TiltablePages

Augmenting tilted papers

Bachelor Thesis

People and Computing Lab
Department of Informatics
University of Zurich

Jan Gugler
14-716-849

University of Zurich[UZH]

PAC

Supervised by
Prof. Dr. Chat Wacharamanotham

Submission: 29 January 2019

Contents

List of Figures

Zusammenfassung

Augmented-Reality-Anwendungen werden in einer Vielzahl von Bereichen immer häufiger eingesetzt. Die Verwendung von Augmented Reality im akademischen Kontext ist jedoch begrenzt. In dieser Arbeit haben wir ein Tool entwickelt, mit welchem Forscher virtuelle Objekte in ihre Experimente einbinden können. Um dies zu ermöglichen, haben wir eine einfach anpassbare Tracking und Projektions-Pipeline implementiert. Diese Pipeline besteht aus drei Teilen: (1) außeraxiale perspektivische Projektion, (2) Kalibrierung und (3) Keystonekorrektur. Die perspektivische Projektion wird anhand von Trackingdaten berechnet, wobei eine 3-D-Illusion aus einer Fläche erzeugt wird. Das resultierende Bild wird auf die gewünschte Projektionsfläche platziert und angepasst, um sicherzustellen, dass ein unverzerrtes Endbild an den Projektor gesendet wird. Zusätzlich wurde in der Unity3D-Engine ein Augmented-Reality-Computerspiel entwickelt, welches die Fähigkeiten der Pipeline präsentieren, sowie ein unterhaltsames Spielerlebnis bieten soll.

Abstract

Augmented reality applications are becoming increasingly commonplace in a variety of fields. However, the use of augmented reality in an academic context is limited. In this thesis, we developed a tool which allows researchers to superimpose virtual objects onto their experiment setups. For this purpose, we have implemented an easily adaptable tracking and projection pipeline, consisting of three parts: (1) off-axis perspective projection, (2) calibration, and (3) keystone correction. This pipeline uses tracking data to compute the off-axis perspective projection, which creates a 3D illusion out of a flat surface. The resulting image is then aligned with the desired projection surface and skewed to ensure a non-distorted final image is sent to the projector. Additionally, an augmented reality computer game was developed in the Unity3D engine, to showcase the pipeline and provide a fun game-play experience.

Acknowledgements

I want to thank all the people who supported me in any way during this thesis. A big thank-you goes to Prof. Dr. Chat Wacharamanotham, for allowing me to work on such a fun project and for the valuable feedback provided during the implementation and writing of this thesis. Furthermore, I would like to thank all the people who tested the game in different states.

A last thank-you goes to my family and friends who helped me to occasionally clear my mind and return to work motivated.

Chapter 1

Introduction

> "I'm excited about augmented reality because unlike virtual reality, which closes the world out, AR allows individuals to be present in the world but hopefully allows an improvement on what's happening presently."
>
> - Tim Cook [2018]

The possibilities and opportunities provided by augmented reality (AR) are becoming increasingly evident as technological advancements are made. AR involves the superimposing of virtual objects onto the user's view of the real world, effectively augmenting our current reality [Azuma, 1997]. AR systems are already in use in a variety of fields, ranging from training apps for neurorehabilitation [Ma et al., 2014] to aiding workers doing manual labor by displaying instructions onto their workspace [Boeing, 2018]. The future of AR technology seems bright, as large corporations like Apple [Apple Inc., 2018] and Google [Google, 2018] continue to invest resources into their own advanced augmented reality systems and applications. Additionally, human interaction with these novel systems will likely become an important research topic in the future [Bacca et al., 2014].

The potential use of AR in academic studies

The majority of augmented reality applications currently available are video game-related [Augmented Reality Games, 2019]. The entertainment value of such video games and their use to show off this emerging technology should not be undervalued. However, augmented reality systems could also be a valuable addition to the academic research toolkit. The use of AR in studies could potentially help reduce

the cost associated with acquiring or constructing one-off objects for an experimental setup. Some of these objects could alternatively be created digitally and superimposed onto the scene with a projector.

Goal 1: Create an augmented reality tracking and projection pipeline

Based on the current research, this thesis aims to address the issue of a missing tool which allows researchers to use augmented reality in their experiments. Thus, the first goal of this thesis is to develop an augmented reality tracking and projection pipeline, which can be easily adapted without changing the underlying programming code. This pipeline involves the use of an optical tracking system to track the position of both the user and the projection surface. The collected tracking data is then used to manipulate a virtual object to create the illusion of the object being 3D. The resulting image is then projected onto the scene.

Goal 2: Develop an augmented reality video game

The second goal is to develop an augmented reality game, which makes use of the tracking and projection pipeline. This game should function as a showcase for the pipeline, as well as provide a fun gameplay experience.

To address these goals, the paper is structured as follows. To start with, an overview of the related work is given in chapter 2. In the following chapter, the hardware and software required to develop both the pipeline and the game are presented. Subsequently, in chapter 4, the function and implementation of the tracking and projection pipeline are described. The augmented reality game, including its development, structure, and systems, is described in chapter 5. In the closing chapter, the conclusion is drawn and the future work is discussed.

Chapter 2

Related work

The goal of this thesis is to create a tracking and projection pipeline, as well as an accompanying augmented reality computer game. Therefore, the literature research consists of three main focus points. First, the focus is placed on existing work done with optical tracking systems, as the implementation requires such a system. Second, existing augmented and virtual reality applications are identified while detailing the challenges their implementations provided. Lastly, the focus is set on the use of motion control in games, due to the developed game using a motion-based control scheme.

2.1 Optical tracking

Optical tracking is defined as a "3D localization technology which is based on monitoring a defined measurement space using two or more cameras" [PS-Tech, 2018]. In this section, literature describing work done with optical tracking systems is presented. An emphasis is placed on how these systems function and their various use-cases.

Blob detection method

In their work, Ribo, Pinz & Fuhrmann [2001] present a stereo vision and marker-based tracking system for augmented and virtual reality applications. They provide a detailed analysis of the technical requirements for their system, focusing on accuracy, performance, and robustness. To meet these requirements, specific hardware recommendations are given. Additionally, Ribo et al. [2001] explain the method

used to extract the marker positions from the images recorded by the infrared tracking cameras. This method, called "Blob Detection", first analyzes the images for bright white spots caused by infrared light being reflected off of the markers. In a second step, the roundness of the found white spots is used to determine if a given spot should be categorized as a marker or if the spot is due to infrared interference. A basic understanding of this method was required to identify the causes of the tracking issues encountered during the implementation of the pipeline.

Importance of accurate tracking

De Amici, Sanna, Lamberti & Pralio [2010] point out the high cost of most commercially available tracking solutions. They show that acceptable tracking results can be achieved without a significant financial investment. For this purpose, they developed a low-cost tracking system, which can be constructed for under a thousand US dollars using refurbished Nintendo Wii remotes. De Amici et al. [2010] emphasize the importance of smooth tracking if users are required to interact with the application. They declare that an accurate and low-latency system is necessary to ensure the desired smooth tracking. Drawing on these insights in relation to the practical work in this thesis, the tracking system used for the pipeline and augmented reality game was configured to provide the most accurate tracking data possible.

Minimizing infrared interference

A look at a use-case for tracking systems is provided by Guerra-Filho [2005]. In his work, he examines how tracking systems are used in the context of motion capture for the film industry. Motion capture enables an actor's movements to be tracked and later transferred to a digital character. Similar to Ribo et al. [2001] the topics of marker detection and infrared interference are examined. To combat the tracking issues caused by infrared interference, special motion capture suits are constructed out of a unique material which does not reflect infrared light. The insights provided by Guerra-Filho's work [2005] were fundamental in choosing tracking objects with suitable surface properties to minimize infrared interference.

2.2 Augmented and virtual reality

In this section, different augmented and virtual reality applications, and the challenges associated with their implementation, are presented. Many virtual reality systems have similar technical requirements and issues as augmented reality systems, as was ascertained

in the extensive research on recent discoveries and findings in these fields. Thus, research from both fields is presented below, though for practical purposes a focus was placed on augmented reality applications.

Generalized perspective projection

Cruz-Neira et al. [1992] created the cave automatic virtual environment (CAVE) interface. The CAVE is a cube-like space with displays as walls. The user is allowed to move around freely inside the space. The displayed images are adjusted continuously by the system to fit the current perspective of the user, creating the illusion of being inside a virtual environment. To achieve this illusion, a mathematical algorithm known as a perspective projection is required. The calculations used in this algorithm were later generalized by Kooima [2009]. His work enables the perspective projection algorithm to be used in many different applications and games. The generalized perspective projection was used as a basis for our implementation of the off-axis perspective projection described in chapter 4.1 "Off-axis perspective projection".

Need for a calibration solution

In their research, Mine et al. [2012] describe how they are integrating augmented reality applications into Disney theme park attractions. They emphasize the need for a robust calibration of both the tracking system itself, as well as the application using the tracking data. The specific context of their work presents additional constraints, such as only being able to recalibrate the system during nighttime when the park is closed. Nevertheless, they make a general recommendation that a quick and straightforward calibration work-flow should exist in every application. This recommendation by Mine et al. [2012] lead to the implementation of a visual calibration tool for the pipeline.

Perceptual characteristics

In their work, Kruijff, Swan & Feiner [2010] discuss common perceptual issues found in augmented reality applications and how they can be fixed. They present their findings, showing that the color fidelity and brightness of the image positively affect the immersion of the user. Additionally, Kruijff et al. [2010] show the lack of a direct relationship between the image resolution and the user's ability to judge the depth of a virtual object. Based on their findings, we decided to focus on a vibrant color palette, rather than outright graphical fidelity while developing the augmented reality game .

2.3 Motion control in games

This section describes literature related to motion control in games, as well as how users perceive this novel control scheme. The specifics of the goals for this thesis make the following knowledge necessary for the implementation of the game.

Evolution of game controllers

In his work, Cummings [2007] presents an evolution of the game controller. He describes how the rise of specific game genres popularized different input methods. Motion control, which has been around since the arcade era with the "light gun" [Kent, 2010], was greatly popularized with the release of the Nintendo Wii and later the Wii U [Nintendo, 2018b]. The Wii made use of a proprietary remote with multiple accelerometers inside, allowing players to interact with their games through movement alone. Cummings [2007] states that such complex motion controllers require games to be developed from the ground up with this control scheme in mind. He emphasizes that adapting an existing game to include motion controls will rarely lead to satisfactory results. Thus, for our implementation of the augmented reality game, motion control was used in all phases of development.

Design recommendations

Bucolo, Billinghurst & Sickinger [2005] examined user experiences with mobile game interfaces. Their study used a mobile game which required players to navigate a ball through a maze using different input methods. One of these input methods was motion control, allowing users to tilt their phones to control the ball. Based on the results of their study, Bucolo et al. [2005] recommend that overly precise movements should be avoided. This recommendation is due to motion controllers rarely allowing for exact control due to input latency.

Intuitive game mechanics

The importance of using an intuitive game mechanic in motion control-based video games is discussed by Champy [2007]. He recommends using a motion needed for everyday real-world tasks as the basis for the motion required to control the game, hereby eliminating the need for the player to learn any new skills. The recommendations by Bucolo et al. [2005] and Champy [2007] were considered during the prototyping phase of the game.

The findings in recent research presented in this chapter offer a strong knowledge basis, which proved important in the practical implementation further detailed in the following chapters.

Chapter 3

Hardware and software setup

Specialized hardware and software are required to create an augmented reality tracking and projection pipeline, and accompanying computer game. For an augmented reality game to maintain the desired illusion that a virtual object is real, the object should be viewable from different angles. Thus, the game needs to be aware of the user's position, which requires a tracking system. The different tracking system types, as well as the specific setup of the tracking system used, are described in the following section. Furthermore, to create a complete augmented reality game, some additional game-specific hardware is required. This hardware and the related issues are described in section 3.2 "Game related hardware". The game itself is developed using a game engine. The final section of this chapter provides an evaluation of different game engines.

3.1 Tracking system

To obtain the tracking data required to create the illusion of a virtual object being real, a tracking system is needed. There are many different approaches to tracking objects in 3D space, with the available systems consisting of two main categories: (1) marker-less tracking systems or (2) marker-based tracking systems.

Marker-less tracking systems

Marker-less systems, such as the ones developed by Leap Motion [Leap Motion, 2018] and Microsoft Kinect [Microsoft, 2018], use in-

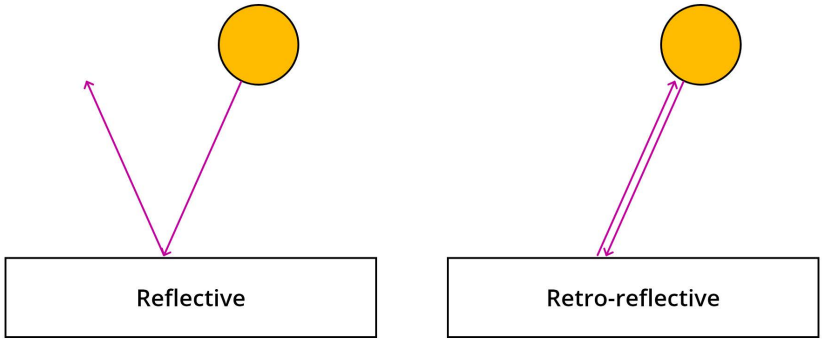

Figure 3.1: Reflective surfaces reflect light at an angle based on the light ray's entrance, whereas retro-reflective surfaces reflect light to the source. Such retro-reflective properties are used for tracking markers.

frared cameras to generate depth maps of their environment. A depth map is an image containing information relating to how far away objects in the camera's field of view are. Such marker-less systems are relatively inexpensive, yet can only provide low accuracy position data when used in large tracking volumes.

Marker-based tracking systems

Marker-based tracking systems, like the OptiTrack system by NaturalPoint [NaturalPoint, 2018b], utilize infrared cameras to track the position of individual markers attached to objects. These markers are coated with a special retro-reflective paint. The need for such a coating is due to the way light behaves when it reflects off a surface. Normal surfaces reflect light at an angle based on the angle of the entering light ray, whereas retro-reflective surfaces always reflect light to the source [Stamm, 1973], as shown in Figure 3.1. Thus, using a retro-reflective coating on the tracking markers allows the infrared light, emitted by an LED ring around the camera, to be directed back towards the lens of the camera. By setting up multiple cameras, each with a different viewpoint, a complex algorithm can calculate a very accurate position of a given marker in 3D space.

Accurate position data is required for the augmented reality game and pipeline to function correctly. Thus, a marker-based tracking system was used. Specifically, the OptiTrack system by NaturalPoint shown in Figure 3.2 was chosen, due to its availability at the ZPAC lab.

Figure 3.2: The OptiTrack system setup in the ZAPC Lab at the University of Zurich, showing the positions of the cameras used for tracking the retro-reflective markers.

Figure 3.3: The Flex 13 camera used by the OptiTrack system, showing the illuminated LED ring around the lens which emits the infrared light used to track the markers.

OptiTrack setup

The setup of the OptiTrack system consisted of four Flex 13 cameras [NaturalPoint, 2018a], shown in Figure 3.3. These four cameras were placed on tripods at different corners of the desired tracking volume. This non-permanent mounting solution allows for the cameras to be moved easily, should the tracking volume change in the future. Additionally, being able to mount the cameras fairly high, minimized infrared interference. Eliminating all possible infrared interference is important, as even little interference can greatly reduce the tracking accuracy of the overall system.

The OptiTrack system provides the necessary tracking data. However, some additional hardware is required for the augmented reality pipeline and game to function. This hardware is described in the following section.

3.2 Game related hardware

In addition to the OptiTrack tracking system, some additional hardware is required to create the augmented reality game. This game-related hardware consists of (1) a projector, (2) a table, and (3) a tracked object, which acts as the controller for the game. The projector was mounted on a wooden beam above the tracking volume, as shown in Figure 3.2, and was used to display the game. The table acted as a second projection surface and was placed in the middle of the tracking volume, below the projector. White cloth was draped over the table, to ensure the colors of the projected image were not distorted.

Requirements for the object used to control the game

The third hardware item required for the game to function correctly is the tracked object used to control the game. This object also functions as the primary projection surface, to showcase the developed tracking and projection pipeline. Thus, three factors had to be considered when choosing the object:

- **Weight** - The object had to be light enough to be held for an extended time while playing the game.
- **Size** - The object had to be large enough to have the game projected onto it while remaining maneuverable.
- **Rigidity** - The object was not allowed to flex while it was moved, as this would distort the image projected onto its surface.

Based off of these three factors, a box lid was chosen, as it possessed the required characteristics and was available in the ZPAC lab. The box lid was fitted with retro-reflective markers, as shown in 3.4, which allowed it to be tracked by the OptiTrack system. Tracking the box lid was initially plagued with issues, which are described in the following subsection.

3.2.1 Issues

The tracking issues experienced with the box lid needed to be resolved to achieve the required tracking accuracy. Through trial and error, the issues were deemed to be caused by the following two factors: (1)

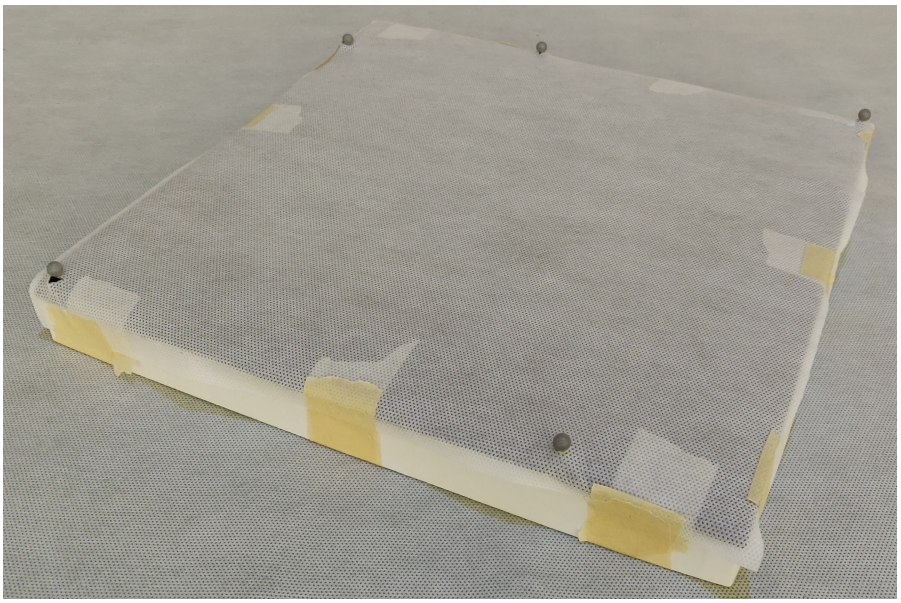

Figure 3.4: The box lid used as a projection surface and controller for the augmented reality game, with the tracking markers glued to it.

the arrangement of the markers on the box lid and (2) the reflective material out of which the box lid is constructed.

Issue 1: Marker arrangement

The arrangement of the markers on a tracked object is used by OptiTrack Motive software to determine the orientation of the object. The software computes the distances between the individual markers, which are then compared to determine the pitch, roll, and yaw of the object [Koch, 2016]. Thus, the distances between the markers should not be equal, as this could result in the software computing a constant incorrect orientation or rapidly switching between possible orientations.

The initial marker arrangement on the surface of the box lid had equal distances between markers. The arrangement consisted of four markers, with a marker in each corner, as shown on the left in Figure 3.5. This symmetrical arrangement resulted in the Motive software not being able to reliably determine if the box lid was rotated 0, 90, 180, or 270 degrees, which produced unusable tracking data.

To fix the tracking issues produced by symmetrical marker arrangements, a unique and non-congruent arrangement is recommended by NaturalPoint [2018a]. We implemented such an arrangement by re-

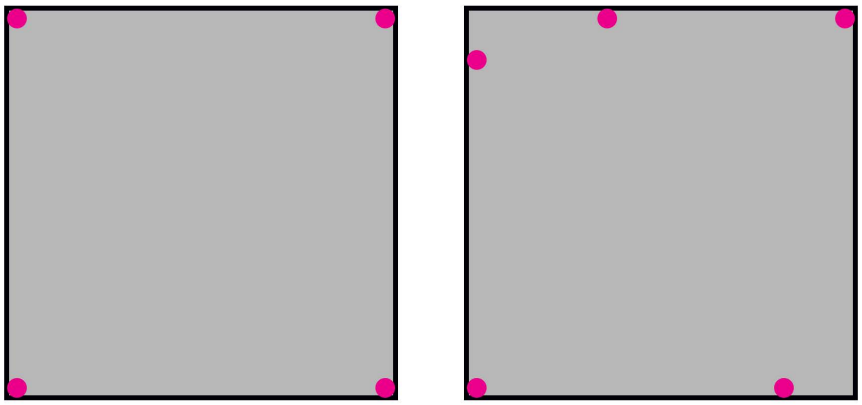

Figure 3.5: Left: The initial marker placement with four markers, which caused tracking issues; Right: The final marker placement with five markers, which fixed the tracking issues.

arranging the markers and adding a fifth marker. The new marker arrangement is shown on the right in Figure 3.5.

Issue 2: Reflective surface material

Having solved the first factor causing tracking issues, the second factor was examined: the reflective material out of which the box lid was constructed. This material caused a large amount of infrared interference, as its reflective properties meant that it scatters the infrared light emitted by the tracking cameras. The interference resulted in the Motive software not being able to reliably extract the position of the retro-reflective markers from the images recorded by the tracking cameras, shown in Figure 3.6. Thus, the software could not reliably track the box lid.

To eliminate the infrared interference, the reflective material of the box lid needed to be covered. We used a black colored felt fabric, as it has a similar texture to the fabric used to create motion capture suits, which are specifically designed not to reflect any infrared light [Sullivan et al., 2015]. The use of such fabric eliminated all infrared interference, though the dark color of the felt made the image projected onto the box lid difficult to see. To make the image more visible, we covered the felt with thin white cloth. The resulting box lid allowed the player to clearly see the projected image, while still minimizing the infrared interference. These alterations resulted in the Motive software being able to reliably extract the positions of the markers, as shown in Figure 3.6. Thus, the tracking issues with the box lid were resolved,

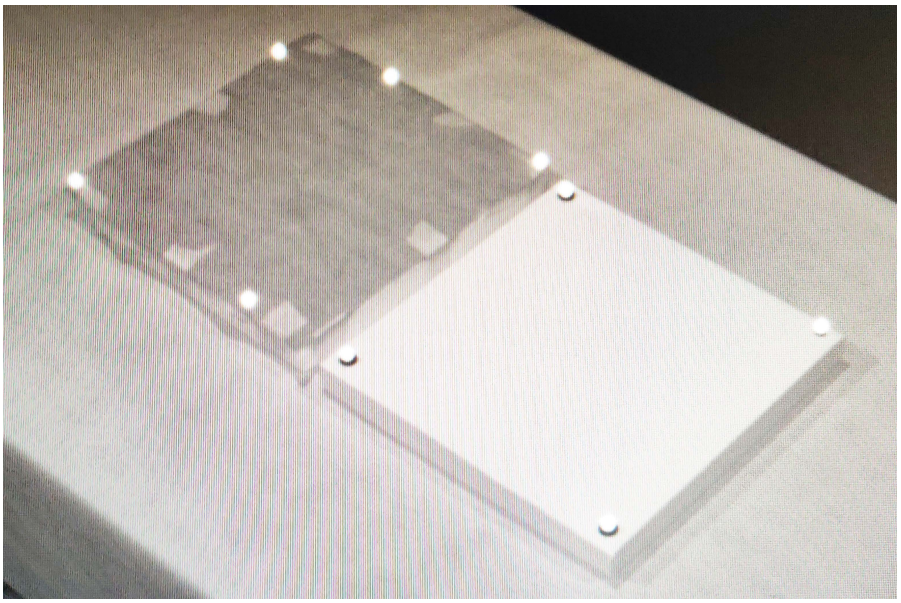

Figure 3.6: Comparison between (1) the box lid covered in black felt fabric and a thin white cloth on the left, and (2) a regular white box lid on the right, as seen by the tracking cameras.

and the tracking system produced the desired accurate tracking data. This data, as well as the presented hardware, is then used by the game engine, which is described in the following section.

3.3 Game engine

The tracking and projection pipeline and the accompanying game are both developed using a game engine. A game engine is a software development tool, which abstracts the details of game-related processes, such as rendering the image or calculating physics simulations [Ward, 2008]. This abstraction allows the people making games (such as programmers, artists, designers, and scripters) to focus on creating a fun and meaningful game-play experience, instead of having to implement basic systems from scratch for every new game [Lewis and Jacobson, 2002].

Visual editor

As only a small amount of people who work on video game are usually programmers, most popular game engines include a visual edi-

tor. A visual editor allows for large parts of the game and pipeline to be constructed and expanded without requiring programming knowledge. As the design goal of the tracking and projection pipeline is to allow future users to change its parameters without changing the underlying code, a good visual editor is especially important.

Engine comparison

The most popular game engines which include a good visual editor [TNW, 2016] and their respective characteristics are as follows:

- **Unreal Engine 4** (UE4), the newest iteration of the Unreal Engine, is developed by the American software and game studio Epic Games [Epic Games, 2018] and is free for non-commercial use. UE4 is known for its photo-realistic rendering and particle systems, which enable developers to create visually impressive games [Karis and Epic Games, 2013]. However, to use these systems to their full potential, a significant learning effort is required [Thinkwik, 2018]. Thus, the Unreal Engine 4 is considered poorly suited for beginner game developers.
- The **CryEngine** is developed by the German studio Crytek [Crytek, 2018] and requires a monthly fee payment, even for non-commercial use. Thus, given the financial investment required, only large studios can afford to develop a game with the CryEngine. Its popularity can be attributed to the versatility of its architecture, which allows it to be used for games of many different genres.
- **Unity3D** is developed by Unity Technologies [Unity Technologies, 2018c] and is the most popular engine currently available [Thinkwik, 2018]. This popularity can be attributed to it being free for non-commercial use, as well as its shallow learning curve for novice game developers. Unity uses easy-to-learn C# as its scripting language instead of the commonly used C++ [Xie, 2012]. Additionally, the architecture utilizes an entity component system of gameobjects, which was constructed to allow most adjustments to be made directly in the visual editor [Unity Technologies, 2018a].

Based on these characteristics and our prior experience working with different game engines, the Unity3D engine was chosen for the implementation of the tracking and projection pipeline, as well as for the augmented reality game. The shallow learning curve for novice game developers and the emphasis on a good visual editor would allow

the pipeline to be adjusted with minimal programming knowledge required.

OptiTrack plug-in for Unity

Additionally, due to the popularity of the Unity Engine, an OptiTrack plug-in exists for it [NaturalPoint, 2018b]. This plug-in allows for easy integration of real-time tracking data directly into Unity. The use of this plug-in allowed us to focus on the development of the pipeline and game, rather than on the complex task of integrating the tracking data into Unity.

To summarize, the chosen hardware and software consists of the OptiTrack tracking system, the game related hardware, and the Unity game engine. The development of the tracking and projection pipeline, as well as the accompanying game, was enabled by using this hardware and software. The pipeline and its implementation are described in the following chapter.

Chapter 4

Tracking and Projection Pipeline

An augmented reality game requires a specific sequence of processes to correctly track the user's movements and display the desired image. This sequence is called the tracking and projecting pipeline. The pipeline uses the OptiTrack system and Unity, which are introduced in the previous chapter. As the setup at ZPAC may change in the future, it is essential that the pipeline can easily be adjusted. Therefore, the design goal of the pipeline is to allow future users to change the majority of parameters without needing to change the underlying code. To achieve this, the developer should be able to make the changes in Unity's visual editor, as it is an easy to understand visual interface. The interface uses a hierarchical structure, which makes it is easy to distinguish between the different parts of the pipeline.

Pipeline overview

The pipeline consists of the three connected parts shown in Figure 4.1: off-axis perspective projection, calibration, and keystone correction. The off-axis perspective projection is used to create a 3D illusion out of a flat screen, by using tracking data and mathematical calculations to transform the image. The next section describes this technique and its specific implementation in Unity. The calibration part uses the image generated by the perspective projection and aligns it to match up with the projection surface. The implementation of the calibration solution is described in section 4.2 "Calibration". The third part, the keystone correction, skews the final image to ensure a non-distorted image is sent to the projector. The technique used to achieve this keystone correction in Unity is described in the final section of this chapter.

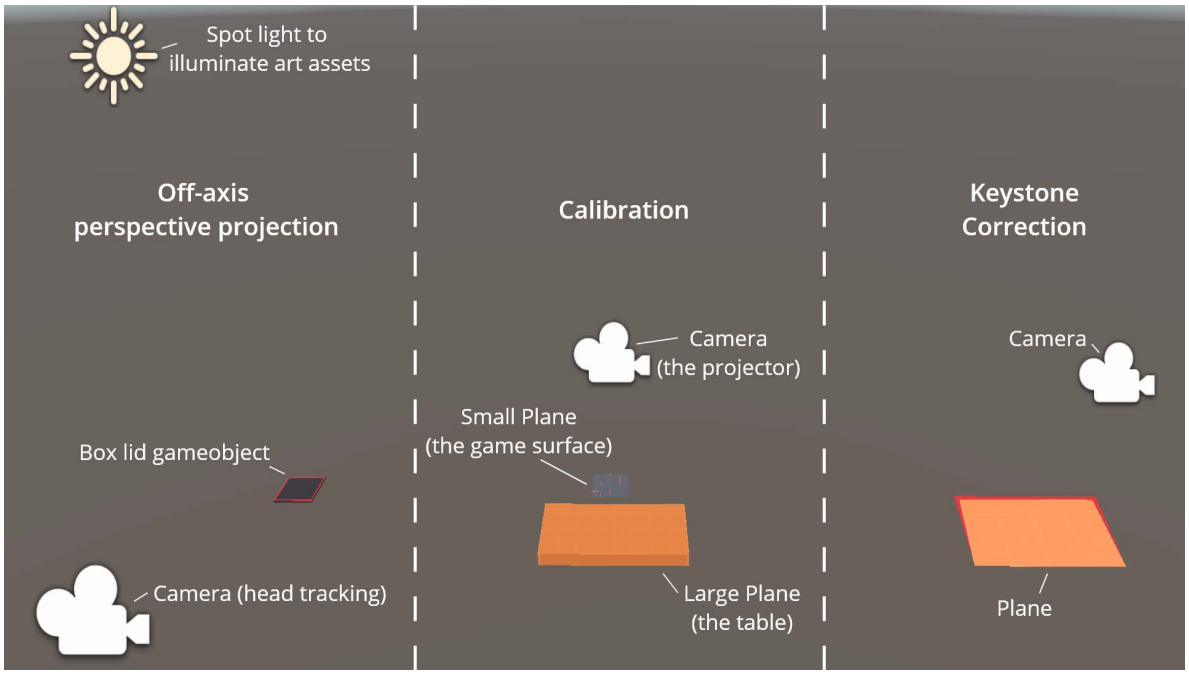

Figure 4.1: An overview of the whole pipeline showing the three parts in the scene view of the Unity3D editor. The names of the visible gameobjects are annotated, to provide a visual guide as to which gameobjects belong to what part.

4.1 Off-axis perspective projection

Foreshortening effect

The first step of the pipeline is the off-axis perspective projection, which creates a perspective illusion, were the player sees a 3D object when looking at a flat display or projection surface [Başar et al., 2009]. To avoid confusion between the mathematical projection and the projection done by a projector, we will refer to projection surfaces as screens in this section, as the principles used apply to both. The perspective projection makes use of the foreshortening effect, which refers to how big humans perceive objects at different distances [Kooima, 2009]. An object far away appears smaller than an object up close, even if both are actually the same size. The foreshortening effect generated by the perspective projection is commonly used in virtual reality systems and environments [Heuser, 2008]. An example of such a virtual reality system is the cave automatic virtual environment (CAVE) by Cruz-Neira et al. [1992] as described in chapter 2.2 “Augmented and virtual reality”. The perspective projection used by the CAVE system is a traditional perspective projection, were the screens position is fixed in 3D space.

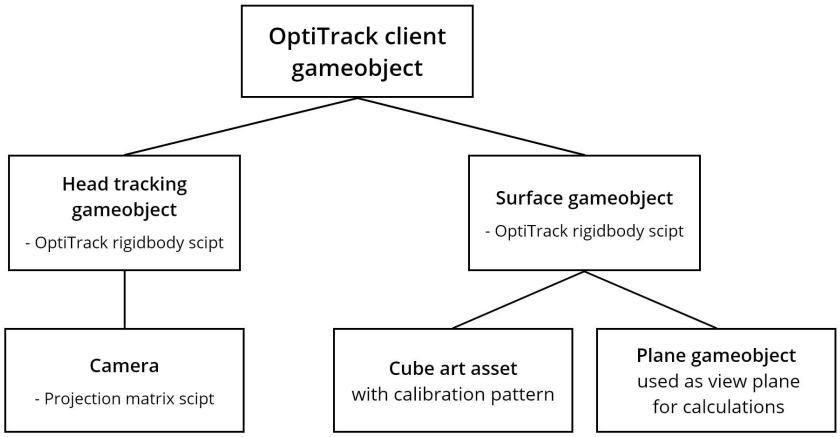

Figure 4.2: Unity setup used for perspective projection, showing the hierarchical structure of the gameobjects and their attached scripts.

On-axis vs. off-axis perspective projection

More advanced virtual reality systems allow for the screen position to change. An example of such a system is the Oculus Rift [Facebook Technologies, 2018], which uses a head-mounted display to track the user's head position in 3D space. The tracking data is subsequently used to compute the perspective projection from the user's viewpoint. The possibility of the screen moving requires the screen position to be implemented as a variable, rather than a fixed value, for the perspective projection calculations. Due to both the screen and the user's head position being able to move in our desired application, an off-axis perspective projection implementation was used. A simpler on-axis perspective projection would only work if the player's head stayed in line with the symmetry axis of the screen [Kooima, 2009]. This approach would be too limiting for the desired application, as the player should be able to move more freely, to make use of the tracking systems capabilities. An off-axis perspective projection allows for arbitrary head movements and screen positions.

Structure in Unity

For the off-axis perspective projection calculations to be executed correctly, a specific structure in Unity is needed. We set up the structure through a tree of Unity gameobjects as shown in Figure 4.2. At the root of the tree is the OptiTrack client gameobject. This gameobject has two children: (1) the surface gameobject and (2) the head tracking gameobject, which both have an OptiTrack rigidbody script attached. This rigidbody script allows Unity to use the coordinates of

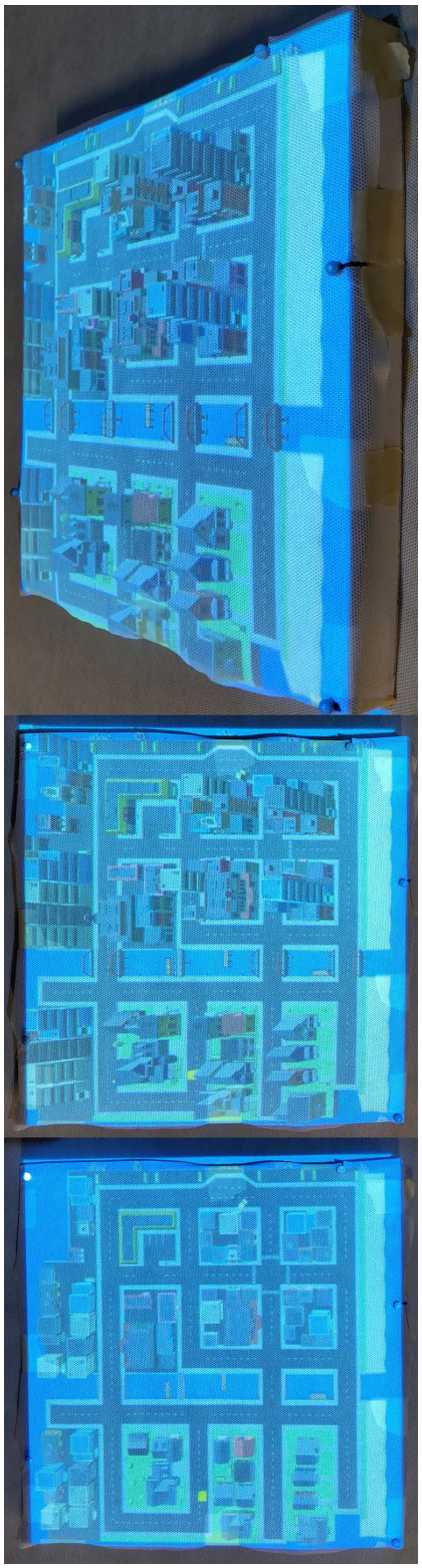

Figure 4.3: Left: No perspective projection; Middle: Perspective projection applied; Right: Perspective projection viewed from the tracked head position

specific rigid bodies streamed from the Motive software, to control the gameobject to which the script is attached [NaturalPoint, 2018b]. The head tracking game object has a camera child object, which has the projection matrix script attached to it. The projection matrix script is used to calculate the perspective projection from the viewpoint of the camera. The surface gameobject has two children: a cube art asset with a calibration pattern on it, and a plane gameobject, which is used in the calculations in the projection matrix script.

Projection matrix script

The projection matrix script computes the off-axis perspective projection and thereby creates the 3D illusion on a flat screen, as can be seen in Figure 4.3. The mathematical formulas needed are described by Robert Kooima [2009]. For their specific implementation in Unity the "Projection for Virtual Reality" Unity Wikibook [2017] was used as a guide. Together they create the basis for our implementation of the projection matrix script.

Our projection matrix script requires three inputs: (1) the camera position, (2) the screen plane and (3) the screen dimension. The screen plane is defined as the screen's surface, in our case, this is the surface of the box lid. All art assets on the screen plane will have the perspective projection applied to them. By adjusting the z-position of the art assets relative to the screen plane, different types of perspective illusions can be achieved. Using the island art asset from the final game (which can be seen in Figure 4.3) as an example, if the screen plane is at the height of the roads, anything above this height (such as buildings and trees) will appear to come out of the box lid. If the screen plane is positioned at the height of the building rooftops, it creates the illusion of depth, as if one could look into the box lid. However, it is vital that the position and orientation of the screen plane match the position and orientation of the real world screen. Otherwise, the calculated perspective will be wrong. Thus, the plane gameobject is used for the screen plane, as its tracked position and orientation inside of Unity reflect the real world position and orientation of the box lid used as the projection surface. The size of the box lid defines the screen dimensions, which the plane gameobject is scaled to match. By using the dimensions of a Unity gameobject instead of hard-coding them in the projection matrix script, the screen can easily be changed by resizing the gameobject in the Unity GUI without needing to change the underlying code.

The projection matrix script first needs to gather the required values. Thus, the coordinates of three corners of the plane game object are

read and assigned to variables. This is done as follows:

```
1 Vector3 pa = plane.transform.TransformPoint(new Vector3
    (-5.0f, 0.0f, -5.0f)); // lower left corner in Unity
     world coordinates
2 Vector3 pb = plane.transform.TransformPoint(new Vector3
    (5.0f, 0.0f, -5.0f)); // lower right corner
3 Vector3 pc = plane.transform.TransformPoint(new Vector3
    (-5.0f, 0.0f, 5.0f)); // upper left corner
4 Vector3 pe = transform.position; // eye position
```

Only three corner positions (p_a, p_b and p_c) are needed, as the location of the fourth one is implicit due to the rectangular shape of the screen. Additionally the camera position is read. We called the camera position the eye position p_e in our implementation, as the camera "sees" the view plane and to make it easier to distinguish from the camera gameobject in the code.

Screen-local-axes

Having the required positions, a way for describing points relative to the screen is needed. To achieve this, we need to calculate the screen-local-axes v_u, v_r and v_n. They are calculated as follows:

$$v_r = \frac{p_b - p_a}{||p_b - p_a||}$$

$$v_u = \frac{p_c - p_a}{||p_c - p_a||}$$

$$v_n = \frac{v_r \times v_u}{||v_r \times v_u||}$$

Together they are the orthonormal basis of the screen, as can be seen in Figure 4.4. The vector v_u points up, v_r points to the right, and v_n points out of the screen. v_n is also the vector normal to the screen.

Frustum extents

Now that we can describe all points relative to the screen, we want to construct a perspective projection matrix which can later be used to create the desired foreshortening effect. To create this projection matrix some additional values are needed: (1) the frustum extents and (2) the distance to the clipping planes. The frustum extents are the distances from the screen-space-origin to the edges of the screen. They are defined l (left), r (right), b (bottom) and t (top) as shown in Figure 4.5. They are calculated using the vectors from the eye position to the respective screen corners, which are shown in Figure 4.6. The

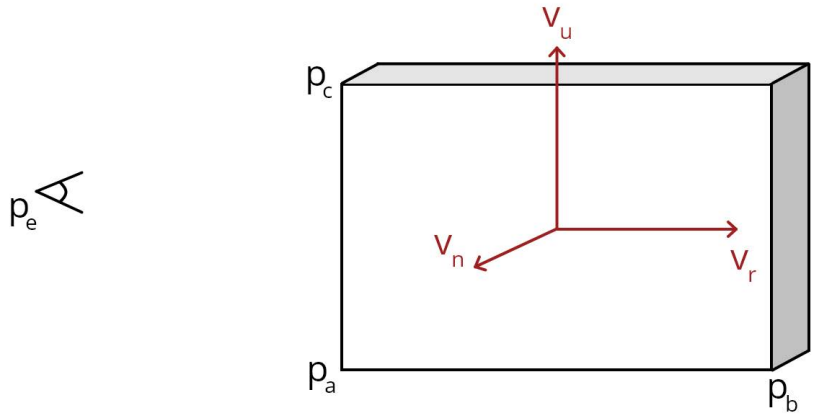

Figure 4.4: The eye position and corner positions of the view plane, with the screen-local-axes emanating from the middle of the screen. Using these axes all points can be described relative to the screen.

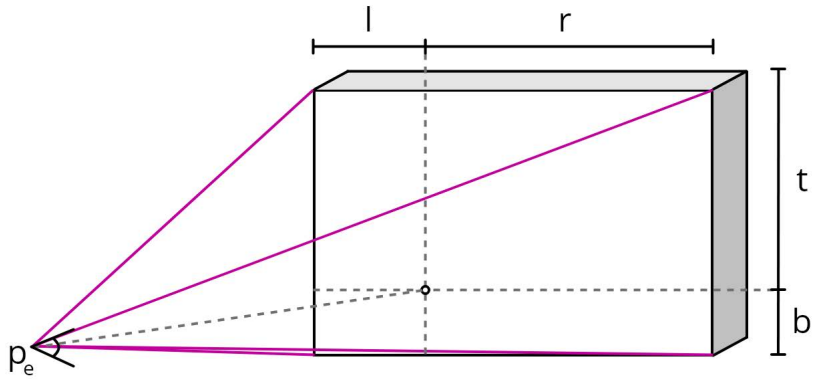

Figure 4.5: The view frustum from the eye position to the corners of the screen, with the frustum extents shown. The frustum extents are needed to create the perspective projection matrix.

calculation is as follows:

$$l = (v_r * v_a)n/d \qquad r = (v_r * v_b)n/d$$
$$b = (v_u * v_a)n/d \qquad t = (v_u * v_c)n/d$$

The second set of values needed for the perspective projection matrix

Clipping planes

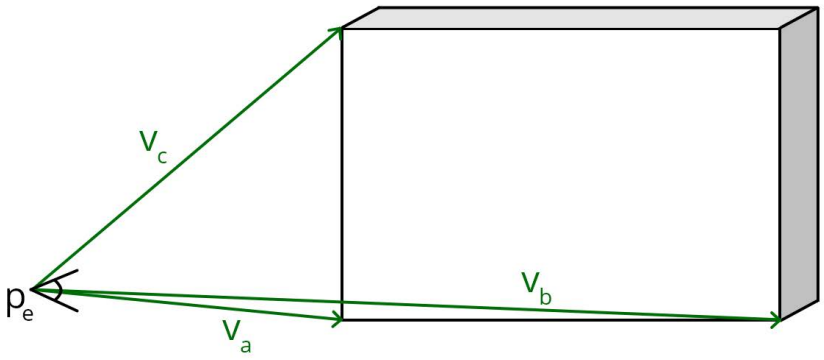

Figure 4.6: The vectors from the eye position to the screen corners.

are the distances to the near and far clipping planes, which are defined as n and f. Clipping planes mark the distance range in which the camera renders vertices [RealIllusion, 2018]. Everything before the near clipping plane and everything after the far clipping plane will not show up in the final rendered image. In our implementation, these values are set inside of Unity and read by the script, to allow easy editing of parameters in the visual editor.

We can now construct the perspective projection matrix:

$$P = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Division by the z component

The desired foreshortening effect requires the x and y values of every vertex to be divided with the value of its z component. As Kooima [2009] stated, this is easiest to do by making use of the fourth component of homogeneous three-dimensional vectors. When such a vector is multiplied by the -1 in the fourth row of P, the negative z value is moved to the w-component of the resulting vector. Thus, if the result is collapsed down, the division by z implicitly occurs.

The mathematical implementation of our perspective projection matrix has two issues: (1) the view frustum cannot be rotated and (2) the eye position cannot be moved horizontally or vertically. Both of these issues need to be resolved for the off-axis perspective projection to be useful for our desired application.

Issue 1: Rotating the view frustum

To solve the issue of not being able to rotate the view frustum, we simply rotate the screen in the opposite direction by the same amount. This rotation is done via a transformation matrix M, with the screen-local-axes vectors v_r, v_u and v_n as columns:

$$M = \begin{bmatrix} v_{rx} & v_{ux} & v_{nx} & 0 \\ v_{ry} & v_{uy} & v_{ny} & 0 \\ v_{rz} & v_{uz} & v_{nz} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The transpose of M: M^T is multiplied with the perspective projection matrix P to resolve the issue of not being able to rotate the view frustum.

Issue 2: Moving the eye position

The second issue that the eye position cannot be moved horizontally or vertically, is due to the way the division by z functions. Our solution is similar in approach to the first issue: we move the screen in the opposite direction. This requires the following translation matrix T:

$$T = \begin{bmatrix} 1 & 0 & 0 & -p_{e_x} \\ 0 & 1 & 0 & -p_{e_y} \\ 0 & 0 & 1 & -p_{e_z} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Final matrix

P is multiplied with T to resolve the second issue. Therefore the final matrix P' is calculated as follows:

$$P' = PM^T T$$

The result of P' is an image with the off-axis perspective projection applied, which functions correctly regardless of the user's head position and the position of the screen.

The transformed image is the final result of the projection matrix script and also of the first part of the tracking and projecting pipeline. While the application is running, the image is rendered to a texture, called a render texture. This render texture is then used in the next part of the pipeline, the calibration.

4.2 Calibration

The second part of the tracking and projecting pipeline is the calibration, which aligns the image generated in the first part, with the real

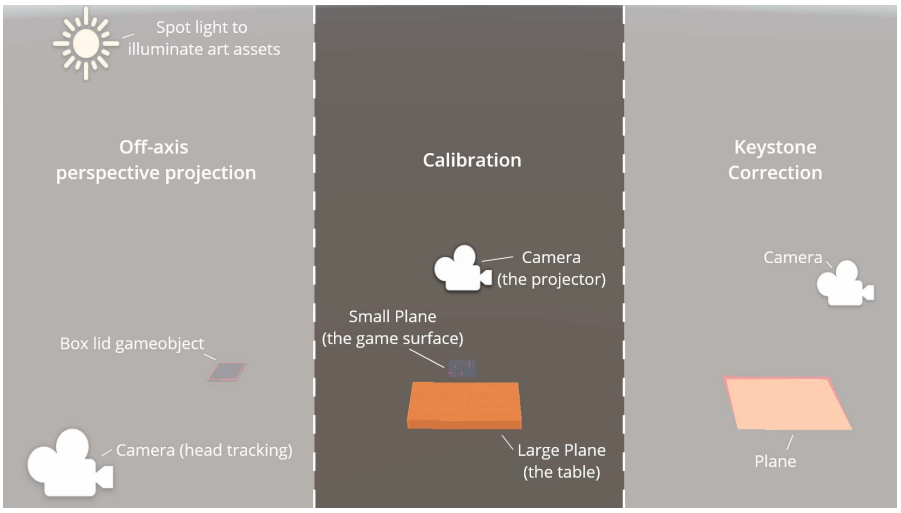

Figure 4.7: The structure of the calibration part as seen in Unity, shown in the context of the whole pipeline.

world projection surface. This alignment is needed, as in the first part of the pipeline only the relative distances between the tracked objects were considered. Thus, the virtual positions of the tracked objects need to be matched up with their real-world counterparts. As the default unit of measurement in Unity is the same as a meter in the real world [Unity Technologies, 2018g], the setup in the ZPAC lab can be easily reconstructed inside of Unity by measuring the dimensions and distances with a measuring tape.

Structure in Unity

The resulting structure inside of Unity consists of a camera and two different sized plane gameobjects, as can be seen in Figure 4.7. The camera represents the mounted projector and films the scene. The large plane gameobject represents the table, around which the tracking system is set up in the ZPAC lab. The box lid is represented by the small plane gameobject, onto which the render texture from the first part is mapped. A benefit of mapping the render texture onto another plane is that only the area on top of the box lid has the perspective projection applied. This mapping allows for non-3D user interface (UI) elements to be displayed on the rest of the available projection surface on the table. We created the large plane gameobject, representing the table first. This plane acts as the baseline for the camera's position. The position of the camera is adjusted to ensure it is the same distance above the large plane, as the mounted projector is above the table. The positions of the large plane and the camera are fixed after the initial

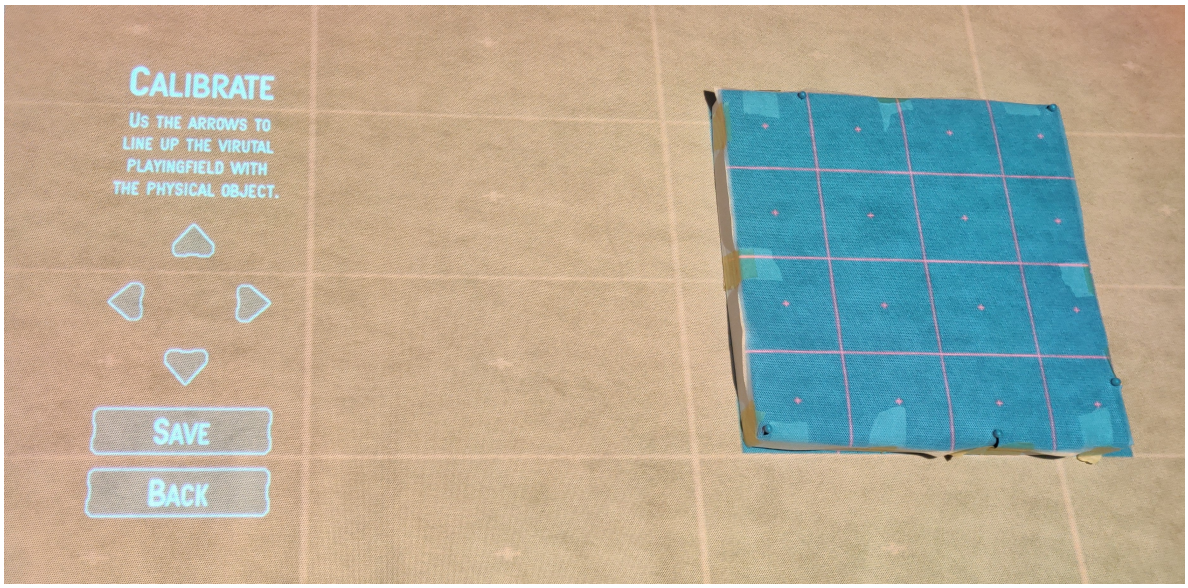

Figure 4.8: The calibration user interface for translating the tracking origin, with the arrows on the left. A good calibration is shown, with the image matching the box lid.

adjustment. However, the small plane's position is controlled by the tracking data of the box lid.

While the projector could be manually moved until the projected image of the small plane matches up with the real world box lid, this is not feasible and goes against our design goal of making an easily adaptable system. If the projector were to be moved, either by accident or if the OptiTrack calibration square is placed at a different location on the table during startup, the projected image will no longer be aligned. As this is relatively common and needs to be accounted for, a quick way to calibrate is required. Thus, a solution inside of Unity was implemented.

Translation of the tracking origin

The calibration tool we implemented makes use of the fact that the tracking origin can be moved. Moving the tracking origin also moves all of the tracked objects, namely the small plane with the render texture mapped to it. The tracking origin inside of Unity is defined by the position of the OptiTrack client gameobject. Thus, to adjust the tracking origin, the OptiTrack client gameobject is translated on the X and Y axes. The Z-axis will never need to be adjusted, as the tracking origin in Unity and the tracking origin in the Motive software cannot differ in their height.

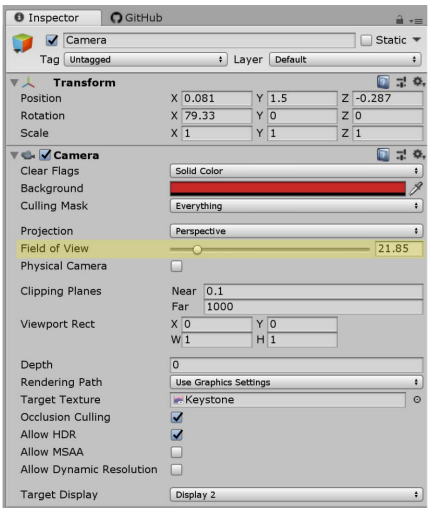

Figure 4.9: The Unity inspector tab of the camera used for adjusting camera parameters, with the field of view parameter highlighted.

Visual calibration tool

Based on the fact that the tracking origin only needs to be adjusted in four directions (up, down, left, right), the implemented visual calibration tool uses arrows, as can be seen in Figure 4.8. This tool allows users to move the tracking origin around in the X and Z axes by pressing arrow buttons until the desired result is reached. Adjustments to the position are made during runtime, so the view is updated in real time, providing immediate feedback to the user. The calibration is complete when the projected image of the small plane, with the mapped render texture, aligns with the box lid.

A complete calibration is saved for further sessions. The saving process was implemented through the Unity PlayerPrefs system, which stores the desired data in a plain-text file [Unity Technologies, 2018d]. Upon start of the application, this text file is read and the position of the OptiTrack client is adjusted automatically to the saved position. This saving process eliminates the need for calibration every time the application starts.

Camera parameters

The presented calibration solution allows for correct alignment when the box lid is placed on the table. For the calibration to remain intact when the box lid is picked up, the camera parameters need to be correctly adjusted during the initial setup. These adjustments are made in the camera gameobject's inspector tab shown in Figure 4.9. The field of view parameter is especially important, as if it is set correctly,

the small plane will always match up with the dimensions of the box lid. To determine the correct field of view parameter for the camera, the projector's field of view (found in the projector's manual) is used as a baseline and adjusted based on the output. If the projected image of the small plane becomes too large to fit on the box lid when the box lid is moved closer to the projector, the camera's field of view needs to be increased. If the projected image does not take up the available space on the box lid, the field of view needs to be decreased. The parameter is set correctly when the projected image always matches the dimensions of the box lid, regardless of how close the box lid is held to the projector. When the box lid is tilted, the camera distorts the small plane based on its perspective, which allows it to remain aligned.

If the calibration solution is used successfully and the parameters are set correctly, the calibration will remain intact regardless of how the box lid is moved. At the end of the second part of the pipeline, the image is positioned at the correct location. This image is again rendered to a texture, which is then used in the final part of the pipeline, described in the following section.

4.3 Keystone correction

The third and final part of the tracking and rendering pipeline is the keystone correction, which skews the image generated in the second part, to ensure that a non-distorted final image is sent the projector. This part is required as the projector is not positioned precisely above the middle of the table, as illustrated in Figure 4.10. Therefore, the camera used to generate the image in the calibration part is offset as well.

Keystone effect

Due to the slight offset from the middle, the projector is not perpendicular to the projection surface, which results in a distorted image. This distortion of the image, with the lower part of the image being wider than the top part, creates a trapezoid shape as shown in Figure 4.11. This distortion is known as the keystone effect [Yadav and Agrawal, 2013]. The amount of distortion can be approximated using the following formula:

$$\frac{cos(\epsilon - \frac{\alpha}{2})}{cos(\epsilon + \frac{\alpha}{2})}$$

ϵ is defined as the angle between the screen and the central light ray

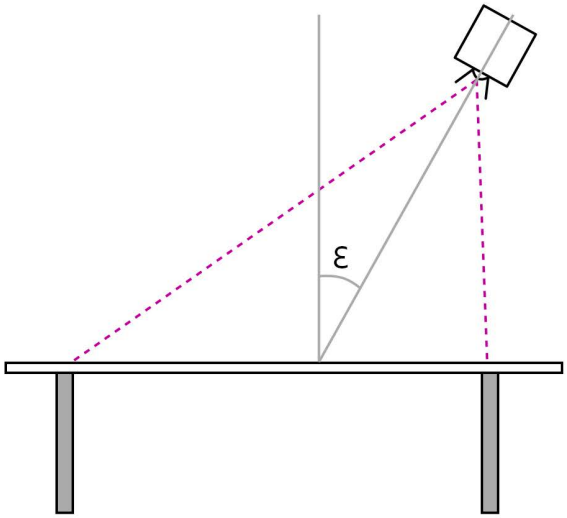

Figure 4.10: The offset of the mounted projector, which causes the keystone effect.

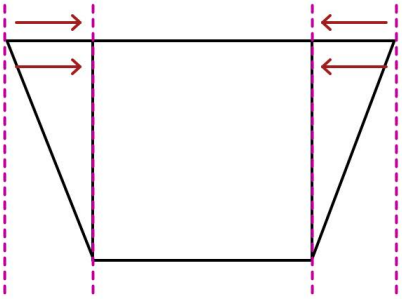

Figure 4.11: The trapezoid shape of the image when the keystone effect is present. The arrows show the skewing done by the keystone correction to create a non-distorted image.

of the projector, and α defines the width of the focus, which is determined by the projector's lens. The formula shows, that with an increasing angle, the amount of distortion will increase as well.

The distortion caused by the keystone effect is a common issue found in many commercial projection setups. This is due to mounting constraints. An example for this, is mounting a projector to the ceiling at an angle and having it project onto a nearby wall. To fix the keystone

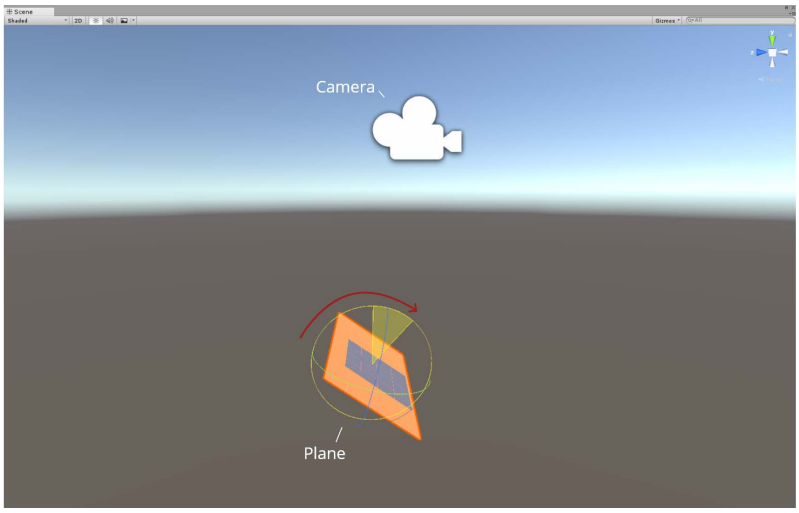

Figure 4.12: To correct the keystone effect the plane is rotated along the perpendicular axis to the camera in the Unity scene view.

effect, most modern projectors make use of built-in software [Yadav and Agrawal, 2013]. The projector used in the ZPAC lab is equipped with such software and can correct the distortion. This correction is done by digitally skewing the inputted image so that the projected image will appear rectangular on the projection surface, as shown in Figure 4.11

However, the keystone effect is still an issue for the image generated by the camera in the calibration part of the pipeline. As the camera is offset slightly from the two plane gameobjects, the captured image will be distorted. This distortion is caused by how cameras deal with perspective. The camera cannot be moved to the middle, as its exact point of view is needed to ensure consistent calibration. Therefore, the keystone effect on the render texture needs to be fixed using a different method before the final image is sent to the projector.

Keystone correction approach

The solution we implemented requires two additional gameobjects: (1) a plane gameobject, with the same dimensions as the large plane used in the calibration part of the pipeline, and (2) a camera. The plane gameobject has the render texture mapped onto it and is filmed by the camera. To fix the keystone effect, the plane is rotated along the perpendicular axis to the camera until the distortion is canceled out. This rotation is done in the Unity scene view, as shown in Figure 4.12. Our approach uses the same camera issue which originally distorted

the image, to create a non-distorted final image. This resulting image is then sent to the projector.

To summarize, the tracking and projecting pipeline consists of three parts: off-axis perspective projection, calibration, and keystone correction. A 3D illusion is created by applying the off-axis perspective projection. The resulting image is then correctly aligned with the real world setup during the calibration part. Lastly, the image is skewed in the keystone correction part, with a non-distorted image being sent to the projector. The design goal set for the implementation of the pipeline was met, as the majority of the parameters can be changed without needing to adjust the underlying code. Therefore the pipeline can easily be reconfigured for a different setup and used for a variety of use cases. One of these use cases is the game developed to show off the pipeline. The game itself, as well as how it was implemented is described in the following chapter.

Chapter 5

Game

An augmented reality computer game was developed in Unity. The game uses the tracking and projection pipeline, introduced in the previous chapter, and the OptiTrack system introduced in the chapter 3.1 "Tracking system". The design goal of the game is to showcase the tracking capabilities of the OptiTrack system and the 3D illusion created by the pipeline, while providing a fun game-play experience.

The implementation of the game was done using a two-phase approach, consisting of a prototyping phase and a development phase. This approach and the work done during the phases are described in the next section. For the final game to function as intended, we implemented a specific structure in Unity. This structure is described in section 5.2 "Game structure". Using this structure, scripts are implemented to control player movement and game-logic. These scripts are described in the final section of this chapter.

5.1 Game development approach

To successfully develop a large software, like a game, a suitable approach needs to be defined. A good approach helps developers stay focused on the required tasks and allows deadlines to be met. A commonly used game development approach was defined by Erik Bethke [2003]. His approach consists of: (1) the preproduction phase, (2) the prototyping phase, (3) the development phase, and (4) the post-release

phase. The preproduction phase involves creating a game concept and an accompanying vision document, which can be presented to game publishers. The prototyping phase involves finding suitable game mechanics and controls for the game concept through testing. The development phase uses the chosen mechanics and expands them into a fully featured game, making use of established game design principles. At the end of the development phase, the game is released. The last phase, the post-release phase, describes the period after the game has been released and focuses on updating the game with additional content.

Due to the short time frame for completing the game as part of a bachelor thesis, the long approach defined by Bethke [2003] was not feasible. Additionally, as there would be no further updating of the game after its completion, the post-release phase was not needed. Thus, the approach we chose consisted of two phases: (1) the prototyping phase and (2) the development phase. These phases and the work accomplished during them are described in the following subsections.

5.1.1 Prototyping phase

The prototyping phase consists of testing out game mechanics and controls with the goal of finding a fun combination [Sicart, 2008]. As part of the design goal of the game is to make use of the OptiTrack system's tracking capabilities, we decided upon using the projection surface itself as the controller. Thus, the tracked box lid functions as the controller for the game, with its movement, as well as pitch, jaw, and roll being the input.

Inspiration

With the control scheme chosen, a suitable game mechanic needed to be found. An early idea was to recreate the tilting maze platform of the "Myahm Agana Shrine" puzzle from the game "The Legend of Zelda: Breath of the Wild" [Nintendo, 2018a], shown in Figure 5.1. The puzzle consists of a floating maze-like platform and a physics-controlled ball. The goal of the puzzle is to successfully tilt the platform, allowing the ball to roll through the maze, while avoiding falling off the sides of the platform. Controlling the tilt of the platform is done by using the accelerometer inside of the Nintendo Switch console. The tilt of the hand-held console controls the tilt of the platform in real-time. As the platform is similar in shape to the rectangular box lid, developing a similar game mechanic seemed plausible.

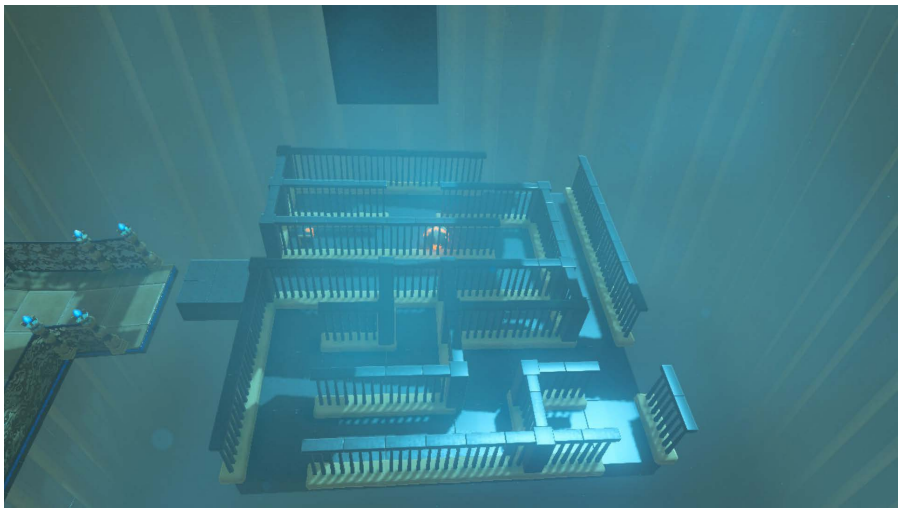

Figure 5.1: The "Myahm Agana Shrine" puzzle in the game "The Legend of Zelda: Breath of the Wild" [Nintendo, 2018a].

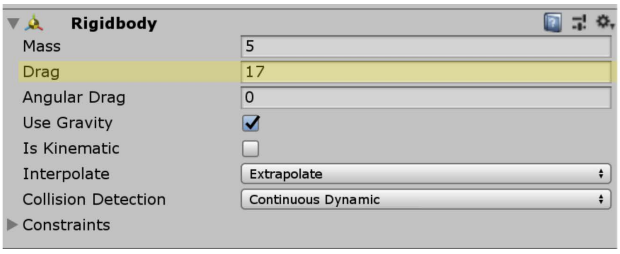

Figure 5.2: The rigidbody inspector tab in Unity, which is used to adjust the drag parameters.

First prototype

To test this potential game mechanic, a prototype was built in Unity. The setup consisted of (1) a plane gameobject, (2) a sphere gameobject, and (3) four cube gameobjects. The plane functioned as the platform and was controlled by the tracking data from the box lid through the OptiTrack system. The sphere was used as the ball and had a rigidbody component attached to it. This rigidbody component allowed the ball to be controlled by the Unity physics engine, making it roll to the lowest point due to the effects of gravity. The four cube gameobjects were placed on the plane to act as obstacles.

With the prototype complete, it could be tested regarding the design goals of the game. During testing, it became clear that the ball rolled very fast, which made it difficult to control finely by tilting the box

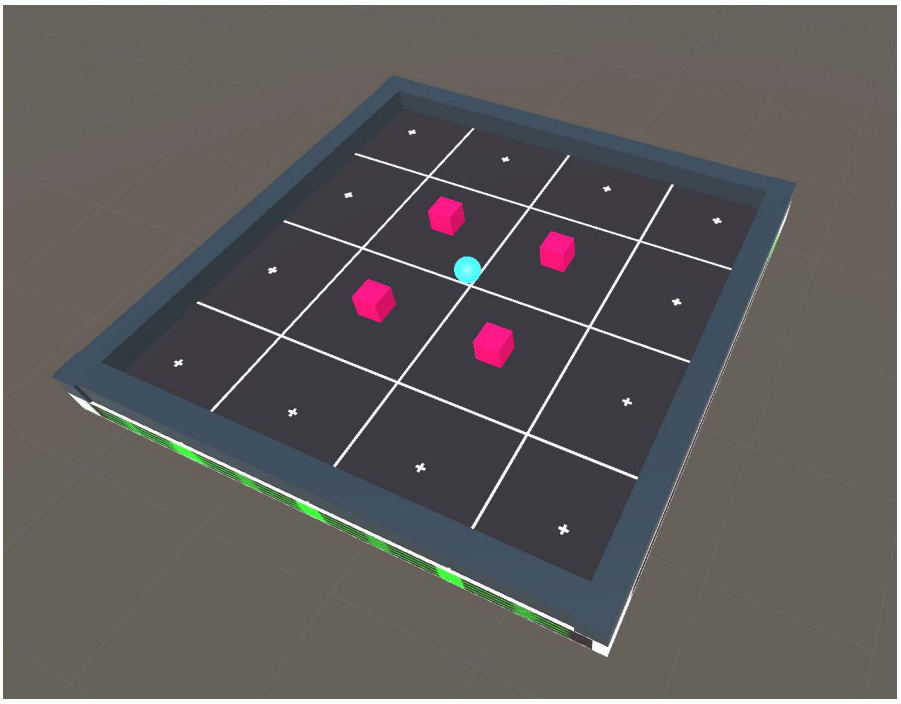

Figure 5.3: The second prototype as seen in the Unity scene view, showing the walls added to the sides of the platform. Additionally, the ball and the obstacles can be seen.

lid. This was fixed by increasing the drag parameter in the rigidbody inspector tab, shown in Figure 5.2, which increased the air resistance applied to the ball. The stronger air resistance slowed the movement of the ball, which made it easier to control and lead to the prototype being more fun. However, the possibility of the ball rolling off the edge of the platform was still a source of frustration. Thus, while the prototype did meet the design goal of making use of the OptiTrack system's capabilities, it did not provide a fun game-play experience due to the numerous fail states.

Second prototype

We implemented a second iteration of the prototype and removed the frustrating fail states by adding walls to the sides of the platform, as shown in Figure 5.3. As navigating the ball around the platform was no longer challenging, due to not being able to fall off, a new challenge was needed. After a brainstorming session, we decided on having the player fight the clock. A score would be increased through the repeated completion of an action before a timer ran out.

Figure 5.4: The taxi art asset, which replaced the ball used to test the game mechanic in the prototyping phase.

The specific action needed to increase the score was decided to be navigating the ball to different points on the platform as quickly as possible. Tilting the box lid without needing to worry about the ball falling off the platform, while still providing the player with a challenge, resulted in the desired fun game-play experience. Having found a suitable game mechanic, which met the design goals of the game, the prototyping phase was completed. The mechanic was then expanded upon in the development phase.

5.1.2 Development phase

After completing the prototype phase, the decided game mechanic is used as the basis for the full game. This involves creating an environment in which the chosen mechanic makes sense by using established game design principles to create a consistent and fun game.

Game environment

To create a believable environment, an in-game reason for making the player repeatedly traverse the available space on the platform, needed to be found. Upon further reflection, we concluded that a taxi fare system would match the chosen mechanic well. Taxis drive to a specific location to pick up a passenger, then drive to another location to drop

Figure 5.5: The final game environment showing the island art asset including as environmental art assets and the road network, as seen in the Unity scene view.

them off and then repeat the process. Thus, the ball was replaced with a taxi art asset, shown in Figure 5.4.

Following the same logic, the platform was replaced by an island art asset, which features a road network for the taxi to drive on, as shown in Figure 5.5. The roads also help signify where the taxi is allowed to drive. Having the game be set on a water-surrounded island also implicitly solves the issue of needing to explain why the taxi is not able to drive off the edges of the play-area. The chosen island art asset also features tall buildings to emphasize the 3D illusion created by the perspective projection described in chapter 4.1 "Off-axis perspective projection".

Sizing of the island art asset

Choosing the correct size of the island art asset is very important, as it can have a substantial effect on the way the game is experienced. The game design principle known as "spacing" describes the importance of sizing game elements correctly. The principle refers to having an understanding of how much space will be available, both inside the

Figure 5.6: The colorful objective markers signaling the pickup and drop-off points, which create a clear goal for the player to work towards.

virtual world, as well as on-screen space [Allmer, 2009]. It also urges developers to take the spatial relationships of elements into consideration, as well as what happens when those spaces are adjusted.

Space on-screen consists of the surface of the box lid, onto which the island is projected. Due to this limited space, the game environment needs to be sized accordingly. A small island would have less space for roads, but it would allow for the whole art asset to be scaled up to fit the available space on the box lid. A large island would have more space for roads but would require scaling down to fit the box lid. As the design goal of a fun game-play experience cannot be met if the player has difficulties seeing what is happening in the game, a smaller island art asset was chosen, which was then scaled up.

Objective markers

Having sized the island art asset, the specific locations the player is expected to navigate the taxi to, in order to increase his score, needed to be defined. These locations function as the objectives in the game, providing players with a goal they can work towards. More concrete goals are usually deemed better if high player participation is desired [Stillman, 2014]. Thus, colorful cube gameobjects were placed onto the road network, to act as objective markers. These cube gameobjects, shown in Figure 5.6, signal where the player should navigate the taxi to. The colors of the markers denote the type of marker: yellow markers are passenger pickup points, and purple markers are passenger

drop-off points. Having these two marker types enforces a sequence which the player has to follow: a passenger must first be picked up, before he can be dropped off. To decrease the fatigue of being required to follow the same game-play sequence repeatedly, the marker locations are chosen in a pseudo-random fashion during run-time. The script used for choosing the marker locations is described in section 5.3.2 "Level manager script".

For the game to be considered complete, additional elements like the menu and settings screens needed to be added. We implemented these screens in the final step of the development phase. These screens are described in section 5.2.1 "Main menu scene".

At the end of the development phase, the game was considered complete. However, for the game to look and function as described above, we implemented a specific structure inside Unity. This structure is described in the following section.

5.2 Game structure

To develop the augmented reality game, using the game mechanics and art assets described in the previous section, we implemented a specific structure. This structure was primarily determined by the way the Unity 3D engine is constructed, and the way our game should function. Unity uses files called "scenes", which can be thought of as unique parts of the game. All elements of the game need to be placed within such a scene [Unity Technologies, 2018f]. Thus, each scene file can contain, for example, the menu screens or a specific level in a game.

Unity scenes

The use of scenes allows for the game to be structured as parts, allowing developers to build up the game one part at a time. Additionally, this segmentation reduces the clutter in the Unity's visual editor, as only the gameobjects and art assets for a specific scene are displayed. If this were not the case, large games would require developers to search through an extensive list of gameobjects every time a parameter needs to be changed for a specific gameobject. Many large games have upwards of fifty scenes, consisting of one scene per level and several for the menus. However, this number can vary greatly, depending on the type of game and if procedural elements are used.

Figure 5.7: The main menu scene which greets the player when starting the game, showing the five buttons.

During the implementation of the game, we created three scenes: (1) the main menu scene, (2) the tracking scene, and (3) the level 1 scene. The main menu scene contains the different menu screens and is shown after starting the game. The scene's structure and how it functions is described in the following subsection. The tracking scene is loaded to align the projected image of the island with the surface of the box lid. This scene is described in subsection 5.2.2 "Tracking scene". After navigating the menu and ensuring good calibration, the level 1 scene can be loaded. It contains the game itself and is described in the final subsection.

5.2.1 Main menu scene

The main menu scene is used to display the main menu, shown in Figure 5.7, when the player starts the game. The scene uses a background image to make the main menu more visually appealing and to connect it to the theme of the game. The menu itself consists of user interface (UI) elements, such as buttons and labels. Unity requires all UI elements to be placed into a canvas gameobject. This canvas gameobject controls their position on the screen, and ensures the buttons do not overlap regardless of the screen's aspect ratio [Unity Technologies, 2018h].

Figure 5.8: The about text is shown after the about button is pressed on the main menu screen, with the back button to navigate back the main menu shown.

There are five buttons displayed when the main menu scene is loaded. These buttons are: (1) start, (2) calibrate, (3) help, (4) about, and (5) quit. The "start" button loads the level 1 scene, which then starts the game. The "calibrate" button additively loads the tracking scene into the main menu scene. The "help" and "about" buttons both open another canvas gameobject, which displays additional information to the user in the form of text, as shown in Figure 5.8. The "quit" button stops the game and returns window focus to the Unity visual editor.

5.2.2 Tracking scene

The tracking scene is used to align the projected image of the island onto the surface of the box lid. To accomplish this calibration, the tracking scene contains a simple version of the tracking and projection pipeline introduced in chapter 4 "Tracking and Projection Pipeline". Thus, the calibration tool can be displayed to allow the player to calibrate the pipeline and to save the parameters. These calibration parameters are then used by the full version of the pipeline in the level 1 scene during game-play.

The calibrate button in the main menu scene triggers the additive

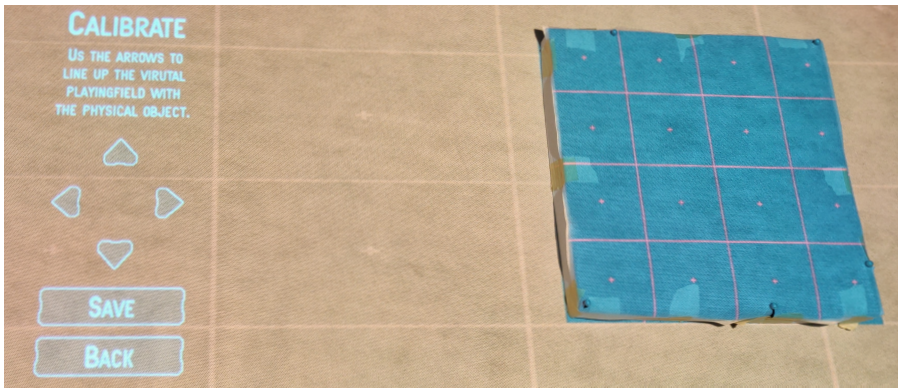

Figure 5.9: The calibration user interface displayed when the tracking scene is loaded additively, showing the text and buttons, which use the main menu scene canvas.

loading of the tracking scene into the main menu scene. This results in both scenes being active at the same time, which is only done is certain special cases. The reason we implemented the tracking scene to load this way, is to allow the canvas gameobject from the main menu scene to be used to display the text and arrow buttons.

5.2.3 Level 1 scene

The level 1 scene is loaded when the start button is pressed on the main menu. This scene contains all the art assets, game-play scripts, and UI elements required for the game itself. Thus, it is considered the scene in which the actual game is stored and played.

Nomenclature

The name of the scene, "level 1", was chosen to allow for an easy to understand nomenclature if the game is ever expanded. This expansion could potentially come in the form of new islands, with different road networks. Each of these islands would require its own scene. These additional scenes could then be named "level 2", "level 3", and so forth, to allow for easy distinction.

Structure in Unity

The structure of the level 1 scene inside of Unity is similar to the structure of the tracking and projection pipeline introduced in chapter 4 "Tracking and Projection Pipeline", as the game uses this pipeline. The main difference is a change in the structure of the off-axis perspective projection part. The cube art asset is replaced with all art

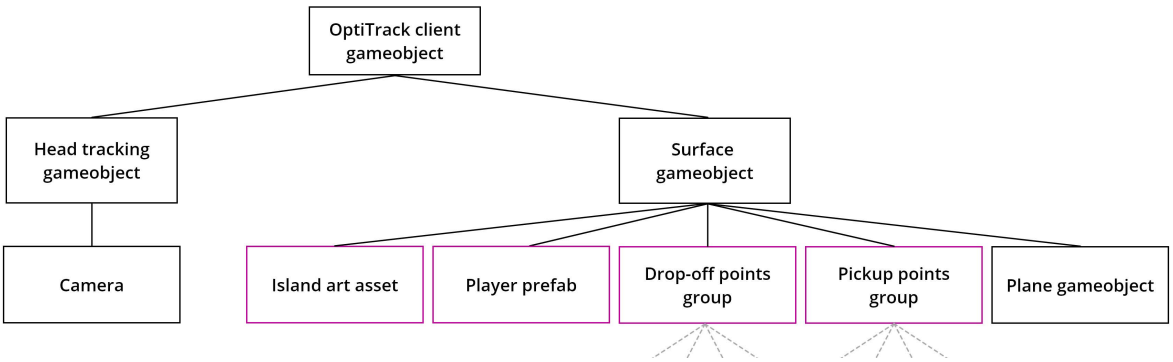

Figure 5.10: The new structure of the off-axis perspective projection, so it applies the 3D illusion to all the art assets and gameobjects during game-play. The new parts are shown in purple.

assets and gameobjects needed for the game, as the 3D illusion should be applied to them. This new structure is shown in Figure 5.10. Thus, the surface gameobject has four new children: (1) the island art asset, (2) the player prefab, (3) the drop-off points group, and (4) the pickup points group. The player prefab consists of the taxi art asset and the scripts required for the taxi to move, which are described in detail in section 5.3.1 "Player prefab scripts". The drop-off and pickup points groups each contain four colored objective markers.

Objective marker structure

Each objective marker is made out of a cube gameobject and a collider component. A Unity collider component is used to detect collisions with other gameobjects [Unity Technologies, 2019]. Thus, this collider component is used to detect if the player has successfully navigated the taxi to an objective marker.

User interface elements

In addition to the gameobjects and art assets which have been added as children of the OptiTrack client gameobject, the level 1 scene also uses various UI elements which are contained in a canvas gameobject. These elements include (1) a start timer, (2) a countdown timer, and (3) a score indicator. The start timer is displayed when the scene is loaded, to allow the player some seconds to adjust his focus before the taxi can be moved. The countdown timer is placed in the top left corner of the available projection space and shows the time remaining. The score indicator is placed below the countdown timer and shows the current score of the player.

The games' structure inside of Unity allows the game to function as

intended and to meet the design goal of showcasing the tracking and projection pipeline. Using this structure, scripts to control the player movement and the overall game-logic were implemented, which are described in the following section.

5.3 Scripts

To create a fun and complete game-play experience, different scripts were implemented. These scripts make use of the game's structure to control and define the way the game functions during run-time. While many different scripts were implemented, two are especially important: (1) the scripts contained in player prefab, as they define how the taxi moves, and (2) the level manager script, containing the game-logic. These scripts are described in the following subsections.

5.3.1 Player prefab scripts

The player prefab is a Unity prefab asset containing the taxi. Unity prefabs are reusable assets, which can contain multiple gameobjects and their components, parameters, and scripts [Unity Technologies, 2018e]. Such prefabs allow for a single item to be instantiated during run-time, instead of having to instantiate every gameobject and script separately.

Structure of the player prefab

The player prefab is comprised of: (1) a sphere gameobject and (2) the taxi art asset, as shown in Figure 5.11. The sphere gameobject has a rigidbody component attached to it, allowing it to be controlled by the Unity physics engine. The taxi art asset has two scripts attached to it, which define how the taxi drives around on the island.

As the player should see only the taxi art asset during game-play, the mesh renderer of the sphere is disabled. Disabling the mesh renderer prohibits the camera from rendering the surface mesh of the sphere, thus making the sphere invisible.

Sphere gameobject

The sphere gameobject is used to control the driving behavior of the taxi. As the Unity physics engine manages the sphere gameobject, it will constantly roll to the lowest point of the island when the player tilts the tracked box lid. This mechanic is used to control the driving

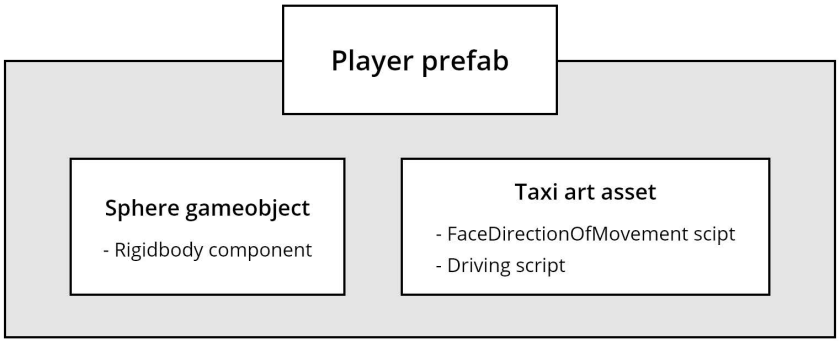

Figure 5.11: The player prefab asset, with the gameobject and art asset it is comprised of shown. Everything inside of a prefab can be instantiated with a single command during run-time.

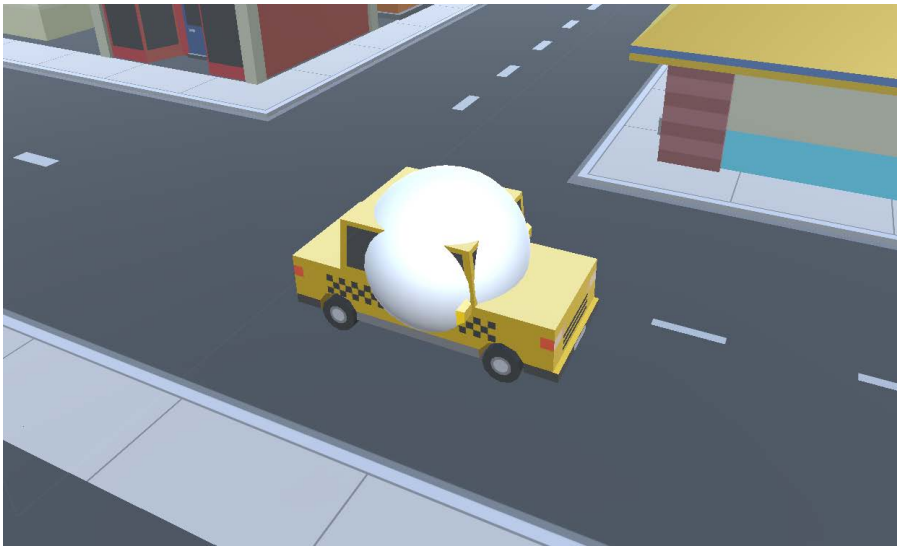

Figure 5.12: The sphere gameobject used for controlling the taxi, with its mesh renderer enabled for visualization purposes.

of the taxi by updating the taxi art asset's position to match that of the sphere, as shown in 5.12.

Driving script

Updating the position of the taxi art asset based on the position of the sphere gameobject is handled by the driving script, which is attached to the taxi art asset. To ensure the position is updated correctly

every frame, the code in the driving script is placed in the LateUpdate function. The Unity LateUpdate function is only executed after all physics calculations have completed Unity Technologies [2018b]. Thus, the position of the physics controlled sphere will have been updated before the driving script accesses these values. The code we implemented in the driving script is as follows:

```
1 void LateUpdate () {
2   spherePositionTemp = sphere.transform.position; //
        Fill a temporary variable with the sphere's
        current position
3   spherePositionTemp.y = spherePositionTemp.y -
        downwardsAdjustment; //As the sphere's origin is
        higher up then the taxi's origin, the temporary
        variable's y value is adjusted downwards. This
        adjustment ensures the wheels of the taxi art
        asset look like they are touching the ground
4   transform.position = spherePositionTemp; //The taxi
        gameobject's position is assigned the values
        stored in the temporary variable
5 }
```

FaceDirectionOfMovement script

The second script attached to taxi art asset, the FaceDirectionOfMovement script, ensures that the art asset is always rotated correctly based on the direction it is moving in. This rotation is required, as a real-life taxi can only drive in the direction it is facing, which should also be the case in the game. To determine the required rotation, the velocity vector of the sphere gameobject is used, as the movement direction can be determined from it. The code used to rotate the taxi art asset is as follows:

```
1 void LateUpdate () {
2   Vector3 movement = GameObject.Find("Sphere").
        GetComponent<Rigidbody>().velocity; //Get the
        velocity of the sphere, as the movement direction
        can be determined from it
3   transform.rotation = Quaternion.Slerp(transform.
        rotation, Quaternion.LookRotation(movement), 0.3F)
        ; //Rotate the taxi gameobject based on the sphere
        's movement direction and add some smoothing to
        the rotation by using the slerp function
4 }
```

Together both scripts allow the player to think that by tilting the box lid, he is directly controlling the taxi when navigating to the objec-

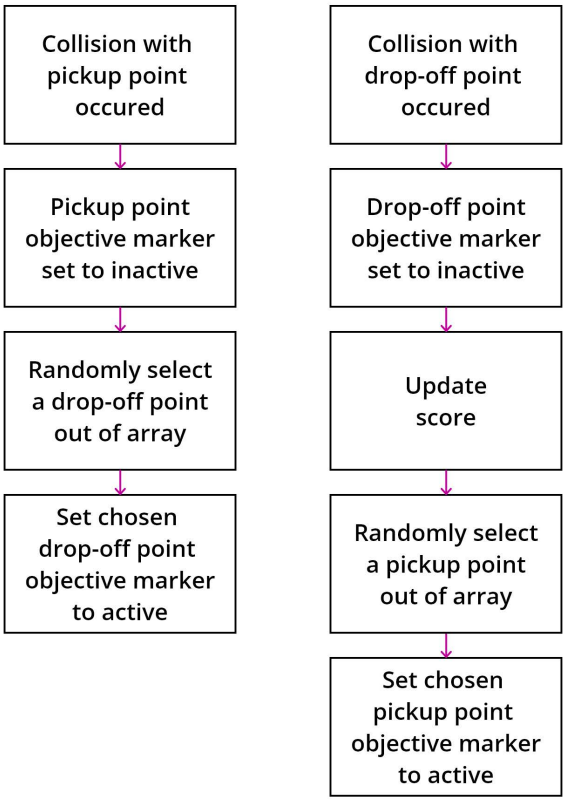

Figure 5.13: The sequence of events, in which the level manager script selects a new objective marker, showing when the score is updated.

tive markers. These objective markers are controlled by the script described in the following subsection.

5.3.2 Level manager script

The level manager script defines the majority of the game-logic and is executed as soon as the level 1 scene is loaded. The level manager script has three main functions: (1) to control which objective marker should be shown to the player, (2) to increase the score if a passenger is successfully picked up and dropped off, and (3) end the game when the countdown timer finishes.

Controlling which objective marker is shown to the player

To control which objective marker is shown to the player, the level manager script first adds all pickup and drop-off points into separate

arrays. All of these points are then set to inactive, making them invisible to the player. When the start timer is finished, the script randomly selects a pickup point and sets it to active, allowing the player to see the first objective marker. As the player navigates the taxi into this objective marker, a collision occurs between the rigidbody component of the sphere (introduced in section 5.3.1 "Player prefab scripts") and the collider component of the objective marker. This collision triggers the level manager script to set the objective marker to inactive, and randomly select a drop-off point from the corresponding array, which it then sets to active. This sequence of events is shown in Figure 5.13, for collisions with pickup points, as well as with drop-off points. The code we implemented for the sequence of events when the player collides with a drop-off point objective marker, including updating the score, is as follows:

```
1  //Check if a drop-off happened
2  for(int i = 0; i < dropoffPoints.Length; i++) { //For
       each drop-off point check if a collision occurred
3    DropoffTrigger dropoffTrigger = dropoffPoints[i].
       GetComponentInChildren<DropoffTrigger>();
4    if (dropoffTrigger.droppedOff) { //Enter if drop-off
       occured
5      dropoffTrigger.droppedOff = false; //So it is only
         triggered once per collision
6      dropoffPoints[i].SetActive(false); //Set the marker
         to inactive, making it invisible
7      score += 100; //Update score
8      pickupPoints[Random.Range(0, pickupPoints.Length-1)
         ].SetActive(true); //Randomly select a new
         pickup point and make it active
9    }
10 }
```

Ending the game

In addition to controlling which objective marker is shown to the player, the level manager script also ends the game when the countdown timer is finished. After ending the game, a message is displayed, showing the score achieved by the player. To allow the player enough time to read his score, a five-second delay was implemented before restarting the game. Restarting the game is accomplished by reloading the level 1 scene, which also reloads the level manager script.

To summarize, we developed the game using a two-phase approach, consisting of the prototyping phase and the development phase. This

approach allowed for the game to be completed in the given time frame. The final game utilizes a specific structure inside of Unity consisting of three scenes. Using this structure, scripts were implemented to control player movement and game-logic. The design goals set for the game were met. The OptiTrack system's tracking capabilities are showcased through use of the tracked box lid as a controller. The 3D illusion generated by the tracking and projection pipeline is used to transform the games' island art asset. Additionally, the game provides a fun game-play experience, meeting the final design goal.

Chapter 6

Summary and future work

6.1 Summary and contributions

As a recapitulation, a summary of this thesis is presented in this final chapter. This thesis showed that it is possible to develop a tool which allows researchers to use augmented reality in their experimental setup. The tool consisted of the tracking and projection pipeline. This pipeline is made up of three connected parts: (1) off-axis perspective projection, (2) calibration, and (3) keystone correction. The structure of the pipeline is such that it can be easily adapted to be used in future studies, without needing to change the underlying code. The pipeline developed in this thesis could potentially contribute to diminishing the cost of expensive experiment setups, by allowing difficult to construct objects to be digitally superimposed onto the scene.

To showcase this tracking and projection pipeline, as well as the tracking capabilities of the OptiTrack system, an accompanying augmented reality game was developed. The final game allows players to control a virtual taxi through the tilting of a tracked surface. The objective of the game is to achieve a high-score, by picking up and dropping off as many passengers as possible in the given time limit.

6.2 Future work

Considering the software developed in this thesis, notably the pipeline and the game, there are some aspects which could be expanded upon in future work. These aspects are: (1) the visual calibration interface of the pipeline and (2) the game's features.

Visual calibration interface

The visual calibration interface of the tracking and projection pipeline, introduced in chapter 4.2 "Calibration", allows for the tracking origin to be translated on two axes. The third axis, the z-axis, cannot be adjusted through the interface, as such an adjustment would only be required in rare cases. Nonetheless, a visual method for this adjustment could be added, to make the pipeline even more versatile.

Game features

Additionally, future work could also include the development of additional features for the augmented reality game. To further showcase the tracking capabilities of the OptiTrack system, additional objects could be tracked and used to interact with the game. An example of this could be fitting small cardboard cubes with tracking markers and attaching these cubes to the surface of the box lid. The cubes could act as roadblocks for the taxi, forcing the player to find alternative routes to the objective markers.

Another possible extension of the game could be the inclusion of different types of taxis. In addition to the yellow taxi currently in the game, players could be given a choice between different vehicles when starting the game. For example, a taxi-van could move slower but reward the player with more points per fare. Experimenting with different vehicle types and adjusting the game-play accordingly, could further increase player engagement.

Bibliography

Matt Allmer. The 13 basic principles of gameplay design, February 2009. URL https://www.gamasutra.com/view/feature/132341/the_13_basic_principles_of_.php.

S. De Amici, A. Sanna, F. Lamberti, and B. Pralio. A wii remote-based infrared-optical tracking system. *Entertainment Computing*, 1(3):119 – 124, 2010. ISSN 1875-9521. doi: https://doi.org/10.1016/j.entcom.2010.08.001. URL http://www.sciencedirect.com/science/article/pii/S1875952110000054.

Apple Inc. Arkit 2, 2018. URL https://developer.apple.com/arkit/.

Augmented Reality Games. Current state of the augmented reality technology, 2019. URL https://www.augmented-reality-games.com/technology.php.

Ronald T Azuma. A survey of augmented reality. *Presence: Teleoperators & Virtual Environments*, 6(4):355–385, 1997.

Uğur Başar, Vahit Sahiner Ali, and Barış Fidaner Işık. Off-axis stereo projection and head tracking for a horizontal display. 2009.

Jorge Bacca, Silvia Baldiris, Ramon Fabregat, Sabine Graf, et al. Augmented reality trends in education: a systematic review of research and applications. 2014.

E. Bethke. *Game Development and Production*. Wordware game developer's library. Wordware Pub., 2003. ISBN 9781556229510. URL https://books.google.ch/books?id=m5exIODbtqkC.

Boeing. Boeing tests augmented reality in the factory, January 2018. URL https://www.boeing.com/features/2018/01/augmented-reality-01-18.page.

Sam Bucolo, Mark Billinghurst, and David Sickinger. User experiences with mobile phone camera game interfaces. In *Proceedings of the 4th International Conference on Mobile and Ubiquitous Multimedia*, MUM '05, pages 87–94, New York, NY, USA, 2005. ACM. ISBN 0-473-10658-2. doi: 10.1145/1149488.1149503. URL http://doi.acm.org/10.1145/1149488.1149503.

Adam S Champy. Elements of motion: 3d sensors in intuitive game design. *Analog Dialogue*, 41(2):11–14, 2007.

Carolina Cruz-Neira, Daniel J Sandin, Thomas A DeFanti, Robert V Kenyon, and John C Hart. The cave: audio visual experience automatic virtual environment. *Communications of the ACM*, 35(6):64–73, 1992.

Crytek. Cryengine, 2018. URL https://www.cryengine.com/.

Alastair H Cummings. The evolution of game controllers and control schemes and their effect on their games. In *The 17th annual university of southampton multimedia systems conference*, volume 21, 2007.

Epic Games. What is unreal engine 4, 2018. URL https://www.unrealengine.com/en-US/what-is-unreal-engine-4.

Facebook Technologies. Oculus rift, 2018. URL https://www.oculus.com/rift/.

Michelle Fitzsimmons. Apple's tim cook: 'ar has the ability to amplify human performance', February 2018.

Google. Google glass, 2018. URL https://www.x.company/glass/.

Gutemberg Guerra-Filho. Optical motion capture: Theory and implementation. *RITA*, 12(2):61–90, 2005.

Nicolas Heuser. "window into a virtual world" screen concept - homemade cave environments. 2008.

Brian Karis and Epic Games. Real shading in unreal engine 4. *Proc. Physically Based Shading Theory Practice*, pages 621–635, 2013.

S.L. Kent. *The Ultimate History of Video Games: Volume Two: from Pong to Pokemon and beyond...the story behind the craze that touched our li ves and changed the world*. Crown/Archetype, 2010. ISBN 9780307560872. URL https://books.google.ch/books?id=PTrcTeAqeaEC.

Peter Koch. Optitrack for vr and unity3d, November 2016. URL http://talesfromtherift.com/optitrack-for-vr-and-unity3d/.

Robert Kooima. Generalized perspective projection. *J. Sch. Electron. Eng. Comput. Sci*, 2009.

Ernst Kruijff, J Edward Swan, and Steven Feiner. Perceptual issues in augmented reality revisited. In *Mixed and Augmented Reality (ISMAR), 2010 9th IEEE International Symposium on*, pages 3–12. IEEE, 2010.

Leap Motion. *Documentation - Technology Overview*, 2018. URL https://developer.leapmotion.com/documentation.

Michael Lewis and Jeffrey Jacobson. Game engines. *Communications of the ACM*, 45(1):27, 2002.

Minhua Ma, Lakhmi C Jain, Paul Anderson, et al. *Virtual, augmented reality and serious games for healthcare 1*, volume 1. Springer, 2014.

Microsoft. Kinect for windows, 2018. URL https://developer.microsoft.com/en-us/windows/kinect.

Mark R Mine, Jeroen van Baar, Anselm Grundhöfer, David Rose, and Bei Yang. Projection-based augmented reality in disney theme parks. *IEEE Computer*, 45(7):32–40, 2012.

NaturalPoint. *Rigid Body Tracking*, February 2018a. URL https://v20.wiki.optitrack.com/index.php?title=Rigid_Body_Tracking.

NaturalPoint. *OptiTrack Unity Plugin*, September 2018b. URL https://v20.wiki.optitrack.com/index.php?title=OptiTrack_Unity_Plugin.

NaturalPoint. Flex 13, 2018a. URL https://optitrack.com/products/flex-13/.

NaturalPoint. Optitrack, 2018b. URL https://optitrack.com/.

Nintendo. The legend of zelda: Breath of the wild, 2018a. URL https://www.zelda.com/breath-of-the-wild/.

Nintendo. Wii u, 2018b. URL https://www.nintendo.com/wiiu/what-is-wiiu.

PS-Tech. Optical tracking explained, 2018. URL http://www.ps-tech.com/3d-technology/optical-tracking.

RealIlusion. Clipping planes of the camera, 2018. URL https://www.reallusion.com/iclone/help/iclone3/10_Scene/Camera/Clipping_Planes_of_the_Camera.htm.

M. Ribo, A. Pinz, and A. L. Fuhrmann. A new optical tracking system for virtual and augmented reality applications. In *IMTC 2001. Proceedings of the 18th IEEE Instrumentation and Measurement Technology Conference. Rediscovering Measurement in the Age of Informatics (Cat. No.01CH 37188)*, volume 3, pages 1932–1936 vol.3, May 2001. doi: 10.1109/IMTC.2001.929537.

Miguel Sicart. Defining game mechanics. *The International Journal of Computer Game Research*, 8(2), December 2008. URL http://gamestudies.org/0802/articles/sicart.

R Stamm. Retroreflective surface, January 1973. US Patent 3,712,706.

Daniel Stillman. Seven principles of game design and five innovation games that work, October 2014. URL http://www.thedesigngym.com/seven-principles-of-game-design-and-five-innovation-games-that-work/.

Steve Sullivan, Kevin Wooley, Brett A Allen, and Michael Sanders. Visual and physical motion sensing for three-dimensional motion capture, September 2015. US Patent 9,142,024.

Thinkwik. Cryengine vs unreal vs unity: Select the best game engine, April 2018. URL https://medium.com/@thinkwik/cryengine-vs-unreal-vs-unity-select-the-best-game-engine-eaca64c60e3e.

TNW. This engine is dominating the gaming industry right now, 2016. URL https://thenextweb.com/gaming/2016/03/24/engine-dominating-gaming-industry-right-now/.

Unity Technologies. *GameObject*, November 2018a. URL https://docs.unity3d.com/Manual/class-GameObject.html.

Unity Technologies. *MonoBehaviour.LateUpdate()*, November 2018b. URL https://docs.unity3d.com/ScriptReference/MonoBehaviour.LateUpdate.html.

Unity Technologies. Unity3d, 2018c. URL https://unity3d.com/.

Unity Technologies. *PlayerPrefs*, November 2018d. URL https://docs.unity3d.com/ScriptReference/PlayerPrefs.html.

Unity Technologies. *Prefabs*, November 2018e. URL https://docs.unity3d.com/Manual/Prefabs.html.

Unity Technologies. *Scenes*, November 2018f. URL https://docs.unity3d.com/Manual/CreatingScenes.html.

Unity Technologies. *Transforms*, November 2018g. URL https://docs.unity3d.com/Manual/Transforms.html.

Unity Technologies. *Vertical Layout Group*, November 2018h. URL https://docs.unity3d.com/Manual/script-VerticalLayoutGroup.html.

Unity Technologies. *Collider*, January 2019. URL https://docs.unity3d.com/Manual/CollidersOverview.html.

Jeff Ward. *What is a Game Engine?*, April 2008. URL http://www.gamecareerguide.com/features/529/what_is_a_game_.php.

Wikibooks. Cg programming/unity/projection for virtual reality, June 2017. URL https://en.wikibooks.org/w/index.php?title=Cg_Programming/Unity/Projection_for_Virtual_Reality&oldid=3231767.

Jingming Xie. Research on key technologies base unity3d game engine. In *Computer Science & Education (ICCSE), 2012 7th International Conference on*, pages 695–699. IEEE, 2012.

Devvrat Yadav and Shashikant Agrawal. Keystone error correction method in camera-projector system to correct the projected image on planar surface and tilted projector. *International Journal of Computer Science & Engineering Technology*, 4(2):142–146, 2013.

Index

Typeset January 27, 2019