CameraTool: Pipeline Optimization for Camera Setup in the Unity Game Engine

Master's Thesis, 2011

Group 1032

by

Bjarne Fisker Jensen

Jacob Boesen Madsen

May 2011

Department of Architecture, Design and Media Technology
Aalborg University

PREFACE

This report is the documentation for the 10th semester project for the masters education in Medialogy, during the fall semester of 2010 and the spring semester of 2011. The report is part part of an extended delivery, covering the production period of 20 ECTS points from 9th semester as well as the full 30 ECTS from the 10th semester.

During the spring semester, the authors of the report participated in the DADIU production of March 2011, which has been an elective part of the 8th as well as the 10th semester Medialogy, for students in the Technical specialization program. The production lasted the entire month, and is considered a project period of 10 ECTS. All of the 10 ECTS came from the semester project. As the extended semester project is intended to be 50 ECTS, 10 has been using to participate in the DADIU production, leaving 40 ECTS of work for the semester project.

The project was planned to allow for the DADIU production in March to be used for testing the product, created within the fall period of the semester, as well as gathering data of usage of the product.

The authors contribution to the game created in the DADIU production can be found in Appendix A, and the game itself can be found Appendix B or on the webpage `http://www.branchgame.com/`.

The product developed for the semester project will throughout the report be refered to as CameraTool.

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

A camera operates by letting light pass through an opening, called aperture, and onto a surface with the ability to record the intensity and color of the light that hits it. Many modifications and improvements can be build on top of this simple system, like focusing the light that hits the recording surface with a lens or controlling the amount of time the surface is exposed. These are the key variables when dealing with still-image photography.

With the introduction of motion picture, the imaging is taken a step further by adding time to the equation, letting the image evolve over time. This meant the introduction of cinematography, which addresses the issues that arise when both the camera and motives in the scene may be in motion. Cinematography is closely related to the art of still image photography, and describe topics relevant in both worlds, such as lighting, aspect ratio, framing, depth of field etc. Additionally, cinematography also describe elements of a moving camera in a scene. The image displayed to the audience is the viewport into the story told in the movie, which means that the camera motion adds a big part to the viewer's emotional reaction.

Many parallels can be drawn from the motion picture camera to the virtual camera used in the world of computers. The virtual camera is an analogy for the real camera in the form that it shows the world through a viewport. Although the technique for creating the image in the virtual world differs from the real world, the cinematographic values are still considered to be effective.

1.1 Introducing Game Cameras

In the movie industry, a film is typically cut into sequences and scenes. Each scene can be shot using one or more cameras, which is later edited together to create the desired combination of shots. Unlike the movie industry, the camera systems used in games are limited to a set of constraints to ensure that the user does not get confused when playing the game. Since all games, per definition, uses some form of user interactivity, the game strives to have intuitive control and not leave the user lost and confused. A part of this is solved using clever (or very simple) camera controls in the games, where the goal is simply to not render to user confused. Common camera placements and angles from movies used in games include: Over the sholder shot, bird's eye shot and point of view

shot. Common for almost all games is that the camera is most often continuous and do not cut between camera angles very often, unlike the movie industry. There are of course exceptions to this, such as camera modes in racing games and point and click adventure games, but these are considered to be an exception to the norm.

In computer games, the function of the camera system is to manage the position of the viewpoint, as well as orientation and other properties, related to the events in the game. The camera is responsible for presenting the game to the player, and is thus highly influential on the player's perception of the game world.

How the world is presented and the effectiveness of the camera system is a major influence of the player's enjoyment and experience of the game. According to Mark Haigh-Hutchinson, the design philosophy of a virtual camera system is really quite simple:

> "An ideal virtual camera system, regardless of the game, is notable by the lack of attention given to it by the viewer" [5]

In contrast to the non-interactive camera in movies, the player is often directly or indirectly in control of the camera in a computer game. And with the constantly unpredictable changes happening in a game, the camera will need to be repositioned constantly, without annoying the player. In a movie, the director often set the stage and directs the action and camera before shooting. Despite this major difference, cinematographic principles are relevant for games as well as movies. Section 2.1 will look into current research in cinematographic principles used for real-time applications, such as games.

The camera system is usually designed to handle all camera specific features of the game, both for cinematic sequences and real time play. Most of the fundamental camera properties, such as position, orientation, aspect ratio, view angle, depth of field, etc. are part of the default camera component in Unity. Other properties of a camera, such as locks to distance, pitch, yaw, etc., as well as presentation styles such as behind the character, over the shoulder, etc. are not part of the default component. The standard package, shipped with Unity, include multiple scripts to help the user with setting up basic camera functionalities like smooth follow and mouse orbit. Other possibilities, such as predictive or reactive cameras are not part of the default Unity package. Cinematic events, such as re-orientation changes, camera shake, and other effects that resemble a humanly operated camera and not a computer generated one, are also not in Unity by default. Some of these will be relevant to look into for the camera setup tool for this report. Section 2.2 will look into the area of camera features, found in current and popular computer games.

1.2 Problem With Game Cameras

Different games approach the virtual camera system in different ways. A first person shooter, such as Counter Strike [6], aim to create a realistic camera system where the camera appears to be mounted on the head of the player. In order to convey the feeling of realism, the camera must obey the laws of physics from the real world. When the player is running, the camera should be moving as if a person is running with the camera, and when a nearby explosion goes off, the camera (and player) should shake from the blast.

Other games, such as car racing games, puzzle games, etc. uses other methods for controlling the camera, and in some games the camera will have a more important function than in others.

For most games, the process of creating a virtual camera system is a task for both the game designer and director, as well as the game programmer.

During the DADIU [7] productions in May 2010, in some of the teams, much focus was on developing the virtual camera systems. Both authers of this report were part of the DADIU productions, as programmers on different teams. During these productions, the game designers and game directors were in charge of the look and feel of the camera system, while the programmers were in charge of actually developing and implementing this into the camera system of the game.

During the productions, the camera changed a couple of times as a reflection of the game designer's and game director's ideas, and each time it caused other parts of the project to be delayed.

Many programming hours were spent on virtual camera development and many more hours on changing the camera multiple times afterwards. A tool which would allow the game designer to prototype a camera setup or even create the final camera setup, without having any programming experience, would be preferred in these situations.

The main problem field is that the creative people in any production, who does not have technical knowledge of the actual development, should still be able to contribute, and will have an idea of a concept they want to realize, or they might just have an idea of what they do not want. [8]

Based on the experience gained in the DADIU production during May 2010, a tool for easing the camera system development, will be created in this project. As the engine used at the DADIU productions since 2009 has been the Unity game engine [9], this is also the selected development platform for the CameraTool developed in this project.

1.3 Unity

Unity is a game engine developed by Unity Technologies [9], which allows for easy game development for multiple platforms, such as Playstation 3, Xbox 360, Nintendo Wii, development for Mac and PC's and Android-based smartphones, as well as Iphones and Ipads. [10]

Unity comes with great capabilities for creation of games in a relative short timeframe, as it is a complete tool with abilities for importing assests in the form of models, textures, sounds, etc. Thus it is a good base for creation of quick prototypes, as well as complete games.

Unity has a number of great features that assists in creating games and not focusing on creating the technology itself. It comes with Nvidia's PhysX for handling physics, OpenGL, and DirectX for 3D rendering and OpenAL for audio. It has built-in particle system generation and animation and multiple other features for developing games, quick and painless. And for the recent Unity 3.0 release, even more features, such as deferred rendering and lightmapping.

Unity uses the open source computing platform Mono, which allows scripting using C#, Javascript and Boo. [11]. The preferred language chosen for development is c#, although the difference between the available languages is very small.

1.4 Project Idea

The tool created in this report focusses on an easy-to-use approach, allowing the game designer to setup the virtual game camera system, with the needed functionality to setup a camera system for most game genres, without the need for a programmer to do the work.

A complete tool for assisting in game camera setup is to be developed for the Unity game engine. It should include functionality as seen in state-of-the-art cameras in current computer games, as well as past popular computer games. The tool should allow the game designer to setup a camera system with extensive functionality, using an intuitive, simple and costumizable interface to allow for different use in different productions.

The ability for a game designer to work on camera design without the assistance of a programmer has a lot of benefits. It will allow the game designer to prototype his ideas without having to wait for the programmer to implement the functionality first. This saves time and money for the overall production. It also frees up a programmer's time to spend on other elements of the game, which also cuts production time and cost overall.

1.5 Project Scope

In Section 2.1, Related Works, current research in game cameras will be investigated and described, followed by Section 2.2, Camera in Computer Games, in which current and popular games will be analyzed in order to uncover the camera features in those games.

Based on this analysis, the interface for the tool will be designed and developed with focus on usability by the end user.

The project is tested on the March 2011 DADIU production, in which the group members also participated, as game programmers in the production of the game Branch [12]. This allowed us to get firsthand experience with DADIU game designers using the tool for camera system setup. The tool were also made available for game designers on the other five DADIU productions.

The intention is for the tool to be used in a real production, by a non-programmer, and preferably a game designer.

1.6 Problem Formulation

Based on the experience gained at the DADIU productions, during March 2010, it is sought to develop a tool which will transfer some of the workload of the camera creation process, from the game designer along with the game programmer prototyping and developing the final camera system for the game, to only the game designer, or even the game director. This tool should seek to allow the game designer to prototype a camera system for the game, in order to determine the best possible solution to implement. If performance in the game is a non-issue, the tool might even be considered as a final camera system for the production.

It is expected that usage of the tool will allow the programmers more time to focus on other parts of the production than camera development. At the same time, the game designer should be allowed to prototype camera systems for the game without having to wait for a game programmer to implement features for the camera.

The tool should be presented to the game designer in a non-technical way, and should be designed with the user in focus, using user centered design principles, in order to design the system to be as intuitive and easy-to-use as possible for the end user, i.e. the game designer or game director.

> It is sought to develop an intuitive and easy-to-use tool for the Unity Game Engine, which allows game designers to setup a virtual camera system for computer games, without needing assistance by a programmer, containing most common functionality found in current computer games.

CHAPTER 2

ANALYSIS

This chapter will describe the analysis conducted in order to be able to properly design a complete camera setup solution to be used in computer games developed using the Unity game engine.

Section 2.1 describes works in the field of cameras in virtual systems, looking into cinematography, and the use of hard and soft constraints for the camera systems as well as camera planning systems. Section 2.2 looks into popular and current computer games and investigates the use of camera properties, to get an understanding of the needs for a camera system in todays computer games. As the tool should be aimed for game designers, and DADIU is a major influence in the creation of such a system, Section 2.2.2 will describe the use of cameras in a number of games created during past DADIU productions.

Sections 2.3 and 2.4 summerizes the chapter and sets limitations for the design of the CameraTool.

2.1 Related Works

Most of the recent literature on camera systems in computer games focuses on research in cinematographic principles for virtual camera systems as well as planning techniques for motion and positioning.

> The consideration of cinematic properties and literature (narrative) should provide cues to automatic generation of camera shots that are both intelligent and meaningful, enabling games that give the look and feel of a film. [13]

The quote describes a desired, though difficult, end result from a virtual camera system. One of the main differences between movies and computer games, is that games are highly interactive, which makes it hard for the game camera system to apply cinematic principles for conveying feelings in the same way as it is used in movies, where the director can set the stage and apply the techniques, and even retake the scene. The use of scene editing in games is a rare feature, as it takes control away from the player, whereas it in movies are part of setting the mood and used as a cinematic tool to convey a narrative. In games it is mostly limited to scripted, non-interactive cut-scenes.

Cinematographic principles used in movies has been developed and evolved for many decades and the language of the camera in movies are used to convey feelings and a narrative. In games, the camera is mostly used to give the player an idea of what is happening in the virtual world meaning that it is often used for functionality rather than aesthetic purposes.

Christie et al. describes three properties of an image; geometry, perceptual and aesthetics [13]. In many computer games, most cameras focus on geometry and a few games try to convey aesthetics as well. Conveying all three properties, as in movies, is a hard task for a camera system in an interactive real-time game world.

The idea of using cinematographic principles in games is a good one, but hard to implement, as the player might find himself in situations where the camera system breaks, and the cinematographic principles cannot be applied. When the camera system breaks, it is often because the camera is in an impossible position, like inside a wall, or breaks other physical laws. One method to avoid breaking the camera system is by using a constraint based approach to set a set a general principles the camera must follow, such as occlusion avoidance, view distance, view angle, etc. This approach gives the camera system a limited freedom for following the avatar and the more constrained the camera system is, the more rigid and inflexible the camera seems to the player. A problem is areas in the game where the constraints might not be solvable, and the camera breaks. Li et al. describes the uses of hard constraints, such as camera collision avoidance and avatar occlusion, and soft constraints, such as viewing angles and view distance, to overcome this problem [14].

Multiple recent litterature describes how to apply and implement cinematographic principles in computer games, and many focuses on constraint based approaches [14] [15] [16] [17] [18].

The idea of a cinematic camera approach has also been used to create an event-driven setup, where two persons are having a conversation, such as typically seen in movies, and in cut-scenes in a game [19] [20]. This makes heavy use of cinematographic principles to setup the camera(s) for the shot. It starts with an establishing shot, and cuts to over-the-shoulder shots of the avatars. Lin et al. also describes a camera module, in which a conversation script is also present [18].

Much of the conversations in games are in scripted cut-scenes of the game, so it makes sense to have a system for easy creation of a camera setup for this.

Bares et al. also describes a method for camera composition based on cinematographic principles, using a WYSIWYG[1] editor, instead of scripting [17].

Automated camera planning is another heavily researched topic in virtual camera systems. Christie et al. describes the expressivity of the parameters for the camera, such as shot type and angle, as well as different solving mechanics and approaches for this [13]. This camera planning is used by others in order to plan navigation through a virtual environment and not lose focus of the cinematic aspect of the camera system [14] [16] [21]. Most of the planning for navigation uses research for robotics to plan the movement through the scene. The approaches described require a pre-planning process, and assumes the scene is static.

In general, most current implementations of cinematographic camera systems for games are heavily tied to the narrative of the game, and creating a general purpose virtual camera system that can be used in most types of games are desirable for the understanding and the narrative of the game, but has deliberately been exluded in this project, to focus on more general camera features as seen in common games, which are

[1] What You See Is What You Get

not tied to a narrative. Instead, this report will aim to develop a complete camera system usable for current modern games. The camera system created will be customizable such that the game designer has the option of adding specific behaviors as events, which he can tailor to the game being created. This way a general purpose tool can be created, with option for customization for the game being developed.

2.2 Camera in Computer Games

Computer games are split into categories based on their game play rather than their visual appeal. Each genre focuses on unique sets of game play values and camera control tends can often be found within each genre. The largest and commonly accepted game genres include the following, according to Wikipedia [22]:

- Action
- Action-adventure
- Role-playing
- Simulation
- Strategy
- Sport

Other unique game genres exist like music, puzzle and party games but they add little or no new camera features to the list seen above and are therefore omitted from the analyses. The following section will describe the different camera trends found in games selecte from a subset of the list of genres.

2.2.1 Popular games within specific genres

The goal of this analysis is to find camera trends used in multiple games, rather than replicating one unique set of features from a specific game. Multiple genres and games are therefore looked into, based on commonly accepted games as well as personal experience. Games will be analyzed, and camera features extracted and described in details. Other games within same genre will then be referenced but since many camera features are the same with very few variations, they will not be described in details.

Action: Half-life and Counter Strike

Half-life [23] and Counter Strike [6] represent the first person shooter genre of the 21th century. They exploded the sales charts and became some of the most played games in modern time. Counter Strike is originally developed as a modification to Half-Life and the games therefore have a lot of engine features in common. The newest demo version of Half-Life 2: Lost Coast were tested as well as Counter Strike: Source. Ingame screenshots from the games can be seen in Figure 2.1. The following camera features were noticed:

- Camera position fixed to avatar position
- Camera orientation is linked to mouse movement
- Position and orientation shakes on small explosions
- Recoil when you shoot with a gun
- Crosshair changes color depending on target.
- Right click to zoom in

(a) Half-Life: Lost Coast (b) Counter Strike: Source

Figure 2.1: Screenshots from Half-Life: Lost Coast and Counter Strike: Source

Like all games in the first person genre, the position of the camera is placed where the head of the avatar would have been. Typically in first person shooters, there is little or no graphics for the avatar, although current games trends to show more and more graphics of the character when posible. This is also the case in both of these games. The mouse input is mapped to the orientation of the camera and can be compared to looking around with your head in the real world. Moving the mouse left and right is mapped to a rotation around the up-axis while moving the mouse up and down is mapped to a rotation around the right-axis. The latter is limited to a rotation between $[-90°, 90°]$ relative to the forward direction.

The games use a small shake of the camera when a bomb detonates nearby. This creates the illusion of a shockwave or small quake and the goal is to render the user a bit confused afterwards. The shakes appear to change the position and orientation of the camera in random directions within a time period of 0.2 to 1.0 seconds depending on the size and distance of the explosion. Afterwards the camera returns to its normal position.

Each bullet shot with the gun results in small recoil, making it hard to aim while shooting continuously. The recoil is rotating the camera view upwards with a small random offset.

Whenever the user targets a friendly unit, the crosshair in the middle of the screen changes to green and it is not possible to fire. The crosshair changes back to normal when the friendly unit is no longer aimed at.

When using a weapon with a zoom-scope, a right click will result in the camera to switch to a different view supposed to imitate the avatar looking through the zoom-scope.

Other top selling games in the genre include the Team Fortress [24] series and the Call of Duty [25] series. Team Fortress 1 and 2 seams to use the same features as listed above. The same goes for Call of Duty although the camera is a lot closer coupled to the movement of the character. The camera is bouncing from side to side when the character is running and takes a step and the camera is generally never kept still at any point. The camera movement when running is applied to the roll axis of the camera, and the frequency and amplitude is dependent on the characters running speed.

Action-adventure: Tomb Raider

One of the most popular games within the action-adventure genre would be the games in the tomb raider series. The game features a third person camera view, which can be seen in Figure 2.2. Unlike the typical first person shooters in the action genre, the action-adventure genre has more focus on exploration and interaction with the environment.

It is often seen that the game designers decides to use third person cameras in this genre to give the player more visual feedback on what the avatar is doing and interacting with. The action element is still present and the avatar often carries a gun or another type of weapon to defend/attack. This is a list of the most important camera features found in the Tomb Raider game:

- Camera always orbit around avatar
- Camera orbit mapped to mouse movement
- On occlusion the camera is translated to nearest none-occluded point
- Smoothed camera movement (delayed from avatar movement)
- Area based cut scenes that controls the camera
- Fade to black on death
- Position and orientation shake on explosions and on high velocity changes

Figure 2.2: Screenshot from Tomb Raider

Like first person shooter games, the mouse is used to control the camera movement. But where the first person shooter games used the mouse input to simply rotate the camera the third person games orbits the camera around a pivot point, which is typically the avatar. This means that obstacles may in some situations occlude the character. To avoid this, the camera is moved to the nearest non-occluded point, whenever the character is not visible to the camera.

When the character moves it seems that the camera does not follow instantly but instead has a small delay followed by an acceleration and deceleration of the camera. This creates a smoothing effect that result in a less rough and more pleasing movement.

When the avatar enters a new area the game sometimes takes over the movement of the camera and makes a quick flyby to give the player an overview of the area he just

entered. When finished, the camera returns to normal and the player can continue to play.

If the avatar is shot or killed in any way, the camera fades to black before the game is restarted.

Just like Half-life and Counter Strike the camera shakes when the avatar is near an explosion or another event that may trigger an unstable view.

Similar features are found in games like Grand Theft auto IV [26] and Demon Souls [27].

Role Playing Games: Diablo II

Role Playing Games (RPG) typically comes with two different camera types; a third person camera like action-adventure or a top down view on the avatar. Many parallels can be drawn between typical action-adventure games and third person camera RPG and it will therefore be omitted. The top down approach is on the other hand rather unique to the RPG and Real-Time Strategy (RTS) genres, and the genre was forever defined with the extremely popular launch of Diablo and Diablo II. A screenshot from Diablo II can be seen in Figure 2.3. Diablo II were analyzed and the following camera features were noticed:

- Camera locked in a 45° angle
- Camera is fixed on character
- Camera distance is fixed
- Instant movement, no smooth
- Occluding obstacles are rendered transparent

Figure 2.3: Screenshot from Diablo II

The game uses an extremely simple camera approach. The camera is positioned in what appears to be a 45° angle above the avatar with fixed height and distance. It is

moved instantly with the character, meaning that the character is always in the center of the screen unlike tomb raider that smoothes the movement. Whenever an obstacle is occluded it is simply ignored or rendered transparent.

The game play video for the Diablo III game, which is set to be released during 2011 or 2012[2], shows similar features for the camera as in Diablo and Diablo II, as well as the addition of camera shake [28]. Similar games in the genre like Baldurs Gate II and Dragon Age Origins does not introduce any new camera features not already listed in this section [29].

Real Time Strategy: Age of Empires III

The age of empires series is one of the largest games within the RTS genre and the third game in the series were chosen for analysis. The following camera features were noticed:

- Locked in an 45° angle
- Translation in same plane as ground
- Translation when mouse touches edges
- Translation with keyboard arrows
- Translation using Crtl + left mouse click + moving the mouse
- Translate camera forward / backwards using mouse scroll wheel
- No rotation possible

Figure 2.4: Screenshot from Age of Empires III

Unlike all the other games analyzed, this game does not evolve around a single character but rather a whole civilization. The camera in the game is not fixed on a single character but can be freely moved around by the player by using a set of input commands. Like Diablo II the camera is fixed in what appears to be a 45° angle above the ground. The camera can be translated around by touching the edges of the screen with the mouse, using the keyboard keys or dragging with the mouse while holding the crtl key. An in-game screenshot from the game can be seen in Figure 2.4.

[2]Or whenever.. it is Blizzard afterall.

The movement is always in the same plane as the ground, i.e. north, south, east and west and mouse scroll can be used to move closer / further away with limitations defined by the game. No rotation is possible in Age of Empires, but other games in this genre have been using this feature, allowing the player to see building and characters from different angles. The features found in Age of Empires III can also be found in games like StarCraft [30] and Empire Earth [31].

2D Platformer: Trine

Trine is a 2D platform game in a 3D engine, meaning that the avatars can only move in the 2D plane like any typical 2D platform game, but the objects are rendered with a perspective, as can be ssen in the in-game screenshot, in Figure 2.5. Even though the movement is locked in 2 dimensions, other actions are not. The camera and other objects may be moved in 3 dimensions. The camera functions were noticed in Trine:

- Smooth follow, delayed from character movement
- Camera focus in front of avatar
- Area based triggers that change distance, orientation and center
- Trigger based position shake of the camera in certain situations

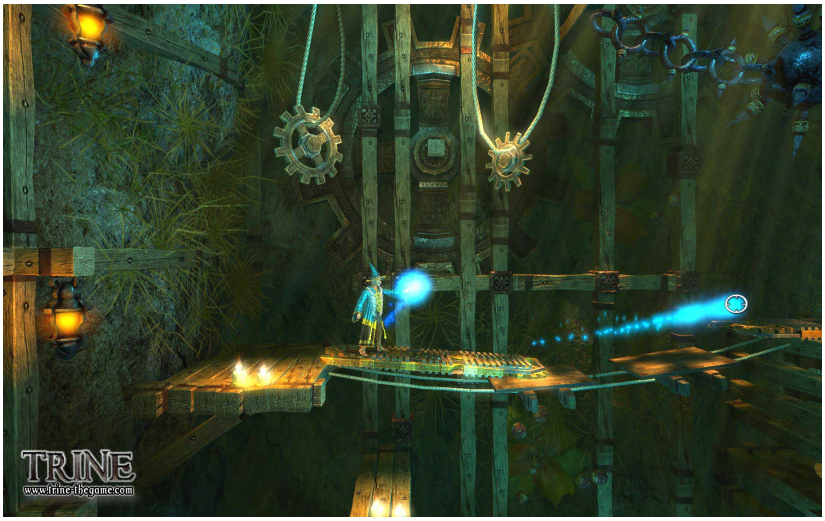


Figure 2.5: Screenshot from Trine

The camera in Trine has a small delay when following the avatar in the game. The center of the screen is not in the avatar itself, but rather a bit in front of the character creating a better view of the environment to come. The avatar sometimes has to walk through a gateway or door occluding the character graphics from the cameram but unlike Tomb Raider where the camera simply moved closer or made the occluding obstacle transparent, the camera simply moves to a position where the avatar is no longer occluded. This function appears to be area based and triggers whenever the character enters a certain area. The games also uses camera shake as seen in Counter Strike and Tomb Raider.

2.2.2 Earlier DADIU games

As the experience gained during the DADIU production in May 2010 served as a major influence in the creation of CameraTool, past DADIU games will be described in this section with repsect to the camera systems found in the games.

Slug'n'Roll

Slug'N'Roll [32], created at the DADIU production 2009 is a third person action game where the player controls a snail in a big garden. The goal is to collect flowers while avoiding the evil killer-snails. Your snail has the ability to retract into a rolling snail house, allowing it to kill the evil killer snails using its momentum. The snail also has the ability to fly by using its skin as a parachute. Both camera methods can be seen in Figure 2.6.

The camera control used in the game is kept to a bare minimum. It is always locked at a certain distance and angle behind the snail and follows instantly when the snail is moving or rotating. When objects are occluding the view of the snail, the camera is moved to nearest non-occluded point. The only time the camera changes angle is when the player enters the parachute mode. The camera then instantly cuts to another position above the snail, giving the player a larger overview of the map beneath.

(a) Slug'N'Roll on ground (b) Slug'N'Roll in the air

Figure 2.6: Ingame screenshot from Slug'N'Roll, created in the 2009 DADIU production

Nevermore

Nevermore [33], DADIU 2009 is a game in the horror genre where you play the Sad little girl, Anna, who gets bullied in the schoolyard. She gets convinced by her little teddy bear that a hopscotch diagram might cheer her up; all she need is some red paint. Without further questions she goes on a rampage in the schoolyard with her teddy bear and an oversized knife to collect the blood of the bullies to use as paint.

The game uses a third person camera where the user is unable to control the camera directly. The camera always keeps the same distance to Anna with her in focus and tries to orbit around her when she rotates so the camera always stays behind the girl. The rotation uses a smoothing algorithm, delaying the camera rotation relative to Anna's rotation.

There is no implemented solution to occluding obstacles and the camera may be stuck behind or inside an object.

When Anna is hit by one of the bullies the camera shakes for around half a second, like seen in the Half-Life, Tomb Raider, etc. Ingame screenshot from the game can be seen in Figure 2.7.

Figure 2.7: Ingame screenshot from Nevermore, created in the 2009 DADIU production

Puzzle Bloom

Puzzle Bloom [34] features a rather advanced camera control compared the other DADIU games from 2009. The game is a top down puzzle game where the user controls a green spirit that has the ability to attach itself and control monsters that inhabit a grey and dull world. The goal is to leap from monster to monster in an attempt to get to the checkpoints where a tree will grow and add color and light to the nearby environment.

The implemented camera is top down and fixed in what appears to be a 45° angle above the character. The camera can be controlled by using the right mouse button to rotate the camera around the character in the XZ-plane and change the distance to the character by using mouse scroll. When the character moves the camera follows by translating the position of the camera if the character moves parallel to the viewing direction, while the camera simply rotates if the movement is perpendicular to the viewing direction. All translations to the camera position use a smoothing algorithm.

When the character enters a checkpoint the normal camera functionality is overridden and the camera moves closer and takes a spin around the character before returning to normal. While doing the spin, the field of view is decreased to show more depth in the rendered image. The illustration in Figure 2.8 shows an in-game screenshot at a checkpoint, just before the game takes control of the camera.

There is no implemented solution to avoid occluding obstacles other than smart level design and the player's ability to move the camera using the controls described above, if the character is not visible.

Figure 2.8: Ingame screenshot from Puzzle Bloom, created in the 2009 DADIU production

Office Rage

Office Rage [35] features an ordinary office that one day snaps and starts breaking stuff placed around the office. The objective is simple: break and destroy as much of the company's property as possible before security arrives. The game uses a simple 3. Person camera placed behind the avatar at all times. A small delay is applied to the rotation, smoothing the movement of the camera when the character is rotated. All other camera functions are kept to a minimum leaving the user unable to zoom, rotate or in any way control the camera. Furthermore the game features no camera shake or any methods for avoiding character occlusion. An example of this can be seen in Figure 2.9.

(a) Office Rage (b) Office Rage (Occluding)

Figure 2.9: Ingame screenshot from Office Rage, created in the 2010 DADIU production

Anton and the Catastrophe

Anton and the Catastrophe was developed by one of the authors of this report, Jacob Boesen Madsen, during the DADIU May 2010 production. The game is about a small boy, Anton, who is tricked by an evil cat, and must collect a weapon and fight the cat in

order to save the city he lives in. [36]

Anton and the Catastrophe make use of an advanced camera. The control of Anton and the control of the camera are linked very closely. The player cannot move Anton directly, only indirectly by moving the camera or placing objects for Anton to react to.

The camera is limited in its movement, as it can only pan, and not rotate. The camera is semi-fixed to Anton, as Anton will move to the center of screen whenever the camera moves away from Anton. The camera cannot move too far away from the main character and has a maximum velocity equal to the maximum velocity of Anton.

When there has been no input (panning the camera) from the player in a 5 second period and the Anton is idle, he will seek out points of interest. The camera will follow Anton and be out of the user's control, as Anton has decided to walk towards it.

When Anton is using bubble gum (an object in the game), he gains the possibility to fly for a couple of seconds. During his time in the air he can be controlled directly by moving the mouse around the screen, and he will follow the mouse.

During the game play, when Anton finds parts of the machine to defeat the cat, the camera zoom in and Anton does a victory celebration. During this period of time, the camera and control for Anton is locked. After the celebration, the camera zooms to the original position and is unlocked again.

A second zoom is implemented for the final boss fight, in which the camera changes zoom and view angle for the fight. If the boss-fight is exited, the camera returns to the original again. In-game screenshot from the game can be seen in Figure 2.10.

Figure 2.10: Ingame screenshot from Anton and the Catastrophe, created in the 2010 DADIU midway production

Sophie's Dreamleap

Sophie's Dreamleap [37] was created by one of the authors, Bjarne Fisker Jensen, during the May 2010 DADIU production. The game is about a little girl, Sophie, who plays the childs game: the floor is made of lava [38], with her big brother. The ground literally turns green and becomes deadly, and Sophie has to jump from furniture to furniture in an attempt to get to her brother.

One of the constraints for the production was that the game should be controllable using only the mouse and left button click. The game is a 3d platformer and the mouse movement was reserved to controlling the character, leaving no input commands available for controlling the camera. This meant that the camera system had to be autonomous, and a trigger based camera system was implemented.

The camera in the game always tries to stay in the ZY-plane that intersects the position of Sophie while keeping her in the center of the screen. This means that the game worls is always viewed from one angle, since the camera orientation always maintain a positive z-axis. A lerp algorithm is used on the camera position to smoothing the movement of the camera when Sophie moves.

To ease the setup of the camera in the game, the position is defined using only a height and distance to the avatar. This meant that the game designer and game director had less variables to focus on when setting up the camera. These variables could then be changed based on areas that Sophie entered, if there was a requirement for a different camera setup for optimal game play. This was done by interpolating between the two setups whenever Sophie enters a trigger zone. The camera would return to normal upon exiting the zone. A screenshot from the editor can be seen in illustration (a) in Figure 2.11, where the triggerzones are colored blue.

An in-game screenshot from the game can be seen in Figure 2.11, in the image to the right.

(a) Editor view of the camera trigger system (blue) used in Sophie's Dreamleap

(b) Ingame screenshot from Sophie's dreamleap, created in the 2010 DADIU midway production

Figure 2.11: Ingame screenshot from Sophie's Dreamleap, created in the 2010 DADIU midway production

2.3 Limitations

This report focuses on the feasibility to develop a toolset inside the Unity game engine that allows a non-technical team member of a computer game production to define, create and setup a camera system with the most common and used camera features found in modern computer games.

Cameras in games are often connected with the user input and responds to the unpredictable real-time actions happening in the virtual world. Therefore all functions and features of the CameraTool are considered to be passive, meaning that CameraTool should not actively affect anything other than the camera itself. Behaviors like moving a character and changing the state of objects, which are not camera-related, in the scene are considered to be out of the scope for this project. The sequence for updating camera position according to Figure 4.1 in the Implementations chapter, 4, is used and it is up to the game programmers in the production team to make function calls and fetch data from the tool if more advanced features are needed.

Active GUI elements, displayed in the camera plane, showing different game play elements like health, time, etc. are considered too unique and outside the scope of a camera setup tool. The tool will however support GUI elements directly related to the simulation of a real camera like dirt splashing up on the lens or the camera fading to black.

Image based effects applied to the rendered image of the camera is not within the scope of this project. Native Unity supports a large portion of the most commonly used image effects like color correction, glow, noise and motion blur. A complete list if native supported image effect can be found in the unity documentation [39].

Research has been investigated in automation of cinematographic qualities and camera motion planning within interactive applications, and discussed in Section 2.1, Related work. Ultimately the goal for the CameraTool is not to develop fully automated cinematographic principles and motion planning camera as described in[13], [14] and [21]. The goal is to let the designer setup the desired cinematographic principles and camera movement using a set of hard and soft constraints in a constraint satisfaction system as mentioned in [14]. The constraints are a set of rules created by the designer, and it is up to the CameraTool to meet these requirements as best as possible depending on the game environment. This results in a more dynamic camera setup and gives more freedom to the user, since automation tends to lock the control to a set of predefined functionality or narrative.

The CameraTool is considered to be a general purpose toolset that allows the user to setup camera functionality seen in AAA games as well as previous DADIU game productions without the use of programming or scripting knowledge. The criterion for success is to save production time in a DADIU production, during the March 2011 production period, in which the authors of this report themselves will participate as game programmers.

It should be noted however, that at no point will the authors have anything to do with using the tool, as it is up to the game designer to make use of the tool. The authors will however be available for bug-fixing the CameraTool, should it be necessary.

2.4 Summary

In the previous section, 2.1 Related work, different articles regarding camera control within games were looked into. Christie et al. discuss the difficulties when trying to

apply cinematographic values taken from the film industry and applying them in games, where focus is put on interactivity [13]. This makes it hard for the designer to convey feelings in games using the same approach as in movies. It can be argued that it is harder to convey a feeling using a camera in a game compared to conveying the same feeling in a movie, since more valuables has to be taken into consideration. By having interactivity in games, the camera and game design must support the fact that the user does not always do exactly what he is supposed to, and the camera system might break.

The game design and camera system should provide a simple overview of the virtual world. Scott Rogers emphasizes the need for a well-functioning and non-breaking camera system in games to avoid leaving the user frustrated and disorientated:

> "Nothing will cause a player to stop playing your game faster than a poor camera." [40]

Li et al. introduces a method where the camera follows a list of soft and hard constraints [14]. The soft constraints are used to define the desired camera position and orientation in the game. These soft constraints are used as guidelines for the system to know what position is the most desired, although they are not always possible to achieve. The hard constraints overrule the soft constraints and are used to secure that the camera system does not break by ensuring that the camera is not rendered inside a wall and the target is always visible. The same approach is seen in multiple articles [14][15][16][17][18].

The different camera setups and cinematographic principles used in movies are thoroughly tested in theory and practice, and a detailed list of the different setups can be found in almost any textbook on the subject. The camera features used in games are however a bit more unclear and undefined, although many games seems to have a lot of features in common, depending on the game genre. In Section 2.2, Camera in Computer Games, a list of the most common and accepted genres used to categorize games were described in an attempt classify the different camera trends within each genre. Popular AAA game titles within each genre were examined and the different camera features were segmented.

The analysis showed multiple feature overlaps across the different games and genres and a list featuring the core camera functionality were created. It should be possible to:

- Set the camera position relative to another object
- Keep focus on a specific target or object, with the ability to add offset
- Pan the camera forward/backwards and left/right relative to the ground plane
- Change the orientation of the camera
- Orbit the camera around an object
- Move the camera closer or further away from the current view
- Trigger a camera shake when certain events accur
- Trigger a recoil effect that temporarily offsets the camera target upwards
- Add crosshair graphics to the camera and change it, based on the target pointed at
- Change the current view to a zoom scope with according graphics
- Translate position of the camera to nearest none-occluded point
- Change opacity on occluding obstacles
- Smooth camera movement
- Create area based cut scenes
- Fade the rendered image to black
- Create area based triggers that alter camera settings

The cameras used in all the games analyzed can be re-created by using a subset of features from the list, and the list are to be used as a guideline when designing the soft and hard constraints that should be available to the game designer through the CameraTool system.

In an attempt to study the actual use of the listed camera features found in AAA games, a selection of previous DADIU games were analyzed. The games were selected based on the complexity of the camera implemented and used in the games. Although the games feature rather unique content and game play, the camera features used did not distinctly deviate from the list of features found in AAA games. This implies that the list above is sufficient enough to recreate the typical camera features found in DADIU games, and that the games might even benefit from having more features available. It should however be noted that students in the DADIU program generally is encouraged to keep camera features to a minimum, to avoid tough debugging and to keep focus on the game play, during the short production time available.

CHAPTER 3

DESIGN

Despite we as a society develop more and more sophisticated tools to assist us in our daily work and errands, which, in theory, should make our lives easier and less complex, the reality is that with modern technology, everything is now more advanced, but also more complex, than ever before. [41]

This paradox of technology, which is here to help and assist, is also a great source of frustration to many people. This has to do with software development and interface design, in the sense that the paradox of technology should not be an excuse for poor design. Saying that a system is advanced or complex should not be an excuse for not designing a proper interface the user can understand. Well-designed object are easy to understand, no matter how complex or advanced the system is. Apple [42] is a great example of a company who understand designing for simplicity and with the end user in focus.

The whole purpose of the development of the CameraTool system is that the user should use less time and be less frustrated when setting up the camera system for his game, or it defeats the whole purpose of the tool in the first place.

This chapter describes the design for the GUI for the CameraTool system. The chapter will describe relevant topics within Interaction Design, Usability Principles and User Centered Design, which will be used as a basis for creating the interface for the system.

In order to integrate the system into Unity's editor, a section describing and analyzing the elements of unity's editor can also be found in this chapter. Following the guidelines for the design, a section describing the interface design for the system is found, followed by the technical design section, in which the technical design for the system is described. This section describes the internal framework for the CameraTool and the considerations made to integrate the system into Unity's component based system.

Throughout the chapter there are illustrations of mockups for the interface. The functionality depicted in the illustations are the functionality which can be found in CameraTool, and are described in detail in the implementation, Chapter 4.

The graphical user interface (GUI) design for the CameraTool system should be created to allow the user an easy and intuitive way of creating the virtual camera system for the game. In order to create a GUI that is as intuitive as possible to use, areas

such as Interaction Design, Usability Principles and User Centered Design will have to be investigated.

Following is a list of considerations made for the GUI design:

- What problem is the tool meant to solve; this should be designed for
- Make it easy to determine what actions are possible
- Unity design guidelines; the tool should feel as part of the Unity editor
- Intuitiveness; the GUI should make sense to the end user
- Low learning curve; only important information should be available at startup
- Gestalt theories; how are items grouped and placed in the interface
- Gulf of execution; the bridge between intended action and actual actions
- People don't figure out how stuff works; they muddle through

A user's first impression with a tool has a lot to say with whether they continue to use it. Not only should it be easy for the end user to learn and use, but it should not frighten a new user away. A lot of different methods will help develop an interface that won't scare the user away. This will be explained in the following sections on usability principles and design principles. In general, in order to design for any user, you will need to know your client, as Scott Goffman described in a talk at the Game Developers Conference 2011 [8]. A user will always want better tools to do his job, but often the same user cannot express or describe what he needs in a tool. It is the job of the developer and GUI designer to figure out how the tool should work, what it should look like, and how it should be used. This however, should not be done without any interaction with the end user. It is preferable to consult the end user, and preferably a more-than-average technical minded part of the end user group, to use for usability and beta testing of the product. One problem with tools development is that a technical person is developing software for a non-technical person to use. And technical people tend to want to be in full control of the software, with as many options as possible, where as any other person only need a miminum of those options during a regular work-week. There need to be a lot of focus on designing the tool for the end user.

As the intended end-user is the game designer or even game director in a production, the CameraTool systems should be created with the designer in focus. As developers are used to a certain terms and ways to work with software, the designer might not be familiar with those terms and used to work in a different way. The road from thought to action might be different for a designer, and the interface of the CameraTool should reflect this. The actions considered by a designer and a programmer, to setup a camera, are most likely different. A programmer might start with breaking the system into bits and pieces, while a designer initially considers the big picture, and might not even beware of these bits and pieces required for the camera to function as wanted. This was experienced firsthand during the DADIU productions in May 2010, in which the creative staff had certain criteria for the game camera system, but had a very limited knowledge of the steps needed to setup such a camera system.

Using terms from cinematography and movies in general bridges some of this gab in understanding and communication. In general, the considerations for the camera by the creative staff are most often a general idea, such as a first person camera system or a third person camera system. This system will often need to be extended, later in the production, as more criteria for the camera system are formulated by the game designer. In the beginning of a production, where the camera system are not in focus (the game design/mechanics often are), the game designer might only have a notion of what is needed, i.e. such as a first/third person camera system, and not specified this

further. There is useful information to gain, for the development of any tool for designers to use. In the production of CameraTool, it might be useful to allow designers to setup a camera system based on this notion. If a game designer wants to implement a third person camera, he might have the notion of a third person camera like Tomb raider [43], but does not immediately consider distance, view angle, limits to rotation, etc. for the camera system. Having a general camera system setup which can be customized might be worthwhile for any game designer and save time spent on prototyping different setups. This bridge, between intention and actual execution is described by Donald A. Normal, as the Gulf of Execution [41], which can be seen in Figure 3.1. The more narrow the bridge, the better for the end user.

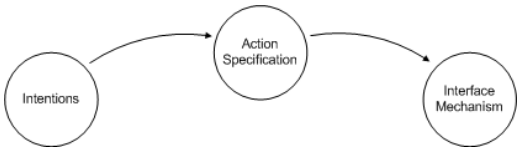


Figure 3.1: Gulf of execution

In addition to the Gulf of Execution, Donald A. Norman describes seven stages of actions, in which you begin by formulating a goal and ends with evaluating the outcome of the system. The seven stages of actions are [41]:

- Forming the goal
- Forming the intention
- Specifying an action
- Executing an action
- Perceiving the state of the world
- Interpreting the state of the world
- Evaluating the outcome

What can be gained from this is that not only should the interface be designed to be as little frustrating and difficult as possible, but the user will also want a feedback on the interaction. If the user performs an action and the state of the world appears the same, the user might be in doubt whether the action was performed at all. The bridge from forming the goal to executing one or more actions should be as narrow as possible, while the user is still in charge [41].

Johnson suggest keeping the conceptual model as simple as possible, but not simpler. Using as few concepts as needed and the design simple, the user will have an easier time mastering the product. However, making it too simple, the user might get frustrated, as there is not enough ability to change options or settings [44]. A tool as displayed in Figure 3.2 might be easy to understand and use, but the user might not feel he is in control of the camera setup, or feel that he lacks choices for the camera system.

A notion of a goal might be as simple as "Create first person camera" and the executing of this is more than a few actions, it is not only important to design the interface which lets the user setup the camera system, but also to design the interaction for the interface. In what order should the tasks be executed, in which order do they appear in the user's mind, and so on. This is worth considering as well, in designing an easy-to-use and intuitive application [41]. When designing the interaction, another important area to consider is the error handling as things go wrong. Even a single action should not

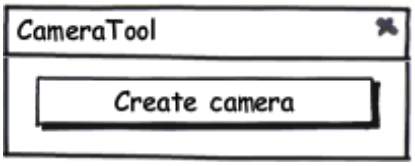


Figure 3.2: An interface can be designed to be too simple if all control is removed from the user

have severe repercussions for the user, and for multiple actions, the user will want to need what went wrong and why. It is the job of the tool to minimize the risk of error and minimize the cost of one [41].

A last item, before diving into a description of the design of Unity's editor, and the reason to do so, is the concept of familiarity. The CameraTool should feel as an integrated part of Unity when possible, and the designer should not feel a difference between the default Unity editor and any plug-ins or editor scripts. In the design of CameraTool, most often the design style of Unity will be taken into consideration in order to improve familiarity. This helps the user identify possible actions and allows for reusability in GUI design elements.

3.1 Design Principles

This section will describe different design principles for the GUI and how to design this interface with the focus being on the end user; the game designer. Before mentioning the full list of considerations taken for the GUI design, the most important thing, to avoid scaring away new users, are the thoughts regarding the interface as it appears the first time it is shown to the user.

A scaring example of a user interface, shown by Scott Goffman during his Game Developers Conference 2011 presentation, is the interface for the 3Ds Max [45] plugin, PolyBoost [1], which can be seen in Figure 3.3.

The interface for PolyBoost might be designed in order to speed up production time for an advanced 3Ds Max user, who has extensive knowledge in using PolyBoost as well, but it is most certainly not designed for ease of use, by a novice user. The complexity of PolyBoost however, should not be the excuse for poor design for new users. The interface could have been created with a novice interface as well as an advanced interface for power users.

In terms of User Centered Design [46], the interface of any tool should aim to help and assist the user as much as possible, and in general, be developed in collaboration with the end user. This involves interview with the user, in order to examine the user's needs, as well as usability testing of the production during the development. The usability testing will be described in the Test chapter, 5.

Based on resources [2] [41] [46] and [47], a list of considerations regarding the GUI has been compiled:

- Make it easy to determine what actions are possible
- Make it easy to evaluate the current state of the system
- As much as possible, it should be possible to operate the system without instructions
- Buttons should have self-explanatory names

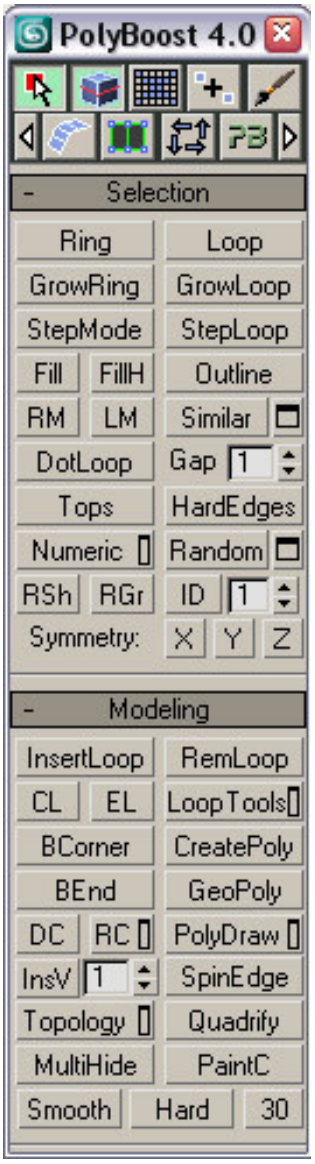


Figure 3.3: The PolyBoost plugin for 3Ds Max [1]

- Use terms the user is familiar with, in order for the user to not have everything in the head
- Use whitespace to group items
- Make items visible
- Create a clear visual hierarchy; appearance of things should portray relationships
- Break the interface up into clearly defined areas; users should easily identify what a section offers
- When all else fails, standardize!

In order to bridge the Gulf of Execution, it should be immediately obvious to the user what the system offers, in terms of **what actions are possible**. It should be clear to the user what is possible to accomplish with the system, and when familiar with the tool, also how this is accomplished using one or more steps. The goal or intention of the user might be to setup a camera. In such a case it should be clear to the user that he can either setup such a camera using a default setup or by setup of the needed functionality from scratch, such as set view distance to zero, enabling mouse rotation, etc.. In the same manner, it should be easy for the user to **quickly evaluate the current state of the system**, allowing him to get an overview of enabled functionality and see the kind of camera created.

Designing the CameraTool for easy and clear view of the current state and available actions, features for the camera should be in separate categories, displaying whether the feature is enabled or disabled.

As much as possible, it should be possible to operate the system without instructions and **Buttons should have self-explanatory names** are similar in the sense that the system should easily explain to the user what each button or slider does, without the user having to guess or read a manual on operating the system. The chain of actions from the user should be intuitive and make sense, and the naming of objects should not hinder the user in achieving his goal.

Use terms the user is familiar with, in order for the user to not have everything in the head is a term described by Donald A. Norman, similar to the topic of being able to operate the system without instructions [41]. He mentions that often the user does not need to remember everything in his head, as long as it is possible to easy relearn the knowledge, when needed. In this case, using terminology familiar to camera terms in the movie industry and terms from the computer games industry in general, it should be possible for the game designer to quickly pick up and relearn the terminology when needed. Labels for groupings and tooltips for items might be beneficial in the process of quickly re-learning the tool. However, using too many tooltips and labels might annoy an advanced user, so it should be considered whether this is necessary or not.

By using **whitespace**, both as indention of objects and as spacing between groups, it should be clear to the user to what category of objects an item belongs. An example of this can be seen in Figure 3.4.

Making things visible to the user, is an interesting point. An item should be visible to the user, but at the same time, it is important not to clutter the user interface in items and buttons, and thus confuse the user, rather than help him. It is important the user knows where to find the item, rather than making everything immediately visible, as was seen in PolyBoost, in Figure 3.3. As mentioned by [47], the number of clicks does not matter for the user, as long as each click makes sense. This will be expanded in the Usability Principles section, Section 3.3.

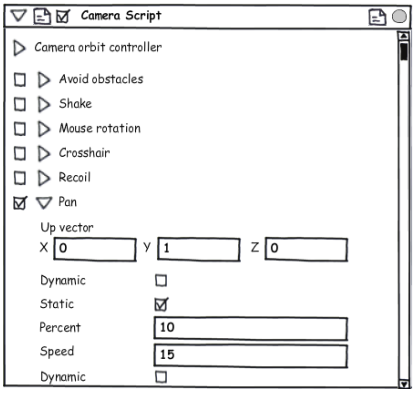


Figure 3.4: Clear visual hierarchy

To create a **clear visual hierarchy** of groupings of items in the interface, things will be grouped in categories, which the user can expand to get more information and see advanced features. This is coupled with makings things visible, as well as it should not confuse the user with a cluttered interface. This also help **breaking the interface up into clearly defined areas**. An example of this can be seen in Figure 3.4.

In order to avoid clutter, other than group objects in a hierarchy and use whitespace to assist the items, it is important to question the use of every item in the interface. A developer will often want as much control of the system as possible, either via the interface, a terminal or third. However, when designing an interface for another person, it is important to remember that this person does not poses the extensive knowledge of the developer, and most likely does not need the same amount of control the developer does. The question: "does it need a button", should be given to every function in the system, and if a button is never used by the end user, it is not needed, and should be removed, or hidden, from the standard interface. In general, if it is not clicked frequently or repeatedly, it might not be needed, and should most definitely not be in the default or simple view mode of the tool.

When all else fails, standardize! Having a lot of design principles to follow is great, but they should not be applied at any cost. If it is not possible in the given context, or the user refuses to work in a certain way, best practice is to follow the conventions of the given program which you are trying to extend. Section 3.2 will look into the general design of the Unity editor's interface.

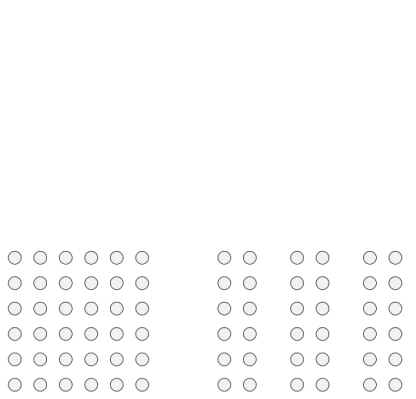
The tool should be placed in the most logical place, or most logical places, depending on the program. Having multiple plug-ins or extensions for a program, it might be a good idea to group those in a single menu bar, called plug-ins, as well as having each also grouped under other menus where they might make sense. An example of this is having an export plug-in in the file menu, near the Save functionality. Users might not have the same train of thought, and having tools available where they make sense, lessens the time user has to look for the menu. This however is a road to menu clutter, and should be considered carefully.

Even the best designed interface does not miraculously make the user an expert in using it or understanding the terms used, in this case camera terminology, if they do not have any experiences with cameras beforehand. For this, the tool should have some sort of help file or guides to use of the tool. For this, a help file can be distributed along with the tool, or an online wiki created, in which the users themselves can contribute with

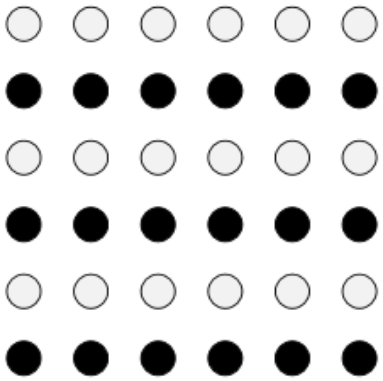
information and guidelines for other users. Given that the tool can be extended, these scripts can be shared as well.

3.1.1 Gestalt Theories

Gestalt theories have already been used in the design principles section, 3.1 of the Design chapter, without mentioning a name for it. The laws of grouping have been used in description of grouping of elements of the same component or the same feature. There are five gestalt laws of grouping [48], which can be seen in Figure 3.5. Of the five laws of grouping, the law of proximity and the law of good continuation are used in the design of the interface. As Figure 3.5 shows, settings within a group are in proximity of each other, while other settings and functionality are separated by the group expand/collapse option. This helps the user in identifying items and settings tied to the same feature or component, and will lessen frustration when looking for a setting of a certain feature.

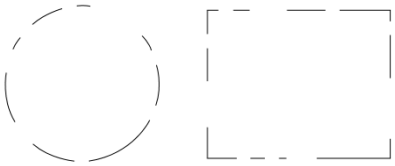


(a) Law of proximity [49]

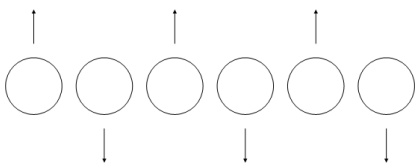


(b) Law of similarity [49]

(c) Law of good continuation [50]



(d) Law of closure [49]



(e) Law of common fate

Figure 3.5: Gestalt laws of grouping

3.2 Unity Design Guidelines

The design of Unity's default editor can be seen in Figure 3.6.

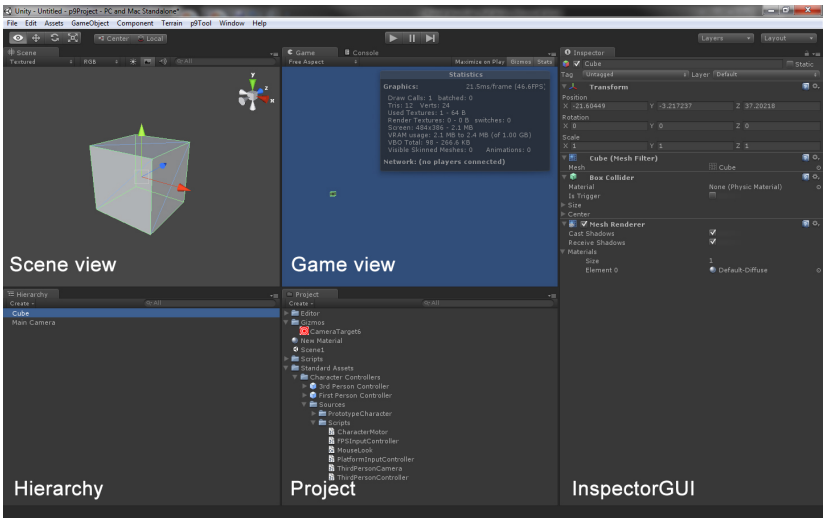


Figure 3.6: Screenshot of the Unity editor (Pro version)

Inside Unity it is possible to change the default inspector layout of your components by creating a custom inspector. Custom inspectors exposes GUI control elements from the GUI, GUILayout, EditorGUI and EditorGUILayout classes avalible through scripting [51]. All inspector GUI elements seen within unity are created using a subset of the elements available in these classes, thus making it easier and straight forward to design a tool in the same design context as Unity.

The unity inspector is split into two vertical columns, illustraded in Figure 3.7, where the rightmost column is primarily reserved to user input controls like toggle buttons, text fields, number selector and more advanced controls like color picker and object selector. The leftmost column is primarily used for showing small descriptive text matching the input control. The native FBX importer, Figure 3.8, illustration A, is a good example of a standardized design within unity. The Importer features a number selector, dropdown menu, toggle boxes, sliders and more, in the right column with matching descriptive text on the left. Note that all objects and text is left aligned creating a distinctive separation of the two columns.

Indention of the text is used to display sub options and controllers to a parenting control objects. This grouping makes it easy to gain a quick overview of the available options, and the indention often comes with the ability to collapse child objects. Examples of intention and collapsing inside unity can be seen in Figure 3.8 illustrations D and E. An arrow pointing to the right or down is used to indicate whenever the object is expanded or collapsed.

Unfortunately the above design guidelines cannot always be met. The design principles are often broken when a GUI element require more than usual space. This is particular visible in Figure 3.8 image B and C where the vertical alignment lines often gets broken by wide input fields or multiple input fields in the same horizontal line. The result is a clustered interface and it gets harder to gain a quick overview. More examples where the design guidelines are broken, can be seen in small things like the indention

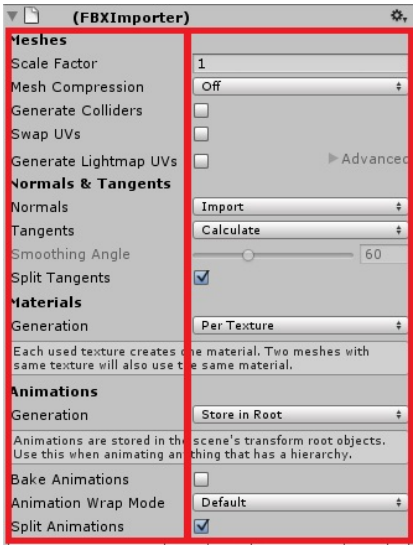


Figure 3.7: The unity inspector is designed using two columns. The leftmost column is used for descriptive text and the rightmost is used for user input objects. Both columns is left-aligned

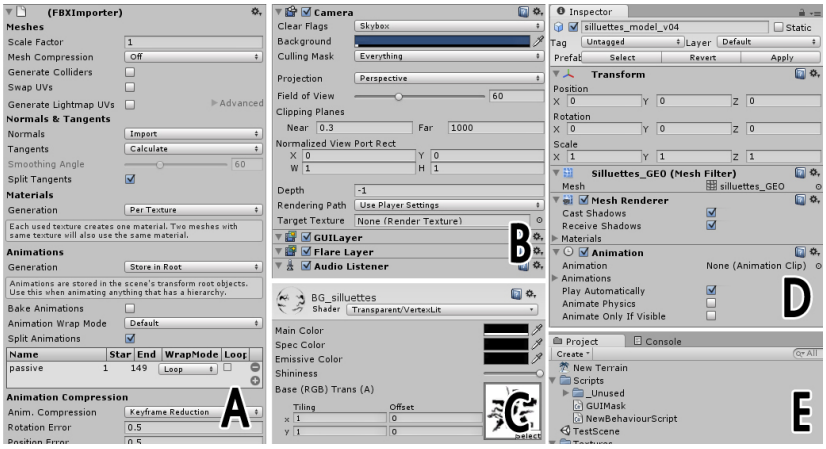


Figure 3.8: Screenshot of the various elements from the Unity editor

of the expand arrow in Figure 3.8, image D and E. The Project menu in image E uses indention of the arrow while the expansion arrow for animations in image D is left aligned without indention. More examples of broken design guidelines can be found through the editor and it reflects that the Unity game engine and interface design is still a work in progress. This makes it hard to make specific demands to the design of the CameraTool, although it should try to respect the guidelines described.

3.3 Usability Principles

This section describes the process of creating an easy to use and intuitive interface for the user. The considerations mentioned here will be used later in creating the interface for CameraTool.

One might think that an easy approach would be to ship the product with an extensive manual, describing the program and act as a user's manual as well as a place to look when in trouble. However, the hard fact is that very few people actually take their time to read a manual, or any instructions for that matter, when dealing with new technology [2]. Instead, people often figure out how things work on the fly, as they use the technology. They might use a manual for debugging or a guide as a tutorial, but for most part, they muddle through.

Despite people in general muddle through, the interface should make use of design and usability guidelines, to make sure the user find what he is looking for. It also gives a better chance the user will understand what is offered to him, and it makes him feel smarter and in control of the program. A positive association with the program or tool will increase the likelihood of him actually using the program or tool.

Realizing this, there are some considerations to be taken when creating the interface for an end user, who does not care, or we will assume he does not care, about reading instructions.

Some general considerations regarding the usability of the system, based on [2] [8] [41][47], are:

- Make it as easy and intuitive for the user as possible
- Provide a good conceptual model
- Make sure only the most important elements are available
- It should be clear how an action can be used together with another action
- All actions should have reactions
- Aim to make errors as cost-free as possible for the user

The first and most important issue is to **make the system as easy and intuitive for the user as possible**. This is tied greatly with the design principles mention in Section 3.1. In order for the user to easily identify options in the interface, items are grouped into categories based on features for the camera, each collapsible, to give an easy overview of the features, and to avoid clutter in the interface. An example of this design can be seen in Figure 3.9. Making the design easy and intuitive is actually tied with everything in this chapter, as all elements should be tied together to design an interface the user can understand, which are easy getting started with, and the user actually wants to use in a production. This only happens if the interface is easy and intuitive, and the user is not afraid that the system will make errors or a mess of his project. This last part, **making errors as cost-free as possible**, are also relevant, as an error which result in a massive cost for the user, will most likely result in the user never, or only reluctantly, use the program or tool again.

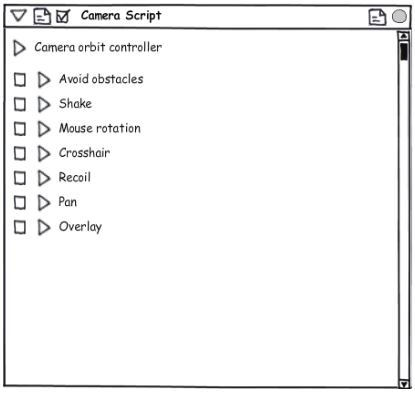


Figure 3.9: The collapsed interface design

The actions from the user's intention to the actual actions taken should be simplified to the least possible actions available, in which the user still feels he is in control, to make the action as intuitive to the user as possible. A one button "Create camera" tool, as seen in Figure 3.2 might not be the choice, as the user will most likely not feel in control.

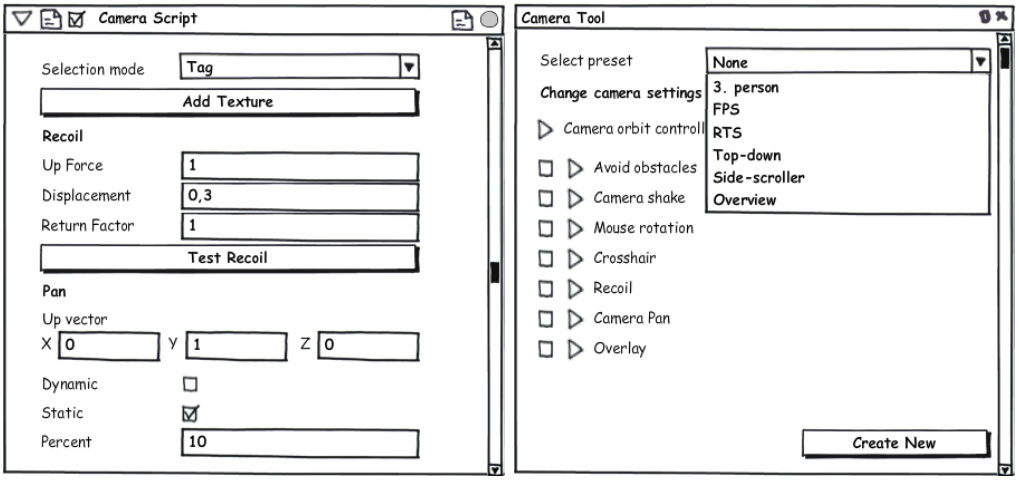
What should be done is to create a good conceptual model for the user to understand [41]. Concepts the user understands in a camera system might be "First Person Camera" or "Third Person Camera", which might be identical to their intended goal, and the user might not automatically break this down to the individual components and features needed for such as camera. Even though most of the terminology will be similar to that used in cinematography and for movies, it might be beneficial to create a conceptual model closer to what a computer player is used to. This could be used for presets, such as first- or third-person cameras, which can be adapted to the need for the production. It could also be used for settings, such as whether the camera should follow an avatar on the screen, or not.

To not confuse new users and allow users to easily learn the tool, the design should focus on a simplified interface, usable for most uses, with option for an advanced view to customize the camera system further. **Making sure only the most important are available** to new users will lessen the risk of scaring them away before they even use the tool. Steve Krug also mentions the importance of this:

> "if something requires a large investment of time - or looks like it will - it's less likely to be used" [2]

This is tied both with the Gulf of Execution and the seven stages of action. The user forms an intention of completing a seemingly simple task, such as creating a first person camera. He then opens the tool and sees a lot of different items, labels, buttons, sliders, etc. He then has to figure out what possibilities he has, and how he should translate this into his intentions and in what order to apply the actions. This requires a lot of time and effort from the user, and he might choose to delegate the job to another person, i.e. the programmer, with a post-it note containing desired functionality. An example of this interface is illustrated in Figure 3.10.

In another scenario, the user forms an intention of completing a seemingly simple task, such as creating a first person camera. He then opens the tool and sees categories, each labeled with a simple word or sentence, such as "Pan options", "Camera shake", etc. and on the top of the tool is a dropdown menu with a list of presets, such as "First



(a) Cluttered GUI (b) Clean and simple GUI

Figure 3.10: Screenshots from the a cluttered (left) and a cleaned GUI (right)

person", "Third person", "Side scroll", etc. This design is less cluttered and also bridges the Gulf of execution, which makes it more likely the user will use it. An example of this interface is illustrated in Figure 3.10.

An element of the design to take into consideration is the amount of clicks the user should need to find and set a setting in the interface. A rule-of-thumbs, for internet usability, states that the user does not want to click more than three times before giving up [52] [53]. Recent research, from the article [47], as well as wikipedia [53], states that this is not true. It is true that the user when surfing the internet does not want to wait and usually give the website a very short lease before leaving. The research shows however, that as long as each click makes sense to a user, and he does not feel he clicks blindly, the three click rule does not apply. This fits greatly with trying to simplify the interface, in order to not scare anyone away immediately.

It should be clear how an action can be used together with another action. When the user breaks the desired intention down into components and features, it should be clear using the tool, how a component can be used together with other components, and how changes to one might change the camera system in the production. In general, the less time the user needs to figure out a task, the better. This is a hard task for any system, but should not be forgotten in the design of the interaction.

As a final note, it is not possible to make everything self-evident, and following these design and usability principles does not allow for a magical design with does everything for the users without him needing to stop and think for a second, but as much as possible of the interface and interaction should be self-evident.

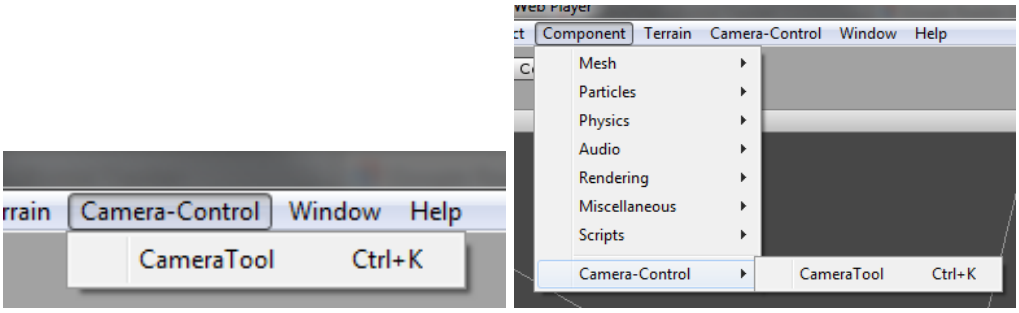
> "If you can't make a page self-evident, you at least need to make it self-explanatory" [2]

This quotation is said about web-usability, but should be resourceful for any interface design.

3.4 Interface Design for CameraTool

This section describes and illustrates the design of the interface for the CameraTool, at the time of the DADIU production in March 2011.

The CameraTool is placed in both Unity's Component menu, as well as its own menu, for easy access. Illustrations of this can be seen in Figure 3.11.

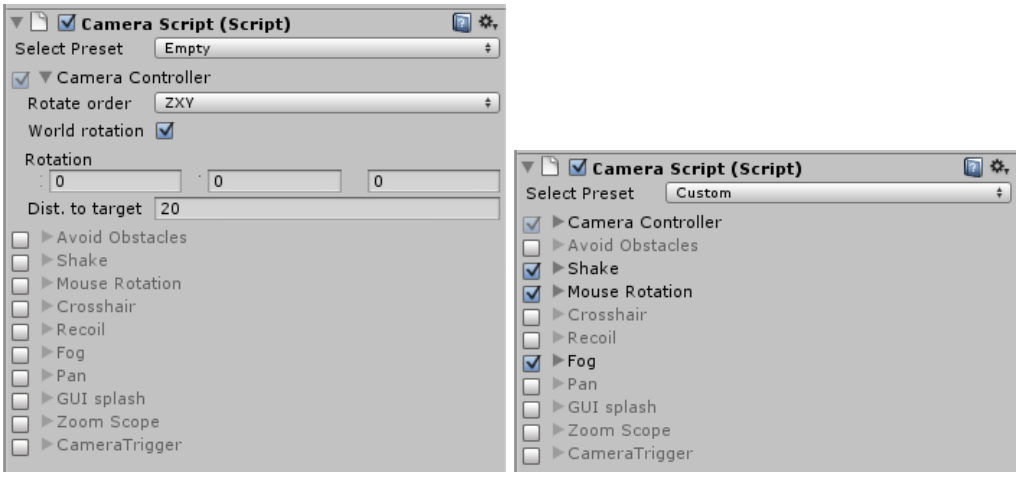


(a) CameraTool menu (b) Component menu

Figure 3.11: Screenshots from the Unity editor menu

It was considered only placing the tool in the Components menu, but was ultimately chosen to place it both places. This might result in menu clutter, if the designer has multiple tools all having its own place in the menu bar, but for most people, it will be a logical place to look for the tool, based on the Usability tests, Section 5.1. It was also chosen to implement a shortcut for adding the component to the camera, which was mostly done for ease of development, and chosen to let it become part of the tool.

When adding the CameraTool component to a camera gameobject, the initial interface is designed as illustrated in Figure 3.12.



(a) CameraTool menu (b) Component menu

Figure 3.12: Screenshots of the initial CameraTool component

Everything, with exception of the Camera Controller, is collapsed initially to not

confuse the user with unnecessary information right away. The Camera Controller is the main object of the component and cannot be disabled. The toggle button is kept in the interface but faded, as the usability tests showed users had a tendency to overlook it if it was collapsed and had no button, thinking it was a label. Figure 3.12 [B] shows the changes in appearance to enabled features of the CameraTool.

To allow for simple creation of a default camera, such as a first or third person camera, presets are at the very top, to allow the user to quickly change the type of camera to a desired preset, or leave it empty and setup a custom from scratch. The current presets, as of the DADIU March 2011 production, are illustrated in Figure 3.13.

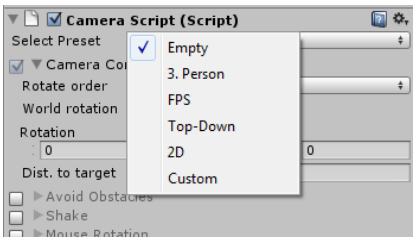


Figure 3.13: The available presets for the CameraTool

As shown in in Figure 3.14, functionality which is disabled is grey and cannot be changed. This allow for the user to quickly gain an overview of current functionality of the camera.

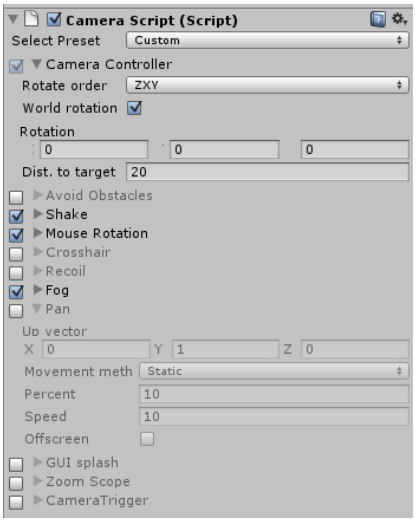


Figure 3.14: Disabled features are greyed out to gain overview of features not yet enabled

Features for the camera can still be expanded to show functionality while disabled. This is designed to avoid users having to enable functionality to inspect it. People are hesitant to add unknown functionality to the camera. This was found in the usability tests.

Figures 3.14 and 3.15 illustrates the use of whitespace and grouping in the CameraTool. Functionality for a specific feature is grouped in an expandable menu which is indented to easily notice what belongs to a certain feature.

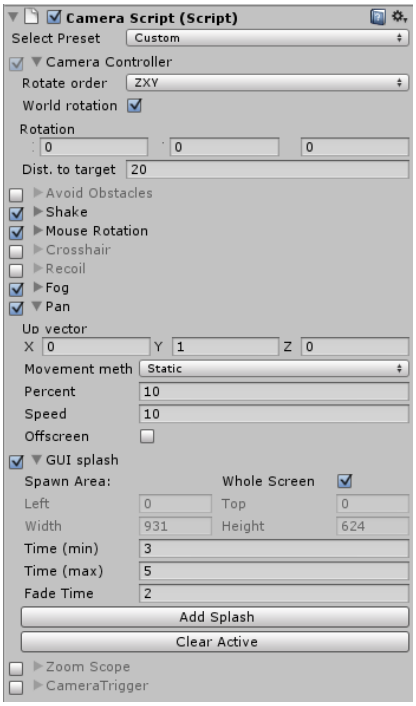


Figure 3.15: Component grouping to seperate features

To allow a user to quickly relearn the tool, as well as assisting first-time users or novices, most of the interface contains tooltips when the mouse hovers over a label. This is illustrated in Figure 3.16. The tooltip should only be visible after the mouse hovers over the object for a specific period of time. This is default Unity behavior, and is most likely designed to avoid annoying regular and advanced users.

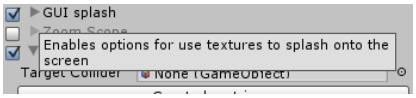
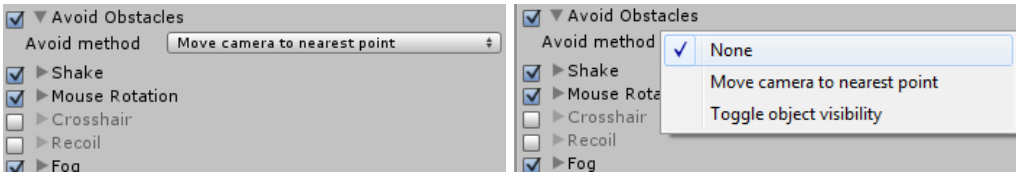


Figure 3.16: Tooltip for the GUISplash component of the CameraTool

For features where it was deemed possible, the components were stripped to a bare minimum, and only the essential feature remained. This was developed for a designer to easy pick and use. This is illustrated in Figure 3.17, displaying the component for avoiding obstacles.

Other functionality designed, which have not been mentioned in the design and usability sections, are the ability to test functionality in play mode, directly from the editor, as illustrated in 3.18, as well as the design of a dynamic menu for camera triggers, which expands or despands, depending on the triggers created using the CameraTool. The latter is illustrated in Figure 3.19



(a) CameraTool menu (b) Component menu

Figure 3.17: Screenshots of the CameraTool's Avoid Obstacles component

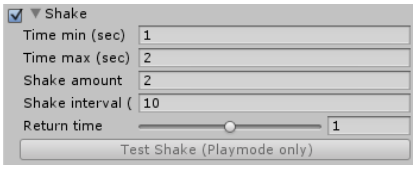


Figure 3.18: Test shake button only avalible in playmode

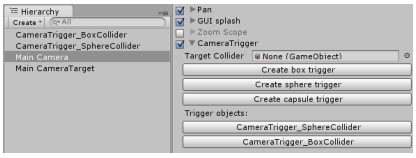


Figure 3.19: Screenshots of the CameraTool's trigger component

CHAPTER 4

IMPLEMENTATION

This chapter will describe the approach taken to implementing the needed functionality found in the Analysis Chapter, 2 in a way that addresses the design described in Chapter 3. The goal for the implementation is to develop a scalable and dynamic system that makes it easy to implement more features, while still keeping focus on GUI interface and user centered design regarding the end user. The different variables and functionalities needed for the CameraTool will be segmented and the best approach for implementing a constraint based system will be discussed. This includes looking into known design patterns used when implementing the different features. Finally a system for generating metrics for the in-field testing of the CameraTool will be presented.

In order to display the view of the game during game play, games use an update loop to update everything from objects positions and orientation, to render to the screen as well as taking player input into consideration. An example of an update loop can be seen in Figure 4.1.

As seen in Figure 4.1, the camera logic happens after most of the other logic in the game, according to Mark Haigh-Hutchinson [5]. This is done to make sure the camera end in the right position, to handle occlusion and/or collisions of the camera. According to the update loop, the camera update in the program execution pipeline is considered to be after all game logic has taken place, and before the rendering module. This also matches game pipeline described in [15] and is considered to be the most common method.

Unity works by having multiple event functions for doing frame and physics based scripting, according to the Unity documentation [54]. The execution order for the different event functions can be seen in Figure 4.2.

As the camera should be able respond to events and movement created as a result of user input, physics and game logic, the critical part of the camera movement logic should be placed in the LateUpdate function. The LateUpdate function is called after all rigid body movement, user input, update event and animations have happened, therefore making it ideal for camera logic and movement.

Nieuwenhuisen et al. describes a virtual camera with a position, an orientation and a zoom-factor [21]. Other basic parameters exist like projection type, sheer, view frustum etc. but since these, as well as the zoom-factor, are irrelevant for the camera movement they are ignored for now. The camera position can be described by using three variables for the position vector in the 3d space. Unlike the position, a rotation in a 3-dimensional

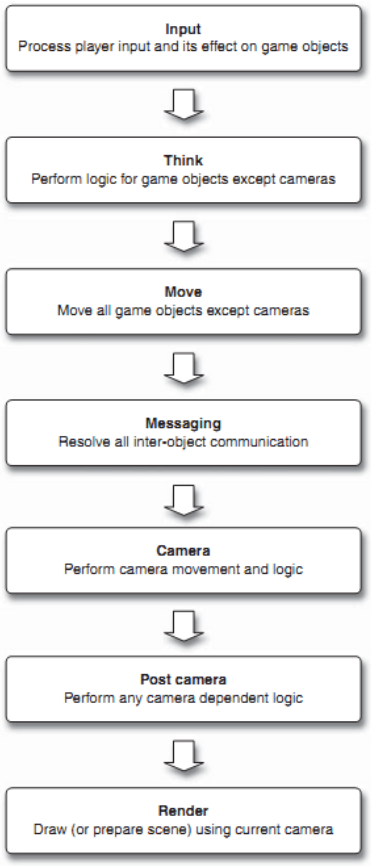


Figure 4.1: General update loop for a computer game

All Awake calls
All Start Calls
while (stepping towards variable delta time)
    All FixedUpdate functions
    Physics simulation
    OnEnter/Exit/Stay trigger functions
    OnEnter/Exit/Stay collision functions

Rigidbody interpolation applies transform.position and rotation
OnMouseDown/OnMouseUp etc. events
All Update functions
Animations are advanced, blended and applied to transform
All LateUpdate functions
Rendering

Figure 4.2: The execution order for event functions in Unity

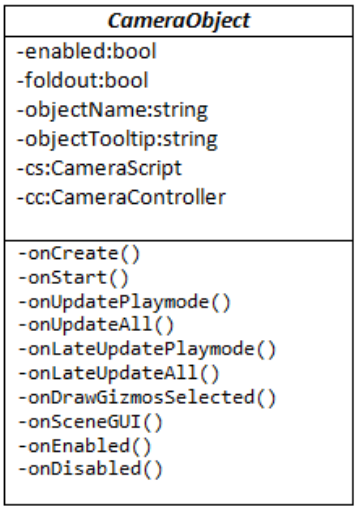


Figure 4.3: The structure of the CameraObject class parenting all components in the CameraTool

Euclidian space can be described in multiple ways. Nieuwenhuisen et al. uses three parameters to describe a point that the camera looks at, used to represent the viewing direction, as well as a parameter that describes the rotation of the camera around the axis spanned by this point and the camera position, also referred to as the twist or roll. This method uses a total of 7 variables to define a camera. Another way of representing the rotation would be to use Euler angles or Quaternions used by the native Transform component in Unity. Internally the CameraScript uses the 7 variables presented in the paper by Nieuwenhuisen et al., but the according to the design principals described in Chapter 3, Design, the tool should also support input in the form of Euler angles, as this is the native method used in the Unity game engine.

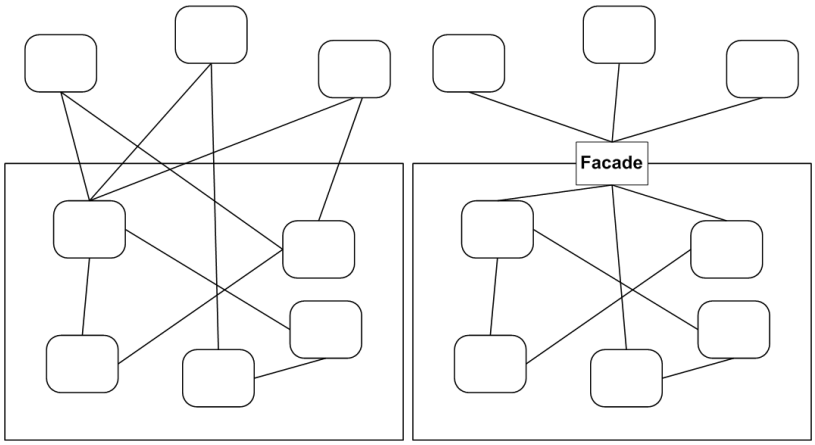
4.1 Design Patterns

To ensure that the code keeps organized and keep focus in reusability and scalability, different design patterns described by Gamma et al., are implemented [55].

Each of the functionality classes are implemented using polymorphism that allows data type and functions to be handled using a uniform interface. All classes extends a class, CameraObject, that defines a set of event driven functions as well as supplying some basic references to the CameraController and the CameraTool interface objects. The extended functionalities can be seen in Figure 4.3.

A controller class, called CameraScript, is used to instantiate and setup all CameraObjects. Since the class extends the unity MonoBehaviour all unity behavior functions are received and delegated to subclasses of the CameraObject type. Code listing 4.1 shows the delegation of the Update message event to all enabled CameraObject classes, and similar implementation is made on all other relevant events.

Code listing 4.1: Delegation of the Update message event to all enabled CameraObject



(a) Complex interface to a complex system

(b) More structured interface to a complex system through the facade pattern

Figure 4.4: Facade Pattern used to supply more structured programming interface to the CameraTool

classes

```
1  private void Update()
2  {
3      foreach (CameraObject co in cameraObjects)
4      {
5          if (!co.enabled) continue;
6          if (Application.isPlaying)
7          {
8              co.onUpdatePlaymode();
9          }
10         co.onUpdateAll();
11     }
12 }
```

Note that the function distinguishes and calls a specific function of the application is in playmode. This is due to the fact that the CameraScript class has been enabled to be executed in edit mode, allowing changes to the camera to be done in non-playmode. Similar delegation happens on all other unity message event function needed by the camera objects.

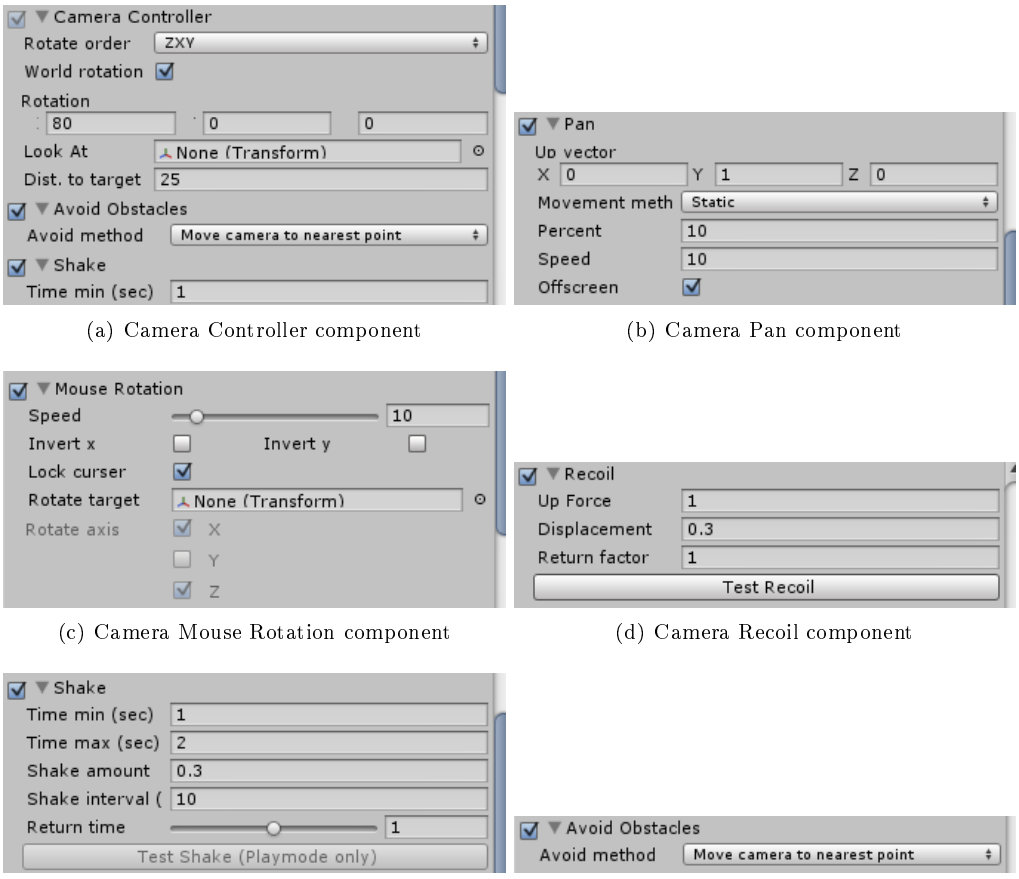
The CameraScript also functions as a facade structure creating a more simplified interface to the CameraObjects allowing a more control and streamlined interface available to the end user. It minimizes the need for communication and dependencies between subsystems. The facade pattern is illustrated in Figure 4.4

4.2 Components

The CameraTool should allow the designer to enable an arbitrary amount of camera features, which means that classes used for modifying the camera position and orientation, may overlap and access the same variables, creating dispute between the classes. This is solved by prioritizing the update routines in each class, based on the scope of the movement desired, starting from global scope and ending with local scope.

The features found in the analysis, Chapter 2 are divided into a set of classes, each responsible for a subset of the features. The classes are ordered in according to the scope of camera movement needed. Furthermore classes responsible for hard constraints, i.e. occlusion check, are listed last as these should be called last in each update event to ensure that the CameraTool does not break.

This section describes the features implemented in the CameraTool. Figures 4.5 and 4.6 illustrates the interface and functionality for the features.



(a) Camera Controller component

(b) Camera Pan component

(c) Camera Mouse Rotation component

(d) Camera Recoil component

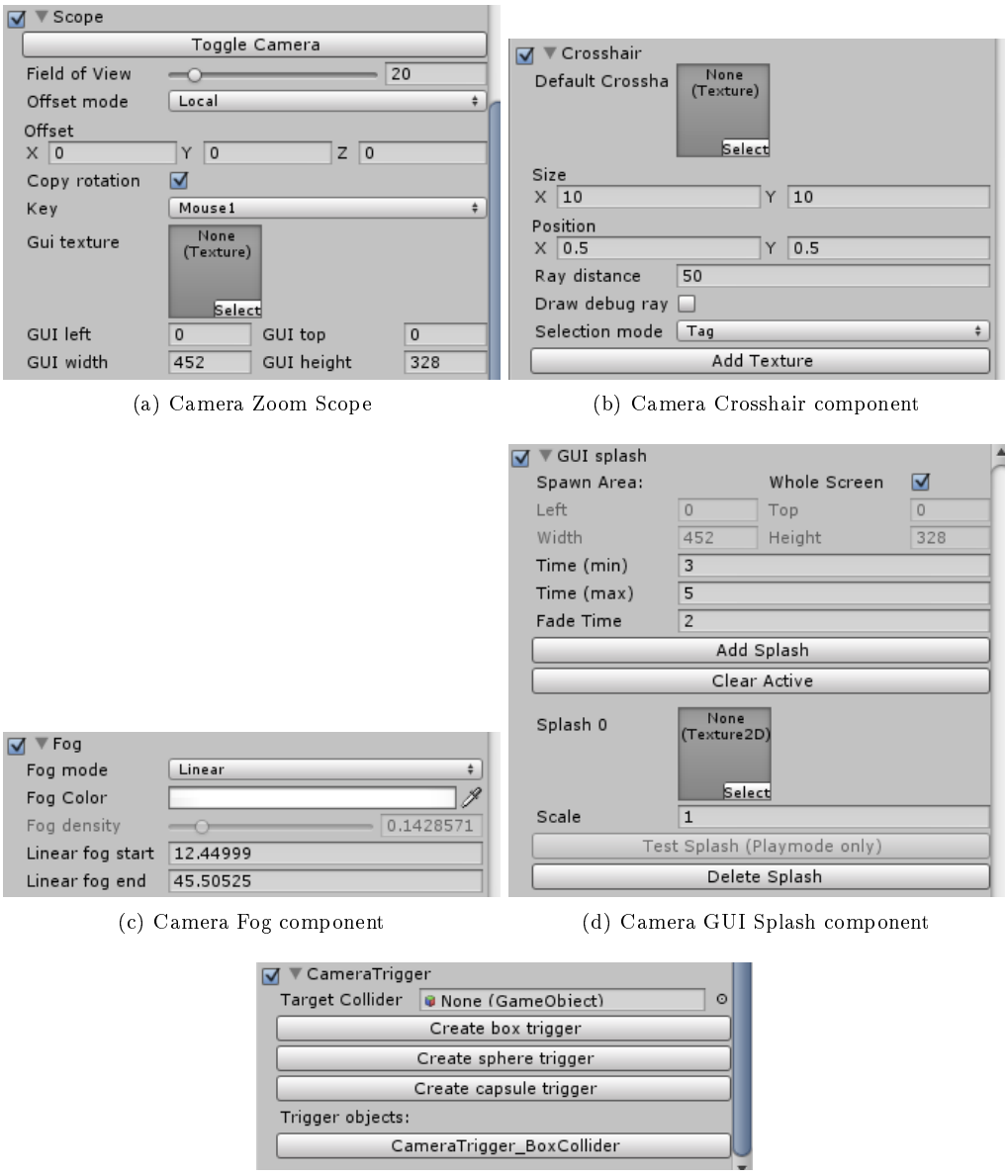
(e) Camera Shake component

(f) Camera Avoid Obstacles component

Figure 4.5: Illustrations of features implemented in CameraTool

4.2.1 CameraController

The camera controller is in charge of the most basic behavior coupled to the CameraTool and it cannot be disabled. It creates a point in the scene, represented by a red and white shooting target, that lets the user select a point of reference when positioning the camera. The position of the camera will always be relative to this point and the CameraController supplies the user with methods for affecting the distance and orientation to the target. Euler angles are used as input parameters letting the user set the desired rotation around each axis as well as choosing whenever the rotation calculations should be done in local or global space. Advanced users may also select the rotation order used to avoid gimbal



(a) Camera Zoom Scope

(b) Camera Crosshair component

(c) Camera Fog component

(d) Camera GUI Splash component

(e) Camera Trigger component

Figure 4.6: Illustrations of features implemented in CameraTool

lock if the default setting, ZXY, is not sufficient enough for the desired setting. The CameraController also features methods for changing the default distance to the target as well as allowing the user to zoom in or out based on user input.

4.2.2 CameraPan

The CameraPan class allows the designer to setup ways for the user to pan around the scene by letting the mouse touch the edges of the screen. This features is most commonly seen in RTS games described in the analysis, Chapter 2, but may also be used for other purposes. The class works by translating the position of the target, defined by the CameraController. It maps mouse movement near the edges of the screen to a plane in space defined by a normal-vector set by the game designer. The available variables features settings for translation speed as well as definitions on how much of the screen should be considered to be edge zones. Multiple ways of mapping the screen space to the movement plane are implemented. The standard static mode has 8 possible movement directions; one for each edge of the screen and a combination of these when the mouse is near a corner. The dynamic mode maps a vector from the screen center to the mouse position to the movement plane when the mouse is inside an edge zone. This enables the possibility to move the CameraTarget in all directions dependent on the mouse position.

4.2.3 CameraMouseRotation

To allow the game designer to setup the camera for the popular camera movement as seen in first and third person shooters, the MouseRotation class was implemented. The class works by changing the variables defined in the CameraController for the rotation thus maintaining the desired distance to the target. This allows for third person cameras to be created by setting a distance larger than zero in the CameraController. If a first person camera is desired, the distance can be set to zero, forcing the camera to rotate around itself. The available variables are used to define the rotation speed, whenever or not the mouse axis should be inverted and if the cursor should be locked to the center of the screen. The designer can also choose what axis should be affected by the mouse rotation.

The standard package shipped with unity includes a simple character controller [56]. This controller allows the designer to quickly add a character-like object into the scene with a simple collider and rigidbody component with the ability move through scripting. The component maps the up-key to forward movement, down to backwards and so forth. To support this component and others like it a rotate target variable is created. GameObjects assigned to this variable are automatically rotated and aligned with the projection of the camera look direction into the xz-plane. If used it means that the character will always move forward relative to the camera position.

4.2.4 CameraRecoil

Camera recoil is implemented to allow the game designer to simulate a recoil effect when the player shoots with a gun or likewise, most often seen in first person shooters. The effect simulates a short burst in an upwards direction, tilting the camera. The effect is implemented in a sandbox environment where the class uses an offset variable to define the total amount of tilt added to the camera, rather than changing the variable for camera direction in the CameraController. This makes the other classes unaffected by the recoil factor as this is only temporary. After the recoil has been added, the script will

automatically return the camera rotation to the unaffected state over a period of time dependent on the return factor. The recoil effect is meant to be called through scripting, but to allow the game designer to test the effect in the early stages a 'Test Recoil' button is added. This button simulates a call to the CameraTool telling it to add a recoil effect, which is useful for testing purpose.

4.2.5 CameraShake

The CameraShake has a lot of similarities to the CameraRecoil class. It only operates on offset variables, leaving the other classes unaffected in the next update cycle. The shake is implemented by translating the camera in a random direction. Every interval, defined by the designer, the random direction will change and the camera will be translated in the new direction. The game designer chooses for how long the shake should last and by what amount the camera should shake as well as the time it takes for it to return to normal after the shake has finished.

Like the CameraRecoil this function is meant to be triggered through scripting, but a button for simulating a shake-call has been added making it easier to test and tweak the correct shake.

4.2.6 CameraAvoidObstacles

To avoid the target being occluded due to objects placed between the target and the camera position, CameraAvoidObsticles class is implemented. The class is the last class modifying the movement and rotation to be called in the update cycle. It works by shooting a ray from the target position towards the camera position. If this ray intersects any colliders the class uses the avoid method defined by the designer. This includes moving the camera to the intersection point thus ensuring that the target is not occluded or disabling the visibility of the occluding object(s). The latter is temporarily implemented for testing. The final version should send a message to all occluding objects allowing the production team choose and implement the resulting reaction.

4.2.7 CameraZoomScope

The function of the CameraZoomScope is to simulate the use of binoculars or a gun scope. It is implemented by adding a second camera to the scene and allowing the production team to switch between these through scripting or bind a key in the interface. The scope camera has the same position as the default camera, with the ability to add an offset, and copies the rotation as well if needed. The field of view on the scope camera can be changed as well as adding a GUI texture, only visible when rendering through the scope camera.

4.2.8 CameraCrosshair

The CameraCrosshair class allows the game designer to add a default GUI texture drawn on top of the rendered image. The size and position of the image can be specified in the inspector. Furthermore the class enables the game designer to specify different textures to be drawn, dependent on what the user is targeting. The system uses Unitys build-in tagging and layering system to distinguish between targets by shooting rays from the camera position through the center of the screen and draws the appropriate texture defined by the game designer. The system supports multiple targets allowing the

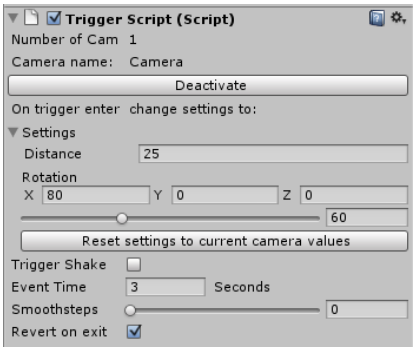


Figure 4.7: CameraTriggerScript component for triggers in Unity

designer to switch texture when the player is targeting specific objects i.e. friendly units or intractable objects.

4.2.9 CameraFog

The CameraFog class simply taps into the Unity build in fog system in the render settings. The system does not add any additional features, but works as a shortcut for the game designer to add fog to the camera in the game. See Unity's documentation on Render Settings for reference [57].

4.2.10 CameraGUISplash

The CameraGUISplash uses the GUI system in Unity to display textures on the screen for a specified period of time before fading out and disappearing again. The system is meant to be used to display things like splatter in the form of mud or blood on the screen. Since the splatter created externally from the CameraTool the class features an interface allowing access to the functions through scripting. Like the other classes, the game designer has the ability to simulate trigger calls in the interface which can be used to test the effect before implementation.

4.2.11 CameraTrigger

The CameraTrigger features a method for changing the settings for the CameraTool over time when a collider enters a specific trigger zone. The class works by instantiating a box, sphere or capsule trigger zone and adding it to the scene. The trigger can be moved into the desiret position in the level. When a target collider, defined in the inspector, enters the trigger zone all settings will be set to the defined variables defined in the trigger zone over a period of time. The script uses a linear interpolation between variable settings, but supports smoothsteps to smoothly interpolate between the two states [58].

When the collider enters the trigger zone a snapshot of the current state of the camera is created to allow the system to revert changes when the collider exits the zone again. The component added to the trigger are illustrated in Figure 4.7

CHAPTER 5

TEST

This chapter describes the process of testing the CameraTool. The interface and interaction of the tool were tested and iterated prior to the DADIU production, March 2011, using usability tests testing the design of the interface, as well as the user interaction. During the DADIU production the tool was offered to all production teams, to ease their camera creation process. The CameraTool was collecting data in order to investigate the usage of the tool in the production teams. After the DADIU production, interviews were conducted with game designers and game programmers from different teams, in order to get a deeper understanding of their camera creation process, as well as feedback on the use of the CameraTool, if they used it.

5.1 Usability Tests

The interface for the CameraTool has been iterated based on usability tests, conducted on people both with and without any Unity experience. All people tested are familiar with games technology and the game creation process, making them suitable as test participants, as no game designers or directors were able to test, due to residing at different and far away locations.

The purpose of the usability tests is to observe use of the tool, in order to understand how to support the tasks and goals of the end user [46]. It is not to disprove one interface or another, but to provide valuable information of the current design of the interface and interaction, to allow for making a judgment based on past experience in designing tools [2]. The usability tests do not focus on the functionality of the tool or the implementation, only on the actual GUI design and user interaction with the tool. The outcome of the usability tests should be an easy-to-use interface, which is intuitive to use by a game designer.

The usability tests were conducted in two rounds, with an initial pilot test, in which the test participants were given tasks and based on the performance of the user, it was discused the pros and cons of the interface and interaction. For the second round, the screen was captured as well as microphone input, using Camtasia Studio [59]. Notes were taken for the tests as well, by an observer. The notes and video material for the tests can be found in the Appendix.

5.1.1 Test Method

The method used for the usability testing is as described by Krug [2], lost-our-lease testing. The difference between traditional testing and lost-our-lease testing can be seen in Figure 5.1.

	TRADITIONAL TESTING	LOST-OUR-LEASE TESTING
NUMBER OF USERS PER TEST	Usually eight or more to justify the set-up costs	Three or four
RECRUITING EFFORT	Select carefully to match target audience	Grab some people. Almost anybody who uses the Web will do.
WHERE TO TEST	A usability lab, with an observation room and a one-way mirror	Any office or conference room
WHO DOES THE TESTING	An experienced usability professional	Any reasonably patient human being
ADVANCE PLANNING	Tests have to be scheduled weeks in advance to reserve a usability lab and allow time for recruiting	Tests can be done almost any time, with little advance scheduling
PREPARATION	Draft, discuss, and revise a test protocol	Decide what you're going to show
WHAT/WHEN DO YOU TEST?	Unless you have a huge budget, put all your eggs in one basket and test once when the site is nearly complete	Run small tests continually throughout the development process
COST	$5,000 to $15,000 (or more)	$300 (a $50 to $100 stipend for each user) or less
WHAT HAPPENS AFTERWARDS	A 20-page written report appears a week later, then the development team meets to decide what changes to make	The development team (and interested stakeholders) debrief over lunch the same day

Figure 5.1: Comparison of test methods [2]

The number of test participants should be three to four per test, and the more tests conducted, the better. This is also illustrated by Krug, as seen in Figure 5.2, where the number of problems found and fixed may be larger, due to the errors from the first test being fixed.

The test of the interface was conducted using a hi-fi prototype of the CameraTool during both of the test rounds. The interface for the tool was created in parallel with the functionality, so it made sense to conduct the test in the Unity editor, using the hi-fi prototype.

During each iteration, test participants were given a set of tasks to complete, covering most of the interface. The interface was then iterated based on the input from the users and evaluations of the tests.

The users chosen for the tests, varied in experience with Unity and in game design in general, and only a few had any experience with camera setup before. This was done to provide information from beginners as well as advanced users of both Unity and camera setup, as this is considered that the average game designer does not necessary knows about camera design or terminology.

During the first round of testing, only notes were taken by an observer for documentation and later iteration of the interface. The second round of tests has been documented using screen-capture during the test, as well as taking notes simultaneously by an observer. The notes and screen-captures can be found in the Appendix.

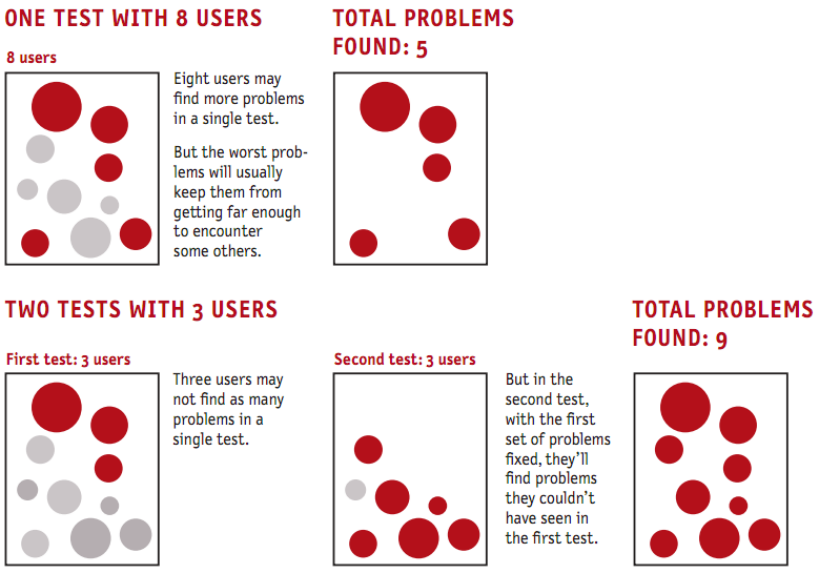


Figure 5.2: Errors found, depending on number of tests [2]

5.1.2 Test Design

The test was created to evaluate both interface and interaction. The user was given tasks to complete, similar to tasks an end user will have, creating a camera. The tasks are complex, such as creating complete first and third person cameras. These tasks are given, as they will resemble a train of thought a game designer will have for a camera. It allows for testing the interaction of the system as well as the entire interface design. Also, it allows for evaluation regarding the Gulf of Execution and how to handle the gap between intend and actions needed to setup a camera system. More simple tasks are given to evaluate the intuitiveness of features in general.

In order to get an understanding of the user's mind when interacting with the CameraTool, the tasks given should resemble tasks a game designer would have in a production setting, such as creating a first person camera for a shooter game. The camera should be attached to the avatar in the game, and the system might make use of camera shake on explosions, shaking when running, recoil when firing, a zoom scope for sniping, etc. The tasks given are to setup complete camera systems with the functionality available at the time of test, and play around with the created camera in Unity. In order to gain an idea of the process in the users mind, from a concept, such as camera type, to actions needed, the test participants are asked to create the camera setups without using predefined cameras from the presets. The scene created for the purpose of the test contained of a ground plane, on which pillars were placed as obstacles and a first person controller native in Unity, the user could move using W, A, S and D keyboard controls . The user should create the camera setup in this scene and play around in the world. The pillars can be used as obstacles for the Avoid Obstacles script, as friends/enemies for the Crosshair script, etc. An illustration of the scene can be seen in Figure 5.3.

In order to advance the difficulty of the test, to get around most functionality of the tool, the first question after adding the CameraTool component to the camera game object, are to setup a camera for a real-time strategy (RTS) game. This is considered to

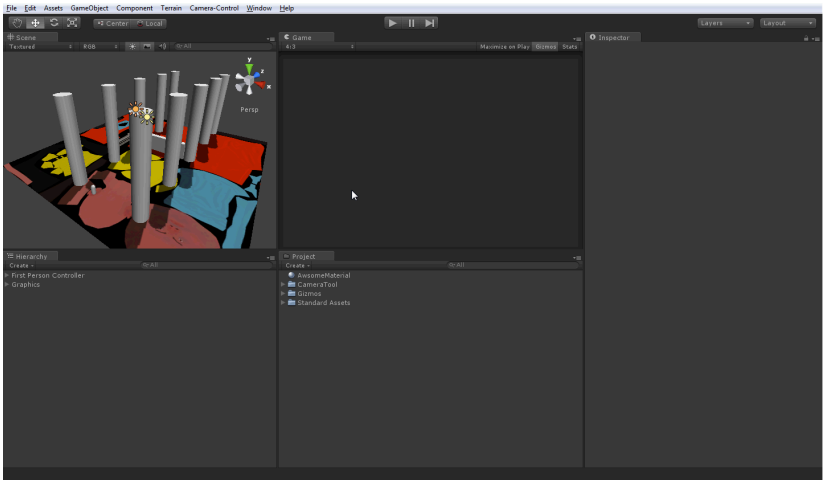


Figure 5.3: Screenshot of the scene created for the usability tests

be one of the simpler cameras, in terms of functionality needed. The camera will need pan using the mouse controller input, as well as setting an up-axis, which is default the Y-axis, setting the angle and distance to the plane. And that is it for a basic RTS camera.

Following the RTS camera, the test participant is asked to reset the component and setup a camera for a common first person shooter (FPS) game, with a crosshair texture in the viewport. This question is followed by changing the default settings for the crosshair to change, based on the target in sight. This task requires some knowledge of Unity as it involves understanding layers and/or tags in Unity. The task of creating an FPS camera is harder than the first task, as this time the cameraTarget should be attached to an actor already equipped with Unitys default CharacterController script, as well as changing the look distance of the camera to zero, to get the first person feel.

The last camera type to be created is the third person camera, following the actor, as seen in games such as i.e. Tomb Raider. For this camera, the test participant should make use of much of the same functionality as for the first person camera. However, the test participant should this time set look distance to something greater than zero, and the camera should not use a crosshair. It should include functionality of mouse rotation. Having created the base third person camera, the camera was expanded in the remaining two tasks, using avoid obstacles on the camera to keep the avatar in the view at all times, as well as creating trigger zones the avatar can trigger upon entering the zones. The triggers are, in the case of these tests, considered to be used along with shake for the camera, and trigger shaking of the camera upon entering a trigger zone.

A full list of tasks to be completed during the tests, along with the functionality for the tool the task should cover, are:

- Create a camera in the scene, and add the CameraTool component
  - Make use of one of the menu bars to append the CameraTool to the Camera game object
- Create an RTS style camera (of your own choice) without the use of presets
  - Toogle Pan functionality and perhaps play around with the settings
  - Change settings of the CameraController, such as look distance and rotation of the camera

  - The scene should be tested in play-mode
- Create a complete first person style camera with crosshair, without the use of presets
  - Change settings of the CameraController, as look distance should be zero
  - The cameraTarget should be set as a child to the player avatar
  - Crosshair functionality should be enabled
  - Mouse Rotation should be added to control mouse input for the player avatar and camera
  - The scene should be tested in play-mode
- Change crosshair based on target in sight
  - Crosshair settings should be changed to change target based on layers/tags
  - The scene should be tested in play-mode
- Create a third person camera (of your own choice) following a character, without the use of presets
  - Change settings of the CameraController, which is always enabled, as look distance should be a non-zero value
  - Mouse Rotation should be added to control mouse input for the player avatar and camera
  - The cameraTarget should be set as a child to the player avatar
  - The scene should be tested in play-mode
- Add avoid obstacles to the camera and test the different methods
  - Avoid Obstacles should be enabled, and one of the methods chosen
  - The scene should be tested in play-mode
- Add triggers events to the camera, and play around with them
  - Camera Shake functionality should be enabled
  - Trigger zones should be created and placed in the scene
  - The scene should be tested in play-mode
  - Trigger settings to interpolate variables should be set

The setup for the tests consisted of the test participant being placed in front of the computer running Unity, with the scene constructed for the purpose of the test. The test computer ran Camtasia Studio to record screen activiy and microphone input [59]. The test leader then provided the test participant with the task of setting up the particular camera system. An observer observed the test and took notes during the course of the test. An illustration of the test setup can be seen in Figure 5.4.

5.1.3 Results

This section will summarize the results from the testing of the interface and interaction, as well as general notes from the tests.

In order to get an understanding of the difficulty of the tasks in the iterated test, illustrations in Figure 5.5 shows the time for the test participants to complete the assignments given, as well as the amount of errors or mis-clicks occurring during the tests. This was only done for the second test, as there are not any video recording of the initial pilot test, and time to complete tasks were not measured, as the key to the first test were dialog with the user, and getting an understanding of the interface design.

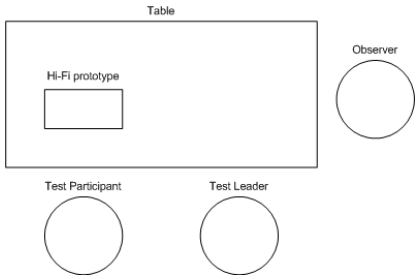
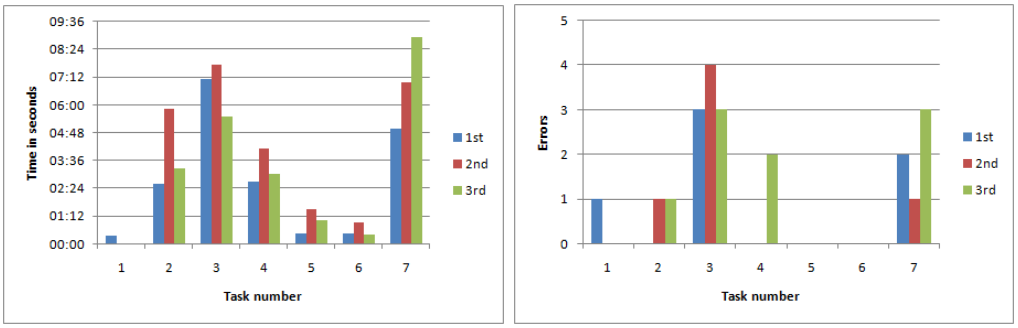


Figure 5.4: The test setup for the usability tests



(a) Time spent on assigned tasks

(b) Errors occuring during assigned tasks

Figure 5.5: List of errors from the second usability test

Given the time illustrated in Figure 5.5, it would seem that the difficulties for the set of tasks are in the correct order. At first, the interface is introduced, and they are asked to setup an RTS style camera. Time would here be assumed to be spent on navigating in the interface, looking for and enabling functionality associated with an RTS style camera. The testing also shows this to be the case, using 2:30 minutes to 6:00 minutes to quickly navigate the interface and setup a camera system for an RTS style game. Looking at the errors made, illustrated in Figure 5.5, in setting up the RTS style camera, the error made by two of the test participants were that try manually tried to move the camera, and not do so using the interface, as the camera position and orientation was locked.

Setting up a first person camera system, the task in assignment three, proved to be more difficult. In average, the test participants spent more time on this assignment. In all fairness, the features needed for this was higher, but the interface should at this point be known to the user. The main problem here was the difficulty in understanding that the camera follows the target, and that placing the target inside the First Person Controller is key to solving the task. As illustrated in Figure 5.5, this assignment caused more errors form the user interaction than the previous. The test participants thought the target could be changed in the editor settings and not manually in the hierarchy. The users not knowing the difference between Scope and Crosshair also caused them to spent time with the wrong feature. Changing the crosshair based on the target in sight was not really any problem for the test participants. The time on this task was mainly spent on playing around with the features of the crosshair and camera tool in general. One user had problems, as he tried to find a solution by looking for a "Set target" functionality in the settings for Crosshair. For him, adding a subsequent texture for the crosshair did

not make immediately sense.

At this point, before assignment five, most of the basics of the CameraTool had been gone through, and setting up a third person camera was error free and easy for all test participants, and so was adding the Avoid Obstacles functionality and testing it. Both these tasks were also quickly completed. The last assignment, adding triggers to the scene by using the CameraTool, caused both a number of errors as well as it took a lot of time to complete for the users. The reason for this might be that the areas of the assignment were new, as it was not only functionality for the camera that should be toggled and set, but a trigger system that should be created and set. As the users previously did all the interaction from the CameraTool in the previous assignments, they thought that interfacing with the triggers also happened through the CameraTool interface. As that was not the case, this caused some errors in interaction from the user. Also, the users did not notice the warning produced when trying to shake the camera, with the functionality not enabled in the CameraTool. This might be because of the test setup; the users are more focused on the tasks and the test, and might not notice general warnings or errors in Unity.

Figure 5.6 displays the errors/faults happending during the tests. It should be noted, that by "error" it is not meant as errors by the user, but as difference in understanding of the user and intent of the developers and the system. By noticing these differences, the tool can be changed to better suit the mind of the end users.

	1st	2nd	3rd
1	1) Attempted to add the component via script	N/A	N/A
2	N/A	1) Attempted to move the camera and not the target	1) Attempted to move the camera and not the target
3	1) Was looking for a feature to change cameraTarget in the interface 2) Got Scope and Crosshair mixed up 3) Did not realize that the camera is infront of the FPC and did not set distance	1) Moved the camera onto the First Person Controller 2) Clicked Empty Preset as a mistake and reset the camera 3) Was looking for a feature to change cameraTarget in the interface 4) Attempted to center the camera on FPC by moving the target to make it fit	1) Tried using constraint rotation in CameraController 2) Was looking for a feature to change cameraTarget in the interface 3) Got Scope and Crosshair mixed up
4	N/A	N/A	1) Did not consider creating new texture for the new target
5	N/A	N/A	N/A
6	N/A	N/A	N/A
7	1) Did not understand the target in the Triggers menu 2) Did not notice the warning that Shake was not enabled	1) Did not notice the warning that Shake was not enabled	1) Attempted to create a trigger using a new GameObject 2) Did not notice the warning that Shake was not enabled 3) Believed the settings for the trigger was in the CameraTool interface

Figure 5.6: List of errors/faults from the second usability test

General Notes

All test participants managed to complete all tasks, but needed assistance solving steps along the way. People did not notice the Camera Controller functionality, as it did not have an enable/disable toggle, due to always being enabled. This lead people to not notice it, as it were different from the functionality they did notice, and only looked

through those with toggle option. This was changed so users to make users aware of the Camera Controller. However, it does not have any functionality, as it is always enabled, and cannot be disabled.

As mention by Scott Goffman [8], the menu should be placed where it makes sense to the user. This was noticed during the tests, where the participants added the CameraTool from both places. Even though the CameraTool menu is displayed in the menu bar, it makes sense to look under the Component menu.

After the first couple of tasks at hand, people started understanding the tool and had little problems navigating the inspector, enabling and disabling functionality. Some even went out of the scope of the task playing around with functionality.

In general, the test participants managed to solve the tasks by muddling through the interface, clicking and testing where it made sense to them. One thing that no participant understood was the Camera Scope feature. The word Scope is too broad, and no one really had any idea what this was. The name was changed to Zoom Scope afterwards, to avoid confusion. In general, people tend not to click something they do not know what is.

The empty preset functionality to reset the camera, was toggled by mistake by a test participant, and the camera was reset. It is generally a good idea to be able to quickly reset a component when used correctly, but have unwanted consequences when done by mistake. The user should either be warned or have the ability to undo.

Following the usability tests, most of the interface for each component was re-created, focusing on a more designer-friendly approach, as some of the terminology might have been too computer-programmer-ish. Labels and tooltips were re-created for the entire tool, to hopefully be more of assistance to the user. The items not enabled were faded out and made expandable without enabling the feature first. As users mentioned, it should be possible to see settings for the feature before enabling it.

5.2 Data Gathering During the DADIU Production

In order to get as much information of the use of the CameraTool during the DADIU production, a function for analysis of use has been implemented in the CameraTool. The idea is to investigate the usage of the tool during the production in order to see what elements have actually been used, and to notice any difficulties in using the tool or single components.

The data gathering is implemented by sending HTTP POST requests to a server hosting a PHP script. The PHP script then connects and stores the data in the POST method to an SQL one-to-many relation database. If available, the name of the computer running the CameraTool is used as a unique identifier. If the computer name is not available a random integer between 0 and MAX[1] is associated with the instance of the tool and used to identify users.

The gathered data includes:

- Computer name
- Class name from which the call is invoked
- Function name from which the call is invoked
- Arbitrary label used to categorize calls
- Arbitrary descriptive text
- Float value for value oriented data

[1] int.MaxValue: 2.147.483.647 [60]

- Any error messages caught by exception handling
- Timestamp for when the data was send from Unity
- Timestamp for when the data was received by the server

Class and function tracking is useful to see what features and functions are most popular as well as seeing if any functionalities, for some reason, is not being uses as intended. The arbitrary label is used to distinguish between tracking events called from within the same function. This could for example be the native unity Start() function that is called when the user enters or exits playmode. An arbitrary text can also be attached to the log-call to add a description and specifications if necessary. In the same way as the text, a number can be tracked using the value property, effective to track the current state of a changed variable. Whenever an exception is caught in the code, the error is automatically logged in the database using the text property to log the error description. Lastly the system logs timestamps from the Unity game engine as well as the current server time. A system for logging all GUI functions, such as the press of a button in the inspector or when a function is enabled, is implemented in the CameraTool. Furthermore all Enter- and Exit-playmode events are logged. The illustration in Figure 5.7 shows an example of some data gathered from a system using the CameraTool during the production.

id	computerName	mac	callClass	function	label	text	error	value	unityEpocTime	timestamp
18	dadiu2-PC		CameraScript	Start	Start	Application is startet with the camerascript enabl...		0	1299078728	2011-03-02 14:12:08
19	dadiu2-PC		CameraScript	Start	End	Application has ended with the camerascript enable...		0	1299078763	2011-03-02 14:12:43
20	dadiu2-PC		CameraAvoidObstacles	onEnabled	enabled	Camera Avoid Obstacles has been enabled		0	1299078774	2011-03-02 14:12:53
21	dadiu2-PC		CameraShake	onEnabled	enabled	Camera Shake has been enabled		0	1299078774	2011-03-02 14:12:54
22	dadiu2-PC		CameraMouseRotation	onEnabled	enabled	Camera Mouse Rotation has been enabled		0	1299078775	2011-03-02 14:12:54

Figure 5.7: Section of the raw data gathered during the DADIU production 2011

5.2.1 Results

The data collected during the DADIU 2011 production showed 9 unique computers that had the CameraTool installed, whereas it was possible to identify 3 of them as being members of the Branch team by looking at the tracked computer name. Unfortunately due to a bug fix during the production, it became impossible to track the computer name. The random high number used as unique identifier was introduced and associated with the instance of the CameraTool. This means that large portion of the 9 unique computers logged, probably belongs to the Branch team since each identifier were reset during the bug fixing. In general it appears that only a few computers have supplied data amounts above a threshold that can be considered to be serious use of the CameraTool. The computer on the Branch production with the initial installation of the CameraTool, used by the game designer, was named kbak-PC which is heavily represented in the dataset.

Since a large portion of the data origins from within the Branch production, it was chosen to discard all data not collected by the computer used by the game designer.

The data shows that the CameraTool was extensively tested during the first 2-3 hours after being presented with the tool. The illustration in Figure 5.8 shows a visual representation of the use of the tool over time. The Figure shows what features has been enabled (green) or disabled (red) as well as when the project has entered or exited play mode. Although it is hard to classify patterns and trends in the use of the CameraTool, having only one test person, it can be seen that most features of the CameraTool has been enabled at some point during the initial 3 hours after installation.

If the CameraTool were to be introduced to a larger audience, like the Unity Asset Store [61], it is believed that the system implemented would supply sufficient amounts of data to create a more detailed description of the use of the CameraTool as well as error occurrences. The system might even show a link between the camera setup and the game genre. The logging of the interface can be used to see if any features are hidden or less used due to a possible design problem as well as noting what features are quickly disabled after enabling them, indicating that the features were unwanted or too complicated to use.

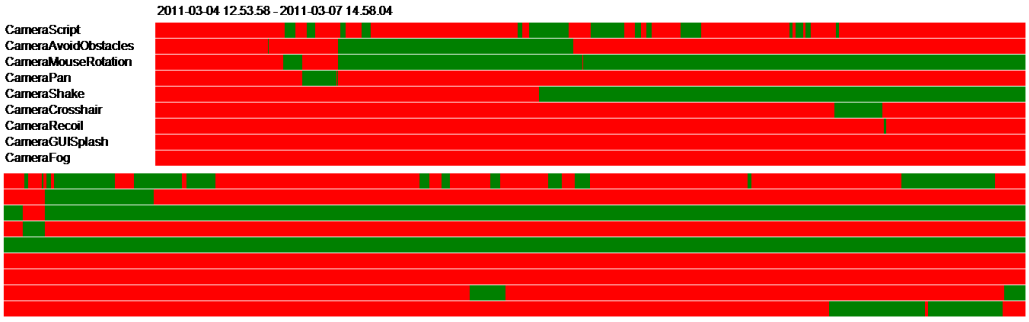


Figure 5.8: Tracking data from the first three hours of work with the CameraTool during the DADIU production. The red color represent periods where the features are disabled and green represents enabled features

5.3 Interviews

In order to maximize the amount of information gained from the use of the CameraTool, game designers, game directors and game programmers from the different DADIU teams have been contacted in order to conduct an interview investigating the usage of the CameraTool in their productions, as well as general questions to camera creation and the productions. Interviews were conducted with people from four of the six different DADIU production teams from 2011.

The interviews were constructed as unstructured interviews [46] and designed according to the person interviewed. As the game designer of Branch was observed during the production, some knowledge of his use of the CameraTool was gained before the interview. This allowed for more focused questions as to how the tool was used and what thoughts were put into it. The questions were designed to gain information on how he felt the tool affected the production and what frustrations he might have encountered during the setup of the camera. This included whenever, or not, he felt like some fea-

tures were missing and if the tool assisted or limited his creativity. Furthermore it was sought to clarify if the design of the interface made sense and how the tool in general felt to use in the production. As this is an unstructured interview the questions asked is meant to lead the interviewee to talk about specific subjects and if the topic is interesting more detailed questions can be considered, an the interviewee is asked to elaborate. The questions should not be kept strict but should rather encourage free talk and let the interviewee lead the conversation and only supply new topics when necessary.

It should be noted that during the DADIU production, the game designer was given the CameraTool along with a minimum of instructions on how to install the tool. No additional help were given on how to use or setup the tool, which allowed the game designer to gain his own opinion on how to use the tool. This meant that feedback on the intuitiveness of the CameraTool could be gained as a follow up on the changes made due to the usability test described in section 5.1.

The questions asked to the game designer on Branch, are:

- Did the tool assist you in creating the camera you wanted for the game?
- Did you notice any limitations in the tool?
- Did the use of the tool give you any new ideas for the camera you hadn't considered yourself?
- How was the tool to use? Did it make sense, and was it intuitive?
- How much time did you spend on setting up the camera for the game?
- Would you prefer the way we designed the tool, or Unity's component based system, by dragging individual scripts onto the camera?
- Did you feel that the tool gave you freedom in creating the camera, or that it locked you into a specific way of working?
- Any other comments?

A more general approach was taken to evaluate the use of the CameraTool in the other DADIU teams since its unsure how big a role the tool played in the production of their games. As with the interview with the game designer of Branch, the questions were also designed to be used during an unstructured interview. The main focus was to elaborate whenever the tool was used or why they choose to neglect it. If they played around with the tool or used it in the production, relevant questions used in the earlier interview should be used as deemed relevant. On the contrary, if the tool were not used or tested, the focus should be changed as to why not. Better methods for promoting the tool if the process were to be reproduced, should be discussed to gain ideas for better distribution in the future. This could include video tutorial, demo scenes etc. and the preferred methods should be noted. At last the interviewee's attitude towards a generalized method for camera setup should be looked into and whenever or not past experience proves or disprove the need for at CameraTool in a game production environment like DADIU.

The persons included for the interview included the game designer from Blendimals [62], game programmer from Raasool [63] and a game programmer from Broken Dimensions [64].

The questions asked to the other interviewees are:

- Did you use the CameraTool in the production?
  - If no, why not?
    - Was the camera design in your game the reason for not using it? Or unwillingness to look into the tool?
    - What should have been done in order for you to use it?

        - Video introduction, tutorial?
        - Anything else?

    - If yes, what features did you use?
        - Who used the tool? (Designer, Director, Programmer?)
        - And how long did it take you to setup?

- Did you consider using the tool to prototype a camera for your game?
- Any comments on the idea of having a tool like this?
- Did any past experience on camera creation affect your choice in whether to use the tool or not?
- Any other comments?

5.3.1 Results

This section summarizes the relevant answers to the interviews conducted. For full answers to the questions, see the Appendix.

The game designer on Branch mentioned that the creation of the camera setup was a lot easier than the first production, where he worked on Anton and the Catastrophe. All the basics for the camera were in the tool, ready to be used. It made it a lot easier to play around with and get a feel of the type of camera setup he wanted for the game, without having to rely on programmers to code the functionality for him first. It made the iteration of the camera setup faster and easier. He mentioned he loved the fact that all the features were available, and considered it a shame most of them did not fit the game play.

He did not have any problems using the tool, mentioning it was somewhat intuitive to use, though he would like more visual representation of some of the functionality in the tool. Also, some of the labels in the tool felt like it were written by programmers, and he had to try the features in order to see what they did. He mentioned the need for a simpler interface, with options for a more advanced interface, as he was somewhat confused the first couple of minutes, playing with the tool.

The time for setting up a complete camera as used in the final game only took a couple of hours, including playing around and getting a feel of the tool, as well as stress-testing it, seeing if it would break during game play. He mentioned that the tool felt robust and he did not feel the system broke when adding or removing features, or during run-time. He also mentioned he loved having an all-in-one tool and not miss out on any functionality.

The main results from the other interviewee's were that none of the interviewed production teams made use of the tool in their production. And only one of the interviewed mentioned actually trying the tool. This was primarily due to the simple camera system they considered for their game. Thus, there would be no reason to use the CameraTool, or even try it out.

Everyone considered it a good idea having the tool to use during a production and that it would help creating or prototyping the camera for the game. However, they also mentioned the need for a video displaying the features for the tool or even a tutorial video for them to actually use it, as they did not know the features of the tool.

It was also mentioned that the reason, in some cases, for the simple camera, was due to the experience from the first DADIU production, as well as focus being more on game play, thus keeping the camera as simple as possible.

General Notes

The results gained from the interviews will in this section be categorized and discussed based on observations made by the authors of the report and knowledge gained from other tests and sources used throughout the report.

The game designer from branch mentions that the camera setup for this year's production was a lot easier than the previous DADIU production he worked on. The fact that he finds it easier is a success criteria for the production of the CameraTool, but many other factors come into play. The last production he did was Anton and the Catastrophe described in the analysis, Section 2.2.2, which uses an advanced camera, compared to Branch. This might be because he learned about the work needed and complexity added when setting up an advanced camera, and just wanted to keep it simple, or the game play just did not call for advanced camera features. Whatever the reason, the fact is that the game designer was able to setup the camera for the game, without programmer assistance, while still feeling that the whole process went smoother, compared to last year. A potential big programming task had also been removed from the programmers as they did not need to be involved in the setup of the camera.

It was said that the tool included all the basic functionality a camera setup tool needs, which indicates that no immediate features were missing thus verifying that the tool covers the most basic functionalities of a camera system. It does however not necessarily mean that all advanced features included in the tool were sufficient, as some of them were not used in the game.

The CameraTool appears to make prototyping faster and easier. This is due to the fact that the tool can be operated without the need of a programmer. This naturally encourage creativity and help the designer to gain a better understanding of what type of camera system is best for the game. During the DADIU production it was observed that after the initial setup of the camera, the game designer, game director and a computer graphic artist spent time together to discuss and test different camera setups before finally ending up with the version included in the shipped version of the game.

The game designer believed that he used about two hours to setup the camera system for the game. This timeframe is confirmed by the data gathering described in section 5.2. The combined hours spend on the camera system in this years production is significant less than either of the productions the authors of this report participated in last year. Compared to the other DADIU 2011 teams it appears that they used the same or maybe a bit less time than last year, and their approach was the same: have a programmer to implement the system and let the game designer tweak the variables.

The tool were believed to be somewhat intuitive to use, indicating the tool did as it was supposed to do. This might be able to be improved by using a more visual approach to the functionalities in the CameraTool. Minimalistic pictograms or visual drawings might help the user understand the functionalities compared to a text-based approach. It was also mentioned that the CameraTool used programmer language when describing functionalities, meaning that some of the functionalities or descriptive text it too technical. Although this has been a priority to use a less technical language and variables to setup the camera, this could indicate that more work could be put into the language used. It should however be noted that the game designer previous had no experience or teachings as to the inner workings of a camera system. The fact that some features and explanations confused him could be considered to be part of the learning curve attached to learning to setup a camera. As mentioned by Norman [41], users muddle through features and functionalities that does not make immediate sense, which were also the case here. The camera system were created and worked, although not all

variables were understood by the game designer at the first impression.

All interviewees seem to think that a CameraTool is a great idea, although none of the other teams used the tool for their production. The general explanation for not using it was that the features and functionalities were not introduced properly beforehand, so they saw no advantage in trying it.

CHAPTER 6

DISCUSSION

In this project, the CameraTool for the Unity Game Engine was created. The vision for the project was to develop a tool, which would allow a game designer to prototype and/or setup the camera system for a game, without needing assistance from a game programmer to implement the needed functionality. The idea is, with camera setup being a combined technical and aesthetic task, the game designer and director should be free to prototype the camera system without being bound by time/programming restrictions.

The desire to develop a tool for designers to use, which frees up programmers to focus on other tasks, came as a reflection of the DADIU productions in May 2010, in which the authors participated as game programmers, respectively on Sophie's Dreamleap and Anton and the Catastrophe. In the 2010 productions, much time were spent on creating very specific cameras for the game, and these cameras were iterated during the productions in order to fulfill the constantly changing demands of the game designer or game director. That meant a lot of time which could have been used on improving the game play were spent on camera programming instead.

With this problem in mind, the problem formulation was:

> It is sought to develop an intuitive and easy-to-use tool for the Unity Game Engine, which allows game designers to setup a virtual camera system for computer games, without needing assistance by a programmer, containing most common functionality found in current computer games.

The problem was limited to creating a tool consisting of the basic features found in common games, in order for the game designer to setup the camera system for most game genres. The Analysis chapter, 2, described this process, as well as looking into recent studies in virtual camera systems. This would be the base for developing the CameraTool.

For the interface development, it was intended for use by game designers and thus should be designed with a non-technical end user group in focus. The development of the interface and the interface interaction was aimed to be as intuitive for the game designer as possible, using usability principles and general user centered design, which was described in detail in Chapter 3, Design.

In order to be able to iterate the design, the interface and interaction were tested using usability tests, as described in Section 5.1. The interface was tested on a target audience similar to the end user group, as no participants from the end user group was

able to participate in usability testing prior to the DADIU productions. The design of the interface was iterated based on the evaluation of the user tests, prior to the DADIU production. The second tests showed that especially the features of controlling the camera relative to the camera target, from the inspector, seemed to confuse people initially, as the camera is locked to the target. This was initially created to make it easy for people to setup camera systems which can act as both first person and third person cameras, by changing the distance to the target. Also, the naming of some features confused the users during the tests. As a result of this, most of the features were re-designed following the usability tests. The tests also shows that the later tasks, with exception of creating camera triggers, seemed easy as the users seemed to understand the interface after having passed the initial learning curve. The task of creating the triggers caused some problems, as the users should distinguish between trigger functionality and camera functionality, but they all managed to solve the task. As later usage by the game designer on the Branch production showed, he did not have any significant problem using and playing with the tool for the camera setup. As he was the only user to use the tool for an extended period of time, it cannot be concluded that the changes to the interface made a significant difference. However, it is believed that the current interface is an improvement to the one prior to the usability tests.

During the DADIU production, it was intended to collect data of the CameraTool usage, in a real production environment. The CameraTool was sent to the game designers in the DADIU productions in the beginning of the production period, for them to be able to use it during the production, if so desired.

The data gathered included usage of functionalities, as well as errors occurring in the CameraTool during the production. The data was sent to a web-server along with a description of the feature and a local computer time-stamp from Unity. Due to the low number of users during the DADIU production, the data collected was too vague to get anything meaningful from. However, it showed a potential in future usage, to collect information of use and errors occurring, to show a relationship between the games created, features used and difficulty of use of the features. This can also provide information of the interface and interaction, as it shows the use of the tool over time. Thus, the actions over time can be seen, to get a better understanding of the Gulf of Execution. It can also be noticed whether certain features are less attractive for the user, or features were unnoticed by the user, and thus never used.

The interviews following the DADIU production focused on expanding on the usage of the tool during the productions, regarding elements of the production that were easier, as well as problems with the CameraTool. It was also examined why certain teams decided not to make use of the tool. The game designer from Branch was interviewed as he created the camera for that production. A game designer and game programmers from other teams were also interviewed regarding usage of the CameraTool, as well as general questions related to the camera created for their game. In the interview, the game designer from Branch mentioned he found it to be great that the CameraTool included as much functionality, as he felt it was a rather complete tool, regarding features. Because of this, he was able to setup a camera without a programmer developing the functionality first. This made prototyping and camera design much easier and faster. The design of the interface seemed intuitive to him, as he had not had any training in the tool before the production, and yet he managed to quickly understand the interface and play around with different camera setups, before creating the final camera setup. However, in the re-design of the interface, post-usability tests, the labels and naming of the functionality might not have been sufficient, as the game designer mentioned some "programmer language" in the CameraTool. This is not definite, as he did not have previous experience with camera and

cinematography terminology in general. He mentioning in the interview that he wanted a more simple interface, with options for advanced features. This has been disregarded, as the interface for CameraTool are, compared to many of the default Unity components, believed to be as simple as it should be. It is considered that having a simpler interface will most likely only annoy regular or experienced users, as it might become too simple.

In general, the CameraTool was being used during a DADIU production, which was the desired outcome at project start, and the game designer managed to setup the camera singlehandedly, without needing assistance of a programmer. And not only was this managed without a programmer; the camera system was setup in only a few hours, compared to days of programming in earlier productions. Moreover, the interview mentioned that the camera system felt robust, even when being stress-tested by the game designer.

As the other interviewees mentioned, their productions did not make use of the CameraTool. They were asked for the reasons for this, and other reasons where speculated, based on prior experience. All interviewees mentioned their game having simple camera setups as the main reason for not using the tool. Based on previous experience at the DADIU productions, the camera setup has become notorious as a difficult subject at DADIU, and the consultation groups mentioning that the simpler the camera, the better. This can be an argument for why a tool such as CameraTool should be used in a DADIU production, as it frees programmers to focus on other elements of the production, without limiting the creativity by the game designer and director.

In order to get more teams to use the CameraTool, it should be considered creating a video displaying functionality of the tool, as well as a tutorial video to display the usage of the tool, in order to make them interested in using the tool in a production. Even if the production team feels the tool should not be used in the final game, the tool might work as a prototyping tool for the game designer and game director to prototype the type of camera desired for their game.

To sum up; the game designer of Branch used the CameraTool to singlehandedly setup the camera system for the Branch game, without assistance of programmers. Thus the tool was successfully used for a DADIU production, both as a prototyping tool and as a mean to setup the final camera for the game. The interface of the CameraTool appeared intuitive to the game designer, and after playing around with it, he had no problem in using the tool, and not even stress-testing broke the tool in the game. It is believed that the desired outcome of the project has been reached.

6.1 Future work

This section describes further work to be done for the CameraTool. Based on the evaluations of the tests and a general feature-list set by the authors of the report, certain features of the toolset should be implemented in the CameraTool, and some existing features should be re-created. This section will dive into this area, and elaborate on each feature.

Camera Target and Camera Manager

Two of the most fundamental features for the CameraTool, one to be re-created and one to be implemented, are the use of camera target and the creation of a camera manager for the scene. As mentioned in Chapter 5, Test, the use of the camera target was initially confusing for the test participants. It was not until after the initial learning curve they understood how to use this feature. The problem arises because the interface to the

CameraTool uses two objects to represent the camera: a camera and a camera target. The target is meant to always follow the avatar of the game, if one is present, otherwise it will be used to represent the viewing direction of the camera. The system differs from other camera systems in the way that the target often is not represented by an object in the 3d space, but rather a viewing direction of the camera. The initial idea was that the game designer should be able to set the camera to a specific distance to a target or avatar. For this to work, the CameraTool should be in control of the position of either the camera itself or the target of the camera. Since a camera system in games often follows an object or avatar, it would make sense to let the game designer have direct control over the camera target and indirect control over the position of the camera through the interface of the CameraTool. Unfortunately this meant that the camera object inside the Unity game engine have to be controlled by the CameraTool in order to maintain a fixed distance to the target. This contradicts the normal way of moving a camera inside Unity, thus generating confusion for the user. Through the observations and tests it was obvious that although the system initially is confusing for most users, it becomes much easier once the concept is understood.

To overcome the initial confusion, a setting could be implemented in the CameraController that lets the user decide what systems should be dependent. This would allow the user to toggle between having the camera position dependent on the target, having the target dependent of the camera position and having both independent. It might even support that the user selects an object, to be used as a target, in the inspector, instead of having to nest the camera target object with the object to follow. The system would then have to compensate with changing the GUI interface and graying out unavailable features, like changing the distance, if a method that makes it impossible to set has been chosen.

The idea behind the camera manager is to have a general class for controlling all cameras in a game, such that the transitions between cameras as well as camera setups become a more fluid process and cameras can be added easily on the spot.

The camera manager should inherit some of the responsibilities from the CameraScript as it is currently. The manager should manage the active camera and have a list of all deactivated cameras in the scene. This way, multiple cameras for different set of camera shots could easily be setup and transitioned between. The general idea of the camera manager is explained in Real-Time Cameras [5]. This would also give the ability to create shots based on cinematographic principles easier for games.

Cinematic cameras

As the related studies mentioned in Section 2.1, there are a number of features which can be added to the CameraTool in order to allow designers to setup more cinematographic camera systems.

One idea is to setup a system for automatic camera planning, using a probabilistic roadmap method [14] [16] [21]. This will require some pre-planning, but for static (or mostly static) scenes, it might make sense for the camera to follow a pre-planned route.

In relation to automatic camera planning, a user should be able to specify a path for the camera over time for certain events in the scene, using a keyframe based approach, much like animators keyframe movement of joints over time in a 3D environment. This would allow the user to create events like airplane flyby over an area, as well as motion for intro-sequences easily.

For obstacle avoidance, the camera could use a more intelligent approach, other than removing obstacles, making them transparent or even moving the camera closer. The

camera could, based on the surroundings, change position to a more desirable location while still have focus on the avatar for third person games. This could later be expanded to also take other elements of the scene, such as enemies or fields of interest, into account.

In-game cinematic sequences, such as conversational events, in which the avatar interact, which cuts between different over-the-shoulder camera shots, as seen in the film industry, is a feature that could be implemented as well, in the CameraTool. Using a camera manager as described would ease the process of creating this. This feature would allow the user to setup an in-game event, which starts a conversation between two or more avatars. The system should be able to place cameras either by the designers choice or automatic based on certain parameters given to the system. The designer should then specify a number of time steps, in which the system should toggle between the cameras used in the conversation event.

Visual interface

As mentioned in the interview following the DADIU production, Section 5.3, the game designer on Branch mentioned the wish for a more visual interface, in order to get a better understanding of the features in CameraTool. Pictograms in the interface could supply with additional information of the feature, as illustrated by example in Figure 6.1.

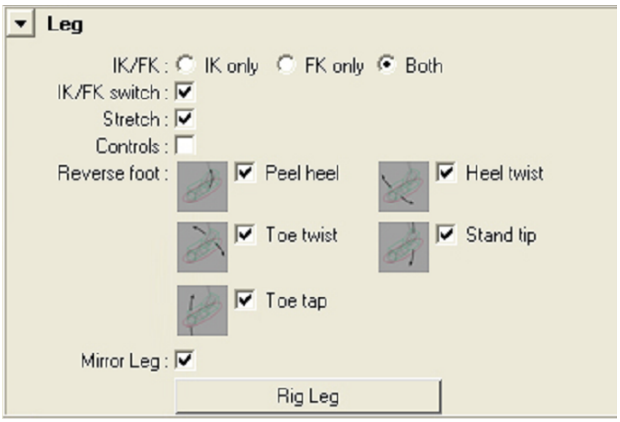


Figure 6.1: Example of pictograms used to illustrate subjects to give people an idea of a feature [3]. This illustration describes features for rigging in Autodesk Maya [4]

The tool could feature gizmos in the editor to better visualize settings in order to assist the user in setting up the camera and to visualize settings easily, without the clutter of the inspector. This will have to be designed for ease of use, in the same way the interface has been designed, as it should not be intrusive, and still make sense for the user.

Each functionality for the tool should have a help/extended setting in a popup menu to allow for rarely used settings and to provide general assistance to a user if in doubt of usage of the functionality.

General notes

Following the implementation of the product, the remaining bugs should be attended and the tool should be released on the Unity Asset Store in order to gain more information

for improvements to the tool, and the tool should implement the functionality described in this section, in a new release cycle.

Tutorial videos for the CameraTool features and general usage should be created for potential users to be able to see the toolset in action without having to download and load the tool into Unity in order ro experience the functionality of the tool. This should be seen as an introduction to the tool and the capabilities it has, as well as a promotional video for the CameraTool.

The CameraTool should be implemented into the Island Demo to allow people to test the tool in a simulated game environment, to test the functionality firsthand and see the robustness of the tool.

APPENDIX A

DADIU 2011 PRODUCTION: BRANCH

This appendix serves as documentation for the DADIU production of March 2011, which was part of the 10th semester Medialogy for the Technical specialization program. The production was in March and lasted the entire month, and was a 10 ECTS production. All of the 10 ECTS came from the semester project, which was a bit shorter than usual. As the project was part of an extended delivery of 50 ECTS, in which 10 ECTS was spent on the DADIU production, the remainder of the production was 40 ECTS points of work.

The game can be played using the Unity web-browser plugin, on `http://www.branchgame.com`.

A.1 Introduction to DADIU

The National Academy of Digital Interactive Entertainment, Det Danske Akademi for Digitalt Interaktiv Underholdning in Danish (DADIU), is a collaboration between universities and art schools in Denmark, in an attempt to educate students in game creation. [7]

The vision of DADIU is to create a number of educations which will positively influence the Danish computer game industry. Students from various educations participate in the program of creating two computer games productions, each production lasting one month, pre-planning excluded. The idea in the production is to simulate a real computer game production, as close as possible to a real production in the computer games industry.

For the productions, two teams are positioned in Aalborg and four in Copenhagen for the production month, with teams of 12-15 people, ranging from animators to programmers, on to project managers and a game director. The full list of people in a DADIU production is:

- Game Director
- Game Designer

- Project Manager
- Audio Designer
- Art Director
- Visual Designer
- Lead Animator
- Animator
- Lead Programmer
- Programmer
- Lead CG Artist
- CG Artist

In order to be able to participate in the DADIU education, one must attend a qualified education in one of the universities of art schools participating in the DADIU program. For Medialogy, in order to be qualified, one must be a master's student on the Technology specialization track, and will only be able to participate as a game programmer.

A.1.1 DADIU 2011, Team 3

Team 3, the team both authors were part of, consisted of 14 people, who all participated in the development of the game Branch [12].

The team were:

- Game director: Mai Sydendal
- Game designer: Kristian Bak
- Project manager: Morten Elkjær Larsen
- Audio designer: Troels Fløe
- Art director: Dennis Art Andersen
- Visuel Designer: Stefan Lindgren Josefsen
- Lead animator: Mikkel Mainz
- Animator: Aimall Sharifi
- Lead Programmer: Bjarne Fisker Jensen
- Programmer/audio programmer: Mads Baadsman
- Programmer/3D import/pipeline developer: Jacob Boesen Madsen
- Programmer: Peter Klogborg
- Lead CG artist: Karen Bennetzen
- CG artist: Bastian L. Strube

As there were four programmers, each with insights in different areas of the production, the tasks were initially split between the programmers, to make the production more effective and to each have sayings in different areas. Bjarne was appointed lead programmer. His responsibilities included general responsibilities for the game with respect to what could and could not be achieved in the production month and when elements should be implemented in order to follow the schedule, set by the project manager. Mads was responsible for implementing everything related to the audio in the game, or delegating the tasks to other programmers, and making certain, along with the audio designer, that all audio elements were in the final game and were implemented as they should. Jacob was responsible for handling import of assets into the Unity engine, using a custom pipeline script, as well as making sure all 3D assets and animations were cleaned up and imported problem free, as well as serving as a go-to-guy for the CG artists and animators when they had questions regarding Maya and/or Unity.

A.2 Branch, The Game

Branch is a game which takes place billions of light years away from the earth in a giant nebula, an interstellar cloud of dust and gas. This nebula has clumped together to a strong floating tree, which is so powerful that it can create stars faster than any other place in the universe. The Tree has clusters of columns, which each has a massive flow of pollen. When all of the columns in one cluster create pollen, a new star is born.

Machines have drained all the energy from our sun and many other galaxies and they are looking for a new way to get resources. They are attacking the giant nebula tree, trying to win it over so they will have a never-ending source of energy. But if they do so, the balance of the universe will be broken and it will be the end of everything that exists.

As a player you have to help the tree defending it over, so they will have a never-ending source of energy. But if they do so, the balance of the universe will be broken and it will be the end of everything that exists.

A.2.1 The idea behind Branch

Branch is created to be a game playable in a lunch-break. It consists of a simple game play which is easy to learn and hard to master. You play as the giant nebula tree which must fight of the invading machines. The plants in themselves are weak, but in great numbers they are strong, where each machine is strong by itself. As a player you must navigate the flow of the plants in order to attack the machines with a strong force, and force them back in the game, taking over their tiles one by one.

There are two ways plants can attack in the game. First, plant-occupied tiles can be lowered and the flow will stream down. Once the flow reach a low point, the plant on the tile will attack a neighboring machine-occupied tile, or in the event that all surrounding tiles are plants, the plant will shoot the pollen into the air, for later use. The tornados can be created from this pollen in the air, which is the second way to attack machines. The tornado will attract pollen from neighboring plants and multiply the effect of the pollen and damage the machine using the pollen.

The rules for the pollen are simple. The pollen will always attempt to flow downward onto neighboring plant or neutral tiles. If the pollen reaches a low and a machine is neighboring the plant, the pollen will either heal the plant or use it as a source for the plant to attack the robot. If there are no neighboring machines and the pollen has reached a low point, the plant will shoot the pollen into the air, generating force for a later tornado.

When you have defeated all robots the game is won.

A.3 Technical Aspect of the Production

This section describes the most technical aspects to take into consideration in the beginning of the production, such as how to handle scaling of the game, what features might be implemented later in the production and how the system should behave to changes. It was also considered what to delegate to different people in the production in order to avoid programmers taking on many non-programming oriented tasks, such as computer setup, asset cleanup, asset importing, redo level setup after feature changes in the game, etc.

Following the initial design, based on the prototype illustrated in Figure A.1, the design of the game was changed to make use of hexagons in the game play, and not squares as illustrated. This lead to the decision to make an underlying framework for the game, utilizing hexagons in a grid, with references to their neighbors, which can be updated dynamically as more hexagons are added to the scene. An illustration of the hexagonal grid structure can be seen in Figure A.2.

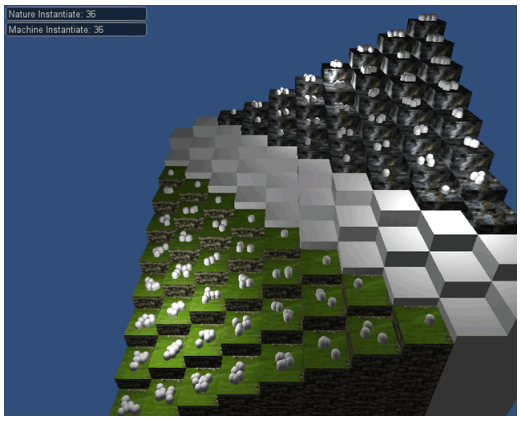


Figure A.1: The first prototype of the Branch game mechanics, developed during pre-production

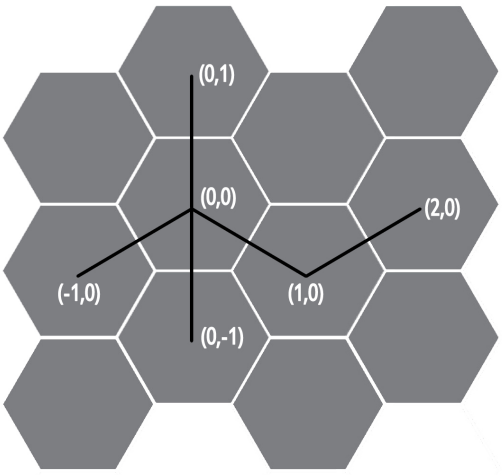


Figure A.2: The hexagonal grid structure for the final Branch game

As the level design most likely would be in the hands of the game designer, an in-game editor was created to allow the game designer to setup the level, with tiles as well as starting points for machines and plants. This was created to allow the designer to create levels without any trouble and to allow easy re-import of entire levels using new art and scripts should any development break the current levels. In order to make sure of this, the ability to save entire levels to file, as well as reload any objects and scripts on the tiles were created for the editor. An illustration of the editor as well as editor display

of a level being created are shown in Figure A.3.

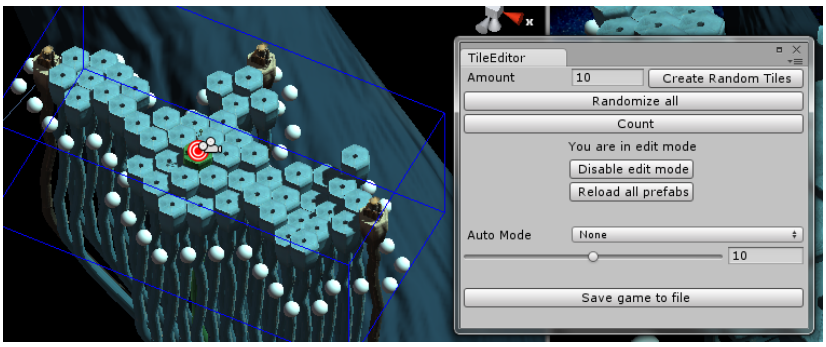


Figure A.3: The level editor for Branch, along with the development of Level 4 of the game

During the production, the editor was extended to be able to create random levels automatically, as well as randomly generate tiles automatically on top of an already designed level, should it be desired. The editor made it possible for the designer to create the world, so to speak, during procrastination. The world is illustrated in figure A.4. This game was even playable, though at low frame rates.

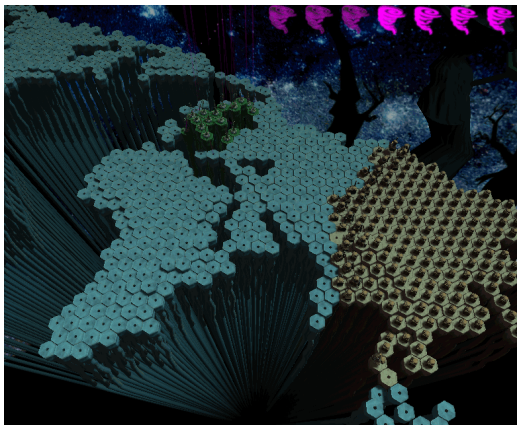


Figure A.4: The level editor made it possible for the game designer to easily create the world

Having the editor created, it was decided that game elements should be developed to support this. The game was developed to split the game into a backend consisting of the editor, the main game loop and the state machines for machines and plants, and have the entire game being playable, with interchanging graphics. For any events being dependent on the animation, this was implemented as animation events.

A.4 The Authors Contributions to the Production

As the authors of the report were half of the programmers on the DADIU production, the contributions for the game were significant. During the production, most features were

developed and changed by more than one person. Therefore this section will describe the features developed for the game by only the authors, where any element in the game in which Mads and Peter also participated in creating will be described in less detail, and any elements which Mads and/or Peter created by themselves will not be mentioned.

A.4.1 Hexagons

The hexagon system for the games was created based on inspiration from GD Reflections blog [65]. The grid created for the hexagons as illustrated in Figure A.2.

The neighbors of each hexagon can then easily be found in the grid, by looking at the relative coordinates, depending on whether the current tile has an even or odd X coordinate, as illustrated in code listing A.1

Code listing A.1: Array of neighbors relavtive to a tile

```
1 private static int[,] EVEN = { { 0, 1, 1, 0, -1, -1 }, { 1, 0, -1, -1, -1,
      0 } };
2 private static int[,] ODD  = { { 0, 1, 1, 0, -1, -1 }, { 1, 1,  0, -1,  0,
      1 } };
```

Each tile will then keep track of itself and its neighbors in order to determine game play actions.

A.4.2 Tile Editor

During the first week of the production, the game design was still a bit fuzzy which made it hard to create large tasks for the programmers since all features were unsure. The only thing that was defined was that the levels should consist of hexagon shaped tiles with the ability to interact with each other. Using this knowledge, a tile editor for the game designer to use for creating levels in the very early stages of the production was implemented. The tile editor works by locating a TileController script in the scene, and if none exists, it creates it and adds the first tile at (0,0) in the tile hexagon coordinate system. All interaction with tiles is locked and the only way of interacting with the tiles in the level is through the interface of the editor. This meant full control of the tile system to the programmers, while still allowing the artists to setup graphics in the game without the fear of breaking the system. The interface to the tile editor are illustrated in figure A.3.

When in edit mode through the tile editor, all tile spots not occupied by a tile is represented by a small sphere as illustrated in Figure A.3. By selecting the sphere, or any other tile, the type can be converted to either a neutral, nature or machine tile. This allows the game designer to expand the level by converting spheres to tiles and setup the starting point of the plant and machines. The editor also featured a method for randomly generating neutral tiles as well as randomizing the height of all tiles in the level, useful when prototyping game play features.

The editor also features a method for saving and loading a level through the editor. The file format used is a simple comma and line break separated text file stored locally on the computer. This proved to be a valuable asset and backup service as all levels suddenly became independent of the unity scene and levels could be loaded created and saved dynamically.

Later in the production the level load and the random generator functionalities were combined to create an in-game level generator. The level generator worked by creating a random amount of neutral tiles within a limit defined by the game designer. Plant tiles

were then spawned in the middle of the map, where machine tiles are crated at the edge of the map for best game play value. Screenshots of a random generated level can be seen in Figure A.5.

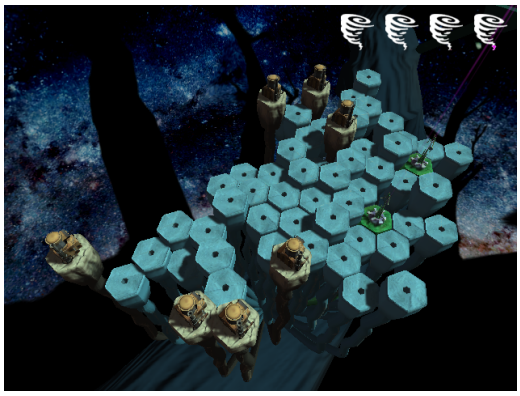


Figure A.5: A random level generated using the editor developed

A.4.3 Tile Behaviour

To implement the basic functionality needed for the tiles, each tile in the game extends a TileBehaviour abstract class which then again extends the Attackable abstract class, visualized in Figure A.6. The Attackable class extends MonoBehaviour and adds a basic interface to the tiles that allows them to be attacked by neighbors as well as keeping track of health, damage taken, healing and whenever or not the tile is dead and has to start its takeover phase. The class structure for the Attackable class can be seen in Figure A.6. The TileBehaviour class adds basic kwnolegde of the tiles surroundings and creates abstract functions for sending and reciving flow, and it is up to the classes that extends TileBehaviour to implement what should happen when flow is received.

MachineBehaviour

Each machine tile in the game has a MachineBehaviour script attacked and has a reference to all neighbors. By choice the machine is implemented using a very simple AI, since the machines are suppose to be dumb, but strong contrary to the plants that are smart, but weak. The machine always moves up and down towards the nearest non-friendly neighboring tile. If none exists it goes into the hibernate state described in subsection A.4.7. This means that the machine will always try to level with the tile that it is attacking. When in the same level, the machine enters an attack routine until the target is dead or the machine is no longer in level with the target. The different states can be seen in code listing A.2.

Code listing A.2: States in the MachineBehavior class

```
1 private enum States
2 {
3     Idle,
4     Move,
5     Attack,
6     Takeover,
7     Rotate,
```

```
8       Spawn,
9       Die,
10      Hybernate
11  }
```

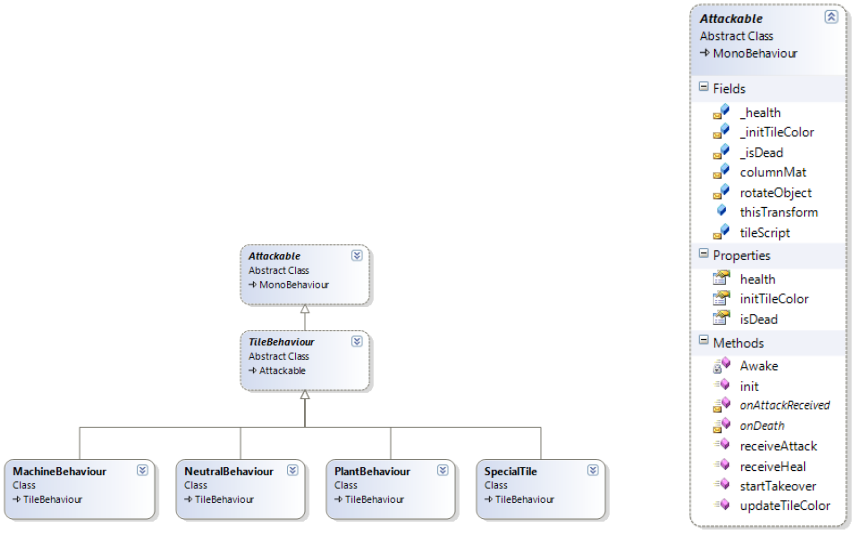
PlantBehaviour

The PlantBehaviour script is attached to all plant tiles, and like all other tiles it has a reference to each of its neighbors. Each plant tile generates pollen, and if it has no enemy neighbors it distribute the pollen evenly to all lower positioned neighbors. If the tile is a local minimum and there are not lower positioned neighbors it shoots the excess pollen in the air to charge the players tornado power.

When the pollen reaches a tile positioned at the frontline with one or more enemy neighbors, it is used by that plant tile to heal and raise attack power. A plant with no healing will always loose to a machine, which means that the player must always reposition the tile height to ensure that all frontline tiles receives a minimum of healing to survive.

A.4.4 Data Gathering

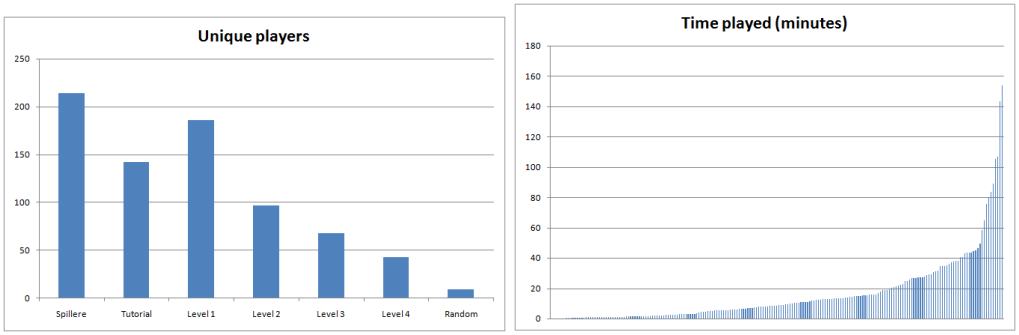
Using the same data gathering system implemented in the CameraTool it was quick to modify the code to track user data during play testing of prototypes and final version of the game. This allowed metrics to be generated to visualize how the users played the game, for how long and how the level progressed. On top of the data gathering, a system allowing the users to give feedback on the game through an in game questionnaire was also implemented. This meant that each dataset could be mapped to the user feedback, allowing the game designer to create levels and game play features based on the results.



(a) Behaviors extending from the Attackable class

(b) Overview of the functionality of Attackable

Figure A.6: Overview of attackable and classes extending it



(a) Overview of the unique number of players at the time of the DADIU presentation, end of March 2011

(b) Overview of time spent playing the game, for each user, at the time of the DADIU presentation, end of March 2011

Figure A.7: Data gathered from the DADIU production 2011

At the end of the production, the data gathered from the latest build was collected and visualized to show general metrics at the evaluation after the production. The data shows more than 200 unique players that loaded the prototype and showed a steady decrease of players completing different levels until the hardest, the random generated level, which less than 5% were able to play. The visualized data of unique players can be seen in Figure A.7 (a).

The time used by each player was plotted into a histogram and sorted on total time played which showed an average playtime of around 20 minutes, which was considered a success since the build only featured a vertical slice of the full game idea. The histogram of the total time played can be seen in Figure A.7 (b).

A.4.5 Pipeline Development

Based on previous experience, it was decided early on in the development, that the artists should not be responsible for adding assets to the asset server, by themselves. The desired workflow for the artists were to create the assets locally on their machines, and then backup using a network server. This was not the most desirable solution, as it could possibly mean lost asset and time, should a computer crash. But for a single month production, setting up SVN and teaching people to use it was considered unnecessary. Instead, people were told to have all work placed on a network drive, in order to have a shared location for assets, as well as a drive to take backup of. This however, meant there were no versioning of assets. A python script was created for the process of taking hourly backup of the server to a local computer, as well as a removable hard drive for offsite backup during the night and weekends.

This was created to use minimal time for setting up the workflow for all people involved in the development, and for limit the amount of new programs and workflow methods.

Another python script was written to automatically import assets from the network server into Unity. The script looked at the name and date of the file, and compared the file to items in a text file, and re-imported new assets. It was a crude way, but made sure the artist did not have to import and upload assets in Unity, and could focus on creating assets, trusting the script took care of the process. This also limited the need for communication when new assets were ready for the game.

An illustration of the entire development pipeline, simplified, as documented by the project manager, are illustrated in Figure A.8

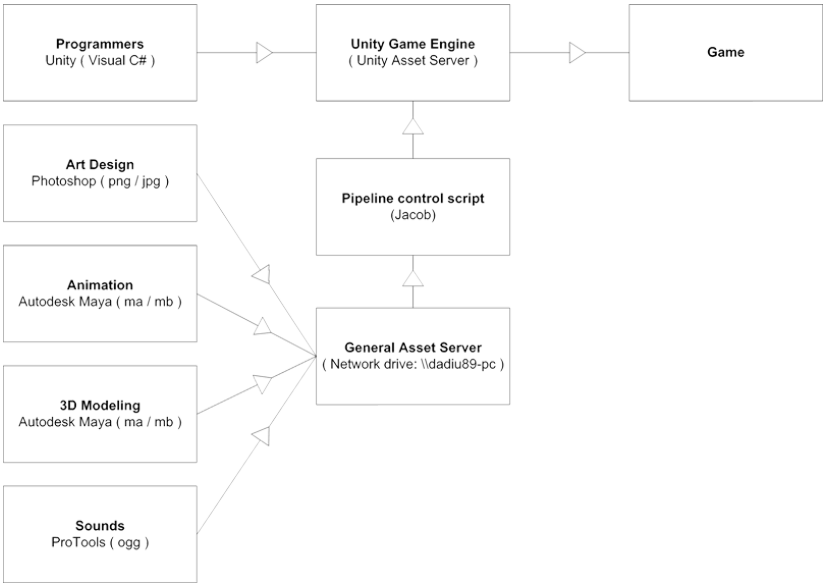


Figure A.8: Simplified development pipeline, as documented by the Branch project manager

A.4.6 Pause Functionality

In order to make sure all game activity stopped as the game pauses, a global message was sent to all game objects, which should implement functionality to receive the call if necessary. An example is the function unPauseGame(), as seen in code listing A.3.

Code listing A.3: unPause function from the game

```
1  public static void unPauseGame()
2  {
3      Time.timeScale = timeScale;
4      Time.fixedDeltaTime = fixedDeltaTime;
5      _isGamePaused = false;
6
7      object[] objects = FindObjectsOfType(typeof(GameObject));
8      foreach (GameObject go in objects)
9      {
10         go.SendMessage("OnResumeGame", SendMessageOptions.
               DontRequireReceiver);
11     }
12 }
```

A similar function is called for pausing the game. This triggers on escape, when showing the main menu and during the tutorial of the game. As this does not happen often, a global message sent is considered not expensive, and any short delay occurring when pausing can be attributed the pausing of the game.

A.4.7 Model Swap Optimization

The computer graphic artists on the team is coming from the animation school in Viborg, where the education is mostly focused on creating 3d animation films. When creating 3d models for animations, not much focus is put on optimizing the 3d models by using a low polygon count and simple rigs, useful when creating computer games. Although the problem was anticipated by the programmers, and a polygon limit of 1200-1500 was set as an absolute maximum for the plant and machine models which turned out to be a challenge for the cg artists. The result was that the two models for the plant and machine ended up having an over-complicated rig compared to the object size and level of details as well as having multiple meshes within each model. Although the models had less than the polygon limit specified, the machine model, which was the worst, ended up using roughly 300 draw calls per model even after mesh sharing techniques were applied. This meant that the game would run at a very low frame rate, due to the high amount of draw calls, if the machine were to have multiple models on the screen, which was unacceptable. At the time the problem was discovered, the models had already been animated and implemented in the game, and a remake of the whole process would take a long time.

The solution became to animate a passive state for the robots, where they would transform to a cube and power down, when not on the front line. The passive model for the machine would then be swapped with a textured cube with significantly fewer draw calls. When the machine detects that no neighbor plants tiles are present it starts to play a power down animation which, on finish, swaps the model to the static, low polygon version. The opposite happens if the machine becomes part of the frontline and the machine needs to start its attack routine. The system was stress tested on an extreme map designed to look like the world which showed that the game was still playable with large amounts of enemies. The illustration in Figure A.4 shows a screenshot from the stress test. The machines started in Japan and started to spread through Asia, and all machines not near the frontline entered the powered down state like expected.

A.4.8 Event Driven Animations

To toggle animation based events, classes for handling this was implemented, to avoid manually setting up animation events in the editor. An example of a setup function can be seen in code listing A.4.

Code listing A.4: Animation event setup code example

```
1 private void setupAnimationStart(string targetAnimationName, string
      functionName)
2 {
3     AnimationClip targetAnimation = anim.GetClip(targetAnimationName);
4     AnimationEvent doneEvent = new AnimationEvent();
5     doneEvent.time = 0;
6     doneEvent.functionName = functionName;
7     targetAnimation.AddEvent(doneEvent);
8 }
```

This also made it animation independent, so animations could be edited or changed during the production, without any trouble.

A.4.9 Miscellaneous

As mentioned, as the authors is half the programming staff at the production, the list of programming done, and elements created significant. Following is a list of elements created or co-created by the authors during the production, which are deemed unnecessary to explain in detail in the report. As a reference, the code mentioned here can be found in the Appendix, B.

- InputController (The initial implementation)
- FlowObject
- GameSettings
- MachineAnimationSetup and PlantAnimationSetup
- MeshCombineUtility
- ScalePulse
- TornadoScript (The initial implementation)
- WorldParticleSystem

A.5 The Production and CameraTool

During the production, the job of creating the camera was in the hand of the game designer. In the initial phase of the production he was given the CameraTool and told to setup the camera for the game using the tool.

This was done both to delegate work from the programmers unto the designer, to have more time for focusing on other aspects of the game development, and also, mainly, to have the CameraTool tested in the production, to see whether it was actually useful in a real production.

Initially the game designer was told he would not get any assistance in the task of setting up the camera. Only if the camera system broke or features were insufficient should he contact the programmers and have them assist him in the needs he had for the camera. During the setup of the camera and initial stages of the production, data was collected of usage of the tool during these parts of the production. The data collection was turned off from this CameraTool after the final setup of the camera, in order to not flood the database with un-useful information, as well as to improve runtime behavior of the game. Although it was not much of a performance increase.

APPENDIX B

DVD CONTENTS

The content on the DVD are organized as follows:

- CameraTool - The code contained in a unitypackage
- Data Gathering - The data gathered at the DADIU production, of usage of the CameraTool
- Interviews - The notes from the interviews conducted following the DADIU production
- Project Report - An electronic copy of the project report
- Project Video - The A/V production for the project
- Usability Tests - The notes from the second usability test round

BIBLIOGRAPHY

[1] Carl-Mikael Lagnecrantz. Polyboost. http://polyboost.com/, May 2011.

[2] Steve Krug. *Don't Make Me Think*. New Riders Press, 2006.

[3] Jacob B. Madsen Dennis K. Risborg and Jonas W. Christensen. Easyrig - a rigging script for autodesk maya. http://projekter.aau.dk/projekter/da/studentthesis/easyrig%2887747109-4f85-4dbe-bdb5-ef5a2ac2854a%29.html, May 2009.

[4] Autodesk. Autodesk maya. http://usa.autodesk.com/maya/, May 2011.

[5] Mark Haigh-Hutchinson. *Real-Time Cameras: A Guide for Game Designers and Developers*. Morgan Kaufmann Publishers, Burlington, MA 01803, USA, 2009.

[6] Valve. Counter-strike: Source on steam. http://store.steampowered.com/css, May 2011.

[7] DADIU. Det danske akademi for digital interaktiv underholdning. http://dadiu.dk/, December 2010.

[8] Scott Goffman. Making tools your team will actually use. http://tech-artists.org/downloads/GDC2011/GDC2011_ScottGoffman_MakingGoodTools.pdf, March 2011.

[9] Unity Technologies. Unity. http://unity3d.com/, December 2010.

[10] Unity. Unity: Publishing. http://unity3d.com/unity/publishing/, May 2011.

[11] Mono Project. What is mono. http://mono-project.com/What_is_Mono, May 2011.

[12] DADIU 2011 Team 3. Branch. http://branchgame.com/, May 2011.

[13] Marc Christie, Rumesh Machap, Jean-Marie Normand, Patrick Olivier, and Jonathan Pickering. Virtual camera planning: A survey. In Andreas Butz, Brian D. Fisher, Antonio Kruger, and Patrick Olivier, editors, *Smart Graphics*, volume 3638 of *Lecture Notes in Computer Science*, pages 40–52. Springer, 2005.

[14] Tsai-Yen Li and Chung-Chiang Cheng. Real-time camera planning for navigation in virtual environments. In *Proceedings of the 9th international symposium on Smart Graphics*, SG '08, pages 118–129, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] Nicolas Halper, Ralf Helbing, and Thomas Strothotte. A camera engine for computer games: Managing the trade-off between constraint satisfaction and frame coherence, 2001.

[16] Steven M. Drucker and David Zeltzer. Intelligent camera control in a virtual environment. In *Graphics Interface '94*, pages 190–199, may 1994.

[17] William Bares and Byungwoo Kim. Generating virtual camera compositions. In *Proceedings of the 6th international conference on Intelligent user interfaces*, IUI '01, pages 9–12, New York, NY, USA, 2001. ACM.

[18] Ting chieh Lin, Zen chung Shih, and Yu ting Tsai. Cinematic camera control in 3d computer games.

[19] Brian Hawkins. Creating an event-driven cinematic camera, part one. `http://www.gamasutra.com/view/feature/2910/creating_an_eventdriven_cinematic_.php`, January 2003.

[20] Brian Hawkins. Creating an event-driven cinematic camera, part two. `http://www.gamasutra.com/view/feature/2909/creating_an_eventdriven_cinematic_.php`, January 2003.

[21] Dennis Nieuwenhuisen and Mark H. Overmars. Motion planning for camera movements in virtual environments. Technical report, `http://archive.cs.uu.nl/pub/RUU/CS/techreps/CS-200`; `http://www.cs.uu.nl/research/techreps/repo/CS-2003`, 2003.

[22] Wikipedia. Video game genres. `http://en.wikipedia.org/wiki/Video_game_genres`, December 2010.

[23] Wikipedia. half-life 2. `http://en.wikipedia.org/wiki/Half-Life_2`, May 2011.

[24] Valve. Team fortress. `http://www.teamfortress.com/`, May 2011.

[25] Inc. Activision Publishing. Call of duty. `http://www.callofduty.com/`, May 2011.

[26] Rockstar Games. Grand theft auto. `http://www.rockstargames.com/grandtheftauto/`, May 2011.

[27] Atlus. Demon souls. `http://www.demons-souls.com/`, May 2011.

[28] Blizzard. Diablo 3 gameplay video part 1. `http://www.youtube.com/watch?v=NQMBIRipp5A`, May 2011.

[29] Bioware. Bioware. `http://www.bioware.com/`, May 2011.

[30] Blizzard. Blizzard. `http://eu.battle.net/sc2/`, May 2011.

[31] Stainless Steel Studios. Empire earth. `http://games.ign.com/objects/027/027182.html`, May 2011.

[32] DADIU 2009. Slug n' roll. http://dadiu.dk/spil/spil-2009/slug-n-roll, May 2011.

[33] DADIU 2009. Nevermore. http://dadiu.dk/spil/spil-2009/nevermore, May 2011.

[34] DADIU 2009. Puzzle bloom. http://dadiu.dk/spil/spil-2009/puzzle-bloom, May 2011.

[35] DADIU 2010. Office rage. http://dadiu.dk/spil/midtvejsspil-2010/anton-og-katastrofen, May 2011.

[36] DADIU 2010. Anton and the catastrophe. http://branchgame.com/, May 2011.

[37] DADIU 2010. Sophie's dreamleap. http://dadiu.dk/spil/midtvejsspil-2010/sophies-dreamleap, May 2011.

[38] Wikipedia. Hot lava (game). http://en.wikipedia.org/wiki/Hot_Lava_(game), May 2011.

[39] Unity. Unity: Image effects scripts. http://unity3d.com/support/documentation/Components/comp-ImageEffects.html, May 2011.

[40] Scott Rogers. *Level Up! The Guide To Great Video Game Design.* Wiley, 2010.

[41] Donald A. Norman. *Design of Everyday Things.* Basic Books, 2002.

[42] Apple. http://www.apple.com/, May 2011.

[43] UGO Entertainment. Tomb raider series. http://tombraider.ugo.com/games/, May 2011.

[44] Jeff Johnson. *GUI Bloopers.* Morgan Kaufmann Publishers, 2000.

[45] Autodesk. Autodesk 3ds max. http://usa.autodesk.com/3ds-max/, May 2011.

[46] Jenny Preece Helen Sharp, Yvonne Rogers. *Interaction Design: Beyond Human Computer Interaction.* Wiley, 2007.

[47] Cameron Chapman. 10 usability tips based on research studies. http://sixrevisions.com/usabilityaccessibility/10-usability-tips-based-on-research-studies/, May 2011.

[48] Margaret W. Matlin and Hugh J. Foley. *Sensation And Perception.* Allyn And Bacon, 1997.

[49] Wikipedia. Gestalt psychology. http://en.wikipedia.org/wiki/Gestalt_psychology, May 2011.

[50] infovis wiki. Gestalt laws. http://www.infovis-wiki.net/index.php?title=Gestalt_Laws, May 2011.

[51] Unity. Extending the editor. http://unity3d.com/support/documentation/Components/gui-ExtendingEditor.html, May 2011.

[52] Foraker Labs. 3-click rule. http://www.usabilityfirst.com/glossary/3-click-rule/, May 2011.

[53] Wikipedia. Three-click rule. http://en.wikipedia.org/wiki/Three-click_rule, May 2011.

[54] Unity. Execution order. http://unity3d.com/support/documentation/Manual/Execution%20Order.html, May 2011.

[55] Ralph Johnson Erich Gamma, Richard Helm and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[56] Unity. Character controller. http://unity3d.com/support/documentation/Components/class-CharacterController.html, May 2011.

[57] Unity. Render settings. http://unity3d.com/support/documentation/Components/class-RenderSettings.html, May 2011.

[58] Wikipedia. Smoothstep. http://en.wikipedia.org/wiki/Smoothstep, May 2011.

[59] Techsmith. Camtasia studio. http://www.techsmith.com/camtasia/, May 2011.

[60] Dot Net Perls. C# int.max and min constants. http://www.dotnetperls.com/int-max-min, May 2011.

[61] DADIU 2011. Unity technologies. http://unity3d.com/unity/editor/asset-store.html, May 2011.

[62] DADIU 2011. Blendimals. http://blendimals.dadiugames.dk/, May 2011.

[63] DADIU 2011. Raasool. http://raasool.dadiugames.dk/, May 2011.

[64] DADIU 2011. Broken dimensions. http://broken-dimensions.dadiugames.dk/, May 2011.

[65] Ruslan Shestopalyuk. Hexagonal grid math. http://gdreflections.com/2011/02/hexagonal-grid-math.html, May 2011.