avainsanat		
dokumentointi, pelikehitys, Unity, opetus, modulaarisuus		

Author (authors)	Degree	Time
Jari Tervo	Bachelor of Engineering	November 2019
Thesis title		
Development of supplementary material for teaching game development		37 pages 5 pages of appendices
Commissioned by		
South-Eastern Finland University of Applied Sciences		
Supervisor		
Niina Mässeli		
Abstract		
<p>The purpose of this thesis was to develop supplementary material for teaching game development for the South-Eastern Finland University of Applied Sciences GameLab. The supplementary material took the form of a game collection where the games were built from modular pieces that could be used for creating your own games, even by a beginner.</p> <p>During the development, however, the approach changed from modularity to individuality, when coding started to become the focus, and the games of the collection were divided into separate items. Each was accompanied by a tutorial piece on how to make the game and some general documentation on its components. The new product in the making was a composition of the games and their documentation.</p> <p>The games of the collection were created with the Unity game engine, using Visual Studio for writing the code and Blender for the graphics. This thesis dealt with the creation process of one of the games, a first-person shooter game inspired a wild west aesthetic and shooting ranges. The thesis also dealt with the documentation made for the game: a generated database made from the code using Doxygen and the tutorial page hosted on the project's website.</p> <p>The usage of the game collection had already started as of writing this thesis. Except for a few issues, the reception was positive, and the collection has been used for teaching new students. However, more testing is required for a definite conclusion of the product's success.</p>		
Keywords		
documentation, game development, Unity, teaching, modularity		

CONTENTS

TERMS AND ABBREVIATIONS.....	6
1 INTRODUCTION.....	7
2 DESIGN.....	8
2.1 Modular game collection.....	8
2.2 Pelipalapeli reformed.....	11
2.3 First-person shooter.....	12
2.4 Tools used.....	14
3 FIRST-PERSON SHOOTER.....	16
3.1 Modular components.....	17
3.2 Player and movement.....	18
3.3 Shooting mechanics.....	21
3.4 Targets and spawning.....	23
3.5 Score and HUD.....	27
3.6 Finishing touches.....	28
3.7 Graphics.....	28
4 DOCUMENTATION.....	30
4.1 Doxygen.....	31
4.2 Tutorials.....	31
5 CONCLUSIONS AND FUTURE DEVELOPMENT.....	33
REFERENCES.....	35
LIST OF FIGURES.....	37

APPENDICES

Appendix 1. Game element table

Appendix 2. Timer class

Appendix 3. Resource class

Appendix 4. Object Pooler class

Appendix 5. Pelipalapeli: commenting guidelines

TERMS AND ABBREVIATIONS

AI	Stands for Artificial Intelligence. A way of simulating human intelligence.
Algorithm	A segment of code that has the purpose of doing something. For example, calculating the speed of an object.
C#, C++	Programming languages derived from C.
Class	A template for creating objects in Object-Oriented-Programming.
Command line	A tool for commanding the operating system directly.
Comment	A text segment in the code with the purpose of guiding the reader.
Function	A block of code that can be called with a single command.
Hit point	A general convention in game development for keeping track of the player's health. Typically, when hit points run out, the player character dies.
HTML	Stands for Hypertext Markup Language. Used for making websites and HTML documents.
HUD	Stands for head-up display. A general convention in video games for displaying data on the player's screen.
IDE	Stands for Integrated Development Environment. Contains a text editor for writing code and usually allows for generating software from said code.
Method	Same as a function but for classes.
Pathfinding	A method of finding one's way for AI.
Quaternion	Used to represent three-dimensional rotation.
Spawning	A concept of creating objects with code. Despawning is the opposite, where something is removed.
Syntax	In programming, denotes rules and principles on how code is written.
UI	Stands for User Interface. Refers to interaction between humans and machines. Encompasses controls, HUD and various other things.

1 INTRODUCTION

This document sets out to explore the overall development and design process of the game collection called Pelipalapeli. The game collection was commissioned by the South-Eastern Finland University of Applied Sciences GameLab in preparation for the next semester. The game collection is to serve as the foundation for teaching game development's basics to new students and its development was the objective of this thesis.

GameLab intends to use the collection in conjunction with traditional teaching methods for preparing the students for the usage of Unity in their own projects and in further studies. One of the intentions is also to lower the threshold for attending game development events, such as game jams, by providing material for the students to rely on.

The document also takes a deeper dive into content that makes up the game collection. Based on the specifications of the final design, a group of games and documentation based on those games were made. This document will deal with the development of one of the games from the collection: a shooting range inspired first-person shooter game.

The game collection's development team consisted of three students nearing the end of their studies, who were approached to work on the project due to their previous achievements. As such, multiple theses were made regarding the project, each approaching it from a different angle. For further information about the subject, it is advised to seek out the theses of Sami Krouvi and Juho Koponen, to get a better overview. Koponen made a comprehensive thesis about the documentation of the project, which this thesis will only go over briefly (Koponen 2019).

2 DESIGN

This chapter details the overall design process of the game collection and its contents. The project itself went through major revisions during its early development, which are explored in further detail as the design is introduced.

The focus then shifts to a game made for the collection. A first-person shooter was made based on the previously mentioned revisions and its design is detailed here, along with the tools used for making it.

2.1 Modular game collection

The concept of a modular game collection came from the commissioners at the start of the project. The idea was to create approximately 10 games that shared components between each other, allowing the end user to create their own games from said components.

Modularization in software development is described as a process of dividing a system into more manageable components, or modules, which is what the project set out to do. It can greatly decrease the complexity of the system, due to it being split up into smaller pieces, making development easier. (Otero 2012, 25.)

As the finished project was intended to be used by first-time game developers, the simplification of the systems provided by modularization was a good start. The components would also have to be well documented and simple to use, with minimal coding required. The games itself would have to be simple and similar enough for the component sharing to work but fundamentally different enough as to not feel like the games were all the same.

To further this goal, rudimentary plans were made about what games the final package would include and what components the different games could share. Appendix 1 details categorization for the various pieces the games could consist of, what games were selected and who would be responsible for piecing it together. The selected game types, or genres, that were chosen to be

represented in the project were chosen for their simplicity and similarity when it came to individual game mechanics. As an example, the first-person shooter and the survival horror games could share control schemes for the player character, that being the first-person camera listed under camera types and two-axis movement listed under movement types.

Early in testing, rudimentary coding was deemed necessary as different mechanics, such as types of movement, were decided to be categorized under various classes. Classes like this and their methods could then be used by the end user, with the heavy lifting being done inside the class whose methods a user could then use with simple syntax. Complex mechanics would be accessed with a couple of words.

Figure 1 showcases one of such classes that was used for testing the concept. It includes straightforward methods for manipulating objects such as MoveSimple, which moves the inserted Unity game object `obj`, along the directional vector `dir` for the distance `v` which represents velocity. The class also includes more complicated methods such as RotateToDirection which returns a quaternion which represents the `dir`-vector's directional rotation in a 3D space and as such, does nothing on its own and requires an algorithm that makes use of the returned quaternion. Instructions as to how it would be done, would have been included in the documentation.

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
7  public static class Action
8  {
15 public static void Jump(Rigidbody rb)
16 {
17     // REMINDER: Needs a way to check for the ground collision (raycast or collider?)
18     // Add parameter for jump height (gotten from math and physics)
19     rb.AddForce(Vector3.up * 15.0f, ForceMode.VelocityChange);
20 }
21
32 public static void MoveSimple(GameObject obj, Vector3 dir, float v)
33 {
34     obj.transform.position = obj.transform.position + dir.normalized * v * Time.deltaTime;
35 }
36
48 public static Quaternion RotateToDirection(Vector3 dir)
49 {
50     // REMINDER: Check if RotateToDirection() could also support 3D-rotation.
51     return Quaternion.Euler(0, 0, Mathf.Atan2(dir.y, dir.x) * Mathf.Rad2Deg);
52 }
53
66 public static GameObject Spawn(GameObject obj, Vector3 pos, Quaternion rot)
67 {
68     if (obj != null)
69     {
70         return MonoBehaviour.Instantiate(obj, pos, rot);
71     }
72     else
73     {
74         Debug.LogError("PPP Error: Could not spawn object. No reference (null).");
75         return null;
76     }
77 }
78
90 public static GameObject Spawn(GameObject obj, GameObject spawner)
91 {
92     if (obj != null)
93     {
94         return MonoBehaviour.Instantiate(obj, spawner.transform.position, spawner.transform.rotation);
95     }
96     else
97     {
98         Debug.LogError("PPP Error: Could not spawn object. No reference (null).");
99         return null;
100    }
101 }
102

```

Figure 1. Proposed Action-class

The concept was ultimately abandoned after its problems started to become more apparent. By increasing the amount of actual coding required to make use of the components, the original idea of the project started to get lost. Instead of building the game from recognizable pieces and connecting them together, the game creation process would be abstract and require things such as writing code that utilized the premade methods. Anything remotely complicated required its own custom script anyway. Complex and specialized systems such as a spawning algorithm or a state machine for game states would have to be custom built case by case, further hurting the concept as a beginner's first touch into game development. Such things would have to be separately explained step-by-step to a beginner, which is exactly where the project was heading towards.

2.2 Pelipalapeli reformed

The project was repurposed with a different approach and more limited modularity. The final concept was much simpler: same list of games, as detailed in Appendix 1, would be created, but this time they would be self-contained. No longer sharing parts, the focus would be on the games themselves and how they were made. However, certain already done modular components were repurposed for the final design, such as the Timer class seen in Appendix 2, to not waste the time it took to make them.

Instead of the parts the games consisted of being generic, they would be specialized and only work within the context of the specific game. The parts could then be repurposed, reverse-engineered and placed differently inside of the game world to create something familiar but still new. The first-person shooter game would not stop being a first-person shooter game but at the very least, it could be customized by the user to better serve their vision.

Each game would be provided with detailed instructions as to how one could recreate it and its parts. Essentially, a tutorial. Because the end users would mostly consist of people who had never programmed before, it was also decided that the code would have to feature extensive commentary. This would also allow for increased complexity when it came to the games themselves. More complex concepts could be taught, since the instructions would be presented in a step-by-step format.

The tutorial based solution however, could introduce the dilemma touched upon by Rosalind Driver. Although a student might follow set instructions and successfully reach their goal, they might learn the wrong things or misunderstand the conclusion. In the scope of the project, the student might successfully complete the tutorial and finish creating the game, but they might get the wrong idea from certain parts of it, such as why something was done. (Driver 1994, 41-48.)

Responsibility for avoiding the problem ultimately falls upon the teachers who will be using the game collection, but it is important to consider from a development standpoint as well. Things that can be misunderstood should be explained in the tutorials so that false information isn't accidentally taught. Explaining why something is done would also break the monotony of simply doing as told in the tutorial.

2.3 First-person shooter

One of the games to be created was a first-person shooter, also known as an FPS, a subtype of the shooter genre which itself is a subtype of the action genre. In these games, the player typically controls a character from a first-person perspective, with the illusion further supported by a hand holding a weapon in front of them, simulating the player's own involvement. Shooter games of all kinds are often violent and can feature complex game mechanics such as different movement options, as opposed to simply walking, and weapons but the game to be created for the collection would be somewhat different. (Mitchell 2012, 31-33.)

The game would take place in a themed shooting range, featuring life sized targets for the player to shoot at. Instead of using systems too complex for a beginner, such as AI pathfinding, the targets would simply appear from a certain point, move a preset distance in the preset direction and then disappear. Shooting them during the time it takes for them to move, would then cause them to disappear and reward the player with points based on the target type. Certain targets would remove points to discourage shooting anything that moves and as such, the different target types would have to be identifiable with a glance. To achieve this, different targets would have a different picture on them.

Shooting mechanics would be accomplished with hitscan, which means projecting a ray in a specified direction and testing for appropriate physics collisions (Jung 2018). The ray in question, would be the normal of the player's camera, or viewport, and would test against collisions with the environment and the targets. Hitting a target would do as specified previously but hitting the

environment would create a bullet hole to visually demonstrate hitting something else.

In Figure 2, the concept is visualized in a quick mockup. Targets to shoot appear from visually obstructed positions and continue to move to a different obstructed position where they promptly disappear. Pictured is also the general layout of the UI, which features the player's ammo count along with their score value. The player's gun would be rendered on top of everything else, separate from the rest of the game world. Two types of target movement options would be featured in the game. First option would be moving from point A to point B and then disappearing, intended to be used for targets that travel longer distance across the game world. The second movement option would differ in that they would return to the starting point before disappearing, with the intention being targets that pop or peek from behind a wall or other obstruction before returning to being hidden before disappearing completely.

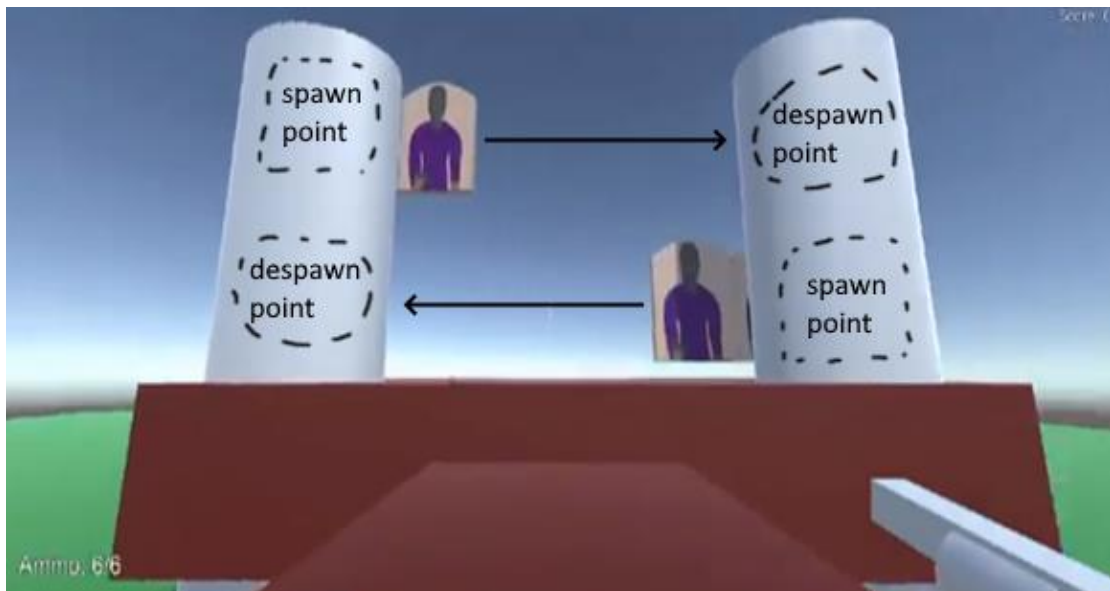


Figure 2. Mockup of the FPS game

The game world would be structured like a corridor in which the player can move freely. The player would begin from what could be considered the entrance and then continue to move towards the exit while waves of targets would spawn at specific points. Reaching the exit would end the game and display the player's final score. Figure 3 demonstrates how the game world would be structured. A

shooting event begins once the player walks inside of its boundaries and ends once the last targets have been defeated or they have destroyed themselves.

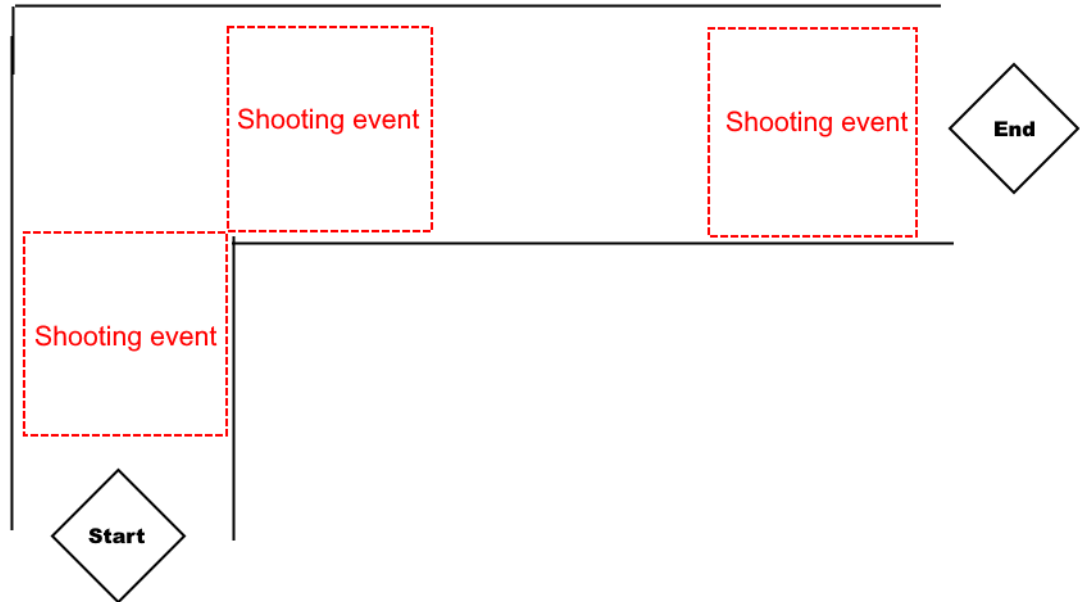


Figure 3. FPS level layout

Once the game ends, the player is prompted to restart. The game cannot be lost, as there is no lose condition such as the player dying or time running out. The score the player gains ultimately has no meaning and won't influence the win condition. It simply serves as an estimation of how well the player did.

2.4 Tools used

This chapter goes through the tools selected and used for realizing the game collection. The tools selected for the project were mainly based on the specifications set by the commissioners. However, some tools were selected by the development team due to having prior experience in their use and showing preference to said tools over their alternatives.

The use of certain tools began later. As the project continued to evolve, more tools were deemed necessary for the continuation of its development. Especially the revisions of the original concept required new tools for handle new

development environments. For example, the inclusion of the tutorials required a platform to build those tutorials on and the tools to realize them.

Unity and C#

Unity was selected as the game engine in conjunction with the development team and the commissioners, due to the team already being proficient in the use of said game engine. Thus, Unity provided a safe basis to build the project's contents with, requiring a minimal amount of additional learning on top of everything else that would be new.

C# is the scripting language used by the Unity game engine and its syntax doesn't differ much from that of C++, which is traditionally also taught to the new students, making it even more suitable for the project. In addition, Unity is freely available for use. Unreal Engine on the other hand, uses C++ but is mostly known for its visual scripting which is why it wasn't considered. The focus on traditional coding made Unity the better choice. (Epic Games 2019; Unity Technologies 2019.)

Doxygen

Doxygen is a tool used for generating documentation for code, from code. It was introduced into the project's scope to further increase the available documentation for students who would be using the game collection. Doxygen can create simplistic HTML pages from code that represents classes, methods and variables of various entities. (Doxygen 2019.)

Git and GitHub

Git is a lightweight version control system and GitHub is a platform used for accessing it. Git was included into the project to replace Unity Collaborate and because the use of Git was going to be a part of the same education scheme as the project's contents, so including it was deemed useful by the project's commissioners. (Git 2019.)

Visual Studio

Visual Studio is an IDE. It was used in conjunction with Unity to write C# code. Visual Studio was selected for its IntelliSense and because the development team had experience in using it. IntelliSense is a system that offers context sensitive guidance and autocompletion when writing code. IntelliSense also works with Unity's own libraries. (Microsoft 2019.)

Blender

Blender is an open source 3D modelling software. It was used in the project to create simple graphics for the featured games. Blender was selected to be used in the project due to the development team being proficient in its use. (Blender Foundation 2019.)

Brackets and HTML

Brackets is a text editor which specializes in web design (Brackets 2019). In the project, it was used for creating HTML documents that house the tutorials, using HTML which is the language for making web pages (W3Schools 2019). The team had used it briefly in a different project, so it was included into the scope at the same time as the tutorials were.

Paint.NET

Paint.NET is an image editing software that was used for making concept art and mockups for the project. It was selected because it was free to use, and the development team had used it before. Paint.NET features multiple useful tools and layer-based editing while still maintaining simplicity in its user interface, making it perfect for quick mock-ups. (Paint.NET 2019.)

3 FIRST-PERSON SHOOTER

This chapter details the creation process of the first-person shooter game created for the game collection. The game was not very complex because it would be used for teaching purposes and as such, there were little to no problems during its development.

In the implementation of the game, the focus was on creating clean and easily recyclable code and modular game objects, allowing the users to create their own FPS games using the knowledge gained from following the tutorial. As such, the focus will be on the code but some things inherent to Unity Engine will be detailed as well. The modular components that were repurposed for the project will also be briefly detailed in this chapter. The chapter will not go through all of them, only the ones used in the creation of the first-person shooter.

3.1 Modular components

Timer

Some systems rely on a Timer class object, which can be seen in full detail in Appendix 2. The Timer class is used for keeping track of time by calculating the time each frame took to complete. Counting time is done manually for each Timer instance, as is checking for its completion, and it was intended to be done each frame. As the Timer was inherently connected to frames per second, it could be inaccurate if performance suffered severely. In the scope of the games made for the collection, it was not a problem, however.

Resource

When the project was still intended to be fully modular, the development team found a modular data object for storing, modifying and comparing simple number values useful. But the most important part was having a clearly defined maximum and minimum values as a part of the data object itself, in addition to the current value. This was useful for handling finite things such as score, ammunition and health with a single variable and no programming knowledge. The Resource class is included in this document without commentary in Appendix 3.

Object Pooler

Object pooling is a programming technique that allows for more control over memory management. Instead of generating a new instance every time something needs to be created and then deleting it once it's no longer needed, an object pooler allocates a pool of objects that are recycled throughout the run of the program. Objects are hidden and deactivated instead of deleted outright

and they are reactivated once needed again. Spawning and despawning are used to describe this process. (Dickinson 2017, 294-297.)

Object pooling wasn't used as much as originally intended in the FPS game. In the end, it was only used for spawning and despawning bullet holes. Spawning targets uses a different system due to a lack of expertise at the time. Appendix 4 details the Object Pooler class used in the project.

3.2 Player and movement

To begin, a player character needed to be created. In this implementation, the player's functionality was split into two scripts. One for the general controls, which included movement and camera control, and another one for handling the shooting mechanics. Figure 4 details the final player object and its child hierarchy. The root object housed two cameras. One was for the player's view and the other one for rendering the player's gun over the first camera. It also housed the object used for detecting collisions using Unity's own colliders. As the camera was deeply tied to the controls, certain measures had to be taken when it came to the code.

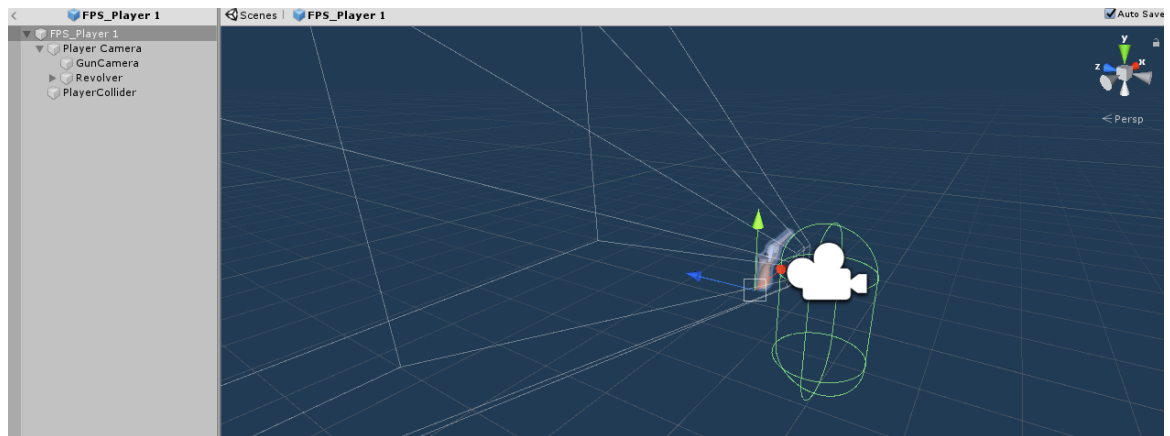


Figure 4. FPS player

Figure 5 details the movement logic used, a method named Move, which is called every physics frame. A movement vector was formed from Unity's base inputs as seen on line 83, meaning that moving the player could be achieved with the basic WASD controls, using the preset controls of the Unity engine. The vector in

question was rotated to match the y-axis rotation of the root object, ensuring that the movement vector would be relative to the rotation of the player character instead of the global axes of the game world; when the player turned, so would the direction of what is considered forward. The movement vector was used as the direction of the velocity that was applied to the player character, to simulate walking. Finally, when no input was detected, the player's velocity would decrease fast but not instantly.

```

74
75
76 private void Move() {
77
78     Quaternion yAngle = Quaternion.Euler(0, transform.rotation.eulerAngles.y, 0);
79
80     Vector3 movementVector = (yAngle * new Vector3(Input.GetAxisRaw("Horizontal"), 0, Input.GetAxisRaw("Vertical"))).normalized * 5;
81
82     if (movementVector.magnitude > 0) {
83         rb.velocity = new Vector3 (movementVector.x, rb.velocity.y, movementVector.z);
84     }
85     else {
86         rb.velocity = new Vector3(rb.velocity.x * 0.90f, rb.velocity.y, rb.velocity.z * 0.90f);
87     }
88 }
89
90
91
92
93
94

```

Figure 5. FPS player movement

In terms of the camera itself, up and down movement was accomplished by rotating the camera on its x-axis based on the mouse's corresponding movement axis as seen in Figure 6 line 70. Left and right movement didn't directly manipulate the camera at all, instead it was achieved by rotating the root object on line 72. The root object's child objects, including the camera, would then inherit the rotation and complete the synergy between HandleCamera and Move methods. By rotating the view, one would also rotate the root object. And by rotating the root object, the forward vector would change to match the camera's view.

```

66
67
68 private void HandleCamera() {
69
70     playerCamera.transform.Rotate(-Input.GetAxisRaw("Mouse Y") * cameraSpeed * Time.deltaTime, 0, 0);
71     transform.Rotate(0, Input.GetAxisRaw("Mouse X") * cameraSpeed * Time.deltaTime, 0);
72 }
73

```

Figure 6. FPS camera logic

Jumping and falling, as seen in Figure 7, were achieved with somewhat of a trick. Instead of using realistic physics and Unity's build-in physics engine's gravity, the player was not affected by gravity in the traditional sense. Instead, when the player was not touching the ground, they would always have a predetermined downwards velocity. Whether the player was grounded or not was tested by

projecting a short ray downwards from the center of the root object and testing for collisions against the ground.

```

95
97 private void Jump() {
99     if(Input.GetButtonDown("SpaceBar") && IsGrounded()) {
100         rb.AddForce(Vector3.up*10, ForceMode.VelocityChange);
101     }
102 }
103
106 private bool IsGrounded() {
108     return Physics.Raycast(transform.position + Vector3.up, Vector3.down, 1.1f, (1 << 10));
109 }
110
112 private void ApplyGravity() {
114     if (!IsGrounded())
115     {
116         rb.AddForce(Vector3.down, ForceMode.VelocityChange);
117     }
118 }
119 }

```

Figure 7. Jumping and falling

In addition, certain things were needed for rendering the player's gun over everything else. First, the gun object needed to be put into a separate layer. Secondly, the depth value of the gun only camera needed to be set to depth only. And thirdly, the culling mask needed to be set up so that the layers were inversed, which means that the gun camera only had the layer where the gun is located at and the player camera had everything but that layer, as seen in Figure 8. (Ivkoni 2010.)

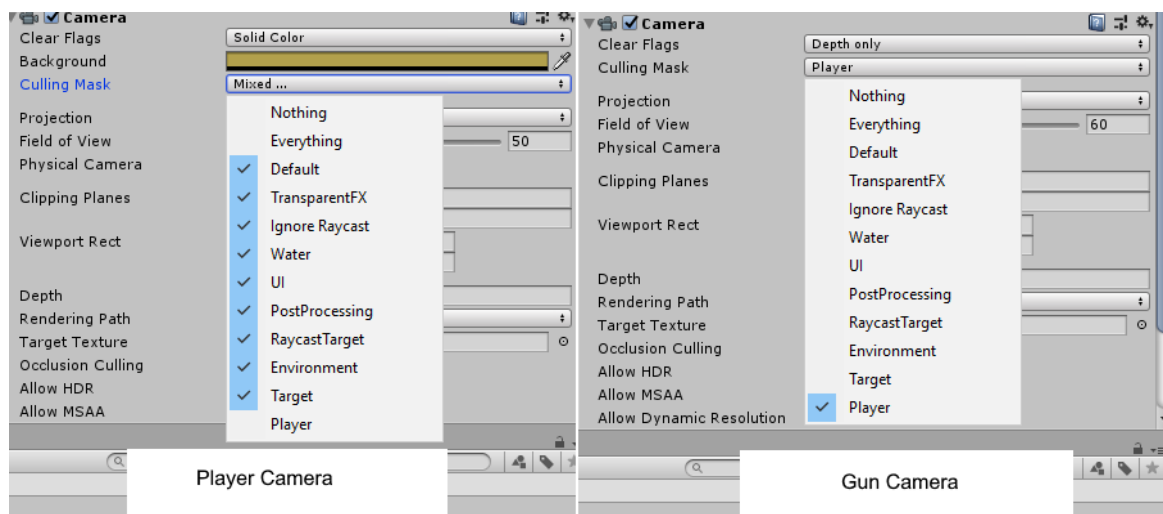


Figure 8. FPS dual camera setup

3.3 Shooting mechanics

The basis for the shooting mechanics were accomplished with a simple state machine seen in Figure 9. While in the ready state, the algorithm tested for appropriate input for firing the gun and whether the player had ammunition left. If the player didn't have full ammunition and initiated a reload with the correct input, the state shifted to reloading. Once reloading was done, the state shifted back to ready. In addition, a failsafe was put in place if the state value was changed incorrectly, to prevent accidental hang-ups if a user following the tutorial did a mistake.

```

58  ▶
60  ▶ switch(weaponState) {
61  ▶     case WeaponState.ready:
63  ▶         Shoot();
64  ▶         break;
65  ▶     case WeaponState.reload:
67  ▶         Reload();
68  ▶         break;
69  ▶     default:
71  ▶         Debug.LogError("PPP Error: Mismatched weapon state.");
72  ▶         break;
73  ▶ }

```

Figure 9. Shooting states

Figure 10 details the entire shooting algorithm. The algorithm checked for input and Resource class instance ammo's value, as seen on line 94. A Timer class instance called firerate also kept shooting limited to a set number of times per second. Following that, the logic would split either to shooting or initiating a reload. For shooting, the algorithm would first initiate the sound and visual effects and then continue to test for collisions with a ray cast on line 109. Hitting the environment would create a bullet hole as shown on line 123 from an object pooler named bulletHolePooler that kept track of bullet hole instances. Hitting a target would increase the player's score by an amount taken from the target instance, which was then stored in the game manager on line 115. Finally, the target's destruction was begun on line 118. If the player's ammo value was less than the maximum and the player gave the correct input, an animation and a sound effect would play as the state was changed to reloading.

```

90
92 private void Shoot() {
94     if (Input.GetButton("Fire1") && firerate.IsFinished() && ammo.GetValue() != 0)
95     {
96         gunAnimator.SetTrigger("shoot");
97         gameManager.GetAudioManager().PlaySound(shootSFX);
98
100         RaycastHit impactPoint;
102         firerate.Reset();
104         DrawBullet();
106         ammo.SetValue(ammo.GetValue() - 1);
107
109         if (Physics.Raycast(transform.position, transform.forward, out impactPoint, Mathf.Infinity, ((1 << 10) | (1 << 11))))
110         {
112             if (impactPoint.collider.CompareTag("Hazard"))
113             {
115                 gameManager.IncreaseGameScore(impactPoint.collider.GetComponent<FPS_Target>().GetPoints());
116
118                 impactPoint.collider.GetComponent<FPS_Target>().DeathSequence();
119             }
121             else
122             {
123                 bulletHolePooler.Activate(impactPoint.point, Quaternion.LookRotation(impactPoint.normal));
124             }
125         }
126     }
128     else
129     {
130         firerate.Count();
131     }
132
134     if (ammo.GetValue() != ammo.GetMax() && Input.GetKeyDown(KeyCode.R)) {
136         weaponState = WeaponState.reload;
137         gunAnimator.SetTrigger("reload");
138         gameManager.GetAudioManager().PlaySound(reloadSFX);
139     }
140 }

```

Figure 10. Shooting algorithm

The reloading algorithm simply counted time with a Timer class instance reloadDuration as seen in Figure 11 line 156. Once the Timer had reached its maximum value, the state was set back to ready on line 148 and the player's ammo was set back to its maximum value, stored in the class instance itself, on line 151. The fire rate timer was also reset, to allow for continuous shooting, even after reloading, on line 150.

```

141
143 private void Reload() {
145     if (reloadDuration.IsFinished())
146     {
148         weaponState = WeaponState.ready;
149         reloadDuration.Reset();
150         firerate.SetCurrent(firerate.GetMax());
151         ammo.SetValue(ammo.GetMax());
152     }
154     else
155     {
156         reloadDuration.Count();
157     }
158 }

```

Figure 11. Reloading algorithm

3.4 Targets and spawning

As explained in the design portion in chapter 2.3. the targets came in two types, simplified here as Line and Pop, linear and pop out or peeking movement respectively. Line type would travel from point A to point B while Pop type would go from A to B and back to A. Figure 12 details both movement types as they were coded. Both featured the same beginning logic, where the distance between the starting position and the current position were compared to the distance between the starting position and the ending position, shown on lines 95 and 118. If the distance was not the same, it meant that the end position had not been reached yet and as such, the target would be moved towards it in proportion to its speed value and the time difference between the current and previous frame, `Time.deltaTime`, as shown on lines 98 and 121. Because the target's movement was completely fixed and only ever moved towards the second point, distance could be used as an accurate measure of position on the line from A to B in this context. If the distance between the beginning and the current point was ever larger than the distance between the beginning and ending point, the goal could be deemed reached.

Once the end point was reached, the logic split, as seen on lines 101 and 124 of Figure 12. In Line type targets using linear movement, the object was simply marked for destruction and begun its death sequence. For the Pop type targets, once the end point was reached, the starting and ending positions were switched around, as shown on lines 133 to 136, and the `toBeDestroyed` boolean set to true on line 139, ready for when the end position was reached for the second time and the death sequence could be initiated.

To further emphasis the logic splitting for the beginners, the code here was formatted to look as similar as possible in both instances even though it's generally considered to be wasteful to repeat code as it can lead to problems such as having to fix the same error in multiple places (Goodliffe 2006, 251-252). It was, however, not an issue in this implementation.

```

91
93 private void LinearMovement() {
95     if (Vector3.Distance(posZero, transform.position) < (Vector3.Distance(posZero, posFinal)))
96     {
98         transform.position = transform.position + (posFinal - posZero).normalized * speed * Time.deltaTime;
99     }
101     else
102     {
104         toBeDestroyed = true;
105     }
106
107     if (toBeDestroyed)
108     {
110         DeathSequence();
111     }
112 }
113
114
116 private void PopOut() {
118     if (Vector3.Distance(posZero, transform.position) < (Vector3.Distance(posZero, posFinal)))
119     {
121         transform.position = transform.position + (posFinal - posZero).normalized * speed * Time.deltaTime;
122     }
124     else
125     {
127         if (toBeDestroyed)
128         {
129             DeathSequence();
130         }
131
133         Vector3 temp = new Vector3();
134         temp = posZero;
135         posZero = posFinal;
136         posFinal = temp;
137
139         toBeDestroyed = true;
140     }
141 }

```

Figure 12. Targets' movement logic

Initializing the targets was done by the spawner. Figure 13 details the initialization method that a spawner would use when creating a target. First, it would set the type and both starting and ending positions based on the spawner itself as seen on lines 75 to 77. Then it would move into a binary state machine, where the speed of the target would be calculated based on distances, type of movement and the delay between spawned objects. In short, the speed of a peeking target would be two times the distance between the start and ending positions in proportion to the spawning delay, meaning that the targets would seamlessly continue peeking one after another. For linear targets, the speed was balanced around moving from start to end in four seconds if the delay was one second and then scale further based on the delay.


```

63
74 public void Initialize(int t, Vector3 start, Vector3 end, float delay) {
75     type = t;
76     posZero = start;
77     posFinal = end;
78
81     if(t == 1) {
83         speed = (Vector3.Distance(start, end) * 2) / delay;
84     }
86     else {
88         speed = (Vector3.Distance(start, end) / 4) / delay;
89     }
90 }

```

Figure 13. Setting up targets

When a target was hit, the player would start the death sequence of a target prematurely, as seen in chapter 3.3. The death sequence itself did nothing more than cause the target to play an animation and begin a countdown to hiding the target object in the game world. The collider was disabled before the countdown even began to prevent multiple method calls during the countdown.

For the spawning, a straightforward system was devised to handle how the targets are created and kept track by the game. A script called FPS_Spawner held references to target instances and instantiated them after a set period called SpawnDelay in set intervals called SpawnRate. The spawner knew the starting and ending locations of the movement which were derived from its two child objects marked as Start and End, which were kept track of by the script. The spawner was set as either Pop or Line, so that the spawned targets knew which movement scheme to use. Figure 14 showcases how the spawner looks inside of the editor as a script component. The Spawned Objects list, along with the starting and ending positions, would be populated by drag and dropping elements into the list in the editor view.

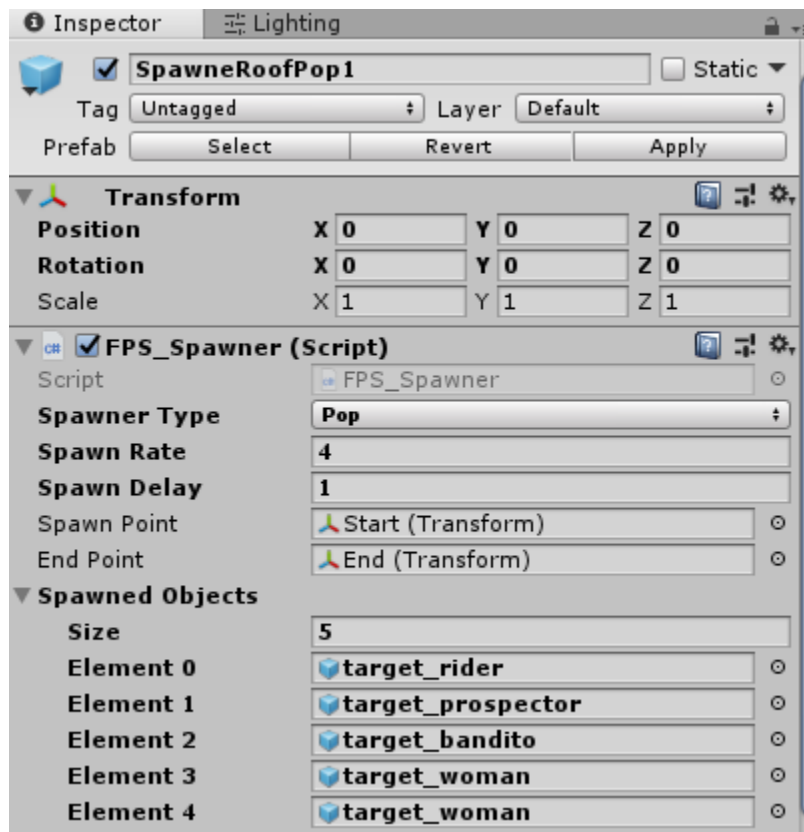


Figure 14. FPS spawner in the editor

Finally, an object was made to keep the player from progressing, until all targets were either defeated or they had run their course. The Encounter Blocker, as shown in Figure 15, consisted of an invisible wall that prevented the player from moving. The blocker was given a reference to the spawner that was timed to be the last one in a sequence, as decided by the user. Because the spawner's algorithm worked by emptying a list, as targets were spawned from it, the blocker only had to check whether it was empty or not. The blocker would also reveal a hidden arrow for a few seconds, to show the player where to go next.

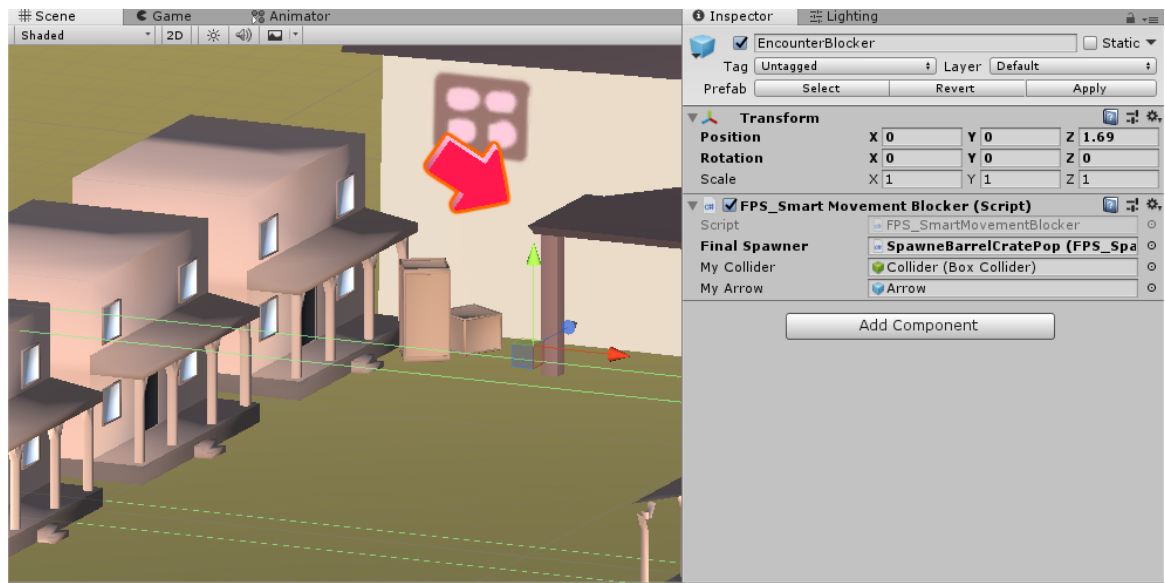


Figure 15. The Encounter Blocker

3.5 Score and HUD

The UI was managed by the game manager and the UI manager. Game and player specific information such as the player's ammo, score from destroyed targets and such was given to the game manager. The UI manager in turn, would update its values based on the game manager. In chapter 3.3, an example of this was already shown. Figure 16 shows the specific lines from the shooting logic, where the game manager was told to increase the score value, based on the target instance's point value, and to update the player's ammo count.

```

76 | gameManager.SetPlayerAmmo(ammo.GetMin(), ammo.GetMax(), ammo.GetValue());
115 | gameManager.IncreaseGameScore(impactPoint.collider.GetComponent<FPS_Target>().GetPoints());

```

Figure 16. Updating the values

The values stored in the game manager were then used to update the UI components, that displayed the values. Figure 17 shows the Update method of the FPS_UIManager script. The values were updated every frame. The text was also formatted, more importantly the score which was limited to four character as seen on line 28, so the value would go from 0000 to 9999.

```

26 | private void Update () {
28 |     scoreText.text = string.Format("Score: {0:0000}", gameManager.GetGameScore());
29 |     ammoText.text = string.Format("Ammo: {0:0}/{1:0}", gameManager.GetPlayerAmmo().GetValue(), gameManager.GetPlayerAmmo().GetMax());
30 | }

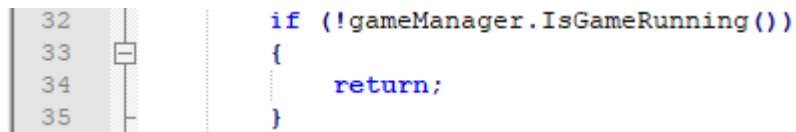
```

Figure 17. Updating the HUD

3.6 Finishing touches

For ending the game session, not much was required. In the intended end point of the game world, as explained in chapter 2.3, an object was created that would tell the game manager to initiate the ending sequence of the game. Once the player entered its radius, the game's state would change, and the ending screen would be displayed.

To prevent unintended behavior, the player's controls, which included shooting and moving, were disabled after the game ended. Figure 18 shows a short snippet of code, which was inserted in the beginning of all player control related logic. If the game had ended, the forced return prevented the following code from being ran.



```
32  
33  
34  
35  
if (!gameManager.IsGameRunning())  
{  
    return;  
}
```

Figure 18. Checking the game state in player controller

3.7 Graphics

For the graphics of the game, rudimentary 3D assets were made in Blender. The theme of the game was decided on wild west and the assets reflected that. Saloons and frontier houses served as the backdrop for the action and cowboys and bandits as the targets to shoot. The models were made with performance in mind, featuring a low amount of detail.

Figure 19 shows the different characters created for the game, used to represent the targets. During the design process, the concept for the targets' graphics was more literal, as discussed and previewed in chapter 2.3. But as the development continued it was decided that each target should have its own unique model instead of the targets simply having a different image painted on them. This would increase readability, as each model would have its own silhouette, color scheme and animations. In Figure 19, the upper row features the targets that give points while the bottom row features those that remove them and are not meant

to be shot at. Targets that are meant to be shot are clearly armed, one with a gun and the other one with dynamite, while the rest are civilians, with one carrying a pickaxe and the other one a baby.



Figure 19. Collage of pictures of the target models

As for the game world itself, its appearance was inspired by various wild west themed movies, games and images. Figure 20 is a collage of various parts of the game world taken in Unity with the fog disabled. Static models created in Blender populated the scene to give it some depth and the appearance of a town in the wild west. Most models were reused throughout the level, such as the houses and the crates.



Figure 20. Collage of screenshots from the FPS

4 DOCUMENTATION

This chapter briefly details the documentation made for the project from a development perspective. As explained in chapter 2, the game collection required heavy documentation due to its contents being used for educational purposes. Each game created for the collection required extensive commentary of the code, along with a tutorial piece on how to recreate the game. The first-person shooter's documentation will be used as an example throughout this chapter and it will go through the creation process of the tutorial created for the FPS game.

The development team created several guidelines and conventions regarding the documentation, both for the layout and flow of the tutorials along with the commenting. This was deemed important to keep large amounts of text organized and readable from a glance.

4.1 Doxygen

As explained in chapter 2.4, Doxygen generates documentation from code. Introducing Doxygen to the project was simple. Once installed, Doxygen could be used with the command line, utilizing its own commands separate from those of the operating system. A configuration file was created either manually or automatically in the source directory, which would control the generation process, such as the input and output directories and what format the generated documents would be in.

In the code itself, Doxygen's document generation was controlled with commands that the IDE would ignore but Doxygen could read. Appendix 5 shows the final commentary guidelines made for the project, featuring Doxygen specific commands in green. Figure 21 shows various parts of the final Doxygen document. Class overview details the script used for handling the FPS player's shooting logic, which was described in chapter 3.3.

The figure displays three panels of Doxygen-generated documentation for a project:

- 1. Class list:** A table listing classes and their descriptions. Examples include:
 - `FPS_EndGameFlag`: First Person Shooter -pelin lopetusalue luokka
 - `FPS_GameManager`: First person Shooter -pelin pelimanageri
 - `FPS_PlayerController`: First Person Shooter -pelin pelaajahalmon ohjaajaluokka
 - `FPS_PlayerShoot`: First Person Shooter -pelin ampuksiluokka
 - `FPS_SmartMovementBlocker`: First Person Shooter -pelin liikkumisen estäjäluokka
 - `FPS_Spawner`: First Person Shooter -pelin peliohjettujen luontoluokka
 - `FPS_Target`: First Person Shooter -pelin maalialueiluokka
 - `FPS_UIManager`: First Person Shooter -pelin UI-manageri
 - `GameObjectGroup`: Peliohjettujen ryhmä-luokka
 - `Lifetime`: Objektin elinikä
 - `Line`: Matemattinen jana
 - `MenuItem`: ...
 - `Chest`: ...
 - `ObjectPooler`: Pakallinen peliohjettujen luokka, joka säilyttää ja pitää kirjaa peliohjetteista
 - `PauseMenu`: ...
 - `Projectile`: Projektililuokka
 - `RCG_CameraController`: Racing-pelin kameran ohjainluokka
 - `RCG_CarController`: Autoiluokka
 - `AxleInfo`: Dataa luokka auton akselille
 - `RCG_Checkpoint`: Valokappale-luokka
 - `RCG_GameManager`: Racing-pelin pelimanageriluokka
 - `RCG_Player`: Racing-pelin pelaajaluokka
 - `RCG_UIManager`: Racing-pelin käyttöliittymän manageri
 - `Resource`: Resurssi-luokka
 - `RUN_Collectable`: Endless Runner -pelin kerättävä esine -luokka
 - `RUN_GameManager`: Endless Runner -pelin pelimanageri
 - `RUN_Obstacle`: Endless Runner -pelin esteiluokka
 - `RUN_Player`: Endless Runner -pelin pelaajaluokka
 - `RUN_UIManager`: Endless Runner -pelin graafisen käyttöliittymän manageri
 - `RUN_WorldFile`: Endless Runner -pelin kentän pitäjän
 - `Spin`: Objektin pyörimisnopeudesta vastaava komponentti
 - `STG_Emitter`: Lapsiluokka Emitter-vihollistyyppille
 - `STG_Enemy`: Isäntäluokka eri vihollistyyppille Shoot Em Up -pelissä
 - `STG_GameManager`: Pelimanageri Shoot Em Up -pelin
 - `STG_PlayerController`: Pelaajan ohjaajaluokka
 - `STG_PlayerHealth`: Pelaajan elämästä vastaava luokka
 - `STG_Projectile`: Projektililuokka Shoot Em Up -pelin varten
- 2. Class overview:** A detailed view of the `FPS_GameManager` class, showing:
 - enum WeaponState (ready, reloading)**
 - Yksityiset jäsenfunktiot:** `void Start ()`, `void Update ()`, `void Shoot ()`, `void Reload ()`, `void DrawBullet ()`
 - Yksityiset attribuutit:** `GameObject muzzleFlash`, `GameObject bulletHole`, `Animator gunAnimator`, `AudioClip shootSFX`, `AudioClip reloadSFX`, `Timer fireRate`, `Timer effectDuration`, `Timer reloadDuration`, `WeaponState weaponState`, `Resource ammo`, `ObjectPooler bulletHolePooler`
 - FPS_GameManager gameManager**
- 2. Class overview cont.:** Continuation of the class overview, showing:
 - Yksityiset tyytit:** `enum WeaponState (ready, reloading)`
 - Yksityiset jäsenfunktiot:** `void Start ()`, `void Update ()`, `void Shoot ()`, `void Reload ()`, `void DrawBullet ()`
 - Yksityiset attribuutit:** `GameObject muzzleFlash`, `GameObject bulletHole`, `Animator gunAnimator`, `AudioClip shootSFX`, `AudioClip reloadSFX`, `Timer fireRate`, `Timer effectDuration`, `Timer reloadDuration`, `WeaponState weaponState`, `Resource ammo`, `ObjectPooler bulletHolePooler`
 - FPS_GameManager gameManager**

Figure 21. Doxygen collage

4.2 Tutorials

The tutorials were introduced into the scope of the project at a very late stage. Initially, they were to be Word documents but to keep students from simply copying code from the tutorial, the development team decided that all code was

to be presented as pictures. Word does not support pop-up images, so the usage of large images clashed with the development platform. As such, HTML was chosen as the platform, since it would also allow for easy hosting of the tutorials on GameLab's website and the database generated by Doxygen could also be integrated to it with a link.

A template for the HTML based tutorials was made by the development team to suit the needs of the project. Figure 22 shows an edited version of the original template, made specifically for this document due to the original being written in Finnish. All the pages of the tutorial website shared a header, which included links to the front page and the Doxygen document, among other things. The front page had links to all the different tutorials.

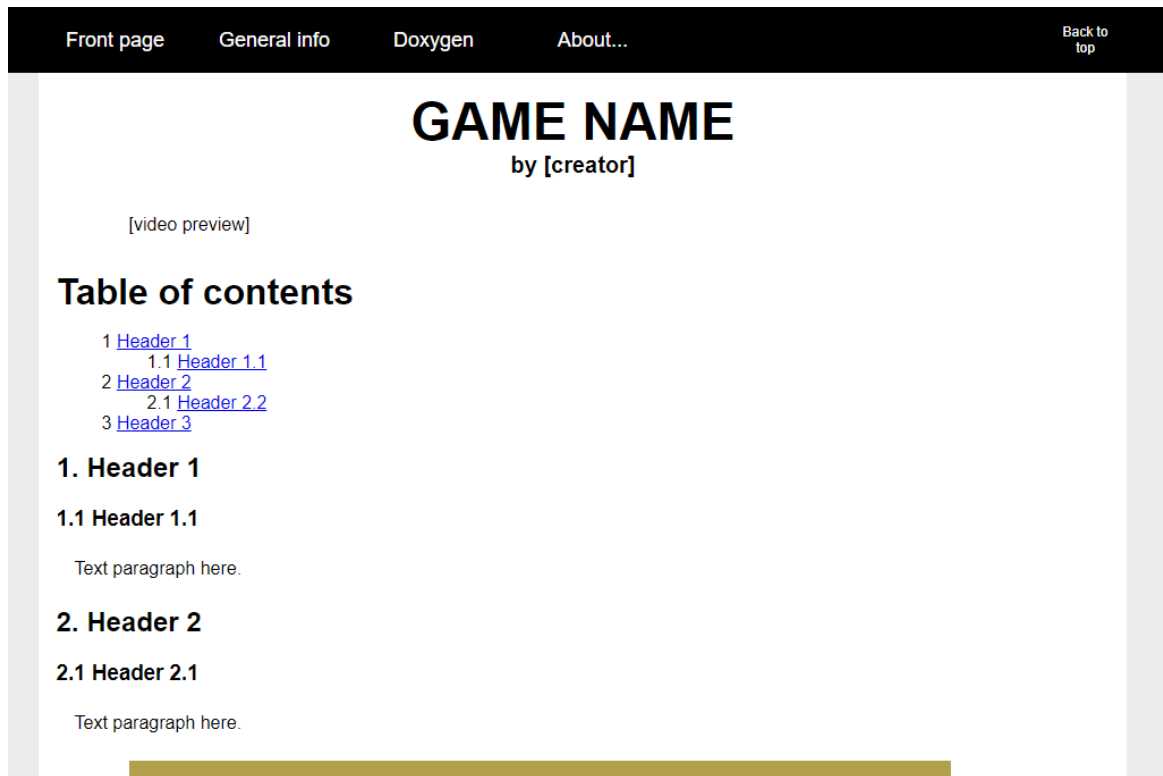


Figure 22. Tutorial template

As for the content of the tutorials, information was presented in logical order. Things that were required for something else to work were done beforehand and jumping between different scripts and game objects was avoided. Most of the time, one thing was finished before moving on to something else. Below is a

translated excerpt from the first-person shooter tutorial, which goes through the creation of the player object. Names used by objects, concepts and variables were bolded to signify their unique nature. Before something was done which could be misunderstood, it was explained further between the steps.

First, we navigate to *Assets/Resources/Models/First Person Shooter* folder in the Project window. Drag the **Revolver4** object into the Scene.

We create a new empty game object into the hierarchy and rename it to **FPS_Player**. Drag the **Main Camera** under this object, making it its child. Next, we drag the revolver to be the camera's child. Rename the revolver object to **Revolver** and **Main Camera** to **Player Camera**.

We create another empty game object and rename it to **Player Collider**. Give it the **Capsule Collider** component.

We need one more camera. Create a new **Camera** object and rename it to **Gun Camera**. Set it as the **Player Camera's** child. The **Gun Camera** is going to render only the gun. The gun is rendered on top of everything else, as is customary in FPS games.

As explained previously, all code shown was presented in pictures. This was done to prevent copy pasting code and skipping steps, which would not only ruin the learning experience but could cause confusion due to missed steps. In the HTML tutorials, images could be clicked on, which would expand the picture and show its caption.

5 CONCLUSIONS AND FUTURE DEVELOPMENT

As of writing this thesis, the development is finished, and the product is in use. When first presented to the commissioners the reception was mostly positive, but some additional points needed to be addressed. A video preview of each game in action was commissioned, along adding required license information to the HTML document. The video preview was to be integrated to the tutorials themselves. The games themselves had no issues.

For the users it was a different matter. Certain tutorials proved to be too difficult and some detail, which felt obvious at the time of writing the tutorial, was missing

from some. There was also an issue with the images in the tutorials which stemmed from foreign characters in the file names.

Overall, the success of the product is yet to be determined and more data is required to come up with a definite conclusion but outside of these issues, nothing else has come up. The games and the tutorials have served their purpose as supplementary material for teaching game development. The level of success is still unknown, but it can be considered a success nonetheless.

For future development, once the issues are fixed, the pool of games and tutorials could be expanded easily. With the foundation of guidelines, templates and working methods already done, expanding would be the next logical step. The collection lacks representation from various popular genres and none of the games feature multiplayer. These could be approached in the future with a new collection of games and tutorials.

REFERENCES

Blender Foundation. 2019. WWW document. Available at:

<https://www.blender.org/> [Accessed 15 July 2019]

Brackets. 2019. WWW document. Available at: <http://brackets.io/> [Accessed 24 September 2019]

Dickinson, C. 2017. Unity 2017 Game Optimization - Second Edition. Ebook.

Birmingham & Mumbai: Packt Publishing. Available at:

<https://ebookcentral.proquest.com/lib/xamk-ebooks/detail.action?docID=5160904>

[Accessed 7 October 2019]

Doxygen. 2019. About. WWW document. Available at:

<http://www.doxygen.nl/index.html> [Accessed 15 July 2019]

Driver, R. 1994. The fallacy of induction in science teaching. In Levinson, R. (ed.)

Teaching Science. Ebook. 1994. London: Routledge. Available at:

<https://ebookcentral.proquest.com/lib/xamk-ebooks/detail.action?docID=237263>

[Accessed 19 August 2019]

Epic Games. 2019. Unreal Engine 4 Documentation. WWW document. Available

at: <https://docs.unrealengine.com/en-US/index.html> [Accessed 26 September 2019]

Git. 2019. WWW document. Available at: <https://git-scm.com/> [Accessed 15 July

2019]

Goodliffe, P. 2006. Code craft: The practice of writing excellent code. Ebook. San Francisco: No Starch Press. Available at:

<https://ebookcentral.proquest.com/lib/xamk-ebooks/detail.action?docID=273481>

[Accessed 27 August 2019]

Ivkoni. 2010. Forum post on the official Unity forum. Available at:

<https://forum.unity.com/threads/rendering-player-and-gun-in-front-of-other-objects.64663/#post-413097> [Accessed 22 August 2019]

Jung, T. 2018. How Do Bullets Work in Video Games? Article. Available at:

<https://medium.com/@3stan/how-do-bullets-work-in-video-games-d153f1e496a8>
[Accessed 19 August 2019]

Koponen, J. 2019. Documentation of modular game collection. Bachelor's thesis.

Available at: <https://www.theseus.fi/handle/10024/169889> [Accessed 5 November 2019]

Microsoft. 2019. Visual Studio 2019. WWW document. Available at:

<https://visualstudio.microsoft.com/vs/> [Accessed 24 September 2019]

Mitchell, B. L. 2012. Game Design Essentials. Ebook. 2012. Hoboken: Sybex.

Available at: <https://ebookcentral.proquest.com/lib/xamk-ebooks/detail.action?docID=818112> [Accessed 2 September 2019]

Otero, C. Software Engineering Design: Theory and Practice. Ebook. 2012.

Auerbach Publications. Available at: <https://ebookcentral.proquest.com/lib/xamk-ebooks/detail.action?docID=1580108> [Accessed 5 November 2019]

Paint.NET. 2019. WWW Document. Available at: <https://www.getpaint.net/>

[Accessed 24 September 2019]

Unity Technologies. 2019. Unity 2019: Performance by default, high-fidelity real-time graphics, and artist tools. WWW document. Available at:

<https://unity3d.com/unity> [Accessed 26 September 2019]

W3Schools. 2019. HTML Introduction. WWW document. Available at:

https://www.w3schools.com/html/html_intro.asp [Accessed 25 September 2019]

LIST OF FIGURES

Figure 1. Proposed Action-class.....	10
Figure 2. Mockup of the FPS game	13
Figure 3. FPS level layout	14
Figure 4. FPS player.....	18
Figure 5. FPS player movement.....	19
Figure 6. FPS camera logic	19
Figure 7. Jumping and falling	20
Figure 8. FPS dual camera setup.....	20
Figure 9. Shooting states.....	21
Figure 10. Shooting algorithm	22
Figure 11. Reloading algorithm	22
Figure 12. Targets' movement logic	24
Figure 13. Setting up targets	25
Figure 14. FPS spawner in the editor	26
Figure 15. The Encounter Blocker.....	27
Figure 16. Updating the values.....	27
Figure 17. Updating the HUD	27
Figure 18. Checking the game state in player controller	28
Figure 19. Collage of pictures of the target models	29
Figure 20. Collage of screenshots from the FPS.....	30
Figure 21. Doxygen collage.....	31
Figure 22. Tutorial template.....	32

Appendix 1

GAMES/ELEMENTS	PLATFORMER (PFR)	TOP-DOWN TWIN-STICK SHOOTER (TTS)	FPS (FPS)	ENDLESS RUNNER (RUN)	2D BATTLE ARENA (BAR)	RACING GAME (RCG)	REAL TIME STRATEGY (RTS)	TOWER DEFENSE (TWD)	SURVIVAL HORROR (HOR)	SHOOT 'EM UP (STG)
CHARACTERS										
Player	Y	Y	Y	Y	Y	Y	Y	N	Y	Y
NPC	Y	Y	Y	N	?	Y	Y	Y	Y	Y
OBJECTS										
Spawner	Y	Y	Y	Y	Y	Y	Y	Y	?	Y
Projectile	Y	Y	Y	N	Y	Y	Y	Y	N	Y
Collectible	Y	?	Y	Y	Y	Y	N	N	Y	N
Props (physics)	N	N	Y	N	N	Y	N	N	N	N
MOVEMENT TYPES										
Move on 2 axes	Y	Y	Y	N	N	N	Y	Y	Y	Y
Move on 1 axis	N	N	Y	Y	Y	Y	N	N	N	N
Rotation control	N	Y	Y	N	Y	Y	N	N	Y	N
Waypoints	Y	N	N	N	N	Y	Y	Y	N	N
Follow	N	Y	N	N	N	N	?	N	Y	N
CAMERA TYPES										
Static	N	Y	N	Y	Y	N	N	Y	N	Y
Follow player	Y	N	N	N	N	N	Y	N	N	N
Rotation camera (3rd person)	N	N	N	N	N	Y	N	N	N	N
Rotation (1st person)	N	N	Y	N	N	N	N	N	Y	N
ACTIONS										
Jump	Y	N	N	?	Y	N	N	N	?	N
Look at target	N	Y	Y	N	N	Y	Y	Y	Y	N
Attack	Y	Y	Y	N	Y	Y	Y	Y	Y	Y
SYSTEMS										
Health & Damage	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Hitbox/Hurtbox	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Spawn/Respawn/Despawn	Y	Y	Y	Y	Y	Y	Y	Y	?	Y
Resource/Currency	?	Y	Y	Y	Y	Y	Y	Y	Y	Y
Timer	N	Y	Y	Y	Y	Y	Y	Y	?	Y
UI (resource, button)	?	Y	Y	Y	Y	Y	Y	Y	N	Y
Animations	Y	N	Y	?	Y	Y	Y	N	Y	?
Pathfinding	N	N	N	N	?	Y	Y	N	Y	?

Made by Juho Koponen, Sami Krouvi & Jari Tervo

Appendix 2

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 /// @brief Ajastin-luokka.
6 /// @details Ajastin laskee nolasta maksimiarvoon. Voidaan käyttää apuna tapahtumien ajoittamisessa.
7 [System.Serializable]
8 public class Timer
9 {
10     [SerializeField] private float max;
11     /// @details Ajastimen tämänhetkinen arvo.
12     private float current;
13
14     public float GetMax() { return max; }
15     public void SetMax(float m) { max = m; }
16     public float GetCurrent() { return current; }
17     public void SetCurrent(float c) { current = c; }
18
19
20
21     /// @details Ajastin-luokan konstruktori.
22     public Timer(float m, float c)
23     {
24         this.max = m;
25         this.current = c;
26     }
27
28     /// @details Laskee aikaa. Tarkoitettu kutsuttavaksi per frame.
29     public void Count()
30     {
31         if (current <= max)
32         {
33             current = current + Time.deltaTime;
34         }
35     }
36
37     /// @details Kertoo, onko ajastin saavuttanut maksimiarvon.
38     /// @return Palauttaa true, kun ajastin on saavuttanut maksimiarvon.
39     public bool IsFinished()
40     {
41         if (current >= max)
42         {
43             return true;
44         }
45         else
46         {
47             return false;
48         }
49     }
50
51     /// @details Nollaa ajastimen.
52     public void Reset()
53     {
54         current = 0;
55     }
56 }
57
```

Made by Juho Koponen, Sami Krouvi & Jari Tervo

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  [System.Serializable]
6  public class Resource
7  {
8      [SerializeField] private float min;
9      [SerializeField] private float max;
10     private float value;
11
12     public float GetMax() { return max; }
13     public float GetMin() { return min; }
14     public float GetValue() { return value; }
15     public void SetMax(float v) { max = v; }
16     public void SetMin(float v) { min = v; }
17     public void SetValue(float v)
18     {
19         if (v < min)
20         {
21             value = min;
22         }
23         else if (v > max)
24         {
25             value = max;
26         }
27         else
28         {
29             value = v;
30         }
31     }
32
33     public Resource(float min, float max)
34     {
35         this.min = min;
36         this.max = max;
37         this.value = min;
38     }
39
40     public void Modify(float a)
41     {
42         value = value + a;
43
44         SetValue(value);
45     }
46 }
```

Made by Juho Koponen, Sami Krouvi & Jari Tervo


```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
7  public class ObjectPooler : MonoBehaviour
8  {
10 [SerializeField] private GameObject pooledObject;
12 [SerializeField] private int pooledAmount;
14 [SerializeField] private bool canExpand;
19 private List<GameObject> pooledObjects;
20
21 private void Start ()
22 {
24     pooledObjects = new List<GameObject>();
25     for (int i = 0; i < pooledAmount; i++)
26     {
27         GameObject obj = (GameObject)Instantiate(pooledObject);
28         obj.SetActive(false);
29         pooledObjects.Add(obj);
30     }
31 }
32
40 public GameObject GetPooledObject()
41 {
42     for (int i = 0; i < pooledObjects.Count; i++)
43     {
44
45         if (!pooledObjects[i].activeInHierarchy)
46         {
47             return pooledObjects[i];
48         }
49     }
51     if (canExpand)
52     {
53         GameObject obj = (GameObject)Instantiate(pooledObject);
54         pooledObjects.Add(obj);
55
56         Debug.Log(this.gameObject.name + " pool size: " + pooledObjects.Count);
57         return obj;
58     }
59
60     Debug.LogError("PPP Error: All objects in this pool are already in use!");
61     return null;
62 }
63
71 public List<GameObject> GetPooledObjects()
72 {
73     return pooledObjects;
74 }
75
85 public void Activate(Vector3 pos, Quaternion rot)
86 {
87     GameObject obj = GetPooledObject();
88     obj.transform.position = pos;
89     obj.transform.rotation = rot;
90     obj.SetActive(true);
91 }
92
101 public void Activate(GameObject spawner)
102 {
103     GameObject obj = GetPooledObject();
104     obj.transform.position = spawner.transform.position;
105     obj.transform.rotation = spawner.transform.rotation;
106     obj.SetActive(true);
107 }
108 }
109

```

PELIPALAPELI

Commenting Guidelines

```
/// @brief Description
/// @details Detailed description
class Example
{
    /// @details Description
    int integer1;

    /// @details Description
    struct Structure
    {
        int integer2; ///< @details Description
        int integer3; ///< @details Description
    }

    /// @details Description
    /// @n Previous continues on a new line
    /// @param i description
    /// @return Returnee's description
    /// @cond DEV
    /// <summary> (is not included in Doxygen documentation)
    /// <para> Short description (tooltip). </para>
    /// </summary>
    /// @endcond
    int Method(int i)
    {
        (do something)
        return i;
    }
}
```