OULUN AMMATTIKORKEAKOULU

Joonas Anttila

CREATING AN ADAPTIVE CAMERA SYSTEM FOR A 3D PLATFORMER GAME

Meeting the requirements of game design and end-user experience

CREATING AN ADAPTIVE CAMERA SYSTEM FOR A 3D PLATFORMER GAME

Meeting the requirements of game design and end-user experience

Joonas Anttila
Bachelor's thesis
Spring 2015
Information Technology
Oulu University of Applied Sciences

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan koulutusohjelma, ohjelmistokehityksen suuntautumisvaihtoehto

Tekijä(t): Joonas Anttila
Opinnäytetyön nimi: Mukautuvan kamerajärjestelmän luonti 3D-tasoloikkapeliä varten
Työn ohjaaja(t): Pekka Alaluukas
Työn valmistumislukukausi ja -vuosi: kevät 2015 Sivumäärä: 40

Tämän opinnäytetyön tavoitteena oli suunnitella ja kehittää mahdollisimman hyvän pelikokemuksen tuottava adaptiivinen kamerajärjestelmä tietokonepeliä varten. Työllä ei ollut varsinaista toimeksiantajaa, mutta varsinainen kamerakomponentti kehitettiin osana pelidemoa, ja sitä myöten tiimin tarpeita varten. Kyseessä oleva peliprojekti on saanut alkunsa Oulu Game Lab -kehitysohjelmasta.

Tarkoituksena oli kehittää toimiva ja sulava kamerajärjestelmä 3D-tasohyppelypeliin. Kamera pyrkii tarjoamaan pelaajalle parhaan mahdollisen kolmannen persoonan perspektiivin ja pelattavuuden kaikissa tilanteissa sekä mukautuvuuden peliympäristön tuomiin eri haasteisiin. Työssä käydään läpi erilaisia pelikameramalleja ja niiden ominaisuuksia, kameran suunnittelussa huomioonotettavia asioita sekä tekniikoita parhaan pelikokemuksen saavuttamiseksi.

Samalla esitellään kamerakomponentin ja sen toimintaan pohjautuvan tähtäyskomponentin rakentaminen, ja tavoitteen saavuttaminen ohjelmallisesti. Tähtäyskomponentin avulla pelaaja pystyy ampumaan vihollisia sekä aktivoimaan eri elementtejä pelissä.

Lopputuloksena syntyi toimiva kolmannen persoonan kamerakomponentti sekä kameran toimintaan pohjautuva pelillinen komponentti, joita käytettiin pelidemossa. Kamera vastaa suunnitteluvaatimuksia ja tavoite saavutettiin onnistuneesti. Kamerajärjestelmää voidaan myös käyttää muihin kolmannen persoonan perspektiivin peleihin, ja se on helposti laajennettavissa ja jatkokehitettävissä.

Työ kehitettiin käyttäen Unity-pelimoottoria ja Microsoft Visual Studiota. Ohjelmointikielenä työssä oli C#.

Asiasanat: peli, tietokonepeli, Unity, pelinkehitys

ABSTRACT

Oulu University of Applied Sciences
Bachelor’s Degree of Information Technology, Software Development

---

Author(s): Joonas Anttila
Title of thesis: Creating an Adaptive Camera System for a 3D-Platformer Game
Supervisor(s): Pekka Alaluukas
Term and year of completion: Spring 2015 Pages: 40

The aim of this Bachelor’s thesis was to design and create an adaptive camera component for a 3D platformer computer game. The camera would need to provide a satisfying end-user experience. The camera component was developed as a part of a game demo which stemmed from the Oulu Game Lab education program.

Game cameras are an essential part of the gameplay experience. A bad camera can be a terrible nuisance for the player. The aim was to design and program a solid camera component that would follow the action with ease and adapt to the game environment, while always providing the best possible view for the player.

The first half of the thesis covers different camera models and their features, techniques and design notions that need to be considered in order to achieve the best possible gaming experience. The second part covers the actual programming part of the camera component. The final part of the thesis explains the development and programming of a targeting component based on the functionality of the camera. The purpose of the targeting system is to allow the player to shoot and handle groups of enemies as well as activate elements in the game world.

The result was a successful third person camera component, plus a separate gameplay element based on the mechanics of the camera. The camera fulfills the design requirements and works as intended. The camera component can be used for other similar third person perspective games too, and it is easily expandable. No real issues or bugs were present during the development, but the collision detection and clipping issues could be improved in the future.

The project was developed using the Unity3D game engine and Microsoft Visual Studio. Programming was done in C#.

Keywords: game development, game, Unity

CONTENTS

GLOSSARY

Collider	Defines the physical bounds of a game object
Gameplay	The structures of player interaction with the game system
Metroidvania	A genre of 2D platformer games which emphasize on action-adventure and exploratory gameplay
Raycast	A function that casts a ray that detects colliders
SSAO	Screen space rendering occlusion, a rendering technique for approximating the ambient occlusion effect in real time
Transform	A component that determines the position, rotation, and scale of every object in a scene

1 INTRODUCTION

This Bachelor's thesis is about creating an adaptive camera component for a 3D platformer computer game called Bonnie the Brave: Space Courier, and how to make it fulfill the needs of the game's design while providing the player the best possible playing experience.

The game project in question stems from the Oulu Game Lab training program. The purpose of the program is to produce new game companies and create jobs in the industry. While the subject of the thesis was not commissioned by anyone officially, the need for the component was commissioned by the game itself and the team around it.

Game cameras are an essential part of the gameplay experience, especially in the 3D platformer genre. A badly constructed camera component can be a massive nuisance and ruin the fun for the player. The camera needs to be able to follow the action with ease while providing the player the best possible view of the game world. A well-built camera system can also be used as a base for other features. The first half of the thesis will cover general camera models found in video games, and what design aspects need to be considered. The second half covers the actual development and programming of the camera component, as well as the programming of a gameplay element based on the functionality of the camera.

The game was developed using the Unity engine and Microsoft Visual Studio 13.

2 ABOUT THE GAME PROJECT

This chapter will cover an overview of the game project and the tools of development. At the time of writing, the game is still unfinished so the content regarding the design is subject to change.

2.1 The project: Bonnie the Brave – Space Courier

Bonnie the Brave – Space Courier is a 3D adventure platformer game designed for the PC and consoles. The following excerpt describing the overall setting of the game is from one of the four design documents: "*The game is built around a compelling and colourful story about an intergalactic postal courier, Bonnie the Brave. The story starts with an altercation with robot space pirates, robuccaneers, and sends our heroine on a journey of discovery through a mysteriously lush crater on a planet otherwise devoid of all life. Bonnie has to use all her wits and abilities, as well as an impressive array of retro-futuristic gadgetry, to survive the crater's challenges. Once she finally manages to collect the pieces of her ship and prepares to leave, something old and primal is stirring beneath the planet's surface and escaping the planet becomes the least of her worries.*" (1, p. 2.)

At the core of the game is a tight and nostalgic platformer. Traditional 3D-platforming is combined with a metroidvania-type open-world structure that supports discovery and exploration. Bonnie collects and finds gadgets and enhancements in the world that help her access new areas in the world and fight against hostiles with different techniques. The game aims for 4-6 hours of playtime. (1, p. 2)

The visual representation of the game is colorful and cartoony. The graphical style is nostalgic but modernized, referencing game franchises such as *Jak and Daxter* and *Ratchet & Clank.* The visuals are supported with an up-beat musical score and a wide array of lush sound effects. (1, p. 2; 2; 3.)

2.2 Development tools

Ultimately, developing a game requires the use of several different programs from possible game engines to modeling and graphics editor tools. This game was developed using the Unity3D development platform plus a separate script editor IDE (integrated development environment).

2.2.1 Unity3D

Unity3D, also known as just Unity, is a very popular game development environment. It includes a physics game engine/renderer and an integrated editor. The Unity engine contains a massive array of features, categorically from animation, scripting and graphics to 3D physics, audio and optimization to name a few. Unity is liked throughout the industry for its speed and efficiency of its workflow. The editor is user-customizable which means it can be modified with free, paid or self-made extensions - making it a flexible and powerful tool. (4.)

One of Unity's leading features is its multiplatform support. This means that a game can be made and built for multiple platforms. The number of supported platforms is growing constantly. Currently Unity supports Windows, Linux, Mac, mobile platforms (iOS, Android, Windows Phone, BlackBerry), web browsers and consoles (PS3, PS4, PS Vita, XBOX ONE, XBOX 360, Wii). (5.)

2.2.2 Microsoft Visual Studio

While Unity comes with a script editor called MonoDevelop, external editors can be used. Unity supports three different programming languages: C#, JavaScript and Boo (5). The scripting language for *Bonnie the Brave* was C#.

MonoDevelop is capable of handling all three programming languages perfectly fine, but out of personal preference and some minor technical features, the editor used for this project was Microsoft Visual Studio 2013. Visual Studio is a comprehensive set of tools and services for developing desktop, device and web applications as well as cloud services. It features a development environment for .NET languages as well as other languages such as HTML, C++ and VB). Visual Studio comes in several versions, each with a different collection of features. (6; 7) The version used for this project was Visual Studio Professional.

In the past, my personal experience with MonoDevelop had not been too great. Some features such as the IntelliSense/auto-code completion performed inconsistently, the performance was poor and crashes were too frequent. I had been using Visual Studio for a few years, and found it much more to my liking. It performs great, has many useful features and is highly editable.

3 CAMERA MODELS

Fundamentally, graphics in a game are meant to display the game world space to the player. That space is displayed from a particular point of view or angle. Designers usually imagine this as a hypothetical camera. The camera creates an image which the player can see. A camera model is the system controlling the behavior and performance of the camera. (8, p. 39.)

Camera models can be dynamic or static. Older games and many small games nowadays use a static camera model. (8, p. 39.) The virtual space is shown from a fixed perspective, just as if the game screen was treated like a game board in tabletop games (8, p. 271). In dynamic models the cameras move in reaction to player actions and events happening in the game world. These models are harder to implement and necessitate more design effort. In turn they provide the player a more cinematic and lively experience. (8, p. 39.) The cinematic analogy is often used when talking about the view on the screen. The term virtual camera stems from this idea that the view on the screen is an output of a camera looking at the game world. (8, p. 271.)

Since *Bonnie the Brave* is a 3rd person platformer game, the focus was on third-person models. Other models did not really fit the scope of the thesis, but they will be discussed in passing.

3.1 Third-person perspective

A 3rd person perspective camera is the most common model in action-adventure and 3D action games. Typically, the camera tails an avatar at a fixed distance. It remains behind and somewhat above the character/avatar as she moves in the game world. This allows the player to see the world beyond the avatar. The advantage of a third-person perspective is that it lets the player to see the avatar in great detail in games with strongly characterized or customizable avatars. (8, p. 273.)

Dynamic third-person camera models also take a more significant amount of work to implement. One of the biggest issues is to decide and implement the camera behavior when the avatar is moving, and more specifically turning. There are a few different options to carry out depending on the game and general preference.

Cameras that keep their position constantly behind the avatar are used in simulator games such as flight simulators or racing games. The camera always looks in the same direction the avatar is looking. This sort of behavior is useful in chasing sequences or high-speed scenarios. Generally games with vehicles as playable avatars use this sort of camera behavior. It does not work well with humanoid avatars because of the possible rapid direction changes. Fast sweeping motions of the camera can cause players motions sickness and disorientation. (8, p. 274.) As an example, imagine a person running around with a 3-meter-long stick with a camera at the end attached to their back.

Another behavior model example can be taken from *Super Mario 64* (9), which is a similar game to *Bonnie the Brave*. The camera will generally follow the avatar, but it allows the player to see the sides of the character when turning (IMAGE 1). It slowly reorients itself and positions back to behind the avatar after the avatar has begun turning. This approach creates a less disorienting experience. (8, p. 274.)

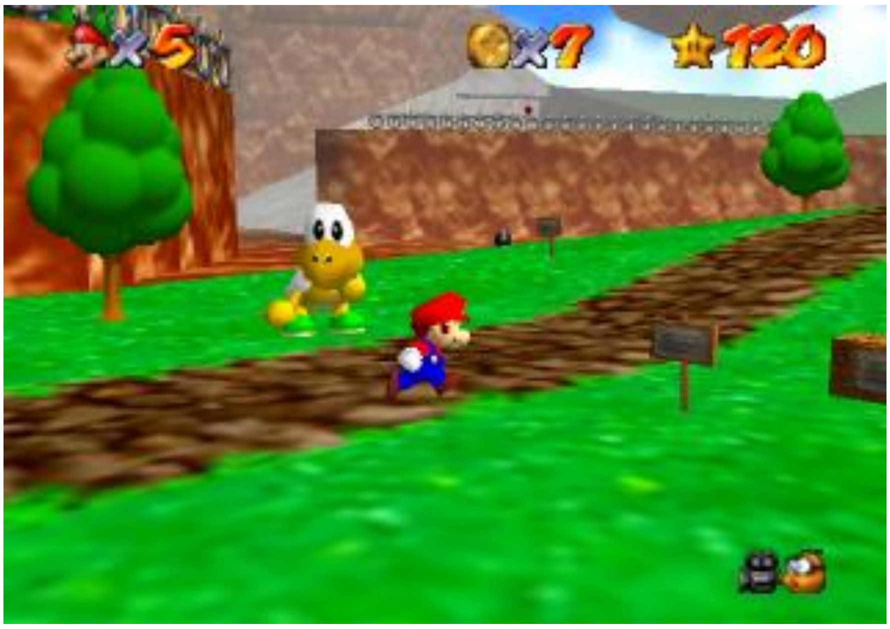


IMAGE 1. The camera showing Mario's side mid-turn in Super Mario 64 (10)

Some games also use a camera model that reorients only after the avatar has stopped moving. While this is the least aggressive way to reorient the camera, the player can command the avatar character to turn back the way she came and run straight at the camera. In this situation the camera will dolly away backwards to keep the avatar in view. It also means that the player is not able to see obstacles in the avatar's way because they are behind the camera. (8, p. 274.) To achieve the best outcome, games can blend the elements of all the behavior models mentioned. Reorientation can be automatic but the player can be given options to reorient manually or switching between different camera modes.

Another common issue with third-person models is obstructing landscape objects. These objects can intrude between the character and the camera and effectively block the player's view. These situations can and should be handled properly to prevent frustrating situations. Obstruction issues can be tackled with several solutions, for example orienting the camera differently to provide a better view or rendering objects semitransparent. (8, p. 274.)

In third-person games players are usually given the option to position the camera manually to get a better look at the surrounding world. The player can use assigned controls to circle the camera around the avatar, focusing on her in the middle of the screen. This way the player can see the game world around the character in any situation. It also enables the players to look at the avatar from different angles. (8, p. 273.) The manual control can assist the player and prevent frustration if the camera does not act optimally. Additionally, it administers the player more freedom and allows a more effective gameplay experience.

3.2 First-person perspective

In first-person perspective models the camera assumes the position of the avatar's eyes. Generally, the player does not see the character's body, except for the avatar's hands and displayed handheld weapons. First-person perspectives provide immersion as they make the players feel like they are in the game world themselves (IMAGE 2). Players do not need to control the camera either, looking around happens by simply moving the avatar. (8, p. 272.)

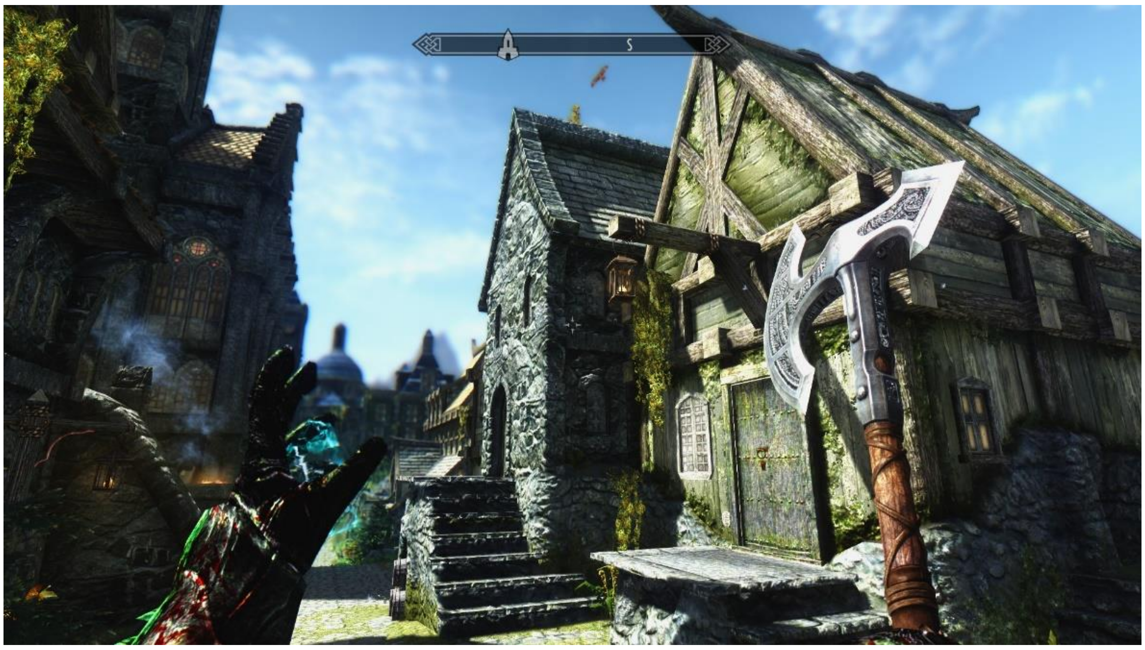


IMAGE 2. The first-person perspective with an immersive depth-of-field effect in "The Elder Scrolls V: Skyrim" (11)

First-person cameras have a lot of advantages. The avatar character does not need to be displayed constantly, which boosts performance as there is less graphical data to render. It also reduces the need for a large number of animations. With first-person cameras players feel that it is easier and more practical to aim and use ranged weapons. There are two main reasons for this. Firstly, the player's viewpoint matches exactly with the avatar's viewpoint, which means that the difference between the player's perspective and the avatar's perspective need not be corrected. Secondly, the avatar's body does not block the view for the player in any way so the player can see clearly. Interacting with the environment and game world is straightforward. The first-person view makes it easy for the players to maneuver and position the avatar accurately when doing tasks like picking up objects, going through narrow doorways or delivering attacks with weapons (IMAGE 3). Positioning the character during combat is easier and more precise in the first-person mode. It also adds a greater sense of immersion. (8, p. 272.)

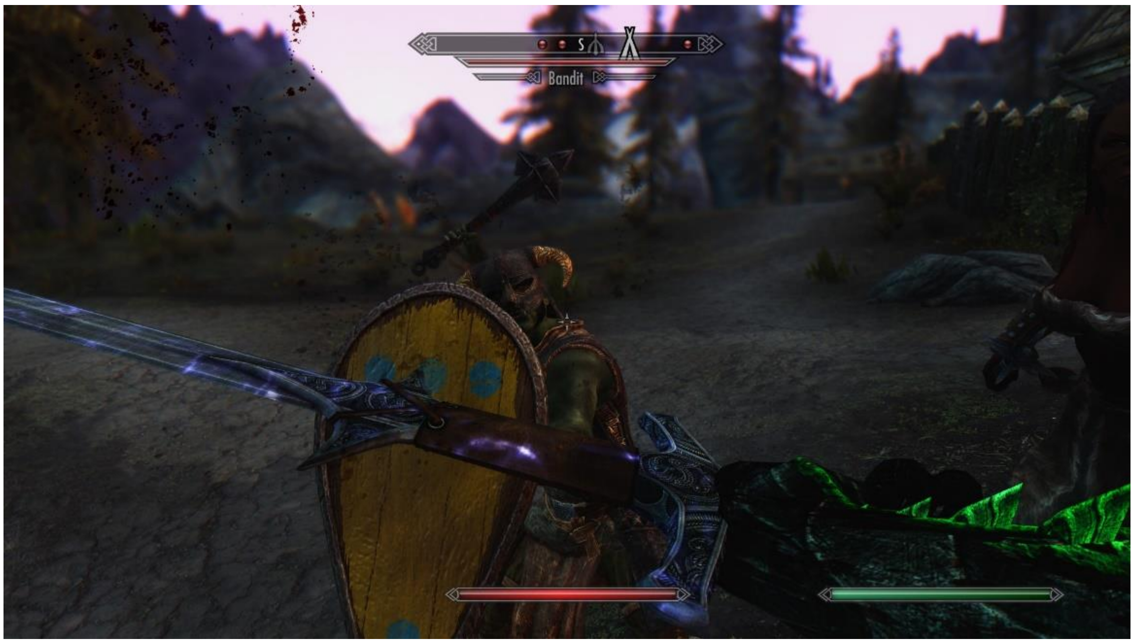


IMAGE 3. The player getting ready to block a melee attack in “The Elder Scrolls V: Skyrim”. (11)

The model also has its disadvantages. The player’s sense of the character’s personality and moods might be diminished since her body language and facial expressions cannot be seen. The character’s personality must be expressed through scripting and voice-acting. Certain types of gymnastic and athletic moves are likewise difficult to pull off in first-person perspectives. Platforming and jumping is easier when the player can see the avatar: Timing perfect jumps can be an issue with the first-person models. In addition, quick movements, rising and falling sensations can induce motion sickness in players. (8, p. 272.)

3.3 Aerial perspectives

Aerial perspective camera models are usually needed for games where the player sees a lot at once, like strategy games or traditional party-based role-playing games. Aerial perspectives prioritize the game world rather than one character. The player usually sees the game world and a lot of objects and characters or units in one view. In multi-present interaction models, where the player can act on different sections of the game world at will, the player needs to be able to move the game view around to see the part they want. The same applies to party-based interaction models, although in these types of games the movement of the camera is usually restricted to the surrounding region of the party. Aerial perspectives can be further divided into top-down and isometric perspectives (IMAGE 4) as well as free-roaming cameras. (8, p. 39, 275-276.)

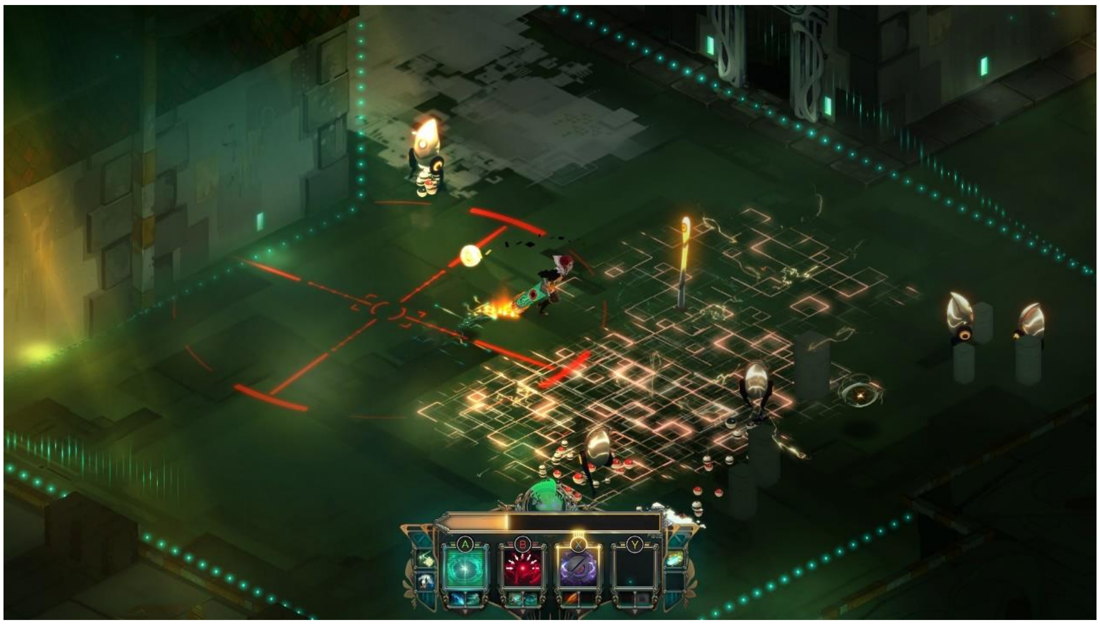


IMAGE 4. The isometric perspective in "Transistor" displaying the player and multiple enemies within the confined encounter area. (12)

4 DESIGN REQUIREMENTS

This chapter details information about what the camera system in *Bonnie the Brave* should cover to produce a smooth and solid gameplay experience. A camera at its worst can make the playing experience unbearable. One good example is an action-adventure hack n' slash game *MediEvil* (13). *MediEvil* had a promising design and lots of personality, but its terrible camera brought the whole experience down. The camera could only be moved if the character was moved. Players had trouble focusing on anything, the camera got stuck to objects, and combat became extremely frustrating.

The purpose of the camera is essentially to act as the player's eyes. Traditionally, when programming a camera, one has to create projection matrices by hand and carry on from there. Luckily, Unity has a built-in camera component that can be used as a base for everything. It already knows how to "see" and automatically acts as the player's eyes. The basic camera component has a couple of adjustable variables and properties, for example the field of view, the size of the viewport and rendering paths. (14, Graphics -> Graphics Overview -> Camera)

However, considering the genre and the type of game *Bonnie the Brave* is, the camera also needs to manage several other things in the spirit of 3rd person cameras, such as moving in the confines of the 3D world, handling collisions with the environment and behaving according to user inputs.

4.1 Basic gameplay behavior

The basic gameplay behavior as a term here covers the standard gameplay aspects in the 3D platformer genre. That is to say, how the camera should behave when taking into consideration basic movement like running and turning, jumping, platforming, and combatting, and when using Bonnie's skills and gadgets.

The camera designed in Bonnie can be categorized as a third person follow camera. The camera needs to follow the character in whichever direction she may move in whichever situation. It is positioned a fixed distance away from the character. The levels in game are open spaces, which means the player can run in all directions, not just forward and "away" from the camera eye. If the camera was to only look at Bonnie's back, the follow motion would be very stiff and probably even disorienting to the player. It would also limit the view during gameplay since small alterations to the character's movement would also move the camera. Therefore, the player's focus point would needlessly change.

The camera needs to offer the best possible view of the moving direction and the surrounding game world to ensure a satisfying playing experience. For this reason it was decided early on that the camera-player movement model would follow close to those of similar games, such as *Jak and Daxter* and *The Legend of Zelda: the Wind Waker* (2; 15). Practically, it means that the camera should be able to see any side of the character when she is moving around. For example, Bonnie can run in small circles in front of the camera with the camera physically not moving, or run directly at the camera. In the case of running directly right or left the camera will look at the character's side as she runs around the camera, or more specifically "orbits" it. With this simple technique, minor changes to the character's movement will not be a problem anymore, and the whole feel and look of the basic gameplay feels more natural.

Movement and jumping are particularly important gameplay factors in a game like this. Thankfully, getting the proper orbiting behavior for basic movement also signifies that general platforming and melee combat will work relatively well too.

In addition to the general behavior, the camera needs to be aware of the surrounding environment. It needs to collide with the bounds of the level and not go through walls or other solid game objects that define the geometry of the world. When colliding, it should also act so that the player's view is not obstructed or compromised. However, the camera should not collide with everything. Things such as leaves and other foliage usually allow cameras to go through them or alternatively they are compensated by other means.

4.2 Specific situations

At times, the player will face situations that require the camera to switch its behavior. These situations are usually events that happen in the surrounding world, by design or brought about by the player themselves.

Dying by dropping into a pit or a chasm is one of these events. In the case of falling it is a good idea to indicate this to the player by doing something with the camera in addition to other indicators like sounds. In *Bonnie the Brave*, if the player falls down, the camera should move from following Bonnie to above her, and look directly down as she falls.

Another good example is specific platforming sections. For example, platforming on a side of a wall can be dramatically better for the player if the camera is further away and showing the whole section from the side. This basically means mimicking a side-scrolling 2D game for a segment of the gameplay. Players can see the continuing and surrounding geometry much better. A comparison can be seen below (IMAGE 5). It also gives the game an appropriate nostalgic old school vibe.

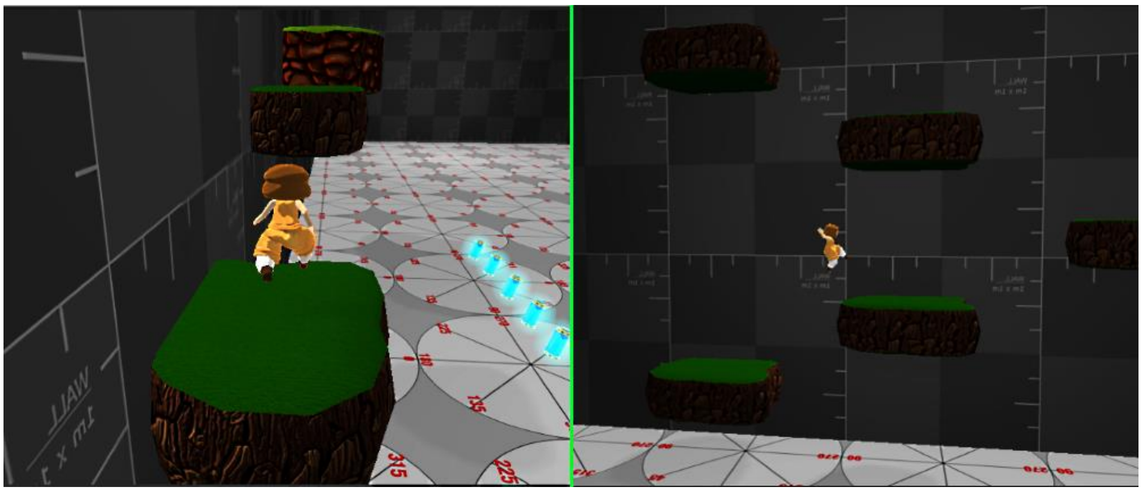


IMAGE 5. Platforming section on a side of a wall. Basic follow camera (left) vs. a specific platforming section camera (right).

Transitions like this can be accomplished by using multiple cameras and switching between them, or just using one camera and changing its behavior accordingly. *Jak and Daxter* handles this sort of single camera behavior alteration extremely well. The player rarely has to worry about not seeing everything they should.

4.3 Player adjustments

The aim was to create a camera that would allow decent enough gameplay without controlling it manually. However, manual controlling as a feature was always planned. Players would be able to rotate the camera around Bonnie using the right analog stick of the controller. As players become skilled, manual controlling allows them to complete actions faster and more accurately. For example, difficult platforming sections and combat scenarios where enemies approach from all directions are considerably easier to handle in this fashion.

4.4 User interface elements

UI-elements essentially mean the graphical elements the player sees in menus or on the screen during gameplay. These are the only occasions where the game uses a different camera. The standard menu camera is fixed so it does not move. Menu and HUD (Heads up display) objects just appear within the viewport.

The team had plans for using a separate camera for the map menu. The player could look at the map and move over it by moving the camera. The functionality was not finished completely for the demo and ultimately, a better solution was thought out.

5 CREATING THE CAMERA SYSTEM

When the design is solid and the vision clear, starting the programming of a component is much easier. We already knew what sort of situations the player has to face and how our camera needs to adapt. Building the component started by creating a simple working prototype-component which would be easy to expand once more features were designed, and the level design and level building progressed. Without a decent camera, early playtesting and level testing would have been a painful chore.

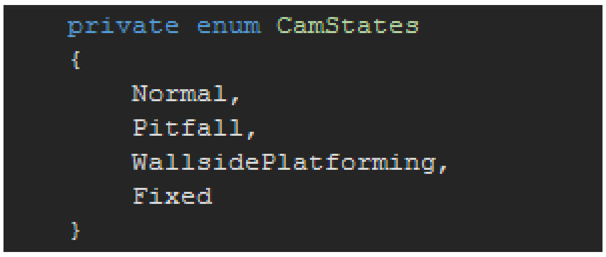
Because of the unique nature of game programming in modern engines, the testing of the components happens as they are programmed. Due to this, documenting the actual programming part differs from the actual work process and the linearity of it. Coding and testing go hand in hand in a rapid manner.

5.1 Scripting the third person camera

At this point of development we did not know our desired field of vision, how snappy the cameras movement ought to be, and how far or at which height the final camera should be related to the avatar. Therefore, it was beneficial to create sensible and adjustable variables. In Unity, this was easily accomplished by making the variables public. They can be adjusted from the Inspector-window even during runtime. This way, values can be tweaked during testing allowing the player instantly see the results.

5.1.1 General logic

The behavior logic that the camera component follows is technically a very rudimentary abstract machine called a finite-state machine (also known as FSM) (16, State). The camera has a fixed set of states. It can only be in one state at a time. Different events or inputs can launch the possible transition from the current state to a state pointed by the transition. Since only a handful of states need to be used, best practice was to use an enumeration for the state machine (IMAGE 6). An enumeration is a type consisting of a set of named constants. (16, State; 17, C# Keywords -> Types -> Value Types -> enum.)



```
private enum CamStates
{
    Normal,
    Pitfall,
    WallsidePlatforming,
    Fixed
}
```

IMAGE 6. The possible states for the camera to be in.

During normal gameplay, the camera is in *Normal*-state. This state is where the functionality of the camera is the most complex. The player has to be able to rotate, orbit and tilt the camera around the player character through inputs. The camera also needs to handle collisions with the level geometry so that it will not clip through objects. But most importantly it has to update its position and rotation with ease so that the player gets a clear vision of the character's headings and current orientation.

When the character enters a wall-side jumping section in the game, the camera transitions to a *Fixed*-state or a *WallsidePlatforming*-state. Unfortunately, because of the time restrictions during the development of the demo, the more intelligent *WallsidePlatforming*-state was scrapped. The functionality was not working flawlessly, so the team settled for the *Fixed*-behavior. In the *Fixed*-state, the camera moves to a fixed position in the world space where it continues to look at the player character from further away, giving the player a better view of the level and current section. The transition to this state is triggered when the player enters a specified area.

The last state that the camera can be in is called *Pitfall*. This state is only needed when the player character dies by falling into something e.g. a pit, a chasm or lava. The transition to this state happens when the player enters a specific trigger collider. The camera proceeds to move above the player and look down at the top of the characters head as she falls further down. It will not physically move with the character, but remains directly above the point in the world space where the avatar entered the trigger. Its only purpose is to indicate to the player that Bonnie has died. IMAGE 7 shows the finished camera transitioning to the *Pitfall*-state, looking at the character as she falls to her demise.

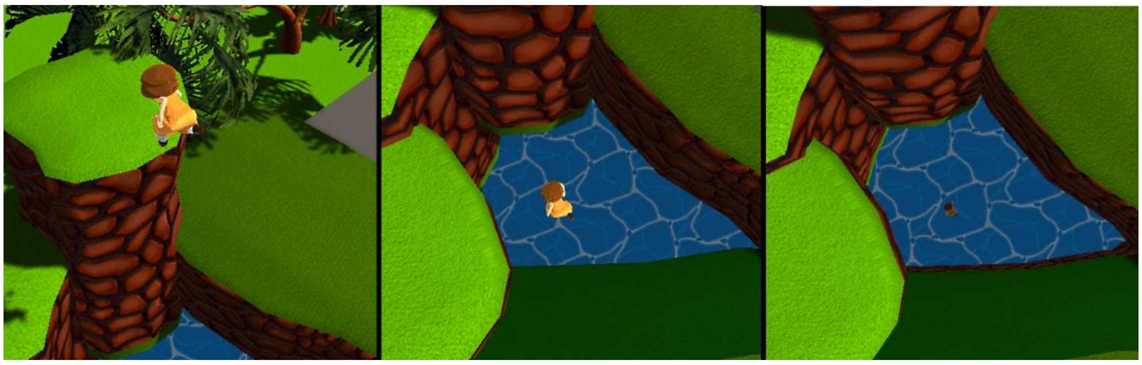


IMAGE 7. Pitfall-state.

5.1.2 Camera movement

As established earlier, the camera needs to smoothly follow the character from a certain distance and behave according to the character's movement and orientation. The character uses physics so its movement functions are updated in Unity's internal *FixedUpdate*-function. *FixedUpdate* is called every fixed framerate frame, and Unity's normal *Update*-loop is called every frame. If the behavior of the camera was placed in either of these loops, it could experience jagged movement because of the fast frame intervals. Therefore, to gain the smoothest outcome, the movement calculations of the camera are updated within Unity's *LateUpdate*-loop. *LateUpdate* is called after all other update-functions. In this way, for the current frame, all possible character movement alterations have been calculated, so the camera can move to its desired position without ragged transitions. (18.) In addition, the camera will not actually follow the character's transform, but rather a separate child transform placed to the character's center point.

The code structure for updating the movement of the camera is actually quite simple. Thanks to Unity's inner mechanics, much can be achieved with little. In the class's *Start*-method, the look direction of the camera is set to the direction our character is looking at, to ensure that at the start of the game the camera is already looking in the correct direction.

At the beginning of *LateUpdate*, the script starts by getting values from the analog control sticks of the game pad. After that the current state will determine how the camera wants to move. In the *Normal*-state, the first thing to set is the target position of the camera. This is the position where the camera wants to be in every frame.

As the purpose of the *Normal*-state is to essentially follow the player, a simple look direction vector is easily achieved by subtracting the current position of the camera from the character's position. Because the target position calculation originates from the character's transform, the camera will always look at the character's back, resulting in a very rigid movement. When the player is running in a circle, the camera is actually orbiting the player, whereas the goal is to get the character to orbit the camera. To reach the intended result, instead of calculating the target position based off of just moving back and up, it will calculate based off of moving a fixed distance from the character, no matter what way she is facing. An added Vector3-type variable was set to be calculated above the character's head. This offset replaces the character's position in the calculations. As a result the look direction is towards the offset which enables the camera to see any side of the character whilst still following her position.

Now that the direction vector from the camera to the character is set properly, it is normalized for a valid direction with a unit magnitude. Essentially, the targeted position is simply calculated by multiplying the normalized look direction vector by the desired away distance, and then subtracting the resulted vector from the desired camera height (character offset + distance up). By adjusting these distance multipliers the changing of the desired camera position is effortless during testing. The variables and parameters of the calculation are visualized in IMAGE 8. Note that in the picture the target position is directly behind the character, whereas during gameplay it can be at any angle around the character, depending on the character's movement and orientation.

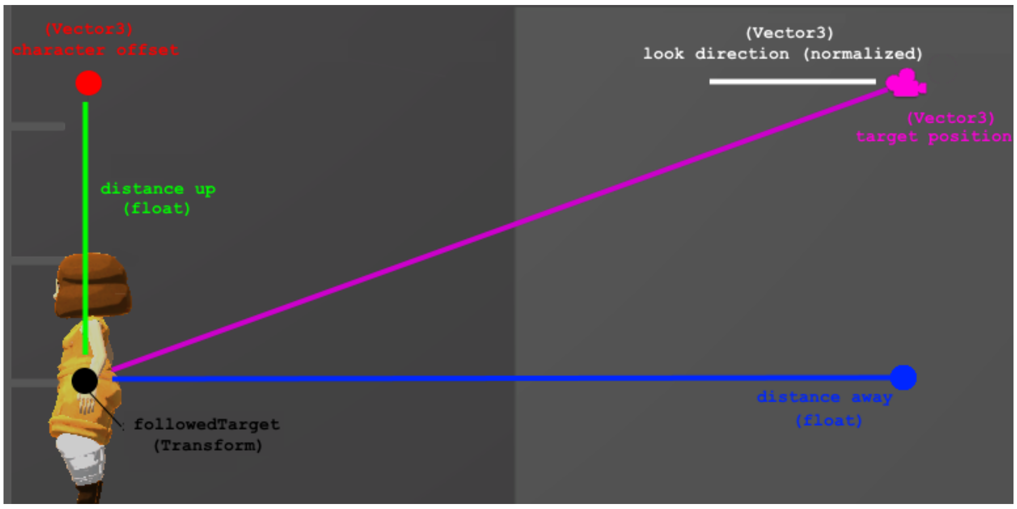


IMAGE 8. The visualization of the parameters used in calculating the target position of the camera every frame.

This approach works fine since most players rotate the camera manually to gain a better view. If the player does not control the position of the camera by hand, the camera is still able to orient to the character's moving direction rather well.

To ensure fluid movement, the camera will need to smooth its transition to the target position each frame. After determining the position for the current frame a function by the name of *SmoothTo-CurrentPosition* is called. This function will take two Vector3-type parameters: the current position and the target position, and gradually transition the vector towards the target value over time. Damping speed and time are variables determined in the class, so adjusting desirable values is quick and easy. Changing these values can have a drastic effect on the snappiness of the camera movement.

Finally, at the end of our *LateUpdate* for the *Normal*-state, the camera object needs to rotate so that its forward vector faces the character. The Transform-class' *LookAt*-function does this adequately enough (19). This concludes the update loop for the main functionality of the camera in the *Normal*-state regarding movement, rotation and general orientation.

As for the *Fixed*-state, the camera does not need to update its position. Upon the player entering a specified area within the level, an event triggers the FSM to change its state. In this state, the target position of the camera is a pre-determined position in the world space which means that the position of the camera only needs to be updated once. The transition is smoothed with a similar function to the previous *SmoothToCurrentPosition*-method. The *Pitfall*-state is triggered in the same manner. The only difference is that instead of a pre-determined position, the target position of the camera is calculated when the player character enters a trigger collider that is set around a pitfall.

5.1.3 Manual control and interactivity

With the features covered above, the camera is now intelligent enough for a standard gameplay. Playing the game is possible albeit unnecessarily arduous. However, with a simple added functionality it is possible to majorly enhance gameplay: the ability of enabling the player to horizontally rotate the camera around the character. Luckily, this can be accomplished with one line of code, thanks to Unity's built-in *RotateAround*-function (19).

Due to issues found in playtesting, more control was added later in the development. In addition to rotating the camera around the character's y-axis, players can also move the camera longitudinally, around the character's x-axis (IMAGE 9). This allows the player to peek over edges by pushing the analog stick forwards, or alternatively look up by pulling the stick back. In IMAGE 9, note the restrictions on the longitudinal axis, the player will not be able to rotate the camera below the character's feet or over her head. Normally, the ground would block the movement of the camera when moving down, but there are situations where this would not apply. For example, the camera looking at the character's back while she stands on a ledge, facing away from the drop.

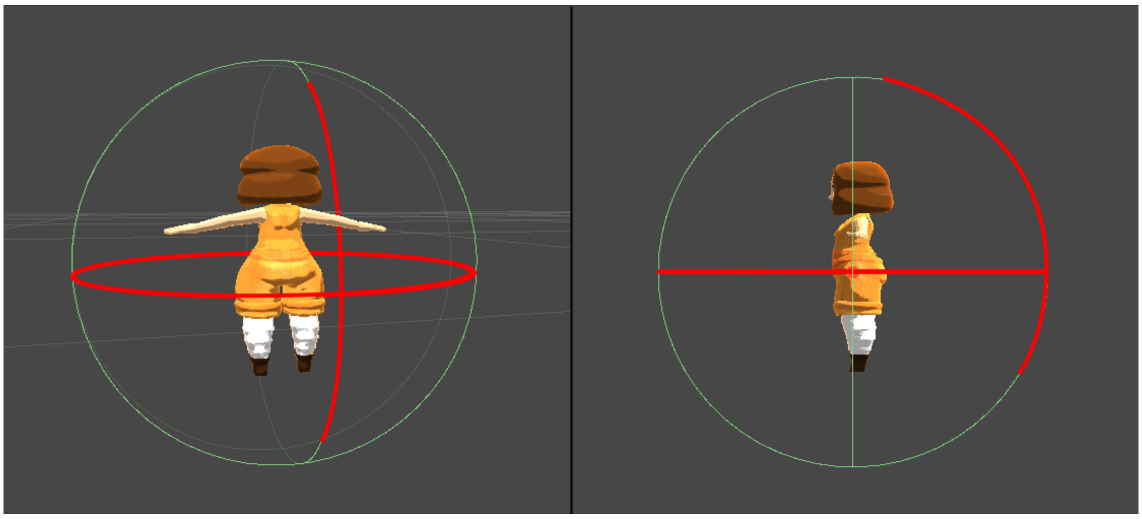


IMAGE 9. The axes along which the player can rotate the camera are depicted in red.

As stated in chapter 3.1, allowing manual camera control boosts efficiency and can prevent frustration. It is faster to navigate tricky jumping sections and handle combat when the player is in control of what they see. Structurally, all the magic happens just before calculating the target position for the frame.

5.1.4 Collision detection

It is important for the camera not to clip through objects and obstruct the vision for the player. Creating a good and smart clipping prevention functionality proved to be a difficult task. The system needs to work all-around in fundamentally every possible scenario. Such utility was eventually removed, since I was not able to make it work well enough in time. We settled on a simpler approach, which universally worked better and did what it needed. Compensation for leaves and other foliage was not included.

To detect collisions with the surrounding geometry, the camera will use raycasting (20). For example, what happens if the character runs “backwards” facing the camera, and the camera backs up to a wall? The camera is casting a ray from the character offset to the target position. When moving backwards, the ray of the raycast will register a hit to the wall since it is obstructing the way to the future target position (IMAGE 10).

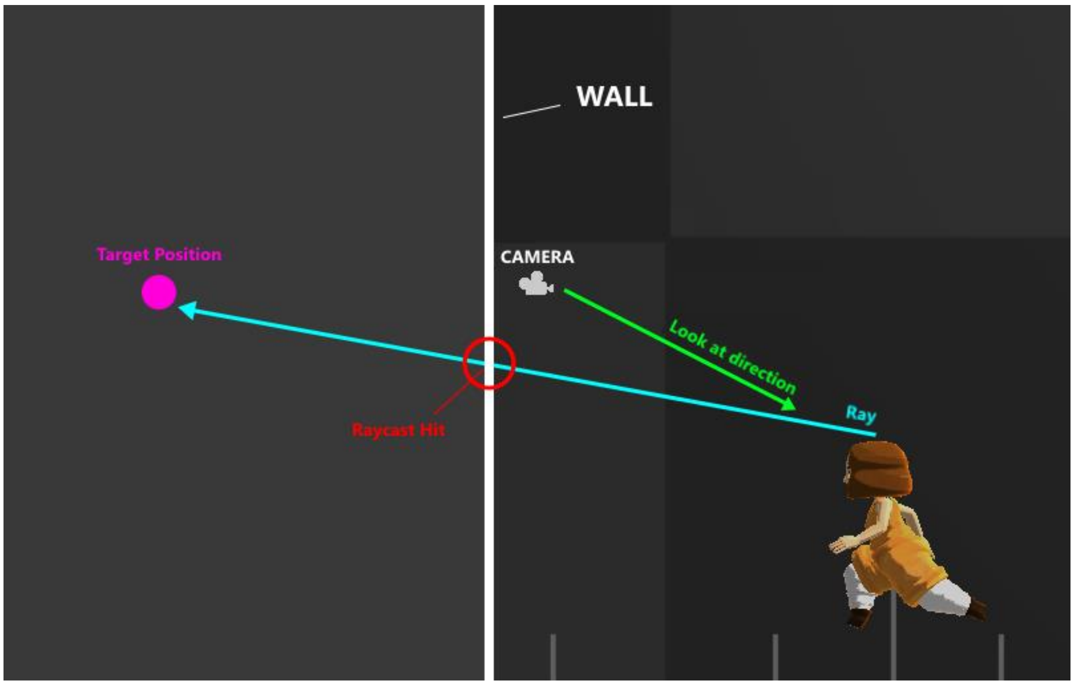


IMAGE 10. The character running “backwards” towards the lens.

At this point, we restrict the movement of the camera further in x-z-coordinate planes, therefore stopping at the wall and effectively “colliding” with it. The character can however still run at the camera. To prevent the character’s face being 1 millimeter from the lens and blocking the view, the camera will move upwards (IMAGE 11). Commonly, the cameras in similar games do this same shift up along the y-axis.

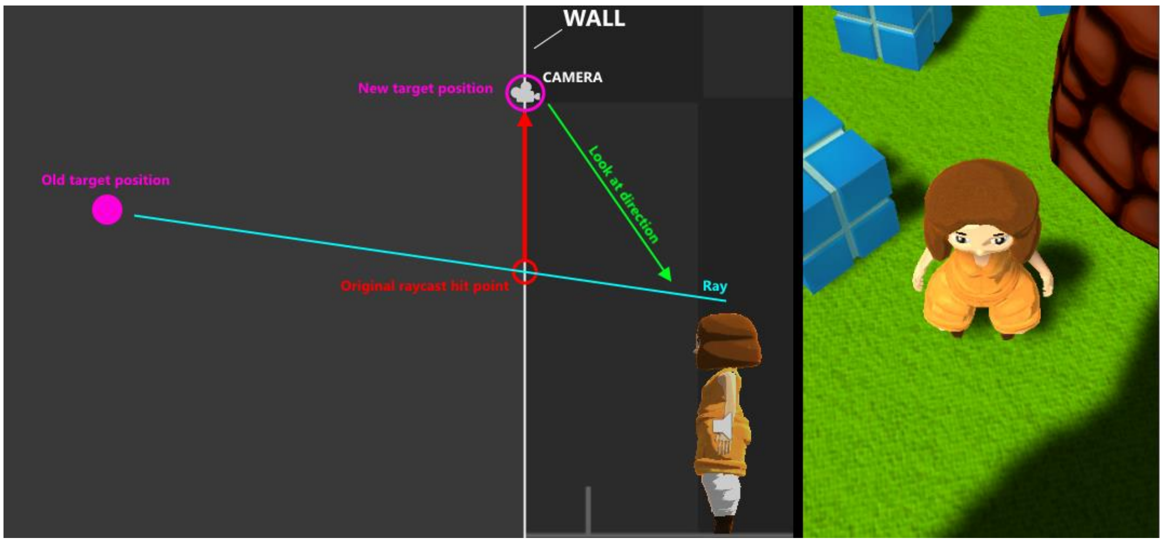


IMAGE 11. The visualized cross section of the camera's shift upwards along the y-axis (left), plus the camera's view from a similar situation in-game (right).

If the player still insists on moving towards the wall, the camera will swiftly flip its position to behind the character, normalizing the view for the player. This can also prevent the camera for getting stuck, moving in a peculiar way or obstructing the view in tight spots.

Testing initiated a need for improving the collision detection method. When the character descends stairs or a hill, the camera gets very close to the ground, which again does not provide the best necessary view for the player. This was compensated by adding another raycast. If the camera gets too close to the ground, it will reposition itself by moving directly upwards.

Another thing added near the end of the development phase was disabling manual camera movement when the camera is colliding with something. In these situations moving the camera by stick seemed to produce weird behavior and sticky movement. Scripting-wise this was extremely simple, only one flag was added.

5.2 Testing

As mentioned earlier, the testing of the camera component was handled during the programming and playtesting. It is crucial to implement features quickly and detect problems equally fast. Personally, I do not recall many issues since they were not documented in any way. The nature of the development style is mostly trial and error, fail fast and fail often. Writing a line of code and running the game in the editor is quick and effective. As for the camera, getting the specific *WallsidePlatforming*-state to work sufficiently was difficult. It was playtested longer and more thoroughly than the other camera features. Due to the nature of the levels, operability was not satisfactory and time for the task ran out. The state was dropped completely but user experiences and issues were noted. Fixing the problem in the future will be an easier job thanks to the gathered information.

Playtesting also initiated the need for new functionalities and tweaks. For example, the ground collision checks for the camera, allowing players longitudinal camera movement for peeking over edges, looking upwards and generally improving the collision behavior in tight spots. Overall, a lot of these tweaks and improvements were found when playing the game. It can be challenging to cover everything in the design, so some aspects of gameplay and playability can remain contingently overlooked.

All in all, playtesting is crucial throughout the whole development process. Unfortunately the “trend” in the industry is that studios release unfinished games to get them out fast. The results are ugly and gamers unhappy, as heaps of bugs and glitches ruin the whole playing experience. The importance of proper testing is utmost. Luckily for the camera component, testing without playing the game is impossible, so it was well tested in many conditions. The performance of the component is solid and light. There are no bugs and inconsistencies in the parts that work. Only the issues with clipping remained, but then again, the particular functionality could not be finished in time.

6 ENEMY TARGETING

Combining components and their mechanics can be a huge timesaver. Of course, such integrations might become a chore, but with a good design it is possible to kill two (or even more) birds with one stone. The basic 3rd person follow -type camera mechanics can easily be utilized and used as a base for other gameplay features.

In *Bonnie the Brave*, the main character Bonnie will acquire a gadget that is essentially a ranged combat weapon. It can be used to shoot enemies, activate platforms, activate buttons that open doors and other similar elements that change the environment in some way. (21, p. 21). This gadget (called the Ankle-Activator) features a semi-automatic aiming system. The functionality of this feature was based on what the camera component (and therefore the player) sees and how it moves.

6.1 Design parameters

Designing the component was a bit tricky. A few important factors needed to be considered before starting the programming of the component. First of all, the team had to decide whether the aiming was based on the character's orientation or on the orientation of the camera. This had no real effect on the difficulty of programming, but rather on the gameplay issues it might bring. It was quickly established that it would be better if the automatic aiming would be based on the camera. Since the player can manually decide where to look at, it felt natural that the looking direction should also define the direction of the aiming vector. This would also mean that it is possible to aim behind the character's back. However, it is quite silly to imagine someone shooting blindly behind their back and hitting a target perfectly. To fix this, it was decided that the character would always face the direction of the aim when the gadget was active.

The aiming system itself is almost completely automatic. The player needs to activate the system by pressing a button and decide where to aim by using the camera controls. The aiming system needs to recognize different elements in the world and automatically aim at them. When it is locked to a target, the player can launch a projectile by pressing another button. The projectile will fly towards the position of the object at the time of shooting. This means that even if the aim is automatic, the launched projectile can miss if an enemy moves out of the way fast enough. The system also needs to handle groups of targets. For example, if the player is aiming at a group of enemies, the system will automatically lock to the closest one to the character. If the player keeps the gadget active and shoots the target and destroys it, the system will need to automatically lock to the second closest target in the group (IMAGE 12). The current locked target is visualized by a beam of light from the character to the targeted object.

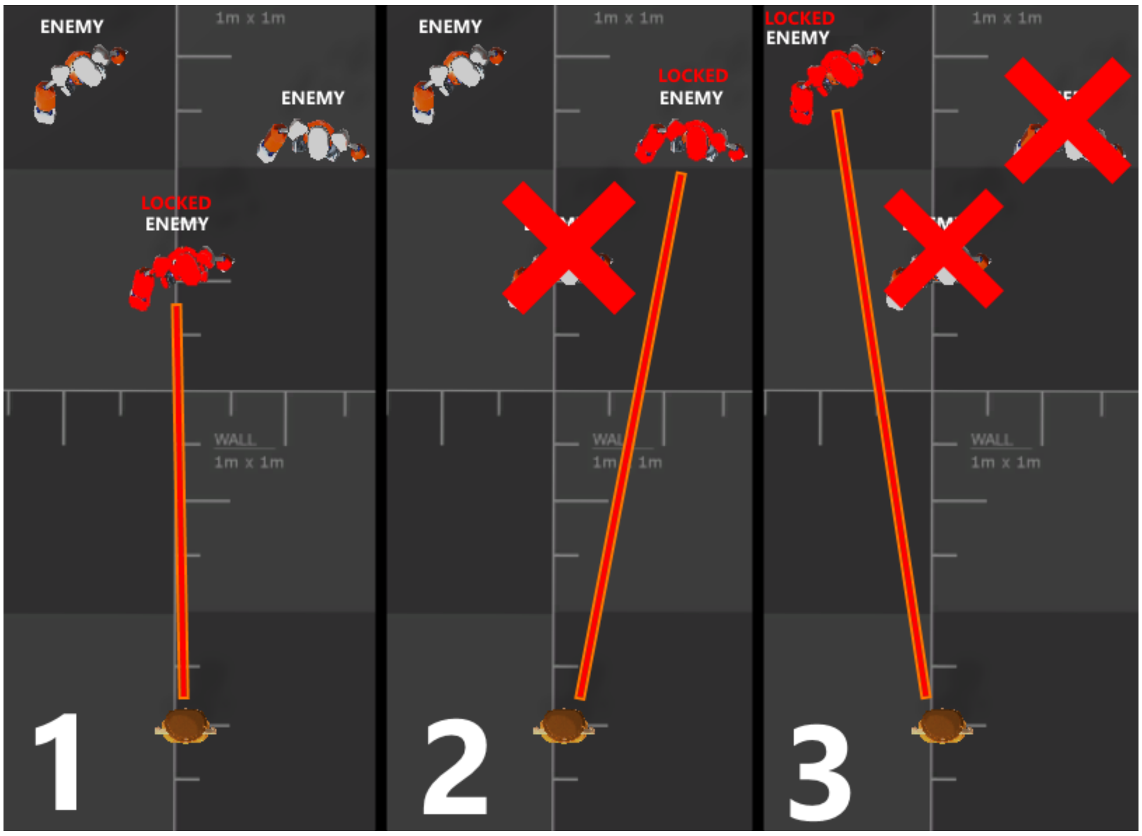


IMAGE 12. The targeting system locking to the nearest enemy in the group.

In consensus: the aiming system needs to detect single objects as well as groups of objects, and quickly lock to the intended target. The player is unable to aim at them manually, but they can decide the general direction of aim by controlling the camera. The aiming system should be able to handle the rest.

6.2 Choosing the best method

The design parameters for the component were quite flexible, which left a few choices of how to craft the system.

6.2.1 ViewPortPointToRay

Unity's Camera-class has a *ViewPortPointToRay*-method that creates a ray through a viewport point of the camera. The ray can then be used to detect hits in the world space. (22.) However, one simple ray will not be enough since the aiming system needs to detect multiple targets at once. Due to this, *ViewPortPointToRay* quickly became redundant. The function would be the simplest solution to use if the player could aim at something manually.

6.2.2 Raycast sweeps and SphereCast

Using rays is a nice way to detect collisions in Unity. However, detecting multiple targets fast or at once requires the use of multiple rays or sweeping one ray across an angle or an area. Since it is exactly what the aiming system requires, a couple of varying functions were created. One of the functions cast multiple rays through several viewport points. Another one was a sweeping function that swept one ray over an angle (a sector). Finally, a method that experimented with the *Sphere-Cast*-function of Physics-class was coded. *SphereCast* is effectively a sphere cast along a ray that returns information on colliders that hit it. (20.)

All of these raycast-based methods were able to return usable data, but whenever the aiming was active, there was a noticeable spike in the performance. If the player keeps the gadget active, the component needs to cast rays and process the hit data every frame. While the performance hit was not crucial, I was certain that a better solution could be found.

6.2.3 Triggers

Another way to detect collisions effectively within Unity is using colliders. That way it is possible to utilize Unity's own event functions. These functions return information regarding the collisions. For example, it is possible to detect when a something enters a trigger collider, leaves it or stays within its bounds. This is a superior way compared to raycasts due to the nature of the aiming system. Instead of constantly casting rays, gaining hit data can be examined through the event functions which are less performance heavy. (23.)

6.3 Basic mechanics

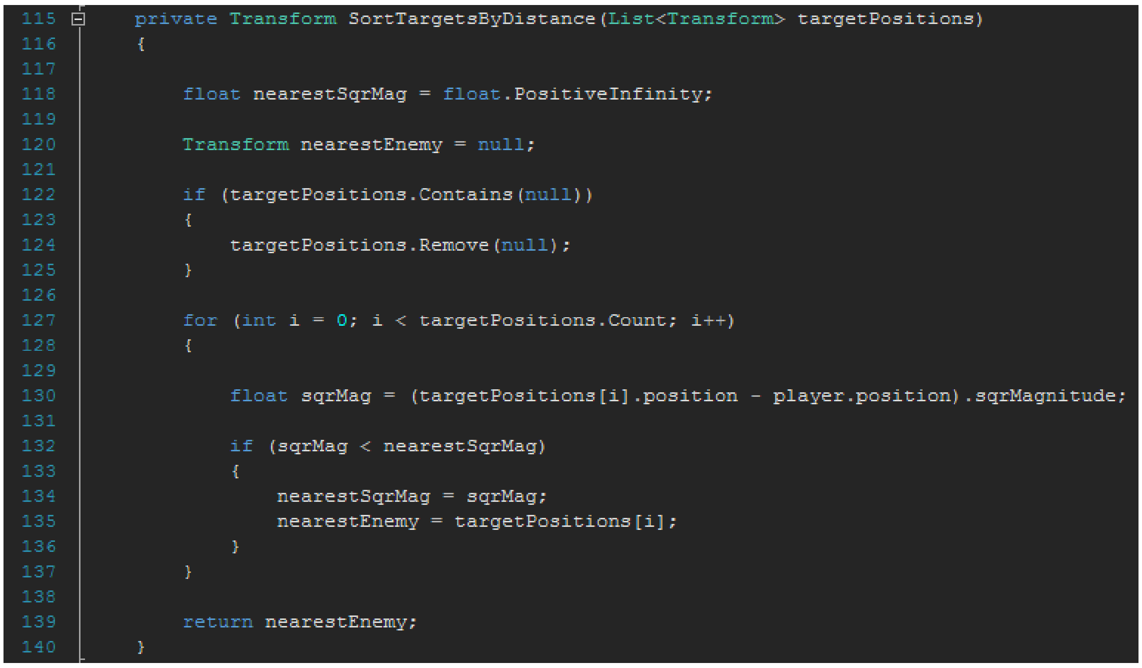
Trigger colliders were the best option to go for. Using a trigger also makes the system simpler in general. A collider object can be added as a child to the camera. Therefore, it automatically moves with the camera and responds to the orientation and rotation of the camera. Whichever way the player is looking at, the aiming system will now automatically work in the intended direction. Changing the bounds of the collider makes it easy to adjust the reach and affecting area of the aim. The collider component is off by default. If the player activates the gadget the collider is enabled. If a viable target enters or is within the bounds of the collider, the aiming system will recognize it and act accordingly.

6.4 Scripting the Targeting class

The player character Bonnie has different resources. When using the Ankle-Activator to shoot, she uses her Energy-resource. These energy points can be imagined as ammo for the character's different gadgets and skills. (21, p. 19). In the Targeting script's *Update*-loop, the first thing is to check for the user inputs and if Bonnie has enough energy points to perform the intended action (shooting, in this case). In addition to resource checks, some other checks are conducted in order to prevent null references.

When the gadget is active, the trigger collider for the aiming system is enabled. As mentioned before, handling the actions happens through Unity's event functions. These functions are *OnTriggerEnter*, *OnTriggerStay* and *OnTriggerExit*. Like the names suggest, *OnTriggerEnter* is called every time a collider enters the trigger, *OnTriggerExit* is called when something exits the trigger's bounds. *OnTriggerStay* is called every frame if an object with a collider touches it. (23.)

Whenever a detectable target object enters our aiming system's trigger, it is added to a list (24, links System.Collections Namespaces -> System.Collections.Generic). This List object contains all the targets that our aiming system recognizes when active. When the aiming is active and all the targets remain within the trigger's bounds, *OnTriggerStay* will call another function that selects and locks to the closest target object. A method called *SortTargetsByDistance* (IMAGE 13) goes through the list of targets and calculates which is the closest to the character.



```
    private Transform SortTargetsByDistance(List<Transform> targetPositions)
    {

        float nearestSqrMag = float.PositiveInfinity;

        Transform nearestEnemy = null;

        if (targetPositions.Contains(null))
        {
            targetPositions.Remove(null);
        }

        for (int i = 0; i < targetPositions.Count; i++)
        {

            float sqrMag = (targetPositions[i].position - player.position).sqrMagnitude;

            if (sqrMag < nearestSqrMag)
            {
                nearestSqrMag = sqrMag;
                nearestEnemy = targetPositions[i];
            }
        }

        return nearestEnemy;
    }
```

IMAGE 13. The SortTargetsByDistance-method in the Targeting-script.

This enables the aiming system to adapt to the character's and enemies' possible movement and will always keep the lock to the closest target (IMAGE 12). A vector from the character to the locked object is then calculated. If the player shoots at the target, the projectile will fly along this vector. If the target is destroyed, it is removed from the list. In the case of multiple targets, *SortTargetsByDistance* does the math again next frame and a new target is locked on to. When an object leaves the aiming system's trigger, *OnTriggerExit* is called and the object is removed from the targets list. If the gadget is deactivated, the list is completely cleared.

The functionality of the system is rather simple, but it works wonders. By delving a bit into different methods on how to reach the same result, it was possible to maximize the outcome and obtain flawless design-fulfilling results. The targeting system performs greatly and demands very little from performance. There were no real problems when developing it, some null reference bugs being the only hiccups. Best of all, the previously created camera created a shortcut and worked as a base for the component. Time consumption was not a hindrance.

The camera component and the targeting component together are a part of a bigger ensemble called a camera rig. With a good base camera, more components and layers can be added to the rig if needed. For the scope of the project and this thesis, it was not necessary add more features.

There were plans of a first person camera mode and given how the camera was built, adding the feature would not pose an issue. Also, effects and other post-processing scripts were pretty much left out. Basic anti-aliasing and SSAO were used and those scripts were provided by Unity technologies.

7 THOUGHTS AND CONCLUSIONS

I had no idea for the subject of the thesis when I started working on the camera component for *Bonnie the Brave*. The game itself was my main priority, not just the camera. As the first camera-related functionalities were implemented in the game, I started seeing the potential for a thesis that truly interested me and was of a proper scope. However, I still had issues with narrowing down the actual subject. For a long time I was thinking about going with a more detailed, technical approach, but ended up discarding that choice. Thanks to a great counsel from the team members, I was able to lock down a fitting scope and subject. The main aim was to develop a camera that would fulfill the design requirements of the game and provide a great gameplay experience.

Overall, the aim of the thesis was achieved, at least in terms of what the project required. The issue with game development is that things can be polished almost infinitely. The collision detection methods and clipping issues were the only parts that needed more work. The system would need to deal with foliage better, perhaps through the use of transparency or just improved positioning. Unfortunately time was scarce during the development and some aspects of components need to be cut down. The development of the other main functionalities did not pose any big issues. I was quite satisfied with the movement and orientation of the camera during gameplay. Moving the character in the game's environment is fluid and easy.

The camera was liked by players and received great feedback. It was not intrusive, performed smoothly and gave the players nostalgia from similar games from the past, which is exactly what Bonnie the Brave was set out to do. Another great thing about the main component is that it can be used for other third-person games, too. Possible future improvements would be adding effects and more cinematic features, such as camera shakes and post-processing effects.

I would like to point out one issue that arose when I started documenting and writing the thesis. Finding proper sources and literature about video game cameras was hard, and it still is. This was mildly annoying as I truly wanted to delve deep into the subject. When I encountered the first eligible source and read the camera design chapters, I found that most of it was already familiar information to me. The fact to the matter is that most people who have played games their whole lives and gone on developing and designing later on, already know how game cameras and other components should work to provide a satisfying experience. They have figured it out through playing great games (and terrible games). There are no strict rules and research about different video game camera types, it is mostly common sense, design decisions and experience. In the end, developers just want to make their games play well.

REFERENCES

1. Piippo, A-M. 2014. Game Design Document – General Information and Overviews. Bonnie the Brave – Space Courier team. Available on request.

2. Naughty Dog, Inc. 2001. Jak and Daxter: The Precursor Legacy. Video Game. Available at: https://www.playstation.com/en-us/games/jak-and-daxter-collection-ps3/

3. Insomniac Games. 2002. Ratchet and Clank. Video Game. https://www.playstation.com/en-us/games/ratchet-and-clank-collection-ps3/

4. Unity Technologies. 2015. A Feature-rich and Highly Flexible Editor. Date of retrieval 1.4.2015. http://unity3d.com/unity/editor

5. Unity Technologies. 2015. Build Once Deploy Everywhere. Date of retrieval 1.4.2015. http://unity3d.com/unity/multiplatform

6. Microsoft. 2015. Application Development. Date of retrieval 22.3.2015 https://www.visualstu-dio.com/features/

7. Microsoft. 2015. Visual Studio with MSDN. Date of retrieval 22.3.2015 https://www.visualstu-dio.com/products/.

8. Adams, E. 2014. Fundamentals of Game Design, 3rd Edition. New Riders.

9. Nintendo. 1996. Super Mario 64. Video Game. Available at: https://www.nintendo.com/games/detail/TOEiNDty4AHQnsUnC2Eyog0kBeitORN

10. Nintendo. 1996. Super Mario 64. Screenshot. Date of retrieval 2.4.2015 http://static.gi-antbomb.com/uploads/scale_small/1/17172/1250631-sm64bobombbattlefield.png

11. Bethesda Softworks. 2011. The Elder Scrolls V: Skyrim. Video Game. Self-taken screen-shots. Available at: http://www.elderscrolls.com/skyrim.

12. Supergiant Games 2014. Transistor. Video game. Self-taken screenshot. Available at: http://www.supergiantgames.com/games/transistor/.

13. SCE Cambridge Studio 1998. MediEvil. Video Game. Remake available at: https://www.playstation.com/en-us/games/medievil-resurrection-psp/

14. Unity Technologies 2015. Unity Manual. Date of retrieval 1.4.2015 http://docs.unity3d.com/Manual/index.html

15. Nintendo. 2002. The Legend of Zelda: The Wind Waker. Video Game. Available at: http://www.zelda.com/windwaker/

16. Nystrom, R. 2014. Game Programming Patterns. Date of retrieval 13.3.2015 http://gamepro-grammingpatterns.com/contents.html

17. Microsoft. 2015. Visual C#, C# Reference. Date of retrieval 22.3.2015 https://msdn.mi-crosoft.com/en-us/library/618ayhy6.aspx

18. Unity Technologies. 2015. Scripting Reference. MonoBehavior. Date of retrieval 13.4.2015. http://docs.unity3d.com/ScriptReference/MonoBehavior.html

19. Unity Technologies. 2015. Scripting Reference. Transform. Date of retrieval 13.4.2015. http://docs.unity3d.com/ScriptReference/Transform.html

20. Unity Technologies. 2015. Scripting Reference. Physics. Date of retrieval 13.4.2015. http://docs.unity3d.com/ScriptReference/Physics.html

21. Piippo, A-M. 2014. Game Design Document – Detailed Descriptions and Asset Breakdown. Bonnie the Brave – Space Courier team. Available on request

22. Unity Technologies. 2015. Scripting Reference. Camera. Date of retrieval 13.4.2015. http://docs.unity3d.com/ScriptReference/Camera.html

23. Unity Technologies. 2015. Scripting Reference. Collider. Date of retrieval 13.4.2015. http://docs.unity3d.com/ScriptReference/Collider.html

24. Microsoft. 2015. .NET Framework Class Library. Date of retrieval 22.3.2015 https://msdn.mi-crosoft.com/en-us/library/gg145045(v=vs.110).aspx