Politechnika Wrocławska

Wydział Informatyki i Zarządzania
kierunek studiów: Informatyka

Praca dyplomowa – inżynierska

Video game with local multiplayer based on Unity Engine

Kamil Wojtysiak

słowa kluczowe:
Unity
Video game
Multiplayer

krótkie streszczenie:
The goal of this thesis is to project and implement a video game that can be played by several players simultaneously on a single machine. Project consists of introduction to the topic and analysis of existing solutions, followed by explanation of applied design choices, description of problems encountered and solved during implementation of the application prototype as well as possibilities for its further development.

opiekun pracy dyplomowej	.... *Tytuł/stopień naukowy/imię i nazwisko*	.... *ocena*	.... *podpis*
Ostateczna ocena za pracę dyplomową			
Przewodniczący Komisji egzaminu dyplomowego	.... *Tytuł/stopień naukowy/imię i nazwisko*	.... *ocena*	.... *podpis*

*Do celów archiwalnych pracę dyplomową zakwalifikowano do:**
a) kategorii A (akta wieczyste)
b) kategorii BE 50 (po 50 latach podlegające ekspertyzie)
* *niepotrzebne skreślić*

pieczątka wydziałowa

Wrocław
2019

Streszczenie

W obecnych czasach gry wideo stały się jedną z najpopularniejszych form rozrywki, której rynkowa wartość od wielu lat stale wzrasta. Dzięki powszechności dostępu do Internetu oraz przez popularyzację modelu wieloosobowej rozgrywki sieciowej, mało współczesnych gier oferuje możliwość przeprowadzenia rozgrywki wieloosobowej lokalnie. Jest to jednak błąd, ponieważ jako istoty społeczne nadal lubimy spotykać się i dzielić naszą rozrywką, taki rodzaj rozgrywki nadal jest potrzebny.

Niniejsza praca prezentuje projekt i implementację projektu gry opartego na tym założeniu. Pracę rozpoczyna wprowadzenie w tematykę oraz przegląd konkurencyjnych rozwiązań. Następne rozdziały przedstawiają proces tworzenia gry: etap preprodukcji, projektowania oraz implementacji. Rozdział poświęcony projektowaniu przyjął formę dokumentu opisującego koncept gry - GDD. Rozdział poświęcony implementacji prezentuje przebieg procesu tworzenia prototypu gry, wykorzystane wzorce projektowe, napotkane problemy implementacyjne oraz metody użyte w celu ich rozwiązania, jak również plany dalszego rozwoju projektu.

Efektem pracy jest działająca gra komputerowa dla dwóch graczy, obsługująca sterowanie za pomocą klawiatury jak i zewnętrznych kontrolerów.

Abstract

Nowadays, video games have become one of the most popular forms of entertainment, whose market value has been steadily increasing for many years. Thanks to the widespread access to the Internet and by popularization of the online multiplayer game model, few modern games offer the opportunity to conduct multiplayer games locally. However, this is a mistake, because as social beings we still like to meet and share our entertainment, this kind of gameplay is still needed.

This work presents the design and implementation of a game project based on this assumption. The work begins with an introduction to the subject and a review of competing solutions. The following chapters present the game development process and its stages: pre-production, design and implementation. Design chapter took the form of a document describing the concept of the game - GDD. Implementation chapter presents the process of creating a game prototype, used design patterns, implementation problems encountered and methods used to solve them, as well as plans for further development of the project.

The result is a playable computer game for two players, with control scheme supporting the use of a keyboard as well as external controllers.

Table of contents

1. Introduction

This chapter serves as an introduction to the topic and terminology of video game development process in the scope of the thesis. It includes description of the goal of the thesis and overview of similar existing solutions, as well as substantiation for the choice of target platform.

1.1. Topic and terminology

1.1.1. Video games

Video game is an interactive form of entertainment which allows input through user interface to generate visual feedback on a video display such as TV screen or computer monitor. Historically, "Cathode ray tube Amusement Device" patented in 1947 is considered to be the first video game created [1]. However, it weren't until early 1980s, that with appearance of video arcades video games became widely known to general public.

1.1.2. Local multiplayer

Local multiplayer is a type of game where several players play simultaneously on a single display device. Many of the first video games were local multiplayer, as they were designed to be played by two players competitively. Most famously known example of such game would be "Pong", manufactured by Atari in 1972 [2]. This type of multiplayer is often achieved by dividing screen into smaller sections, either horizontally, vertically or dynamically, with a technique called "split screen" [figure 1.1].

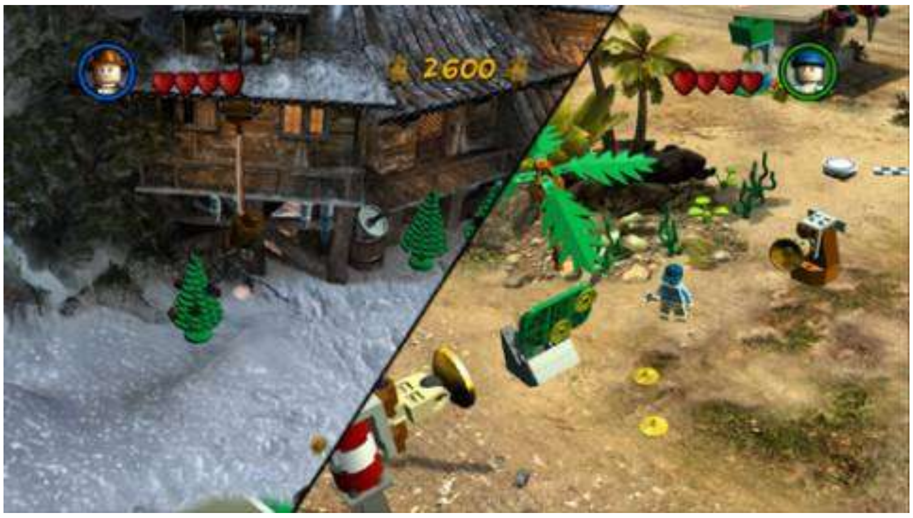


Figure 1.1. Dynamic split screen technique used in "Lego Indiana Jones", access 08.11.2019
source - https://www.giantbomb.com/dynamic-split-screen/3015-6795/

Other techniques of approaching local multiplayer include designing a game with static gameplay area which allows to display all action happening in the entire level at once (e.g. Pong, Bomberman), only allowing players to move in turns with a playstyle called "hot seat" (e.g. Worms, Heroes), or by using a specialized algorithm to allow camera to track all player characters at the same time (e.g. Super Mario Brothers, Mortal Kombat), either by limiting their movement possibilities or by panning out when characters stray too far from each other.

1.1.3. Platformers

Platformers, or platform games, originated in early 1980s. The term describes games where jumping on suspended platforms and avoiding obstacles is an integral part of the gameplay. Term was coined in 1983 [3] and is considered as a valid video game genre since. While genre is developed in 3D as well as 2D graphics, “side scroller” or profile view type camera is prevalent choice in this genre, as three-dimensional view has proven to make measuring jump distance between platforms exceedingly difficult.

1.1.4. Fighting games

Fighting games are a type of action game where two (or sometimes more) on-screen characters fight each other [4]. The genre revolves around a fighting match divided into rounds, where characters are usually restricted to two-dimensional plane of movement and must time their attacks, blocks and special moves to defeat their opponent. Most fighting games revolve around brawling and exaggerated versions of multiple martial arts, but some titles allow characters use of melee or even ranged weapons (e.g. Soulcalibur).

1.1.5. Asymmetric character design

Asymmetry in context of video game character design refers to the player or players having different abilities from the start of a match. In modern approach we assume video game-style asymmetry also encapsulates the possibility of choosing one of asymmetrical characters or forces (i.e. playable race in strategy games) before the match begins [figure 1.2]. A perfect counterexample of asymmetrical game would be chess, where players start the match with identical forces. There are several reasons as to why asymmetrical design became prevalent in modern gaming: it is used for adding more variety, supporting different play styles or giving players more options to customize their experience. [5]

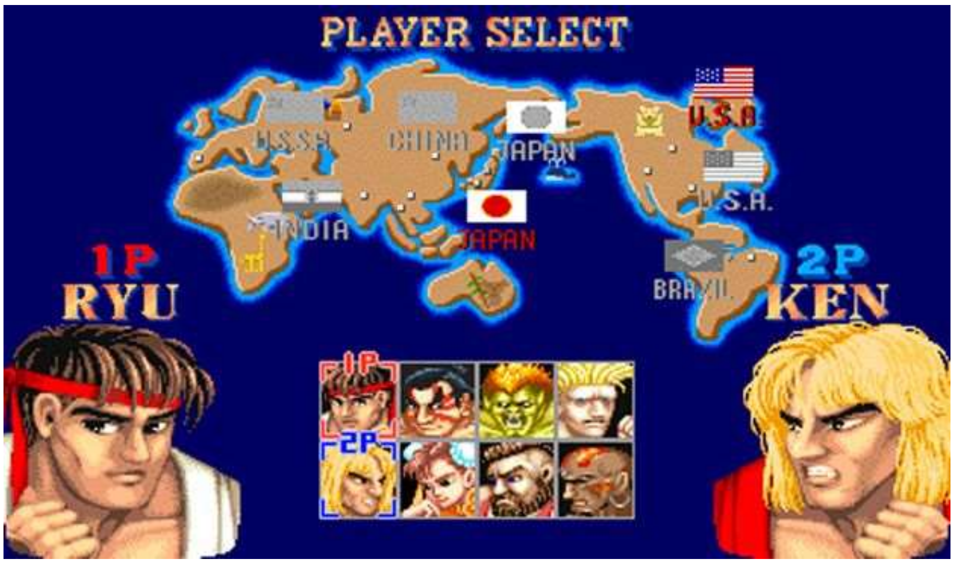


Figure 1.2. Character selection screen from “Street Fighter”, access 08.11.2019
source – own

1.1.6. Game Design Document

A game design document (often abbreviated GDD) is a detailed living software design document of the design for a video game [6]. It is mainly used to organize work and track progress on the project. Since production of video games is subject to frequent and high impact changes, GDD is vital tool for cooperation between artists, designers and programmers, and to ensure all parts of the project are up to date and in accordance with current vision of the project.

1.2. Thesis goal

The goal of the thesis is to create a project and implementation of two-dimensional fighting platformer game with local multiplayer and asymmetric character design. The game should support and accommodate for two to four players playing simultaneously on the same computer. Overall design decisions should support readability of what is happening on screen and maintain comfortable control scheme regardless of number of players.

The scope of the project includes creation of playable prototype and deployment on PC. Since Unity engine is used for this goal the game can later be deployed to numerous different platforms fairly easily. It will consist of designing appropriate and consistent theme as well as game rules and asymmetrical character classes, preparing graphical and audio layers, interface and animations.

The thesis consists of four chapters. First one is an introduction to the topic, analysis of several similar existing solutions from the last decade and explanation of chosen target platform. Second chapter describes preproduction phase of a video game and its differences when compared to a design process for functional or business applications produced to solve a particular real-world problem. Third chapter explains reasoning behind taken design choices and serves as a detailed Game Design Document for the project. Last chapter is used to show the implementation phase of the project, describing chosen tools, solutions, design patterns, encountered problems and their solving methods. Since modern video games require post-release maintenance and production of additional content to stay relevant in the market, ending of this chapter showcases possibilities for further development of the project.

1.3. Overview of existing solutions

1.3.1. Nidhogg

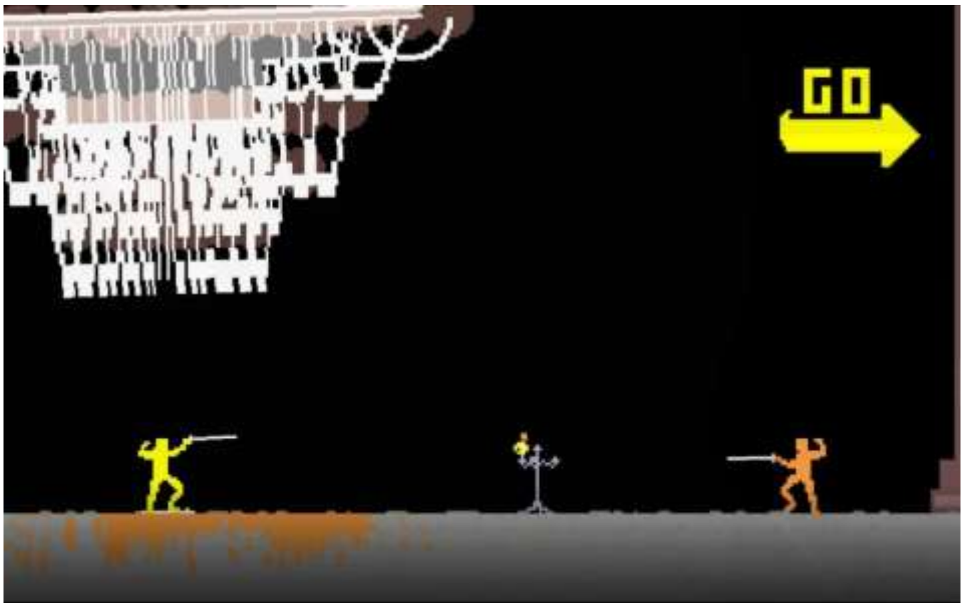


Figure 1.3. Gameplay screenshot from "Nidhogg", access 08.11.2019, source - https://store.steampowered.com/app/94400/Nidhogg/

Nidhogg [figure 1.3] was released in 2014 by a two-manned studio Messhof. It presents a tug-of-war type of gameplay, where players try to reach their opponent's side of the screen while protecting their own. It's presented with extremely simplified graphics, but the simple and fluid controls mixed with attention to details such as permanent blood stains make for a fast-paced and engaging gameplay.

Advantages:

- Easy control scheme
- High responsiveness
- Readability of game status
- Audiovisual feedback (special effects)
- Advanced movement and combat techniques (e.g. throwing weapon)

Drawbacks:

- Lack of variety, no character classes and only one weapon type (sword)
- Restriction of movement to horizontal plane only
- No additional game modes
- 2 players limit

1.3.2. Brawlhalla

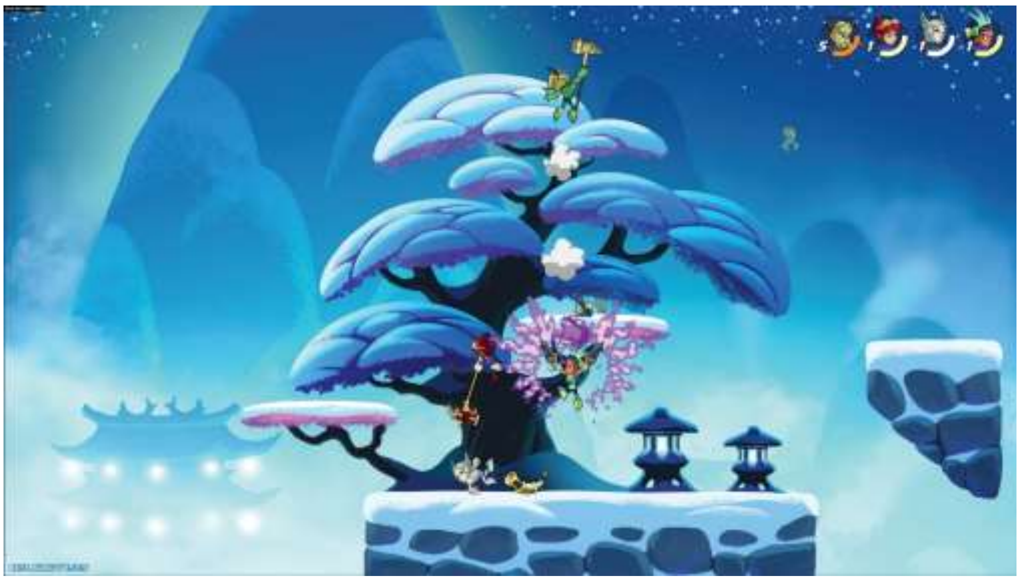


Figure 1.4. Gameplay screenshot from “Brawlhalla”, access 08.11.2019
source - https://www.gry-online.pl/S055.asp?ID=312337

Brawlhalla [figure 1.4] is a free-to-play game released in 2017 by Blue Mammoth Studio. It allows players to choose one of 44 playable characters with 2 of 12 available weapons assigned to them. Player characters can’t inflict damage directly, and must instead try to knock each other off the platforms.

Advantages:

- Art style and animations
- Large amount of unlockable characters
- Multiple game modes
- Up to 8 players in one match

Drawbacks:

- Unreadable user interface due to its size and positioning
- Long play time required to unlock new characters
- Little difference between characters except cosmetic
- Visual clutter with more characters on the screen
- Online multiplayer only

1.3.3. Rivals of Aether

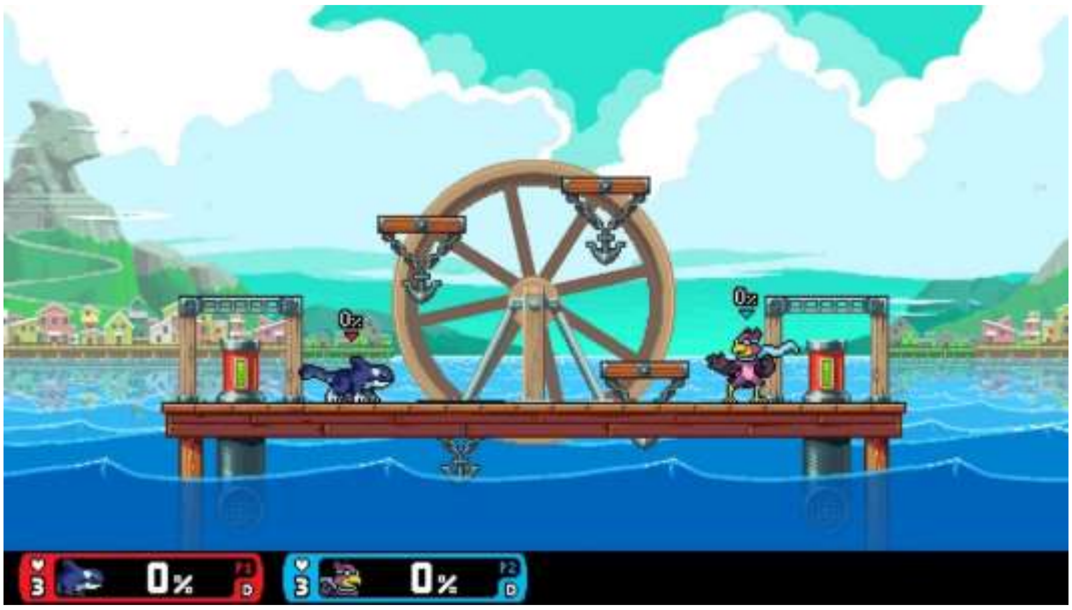


Figure 1.5. Gameplay screenshot from "Rivals of Aether", access 08.11.2019
source - https://store.steampowered.com/app/383980/Rivals_of_Aether/

Rivals of Aether [figure 1.5] was produced in 2017 by a single developer Don Fornace. It's simple graphics are offset with extremely polished gameplay. As in most fighting games released after Super Smash Bros, the objective is to throw other players off the platforms. Game offers 12 distinctive characters to choose from, and focuses mostly on 1 versus 1 fights, but has recently added a 2 versus 2 option.

Advantages:

- Fluid gameplay
- Fair amount of playable characters
- Unique moves for each character
- Interactable arena elements
- Has both local and online multiplayer

Drawbacks:

- Only two actual game modes: 1v1 or 2v2
- Very small gameplay area – entire arena fits in one screen
- Online multiplayer suffers heavily from latency problems
- Many characters are locked behind a paywall in addition to the base game price

1.3.4. Super Smash Bros Ultimate

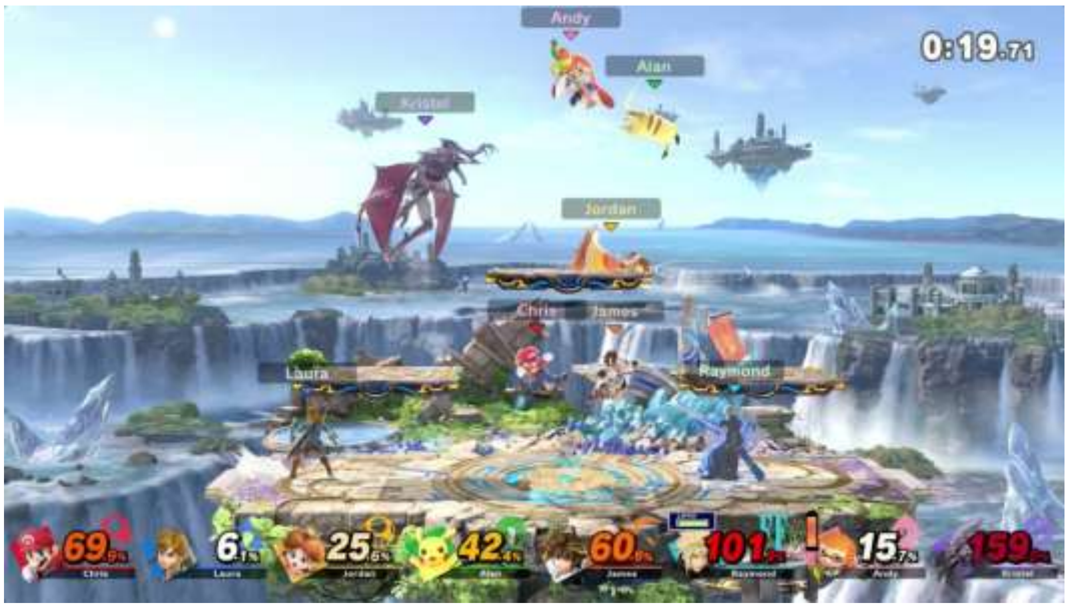


Figure 1.6. Gameplay screenshot from "Super Smash Bros Ultimate", access 08.11.2019
source - https://www.newgamenetwork.com/media/26769/super-smash-bros-ultimate/

Super Smash Bros Ultimate [figure 1.6] is the latest addition to publicly acclaimed Super Smash Bros series. It was released in 2018 by Namco Bandai Studios. It is considered the largest and most complete fighting platform game to date, and series as a whole set up most of the standards for the entire genre. Unfortunately, it is available exclusively on Nintendo Switch console.

Advantages:

- 1-8 players
- Has local and online multiplayer
- Many unique characters to unlock
- Advanced graphics
- Golden standard for the genre

Drawbacks:

- Problems with online matchmaking and latency
- Unlocking some characters is very difficult and time consuming
- Often unreadable and chaotic gameplay
- Its proven formula has been repeated over and over
- Only on Nintendo Switch

1.4. Target platform

Choice of the first target platform for the project was dictated by the availability of equipment and background check of existing solutions. Due to the nature of those systems, there exist noticeably more games offering local multiplayer on consoles rather than on PC. There are only a few similar solutions in the market right now, creating a niche for such product. More games of this type used to be available on PC in the past, which creates an opportunity to increase consumer interest by relying on nostalgy factor. According to market research, popularity of personal computers as a gaming device will remain stable in upcoming years, while at the same time net worth of the gaming industry is estimated to be increasing constantly [figure 1.3]. At the same time, developing this project with Unity engine makes it relatively easy to deploy it on other platforms without much additional effort, especially since it will already include built-in cross-platform controller support. Due to low-demanding graphics and simple control scheme of the project it could even be ported as a mobile version in the future of its life cycle.

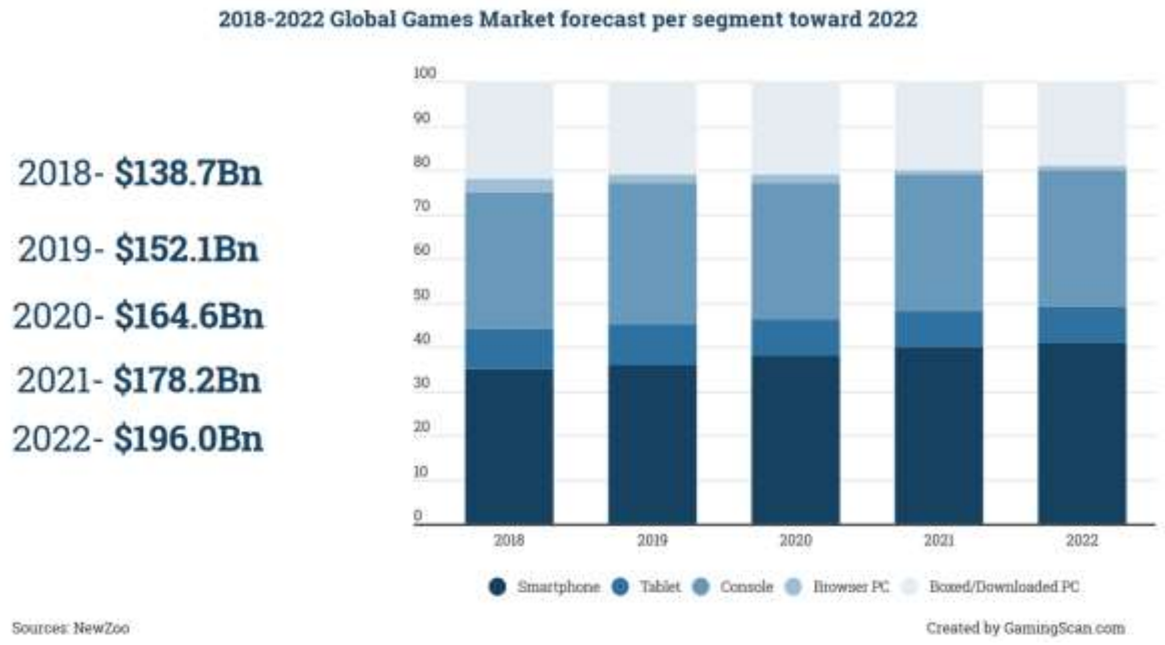


Figure 1.7. Value forecast of the gaming industry, access 08.11.2019
source - https://www.gamingscan.com/gaming-statistics/

2. Project

The development process of a video game is vastly different than that of a non-game project. First of all, most games are not created with the goal of solving any real world problem (with a notable exception of the therapeutic and simulator games niches), nor to improve an analytical or business process. Moreover, video game project development is a very fluid process, in which even the project concept is subject to drastic changes. Most video game projects start from a general concept or idea, and most functionalities, mechanical and gameplay details are defined and adjusted later in the production. In a ser testing and verifying which parts of the project are fun to engage with. Because of this, creating a list of functional requirements as a base on which a video game project will be built is not an appropriate approach.

The other major difference between a video game and a 'regular' application is the internal project structure. According to research [7], over 50% of video game project files are labeled as 'multimedia', while non-game projects on average consist of such files in less than 5%. Because of use of video game engines, the amount of code and documentation in video game projects is also substantially lower than in other applications [figure 2.1].

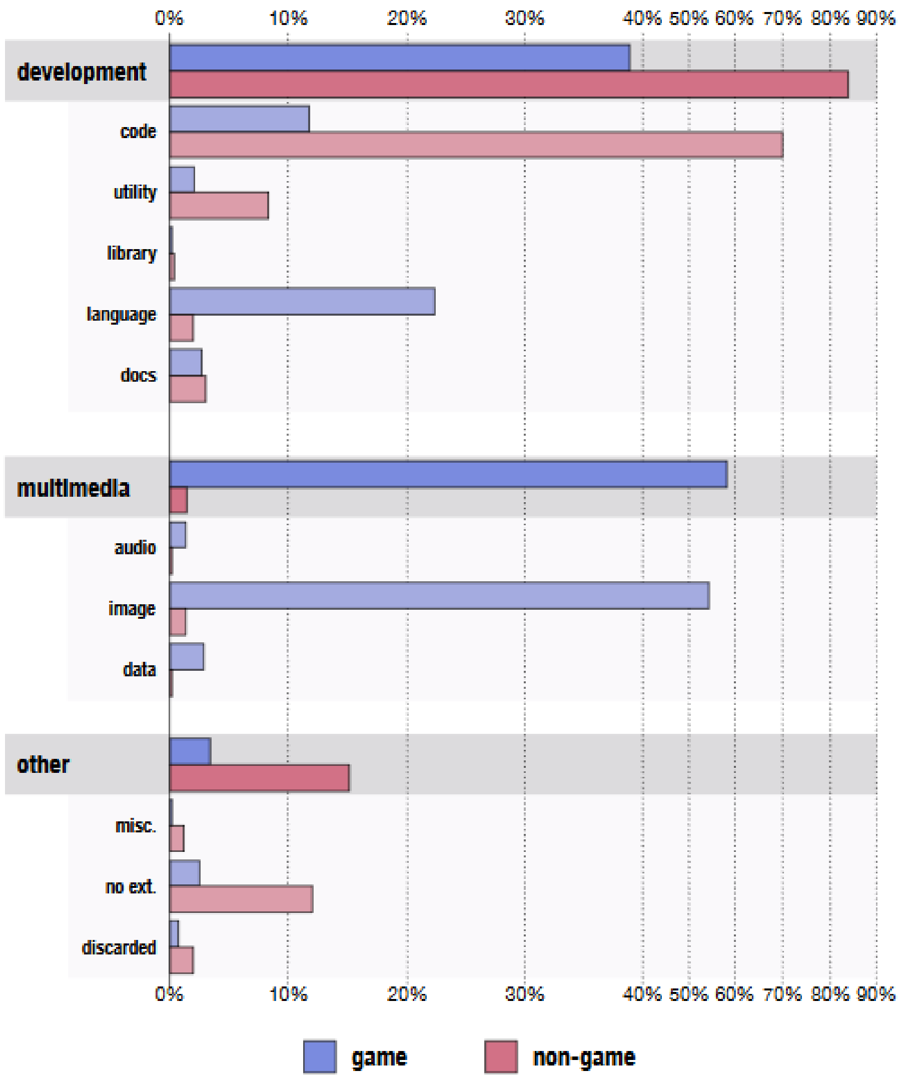


Figure 2.1. Distribution of project files in categories, access 08.11.2019
source - https://dibt.unimol.it/staff/fpalomba/documents/C32.pdf

Because of these structural differences, video game development usually requires a coordinated effort of artists and specialists from various fields. For a smaller, independent game developers, this means that they must either be adept in a much broader range of skills, or consider outsourcing some of the work to keep the project manageable.

2.1. Main concept

The game is a 2D platformer, arena type fighting game which allows two to four players to play simultaneously on the same device. Players will be able to choose their character's class and customize it before a match. Character classes are completely different from each other, and have distinct powers and modes of locomotion. The players should be able to change the match rules, such as team allegiance or characters health value. Players fight each other on enclosed arenas, gaining and losing points and competing for set amount of them which leads to victory. The arena will include special, recharging power-up spots with randomly assigned effects. Players can compete to capture those spots to transform the arena and/or give the capturing player an advantage. Furthermore, the game should have simple and responsive controls and be compatible with external controllers such as game pads.

2.2. Business analysis

Video game of this genre are fairly popular and prevalent on most consoles that allow the use of multiple controllers, but not many of them were released on personal computers within last few years. From those available on PC, even less allow for a local multiplayer gameplay. For those reasons, I believe there is a niche on the market which this project aims to fill. It's a small and simple game, which means it does not cost much to produce compared to most large budget titles. The comparison of release prices of similar titles on Steam platform within last 5 years was performed [table 2.1]:

Table 2.1. Fighting games' release price comparision, access 08.11.2019
source – Steam

Game	**Release price**
Nidhogg (2014)	€9.99
Nidhogg II (2017)	€14.99
Brawlhalla (2017)	Free to play
Brawlhalla - character pack	€8.99
Rivals of Aether (2017)	€14.99

In conclusion, as a quick, fun game to play occasionally with a group of friends, the project should do well within a low to medium price range for an independent game of this type, between €5-15.

3. Design

3.1. Main theme

The idea for main theme of the project came from several complementary conclusions. Firstly, for a fighting game, theme does inevitably have to revolve around conflict – albeit it does not have to be a physical one. Moreover, abstract depiction of non-physical conflict might help differentiate this game from others in the genre. Most fighting games tend to use either fantasy (Nidhogg, Rivals of Aether), or multiverse (Brawlhalla, Super Smash Bros) setting. The latter one works well for introducing many diverse characters from very different environments, but undermines the coherence of the audiovisual design. Fantasy setting might help with establishing a game world with consistent style and mythology, but alas has been overused due to ease of creating conflict (e.g. good versus evil, cat kingdom versus dog empire) and introducing diverse character powers by describing them as magic.

With these observations in mind, I have decided to create main theme for my project drawing inspiration from the subject closest known to me, computer science. Hence, main theme of my game revolves around the topic of computer security: selection of virtual entities – computer viruses, hackers and security software battle to take control over a host machine. The fight in this setting is symbolic and instead of physical combat, represents the act of digital warfare – exploiting glitches, using hacking and security tools, revoking access, etc.

3.2. Environment

The audiovisual design for environment in my project has to represent a digital 'inside' of the computer. Because of the nostalgic arcade gameplay basis for this type of game, I have decided to stay with retro-futuristic vision of cyberspace, as could be seen in 1982 movie Tron [figure 3.1] or hacking sequences from cyberpunk themed Shadowrun games [figure 3.2]. This decision would allow me to stay consistent with the main theme of the project, while not being overwhelming in required workload for me as a single developer.

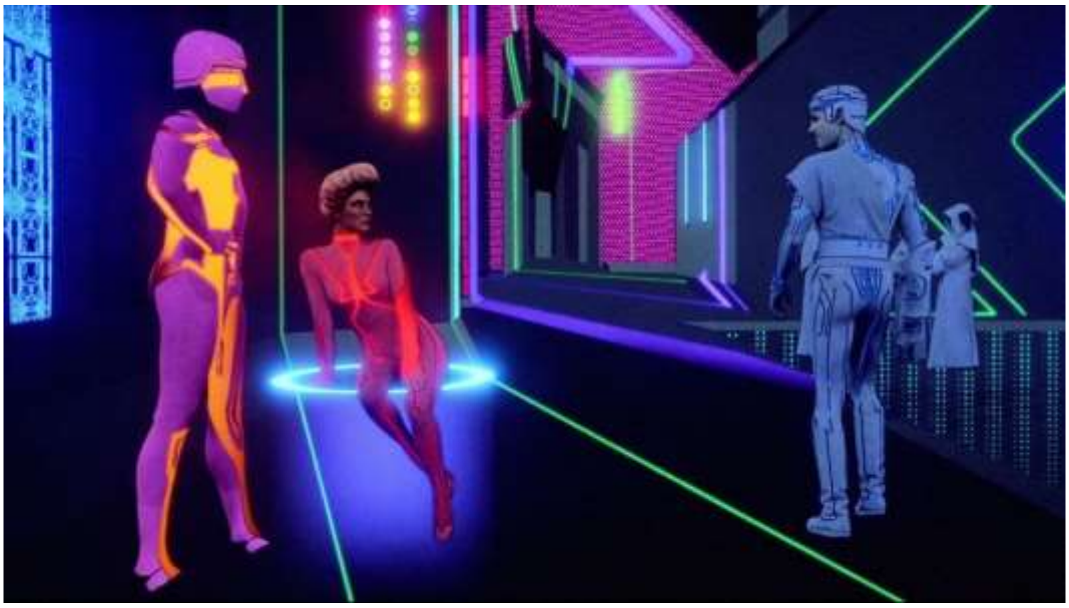


Figure 3.1. Screen from movie "Tron", access 08.11.2019
source - http://www.qwipster.net/2018/08/30/tron-1982/

Figure 3.2. Screenshot from "Shadowrun: Dragonfall", access 08.11.2019
source - https://store.steampowered.com/app/300550/

3.3. Character classes

Because of the small scope of the project, I decided an optimal number of character classes would be three – this would allow them to be as different from each other as possible, and would fulfill a well-known triad of archetypes – a warrior, mage and rogue. Additional character classes could still be added post-release in the further development cycle of the project, but to design them as distinctly and asymmetrically as possible, I have chosen to design exactly three main characters for the base game. Visually, monochrome design with bright white "energy" color will allow for easy player color customization in later development stages.

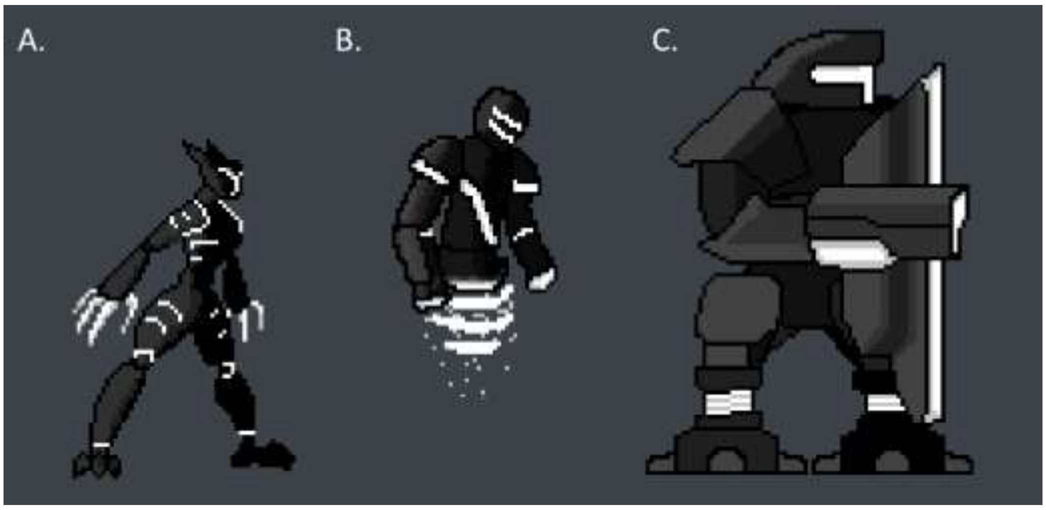


Figure 3.3. Main character classes concept art, source - own

3.3.1. Glitch

Glitch [figure 3.3 A] fits an archetype of the rogue. It is a fast-paced, melee-oriented character with teleportation ability, high jumps and quick claw attacks. As a balancing factor, it has less health than any other class, but can escape danger through his faster movement speed and teleportation ability. Lore-wise, Glitch represents a malicious actor such as a hacker or computer virus, spreading quickly and violently through the cyberspace, using glitches in the system (teleport ability) to circumvent security measures and gain access to restricted data (move through platforms and avoid attacks). Its abilities are named after various breaches in security and hacking methods:

Brute Force – fast melee attack, use claws to hit targets in quick succession. Low energy drain.
DDoS – charged melee attack, hold attack button to prepare a devastating attack launching target in the air. High damage, very high energy drain.
Backdoor – teleport in a chosen direction or momentarily disappear from the system to avoid damage. Can be used to move through obstacles. Moderate energy drain.
Injection – move through enemy entity and destabilize its integrity, dealing damage. High energy drain. Doesn't deal damage if energy requirement is not met.

3.3.2. Hunter

Hunter [figure 3.3 B] is the character fitting the archetype of a mage – has medium health pool and movement speed, and focuses on attacking enemies from range. Hunter uses unique movement method – flying, in a manner similar to a jetpack. Hunter is meant to represent an avatar of human user, or system wizard – he has more privileges in the system than other characters (flying), and must manually find and resolve problems within the system (aim with ranged weapon). Hunter's abilities are named after tools and methods which can be used by a system administrator:

Defragment – send a projectile moving in a straight line in front of the Hunter, dealing damage to enemy entities on hit. Moderate energy drain.
Check Disk – area denial tool - throw a floating disc functioning as a proximity mine. Throwing any new disc detonates previous one. High energy drain.
Scan – shoot a homing projectile which will move in a straight line towards the first enemy entity within its line of sight. Very high energy drain
Superuser – use your privilege within the system to move around without restrictions. Allows Hunter to fly in a manner similar to a jetpack. Very low energy drain.

3.3.3. Warden

Warden [figure 3.3 C] class fulfills the role of a warrior in the game. It's slow movement and high health pool represents it's heavy armor – security measures. Due to its weight, it can only move vertically using rocket thrusters in his legs – which consumes energy. Warden wields a shield, which makes him invulnerable to frontal attacks as long as its energy bar is not empty, as well as a short-ranged weapon, similar to a flamethrower or a shotgun, to deal with intruders when they get into its line of sight. Warden is designed to represent a security system, such as antivirus software, and has abilities named after cyber security measures:

Shredder – fire primary weapon, damaging every enemy entity caught within a cone of fire in front of the Warden. Moderate energy drain.
Safe Mode – raise shield, creating an impenetrable barrier protecting the Warden from incoming damage from all directions for a short while. Very high energy drain.
Firewall – charge thrusters in wardens legs to perform a propelled upward thrust, allowing him to move vertically and leaving behind a blazing trail for a few seconds, which will damage any enemy entities on contact. High energy drain.
Disinfect – air attack, use after propelling Warden upwards to fall down with great force, releasing a repelling shockwave and killing any enemy caught directly under Warden's feet instantly. Low energy drain.

3.4. Power-ups

Basic gameplay loop of a fighting game doesn't offer much in the way of variety, and since the game is being designed as a source of friendly entertainment and not an esport tournament platform, it would strongly benefit from introducing an element of random chance and more player interaction into the match. To balance the randomness of the power-ups in this game, I am designing them in the form of temporary control points, so that every player gets enough time and a chance of taking the bonus for themselves (instead of it being based on whichever character was closest to the power-up at the moment of its appearance). Player must stay within close proximity to the control point for a given interval of time (about 10 seconds) without it being conquested by other players [figure 3.4 a,b,d]. If other player manages to interrupt other player's capture, they will start 'undoing' first player's work [figure 3.4 c] and returning point to its original state [figure 3.4 a], after which conquesting player can capture the control point for themselves. After a successful capture of the control point, power-up will be awarded to the player for a given amount of time, or until it's resources are depleted. After that, the control point will enter a recharge period of 10 seconds [figure 3.4 e,f,g,h], after completion of which it will randomly assign itself a new type of power-up and become available for capture again, thus repeating the cycle and staying relevant for the rest of the match.

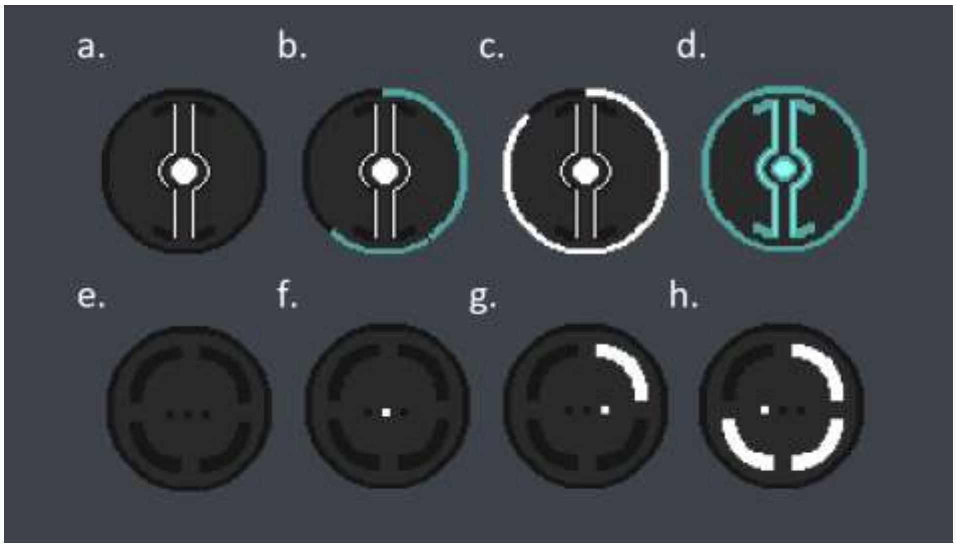


Figure 3.4. Life cycle of a control point, source - own

At the start of the game, all control points start in recharge mode and are assigned a random power-up type. Power-ups stay in white color while not being actively captured, and will change to player character's energy color during the capture and after being conquested. In the base version of the game, there will be 5 thematically fitting power-ups that can be assigned to an active control point:

3.4.1. Packet Tracer

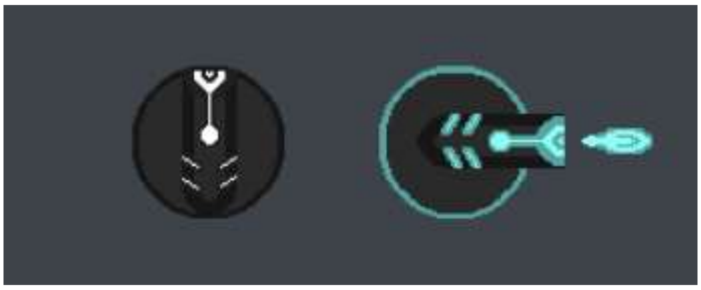


Figure 3.5. “Packet Tracer” power-up, source – own

Packet Tracer [figure 3.5] is an offensive power-up type. After capturing control point with this power-up, sentry gun will be activated at its position. Sentry will wait in idle state, rotating its barrel part slightly, until any enemy player enters it's line of sight within a given detection range. After spotting an opponent, Packet Tracer will turn its barrel in his direction and fire a volley of three projectiles with limited homing ability. Projectiles can be avoided by moving quickly out of their way and leading them to hit into terrain, otherwise will damage target if he's caught off guard. Any kill achieved by this sentry is treated as capturing player's kill for score calculation. Once activated, sentry turns off and capture point enters recharge mode, giving another player option to be captured, otherwise will stay in alert mode for 30 seconds before recharging. Implementation of this timer is designed to prevent ‘camping’ strategy, where player would avoid entering combat and wait at an advantageous position near his activated sentry gun, which would slow down the match and lead to a less exciting gameplay.

3.4.2. Overclock

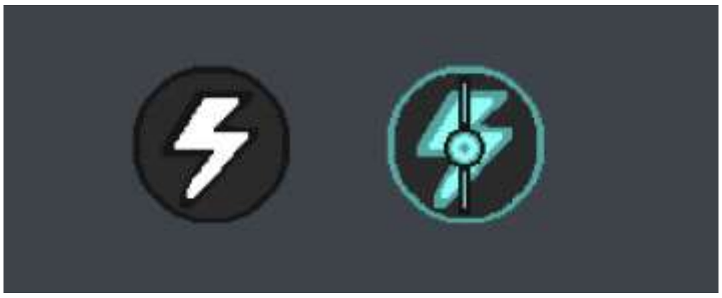


Figure 3.6. “Overclock” power-up, source – own

Overclock [figure 3.6] is a high-risk, high-reward power-up type. Once captured, its arm starts rotating, generating an energy refilling zone around it, marked by a ring of particles. It will increase energy regeneration rate for all player within its range. Energy regeneration is increased by 50% for everyone except the capturing player, whose regeneration rate is doubled. Because it affects opponents positively, players are required to analyze their situation and decide whether activating this power-up will be worth giving enemies additional energy as well. Standing their ground within the Overclock zone will give any player an advantage, but is also very likely to attract other players attention. Once captured, this power-up will activate automatically and keep working for 20 seconds before deactivating and entering recharge mode.

3.4.3. Syscheck

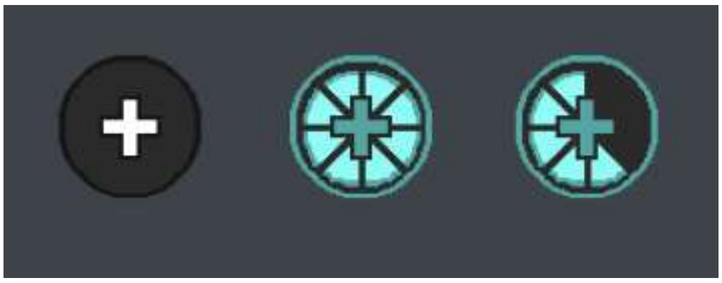


Figure 3.7. “Syscheck” power-up, source – own

Syscheck [figure 3.7] is a support power-up type. After capture, control point with this power-up will heal one health point of any wounded player in its immediate proximity up to eight times, with three second delay between each heal. It will prioritize healing the capturing player’s character over others. As with System Tracer, to prevent a camping situation, standing on the activated Syscheck power-up with full health will still drain it’s charges without giving any benefit (except preventing its use from other players). After activation, it will remain active until all its charges are depleted, after which control point will enter recharge mode.

3.4.4. Quarantine

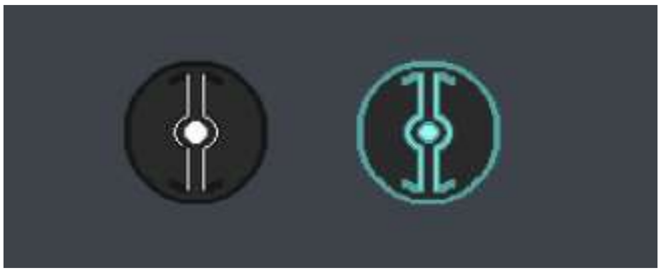


Figure 3.8. “Quarantine” power-up, source – own

Quarantine is a defensive power-up type. After capturing a point with this power-up, a protective zone will appear around it, blocking any projectile that tries to enter it, and preventing use of powers by draining all energy from everyone within its perimeter. This power-up is designed as a last resort getaway, giving player a safe zone and a moment to breathe, it can also be used to get some space between him and his enemies. With smart timing, it could even be used to tactically drain enemy’s energy. It’s powerful defensive capabilities are balanced with a drawback of leaving the player vulnerable while his energy regenerates. After capture, Quarantine automatically activates and raises its protective barrier for 7 seconds, after which control point enters a recharge mode.

3.4.5. Short Circuit

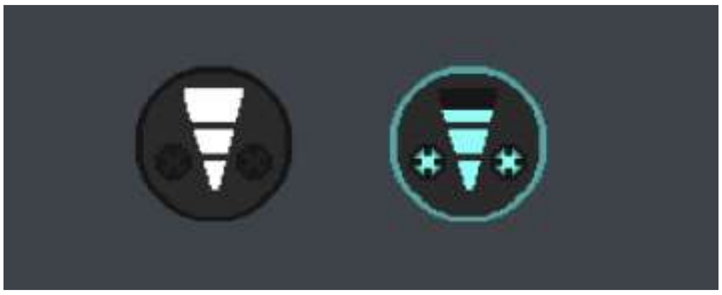


Figure 3.9. “Short Circuit” power-up, source – own

Short Circuit [figure 3.9] is an area denial power-up type. Once captured, it emits particles reminiscing of electrical discharges, and will emit an impulse of electricity around it every second while the charge meter in its middle depletes, displaying information how much longer it will last. Players damaged by the Short Circuit will have some of their energy bar restored, creating possibility for interesting trade-off decisions. Capturing player won’t receive damage from the first two impulses. This power-up’s main role is to create a dangerous terrain, which with proper tactics can be used in either offensive or defensive manner. Adding a positive effect on top of a negative one allows for further increase of possible creative uses. Kills achieved with this power-up do award any additional score. Short Circuit activates automatically after being captured and lasts 10 seconds before entering a recharge mode.

3.5. Game rules

3.5.1. Health

To differentiate this project from fighting games based on Super Smash Bros formula, as well as to make player actions and attacks feel more consequential, I have decided to introduce character health points into the game. Since basic premise of gameplay will involve a tug-of-war scoring system, small, single digit amount of health points will help with achieving a balanced pacing of the game. Making characters die from a single hit would reduce variety in character classes and make fights feel frantic, while giving characters too much health would make attacks feel weak and inconsequential. All character classes start with a different health pool: 3 health points for Glitch, 4 for Hunter and 5 for Warden. Combined with damage mitigation skills and techniques described in this chapter, this should introduce enough difference between characters to make them naturally drift towards different playstyles: Warden should not be afraid to trade hits with other characters, but on the other side his slower movement makes him prone to flanking maneuvers. To make the health system accessible and clearly defined, most attack types of every class deal exactly one point of damage to the target. When character’s health pool reaches zero, they are defeated and must wait a set cooldown period before being respawned into the match. Cooldown period is intentionally designed to be very short, as to not interrupt the flow of the game. Drawing inspiration from the Nidhogg blood stains mechanic [figure 1.3, p.3], defeated character will leave a corpse behind, creating a visual measure of tracking the match progression as more corpses will start filling the arena.

3.5.2. Energy

Using most characters' powers will drain their energy, which is introduced as a resource-management mechanic. Unnecessary repeated use of powers will leave a player character defenseless while his energy regenerates, requiring a more thoughtful input pattern from the player. All characters start with the same amount of energy, and their energy will automatically regenerate at identical rate when not using any powers. Movement does not restrict energy regeneration, while use of any energy draining power applies a short cooldown period preventing further regeneration – this cooldown is designed to reduce viability of tactics based on exploiting regeneration system by repeatedly using power with lowest energy cost. Different powers will drain different amounts of energy, some of them will require certain amount of energy to even activate while others will drain energy continuously, with their duration dependent on the amount of energy available to the player. Upon respawning, character's energy pool is fully replenished to give them chance to retaliate after being defeated. Some players might not find such strategic resource-management elements enjoyable, for those players an unlimited energy mode is available to choose before the start of the match. In this mode, using powers will not drain characters' energy.

3.5.3. Score system

Main objective of the match is to be the first player or team to gain 5 points. A point is awarded for the team upon defeating another player, but a point is also lost when controlled character dies. Environmental deaths will not award any points, but some power-ups are treated as controlling player's kills and will increase their score. 5 points objective might seem easy to achieve, but because of point loss mechanic it works more in a back and forth manner, leading to high tension moments whenever one of the players gets close to winning match, and allows losing players to make a comeback by simultaneously lowering their opponents score while increasing their own. In a match with more than two players, player with highest score will also attract other player's attention, making that final push even more difficult.

3.5.4. Teams

In accordance with asymmetric design, players are given the freedom to form their own teams however they want, based on their chosen character color. Players with the same color are automatically assigned to the same team, will share the same score counter, and will not be able to damage each other. This setup will allow for more match setups, for example new players can try to take on a more experienced player together, or two versus two match can be performed. Teams aren't required to have the same combat potential, but additional options will allow players to tweak game balance however they want. If two players with the same character class will join the same team, their characters will be differentiated with a different hue of the chosen team color.

3.6. Control scheme

The main focus of this project is delivering a seamless, comfortable gameplay experience for up to four players playing simultaneously on the same computer. For that reason, control scheme must be thoughtfully planned to logistically leave enough room for each player, while at the same time not relying solely on external controllers, should users want to play just using their computer. Control scheme for the project was designed after classic NES controller [figure 3.10], in accordance with implemented retro design approach. It was also chosen because of the innate limitation of computer keyboards, most of which won't track more than 5 simultaneous key presses, which could lead to problems with input reliability. Control buttons for each character constitute of four dimensional inputs and two action buttons, A and B, which can be combined to increase amount of possible player inputs.

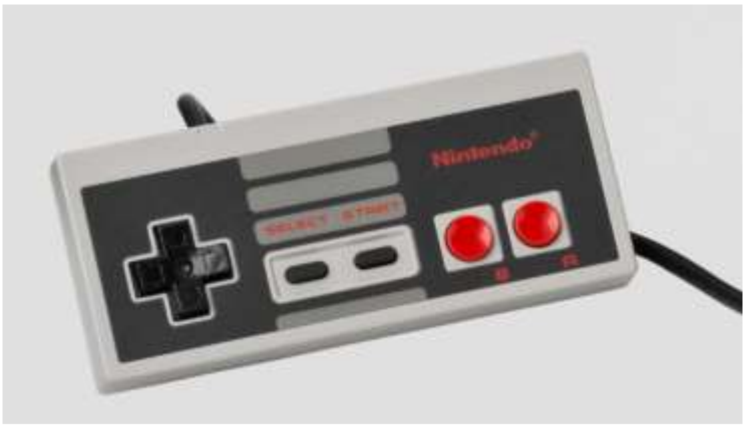


Figure 3.10. Nintendo Entertainment System controller, access 08.11.2019
source - https://en.wikipedia.org/wiki/Nintendo_Entertainment_System

3.6.1. Keyboard

Because of limited space around one keyboard and aforementioned hardware limit for simultaneous key presses, I have decided to limit amount of players who can use it to control their characters to two. From then, priority task was to separate players input keys as far as possible from each other without compromising their comfort (i.e. crowding each player's input area). Optimized solution [figure 3.11] required taking advance of numerical keypad present on the right-hand side of most modern keyboards. This solution would fall short for some keyboard types, such as many present in portable computers (laptops), therefore my control scheme had to include an alternative key mapping choice for player using the right-hand side controls for increased accessibility.

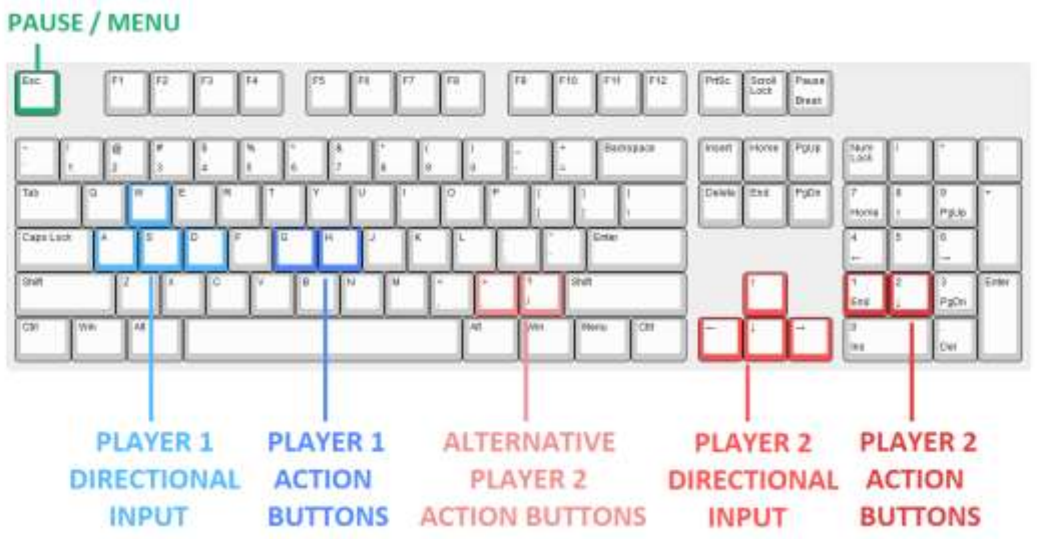

PAUSE / MENU
PLAYER 1 DIRECTIONAL INPUT
PLAYER 1 ACTION BUTTONS
ALTERNATIVE PLAYER 2 ACTION BUTTONS
PLAYER 2 DIRECTIONAL INPUT
PLAYER 2 ACTION BUTTONS

Figure 3.11. Key mapping for the keyboard, source - own

3.6.2. Gamepad

Designing control scheme for gamepad controllers [figure 3.12] was much easier task – all I had to do was follow the basic NES controller [figure 3.10] layout, to maximize amount of controller types supported by the product. With use of proper drivers, project should offer support for as many different controllers as possible. Because many players are accommodated to using left analog stick for character movement, game will map it as directional key presses after detecting a modern type of controller, giving players an alternative solution. Controllers will also allow menu interaction by assigning "Action A" as a confirm button and "Action B" as a return button, to allow complete interaction with the game without the need to get up in a situation where all players are sitting further away from the screen.

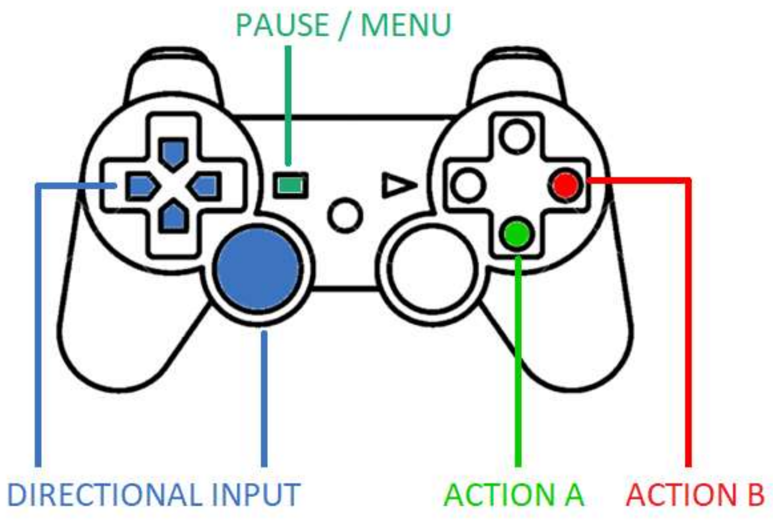


Figure 3.12. Key mapping for the controller, source - own

3.6.3. Characters controls

Aim of asymmetric character design is to make them feel and control differently based on chosen class: for just basic movement, Glitch will utilize jumps, Hunter his ability of flight and Warden must take a moment to charge his boosters to perform a jump, or use a Firewall power by shooting at his feet, akin to the rocket-jumping technique introduced in 1996 by an Id Software game "Quake" [8]. I have prepared a comprehensive list of characters' actions based player inputs [table 3.1], to highlight differences in different classes control schemes, as well as a practical guide for the upcoming implementation phase.

Table 3.1. Control scheme for different character classes, source – own

Buttons	Glitch	Hunter	Warden
Left/right	Move	Move	Move
Up	Jump	Fly up (Superuser)	Aim up
Down	-	Fly down	-
A	Attack (Brute Force)	Shoot (Defragment)	Shoot (Shredder)
hold A	Charged Attack (DDoS)	Homing missile (Scan)	Charge jump
B	Disappear (Backdoor)	Throw mine (Check Disk)	Raise shield (front)
hold B			Barrier (Safe Mode)
A + Left/right	Dash attack (Brute Force)	Same as A	Shoot (Shredder)
A + Up	Same as A	Same as up and A	Shoot up (Shredder)
A + Down		Same as down and A	Firewall jump (Firewall)
B + Left/right	Teleport (Backdoor)	Same as B	Same as B
B + Up			Raise shield (up)
B + Down		Place mine (Check Disk)	Barrier (Safe Mode)

As evidenced, despite limited number of buttons, combining their presses and introducing distinction between button tap and hold (i.e. short and long presses) opens up noticeably more input options to the players.

3.7. User interface

Well-designed user interface is an essential part of any video game project. Even the most playable title can deter players' interest with poorly designed or lazy looking interface. Especially with game requiring split-second decisions such as this one, practical and readable UI is critical to assure user satisfaction with the final product. Most interface design in this subchapter was done with the Balsamiq Mockup 3 tool.

3.7.1. Main menu

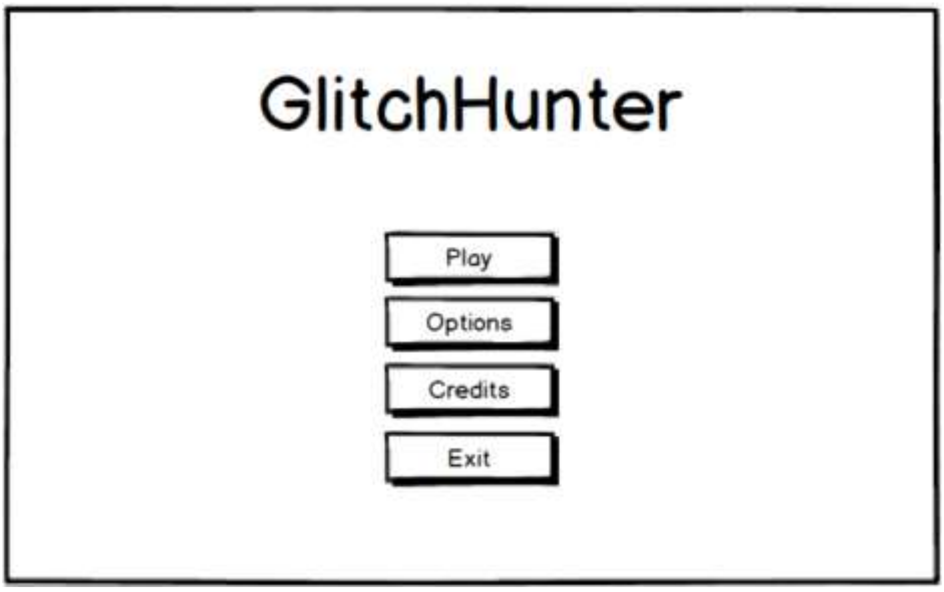


Figure 3.13. Mockup of the main menu, source - own

Main menu [figure 3.13] is the first thing players will see and interact with. It should be kept simple, but provide access to vital functions of the program. It is important to put the play button as the first position in the menu and exit as a last one to stay in accordance with video game standards. It is important to note that by giving users option to control menu with various input methods, we increase chance to mis-click elements, and therefore exit button should not close the program immediately, but display a prompt asking user to confirm their will to exit [9].

3.7.2. Match settings

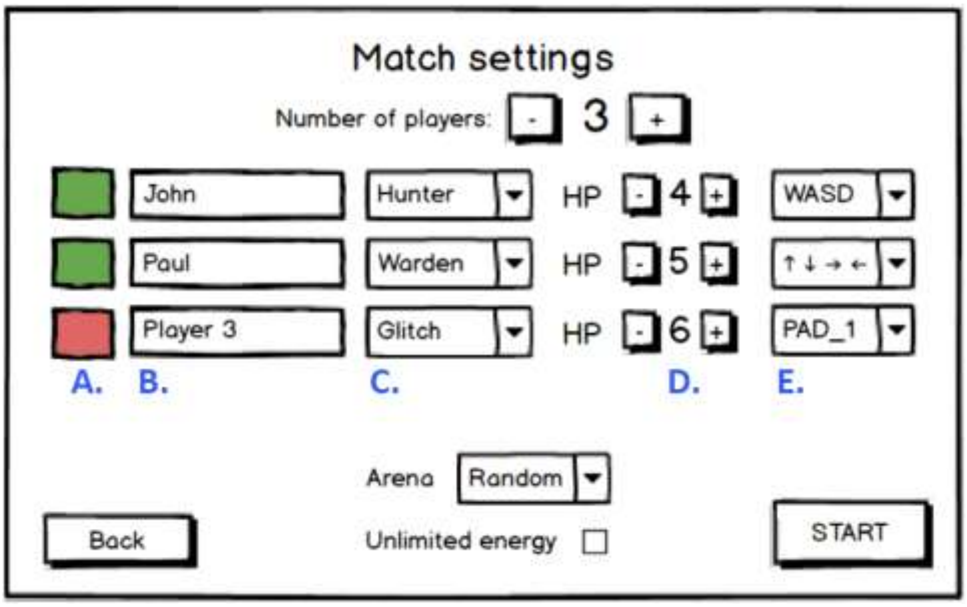


Figure 3.14. Mockup of the match settings menu, source - own

Match settings menu [figure 3.14] is displayed after pressing the Play button from the main menu. Set of buttons at the top of the screen allows players to increase or decrease number of participating players. Chosen amount of players (between 2 and 4) has their character options displayed in rows: players can pick their team color [A], enter character name [B], choose character class [C], modify amount of health points [D], and choose their preferred control method [E]. At the bottom of the menu, players can also choose their preferred arena type, and optional unlimited energy play mode. Start button will begin a match with chosen settings, but will remain inactive if there are less than two teams chosen.

3.7.3. Game options

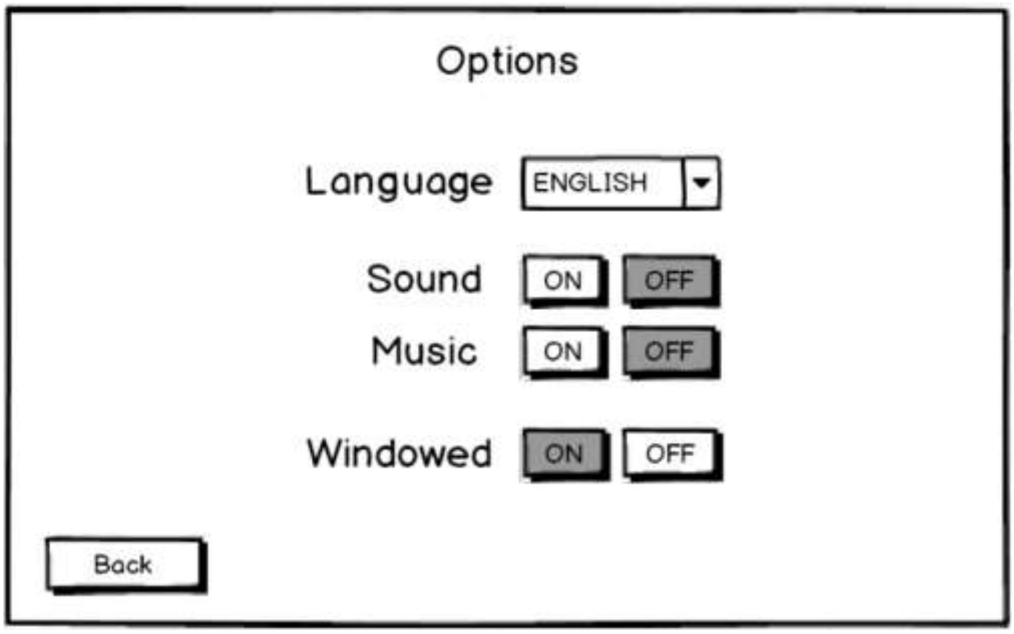


Figure 3.15. Mockup of the options menu, source - own

Due to the low-demanding graphical side of the project, it will be able to run on most computer setups without performance issues, therefore we can omit graphic settings in this menu. What we’re left with is language setting for translations, which can be done with low effort due to very low amount of written text shown to the player, basic audio settings for players who prefer choosing their own background music, and an option to view game either in full screen or windowed mode [figure 3.15].

3.7.4. Credits

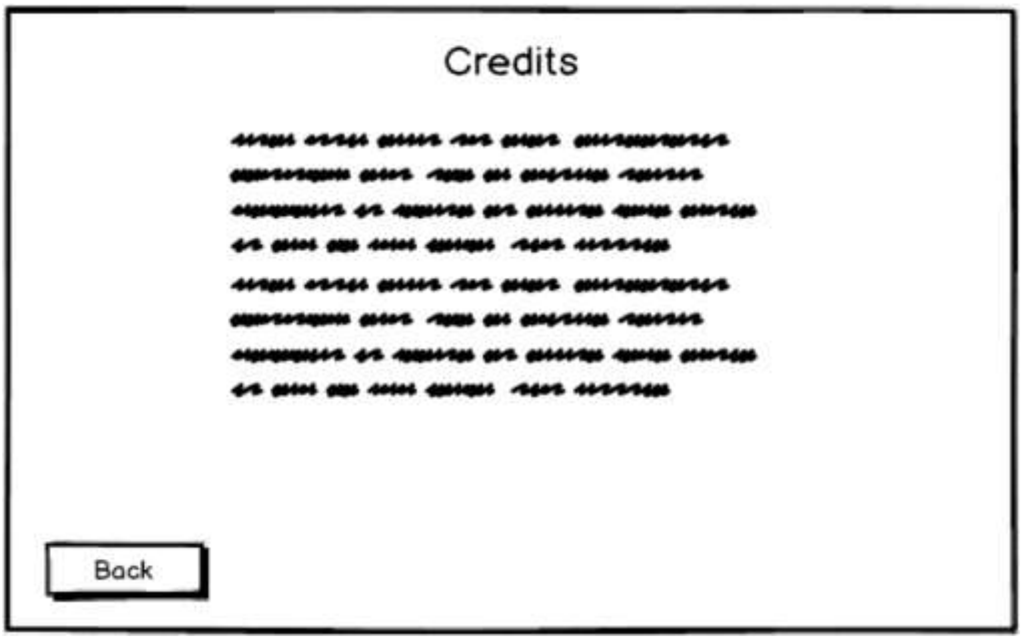


Figure 3.16. Mockup of the credits, source - own

Credits space [figure 3.16] is often skimmed over during development of small video game projects. Proper accreditation for all used art work and third party assets is however very important in any real world project we want to release. It is also a huge part of game development etiquette and can in some cases draw unwanted, negative attention to the developer when forgotten about.

3.7.5. Pause menu

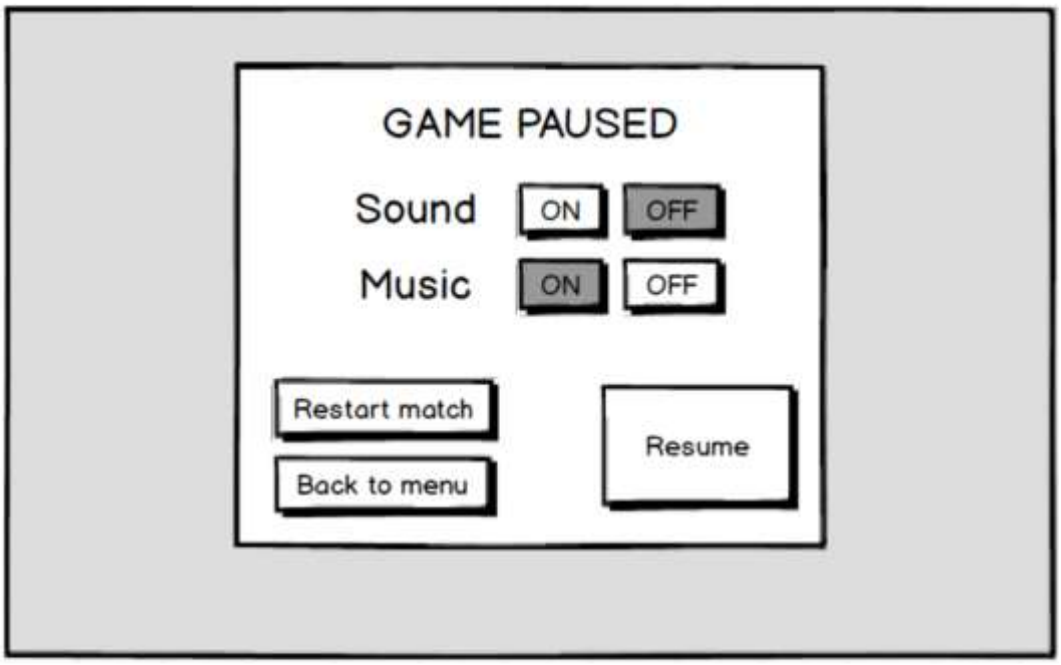


Figure 3.17. Mockup of the pause menu, source - own

Pause menu [figure 3.17] is the window that will be displayed whenever game is paused during gameplay. It gives players access to basic audio settings, as well as options to resume game, restart the match with current player settings, or return to main menu. Since the latter two options will end any ongoing match and any progress made by the players will be lost, a confirmation dialogue box should be displayed before allowing players to exit an ongoing match [9].

3.7.6. Status bar

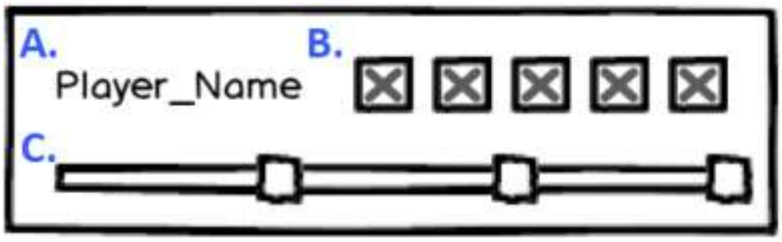


Figure 3.18. Mockup of the player status bar, source - own

Status bar [figure 3.18], also known as Heads Up Display (HUD), is the part of the interface that displays information about player character. For a fast-paced game with multiple players, my priority in its designing was it's readability. Status bar includes player name [A], current amount of health points [B], and energy bar [C]. Maximal health point amount in this project is low enough to allow displaying them separately, which makes their amount easier to gauge with peripheral vision, rather than requiring player to refocus their sight to read a numerical value. For the same reason, energy bar that visually fills up from the edge of the screen was preferred solution to the numerical display. Three notches on the energy bar light up when certain thresholds of energy are reached, further signalizing to the player what powers are available for their use at the moment.

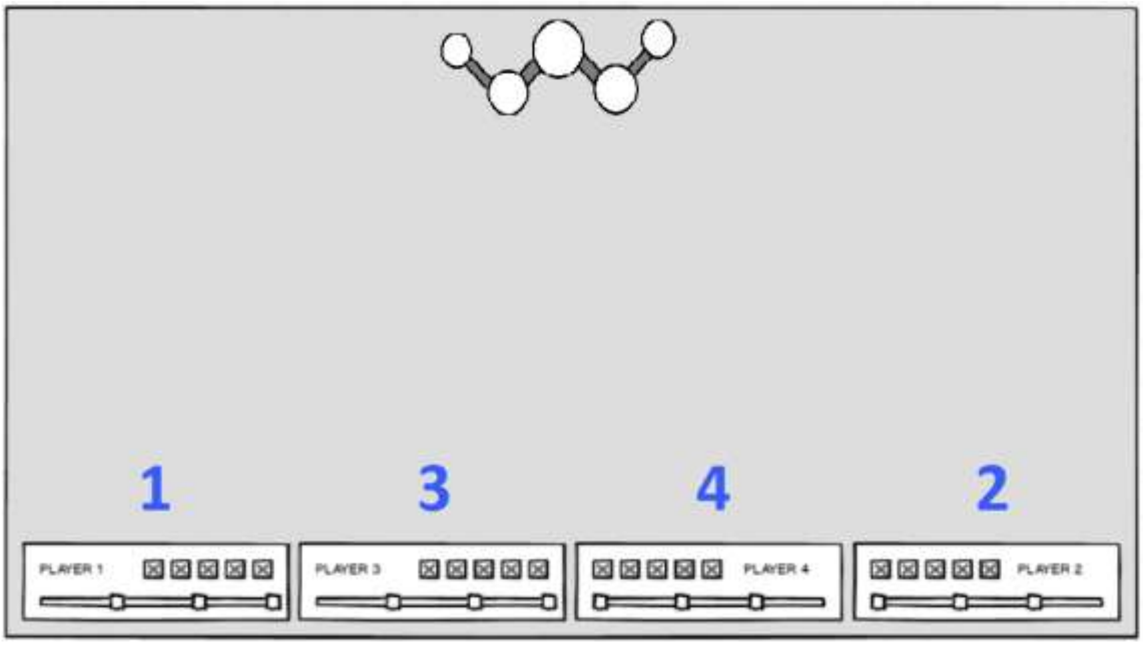


Figure 3.19. Positioning of the players' status bars on the screen, source - own

In fighting games, status bars are usually displayed on either top or bottom part of the screen, I have chosen the latter to leave top position for the score counter. Status bars that are closer to the right side of the screen are mirrored to introduce a pleasant symmetry to the interface and make it look more appealing by reducing its visual repetition.

Because a game match can include a varying number of players, proper order of introducing new status bars [figure 3.19] must be followed: if there are only two players, it makes more sense for their status bars to be on the opposite edges of the screen rather than displaying them on only one half of it.

3.7.7. Score counter

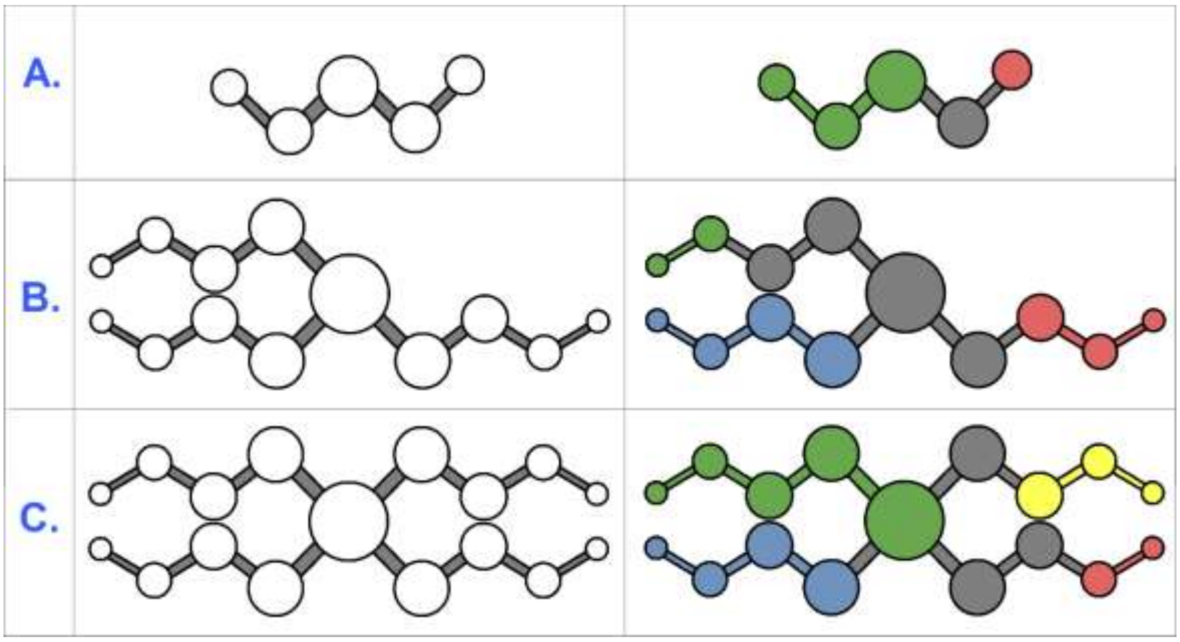


Figure 3.20. Mockup of the point counters, source - own

To better display the tug-of-war nature of the scoring system, I have designed a visual score counter, which fills up with the team color of scoring character [figure 3.20]. Thematically, it is meant to resemble nodes in a computer network, to which characters gain or lose access. Purpose of the characters is to gain control over local computer network, which is symbolized by completely filling a counter for two-team matches [A], or getting access to the central node on the three- [B] or four-team [C] counters. This score counter design fits well with the main theme of the project while also making it easy to quickly notice which players are closing to victory by just checking the dominant color on the counter. To make this visual clue more pronounced, nodes closer to the central node are larger than the further ones.

4. Implementation

4.1. Game engine

For the needs of this project, I have chosen the Unity engine in version 2018.3.5f1. Unity is a cross-platform game engine developed by Unity Technologies. After its initial release in 2005, it has seen continual support from its developers, extending its capabilities, optimization techniques and accessibility. As of 2018, the engine had been extended to support more than 25 platforms. The engine can be used to create three-dimensional, two-dimensional, virtual reality, and augmented reality games, as well as simulations and other experiences [10]. I have chosen unity due to its versatility and possibility of extending its functionalities with many readily available extensions and libraries. It is well suited for a two-dimensional game development, without being oversimplified as some 2D-only engines (e.g. Godot, RPG maker) tend to be. It’s biggest competitor, Unreal Engine, is focused mainly on development of three-dimensional games with high quality graphics, and therefore lacks many inbuilt 2D functionalities Unity offers. Unity Engine was written in C++ programming language, but uses C# as its default scripting tool.

4.2. Tools

Unity Engine itself offers a wide variety of built-in tools, from physics engines, special effects and animators to image and audio correction. I tried to use native engine functionalities wherever applicable, hence list of external tools turned out as fairly short.
To reduce additional costs of the project all tools used are either freeware, or were used in their free (community) version.

4.2.1. Visual Studio 2017

Microsoft Visual Studio is an integrated development environment (IDE) from Microsoft. It supports 36 different programming languages, including C# which is used for this project. Visual Studio 2017 comes preinstalled with Unity Engine as a default script editing software. Its extensive list of functionalities and built-in support for Unity functions and libraries make it a great tool when programming game logic.

4.2.2. GIMP 2.10

GNU Image Manipulation Program is a free and open-source raster graphics editor used for image retouching and editing, free-form drawing, converting between different image formats, and more specialized tasks [11]. It is used for creation and manipulation of 2D graphics used as assets in the project. It was favored over simpler graphic editors due to its advanced functionalities such as layering, grid snapping and transparency.

4.2.3. Audacity

Audacity is a free and open-source digital audio editor and recording application software. It was used for fine-tuning and manipulation of audio assets used in the project, as well as converting some of them from file types unsupported by Unity Engine.

4.2.4. Bfxr

Bfxr is a free custom sound effects generator. It was used for creation of audio assets for this project. Sound samples generated by this program are reminiscent of those from NES era games, making it a perfect candidate for the retro style chosen for this project.

4.3. Third party assets

The scope of any multimedia project rarely involved creation of every single asset by hand. Use of third party assets is a widely used practice and helps to greatly decrease development time of the project. Because of the relatively simple design I could create most of used assets by myself, and only resorted to third party ones where it did save a lot of time and was not vital to the main scope of the project.

4.3.1. InControl input manager

InControl is an input manager for Unity3D that standardizes input mappings across platforms for common controllers. It is written in C# and strives to make it easy to add cross-platform controller support to your game [12].
Since the scope of the project involves creating a video game, and not writing hardware driver adapters for many different controller brands, I have chosen to use this available third party solution. Most current version costs $40 at the time of writing, but the version I will be using is an outdated one, free and open source, which support has been discontinued by the developers. Nonetheless, it still served the needs of this project with flying colors.

InControl 1.4.4 by pbhogan, 2017
https://github.com/pbhogan/InControl [Access 06.05.2019]

4.3.2. Camera shake script

After conducting player tests, I needed a quick solution to implement an additional camera effect to improve the visual side of the project. This third party script is very simple, but was used to save time when introducing improvements in late stage of the project implementation.

CameraShake by ftvs, 2016
https://gist.github.com/ftvs/5822103 [Access 19.10.2019]

4.3.3. Graphics

The only graphical third party asset used is an animated background, rest of the graphics was created for the needs of the project. Chosen animation is free for commercial use.

Animated background – **Cyan Square** by FreeFootageParadise
https://www.youtube.com/watch?v=f8I7-Q09MRk [Access 31.01.2019]

4.3.4. Fonts

Since Unity engine does not come with a variety of inbuilt font styles, I had to choose and import one that fits the project. Website “1001 fonts” offers a lot of readily available font options, from which I have chosen one with simple 8bit design to further enhances retro-arcade style of the game. “Free for commercial use” license of the font ensures it won’t cause any copyright related issues after project deployment.

Notalot35 font by Allison James
https://www.1001fonts.com/notalot35-font.html [Access 31.01.2019]

4.3.5. Music

Creation of custom music score was not possible in the time and scope of the project, therefore I made use of available free soundtrack options for the needs of the project. Only files with Creative Commons 0 (public domain) or BY (free under author accreditation) licenses were used [13]. Tracks used:

Main menu - **Gamerstep Melody** by Clinthammer
https://freesound.org/people/Clinthammer/sounds/179523/ [Access 19.10.2019]

Combat music 1 – **Rhinoceros** by Kevin MacLeod
https://incompetech.filmmusic.io/song/4284-rhinoceros/ [Access 19.10.2019]

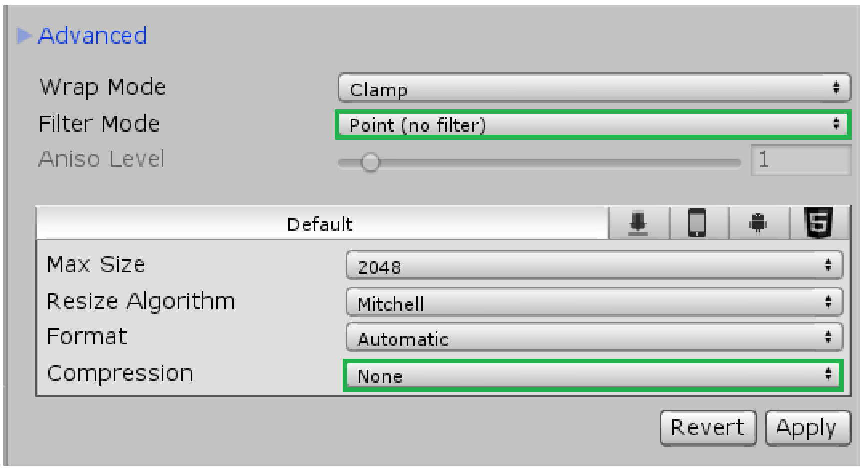
Combat music 2 – **Harmful or Fatal** by Kevin MacLeod
https://incompetech.filmmusic.io/song/3859-harmful-or-fatal/ [Access 19.10.2019]

Combat music 3 – **In a Heartbeat** by Kevin MacLeod
https://incompetech.filmmusic.io/song/3907-in-a-heartbeat/ [Access 19.10.2019]

4.4. Graphics

4.4.1. Sprites

Sprites used in the project were prepared using GIMP 2.10 and exported to a .png format, which is useful due to its higher quality and transparency support lacking in .jpeg, and smaller memory use than .bmp. At the same time, scalable vector graphics were unfitting for the needs of the project, due to very small size of pixel art assets and no need to smoothen up their edges – in fact, to keep pixel art sprites from being anti-aliased to keep their sharp edges, change of default Unity's advanced sprite options [figure 4.1] was in order: Filter Mode had to be set to Point (no filter), as well as Compression level to None. This prevented graphics from rendering as blurry [figure 4.2].



Wrap Mode: Clamp
Filter Mode: Point (no filter)
Aniso Level: 1

Max Size: 2048
Resize Algorithm: Mitchell
Format: Automatic
Compression: None

Figure 4.1. Advanced Unity sprite settings, source – own

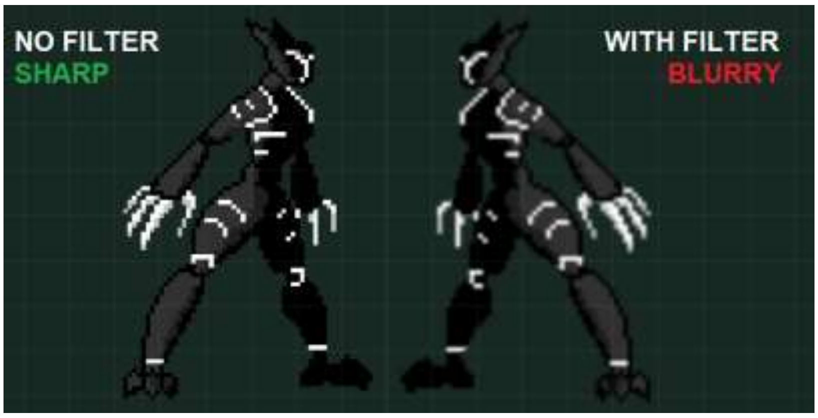
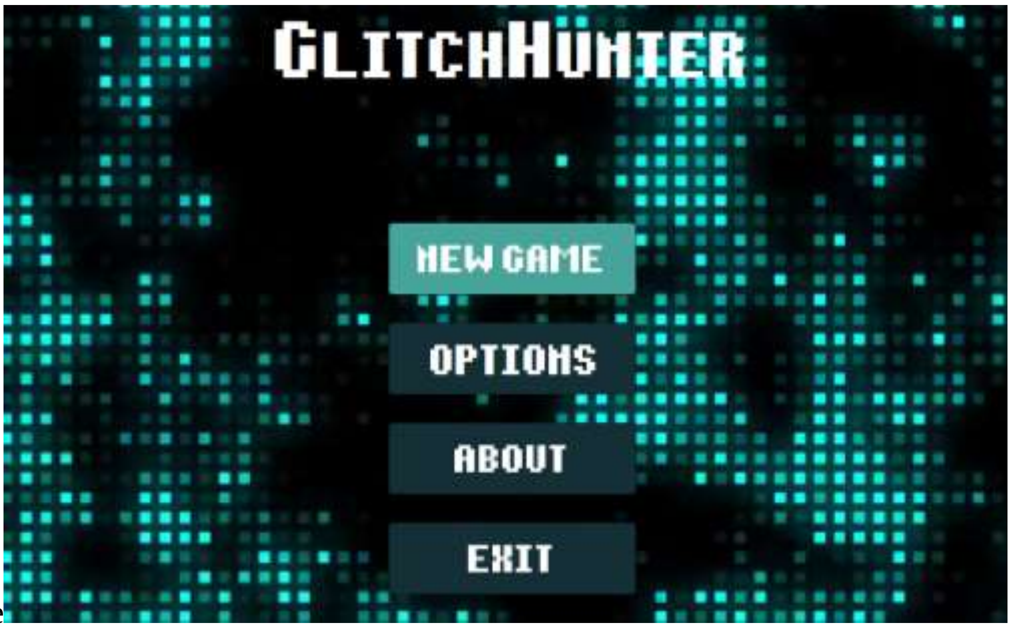


Figure 4.2. Sprite sharpness difference, source - own

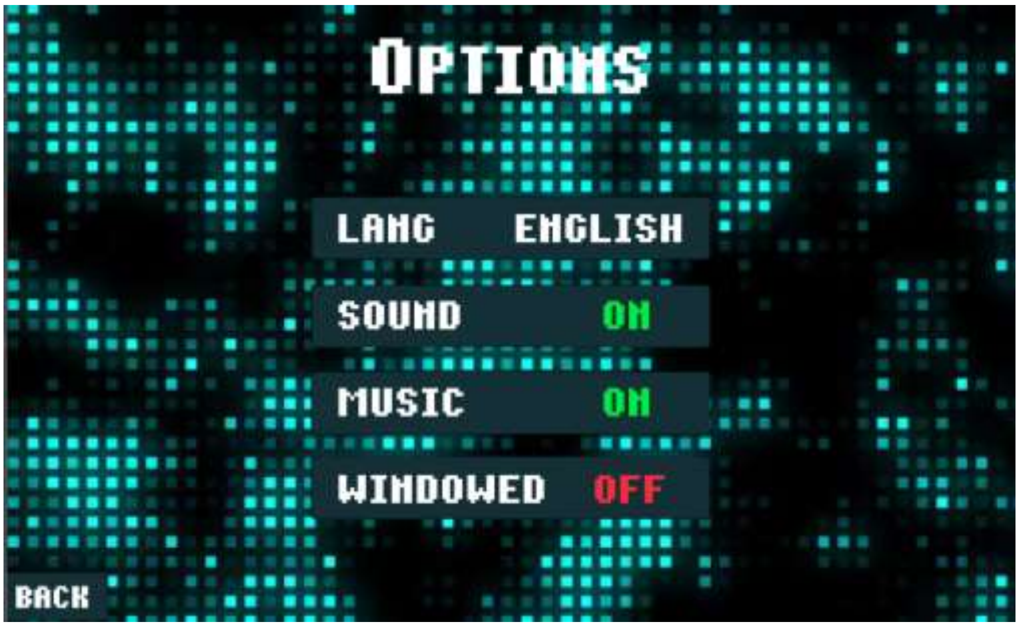
4.4.2. User Interface

Menus [figures 4.3, 4.4] and interfaces [figures 4.5, 4.6] were prepared according to mockups from subchapter 3.7. An animated background was used in order to make otherwise simple and bland menu look more interesting. Button highlight feature was added for a visual pointer while using the menus with a controller. Exit button highlight hue was changed to red to differentiate its role and to serve as a warning visual cue before pressing it.



e

Figure 4.3. Main menu, source - own



LANG ENGLISH
SOUND ON
MUSIC ON
WINDOWED OFF

Figure 4.4. Options menu, source – own

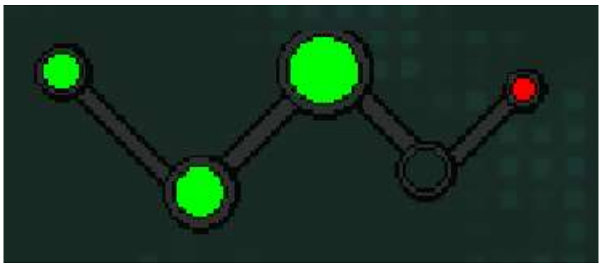


Figure 4.5. Two team point counter, source - own

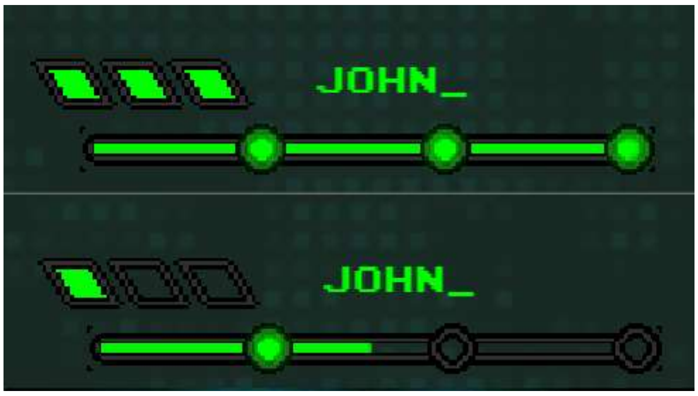


Figure 4.6. Player status bar, source – own

4.5. Animation

In 2D game development there are two main animation styles: spritesheet animation [figure 4.7] and skeletal animation [figure 4.8]. While the skeletal method has its advantages, such as smooth animation with the use of key frames and tweening, the more traditional, spritesheet method was more fitting with the retro theme of the project. This method requires drawing each frame of the animation separately. In most 2D games with low-fidelity graphics just a few (4 to 6) frames are enough to convey information about characters action.

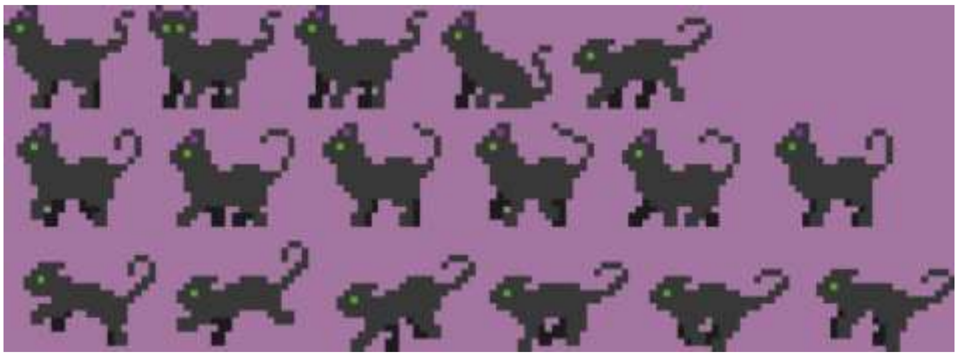


Figure 4.7. Spritesheet animation example, source – own

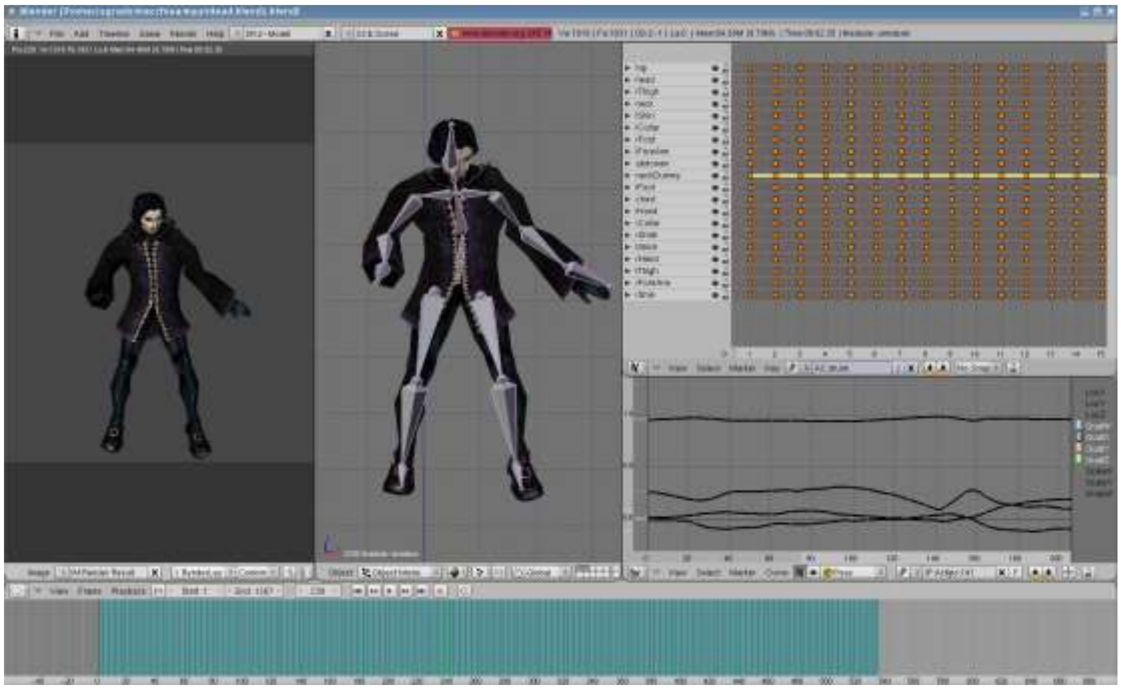


Figure 4.8. Skeletal animation example. access 08.11.2019
source - http://alumni.cs.ucr.edu/~sorianom/cs134_09win/lab5.htm

For the application prototype, complete spritesheet for one character class was prepared [figure 4.9]. It contains frames used in following animations: standing idle, running, attacking, jumping, falling, dying, and teleporting.

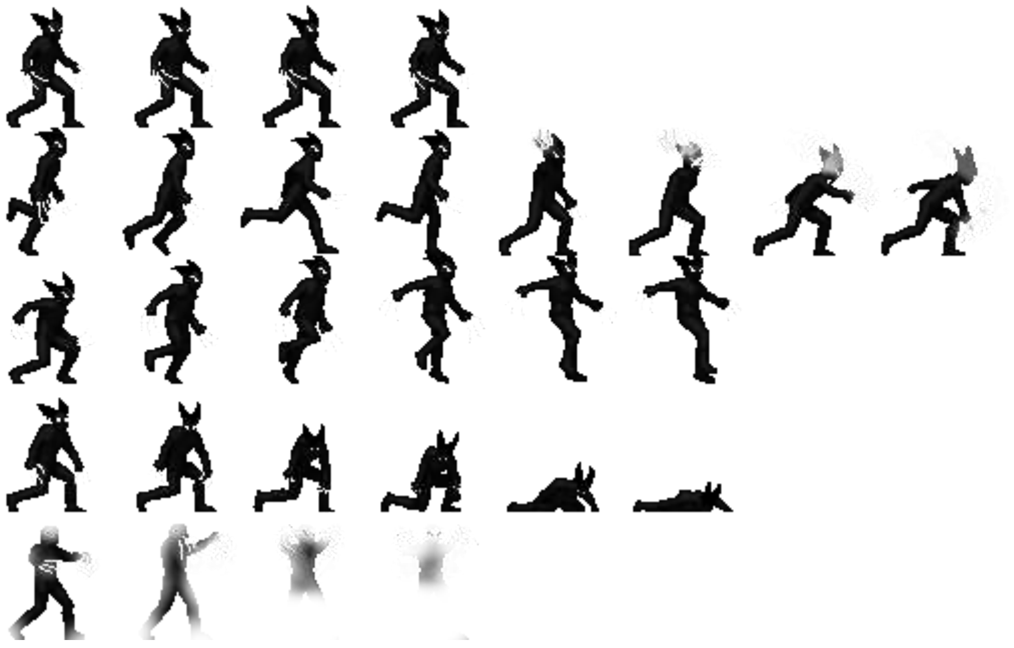


Figure 4.9. Glitch character animation spritesheet, source – own

After preparing the spritesheet and importing it into the Unity project, next step was to create animations by highlighting the graphical component of player character object and using Unity’s Animation window [figure 4.10].

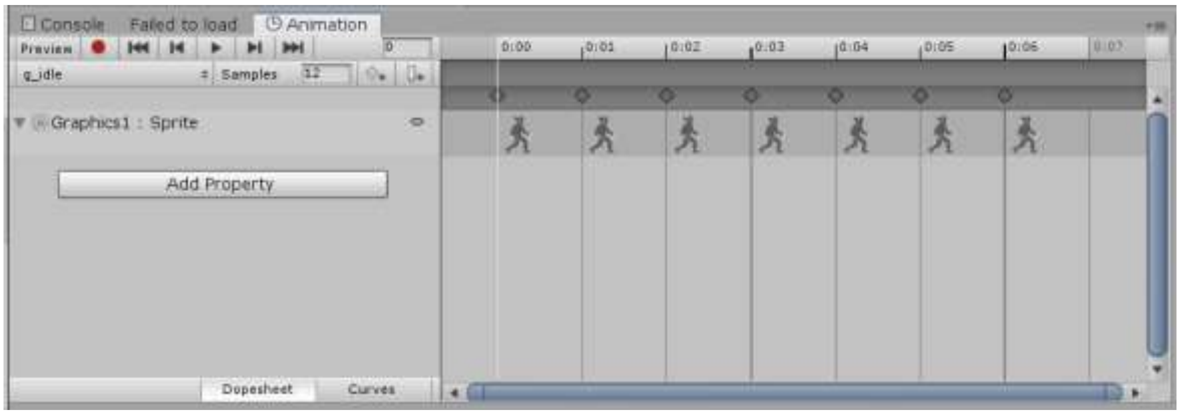


Figure 4.10. Unity Animation window, source – own

During the previous step, Unity automatically creates an Animation Controller – a component which decides which animations are going to be displayed, as well as their speed, delays, overlapping, and many other attributes. The Animation Controller component can be viewed and edited through Unity’s Animator window [figure 4.11].

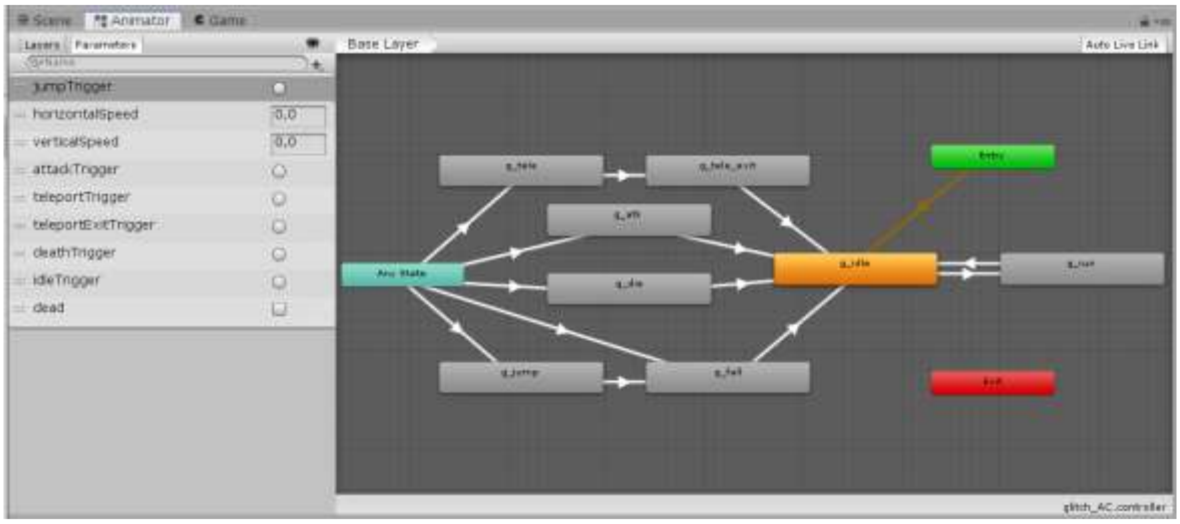


Figure 4.11. Unity Animator window, source – own

Animator window is divided on two parts: on the right side all animation states created for the given object are displayed, this is where we can access detailed options for each animation and create animation transitions, displayed as one-directional arrows. On the left side of the window is a list of parameters, which serve as an interface between Animation Controller and rest of the application. Parameters can be set through the application code during runtime, which in turn is used by the Controller to perform animation transitions. Above image shows animation controller created for the purpose of animating the Glitch character.

4.6. Audio

Audio layer is vitally important for any multimedia project. Because of the limited time frame of the project and lack of proper skill, music layer was chosen from the available external sources, listed in subchapter 4.3.5. Sound effects were generated manually using free Bfxr program [figure 4.12], which with the amount of clip samples and modulation options has proven to be a flexible and reliable tool.

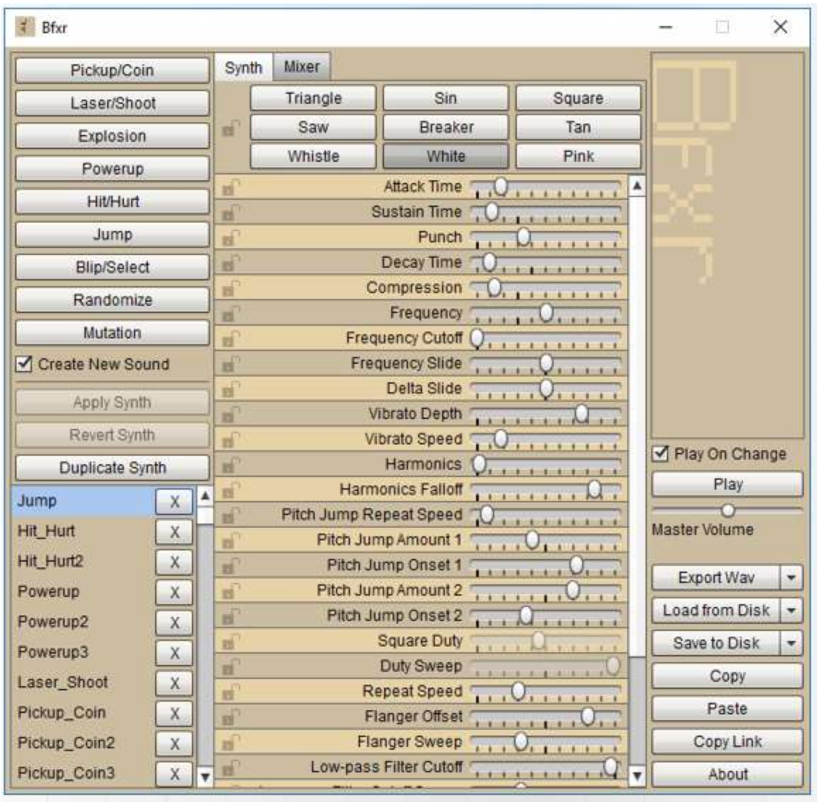
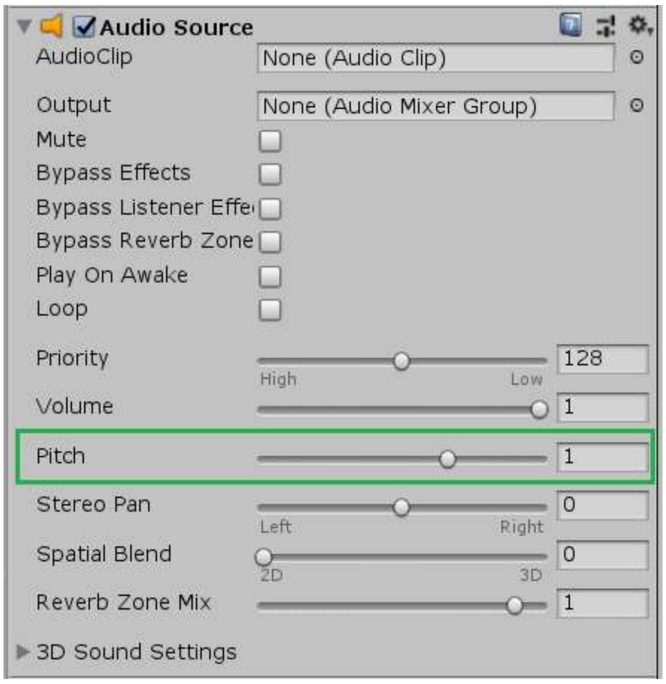


Figure 4.12. Bfxr program view, source - own

Unity uses Audio Source component [figure 4.13] as a mean of introducing sound onto the game scene. Naïve approach often taken by novice designers assumes that each audio clip requires its own audio source, which worsens program optimization and might introduce interference problems in case of many audio sources playing at once.

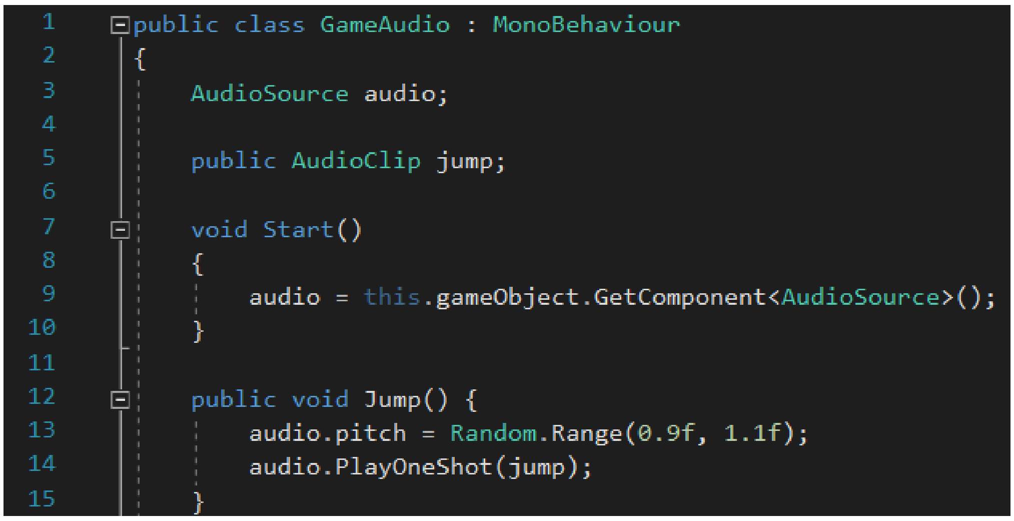

Audio Source

AudioClip	None (Audio Clip)
Output	None (Audio Mixer Group)
Mute	☐
Bypass Effects	☐
Bypass Listener Effe	☐
Bypass Reverb Zone	☐
Play On Awake	☐
Loop	☐
Priority (High – Low)	128
Volume	1
Pitch	1
Stereo Pan (Left – Right)	0
Spatial Blend (2D – 3D)	0
Reverb Zone Mix	1

3D Sound Settings

Figure 4.13. Audio Source Unity component, source – own

Proper approach [listing 4.1] would be to use one audio source for one object, reference list of all sound clips used by it during the scene load and then play the required clip with PlayOneShot( ) function. Many beginners make the mistake of assigning a clip to the audio source and activating it with Play( ) function, which does not allow to use multiple audio clips in quick succession and introduces audio distortion if another clip is played on the same audio source before the previous one ends.



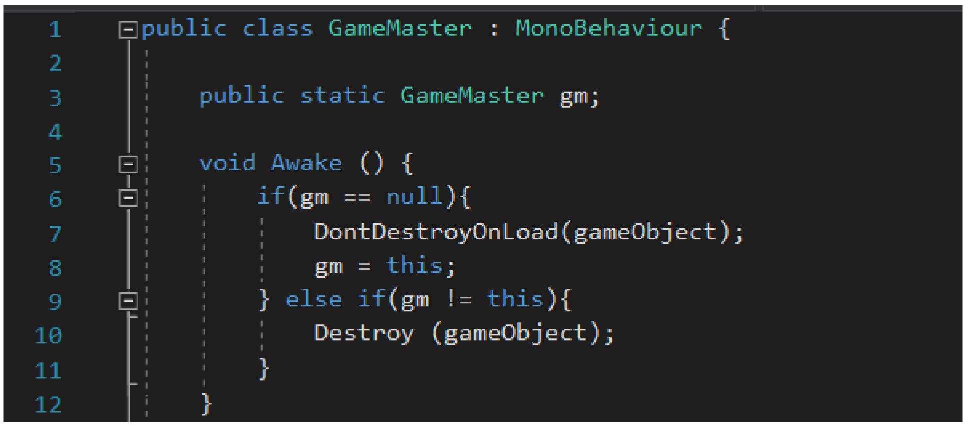
```
public class GameAudio : MonoBehaviour
{
    AudioSource audio;

    public AudioClip jump;

    void Start()
    {
        audio = this.gameObject.GetComponent<AudioSource>();
    }

    public void Jump() {
        audio.pitch = Random.Range(0.9f, 1.1f);
        audio.PlayOneShot(jump);
    }
```

Listing 4.1. GameAudio script code fragment, source – own

At last, simple sound effects like those used in the project, which are going to be played tens or hundreds of times during single game match introduce the problem of repetitiveness. To introduce a variety and avoid making effects sound exactly the same, modifying the pitch value [figure 4.13] is optimal solution, which does not require producing any new clips with additional sound variations. After testing, I have found that modifying the pitch programmatically within ±10% of its base range (figure 4.14, line 13) gives the best results.

4.7. Programming

Game's core systems were implemented using C# language and Unity design patterns. One of the design patterns used was Singleton behavior [listing 4.2], which serves as a guard against duplication of critical scripts during runtime. It works by keeping a reference to the static instance of itself, if there is no static instance, it becomes one. If the static instance already exists, it forces any further instances to destroy themselves. Despite being considered harmful by some critics [14], in this case it does not impact performance negatively because it performs its check only during the scene load, and is an important part of any project that works by using and reloading multiple scenes, which could break if two or more instances of a critical script were active at the same moment. Nonetheless, this design pattern does introduce a global state to the program and should be used very sparingly.



```
public class GameMaster : MonoBehaviour {

    public static GameMaster gm;

    void Awake () {
        if(gm == null){
            DontDestroyOnLoad(gameObject);
            gm = this;
        } else if(gm != this){
            Destroy (gameObject);
        }
    }
```

Listing 4.2. Singleton behavior code fragment, source – own

Another design pattern prevalent in Unity engine is the proper use of Unity's Event Functions, in particular Awake( ), Start( ), Update( ), FixedUpdate( ) and LateUpdate( ) [15]:

- **Awake** – Awake is used to initialize any variables or game state before the game starts. Awake is called only once during the lifetime of the script instance.
- **Start** – Start is called on the frame when a script is enabled just before any of the Update methods are called the first time.
- **Update** – Update is called every frame. It should be used for anything related to display and player input.
- **FixedUpdate** –has the frequency of the physics system; it is called every fixed frame (frame-rate independent) - 0.02 seconds (50 calls per second) by default. Its intended use is for physics system calculations.
- **LateUpdate** – LateUpdate is called after all Update functions have been called. This is useful to order script execution. For example a follow camera should always be implemented in LateUpdate because it tracks objects that might have moved inside Update.

4.8. Encountered problems

During the implementation phase of the project, various types of problems were encountered. Part of them were technical, other were related to gameplay balance and user experience. Many of those problems were found thanks to user testing which started very early and had an important role throughout the entire implementation process. Below I'm listing some of the encountered problems and methods used to solve them:

4.8.1. Preserving information through the scene load

Problem description:

By default, during a scene load in Unity all existing object in scene hierarchy are deleted, and new ones are created from the new scene file. This makes preserving information, such as selected player options or preferences impossible to do within the properties of an instantiated object.

Solution:

Chosen solution to this problem was to use PlayerPrefs class, which is a part of Unity Engine Core Module. It stores and accesses player preferences between game sessions [16]. Using PlayerPrefs is not intended to preserve game state information, but is an efficient, simple solution for preserving a limited number of player preferences, such as game options. The project doesn't need to preserve any other type of information, which made this solution an obvious choice. This introduced additional feature to the project, by allowing to preserve match settings between game sessions and removing the need for players to reenter their data and preferences after restarting the game match.

4.8.2. Camera stutter

Problem description:

Camera movement realized with a custom script manifests noticeable delay and stutter during fast movements of tracked player characters.

Solution:

After analyzing the camera tracking script, the fault was found in Mathf.Lerp( ) function used for the linear interpolation of camera position. Function updated its status based on static time interval, which led to noticeable artifacts with faster movement. The solution was to replace the standard function with manually added code calculating the camera position, and assigning this position to main camera transform during the Update( ) event function to ensure it is done in synchronization with the frame rate, so that no stutter is visible. While the calculation is now made more often, it is simpler than that of interpolation function, hence the solution did not introduce any noticeable performance drop.

4.8.3. Many-to-many connection between scripts

Problem description:

Because of the player choice in regard to their character class and control scheme, a problem of many-to-many connection between the controller script and character class script has arisen [figure 4.14]. Naïve solution would involve duplication of all character class scripts for each player separately, which would introduce a lot of redundant code and would not comply with clean code programming pattern.

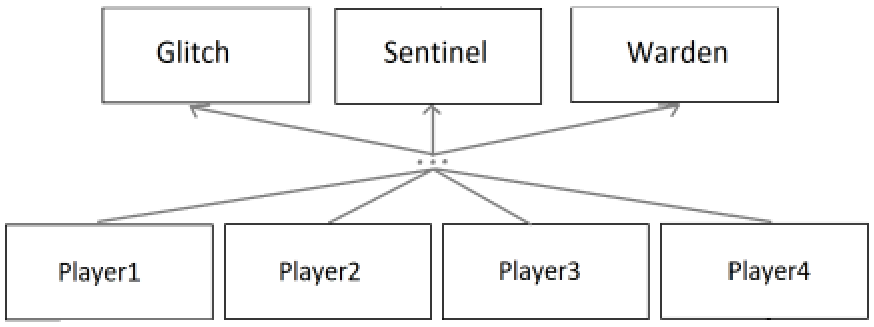


Figure 4.14. Many-to-many script connection, source - own

Solution:

The solution was to use the inheritance property of the object-oriented C# language. A new class called CyberSuit was created as a superclass to the character class scripts [figure 4.15]. This allowed to use it as an interface between the control scripts and character behavior, without the need to assume the character class chosen by the player in the PlayerController script. Additionally, PlayerInput scripts were introduced for all possible input methods to make PlayerController scripts independent from the control scheme chosen by the player. This helped in avoiding even more code duplication, which would occur if I tried to involve every possible input method in each PlayerController script.

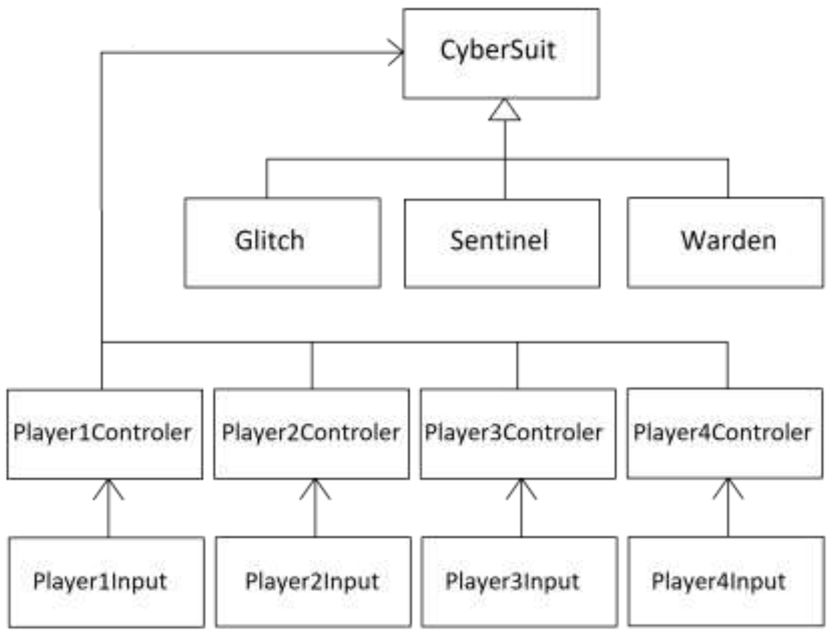


Figure 4.15. Improved script hierarchy, source – own

4.8.4. Using numerical keyboard in control scheme

Problem description:

Testers reported inability to use numerical keys for controlling their character due to not having numerical keypad on their keyboard or not having pressed the NumLock button.

Solution:

Most user-friendly solution to combat the NumLock problem is to automatically set NumLock state to on when launching the program. On Windows it can be achieved by using Microsoft API and modifying the "user32.dll" file [17]. The problem of not having a physical numerical keypad could only be solved by allowing the player using right-hand side of the keyboard to use an alternative control scheme, as described in subchapter 3.6.1.

4.8.5. Animations getting stuck in loop or wrong state

Problem description:

Players' characters get stuck in a wrong animation cycle, such as activating running animation after dying or getting stuck in falling animation.

Solution:

The source of the problem was tracked to "Any State" node of the Animator component [figure 4.11, p.34]. Because of using floating point values like 'HorizontalSpeed' as parameters, Animations could be called out of order even after enabling the 'HasExitTime' option and setting exit time to max value for animations that are not meant to be overwritten. The solution was to add a new boolean parameter within the Animator, and check its value as an additional condition for animation transitions, particularly from non-skippable animations and the "Any State" node.

4.8.6. Glitch character teleporting out of gameplay area bounds

Problem description:

Because of its unique teleport ability, Glitch character class allows it to pass through the gameplay area bounds implemented as a physical barrier. This is unintended behavior, can make the character stuck and breaks the game balance.

Solution:

The solution was to introduce empty "guardian" objects on the lower-left and upper-right corners of the playable area. When a glitch character wants to teleport, its teleport destination location is compared with positions of guardian objects present on the arena, and prevents the character from teleporting where he shouldn't by using them as limits – the teleport destination is set to a position just within the level boundary. Ray tracing solution was taken into consideration but quickly abandoned, because the ability was intended to pass through other physical objects on the scene, which would make a ray tracing approach overly complicated.

4.8.7. Attacks feel weak and unsatisfying

Problem description:

Testers reported that performing a successful melee attack does not feel satisfying, and it is not clear whether the attack actually hit the enemy.

Solution:

This problem had to be addressed on multiple layers. First thing that was added to make the attacks more pronounced was a physical pushback of the attack target in the direction opposite to the attack. This would also work as a balancing factor, preventing immediate execution of next melee attack by the attacker. On the audio layer, melee attack sound was divided into three sound samples – swooshing sound of attack being performed, sound of attack hitting an object, and sound of target character being wounded. On the visual layer, particle effect was set to emit an outward burst of particles from wounded target. Finally, the last tool for creating the impression of force and impact was implementation of additional script that performs a camera shake whenever any character present on the screen receives damage. User testing performed after introducing these changes confirmed that expected results were achieved and the problem is no longer present.

4.8.8. Camera gets stuck with multiple scripts

Problem description:

Camera shake script implemented while solving the previous problem interferes with player tracing script and makes camera stop following players on vertical axis properly. Camera started acting erroneous and unpredictable.

Solution:

Upon closer inspection of both scripts, the source of the problem was found in that camera shake script works by modifying the "localposition" value of the camera object. Because both scripts were updating camera position during Update( ) event function, shake script sometimes overwritten the calculations done in tracking script before the end of the frame, thus preventing them from being applied. The solution was to make use of the call order of Unity Event Functions [15]. The code of camera shake script was moved to execute in LateUpdate( ), which is called after all other Update( ) functions, making sure it allows the calculations of the tracking script to complete and execute before applying its position modifications.

4.8.9. Players getting killed immediately upon respawn

Problem description:

Upon being defeated, characters respawn at their last known position, which allows other characters to ambush them right after being respawned. Moreover, character who died to environmental damage (e.g. falling on spikes) gets respawned right on top of the dangerous element and get stuck in a loop of being killed repeatedly.

Solution:

First tested solution was to respawn players at their static spawn points, but it slowed down the pacing of the game, caused camera to abruptly change its position and did not prevent players from being ambushed. Instead, the chosen solution was to randomly change character position within a certain range from their last known position upon being respawned. This solution makes it impossible to predict exact location of the next character's respawn point, but ensures that the player won't get spawned too far from the action and that camera will not have to suddenly shift to a far away spawn point. The guardian objects from earlier teleportation problem (subchapter 4.8.6) have again proven useful as limiters to prevent players from respawning outside the gameplay area.

4.9. Implementation result

The result of the implementation phase is a playable video game prototype fulfilling the role of minimum viable product. Due to time restrictions it was not possible for me as a single developer to create a full version of functional video game, but most important functions of the project: core gameplay, scripts, graphics, animations, user interface and audio have been implemented. The only thing left to finish the project is adding more content to the prototype, such as making other character classes playable or creating more arena levels or a way to randomly generate them.

The finished prototype [figure 4.16] allows to perform a one versus one match and supports both keyboard and controller control schemes. Because of static number of active players, it is even possible to change between controller and keyboard during program runtime.

Playable prototype has been uploaded to the service *itch.io* and is available for download at:
https://kamilwojtysiak.itch.io/glitchhunter

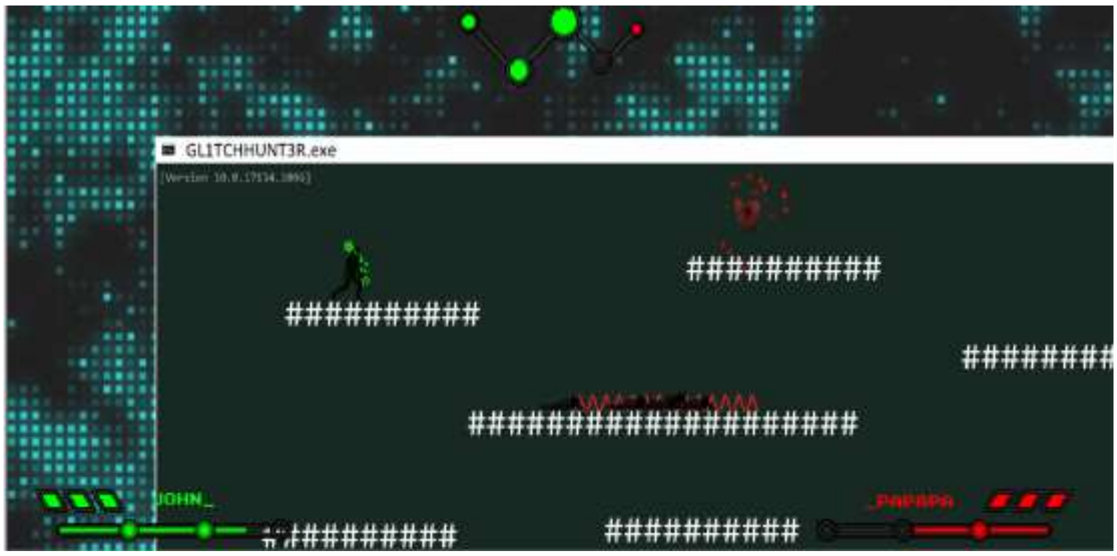


Figure 4.16. Project prototype gameplay, source – own

Due to low graphical requirements of the project, prototype works on a low-end computer with stable framerate of around 80 frames per second, without any noticeable input lag. This not only ensures the game is playable on most modern computer setups, but also shows a promise for porting the project to other platforms as well.

4.10. Further development

First step for the project will be of course finishing it with accordance to Game Design Document prepared in Chapter 3. Creating an enemy AI would be a welcome addition to introduce single player gameplay. Because of choosing Unity engine, process of porting the project to other platforms will be fairly straightforward, and the project's native support for various types of external controllers makes it worth investigating. This type of video game would play nicely on any console which supports use of multiple controllers. Because of very simple control scheme, after implementing a support for play over a local network, either through Wi-Fi or Bluetooth, it could even be ported to handheld devices such as smartphones. Content updates could be performed with low effort by adding new character classes, power-ups and play modes over time to keep players coming back to the game.

Summary

In conclusion, the goal of the thesis was reached – a playable early version of the game has been implemented, which can serve as a foundation to build a fully functional version of the game on.

I have chosen this topic to learn more about the process of video game production. During writing of this thesis I have learned about substantial development principles, such as analyzing the business potential of the project and performing comparison of existing solutions. A lot of different components constitute to a video game project, and it’s very hard to remember about all of them during the design phase alone. During implementation, I have met with several technical problems and was prompted by users to implement functionalities that were not predicted in the original design (e.g. player pushback and camera shake upon landing an attack). Video games have unique implementation problems such as game balance, which are not present in non-game application projects, but are a very important aspect of the finished product experience, especially ones with competitive gameplay. Because sole purpose of video games is entertainment, it is very important to ask for user feedback, starting as early in the development process as possible.

In the thesis, phases of video game development process were shown, from substantiating the general idea for the product, through project, design to implementation. User testing was performed throughout the entire development process, gathering feedback and implementing changes right after functionalities implementation, this approach allowed to avoid large scale changes in the late stages of the project due to user dissatisfaction. The design chapter was written in a form of Game Design Document, and can be used to finish the prototype and release the project in a video game store of choice in accordance to the business analysis performed in the project chapter.

Overall, creation of this video game started with finding an idea for a game that I would like to play myself with my friends, and the satisfaction of potential users with the created prototype allows me to have an optimistic look on public reception of the title upon its eventual release.

References

[1] J. T. T. Goldsmith, *Cathode ray tube Amusement Device*. USA Patent 2455992, 1947.

[2] D. Ellis, *A Brief History of Video Games*, Random House, 2004.

[3] T. Minkkinen, *Basics of Platform Games*, 2016. [Online]. Available: https://www.theseus.fi/bitstream/handle/10024/119612/Thesis%20-%20Toni%20Minkkinen.pdf [access 8.11.2019].

[4] A. Rollings, E. Adams, *Fundamentals of Game Design*, Prentice Hall, 2006.

[5] K. Burgun, *Asymmetry in Games*, 2015. [Online]. Available: http://keithburgun.net/asymmetry-in-games/ [access 08.11.2019].

[6] B. Bates, *Game Design*, Thomson Course Technology, 2004.

[7] L. Pscarella, M. Di Penta, F. Palomba, A. Bacchelli, *How Is Video Game Development Different from SoftwareDevelopment in Open Source?*, University of Zurich, 2018.

[8] A. Bailey, *From whence came that rocket?*, Speed Demos Archive, 2016. [Online]. Available: http://quake.speeddemosarchive.com/quake/qdq/articles/WallHug/rjump.htm [access 08.11.2019].

[9] K. Yiffrah, *Are you sure you want to do this? Microcopy for confirmation dialogues*, UX Design, 2019. [Online]. Available: https://uxdesign.cc/are-you-sure-you-want-to-do-this-microcopy-for-confirmation-dialogues-1d94a0f73ac6 [access 08.11.2019].

[10] S. Axon, *Unity at 10: Game development has never been easier*, Ars Technica, 2016. [Online]. Available: https://arstechnica.com/gaming/2016/09/unity-at-10-for-better-or-worse-game-development-has-never-been-easier/ [access 08.11.2019].

[11] A. Peck, *GIMP: From Novice to Professional*, Physica-Verlag, 2006.

[12] P. Hogan, *InControl: Introduction*, Gallant Games, 2019. [Online]. Available: http://www.gallantgames.com/pages/incontrol-introduction [access 08.11.2019].

[13] Creative Commons, *The licenses*, About the licenses, 2016. [Online]. Available: https://creativecommons.org/licenses/ [access 08.11.2019].

[14] M. Hevery, *Clean Code talks - Global State and Singletons*, 2008. [Online]. Available: https://testing.googleblog.com/2008/11/clean-code-talks-global-state-and.html [access 08.11.2019].

[15] Unity Technologies, *MonoBehaviour*, Unity Scripting API, 2019. [Online]. Available: https://docs.unity3d.com/ScriptReference/MonoBehaviour.html [access 08.11.2019].

[16] Unity Technologies, *Player Prefs*, Unity Scripting API, 2019. [Online]. Available: https://docs.unity3d.com/ScriptReference/PlayerPrefs.html [access 08.11.2019].

[17] User pako, *Turn Numlock On At Start*, Unity Answers, 2015. [Online]. Available: https://answers.unity.com/questions/983993/turn-numlock-on-at-start.html [access 08.11.2019].

List of tables

List of code listings

List of figures