Politechnika Wrocławska

Faculty of Computer Science and Management
Field of study: **COMPUTER SCIENCE**

Bachelor Thesis

Dungeon crawler game with procedural generated world in Unity

Piotr Wątorski

keywords:
video game, procedural generation, dungeon, Unity

short summary:
The paper focuses on the design and implementation of a computer game with a procedurally generated world in the "dungeon-crawler" genre. Gameplay mechanics, fragments of implementation, tests and development plan were described in the work.

Supervisor	Marek Kopel, PhD		
	Title/ degree/ name and surname		
The final evaluation of the thesis			
Chairman of the Diploma Examination Committee	.....	.....	…….
	Title/ degree/ name and surname	*grade*	*signature*

For the purposes of archival thesis qualified to: *

a) Category A (perpetual files)
b) Category BE 50 (subject to expertise after 50 years)

* Delete as appropriate

stamp of the faculty

Wrocław, 2020

Abstract

The main goal of this thesis was to implement a computer game about exploring a dungeon generated using procedural generation techniques. Thesis began with an introduction to the problem followed by a review of existing solutions. Afterwards, functional and non-functional requirements along with gameplay assumptions were formulated. The most interesting systems implementations and corresponding results were presented. The thesis also contains a description and outcomes of performed performance tests. The solution prepared as part of this thesis is a fully playable game.

Streszczenie

Głównym celem pracy było zaimplementowanie gry komputerowej o eksploracji lochów tworzonych przy użyciu technik generowania proceduralnego. Pracę rozpoczęto od wprowadzenia w tematykę problemu, po którym nastąpił przegląd istniejących rozwiązań. Następnie sformułowano wymagania funkcjonalne i niefunkcjonalne oraz założenia rozgrywki. Przedstawiono najciekawsze implementacje systemów i odpowiadające im wyniki. Praca zawiera opis i wyniki przeprowadzonych testów wydajnościowych. Produkt przygotowany w ramach tej pracy jest w pełni grywalną grą komputerową.

Contents

Introduction

The global gaming industry is developing dynamically. According to techjury.net [1] it was worth 151.55 billion USD in 2019 and is estimated to be growing exponentially at a rate of over 9% per year from 2020 to 2025. Video games have become one of the most often chosen forms of spending free time.

For the longest time games produced by major publishers were the most popular – this type of games is called “AAA” or “Tripple-A”. The other type are indie games, which are created in small teams, often by just one person. Market analysis [2] indicates that this branch owes its success mainly to its specific development process and rise of digital distribution channels, through which publishing games is much cheaper. New methods of financing the production are also significant. Crowdfunding and paid pre-release or test versions of the game allow small developers to dedicate more time to improving their product.

Goal of the thesis

The aim of the thesis is to create a computer game of the dungeon-crawler genre with a procedurally generated world. For this purpose, it is necessary to get acquainted with the characteristics of the genre, procedural generation algorithms and existing market solutions. On this basis, the product will be designed and implemented with the use of appropriate tools.

Scope of the thesis

The scope of the thesis includes the preparation of a project, specifying functional and non-functional requirements and assumptions of the game, as well as implementation in the form of a fully playable product. The developed game is to have mechanics typical of the dungeon-crawler genre. A unique underground maze will be generated for each game, using appropriate algorithms. The player will have ability to explore the said labyrinth. The essential graphics and assets will be prepared or acquired, and the necessary tests will be carried out to check the game’s performance.

1. Thesis subject

1.1. Dungeon-crawler genre

A "dungeon crawl" or "dungeon-crawler" is a type of scenario in role-playing games in which characters navigate an underground labyrinth environment (a "dungeon"), fighting various monsters, exploring, solving puzzles, and looting any treasure they may find. Article describing this topic [3] suggests that due to potential simplicity and the limited expectations most players have for plot and logical consistency in dungeon crawls, they are fairly popular in role-playing video games. Such games typically consist of endless procedurally generated dungeon terrain, randomly placed monsters and treasures scattered throughout the structure.

1.2. Procedural generation

Procedural generation is a method of creating data using algorithms instead of doing it manually. Usually the process is a combination of hand-made assets and algorithms synergised with computer-generated randomness (noise) and processing power.

According to a comprehensive topic overview [4] in video games, it is used to automatically create large amounts of content in a game. Depending on the implementation, advantages of procedural generation can include smaller file sizes, larger amounts of content, and controllable randomness for less predictable gameplay.

2. Analysis

2.1. Review of existing solutions

2.1.1. Darkest Dungeon

Darkest Dungeon was developed by Red Hook Studios and published by Merge Games. The project started during early 2013 and was released for Windows and OS X in January 2016, which followed a year-long early access development period [5].

A review in Ign [6] describes it as a role-playing game where the player manages a team of heroes and adventurers to explore titular dungeons and fight the creatures within (Figure 2.1). Each hero belongs to one of fifteen character classes, and has their own statistics and skills that can be upgraded over time. If a hero dies while exploring a dungeon, that hero is lost for good.

The game has received overwhelmingly good reviews and was released on all modern consoles and operating systems [5]. Over 650 thousand copies were sold just during the release week and 2 million copies on all systems worldwide - according to market analysis [7] and developers [8].

Figure 2.1: A battle in the Darkest Dungeon, access 2020.11.16 [9].

2.1.2. Legend of Grimrock

Legend of Grimrock was developed and published by a small indie studio Almost Human. It was released for all modern operating systems. Legend of Grimrock II, a sequel, was released in October 2014.

Nathan Meunier in a review for Ign [10] describes Legend of Grimrock as a first-person grid-based dungeon crawler game with tile-based movement and realtime game mechanics. Players control a party of one to four characters which they move through a 3D rendered grid-based dungeon, a style of gameplay popular in RPG games, as seen in Figure 2.2. Gameplay is a combination of puzzle solving and combat. Characters in the party gain experience by slaying monsters they encounter, allowing them to increase in level and skills to enhance combat abilities. Equipment is obtained through exploration and solving of puzzles throughout the dungeon. Many of the harder puzzles throughout the game are designed as bonuses, being optional to the progression through the dungeon but granting superior items and equipment for solving them.

The game received very good reviews and sold over 900 thousand copies worldwide [11][12].

Figure 2.2: First person view of a fight in Legend of Grimrock, access 2020.11.16 [13].

2.1.3. Rogue

Rogue was developed by Michael Toy, Glenn Wichman, Ken Arnold and Jon Lane throughout the whole nineteen-eighties. Rogue is one of the first computer dungeon crawler games ever made that paved the way in that genre and inspired many others. It has text-based console graphics Figure 2.3 and simple keyboard controls as described in article for edge-online [14].

In the game player, as a brave adventurer, explores an uncharted, underground dungeon. The goal is to reach the bottom of the maze, find a specific item and then return. The dungeon is filled with monsters trying to stop the player from achieving the victory. The game gets progressively harder the deeper the character ventures.

Rogue became popular in the nineteen-eighties among college students and computer users in general. In a ranking from 2009 [15] it was declared "6th Greatest PC Game" by PCworld, as it created a foundation for a noticeable part of the gaming industry.

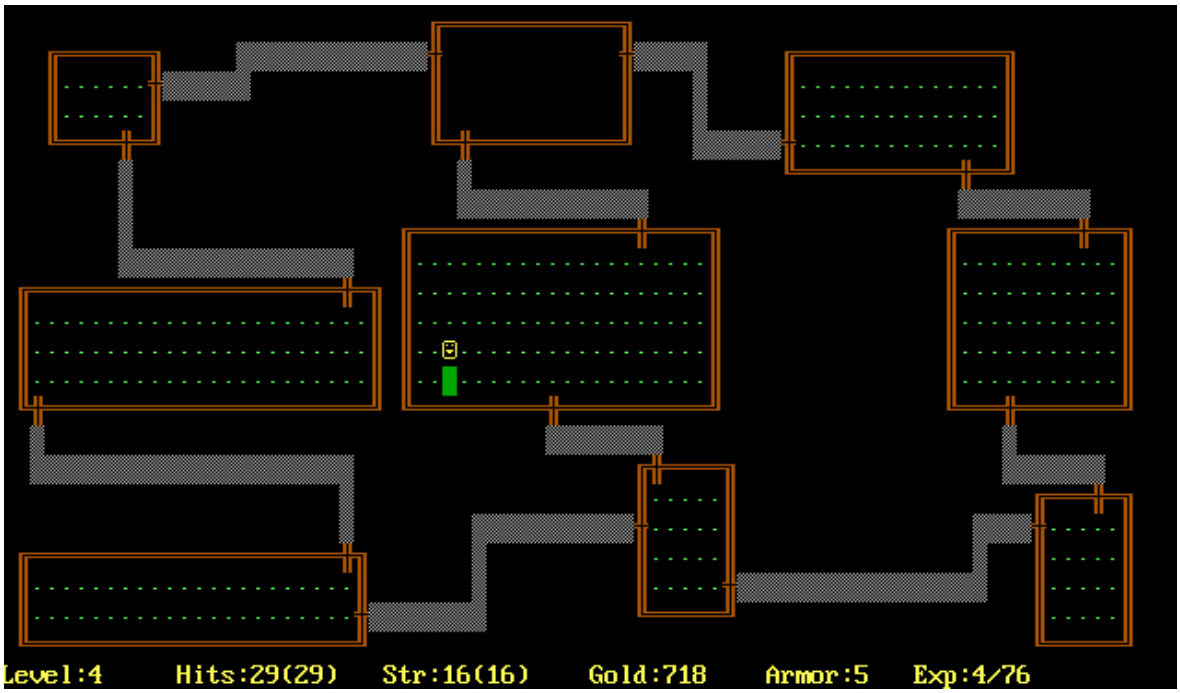


Figure 2.3: Gameplay screenshot of the game Rogue, access 2020.11.16 [16].

2.1.4. Minecraft

A computer game created by Markus Persson and developed by Mojang Studio. The first public test version was released in 2009. Thanks to social media before its official premiere in 2011, it became a world-phenomenon as stated by Matt Silverman in article for Mashable [17].

The ability to modify terrain combined with a procedural world generator were widely praised. Although the graphics may seem unappealing they allow for greater complexity in the terrain generation.

Tom Warren in an article for The Verge [18] describes the game as universally praised. Through over 10 years of its existence on the market it sold over 176 million copies worldwide and still has over 100 thousand active monthly users.

Figure 2.4: Screenshot of Minecraft world, taken 2020.11.17 (own source)

2.2. Technologies

2.2.1. Game Engine

A game engine is a software-development environment designed to build video games for consoles, mobile devices, and personal computers. Game engine typically provide a rendering engine for 2D or 3D graphics, a physics engine or collision detection, scripting, animation, artificial intelligence, networking, sound, threading and many more. This allows for easy and rapid development of games. According to a market analysis [19] there are two most prominent game engines on the market Unity and Unreal Engine.

The Unreal Engine is designed to create large AAA projects with help of big development teams. At the cost of lower accessibility it allows for near photo-realistic graphics. Due to the high amount of work required to finish a product in this environment it was determined to be unsuitable for a single developer.

Unity[1], a integrated environment for creating 2D and 3D games and applications was used to implement core gameplay mechanics. Newest version at the time was Unity 2020.1.4 and this distribution was used. Thanks to extensive documentation, numerous official guides and tutorials, friendly and intuitive user interface and the fact that it is free of charge for personal use it is the most popular game engine for independent creators and small studios according to praticle for PC World [20] and Unity documentation[21]. Over the years Unity gathered a very active community of creators. Thanks to their work the game engine can be altered and expanded with numerous free as well as paid plugins. Those extensions can be found in the *Unity Asset Store*[2].

2.2.2. Additional tools

The Unity Asset Store provides a great amount of models, scripts, shaders, animations, network infrastructures for multiplayer and even whole game modules. Using already made and well-tested resources allows small teams to make progress faster and improve the overall quality of the product with little-to-none problems. Developers are not restricted only to ready-made solutions and can use custom made assets. Unity scripts can be written in C# programming language or in one of many available *visual scripting systems*[3] provided in Asset Store. For this project only the text-based scripting will be used.

2.2.3. Programming environment

Visual Studio 2019[4] was selected as a programming environment. It has full support for Unity, in fact the game engine is natively distributed with this editor [22]. Assets created in Unity will automatically show up in Visual Studio solution and vice versa. One of the most important features is the game runtime debugging, that allows for standard-like debugging in a running game. Visual Studio is robust, well documented and very efficient environment for game development in Unity.

[1] https://unity.com/
[2] https://assetstore.unity.com/hp
[3] https://unity.com/how-to/make-games-without-programming
[4] https://visualstudio.microsoft.com/pl/vs/

3. Design

3.1. Functional requirements

After careful analysis of the most prominent games in the dungeon crawler genre as well as those having proceduraly generated worlds, key features were formulated as functional requirements:

- Procedural, semi-random dungeon generation
- Ability to explore the dungeon
- Ability to improve character equipment and skills
- Fighting system based on character statistics
- Turn-based moving and fighting
- Ability to replay a specific dungeon
- Seemingly endless dungeon
- Player should be able to see only what the character can

3.2. Nonfunctional requirements

The most important non-technical features were formulated as non-functional requirements:

- Smooth gameplay
- Intuitive interface and controls
- Mouse and keyboard support
- Ability to play offline
- Adware free

3.3. Gameplay

B. Bates in a book on game design [23] states that the modern game industry does not follow standard software development methodologies, as they do not apply to projects of this type. Instead of defining strict use-cases it was decided to describe gameplay and its structure:

Before the game starts player can select the initial seed to allow for replayability. After choosing the value player receives a proceduraly generated dungeon, which consists of rooms and corridors filled with enemies. Player views the game world from a floating camera attached to the main character.

Player can explore the dungeon and fight monsters by pointing at reachable destination or available enemy. Each adversary grants equipment and consumable items as well as experience to increase character skills. Enemies get stronger the further from the initial area player gets. Each movement and attach should cost a certain amount of stamina points that refill every turn.

After successful fight player can take opponents items to inventory and eventually improve

the characters' equipment.

Game ends when the player gives up or his character loses all health point. The latter situation can occur only during fights. Health can be replenished using consumable items and due to natural regeneration.

3.4. Graphics

Bearing in mind the requirements and the assumptions of the game, it was decided to choose a simple 3D graphics with consistent and tilable textures. Abandoning resource-heavy solutions allows for robust and quick generation of the game world as can be seen in Minecraft (see Figure 2.4).

3.5. UI design

Based on the previously formulated functional and non-functional requirements as well as the assumptions of the game, six interface designs were prepared: main menu, setup menu, game screen, character window, pause screen and death screen. Additionally in-game actions and interactions had their representations designed.

3.5.1. Main menu

The first screen (Figure 3.1) should allow player to start the game setup and close the application. There are two buttons *Start the game* and *Exit* allowing player to begin setup and leave the game respectively. The upper part of the screen is occupied by the product title. The menu should be semi-transparent to allow to view a 3D render of a dungeon entrance.

Figure 3.1: Main menu

3.5.2. Setup menu

The setup screen (Figure 3.2) should allow player to provide a base seed for the generator, allow to enter the dungeon and to go back to the main menu. There is one text input field and two buttons *Enter the dungeon* and *Exit* allowing player to start the game, provide the seed and leave the to main menu respectively. The upper part of the screen is occupied by the product title. The menu should be semi-transparent to allow to view a 3D render of a dungeon entrance.

Figure 3.2: Setup menu

3.5.3. Game screen

The game screen (Figure 3.3) should allow player to choose to move and attack, as well as to open the character window, finish turn and pause the game. Basic character status should also be clearly visible on this screen. There are five buttons *Move*, *Attach*, *Character*, *Finish turn* and *Pause*. The first two allow player to take action in the dungeon, that is to move and attack respectively. The *Character* button opens the character window. The *Finish turn* button begins the sequence of enemy movements and the *Pause* button stops the game and displays pause screen. Bars at the bottom show character status, that is health and stamina points.

Figure 3.3: Game screen

3.5.4. Character window

The character window (Figure 3.4) should allow player to view character equipment, inventory, health, stamina and skill points as well as current experience and level. On the right there are *Equipment slots* that allow player to chose what the character is currently using as his gear. *Inventory slots* are located at the bottom of the window. Player can store there items for later use. In the lower-right corner there are two slots for consuming consumable items and throwing unnecessary gear away. The rest of the window is taken by information about the character. This includes statistics: level, experience, health, stamina, attack and armor as well as skill section with available skill points and current skill distribution. The latter section allows to use free points to improve character performance.

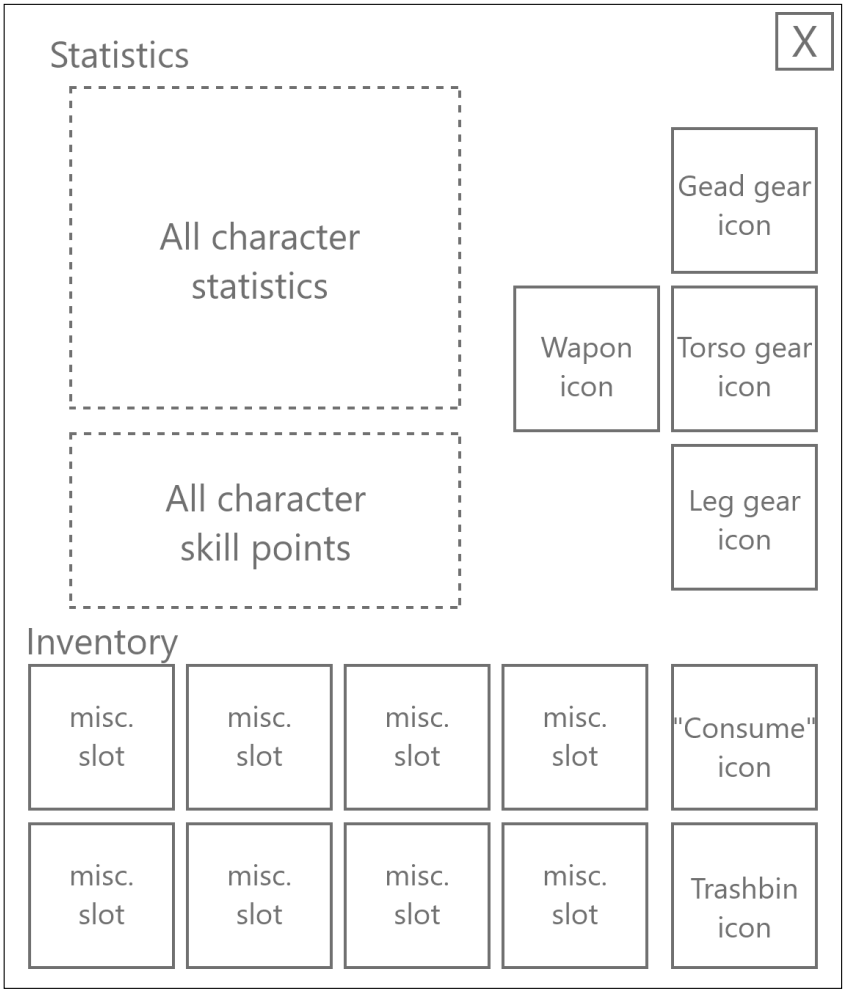


Figure 3.4: Character window

3.5.5. Pause screen

The pause screen (Figure 3.5) should allow player to return to the game and also to give up. There are two buttons *Resume* and *Give up* allowing player to resume the game and give up respectively. The upper part of the screen is occupied by the pause indicator title. The menu should be semi-transparent and partially blacked out to ensure the player that the game is paused.

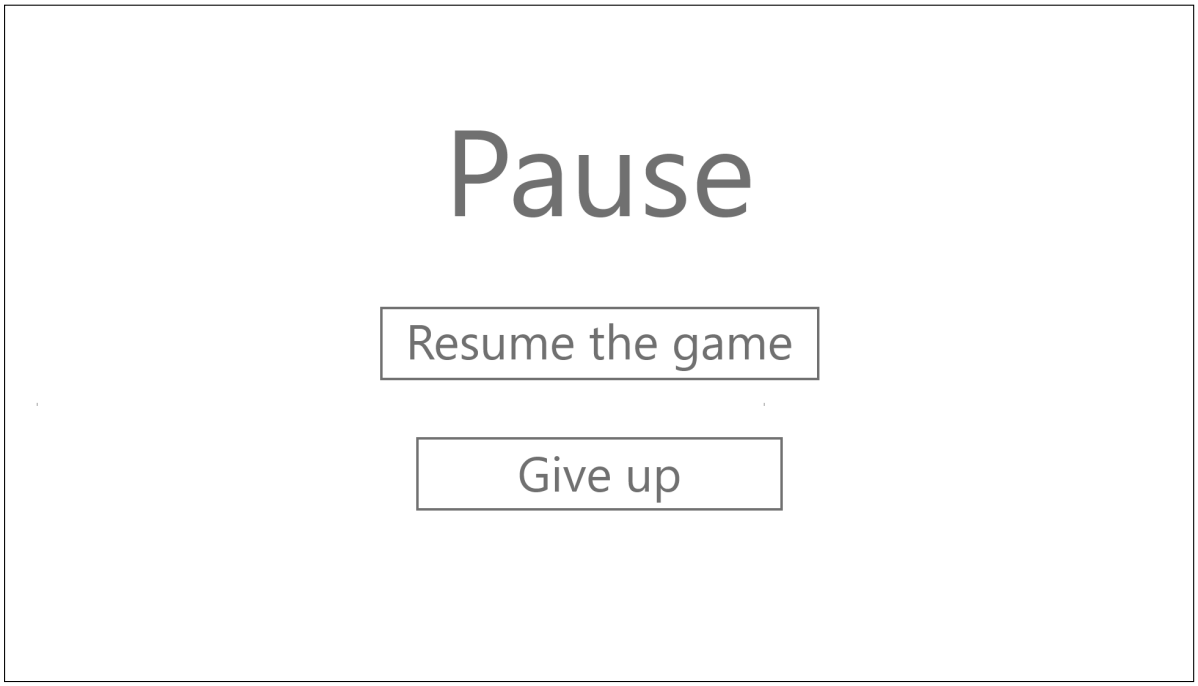


Figure 3.5: Pause screen

3.5.6. Death screen

The death screen (Figure 3.6) should allow player to go back to the main menu and to view playthrough information on amount of beaten opponents, tiles walked and similar statistics. There is one button *Exit* that allows player to leave to the main menu. The upper part of the screen is occupied by the game over indicator, while the middle section displays the aforementioned statistics.

Figure 3.6: Death screen

3.5.7. Action representation

Character can perform two in-game-world-actions, that is move and attack. Both should be intuitive and their impact should be viable before the action is taken. The in-game action indicators should be obvious and distinguishable, so that the player has no doubts on what will happen. There are two indicators: on the screen and on the dungeon floor. The first represents how much stamina will a certain action cost. The latter one shows exactly how the action will be performed, that is what path will the character take on the grid and which enemy will be attacked.

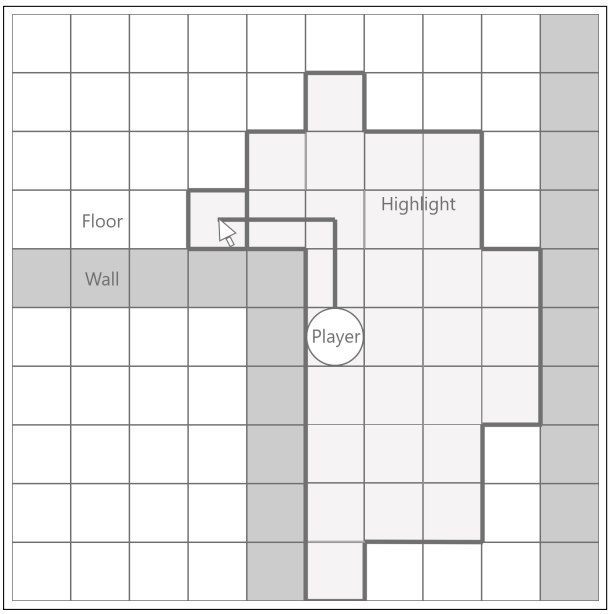


Figure 3.7: Movement indicator from top down view

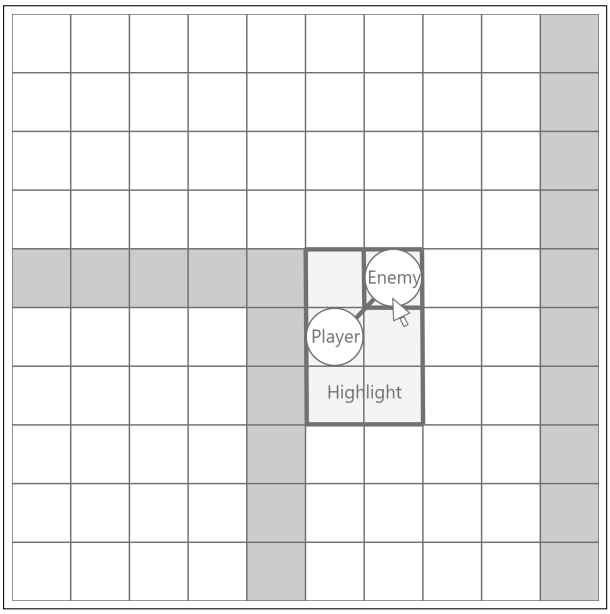


Figure 3.8: Attack indicator from top down view

4. Implementation

4.1. World structure

The 3D dungeon is generated proceduraly using a custom tool created for this purpose. The maze consists of corridors of universal width (6 tiles) and rooms of varying dimensions. The visible representations of the dungeon (*meshes*[1]) are based on a custom voxel system (see Section 4.2.5) and consist of blocks with different textures, although most of the volume is filled with empty space. A built in light and shadow system with proper tweaks allows to hide certain parts of the dungeon.

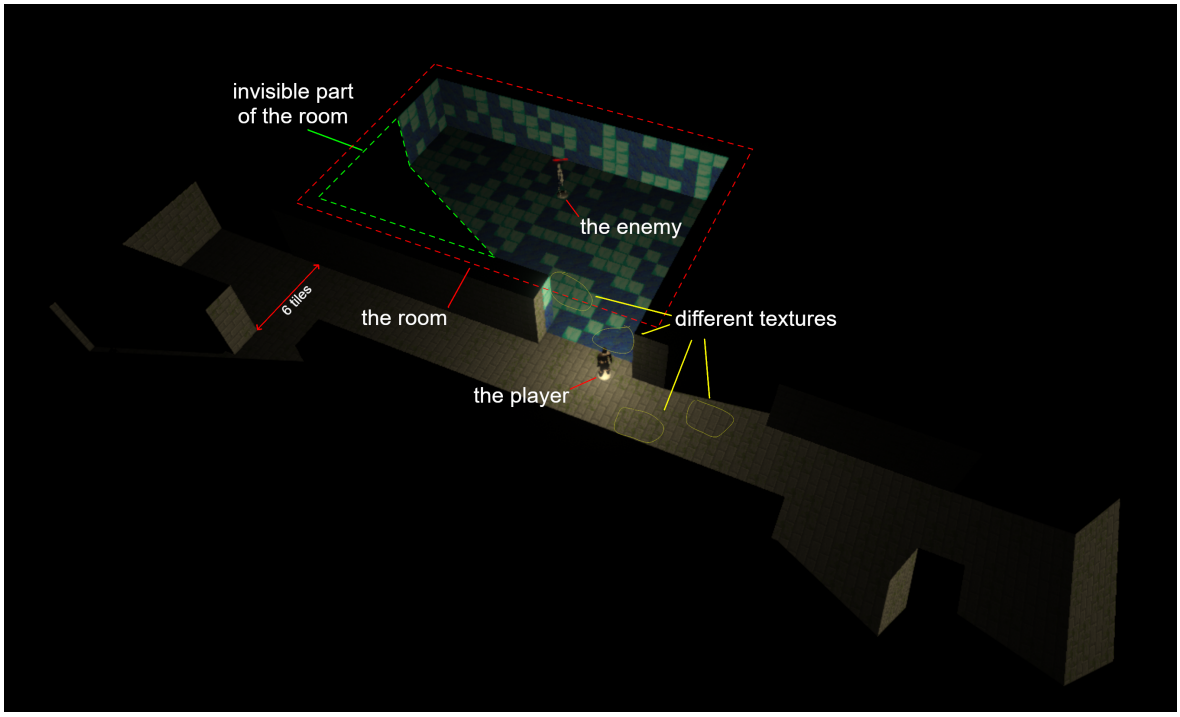


Figure 4.1: World structure overview

[1] https://docs.unity3d.com/ScriptReference/Mesh.html

Although blocks differ they all use only one texture file that combines all possible looks. Such file is called a *Texture Atlas* (see Figure 4.2). According to a problem overview [24] using a combined image can greatly improve the performance thanks to texture and object batching in rendering pipeline. The strange 'tiled' look of the texture is a solution for the the *texture bleeding* problem (see subsection 4.7.2). Thanks to the way the chunks and meshes are created (see subsection 4.2.2) the renderer can optimise the process even more.

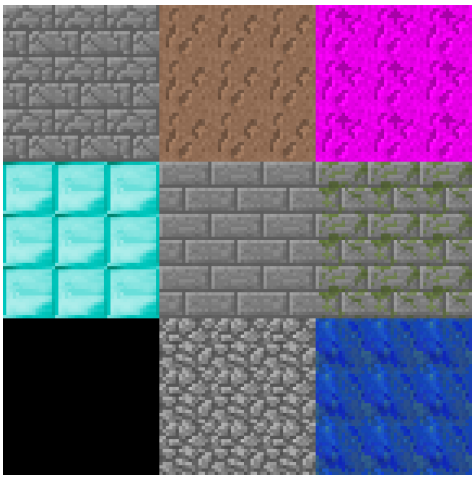


Figure 4.2: The atlas used in the project

4.2. Generation

4.2.1. Procedural generation

The main goal of the procedural generation implementation was to increase replayability by providing the player with virtually endless amount of unique dungeons to explore. Restricting player to play only a handful of man-made labyrinths would result in very low replayability and hence low enjoyment. The same goes for restricting the size of the maze by setting up some bounds on the maximal values. Using a procedural generator solves both problems and takes a huge amount of creative workload from the developer.

The generator and all classes in generation hierarchy (see subsection 4.2.2) use Perlin noise, a well known and proven algorithm for semi random procedural generation. Due to repetitive nature of Perlin noise over larger distances a native to C# *System.Random* class was used for additional controllable randomness. Every random number generated in any of the algorithms is created using combination of both.

4.2.2. Generation hierarchy

The hierarchy is divided into six logical classes: SuperChunk, Chunk, Structure, Cell and Tile. Additionally the Block class is used as a base for mesh generation. The hierarchy does not reflect the class inheritance, the arrows reflect aggregation relationship.
The Dungeon communicates with its elements through SuperChunks using prepared interfaces. Thanks to such division all unnecessary and possibly error-generating interactions were disposed. For example Tiles and Chunks do not interact in any way, as there is no need for such an interaction.

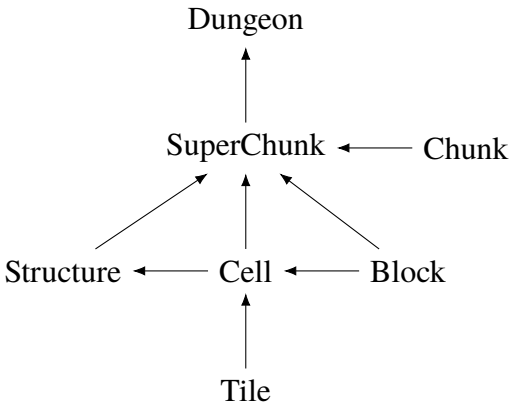


Figure 4.3: Dungeon logical hierarchy

- **SuperChunk**
  SuperChunk is a class for managing a collection of Chunks, Cells and Blocks. It consist of:
  - 16 (4x4) Chunks
  - 1024 (32x32) Cells
  - 262144 (256x256x4) Blocks

  SuperChunk is responsible for local[2] dungeon generation. It keeps track of all Cell and Chunk instances that should stay under it in the hierarchy. Additionally it contains an array of pointers to specific instances of the Block class. Methods responsible for retrieving certain objects at specific in-game locations were implemented for the ease of use.
- **Chunk**
  Chunk is a class responsible for the in-game representation of a certain part of the dungeon. Due to Unity Engine and typical user hardware technical limitations the limit of vertices per mesh is equal to 65535 as stated in the documentation [25]. For that reason it was decided to limit the chunk to 8x8 cells (64x64x4 blocks) which, for the worst case scenario (a 3D checker pattern), leaves every block with 3 vertices to spare. This class in derives from native Unity class *UnityEngine.MonoBehaviour*[3] that is needed if deriving class has to directly
- **Structure**
  The Structure is a helper class responsible for keeping track of cells that are inside rooms and for creating exits of said rooms.
- **Cell**
  The Cell is responsible for updating local pathfinding and for building the corridors and rooms according to SuperChunk instructions. It also uses a simplified flood algorithm to detect if it should be visable by the character and if enemies on it should be active.
- Tile
  The Tile is a backbone of pathfiding. It heep track of its neighbors and 'living' entities located directly over it. The second responsibility of this class is to display action indications on the dungeon floor (Action indicators see subsection 3.5.7).
- Block
  The block is an **abstract** basic building element of the game-world. There are two types of blocks: air and solid. The first one is transparent and does not contribute to mesh building, while the latter one has its representation displayed as a part of a certain chunk mesh. Only one instance of every block is initialized - all SuperChunks keep only the refferences to those few objects in their Block arrays. The few instances of this class that are present in the memory contain their references to the appropriate points describing correct texture position in the *Atlas* (see Figure 4.2).

4.2.3. Perlin noise and seed generation

Perlin noise is is a type of gradient noise developed by Ken Perlin. It was designed to look more natural than other algorithms at that time. An example is shown in the in Figure 4.4 and Figure 4.5 below:

[2] Inside its volume
[3] https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

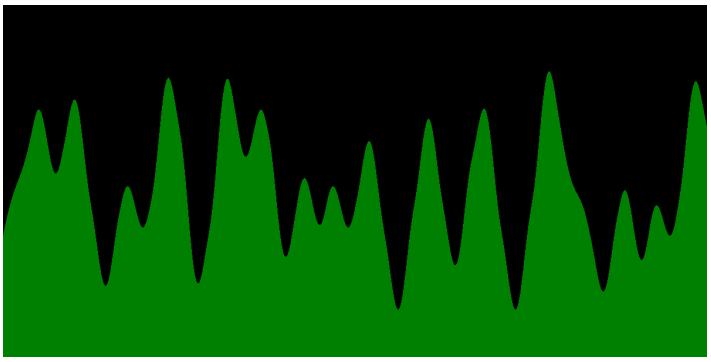


Figure 4.4: Perlin noise values (own source)

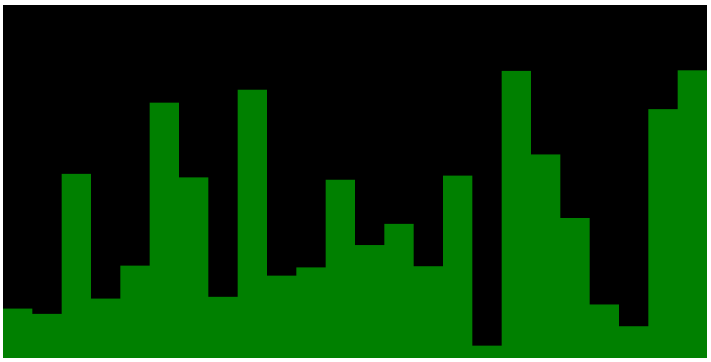


Figure 4.5: Random values (own source)

Perlin noise is used extensively in games with procedural generated worlds and to create unique textures. One of the most prominent examples is Minecraft. Its developer in his blog [26] describes the process that uses multiple layers of 3D noise.

For this project a native Unity Engine version of 2D Perlin Noise was used [27]. Due to the determinism requirement (see section 3.1) it is a perfect solution. Thanks to the fact that two coordinates describe a certain noise value it can be used as a base for location specific seed generation. Such a seed is later used in *System.Random*[4] class instances specific to each SuperChunk. For this to be possible a method was developed to change values from range [0.0, 1.0] to the range of System.Int32[5].

Provided coordinates x and y are multiplied by a value close to 1 to ensure more unique values, as the Perlin noise is limited by its permutation matrix size and loops back when the *'border'* is reached.

The algorithm samples 16 points. Each consecutive point is multiplied by a corresponding power of 2 and added to the final value. The maximal value that can be reached is always less than 2^{16} (25536) so it is normalized to match the range of Int32 in the return statement.

[4] https://docs.microsoft.com/en-us/dotnet/api/system.random?view=net-5.0

[5] [-2147483648, 2147483647],
see: https://docs.microsoft.com/pl-pl/dotnet/csharp/language-reference/builtin-types/integral-numeric-types

Listing 4.1: Position seed creation

```
public static int RandomInt(float x, float y){
    float val = 0;
    float multiplier = 1;
    float maxValue = 65536;
    float resolution = 4;
    x *= 1.001f;
    y *= 1.001f;
    for (float dx = 0; dx < resolution; dx++){
        for (float dy = 0; dy < resolution; dy++){
            val += multiplier * Mathf.PerlinNoise(
                                x + dx / resolution,
                                y + dy / resolution);
            multiplier *= 2;
        }
    }
    return (int)(int.MaxValue * (val / maxValue));
}
```

Listing 4.2: Combined seed creation

```
Seed = Perlin.RandomInt(superChunkPosition.X,
                        superChunkPosition.Y);
long longSeed = (((long)Seed) + ((long)Dungeon.Seed));
Seed = (int)(longSeed % int.MaxValue);
RNG = new System.Random(Seed);
```

The SuperChunk generates its perlin seed using its *SuperChunkPosition* where neighbouring instances coordinates differ only by 1 in only one direction. It is an additional way of conserving seeds. Such algorithm can create only one dungeon, so a second seed, provided by the player, is used in combination with the generated one so that in theory over 4 bilion unique mazes can be explored. The combined seed is used to create a System.Random instance, that later is responsible for the generation of the whole SuperChunk.

4.2.4. Dungeon structure generation

The dungeon generation algorithm can be divided into 2 basic stages *initializing* and *generating*.

The first one occurs while the SuperChunk is initialized and involves creating all its direct children. Arrays of Chunks, Cells and Blocks are created and initialized additionally sizes of rooms are determined and their regions marked. The algorithm is designed so that rooms occupy no more than $\frac{1}{3}$ of the space of the SuperChunk area. It starts with a certain amount of *'room material'* or *'juice'*:

$$juice = \frac{SuperChunk.width^2}{3}$$

Structure dimensions are randomly generated using System.Random class instance previously initialized with the combined SuperchunkSeed. The randomly selected dimensions depend on current *'room material'* and may be swapped depending on the local System.Random instance:

Listing 4.3: Structure size random generation

```
xSize = RNG.Next(2, Arit.Clamp(juice / 2, 2, 8));
ySize = RNG.Next(2, Arit.Clamp(juice / xSize, 2, 8));
if(RNG.Next(2)==0)
{
    Manipulator.Swap(ref xSize, ref ySize);
}
```

After the dimentions are set the algorithm finds all positions inside the SuperChunk that the Structure can fit so that the rooms are at least one tile apart. The last step is to mark affectec cells as visited and to connect inside and outside cells of room exits.

The second stage begins if any part of a SuperChunk has to be displayed in the game-world. In this part Cells are assigned connections to neighbours using a *randomized depth-first search* described in Listing 4.4.

Listing 4.4: Building local mazes

```
create_maze(start_cell)
    path_stack.push(start_cell)

    while (not path_stack.empty)
        curr_cell = path_stack.pop()
        neighbors = cell.get_unvisited_neighbours()

        if (not neighbors.empty())
            neighbour = select_random(neighbors)
            connect(curr_cell, neighbour)
            set_visited(neighbour)

            path_stack.push(curr_cell)
            path_stack.push(neighbour)
```

After the depth-first search algorithm finishes running the SuperChunk is in a state where every place is reachable from any other location (in the SuperChun volume). Because of that the maze may be too confusing or boring, as the player will eventually have go back from a very long dead-end section of the maze. This problem is solved by randomly selecting Cells (with no Structure on top) to connect them to their random neighbour. This guarantees that there are multiple ways that lead from one place to another.

Next the now-generating-SuperChunk initializes its uninitialized neighbouring SuperChunks. This operation is needed, because without it edge cells would not have neighbours to connect to in the following step.

As stated earlier player can access any point from any other point in the confines of a single SuperChunk, but there are no connections to its neighbours. This problem is solved in a way described in Listing 4.5. Due to the fact, that the connection are created only in two

Listing 4.5: Connecting superchunks

```
connect_in_direction(superchunk, dir)
    neighbour = superchunk.get_neighbour_in_dir(dir)
    if (connection_exists(superchunk, neighbour))
        return
    if (dir == 2 or dir == 3)
        swap(superchunk, neighbour)

    RNG = make_RNG_with_seed(superchunk.seed)
    edge_cells = superchunk.get_edge_cells(dir)

    for(cell in edge_cells)
        if (RNG.chance(0.2) == true)
            cell.connect_to(dir)

    cell = RNG.select_random(edge_cells)
    cell.connect_to(dir)
```

determined directions all Superchunk to Superchunk random choices are deterministic, as there is only one way to create such a connection.

The last generation step is to populate the Block array of the generated SuperChunk. Each Cell, knowing its connection status as well as structure status is able to rise walls in proper places using a SuperChunk method for filling a specified volume.

The final stage consists of generating and assigning meshes based on the aforementioned Blocks array. This step is descriped in the following subsection.

4.2.5. Voxels and meshes

Due to technical limitations regarding the maximal mesh vertex number (mentioned in subsection 4.2.2) all SuperChunks are divided into 16 Chunks. Each chunk is assigned a mesh generated on the base of the Block array in its parent.

To create said meshes a voxel-to-mesh algorithm was implemented. It took over 300 lines of compact code, but it boils down to the following algorithm:

Listing 4.6: Mesh generation algorithm

```
generate_mesh(blocks)
    mesh = new mesh
    for(block in blocks)
        submesh = new mesh
        for(side in block.sides)
            if(is_visible(side))
                submesh.add_surface(side.position, side.texture)
        mesh.merge_with(submesh)
    return mesh
```

Thanks to the derived properties of the MonoBehaviour class Chunks can have their representation present in the game-world. It provides all possible methods needed for manipulating, transforming and translating objects. This combined with possibilities given by the optional *UnityEngine.MeshFilter*[6] and *UnityEngine.MeshRenderer*[7] members of a MonoBehaviour instance provides a perfect set for showing individual parts of the Dungneon using previously generated mesh.

4.2.6. Optimisation

Each operation related to the dungeon generation takes time. Surprisingly creating meshes is much more resource heavy than any other stage of the process. The obvious solution is to just move the time consuming tasks to a parallel thread. For even more optimisation both processes are implemented asynchronously.

Due to technical limitations the appearance of a given MonoBehaviour can be altered only in the main Unity thread, preferably in the *MonoBehaviour.Update*[8] method. For that reason a scheduler was implemented for managing communication between the threads.

[6] https://docs.unity3d.com/Manual/class-MeshFilter.html
[7] https://docs.unity3d.com/Manual/class-MeshRenderer.html
[8] https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html

4.3. Items

Most role playing games have some form of an inventory system. The general purpose of such mechanic is item management however might it behave differently in a each title. For this project a traditional RPG-style, slot-based inventory system was implemented. It is one of the most intuitive equipment types. The player can see and interact easily with items. Standard drag-and-drop functionality was designed and implemented into the system along with a swapping mechanic.

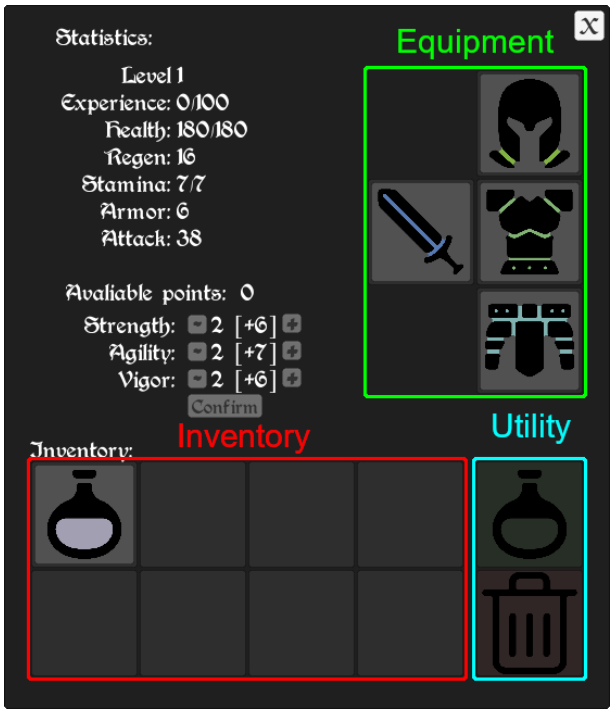


Figure 4.6: The Character Window

4.3.1. Definitions

- **Item** is a game object that is not a direct part of the game-world nor is it a *living* entity, but it has properties that might affect player or an enemy.
  - **Gear** consists of items that can be worn, that is all armor parts as well as weapons
  - **Consumables** are items that can be used only one time and, in contrary to gear, have temporary effects.
- **Inventory** is a virtual space that every living entity can use to store any type of items.
- **Equipment** differs from Inventory in one aspect, specific items can be stored only in corresponding slots. That is: helmet on head slot, armor on armor slot, weapon in a weapon slot and lower armor in legs slot.
- **Utility zone** - in this project it destroys or uses (consumes) items. There are only two slots, one for consuming items and second for destroying them.

4.3.2. Generation

Items are proceduraly generated by the entity that they are going to spawn on. Due to the determinism requirement the process is similar to the initial SuperChunk generation.

For each new entity a temporary System.Random instance is initialized with a combination of initial and location seeds.

Each item is generated simillary to how the Structures are generated. The empty Equipement class starts with a certain amount of *power* that it distributes randomly between each items' statistics (see subsection 4.4.2) using previously mentioned System.Random class. The initial power is linearly dependent on the distance from the dungeon centre:

$$power = \frac{distance}{8} + 6$$

Such a distance to equipment and hence enemy power ensures that the player always has a worthy opponent to face in the dungeon, no matter how much gear and skills he accumulates. Together with item statistics a random color is generated and applied to the item visual representation, so that it is easier for player to distinguish between individual items in the inventory.

4.4. Entities

Entities are in general all game objects that can move and use or wear items. There are only two types of entities: the character and enemies. Each entity has certain statistics, that are dependent on the equipment it possesses.

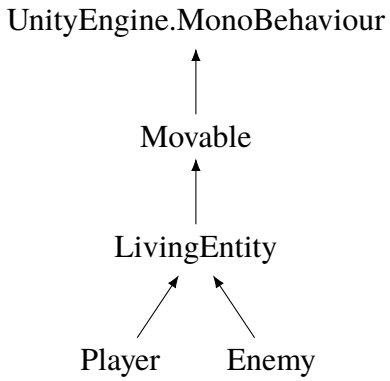


Figure 4.7: Entities class hierarchy

4.4.1. Movable

Movable is a class responsible for requesting and interpreting walk paths as well as displaying the movement in the game world. It derives from *UnityEngine.MonoBehaviour*. Due to asynchronous nature of pathfiding implementation (see section 4.5) a method for accepting pending paths was designed. It uses thread-safe implementation of C# *System.Collections.Generic.Queue*[9] for unpacking incoming movement orders. The main functionality is located in the *Update* method. There, during each frame the object moves in direction described by a current suborder, until it reaches the subgoal, then next suborder is dequeued. This continues until the object reaches its final destination.

4.4.2. Entity statistics

- **Health and MaxHealth** are measures of the entity well being. If the first value reaches 0 the entity dies.
- **Regeneration** is a measures of how much the entity's health is restored each turn until reaching MaxHealth
- **Stamina and MaxStamina** are entity action points. If the first value reaches 0 the entity cannot perform any stamina consuming actions, that is: moving and attacking. This measure refreshes every turn.
- **Strength** directly impacts the attack statistic of an entity as well as how much heath the entity has.
- **Agility** directly impacts the amount of action points an entity can use in one turn.
- **Vigor** directly impacts the amount of health and health regeneration of an entity.
- **Armor** points negate incoming damage.

[9] https://docs.microsoft.com/pl-pl/dotnet/api/system.collections.generic.queue-1?view=net-5.0

- **Attack** points represent the amount of damage an entity can deal in one attack. Each attack costs 5 stamina points.

The exact values are calculated using following equations:

$$attack = weaponDamage + strength * 4$$

$$armor = helmetArmor + upperArmorArmor + lowerArmorArmor$$

$$maxHealth = 100 + (strength + vigor) * 5$$

$$regeneration = vigor * 5$$

$$maxStamina = 5 + \frac{vigor}{4}$$

4.4.3. Living entity

LivingEntity is a class responsible for keeping track of entity statistics (see subsection 4.4.2). The class derives from *Movabe* and overrides its methods (while keeping original functionality) to allow easier management of stamina: If a movement order gets dequeued from order queue it firstly goes through the *LivinEntity*, where a propper amount of stamina is 'used'. The class also contains methods for receiving damage, performing attacks and for managing death.

4.4.4. Character

The character is an entity that is controlled by a player. The class has all necessary player-character interfaces implemented:

- Movement
- Attacking
- Action indicators (see subsection 3.5.7)
- Showing inventory

It derives from *LivingEntity* and appends some of the methods: The *OnDeath* method instead of destroying the gameobject finishes the game and initializes the death screen. All functions related to movement and attacking save corresponding statistics to show them on the death screen.

4.4.5. Enemy

Enemy class derives from *LivingEntity* and implements the necessary methods for receiving orders from the *Hivemind* (see section 4.6), that controls it. Additionally it overrides the *OnDeath* from *LivingEntity* and appends its functionality to give the character some experience for killing it.

4.4.6. Inventory

Each entity has two storage instances: inventory and equipment. The enemies spawn with incomplete item set, while the character initializes with a full gear and one consumable potion. Character item management is done with typical for RPG-games, drag-and-drop system. It uses classes and event interfaces provided by Unity. In addition to simple dragging items to empty positions a swapping method was developed.

4.4.7. Fighting system

The fighting system is based on entity statistics. Entity can choose a tile both fully and diagonally adjusted. If the space was occupied by a LivingEntity instance the action is performed. The attack power is calculated as a difference between attacker attack statistick and receiver overall armor. If the latter one is lower than the first one the reciving entity loses corresponding amount of health. If enemy dies character receives amount of experience points equal to sum of all enemy skill values. If player dies the death screen is displayed and game ends.

4.5. Path finding

Pathfinding relies on Tiles and a Pathfinder class to work. It is responsible for building path maps and for showing character range. Both processes work on separate threads and use the order scheduler to function. The Pathfinder implements a method for initializing appropriate Cells and Tiles. There are two types of pathfiding implemented A* pathfinding and a custom flood algorithm.

4.5.1. A* pathfinding algorithm

The A* algorithm was implemented. It is a graph traversing algorithm, that finds the path by looking for least costly neighbour with the lowest heuristic score (in that case distance to finish) and moving onto the neighbour. It is used only for Enemy movement.

For the algorithm to work a sort of a graph has to be created. It is based on Tile instances. Each tile knows its neighbours, both diagonal and straight. It also keeps track of 'walkability' of current node - if there is some entity or a wall over the tile it is not 'walkable'.

The *Pathfinder* instance finds Cells in a sqyare of 11 on 11 cells with the middle on the player position. The set is compared with set from precious iteration. Based on the results:

- **Uninitialized cells have their tiles initialized** by making tiles know their neighbours, setting parents to null, setting score to maximal possible value and checking walkability
- **Initialized cells that are outside of the zone have their tiles delegated to a destroying thread**
- **Cells in both sets have their tiles reset** by setting parents to null, setting score to maximal possible value and checking walkability.

When the graph is all set up the Pathfinder instance executes a following algorithm:

Listing 4.7: A* pathfinding algorithm

```
function reconstruct_path(current)
    total_path = {current}
    while current.parent is not empty:
        current = current.parent
        total_path.prepend(current)
    return total_path

function A_Star(startTile, finishTile)
    openSet = {start}
    startTile.gScore = 0
    startTile.fScore = distance_between(startTile, finishTile)
    while openSet is not empty
        current = get_node_with_lowest_fscore(openSet)
        if current = finishTile
            return reconstruct_path(current)
        openSet.Remove(current)
        neighbours = current.get_walkable_neighbours()
        for each neighbor in neighbours
            cost = walk_cost(current, neighbor)
            new_gScore = current.gScore + cost
            if new_gScore < neighbor.gScore
                neighbor.parent = current
                neighbor.gScore = new_gScore
                distance = distance_between(neighbor, finishTile)
                neighbor.fScore = neighbor.gScore + distance
                if neighbor not in openSet
                    openSet.add(neighbor)

    return failure
```

The path is then enqueued in the scheduler for a specified *Movable* instance, that takes care of the movement.

4.5.2. Custom flood algorithm

For the player movement and for the range and path to be displayed a custom flood algorithm was implemented. It works in similar fashion to the A*, but instead of a target it has maximal energy.

For this algorithm the Pathfinder initializes a smaller graph for a 7 on 7 Cell square. The area is smaller for optimisation reasons, as the weighted flood algorithm for pathfinding has a complexity of over $O(N^2)$. With the selected size the whole region is mapped in less than 100 ms on a standard PC.

With a graph initialized the Pathfinder selects a Tile on which the player stands on and performs the flood in the folowing way:

Listing 4.8: Custom flood algorithm

```
function flood(startTile, energy)
    open_set = {startTile}
    while open_set is not empty:
        tile = open_set.pop()
        tile.reachable = true
        neighbours = tile.get_walkable_neighbours()
        for neighbour in neighbours
            move_cost = tile.cost + cost_between(neighbour, tile)
            if move_cost <= energy
                if move_cost < neighbour.cost
                    if not open_set.contains(neighbour)
                        open_set.add(neighbour)
                    neighbour.parent = tile
                    neighbour.cost = move_cost
```

This creates a set of Tiles where each Tile has its parent set to the next node on a shortest path to the position where character stands. Thanks to the *Tile.Reachable* flag it is easy to show the reachable region on the dungeon floor. All visible Cells are ordered to refresh the grid display on corresponding Tiles. Player path is displayed using *UnityEngine.LineRenderer*[10].

4.6. AI

Enemies have to be well coordinated hence a hivemind-like artificial intelligence was designed to centrally manage monster combat and movement. Every created Enemy is registered in the Hivemind and put in a list of managed drones. If a given Enemy is defeated it is unregistered from the AI. The Hivemind runs a separate thread that distributes orders. It is an implementation of a algorithm.

[10] https://docs.unity3d.com/Manual/class-LineRenderer.html

Listing 4.9: Custom flood algorithm

```
while game_is_running()
    if enemy_turn == true
        enemy_turn = false
        active_enemies = {}
        for enemy in enemies
            cell = enemy_get_cell()
            if cell.active
                active_enemies.add(enemy)
        active_enemies.sort_descending_by_energy()
        while active_enemies[0].energy > 0
            if enemy.can_attack_player()
                enemy.attack_player()
            else
                path = find_path_to_player()
                if path.energy > enemy.energy
                    path.shrink_to_energy(enemy.energy)
                pathfinder.reserve_tile(path.finish)
                enemy.move(path)
            if enemy.attack_cost > enemy.energy
                enemy.energy = 0
        active_enemies.sort_descending_by_energy()
    else
        sleep(100ms)
```

Firstly the Hivemind finds all active drones, then for each one that has positive energy value it tries to order an attack. If this does not succeed it tries to move as close to the character as it can with provided energy. Because the orders are given sequentially Hivemind can reserve the target tile in Pathfinder instance, so that it will not try to find path to this position. If the Enemy cannot perform any action in given turn it is sent at the end of the list. This setup ensures the enemies will not bump into each other and will chase the character efficiently.

4.7. Implementation problems

The gaming industry notoriously struggles with unpredictable errors and bugs. Those might be result of engine malfunctions, environment incompatibilities and plain implementation errors. Even the biggest companies face those unexpected problems during development process as can be seen on example of Cyberpunk 2077 [28]. Limitations of the game engine caused problems during the implementation. Some of them (with corresponding solutions) have been described below.

4.7.1. Meshes

Mesh generation takes relatively a lot of time. In the initial one-thread implementation the game would freeze for a split second whenever a new mesh was created or an old one was updated. This is unacceptable in a modern game and needed to be fixed.

The performance tests determined that that for a single mesh it takes on average 74ms (see mesh generation performance tests section 5.3) to assemble from voxel representations. It is over 4 times longer than a single frame should take[11]. In addition usually quite a few meshes are requested at once so the freezes would took over half a second. To solve this problem an asynchronous solution was developed:

Meshes are based on the Block array corresponding to a part of a corresponding SuperChunk, so if any change occurs in the said array a proper Chunk has a outdatedMesh set to true. Meanwhile in the background an asynchronous worker continuously goes through a list of loaded Chunks and tries to update them by invoking *Chunk.UpdateMesh()*, a method that regenerates the visual representation. The resulting mesh is then saved in a temporary variable for Update thread to pick it up and replace the old instance with the new one. After the process is done *outdatedMesh* flag is set to false and *changeMesh* flag is set to true.

Every two frames each loaded Chunk with changeMesh flag set to true replaces its outdated mesh with the new one. This process is necessary due to the technical limitation of the game engine - visual representations can be only updated in the main thread (Update function of MonoBehaviour).

This ensures that every chunk is updates as soon as possible and the whole process does not affect gameplay in a negative way.

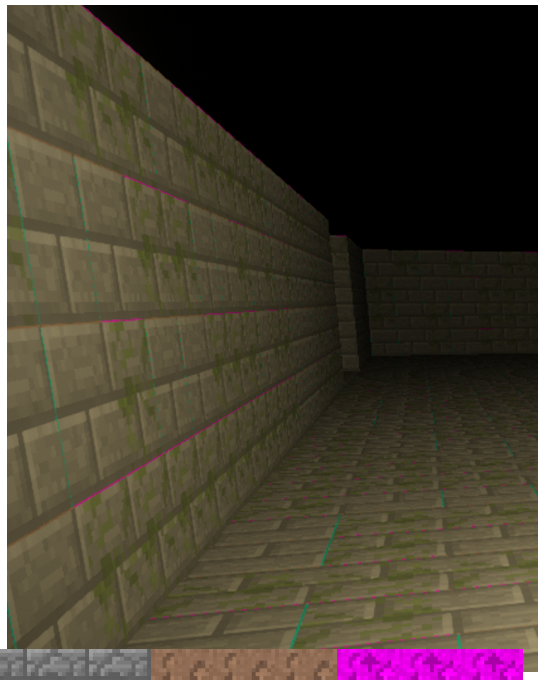
[11] for smooth gameplay at least 60 frames per second are needed, this means a single frame lasts only 16 ms

4.7.2. Texture bleeding

Texture bleeding is a common problem in the gaming industry and affects every project using texture atlases. The renderer uses said textures and for better performance translates and rescales them. This might lead to blending eadges of separate pictures. The effects of such process can be seen in Figure 4.8.

One of the solutions is to space apart the individual textures[12]. But this proven to be ineffective. The edges would still bleed the transparency of empty spaces or color if the padding was done with one. For the texture to look natural with the bleeding the same image should blend with it.

Finally the issue was solved by creating texture padding by tiling each image in a following way:



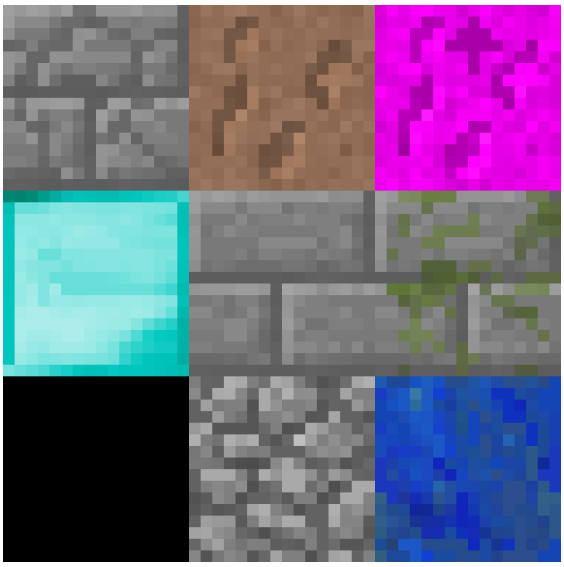


Figure 4.9: The problematic atlas

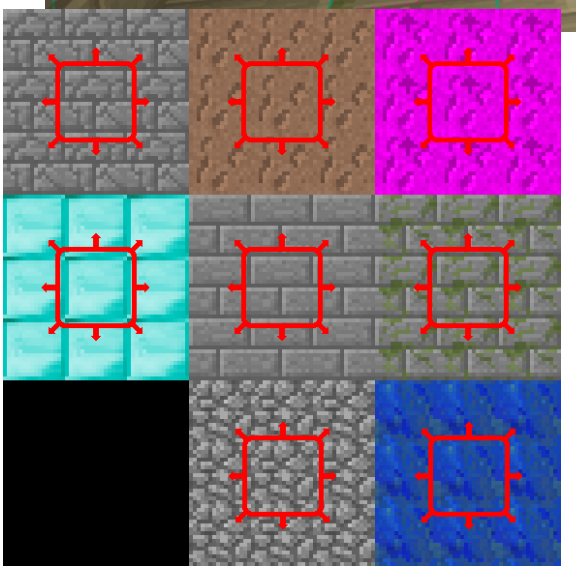


Figure 4.10: Atlas with tiled padding

The red tiling indicators are not present on the final texture.

[12] http://wiki.polycount.com/wiki/Edge_padding

4.7.3. Healthbar shadows

Enemy healthbars are implemented using *UnityEngine.UIElements.Image*[13] and *UnityEngine.Canvas*[14] instances. Due to that they do not revieve shadows, as the UI elements have a separate rendering pipeline.

This created an issue, because players perception is limited only by the shadow system provided by Unity. Hence the healthbars were visable even if they should be hidden from player. This can be seen on a following figure:

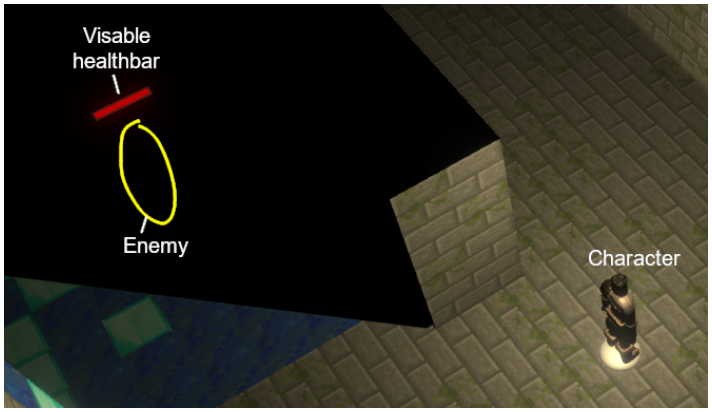


Figure 4.11: Incorrectly visible healthbar

To combat this issue a solution was developed based on a built-in Raycast[15] system. Each frame every active Enemy instance tries to 'shoot' a ray in the exact player direction. This operation is heavily optimised in Unity Engine and takes virtually no time to complete hence it has no impact on performance. If a given ray hits a wall the healthbar is turned off, otherwise it stays on.

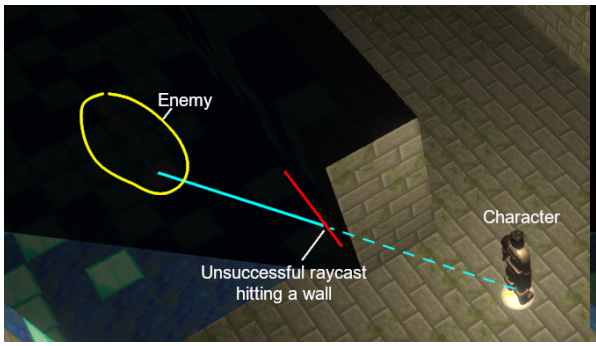


Figure 4.12: An unsuccessful raycast

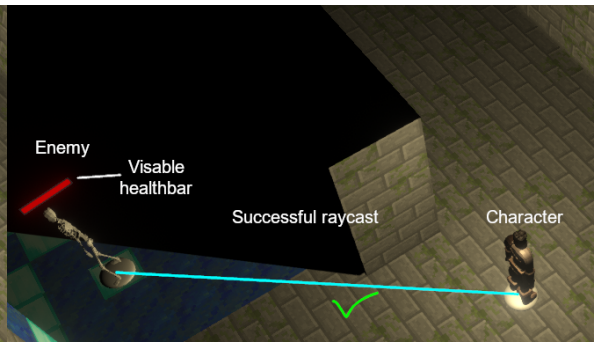


Figure 4.13: A successful raycast

The screenshot in Figure 4.12 was taken with additional light source enabled to improve visibility of the obstacle and enemy position. This does not reflect how the shadows behave normally.

[13] https://docs.unity3d.com/2020.1/Documentation/ScriptReference/UIElements.Image.html
[14] https://docs.unity3d.com/2020.1/Documentation/ScriptReference/Canvas.html
[15] https://docs.unity3d.com/ScriptReference/Physics.Raycast.html

4.7.4. Pathfinding

Pathfinding is very important feature in the game and should work flawlessly throughout the whole gameplay. There were two issues connected strictly to this application element. Firstly the game was freezing slightly and loosing frames on each path request. The problem was simply solved by implementing asynchronous solution and adding appropriate methods to the scheduler so it can interact with the Pathfinder class instance.

The second problem was also the game lagging, but because of different reason. When a new chunk was loaded in the pathfinding range exactly 4096 new Tile instances had to be prepared and instantiated. As stated earlier usually a few Chunks are requested at once so this value quickly rises. While the preparation was lightning fast and did not affect the performance the spawning did. Additionally each unloaded Chunk has to despawn its Tile instances and this also takes a bit of time. To resolve this issue an asynchronous solution was developed in combination with optimisations described in section 4.5.

Unity provides exact amount of time passed from the start of the frame[16]. Using that information it is easy to calculate how much time is left before the frame should end assuming stable 60 frames per second:

$$t_{left} = max(0; \frac{1}{60} - t_{delta})$$

Based on this value on every frame the time remaining is spent on spawning and despawning enqueued objects:

Listing 4.10: Optimised spawning algorithm

```
spawn_and_despawn(spawn_orders, despawn_orders, frame_len)
    should_go = spawn_orders.count + despawn_orders.count > 0
    do_despawn = false
    while should_go == true
        do_despawn = despawn_orders.count > 0 AND NOT do_despawn
        if do_despawn == true
            object = despawn_orders.dequeue()
            destroy(object)
        else
            if spawn_orders.count > 0
                order = spawn_orders.dequeue()
                object = spawn(order.element_to_spawn)
                order.reciever.recieve_object(object)
            else
                should_go = despawn_orders.count > 0
        should_go = should_go AND Unity.time_delta < frame_len
```

[16] https://docs.unity3d.com/ScriptReference/Time-deltaTime.html

The presented algorithm takes care of spawn and despawn scheduling. using provided first-in-first-out order queues it tries to perform as much operations as possible while alternating between instantiating and destroying objects. The spawning is a bit more complicated as alongside the instruction on what to create it requires an object that implements an *IReciever* - interface that takes care of managing newly spawned objects. In that case it is the *Pathfinder* class instance that later distributes the Tiles to corresponding Cells.

4.7.5. Confusing items

Player should not be confused by items. For that reason a deterministic random solution was prepared. Each item in the whole game has a specific color dependent on the *location seed* provided by the entity it is possessed by. The color coding was added as a simple texture filter based on the built-in Sprite system in the Unity Engine.

4.8. Final result

The first picture shows the final version of the main menu with a dark outline of a dungeon entrance barely lit up by two torches in the background.

Figure 4.14: Final version of the main menu

Figure 4.15 shows the final version of the setup screen with input field to type a numeric seed into as well as buttons for starting the game and leaving to main menu.

Figure 4.15: Final version of the setup menu

Figures 4.16 and 4.17 show the movement range, path and target together with attack indicators displayed on the dungeon floor.

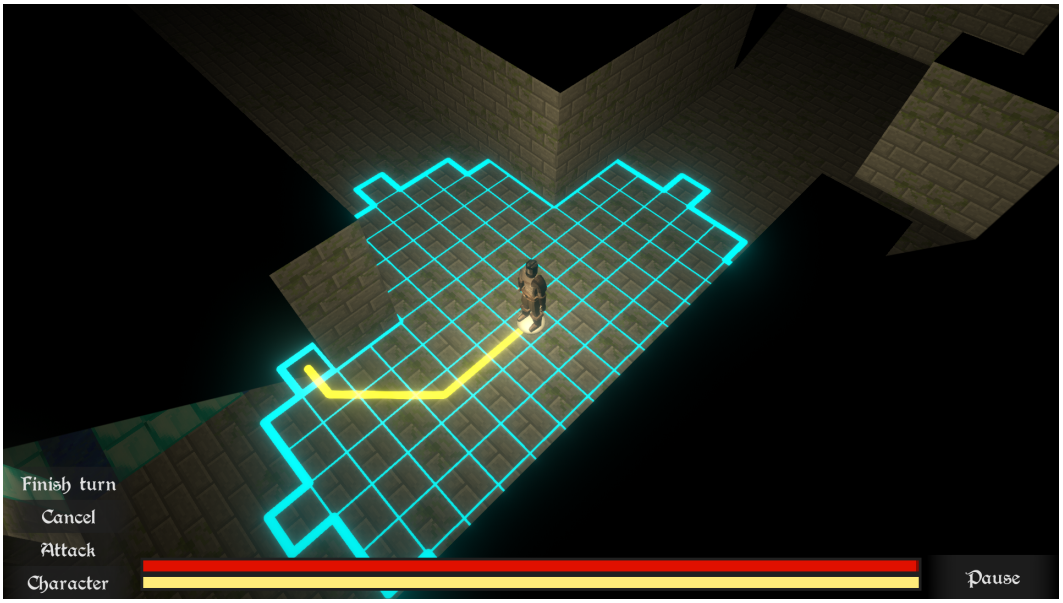


Figure 4.16: Final version of movement and path indicators

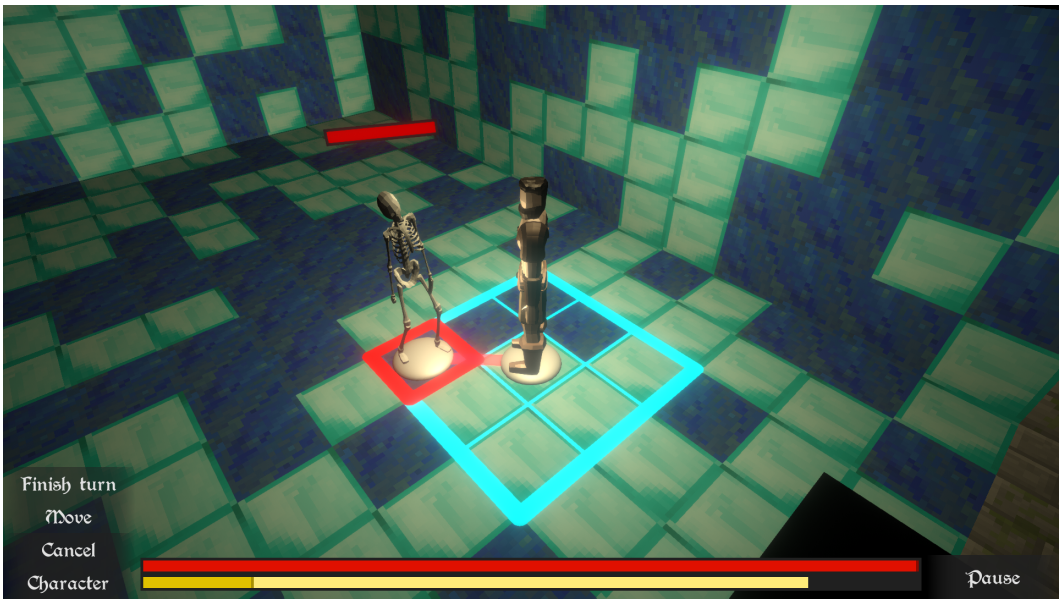


Figure 4.17: Final version of attack indicator

Figure 4.18 shows character screen in the final form. A potion and few items can be seen in the inventory. The middle section is reserved for statistics. Additionally an item tooltip is also visible on the screenshot.

Figure 4.18: Final version of the character screen

Figure 4.19 show the final version of pause menu with options to continue the game and to give up and leave.

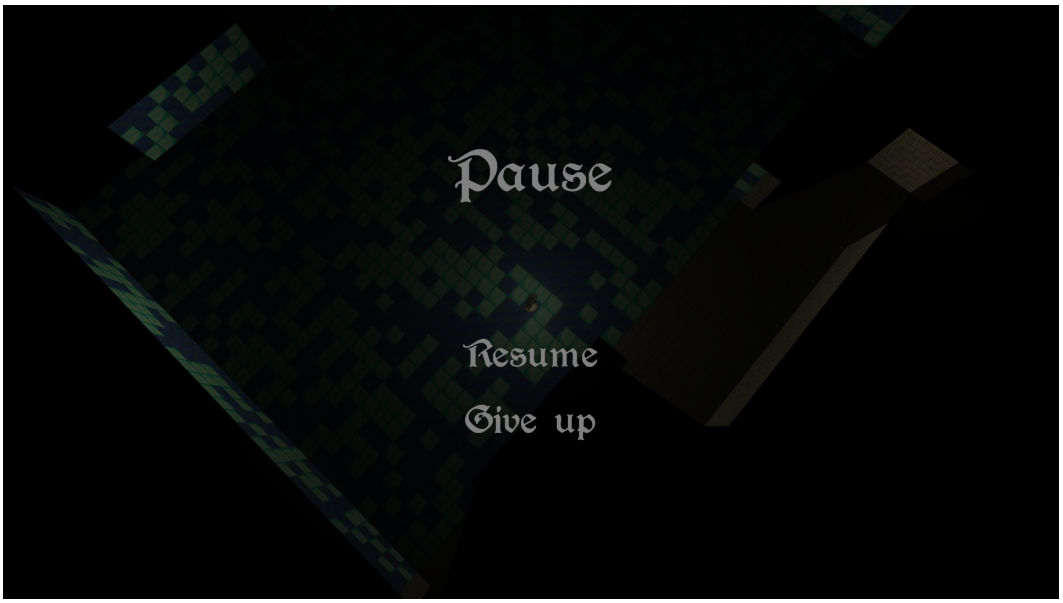


Figure 4.19: Final version of the pause menu

The final figure 4.20 shows the final version of the game-over screen with selected playthrough statistics displayed.

Figure 4.20: Final version of the game-over screen with some statistics

Assets used in the project:

- *Low Poly Fantasy Warrior*[17]
  A stylized model of a fantasy adventurer.
- *Low Poly Skeleton*[18]
  A stylized model of an skeleton monster.

[17] https://assetstore.unity.com/packages/3d/characters/humanoids/low-poly-fantasy-warrior-127775
[18] https://assetstore.unity.com/packages/3d/characters/low-poly-skeleton-162347

- *Altar Ruins Free*[19]
  A pack of fully textured stone ruin models.
- *Procedural fire*[20]
  An out of the box fire particle system ready to be placed in the game world.

Icons and some textures were prepared in a free graphics editing application *Gimp*[21]

[19] https://assetstore.unity.com/packages/3d/environments/fantasy/altar-ruins-free-109065
[20] https://assetstore.unity.com/packages/vfx/particles/fire-explosions/procedural-fire-141496
[21] https://www.gimp.org/

5. Tests

Performance and correctness tests were performed as a part of the project. Different processes were measured end evaluated as well as overall gameplay smoothness based on how many frames per second the game can show. Additionally a playtesting process was performed. The performance tests were performed on a personal computer with following specification:

- Operating system: Windows 10
- Processor: AMD Ryzen 5 1600
- Graphics card: GeForce GTX 1060
- RAM: 16 GB DDR4

All time measurements were performed using *System.Diagnostics.Stopwatch*[1] class.

Due to specificity of the gaming industry not much can be tested using traditional methods. For that reason playtesting is strongly recommended by B. Bates [23].

5.1. Playtesting

According to a comprehensive systematic review [29] playtesting is usually divided into three stages:

- Closed stage - an internal testing process not available to the public
- Beta stage - this type is usually performed during the final stages of testing just before going to market with a product. It is often run semi-open in order to find any last-minute problems
- Open stage - in general it is a test open to anyone who wishes to join or to recruited testers from outside the design group

Playtesting was performed in three stages corresponding to those mentioned earlier. All found errors were corrected.

5.1.1. Closed tests

Closed tests were performed each development week and involved testing newly implemented functionalities. Errors found in this stage:

- Texture bleeding
- Healthbar shadows
- Pathfinding optimisation problems

[1] https://docs.microsoft.com/en-gb/dotnet/api/system.diagnostics.stopwatch?view=net-5.0

5.1.2. Beta tests

Beta tests involved 4 participants who would play on average 45 minutes of the game in one sitting. This stage was performed after the development finished. Errors found in this stage:

- Enemies finishing movement on the same tile
- Meshes taking too long to generate

5.1.3. Open tests

Open tests involved 4 previous testers and 4 new. Both groups played on average 45 minutes of the game trying to find errors and problems. This stage was performed after the errors found in beta tests were repaired. Problems found in this stage:

- Items were too similar

5.2. Frame rate

Unity Profiler[2] is a built-in service for gathering information on cpu and memory usage, frame rate and direct method time consumption. Its usage is restricted to the development mode in Unity Editor so the final product should behave on average two times slower than the Editor, as it consumes 8 to 14 ms per frame so for purpose of gameplay frame rate test it was turned off.

Figure 5.1: Example Editor time consumption

The first part of the test included the initial scene loading, spawning Chunks and Tiles in the initial area using the optimised method described in subsection 4.7.4. As can be seen in Figure 5.2 due to the high number of elements to initialize the process was spread over 16 frames after which the frame-rate rises to over 120. The spawning scheduler heavily depends on the frame length hence the editor time could not be ignored in this test.

The second part (see Figure 5.3) was performed with the Profiler ignoring Editor time as no frame length depended process was measured. the graph shows frame length as well as frame-rate during a typical playthrough with a small 2 ms drop in efficiency due to over 1200 Tiles updating at once.

[2] https://docs.unity3d.com/Manual/Profiler.html

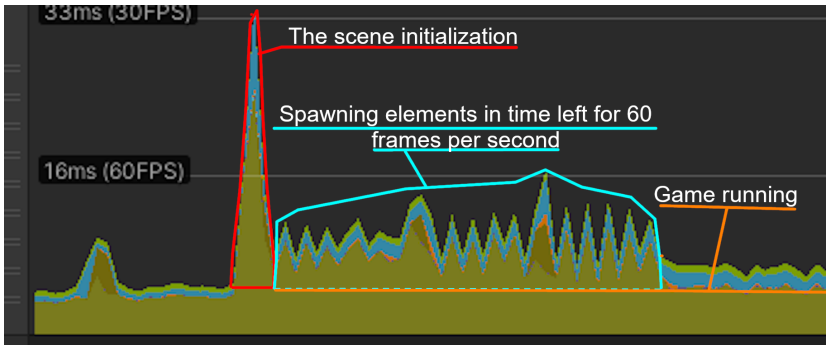


Figure 5.2: Test in development mode with editor enabled showing optimised spawning

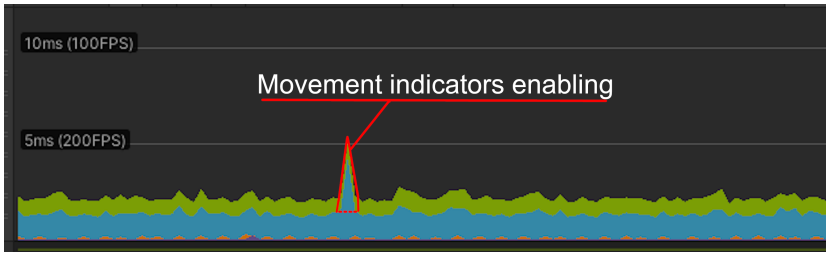


Figure 5.3: Test in simulated playmode showing smooth 400 frames per second gameplay

Both tests were taken in development build, this means that the final build is even better.

5.3. Mesh generation

There were two kinds of tests performed for mesh generation. That is structure tests and performance tests. To ensure the correctness of measurement in total: 1024 SuperChunks were fully generated, that gives 16384 individual meshes that were checked.

5.3.1. Mesh structure tests

In this test an amount of vertices in a Chunks was measured to ensure that the engine limitations are not reached in any point.

Measure	Minimum	Average	Maximum	Engine limit
Vertex count	4640	9736	17472	65536
Percent of limit	7%	14.8%	26.6%	-

Table 5.1: Vertex measurements

The test shows that the engine limits were not reached on a huge sample of real meshes.

5.3.2. Mesh generation performance tests

Mesh generation times were very important as they had very negative impact on the overall game performance. According to measurements visible in Table 5.2 the average time is 74 ms per mesh. The results are nowhere near the target values for a single mesh update per frame.

Measure	Minimum	Average	Maximum	Target
Generation time	46 ms	74 ms	103 ms	< 16 ms
Corresponding frame-rate in frames per second for one mesh per frame	21 fps	13 fps	9 fps	60 fps

Table 5.2: Mesh generation time measurements

During average movement 4 meshes need updating - that is the mesh where the character currently stands and meshes around that position due to changing visibility. This resulted in very noticeable freezes reaching over 350 ms after each movement. This problem was fully solved (see subsection 4.7.1).

Conclusion

All goals of the thesis has been reached. The resulting game has mechanics typical for the dungeon crawler genre. The game world is generated procedurally and each one is different. The algorithm is deterministic and gives possibilities of replaying specific scenarios. The game has been released for personal computers and, according to the results of the tests, runs smoothly.

Although effort has been made to develop easy to expand systems, but due to limited resources graphics might be found simple or repetitive. It wasn't always possible to find the perfect icons, textures or models that would be free for academic use.

The scope of issues covered in the work is very wide, as creating a playable product was required. Fortunately, the use of tools such as Unity and Voxel or Mesh systems significantly accelerated implementation, allowing to focus on the higher level aspects of implementation.

The product is in a late stage of development, is fully playable and can be considered a release candidate. However, the solutions used make it quite easy to expand. Adding additional content in the form of items with special properties or enemy types with specific behaviours is very simple and requires only extending already functional classes.

Possible future development

The next step in development could be the addition of game-state saving to allow players to save and load a given adventure. Non-playable-characters could be added in a form of lost adventurers or trades close to the entrance. Environment manipulation and destruction combined with magic system typical for role-playing games is also a very good concept to introduce. Graphics overhaul also should be strongly considered to create a more unique and dark atmosphere.

Thanks to the Unity Engine, it is possible to create ports for other gaming platforms with little work. The chosen graphic style and low resource consumption make an expansion onto mobile devices quite natural. Only thing that needs to be slightly adjusted is the interface.

Over time, the game might follow in the footsteps of famous projects, described during the analysis of competing solutions. All of the started very small and were developed over the years.

Bibliography

[1] Teodora Dobrilova. *How Much Is the Gaming Industry Worth in 2020?* techjury.net. 2020. URL: https://techjury.net/blog/gaming-industry-worth/ (visited on 11/16/2020).

[2] Bartosz Lewandowski oraz Maria B. Garda. "Indie games: fenomen niezależnych gier komputerowych". In: *Przegląd Kulturoznawczy* 2011.Numer 1 (9) (2011). URL: https://www.ejournals.eu/Przeglad-Kulturoznawczy/Przeglad-Kulturoznawczy-2011/Numer-1-9-2011/art/1281/.

[3] Nathan Brewer. *GOING ROGUE: A BRIEF HISTORY OF THE COMPUTERIZED DUNGEON CRAWL*. https://insight.ieeeusa.org/. 2016. URL: https://insight.ieeeusa.org/articles/going-rogue-a-brief-history-of-the-computerized-dungeon-crawl/ (visited on 11/16/2020).

[4] Julian Togelius et al. "What is Procedural Content Generation? Mario on the Borderline". In: *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*. PCGames '11. Bordeaux, France: Association for Computing Machinery, 2011. ISBN: 9781450308724. DOI: 10.1145/2000919.2000922. URL: https://doi.org/10.1145/2000919.2000922.

[5] Kris Graft. *Road to the IGF: Red Hook Studios' Darkest Dungeon*. gamasutra.com. 2016. URL: https://www.gamasutra.com/view/news/265011/Road_to_the_IGF_Red_Hook_Studios_Darkest_Dungeon.php (visited on 11/16/2020).

[6] Dan Stapleton. *Darkest Dungeon Review*. ign.com/. 2016. URL: https://www.ign.com/articles/2016/01/27/darkest-dungeon-review (visited on 11/16/2020).

[7] Darryl Dyck. *Darkest Dungeon developers find success in B.C.'s growing gaming industry*. theglobeandmail.com. 2016. URL: https://www.theglobeandmail.com/news/british-columbia/darkest-dungeon-developers-find-success-in-bcs-budding-gaming-industry/article28409660/ (visited on 11/16/2020).

[8] *Happy Holidays from the Hamlet!* darkestdungeon.com. 2017. URL: http://www.darkestdungeon.com/happy-holidays-from-the-hamlet/ (visited on 11/16/2020).

[9] *Darkest Dungeon Review*. ign.com. 2016. URL: https://assets1.ignimgs.com/2018/06/19/darkest-dungeon-the-color-of-madness---03-1529441030193.jpg?crop=16%5C%3A9&width=888 (visited on 11/16/2020).

[10] Nathan Meunier. *Legend of Grimrock Review*. ign.com. 2012. URL: https://www.ign.com/articles/2012/04/09/legend-of-grimrock-review (visited on 11/16/2020).

[11] *LEGEND OF GRIMROCK*. metacritic.com. 2012. URL: http://www.metacritic.com/game/pc/legend-of-grimrock/critic-reviews (visited on 11/16/2020).

[12] *Sales of Legend of Grimrock dungeon crawling game series are still going on strong*. gamershell.com. 2014. URL: http://www.gamershell.com/news_169815.html (visited on 03/16/2018).

[13] *THOUGHTS: LEGEND OF GRIMROCK*. 2012. URL: https://www.scientificgamer.com/blog/wp-content/uploads/2012/04/miss.jpg (visited on 11/16/2020).
[14] Edge Staff. *The Making Of: Rogue*. edge-online.com. 2009. URL: http://www.edge-online.com/features/making-rogue.
[15] Benj Edwards. *GOING ROGUE: A BRIEF HISTORY OF THE COMPUTERIZED DUNGEON CRAWL*. pcworld. 2009. URL: https://www.pcworld.com/article/158850/best_pc_games.html#slide6.
[16] *Rogue screenshot*. wikipedia.com. 2008. URL: https://en.wikipedia.org/wiki/File:Rogue_Screen_Shot_CAR.PNG (visited on 11/16/2020).
[17] Matt Silverman. *Minecraft: How Social Media Spawned a Gaming Sensation*. mashable.com. 2010. URL: https://mashable.com/2010/10/01/minecraft-social-media/?europe=true (visited on 11/16/2020).
[18] Tom Warren. *Minecraft still incredibly popular as sales top 200 million and 126 million play monthly*. theverge.com. 2020. URL: https://www.theverge.com/2020/5/18/21262045/minecraft-sales-monthly-players-statistics-youtube (visited on 11/16/2020).
[19] Marcus Toftedahl. *Which are the most commonly used Game Engines?* gamasutra. 2019. URL: https://www.gamasutra.com/blogs/MarcusToftedahl/20190930/350830/Which_are_the_most_commonly_used_Game_Engines.php (visited on 11/16/2020).
[20] Pietro Polsinelli. *Why is Unity so popular for videogame development?* pcworld. 2013. URL: https://designagame.eu/2013/12/unity-popular-videogame-development/ (visited on 11/16/2020).
[21] *Unity User Manual for 2020.1*. unity. 2020. URL: https://docs.unity3d.com/2020.1/Documentation/Manual/ (visited on 11/16/2020).
[22] *Visual Studio C# integration*. unity. 2020. URL: https://docs.unity3d.com/Manual/VisualStudioIntegration.html (visited on 11/16/2020).
[23] B. Bates. *Game Design*. Premier Press, 2004. ISBN: 9781592004935. URL: https://books.google.pl/books?id=f7XFJnGrb3UC.
[24] Ivan-Assen Ivanov. *Practical Texture Atlases*. gamasutra.com. 2006. URL: https://www.gamasutra.com/view/feature/130940/practical_texture_atlases.php (visited on 11/16/2020).
[25] *Max vertices*. unity. 2020. URL: https://docs.unity3d.com/2020.1/Documentation/ScriptReference/Mesh-indexFormat.html (visited on 11/16/2020).
[26] Markus Persson (Notch). *Terrain generation, Part 1*. 2011. URL: https://notch.tumblr.com/post/3746989361/terrain-generation-part-1 (visited on 11/16/2020).
[27] *Mathf.PerlinNoise*. unity. 2020. URL: https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html (visited on 11/16/2020).
[28] Andy Chalk. *CD Projekt shoots down a new Cyberpunk 2077 delay rumor*. 2020. URL: https://www.pcgamer.com/cd-projekt-shoots-down-a-new-cyberpunk-2077-delay-rumor/ (visited on 11/16/2020).
[29] Rosa Yanez, Daniel Cascado-Caballero, and Jose Luis Sevillano. "Academic methods for usability evaluation of serious games: a systematic review". In: *Multimedia Tools and Applications* 76 (Feb. 2017), pp. 1–30. DOI: 10.1007/s11042-016-3845-9.

List of Figures

List of Tables

Source code list