Co-op split screen game with Unity 3d and Cinemachine plugin

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Bachelor's thesis

HAMK university of applied sciences
Bachelor of Business IT

2021

Pasi Saarikalle

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

ABSTRACT

HAMK university of applied sciences
HAMK university of applied sciences

Author Pasi Saarikalle **Year** 2021

Title Co-op split screen game with Unity 3d and Cinemachine plugin

Supervisor(s) Lasse Seppänen

TIIVISTELMÄ

Opinnäytetyön tavoitteena oli tehdä niin sanottu jaetun näytön moninpeli, käyttäen pohjana Unity3d-pelimoottoria. Pelissä käytetään kameraominaisuuksissa vahvasti Cinemachine-nimistä lisäosaa Unitylle. Opinnäytetyön pääpainopiste oli jaetun näytön moninpelin kehityksessä, mutta siinä käytiin läpi myös muita pelin toiminnalisuuksia, kuten tekoälyn peruspiirteitä tässä projektissa.

Teoriaosuudessa käytiin hieman läpi pelimoottoreiden historiaa. Opinnäytetyössä käytiin myös läpi itse Unity3d-pelimoottorin historiaa, sekä sen ominaisuuksia. Teoriaosuudessa on myös kirjoitettu pelissä vahvasti läsnä olevasta lisäosasta, Cinemachine:stä. Myös pelissä käytettyjä ohjelmointikieliä on avattu hieman.

Käytännönosuudessa kuvataan tämän projektiryhmän lähestymistapa pelin suunnitteluun ja projektin hallintaan. Käytännönosuudessa on myös käyty läpi pelin piirteitä, ja pelin kulku pelaajan näkökulmasta. Teoriaosuudessa kerrotaan myös läpi pääpiirteittäin pelin keskeisimmistä mekaniikoista. Käytännön osuus painottuu kuitenkin jaetun moodin kehittämiseen pelissä ja niihin haasteisiin, joita pelin kehittäminen kahdelle pelaajalle toi mukanaan.

Avainsanat Pelinkehitys, Unity 3d, C#, Moninpeli, Cinemachine

Sivut 38

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Name of degree programme
Campus

Author Pasi Saarikalle **Year** 2021

Subject Co-op split screen game with Unity 3d and Cinemachine plugin

Supervisor(s) Lasse Seppänen

ABSTRACT

The goal of the thesis was to figure out how a so called split-screen game is made using Unity3D. The game itself was made with Unity game engine and heavily utilizes a plugin called Cinemachine for its camera work. The main subject of the thesis was focused around the split-screen feature of the game, but it is also briefly going to go over some of the more essential features of the game project, like the logic behind the artificial intelligence of the Non-player Characters (NPCs), as well as some of the core gameplay features, such as the reputation system that was implemented into the game.

The theory part of the thesis went over the history and some of the milestones of a modern game engine. In the theory the thesis also went over the Unity3D game engine and its basic features. The plugin that we used, Cinemachine and its uses, was also written about. In the theory part the programming languages used in this game are also going to be talked about.

In the practical part of this thesis the author wrote about the teams approach on developing the game. The core mechanics were also written down so the game and its mechanics would become more apparent.

Keywords Game Development, Unity 3d, C#, Split Screen, Cinemachine

Pages 38

List of concepts

Unity 3D = A multi-platform game-engine made for developing 3d games.

C# = C# is a .NET based programming language.

GUI = Graphical user interface. Form of User Interface that allows the user to interact with the program or in this case, the game.

Cinemachine = Cinemachine is a plugin for Unity 3d, used to control camera movement in-game.

Split screen = Split Screen is a term used for local, co-operative game mode, where two or more players share the same screen, but the screen is split amongst the players.

Co-op = A multi-player game mode, where players have to cooperate to achieve a goal.

Asset = An asset in the context of a vide game development is anything that goes into a video game – animations, sound files, character models, scripts etcetera.

Panda BT = Panda BT is a behaviour tree model used to develop artificial intelligence.

NavMesh = NavMesh is Unity game engines navigation mesh that is mostly used to navigate artificial intelligence or player characters.

Mesh = In the context of game developing, a mesh is a part of a 3D/2D model. A 3D model is built from different meshes and can be textured.

SCRUM = Scrum is a project management framework that is used a lot in software development.

Contents

1 INTRODUCTION

The game industry is at the moment of writing, bigger than ever. No matter the age, most people have at some point played a video game. The uprising trend of online multiplayer games has drastically reduced the market for local multiplayer games, such as split screen co-op games; people do not have to gather around one television to play with friends and family anymore.

During my exchange program at HAN university of applied sciences, we had a project where we made a game. The game is a medieval, fantasy style, innkeeper simulator, where the player has to serve people from across the world and make a profit. We wanted to go back to the roots of co-operative gaming and include a split screen game-mode for people to play together while physically being present. We came to the conclusion that we wanted to utilize Unity3d-plugin called Cinemachine.

Cinemachine is a plugin that gives greater and more seamless control over the camera movement inside the game. It can also be utilized in cutscenes.

I will be researching the next topics in this thesis:

- How to create a seamless transition between a split screen and a shared screen?

- How to utilize cinemachine in a split screen setting?

- How is the occlusion of the walls and objects optimized in a game with two cameras and points of view?

- How does the split screen affect the rotation and placement of the GUI elements (such as speech bubbles)?

2 UNITY

Unity is a game engine, developed by Unity Technologies. The game engine was released in June 2005 as a Mac OS X-exclusive game engine. It is mainly used to make 3D (three-dimensional) or 2D (two-dimensional) video games, simulations, augmented reality games and virtual reality experiences. At its dawn, it was mainly used to develop video games, but later on has been adopted by industries other than gaming; film, architecture and construction, for example. (Unity, 2017)

Unity technologies constantly update the game engine and it has seen many different iterations during its lifespan, the latest stable build being 2020.1.5, which was released in September of 2020. (Unity, 2017)

2.1 Game engine

A game engine (also referred as game architecture, game framework or gameframe) is framework to build a software, or more specifically, a game onto. At its core, a game engine usually consists of graphics renderer or render engine, physics engine, sound engine, collision detection, scripting and localization support. It also typically includes video support for in-game cinematics and in-game cutscenes. (Jeff Ward, 2008.)

Game engines give an array of different visual development tools and software components. These tools help game developers to create game experiences, without having to create a lot of the game infrastructure from the scratch. (Jeff Ward, 2008.)

There are many types of game engines; some are free to use, some are not. There are also many custom game engines, made for specific games by their developers that are only used in-house. These are typically more used by the big companies, rather than independent developers. Many game engines also provide customizability extensions, such as plugins or

add-ons. These extensions are meant to extend development capabilities, or increase efficiency in certain areas of development. (Stelios Xinogalos, 2017.) Different game engines often offer unique features, while still having similar core functionalities. (Stelios Xinogalos, 2017.)

2.1.1 History of game engines

Before the rise of game engines, developers had to create their entire game infrastructure from scratch. Every time a game was developed, every asset, physics engines, graphics renderers, audio engines etcetera had to be built from nothing and it was slow. That created a desire to create a framework to develop games upon that would have the basic infrastructure of game development already on it. This has increased the efficiency and speed of which games can be developed. (Jeff Ward, 2008.)

One of the first “modern” game engines was Wolfenstein 3D engine. It was developed in 1991 by John Carmack. This was the engine that the game Wolfenstein 3D was built upon, and if you have ever played any first-person shooter, this is the engine that popularized the genre. John Carmack also developed an engine in 1993, called “IDTECH1”, or more commonly known as “Doomengine”. It is the game engine that was used in creation of the popular DOOM game, shown in figure 1. This game engine was a step forward in the graphics engine technology. (Michael Hitchens, 2011.)

Figure 1 Video game DOOM

2.1.2 Graphics engine

In more simplistic terms, graphics mean the visual representation of the game. The rendering capabilities of the game engine dictates the appearance of the game. This means 3D or 2D models, animations, textures and other various asset types. These are commonly created on third-party software, such as Blender or Maya. The engine usually then gives integration capabilities to integrate the files straight to the desired project. (Michael Hitchens, 2011.)

2.1.3 User input

Game engines commonly provide a built-in input detection system for the most common types of inputs such as keyboard, mouse and controller. Engines also give various ways to detect said inputs such as key presses, key releases and holding a key. Nowadays, with the popularization of virtual reality (VR) stations, game engines have also started to integrate virtual reality control inputs into their infrastructures. (Michael Hitchens, 2011.)

2.1.4 Audio in a game engine

Audio is also a basic pilar of games. Game engines typically support the direct use of the most common audio file formats like .wav, .mp3, .M4A and .wma. Like the graphical assets, audio assets are usually recorded and made with third-party software, such as Audacity. These files can then be integrated into the game engine. (Ronny Mraz, 2020.)

2.1.5 Networking Frameworks

Online capabilities in a video game is a desired feature in modern video games. Many of the biggest game engines have integrated some sort of networking frameworks inside their engines. The basic idea is that all the players that are in the same session, would retain the same “Game-state”. Many of the modern game engines come with components that helps to build the client to server connection. (Ask Gamedev, 2018, video.)

2.1.6 Scripting in game development

Games objects need to be attached with a script to behave in a certain way like reacting to a player character, for example damaging the player when the player hits the game object. Most of the modern game engines offer a plethora of scripts to speed up the process of developing a game. Game engines also often include their own code libraries with various functions. (Walker White, Christoph Koch, Johannes Gehrke, and Alan Demers, 2009.)

2.1.7 Physics engine

Game Engines offer a support for physics engines. Physics are important to make a game feel realistic. Physical forces such as gravity are calculated in the physics engine through mathematical problems, and the physics engine does that for the game. Physics engine does this, trying to simulate real-world physics, to make the movement of the game objects feel life-like. (Partha Sarathi Paul, Surajit Goon, Abhishek Bhattacharya, 2012.)

2.2 **Unity3d**

Unity3d is a specifically made iteration of Unity game engine, designed for developing cross-platform, 3D games, simulations, virtual reality experiences etcetera. Unity3D is a free program, excluding a *Pro edition* with more tools and features, which costs 150$ a month and the Unity Plus, which is 40$ a month. (Unity, 2017)

Unity3D is a simplistic editor, while still offering powerful tools for expert game developers. Unity3D also has a wide variety of open-source plugins, which are easily installed from the editor itself. (Peng Xia, 2014.)

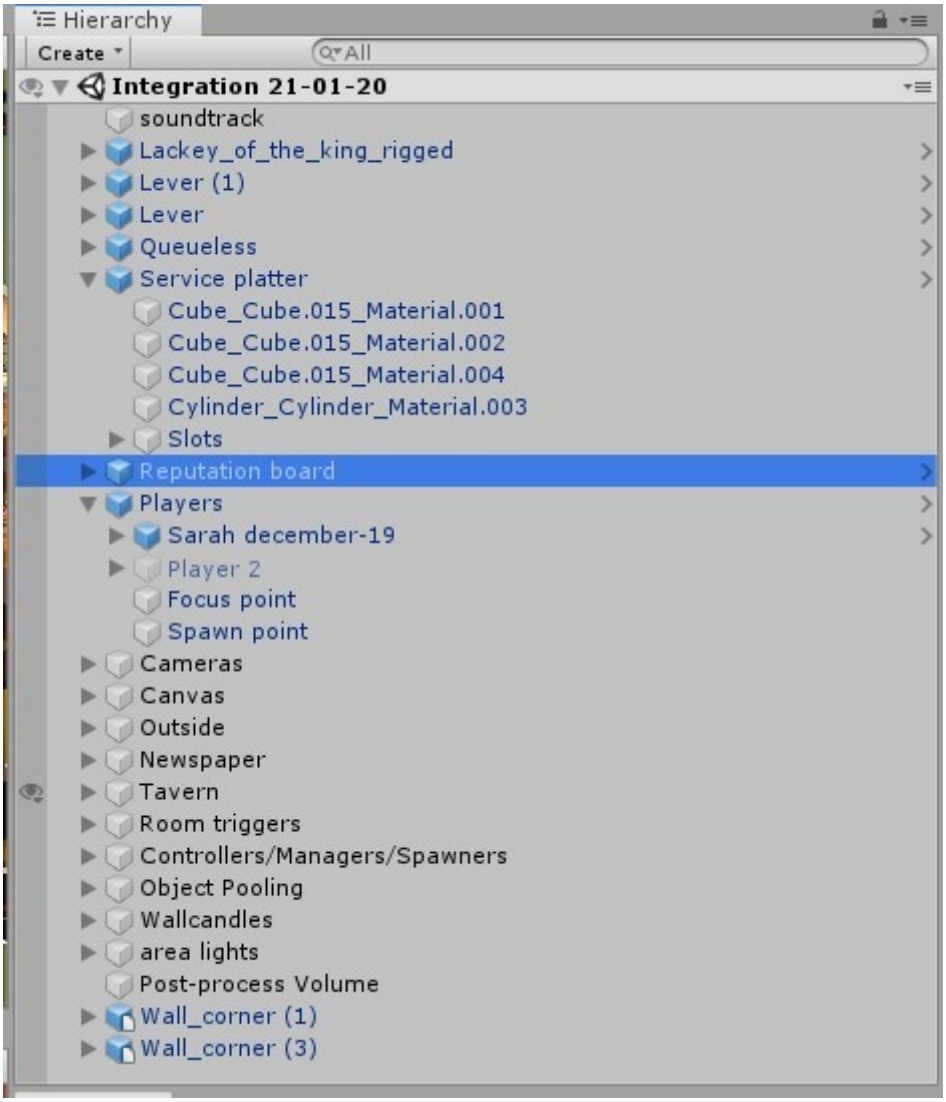
The scripting in the game engine is mainly done with three programming languages, supported by Unity; UnityScript, C# and Boo. Scripts in Unity are known as *behaviours.* They can be directly linked to an asset in a project to make them interactive. (Peng Xia, 2014.)

Unity is compatible with most of the 3D applications and most of the major Audio file formats such as .Wav, .Mp3 (Peng Xia, 2014.). Unity3D is one of the two most widely used game engines in the market as of August 2020.

2.3 **The editor application**

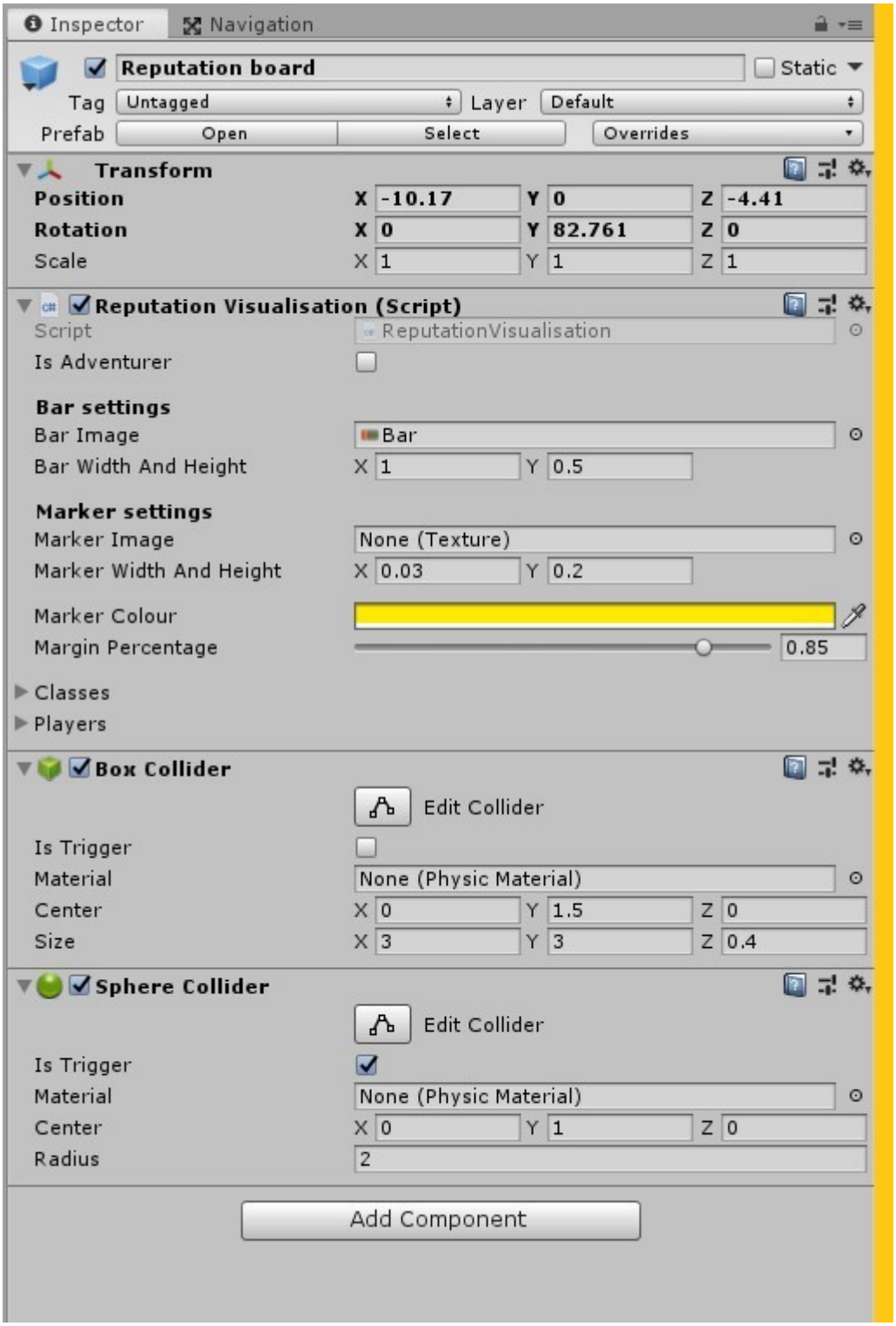
Unity3D's editor application is simplistic enough to learn quickly, and it is easy to navigate. Its windows and panels can be dragged and dropped to make the editor most suitable for the user.

Figure 2 Unity hierarchy panel



The hierarchy panel (see figure 2) indicates, where the assets are stored in the scene. It also displays the hierarchy between the assets and objects. For example, if some object is other objects “child”, it will be shown under the parent object. This makes it convenient to make out the hierarchy of objects. It is also possible to hide objects temporarily in this window. This makes said object invisible during the editing process. (Peng Xia, 2014.)

Figure 3 Unity Inspector Window



The inspector panel (figure 3) gives the option to inspect individual elements of the object: from position and rotation to ragdoll physics and gravitational properties. This window allows the developers to tweak the

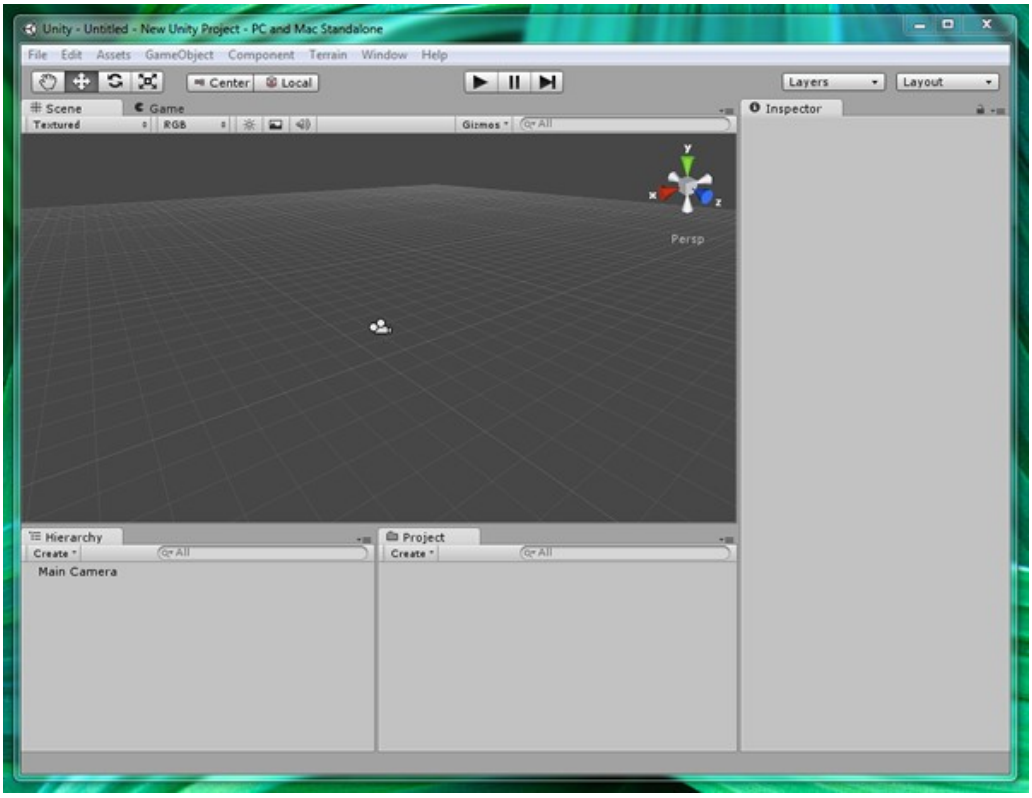
individual properties of the game objects, and attach new elements to them, such as scripts. (Peng Xia, 2014.)

The projects panel shows the many different assets of the current project. They can be put into a folder to keep them organized. Here is where all the scripts, 3D models, audio files and all the other assets of the project are located. (Peng Xia, 2014.)

The scene window, or the main window, is a 3d view of the scene that is being developed. The assets can be physically dragged and dropped into the scene from that view. It can be panned, rotated, or zoomed in the window. The scene window also functions as a play-mode window, where the developing party can play test the project. (Peng Xia, 2014.)

There are also some basic menus and play scene button on the top of the screen, where it is possible to save and load up projects, import assets, choose the properties for the game objects and more. These are seen in the figure 4.

Figure 4 Unity Editor appearance



2.4 **Assets**

Assets are everything the project uses in the development, from 3d models to scripts and audio files. Aside from a few basic shapes and scripts, Unity cannot create its own assets, but they are usually made with a third-party program like Maya3d. They are easily imported to the Unity3D editor from the editor itself. Assets are easily imported into the editor by just dragging and dropping them into the asset panel inside the editor. (Manninen, Thurlin, 2007.)

The assets go into the Project panel inside the editor. Here it is possible to organize them into folders, and make adjustments to the assets properties, before using them in a project (Manninen, Thurlin, 2007. Unity accepts most of the popular 3d file formats and audio formats. All of the common image file formats are also supported by Unity3D.

2.5 **Scenes**

A scene is where most of the actual “building” happens. It is a 3D field, where it is possible to drag and drop assets to a scene and move them around. It is also a viewport, on how the project will look when the play button is pressed. When a new project is created, the Unity3D editor will automatically make an empty scene, to which the project can be developed upon. A project can also use multiple scenes during runtime. It is possible to hide assets on the scene by clicking on the eye-icon of said object on the hierarchy panel. (Peng Xia, 2014.)

The hierarchy panel will show the hierarchy of the used assets in the scene. From there it is also easy to add scripts and properties to the assets. (Peng Xia, 2014.)

2.6 **Scripting in Unity3D**

Scripts are called *behaviours* in Unity3D. It is accurate, since scripts make assets interactive and thus, giving them behaviours. Scripting in Unity3D is mainly done with one of the three Unity3D supported scripting languages: C#, Boo or JavaScript. The scripts allow to create custom triggers for game events or respond to user input in the desired manner. There was a custom script for Unity, called UnityScript. It was very similar to JavaScript but was designed for Unity specifically. Unity stopped the support for UnityScript in 2017 however. A new script can be created inside the asset menu and then drag the script to an asset, giving the object a script or a behaviour. (Peng Xia, 2014.)

It is also possible to give objects attributes in a script that can be changed straight from the editor. For example, to make a player movement script that would allow the player to move, and the developer would give it an attribute inside the script called “speed” that would determine how fast the player moves, it could be made it public so that it could be changed

directly from the inspector panel. By default, Unity uses Visual Studio as its script editor software. This can be changed from the External Tools panel (Unity>Preferences).

A script, by default, derives from a class named MonoBehaviour. The MonoBehaviour is a built-in class that allows the script to work with Unity. There are also two default functions inside the class called Update and Start. The main difference is that the Update function updates and goes through the function every frame update of the game object, while the Start function only gets called before the actual gameplay, once. This means that any code that needs to get called continuously, such as movement script, should be inside the update function. The Start function is a good function to initialize any code before the gameplay. These functions are shown in code-snippet 1. (Georgi Invanov, 2016.)

Code snippet 1: Unity default script

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Thesis : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

3 CINEMACHINE-PLUGIN FOR GAME DEVELOPMENT

Cinemachine is a free plugin for Unity game engine. It is used to control the movement of the camera inside a project. It can be used for variety of purposes. First-person shooters to follow cameras, Cinemachine can be used in 2d-space and 3d-space alike. Cinemachine can also be used to make nice, professional-looking cutscenes. It was made as a way to replace extensive camera-logic development in game development. Cinemachine and Unity has also won an Emmy Award for Technology and Engineering. (Unity, 2017)

The procedural nature of the modules of Cinemachine makes it bug-resistant. When making adjustments inside a project (for example changing objects animation) Cinemachine adjusts its behaviour dynamically. This gives Cinemachine the edge over hard-coded cameras; there is no need to manually change the code when making adjustments in the scene of the project. (Unity, 2017)

3.1 **Behaviours**

Cinemachine has four main purposes and behaviours: composer, transposer, free-look and 2d-specific features. (Unity, 2017) Composers main task is procedural targeting and to automatically rotate the camera to keep the desired objects in any specific position inside the screen. It also allows real-time procedural composition with controls, width and height damping and frame adjusting.

Transposer or follow camera is a camera element that can be attached to any object to follow it. It is possible to configure how the camera will follow the object. (Unity, 2017)

A Free-look camera is an orbit camera, which is mainly used to manage third-person games with its orbital camera rig. The camera can be

configured with many controls like orbital speed, shape and input type. (Unity, 2017)

Cinemachine also supports many 2d-specific features. These include 2d framing, where it is possible to track and follow certain objects, as well as 2d orthographic rendering (orthographic rendering is similar to the classic “perspective view”, but in 2d-space). (Unity, 2017)

Cinemachine has a special feature, where it can save and preserve the adjustments made in Play Mode (when the project is running inside the editor). Normally the developer would need to stop the play mode to make these adjustments. This is a very convenient feature, as it allows the developer to adjust and tweak the camera positions in runtime. (Unity, 2017)

3.2 **Installing Cinemachine**

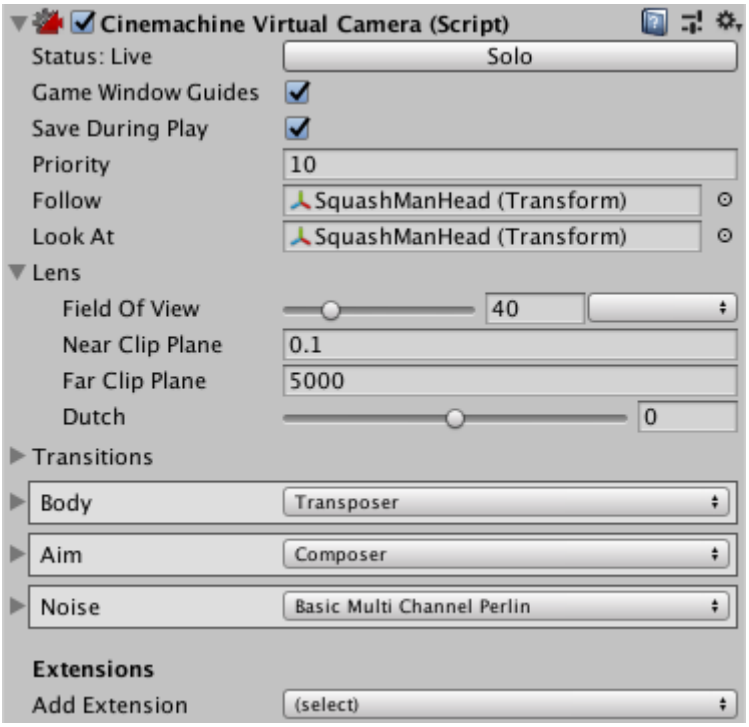
Cinemachine is easy to install. In the top-menu of the editor, got to the Window-tab, select package manager and select Cinemachine for installation. It is also free and available for commercial use.

3.3 **Setting up the cameras**

Setting up the cameras with Cinemachine is simple. In the Unity menu, choose Cinemachine and create virtual camera. Unity creates a new GameObject that has a component inside, called Cinemachine Virtual Camera.

There are properties inside the Cinemachine Virtual Camera component. The two main ones are called *Follow* and *Look At.* They can be assigned to a Game Object into either of the properties, for the camera to follow, or look at a certain game object. There are also other attributes, such as *Field of View* which adjusts, how much the camera sees in front of itself.

Figure 5 Cinemachine Virtual Camera properties



The other properties are shown in the figure 5. One of the more important properties is the Status attribute. The virtual camera has three following states: Live, Standby and Disabled.

Live state indicates that the camera is controlling a Unity camera GameObject that has a Cinemachine Brain script attached. If two cameras are blended into each other, both cameras will have their Status: Live. (Unity, 2017)

Standby state means that the Virtual Camera is not currently controlling a Unity camera object. It is however, still following and targeting the desired targets. (Unity, 2017)

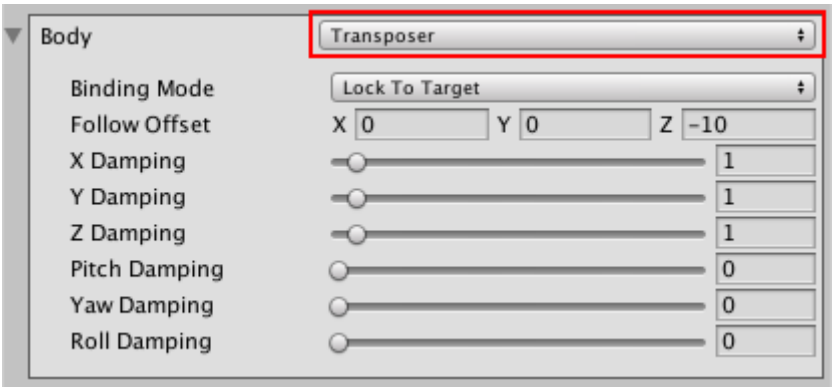
Disabled state indicates that Virtual Camera is not actively following or targeting any GameObject, nor does it control the Unity Camera. It also does not use any processing power, so it is a good practice to disable cameras that are not being worked on at the moment, since the cameras can take a good amount of power. It is still active on the scene however,

and the changes that are made to the virtual camera, are being preserved. (Unity, 2017)

3.4 Body, Aim and Nose properties

Inside the Cinemachine Virtual Camera there are three properties called *Body, Aim* and *Nose*. These are the logic properties of the Virtual Camera and these specify, how the virtual camera is going to track, follow, target and rotate the camera. These properties are also transferred to the Unity camera GameObject with the Cinemachine Brain when the virtual camera takes control of said Unity camera. (Unity, 2017) The Body property specifies which algorithmic logic the Virtual Camera is going to use to move the Virtual Camera around (see figure 6).

Figure 6 Cinemachine Virtual Camera Body properties



There are six different algorithms to move the camera. "**Transposer**" moves in relation to the follow target. "**Do nothing**" keeps the camera still. "**Framing Transposer**" is similar to "transposer", but it keeps the fixation in relation to the screen-space. "**Orbital Transposer**" is again, similar to "transposer", but it allows the player input to dynamically adjust the camera (I.E zoom out, zoom in). "**Tracked Dolly**" moves the camera along a pre-determined path. This is good when making in-game cutscenes or other more cinematic scenes. "**Hard Lock to Target**" uses the exact location of the follow target. (Unity, 2017)

4 PROGRAMMING LANGUAGES USED IN THE GAME

C# (C-sharp) is a common .NET based programming language, developed by Microsoft in 2000. It is a simplistic, object-oriented programming language, and is related to C and C++ program languages. (Veli-Matti Sivonen, 2004)

4.1 .NET core

.NET Core is a free, open-source version of .NET framework. The framework is a framework for developing software and running applications on Windows. Unlike .NET Framework, .NET Core is cross-platform, meaning it runs on macOS and Linux operating systems as well.

.NET Core Framework is mainly used to develop mobile, desktop, and web applications, as well as games. It was designed with a purpose to make it lightweight and cross-platform, and has all the features that are required to run basic .NET Core applications.

.NET Core is used over the traditional .NET Framework, mainly because it is cross-platform. Also, developing for different Windows platforms is more compatible with .NET Core. Also, because of the wide array of different devices that are used with the software, .NET Core functions as the single framework that works globally between web applications, Windows Phones and Windows Desktop.

4.2 C#

C# is modern, object-oriented programming language, inspired by other object-oriented programming languages, like C++, SmallTalk and Java. The goal of C# was to create a programming language that has support for the first class functions to increase stability for the developed program. The goal was also to create a programming language that combines the

productivity of Microsoft Visual Basic and the productivity of C++. (Veli-Matti Sivonen, 2004.)

As a programming language, C# is interconnected with with .Net framework. It does not contain its own libraries for example but gets the from .Net class libraries. The .NET class libraries are mostly made with C# though, so they are symbiotic in a way. (Veli-Matti Sivonen, 2004.)

5 MAKING A SPLIT SCREEN GAME

In this chapter the thesis will go through how the project-team made the split screen feature to the game utilizing the Cinemachine camera plugin, and will go over briefly how the team made the necessary tweaks to the script to make the player experience as smooth as possible. This example is made using an already existing (work in progress) video game that the team spent 12 weeks making.

The game idea came to fruition from the concept owner. He had an idea for a game that would be a tavern simulator in a high-fantasy world that was filled with dwarfs, warriors and wizards for instance. The first week the team spent forming the project groups and brain storming ideas for the game. The project had four main teams: The programmers, the game designers, the art team and the sound team. I was one of the eight programmers working on the game.

The project management tool of choice was SCRUM. The team had a daily morning meeting within the teams, and on Mondays and Fridays there was a meeting as a full team, discussing what the team would do the following week and what the projects focus would be. Fridays were more for the retrospective brain storming. The course also had a biweekly event, where the team showed the whole course the projects progress, and got feedback on the game. So the team had a nice feedback loop going for the project.

5.1 Overview of the split screen feature

The game concept is a tavern simulator, where the player plays as a bartender that tends a medieval tavern that is set in a fantasy setting. The player has a daily quota for customers served that has to be met in order to progress to the next day. The player has to serve the customers with food, drinks and weaponry for their adventures, and keep the tavern tidy

by cleaning up the place. The game is playable in a single player mode, but it can also be opted-in for a split screen mode where the player can play with a friend in local co-op mode.

Figure 7 Split screen mode in Behind the Tankard

In figure 7 is a demonstration of the split screen feature. “Player one” controls the left screen, and “player two” controls the right screen. All the assets used are made by the team utilizing Maya3d and Blender that are then imported into Unity3d. These ”modes” can be switched back by pressing *f2* key. This causes the other playable character to disappear and the screen going back to one player view only.

5.2 **Making of the base game**

Before the team started to make the game, it was crucial to plan all the features the project-manager wanted to implement into it, and plan how the team wanted to approach the game making process. The team had the project split into four smaller teams: Programmers, Art department, sound engineers and game designers. I worked in the programming team.

The programming teams first step was to make a character “dummy” and a test-level and make the character move. The programmer-team created

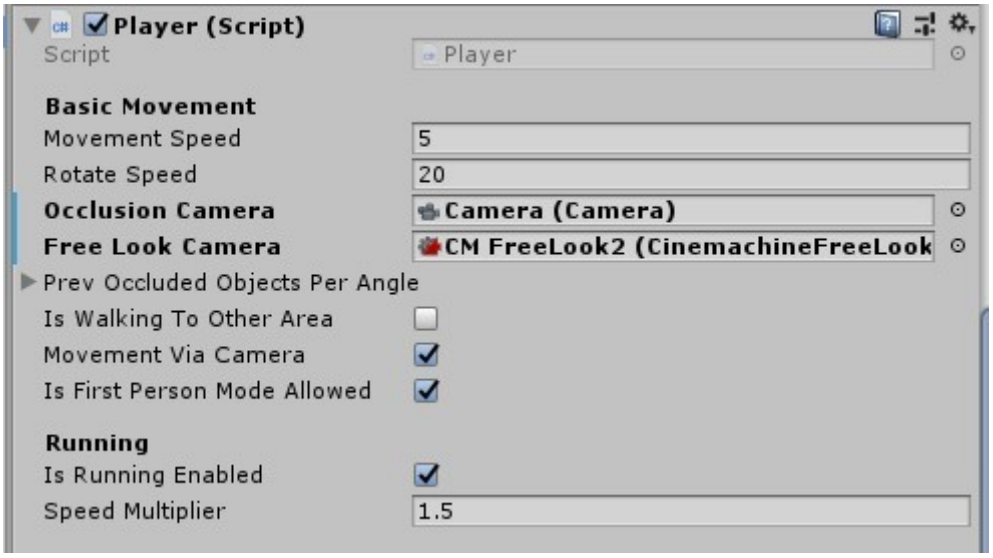
a player-dummy using the basic unity3d object assets for a dummy and attaching a simple movement script for it. The games first iteration of the movement in the game was a so called “click to move” movement, where the player click a spot the player wants the character to move and it would move there by using a so called NavMesh-system. NavMesh system is ‘’s built in feature that creates an invisible navigation mesh on top of a plane and is mainly used for this type of movement, as well as AI’s movement. The team then decided to scrap that in favour of the more classic “WASD”-movement. This means that “w” key is pressed to move forward “a” to move left, “d” to move right and “s” to move backwards. This was the desired movement mechanic, because the project manager wanted to integrate full console-controller support for the game and by using this method of movement in the game, it made it easier to integrate full controller support for the game.

Code Snippet 2: player movement

```
private void Move()
    {
        float xInput = Input.GetAxisRaw(horizontal);
        float zInput  = Input.GetAxisRaw(vertical);

        // Set the animation state
        SwitchBetweenIdleAndWalkingAnimation(xInput, zInput);

        // Get the direction the player wants to move in
        Vector3 direction = movementViaCamera
            ? Vector3.ProjectOnPlane(OcclusionCamera.transform.forward, Vector3.up) * zInput
+
              Vector3.ProjectOnPlane(OcclusionCamera.transform.right, Vector3.up) * xInput
            : transform.right * xInput + transform.forward * zInput;

        // Get movement speed
        float speed = isRunningEnabled && Input.GetButton(sprint) &&
!_inventory.HasInventory()
            ? movementSpeed * speedMultiplier
            : movementSpeed;

        // Get gravity
        Vector3 gravity = _controller.isGrounded
            ? Vector3.zero
            : Physics.gravity * Time.deltaTime;

        // Move the character
        Vector3 displacement = speed * Time.deltaTime * direction.normalized + gravity;
        _controller.Move(displacement);

        // Rotate the character
        if (movementViaCamera)
        {
            Quaternion targetRotation = Quaternion.LookRotation(direction +
transform.forward);
            transform.rotation = Quaternion.Slerp(transform.rotation, targetRotation,
rotateSpeed * Time.deltaTime);
        }
    }
```

Code Snippet 2 is a code snippet from the player script that controls everything to do with the player-character. This particular method is the projects movement script. It checks the direction, where the camera is pointed using vectors, and changes the direction where the player character moves when going forward. For example, if the player is going forwards towards north, but rotate the camera towards east, then the player character will go east when pressing “w”, instead of continuing to go north. There is also a conditional that checks if the player presses the “shift” key that enables “running” and raises the movement speed of the player character.

Figure 8 editor options



In figure 8 the public modifiers are shown. This is made by setting the float variables like Movement Speed and Rotate speed public. In this case it was desired to have public modifiers like Movement speed, which controls the player characters movement speed, while rotation speed sets the characters rotation speed around its axis. The other checks are just there for debugging purposes. Movement Via Camera defines, whether or not the directions of the players are defined via the cameras direction, instead of the characters direction it is facing. The game also has a “first-person” mode, which means that it is possible to zoom into the player characters perspective.

The programmers started to implement the world-interaction system. The team wanted the player to be able to pick up items from the ground and for example: take a clean keg from the counter, go fill it up with a brew and serve it to the customer. The customers were all NPCs (*Non-player-controlled characters).* The interaction system was made with two scripts called Inventory and PickUpItem. The Inventory script created a list of the items, based on what the player was already carrying and created some conditions for the player character (Maximum capacity, or is the player already holding an object) and the PickUpItem was a script that was attached to every interactable item. The PickUpItem script transferred the parent object to the player character and added the item to the list created

inside the Inventory script. The PickUpItem script also had a method to drop or throw the item the player character was holding, as well as interacting with the player's current target with the item the player was holding. For example, if the player character is holding a plank of meat in its hand, and the player character is targeting a customer and the player presses the interaction key, it would try to serve it to the customer. Or if the player character was holding a broom and would interact with it on a pile of dirt, it would sweep it.

It was also necessary to create an artificial intelligence (AI) for the non-player controlled character (NPC). This project used Panda BT Behavioural tree model for that. Behavioural tree is a logical model created to mainly create game AI. The Games NPC's utilizes Unity's NavMesh-system for their movement. After spawning the NPC, the NPC will have a chance to walk into the bar, and order something from the counter. The NPC will then take a seat, and wait for five minutes for its order to be fulfilled. If this is not met, it will leave the tavern and the reputation with the NPCs faction will lower and will cause the said faction to come to the tavern less frequently as well as paying less money. The NPCs also have a chance to go to an armory counter and ask the player to sell them swords. They also have a chance to start fighting each other as well as leaving dirt tracks around the tavern that the player needs to clean up.

The Reputation system is a system that was made to give a fun meta-game (a game within a game) for the player experience. The NPC spawns with a certain faction: Noble, Hunter, Paladin, Mage, Warrior, Peasant, Cleric and a Warlock. The reputation for each faction is indicated on a notification board outside the tavern and will show the player its standing with each faction (see figure 9). The Reputation will lower or rise, depending on how satisfied the NPCs are with their order. This is a sum of how fast the player serves them, whether or not their order was correct and how tidy their tavern is. The players standing with the faction will affect how much the

said faction will visit the tavern. For example, if the players standing with the peasants is bad, they might not visit the tavern at all and thus making it harder to reach the daily quota.

Figure 9 Reputation Board

There are also a set of races the NPCs can spawn as: Elf, dwarf or Human. These races have different modifiers, for example elves are much more impatient and will leave if the tavern is dirty, where dwarfs do not change their mood depending on the tidiness of the tavern.

5.2.2 The flow of the game

The game starts with an NPC called kings lackey coming to the tavern, giving the player the daily quota that the player must fulfil in order to progress to the next day. The daily quota is a pre-set amount of money that the player has to make in a day (in-game day has been set to 10 minutes at this point). Then the customers start coming. The customers are all NPCs, and they will go queue on the counter. After the player takes the NPSs order, an icon of their order will appear above their heads, and they will go and take a seat on a free table. Once seated, the timer starts for the player to serve them and keep them happy. The player must pick a correct vessel for the serving, whether it is a mug for a brew, or a steak

plank for dinner. The player then can go fill the said vessel with a corresponding workstation (keg for beer, oven for meat). The player go es and serves the customer, and if it was done in under five minutes, the customer stays happy and is indicated with a thumbs-up icon on top of their head. (shown in figure 10). When the customer leave, they will either raise the reputation with the said faction or lower it, depending on whether or not the player served them in time, or whether or not the tavern was clean enough. There are also adventurers, looking to buy weapons. The team made a separate counter for the armory part of the tavern. The basic idea is the same as with the food and drinks, the NPC will queue in front of the armory and an icon on top of their head will indicate what they want. For now, the armory only serves swords.

There is also a random chance to spawn a wandering merchant during a day, which will sell the player some goods (like more tankards, more stake planks). The player buys these using the currency made during the day. This will of course take away from the goal of the day. After the time is up, and the player has collected the coins for the quota, the game will progress to the next day and the quota will be bit tougher to accomplish. That is the main flow of this game.

Figure 10 satisfied customer

5.3 Wall occlusion

The project team wanted to have an occlusion mechanic in the game to improve the feel of the game. What that means is that if the player stands next to a wall or under a roof, the wall and roof disappears so that the player can have a better vision of the player character. Making a wall-occlusion mechanic in its core is straight-forward. The wall that is desired to be occluded is set as a layer, and for example, if a player is facing a wall, and the camera is on the other side of the wall, the walls mesh is disabled, rendering it as an invisible wall. The problem arises when there are two players and two separate cameras and it is not desired for the culling to be global, meaning the mesh needs to render invisible only to the relevant camera. For example: Player one is in a different room, and the roof is culled invisible to improve the player ones visibility, but the roofs mesh for player two is desirable for aesthetic purposes if player two is in the other room and culling the roof would not grant any visual benefit for the player.

The programmer team did this with a couple of scripts; a so called Occlusion trigger script, which is a script that was added to an element “above” each room-plan that would act as a detector where the players were moving and where the camera was facing. Nearly every wall between the player and the camera was made invisible in the tavern.

Code Snippet 3: occlusiontrigger script

```
private void Awake()
    {
        AllOcclusionAreas = FindObjectsOfType<OcclusionTrigger>();
    }

    private void OnTriggerExit(Collider other)
    {
        DisableOcclusionOnExit(other);
    }

    public void OnTriggerStay(Collider other)
    {
        IsPlayerWalkingToOtherArea(other);
        HandleOcclusion(other);
    }

    public void OnTriggerEnter(Collider other)
    {
        AddPlayerToArea(other);
    }

    private void AddPlayerToArea(Collider other)
    {
        foreach (Player player in players)
        {
            if (other.gameObject == player.gameObject)
            {
                playersInArea.Add(player);
            }
        }
    }
    private void IsPlayerWalkingToOtherArea(Collider other)
    {
        foreach (OcclusionTrigger area in AllOcclusionAreas)
        {
            if (area.gameObject != other.gameObject) continue;
            if(area.gameObject == gameObject) continue;

            foreach (Player player in playersInArea)
            {
                if (area.playersInArea.Contains(player))
                {
                    player.isWalkingToOtherArea = true;
                }
                else
                {
                    player.isWalkingToOtherArea = false;
                }
            }

        }
    }
```

In Code Snippet 3 there are methods that checks how many players are in the area that the script is added to. The OnTriggerEnter checks if the player character enter the triggers area, and adds the player character to a list via the AddPlayerToArea method. The list is just a public list that keeps track

of the players inside the trigger area and makes the occlusion for both players easier.

The OnTriggerExit is a built-in command from the unity3d library that allows to make triggers that check if an action happens on trigger, whether it is exiting the trigger-area, entering it, or staying on it. OnTriggerExit specifically checks, if a playable character leaves the trigger-area and removes it from the list that was set earlier that keeps track of the players in the area of the trigger. This method then calls another method (DisableOcclusionOnExit) that removes the occlusion of that area.

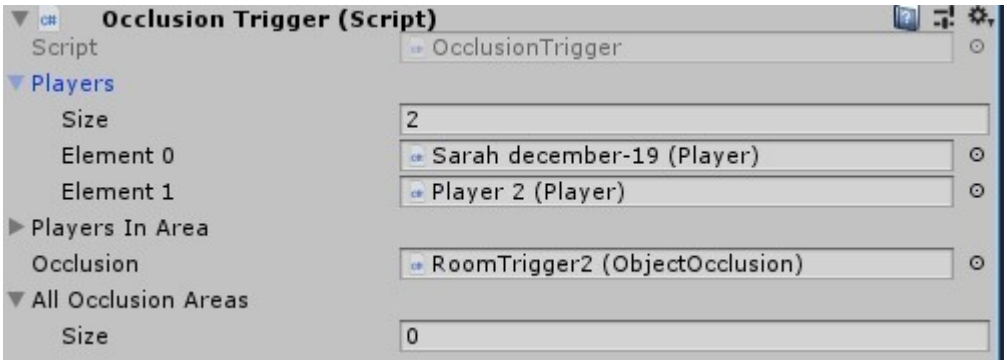
Code Snippet 4: HandleOcclusion

```
private void HandleOcclusion(Collider other)
    {
        int playerNr = 1;
        int otherPlayerNr = 2;
        foreach (Player player in players)
        {
            if (other.gameObject == player.gameObject)
            {
                if (occlusion.HasPlayerChangedAngle(player) || !player.isWalkingToOthe
                {
                    SelectOcclusion(playerNr);
                    occlusion.ShowOnExit(player, _chosenOcclusion,
GetOccludeLayer(otherPlayerNr), ObjectOcclusion.OcclusionAction.Stay);
                    EnableOcclusionForPlayer(player);
                }
            }

            playerNr++;
            otherPlayerNr--;
        }
    }
```

In code snippet 4 is the method that handles the logic behind the occlusion. It takes the information of the players in the area and asks if the player has changed the camera angle or if the player is not walking away from the trigger. It then sets the occlusion for the player by calling another method from another class, ObjectOcclusion. The method is called ShowOnExit, and it checks the players camera angle and calculates the layers that should be occluded for each player.

It is made this way because development team did not want the occlusion to be “global” but rather different for each player. So if *player A* is located in the kitchen near the southern wall, the southern wall and the kitchen roof would be invisible for *player A* only, while *player B* would still see the walls and the roof of the kitchen, if player two would be for example, outside or in a different room.

Figure 11 occlusion trigger, public modifiers



In Figure 11 the public modifiers that were added to the Occlusion Trigger script are shown. The main thing here is the Players value, which can be changed from the editor and will determine how many players are affected by the wall occlusion. If it is wanted to raise the player limit to three players for example, it can be added to the size and assign the player model to a new element slot that will appear and said player will also be affected by the occlusion. At this moment the game does not have support for more than two players, since it proved to be a difficult task to implement that as well as all the 2D-elements that needed to face the player camera.

After the scripts were done, the programming team created “trigger planes” for each room. The trigger planes are empty planes without any mesh or textures that functions only as a trigger to see when players enter the room or leave the room. The trigger planes were placed on top of each rooms floor. The OcclusionTrigger and ObjectOcclusion script were then

assigned to these trigger planes. The boxes on the modifiers of the collider to make it a trigger were checked (Is Trigger checkbox inside the editor).

Figure 12 Occlusion example in practise

In Figure 12 an example of the occlusion can be seen. The walls and kitchen door have disappeared for player one (the left screen) to give the player a better visual of the surroundings, while player two (on the right) is facing away from the kitchen, so the door and wall are still in place, since it is not hindering for the player experience to have the visuals of the walls there.

5.3 **2D-elements**

In the game there were many 2D-elements such as the reputation gain icons, mood change icons, money gained icons, order icons. The problem with the split screen mode was that the 2D-icons were always facing the player ones camera, and it hindered the visual feedback for player two. The team wanted to make it so that it would always face the camera both players. The approach to it was spawning individual 2d-elements for both players, and hiding the other players 2d-elements.

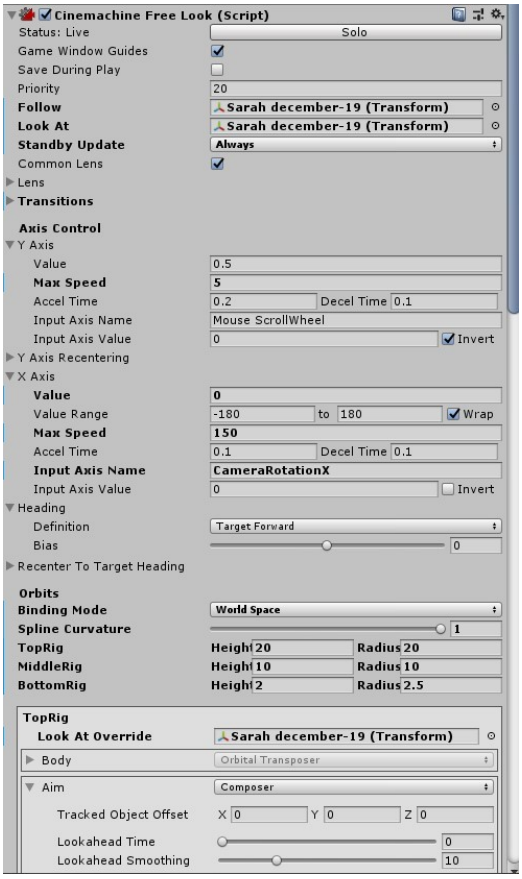
5.4 **How the Cinemachine plugin was used in practice**

When the project started, the team had the split screen mode in mind. The first iteration of the in-game camera was the basic Unity Camera

component that was simply attached as a child object to the player character. This did not work quite well, and the camera movement felt stiff and clunky, from the subjective perspective of the team. The team looked into different plugins to use for the camera work in the project, as it was desired to have more dynamic control of the cameras movement. The project management decided to pick Cinemachine plugin because it suited the games needs and it felt like it was the least labor intense and the team was on a deadline. The modifiers are already scripted in to the Cinemachine brain script, and the modifiers were already there, so the programmer team did not have to do much manual scripting.

The First thing that was created was a FreeLook camera by selecting the Cinemachine Tab on the top. The team set it up where it was desired to be approximately in relation to the player character. The team chose freelook camera, because it was wanted to be able to rotate the camera freely around the player. This could be done with other types of cameras in Cinemachine (e. g. Virtual Camera), but this suited the games needs well, and was simple to set up. After setting up the relative position of the camera, the built-in modifiers of the Cinemachine free look camera had to be tweaked to suit the games needs.

Figure 13 FreeLook Camera 1 modifiers



In figure 13 the modifiers the project had for the camera are shown. These modifiers come straight from the game object itself, as it comes with a built-in script. The main thing to do first was to select the follow target (a target that the camera follows) and the look at point (where the camera fixates its lense). For us, these were the same object, the player character. Sarah December-19 is just the name the art team gave for the version of the character, and the transform indicates that it reads the objects position in the plane, rather than its mesh, for example. This just makes it more precise, as it takes the pixel accurate position of the character.

The Y Axis modifiers are there to control the inward movement in the cinemachine free look camera. In practice this just means that the settings for the camera zoom can be tweaked. For this project, the team wanted to have it in the mouse scroll, as it felt the most intuitive.

The X Axis modifiers are there for the horizontal movement. In this case, it meant the orbiting movement around the player. The team tweaked around with the settings until the speed suitable for this game was found.

There was also some fine tuning with the orbit modifiers, especially in the Spine Tension, to smooth out how the camera moves when zooming in.

Now the game had a working camera for the player one. It would constantly be following player one and fixating on it so that the player could still freely rotate and pan the camera, in relation to the player character.

The second player camera was just copied from the Cinemachine free look camera, except changed the fixation point was set to player twos character.

5.5 **The split screen mode**

The team made the split screen feature quite simple. The project lead did not want the player two to have it is individual money system or daily quotas, but rather to have a global money system for both of the players. This would make hopping in and out of split screen more stream-lined and intuitive for the games purpose. The goal was to make a simple system that would allow the players friend or family member to just grab a controller and hop in the game.

Code Snippet 5: Split screen script

```
public class CooperativePlay : MonoBehaviour
{
    public static bool inCoop = false;

    public bool coopAllowed = true;
    public GameObject playerPrefab;

    public delegate void EnterCoop();
    public event EnterCoop enterCoop;

    public delegate void LeaveCoop();
    public event LeaveCoop leaveCoop;

    [Header("Controls")]
    public string coopButton = "Splitscreen";

    // Update is called once per frame
    void Update()
    {
        if (!coopAllowed)
            return;

        if (Input.GetButtonDown(coopButton))
        {
            if (!playerPrefab.activeSelf)
            {
                inCoop = true;
                playerPrefab.SetActive(true);
                enterCoop.Invoke();
            }
            else
            {
                inCoop = false;
                playerPrefab.SetActive(false);
                leaveCoop.Invoke();
            }
        }
    }
}
```

The split screen feature is made with a class called CooperativePlay, shown in code snippet 5. It is a simple script that has Booleans to check if the split screen feature is enabled that was made a public value, so that the developers can modify it from the editor itself. The project also has a public GameObject modifier that is set to the player two asset. The player two model is the same as the player one model, except it is recoloured to better distinct the two players.

The basic idea of the script is that it checks if the split screen button is pressed, and if it is true, it then spawns the second player and splits the screen in two, as seen in figure 9.

5.5.2 Challenges of split screen game mode

Even though the split screen feature and its development was not that complicated, the optimization of certain features proved to be challenging. The wall-occlusion mechanic for both players individually was difficult, as it relied on separating the occlusion for each of the camera. The first iteration of the occlusion-system was completely scrapped as it was a global system for both players, and it ended up hindering the gameplay for two-player mode, both in aesthetic aspects of the game, as well as gameplay wise.

The project also had a lot of 2D elements in the game like the icons that display on top of the NPCs (the reputation increases, the mood icons and money gained, as well as the orders that the NPCs placed). The 2D elements needed to be done in such a way that the specific 2D element was always facing the camera, whether or not the mode was two player-mode or a single-player mode.

6 RESULTS

This thesis was made to figure out the best approach in making a split screen game with unity and how to utilize the Cinemachine plugin with it. The game was mainly made in 12 weeks and continued as a hobby.

The team made a working game that had a working two-player split screen mode in it. Although challenging, the project team eventually got the wall occlusion to work properly. The game itself needs some more work if the team would ever want to ship it. The game is poorly optimized right now, and it stutters if played on a low-end computer. There were some features that the team also wanted to expand upon. The team had an idea to have a mechanic in the game that would let the player rent a room for an NPC, and it would give the player in-game currency, but it was not implemented because of the time limit. The project team also wanted the player to be able to customize the tavern, but for now that feature is scrapped completely. After this project, the team continued to work on the game as a hobby for a while, but people were too busy with school or work and the teams budget ran low, so for now the production of the game is seized. The team is however very happy with what was made.

The game was presented in the HAN-university event, where students got to show their projects they had been working on. This game won the crowd-favorite award. The team had set up a lounge where people could play the game and the crowd really enjoyed playtesting the game.

7 SUMMARY

Making an intuitive split screen game with unity and utilizing cinemachine plugin to develop this functionality was one of the main goals of this thesis. The theory part will give the reader some basic understanding of unity as well as the brief history of game engines in general. The thesis will also go over the main scripting language used in Unity game development: C#.
The research questions are answered throughout the practical part of the thesis, although the 2D elements were not that in-depth or complicated.

Writing the theoretical part of the thesis gave the author of this thesis more in-depth understanding of cinemachine. Cinemachine is a great plugin to lessen the workload required for camera work in game development and offers powerful tools to build a game with intuitive camera work.

The thesis also showcases most of the main mechanics made in the game and goes over some of them and how they were built in practice. This thesis will not go over the more in-depth code behind the functionalities, but rather tries to give the reader a general understanding on how we built these mechanics and how they can be utilized in game development.

The game development included in this thesis gave the author an opportunity to try and hone his understanding and knowledge of game development. The pursuit of a career in game developing is something that is a very exciting journey, and will be something that the author will heavily consider pursuing. The thesis also provided some valuable knowledge about some of the more unknown programming techniques for him, such as iterfaces. Making a game is a challenge on its own but the author being a relatively young programmer made it even more of a challenge. At the same time, it was very much a learning moment for the future.

REFERENCES

Unity. 2017. About cinemachine. docs.unity3d.com
https://docs.unity3d.com/Packages/com.unity.cinemachine@2.1/manual/index.html#about-cinemachine
Read: 28.01.2021

Siivonen, Veli-Matti. 2004. Ohjelmointikielten periaatteet: C#-kieli. cs.helsinki.fi
https://www.cs.helsinki.fi/u/pohjalai/k04/ohpe/seminar/Sivonen-CSharp.pdf
Read: 13.12.2020 – 15.12.2020

Ward, Jeff. 2008. What is a Game Engine? gamecareerguide.com
https://www.gamecareerguide.com/features/529/what_is_a_game_.php?page=1
Read: 12.12.2020

Hitchens, Michael. 2011. A Survey of First-person Shooters and their Avatars. Gamestudies.org.
http://gamestudies.org/1103/articles/michael_hitchens
Read: 04.01.2021

Mraz, Ronny Mraz. 2020. The Elements of Videogame Audio. Splice.com
https://splice.com/blog/elements-video-game-audio/
Read: 05.01.2021

Ask Gamedev. 2018. Game Engines Explained. Youtube.com.
https://www.youtube.com/watch?v=LMRZBKhQcRc&ab_channel=AskGamedev
Watched: 16.12.2020

Sarathi, Paul, Partha. Goon, Surajit. Bhattacharya, Abhishek. 2012. History and comparative study of modern game engines. Researchgate.com
https://www.researchgate.net/publication/259496289_History_and_comparative_study_of_modern_game_engines
Read: 02.02.2021

White, Walker. Koch, Christoph. Gehrke, Johannes. Demers, Alan. 2009. Better Scripts, Better Games. queue.ac.org
https://queue.acm.org/detail.cfm?id=1483106
Read: 05.11.2021

Xia, Peng. 2014. 3D Game Development with Unity A Case Study: A First-Person Shooter (FPS) Game. Theseus.fi
https://www.theseus.fi/bitstream/handle/10024/71525/Xia_Peng.pdf?sequence=1&isAllowed=y
Read: 28.11.2020

Manninen, Tony. Thurlin, Tomi. 2007. The Value of Virtual Assets – The Role Of Game Characters in MMOGs. Researchgate.net
https://www.researchgate.net/publication/26493733_The_Value_of_Virtual_Assets_-_The_Role_of_Game_Characters_in_MMOGs
Read: 27.11.2020

Invanov, Georgi. 2016. Introduction to Unity Scripting. raywenderlich.com.
https://www.raywenderlich.com/980-introduction-to-unity-scripting
Read: 28.11.2020

APPENDIX HEADING