Department of Computer Science
Faculty of Science
Palacký University Olomouc

BACHELOR THESIS

Procedurally generated content in computer game

2019

Serhiy Kudryashov

Supervisor: Mgr. Petr Osička, Ph.D.

Study field: Applied Computer Science, full-time form

Bibliografické údaje

Autor:	Serhiy Kudryashov
Název práce:	Jednoduchá hra s procedurálně generovaným obsahem
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2019
Studijní obor:	Aplikovaná informatika, prezenční forma
Vedoucí práce:	Mgr. Petr Osička, Ph.D.
Počet stran:	54
Přílohy:	1 CD
Jazyk práce:	anglický

Bibliograhic info

Author:	Serhiy Kudryashov
Title:	Procedurally generated content in computer game
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2019
Study field:	Applied Computer Science, full-time form
Supervisor:	Mgr. Petr Osička, Ph.D.
Page count:	54
Supplements:	1 CD
Thesis language:	English

Anotace

Cílem práce bylo prozkoumát možnosti procedurálního generování obsahu v počítačových hrách. Na příkladu jednoduché hry zjištěné metody implementovat a okomentovat jejich vhodnost, případně různé metody porovnát.

Synopsis

The main goal of the thesis was to analyze the possibilities of a procedurally generated content in computer games, to implement a simple game with the use of obtained techniques, to comment suitability of used methods and to provide their comparison.

Klíčová slova: procedurálně generovaný obsah; procedurální generování; hra; algoritmy

Keywords: procedurally generated content; procedural generation; game; algorithms

I wish to express my sincere gratitude to my supervisor Mgr. Petr Osička, Ph.D. for his essential recommendations and guidance throughout this thesis. I am grateful to all members of the Department of Computer Science for continuous inspiration during all the years of my study at Palacký University Olomouc. I also thank my parents and friends for their unceasing support and motivation.

I hereby declare that I have completed this thesis including its appendices on my own and used solely the sources cited in the text and included in the bibliography list.

date of thesis submission author's signature

Contents

1 Introduction **1**

2 Computer games **2**
2.1 Game development process . . . 2

3 Procedurally generated content **4**
3.1 Random generation . . . 5
3.2 Benefits and pitfalls of PCG in games . . . 5
3.3 Usage of PCG in games . . . 6
3.3.1 Space saving . . . 7
3.3.2 Game world elements . . . 7
3.3.3 Animation . . . 7
3.3.4 Audio . . . 8
3.3.5 Levels . . . 8
3.4 Conclusion . . . 9

4 Game implementation **10**
4.1 Title . . . 10
4.2 Game design . . . 11
4.3 Visual representation . . . 12
4.3.1 Animation . . . 12
4.4 Unity . . . 13
4.4.1 DOTS . . . 13
4.4.2 ECS . . . 14
4.4.3 C# Job System . . . 15
4.4.4 Burst Compiler . . . 17
4.5 Procedurally generated content in the game . . . 17
4.5.1 Dungeon Generation . . . 17
4.5.1.1 “Naive” approach . . . 18
4.5.1.2 Cellular automaton . . . 22
4.5.1.3 BSP-tree . . . 28
4.5.1.4 Comparison . . . 33
4.5.1.5 Tile map . . . 34
4.5.2 Object placement . . . 35
4.5.2.1 Player spawning . . . 35
4.5.2.2 Poisson disk sampling . . . 35
4.5.3 Objectives generation . . . 38
4.5.4 Leveling and stats . . . 39
4.5.5 Items generation . . . 40
4.6 Basic game mechanics . . . 41
4.6.1 Player . . . 41
4.6.2 Camera setup . . . 41
4.6.3 Minimap . . . 41

4.6.4 AI . . . 41
4.6.5 Pathfinding . . . 42
4.6.6 Dungeon transitions . . . 42
4.6.7 Saving and loading the game . . . 43
4.6.8 Tutorial level . . . 43
4.6.9 Game settings . . . 44
4.7 User interface . . . 45
4.7.1 Main menu . . . 45
4.7.2 In-game interface . . . 45

Závěr **47**

Conclusions **48**

A Contents of the enclosed CD **49**

Acronyms **50**

References **51**

List of Figures

1 Games that use PCG for space saving . . . 7
2 Games with procedurally generated world elements . . . 7
3 Games with procedural animation . . . 8
4 Games with procedurally generated levels . . . 9
5 Screenshots of the implemented game . . . 10
6 Game's icon . . . 10
7 Use case diagram . . . 11
8 Main character . . . 12
9 Chest opening animation . . . 12
10 ECS Performance comparison . . . 17
11 Parallel execution of jobs . . . 21
12 Final results of the "Naive" dungeon generation . . . 21
13 Neighborhood types . . . 23
14 Bresenham's line algorithm . . . 25
15 Cellular automaton map creation process . . . 26
16 Final results of the Cellular automaton dungeon generation . . . 27
17 BSP-tree creation process . . . 30
18 Digging the corridor . . . 31
19 BSP-tree rooms connection . . . 31
20 Final results of the BSP-tree dungeon generation . . . 32
21 Creating the tile map with Unity . . . 34
22 Generated maps converted to tile map . . . 34
23 Poisson disk sampling on a 2D plane . . . 37
24 PDS problem in dungeons . . . 37
25 Results of PDS in the dungeon . . . 38
26 Quests generation in the game . . . 39
27 Generated items . . . 40
28 Setting up the camera with the Cinemachine . . . 41
29 Generated navigation mesh for the dungeon . . . 42
30 Dungeon cut-scene created with Timeline tool . . . 43
31 Screenshot from the tutorial level . . . 44
32 Main menu and settings menu . . . 45
33 In-game interface . . . 46

List of Tables

1 Comparison of implemented algorithms . . . 33

List of source codes

1 Example of a simple ECS component 14
2 Example of a simple ECS system 15
3 Example of a simple ECS system with a job 16
4 “Naive” approach implementation in ECS 19

1 Introduction

The game industry, as well as the gaming community today, has reached a massive size in the entertainment industry and it is hard to ignore its influence on media and our lives, which is increasing every year [1].

Computer games always took a special place in my life and were one of the main reasons why I am so highly interested in programming and computer science overall. Since childhood, one of the greatest mysteries for me was how computer games are made and that is what set me on an exciting journey of studying computer science. I've done some simple games before and was excited during the process, but I always wanted to make a more complex game. Therefore, it is not surprising that I took the opportunity to try and do this as a bachelor thesis.

The main focus of this thesis is procedurally generated content in computer games. One of the goals was to implement a game with the use of techniques obtained during the research. The first part of this work describes commonly used algorithms and methods of procedural generation. The second part provides detailed information on the process of game implementation.

The implemented game itself is a single-player 2D roguelike, which is a very popular genre among games built with this approach. Moreover, the game not only demonstrates procedurally generated elements but also tries to provide an entertaining and enjoyable experience to the player.

The reader of this thesis will be introduced to the problem of procedural generation and integration of procedurally generated content into the game. It is expected that the reader has a basic understanding of algorithms and data structures.

2 Computer games

Procedural generation requires an understanding of what computer games actually are, how they are made and why they can benefit from procedurally generated content.

A video game is a software with the main goal of entertaining its users. A game involves interaction with a user interface to generate visual feedback on a video display device [2].

A computer game is a video game played on a personal computer. PC game platform is known for its overall higher performance, which can result in better image quality, higher frame rates, bigger and more complex game worlds. This platform also provides a wider range of peripherals. But this all comes for a price of increased hardware cost [3].

2.1 Game development process

Game creation is a full software development process [4]. Hence, well known and widely used in software development techniques may be and should be used during game development. Although, an iterative design, especially iterative prototyping approach has proved itself as a better option here. The general idea is to create a basic functional prototype and continue from that point to the final product by adding new features and removing unwanted or unexpected behaviour [5].

Proper planning and time management is required for a game to be successful. Otherwise, due to lack of time, budget exhaustion or numerous bugs, failure is inevitable.

That is why it is very important to keep in mind the basic stages of the game development process [5], [6]:

1. *Pre-production* – probably, the most important part. At this stage, the idea and the main concept of the game is discussed and documented. Also, the first prototypes are created here. Those prototypes can serve as proof of concept and can tell if the game has any chance to succeed or just may lead to waste of time, money and human resources. Moreover, successful prototypes may be reused later in the process.

2. *Production* – the main phase of actual development. Here game elements are implemented. This includes programming, modelling, level creation, art and audio production and testing.

3. *Milestones* – used to track product progress. Milestones are important timestamps of the game's lifecycle. Reaching these points in time makes easier to analyze what was already done, and what lies ahead. Most known milestones are Alpha, Beta and Gold master (final build that will be used for production).

4. *Post-production* – lifecycle of the game after the official release. Here bugs and problems found after launch are solved. New content and features may be added over time, and they will go through the identical phases up until shipped to end-user.

As can be clearly seen, the whole process of the game development is complex, time-consuming and requires a lot of human resources. Creating a playable and immersive game world requires a lot of effort and hard work.

But what if some time could be saved by letting computers do what they do best – compute? What if algorithms could be used to place objects on the level? That would probably save hours or even days for a level designer. Moreover, it may provide some interesting results for further inspiration. Or how to create an infinite world that will feel natural and unique? This leads to the main topic of this thesis – procedurally generated content.

3 Procedurally generated content

First of all, it is necessary to clarify some terminology. Procedurally generated content is content, that was created with Procedural Content Generation (PCG) techniques. That leads to the question, what is PCG? There are two parts in this terminology that need explanation – procedural generation and the content itself.

Procedural generation is a process of creating data not manually, but with an algorithm [7]. An algorithm is a sequence of computational steps that transform the input values into output [8]. Content is data that are presented to the player, such as text, levels, models, items, game rules, quests, music, maps, textures etc [9], [10].

With all that in mind, PCG can be defined as "the algorithmic creation of game content with limited or indirect user input" [11]. In other words, PCG is a process, that takes some input values, for example, the size of the level to generate, then through a series of defined computational steps converts those values into a game content.

Even though PCG found its place in a wide range of areas, it is still expected from PCG to satisfy some common properties [10]:

- *Speed* – whether generation takes place during development or the actual gameplay, speed will always be one of the most crucial aspects of PCG. The main goal here is to find adequate time boundaries. It is unacceptable to make the player spend most of his time at loading screen, neither it is possible for a developer to spend hours of waiting due to rerunning PCG tool until a usable result is created.

- *Reliability* – may be more important for one type of content than others. This requires some kind of quality criteria. An object spawned in a wrong place may block the player from reaching an expected destination, which is a big failure.

- *Controllability* – content generators must provide some way to control the process, so user, whether it is a player or other algorithm, has the ability to affect the result. An object spawner may take into account how far away from each other user wants to spawn objects.

- *Expressivity and diversity* – it is expected from generators to provide a wide range of content with elements that feel unique.

- *Creativity and believability* – more pleasant content for the player is the one, that does not seem unnatural to him. The main goal is to make the player believe that the game world is real. Even little detail, such as a weird-looking tree's branch, may ruin the immersive experience.

It goes without saying, that tradeoffs are always present in the implementation of a content generator, as in any software, such as speed over quality, or reliability over believability [10].

3.1 Random generation

Alongside with procedural generation, the term random generation is often mentioned. Sometimes procedural generation is mistaken for random generation and it may lead to confusion. It is important to understand the difference between those two.

PCG, as was mentioned earlier, is a process that utilizes some algorithms. Algorithms are definite – for the same inputs the same output is given [12]. Same is true for PCG, if generator receives the same inputs, it will produce the same content and there is no randomness involved [7]. But one of the goals of PCG is to provide unpredictable results, thus some level of randomness must take place. We can achieve that, by altering our inputs with random numbers, or by changing the order of commands execution depending on some random values. This is where the boundary between random and procedural terms is erased. Often, when it is said "procedurally generated", it is meant procedurally generated with utilizing randomness [7]. As we can see, definiteness is still saved – the same random values with the same input will result in the same output.

Obtaining a random number seems like a simple task from a human perspective. But for computers, in fact, it is a very complex problem. This is because computers are deterministic. Hence, random numbers generated by computers may seem like random, but actually, are pseudo-random, because they are a result of some complex algorithm, which is again definite [7].

Procedural content generators usually do not require every single random value as an input. Instead, they may ask for one single number commonly known as a seed. A generator will use this number to drive algorithm execution [9]. For example, most of the random number generators require seed as an input. If the same seed is provided, the same numbers will be generated. It is common practice, that number generators take current system time as a default seed. So, by giving randomized seed to content generator we will obtain unpredictable and dynamic results.

3.2 Benefits and pitfalls of PCG in games

It is always debatable why, when and where one should use PCG. Understanding of what PCG may give is crucial to make the right decision.

These are strong sides of PCG [13], [10]:

- *Time-saving* – generators can create a massive amount of content in a short time.
- *Expandability and flexibility* – changing or adding new features will have an impact on every output.
- *Replayability* – generators may provide a similar, but a unique experience at the same time.

- *Reusability* – generators can be reused between applications. Depending on the context, generator can create a completely different experience.
- *Individual experiences* – there is an opportunity for every player to immerse themselves in a unique experience.
- *Creativity* – content provided by the generator may be far beyond human imagination.
- *Overcoming technical limitation* – PCG can be used to produce an amount of content that cannot be stored, therefore be a form of data compression [11].
- *Fun* – the process of creating PCG systems can be very exciting and a great challenge at the same time. "What greater than to create a creator?" [13]

But everything comes with a price. PCG "can be a black hole, a slippery slope, a project risk, and a dark abyss"[13]. These are some known risks that need to be considered [13]:

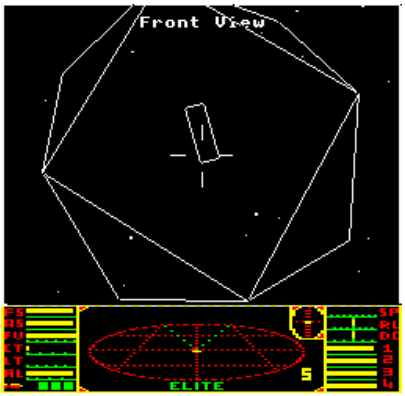
- *Quality assurance* – game with PCG may not work as intended 100% of the time. Despite the creation of a sufficient number of automated tests and test scenarios, it is easy to miss the bug. At the same time, excessive generator restriction will lead to less variant experience. Finding that "golden" spot may be a hard task to accomplish.
- *Time restrictions* – PCG should have saved time, but this is not always the case. Implementing, tweaking and debugging the generator may be more time consuming then it was expected.
- *Multiplayer* – online competitive games have one very important and hard to achieve property – balance. It is extremely difficult to create a generator in such a way, that it could not give advantage to one player over another.
- *Over reliance on PCG* – the game will fail, if there is nothing for the player to do other then wandering through infinite worlds. The final touches must be made by hand to ensure the best experience.

3.3 Usage of PCG in games

This section provides a brief overview of popular games and how they made use of PCG.

3.3.1 Space saving

Storage limitation is one of the problems that PCG can solve. For example, *Elite* took an approach of saving space by storing the seed numbers that were used to create eight galaxies [10]. Another example is *.kkrieger* that uses only 97,280 bytes of disk space. It was achieved by storing textures by their create history instead of per-pixel basis and then recreating them on load [14].



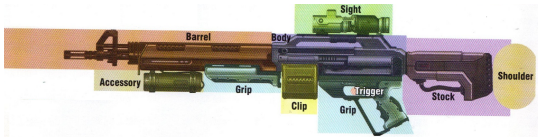
(a) Elite (1984). Source: [15]

(b) .kkrieger (2004). Source: [16]

Figure 1: Games that use PCG for space saving

3.3.2 Game world elements

PCG often used to fill the world with unique elements to explore. Such may be items, weapons, characters, text, flora and fauna. For example, guns in *Borderlands 2*. This game has around 17.75 million weapons to use. Weapons are created by combining small pieces and adding stats and features to the final result [17]. Another example are creatures from *No Man's Sky*. Those are created in the same manner, by constructing from basic parts [18].



(a) Borderlands 2 (2012). Source: [19]

(b) No Man's Sky (2016). Source: [20]

Figure 2: Games with procedurally generated world elements

3.3.3 Animation

Models created by PCG techniques can be impossible to animate by hand. One of the widely known techniques of procedural animation is ragdoll physics. Ragdolls take advantage of a skeletal model structure by altering the position of

bones depending on the applied forces. *Hitman: Codename 47* was a pioneer in integrating ragdolls into game mechanics [21]. *Spore*, on the other hand, took another approach. Their animation tool allows animators to describe motion using familiar posing and key-framing methods. The system records the data and then at runtime, apply those data to specific characters to yield pose goals [22].

(a) Hitman: Codename 47 (2000). Source: [21]

(b) Spore (2008). Source: [23]

Figure 3: Games with procedural animation

3.3.4 Audio

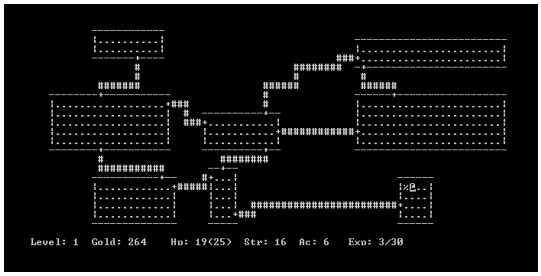
At first glance, the audio experience may not seem so noticeable, but in fact it is a really important part of the game and adds a unique feeling. Procedural music can give even more to this. Currently played audio can be adjusted to suit gameplay context. For example, when there is action on the screen, music can be more intense. It can make the player be more concentrated. Also, when the player is peacefully wandering the world, music can be more calm and relaxing to let the player enjoy the environment. The other approach is to generate music at the runtime. This can be achieved by using neural networks for example. *Spore* and *No Man's Sky* both have procedural music, that creates a unique audio experience for the player [24].

3.3.5 Levels

Finally, level creation is for what PCG is known the most. The level implies any game area, such as dungeons, terrains, planets and even whole galaxies. Wide range of algorithms is used to achieve this. *Rogue* was one of the first games with PCG. It is a dungeon-crawling roleplaying ASCII[1] game that features dungeon generation, items and enemies spawning. Every time the player descends the stairs, new level is generated. Rogue created a whole new genre – roguelike

[1] American Standard Code for Information Interchange

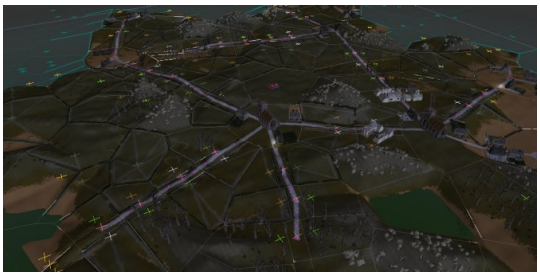
[25]. *Minecraft* creates almost endless world to explore. It is achieved by using a variant of 3D Perlin noise to create terrains [26]. *Diablo 3* creates dungeons by combining premade parts [27]. Developers of *Sir, you are being hunted* used Voronoi diagrams to create beautiful landscapes [28].



(a) Rogue (1980). Source: [29]

(b) Minecraft (2011). Source: [26]

(c) Diablo 3 (2012). Source: [27]



(d) Sir, you are being hunted (2014). Source: [28]

Figure 4: Games with procedurally generated levels

3.4 Conclusion

To sum up, PCG is a process of creating game content with the help of algorithms. A content generator can be driven by limited or indirect user input. Generation can be randomized, by providing random seed value, to achieve more unpredictable results. PCG has found its place in multiple areas. Games that implement PCG techniques can create a unique, interesting and almost infinite world, and also may save time if done right.

4 Game implementation

From now on, this thesis will cover the game design and implementation of gameplay mechanics as well as some known PCG techniques on the example of a simple game.

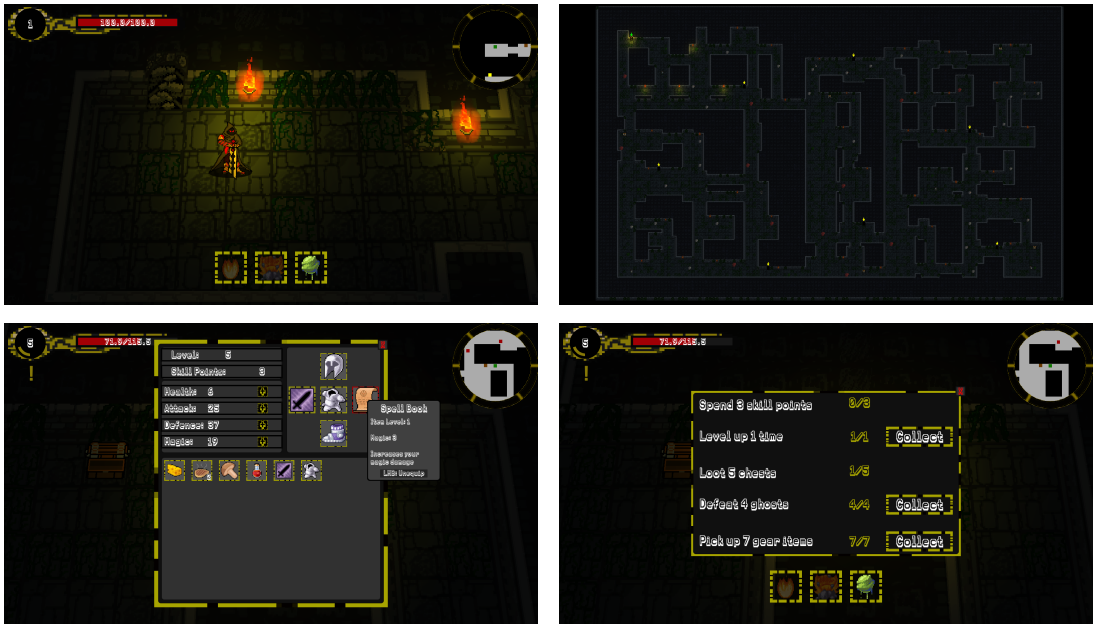


Figure 5: Screenshots of the implemented game

4.1 Title

Figure 6: Game's icon

For a long time my project was called "BP" as an abbreviation from "Bakalářská práce", that is, "Bachelor thesis" in Czech. At some point in time, I decided to give the game a name, so this abbreviation and game's visual aesthetics led me to the title – "Beyond Pixels". The title also shows the main goal of this thesis – to describe what lies beyond those pixels of a procedurally generated content.

4.2 Game design

The game itself is a 2D roguelike with pixel art graphics that features dungeon generation, enemies and items spawning, level and stats progression system, a simple system of generating tasks for a player to complete and tutorial level. The game also has the ability to save and load the current progress of the player. Such common genre was chosen in order to sustain the main focus on implementing PCG methods and gameplay mechanics, than on visual aesthetics and game design.

The gameplay is similar to roguelike games. The main goals of the player are to explore dungeons, kill enemies, collect items, gain experience points. Dungeons are connected with staircases. When a player passes through a connection, a new dungeon is generated, the passage is blocked, and the player is forced to search for the next ladder to continue. Enemies and items in each dungeon adapt to the level of the player and his characteristics.

A more detailed overview of the game from the player's perspective can be described in the use case diagram (see figure 7). It is also a way to specify the requirements and functionality that the game provides.

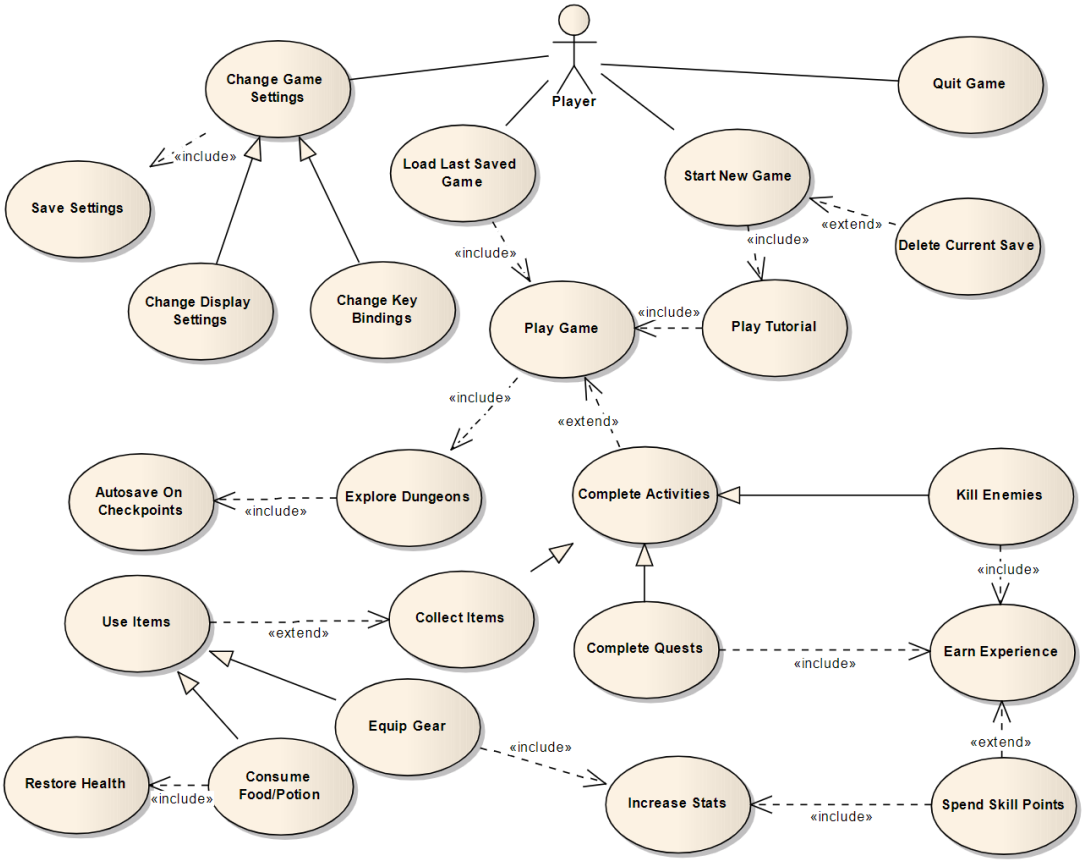


Figure 7: Use case diagram

4.3 Visual representation

Figure 8: Main character

The game completely consists of 2D pixel art sprites. Pixel art is a type of digital art, where the image is edited on the pixel level [30]. It is usually drawn by hand, but not necessarily. The size of sprites is often small, for example, 32x32 pixels. Pixel art graphic was chosen for several reasons:

- Fast to create and adjust.
- Suits the genre and pays tribute to old roguelike games.
- Does not require a high level of details.
- Light-weight and cheap to render.

As with any other form of art, pixel art requires practice and talent. All sprites in this game were drawn by me, with the exception of icons taken from open game art resource[2]. The font in the game is the free font "Karma Future"[3]. It has a pixelated style and suits the game really well.

4.3.1 Animation

The way animation is created in this game is a common approach to pixel art animation. The animation consists of frame-by-frame hand-drawn sprites that change with a certain time step.

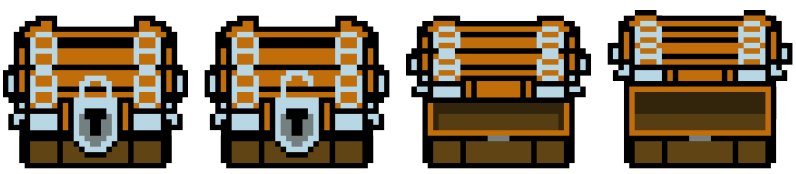


Figure 9: Chest opening animation

[2] 496 pixel art icons for medieval/fantasy RPG by 7Soul1
[3] Karma Font Family by Raymond Larabie

4.4 Unity

The decision to use the engine was made in order to concentrate more on the main target – procedurally generated content. The choice fell on Unity, a game engine developed by Unity Technologies [31]. Unity engine offers a lot of features [32], and those were the most convincing:

- Free for personal use.
- Multiplatform editor that has the ability to build applications for more than 25 platforms.
- All-in-one editor that offers a wide range of tools for 3D and 2D development.
- Built-in User Interface System.
- Large amount of official and made by community tutorials available.
- .NET Framework[4] 4.6 scripting runtime with C#[5] 7.3 as the main scripting language.
- Ability to make standalone builds with IL2CPP[6].

4.4.1 DOTS

Data-Oriented Technology Stack (DOTS) is a new chapter in the history of Unity. The Unity team set as their priority "performance by default", and now they are rebuilding the core of the engine. What they want to achieve with DOTS is to give developers the ability to write high-performance multithreaded code with the least effort [33]. DOTS consists of 3 parts that can make this possible [33]:

- Entity Component System
- C# Job System
- Burst Compiler

DOTS, for now, is in preview, hence, it has limited functionality and some features may be changed or removed in future updates. Minimal required Unity version is 2019.1.0f1, and this project is using 2019.3.0a6, where some minor bugs were fixed [34].

[4] .NET Framework
[5] C# Guide
[6] Intermediate Language To C++ in Unity

4.4.2 ECS

Entity Component System (ECS) is an architectural pattern, that follows composition over inheritance principle. In ECS game objects are represented as entities that consist of components. The behaviour of entities is driven by those components. It gives entities the ability to change their behaviour at runtime, by adding or removing components. Systems are then the place where behaviour is defined. The system will search for entities with certain components and apply defined operations on them. Such an approach "eliminates the ambiguity problems of deep and wide inheritance hierarchies that are difficult to understand, maintain and extend" [35].

Unity's ECS implementation highly benefits form data layout. Entities with the same set of components are stored in memory together. Such a set is called an archetype. Adding or removing a component from the entity switches its archetype. Unity's ECS allocates memory in chunks. Every chunk than contains component data for entities with the same archetype. An archetype has a list of chunks in which entities of this archetype are stored. In order to access the data of the entities components, ECS will cycle through these chunks, and within each of them, it performs a linear loop over tightly packed memory [36].

Because ECS is still in preview mode, it lacks basic features, such as animations, physics, rendering, networking. So the way it is used in this game is called "Hybrid" approach. Old Unity's *GameObjects*[7] are used to represent entities data. Unity has built-in support for such approach. In order for it to be functional, *GameObjectEntity*[8] component must be added to *GameObject*. It will create a bridge between ECS and Unity's *GameObject*, that allows us to use animation, physics and other existing non-ECS components inside ECS. Unity's ECS also has limitations on what can be stored inside components. Components must be structs that can contain only *blittable types*[9] and no reference types or methods[37].

Source code 1 shows an example of a simple component and source code 2 shows an example of a simple system, that moves the object with the use of Unity's physics. On every frame system filters entities that have *Movement* and *RigidBody2D*[10] components. It will calculate velocity based on speed and direction, that entity is heading, and apply it to entity's rigidbody.

```
1 public struct MovementComponent : IComponentData
2 {
3   public float2 Direction;
4   public float Speed;
5 }
```

Source code 1: Example of a simple ECS component

[7] Unity's GameObject
[8] Unity's GameObjectEntity
[9] Blittable types
[10] Unity's Rigidbody2D Component

```
1  public class MovementSystem : ComponentSystem
2  {
3    private EntityQuery group;
4
5    protected override void OnCreate()
6    {
7      this.group = this.GetEntityQuery(new EntityQueryDesc
8      {
9        All = new ComponentType[]
10       {
11         ComponentType.ReadOnly(typeof(MovementComponent)),
12         typeof(UnityEngine.Rigidbody2D)
13       }
14     });
15   }
16
17   protected override void OnUpdate()
18   {
19     this.Entities.With(this.group).ForEach((Entity entity,
20       ref MovementComponent movementComponent,
21       Rigidbody2D rigidbody) =>
22     {
23       var velocity = math.normalize(movementComponent.Direction) *
24         movementComponent.Speed;
25
26       rigidbody.velocity = velocity;
27     });
28   }
29 }
```

Source code 2: Example of a simple ECS system

4.4.3 C# Job System

Job system was created to give developers the ability to fully utilize all available CPU cores. C# Job System is actually a wrapper around the C++ Job System written in native code [33]. It features built-in scheduler that is responsible for creating and executing jobs on worker threads. Job system manages dependencies and has safety checks enabled by default to prevent accidental race conditions. The race condition problem is solved by sending each job a copy of the data instead of the reference. This approach isolates the data, but has a limitation – a job can receive only blittable data types, because blittable types do not require conversion when passed between managed[11] and native code [38], [39]. It also isolates the results of a job and to overcome this, using of a special *NativeContainer*[12] is required. *NativeContainer* is a safe C# wrapper for native memory, that contains a pointer to an unmanaged allocation and provides the way for the job to share the data with the main thread instead of working with

[11] More about managed code
[12] More about NativeContainer

the copy. Since the memory is unmanaged, it is necessary to manually deallocate the data by calling the *Dispose* method. Unity's ECS provides different types of containers to use, such as arrays, hashmaps and queues [40]. Those types are heavily used in this game.

Source code 3 demonstrates an example of a simple system with a job in ECS. The job checks current health value and if it is less than or equal to 0 adds *KilledComponent* to the entity.

```
1  public class DeathSystem : JobComponentSystem
2  {
3    [ExcludeComponent(typeof(KilledComponent))]
4    private struct DeathJob : IJobForEachWithEntity<HealthComponent>
5    {
6      public EntityCommandBuffer.Concurrent CommandBuffer;
7
8      public void Execute(Entity entity, int index,
9              [ReadOnly] ref HealthComponent healthComponent)
10     {
11       if (healthComponent.CurrentValue <= 0)
12         this.CommandBuffer.AddComponent(index, entity, new
               KilledComponent());
13     }
14   }
15
16   private EndSimulationEntityCommandBufferSystem endFrameBarrier;
17
18   protected override void OnCreate()
19   {
20     this.endFrameBarrier = World.Active.GetOrCreateSystem<
           EndSimulationEntityCommandBufferSystem>();
21   }
22
23   protected override JobHandle OnUpdate(JobHandle inputDeps)
24   {
25     var destroyJobHandle = new DeathJob
26     {
27       CommandBuffer = this.endFrameBarrier.CreateCommandBuffer().
             ToConcurrent(),
28     }.Schedule(this, inputDeps);
29     this.endFrameBarrier.AddJobHandleForProducer(destroyJobHandle);
30     return destroyJobHandle;
31   }
32 }
```

Source code 3: Example of a simple ECS system with a job

4.4.4 Burst Compiler

Burst is a compiler, capable of creating highly-optimized machine code from C# jobs by using LLVM[13]. It is working on a small subset of .NET and, therefore, adds more limitations to jobs – usage of any managed objects/reference types inside job is not allowed. In order to compile the job with Burst, the job must be marked with the *BurstCompile* attribute [41].

Benefits of using ECS with Job System and Burst were described in the official Unity's video series [42] on the simple example. In the example a large number of space ships are being spawned. Each ship is constantly moving forward. Figure 10 shows the results of the simulation.

(a) Old GameObjects (b) ECS + Jobs + Burst

Figure 10: ECS Performance comparison. Source: [42]

4.5 Procedurally generated content in the game

Starting from this section, as needed terminology and the way this game is implemented are covered, follows description and implementation of PCG techniques and gameplay mechanics.

4.5.1 Dungeon Generation

Dungeon generation problem	
Task:	Generate unique 2D map on every request.
Input:	Dungeon size, seed value.
Output:	2D array that represents the dungeon.
Requirements:	1. Must provide a map with fully connected playable areas. 2. Map contains a minimal amount of a playable area. 3. Map is closed on borders. 4. Fast enough to execute at runtime.

Each element in the array is called tile. Tiles contain integer value, where 1 is a wall and 0 is a floor, indexes in the array then represent X and Y position

[13] More about LLVM

in 2D space. Fully connected means that the result can not contain floor tiles, that are unreachable from any point of the map.

4.5.1.1 "Naive" approach

When thinking about level generation, this approach may be the first one that comes to mind and serves as the starting point into PCG problems. Different versions of this algorithm are being referred to as Agent-based dungeon growing [10]. The main idea is to create random room, peek random direction and create corridor from the room in that direction, then repeat as many times as needed.

Algorithm 1 "Naive" approach

```
1:  procedure CreateMapNaive(mapSize)
2:      tilesArray ← [mapSize.heigth, mapSize.width]
3:      roomCounter ← 0
4:      desiredRoomAmount ← GetRoomAmount(mapSize)
5:      currentPosition ← GetRandomPosition()
6:      while roomCounter < desiredRoomAmount do
7:          room ← CreateRandomRoom(currentPosition)
8:          direction ← PickRandomDirection()
9:          corridor ← CreateCorridor(room, direction)
10:         SetRoomTiles(room, tilesArray)
11:         SetCorridorTiles(corridor, tilesArray)
12:         currenPosition ← corridor.EndPoint
13:         roomCounter ← roomCounter + 1
14:     end while
15:     return tilesArray
16: end procedure
```

Overlapping of rooms and corridors may occur, but it is not a problem, but rather a positive effect, that can lead to unpredictable and non-linear dungeons. The main thing to be aware of while implementing this algorithm, is map borders. The algorithm should not exceed them and must behave properly when reaches any. Also, when picking direction generator must assure that it does not choose a direction, which leads backwards. Connectivity is provided from the way the map is created.

The number of steps that this map generator executes depends only on the desired number of rooms. It uses map size only for restriction purposes. But, obviously, the number of rooms must be in some way bound to the map size, to utilize as much available space as possible. This implementation uses a simple formula, obtained by experimenting with the generator. For a 100×100 map, which is 10000 square units, 25-35 rooms result in pleasant output for most of the cases. So, on each square unit, the generator must spawn around 0.0025-0.0035 rooms.

But how can ECS and jobs be used here, to benefit from multithreading? First of all, the generator will be represented as a system, rather than a normal

procedure. An entity is used to provide the system with all required inputs. Input values are stored in the component. The system will then pick up this entity, read the data from the component and start the generation process.

The generator will create the first room and corridors from that room in all 4 directions. Then it executes 4 jobs, each having the corresponding corridor passed to it. Each job creates new rooms and corridors in parallel, starting from the end of the received corridor, and stores them in the queue. Actual tiles in the array are set later. To set the tiles, jobs can be created for every single room and corridor, thus utilize all available cores. A brief overview of how this is done in ECS is shown in source code 4.

```
1  public class BoardSystem : JobComponentSystem
2  {
3    [BurstCompile]
4    private struct CreateRoomsAndCorridorsJob : IJobParallelFor
5    {
6      [WriteOnly] public NativeQueue<RoomComponent>.Concurrent Rooms;
7      [WriteOnly] public NativeQueue<CorridorComponent>.Concurrent
           Corridors;
8      [DeallocateOnJobCompletion] [ReadOnly] public NativeArray<
           CorridorComponent> FirstCorridors;
9
10     public BoardComponent Board;
11     public int RoomCount;
12     public int RandomSeed;
13
14     public void Execute(int index)
15     {
16       var random = new Random((uint)(this.RandomSeed * (index + 1)));
17
18       var firstCorridor = this.FirstCorridors[index];
19       var room = this.CreateRoom(this.Board, firstCorridor, ref
             random);
20       this.Rooms.Enqueue(room);
21
22       for (var i = 0; i < this.RoomCount; i++)
23       {
24         var corridor = this.CreateCorridor(room, this.Board, ref
               random);
25         room = this.CreateRoom(this.Board, corridor, ref random);
26         this.Rooms.Enqueue(room);
27         this.Corridors.Enqueue(corridor);
28       }
29     }
30   }
31
32   // Properties declaration
33
34   protected override JobHandle OnUpdate(JobHandle inputDeps)
35   {
36     // Reading data from component and generator initializations
37
```

```
38      var roomCount = var roomCount = (int)(randomSize.x * randomSize.y
            * random.NextFloat(0.0025f, 0.0035f));
39
40      // There is no 2D NativeArray, so using flatten array instead
41      this.Tiles = new NativeArray<TileType>(board.Size.x * board.Size.
            y, Allocator.TempJob, NativeArrayOptions.UninitializedMemory)
            ;
42      // Everything is a wall at the beginnings
43      for (var j = 0; j < this.Tiles.Length; j++)
44        this.Tiles[j] = TileType.Wall;
45
46      this.RoomsQueue = new NativeQueue<RoomComponent>(Allocator.
            TempJob);
47      this.CorridorsQueue = new NativeQueue<CorridorComponent>(
            Allocator.TempJob);
48      var firstCorridors = new NativeArray<CorridorComponent>(4,
            Allocator.TempJob);
49
50      // setup fist room and corridors to all 4 directions
51      var firstRoom = CreateRoom(board, ref random);
52      this.RoomsQueue.Enqueue(firstRoom);
53      for (var c = 0; c < 4; c++)
54      {
55        firstCorridors[c] = this.CreateCorridor(firstRoom, board, ref
              random, c);
56        this.CorridorsQueue.Enqueue(firstCorridors[c]);
57      }
58
59      // Creating rooms and corridors in 4 parallel jobs
60      inputDeps = new CreateRoomsAndCorridorsJob
61      {
62        Board = board,
63        Rooms = this.RoomsQueue.ToConcurrent(),
64        Corridors = this.CorridorsQueue.ToConcurrent(),
65        FirstCorridors = firstCorridors,
66        RoomCount = roomCount / 4,
67        RandomSeed = random.NextInt()
68      }.Schedule(4, 1, inputDeps);
69
70      //Further execution and map controls
71      return inputDeps;
72    }
73  }
```

Source code 4: “Naive” approach implementation in ECS

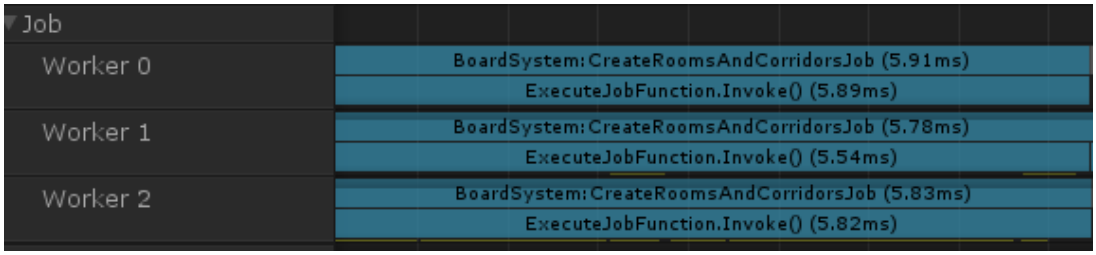


Figure 11: Parallel execution of jobs

This implementation has some further polishing steps:

- The initial room is spawned in the center of the map, to give more space for expanding.
- While picking a direction, there was added a 25% chance to move further from the center, depending on current quadrant of the map.
- Generator still may not utilize a lot of available space, so in the final step it finds actual boundaries of the playable area and cuts it out.

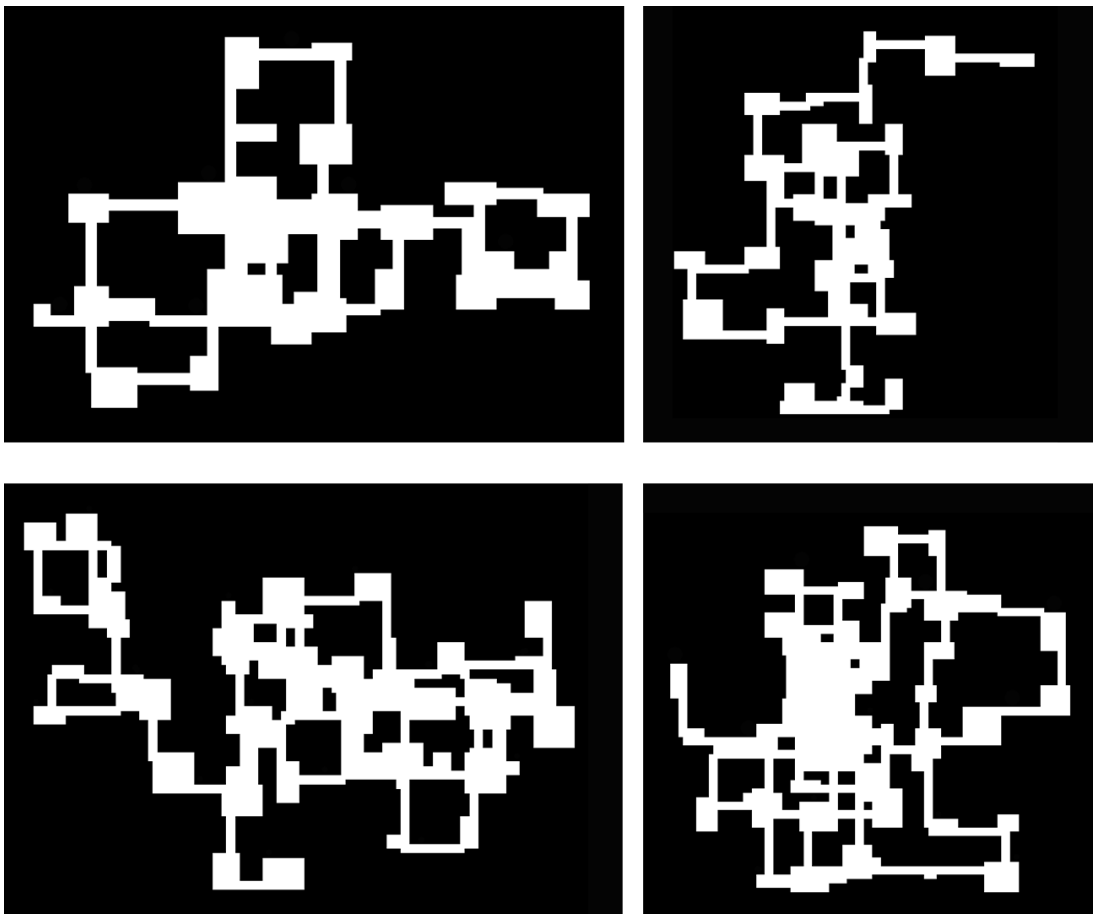


Figure 12: Final results of the "Naive" generation, where white is a playable area

Considering the worst-case scenario of having only 1 available core (WebGL [14]) and a squared map of size $N \times N$, which is n square units, time complexity can be determined as follows:

1. Creating the first room and first 4 corridors is done in time $O(s)$.
2. The number of desired rooms is $0.0035n$ at maximum (assume that is R). The loop to create rooms and corridors will execute R times, each iteration will create 1 room in time r steps and 1 corridor in c steps, thus $O(R(r+c))$.
3. Loop over all rooms. Every room has defined maximum size of M square units, that it can not exceed. For each room system will set corresponding tiles in $O(M)$, thus $O(R \cdot M)$.
4. Loop over all corridors. A total number of corridors is $numbersOfRooms+2$. Corridors have defined maximum length L. For each corridor system will set corresponding tiles in maximum L steps, thus $O(L(R+2))$

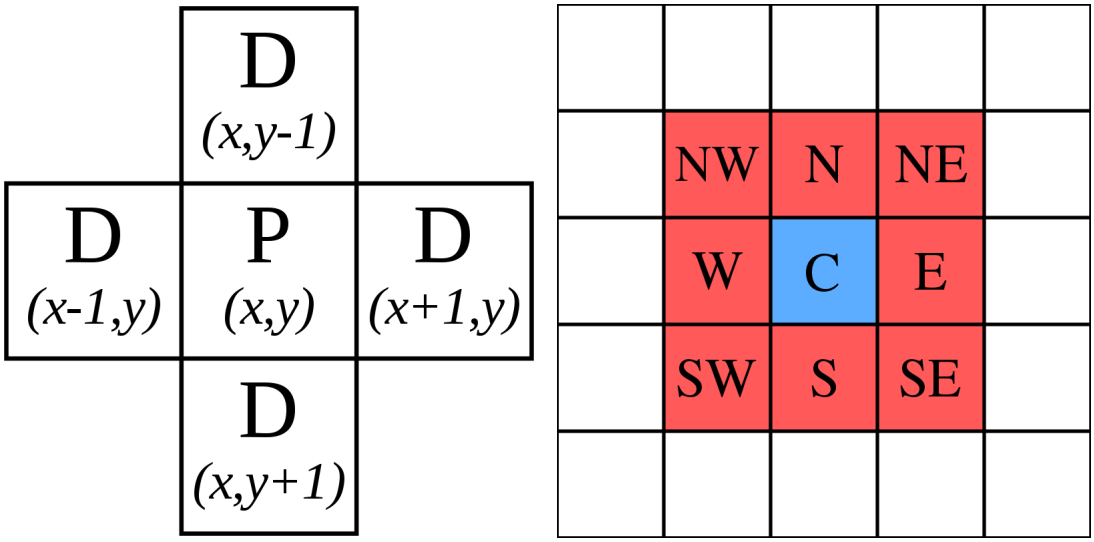
Adding all together, time is $O(s + R(r + c) + R \cdot M^2 + L(R + 2))$. Creating rooms and corridors is done in a constant number of steps, hence can be emitted from the equation. Maximum room size and corridor length are limited by a constant, so can be also emitted, but those values must be chosen wisely. The final time complexity then:

$$O(R + R + R) = O(n) \tag{1}$$

4.5.1.2 Cellular automaton

A Cellular Automaton (CA) is a discrete computational model, that consists of an n-dimensional grid, a set of states and a set of transition rules. Each cell in a grid has defined a state. CA evolves in generations, by changing the states of its cells. At each generation, the cell calculates its state based on the state of surrounding cells. A set of cells that have an affect on the current cell is called the neighbourhood. The most known types of neighbourhoods are *von Neumann neighborhood*[15] and *Moore neighborhood*[16]. CA is a point of interest for many scientific fields, such as computer science, biology, physics and mathematics [10], [43].

[14] WebGL specifications
[15] Von Neumann neighborhood
[16] Moore neighborhood



(a) Von Neumann neighborhood. Source: [44]

(b) Moore neighborhood. Source: [45]

Figure 13: Neighborhood types

This implementation uses the Moore neighbourhood and the simplest case of CA, where a cell can be only in 2 states – *on* or *off*. In the initial generation, each cell gets a random state. At further generations, every cell will check its neighbourhood. If more than 4 cells are currently in the *off* state, the cell will be turned *off*, otherwise, it will be turned *on*. After several generations, cells with the same state will be grouped together, forming *regions*. Regions with cells that have *on* state are considered as playable area and are called *rooms*. Rooms can be separated from each other, therefore must be determined and connected.

Algorithm 2 Cellular automaton

```
1:  procedure CreateMapCellular(mapSize, genCount, randomFillPercent)
2:      cellsArray ← [mapSize.heigth, mapSize.width]
3:      counter ← 0
4:      RandomFillMap(cellsArray, randomFillPercent)
5:      while counter < genCount do
6:          CalculateNextState(cellsArray)
7:          SetNextState(cellsArray)
8:          counter ← counter + 1
9:      end while
10:     roomsArray ← FindRooms(cellsArray)
11:     ConnectRooms(roomsArray, cellsArray)
12:     tilesArray ← ConvertToTiles(cellsArray)
13:     return tilesArray
14: end procedure
```

In algorithm 2 *randomFillPercent* parameter decides the chance of the cell

being *off* in the initial generation. Larger value will result in more cells being *off*.

To find actual rooms this implementation uses the iterative version of the *flood fill algorithm*[17]. The algorithm scans the whole map. When it encounters the cell with the *on* state that has not been processed yet, it starts gathering all surrounding cells with the *on* state and adds them to the queue. Then the process repeats for every cell in the queue. All cells collected in that way will form the room.

Algorithm 3 Flood fill

```
1:  procedure FindRooms(mapSize, cellsArray)
2:      flags ← [mapSize.heigth, mapSize.width]
3:      rooms ← ∅
4:      for y ← 0, mapSize.heigth do
5:          for y ← 0, mapSize.width do
6:              if flags[y, x] = false and cellsArray[y, x].State = on then
7:                  room
8:                  cell ← cellsArray[y, x]
9:                  flags[y, x] ← true
10:                 room.Cells ← CollectCells(cell, flags, cellsArray)
11:                 rooms.Add(room)
12:             end if
13:         end for
14:     end for
15:     return rooms
16: end procedure
17: procedure CollectCells(startCell, flags, cellsArray)
18:     foundCells ← ∅
19:     queue ← ∅
20:     queue.Enqueue(startCell)
21:     while queue.Length > 0 do
22:         cell ← queue.Dequeue()
23:         for all c ∈ GetCellsAround(cell, cellsArray)) do
24:             if flags[c.y, c.x] = false and c.State = on then
25:                 foundCells.Add(c)
26:                 queue.Enqueue(c)
27:                 flags[c.y, c.x] ← true
28:             end if
29:         end for
30:     end while
31:     return foundCells
32: end procedure
```

Connection between all collected rooms is established in 2 stages. First of all, the generator creates corridors between pairs of the nearest rooms. It then

[17] Flood fill algorithm

checks the connection between all the rooms and creates new ones, if necessary.

In the first stage, the algorithm loops over all rooms. For each room, it finds nearest by comparing tiles of that room with tiles of every other room and stores the best result. The corridor is created between nearest tiles. To track which rooms are connected, the algorithm uses a table in which rows and columns represent rooms. Each value in the table cell indicates whether the rooms are connected, where 0 means no connection and 1 means there is a connection. In the initial state, the table has all values set to 1 on the main diagonal, which means that room is connected with itself. When the algorithm connects 2 rooms it sets appropriate cell to 1, and it also sets a proper cell value for all rooms that are connected to any of those rooms. Thus, the algorithm is able to determine whether rooms are connected directly, or through other rooms. When searching for the nearest room, the algorithm additionally checks if rooms have already been connected, and if so, it skips to the next one. There was added an additional condition to the *flood fill algorithm* – the cell is added to the result only if there is at least one *off* cell around. With this condition the algorithm will add cells that are most likely room boundaries. With this approach some unnecessary iterations can be skipped when searching for the nearest room.

In the second stage, the algorithm marks the first room as the *main room* end verifies if every room is reachable from the main room by simply checking table values. If it finds the room that is not reachable, it stores all reachable rooms in one list and all unreachable in another list. Then it finds 2 closest rooms between those lists and creates the connection. It repeats the process until there are no unreachable rooms left. This approach can create the natural looking cave, rather than chaotic dungeon.

To dig the actual corridor between 2 points on the map *Bresenham's line algorithm*[18] was used.

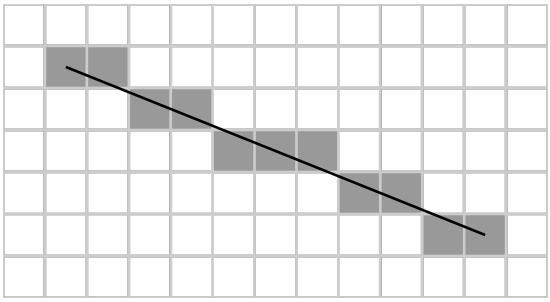
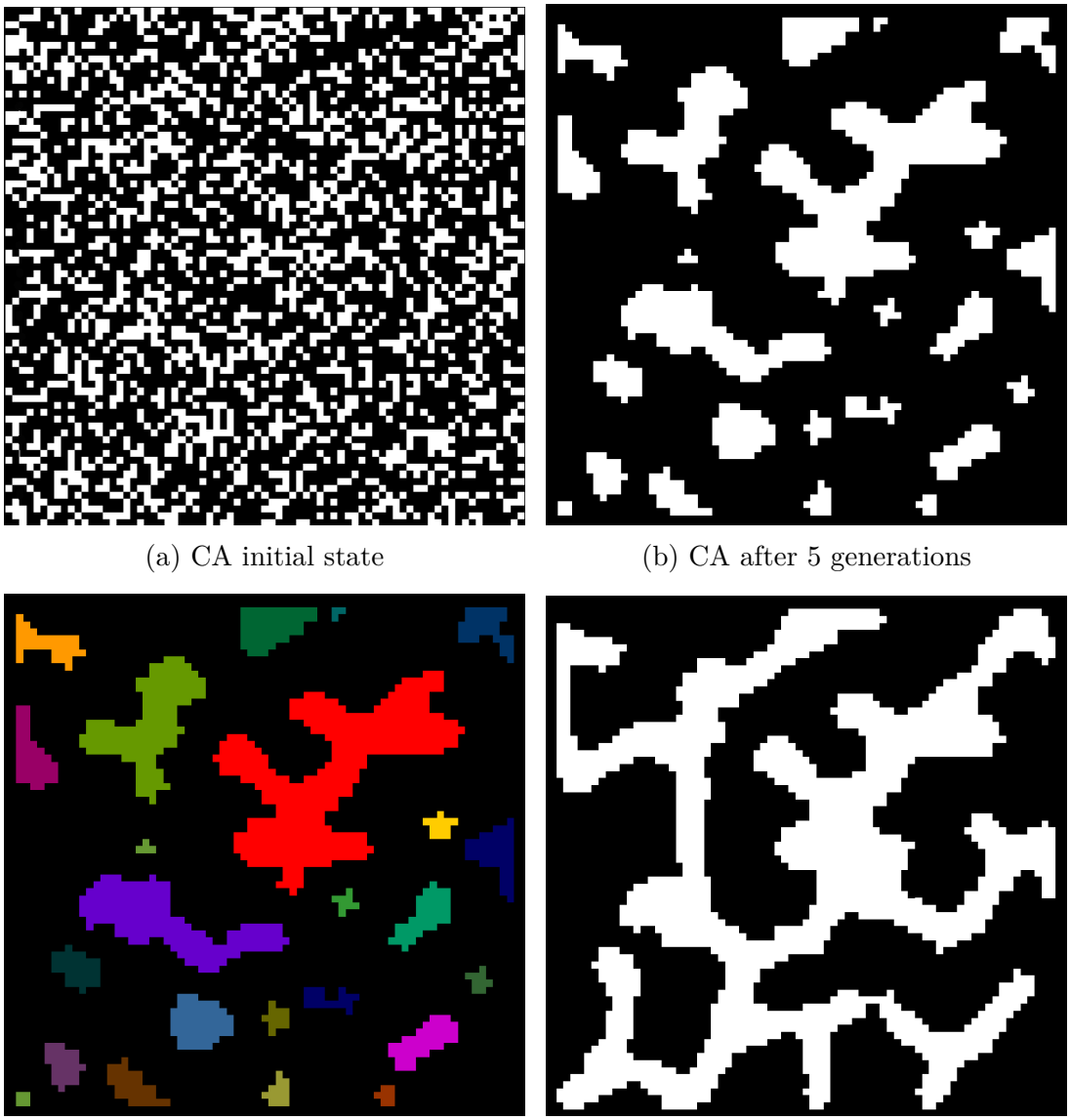


Figure 14: Bresenham's line algorithm. Source: [46]

[18] Bresenham's line algorithm



(a) CA initial state

(b) CA after 5 generations

(c) Collecting rooms

(d) Result with connected rooms

Figure 15: Cellular automaton map creation process. Map size is 75×75 and $randomFillPercent$ set to 65

ECS and jobs were used to speed up the process. Iteration over the map is done by creating a job for every row. Each job then processes its row in parallel. That approach used on initializing the CA with random states and when calculating states for the next generation. Flood fill was not parallelized, to avoid race conditions when setting the flags and to avoid multiple processing of the same room. Jobs were used when searching for the nearest rooms by splitting the map on smaller quads. Each job connects rooms, that lie within its quad. After that one single job connects all rooms together.

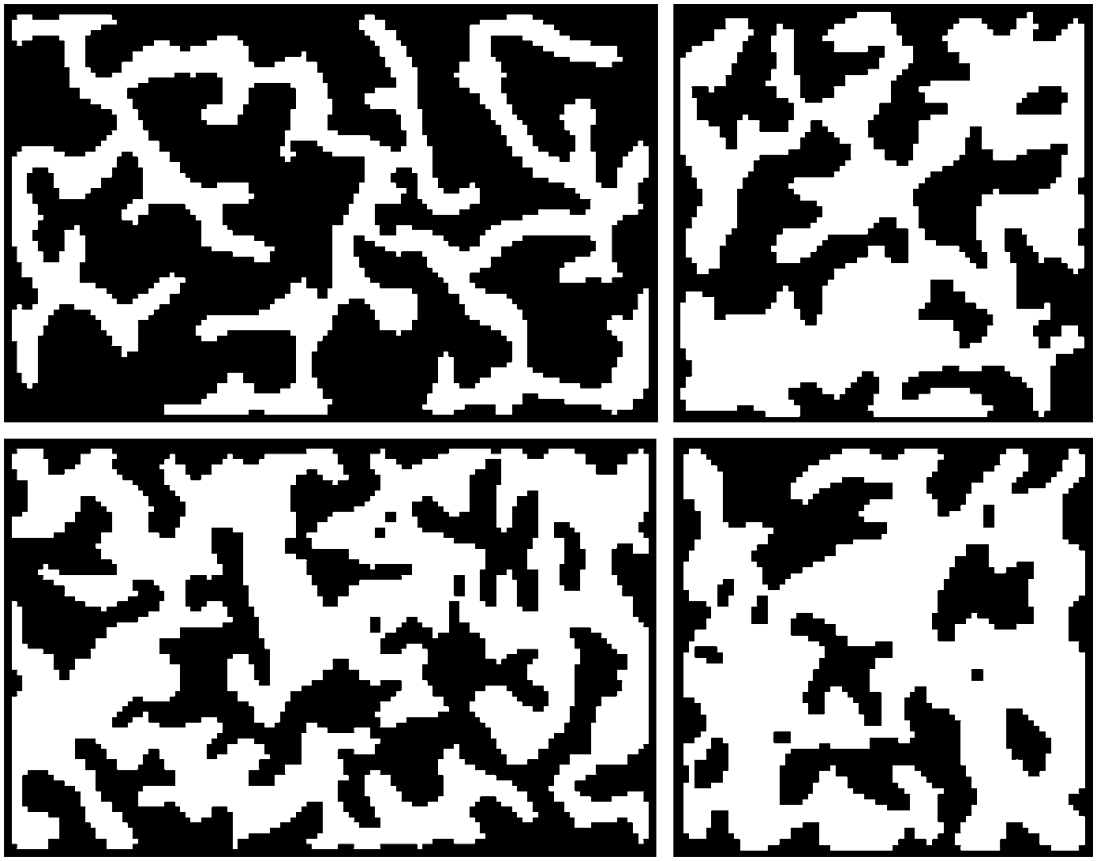


Figure 16: Final results of the Cellular dungeon automaton generation, where white is a playable area

Considering the worst-case scenario with 1 core and a map of size $N \times N$, which is n square units, time complexity:

1. Initializing the CA with random states is $O(n)$.

2. This implementation uses 5 generations to evolve, so $O(10n)$, because it needs one pass to calculate the next state of every cell and additional pass to set new state.

3. Flood fill will scan the entire map in $O(n)$. When it encounters not processed cell, the number of steps is proportional to the number of cells in the filled area. When the whole map is a flooding area, then it is $O(n)$. Since flood fill does not execute the gathering on already processed cells it will be $O(2n)$ at maximum.

4. The number of steps required to connect the rooms depends on the number of rooms and the size of each room. The number of rooms in CA is not strictly bound to the size of the map. It also depends on the $randomFillPercent$ and how CA evolves through generations. With the low $randomFillPercent$ value, the map can have a single room, that covers

almost the entire map, and with the large value map with a single room but of small size may also be generated, both resulting in $O(1)$. But in average CA will create more separated rooms as grid size is growing. Assume that the number of rooms is R. Finding the connection between rooms is done by comparing distances between every cell in each room. If rooms are the same size, they will have at maximum $\frac{n}{R}$ cells each. To create the actual corridor, the algorithm needs to set cells between closest rooms to on state. When rooms are in opposite corners, the length of the corridor is N at maximum. Hence, connecting all rooms is $O(R^2(\frac{n}{R} + N))$.

5. After that, one generation is lived, to remove unnecessary cells created by digging the corridors, which is $O(n)$.

Summing all up, time complexity is $O(n+10n+2n+R^2(\frac{n}{R}+N)+n)$. Assuming that the number of rooms is bound to the map size, and is growing proportionally, than $R = c{\cdot}n$, where c is a constant such that $0 < c < 1$. Removing the constants, the final result in the worst-case:

$$O(n + n + n + n^2 + n) = O(n^2) \tag{2}$$

4.5.1.3 BSP-tree

Binary Space Partitioning (BSP) is the most popular method for space partitioning. Space partitioning is a subdivision of space (typically 2D or 3D) into disjoint subsets. To store subsets and operate on them, trees are usually used. BSP divides the space recursively into two subsets, which allows using of binary trees. Such a binary tree is called *BSP-tree*. They are commonly used in rendering, raytracing and collision detection to operate on already existing data, rather than to create new ones. But because such subsets are disjoint, they can be used to represent rooms in the dungeon [10].

Each node of the BSP-tree holds a certain area of the map. The root node holds the entire map. The algorithm starts in the root node, where it splits the space held by the root in 2 parts vertically or horizontally. Those parts are assigned to the left and right child nodes. Then it repeats the process recursively until space can not be divided further. Actual rooms are created in the leaf nodes. Visualization of the BSP process is shown in figure 17.

Algorithm 4 BSP-tree

```
1: procedure CreateMapBSP(mapSize, minSpaceSize)
2:     tree ← InitTree(mapSize)
3:     SplitNode(tree.Root, minSpaceSize)
4:     RandomFillMap(cellsArray, randomFillPercent)
5:     level ← tree.Height
6:     while level > 0 do
7:         ConnectNodes(tree, level)
8:         level ← level − 1
9:     end while
10:    tilesArray ← ConvertToTiles(tree)
11:    return tilesArray
12: end procedure
13: procedure SplitNode(node, minSpaceSize)
14:    if node.Space/2 < minSpaceSize then
15:        CreateRoom(node)
16:        return
17:    end if
18:    DivideSpace(node)
19:    SplitNode(node.LeftChild, minSpaceSize)
20:    SplitNode(node.RightChild, minSpaceSize)
21: end procedure
```

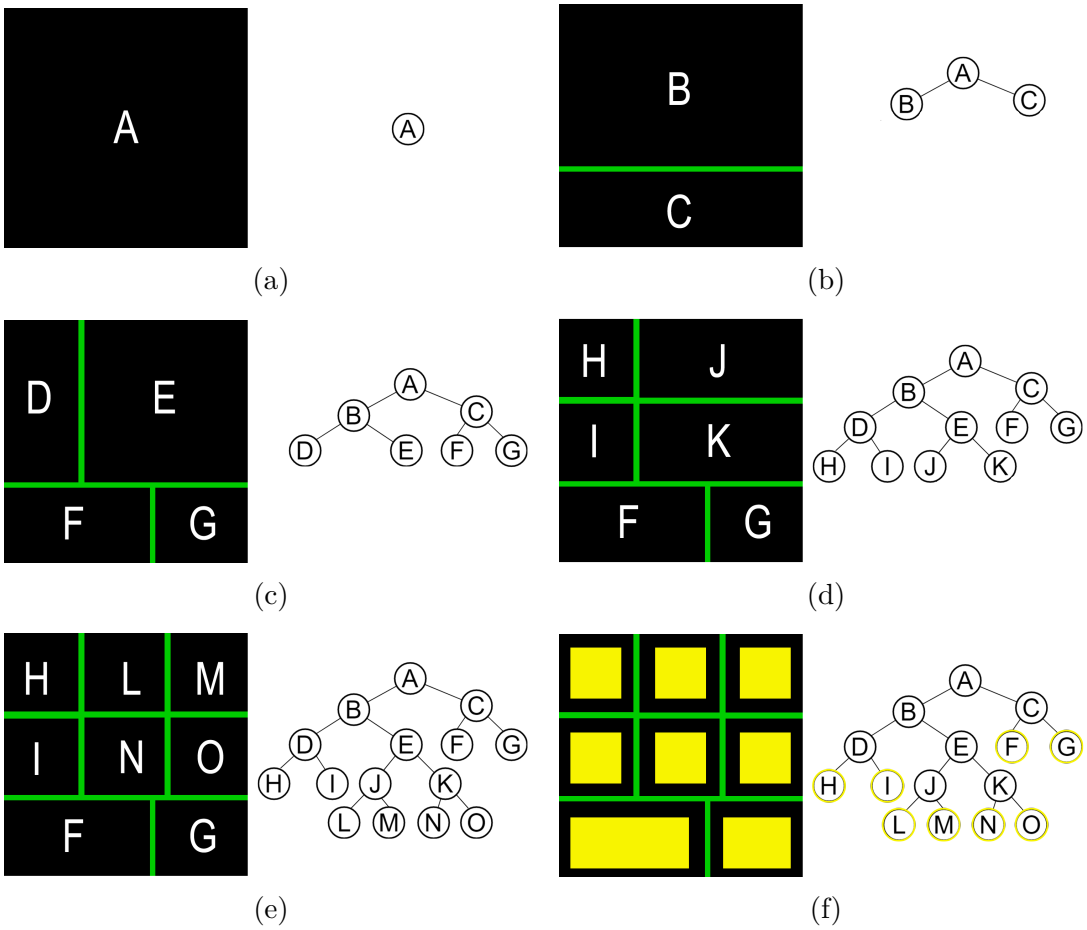


Figure 17: BSP-tree creation process

The algorithm connects rooms by creating corridors between left and right child of every node in each level of the tree starting from the penultimate level. The last level is skipped because leaves can not connect anything. When the algorithm reaches the root, all rooms are connected. To create the connection between 2 rooms, it picks random points in each room and digs the corridor between them. If those points are not aligned vertically or horizontally, it digs the corridor with a turn as shown in figure 18.

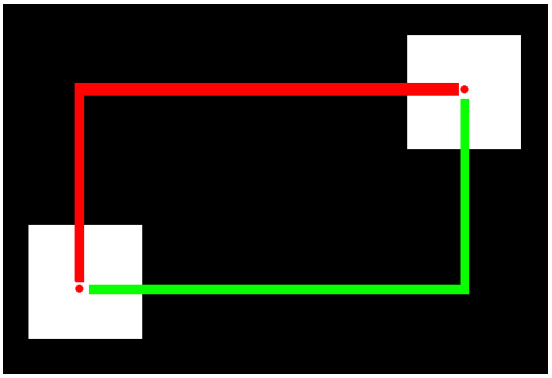


Figure 18: Digging the corridor

Actual rooms lie in leaves. When connecting 2 nodes, the algorithm will recursively traverse the tree down from those nodes, until it reaches the nearest leaf. Then it creates the actual corridor between rooms of those leaves.

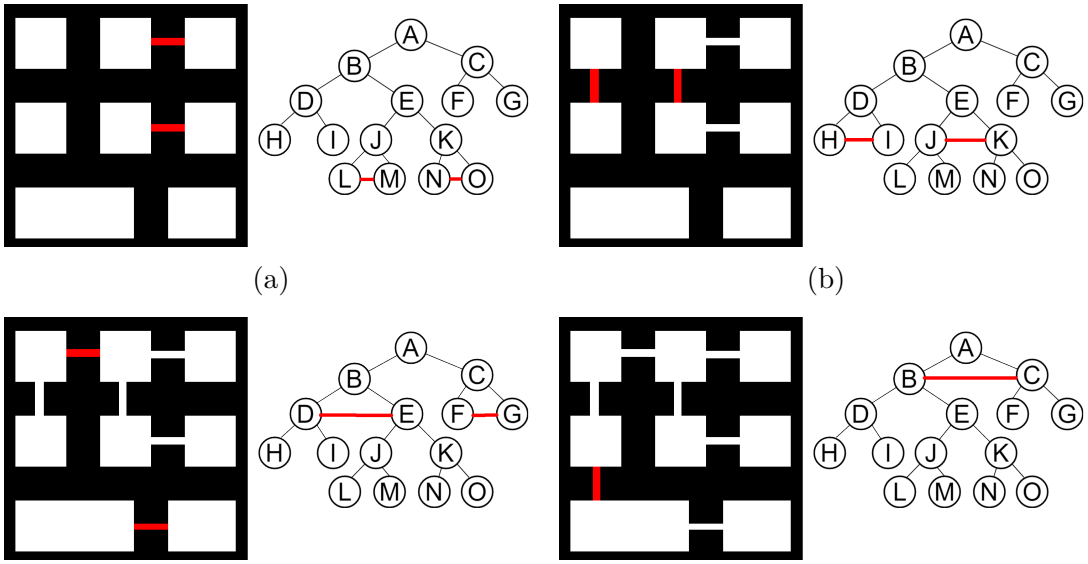


Figure 19: BSP-tree rooms connection

ECS and jobs are used here to create rooms, corridors set map tiles. To use the tree in the jobs, it is converted to the array. The BSP-tree is not guaranteed to be perfectly balanced, so the array is filled with additional null nodes. To find corridors in each level, a job is created for every node, starting from the penultimate level. Every job is responsible for connecting node's left and right child.

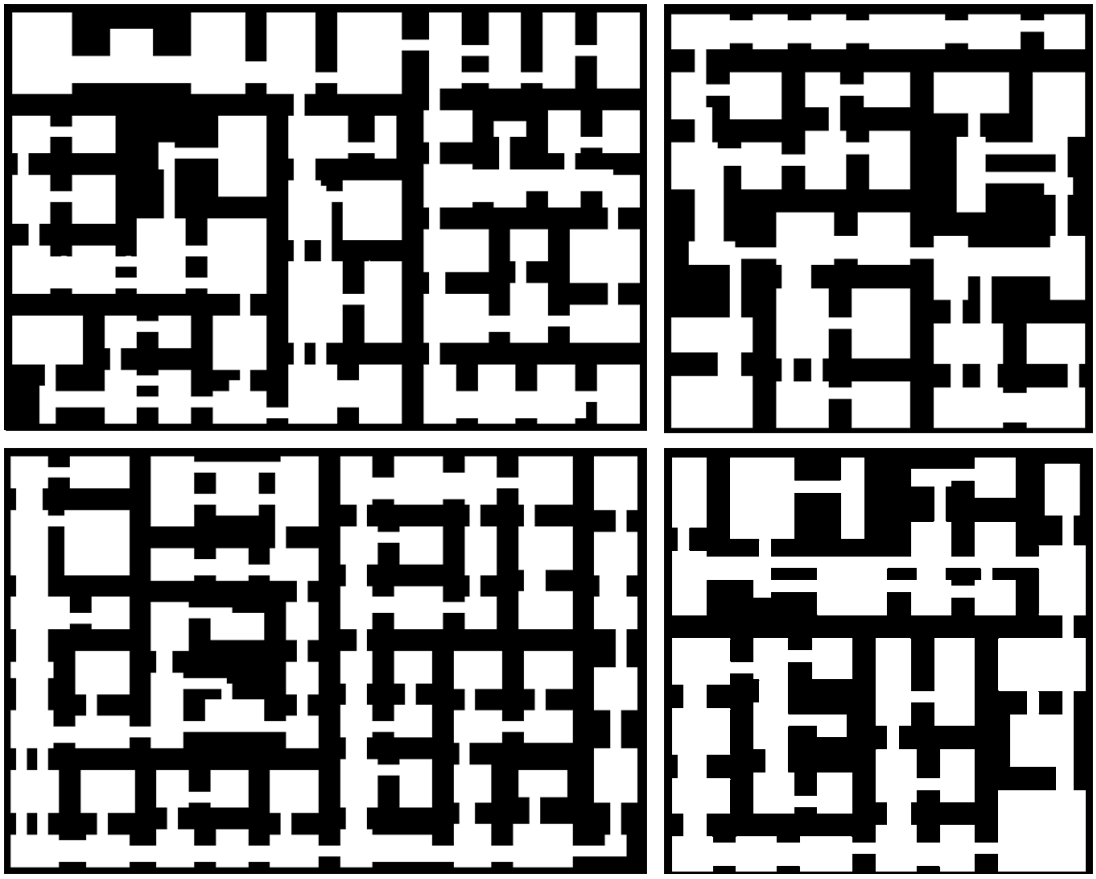


Figure 20: Final results of the BSP-tree dungeon generation, where white is a playable area

Considering 1 core and map size $N \times N$, which is n square units, once more, time:

1. If space is divided in 2 exact same parts all the time and minimal space size is m square units, then the tree will become a perfect binary tree and the map can fit at most $\frac{n}{m}$ rooms, thus the number of leaves is also $\frac{n}{m}$, assuming that is L. Total number of nodes in such tree is then $2L-1$. The height of the tree is than $h = \log_2(2L)$.

2. Splitting the space is done in non-leaf nodes. There are $L-1$ of them in this tree So $O(L-1)$ for the BSP-tree creation.

3. Tree is converted to the array, which is $O(2L-1)$.

4. Rooms are created in leaves and there are L of them. So $O(L)$ is the amount of work needed to create rooms.

5. Rooms are connected in each level starting from the bottom. Only non-leaf nodes are responsible for the corridor creation (left child connects to right child) and the root node stops the process. So, $L-1$ nodes that will

execute connection searching process. To connect the nodes, nearest leaf needs to be found, that is done in h steps maximum, since all leaves have the same depth. To connect left and right child nodes, a leaf needs to be found for both of them. So $(L-1) \cdot 2h = 2(L-1)(\log_2(2L)) = O(L \log_2(L))$, corridors are stored in the queue as a pair of points.

6. Setting the tiles in the array then executed on each room and each corridor. There are L rooms of m square units each, so $L \cdot m$ to set room tiles. Corridors are created in each level between siblings, so there are $(L-1)$ corridors created overall. Assuming that maximal length of the corridor can grow up to the N – $O(L \cdot m + N(L-1))$ to set actual tiles.

Summing all up, when BSP-tree becomes a perfect binary tree, the time is $O((L-1) + (2L-1) + L + L\log_2(L) + L \cdot m + N(L-1)) = O(\frac{n}{m} + \frac{n}{m} + \frac{n}{m} + \frac{n}{m}\log_2(\frac{n}{m}) + n + N\frac{n}{m})$, therefore:

$$O(\frac{n}{m}\log_2(\frac{n}{m})) = O(n\log_2(n)) \tag{3}$$

When tree becomes unbalanced, searching of the leaf node becomes linear and overall time complexity moves towards $O(n^2)$.

4.5.1.4 Comparison

To test the running time of implemented algorithms, 100 dungeons of size 500×500 with randomized inputs were generated by each. Results are in seconds. Test was run as a standalone build using IL2CPP on the i3 1.9Ghz Dual-core 4 threads CPU.

	"Naive"	CA	BSP-tree
Minimal	0.1	0.49	0.12
Maximal	0.98	69.38	2.61
Average	0.19	15.22	0.54

Table 1: Comparison of implemented algorithms

In the game map size varies from 75×50 up to 150×150 tiles, such a map is large enough to give the player space to explore, and it is not overwhelming at the same time. Implemented algorithms will create a map of such size in a very similar time span, that erases the difference between them. If bigger maps are required, there should be done more optimization steps, for example, creating the dungeon in smaller parts and then merging them together, or generating parts of the map only when a player reaches the border of the current part.

As for dungeons themselves, "Naive" generator is fast, but does not provide a lot of control and creates chaotic maps. BSP-tree is a reasonable choice, as it creates maps that look like real man-made dungeons or prisons. CA generates natural-looking caves. CA with a bigger number of states can generate more

diverse maps, for example, 4 states in the CA could represent ground, bushes, rocks and trees to create a forest map.

4.5.1.5 Tile map

Results obtained from dungeon generators now can be visualized. The game has a set of hand-drawn tiles, 32×32 pixels each. Such a set is called a tilesheet. Every element of the array that the generator provides is represented by those tiles. Depending on the environment of the element, the most suitable tile is picked. This can be done manually by iterating through the whole map and scanning all neighbours of the current element, or by using Unity's built-in tool called *Tilemap component*[19] and a additional Unity's package that is called *2d-extras*[20]. With this tool, the rules for tiles can be set directly in the editor. It also offers the ability to generate a collider for the tile map.

(a) Tilesheet (b) Tile map rules in Unity

Figure 21: Creating the tile map with Unity

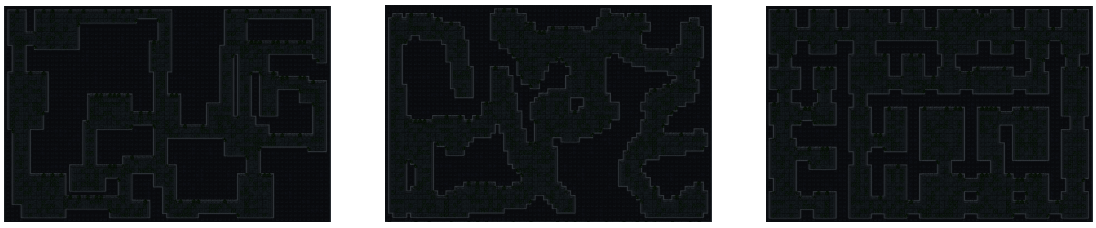


Figure 22: Generated maps converted to tile map

[19] About Unity's Tilemap component
[20] 2d-extras Github repository

4.5.2 Object placement

Object placement problem	
Task:	Determine positions on the map where objects should be placed.
Input:	Generated 2D map.
Output:	Array of positions.
Requirements:	1. Positions can not overlap with each other. 2. Positions should uniformly cover as much space as possible. 3. Fast enough to execute at runtime.

4.5.2.1 Player spawning

The position of the player must be determined first, so that it can be used later, for example, when placing exits from the dungeon it is undesirable to spawn one directly next to the player.

The player's starting position on the map is chosen in a simple manner. Random corner of the map is picked. From that corner, the procedure starts to search for a valid tile. When such tile is found the player is spawned and other systems can take into consideration his position. Placing the player in the corner of the map may help him navigate faster in a new dungeon.

4.5.2.2 Poisson disk sampling

Poisson disk sampling (PDS) is a technique that produces blue noise sample patterns. Blue noise found the application in computer graphics, particularly in rendering. This type of noise is used to create particle systems, motion blur or depth-of-field effects. PDS distributes the samples so that they are at some minimum distance r from each other [47]. This game uses a 2D modification of the PDS to determine the positions of objects. The algorithm initializes the background grid, that is used to store samples. PDS places the initial sample in the random position, adds it to the active list and marks the corresponding cell in the grid. While there are samples left in the active list, pick a random sample S. Pick a random direction and a random distance in between r and $2r$. Move to a new position from the S. If the position is valid for a new sample to be placed – create a new sample, add it to the active list, mark the background grid and repeat the process. If the proper position was not found in a defined k steps – remove S from the active list and continue the execution. Validation of the position is done by searching for existing samples around the position. Since k and r are constants overall time complexity of this PDS implementation is $O(n)$, where n is the number of tiles that PDS receives as input [47].

Algorithm 5 Poisson disk sampling

```
1: procedure GetPositionsPDS(mapSize, mapTiles, radius, limit)
2:     grid ← [mapSize.height, mapSize.width]
3:     activeList ← ∅
4:     sample ← GetRandomSample(mapTiles)
5:     activeList.Add(sample)
6:     grid[sample.Position.y, sample.Position.x] ← sample
7:     while activeList.Length > 0 do
8:         sample ← GetRandomSample(activeList)
9:         counter ← 0
10:        candidateFound ← false
11:        while counter < limit do
12:            newPos ← sample.Position + randomVector
13:            if IsValid(newPos, grid) then
14:                newSample ← CreateSample(newPos)
15:                activeList.Add(newSample)
16:                grid[newSample.Position] ← newSample
17:                candidateFound ← true
18:                break
19:            end if
20:            counter ← counter + 1
21:        end while
22:        if !candidateFound then
23:            activeList.Remove(sample)
24:        end if
25:    end while
26:    positions ← GetSamplePositions(grid)
27:    return positions
28: end procedure
29: procedure IsValid(position, grid, radius)
30:    for all s ∈ GetSamplesAround(position, grid, radius) do
31:        if GetDistance(s.Position, position) < 2 * radius then
32:            return false
33:        end if
34:    end for
35:    return true
36: end procedure
```

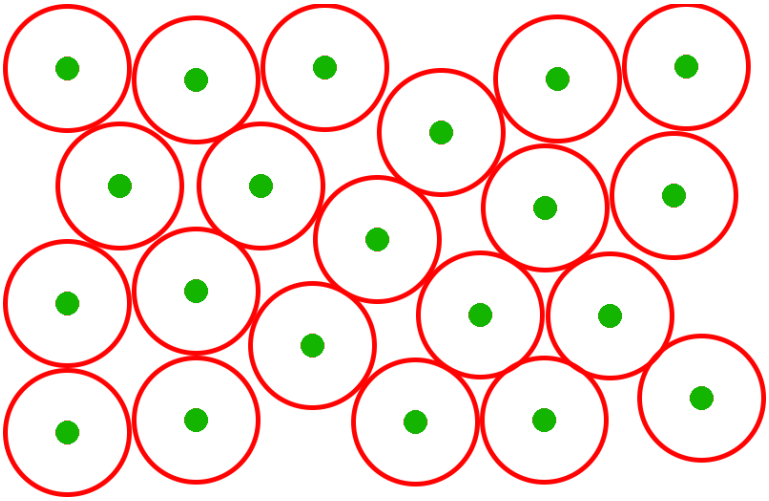


Figure 23: Poisson disk sampling on a 2D plane

PDS by itself if not suitable for calculating the positions in the dungeons, because it does not take into consideration the obstacles like walls. There may occur a situation when PDS will not cover the whole area due to that. Example of such a situation is visualized in figure 24, where PDS is not guaranteed to reach the room on the right. This limitation has been overcome by keeping track of all valid tiles that have not been covered by PDS yet. When the active list gets emptied, there is an additional verification assuring that there are no uncovered tiles left. If there is any, it is added to the active list and the PDS continues the execution.

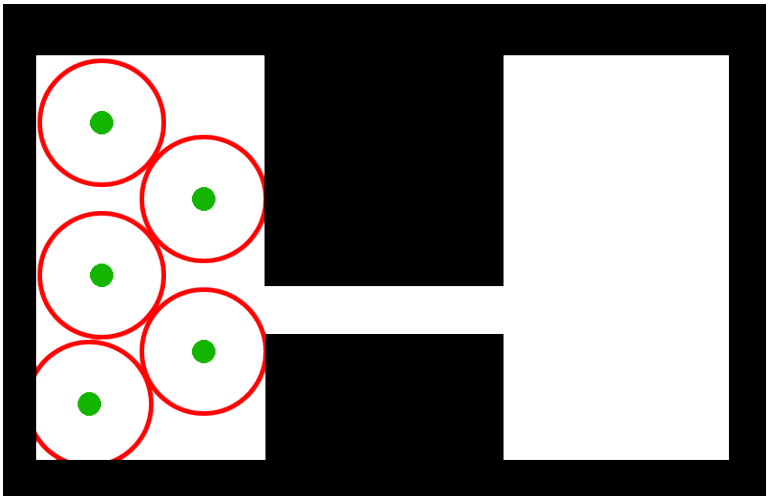


Figure 24: PDS problem in dungeons

This game extends PDS further and allows samples to have different radiuses. Enemies, dungeon exits, chests, torches on the walls and cages are spawned with the use of PDS.

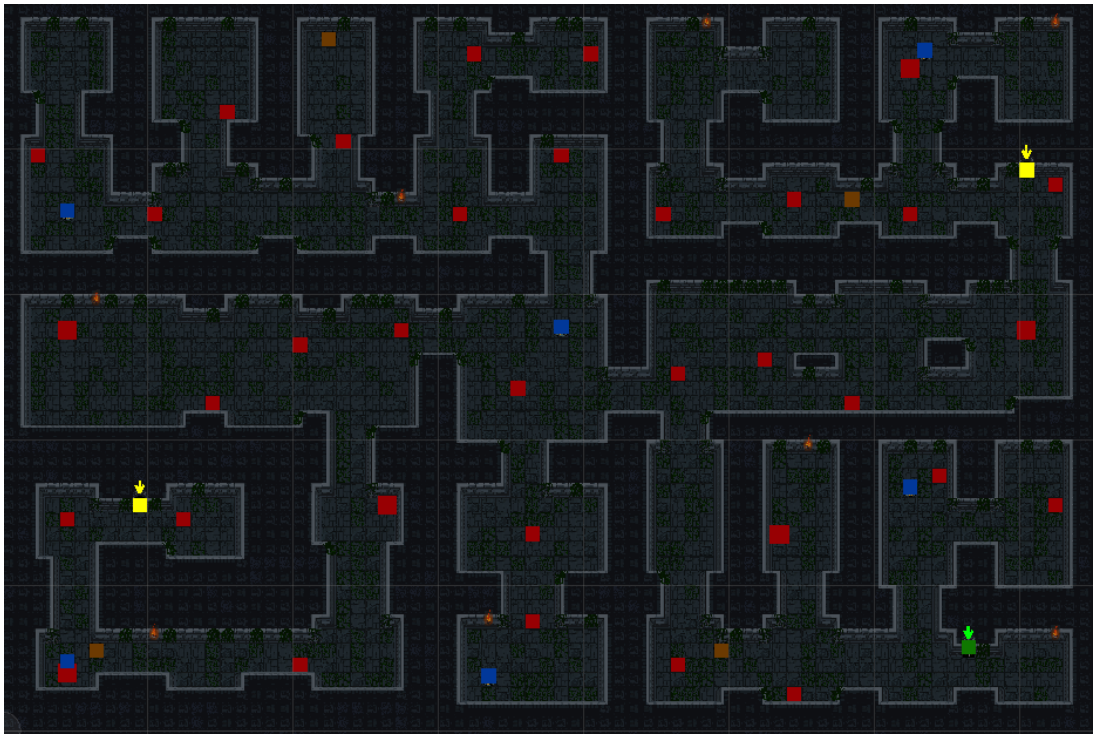


Figure 25: Results of PDS in the dungeon, where yellow squares are exits, red – enemies, brown – chests, blue – cages and the green square is the player

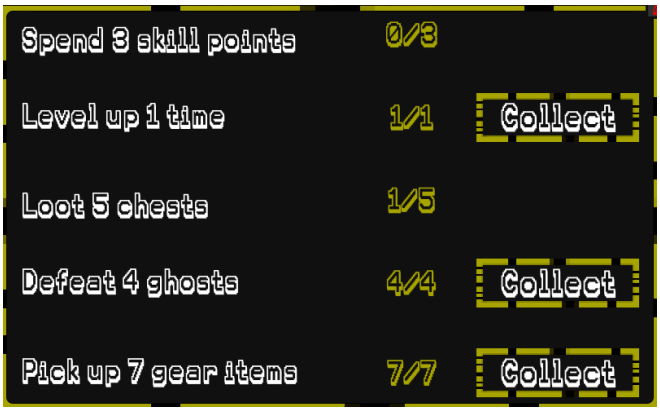
4.5.3 Objectives generation

Without a clear target in the game, player may feel bored overtime. Quests and different objectives in games serve as a way of creating variety to the gameplay and providing the reason for the player to play. Completing the missions gives the player a feeling of accomplishment and stimulates him to play more.

This game features a simple set of objectives for the player to complete, like killing enemies, gathering items etc. The player holds 5 active quests. When any quest is completed, a new one will be generated instantly. The system ensures that duplicate quests are not created. Those tasks are generated at runtime by using formal grammars. Grammar G consists of a terminal T and non-terminal N symbols, a set of rewriting rules R and a starting non-terminal symbol S. Quests are generated by applying rewrite rules until there are no non-terminal symbols left, beginning from the start symbol.

$$
\begin{aligned}
G &= \langle N, T, R, S \rangle \\
N &= \{\text{:Quest, :Kill, :Find, :Release, :Loot, :Item,} \\
&\quad \text{:LevelUp, :SpendSkillPoint, :randomNumber}\} \\
S &= : Quest \\
R &= \{ \\
: Quest &\rightarrow \text{:Kill} \mid \text{:Find} \mid \text{:Release} \mid \\
&\quad \text{:Loot} \mid \text{:LevelUp} \mid \text{:SpendSkillPoint} \\
: Kill &\rightarrow \text{Defeat :randomNumber enemies} \\
: Find &\rightarrow \text{Pick up :randomNumber :Item} \\
: Release &\rightarrow \text{Release :randomNumber chickens} \\
: Loot &\rightarrow \text{Loot :randomNumber chests} \\
: LevelUp &\rightarrow \text{Level up :randomNumber times} \\
: SpendSkillPoint &\rightarrow \text{Spend :randomNumber skill points} \\
: Item &\rightarrow \text{food item} \mid \text{potion} \mid \text{gear item} \\
: randomNumber &\rightarrow 1, \ldots, 15\}
\end{aligned}
$$

(a) The grammar used in this game



(b) Generated quests

Figure 26: Quests generation in the game

4.5.4 Leveling and stats

The game features basic leveling mechanics, to let the player track the progression and to evolve the character. The amount of experience needed to reach a new level doubles with every level. Experience can be earned by defeating enemies, opening chests and cages, completing objectives. When a new level is reached, the player receives a skill point that he can spend to improve one of the four available characteristics. Those stats are:

- Attack – responsible for the amount of damage dealt by the weapon.
- Defence – responsible for the amount of incoming damage dealt by the weapons.
- Health – responsible for the amount of total health.
- Magic – responsible for multiple things at once that are relative to the magic. This stat adjusts the amount of incoming and outgoing damage dealt by spells, spell cooldowns and spell cast times.

Enemies have the exact same set of characteristics. When the new dungeon is generated, the level of the enemies will be set depending on the current level of the player. Then available skill points will be spent randomly, creating more varied enemies. This technique is called auto-leveling, and in the bigger games allows players to explore the world without limitations.

4.5.5 Items generation

There are different types of items in the game: so-called consumable items and gear items. Consumables are presented in the form of food and potions, that have only one purpose – restore lost health points. Gear items affect characteristics of the owner. These items are swords, helmets, chest armour pieces, boots and spellbooks. Every item can hold up to 4 stat modifiers at the same time with a certain modifier always present, such as attack on a sword. When an item is generated it will receive a guaranteed modifier. Then there is a 50% chance to receive the second modifier. If the item gets the second modifier, then it has a 30% chance of getting the third and after the third one, item has a 10% chance to receive the fourth. Items can be dropped from enemies and chests. When chests and enemies are spawned, the system will give those randomized items. Enemies will equip gear items, hence create even more differences between them. When an enemy is killed, each item belonging to the enemy, with a probability of 15% will drop out. Chests will drop everything that they store.

Figure 27: Generated items

4.6 Basic game mechanics

4.6.1 Player

The player has the ability to move in 4 directions, do melee attacks, collect, use and equip various items found in dungeons, spend skill points on character stats, cast 1 of the 3 available spells, 2 of them being damage dealing and 1 being self-healing, complete objectives and explore generated dungeons.

4.6.2 Camera setup

There is a trivial camera setup done by using Unity's *Cinemachine*[21] tool. This tool allows to adjust different camera settings directly in the editor and has a lot of features, such as defining the target that the camera must follow, setting the field of view, adjusting the soft and dead zones and creating smooth transitions.

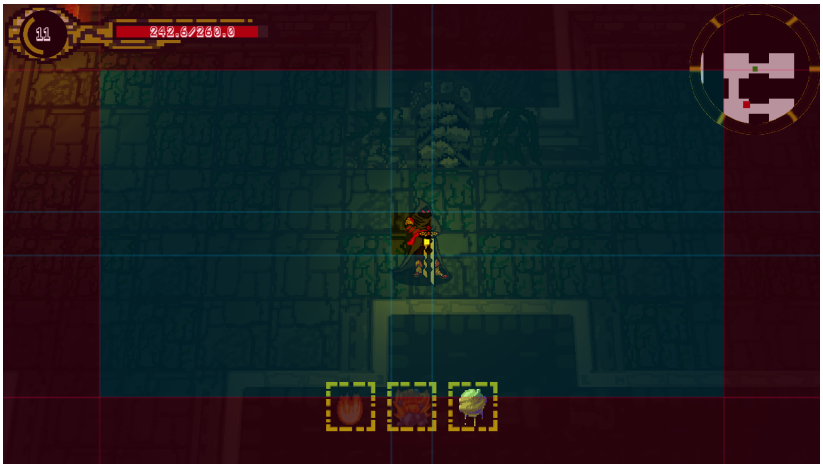


Figure 28: Setting up the camera with the Cinemachine

4.6.3 Minimap

The minimap is done with the help of an additional camera attached directly to the player. This camera renders to the texture, that is later used as a simple image in the UI. Different game objects, such as enemies and chests, have an image attached to them visible only to that camera. Those images are actually what can be seen in the minimap.

4.6.4 AI

NPCs are driven by the state machine with 5 states:

- Idle state – NPC stays for a random amount of time doing nothing.

[21] More about Unity's Cinemachine

- Inspect state – NPC chooses a random point and moves towards it for a random amount of time.
- Evade state – NPC goes back to its initial position.
- Follow state – NPC has a target and will pursue it until the target escapes the range or NPC gets close enough to attack.
- Attack state – NPC will attack the target if the attack cooldown expired and the target is within the attack range.

The state machine is also implemented in ECS, where different states are driven by systems, for example, if the entity has a *FollowStateComponent*, the *FollowStateSystem* will be executed and not others.

4.6.5 Pathfinding

NPC has the ability to navigate the dungeons with the Unity's pathfinding tools called *NavMeshComponents*[22]. When the dungeon is generated, new navigation mesh is baked at runtime and used later by agents to calculate the path to the destination. Although Unity's *NavMesh* was primarily designed to be used in a 3D environment, a rotated 3D plane can be used to match 2D's up vector.

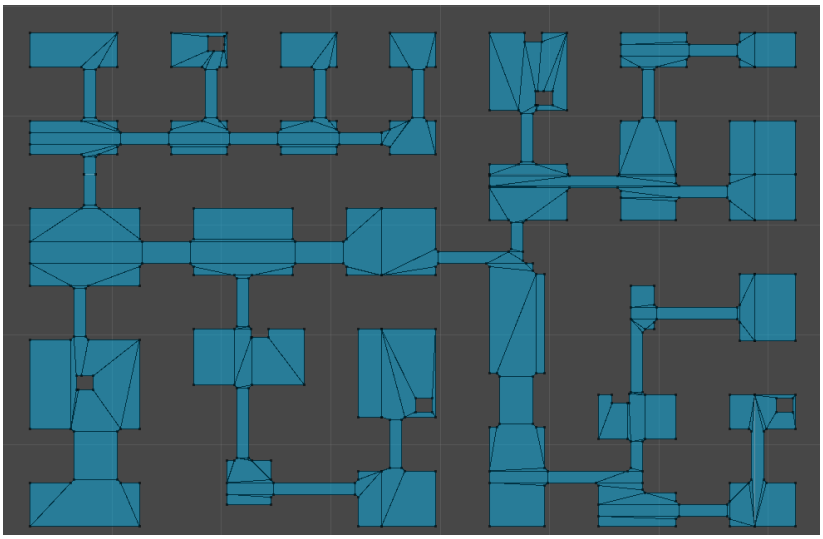


Figure 29: Generated navigation mesh for the dungeon

4.6.6 Dungeon transitions

When the player enters the dungeon a simple cut-scene starts. In the beginning it reveals the whole map, letting the player take a look at the overall map layout and where he should go. When the player leaves the dungeon a similar cut-scene

[22] NavMeshComponents Github repository

will be triggered. Cut-scenes were created with the Unity's *Timeline*[23] tool that allows combining already existing animations as well as creating new ones.

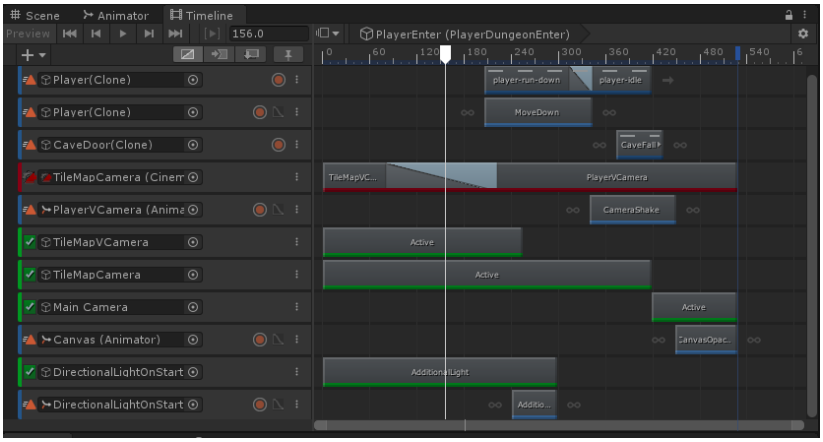


Figure 30: Dungeon cut-scene created with Timeline tool

4.6.7 Saving and loading the game

A saving mechanic must be present in the game, so the player is able to return to the game later. While using ECS most of the data is stored in components that are structs with blittable types, thus the data is already prepared for the serialization. In this game player's level, stats, items, objectives and their progression are saved automatically when the player leaves the current dungeon. Then, when he enters a new dungeon, these data are loaded back. If the player dies in the dungeon he will lose the progress in the current dungeon. This approach adds a little bit of the roguelike's fear of losing gained items and experience. Dungeons are not saved to remove the ability to reload the same dungeon until it will be learned and mastered.

4.6.8 Tutorial level

Whenever the player starts a new game, the tutorial will be loaded. The tutorial level presents the basic game mechanics, guides through the starting area and has some storytelling mechanics implemented.

[23] More about Unity's Timeline tool

Figure 31: Screenshot from the tutorial level

4.6.9 Game settings

In the main menu, there is an options button that opens the game settings menu, where the player is able to toggle full-screen mode, change the resolution and remap key bindings. Settings are saved locally in the XML format when the settings menu is closed, and are loaded back when the player returns to the game.

4.7 User interface

Unity offers a lot of UI components and tools that make the process of creating the interface easy and straightforward.

4.7.1 Main menu

When the player starts the game for the first time he is welcomed with the main menu that will have 3 buttons: New Game, Options and Quit. While opened from the game there will be an additional Resume button that simply closes the menu. If there is an existing saved game, the Load Last Game button will appear that allows the player to load the last saved game. New Game button will start a new game, and if there is an existing save, the confirmation dialogue will pop up, asking the player if he wants to delete the current save and start anew. Quit button will close the game. Options button will bring the game settings menu, where the player has the ability to toggle fullscreen mode via checkbox, change the resolution by picking the value from the dropdown, and change keybindings by clicking on the desired keybind button and pressing the key he wants to assign.

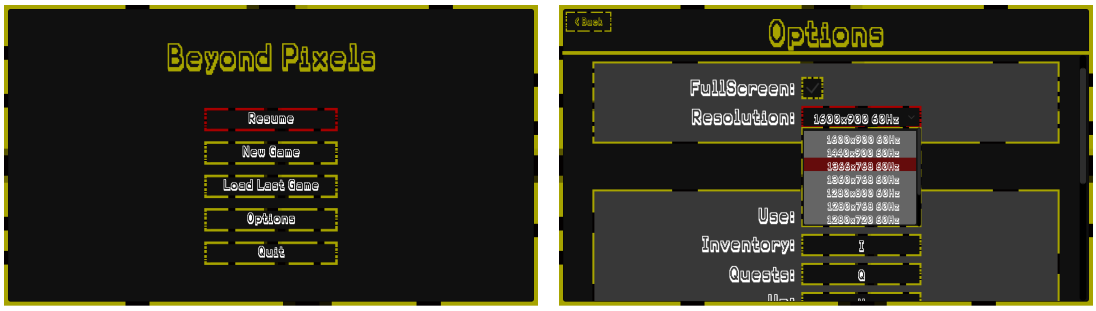


Figure 32: Main menu and settings menu

4.7.2 In-game interface

In the game, the player has an overview of the character's health, current level, level progress and if there are any completed objectives in the top left corner. The minimap is displayed in the top right corner. At the bottom, there are clickable spell buttons and a casting progress bar. When hovering with the mouse over a spell button, the tooltip with information about the spell will appear.

Enemies have their health displayed directly upon them, and the currently selected enemy will have a red circle underneath him.

The player can open the character's info menu where the inventory is located as well. Here the information about stats is visible. If the player has skill points available, he can spend them by clicking a button next to the stat he wants to invest points into. On the right side of the window, there are slots for equipped gear and in the bottom part, there is a player's inventory. Items in the inventory are sorted, and items like food and potions are stacked. The inventory does not

have a capacity, and when items will not fit on the initial grid the scrollbar will appear on the right side.

Opening a loot bag will bring a small window with a grid containing dropped items that are sorted in the same way as in the player's inventory. Hovering over the item will bring the tooltip with item's description, its stat modifiers and a comparison with the currently equipped item.

The objective window will show information about active tasks and their progress. Whenever the task is completed Collect button will appear that allows the player to receive the experience for completing the objective.

(a) In-game interface elements

(b) Inventory and character window

(c) Objectives window

Figure 33: In-game interface

Závěr

Tato práce popisuje možnosti procedurálního generování obsahu v počítačových hrách a možná rizika, která by měla být zohledněna při implementaci procedurálního generování. Na příkladech populárních her býli ukázány oblastí ve kterých se používá procedurálně generovaný obsah.

Známé algoritmy a techniky byly implementovány a předvedeny na příkladu jednoduché 2D roguelike hry. Vhodnost použitých algoritmů byla okomentována, různé metody byly porovnány. Součástí hry jsou procedurálně generováne dungeony a předměty, rozmístění nepřátelů a předmětů, systém generování úloh, možnost uložení a načtení postupů hráče. Hra je postavená s využitím herního enginu Unity v novém Entity Component Systemu.

Ještě zbývá hodně prostorů pro další vylepšení, jako je přidání zvuků, vytváření nových spritů, přidání objektů, hlavolamů a pastí do dungeonu, aby nevypadali prázdní, přidání ručně vytvořených úrovni, které by měli elementy vyprávění. A samozřejmě, ještě zbývá mnoho technik a algoritmů, které se dá použít pro procedurální generování obsahů.

Conclusions

This thesis describes the possibilities of the procedural content generation in computer games as well as the possible risks that should be considered when implementing PCG. On the examples of popular games were shown areas where procedural content generation may be used.

Well known algorithms and techniques for PCG were implemented and showcased on the example of a simple 2D roguelike game, such as an agent based dungeon growing, cellular automaton, BSP-tree, Poisson disk sampling and grammars. Details of those algorithms were described and compared. The game features dungeons and items generation, enemies and objects placement, leveling and stats progression system, a simple system for quests generation, ability to save and load the current progress of the player and a tutorial level. The game is built with the use of Unity game engine in Unity's new Entity Component System.

There is plenty of space left for further improvements, such as adding an audio, creating more diverse tile sheets, adding objects, puzzles and traps to the dungeons so they do not feel empty, blending premade levels between dungeons with storytelling elements. And, of course, there are a lot of algorithms and techniques that can be used in PCG.

A Contents of the enclosed CD

The enclosed CD contains:

`bin/`
: Contains the standalone game build. The game can be started by executing *BeyondPixels.exe*.

`doc/`
: Contains the text of this thesis in PDF format as well as the attachments and files needed to generate the text.

`src/`
: Contains the complete project and source codes of the game.

Acronyms

ASCII American Standard Code for Information Interchange is a character encoding standard for electronic communication

BSP Binary Space Partitioning

CA Cellular Automaton

DOTS Data-Oriented Technology Stack

ECS Entity Component System

IL2CPP Intermediate Language To C++

PCG Procedural Content Generation

PDS Poisson disk sampling

References

[1] ESPORTS, LPE. *The Video Games' Industry is Bigger Than Hollywood* [online]. 2018 [visited on 2019-7-3]. Available from WWW: ⟨https://lpesports.com/e-sports-news/the-video-games-industry-is-bigger-than-hollywood⟩.

[2] WIKIPEDIA. *Video game* [online]. [visited on 2019-7-3]. Available from WWW: ⟨https://en.wikipedia.org/wiki/Video_game⟩.

[3] WIKIPEDIA. *PC game* [online]. [visited on 2019-7-3]. Available from WWW: ⟨https://en.wikipedia.org/wiki/PC_game⟩.

[4] BETHKE, Erik. *Game development and production*. Texas: Wordware Publishing, Inc., 2003. 437 pp. ISBN 1-55622-951-8.

[5] BATES, Bob. *Game Design*. Second Edition. Boston, MA: Thomson Course Technology, 2004. 377 pp. ISBN 1-59200-493-8.

[6] MICHAEL E. MOORE, Jeannie Novak. *Game Development Essentials: Game industry career guide*. 2010. 320 pp. ISBN 978-1-4283-7647-2.

[7] GREEN, Dale. *Procedural Content Generation for C++ Game Development*. Birmingham, UK: Packt Publishing Ltd, 2016. 304 pp. ISBN 978-1-78588-671-3.

[8] CORMEN, T. H.; LEISERSON, C. E.; RIVEST, D. L.; STEIN, C. *Introduction to algorithms*. Second Edition. 2001. ISBN 0-262-03293-7.

[9] WATKINS, Ryan. *Procedural Content Generation for Unity Game Development*. Birmingham, UK: Packt Publishing Ltd, 2016. 260 pp. ISBN 978-1-78528-747-3.

[10] NOOR SHAKER Julian Togelius, Mark J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. 2016. 237 pp. ISBN 978-3-319-42714-0.

[11] JULIAN TOGELIUS Emil Kastbjerg, David Schedl; YANNAKAKIS, Georgios N. What is Procedural Content Generation? Mario on the borderline. [online]. 2011, [visited on 2019-7-5]. Available from WWW: ⟨https://course.ccs.neu.edu/cs5150f13/readings/togelius_what.pdf⟩.

[12] KNUTH, D. *The Art of Computer Programming: Fundamental Algorithms*. Third Edition. 2004. ISBN 0-201-89683-4.

[13] TANYA X. SHORT(EDITOR), Tarn Adams(Editor). *Procedural Generation in Game Design*. First Edition. 2017. 338 pp. ISBN 978-1-498-79919-5.

[14] WEN, Howard. *Interview: Frugal Fragging with .kkrieger* [online]. 2005 [visited on 2019-7-3]. Available from WWW: ⟨https://www.gamasutra.com/view/feature/130690/interview_frugal_fragging_with_.php⟩.

[15] WIKIPEDIA. *Elite screenshot* [online]. [visited on 2019-7-3]. Available from WWW: ⟨https://en.wikipedia.org/wiki/Elite_(video_game)#/media/File:BBC_Micro_Elite_screenshot.png⟩.

[16] WIKIPEDIA. *.kkrieger screenshot* [online]. [visited on 2019-7-3]. Available from WWW: ⟨https://en.wikipedia.org/wiki/.kkrieger#/media/File:Kkrieger_screenshot.jpg⟩.

[17] WIKI, Borderlands. *Borderlands 2 Weapons* [online]. [visited on 2019-7-3]. Available from WWW: ⟨https://borderlands.fandom.com/wiki/Borderlands_2_Weapons⟩.

[18] ALEXANDRA, Heather. *A Look At How No Man's Sky's Procedural Generation Works* [online]. [visited on 2016-10-18]. Available from WWW: ⟨https://kotaku.com/a-look-at-how-no-mans-skys-procedural-generation-works-1787928446⟩.

[19] WIKI, Borderlands. *Borderlands Weapons image* [online]. [visited on 2019-7-3]. Available from WWW: ⟨https://borderlands.fandom.com/wiki/Weapons?file=Weapon_Components.png⟩.

[20] WIKIPEDIA. *No Man's Sky screenshot* [online]. [visited on 2016-10-18]. Available from WWW: ⟨https://en.wikipedia.org/wiki/No_Man%27s_Sky#/media/File:No_mans_sky_screenshot.jpg⟩.

[21] MUSINGS OF A MARIO MINION. *How Does Ragdoll Physics Work?* [online]. [visited on 2016-9-15]. Available from WWW: ⟨https://musingsofamariominion.com/2016/09/15/how-does-ragdoll-physics-work/⟩.

[22] HECKER, Chris; RAABE, Bernd; ENSLOW, Ryan W., et al. Real-time Motion Retargeting to Highly Varied User-Created Morphologies. 2008. Available also from WWW: ⟨http://chrishecker.com/images/c/cb/Sporeanim-siggraph08.pdf⟩.

[23] HETHERINGTON, Janet. *The Evolution of 'Spore'* [online]. [visited on 2016-9-15]. Available from WWW: ⟨https://musingsofamariominion.com/2016/09/15/how-does-ragdoll-physics-work/⟩.

[24] CUTAJAR, Simon. An Introduction to Procedural Musicin Video Games by Karen Collins Summary. Available also from WWW: ⟨http://chrishecker.com/images/c/cb/Sporeanim-siggraph08.pdf⟩.

[25] WIKIDOT. *Rogue* [online]. [visited on 2019-7-3]. Available from WWW: ⟨http://pcg.wikidot.com/pcg-games:rogue⟩.

[26] FINGAS, Jon. *Here's how 'Minecraft' creates its gigantic worlds* [online]. 2015 [visited on 2019-7-3]. Available from WWW: ⟨https://www.engadget.com/2015/03/04/how-minecraft-worlds-are-made/⟩.

[27] DIABLOWIKI. *Diablo: Randomization* [online]. [visited on 2019-7-3]. Available from WWW: ⟨https://www.diablowiki.net/Randomization#Map_Generation_and_Size⟩.

[28] BESCHIZZA, Rob. *Procedurally-generated British countryside* [online]. 2012 [visited on 2019-7-3]. Available from WWW: ⟨https://boingboing.net/2012/07/04/procedurally-generated-british.html⟩.

[29] ROGUEARCHIVE. *Rogue screenshot* [online]. [visited on 2016-9-15]. Available from WWW: ⟨https://britzl.github.io/roguearchive/⟩.

[30] WIKIPEDIA. *Pixel art* [online]. [visited on 2019-7-3]. Available from WWW: ⟨https://en.wikipedia.org/wiki/Pixel_art⟩.

[31] UNITY TECHNOLOGIES. *Unity* [online]. [visited on 2019-7-14]. Available from WWW: ⟨https://unity.com/⟩.

[32] UNITY TECHNOLOGIES. *The world's leading real-time creation platform* [online]. [visited on 2019-7-14]. Available from WWW: ⟨https://unity3d.com/unity?_ga=2.185105069.587496795.1563040448-1628506034.1548607113⟩.

[33] UNITY TECHNOLOGIES. *Performance by default* [online]. [visited on 2019-7-14]. Available from WWW: ⟨https://unity.com/dots⟩.

[34] UNITY TECHNOLOGIES. *Official ECS Github repository* [online]. [visited on 2019-7-14]. Available from WWW: ⟨https://github.com/Unity-Technologies/EntityComponentSystemSamples⟩.

[35] WIKIPEDIA. *Entity Component System* [online]. [visited on 2019-7-14]. Available from WWW: ⟨https://en.wikipedia.org/wiki/Entity_component_system⟩.

[36] MEIJER, Lucas. *On DOTS: Entity Component System* [online]. 2019 [visited on 2019-7-14]. Available from WWW: ⟨https://blogs.unity3d.com/2019/03/08/on-dots-entity-component-system/⟩.

[37] UNITY TECHNOLOGIES. *General Purpose Components* [online]. [visited on 2019-7-15]. Available from WWW: ⟨https://docs.unity3d.com/Packages/com.unity.entities@0.0/manual/component_data.html⟩.

[38] UNITY TECHNOLOGIES. *What is a job system?* [online]. [visited on 2019-7-15]. Available from WWW: ⟨https://docs.unity3d.com/Manual/JobSystemJobSystems.html⟩.

[39] UNITY TECHNOLOGIES. *The safety system in the C# Job System* [online]. [visited on 2019-7-15]. Available from WWW: ⟨https://docs.unity3d.com/Manual/JobSystemSafetySystem.html⟩.

[40] UNITY TECHNOLOGIES. *NativeContainer* [online]. [visited on 2019-7-15]. Available from WWW: ⟨https://docs.unity3d.com/Manual/JobSystemNativeContainer.html⟩.

[41] UNITY TECHNOLOGIES. *Burst User Guide* [online]. [visited on 2019-7-15]. Available from WWW: ⟨https://docs.unity3d.com/Packages/com.unity.burst@0.2/manual/index.html⟩.

[42] UNITY TECHNOLOGIES. *Intro To The Entity Component System And C# Job System* [online]. [visited on 2019-7-15]. Available from WWW: ⟨https://www.youtube.com/playlist?list=PLX2vGYjWbI0S4yHZwjDI1boIrYStpBCdN⟩.

[43] WIKIPEDIA. *Cellular automaton* [online]. [visited on 2019-7-20]. Available from WWW: ⟨https://en.wikipedia.org/wiki/Cellular_automaton⟩.

[44] WIKIPEDIA. *Von Neumann neighborhood image* [online]. [visited on 2019-7-20]. Available from WWW: ⟨https://en.wikipedia.org/wiki/Von_Neumann_neighborhood#/media/File:Von_Neumann_neighborhood.svg⟩.

[45] WIKIPEDIA. *Moore neighborhood image* [online]. [visited on 2019-7-20]. Available from WWW: ⟨https://en.wikipedia.org/wiki/Moore_neighborhood#/media/File:Moore_neighborhood_with_cardinal_directions.svg⟩.

[46] WIKIPEDIA. *Bresenham's line algorithm image* [online]. [visited on 2019-7-20]. Available from WWW: ⟨https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm#/media/File:Bresenham.svg⟩.

[47] BRIDSON, Robert. Fast Poisson Disk Sampling in Arbitrary Dimensions. Available also from WWW: ⟨https://www.cs.ubc.ca/~rbridson/docs/bridson-siggraph07-poissondisk.pdf⟩.