BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

DESIGN AND IMPLEMENTATION OF A SURVIVAL COMPUTER GAME

NÁVRH A IMPLEMENTÁCIA SURVIVAL POČÍTAČOVEJ HRY

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **ONDREJ KOVÁČ**
AUTOR PRÁCE

SUPERVISOR **doc. Ing. MARTIN ČADÍK, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

146142

Institut:	Department of Computer Graphics and Multimedia (UPGM)
Student:	**Kováč Ondrej**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**Design and Implementation of Survival Computer Game**
Category:	Computer Graphics
Academic year:	2022/23

Assignment:

1. Explore the history and state of the art of design of survival computer games.
2. From the existing game engines, choose the one suitable for implementation. Familiarize yourself with the chosen game engine and describe its features.
3. Design a new survival game based on the findings.
4. Implement the proposed game and experiment with different designs and game mechanics in successive iterations.
5. Present the resulting game in the form of a poster and a short video.

Literature:

- Koster, Raph. Theory of fun for game design. O'Reilly Media, Inc., 2013.
- Schell, Jesse. The Art of Game Design: A book of lenses. CRC press, 2008.
- Unity Learn. Unity, https://learn.unity.com/.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Čadík Martin, doc. Ing., Ph.D.**
Head of Department:	Černocký Jan, prof. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	10.5.2023
Approval date:	24.1.2023

Abstract

This bachelor's thesis focuses on the design and implementation of a post-apocalyptic, survival computer game with RPG elements named "Beneath the Mushroom Clouds". The thesis contains a short introduction to the game industry and genres describing the game, general overview of game engines and specific aspects of the Unity game engine. Furthermore, it describes the design and implementation of some of the individual elements that make up the game. The implementation of these elements is described conceptually, with minimal examples of actual code.

Abstrakt

Táto bakalárska práca sa zameriava na návrh a implementáciu postapokalyptickej survival počítačovej hry s RPG prvkami nazvanej "Beneath the Mushroom Clouds". Práca obsahuje krátky úvod do herného priemyslu a žánrov opisujúcich hru, všeobecný prehľad herných enginov a špecifické aspekty herného enginu Unity. Ďalej popisuje návrh a implementáciu niektorých individuálnych prvkov, ktoré tvoria danú hru. Implementácia týchto prvkov je popísaná konceptuálne s minimálnymi ukážkami kódu.

Keywords

Computer game, Game design, Game implementation, Unity, BTMC, Survival, Post-apocalyptic

Kľúčové slová

Počítačová hra, Návrh hry, Implementácia hry, Unity, BTMC, Survival, Postapokalyptická hra

Reference

KOVÁČ, Ondrej. *Design and Implementation of a Survival Computer Game.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Ing. Martin Čadík, Ph.D.

Design and Implementation of a Survival Computer Game

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Martin Čadík. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....

Ondrej Kováč

May 9, 2023

Contents

Chapter 1

Introduction

Gaming industry is one of the **biggest** and **fastest-growing** entertainment industries in the world. It has already surpassed the music and the film industry [20]. In 2021, the gaming industry's worth was estimated to be roughly **$180 billion** [20] [21] [15]. While the majority of this sum is generated by the largest and well known game development studios such as *Microsoft*, *Sony* or *Nintendo* [4], majority of games are created by small, **indie** developer studios and individuals.

In recent years, we have seen an increase in games focused around one particular genre: **survival**. These games feature an open-world environment where the main objective is to keep your character alive for as long as possible by scavenging for resources. These games are often designed to be punishing and unforgiving, with all of the player's progress lost when the character inevitably dies. While to some this may seem like a terrible way to entertain themselves, these games are increasingly more popular amongst players. This may be attributed to the base survival instincts of humans and to the desire for freedom to create one's own story [11] [19].

This thesis focuses on the design and implementation of **„Beneath the Mushroom Clouds“ (BTMC)**, a **hardcore**, **post-apocalyptic**, **survival** computer game implemented using the **Unity** game engine and written in the **C#** language.

Chapter 2 briefly summarizes video game **genres** relevant to BTMC and mentions other games that served as inspiration for its creation.

Chapter 3 provides a general overview of **game engines** and specific aspects of the **Unity** game engine used in the implementation of this game.

Chapter 4 contains the **initial design** of the game, as it was described in its **game design document (GDD)**, together with notes about changes made during implementation.

Chapters 5 - 9 consist of conceptual explanations of how individual elements which make up the game were **implemented**.

Chapter 10 contains a short description of **user testing** of the game.

Chapter 11 summarizes the state of the game at the time of writing of this thesis and the author's newly **gained experience** in the design and implementation of video games.

Chapter 2

Video Game Genres

In this chapter I will talk about the different genres of video games, specific examples and games that inspired the creation of BTMC.

These days, video games can hardly be described by a single genre. Most games are a **blend of different genres** which should give us an idea of what to expect from the game. The most prominent genres in the modern video game industry are[13]:

- Sandbox
- Real-time strategy (RTS)
- Shooters (first-person and third-person)
- Multiplayer online battle arena (MOBA)
- Role-playing (RPG, ARPG, and more)
- Simulation and sports
- Puzzlers and party games
- Action-adventure
- Survival and horror
- Platformer

2.1 Related Game Genres

BTMC would be best described as a blend of the following genres: **Survival**, **Post-apocalyptic**, **Role-playing game (RPG)**, **Third-person Shooter (TPS)**. In the following sections I will broadly explain these genres.

2.1.1 Survival

Survival is the **primary** genre of BTMC and all of the games which served as inspiration contain some aspects of survival.

Character's needs

The most prominent mechanic in almost all of the video games that fall into the survival genre are the fundamental **needs** necessary for survival of the player character. These **three** needs will be found in almost any survival game:

- **Health** - An indication of how „alive" your character is. This is in most games represented by **hit-points**. If your hit-points drop to zero your character dies. You will often get injured in survival games and therefore you need to **heal** your character. In many newer survival games a more complex health system is being implemented compared to older games. Instead of simply taking some cure-all pills to increase your hit-points or bandaging the same arm over and over again, you have to address a *specific wound* by using *specific medicine*. Bandages for bleeds, splints for broken bones, antibiotics for infection and so on. This creates a much more interesting and nerve-wrecking experience, where the player might have all sorts of medical items on hand but not the one they need at that moment.

- **Hunger** - Everyone needs to eat and the player character is no exception in most survival games. Hunger can be implemented in many ways. In some games, it is simply a **hunger bar** which is filled by eating food and depleted over time. Once the hunger bar is depleted, the character starts slowly losing health and dies (*Don't Starve*). Some games take a bit more *sophisticated* approach to hunger. For example, a game might not consider only the caloric value of food but also what **macro-nutrients** the food contains (*Project Zomboid*). Some games might even require your character to keep balanced **vitamin levels** in their bloodstream(*SCUM*). A low hunger bar might also result in **debuffs**, such as no health regeneration, inability to run or reduced damage.

- **Thirst** - The one resource even more important than food is water. This need is very commonly implemented in the same way as hunger. Either as a simple thirst bar or a more complex system with negative effects associated with dehydration.

While these are the main needs found in almost all of survival games, some games implement other needs as well, such as:

- Sleep
- Body Temperature
- Boredom
- Socialization
- Defecation

Then there are games that have some **specific** need which is often not found in other games and serves as an interesting **gameplay mechanic** or a **story device**, such as sanity (*Don't Starve*) or malaria that needs to be regularly treated with medication (*Far Cry 2*).

Figure 2.1: Examples of status bars in different games - Don't Starve (left), The Forest (center), Unturned (right)

Dangerous scenarios

Most survival games rely on dangerous scenarios to create an engaging experience, especially if the game doesn't heavily rely on story. **Hostile NPCs** are a common element that serve to create these situations. These NPCs can take many forms, such as zombies, wild animals, large sea creatures, mutants, or hostile humans.

However, some survival games do not need to rely on NPCs at all. In **multiplayer** survival games, often all that needs to be done is to throw players into a survival scenario and leave them to fend for themselves. Soon enough they will either start turning on each other or forming groups. In some games this is the core gameplay loop. Find a group, collect resources, build a base and *raid* other player's bases (*Rust*).

Inventory system

In vast majority of survival games, the player is limited by their **inventory capacity**. This forces the player to consider what they will be carrying with them, how many resources they can gather and so on.

There are many different ways to implement an inventory system and limit the amount of items the character can carry. These are the most common inventory systems:

- **Grid-based system** - The character's inventory is split into individual **cells**. These cells form a **grid** into which the player can place the items they find. While in some games each item takes up only one cell (*Minecraft*), other games use this system to simulate the **volume** of objects. In these games, larger items span over multiple cells which means that the **organization** of inventory is important(*DayZ*).

- **Weight system** - Instead of focusing on the size of the carried items, their **weight** is what matters (*Project Zomboid*). These games will often include a specific debuff - **over-encumbrance**. When the character is carrying items over their weight limit they will slow down, won't be able to run and so on.

- **Combination of grid-based and weight system** - This is perhaps the most realistic, widely used inventory system. The player has to consider both the size of the item and its weight (*Escape From Tarkov*).

- **Maximum amount of specific resource** - Some games simply give the player specific **inventory containers** for each individual resource in the game. For example, the player can carry 20 sticks, 50 stones and 100 leaves. It does not matter what resources they have, the player can only carry a specific maximum amount of each resource.

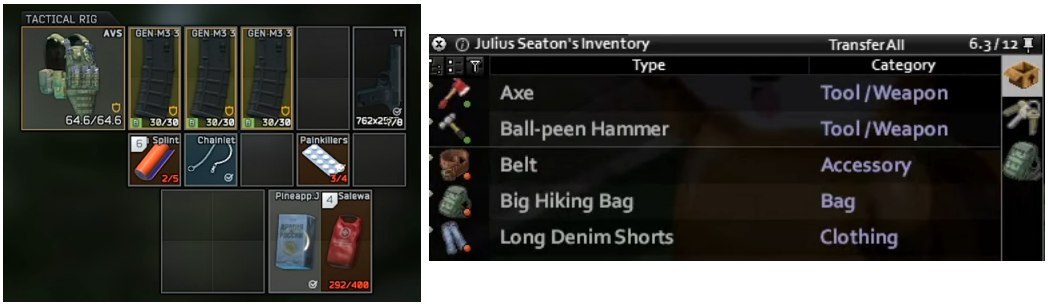

Figure 2.2: Examples of inventory systems - Combination of grid-based and weight system from Escape From Tarkov (left) and a Weight system from Project Zomboid (right)

2.1.2 Post-Apocalyptic

Post-apocalyptic is not only a genre of video games, but one of the genres of fiction as a whole. Instead of directly affecting gameplay, such as genres mentioned above, it defines the **setting** of the game. Works of post-apocalyptic fiction take place after some catastrophic event has occurred. The main characteristics are struggles to survive in a dangerous, devastated world filled with ruins and rubble of buildings. Without basic necessities people are accustomed to in modern times such as electricity, running water, readily available food or laws, humans might often turn on each other. This may cause further instability and a complete collapse of societal structure[14] [8].

Anyone can very quickly recognise this setting as perfect for survival games. Actually most survival games fall into either the post-apocalyptic genre or a *stranded-on-an-island* type scenario. While survival games in which the player gets stranded away from civilization will put their focus on surviving in **nature** by gathering wood, stones and hunting wild animals, post-apocalyptic scenarios often require scavenging in the **ruins** of the civilization that once was.

2.1.3 Role-playing game (RPG)

The RPG genre has a long history, predating the popularization of video games. Role-playing games used to originally be **tabletop** games. The origin of this genre is a bit debatable, however the first commercially available game recognized as an RPG was *Dungeons and Dragons (D&D)*, which was developed in 1974.

It took just a few years before the first RPG *video games* were developed. The first RPG video games were not created commercially for home computers but instead ran on large **mainframes** in American universities. All of these games were somewhat influenced by D&D. A large boom in the RPG video game industry began in the '80s with games such as *Rogue*, *Akalabeth: World of Doom*, *Wizardry* or *Ultima* serving as the foundation for the RPG video game genre [12] [5] [2].

While almost all of the early RPG titles are set in *fantasy* worlds with different races of humanoids or magical creatures RPGs with other themes were created. The main characteristic of an RPG are **non-linear storytelling**, **quests**, **encounters** and **developing characters**.

2.1.4 Third-person shooter (TPS)

While BTMC is not a shooter per say, a large part of the gameplay loop and enjoyment comes from encounters with enemies. These encounters will often include **firearms** or other ranged weapons. Shooter games are one of the most prominent genres of video games. Most of these shooters are first-person shooters (FPS). That means that the player is looking through the eyes of the character. In TPS, players see their character from a different perspective but still control the character's aim.

2.2 Inspiration from other video games

BTMC draws a lot of inspiration from other video games. That's why I believe they should be briefly mentioned in this thesis together with the mechanics and themes that each game brought into this project.

2.2.1 Project Zomboid

Project Zomboid is one of the main inspirations for this game. The game was first released in 2013 by an indie development studio *The Indie Stone*. It's a post-apocalyptic, open-world, zombie survival game with very deep internal mechanics.

The features inspired by this game are:

- Post-apocalyptic setting
- Character skills
- Complex healing and debuff system
- Needs such as sleep or body temperature

2.2.2 Darkwood

Darkwood is a survival horror game developed by *Acid Wizard Studio*. It was first released in 2014 as an early access with a later full release in 2017.

The features inspired by this game are:

- The overall graphical design
- Field of view/Fog of war system

2.2.3 Neo Scavenger

Neo Scavenger is a turn-based, RPG survival game set in a post-apocalyptic wasteland. It was developed in 2014 by *Blue Bottle Games*.

The features inspired by this game are:

- Origin story of the player character
- Visual aspect of the healing system

2.2.4 This War Of Mine

This War Of Mine is a survival game set in a war torn country. It was developed by *11 Bit Studios* in 2014.

The features inspired by this game are:

- Dark, morally grey encounters during the game

2.2.5 Escape From Tarkov

Escape From Tarkov is a tactical FPS game developed by *Battlestate Games*. It was released in 2017 and remains in beta to this day.

The features inspired by this game are:

- Complexity of ranged weapons
- Combination of grid/weight inventory system

Chapter 3

Unity Game Engine

3.1 Game Engines

A game engine is an environment which provides a layer of **abstraction** for game developers when developing video games. These environments provide a suite of visual development tools and reusable software components to shield developers from the complicated tasks that need to run in the background for the game itself to work. Tasks which these components simplify are, for example, reading the player's **Input**, rendering **Graphics** onto the player's screen, simulating **Physics** of objects, providing tools for simplified **AI** creation, implementation of **Sounds** and support for **Networking** services for online multiplayer [18].

Back in the early stages of gaming industry, most studios developed their own, proprietary game engines, specifically designed for one game[1]. However creating a game engine is quite a difficult task and depending on how complicated the game engine is and what libraries and tools are used in the development, it can take anywhere from a couple of days to a couple of **years**[9]. This endeavour is, of course, quite costly. That's why many studios instead choose to work with commercially developed game engines. The best-known commercially available game engines include **Unreal Engine**, **Unity** and **CryENGINE**[17].

3.2 Unity

While there were many commercially available game engines to choose from, I chose Unity for two main reasons. The first reason was **prior experience** with the game engine and the second reason is that it is widely considered as one of the most **beginner-friendly** game engines. In this section I will explain the basic concepts of Unity needed to understand the rest of the thesis.

3.2.1 Unity IDE

While Unity is quite a complex game engine and offers a multitude of tools in its IDE, I will mention only those which were used in the development and will therefore help to make explanations in chapters 5-9 more clear.

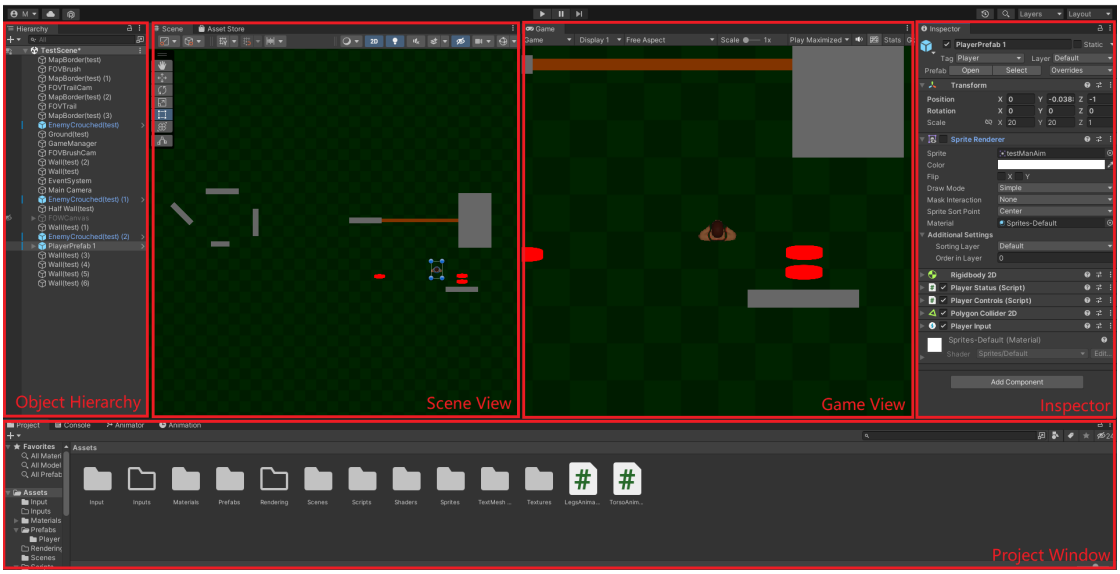

Figure 3.1: Example of Unity IDE configuration from early stages of implementation of BTMC

Hierarchy window

In the Hierarchy window, we can see the currently opened **Scene** and **Game Objects** in the Scene. In this hierarchy window, we can create **hierarchy trees**. This means assigning **child** Game Objects to **parent** Game Objects within the scene.

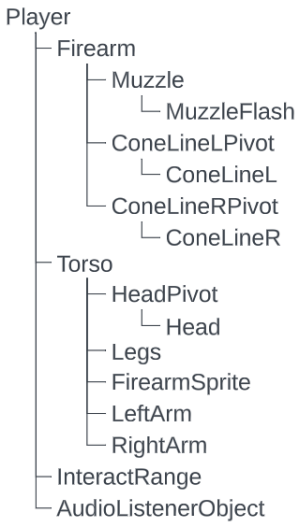

Figure 3.2: Example of the object hierarchy for the Player Game Object in BTMC

The Hierarchy window therefore serves a very important purpose - it is a structured **navigation** for finding Game Objects without having to manually find them in the **Scene View** and it allows the developer to group multiple Game Objects into one.

Scene View

In the scene view, the scene can be seen from a perspective which makes it easy to edit. The developer can navigate the scene and interact with the objects „by hand“.

Game View

This is the view the player will see when playing the game. It renders whatever the **Main Camera** sees. In the IDE there is an option to change the editor to **Play Mode**. In the Play Mode the IDE allows the developer to play the game as if it were built, without actually having to build the game and run it. This is of course a huge advantage which saves a considerable amount of time, due to the fact that building the game can be a lengthy process.

Inspector

In the Inspector window, an individual **Game Object** can be edited. Each Game Object has a set of **Components** attached to it the inspector makes it very easy to add, edit and remove individual Components of Game Objects.

Project window

In the Project window, the user has access to all of the **assets** for the game. These are the individual files such as scripts, sprites, materials, textures, sounds and so on. It is designed as a regular **file explorer** with folders and files, common for many operating systems.

Animation

Animation window lets the user create animations. There are many different ways to animate various objects in video games from moving, rotating and scaling objects to swapping sprites and so on. By using **animation events** the user is able to trigger methods from scripts at specific points in the animation. The animation window also contains a **Curves** section which allows the user to modify the process of changing individual values in the animation.

Animator

The animator window is used to specify when different animations are meant to be played. It works as a **finite state machine**, where individual animations represent the states. The user can then specify transitions between these states which are triggered either automatically when the previous animation reaches an end or after a trigger or a variable is changed through code.

3.2.2 Game Objects

The main building blocks of each game created in unity are Game Objects. Without them, no game could be created. For example, Game Objects are all of the „physical“ objects the player sees. Things such as buildings, NPCs, cars, ground and so on. But these are not the only Game Objects. All of the UI elements are also Game Objects - HUD, inventory screens, main menu and so on. There are even Game Objects that the player does not even see but are sometimes necessary for background scripts or are used to group other Game Objects. Lastly, without Game Objects, the player couldn't really see anything because even the Main Camera is a Game Object.

Each Game Object has a **Name** and can be assigned a **Tag** and a **Layer**. Tags and Layers can in some cases be used interchangeably. However there are some key differences. There is only a **limited amount** of layers - 32 out of which only 27 can be defined by the user, while there is a virtually unlimited amount of tags that can be created. Layers can also be used to specify which objects should be **visible** to Cameras or hit by **Raycasts**.

3.2.3 Components

The Importance of Game Objects has been explained in the previous subsection. However Game Objects are mostly useless without Components. Each Game Object has to have at least one Component and that is the **Transform** Component. This component determines the objects **position**, **rotation** and **scale**.

There are however many other Components a Game Object may have. Here are just a few examples that I deem the most important.

Renderer

There are many different kinds of Renderers but all of them serve the same main purpose. This component makes it possible to **see** the game object. Without some kind of Renderer Component the Game Object cannot be rendered onto the screen and therefore will be invisible.

In 2D games it is common to use the **Sprite Renderer** for most visible Game Objects. A Sprite Renderer uses a sprite from a file as the visual representation of the object. The Sprite Renderer offers a few options to adjust the look of the sprite.

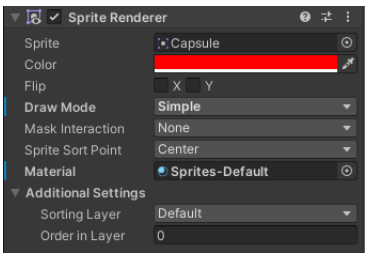

Figure 3.3: Example of a Sprite Renderer Component

Collider

Just like the Renderer Components there are multiple different types of Colliders (Box, Capsule, Polygon etc.). These Colliders can often be separated to two main categories - **2D** and **3D** colliders. It should come as no surprise that for BTMC, only 2D colliders are used.

Colliders are needed to register **collisions** with other objects. While the Collider itself does not handle the physics of collision, it is there to **detect** the collisions.

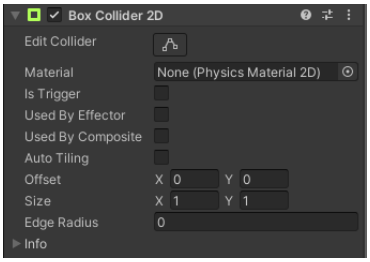

Figure 3.4: Example of a Box Collider 2D Component

Rigidbody

There are only two variants of a Rigidbody Component - **2D** and **3D** Rigidbody. Rigidbody is a component which gives the Game Object **physical properties** such as **Mass**, **Gravity Scale**, **Angular and Linear Drag** and so on. Rigidbody also allows to apply **forces** onto the Game Object. When combined with a Collider the object can collide and physically **interact** with other Game Objects.

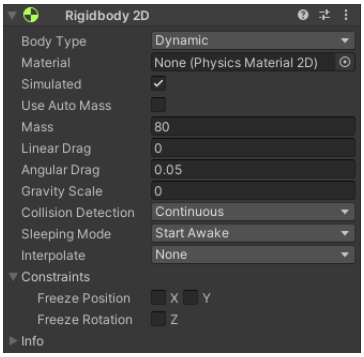

Figure 3.5: Example of Rigidbody 2D Component

Camera

The Camera component allows the Game Object to **render** certain objects from the scene. The component has a multitude of options to edit what the camera sees, how it sees those items, where it renders them and so on.

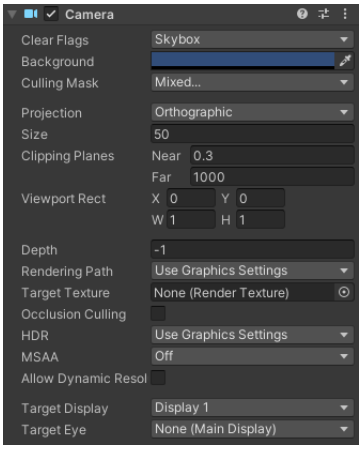

Figure 3.6: Example of Camera Component

Scripts

Besides choosing one of the default components offered by Unity, a **manually written** script in **C#** can be attached to the Game Object. It is common practice for each script to contain the definition of a specific **class**. These classes often inherit from the **MonoBehvaiour** class and redefine its behaviour. The most commonly redefined methods are:

- **Start()** - Code in this method is executed on the first frame when the script is activated.
- **Awake()** - Code in this method is executed when the Game Object to which this script is attached is activated, regardless of whether the script itself is active.
- **Update()** - Code in this method is executed each frame, if the script is active.
- **FixedUpdate()** - Code in this method is executed **independent** of the frame-rate. Instead it is called in sync with the **physics system**, which of course makes this method ideal for simulating physics correctly.

Besides the inherited methods the developer may of course create their own methods.

3.2.4 Raycasts

Raycasts are part of Unity's physics system and they work pretty much like a **laser beam**. They have a **point** from which they **originate** and a **direction** at which they're **aimed**. Raycasts then take note of each Collider they pass through or stop at the first hit collider. A **Layer Mask** may also be assigned to a Raycast which means that the Raycast will only detect collisions with Colliders that are attached to Game Objects that belong to specific Layers. I have decided to mention them here as they play a huge role in BTMC. They are used for a Field of View/Fog of War system, to determine bullet trajectories, detection of the player by the hostile NPCs and so on.

Chapter 4

Game Design of BTMC

The design of BTMC began with a **Game Design Document (GDD)**. GDD broadly outlines what the game is supposed to be. Since this document is created **before** the development of the game even starts, it is usually quite **inaccurate** and the document goes through **numerous changes** and iterations. In this chapter I will be referencing the GDD for BTMC.

I would also like to mention that this design chapter outlines what the game is supposed to look like in its final stage. At the time of writing of this thesis the game is far from a finished product and therefore a lot of the mechanics in this section are not yet implemented. See chapter 11 for the summary of the final product.

4.1 Genre

The genre of the game was one of the first aspects that needed to be chosen. In the GDD the game was described as a **post-apocalyptic, single-player, hardcore survival game**. This categorization was mostly preserved. The game was however further categorized as an **RPG** due to the character's **skills** and **abilities** and a **Third-person shooter**, since firearms will likely play a great role in the gameplay for most players.

4.2 Game Engine Choice

Unity was the chosen game engine from the beginning of design. As mentioned before, Unity is considered to be one of the most **beginner-friendly** game engines, while providing plenty of tools to create a game of this scope. Unity was also chosen due to **prior experience**.

4.3 In-Game World

While „post-apocalyptic“ is a good description in terms of genre it is quite a broad term. The specifics were already outlined in the GDD before implementation.

4.3.1 Setting

The game is set in a future in which most of humanity has been wiped out by an **atomic war**. Most of the people who survived the initial explosions died in the coming years due to health complications from exposure to radiation, missing infrastructure, food shortages caused by a nuclear winter and anarchy that ensued.

4.3.2 Story

The main character (player) was a subject in the research of **cryogenics**, frozen in a cryogenics pod long before the war even started. Stored in an underground research facility, they were able to survive the blasts and the experiment was a success! The pod automatically defrosts the main protagonist and opens roughly **5 years after the war**. Disoriented and confused, they step out of the pod to a new, harsh world.

4.3.3 World Structure

The map of the game was designed to be composed of 4 areas:

- **Forest** - Hostile area, patrolled by bandits composed mostly of trees, roads and occasional houses
- **Derevstok** - A small, walled-off village in the Forest. A safe area with traders and quest givers.
- **Kumensk** - Hostile area, patrolled by the remnants of military. Mostly consisting of larger buildings and city streets.
- **Market** - A secure town square in Kumensk with traders and quest givers.

4.4 RPG Elements

While the game was not originally labeled as an RPG, the GDD already considered character skills as part of the game. The originally designed skills were:

- **General** - Strength, Fitness, Agility, Awareness
- **Offensive** - Ranged, Melee, Trapping
- **Miscellaneous** - Lockpicking, Looting, Medical

4.5 Survival Elements

4.5.1 Character Needs

In the GDD the character was designed to have these needs:

- **Health**
  - Lowered by injuries.
  - Recovers over time if the player is well fed and hydrated.
  - If health drops to zero, the character dies.
- **Stamina**
  - Lowered by melee attacks and sprinting.
  - Replenished quickly over time.
  - After dropping to zero the character cannot sprint and their attacks are weakened. It has to fully replenish before these debuffs are removed.
- **Hunger**
  - Lowers slowly over time.
  - Replenished by eating food.
  - If hunger falls below 50% the character does not replenish health.
  - If hunger reaches zero, the character becomes malnourished and starts slowly losing health.
- **Thirst**
  - Lowers slowly over time.
  - Replenished by drinking fluids.
  - If thirst falls below 50% the character does not replenish health.
  - if thirst reaches zero, the character becomes dehydrated and starts slowly losing health.
- **Body Temperature**
  - Regulated by standing close to a heat source or by removing clothing.
  - If the character is too cold, they will enter a state of hypothermia and start slowly losing health. Their hunger depletion will increase due to increased calorie requirements.
  - If the character is too hot, they will enter a state of hyperthermia and start slowly losing health. Their thirst depletion will increase due to excessive sweating.
- **Tiredness**
  - Lowers slowly over time.
  - Replenished by sleeping.
  - Character becomes tired if the value drops below 20%. This applies various debuffs.
  - If the value drops to zero, the character faints and sleeps on the ground for a short period of time.

4.5.2 Health System

The Health System in BTMC was designed with a little bit of *complexity*. While the health is the main indicator of the state of the character, it is not just a simple number reduction on each hit and an addition when using medicine.

The Health Status of the character is composed of **6 body parts**. All of the **limbs**, **torso** and the **head**. Each of these parts has different injuries that have to be treated using different medical items:

- **General injuries** (possible to appear on any body part)
  - **Gunshot** - Caused by ranged weapons
  - **Cut** - Caused by sharp melee weapons
  - **Bruise** - Caused by blunt melee weapons
- **Limb injuries** (arms and legs)
  - **Fracture** - Caused by blunt melee weapons
- **Other injuries**
  - **Bleeding** - Damages health over time, caused by gunshots and cuts. Can be either heavy or light.
  - **Infection** - Damages health over time and causes severe pain. Caused by improperly treated gunshots and cuts.
  - **Pain** - Applies a broad range of debuffs (lower accuracy, speed, carry weight, etc.) based on severity of the pain. Caused by all kinds of injuries.
  - **Radiation Poisoning** - Causes pain and damage to health over time. Caused by movement in irradiated areas. Effect lingers for a short period of time even after leaving irradiated area.
  - **Food Poisoning** - Causes pain. Caused by eating improper food.

In the GDD these different health items were designed:

- **Bandage** - stops light bleeds and heals them over time, fast use
- **Tourniquet** - stops heavy bleeds but does not heal them, fast use
- **Needle and thread** - stops heavy bleeds and heals them over time, slow use
- **Disinfectant** - prevents infection, fast use
- **Painkillers** - combats pain effects, fast use
- **Splint** - heals fracture over time, slow use
- **Antibiotics** - combats ongoing infection, fast use
- **Blood transfusion** - fills health bar, slow use

4.5.3 Shelter

A kind of a **shelter system** was designed in the GDD. Since the character gets tired over time, they will need to **sleep**. The **quality** of sleep will depend on where they sleep (the ground, a bedroll, a bed, etc.). Since night temperatures in the game will fall well below the freezing point, the player will have to find suitable shelter with a **heat source** to sleep. Sleeping also makes the character vulnerable and various **noise traps** or **lethal traps** can be set up to wake the character up or protect them.

4.6 Combat System

Combat in BTMC was designed to be split into two categories - **melee** and **ranged**.

4.6.1 Melee Combat

Melee weapons were designed to have these 4 attributes:

- **Weight** - Affects attack speed and stamina consumption.
- **Damage** - Affects how damaging the weapon is.
- **Reach** - How far can the character swing the weapon.
- **Type** - Blunt or sharp. Determines types of injuries that can be caused by the weapon.

4.6.2 Ranged Combat

The ranged combat in BTMC was designed to combine the **player's skill**, the **character's ability** to use a ranged weapon and a bit of **random chance**. The accuracy of ranged weapons is designed as a **cone** which represents where the bullet might fly. Since there is no way to aim up and down in a top-down, 2D game, the body part that was hit when the bullet connects is determined *randomly*. It is not just pure randomness though. The abilities of the character play a role in determining what part of the enemy was hit. Characters with **better ranged abilities** have a **higher chance** of hitting critical body parts.

Each weapon in BTMC uses a different **ammunition type**. Furthermore some weapons require **magazines** to function and these magazines have to be **filled before use** in the inventory.

4.6.3 Stealth

In BTMC it is not always wise to charge head on into combat. Sometimes it's better to stay low and quiet. Moving in stealth also provides nice bonuses to attacks on unsuspecting enemies. Melee attacks from stealth cause **instant kills** and stealth attacks with ranged weapons are much more accurate and have a higher chance to hit critical body parts.

4.7 Non-Playable Characters

NPCs in BTMC were designed to be split into 4 categories:

- **Traders**
  - Static, bound to one place
  - Friendly
  - Trade items for in-game currency
  - Reward player for completing quests
- **Civilians**
  - Some are static and some are mobile
  - Friendly
  - Some may provide quests for the player
- **Bandits**
  - Mobile
  - Form groups
  - Hostile
  - Usually equipped with melee weapons with occasional low-end ranged weapon
- **Soldiers**
  - Mobile
  - Form groups
  - Hostile, more dangerous than bandits
  - Equipped with military-grade gear

4.8 User Interface

4.8.1 HUD

The HUD in BTMC was designed to show the following information:

- Status Bars for character's needs
- Injuries
- Buffs/debuffs
- Equipped weapon
- Stance of the character
- Added information from equipped gear such as:
  - Time - if the character is wearing a wrist watch
  - Radiation readings - if the character has a Geiger Counter equipped

4.8.2 Inventory Screen

The Inventory Screen was designed as a menu that contains 4 tabs:

- **Inventory**
- **Health**
- **Map**
- **Tasks**

4.8.3 Inventory Tab

The inventory tab was designed to be split into 3 parts:

- **Equipped**
- **Inventory**
- **Ground**

The „Equipped“ part of the Inventory Tab will showcase a character outline with items that are currently equipped on the character such as clothes, backpacks, weapons and so on.

The „Inventory“ part of the Inventory Tab will be split into multiple grids depending on what kinds of clothes the character has equipped. Some clothing items will apply a **weight reduction** to items carried within them. For example, a hiking backpack is much more comfortable to wear than a cheap satchel, so items carried inside the backpack will not weigh down the character as much.

The „Ground“ part of the Inventory Tab will show items on the ground in the vicinity of the character.

4.8.4 Health Tab

The Health Tab is designed to look very similar to inventory tab. The „Equipped“ part of the screen will instead show **current injuries** of the character and the „Inventory“ part of the screen will show only the available **medical items**. The „Ground“ part of the screen works just like it did before.

4.8.5 Map Tab

The Map Tab will show the player the map of the game, provided the character has a map in their inventory. Otherwise the tab will not show anything.

4.8.6 Tasks Tab

The Task Tab will show the progress of active quests, given to the player by traders and civilians.

Chapter 5

Implementation Basics

Each section in this chapter conceptually describes implementation of an individual **concept** or a **mechanic** in BTMC. In some of these sections, there will be mentioned a video tutorial, on which the implementation is based on. I would like to note that while these tutorials served as the **base** for said mechanics and as such are mentioned as sources, they were *changed*, *adapted* and *expanded* significantly during implementation.

5.1 Project Setup

Before implementing the project I first had to set up my work environment. I installed the **Unity Hub** application and created a new project. I have downloaded the newest Unity Editor release at the time (**2021.3.9.f1**) and created a new project using the default **2D Template**. A new project starts with a **Sample Scene** and a **Main Camera** Game Object.

5.1.1 Game Objects in 2D Space

In Unity Game Engine there is no true 2D workspace. 2D games are created in a 3D space and only appear to be 2D. In BTMC this is achieved by pointing the camera **perpendicular to the XY-plane** (parallel to the Z axis). Therefore the player character and all of the visible Game Objects in BTMC are 2D sprites aligned **parallel to the XY plane**. Since the objects are just 2D sprites, they have no depth (they are infinitely thin in the Z axis).

This means that if two Game Objects should appear below (or above) one another, their Z axis position has to be set to different values. Another thing that needs to be ensured is that if two visible objects can appear in the same 2D space (they do not collide), they must have different Z axis values to avoid a common visual bug called *Z-fighting*.

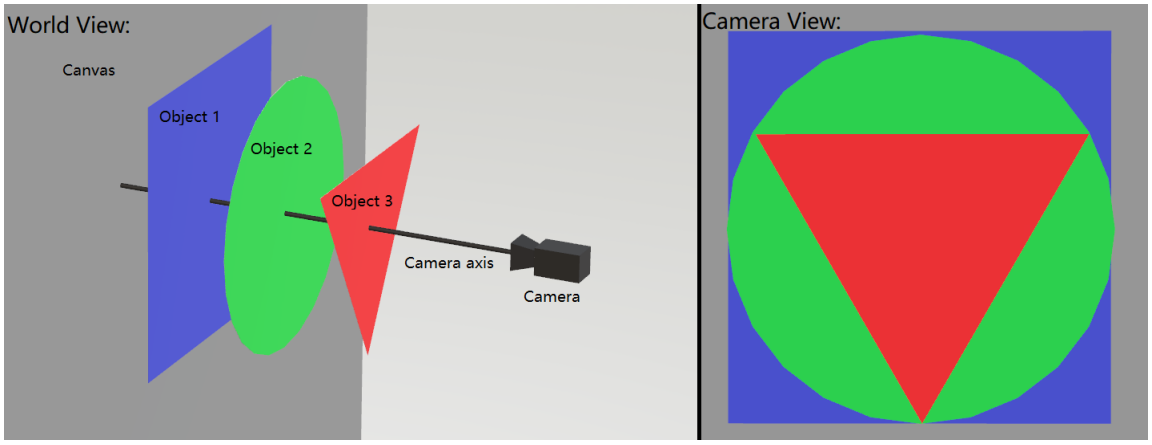

Figure 5.1: Using a camera to display 3D space as 2D space

5.2 Camera System

5.2.1 Camera Movement

The **Main Camera** renders objects placed in front of it onto the screen of the player. It can be considered one of the most important objects in any game created in Unity.

In BTMC the Main Camera is configured to always **look at the player character** and keep the player in the **center** of the screen. There are exceptions to this however.

First of all, the camera does not follow the player perfectly. For a better visual experience, the camera is set to follow the player **smoothly**. This is achieved by delaying the camera movement using the **Lerp** method. This method linearly interpolates between two points. To move the camera closer to the player character each frame, the current position of the camera and the current position of player character are supplied as the two points. Then, the time difference between the current and previous frame (stored in Time.deltaTime) is used to gradually move the camera.

The camera also does not keep the player character in the center of the screen when the player is **„looking around“** using the right mouse button (see 6.3.2). This function lets the player see farther in a particular direction than what is normally visible to them. In this case the camera will calculate a **point** in the world **between** the position where the player is aiming and the position of the player character. The camera will then position itself to look at this point.

5.3 Objects

In this section I will refer to objects as being „below“ or „above“ each other. However, as mentioned in subsection 5.1.1, what this truly means is that their Z axis position values are different.

5.3.1 Visible Game Objects

I have split the visible Game Object in BTMC into two categories. **World Objects** and **UI Objects**. World Objects are objects that **represent real-life objects** in the game, such as characters, walls, containers and so on. UI Objects are objects that the player sees, however they **do not truly exist in the game world** such as UI and HUD elements.

Visible World Objects:

- **Ground** - Only one Game Object is assigned to this category and that is the Ground. The Ground is a large plane that spans the entire playable area and is always below all other Game Objects.
- **Ground Objects** - Ground objects such as rugs, trash on the ground and so on are objects which serve a decorative purpose and can be walked on by the player character.
- **Player** - The player is composed of 4 visible Game Objects. The Legs, the Torso, the Head and the FirearmSprite. This is done due to the way Animation (see 9.1.1) and Hitboxes (see 7.2.5) are implemented in BTMC.
- **Non-playable characters** - NPCs are composed in the same way as the Player in terms of visible Game Objects.
- **Full Obstacle** - Game Objects that the in-game characters should **not** be able to see through, pass through or shoot through, such as walls and tree trunks.
- **Half Obstacle** - Game Objects that the characters can see through (if currently standing), shoot through, however cannot pass through, such as half walls, barrels, sandbags and so on. These are the objects that can be used as cover during ranged combat (see 7.2.4).
- **Low Obstacle** - Game Objects that the characters can **always** see through and shoot through, however they cannot pass through such as chairs, tables, chain link fences and so on.
- **Interactables** - Objects that the player can **interact** with such as containers, doors or beds.
- **Fog of War** - FOW is a large image on a World Canvas which represents areas that the player does not see (see 6.3).

Most of the visible UI objects belong to one layer called **„UI“**, which is one of the default Unity layers.

Visible UI Objects:

- **Menu Screen** - Main Menu screen, Pause Menu screen and Options screen which mostly consist of buttons and other interactable UI elements (see 8.1)
- **HUD Objects** - Status bars, HUD devices, player stance, equipped weapon information, rest menu and rest screen (see 8.2).
- **Inventory Screen** - Screen where the player manages their items, equipment and health (see 8.3)

UI Objects are always displayed **above** all World Objects.

5.3.2 Invisible Game Objects

Sometimes Game Objects are needed, which are **invisible** to the player. These Game Objects exist simply to **hold certain scripts**, serve as a **grouping object** for other Game Objects or have some other **special function**. There are many of these invisible objects in BTMC and will be mentioned and explained later in this chapter, in the sections which they are relevant to.

Chapter 6

Player Implementation

6.1 Player Controls

6.1.1 Reading Input

Input System

At the time of writing of this thesis, there are two most common ways to read the player's input in Unity. The older way of getting the input directly from an input device or through the old **Input Manager** and a newer way of getting the input indirectly through an **Input Action** using an **Input System package**.

While the new Input System is generally harder for new users to grasp it brings multiple advantages. I personally chose it for two reasons. Namely it **saves computer resources** (provided it's used correctly) and it offers a relatively easy way to change **Action Maps**. There there are multiple ways to read player's input via the new Input System Package. I have chosen the often encouraged way of **sending messages** to scripts when a player input is detected.

Sending Input Messages

To send input messages, I first had to add a **Player Input** component to the player Game Object. I have also created a script named `PlayerControls` and attached it to the player. It is important that the Player Input component and the script are attached to the same Game Object, otherwise the Player Input component will not be able to send the messages to the script. A considerable advantage of this message system is that the Player Input component will automatically find and send a message to any script attached to the same GameObject that contains a method named **On**[**Input Action Name**].

Next, I created a new **Input Actions File** in the game's Assets folder. All of the Action Maps and Input Actions will be stored in this file. I have also attached this file to the Player Input component mentioned earlier. The first Action Map I have created was the **Player** Action Map. In this action map are stored all of the controls associated with controlling the player character.

The second Action Map in the game is the **UI** Action Map. This action map is used whenever the player is interacting with the UI of the game, such as the Inventory Screen or the Pause Menu.

Character Movement

Now, for the sake of simplifying the process I will focus only on the *character movement*. This was however done for all of the other controls.

In the Player Action Map I have added a new Input Action named **Move**. Since the player will move the character using the [W] [S] [A] [D] control scheme I decided to read the Move action as a **Digital Normalized 2D Vector** and I have added a new **Up/-Down/Left/Right Composite** to the Move action. To this composite I have assigned the corresponding buttons. With this setup the Input System will read any combination of said buttons as a 2D vector (buttons opposite each other cancel out).

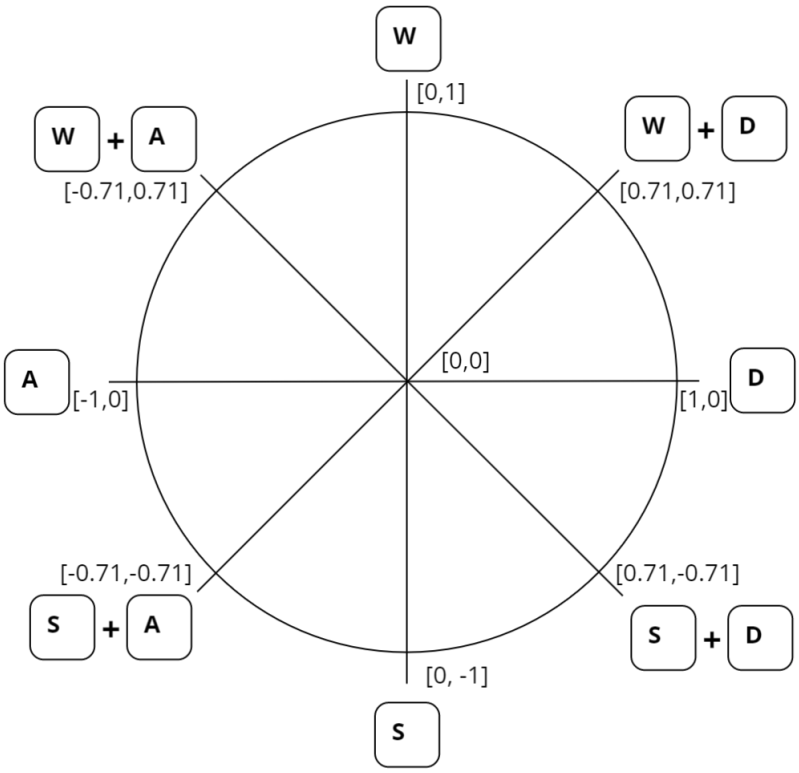

Figure 6.1: Unit-circle showing combinations of pressed movement keys and corresponding normalized vectors.

The next thing I had to do was to create a new method in the Player Controls script named `OnMove`. Now every time the player presses any of the movement buttons the `OnMove` method is called and the 2D vector is passed to it as an argument. In this case the method applies a force to the Rigidbody of the player Game Object in accordance with the direction of the vector.

6.2 Player Status

The status of the player is determined by many variables. These variables are stored, updated and managed by a class named `PlayerStatus`. This class also manages local and global **status effects**, current status of body parts and the like.

6.2.1 Basic Status Variables

Basic status variables are:

- **Stance and movement variables** - Whether the player is sprinting, running, crouched, resting and so on
- **Speed variables** - Sprint speed, walk speed and crouch speed
- **Shooting Ability** - Ability to use firearms (see 7.2.2)

These variables mostly serve as **flags** or are used in calculations by **other scripts**. The Shooting Ability and Speed variables are changed with injuries and pain (see 6.2.3).

When the player character is sprinting they lose Hunger, Thirst and Tiredness points at a **faster** rate.

6.2.2 Survival Status Variables

There are **6** survival status variables the player needs to manage to ensure the survival of the player character:

- **Health**
- **Stamina**
- **Body temperature (currently not implemented)**
- **Hunger**
- **Thirst**
- **Tiredness (fatigue)**

Health

Undoubtedly the **most important** survival status variable. When the player character's health drops to **0**, the player character **dies**. There are a few ways to lose health such as not managing the hunger or thirst of the player character or from injuries sustained in fights with NPCs.

There is however **only one** way to regenerate health. The player character's health is regenerated if they currently have the **Nourished** status effect.

Health is capped at 100 but can be reduced by **injuries**.

Stamina

Stamina is depleted by sprinting and regenerated when not sprinting. When the player runs out of stamina, they will be **unable** to sprint. They regain the ability to sprint after stamina has regenerated above **20%**.

Stamina is capped at 100 but can be reduced by **infection**.

Body Temperature

While the body temperature mechanic is quite easy to implement, currently the game simply lacks enough clothing items to make it viable.

Hunger

Hunger is periodically drained and is replenished by consuming **food**. When hunger drops to zero, the player character gains the **Starving** status effect.

Thirst

Much like hunger, the thirst is also periodically drained and replenished by drinking water. When thirst drops to zero, the player character gains the **Dehydrated** status effect.

Tiredness

Tiredness or fatigue is also periodically drained and is replenished by **resting**. When the character's tiredness drops below **20%** they gain the **Tired** status effect. When tiredness drops to 0 the character will **faint** until tiredness regenerates back to **20%**.

6.2.3 Health System

The health system in BTMC works based on **status effects**. Status effects can be split into **local** and **global**.

Local status effects

Local status effects are bound to a **specific body part**. There are 6 distinct body parts in the game:

- **Head**
- **Torso**
- **Left Arm**
- **Right Arm**
- **Left Leg**
- **Right Leg**

All of these body parts can be affected by either one, or a combination of these local status effects:

- **Open Wound**
- **Bleeding**
- **Clean Bandage**
- **Dirty Bandage**
- **Disinfected**
- **Infection**
- **Stitched Wound**
- **Pain**

Open Wound effect is gained after the character is **shot**. Each open wound lowers the character's maximum health by **10** points.

Bleeding effect is first gained when the character is shot and regained every time, the player removes a bandage from an Open Wound. It causes the player to **periodically lose health** for each bleeding wound.

Clean Bandage effect is gained after the player uses a **Clean Bandage** to bandage their wound. Clean bandages stop bleeding and help **prevent** infection from **infection rolls**. They also help disinfected body parts stay disinfected for much longer.

Dirty Bandage effect is gained after the player uses a Dirty Bandage to bandage their wound or after a Clean Bandage **loses its durability** and turns into a Dirty Bandage. Dirty Bandages also stop bleeding but they **do not** prevent infection. They also help the body part stay disinfected but for a shorter period of time, compared to Clean Bandages.

Disinfected effect is gained after the player uses an **Antiseptic** on an Open Wound or a Stitched Wound. Disinfected body parts have **no chance** of getting infected from **infection rolls**.

Infection is gained after a body part gets infected. Gaining infection is based on randomized **infection rolls** that happen every few in-game minutes. Every Open Wound or a Stitched Wound has a chance to get infected. The chance starts at **0%**. If a wound **does not** get infected on a roll, **2%** is added to the chance. This chance only accumulates for Open Wounds but the rolls still happen for Stitched Wounds. If the wound is bandaged with a Clean Bandage, the chance only accumulates by **0.5%**. When an Antiseptic is used, all accumulated chance is **reset** to 0% and as long as the body part is disinfected, it does not accumulate infection chance. Once a body part gets infected it accumulates **infection** and lowers the maximum stamina by **10** points. To heal this infection, the player has to take **Antibiotics** which lower the infection over time.

Stitched Wound is gained by using a **Suture Needle** on an Open Wound. This stops bleeding and the accumulation of infection chance, but as mentioned before, the wound **can** still get infected. Stitched wounds lower the maximum health by 5 points and heal over time, at which point the body part is **completely** healed.

Pain is caused by Open Wounds and can only be prevented by taking **Painkillers**. Each body part that has the Pain effect **lowers** the shooting ability.

Global Status Effects

Global status effects are applied to the **entire** body, not just a specific body part. Global Status Effects:

- **Nourished**
- **Starving**
- **Dehydrated**
- **Over-encumbered**
- **Tired**
- **Painkillers**
- **Antibiotics**

Nourished effect is gained by keeping the character's thirst and hunger above **90%**. This status effect is the only way the player character can regenerate lost health. Removed when either of these values drops below the threshold.

Starving effect is gained when the character's hunger drops to zero. Causes the player to lose health over time and applies the **Pain** effect to the torso. Removed by eating.

Dehydrated effect is gained when the character's thirst drops to zero. Causes the player to lose health over time and applies the **Pain** effect to the head. Removed by drinking.

Over-encumbered effect is gained when the character carries too much weight. Disables sprinting. Removed by removing items from the inventory.

Tired effect is gained when the character's tiredness drops below **20%**. Lowers stamina regeneration and slightly decreases shooting ability. Removed by resting.

Painkillers effect is gained by using **Painkillers**. Removes Pain from body parts as long as it is active. Removed over time.

Antibiotics effect is gained by using **Antibiotics**. Slowly removes infection from body parts as long as it is active. Removed over time. Wounds that have been infected for too long may require **multiple** antibiotics.

6.3 Field of View and Fog of War

6.3.1 Player vision and character vision

In most first-person games and some third-person games, what the player sees and what the character sees are the same thing. In BTMC however these are two different things. As mentioned in subsection 5.2.1, what the player sees in BTMC is a **top-down view of the area** around the player character. What the character sees determines what **will be shown** to the player.

Therefore in this section when I refer to the **Field of View** (FOV), I am referring to the FOV of the player character.

Since BTMC is supposed to be a hardcore survival game, it would not make much sense to have the player see enemies that are outside of the FOV of the player character or areas they have not yet explored. That's why I have implemented a **Field of View/Fog of War** system.

As evident by the name, this system is works as a combination of 2 separate concepts - Field of View (FOV) and a Fog of War (FOW). The implementation of FOV is based on a tutorial created by a user named *Code Monkey* [7] and FOW is based on a tutorial created by a user named *Santzo84* [16].

This system is implemented using:

- FOV Object,
- FOV Brush Camera and FOV Brush texture
- FOV Trail Camera and FOV Trail texture
- FOW Canvas and FOW Object
- FOW Shader

6.3.2 Field of View

Field of View is a **„cone“** of vision of the player character. When an obstacle is placed into FOV it obstructs the vision and therefore nothing behind this object can be seen. Obstacles that **always** block the vision are called **Full Obstacles** and obstacles that block the vision **only when crouching** are called **Half Obstacles** (see section 5.3.1 for examples).

To achieve this effect I have used **Raycasts**. Each frame, multiple **Rays** are cast out of the player character towards where the character is looking, each at a **slightly different angle**. These Rays „fly“ towards the position they were aimed at, until they reach the maximum distance called the **FOV Distance**. If they collide with any object between their origin and destination, that object will be added to an **array of collided objects**.

If the object they collided with is supposed to block the character's vision they will mark the point where they hit the object as their **end point**.

There is also another set of Rays with much shorter maximum distance which symbolize, what I like to call, a **Field of Perception** (FOP). FOP is the area behind the player character. Therefore it is not what the player character sees but rather what the player character **should be aware of**.

FOV and FOP are two angles, which add-up to **360°** around the player. FOV is always smaller but FOV Rays are much longer than FOP Rays.

When the Player presses the right mouse button, they can **look around**, which lets them see further (FOV Distance is increased) but FOV is much more narrow (FOV is an even smaller angle compared to FOP).

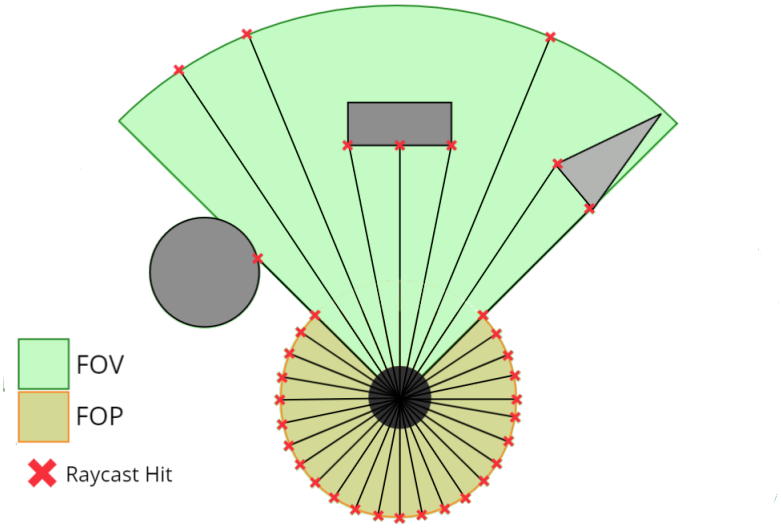

Figure 6.2: Example of FOV and FOP Rays with Raycast Hits.

Each frame there is therefore an **array of points** in the world at which the Rays either collided with an obstacle or reached the maximum FOV (or FOP) Distance. These points are then connected and an outline of a **Mesh** is created. This Mesh symbolizes which areas are currently visible to the player character. This mesh is then rendered and placed onto the **FOV Object**. The FOV Object is rendered in **bright red** color and placed into the **FOV Layer**.

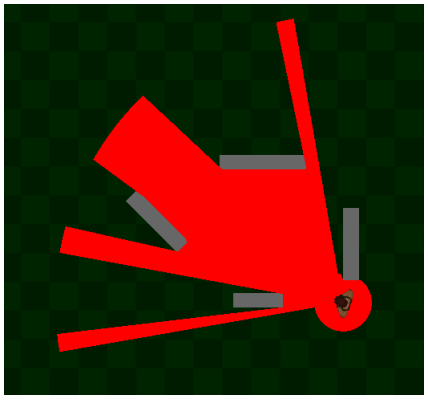

Figure 6.3: Example of FOV Object blocked by Full Obstacles

6.3.3 FOV Cameras

Two cameras are used in this FOV/FOW system. Both of these cameras are positioned to see the **entire playable area** and never move. They also differ from the Main Camera by storing what they see into a **Render Texture** instead of displaying it onto the player's screen.

Both Cameras are set to only render the red **FOV Object**.

The **FOV Brush Camera** stores what it sees into a large texture with a black background called **FOVBrush**. The **FOV Trail Camera** also stores what it sees into a large texture with a black background called **FOVTrail**.

The main difference between these two cameras is that the FOV Brush Camera **overwrites** the texture at the end of each frame with black background while the FOV Trail Cmaera does not. This means that while the FOVBrush texture stores only the **current location** of the FOV, the FOVTrail texture **„remembers“** which areas were already explored.

Figure 6.4: Example of FOVBrush Texture (left) and FOVTrail Texture (right) after some movement.

6.3.4 Fog of War

Now by combining these two textures I was able to create the desired FOV/FOW effect. Inside the **FOW Canvas** is stored a **FOW Game Object** containing a **Raw Image** component covering the entire playable area. The FOVTrail texture is placed into this Raw Image. This would however only cause a large, black and red image to be seen on the screen.

To get the desired effect I have created a **FOW Shader** and a new **FOW Material** which combines these two textures into one. This material is then used on the FOW Object.

6.3.5 FOW Shader

The input of the FOW shader are the two textures created in subsection 6.3.3. This shader will combine the textures into one Material in the following steps.

While the shader works with the textures **on a pixel basis**, I will explain this process by talking about the textures as a whole as I believe it makes the process easier to understand.

First the textures are **blurred**. I have tried two different approaches. The first approach is the usual way of blurring objects, which is often referenced online. The second one is designed by myself due to optimization reasons.

The first approach is to use a simple blurring shader such as a **Box Blur**. The simplest Box Blur, which takes the values of the blurred pixel and the **4 corner pixels** around it and averages them provides only a very weak blur if it is only used once. That's why it is a common practice to blur the texture, downscale it, blur it again, upscale it and finally blur it again. This provides a satisfactory blur and usually is not very resource heavy.

However due to the special circumstances of this FOV/FOW system, this required to use this method on **both** of the cameras, **before** they even stored what they render into textures. I have not figured out how to properly apply this method, so that it is only performed once. Furthermore Unity has a slight performance problem with using **post-processing effects** (such as blurring) on individual cameras. This caused major performance issues down the way and caused the FPS of the game to be **unstable** and **fairly low** considering the simplicity of the project ($\sim$ 60 - 300 FPS at the time of development).

I have therefore devised a way to blur the textures directly **inside the custom FOW shader** without the need for downscaling and upscaling. My method uses a blur which is **much** less optimized, because it averages 33 pixels per pixel instead of 5 pixels. This is done to make the blur stronger without the need for downscaling and upscaling.

This shader can **definitely** be optimized and improved, but I'm not familiar with coding shaders. However in the end result, it caused the FPS to almost **triple** and **stabilize** ($\sim$ 900 FPS at the time of development).

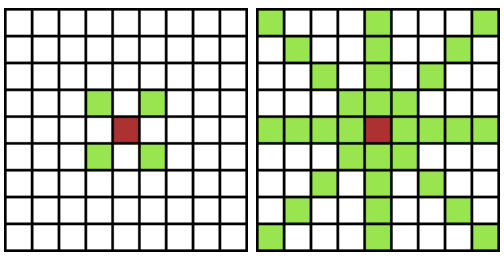

Figure 6.5: Box blur shader (left) and my custom shader (right). The red pixel is the currently calculated pixel and the green pixels are taken into account when averaging.

After the textures are blurred, red pixels on the FOVBrush texture are turned to green pixels. This is done by simply swapping the green and red values of the red pixels.

Then the shader **adds** the colors of both textures together. I will illustrate the possible pixel color combinations using RGBA values: [**RED, GREEN, BLUE, ALPHA**]. Note that the values are capped at 1 and anything larger will be stored as 1.

- Black + Black: [0,0,0,1] + [0,0,0,1] = [0,0,0,1]
- Black + Red: [0,0,0,1] + [1,0,0,1] = [1,0,0,1]
- Red + Green: [1,0,0,1] + [0,1,0,1] = [1,1,0,1]

Here can be seen that areas **undiscovered** by the player will be colored **black**, **discovered** areas will be colored **red** and **currently visible** areas will be colored **yellow**.

Once this combined texture is created, the following formula is applied to create the final material:

$$A = 1 - 0.5R - 0.5G$$

When the final material is created, it is **fully black** and only **alpha value** of each pixel is modified. **Undiscovered** pixels will be fully black **(alpha = 1)**, **discovered** pixels will be semi-transparent **(alpha = 0.5)** and **currently visible** pixels will be fully transparent **(alpha = 0)**.

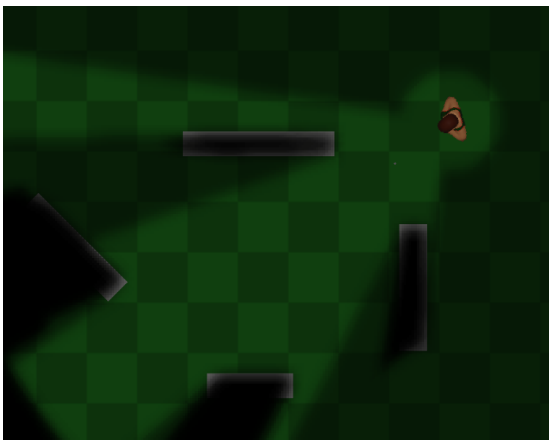

Figure 6.6: Final result of the FOV/FOW system

6.3.6 Disadvantages

While this approach produces the result I was looking for, there are multiple disadvantages.

Computing Raycasts

Raycasts in large numbers can be quite taxing on computer resources. In the default settings of the game, there are 50 rays for FOV and 50 for FOP. The player is able to customize the amount between 50 and 200 rays for FOV. These rays are calculated each frame and for each ray all collisions have to be checked. Decreasing the amount of rays will however cause the FOV edge to have **visible edges** instead of appearing round. This will also cause thin objects to possibly „slip“ between the rays and therefore not obstruct FOV.

Large Textures

Since the FOV textures are stretched across the entire playable area, they cannot be in a too low resolution because the pixels would become too apparent even with the blurring. Planned map size is currently at least 200x200 in-game metres per area and for the FOW to appear correctly I am using textures with 4096x4096 pixels. Usually, each pixel needs 4 bytes to store all 4 RGBA values. This means that each texture would take up **64MB** of space. I have managed to reduce this number by changing the color format of the textures to **R8_UNORM** which only stores the red value of the RGBA format. This means that each pixel now only takes **one byte** and each texture **16MB** of space.

Chapter 7

Implementation of NPCs and Ranged Combat

7.1 Non-playable Characters

Non-playable characters (NPCs) will play a large role in the final game. NPCs will vary in difficulty, hostility and their overall function. At the time of writing of this thesis there is only one type of NPC in the game and that is a simple **„Hostile NPC“**.

7.1.1 Pathfinding

Pathfinding is in my opinion the **most important** aspect of any **non-static** NPC. It is used to determine the best path between the NPCs current position and their destination. Unity provides their own pathfinding tools but those are designed for 3D games. They take into account the size of the character, how well they scale slopes and so on. Since BTMC is a 2D game however, I had to implement my own pathfinding algorithm based on the **A*** algorithm for finding the shortest path between nodes on a graph or a grid .

A* Algorithm Prerequisites

The implementation of said algorithm is based on an online tutorial created by a user named *Code Monkey* [6].

the A* algorithm works by assigning 3 values (costs) to each node:

- **G Cost** - The distance between the node and the **start** node.
- **H Cost** - The estimated distance between the node and the **end** node. The distance is estimated using a suitable **heuristic function**.
- **F Cost** - Sum of the G cost and F cost of the node.

In BTMC each of these nodes is represented by a `PathNode` class. Besides holding information about its costs, it also holds information about whether it is **walkable**, **occupied** and **from which node did the algorithm reach this node**.

These nodes are all part of one instance of a large grid which is represented by a `PathFindingGrid` class. This class contains 200x200 nodes per map area, which means that each node represents a **square metre** of space. I have found this to be a good balance of the amount of nodes and precision of pathfinding. A grid too large will be more taxing

on computer resources when calculating the path, but a smaller one will cause the NPCs to not be able to pass narrow spaces.

The heuristic function used in the calculation of distances must be **admissible**. This means that the expected H cost **cannot overestimate** the actual cost to reach the end node. In BTMC the heuristic function is based on a modified **Euclidean Distance** heuristic. In Euclidean Distance, the path can pass through tiles **vertically**, **horizontally** and **diagonally**. The Euclidean Distance calculates the cost of moving diagonally as a square root of the sum of moving first horizontally squared and then vertically squared (Pythagorean Theorem).

In BTMC, the costs of moving between two tiles are set to **10** for vertical and horizontal movement and **14** for diagonal movement. This saves computer resources by avoiding the use of floats and the square root function at the cost of accuracy (14 is not the exact square root of 200). This inaccuracy is very much negligible.

Lastly, for an A* algorithm to work, two lists are needed. One of them named **Open List**, which holds all nodes that are scheduled for evaluation by the algorithm and a **Closed List** which holds all of the nodes that were already evaluated.

In BTMC, the Closed List is implemented as a **HashSet**. This works because there is always **only one** instance of the node in the Closed List and the only thing that needs to be checked is whether a node **is** or **isn't** in the list. This makes HashSet a perfect candidate for a bit of optimization as HashSets are **faster when searching** for items within them compared to Lists.

Valid Nodes

The algorithm works slightly differently in BTMC than it does in theory as it performs additional checks. Therefore I would like to define a **valid** node. A node is valid when it is both **walkable** and it's **not occupied**.

Walkable nodes are nodes that the NPCs can walk on. These are set when the scene is first loaded. It works by creating a 200x200 2D array of Booleans. Then, all of the **obstacle** objects (see 5.3.1) in the scene are found. Since these are 2D sprites there is always a **rectangle** that can encapsulate them. The positions of the **corners** of said rectangle are determined in World Space and then turned into the Grid Node coordinates. These Grid Node coordinates are **sorted** and **iterated through** from left to right and from top to bottom and any Path Nodes covered by the obstacle are marked as not walkable in the Boolean array. This array is then passed to the `PathFinding` class when its Instance is created and the `PathFindingGrid` is initialized.

Occupied nodes are **reserved** by other NPCs during pathfinding. NPCs reserve the **final** node in their destination after it is found by the A* algorithm. They also reserve the nodes they are **currently standing on**. This is done to prevent NPCs from ending up on the **same** node when trying to get to the same destination and therefore clipping through each other. This is not a perfect system however, as the NPCs will still pass through each other **while walking** if they are walking along the same path.

A* Algorithm

The algorithm in BTMC works in the following steps:

1. Performs validity checks for the end node. If the node is **not valid**, the algorithm tries to find a **suitable neighbouring node**, which can be used instead. If it fails to find a suitable node, the algorithm ends in **failure**.
2. Sets the **G Cost** of **all** nodes to the **maximum** value of int.
3. Sets the **G Cost** of the **start** node to **0** and estimates its **H Cost**.
4. Gets the node with the **lowest F Cost** from the Open List (**Current Node**).
5. If the Current Node is the **end** node, moves to step **14**.
6. Removes the Current Node from the Open List and adds it to the Closed List.
7. Gets the list of **neighbours**.
8. If the neighbour list is empty, moves to step **13**. Otherwise gets one of the neighbour nodes (**Current Neighbour Node**).
9. If the Current Neighbour Node is in the Closed List, moves to step **8**.
10. If the Current Neighbour Node is **not valid**, adds it to the Closed List and moves to step **8**.
11. Calculates the **potential** new G Cost of the Current Neighbour Node as a **sum** of the G Cost of the Current Node and the cost of moving to the Current Neighbour Node.
12. If the potential new G Cost is **lower** than the G Cost stored in the Current Neighbour Node, **updates** the G Cost to the new value and sets the **„came from“ node** as the Current Node. Calculates the H Cost, adds the Current Neighbour Node into the Open List and moves to step **8**.
13. If the Open List is **not empty**, move to step **4**. If it **is empty**, no path can be found between the start and end node and the algorithm ends in **failure**.
14. If this step was reached, it means the path to end node was found and now the path is traced back using the **„came from“ nodes** stored in each node along the path. The result is a **list of nodes which represent the path** and the algorithm ends in **success**.

This algorithm works quite well, however considering the size of the map, if there ever was an **unreachable** node or a node with a particularly **complicated** path to it, the algorithm might try to search a large portion of the **40 000** nodes that tile the map to find a path. This would, of course, cause a **massive** lag, especially if multiple NPCs attempted to find the path at the same time.

There are numerous methods to try and pre-determine whether a node is unreachable, before attempting to find the shortest path using the A* algorithm. These methods would however slow down each pathfinding attempt in which the node **is** reachable, which is the majority of cases. So to make sure the game does not freeze in these special cases and to

not slow down the algorithm any further I have added a **maximum available F Cost**. Since F Cost of nodes accumulates steadily throughout the algorithm, setting a limit means that the algorithm will stop if it takes too long.

At the time of writing of this thesis, the maximum F Cost is set to **1000**. When we consider the size of the map, and the costs of moving from one node to another, this means that if the algorithm did not find a path that is **shorter than 100 in-game metres**, it ends in failure. Even with this limitation, these cases will produce a small stutter when multiple NPCs are attempting to find the path at the same time.

Path Modification

While the A* algorithm finds the shortest path in a grid based environment, it is still **bound** to move in only **8 directions**. To make the path the NPCs take look more **natural** and, in many cases, a bit **shorter**, I have added a **modification** to the final determined path.

When the path is calculated as a set of nodes, there are possibly nodes which are, in a sense, **redundant**. What I mean by that is that when the node coordinates are converted into in-game positions, which are used when actually moving the NPC, there are positions that are **between** two other positions that **do not have any obstacles between them** and therefore do not need to be there.

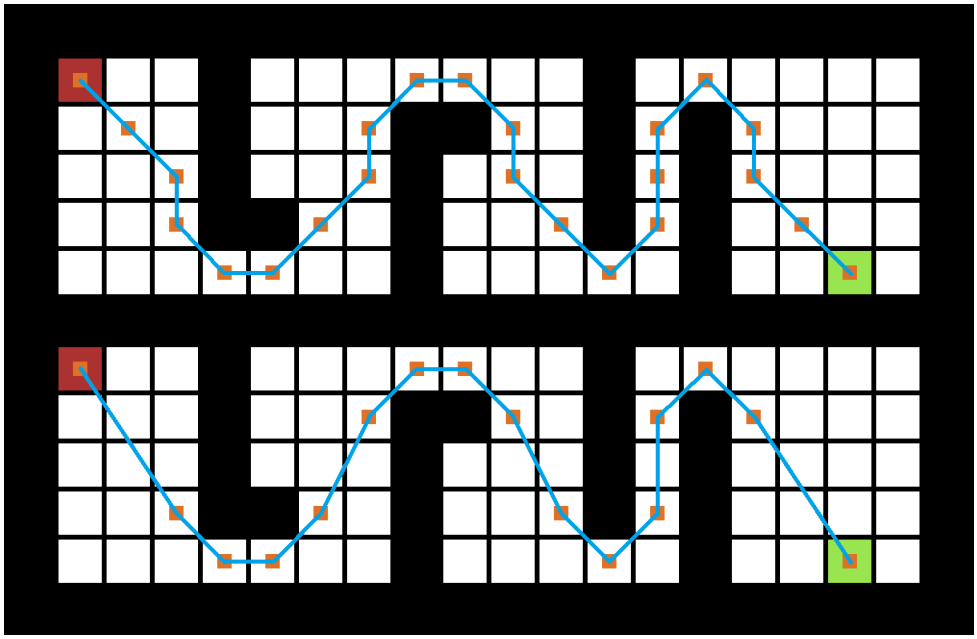

Figure 7.1: Example of a raw A* algorithm path (top) and a modified path (bottom).

While in the figure above, it does not seem like much has changed, in-game it makes a lot of difference. The basic 8 dimensional movement that the raw A* algorithm produces is **very easy to spot** on actual NPCs in-game, as they often **do not** take straight paths to their destination.

This modification has a considerably bigger impact on **larger**, **open** areas. Especially when the NPC is moving at a roughly **22.5°** angle from one of the 8 basic directions.

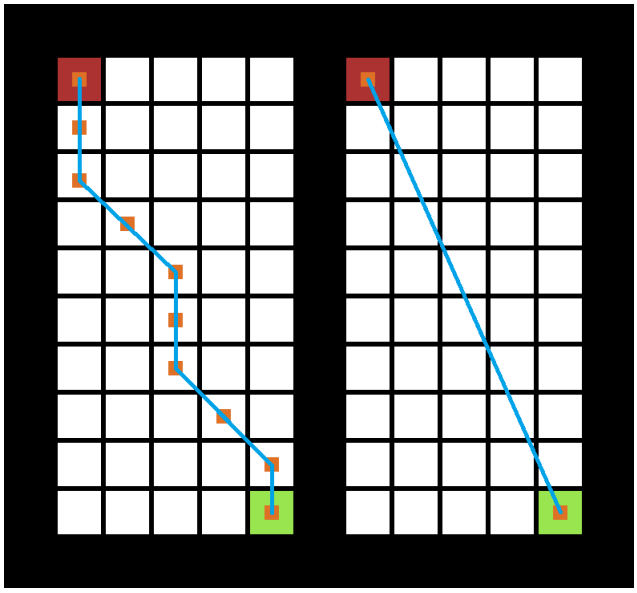

Figure 7.2: Example of a raw A* algorithm path (left) and a modified path (right).

In the figure above, if we consider the tile to be 1 square metre as it is in BTMC, the path traveled by the raw A* would be roughly **10.65** metres, while the modified path is roughly **9.84** metres. Not a huge difference but the real value of this modification is in the fact that the NPC takes a **straight path** to the destination.

In short, this modification algorithm works by creating a start index and an end index. Then it shoots a **Raycast** between the nodes on said indexes. If there **are obstacles** in the way, the algorithm **increments the start index** and tries again. If there **are no obstacles** in the way the algorithm **removes all of the nodes** in between the start index node and the end index node, sets the **end index as the current start index** (as there surely are no more nodes to be removed after the current start index), **resets the start index** and does the process again. The algorithm ends when the start index is **one less** than the end index, which means that no nodes can be removed anymore.

7.1.2 NPC Status

In many ways the NPCs in BTMC are just a **very simplified** version of the player character.

Health

At the time of writing of this thesis, the NPCs in BTMC do not have as complex of a health system as the player character does. They do no need to take care of their basic needs and they do not get injured in the same way as the player does (see 6.2.3).

The NPCs have only their **health** variable and information about **which of their body parts were hit** to simulate the sort of debuffs the player receives (such as their shooting ability or their speed lowered).

Visibility

Since the player should only be able to see NPCs that are currently in the FOV of their character, the NPCs turn their visibility **on** and **off** based on whether or not any of their colliders are currently hit by a **Raycast** from the player character's FOV.

Death

If the NPCs health drops to zero, the NPC **dies**. It is immediately destroyed and a **Dead Body container** is instantiated where it died. This container plays a death animation as soon as it is instantiated. A randomized set of **items** is spawned into the container along with a number of **forced** items, such as the weapon the NPC was equipped with, along with ammunition and magazines for that weapon.

7.1.3 NPC Firearms

The NPCs use a **modified** version of the same firearm script as the player. All of the calculation used by the firearm of the player, such as hit chances, weapon accuracy and so on apply to the NPCs as well. However there are core **differences** in the way the firearms are implemented for NPCs.

Firearm Item

The firearm the player is currently using is determined by what firearm the player has currently **equipped** in their inventory. Since NPCs **do not** have any inventory, their firearm is **randomly** chosen and assigned to them when they first spawn. This is much the same Inventory Item (see 8.3) as the player uses, but instead of being stored in the inventory, it is attached to the NPC Game Object **directly**.

Firearm Operation

The main difference between the firearm script of the NPC and the firearm script of the player character is that in the case of the player character, the Player needs to be **limited** by the current state of the weapon (for example unable to shoot if the weapon is empty or the weapon is currently being reloaded). In the case of the NPC however, they can be simply programmed **not to attempt to shoot** if the weapon is not ready to fire. This makes their implementation much simpler and less taxing on computer resources.

7.1.4 Hostile Behaviour

As mentioned at the beginning of this section, at the time of writing of this thesis, the only implemented NPC type is a **hostile NPC**. The behaviour of NPCs is implemented as a finite state machine and is currently implemented with 5 behavioural states: **Idle**, **Patrol**, **Attack**, **Chase** and **Search**.

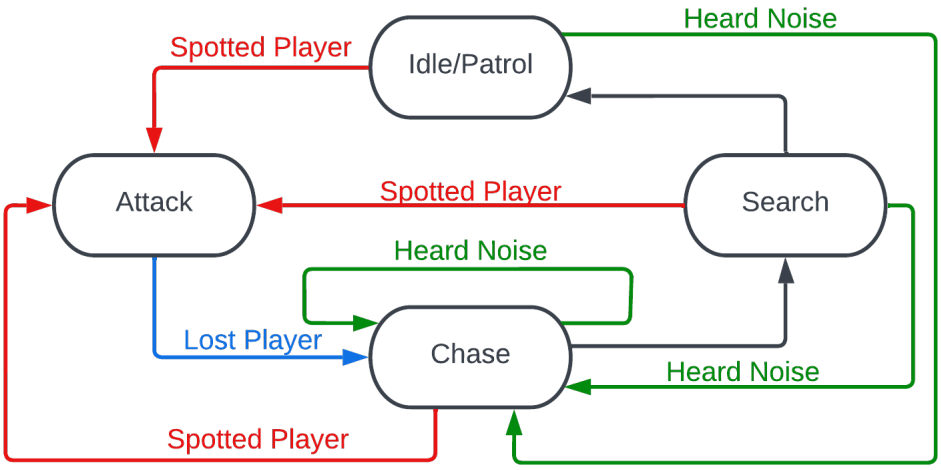

Figure 7.3: State machine of a hostile NPC.

Idle State

This is one of two **default** states of NPCs. In this state the NPC stands in the position it was spawned in. The NPC will periodically turn their head and **look around**.

Patrol State

This is the other **default** state of NPCs. In this state, the NPC has a **list of positions** set as its patrol locations. The NPC will move from position to position, briefly looking around each one.

Attack State

This state is entered when the NPC **spots** the player character. The NPC will **equip** their firearm if it was not equipped already. During this state, the NPC will **always** attempt to look at the player character and **fire** at them.

Before the NPC attempts to fire at the player, it will assume an **attack position**. This means that the NPC will attempt to find the **closest** pathfinding node to its current position and stand on it. Since occupied nodes are not considered valid (see 7.1.1), this helps with **clumping of enemies** when they attack the player. The enemies will still stand quite close to each other but will have a personal space of **1 square metre**.

While the NPC is in an attack state, it stays in **one spot**. This means that the player could potentially walk all the way in front of the NPC and since firearms in BTMC shoot

their bullets from the **muzzle**, the player could push **past** the muzzle and avoid damage. To get around this, the NPCs will **back away** from the player if the player gets too close. No path finding is used in this case, the NPCs simply move in the **opposite direction** of the player. While doing so, they shoot **Raycasts** behind them to detect any obstacles. Therefore the NPC stops backing away if they are **backed into an obstacle**.

Chase State

The chase state is entered either when the NPC **loses track** of the player in Attack state or **hears a noise** in any state besides the Attack state. When the Chase state is entered for the first time, the NPC will try to reach the **player character's last known position** or the **location of the heard noise**.

Search State

The search state is entered exclusively **after** the Chase state in case the player character was **not found**. When search state is first entered a random set of **5 locations** in the area around the last known position or the position of the heard noise are generated. The NPC will then **search** these 5 areas. If the player was not found, the NPC returns back to its **default state**.

7.1.5 Player Detection

Currently there are 2 ways an NPC may detect the player's presence:

- **Vision**
- **Hearing**

NPC Vision

The vision of NPCs works comparably to the player character. There is one key difference however. While **dozens** of Raycasts have to be computed to determine the FOV of the player character, the FOV of the NPC is much simpler. Instead of having many Raycasts calculated each frame, a Game Object in the **shape of a FOV cone** is placed onto the NPC. This FOV object has a **collider** component, set as a **trigger** attached to it. By using this trigger collider, the FOV object can detect when the player character **enters** it. When the player character enters the FOV object, a **single Raycast** is shot out of the NPC towards the player character each frame. If the Raycast hit an Obstacle along the way, it means that the player is **not** actually **visible**. Whether or not Half Obstacles block this Raycast is determined by **both** the stance of the player character and the NPC.

NPC Hearing

Besides seeing the player character, the NPCs may **hear** a gunshot, weapon manipulation or the sounds of movement. For this purpose I have created two classes: `NoiseOrigin` and `NoiseReceiver`. The Noise Origin can simulate noise by using an `OverlapCircleAll` method, which works similarly to Raycasts. Instead of being a line that detects all objects along its path, the Overlap Circle detects all objects within a **circular area**. The noise generator will only detect colisions with objects in a **„NoiseReceiver“** layer. Each NPC

has one of these Noise Receiver objects attached to it. This way when a noise is generated by a Noise Origin, **all** Noise Receivers within this circle are notified. The size of this circle is **variable** and depends on the **loudness** of the noise.

7.2 Ranged Combat

7.2.1 Firearms

There are currently 4 firearms in BTMC. Each firearm has its advantages and disadvantages. There are numerous ways the firearms are balanced such as damage, accuracy, fire-rate, weight, rarity, ammo scarcity and so on.

Firearms:

- **Pistol** - semi-automatic, medium damage, medium accuracy, plentiful ammunition, low weight, uses magazines
- **Assault rifle** - fully/semi automatic, high damage, high accuracy, scarce ammunition, high weight, uses magazines
- **Pump shotgun** - pump action, low damage pellets, bullet spread, plentiful ammunition, high weight, internal magazine
- **Hunting rifle** - bolt action, very high damage, very high accuracy, scarce ammunition, high weight, internal magazine

7.2.2 Firearm mechanics

All firearms are represented as the same **Firearm Game Object** attached to the Player Game Object. The Firearm is composed of **3** Game Objects. The **Muzzle** and two **Cone Lines**. The cone lines are only for debugging purposes and show the current **accuracy** of the firearm. The Muzzle is the object from which **bullets** originate. To the Muzzle is also attached a **Muzzle Flash** object which renders different muzzle flash sprites as the firearm is fired.

All firearm functionality is implemented in a single script called `FirearmScript` attached to the Firearm Game Object. The script uses numerous variables to change the behaviour of the currently selected firearm such as damage, accuracy, fire rate and so on. There has to be constant communication and synchronization between the **Firearm Script**, **Inventory Controller** and the **HUD Controller** as they are all intertwined. It also calls methods of the **Human Animation Controller** and the **Audio Manager**.

The general information specific to all firearms of the same type, such as whether it uses magazines, fire rate, accuracy, damage and so on are stored in the **ItemData** Scriptable Object (see 8.3) associated with the currently equipped firearm. Information about the current firearm as a standalone object such as current ammunition status are stored in the **InventoryItem** script of the firearm. The Firearm Script calls methods of the Inventory Controller such as firing a round, cycling of the firearm or reloading, which update the current state of the firearm in the InventoryItem script.

Firearm accuracy

The **accuracy** of a firearm is determined by the **firearm**, the **ability** of the player character and current recoil accumulated from **consecutively fired rounds**. Firearm accuracy is

represented by a **cone** originating from the **muzzle** of the firearm and centered towards the **point** where the player is aiming. The **degrees** of the cone are calculated each frame.

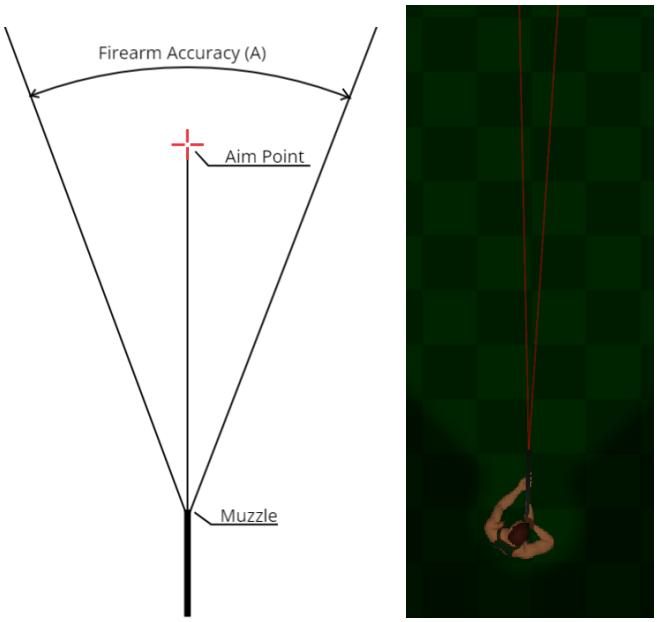

Figure 7.4: Aim Cone sketch (left) and an actual in-game aim cone displayed using cone lines (right)

Formula for calculating the firearm accuracy:

$$A = A_{base} * S + R$$

Where:

- ***A*** - Calculated weapon **accuracy** in degrees
- $\boldsymbol{A_{base}}$ - The **base** accuracy of the weapon. This is determined by the **firearm**.
- ***S*** - The current **shooting ability** of the player character. This is affected by injuries of the player character. This value is always ranges from 0.1 - 1 where 0.1 is the best possible ability.
- ***R*** - Current **recoil** of the firearm - accuracy modifier based on **consecutively fired shots**.

Firearm Recoil

It is expected that when someone is quickly firing from a firearm, they will be as accurate as when they are firing slowly. This is why I implemented firearm **recoil** and **cooldown** into the game.

The output of the recoil system is the **Recoil modifier** used when calculating the accuracy of the firearm.

Formula for calculating the firearm's recoil:

$$R = (R_{base} - 0.5 * (1 - S)) * B_{con}$$

Where:

- $\boldsymbol{R}$ - Current **recoil** of the firearm
- $\boldsymbol{R_{base}}$ - The **base** recoil increment per fired bullet. This depends on the firearm that is used (for example a shotgun has a much more powerful recoil per bullet than a pistol).
- $\boldsymbol{S}$ - The current **shooting ability** of the player character. This is affected by injuries of the player character. This value is always ranges from 0.1 - 1 where 0.1 is the best possible ability.
- $\boldsymbol{B_{con}}$ - Amount of **consecutively fired shots**. This value is **capped** at a certain amount and recoil does not accumulate past that amount.

0.2 seconds after the player stops shooting, the weapon recoil slowly decreases. The **cooldown rate** is one consecutively fired round each 0.1 seconds. By lowering the amount of consecutively fired rounds, the recoil calculation will return smaller values.

The presence of $\boldsymbol{S}$ in both of these equations means that the shooting ability of the player character has an effect on both the accuracy of the first bullet fired from the weapon and on the ability to control the weapon's recoil.

7.2.3 Bullet Mechanics

How weapons fire bullets is covered in the previous section. In this section I will explain how the bullets themselves behave.

Bullet Deviation

Each time a weapon is fired a **bullet deviation** is calculated based on the current weapon accuracy. This value, calculated in **degrees**, determines the deviation of the fired bullet from the point at which the player is aiming. This deviation is calculated for **each fired bullet** by first deciding whether the bullet will deviate from the point of aim to the **left** or to the **right**. Then a **random** number is picked between 0 and **half** of the current weapon accuracy.

I would also like to establish two new terms to make the explanations in the following sections easier: A **shooter** and a **victim**. A shooter is a person **firing a bullet** and a victim is the person who was **hit by a Bullet Raycast**. The player character and the NPCs can be be both in the role of a shooter or a role of a victim in different scenarios.

Bullet Raycast

After the Bullet Deviation is calculated for a bullet a **Ray** is fired, originating from the **muzzle** of the weapon towards the **point** where the player is aiming **plus the bullet deviation**. Each object that's hit by this Ray is added to an **array of collided objects**. Depending on the type of the object that was hit, a different **action** is performed.

Bullet Impact Point

There are 3 types of objects recognized by the bullet Raycast:

- **NPC or Player**
- **Half Obstacle**
- **Anything else**

When a **Half Obstacle** is detected on the path there are two possible outcomes. Either the bullet **hits** the obstacle or the bullet **flies over** the obstacle. This is based on the **distance** between the shooter and the Half Obstacle.

If the shooter stands:

- **less than 5** metres away from the obstacle, there is a **0%** chance of hitting the obstacle.
- **between 5 and 15** metres away from the obstacle, a **formula** is used to calculate the chance between **0%** and **50%** with a **linear** drop-off.
- **more than 15** metres from the obstacle, there is a **50%** chance of hitting the obstacle.

Formula for calculating the chance of a bullet hitting a Half Obstacle, when the shooter is standing 5 to 15 metres away from the obstacle:

$$P = (D - 50) * 0.005$$

Where:

- ***P*** - **Probability** of the bullet hitting the Half Obstacle
- ***D*** - **Distance** between the shooter and the Half Obstacle in **dm**

If the bullet **hits** the obstacle the point where it hit is registered as the bullet's **impact point**. Otherwise a distance between the shooter and the wall is **stored** for future calculations and bullet **flies over**.

When the bullet Ray passes through an **NPC or the Player (victim)**, it does not automatically mean that the victim was hit. Whether the victim was hit depends on multiple factors. These factors and possible outcomes are explained in subsection 7.2.4.

If the bullet **hit** the victim, it takes notes of each hit-box it passed through and calls a method on the victim's hit-box passing all body parts the Raycast passed through. Otherwise the bullet **flies over** the victim.

If the bullet Ray hits **anything else** other than Half Obstacle, NPC or the Player, the point at which the bullet hit the object is considered the bullet's impact point.

Bullet Impact and Muzzle Flash

Once the bullet's impact point is determined, a **Bullet Impact Game Object** is created from a prefab and is **positioned at the impact point and rotated away** from the origin of the bullet. This object has a **Sprite Renderer** component and an **Animator** with animations of different bullet impacts. Based on what type of obstacle was hit, a different bullet impact animation is selected. After the animation is done playing, the object is **destroyed**.

Besides the impact point a **Muzzle Flash Coroutine** is started. This coroutine selects a Muzzle Flash Sprite from an **array** of sprites based on the currently selected weapon. Each weapon has **3** muzzle flash sprites which alternate between shots. This ensures that weapons that can fire rapidly, such as the Pistol or the Assault Rifle have different muzzle flashes for consecutive shots and therefore look more natural.

After the muzzle sprite is selected, the Muzzle Flash Game Object attached to the Muzzle of the Firearm (see subsection 7.2.2) has its **Sprite Renderer** set to the selected sprite. After that the Sprite Renderer is enabled together with a **2D Light Source** that emits from the Muzzle Flash. After 0.05 seconds the flash and the light are turned off again. This produces, in my opinion a quite pleasant firearm flash which illuminates surrounding area and casts shadows.

7.2.4 Taking cover

An obvious way the victim can take cover from projectiles is to break the line of sight between them and the shooter by hiding behind a tall object (a **Full Obstacle**). This however, may not always be possible and even if it is, the victim loses the ability to return fire without leaving the cover.

In this section I will explain the implementation of taking cover from firearms without utilizing full obstacles. As mentioned in the subsection 7.2.3, just because a bullet Raycast passed through a hitbox of the victim, it does not mean that they were hit by the projectile. There is also a possibility that the projectile flew over.

Stance

There are two stances that the victim can take - **standing** and **crouching**. The stance has a considerable effect on whether they were hit by the bullet or the bullet flew over them.

If the victim is **standing**, there is a **100%** chance that they will be hit, regardless of the distance between them and the shooter.

If the victim is **crouching**, there are multiple outcomes determined by the distance between them and the shooter:

- **5 metres or less** - there is a **100%** chance that the shooter will hit the victim (stance has no effect at this distance).
- **5 to 15 metres** - **linear** drop-off from **100%** to **80%**
- **15 metres or more** - there is an **80%** chance that the shooter will hit the victim.

Formula for calculating the chance of hitting a crouched person between 5 and 15 metres away:

$$P = (1 - (D - 50) * 0.002)$$

Where:

- P - **Probability** of the bullet hitting the victim
- D - **Distance** between the shooter and the victim

This chance is **further modified** depending on whether the bullet passed a **Half Obstacle** on its trajectory.

Half Obstacles

If a projectile **passed** a half obstacle before reaching the victim a special formula is used to simulate **hiding behind cover**. The output of this formula is a **Hit Chance** modifier. This is only applied if the victim is **crouched**.

Standing victim has an implied cover from the fact that the projectile might hit the half obstacle. If it flew over the half obstacle, it is likely that it hit the standing victim.

If the victim is **crouched** however, the projectile could've flown over the half wall **and** over the victim as well. Furthermore there is a certain **safe distance** behind the half cover, where it should be impossible for the shooter to hit the victim.

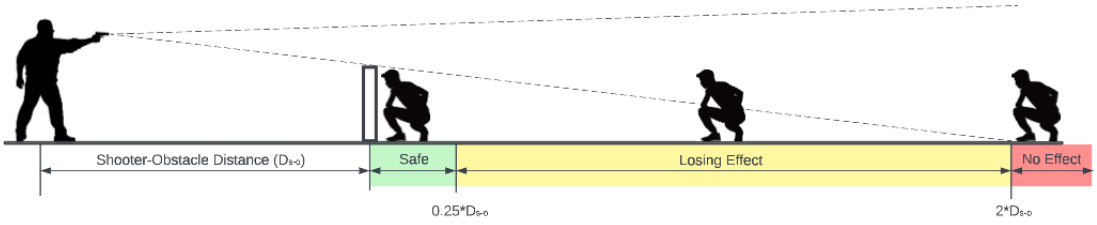

Figure 7.5: Illustration of taking cover by crouching behind a half obstacle.

As shown in the figure, there are **3 areas** behind the half obstacle. The sizes of these areas are determined by the distance between the shooter and the half obstacle (Shooter-Obstacle Distance - $\boldsymbol{D_{s-o}}$).

- **Safe Area** - If the victim is crouched in this area the Hit Chance modifier is **0**. This area spans between the **half obstacle and 0.25 * $\boldsymbol{D_{s-o}}$** behind the half obstacle.
- **Area where cover is losing effect** - If the victim is crouched in this area the formula below is used to determine the Hit Chance modifier. This area spans between **0.25 * $\boldsymbol{D_{s-o}}$ and 2 * $\boldsymbol{D_{s-o}}$** behind the half obstacle.
- **Area where the cover lost all effect** - If the victim is crouched in this area, the Hit Chance modifier is **1** and therefore the cover has no effect. This area starts at **2 * $\boldsymbol{D_{s-o}}$ behind the half obstacle and continues forever**.

Formula for calculating the Hit Chance in the Area behind cover where the cover is losing effect:

$$P = ((D_{v-o} - 0.25 * D_{s-o})/(1.75 * D_{s-o}))$$

Where:

- $\boldsymbol{P}$ - Hit Chance modifier
- $\boldsymbol{D_{v-o}}$ - Distance between the Half Obstacle and the victim
- $\boldsymbol{D_{s-o}}$ - Distance between the Half Obstacle and the shooter

The Hit Chance modifier is then multiplied by the chance that the bullet hit the enemy based on the **distance between the shooter and the (crouched) victim**.

7.2.5 Hit-boxes

Hit-boxes in BTMC are **colliders**, that serve to detect where a bullet hit a character. They are controlled by the methods of a `HumanHitbox` class used by both both the player character and NPCs. Switching of the colliders is done through **Animation Events**.

Each character is split into 5 hit-boxes:

- **Head**
- **Torso**
- **Legs**
- **Left Arm**
- **Right Arm**

Each of these body part's hit-boxes is implemented using a **2D Collider**, either a **Polygon Collider** or a **Capsule Collider**. One of the problems was that it is quite hard to change a polygon collider at runtime and therefore, body parts that use polygon colliders have multiple different colliders which are turned on and off depending on what hit-box should be used. Since composing animations for each frame of every animation would be quite tedious, there are only **4 composed hit-boxes**:

- **Standing Idle**
- **Crouching Idle**
- **Short Weapon Equipped**
- **Long Weapon Equipped**

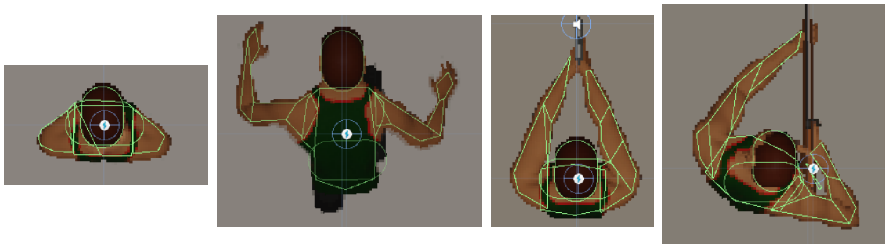

Figure 7.6: Hitboxes (green lines) from left to right: Standing Idle, Crouching Idle, Short Weapon Equipped and Long Weapon Equipped.

Once the firearm script of a player or an NPC sends information about which body parts were hit by a Raycast, the hit-box uses **probability** to determine which body part was hit as in a 2D top down game, the Raycast may pass through **all** hit-boxes at once. It splits individual hit-boxes into **main** hit-boxes (head, torso, legs) and **arm** hit-boxes (left arm, aright arm). The hit-box follows this criteria when deciding which body parts were really shot:

- If the bullet Raycast passed through **multiple** main body parts, **one** of these body parts is selected. This means that head, torso and legs **cannot** be hit in the same shot as it would not make much sense. The torso is most likely to be hit and the head least likely. If the legs are hit, there is a **50/50** chance on which leg was hit.
- If the bullet Raycast **did not** pass through a main body part but did pass through both arms (very likely when holding a firearm), each arm has a **50%** probability it was hit (but **at least one** has to be hit).
- If the **torso** was determined as a hit and the raycast passed through either of the arms, there is a **20%** chance that the arm was hit.
- Lastly if there is **only one** body part that the Raycast passed through, then there is a **100%** chance it was hit.

This seems a bit complicated because these probabilities were **clashing** with the previous implementation of missing a shot due to factors such as stance or cover. Therefore if the `FirearmScript` determined the shot as a hit, then the hit-box has to determine at least one body part as being **truly shot**.

Chapter 8

User Interface Implementation

Just like most other games, BTMC uses **User Interface** (**UI**) to communicate important aspects of gameplay to the player and read their input **outside** of gameplay. UI in BTMC can be split into 3 main sections:

- **Menu** - UI used to interact with the game itself **outside of the actual gameplay**. This section would include the Main Menu screen, Pause Menu screen and Options Screen.
- **HUD** - HUD (Head-up or Heads-up Display) is used to communicate valuable information to the player **during** gameplay. It is used for aspects that cannot be easily shown via in-game objects or mechanics (such as Hunger, Thirst Tiredness) or information that may not always be apparent from the game (Stance or Selected Weapon).
- **Inventory** - The Inventory screen provides interface for the **interaction** with found **items**, player **equipment** and **health** screen.

8.1 Menu

Menu screens in BTMC use layouts and buttons conventional for most games.

8.1.1 Main Menu

The main menu in BTMC is the first loaded scene when the game is started. It consists of the following buttons:

- **Demo Level** - Level created to showcase the state of the game at the time of writing of this thesis.
- **New Game** - Currently unavailable, will be used to start a new game when the game is finished.
- **Load Game** - Currently unavailable, the game is currently unable to save progress.
- **Options** - Used to change visual and audio aspects of the game.
- **Quit Game** - Turns off the application.

8.1.2 Pause Menu

Pause menu is entered from the game when player presses the **Esc** button. It consists of the following buttons:

- **Resume** - Closes the pause menu and resumes the game.
- **Save Game** - Currently unavailable, will be used to save the game to a save slot.
- **Options** - Used to change visual and audio aspects of the game.
- **Main Menu** - Returns the player to the Main Menu.
- **Quit Game** - Turns off the application.

8.1.3 Options Menu

Accessible from both the Main Menu and the Pause Menu. It is split to 2 sections:

- **Video** - Used to change resolution, quality preset, fullscreen option and the amount of FOV Raycastas.
- **Audio** - Used to control master volume, sound effects volume and inventory sounds volume.

8.2 HUD

The HUD in BTMC is separated to these main sections:

- **Upper Status Bars**
- **Lower Status Bars**
- **Stance**
- **Devices Section**
- **Weapon Section**

Figure 8.1: Overview of the main sections which make up the HUD.

8.2.1 Upper Status Bars

The Upper Status Bars contain **3** horizontal status bars that need to be long enough to communicate their status visually:

- **Health Bar** - The overall health of the player character and arguably the **most important** status bar. If this bar reaches zero the player character dies. Therefore it is positioned as **the first** status bar and also accompanied by a **number** to show the value more accurately.
- **Stamina Bar** - The current stamina of the player character. It is important for players to manage their character's stamina carefully, so that they do not end up in a fight or flight scenario out of stamina. Since stamina fluctuates **the most** from all of the status bars I have decided to omit a number showing the precise amount since it would only serve as a **distraction**.
- **Body Temperature Bar** - This status bar differs from all of the other status bars, because it does not fill up like the rest. Instead it shows a **static background** and a **slider** that indicates how cold or hot the player character feels (it does not display the surrounding temperature).

Health bar and Stamina Bar are also accompanied by **arrows** that indicate how fast their value is dropping or rising (3 arrows each way). This also serves as a good indicator for the player that something could be very **wrong** with the character (for example 3 left facing arrows on the health bar indicate a strong bleed). These values may also have their maximum value lowered by **injuries**, which is shown by a **hatched** area at the end of the bar.

Figure 8.2: Stamina Bar displaying two draining arrows (moderate amount of drain) and lowered maximum stamina by 10 points.

8.2.2 Lower Status Bars

The Lower Status Bars contain **3** short, vertical status bars that are not as important to be judged visually and fluctuate much less compared to the Upper Status Bars:

- **Hunger Bar** - The current nourishment level of the player character.
- **Thirst Bar** - The current hydration level of the player character.
- **Tiredness Bar** - How tired the player character is (full bar means that the character is fully rested).

Since these Status Bars are much shorter and their fullness might be a bit harder to judge visually compared to the Upper Status Bars, all of them are accompanied by a **number** showing their precise value. They also lack the arrows present in the Upper Status Bars. While the rate at which these bars fill up or deplete can change based on factors such as current body temperature or equip load, it is usually not as acute. Also the arrows are not very aesthetically pleasing in the very small Status Bars.

All of the status bars (beside the Body Temperature Bar) also change their **color** to bright red when their value drops below a certain threshold (less visible on the Health Bar as it is already red but there is still a difference). The Bars accompanied by a number also change the color of the number to **bright red** at this point. This threshold is usually **20%**.

8.2.3 Stance

The Stance is a simple outline that shows the current stance of the player character (standing or crouching). While this may seem a bit redundant as the player character can be seen, due to the artistic inexperience of the author when it comes to drawing, human anatomy and animations it is best to **make sure** that the player knows if their character is currently standing or crouching.

8.2.4 Devices Section

The devices section is used to present the player with additional information provided the player character **is wearing** such devices. Otherwise this section is **empty**. There are currently only two such devices in the game:

- **Watch** - A digital wrist watch, which displays current time, date and surrounding temperature.
- **Geiger Counter** - A device used to measure surrounding radiation in mSv (milisievert). Radiation is not currently implemented.

8.2.5 Weapon Section

The weapon section is only visible if the player is currently **holding a weapon**. It provides the player with all necessary information about the weapon the player character is currently using. The weapon section contains the following information:

- **Weapon Outline** - The outline of the used weapon. This outline serves to quickly visually inform the player about which weapon is currently selected without reading.
- **Weapon Name** - The name of the selected weapon.
- **Firing mode** - Which firing mode is selected on the firearm. The only weapon currently in the game which supports a selective fire mode is the Assault Rifle.
- **Magazine Bar** - A bar which uses a row of bullets to symbolize the amount of ammo in the magazine. This Ammo Bar depletes a bit unconventionally from bottom to top which is supposed to simulate the movement of bullets in a magazine which are pushed **towards** the chamber.

- **Chamber** - This visually tells the player whether the weapon is currently chambered and ready to fire.
- **Ammo Count** - The current ammo count and the capacity of the magazine. The current ammo count also takes into account the **chambered round** and therefore on a full magazine and a chambered round, the text will display an ammo count higher than the maximum capacity by one (for example 31/30).

Figure 8.3: Overview of the weapon section of HUD.

8.2.6 Rest Menu

When the player interacts with a bed, a rest menu is opened in the middle of their screen. In this rest menu the player may choose to rest for a **specified** amount of time (1-8 hours) or until the character is **well rested**. This menu shows the player their **current** and **projected** values of hunger, thirst and tiredness after the rest.

Figure 8.4: Rest menu.

8.2.7 Interaction Text

When there is an **interactable** object within the player character's **interact range**, a text is displayed above them. The text is composed of the name of the interactable object in brackets and the action that will be performed. The interaction is always tied to the **closest** interactable object in range besides one special case when there is a dead body next to a door. In this case, **the door has a higher priority**. This is done, because when an NPC dies close to a door and therefore a dead body is spawned in their place the player may to be unable to open doors and escape a potentially dangerous scenario or get stuck in a room with a closed door.

Figure 8.5: Example of an interaction text for a bed.

8.3 Inventory Screen

The implementation of the Inventory System is based on a tutorial created by a user named *Greg Dev Stuff* [10].

The Inventory in BTMC is split into 4 main sections:

- **Equipment** - Used to interact with items and manage equipment of the character.
- **Health** - Used to interact with items and manage treatment of the character.
- **Map** - Currently not implemented. Will show the map of the region.
- **Journal** - Currently not implemented. Will be used to manage quests.

The Inventory System in BTMC is quite complex and therefore in this section I will try to abstract the individual components as much as possible while still keeping the explanation thorough.

Figure 8.6: Screenshot of the Inventory Screen showcasing all of the different components.

8.3.1 Item Grid

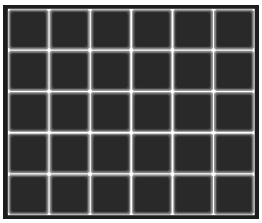

Figure 8.7: 6x5-tile item grid.

Item grid is one of the **main** components of this Inventory System. It simulates **space** for storing items. The grid is split into individual **cells** and always has a rectangular shape. Visually the grid is implemented as a tiled image, using the individual inventory slot sprites as tiles.

The grid contains a two dimensional array of references to **Inventory Items**. This array keeps track of what item is stored in each tile. Since the grid is just a **singular** object and not actually a grid of individual slots, it needs to somehow calculate where to store items or what item was selected when the grid was **clicked on**.

To find out which tile was clicked on, the grid uses a **function** which uses the **screen position** of the mouse to and converts it to the **grid position**. Then a secondary function is applied to figure out **which tile** was clicked. This causes a minor issue with different resolutions. Since the game was meant to be played at **FullHD** (1920x1080) resolution, clicks in any other resolution will not work properly when using the **raw** mouse position. This means that some of the values have to be **normalized** and **converted**.

To manipulate items, the grid provides functions to **get**, **grab** and **place** items. All of these functions require the tile coordinates. When the player attempts to place an item into the grid, multiple checks have to be performed. The outcome of these checks determines whether the item will be placed or not. Afterwards, all of the tiles that are occupied by said item are set to **reference** that item (using the aforementioned two dimensional array of references). When an item is grabbed from the grid, these references are set back to **null**.

Two main checks for item placement are the **boundary** check and the **overlap** check. Boundary check makes sure that the placed item does not protrude outside of the **boundaries** of the grid, while the overlap check makes sure that the item does not **overlap** with an item that is already placed. The overlap check also performs a check for **stackable** items which is one of a few cases when item overlap should be accepted (other cases often include manipulation with items related to weapons such as ammo and magazines).

Sometimes the grid is part of an **equipable** item which also serves as a **container**. In this case, the grid must be able to **save** and **load** its items. This is because when such an item is equipped to the character, this grid is **spawned** from a prefab into the inventory screen. When this item is unequipped, the grid is **destroyed**. If the items could not be saved, they would simply be destroyed with the grid. These items are saved in a array of references, that is being stored by the **item itself** in its `InventoryItem` script.

The grid also features two helping functions which are used when reloading firearms or loading magazines with ammunition. One of these functions serves to **find ammunition** items of certain type, by simply checking the entire array of item references and returning the first found item matching the criteria. The other function attempts to find the **best (fullest) magazine** using the same technique.

Lastly there is also a function which finds **available space** for an item of certain size.

8.3.2 Inventory Item

Figure 8.8: Assortment of different Inventory Items placed in an Item Grid.

Inventory items represent the real-life objects within the game. Their implementation is split into **two** main files. The `InventoryItem` script and the `ItemData` scriptable object. Inventory Items can take up one or more inventory slots, but just like the Item Grids the shape of the area they take up is **always rectangular**.

The Item Data is used to keep **static** information about objects, that are of the **same type** and share certain attributes. It **does not** hold information about an **individual** object (for example whether an item is a magazine is stored in Item Data while how much ammunition is currently inside it is stored in the Inventory Item script).

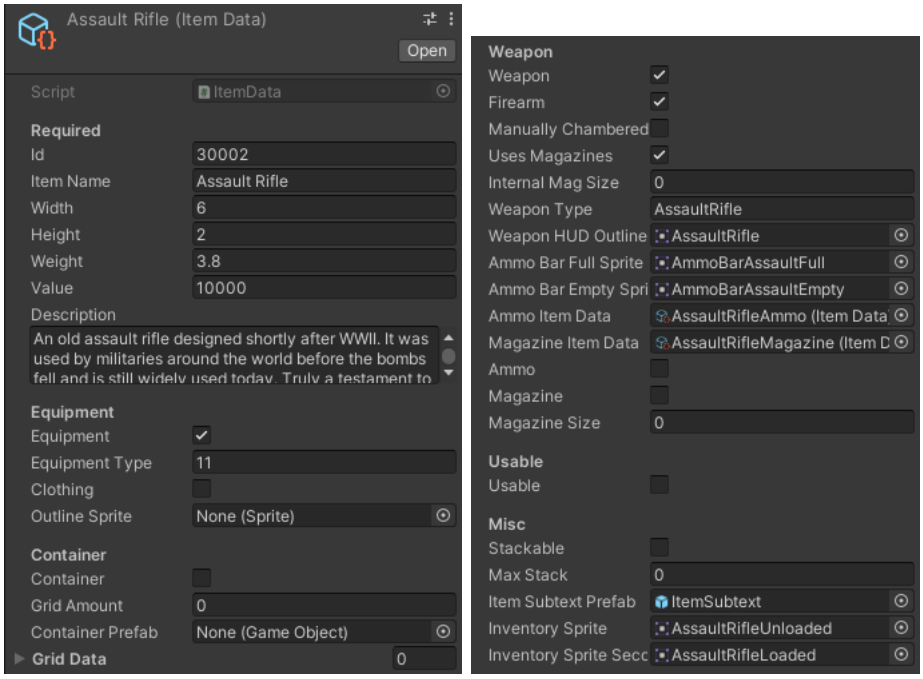

Figure 8.9: Example of the Item Data for an Assault Rifle item split into two parts.

There are many different item types (which often overlap) in the game such as:

- **Equipment** - Items that can be equipped to the player character such as clothing, weapons or devices.
- **Container** - Items that are able to hold other items such as backpacks, chest rigs, pants with pockets and so on.
- **Weapon**, firearm, magazine, ammo - Items tied to the combat system.
- **Stackable** - Items, that can be stored as one item, up to a maximum stack count such as ammunition.

The Inventory Item script implements the functionality of inventory items and it stores information that are **specific** to each **instance** of an item. Information about a specific instance consists of variables such as current position on grid, how many items are in the current stack, if the item is equipped and so on.

The general functions available to all Inventory Items are the function to set the basic item variables and the ability to **rotate** an item. The rest of the functions available to items are specific to certain item types.

The rotate function works simply by rotating the object in the game by **90 degrees** if the item is not rotated and rotating it **back** to normal if it is. If the item contains an **Item Subtext**, its position has to be updated as well. When retrieving the dimensions of an object, its rotation has to be considered and therefore if the item is rotated, the values for width and height are **swapped**.

Inventory Item script implements functionality and variables for all sorts of different items. Here are some examples of different item types and functionality provided for them:

- **Container** - Array of two dimensional arrays of items and a function to save and load items from grids that belong to this item.
- **Stackable** - Variable for current stack count and functions to set, subtract, add to the stack and update the text of the stack.
- **Magazines** - Variable for current ammo count and functions to load and unload ammunition from magazines
- **Weapons** - A large amount of functions for manipulation with magazines, chambers and firearm functions

Stackable items, magazines and weapons also use an Item Subtext. This is a small text, always located in the **bottom right tile** of the item which gives information either about the amount of items in a stack or the fullness of magazine or a firearm.

8.3.3 Item Highlight

Figure 8.10: Example of the Item Highlight on hover (left), valid placement (center) and invalid placement (right).

Before an Item is selected a **white** colored **Highlight** lets the player properly see which item is going to be selected on click.

If the player is already holding an item, they can clearly see which tiles will be **occupied** after placement. Using **red** and **green** color it also shows whether the tiles are free or occupied. The Highlight has many exceptions to this rule however because some combinations of overlapping items are **valid** (for example hovering stackable items of the same type over each other or hovering a round over a weapon to chamber the weapon).

8.3.4 Container Items

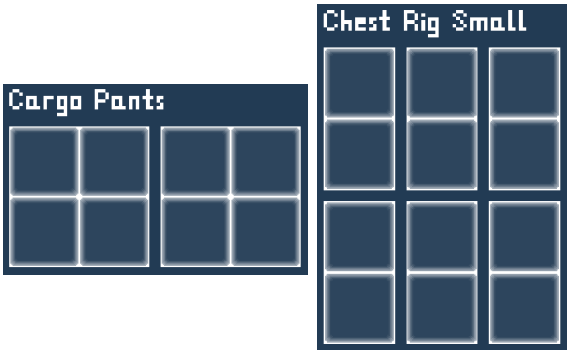

Figure 8.11: Container prefabs for Cargo Pants and Small Chest Rig

Container items are **equipable** clothing items that expand the character's inventory. They use a **prefab**, which contains their name and structure of one or more **Item Grids**. The item grids are structured to somewhat simulate the storage space of said items. In the case of the example above, the Cargo Pants have two large pockets, while the Small Chest Rig has 6 small compartments for magazines.

These prefabs are created in the Inventory Section in the middle of the screen and always follow the **same order** in terms of the type of equipment. This helps to achieve some sort of consistency. The order is:

1. Chest Rig
2. Torso Top Layer
3. Pants Top Layer
4. Backpack

This ordering is further important because different containers have different **capabilities**. For example when the player wants to use an item quickly during gameplay without opening their inventory (for example reloading a magazine), that item has to be placed in either the **chest rig** or **pockets**. Accessing items in the backpack requires the player to open their inventory. That's why containers which provide quick access to items are positioned at the **top** of the screen.

8.3.5 Context Menu

Figure 8.12: The entire possible context menu split into two halves.

The player is presented with this Context Menu, whenever they **right-click** on an Inventory Item. This menu is stored as a prefab and is used for different interactions with items. When the Context Menu is created, the menu decides which buttons should be shown and which shouldn't based on the **attributes** of said item.

Info - All items. Opens the Item Info Window.

Equip - Equipable but unequipped items. Equips the item if the corresponding item slot is free, otherwise does nothing.

Unequip - Equipped items. Unequips the item, first it tries to place the item into the character's inventory. If not possible places item into the container/ground Item Grid. If that is also impossible, nothing happens.

Open - Container items which are not equipped and not opened. Open the Container Item Window.

Close - Container items which are not equipped and opened. Closes the Container Item Window.

Use - Usable items, such as food, drink and medical items.

Attach Magazine - Firearms which use magazines and do not have a magazine inserted. This button tries to find the best magazine in both the inventory and container/-ground grid and inserts it into the weapon. If no magazine was found, nothing happens.

Remove Magaizine - Firearms which use magazines and have a magazine inserted. This button removes the magazine from the weapon and uses it as the Selected Item.

Chamber Round - Firearms, which have an empty chamber and an open bolt. Attempts to find a fitting bullet in the character's inventory or in the container/ground grid. If no ammo is found, nothing happens.

Clear Chamber - Firearms, which have a full chamber and an open bolt. Removes the round from the chamber and uses it as the Selected Item.

Rack Firearm - Firearms, which have an empty chamber and have ammo in the magazine. Takes one bullet from the magazine and inserts it into the chamber.

Open Bolt - All firearms, opens the bolt/slide/receiver of the firearm if it is closed. If there is a round in the chamber, it is removed and used as the selected item.

Close Bolt - All firearms, closes the bolt/slide/receiver of the firearm if it is open. If the has a magazine and ammo in it, this chambers a round.

Load Ammo - Firearms with internal magazines and magazines which are not full. Fills the weapon's internal magazine or a magazine with ammo from the inventory or the

container/item grid. This function tries to look for ammo until the magazine is either full or no more ammo was found.

Unload Ammo - Firearms with internal magazines and magazines which are not empty. Removes all ammo from the magazine and uses it as Selected Item.

Split Stack - Stackable items which have more than one instance of said object in them. Splits the stack into two halves and the new half is used as the Selected Item.

Destroy - All items. Destroys the item. If the item is a container or a firearm, all contents are destroyed as well.

8.3.6 Inventory Windows

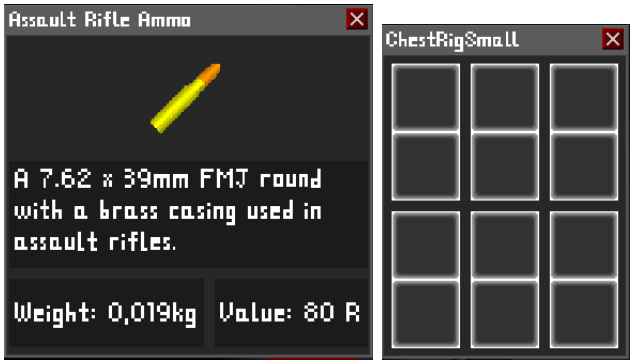

Figure 8.13: Item Info window (left) and a Container Window (right).

There are two types of inventory windows that can be opened using the **Context Menu**. The **Item Info** window and a **Container** window.

The Item Info window, which can be opened for all items contains the **image** of the item, item's **description**, various **information** about the item depending on the type of the item, weight and value. Only **one** info window can be opened per item.

The Container Window can be opened for Container Items that are **not currently equipped**. It allows the player to place items into these Containers without the need to equip them. Only one container window can be opened per item. This is done to circumvent the need to implement **synchronization** between different grid instances of the same item. If the item is moved or equipped the window will automatically **close** to prevent all sorts of bugs and exploits (such as storing the item into itself).

Both of these windows can be moved around and closed using the top of the window which displays the item's name and an „X“ button.

8.3.7 Item Slot and Equipment Outline

Figure 8.14: Example of Item Slots and Equipment Outline without and with items.

Item Slots are used to **equip** certain items. They are conceptually similar to the Item Grid and use functions for item manipulation such as **get**, **grab** or **place**, but instead of using cells, they simply hold **one** Inventory Item.

There are total of 14 item slots present in the game. They can be split into 3 categories:

- **Clothing**
- **Weapons**
- **Devices**

Clothing Slots: **Head**, **Chest Rig**, **Torso Base Layer**, **Torso Top Layer**, **Gloves**, **Backpack**, **Legs Base Layer**, **Legs Top Layer**, **Socks**, **Footwear**.

Weapon Slots: **Primary Weapon**, **Secondary Weapon**

Equipment Slots: **Watch**, **Geiger Counter**

Since the Item Slots vary in their size, and the items which they hold cam also vary in size, the items placed into these slots are visually **resized** and **rotated**, to be visually appealing and resized **back** when they are removed from the slot.

This section of the inventory also contains the **Equipment Outline** which shows the individual items equipped on the player character. This is **purely aesthetic**. This works by having an Outline Sprite assigned to items in their Item Data. These sprites are carefully created in such a way that they **do not** need to be individually resized and can be simply swapped.

8.3.8 Health Screen

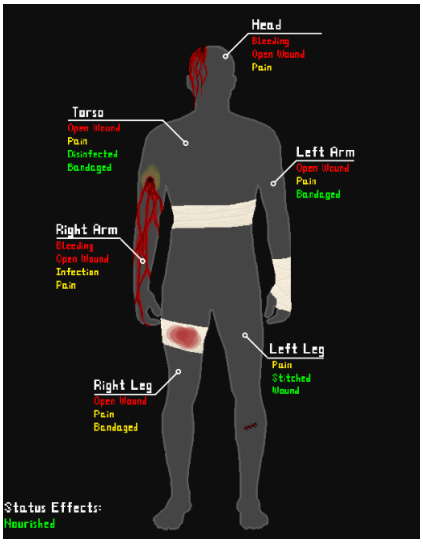

Figure 8.15: Health screen with various injuries on each body part.

The health screen looks exactly like the equipment screen but instead of the equipment outline, there is an outline showing the **current injuries** of the character. This outline shows the current **local status effects** for each body part and the **global status effects**.

The player can use local healing items (**Clean Bandage**, **Dirty Bandage**, **Antiseptic** and **Suture Needle** by dragging them onto the affected body part. There are also global healing items (**Painkillers** and **Antibiotics**) which the player consumes by using the Use option in the Context Menu.

For the health screen, I have **reused** the same Equipment Slots used in the Equipment section. They are just made invisible and have special behaviour, such as destroying one-use items and hiding the bandage items.

8.3.9 Inventory Controller

The Inventory Controller script works as an **interface** between the inventory system and the rest of the game and also as an interface between the different components of the inventory system itself. It handles functionality between Item Grids, Item Slots, Inventory Items, Context Menu and so on.

This script is the most **complex** and arguably the **most important** from the entire inventory system. This script holds all of the high level information about the current state of the inventory such as what the player has equipped, references to opened windows, what item is currently selected and many more.

Listing all of the functionality of this script would be overwhelming for this document but broadly speaking it handles communication between different components mentioned above by performing checks, controlling edge cases and so on.

It handles spawning of container prefabs and outline sprites when items are equipped, communicates with the HUD, handles the color and placement of the Item Highlight, opens the Context Menu and Inventory Windows, handles weapon selection and much, much more.

This script also implements functionality of **hotkeys**, which provide quality of life improvements when using the inventory. These hotkeys are:

- **Quick Transfer** - By Holding Shift by default, the player can quickly transfer items between the container/ground grid an the inventory grids. If an equipped item is clicked it is automatically transferred to the inventory.
- **Quick Equi**p - By Holding Alt by default, the player can quickly equip or unequip items by clicking on them.
- **Quick Split Stack** - By holding Ctrl by default, the player can quickly split a stack of items in half.

8.3.10 Firearms

Firearms in BTMC are implemented to reflect their real life functionality as much as possible. They use magazines, individual rounds, they have to be chambered to fire and so on. This makes them the most **complex type of items** in the game and therefore they need a lot of functionality.

For example, to avoid tedious work with the Context Menu, the player is able to load magazines into weapons by simply **dragging the magazine over the weapon** in the inventory. Rounds can also be dragged over weapons to chamber them or dragged over magazines to fill them up.

To illustrate the complexity of firearms I have drawn a decision tree which illustrates all of the different tasks that need to be performed during a **quick reload** of a firearm that uses magazines.

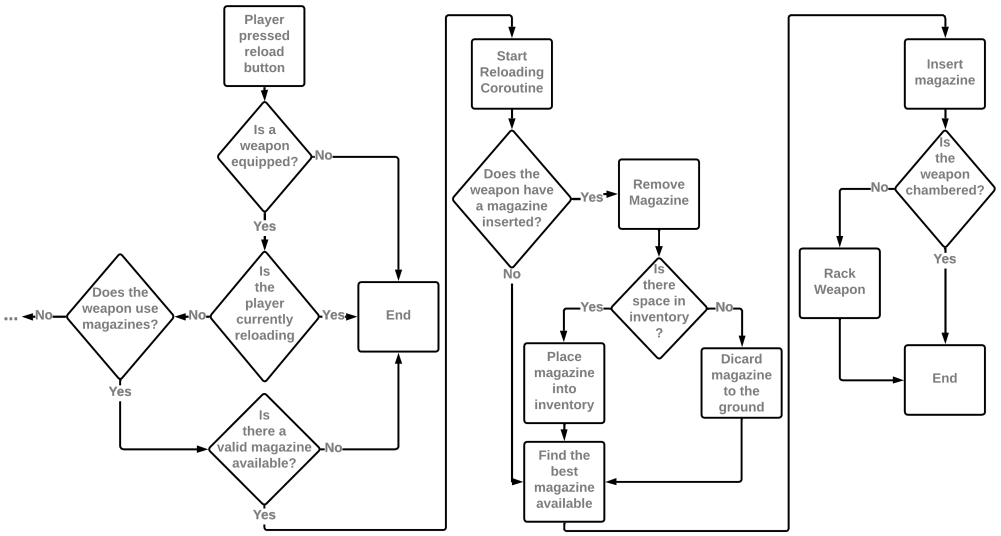

Figure 8.16: The decision tree for reloading a weapon using magazines.

It is also worth noting that this tree does not showcase many checks such as when the player switches weapons **during reloading**, it does not showcase animation activation, HUD updates and already uses abstract functions such as finding the best magazine. This still makes this decision tree a **very simplified** look at weapon reloading.

Chapter 9

Implementation of Animations and Sounds

9.1 Animations

9.1.1 Character Animations

Both the player character and the NPCs are composed of 4 visible Game Objects: the **Head**, the **Torso**, the **Legs** and the **Firearm Sprite**. Each of these objects behaves a bit differently concerning their **rotation**, **position** and **sprite sheet animations**.

I chose the approach of animating using sprite sheets, because I think it looks really well with this type of 2D game. It does have a major downside however. For each frame of the animation, a new, slightly different sprite has to be **„drawn by hand“**. This has proven to be extremely time consuming. Furthermore if the player character wears different clothing that needs to be visible on the player character, an entirely new set of animations has to be drawn for that specific type of clothing. At the time of writing of this thesis however, clothing items are **not** visible in-game.

As I have no art training and lack the skills to properly draw and shade human anatomy, the character animations were created in a sort of **stop motion capture**. I have created a rough human model in a 3D software named **Blender**. This human model was created with an **armature**, which let me move its limbs in a natural manner. I roughly painted the model within the software using **texture painting**. With this set up, I could simply **arrange the the model** into one of the frames of the animation I was creating, **screenshot** it from above, **pixelate** it to fit the aesthetic of the game and edit it in any regular paint software. This process is done a few times for each sprite sheet.

To control which animation should be played at a specific time, I used an **Animation State Machine** provided by Unity, which allows me to control the animations by transitioning between states using **control variables**. This state machine is composed of many sub-state machines and is frankly too complicated to be reasonably illustrated here.

To have a better control of these animations, the animator of the characters is controlled by a shared `HumanAnimationController` script. This script contains methods which play the correct animations for movement or weapon manipulation and also make sure that **animation sounds** are played at the right time. These methods are then called at the correct time from different scripts (such as the firearm script calling a weapon equip animation when a new weapon is selected).

The Head

The Head Game Object is mostly only **rotated** and **moved** in relation to the rest of the body. Currently, the only exception is when the character us using a **long firearm**. In this case, the head's sprite is swapped for a sprite in which the head is slightly leaning to the right. The Head always rotates **towards the cursor**.

When the player character is standing, walking or running, the Head is positioned in the **center** of the Player Game Object. When crouched however, the Head has to be **moved** slightly in relation to the center of the Player. This movement is done through the **Animator** component of the Torso Game Object. When the sprites of the Torso are changed firing crouching down or standing up, the Head is moved to a new position, so that it always stays at the end of the neck.

The Torso

The Torso Game Object uses **multiple** sprite sheets for different actions. Currently animated actions are the following:

- **Walking (standing and crouching)**
- **Running**
- **Crouching Down**
- **Standing Up**
- **Equipping a long firearm (standing and crouching)**
- **Equipping a short firearm (standing and crouching)**
- **Death Animation**

There are still many animations that have to be added, such as all weapon manipulation animations. These animations are, at the time of writing of this thesis, replaced by **placeholder** animations composed of different frames of the weapon equip animations.

Just like the Head, the Torso also rotates towards the cursor, however it moves only after the Head has moved in a certain direction for at least **30 degrees**. This makes the character first rotate their head and only after they reach said threshold, they rotate their body.

Figure 9.1: Torso sprite sheet for walking animation in BTMC.

The Legs

The legs also use multiple sprite sheets for movement animations, just like the Torso. When the character is standing still, the Legs rotate **in sync with the Torso**. If the character us moving, the Legs are rotated in the **direction of the movement vector**.

The Firearm Sprite

The Firearm Sprite Game Object is a part of the character in terms of animation. It is responsible for animating the currently equipped firearm in the character's hands. The Torso weapon animations are played **in sync** with the Firearm Sprite animations. This is important to ensure, so that the character is not, for example, shooting with empty hands. The animations are created to fit each other with a **pixel-perfect** precision. The Animator for the Firearm Sprite uses `Animator Overrides`, which use the same state machine but replace the animations with different ones, based on the weapon the character is currently using.

Figure 9.2: Torso sprite sheet for equipping a long weapon and an assault rifle sprite sheet.

Figure 9.3: Final in-game result (together with the head sprite).

9.1.2 Environment Animations

Environment animations are animations of visible objects in the environment. These are much simpler than character animations. Examples of environment animations are animated **containers**, **doors** or **fires**.

Figure 9.4: Sprite sheet for an Ammo Crate container.

9.2 Sound Design

The sound design in video games is, in my opinion, responsible for a large part of **immersion**.

9.2.1 Sound Implementation

The implementation of sound is based on a tutorial created by a user named *Brackeys* [3].

This system uses an `AudioManager` class which is responsible for swapping audio clips on different **Audio Sources** attached to objects within the game.

There are 2 major audio clip categories in BTMC:

- Clips that **cannot overlap** - These clips will be canceled by other audio clips played on the same game object. For example weapon manipulation sounds.
- Clips that **can overlap** - If these clips started playing, they will finish playing. These are various inventory sounds, gunshots, walking sounds and so on.

9.2.2 Audio Listener

Each scene in Unity should have only **one** Audio Listener. This Audio Listener is in most games attached to the **Main Camera**. In BTMC however, I wanted to implement **directional** sound based on the position of the player character.

This was hard to implement using a camera which sees the scene from far up. For this reason, the audio listener is attached to the **player character**. If the component was attached directly to the character however, the sound would also depend on the current **rotation** of the player. While this could be more realistic as sounds behind the character would be quieter, this also created a very **confusing** experience.

Therefore the Audio Listener is attached to an **Audio Listener Game Object** which is attached to the player and always faces the **same direction** as the camera. This produces the desired directional sound behaviour.

9.2.3 Randomized pitch

Some audio clips have **randomized pitch** within a certain range when they play. This is especially important for sounds like gunshots, which are played in quick succession. If they did not have randomized pitch, they would sound extremely unrealistic as real gunshots fired from the same weapon **do not** sound the same.

9.2.4 Sound recording

I wanted to record as many of my own audio clips as possible. I managed to record audio for some of the **weapon** sounds, **inventory** sounds, sounds for **item interaction** and so on. Despite this, there were sounds I couldn't record as I either **did not have access** to those specific items or good enough replacements (pump shotgun) or I did not have good enough microphone to records the sounds in a **satisfactory** quality (gunshots).

The sounds I couldn't capture myself were downloaded from a website called **Pixabay**, which holds a large range of royalty free sounds for both personal and commercial use without the need for mention of their site or the creators as a source.

The sounds I did record were cleaned of background noise, cut and edited using a free audio editing software called **Audacity**. Despite using a relatively cheap microphone for

recording, the audio sounds, in my opinion, unexpectedly good and, in many cases, is much better than the audio I downloaded from the internet.

Figure 9.5: Photos of the recording of a pistol rack (left) and empty casings hitting a wooden floor (right).

Chapter 10

User Testing

Even though BTMC is, at the time of writing of this thesis, far from a finished product, user testing and overall input of potential customers is an important aspect of the development of any video game or a software product.

User testing for BTMC was done on a sample of **10** people. This is quite a small sample but considering the very early stage of development, I was not yet comfortable providing the game to a larger audience. These people were given:

- A **rough description** of the game's mechanics, such as combat, inventory screen, health system and so on.
- A list of **known bugs** which are easy to find but are either rather hard to remove or they are not too detrimental to gameplay
- A list of features that were **not implemented** and their lack of implementation could be mistaken for a bug
- The **control scheme** of the game

10.1 Demo Level

Considering that BTMC does not yet have enough features and content to be played as originally intended, I have created a **demo level**. This demo level is supposed to serve as a demonstration of the implemented mechanics mentioned in chapters 5 - 9.

This level is created as a singular scene, composed of multiple buildings populated with hostile NPCs. The player is spawned in a small house in the center of the map. In this house, there are a few **pre-determined** starting items such as one container item, so that the player can utilize the inventory, basic healing items and a firearm with ammunition.

The end goal of this level is to simply survive **for as long as possible**. The player can try to clear buildings, which periodically **re-spawn** enemies and loot. The player can then obtain resources by looting either container objects within those buildings or the dead bodies of killed NPCs. When the player character dies, they are shown a death screen with the cause of death and the amount of time they survived for.

10.2 Bug Reports

One of the most important reasons user testing is needed in early development is the discovery of **bugs**. When there is only one person testing the video game, especially when the person is a developer of the game, they might get a figurative **tunnel vision**. What this means is that they may focus on certain mechanics, where they expect most bugs and not explore and test other parts of the game.

This also allows for the game to be tested on a variety of hardware and operating systems, which can reveal bugs and performance issues that may be unique to certain configurations.

In the case of BTMC however, there were **hardly any** bugs reported by the players. Most of these bugs were connected to the game being played in **different resolutions**, which causes misalignment of UI elements.

10.3 User Input

Besides bug reports, getting input about the **playability** of the game is also quite important. Since the people who tested the game were made aware of the early stage of development, they understood that balance, difficulty and amount of content were not the priority of testing. There were however mentions and proposals about different **quality of life improvements**, such as items automatically stacking when quickly transferred into the inventory.

10.4 Performance

10.4.1 FPS

Frames per second or **FPS** is one of the most commonly used metrics to describe how well a game performs on different devices. For computer games, an FPS count **over 60** is in most cases considered satisfactory. On lower FPS counts, the game may appear unresponsive or the player's input may feel „sluggish".

When the FPS count drops **below 24**, individual frames become noticeable, which results in a choppy and fragmented appearance of the game, making it difficult for the player to have a smooth and immersive experience.

In the table below are listed different middle-end and high-end computers on which the game was tested, together with observed FPS counts.

CPU	GPU	RAM	OS	FPS Count
Intel Core i5 - 4460 3.2 GHz	AMD Radeon RX 470	8GB	Windows 10	90-110
Intel Core i7-9750H 2.3 GHz	NVIDIA GeForce GTX 1050	16GB	Windows 11	90-110
Intel Core i5 - 4670K 3.4 GHz	NVIDIA GeForce GTX 1070	16GB	Windows 10	100-120
Intel Core i7-11800H 2.6 GHz	NVIDIA GeForce RTX 3070	16GB	Windows 10	150-210
AMD Ryzen 5 6600H 3.3 GHz	NVIDIA GeForce RTX 3060	16GB	Windows 11	140-180
AMD Ryzen 5 3600X 3.8 GHz	NVIDIA GeForce RTX 3080	16GB	Windows 10	150-200
AMD Ryzen 7 5800X 3.8 GHz	NVIDIA GeForce RTX 2070	32GB	Windows 11	220-250
AMD Ryzen 7 5800X3D 4.5 GHz	NVIDIA GeForce GTX 1070Ti	16GB	Windows 11	180-200
AMD Ryzen 9 5900X 3.7 GHz	NVIDIA GeForce RTX 3080	32GB	Windows 11	180-220

I have also decided to test the game on old, low-end devices which were not designed to run video games.

CPU	GPU	RAM	OS	FPS Count
AMD Athlon II X2 250e 3.00GHz	AMD 760G (Integrated)	4GB	Windows 7	< 10
Intel Celereon CPU N3060 1.6GHz	Intel HD Graphics (Integrated)	4GB	Windows 10	20-30

I have identified **NPCs** as the main source of lower performance. Currently, NPCs are quite badly optimized, especially regarding **pathfinding**. In the current rendition of the demo level, there are a maximum of **29** NPCs active at any given time. In later versions of the game, there is likely to be some sort of **chunk system** implemented. In this system, the map is split into different areas or chunks. This way, each NPC or any performance intensive part of the game can be **assigned** to a chunk. Using this system, it would be possible to **deactivate** them when the player is not within that chunk of the map. This would of course save a lot of computer resources, if done properly.

10.4.2 Rapid Access Memory

Another important metric to consider is the usage of **Rapid Access Memory** or **RAM**. I have analyzed the RAM usage of BTMC using an experimental package from Unity called **Memory Profiler**. This tool allows developers to take **snapshots** of RAM when the game is running and see which parts of the game take up the most space.

Through this tool I have found, that at the time of writing of this thesis, BTMC uses roughly **200MB** of RAM during runtime. This number starts a bit smaller but due to the loading of resources (such as loading an image of an item, when it is spawned for the first time), it slowly rises.

The largest chunk of memory (currently roughly 140MB) is used by **Render Textures**. These include the render textures used in the FOV/FOW system (see 6.3), but also Render Textures created by Unity, which are needed to render the scene.

Other considerable chunks of memory are consumed by regular **Textures** such as the ones used for items and UI elements. Since BTMC is designed in something of a pixelated art style, these textures do not consume too much memory individually, but they do add up during gameplay.

Chapter 11

Conclusion

In conclusion the process of design and implementation of this video game has been quite a learning experience.

Firstly I would like to address the **current state** of the game. I avoid using the term „final product“ because the development of BTMC **will continue** beyond the scope of this thesis. There were a lot of things planned for BTMC that I never had any chance of completing on time. The project was simply **too ambitious** from the beginning. This was one of my first learning experiences in this project. **Learning to manage one's expectations about a project and being able to say no to ideas about new features (even one's own ideas)**.

Some of the features were **greatly expanded** in their functionality compared to what was originally planned. Features such as the Inventory System or the Firearm Functionality turned out better than expected but of course, this took time away from all the other features that were never implemented due to time constraints. Features such as melee combat, non-hostile NPCs, character creation, RPG elements, quests and much, much more.

I originally planned to focus on the technical side of the game first and **after** that was done, I planned to focus on the creation of Assets, Sounds and the like. During the implementation I have realized that the art and technical side of the game go **hand-in-hand** and it is very hard to implement one after the other. This is because, without some **major prior experience** with game development, it is quite hard to have an idea about implementation of certain technical aspects without a reference to what the final game will look like.

I have also **severely underestimated** the importance of **art** in video games. Until this project I looked at video games as a product of **programming**. Now I feel like programming is definitely a smaller portion of a video game than I originally thought. Even on a game, as visually simple as BTMC, I spent **dozens and dozens of hours** drawing items and objects, creating animations, mixing sounds and other things that have next to nothing to do with programming. From now on I am looking at games, first and foremost, as **pieces of art**. I feel like artists, writers, sound designers, music composers, animators and other non-programming roles really do not receive enough credit in the video game industry, despite being such a **huge** part of every single video game.

On the technical side of things, I have learned the importance of good **code segmentation** and **compartmentalization**, creation of **proper APIs** for individual classes and **reusable** code. Many times throughout the development I have written a piece of code that applied to a **specific** problem, only to need, pretty much the same code for a different

application. This meant either **rewriting** of the original implementation to accommodate both use cases or **copying** functionality from the original code, to the new one. Another large problem was the considerable **difference** between my programming skills in the beginning of the project and towards the end. In the beginning I was much less experienced, lacked knowledge about different built-in Unity methods and overall structure of C#. This means that the older code feels like it was written by someone else and when I needed to apply the functionality of scripts that were months apart in creation, it was not an easy task.

Lastly I would like to address the importance of **planning**. **Experienced** programmers or project leaders always mention how quite a bit of time should be spent on planning as it will save a great deal of time during development. While this is true, I would like to point out that this does not necessarily apply to **inexperienced** people. While I could have spent longer planning BTMC, and it **may** have resulted in more realistic expectations, planning for things you have little to no experience with is mostly **futile**. Many of my plans were not met simply because I had no idea how some of the features **could be** implemented, what were the **limitations** of the software or **how much time** certain aspects of development take. If I planned the game and its development again I surely would prepare a much more realistic plan.

Despite all of the problems listed above I am still **very happy** with the results and I plan to continue my work on BTMC in the future months and years, until it becomes the game that I originally envisioned and more.

Bibliography

[1] Arm. *Gaming Engines* [online]. 2022 [cit. 2023-1-18]. Available at: https://www.arm.com/glossary/gaming-engines.

[2] Beresford, P. A history of RPGs. *Den Of Geek* [online]. january 2011, [cit. 2023-1-19]. Available at: https://www.denofgeek.com/games/a-history-of-rpgs/.

[3] Brackeys. *Introduction to AUDIO in Unity* [online]. May 2017. Available at: https://www.youtube.com/watch?v=6OT43pvUyfY.

[4] Cheema, R. 5 Biggest Video Game Companies In The World. *Insider Monkey* [online]. november 2022, [cit. 2023-1-16]. Available at: https://www.insidermonkey.com/blog/5-biggest-video-game-companies-in-the-world-1090614/?singlepage=1.

[5] Cobbett, R. The history of RPGs. *PC Gamer* [online]. may 2021, [cit. 2023-1-19]. Available at: https://www.pcgamer.com/the-complete-history-of-rpgs/.

[6] CodeMonkey. *A* Pathfinding in Unity* [online]. October 2019. Available at: https://www.youtube.com/watch?v=alU04hvz6L4.

[7] CodeMonkey. *Field of View Effect in Unity (Line of Sight, View Cone)* [online]. October 2019. Available at: https://www.youtube.com/watch?v=CSeUMTaNFYk.

[8] Cornett, B. The Post-Apocalyptic Genre: Definition, Characteristics, Examples and More. *BRANDON CORNETT* [online]. october 2022, [cit. 2023-1-19]. Available at: https://www.cornettfiction.com/post-apocalyptic-genre-explained/.

[9] Deckhead. How Long Does It Take To Make A Video Game? *HP Development Company* [online]. february 2020, [cit. 2023-1-18]. Available at: indiegamedev.net/2020/02/05/how-long-does-it-take-to-make-a-video-game.

[10] GregDevStuff. *Grid Inventory in Unity Tutorial Tile based inventory in Unity* [online]. October 2021. Available at: https://www.youtube.com/watch?v=2ajD1GDbEzA.

[11] Houghton, D. Why are there so many survival games? And why do we love the pain they bring? *GamesRadar+* [online]. october 2014, [cit. 2023-1-16]. Available at: https://www.gamesradar.com/why-are-there-so-many-survival-games-and-why-do-we-love-pain-they-bring/.

[12] North, J. A Brief History of the RPG. *PC Gamer* [online]. april 2020, [cit. 2023-1-19]. Available at: https://medium.com/super-jump/exploring-video-game-genres-role-playing-games-5dd55221d16d.

[13] Pavlovic, D. Video Game Genres: Everything You Need to Know. *IndieGameDev* [online]. july 2020, [cit. 2023-1-19]. Available at: https://www.hp.com/us-en/shop/tech-takes/video-game-genres.

[14] Pérez Latorre Óliver. Post-apocalyptic Games, Heroism and the Great Recession. *Game Studies* [online]. december 2019, vol. 19, no. 3, [cit. 2023-1-19]. ISSN 1604-7982. Available at: http://www.gamestudies.org/1903/articles/perezlatorre.

[15] Research, G. V. *Video Game Market Size & Share Growth Report, 2030* [online]. 2022 [cit. 2023-1-16]. Available at: https://www.grandviewresearch.com/industry-analysis/video-game-market.

[16] Santzo84. *Unity 2019 - 2D Fog Of War Tutorial using Render Textures and Shaders* [online]. October 2019. Available at: https://www.youtube.com/watch?v=MUV9Nr-cIGU.

[17] Sibony, J. The Best 7 Gaming Engines You Should Consider for 2022. *Incredibuild* [online]. february 2021, [cit. 2023-1-18]. Available at: https://www.incredibuild.com/blog/top-7-gaming-engines-you-should-consider.

[18] StudyTonight. *Game Engine and History of Game Development* [online]. 2022 [cit. 2023-1-18]. Available at: https://www.studytonight.com/3d-game-engineering-with-unity/introduction.

[19] Sullivan, S. Why Are Survival Games Popular? *MMOs* [online]. october 2015, [cit. 2023-1-16]. Available at: https://mmos.com/editorials/why-are-survival-games-popular.

[20] Townley, D. Gaming's singularity: how games media is taking over the world. *The Drum* [online]. may 2022, [cit. 2023-1-16]. Available at: https://www.thedrum.com/opinion/2022/05/16/gaming-s-singularity-how-games-media-taking-over-the-world.

[21] Wijman, T. The Games Market and Beyond in 2021: The Year in Numbers. *Newzoo* [online]. december 2021, [cit. 2023-1-16]. Available at: https://newzoo.com/insights/articles/the-games-market-in-2021-the-year-in-numbers-esports-cloud-gaming.