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 2D VIDEO GAME IN GAME ENGINE GODOT

NÁVRH A IMPLEMENTACE 2D VIDEOHRY V HERNÍM ENGINE GODOT

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **TOMÁŠ KURTIN**
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

146143

Institut:	Department of Computer Graphics and Multimedia (UPGM)
Student:	**Kurtin Tomáš**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**Design and Implementation of 2D Video Game in Game Engine Godot**
Category:	Computer Graphics
Academic year:	2022/23

Assignment:

1. Explore the history and state of the art of 2D game design.
2. Choose a game engine suitable for implementation. Familiarize yourself with the chosen game engine and describe its features.
3. Design a new platformer game based on the gained knowledge.
4. Implement the designed game and experiment with different designs and game mechanics in successive iterations.
5. Present the game in the form of a poster and a short video.

Literature:
Dodá vedoucí práce.

- 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.
- Godot Engine, https://godotengine.org/

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:	31.10.2022

Abstract

The goal of this thesis is to design and create new 2D game in game engine Godot. The work begins with design, which focuses on researching already existing games. From the knowledge gained, the set of the skills which the player would be able to use was created. In the next step, the world and the story happening in it were designed. Speed was chosen as a main mechanic, because of slow feeling games from the Metroidvania genre. Speed generates energy, which strengthens attacks and allows the player to use other abilities. The whole design was then implemented in Godot game engine, which is a game development environment. The implementation focuses on telling the story and on smoothness and fastness of movement.

Abstrakt

Cílem práce je narvrhnout a vytvořit novou 2D hru v herním engine Godot. Práce začala návrhem, který se zaměřil na zkoumání již existujících her. Podle získaných poznatků byla vytvořena sada schopností, které hráč bude moci použít. Dále byl vytvořen návrh světa a příběhu, který se v něm odehrává. Jako hlavní mechanika nové hry byla zvolena rychlost z důvodu pomalu působících her z žánru Metroidvania. Rychlost generuje energii, která posilňuje útoky a umožňuje hráči provádět další schopnosti. Celý návrh byl poté implementován v herním engine Godot, který je vývojovým prostředím pro hry. Implementace se zaměruje na vyprávění příběhu a na plynulost a rychlost pohybu.

Keywords

Godot, 2D, videogame, game, platformer, Metroidvania

Klíčová slova

Godot, 2D, videohra, hra, plošinovka, Metroidvania

Reference

KURTIN, Tomáš. *Design and Implementation of 2D Video Game in Game Engine Godot.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Doc. Ing. Martin Čadík, Ph.D.

Rozšířený abstrakt

Práce se zaměřuje na návrh a vývoj 2D plošinové hry z žánru metroidvania v herním engine Godot.

Godot je vývojové prostředí, zvané též herní engine, zaměřující se na tvorbu her.

Jelikož rád hraju plošinové hry (hry, většinou 2D, kde se skáče z plošiny na plošinu), tak jsem se rozhodl jednu takovou vytvořit. Jako podžánr jsem si vybral Metroidvania hry. Jedná se o slovo, které pochází ze 2 her Metriod a Castlevania, které definovaly tento žánr. Častými prvky pro Metroidvania hry jsou: průzkum prostředí, získávání schopností, boj s příšerami a odemykání bran. Brány oddělují hráče od zbytku hry, dokud nemá určitou schopnost nebo znalost. Metroidvania hry se často zaměřují na průzkum světa, se skrytými místnostmi a různými detaily ohledně příběhu.

Pro návrh byly vybrány známe hry a následně prozkoumány. Zkoumáno bylo hlavně z jakého důvodu jsou zábavné a co je dělá dobrými. Ze zkoumání bylo zjištěno, že dané hry se z hlediska pohybu zaměřují na pohyby, které hráči umožňují v pozdějších fázích lépe překonávat terén, a na každé místo se lze dostat několika způsoby. Z důvodu zaměření se na rychlý pohyb byly prozkoumány také hry s rychlým pohybem. Rychlý pohyb častěji způsobuje úmrtí postavy za kterou hráč hraje, jelikož není tak moc ovladatelná. Pro vyrovnání šancí hráče jsou tyto hry strukturovány do kratších úrovní, kde tolik nevadí ztráta postupu. Co se týká příběhu, mají Metroidvania hry tendenci mít záhadný děj, který se postupem času hráči odhaluje.

Tvorba příběhu probíhala postupně. Začala vytvořením světa, který by se dal označit post apokalyptickým světm, kde démoni zmocnili světla ze slunce a všude je proto tma. Poté byla stvořena hlavní postava, hrdinka, která si o tomhle světě nic nepamatuje a tak se vydává za dobrodružstvím, aby zjistila kde je a proč tam je. Po vytvoření hrdinky mohly být vytvářeny její schopnosti. Základem pohybu je běh a skok. Byl vytvořen i dash (rychlé vymrštění), který je možný provést do 8 základních směrů (jako na kompasu). Dalším hlavním prvkem je souboj. Hrdinka má k dispozici magickou zbraň, která je schopna měnit svůj tvar do jiných zbraní (meč, sekera a dýky). Tyto zbraně se poté chovají odlišně, mají různý dosah, poškození a dobu použití. Pro ozvláštnění průběhu hry se ve hře vyskytuje strom schopností, takže každý průchod může být odlišný od toho minulého. Schopnosti se dělí na čtyři typy (Síla, Obratnost, Intelekt, Stín), kdy každá má svůj hlavní efekt a dále přidává ještě bonusové vlastnosti dle typu. Pro vyvážení rychlého pohybu a tedy menší ovladatelnosti se hráč při smrti vrátí kousek v čase aby se mohl chybě vyhnout.

Hra je přístupná díky tutoriálu, který učí hráče základy ovládání. Turoriál hráče nechá chvíli zkoušet a poté, když hráč neví, tak ho navede správným směrem. Hru můžou také hrát lidé, kteří nerozumí anglickému jazyku, díky možnosti přepnout ji plně do jazyka českého.

Příběh je vyprávěn formou dialogů, které hráč může přepínat sám, nebo si je nechat přehrávat automaticky. Důležitým faktorem je osvětlení hlavní postavy, přičemž celá hra kromě blízkého okolí je zatmavená. To vytváří pocit nejistoty, kdy hráč neví co se přesně skrývá ve stínech.

Hra byla otestována sledováním hráčů. Chyby, kdy hráči hráli hru jinak než bylo zamýšleno, nebo nepochopili dané prvky byly následně opraveny.

Design and Implementation of 2D Video Game in Game Engine Godot

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.

.....
Tomáš Kurtin
May 8, 2023

Acknowledgements

I would like to thank Mr. Martin Čadík for helping me through the thesis. I also would like to thank my friends for testing the game and my family for helping me with corrections.

Contents

Chapter 1

Introduction

I love playing video games. I like games with good story, fighting mechanics, and good movement. Recently I have played quite a lot of platformer/metroidvania games and pretty much all of them have precise, but for me pretty slow feeling movements. So I would like to try building a game based on fast movement. Alongside fast movement, I would also like to try to use the speed that the player would build for other aspects of the game. This work will focus on fast movement and the use of it in combination with other mechanics. I would like to try to make a game that is not only about moving fast, but also a game that rewards the player for not stopping. The player's attacks will be stronger, their abilities will get stronger, etc. With this, I hope that the player will enjoy the movement more and spend more time running than planning what to do next or waiting for the enemy to turn their back towards them.

For implementation, I have a game engine called Godot. It is an open-source game engine for creating games. Mainly for 2D games, but with its help, 3D games can be built, too. I don't know much about game making and Godot, so this work will also serve as a test to how beginner-friendly Godot is.

This work covers the entire process of making a game, beginning with coming up with an idea and designing around it. The second half covers the implementation of the designed game. Chapter 2 covers key concepts used in this thesis, the next chapter (3) is about designing the game. Chapter 4 is about implementation in Godot.

Chapter 2

Key concepts

This chapter sums up the basics needed for the work that I needed to study.

2.1 Movement

This work is about fast and efficient movement, so this was one of the first things I had to study. This section is based on the games I played.

A lot of platformer games consist of basic movements, which can be found in almost all of them (or all of the modern ones). This includes running / walking, jumping, wall sliding, and wall jumping. They also have some special move that is there to make the game special in some aspect and differentiate it from others. Ori and the Blind Forest has its bash (Ori can „repel“ themselves from enemy or enemy projectile), it allows the character to evade projectiles. The player also gains a lot of movement freedom and makes playing the game more enjoyable. The hollow knight has dashes, normal and one obtained from `Crystal heart` (it is a super dash). The normal one allows the player to dodge attacks, but the super dash allows the player to travel long distances in one infinitely long dash. The player charges it on a wall and, when released, the character travels in one direction until it is canceled or reaches the opposite wall. This allows to open new routes which player couldn't access without it. Next, there is Celleste, which has eight directional dash, which can be recharged by stepping on the ground or picking a special orb. Cellest can also climb walls, but only for a limited period of time (the character gets tired). This combination allows players to move freely in the air and correct their mistakes on the wall by climbing up.

Special moves allows to add something different alongside basic movements. They also make the game feel special because they offer something that is not common in other games. It is good to add one or two of them, but then it can be hard to remember all the movements and how to use them.

Another way to differentiate the gameplay is to drastically modify the basic movement. The character cannot jump (good for puzzle games) or jumps really high like in Super Meat Boy. Super Meat Boy is a good example of how changing default movement can make a unique game. They sped up the character and gave it a high jump. This makes the character fly through the air, which contributes to the fast movement feeling. The character is also slippery because it is a cube of meat, which adds more to the fast-moving feeling.

Games usually tend to adding special move because with basic movements pretty much standardized it means that character will be controllable. On the other hand, Meat Boy

feels like you don't have much control of where it will land or when it stops. Developers want to give the player movements that feel like the character is under perfect control. In my opinion, this is where slow-feeling movement comes from. Also, games with enemies force the player to stop when they want to fight them (and player usually wants to fight them, loot, experience, etc.).

2.2 Godot [1]

Godot is an open source free game engine provided under MIT license. It is multi-platform, meaning it supports making games for Windows, MacOS, Linux, or Android.

2.2.1 Programming languages

Main languages are `Gdscript` and `C#`. `Gdscript` is developed purposely for Godot. It is similar to other script languages like `Python`. Due to the tendence of script languages to be less performant than low level languages it is not for creating big, high-end ralistic games. On the other hand, it can be useful for faster and user-friendlier creating smaller games and prototypes. `C#` can be used for faster and more optimized games, but the programmer may encounter difficulties, as it is relatively new to Godot.

2.2.2 Nodes, Trees and scenes [2]

Main bits of the game are the nodes, which are grouped together into trees. The nodes can range from sprites (textures) to buttons for menus. The programmer puts these nodes together, forming trees.

The scene is made from a tree. The main advantage of the scene is that it can be saved to a disk and used later. Using a scene can mean running it directly (levels in the game) or loading multiple scenes into another (multiple same enemies, at different places). Using scenes in other scenes also means that the programmer can divide the game into logical parts. The whole program can then be more readable due to the smaller script size when it does not contain everything.

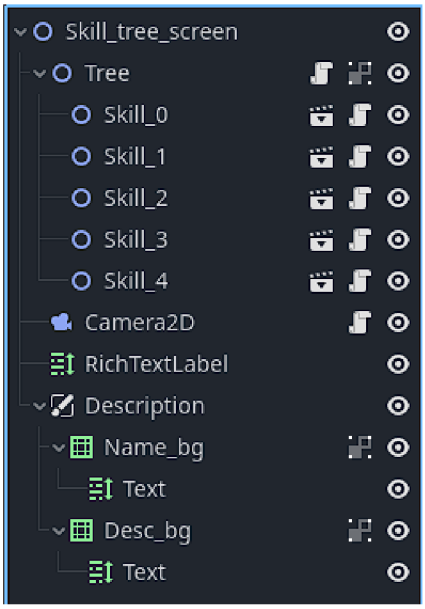

Figure 2.1: Screenshot from the engine

The image 2.1 shows how one scene can look. The entire tree can be seen on the picture, as well as individual nodes (tree, text, description,...). Skills are also a scene, which was created separately and inserted multiple times with different parameters.

2.2.3 Signals [2]

Signals are a way of communicating between nodes and scripts. One script / node sends a signal when some event occurs (Timer finishes, collision detection, ...), and another receives it and performs its action. Some of the node types have their predefined signals on which the programmer can connect using the editor. Programmers can also define their own signals, emit them, and connect to them in different scripts.

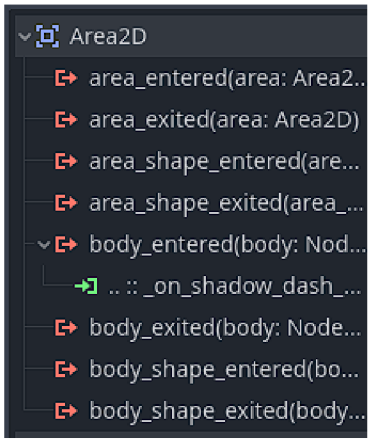

Figure 2.2: Area 2D signals in editor

Figure 2.2 shows how predefined singulars look for a Area2D node. The red symbols are the signals that are named after them. The green symbol signals the connection to another script and the method it will use in the connected script.

```
signal input_switch(input: Inputs)

input_switch.emit(Inputs.Keyboard)

PlayerVariables.input_switch.connect(_on_input_switch)
```

Figure 2.3: Three parts of custom signals

Figure 2.3 shows the second option for signal usage. The first line shows the signal definition with parameters. The second line shows the signal being emitted. And finally, there is connection to the signal in different scripts. These lines are cut from code to visualize how the signals work; they are not in this order in the code, nor are they in the same files.

2.2.4 Singletons (AutoLoad) [3]

Singletons are a way to share variables between scripts. It substitutes global variables from other languages. Scripts with important information (player variable,...) are loaded at the beginning of the game start, and other scripts can use them. This is important when switching scenes, because on the scene switch everything is reloaded and stored data in variables are lost. Except for AutoLoaded scripts, which will keep their data until the game is closed.

2.3 Metroidvania

Metroidvania is a genre of games. The name comes from two games that defined its principles, Metroid and Castlevania. Usually it is 2D and consists of jumping from platform to platform, killing enemies, unlocking abilities or skills, and uncovering the world and its mysteries.

The main mechanics are keys and gates. Gates lock the player from the rest of the map. Usually they do not look like typical gates (big doors), rather than that they are covered behind high wall, which player cannot jump over yet, long pit, which player cannot jump over, or seemingly unbreakable wall. This can create annoying dead ends if done wrong. If done right it can reward the player for noticing and remembering locked gates, but in the same time it can feel like everything is locked and there is nowhere to go if the player does not find the correct key.

To unlock the gates, the player needs to find a key, which can be a specific item or ability that allows them to overcome the obstacle. Keys are usually guarded by bosses, which are skill checks and blocks player until they are beaten. Sometimes keys are hidden behind a puzzle.

The world is an important part of a Metroidvania game. typically large world with many hidden places for the player to explore. Metroidvania also relies on a good story, which slowly unlocks the world and its parts.

Chapter 3

Game design

Game design is in my opinion the most important part of making a game (video game, board game, etc.). If the game is well designed, people will like it, even if it is not completely perfect. Many people can forgive non-perfect graphics or music that is not on the same level as Mick Gordons (Doom) or other high-skilled musicians in game industry. Yes, some people will not play the game if it does not have the highest graphics possible. From my experience, I would rather play a game that has decent story and with controls that aren't horrendous rather than one that has top-notch graphics but is badly optimized. Of course, there are games that build mainly on visual experience. For example, Firewatch builds its popularity on good story and good looking graphics. Its controls are refined to the max, but the movements that the player can do are pretty simple, walking, and interacting with objects. On the other hand, there is Undertale, which for most people does not have the most appealing graphics, but its story is so good that it build a large fanbase. The big aspect is that the story adapts player action and that fights are not necessarily just about killing. The common thing of these games is that they had an interesting idea which they refined to the maximum. The other aspects of the game are good, but nothing outstanding, these games are built to suit some sort of people. Someone might hate Firewatch for its simplicity (just walking and talking), someone might not like Undertale because the graphics do not suit him. In my opinion this is probably why these games are good, they don't try to be top notch in everything or to please all players. They know what kind of people would play the game and with that in mind the game is made.

In this stage (design), the game is being shaped from small pieces and starting to form a large picture depicting the game as a whole. First, it is good to brainstorm all ideas separately or as a mind map, and then start to glue them together. It will not be perfect, some ideas will be missing and some will be excessive. It is not a problem because the design is just a guide of what the final product could look like, not what it will look like. If someone reads the game design, they should be able to say what the game is about.

In my opinion, game design should consist of:

- Genre
  - It narrows the direction the game could take and tells players what to expect.
- Story and World Setting
  - Good stories make the player move through the world with purpose and not just roaming around.

- Mechanics
  - What can a player do to interact with the world?
- Something new
  - Something that will differentiate the game from others.

3.1 Inspiration

Inspiration plays an important role in what the result could look like. What we play, watch, or read can affect the final product. This inspiration could be divided into 2 categories – intentional and unintentional. The first is planned as „I borrowed this from the xx game because it is a good mechanic.“ The programmer can then say that it is not their idea, but that they liked it enough to incorporate it into their game. The second comes from forgotten experiences; these might be dangerous because we can say that it is our work, when in fact it is not. On the other hand we remember these for some reason and they are not attached to some game – they are good on their own not because they are part of something bigger.

Hollow knight

„Forge your own path in Hollow Knight! An epic action adventure through a vast ruined kingdom of insects and heroes. Explore twisting caverns, battle tainted creatures, and befriend bizarre bugs, all in a classic, hand-drawn 2D style.“ [12].

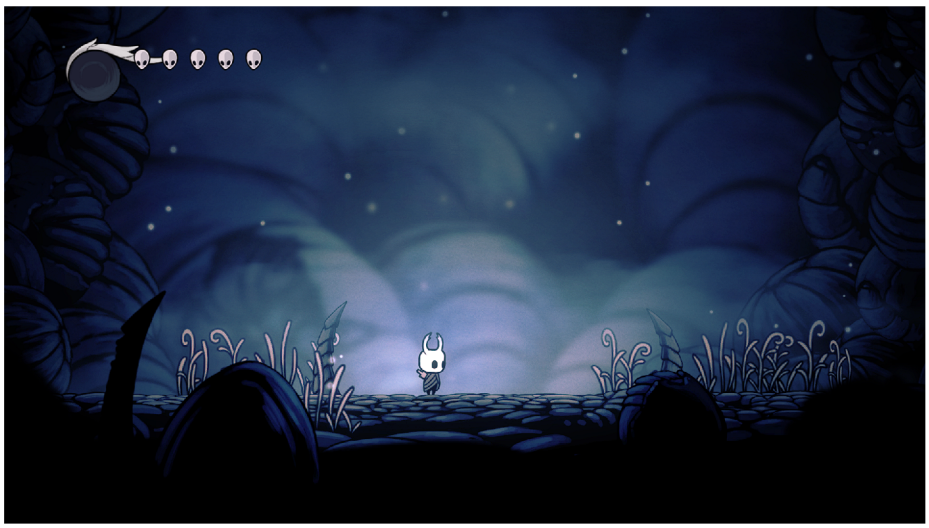

Figure 3.1: Screenshot from the game

Hollow knight is metroidvania from 2017 which was made by Team Cherry[1]. It is about a knight (from bug like empire). As a player you don't know anything about him, or about the world, so you try to get a grip of what it is all about and help the little one on their journey. Jumping from platform to platform, fighting enemies, or gaining new abilities.

[1]https://www.teamcherry.com.au/

These can be considered main mechanics, which could help you to get to know what it is all about and could help the knight to achieve their goal be it whatever.

I really liked the mystery atmosphere, where you don't know where you are or what your goal is. The main part of the game is its unforgiving combat mechanics. They are hard, but when you master them they feel as natural as walking.

Ori and the Blind Forest

„'Ori and the Blind Forest' tells the story of a young orphan destined for heroes, through a visually stunning action platformer created by Moon Studios for PC“ [9].

Figure 3.2: Screenshot from the game

I really liked Ori for its visual and audio style. Although i wrote that graphics do not matter so much, here it ads another level to a good game. The story here is told by the narrator. Even when creatures don't speak or just make random squeaks or other sounds we can still be sure what they are saying. It is not heavy-fight-oriented, the player mainly just mashes one button and evades enemies. On the other hand, platforming is well thought out and there are many ways to get to one place. The player has many moves that help him to get from one place to another. Some of the moves are double jumps, dash, glide, and much more. It feels good to get to where you want and with more and more abilities you can get there faster and faster.

Super Meat boy

„The infamous, tough-as-nails platformer comes to Steam with a playable Head Crab character (Steam-exclusive)“ [13]!

Figure 3.3: Screenshot from the game

Super Meat Boy is a quick platformer where you play as a Meat Boy who is trying to save the Bandage Girl stolen by a villain. In your way are saws, lasers, rockets and various monsters. The game is divided into levels that the player repeats until reaching finish. So they run, die, and again, and again, and .... This style is quick and gives the player the opportunity to test various paths and choose one that suits them. Build on short levels (where the longest lasts for a few minutes at max). Because of that movement can be sloppy, there can be unknown traps and death is waiting „behind every corner“. It is meant to be a hard game, but at the same time the difficulty does not punish the player too much. If you die, then what? You lost about 20 seconds of progress, and now you know that you shouldn't jump there so you try something else.

I would like to achieve this fast-paced platforming seen in this game. But if I want to achieve the same feeling, I must figure out how to implement dying that will not punish the player too much (sending them to a location that is 3 minutes of running).

Ghostrunner

„Ghostrunner offers a unique single-player experience: fast-paced, violent combat, and an original setting that blends science fiction with post-apocalyptic themes. It tells the story of a world that has already ended and its inhabitants who fight to survive“ [11].

The main concept of this game is to run and kill enemies with katana. The enemies fire bullets at you, and you try to dodge or return bullets to them. Bullets are very fast to give players a chance against them. The game implements slow motion. For a short period of time, the player sees bullets as if they were really slow, and the character is capable of faster movement. It has the same concept of levels as Super Meat Boy, where you repeat them until you beat them.

This game gave me a topic to think about, slow motion. It could help the player better control the character or plan what are the possibilities in critical situations. On the other hand, it might make the game which should feel fast paced feel slow paced.

3.2 World building

This section is about creating the world. The good world consists of a believable story, which should not be overshadowed by gameplay or be the only part of the game. Rather than that, it should complement the gameplay, and with it create a unique and enjoyable game. So I looked up an article on planning the world and then with its help wrote the main idea and background of my world.

How to plan the world [5]

1. Create the world
    - The world is the most important part, as it will shape how characters behave, how the backgrounds will look like (alien, futuristic,...) or simply what kind of people live there and how they behave
    - Well planned world guides you through other thinking processes and gives boundaries in implementation (for example, in art). It also ensures that the ideas will not get too spread out into many different things (it then becomes a mess)
    - How does this world look? -> Cities, continents, theme, time (stone age,...), and some specialties (magic,...)
    - How do people live there? -> nations, religion, are they mainly poor or rich?
    - Is there a huge danger? -> evil beings, war
2. Make the characters
    - Not only the player character, but also some important characters that could help progress through the storyline.
    - How old are they?
    - What is their past? – relations with family, lived in poverty, got injured,...
    - Do they have special skills? – mainly related to story and gameplay
    - How do they look and behave?
    - Add both sexes to create diversity if there is no reason to have only women or men.
3. Add the main plot.
    - What main character (hero) wants to achieve? – bring down some big bad evil guy, save their friend or help to end the war
    - How could they achieve their goal?
    - Is there someone else who could help them?
    - Are they the only ones that could do it? (probably yes, but they can race others, for example)
4. Add fill in between
    - Easy-to-understand events: blocked path, helping villagers, searching for lost pet,...

- Something that will move the story forward
- Something that tells more about main character or other important characters

5. Make a storyboard
    - Just draw how it could look like
    - Not finished drawings, rather than that lot of sketches.
6. Implement part of it to the game
7. Repeat
    - After a few iterations, it is good to look back at what you have done and think for a brief moment about it.
        - Does it go as planned?
        - Does it make sense together?
        - Is it easily understandable? (no complicated parts that were not meant to be complicated)
        - Does it follow the main story? (It did not turn off the path you had planned at the beginning.)

My world

Whole world is covered in darkness, the only light sources are things imbued with magic.

The sun was giving off light, warmth and magic. The last thing (magic) was discovered only recently. The Sun was slowly charging Earth with its energy, allowing some creatures to gain special powers. More and more people saw animals breathing fire, bending water, and using other elements to their advantage. Humans wanted to use this power, too, so they began to study different samples that were imbued with magic. Slowly harnessing the power of wanting to use more of it. One day a mysterious person showed up and created an orb. It was linked to the Sun, allowing for a better connection and better power transfer. After this, she was declared a Sun queen. Soon magic began to flow more and more into Earth, and things began to be imbued with stronger and longer lasting magic. New material was discovered, metal that did not contain any magic itself, but could amplify the magic powers of its wielder.

Peace did not last long; creatures from deep within were awakened by the large amount of magical energy and started attacking humankind. People resisted for a while, but stronger and stronger enemies started to appear. Worst case, the demon king woke up and wanted to absorb power stored in the orb. It was already too late; he started devouring the orb. Sun Queen managed to seal the demon away, but it was not without consequences. At the last moment the demon managed to release his power into the orb and corrupting it. The darkness began to spread. Corruption spread to the Sun via the ling that was created between it and the orb. Sun lost its ability to shine light. With her last breath, the Sun Queen deployed a large barrier that prevented stronger creatures and darkness from escaping. But as time went on, the barrier became weaker and darkness began to slip through. The weaker barrier even allowed weaker creatures to escape, which then plundered the world.

3.3 Story

With built up world story was next on the list. The story is told from the beginning of the new game to its end. It mainly focuses on player character or some significant character. In games, a large part of the story is created when players play the game. This is the best part of the games, the player creates the story! Another part of telling the story is through dialogues, books (or other written notes), flashbacks, and hidden things in the background.

3.3.1 My story

Main character wakes up in darkness and does not remember how she came to this point or much information about herself. The mysterious character who is imprisoned in some other dimension helps her escape from monsters. He does not know why or how, but he can see her in some kind of projection and speak to her. After that she found a village. When she talks to the hunter, she gets to know the basic story of the world. For example, the world is mostly covered in darkness, except for places that have some kind of magical street lamp. She seems to speak the same language as them, although she cannot translate some words – she seems to know it, but cannot just understand the meaning of it, maybe she knew them, but has forgotten them as some other things she forgot.

After asking in the village, she decides to go on a journey to recover her memories and help people. Her first quest is to find ring for a local shopkeeper; he agreed to give her something to defend herself on her journey. She is attacked by some creatures, and her friend helps her again (he gives her some kind of new power, shadow power). Then they go on a journey after finding out what this place is really about, and she tries to find some way to free her friend. In the end, she frees him and then he turns against her – he is some kind of demon and he was just using her.

3.4 Art

Art makes our games unique and memorable. Imagine a 2D multiplayer game with guns, where you shoot hordes of enemies but with the look of Mario (or too similar). People will look at this game and think: „Hmm, another Mario game.“ In fact, it has nothing to do with Mario, and, looking like it, it will be categorized as another game. So, the main point of art is to create a distinctive piece that players will at least not hate.

Good art can guide the player through the world, help them recognize objects that can be interacted with, or give a hint about what they can expect from the enemies. Good art shows enough details to inform the player, but not too much to distract them.

3.4.1 Pixel Art

Pixel art comes from old consoles and monitors, which had small resolutions and small memory to store assets. Modern devices do not need to care about these problems, but pixel art is still something. People love older times, which is why when they see some pixel art game they think:„Ahh, good old games[2].“ Pixel art might seem to someone awful, because it is blocky and doesn't have much detail, but many love it for its simplicity.

At first it might seem as easy job to make something with pixels – you don't need to be skilled in line making; not so much shading and sprites are small, so it will not take too

[2]Not confuse with GoG

long. In fact, it is not as simple as some may think. Yes, drawings are small, and when you get into it it takes less time than full hand-drawn art. But it is hard to make something recognizable when having a small resolution (64x64px) and with (much) larger resolutions, it starts to look less like pixel art and can be confused with hand-drawn. Lines are not exactly continuous, and one bad pixel might make it look like it is not straight, or it has more material in one place. However, programs exist for these, so making art is more about making than deciding where to put one pixel.

As mentioned above, resolution makes a difference in the final product. Too low, you could make your character box too high and people will not think of it as pixel art. Sprites can start from the size 32x32px and go even beyond 256x256px. In the end, it all depends on how it looks and its author(s) is(are) happy with the final result. Smaller resolution means blocky textures with less detail, and higher resolution means more details but less „old“ pixel art.

Figure 3.4: Example of pixel art from the game

3.4.2 Hand drawn art

Second popular style is drawn by hand. The artist draws it on paper or digitally (mainly digitally), and the assets are then inserted into the game.

Traditional

Traditional means on paper or made in reality. This style is very time-consuming. It can make your artwork different, but modern art programs can mimic traditional artwork to the point that the viewer will not be able to recognize it (if the artist is capable enough with software). So it is almost every time better to go digitally. Time to draw traditional animation comes from drawing it frame by frame, mostly without using components. That is, if a character has a sword artist can copy paste it and slightly modify it (rotate, shorten,...) they must draw it again and again. There exist styles where you cut your assets into pieces and take photos by moving them slightly or making them in the program, but they do not suit every game (mainly made for comedic games). In traditional art, artists are not limited to paper only; they can model assets from clay, trash, wires, etc. Then these games look unique and people can see the effort behind these.

Digital

Lot of games are made digitally, because the effort put into making, animating and importing assets is less than traditional. It is also less painful because many parts can be copied and tweaked to suit the pose or place. Digital programs also help the artist with their drawings. They add onion skin for animation, correction to lines to look more smooth, artists can take steps back (ctrl+z) and try more bold moves with this, and so much more.

3.4.3 Render from 3D

This style is slowly making its way into games. The artist makes a 3D model, which has bone structure (so it can be moved) and renders it 2 dimensionally with the program. This means an effective workflow, because they need to make only one asset and then animate it by moving parts. The result can look like pixel art or hand-drawn art, depending on the program used to render. It can also be scaled, and the animations can be smoother. Animations can transition between each other by continuing to move elsewhere, not just cutting to a new animation.

The down side of this approach is the program used. It dictates how the result would look, so it may not be exactly what the artist wanted. And if one program is popular, some games might start to look similar.

3.5 Problems

This section is about specific things that bother me in other games and how they could be corrected.

Too exact tutorial

This problem is about tutorial that tells player what to do, what to click on and so on. In the end, it feels like the game is playing itself and the player is there just to follow the text.

It seems pretty easy to solve. Just do not tell the player how to control their character. But it does not solve the problem with special moves, such as dashes or others, that have some special key or combination of keys. And some players might be completely new to gaming ant they will not know which keys to press to even move.

In my opinion the best balance would be to keep it minimal – tell only special moves. And if the player does not seem to understand what they should be pressing after some time, then give him a hint (if they will not move in the first couple of seconds, then it is better to tell him that they should use WASD or controller stick). These hints should be optional and are not required to progress.

Dialogues

Too much dialogues or too long texts in dialogues are annoying. After a few of them, I start to care less about the lore of the game and just want to skip them. Another annoying thing with dialogues is when it stops you from moving; if you chat with someone, why do you have to stand there like a pole?

Dialogues must be short and concise, and there must be as few dialogue windows as possible to tell the message. The character should be able to move during some dialogues.

This would be pretty hard for me because I want to hide some information in dialogues and my texts usually end too long. For this, I should go through them multiple times and try to keep it minimal.

Another solution is a silent game, which explains itself: absolutely necessary dialogues and a lot of information hidden in the background (not just background as background, but even in secret areas).

Also, if possible, the dialogues should reflect the actions of the players. Did the player complete the quest quickly? NPC should compliment him or give him another quest. If it took too long, they might not want the item and just refuse to give the reward, leaving

the player with a useless item. Did the player help some other villager, then give him a discount, and be friendlier with the player character. This would add a lot of dialogues which the player will not see during their playtime. Because of that developers might refuse to do this, it is more work which will probably not be seen. But this helps the player create a unique journey. It gives them the feeling that they contribute to the world and are part of it. And with it, NPCs might be seen as somewhat living creatures, not just a standard NPC who knows only its dialogues and does not react to anything.

Dead places

Long corridor without anything going on. The player simply presses one button and runs. Everything should be more lively if there is no reason to make it look like there is no one, to build tension, or just show dead place.

The game should catch the player attention every so often, so to achieve that enemies can move, friendly NPCs can do some activity in the background or something in the background can move (leaves, bugs, animals, etc.).

Contact damage

A lot of games implement contact damage, which is damage caused by touching enemies. It keeps the player from getting too close to enemies, but, on the other hand, it sometimes feels strange (especially when it seems that the enemy does not have a way of hurting you).

For this problem, I would like to almost completely remove contact damage, except when the enemy has thorns or other visible ways of doing damage. Most of the damage should come from attacks and not from hitting enemies.

Randomness

This one I have mixed feelings about. I hate that you cannot do anything to beat random things, but I can sort of accept them. Without randomness, the game would be good for speedrunning, where you want to know what will happen. But it is good to spice things up and make multiple playthroughs slightly different.

I would love to lower the number of random things by timing, a well-timed attack would do more damage, etc. But I will leave some things to random chance.

3.6 Mechanics

In this section, the mechanics of the game are described. This includes movement, skills, and attributes. It is a summary of them and a commentary on why they are important. Implementation of them is discussed in the next chapter.

3.6.1 Movement

This game is about building speed and maintaining it. With this, the movement needed to feel fast but also controllable. It also needed multiple-move mechanics, so it will not feel boring after a certain time when you only run or jump. Therefore, the set of moves consists of running, walking (for villages), jumping, dash, slide, and different jumps.

Running

For effect of fast running, the character should have a fast acceleration and slower deceleration, but not too slow to feel that the player still has control over the character.

3.6.2 Jumping

I wanted the game to look fairly realistic as far as basic movement is concerned. But this needed change. If an average person can jump say 1 meter high, then a character needs to jump half of its height (estimate). The problem is that my tiles are exactly that height, so the character would jump 1 tile high. I tested it on my prototype and it felt boring. A much better result was when the character was able to jump multiple heights. It does not look realistic, but in my opinion it looks better than that and is more pleasant.

Then there is wall jumping. When the player touches the wall, they can jump off it. This was needed to keep moving in a different direction pretty quickly. When the player runs and wants to go another way, the character needs to slow to zero and then accelerate in a different direction. This means it is pretty slow; however, when they jump to the wall it stops him from zero immediately, and they can do wall jump to quickly accelerate in opposite direction.

Multiple jumps were needed to help the player unlock shortcuts and help them in fights, as they can jump over attacking enemy.

Chapter 4

Implementation

This chapter focuses on implementing parts of the game in the Godot engine.

4.1 Prototype

Before working on the game, I started making prototype. This approach helped me better visualize my ideas and showed which things work together and which not.

The main point of the prototype is to make a rough sketch of the game. The word sketch is important here. It should have the main mechanics and the bare minimum of content. That is, no story, no flashy graphics, no music...[1]. It should serve as a sandbox, where you test important parts of the game – movement, main mechanics (interaction with objects for puzzle game, combat for action game, etc.). When it is not the final product, you do not need to worry about throwing away the code because it costs you maybe a week or month at most, not years. If done correctly, it should reveal failing points, not all of them, but the most visible ones probably yes.

So I have started by making the movement clean. Decide on the right values for acceleration, jump height, friction, and other things. Next on my list was the dash, specifically the eight-directional dash, which can be applied in eight basic directions[2] Then I made an attack, the duration of which was based on the type of weapon and the damage scaled with speed. This meant that different weapons behaved slightly different, although they had the same attack. For scaling damage with speed, I then implemented an energy bar that charges by continuously running. But when the player stops after a moment it will deplete itself. This mechanic would reward players who can move without stopping, which was my goal. To test if attacks work, I added crate. The final thing I wanted to test was different attack types (upward, ranged, magic).

The prototype helped me decide on the right movement curves that felt like the character is fast, but was not too annoying. It also added different casting times to attacks, which I did not plan, but I liked it.

[1] If it is important part of the game

[2] up, down, left, right, up-right, up-left, down-left, down-right

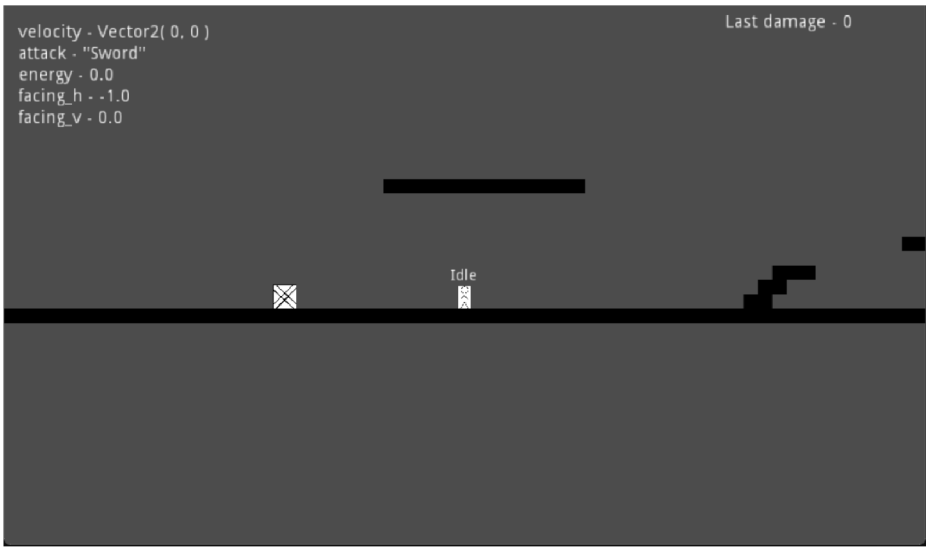

Figure 4.1: Screenshot of the prototype

Figure 4.1 shows how the game looked in one of the first stages. There are labels showing the debug information and there are no graphics.

4.2 Movement

In the prototype, at first, I have implemented movement in the player script. But after a while it became obvious that it will be hard for later adjustments, additions, or taking something off if it seems to be breaking the game. After searching for a while, I have found about using state machine for movement[10]. So, I drew a finite-state machine (FSM) for movement and started implementing it.

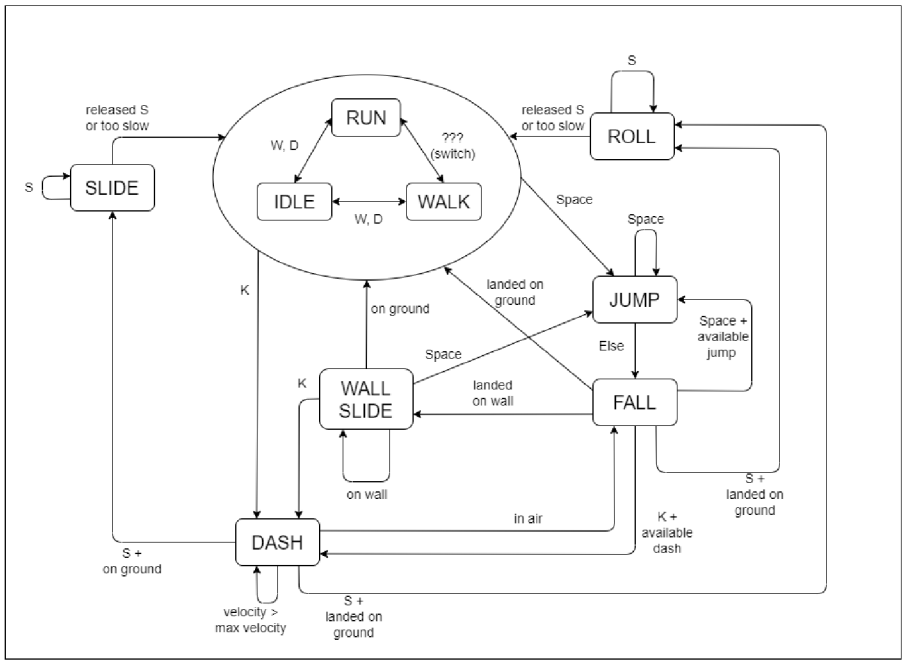

Figure 4.2: Finite state machine design

For the implementation, I searched for an article on the implementation of a finite state machine in Godot [14] and used the code to make my code more manageable. It helped to split the code into parts for each state (RUN, JUMP,...) and make the main script for the player much cleaner. With the help of the tutorial, I created the main structure that manages the states. It keeps the enum value of the current state, calls the function `process` from the state, and switches between states. With the basic structure, I could proceed to implement the states on my own.

4.2.1 Running and walking

Running and walking are basically the same; one is faster. Both of them consist of acceleration and deceleration. In physics, it could be described as the initial speed v_0 added or subtracted (depending on whether it is accelerating or decelerating) to acceleration a in time t.

$$v = v_0 \pm a \ * t$$

In Godot everything runs in a loop called `physics_process`. Due to that, we can add or subtract acceleration to the previous velocity value, so we do not need to worry about time. On the other hand we cannot say that the acceleration of the character is 2 m/s, because meters does not really exist there. And if they do, it might not feel good. This is where prototype comes in, because in there the acceleration value can be tested.

One big problem which is not considered in physics formula is max speed. Speed of light is considered to be the maximum speed of everything, but if formula is applied then there is no stopping it can go way beyond that value[3]. That is a problem, especially when people can run at much slower speeds. Therefore, the maximum speed is needed for each character. In Godot, we can compare this value with the actual speed by the functions `min()` or `max()`, which returns a minimal or maximal value of the two given.

[3] $c = 300000 km/s$

To slow down friction was used. It works basically the same as acceleration, but it can be changed based on the terrain on which the player is. Friction can also be used in acceleration and to simulate moving on ice. For that, friction must be in the range $< 0, 1 >$, where 0 is no friction and 1 is absolute friction. Acceleration is then multiplied by friction before it is added to the current speed.

Why walking

This game is about smooth movement at high speed, so why implement walking?

In villages there is no need to rush, it is a safe zone, where monsters cannot attack, so to visualize that character needs to be relaxed. Walking also allows for more precise movement because it is not so fast. The player can easily talk to NPCs or use items without overshooting or undershooting them.

For some players, it could be frustrating that they need to go at slow speed when they maxed out their maximum speed, acceleration, etc. That is why a switch key is needed that switches between running and walking states.

4.2.2 Jumping and falling

Player can jump when the character is on the ground. The function `is_on_floor()` checks if `CharacterBody` is colliding with floor. When the player presses the jump button (space bar), the characters jump. This is simulated by setting the y-value of the player speed vector to a high negative number (the negative is up). Gravity slowly applies its effect and adds itself to velocity, thus slowing the jump as it approaches its peak.

When the character reaches maximum height or the player releases the space bar, the state is changed to `falling`. When the space bar is released, the velocity is also divided by three, to break jump roughly at the place, but not too drastically (if it was changed to 0 it would look like there was an invisible wall).

Falling is just applying gravity. The player also wants to move the character even when it does not make sense in real life. This has become a standard in platformer games, allowing players to correct their mistakes and not overshoot or undershoot the place where they planned to land. Falling ends when the player land on the ground or on the wall.

Coyote time was used for better responsivity. It is a short period of time that begins after the character runs off the floor. At this time, the player can still press the jump button and make the character jump even if it is not standing on the ground. It has been named after a coyote in Looney Tunes, who often runs off the cliff and stands there in air for a while. It is there because the computer is precise and one pixel means that changing between can jump and cannot jump. Meanwhile, people are not that fast and often think they should jump, but they just fall because Coyote time is not implemented.

When a player falls from high place (or too fast), a bad fall occurs, making the player slow down and his energy is set to zero. The player can avoid a bad fall by preparing for the fall with looking down. When the key is pressed, the timer starts and the variable `rolled` in the player is set. On timer timeout rolled variable is reseted. If the variable is true on landing, the player will roll and no bad fall will occur.

4.2.3 Wall slide and jump

When player jumps or falls on the wall, the character starts to slowly slide down.

When the jump action is pressed, the state is switched to jump, but also adds force to repel from wall. The velocity then changes depending on the move key that the player holds. Moving from wall adds to repel force and moving into wall weakens it.

For checking if the player is on wall or near it, the Ray cast was used. It detects collision with objects without actually colliding, so the player can move more even if ray cast have already collided. To check if the Ray cast collides, we have a method `.is_colliding()`, which returns boolean value.

Coyote timer was also needed for the wall jump, even with raycast detecting it further from the wall. The problem was that when the player jumped from the wall and pressed the opposite key, it did not detect the opposite key. This detection is now also in the jump state.

4.2.4 Dash

Character can dash in 8 different directions. For that game stores which direction the player is facing and when player hits dash key, it gives character bigger velocity in facing direction. Dash is limited by maximum count to avoid the player flying through the air infinitely. The dash count recharges when the character touches the ground and has a lower velocity than the maximum. The second part of that condition is for dashes on the ground, so they do not recharge immediately.

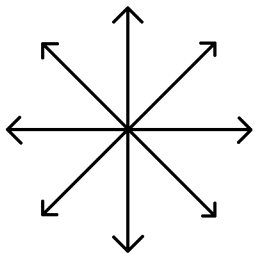

Figure 4.3: Directions of dash

Upon entering, the script checks if the player can perform a dash. If they can, the dash continues normally; otherwise, the player returns to the previous state.

4.3 Energy

Player gains energy when moving. This feature was implemented to reward players for continuous moving. Energy is added with the speed factor in mind, meaning that a higher speed adds more energy. When the player stops, the energy begins to drain.

Energy adds damage to attacks, so a player who moves will do more damage than one who stands next to the monster and moves minimally.

4.4 Fighting

Fighting enemies is one way to fill the big world and give the player something to enjoy while running to their next destination. To differentiate itself from other games, an energy bar was implemented. It charges itself while the player is running, and soon after the

character stops, it starts to deplete. The higher the velocity, the faster it fills. The energy bar adds to the attack power, making it stronger.

The player has a wider spectrum of weapon types, which they can use and change between them even in fights. Different weapon means different use times and different damage. Daggers deal fast but low damage. The hammer is slow to use, but when it hits, it deals massive damage. For now, all attacks have the same reach, but this will change with animations.

In prototype, there is a basic type of enemy, a box, which can detect the hit and the amount of damage it suffered.

Attacks also deal more damage with the full energy bar.

For each weapon, its animation, damage, and specific hit area were needed. At first, all was in one script, but after cleaning the states in movement, this was the next thing to separate. So, I remodeled the state script into the weapon script and moved all weapons to their nodes. All of them can pretty much call the same methods, and if they need something different, their script implements it.

Figure 4.4: Fight with a boss

4.5 Skills

Skills can bring a unique experience to the game. The players unlock them in the order they want. Skills are usually located in a skill tree, so I looked for ideas on how to create one in Godot and found a tutorial from JaviDev [8]. The skill tree consists of nodes that the player can buy to give them some upgrade. In the tutorial, the mouse is used to navigate the skill tree. My game uses mouse only in the main menu, and even there it is optional, so I coded it so that the player navigates with the keyboard. Each node has a dictionary consisting of „left“, „right“, „up“, and „down“, where the indexes of the respective nodes can be specified.

Another thing that I modified is purchasing skills. On pressing the buy button, bar starts to load, illustrating that buy is in process. The player can then change their mind in the middle of the process and do not buy the skill. The bought nodes call the Skills singleton function, telling it all the details about it that are needed for the player to upgrade.

All skills are under the `Tree` node, which is their main controller. It handles player input and moves between nodes and manages buying the node.

The skill tree can also be in multiple languages. For skills, a JSON file is created in the language folder, and for node represented with id, the name and description can be specified. When the skill tree is loaded, it loads the file with the language specified in Settings.

Skills are divided into four categories: Strength, Dexterity, Intelligence, and Shadow. Each category adds a slight bonus to related effects as a bonus to skill own power. This mechanic is hidden and is up to the players to figure it out.

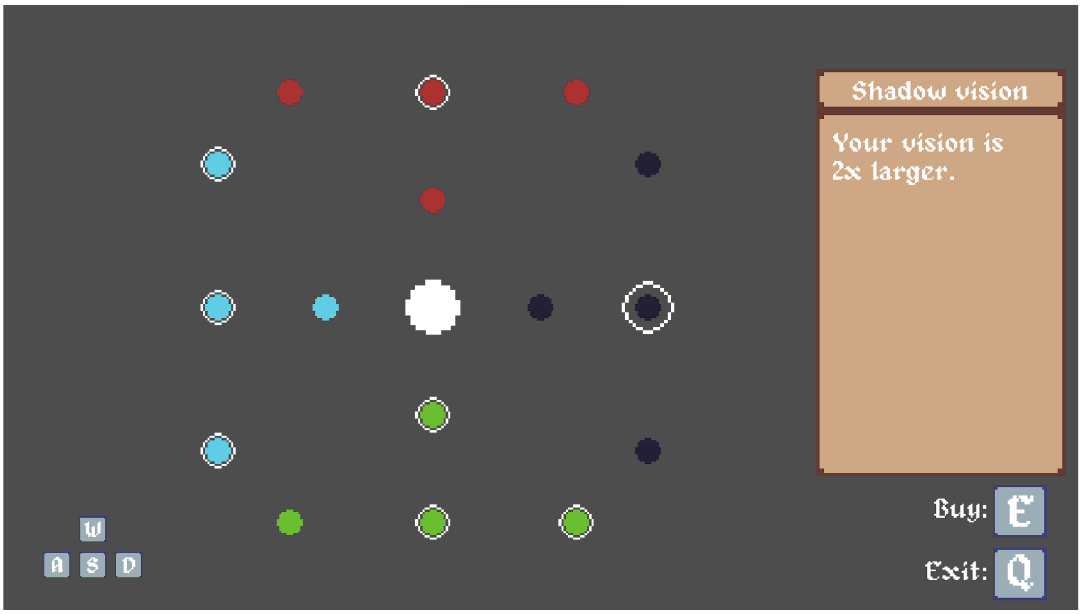

Figure 4.5: Skill tree

Figure 4.5 shows how the skill tree looks. It is divided into four quadrants by type (red = Strength, green = Dexterity, Blue = Intelligence, black = Shadow). Nodes with a white circle around them are already purchased. The black one with the larger white circle is currently selected. On the right is the name of the skill and its description. In the figure, hints to controls can also be seen and the white node, which is the starting node without effect.

4.6 Enemies

Fighting is good to fill in travels from place to place, but without enemies it is practically useless. The enemies come in different sizes, body shapes, and have different weapons. All this is to create a diverse world. Enemies usually have places to go and walk from one side to another, creating some kind of habitat feeling.

4.6.1 Small enemies

Forest is filled with wolves. They are running around and attacking the player on sight. They have their brains and decide what to do next. If they see the player, they stop and prepare to attack. If they are far from the player, they dash towards them and bite them.

Figure 4.6: Wolf

The enemy first updates its reference to the Enemies singleton, so that it starts to save its position and state. When the enemy is far from the player or dead, it does not process. When player is not in line of sight they walk from left to right and from right to left, until player is seen. At the moment of spotting the player they prepare to attack. Movement stops, to signal that they will attack. If they are close to the player, they will start using `bite()` otherwise they will use dash closing their distance to the player.

4.6.2 Bosses

Bosses borrowed powers from dark lords, making them several times stronger. They don't command their minions, but a lot of creatures concentrate around them to absorb some of the borrowed power. They protect significant points of interest or treasures. In the game, they are also kind of a skill check for the player to overcome and continue in the story.

The first boss is an ogre. With a big body, he moves slowly and attacks with his mace. He does not have fast reflexes, but his blow is fatal. He does have two swings, from above and around his body, so no place around him is safe. I have chosen this behavior for the first boss to test player fighting. It is not the hardest boss, but it can take a few attempts to beat him.

Figure 4.7: Ogre

As for the function, the boss moves from side to side like the wolf. For player detection, two Area2D were used to detect the player. Each has a shape of the final attack. On entering, the attack starts and the ogre starts to wind up its mace. Mace has two positions to signal to the player which attack would come. After that, the animation is played, and the player is detected by another Area2D. To ensure that the detection works even after entering the areas, disable and enable each frame.

4.7 Life system

Player is weak and enemies are deadly. With this player dies in one hit and enemies take few hits to be fully beaten. The player needs some way to balance this, that is, the rewind system.

When the character dies, one charge from the watches is used, and time starts to rewind back. The character saves their velocity for each frame in the `PlayerVariables` rewind array. When needed all inputs are disabled, player starts to load velocities from rewind array from most recent to oldest. The loaded velocities are then negated (they are the opposite), and the character moves back. To signal to the player that their character died, the pause icon appears in the bottom right corner. At the end of the rewind, the processing is still stopped and waits for player input to make a move. The player can calmly think about how to escape death and then begin to move. When the watch charges are depleted, the player is respawned at the last checkpoint [4.15.6].

4.8 Non playable characters

Non playable characters or shorter NPCs are as name suggest characters that the player cannot control. NPCs are usually some villagers who trade with the hero, give them quests, or just chat with them, giving away hints or parts of the story.

I have separated NPC into scene, which has exported texture, with this NPCs can be copied and their texture differs. Each NPC has a dialogue [4.8.1] which can be switched with `switch_dialogue()`. NPCs also have a `Area2D` (Interaction) around them. This area signals when body enters and the NPC can then show if they have dialogue with dialogue bubble sprite. The crossed dialogue will show if they do not have any dialogue loaded, meaning they will not appear in the player at the moment.

Each NPC that needs to say multiple dialogues has another node as its parent. This node serves as a controller, so they all inherit from `Dialogue_controller`. The controller class has one parameter called `prefix`. It differentiates between different characters when saving a dialogue that ended. On ready, the script loads the first non-seen dialogue to the character with the help of saved events. When the dialogue ends, `on_play` is called, which can run other events (show key). And then each of the controllers decides what script to play next. For quests NPCs can `wait` until the player does certain action or brings them an item after which they cancel waiting and continue with the given script. I wanted the NPC to have an unlimited number of dialogues and to be able to set them using `export_file`. I searched for listing all exports and found an answer in the godot forum[4], but the usage code is different from the one in the comments. So, assuming that it changes, I created `dialogue_exports_finder` export, which the scripts looks for and saves its usage code. After that script runs through property list once more searching for items with found usage code and items that have prefix „d_“, to eliminate `prefix`, `dialogue_exports_finder` and eventually others exports in future.

[4]https://godotengine.org/qa/38526/how-to-list-the-export-vars-of-a-node

Figure 4.8: Interacting with NPC

4.8.1 Dialogues

Dialogues are for communication between characters and NPCs, but they also inform the player about what is happening and tell them the story. Dialogues can be in the form of a dialogue bubble, a bubble with a picture, or in the form of sound. I have chosen the form of a buble with a text and took inspiration from the DevWorm video [6]. It uses `Nine patch rects` as a bubble, because they can stretch infinitely and fill the needed pixels. Inside are Rich Text Labels. I have added added text coloring to further distinguish between talking characters. Also, a way to switch between languages was added. All JSON files need to be in a specific language folder, and the dialog will change part of the path to load the correct file.

The goal was for the player to be able to move while using dialogues. For that reason, I created the autoplay option, which is an opt-out option. Specifying in the JSON file the „time“ option for each row allows switching to the next sentence without player interaction. The player can also skip manually if they are a quick reader or disable this option on the main screen to only play dialogues manually. However, most dialogues have the disable area, which ends them in whichever phase they are in. This is so that the next dialogue can be played and also to make sure that dialogues will not obstruct the view most of the time. After testing [4.14] I added a function to extend the time of the first script in the dialog. This gives the player some time to stop and read the dialogue that came from nowhere. On dialogues where the player interacts to start the dialogue (NPCs), this option is disabled.

Figure 4.9: Dialogue in game

4.9 Accessibility

Making the game accessible to more people results in more people who can possibly play the game. For that reason, I would like the game to be available in two languages, English and Czech. For this language, settings were added. The user can switch between available languages. This approach also allows for future expansions where everything can be copied into a new folder and translated.

For people who do not like to play on a keyboard a controller support was added.

The game is also welcoming to new players, with a tutorial [4.13.1] that teaches basic controls. The control hints switch accordingly to what the player is using. This can be seen in figure 4.10, where it is the same hint for moving around.

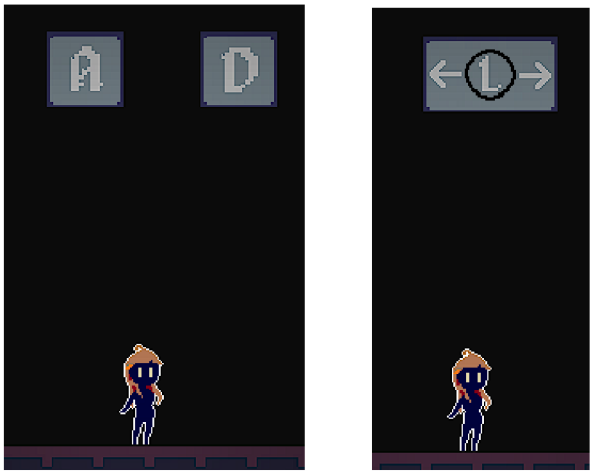

Figure 4.10: Same control hit for keyboard and for controller

Figure 4.11: Czech translation of the first dialogue

4.10 Heads-up display

Heads-up display (or HUD) shows important information for player (health, money,...). I went for a minimal design, which shows player a bar with energy (upper right corner), active buffs (upper left corner), and currently selected weapon (bottom left corner) as can be seen in figure 4.12.

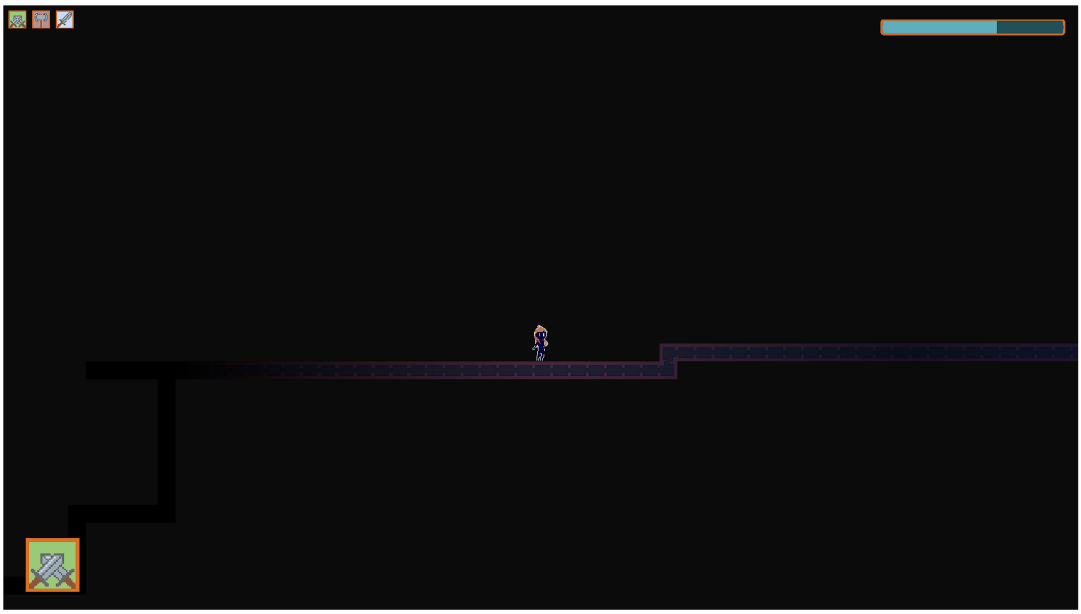

Figure 4.12: HUD

All elements are activated at certain places, so that the player is not overwhelmed by things he does not know about. For example, the energy bar is unlocked when the player is first told about how to collect energy and what it can be used for.

4.11 Saves

Saves are needed for the time when the player closes the game and reopens it later. Necessary variables are stored in AutoLoaded scripts. This means that all scenes can access them and make changes to them. Changes are saved upon switching to another scene (for player variables) or on change (events). Everything is stored in JSON files that are created in the `appdata` folder. When the player starts the game, everything is loaded from the files, so the player can start from where they left.

4.12 Controls

For controls I have chosen to implement two main ways of controlling the player. The first is with a keyboard and the second is with the Xbox controller. I have chosen these two mainly because I often use them for gaming. Due to Godot control handling other controllers than from Xbox should be usable. The only problem is that the key hints would be incorrect.

4.12.1 Layout

Efficient and easy to use layout is in my opinion the most important thing in control. With poor layout control, I often feel pain in my hands, so that is a problem that I wanted to avoid. Also, a good layout is easily memorable and, if possible, uses the unwritten rules (`WSAD` or arrows for movement,...), which the players already have memorized.

For the keyboard, I have chosen `WSAD` for movement. The player moves to the left and to the right using `AD` and control where are they looking (up or down) with `WS`. The

jump is on Spacebar and the dialogues and menu interactions are on E. The last commonly used key is Esc, which pauses the game. Regarding a special set of abilities, I have chosen the opposing J K L. This segment was chosen because it is the starting position of the right hand for typing (the index finger on J). The use of keys is: J – attack, K – dash, and L for changing weapons. These abilities were chosen because they are important and might be used frequently. Other keys are Ctrl – Toggling sprint, Q – Quick cast buffs and quit skill tree, and P to open skill tree.

In the controller, movement is performed on the left joystick, the same principle as on the keyboard. As for the commonly used keys for actions, A is for jumping and menu confirmation, B is for menu closing, and menu (three lines) is for pausing the game. Other keys and their functions are: B – attack, Y – toggle sprint, X – dash. LB – dialogues and interaction, LT – quick cast buffs, RB – change weapon, and RT – opening skill tree. For the A B X Y buttons frequently used abilities were chosen, the same as on the keyboard, except for the swapped change weapon and the toggle sprint. I have decided to put changing weapon on RB because if the player is fighting using B and wants to change the weapon in middle of B mashing, the RB is more accessible than Y. I did not use the D-pad for movement because I might end up using it for some other actions in the later process.

It may seem that there are too many keys and options, but I have tried to lead them one by one in the tutorial [4.13.1].

4.13 Tutorial

Tutorials are boring and can be tedious. I tried to make it as little pain as possible. Most of the tutorial hints are hidden behind the story and most of them can be skipped. All the hints also check if the player already did the action; it is meaningless to show them how to move when they already moved. This ensures that people new to gaming will be able to play the game, and people somewhat familiar with controls will not see all of them.

4.13.1 Movement

The first thing the player learns is how to move. Without moving, the player will not get to the other contents. The player gradually goes from walking to dashing and learning needed keys. Each action has its own place, which serves as a knowledge check, to which the player is able to use the action.

Figure 4.13: Jump hint

On figure 4.13 player did not know how to jump. So after a short period of time the game showed him the button that they should press to move further. If the player already jumped, the keyboard hint would not be visible. Also, if the player changes to controller, the hint will change accordingly.

In addition to standard moves, the tutorial also teaches some special ones. One of them is running. It shows the player that they can switch between running and walking, which is practical when the player wants precise movement (for example, for interaction with an NPC). Then it shows the player that if they falls from a high place, there is no fall damage, but it slows the character down. It teaches the player how to roll, although it is not needed and the player probably will not use it until later in the game. The player also learns how to interact with the object and that if the character dies, they are taken back in time. The last thing from the movement is the dash. The dash is shown only in one direction for the player to figure out the rest.

4.13.2 Fighting

Second part of the tutorial teaches how to fight. The player needs a weapon, so by talking with NPCs and doing some quest, they can obtain one. It teaches the player that there are quests and helps them better memorize how to interact. Now, with the weapon, the player can learn to swing it, which can be used against enemies in the forest where they will visit their next place. By witch they learn to swap weapons, obtain an energy bar, and learn to do an air dash. At this point, the player should know all about controlling the game, and the last challenge is the boss.

The player is held by dialogues and checks if they know all needed moves. Dialogues check if the player has already used the ability or if they picked the item. If the player did not pass the test yet, the dialogues will wait, and when they do so, another dialogue can be played, ensuring that the player will learn things in order.

4.14 Testing

Testing was a crucial step. It helped reveal errors not only in the implementation but also in the design. I was constantly testing the game to see if everything is as planned and fixed a lot of bugs. For example, bad dialogues, enemies stuck in place, etc. Another important part of the testing was the playtest by others. I gave the game to my friends and watched them play it and listened to what they were saying. In my opinion, this test was the most ideal. I could watch the players play the game and take notes, what is going well and what seemed painful to them. After the testing I asked them more questions, what they liked and what not. The first one allowed me to spot technical bugs, while the second revealed errors in design.

Watching others helped me correct these things:

- Too fast dialogues
  - If dialogues appeared from nowhere, players did miss the beginning
- Need for death screen
  - The players did not notice that they died and were taken back to the checkpoint.
- Wall Jump Help
  - The player struggled with wall jumps
- Hide mouse
  - Some players thought that they needed the mouse when they saw it on the screen and clicked on things.
- Missing dialogues
- Errors and misspells in dialogues
- Level design
  - Players were lost or could not make the jump
- Too much on screen

4.15 Singletons

Singletons [2.2.4] are ways of loading variables and making them available for all scripts. In the work, they were used mainly to store the data needed for saving. Almost all of them implement save, load, and reset. Save collects all important variables into one dictionary which is then saved in a file. The files are saved in `"user://saves/save.json"`, where `save.json` is a separate file for each save singleton. Load, reads the file, and loads data back into variables before using them by other scripts. Reset resets the variables to their initial values and then saves them so that they will be in the save files.

4.15.1 Player_variables.gd

This script saves everything related to the player. The main part are attributes, which are divided into several dictionaries: `base_atributes` – initial values, `bonus` – how much to add to base attribute, `atributes` – current values of attributes, `reset_atribtues` if the attribute should reset on `get_atrib()`, `atribtue_add` – if bonus is addition or multiplication. With these player attributes, they can be easily modified; just add them to `bonus` and the whole attribute will be upgraded. There are many attribute names, so the next step was to separate them into another singleton `Atributes.gd`, which contains them all in enum. This ensures that the name is not misspelled by one letter. Player variables also contain methods for getting, setting, and resetting attributes.

Another function of the Player variables script is to store a rewind array. And the last but not least function is monitoring if input method was changed (keyboard to controller or vice versa) and emits a signal so that the hint sprites can change to different.

4.15.2 Events.gd

Through playing the game, the player encounters different events. Picks items, performs certain actions, or hears some dialogue. Events save all of that. They also have functions for checking if an event has already happened and one that replaces older events with one new. `replace_events()` looks all event with given prefix and removes them from the array, then adds a new event instead of them. This method helps to clear the event array and relieve the search for events.

4.15.3 Settings.gd

This script saves Player settings. There are currently only two settings: auto-play dialogue and language. The reset function is missing because resetting the settings in a new game is not desired.

4.15.4 Skills.gd

Player can acquire skills [4.5] in the skill tree. When skills are purchased, the `acquire` method is called from the skill tree and adds the skill and its power. The power of the skill is added directly to `bonus` from the Player variables. Then there is the hidden node power, which is first calculated with `add_atrib` and then added to the bonus with `add_skill_bonus`. Adding node bonus depends on current weapon type, so the last function checks weapon type and with `get_power` the addition to `bonus` is counted. The player receives a stronger bonus from the dexterity nodes (green) when wearing the dexterity weapon (green – daggers), and the same goes for other types.

Skills also recounts bonus from skill nodes whenever the player changes their weapon.

4.15.5 Enemies.gd

When player dies they rewind the time with watches or dies. When rewinding, enemies need to be rewinded as well. All enemies use `update_ref` when ready. This method updates the reference to the enemy into an array so that it can be used later. Also, the last data is restored if the enemy is already saved in the `enemies` array. It restores the position and sets if the enemy is already dead. After that, the enemy is saved in the array `enemies`.

Each frame `_process` is called and enemies are saved. The distance is also measured from the player, and when the enemy is far away, its processing is stopped. That is, the enemy will not consume as much computing power as when it is fully functional.

4.15.6 Saves.gd

Saves manages the entire saving of games. It saves and loads other singletons and also makes respawn saves. When the player dies, they are spawned at the checkpoint, and with the character it resets all the attributes and variables from the time when they entered the respawn point. This ensures that the player can die to travel to the village with the item they just picked.

4.16 Miscellaneous

This section is about all other small things that this work implements, but are too small or have a smaller impact than previous sections.

4.16.1 Text font

The game has pixel art graphics, so I wanted a font that would match the graphics. I have found the Alagard font[5] that is in my opinion the perfect hit. The only disadvantage was that it does not support Czech symbols (č, š, í, etc.). So I added them with Glyphr studio online[6]. With this modification, the Alagard font can be used for Czech translation without printing Czech symbols in the default font.

4.16.2 Visual effects

For visual clarity, basic effects were added. When the player hits, the enemy damage circle starts to play its animation. Another thing that has animation is the attacks. When the player or enemy attack animation starts to play, signalizing where the attack hits.

4.16.3 Sound effects

The game felt empty even when everything is hidden in darkness and this sound emptiness creates the atmosphere to some extent. Using bfxr[7] basic sounds (jump, dash, fall from height, hit) were added.

4.16.4 Engine version

I have started the work on Godot 3.5, but at the beginning of March Godot 4 came out [7]. I have upgraded the following guide from Godot documentation[8]. Even with auto-updater, some lines have to be changed manually (for example, `@export`). The last version on which the game was compiled is Godot 4.0.2 stable (v4.0.2.stable.official [7a0977ce2]).

[5] https://www.dafont.com/alagard.font
[6] https://www.glyphrstudio.com/online/
[7] https://www.bfxr.net/
[8] https://docs.godotengine.org/en/stable/tutorials/migrating/upgrading$_t o_g odot_4$.*html*

4.16.5 Executable

I have exported the project to Windows machines, creating an `.exe` file. In the testing, I have found that I have been using the Vulkan render driver. This meant that starting the `.exe` file to run the game also needs to run Vulkan. On the computers on which I have tested it, only mine had Vulkan and others needed to be run in OpenGL3 via console. So, I created the second executable using OpenGL3, so that users without Vulkan could use this executable.

4.16.6 Lighting

The world is supposed to have nearly no light, so everything is covered in darkness. Light is only around the player, so that they can see on the jumps and on things that need highlight (fireplace, village).

Player has `PointLight2D` node with texture from Godot documentation[9]. For the darknes effect at the edge of the screen where the texture from `PointLight2D` did not reached `CanvasModulate` was used. The background is set to black with `ColorRect` inside `ParallaxBackgroun`, so it moves with the player.

Figure 4.14: Lighting around player

On figure 4.14 Light can be seen around the player and the edges of the screen are almost hidden.

4.16.7 Screens

In game there are multiple special screens that inform the player or they can interact there. The first screen that the player sees is the main screen (Figure 4.15), from which the player can change settings, run a new game, continue the last game or end the game. The main screen also serves as a pause screen. The information screens are as follows: Death screen and Finish screen. They inform the player about his situation.

[9]https://docs.godotengine.org/en/stable/tutorials/2d/2d$_l ights_a nd_s hadows.html$

Figure 4.15: Main menu screen

4.16.8 Level design

Level was designed to be fairly straightforward as it is a tutorial place, where the player should learn the basics and not be lost. Usually there are at most 2 places to go, and it is on the player to make the decision. The place they should go next is usually made more appealing or is mentioned in some dialogue. And if the player goes in the wrong direction, at most the player will die and have another try to go the correct way. There are some places where the player can skip some path if he knows the mechanics. For example, there is a way to get to the witch (house at the top), from the village without entering the forest (area on the right), which is the way the game tells the player to go. The almost complete layout can be seen below in figure 4.16

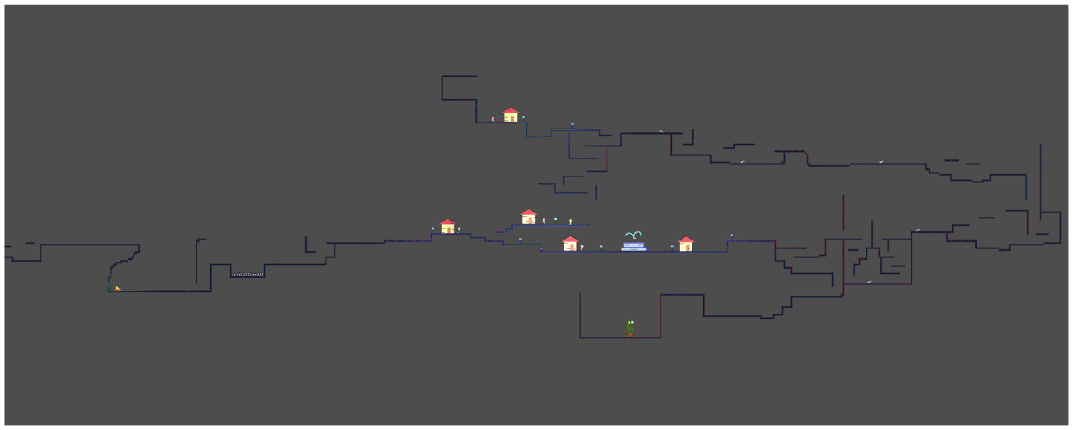

Figure 4.16: Level layout

4.16.9 Art

The sprites were made in Aseprite[10]. At first, I wanted to have more animations, but I did not get more. Only the animated sprite is the player character, with its Running and Idle animation. Animations are played with `AnimationSprite2D`. Animations were created with the help of AdamCYounis video [4]. I started by blocking the body elements and slowly animating the body to look somewhat correctly. Then added the texture for the player.

4.16.10 Out of bounds

Game detects when the character is out of bounds by checking their speed. If it is too fast, the player is probably out of the map, so the game respawns him.

4.16.11 Work correction

After writing this work I have corrected it using writefull.[11]

[10] https://store.steampowered.com/app/431730/Aseprite/
[11] https://www.writefull.com/

Chapter 5

Conclusion

The goal of this thesis was to design a 2D platformer game with the main aspect being speed. The game was designed with the help of researching well-known games in the metroidvania games (Hollow knight and Ori and the Blind Forest) and games that are fast paced (Super meat boy and Ghost runner).

The game tries to be as user-friendly as possible, with the tutorial helping new players and the ability to play the game completely in Czech. Players can also choose whether to play on the keyboard or on the controller.

The story is told through dialogues, which can be played automatically (the player does not need to skip to the next), or this option can be disabled if the player reads slowly and is missing dialogues. To create an atmosphere of place that does not have light from the sun, everything is covered in darkness, with players having a circle of light around them. This creates the feeling of unknown, where the player does not properly see what is in front of them and if there is some danger.

For replayability, the player can buy different skills; they are currently all free, so the player can test them all.

The game can be played on windows machines with Vulkan or OpenGL3 engine. It consists of the tutorial which shows the player how to move and fight, some quests, and fight with the first boss.

For the future expansion, I am planning on adding other sections to the game, expanding areas, story and enemies. The enemy AI would need some tweaking and upgrading. As for the look of the game I am planning on upgrading the graphics, to have more animations and effects.

Bibliography

[1] *Frequently asked questions* [online]. [cit. 2023-04-25]. Available at: https://docs.godotengine.org/en/stable/about/faq.html.

[2] *Scenes and nodes* [online]. [cit. 2023-04-25]. Available at: https://docs.godotengine.org/en/3.1/getting_started/step_by_step/scenes_and_nodes.html.

[3] *Singletons (AutoLoad)* [online]. [cit. 2023-04-25]. Available at: https://docs.godotengine.org/en/3.1/getting_started/step_by_step/singletons_autoload.html.

[4] AdamCYounis. *Pixel Art Class – Run Animation Tutorial [Part 1] | First Pass* [online]. 2020 [cit. 2023-11-8]. Available at: https://www.youtube.com/watch?v=LPBvrdJ_1a8.

[5] Derk. *How to write a good game story and get filthy rich* [online]. [cit. 2022-12-11]. Available at: https://paladinstudios.com/2012/08/06/how-to-write-a-good-game-story-and-get-filthy-rich.

[6] devWorm. *How to Create a DIALOGUE SYSTEM in Godot* [online]. 2022 [cit. 2023-04-26]. Available at: https://www.youtube.com/watch?v=7CCofjq_dHM.

[7] Godot Engine - Steam. *Godot 4.0 is released,...* [online]. 2023 [cit. 2023-5-7]. Available at: https://store.steampowered.com/oldnews/?appids=404790&appgroupname=Godot+Engine&feed=steam_community_announcements.

[8] JaviDev. *Camera movement – Skill Tree Tutorial [1]* [online]. 2021 [cit. 2023-5-7]. Available at: https://www.youtube.com/watch?v=HR-eoZS44vY&t=1s.

[9] Moon Studios GmbH. *Ori and the Blind Forest* [online]. 2015 [cit. 2022-12-04]. Available at: https://store.steampowered.com/app/261570/Ori_and_the_Blind_Forest/?l=czech.

[10] Nystrom, R. *Game programming patterns – States* [online]. [cit. 2022-12-17]. Available at: https://gameprogrammingpatterns.com/state.html.

[11] One More Level. *Ghostrunner* [online]. 2020 [cit. 2022-12-04]. Available at: https://store.steampowered.com/app/1139900/Ghostrunner/.

[12] Team Cherry. *Team Cherry – games* [online]. [cit. 2022-12-03]. Available at: https://www.teamcherry.com.au/games.

[13] Team Meat. *Super meat boy* [online]. 2010 [cit. 2022-12-04]. Available at: https://store.steampowered.com/app/40800/Super_Meat_Boy/.

[14] The Shaggy Dev. *Implementing state machines and the state pattern in Godot* [online]. 2021 [cit. 2022-11-8]. Available at: https://shaggydev.com/2021/11/15/state-machines-godot/.