How to build a video game using Procedural Content Generation?

Author: Claudiu Mihai Jechel
Student ID: 10479594

Project Supervisor: Dr. Ke Chen

A report submitted to the University of Manchester for the degree of bachelor of science in the faculty of science and engineering.

School of Computer Science
2023

Contents

List of Tables

List of Figures

Abstract

Procedurally content generation techniques for video game development were created to aid the developer in building a game. This report aims to provide background information and theory concerning the process of building a video game from scratch without designing maps, thus focusing on the usage of generation algorithms. An overview of maze generation algorithms, game engines, assets generation, work environment and game logic design will be given in the following chapters while developing a 2D game. In the end, the results will be shown and evaluated, taking into perspective the performance of the chosen technique, a comparison between similar algorithms, the user interface analysis using User Experience rules, and further improvements.

Abbreviations

PCG	Procedural Content Generation
FPS	Frames Per Second
GUI	Graphical User Interface
OOP	Object-Oriented Programming
PPM	Pixels Per Meters ratio
JDK	Java Development Kit
IDE	Integrated Development Environment
UI	User Interface
DFS	Depth-First Search
QA	Qualitative Assessment
UX	User Experience

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I thank my supervisor, Dr Ke Chen, for supporting, checking in on me and for his advice this year. I would also like to thank my tutor, Dr Christoforos Moutafis, for his support during my first two studies and the placement year.

I send my parents and grandparents lots of love and hugs for raising and loving me. Last but not least, I would like to thank all my friends who believed in me and stayed along during good and challenging times.

1 Introduction

Video games are one of software development's central and oldest areas. Some of the first titles appeared in the late '40s; they were either too expensive or too big to be sold to the public(History.COM Editors, 2017). Most of them were the results of military and governmental laboratory research. For example, in 1952, British professor A.S. Douglas created OXO (OXO (video game), 2021), as shown in Figure 1, also known as noughts and crosses or tic-tac-toe, at the University of Cambridge.

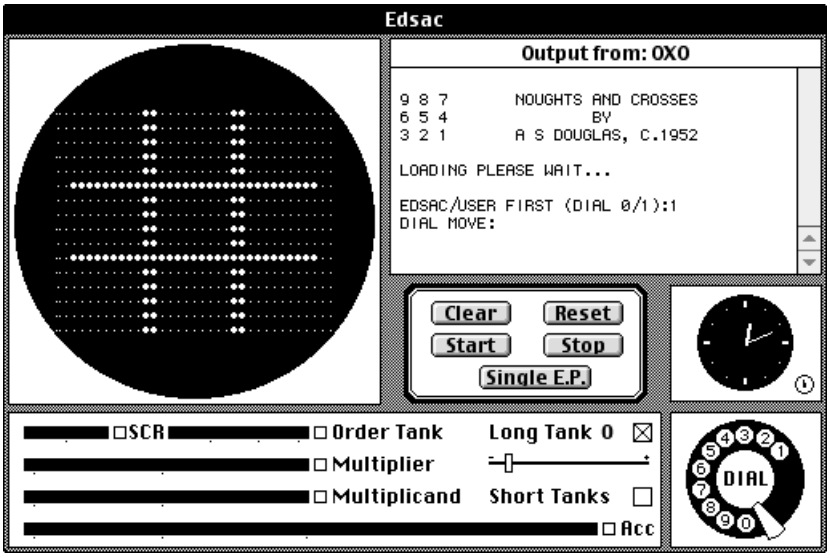

Figure 1: A.S. Douglas's OXO (OXO (video game), 2021)

In 1958, William Higinbotham created Tennis for Two (De, Cruz and Ryan, 2015), as shown in Figure 2. The user had to use an oscilloscope, vacuum tubes and transistors to play the game. It used integrating circuitry to find the path of a ball as it bounces back and forth between two sides of a court, represented by a pair of controllers. The whole process is based on military computers designed to calculate the ballistic path of missiles.

Figure 2: Tennis for Two (Wikipedia Contributors, 2019)

In 1962, Steve Russell at the Massachusetts Institute of Technology invented Spacewar!, the first video game with multiple computer installations, as shown in Figure 3.

Figure 3: Spacewar! (Mary Bellis, 2019)

Earlier creations can be classified as games, such as the Cathode-ray tube amusement device (Cathode-ray tube amusement device, 2022) and the Nimrod computer (Nimrod (computer), 2022), designed to play the game of Nim (Nim, 2020), as shown in Figure 4. However, they were built to show off computing power or for academic purposes.

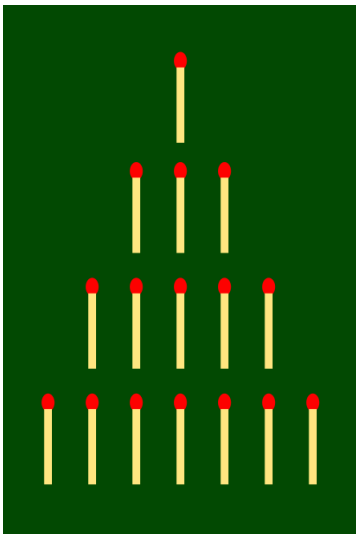

Figure 4: Nim (Nim, 2020)

This changed in 1972, when Ralph Baer, the father of video games, released the Odyssey console. Soon after, various companies started to emerge, creating some of the most-known titles in the world. The Atari company released Computer Space, the first commercial arcade game. They went further, and on November 29, 1972, they

released Pong. By the end of that year, they had sold more than 8000 Pong arcade machines. One exemplar can be seen in Figure 5.

Figure 5: Pong Arcade (Wikipedia Contributors, 2019a)

Between the 1970s and 1980s, the world got to know Space Invaders, Asteroids, Pac-Man, Donkey-Kong, Defender, Galaga, Mario Bros and many others, establishing the industry of video games. With technology improving over time, the first 3D games started to appear, giving birth to modern-day franchises, such as Metal Gear, Resident Evil, Halo, Grand Theft Auto (The original poster is shown in Figure 6) and God of War.

Figure 6: Grand Theft Auto V (Wikipedia Contributors, 2019a)

To sum it up, as the technology behind the game development improved, the video games industry expanded, overwhelming our society. Although it is fun to play these games, one could venture and wonder how they are made.

1.1 History of game development

Moving on to a short history lesson about game development, there was no high-level source code at the beginning of the video games era. Take Pong for example. It was built using hardware circuitry. The original schematics are shown in Figure 7.

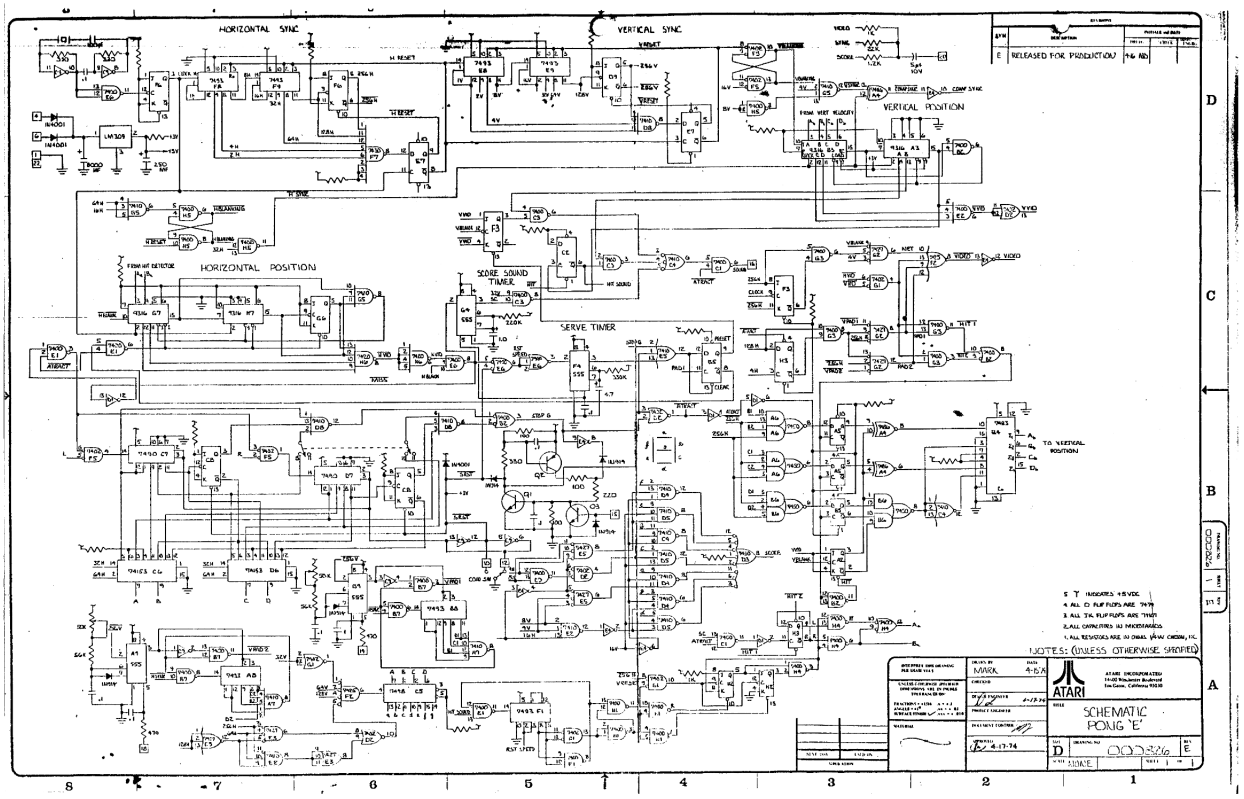

Figure 7: Pong Schematics (Museum, no date)

The complexity behind such a surprisingly simple game is astonishing. Thus, the evolution and simplification of the technological tools was imperative, in order to increase the games quality and difficulty. For example, a game such as Assassin's Creed Valhalla would be unreasonablyhard to create without the development of new, more intuitive toolswhich used accessible coding languages. Thus, the industry moved on to using Assembly. However, this is still considered complex for today's standards. Out of curiosity, one could browse the source code of the Prince of Persia game here.

Thankfully, things have evolved, and in 1985 the game developers got the closest thing to a game engine: Garry Kitchen's GameMaker (Balasubramanian, 2022). Everyone has started using them since then, and companies are creating engines to serve their specific needs. However, they all share the same conventions regarding how a game works. Nowadays, a developer has a variety of options when choosing an engine: Unreal Engine, Unity, libGDX, Phaser and many others. Another honourable mention is RAGE (Rockstar Advanced Game Engine (no date)), developed and used by RAGE Technology Group, a Rockstar Games' Rockstar San Diego studio division. Unfortunately, it is not available to the public.

While these, together with other tools built for image processing and animation

drawing, significantly decrease the amount of effort needed on the developer's side,the time required for designing and releasing a complete game, is still too much. Fortunately, there are techniques that can further help the developer.

1.2 Procedural Content Generation

Procedural Content Generation (PCG) in Video Game Development 'is the algorithmic creation of game content with limited or indirect user input' (Noor Shaker et al., 2018). We trade the work done by the developers and graphics designers with an algorithm capable of generating content. Content in a video game can be anything: characters, maps, obstacles, various objects and events, music, and images. One of the best examples of games that use PCG is Minecraft (Figure 8).

Figure 8: Minecraft (Wikipedia Contributors, 2019b)

The levels are procedurally generated using scaled 3D Perlin noise with linear interpolation. Ken Perlin invented the algorithm, and it can be used to generate textures and terrain. Biomes are built from a Whittaker diagram, which divides up a procedurally generated landscape based on different values. For example, in the diagram area that represents the jungle biome, different values for temperature are displayed at different heights so that some animals and objects spawn if the conditions around are met. Likewise, based on the value of the biome, the player might be caught in a rain or snowstorm. Indeed, Minecraft is very good at showing off PCG techniques, but so are other games: Microsoft Minesweeper, Valheim, and Terraria.

1.3 Traditional vs. Modern PCG techniques

The PCG techniques fall into two categories: traditional (Grammars, L-systems, search-based approaches, Fractal methods, Perlin noise functions, cellular automata) and modern (based on machine learning). This paper will discuss the process of game

development using a traditional PCG technique for maze generation. The aim is to provide the required background and theory to the reader about:

- The maze generation algorithm
- How a game engine works
- The process of building game assets
- Setting up the working environment
- Designing the game logic

At the end of this report, the reader should be able to understand and reproduce the game Spatium Exploratio, a 2D top-down adventure game. The game takes the player on planet Pluto, where he has explore procedurally generated mazes, to find a broken satellite.

Ultimately, the results will be evaluated in terms of the algorithm performance. A comparison between similar algorithms will be done. The reader will also get an User Experience (UX) analysis of the interface. Further improvements for the will be proposed.

2 Background

As already mentioned, the scope of the game is for the player to find the exit of a maze. Figure 9 shows an example of one of the mazes generated.

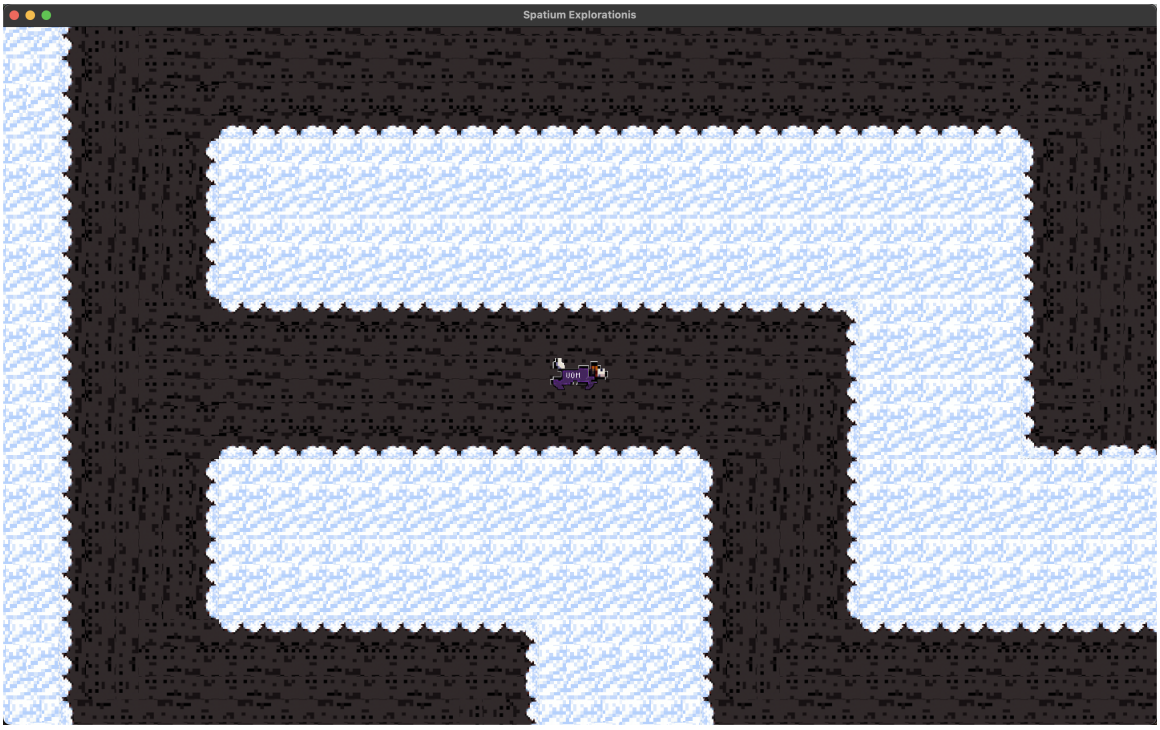

Figure 9: Spatium Exploratio

2.1 The Depth-first search algorithm and its randomized implementation

First of all, it is important to define a few concepts. A graph is a pair G = (V, E), where V is a finite set and E is a set of subsets of V of cardinality exactly 2. The elements of V are called vertices, and the elements of E edges. The standard notation of an edge u, v is (u, v). We say that u and v are neighbours if $(u, v) \in E$. We say v and e are adjacent, if $v \in e \in E$.

Begin by defining the Vertex class as per Table 1:

Vertex
boolean visited
integer i
integer j
array<Vertex> neighbours

Table 1: The Vertex class

At initialization, the visited variable is set to false. If the vertex is visited, it is changed to true. The variables i and j are used to store the vertex's position. In the

end, an array of vertices is used to store and keep track of the vertex's neighbours.

With the Vertex class done, the next step is the Maze class which defines the graph. Its definition is shown below in Table 2.

Maze
integer rows array<array<Vertex» vertices

Table 2: The Maze class

Any object made using the blueprint of the Maze class will store the graph in the form of an array of arrays of vertices (also known as a matrix of vertices) of size rows x rows. The value rows is stored in the rows variable.

The randomized depth-first search algorithm is shown below (Krimgen, 2020):

```
function createMaze():
    startVertex <- Vertex(0,0)
    randomizedDFS(startVertex)
end

function randomizedDFS(vertex):
    markVisited(vertex)
    nextVertex <- randomUnvisitedNeigbour(vertex)
    while nextVertex != null do:
        connectCells(vertex, nextVertex)
        randomizedDFS(nextVertex)
        nextVertex <- randomUnvisitedNeigbour(vertex)
    end
    return
end
```

This is an easy solution for generating a maze with an underlying graph structure which has to be integrated into the Maze class. Therefore, the results are the updated blueprints for the Vertex and Maze classes, as per Table 3 and Table 4.

Vertex
boolean visited integer i integer j array<Vertex>neighbours
Vertex(int i, int j) void visit() boolean isVisited() void setNeighbour(Vertex v)

Table 3: The Updated Vertex class

Maze
integer rows array<array<Vertex>>vertices
Maze(integer rows) void createMaze() void randomizedDFS(Vertex v) Vertex randomUnvisitedNeighbour(Vertex v) void connectCells(Vertex v, Vertex nextVertex)

Table 4: The Updated Maze class

The updated versions now also have constructors and the other required methods.

2.2 The mechanics of a game engine

A game engine is a black box that takes care of several items (Glaiel, 2021) for the developer:

- **System Initialization**: opening a window, obtaining an OpenGL context and initializing audio.
- **Frame Timing Control**: Setting the number of frames per second (FPS) to be rendered (how many times the textures are redrawn). The most common values for FPS are 30, 60 and 120. The bigger the value, the more fluid the animation will look. However, the memory used increases as well.
- **Input**: Detecting the user input (Harper, no date) varies on the system that is running the game. Generally, this is done by combining a Graphical User Interface (GUI) with a keyboard and a mouse. However, there are alternatives. Instead of a mouse, the user's system might have a trackpad, a trackball, or a joystick. Input can be obtained through haptic interaction, where the user holds the end-effector, which can output vibrations of different magnitudes when interacting with a game object. Some games use speech input, such as Lurking or Scream Go Hero. If the developer is building a horror movie, he/she might want to integrate a feature where the microphone detects sounds from the user such that the villain spawns in the vicinity of the player. The input will come through touch interfaces if the game supports the Nintendo Switch console or mobile phones. Once seen as an area for research, Gesture Recognition has made its way into the video game industry, although it is less popular. A good game engine will support most of these types of input.
- **Rendering**: For a 2D game, the bare minimum renderer will draw textures on the screen. Shaders, vertex buffers, render targets, meshes, and materials are necessary if the game is expected to have higher performing graphics. For complete customization, a framework such as OpenGL or Vulkan will suffice, as plenty of documentation exists. However, there are alternatives that facilitate the process and speed up the technical learning curve. For example, using the Ogre3D library would potentially speed up the coding process and reduce the time needed for implementation.

- **Math and Physics Utilities**: Integrating a math engine helps build a physics engine and detect collisions. A physics engine helps simulate rigid bodies, articulated rigid bodies - more rigid bodies connected together, like the bones of a skeleton, deformable objects - such as a building collapsing due to an earthquake. Another vital area where having a physics engine will benefit the developer is liquids simulation. A good engine will have different methods for this, such as Particle-Based, Grid-Based or Hybrid Fluid Simulation.
- **Support for Object-Oriented Programming (OOP)**: Using OOP is the default methodology for designing and building video games. Every component of a game can be conceptually interpreted as an object (instance of a class) built after a blueprint (the class).
- **Scene Management**: Video games often have different screens: the main menu, the settings page, and the gameplay screen. The option of separating these will keep a clean code base.
- **Audio**: Usually, the basic customisation revolves around turning the sound effects on and off. Most of the time, the developer will need to create and edit the sound files using external tools.
- **File Loading and Management**: A basic manager that allows uploading and structuring external files, such as the images used for the textures, animations made outside the engine and audio files.
- **Networking**: A system that allows connection to a server that manages data exchange. While is it a fascinating topic, it requires a deeper understanding and an extensive discussion concerning the software running on the server, the databases and the security protocols, all which are outside of this report's aims.
- **Serialization**: This is necessary when the game requires a way to save data locally for further use in the future. It can be obtained quickly, depending on the used frameworks and programming languages.

These are some essential features a game engine must have, although there can be more. Now that we know what we want from a game engine unless we might want to build one from scratch, it is time to choose one. Choosing depends on the developer's needs and preferences. For our 2D game, we will work with libGDX, an open-source Java framework for game development. Why this particular engine? While other modern game engines have a GUI, which means the learning curve will be pretty steep, libGDX is based on writing lines of code, a lot of them!

2.3 LibGDX

The libGDX game engine supports all of the above features along with further ones specialised for 3D development. Moreover, it is also cross-platform, meaning it will translate the code written to be compatible with multiple platforms: Desktop (Windows and macOS), IOS, Android and Webpage — no need for rewriting the game for a different device. For the game presented in this report, there is a particular interest in System Initialization, Frame Timing Control, Input, Rendering, Math and Physics Utilities, OOP Support, Scene Management, Audio, File Loading and Management.

These topics will be encountered when discussing the development process.

Firstly, it is a good idea to get used to the lifecycle of a screen in the game engine. When building a new screen object, for example, the Gameplay screen, it must implement the Screen interface given by the engine. This interface looks as per Table 5:

Screen
void show()
void render(float delta)
void resize(integer width, integer height)
void hide()
void pause()
void resume()
void dispose()

Table 5: The Screen interface

The libGDX engine calls all these methods which contain the game logic. These are the methods and their functionality:

- The **show()** method: This is called only once when the screen the method belongs to becomes the current screen. This method is the ideal place to load all the assets and initialize the screens' objects. Some of these things can be done in the constructor of the custom screen class, but not all of them, as some of the engine's components are loaded between the calls of the constructor and the show methods.
- The **render(float delta)** method: This is called each time the frame renders. If the FPS is set to 60, it will be called 60 times per second. Here is where the objects are being drawn, and the game's logic is updated. The delta parameter represents the time that has passed since the last call of the render method.
- The **resize(integer width, integer height)** method: libGDX closely monitors the width and height of the game's window. If the user resizes it, the engine will call this method. Inside this method is the place for resizing our textures or loading new ones if we have created different images for different resolutions.
- The **hide()** method: It is called when the screen is no longer the current screen; It helps save memory since we can stop various processes: drawing textures, detecting collisions, updating positions and calculating the effects of the interactions between objects.
- The **pause()** method: It is called when the game's window is no longer the main one on the computer/device where the game is being run. Also called by default before calling the dispose() method;
- The **resume()** method: It is called when the game's window becomes again the main one.
- The **dispose()** method: It is called when the screen is destroyed. Here we must dispose of all the assets we have loaded for our screen: images, videos and sounds; otherwise, they will occupy memory until the whole game is closed.

More information about the Screen interface can be found here and here. The complete documentation is available on GitHub.

2.4 Box2d - a physics library

Given the maze as the main component of the game, it is understandable the player must guide the main character between walls in order to escape. Thus, collision detection is another very important aspect in the game's development. If the player moves the character in the direction of the wall, the expected response is to restrict movement in that direction and force the player to change positions, rather than move within walls. This can be achieved as follows:

1. Get the coordinates of the edges of the character.
2. Get the coordinates of the corners of each wall.
3. Check if the coordinates of the character overlap with any coordinates of the walls.
4. If they do, stop the character's movement towards the direction of the overlapping wall.

An implementation from scratch might be considered an overkill, especially if the character has a more complex design with multiple edges and unique shape. One such example is Figure 10.

Figure 10: A dog from the University of Manchester

In the case of an object like this (i.e. a dog), refining the shape with multiple precise edges could become problematic and too complex to be considered advantageous. For example, animating the character would require multiple frames and extensive customisation of the collision detector (Figure 11). Moreover, if the dog object would be replaced with a significantly different shaped object, restructuring the collision detection would require plenty of effort.

Furthermore, another problem with this approach is that the collision is detected once the objects have overlapped. This might result in unpleasant graphics and can be avoided by adding a few extra pixels to each coordinate when doing the check to give the impression of a natural impact.

A more straightforward solution would be to find a regular shape in which we could fit our character, as presented in Figure 12:

Thus, by fitting the dog object in a rectangle, only four corners need to be checked for collision against other game objects. Additionally, it is much easier to insert textures and animations inside such a shape and then scale accordingly. This technique is known as using Bounding Volumes for collision detection. There are multiple types of bounding volumes, such as:

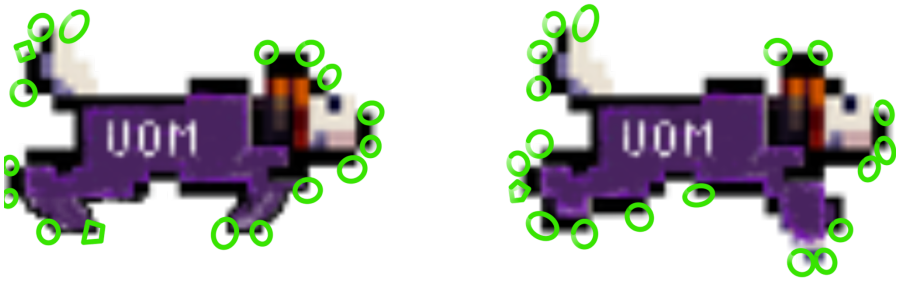

Figure 11: An animation for a walking dog from the University of Manchester

Figure 12: The character inside a square

- **AABB** (Figure 12)
- **OBB** (Figure 13)

Figure 13: The dog inside an OBB Bounding Volume

- **k-DOP** (Figure 14)

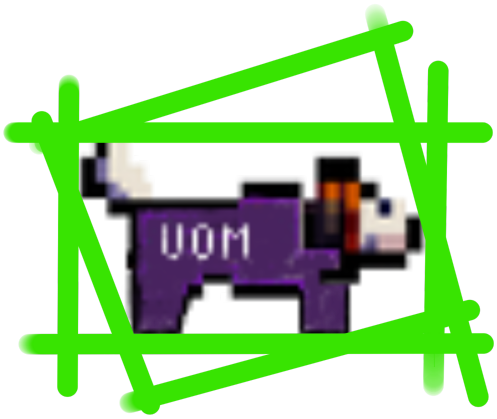

Figure 14: The dog inside an k-DOP Bounding Volume

- **Sphere** (Figure 15)

Figure 15: The dog inside a Sphere Bounding Volume

Since Box2d offers a collision detection system that uses Bounding Volumes, a Sphere will be used. This will allow for a smooth interaction with other Rectangles.

2.5 One more thing about the Box2d library ...

LibGDX uses pixels as its unit of measurement because it is a natural choice for 2D graphics. On the other hand, Box2D uses meters as its unit of measurement because it is a standard unit in physics.

When using both LibGDX and Box2D together, converting between pixels and meters is vital. This is where the pixels per meters ratio (PPM) becomes imperative. The PPM ratio is used to convert between the two units.

To convert a value in pixels to meters, divide the value by the PPM ratio. To convert a value in meters to pixels, multiply the value by the PPM ratio. The following ration will be used:

$$PPM = \frac{100pixels}{1meter} = 100$$

More details about the Box2d library can be found on their website or their specific libGDX documentation.

2.6 Animations

An animation is made of multiple frames that are shown in a sequence, with time gaps in between. An animation of a walking dog can be achieved by capturing multiple phases of its movement and playing those images in a loop, as shown in Figure 16:

Figure 16: The dog's movement captured as an animation

To get a fluid transition between frames, each one must have a duration of a quarter of a second. More details about the animation process in libGDX are found in its documentation.

2.7 Game Settings

Most of the games have a menu that allows the player to customize different aspects of the game. This is also known as the Settings menu. It is obvious that in a game where a maze is generated, there should be the option of configuring the size of it. Pre-production versions should also incorporate a debug mode.

2.8 Debug mode

During the development process, it is useful to see details of the game that do not appear in the final release version, such as the Bounding Volumes in which the image are wrapped. This is known as building and running the game in debug mode. In Figure 17, the debug mode is turned on.

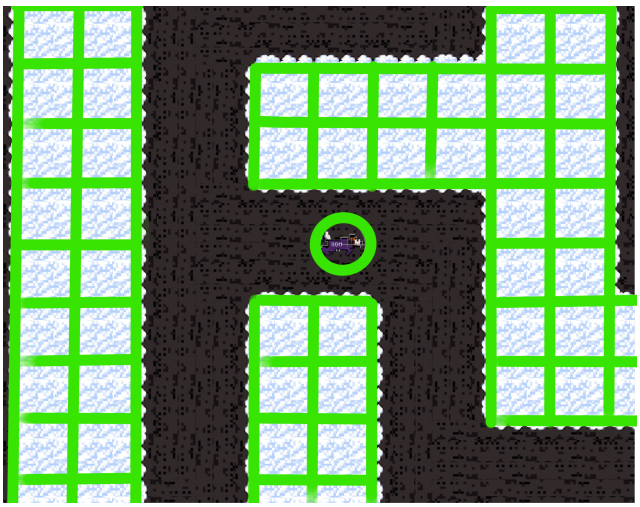

Figure 17: Debug mode is turned on

2.9 Camera

A camera represents the player's eye into the game, an analogy to the real world camera. It is used in libGDX to render the screen. There are two types:

- **PerspectiveCamera**: for 3D games
- **OrthographicCamera**: for 2D games

Building a 2D game will require an orthographic camera. It builds a orthographic (parallel) projection. Simple actions are available with this camera: movement, rotation, zooming in and out and changing the viewport. Using a camera eliminates the need to manually move objects using matrix operations. All the matrix operations are hidden in the implementation.

More details about the PerspectiveCamera can be found here.

2.10 Viewport

A viewport represents how what the camera sees is displayed on the screen. In libGDX, there are four types:

- **StretchViewport**
- **FitViewport**
- **FillViewport**
- **ScreenViewport**
- **ExtendViewport**
- **CustomViewport**

For this game, two cameras will be required, each having a custom viewport. One of the cameras has to look at the images being drawn. The other one will look at the objects bounding volumes. Combining the two cameras will result in seeing both the objects and their bodies (Bounding Volumes) overlapping, as per Figure 17.

For more details about the libGDX viewports, see the documentation.

3 The setup

Since the theory has been established, the next step is setting up the working environment.

3.1 Java

LibGDX is written in Java, therefore the installation of the Java Development Kit (JDK) is required. The latest version of the JDK can be downloaded from Oracle's website. The minimum JDK version supported by libGDX at the moment is 11. LibGDX also uses Gradle, which does not support JDK 19 yet. As a consequence, the JDK's version must be between 11 and 18.

Tutorials for installing the JDK can be found online:

- Windows
- macOS
- linux

Tutorials for using Java can be found on the w3schools's website.

3.2 Integrated Development Environment

An integrated development environment (IDE) is required. An IDE is an editor for Java files, which makes developing applications convenient. There are multiple options for a Java IDE. All of them are different, but in the end they all help achieving the same job.

- **Android Studio**: The obvious choice since it is free and provides support for developing applications for Android. The JDK is provided by Android Studio, so one less problem to take care of. Find it here.
- **Intellij IDEA**: It also requires JDK 11+. Find it here.
- **Eclipse**: Find it here.
- **Other IDE**: The developer can also opt for other IDEs, such as: NetBeans, Visual Studio Code or AIDE. However, these are not commonly used in the libGDX community and they might arise issues which do not have solutions on the internet.
- **Commandline**: LibGDX applications can be developed without the use of an IDE, writing the Java files in a simple editor like Notepad or Vim. This is not recommended, since an IDEs can provide features like code completion and error checking. However, libGDX applications can be built and executed via the commandline.

The community's preferred choice is the **Android Studio editor**.

3.3 LibGDX

The next step is to create the project. There are multiple ways, however, the preferred one is downloading the tool developed by the libGDX founders. It can be found on their website.

3.4 Creating the project

The process goes as follows:

1. Open **gdx-setup.jar**. The following window will pop up (Figure 18):

Figure 18: The libGDX project setup tool

2. Next step is to fill in the fields:
   - **Project name**: The game will be called **Spatium Exploratio**.
   - **Package name**: Usually, this is the developer's website name in reverse. For this game, go with **com.spatium.exploratio**.
   - **Game class**: This is the main class in the game, namely the **GameMain** class.
   - **Output folder**: This is the location where the project will be saved.
   - **Android SDK**: This must be filled with the path to the Android SDK. This is only required if the game will be released on the Android platform. To get the Android SDK, there are two choices:
     - Manually download the Android SDK.
     - Download Android Studio, and:
       (a) Open **Android Studio**.

(b) Go to **Settings (Windows) / Preferences (macOS)**.

(c) Go to **System Settings -> Android SDK**. The path is displayed at this location (Figure 19).

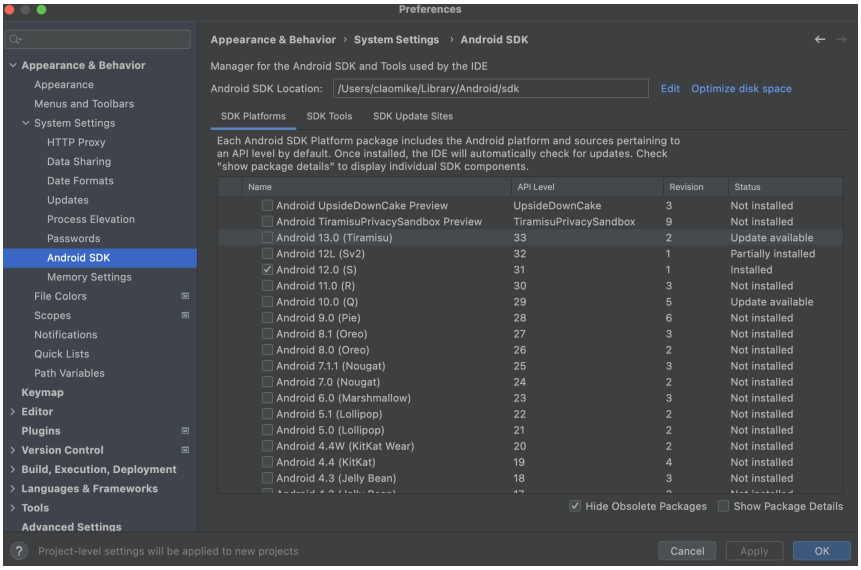

Figure 19: Android SDK path

(d) Copy and paste the path inside the libGDX tool.

- **Supported Platforms**: The **Desktop (LWJGL)** box must be checked. In case the Android SDK is installed, the **Android** box can be checked as well. To deploy the game on **IOS**, the Xcode IDE has to be installed on a machine running macOS. For deploying the game in the format of a webpage, the **HTML** box must be checked.
- **Official Extensions**: These are, as the name suggests, the official extensions supported by libGDX.
  - **Bullet**: It is a 3D Collision Detection and Rigid Body Dynamics Library. The Library is Open Source and free for commercial use. This extension is a Java wrapper for the C++ engine. More tutorials about it can be found on the libGDX website, Bullet's website or on the Bullet's forum.
  - **Freetype**: It is an extension desgined to replace the default way of drawing text. A tutorial can be found at here.
  - **Tools**: A set of tools with a particle editor for both 2D and 3D, and bitmap font and image texture packers.
  - **Controllers**: This extension provides some features, including:
    * Enumerate connected controllers;
    * Support for buttons, axes, vibration and more typically used functionality on game controllers;
    * Listen for controller events globally or per controller;
    * Poll controller state.

    A tutorial can be found at here, while the repository can be found on GitHub.

- **Box2d**: One of the most popular physics libraries for 2D games and has been ported to many languages and many different engines. The Box2D implementation in libGDX is a thin Java wrapper around the C++ engine. A tutorial can be found here. The documentaion is available here.
- **Box2dlights**: A 2D lighting framework that uses box2d for raycasting and OpenGL ES 2.0 for rendering. Created by Kalle Hameleinen, it can be found here.
- **Ashley**: A tiny a high-performance entity framework, making the API easy and transparent to use. Check its repository.
- **Ai**: An artificial intelligence framework for game development with libGDX. Check the repository for more details.

At this moment, only **the Box2d library is vital**.

- **Show Third-Party Extensions**: If pressed, a list of other external libraries are shown for integration with our game. These are not maintained by the libGDX team, however, useful tools can be found there.
- **Advanced**: Shows advanced settings for the project.

At the end, the fields should look as per Figure 20:

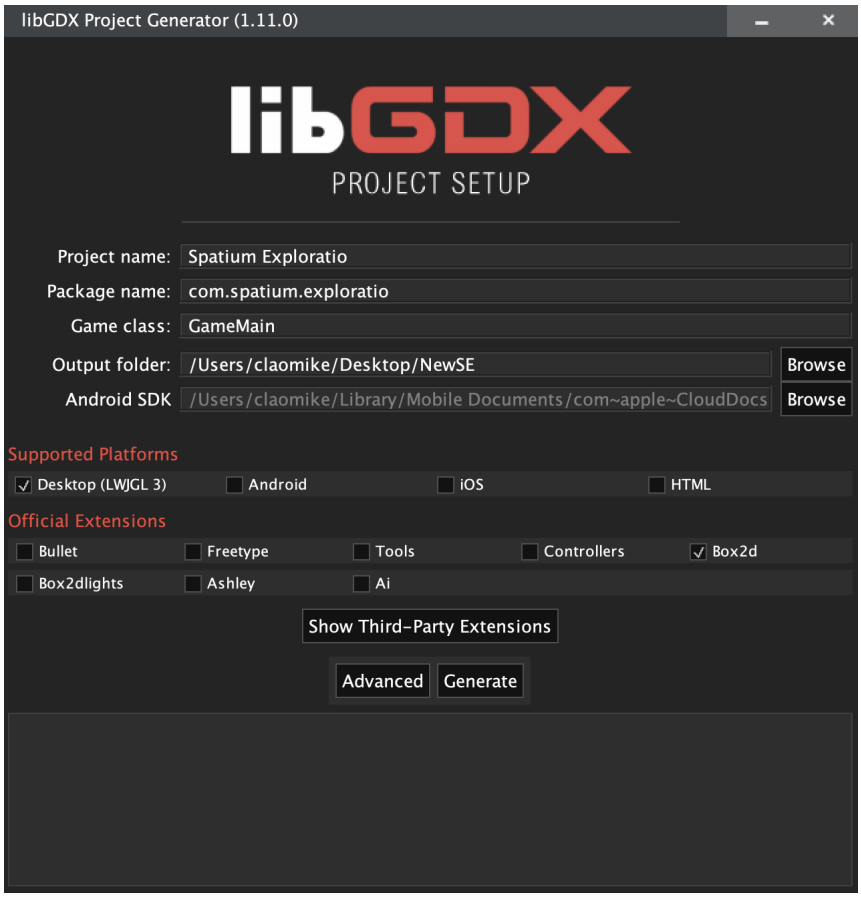

Figure 20: The libGDX tool with all of its fields filled

3. Click **Generate** and wait for the project to be generated, then close the libGDX tool.
4. Finally, open Android Studio and then the project folder.

3.5 Gradle

Gradle is an open-source build tool that is used by libGDX. It manages dependencies, builds and packages the program and performs other build related tasks. A small inconvenience is that the default way that Gradle is setup by the libGDX tool gives a few unwanted warnings, due to some deprecated features. It is a minor inconvenience and it should be addressed by the libGDX team. In the meantime, replace the original contents of the **desktop/build.gradle** file with the following script:

```
plugins {
    id 'java'
}

sourceCompatibility = 1.8

sourceSets {
    main {
        java.srcDirs = ['src']
        resources.srcDirs = ['../assets']
    }
}

ext {
    mainClassName = "dev.clao.DesktopLauncher"
    assetsDir = new File("../assets")
}

import org.gradle.internal.os.OperatingSystem

task run(type: JavaExec) {
    setMainClass(mainClassName)
    classpath = sourceSets.main.runtimeClasspath
    standardInput = System.in
    workingDir = assetsDir
    ignoreExitValue = true

    if (OperatingSystem.current() == OperatingSystem.MAC_OS) {
        jvmArgs += "-XstartOnFirstThread"
    }
}

task debug(type: JavaExec) {
    setMainClass(mainClassName)
    classpath = sourceSets.main.runtimeClasspath
    standardInput = System.in
    workingDir = assetsDir
    ignoreExitValue = true
    debug = true
}
```

```
task dist(type: Jar) {
    duplicatesStrategy(DuplicatesStrategy.EXCLUDE)
    manifest {
        attributes 'Main-Class': mainClassName
    }
    dependsOn configurations.runtimeClasspath
    from {
        configurations.runtimeClasspath.collect {
        it.isDirectory() ? it : zipTree(it) }
    }
    with jar
}

dist.dependsOn(classes)
eclipse.project.name = appName + "-desktop"
```

This should clear all the warnings.

3.6 Run the project

The last step before being able to build and run the game is to setup a configuration. Go to the toolbar on the right hand side and hit the configurations menu (Figure 21).

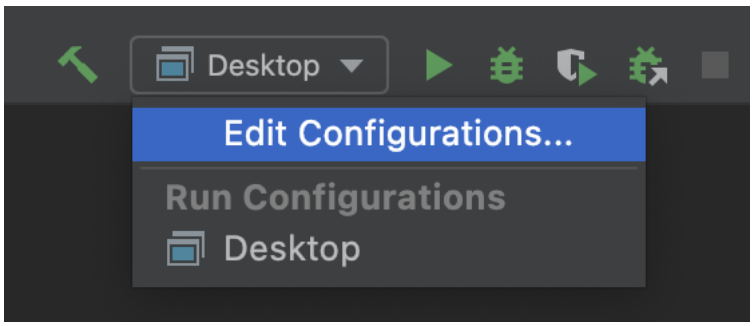

Figure 21: The top toolbar in Android Studio

Add a new configuration of type Application. A few fields must be filled in:

- **Name**: Desktop.
- **Store as project file**: check this box. This prevents having to recreate this configuration each time the editor is re-opened.
- Select the JDK version **11**.
- **-cp variable**: The structure of this field is:

  application name + deployment platform + main

  For the desktop platform, **Spatium Exploratio.desktop.main** must be selected.

- This step only applies if the development is done on a macOS device. Press Modify options and then check the **Add VM options**. Now a new field has appeared which must be filled in with the **-XstartOnFirstThread** parameter.
- Now select the main class: **com.spatium.exploratio.DesktopLauncher**. The configuration should look as per Figure 22:

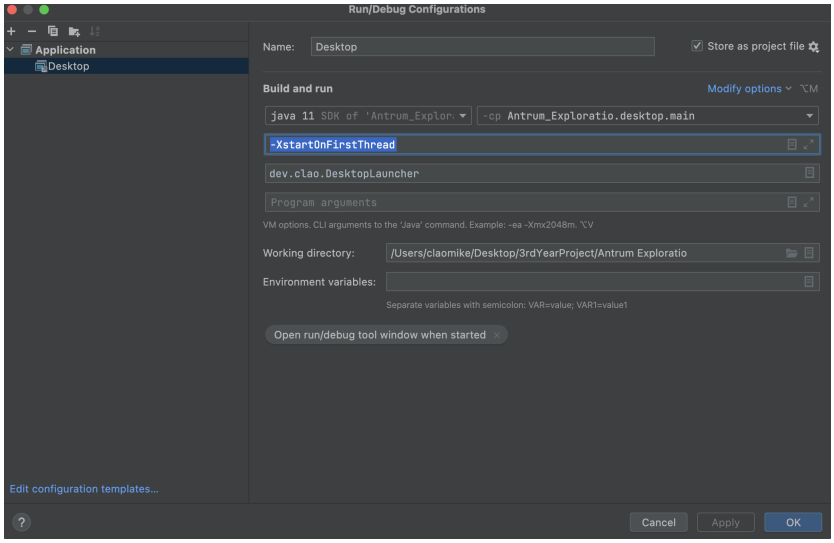

Figure 22: The Desktop configuration

- Click **Apply** and then hit the **OK** button.

Everything is setup now. After building and running the project, a new window with a red background and a default image of a smiley face will appear.

3.7 Version control

For version control, a solution is to use Git, a free and open source distributed system. Details about Git can be found on its website.

While this helps managing the work flow, it is a good idea to save a copy of the project on a regular basis in multiple locations. Therefore, use GitHub. It is free and good for making sure the project will not disappear due to unforeseen incidents.

3.8 Image editing tools

The game will need some images for the character, the walls, the walking path and other objects. There are plenty of tools out there that will get the job done, the preferred choices are:

- The Pixilart editor - It is free to use.
- The iPad version of the Procreate editor.

3.9 Audio editing tools

The game will need some audio files, such as a soundtrack for the main menu and sound effects. There are plenty of tools out there that will get the job done, such as the iPad version of the Garage Band.

4 Design

4.1 Project structure

First thing in the development process is structuring the project. There are three main parts:

- **The Assets folder**: This contains all the images, audio, font and JSON files.
- **The core package**: This includes the game logic.
- **The desktop package**: Contains the Desktop Launcher. This allows customising the window's width, height, size, title and the FPS value.

4.2 The Maze

In subsection 2.1, a maze is defined at a conceptual level. When initializing it, it will look as per Figure 23:

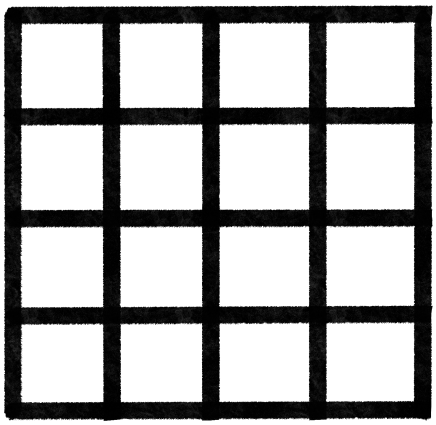

Figure 23: The Initial state of the maze

After applying **the DFS algorithm for generating a maze** to it, it could look as per Figure 24:

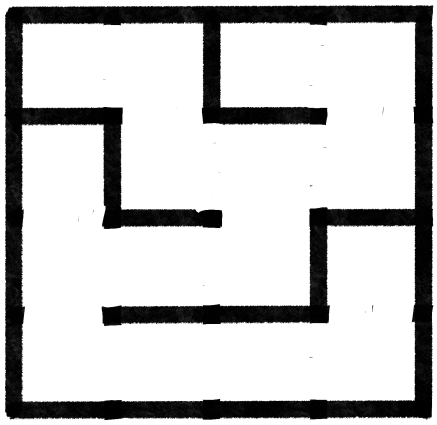

Figure 24: A possible arrangement of the maze

The way to map this maze, which again, is in form of a graph at the moment, is to map each vertex to a 4x4 matrix. So for each vertex, we will draw 16 images, each one representing either a wall, or a path block.

4.3 The images

Four images are needed at first for drawing the maze:

- A **block of dirt** (Figure 25);

Figure 25: Dirt block

- A **block of snow** (Figure 26);

Figure 26: Snow block

- A **block of dirt with snow on the edge**. Using the preferred editing tool, crop a small part of the snow block and attach it to the dirt block, obtaining Figure 27:

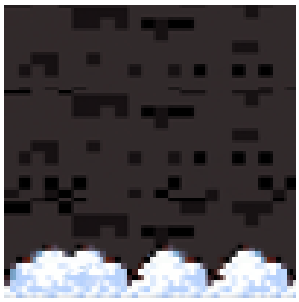

Figure 27: Dirt block with a snow edge

- A **block of dirt with snow on two consecutive edges**. Adding another snow edge to the previous image (Figure 28):

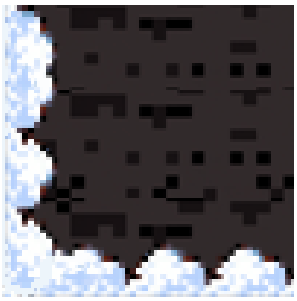

Figure 28: Dirt block with a snow corner

The advantage of this approach is that by rotating the last two images, new ones are obtained, covering all the directions in which a snow edge or corner might be facing. The result is as shown in Figure 29:

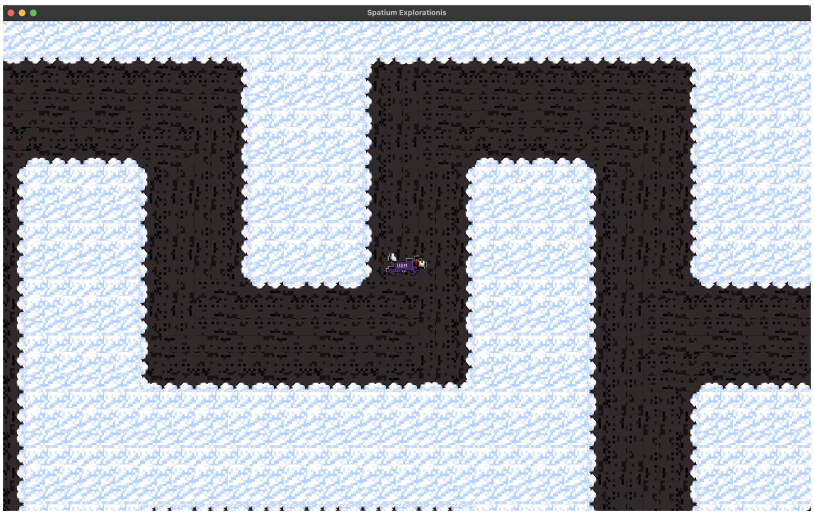

Figure 29: Example of using the generated images

These are the terrain images. Moving on to **the character**, the base image, representing the **rest state** is shown in Figure 30.

Figure 30: The main character

Of course, an **animation** must be implemented for the **walking state**. Using the theory provided in subsection 2.6, the corresponding sequence of frames is presented in Figure 31:

Figure 31: The main character's walking state

There are two other game objects that need images:

- A **spaceship** (Figure 32).

Figure 32: The spaceship

- A **satellite** (Figure 33).

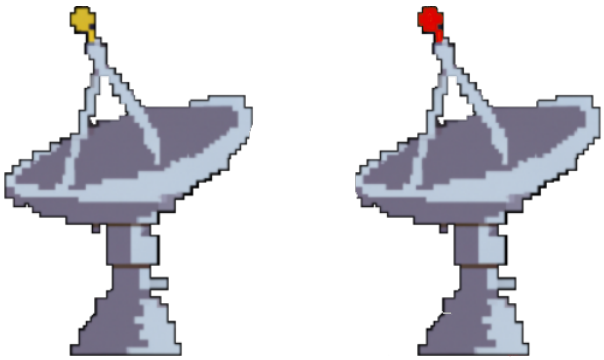

Figure 33: The satellite's animation

The game is going to have a Main Menu screen, a Settings screen, a Credits screen, and a Pause and an End in-game menus.

- The **Main Menu screen**: It will need a background image for this, as per Figure 34:

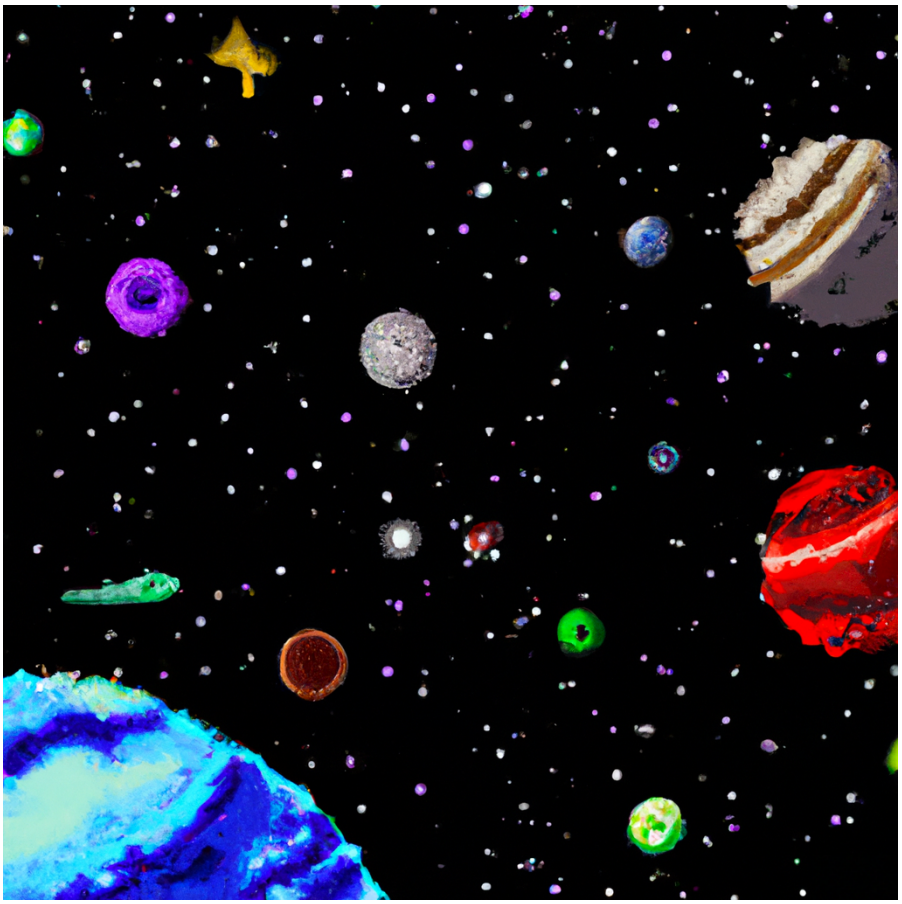

Figure 34: The main menu background

The same background will be used for the credits and settings screens.

- The **Settings screen**: It will need a slider that allows the player to switch between different sizes of the maze. It is composed of two elements: a slider and a knob; as per Figure 35 and Figure 36:

Figure 35: The background of the slider

Figure 36: The knob

The resulting slider inside the game will look as per Figure 37:

Figure 37: The complete slider

- For the **Pause and End game menus**, a background image for displaying buttons on top of is necessary, as per Figure 38:

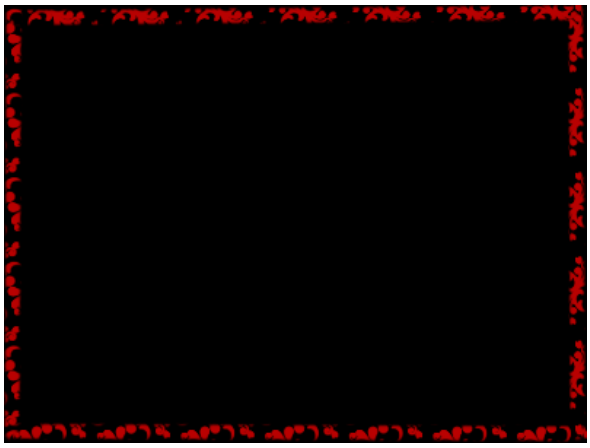

Figure 38: The pause and end menus background

These are all the required images.

4.4 The soundtrack

Creating the music and sound effects for a game usually requires an expert in the area. However, using the Garage Band app and some creativity, the process of creating a simple song becomes flawless, as shown in Figure 39.

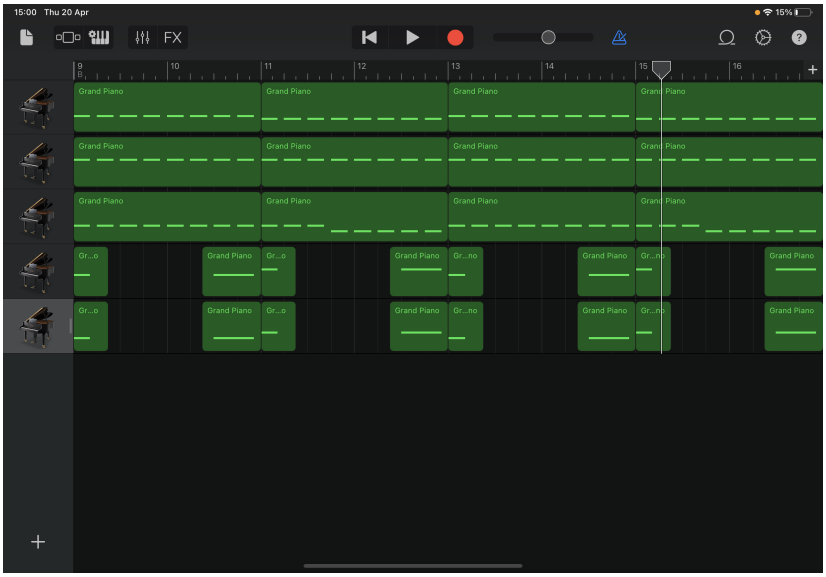

Figure 39: Garage Band

Sound effects can be created using the same application or using any device to record real life sounds.

4.5 Drawing text

There are multiple ways of drawing text in a libGDX game. One way is to use an extension, such as the Freetype library. Another way, the default one, is to import a .tff file. There are plenty of fonts available on the internet that can be used for free for personal and commercial use. The Mabook font, created by Syafrizal a.k.a. Khurasan, can be be found here.

4.6 The game logic

Everything starts with the targeted deployment platform, the Desktop platform. Configure its launcher, the **DesktopLauncher** class (Table 6), by setting the values for the FPS, title, width and height of the window, and the main class of the game.

DesktopLauncher
int FPS String GAME_TITLE int DEFAULT_WINDOW_WIDTH int DEFAULT_WINDOW_HEIGHT GameMain game
static void main (String[] arg)

Table 6: The DesktopLauncher class

The main class is called **GameMain** (Table 7), as configured in subsection 3.4. This class must extend the **Game** class, provided by the game engine. This class has the following variables:

- A variables of type **Constant**.
- A variables of type **SpriteBatch**.
- A variables of type **CustomMusic**.
- A variables of type **GameSettings**.

GameMain
Constants constants SpriteBatch batch CustomMusic customMusic GameSettings settings
void playMusic() void create () void dispose() void exit() void goToScreen(Screens s)

Table 7: The GameMain class

The **Constants** class contains all the constant values used within the game. One can be declared as follows:

```
public static final CLASS VARIABLE_NAME = VALUE;
```

The **SpriteBatch** class is a class provided by the engine, used for drawing textures. It should be only one instance of this class, shared across the whole game.

The **CustomMusic** (Table 8) class takes care of playing music. This object is able to load an audio file, play, stop and dispose of it. It also checks whether the music is still playing or not.

CustomMusic
Music music
CustomMusic(String filepath) void play() void stop() void dispose() boolean isPlaying()

Table 8: The CustomMusic class

The **GameSettings** (Table 9) class holds the user's settings (Figure 40). The main settings in our game are the size of the maze and the debug mode state, as described in subsection 2.7.

GameSettings
boolean DEBUG_MODE int MAZE_SIZE
GameSettings(Constants c)

Table 9: The GameSettings class

Figure 40: The user's settings

Extending the **Game** class, the **GameMain** (Table 7) class, has to override two methods, namely:

- the **create()** method: Its implementation must load the constants, the music, the game settings and set the initial screen. The initial screen will be either the Main Menu screen or the Gameplay screen, depending on the value of the

DEBUG_MODE setting. In case the DEBUG_MODE is true, the initial screen will be the Gameplay screen.

- the **dispose()** method: Its implementation should dispose of the sprite batch and the music previously loaded.

The GameMain class also must exit the game and switch between screens.

As described in subsection 2.3, the player will see is a specific rendered screen. It must implement the **Screen interface** (Table 5). Therefore, a class called **SimpleScreen** (Table 10) must be created. This will be the base class for all the other screens. It must get a reference to the instance of the GameMain created previously, and two variables: an **OrthographicCamera** and a **BitmapFont**.

SimpleScreen
OrthographicCamera mainCamera BitmapFont font
SimpleScreen(GameMain game) void show() void render(float delta) void resize(int width, int height) void pause() void resume() void hide() void dispose()

Table 10: The SimpleScreen class

The GameMain instance, since it holds the constants, music and the sprite batch, will be often needed, therefore we must pass it by reference between screens and objects. The BitmapFont instance loads the font file and draws text. The SimpleScreen instance sets up the font and the camera.

The game will contain the following screens: the **Main Menu screen**, the **Settings screen**, the **Credits screen** and the **Gameplay screen**. The Gameplay screen will be build after the **GameplayScreen class** (Table 21), which will extend the SimpleScreen class. The other screens will extend another class, namely the **UIScreen**. This one also extends the SimpleScreen, but adds specific features.

The **UIScreen class** (Table 11) must set up a stage, using the Stage class, provided by the game engine. This the location to place buttons, labels and other User Interface (UI) components. It also sets the background image.

UIScreen
Stage stage BackgroundImage backgroundImage
UIScreen(GameMain game) void show() void render(float delta) void resize(int width, int height) void dispose()

Table 11: The UIScreen class

The Main Menu screen (Figure 41) is built using the **MainMenuScreen class** (Table 12). This screen starts playing the soundtrack, adds a label with game title, together with a few buttons:

- **Play button**: sets the current screen to the Gameplay screen
- **Credits button**: sets the current screen to the Credits screen
- **Settings button**: sets the current scree to the Settings screen
- **Exit button**: exists the game and terminates the program.

Figure 41: The main menu

MainMenuScreen
MainMenuScreen(GameMain game) void show() void render(float delta)

Table 12: The MainMenuScreen class

Each button will be an instance of the **CustomButton class** (Table 13), which extends the **TextButton class**, provided by the game engine. The custom buttons receive a string input and the font type, both needed for drawing the text. Then, an action is attached to each of them, which will be triggered if the buttons are pressed.

The title will also be an instance of the **CustomButton class** (Table 13), however, it will be disabled and no action will be attached to it, becoming just a simple label.

CustomButton
CustomButton(String text, BitmapFont font)

Table 13: The CustomButton class

To position the buttons on screen, an instance of the ButtonTable (Table 14) class is required. This class extends the **Table class**, provided by the game engine. The Table class works with the **Actor class**, provided by the game engine. However, an improved implementation of the **ButtonTable class** (Table 14) will also include a feature that allows adding a CustomButton instance to the table, while also setting the size of the bottom padding, setting the width of the button and moving to the next row.

ButtonTable
ButtonTable(float x, float y) void addActor(Actor actor, float bottomPadding, boolean newRow, float width)

Table 14: The ButtonTable class

The stage will take care to draw the table.

Drawing the game's version can be achieved the same way as drawing the title, or by using the font's draw() method.

The **Credits screen** is created after the **CreditsScreen class** (Table 15). Using the same technique as for the Main Menu screen, the credits will be drawn using instances of the Stage, CustomButton (Table 13) and ButtonTable (Table 14) blueprints.

CreditsScreen
GameMain game
void show() void render(float delta)

Table 15: The CreditsScreen class

The Settings screen is built using the **SettingsScreen class** (Table 16). It offers the player the possibility of setting the size of the maze and whether he wants to load the Gameplay screen with debug mode activated. It will display the title, the slider for setting the maze size and a checkbox for setting the debug mode on or off. All these components are Actors, so the same technique for positioning on the screen as before will have to do.

The title is an instance of the CustomButton (Table 13) class.

SettingsScreen
GameMain game
void show() void render(float delta)

Table 16: The SettingsScreen class

The slider's class is called **SliderWithTitle** (Table 17). It will extend the ButtonTable class (Table 14) and have three attributes:

- A **String** variable representing the initial text to be displayed.
- A **CustomButton** variable representing the title label of the slider.
- A **CustomSlider** variable class representing the actual slider.

The SliderWithTitle class (Table 17) must also be able to update the title with the value of the slider, whenever the player moves the knob.

SliderWithTitle
String text CustomButton title CustomSlider slider
SliderWithTitle(String text, BitmapFont font, String backgroundFilepath, String knobFilepath, float minValue, float maxValue, float stepSize, float width, final GameSettings settings) void changed(ChangeEvent event, Actor actor)

Table 17: The SliderWithTitle class

The **CustomSlider class** (Table 18) extends the **Slider class**, provided by the game engine. An instance of the CustomSlider class needs at initialization the following parameters:

- The path to the file containing the background for the slider.
- The path to the file containing the background for the knob.
- The minimum value of the slider.
- The maximum value of the slider.
- The step size value of the slider.
- The initial value of the slider.
- The width of the slider width.

The **CheckBoxWithTitle class** (Table 19) is the blueprint for the checkbox object. It extends the ButtonTable class (Table 14) and contains two variables: a CustomButton - the title label; and a CustomCheckBox - the checkbox.

CustomSlider
CustomSlider(String backgroundFilepath, String knobFilepath, float minValue, float maxValue, float stepSize, float initialValue, float width)

Table 18: The CustomSlider class

CheckBoxWithTitle
CheckBoxWithTitle(String title, BitmapFont font, float width, float height, GameSettings settings)

Table 19: The CheckBoxWithTitle class

The **CustomCheckBox class** (Table 20) extends the Actor class and holds a boolean variable, to store the state: checked or unchecked (true or false). An object of this class is a square. If pressed, it will draw or delete a check inside the square, and update its state. Both the square and the check are built using the **ShapeRenderer class**, provided by the game engine.

CustomCheckBox
boolean isChecked
CustomCheckBox(float width, float height, final GameSettings settings) void draw(Batch batch, float parentAlpha) void generateShapeRenderer(Batch batch)

Table 20: The CustomCheckBox class

The **Gameplay screen** is built after the **GameplayScreen class** (Table 21), which extends the SimpleScreen class. This class holds the following:

- A **boolean isPaused** variable: Set to false by default. Assign true if the pause menu is being shown on the screen.
- A **boolean escapeKeyPressed** variable: Set to false by default. If the Escape key is pressed, assign true and the Gameplay screen will show/dismiss the pause menu.
- A **boolean endOfTheGame** variable: Set to false by default. When the collision system detects that the player has hit the satellite, it will set its value to true, and the Gameplay screen will stop processing the user input and show the end menu.
- A **World** variable: Used for calculating the effects of the interactions between the game objects.
- A **FollowingCamera** variable: A camera that follows the character.
- A **Terrain** variable: Generates the start and end platforms, and the maze.
- A **Spaceship** variable: An object placed at the beginning of the maze to stop the player exiting the map.

- A **Player** variable: The player's character.
- A **Satellite** variable: An object placed at the end of the maze, to mark the finish point.
- A **Stage** variable: It is used for drawing the pause and end menus.

An instance of the **GameplayScreen class** (Table 21) must be able to process the player's input, show and dismiss the end and pause menus when necessary, draw, dispose of and update the position of all the objects, and resize the camera.

GameplayScreen
World world FollowingCamera camera Terrain terrain Spaceship spaceship Player player Satellite satellite Stage stage boolean isPaused boolean escapeKeyPressed boolean endOfTheGame
GameplayScreen(GameMain game) void show() void showEndMenu() void dismissPauseMenu() void render(float delta) void resize(int width, int height) void dispose()

Table 21: The GameplayScreen class

The pause and end menus are instances of the **PauseMenu** (Table 22) and **EndMenu** (Table 23) classes. Both classes differ in terms of what buttons display and what actions they trigger, but they both extend one class: the GameMenu class (Table 24), which extends the ButtonTable class (Table 14).

The pause menu (Table 22) has the following two buttons: **Resume** (which closes the menu and resumes the game play) and **Main Menu** (which sets an instance of the MainMenuScreen class as the current screen).

PauseMenu
PauseMenu(GameMain game, BitmapFont font, final GameplayScreen screen)

Table 22: The PauseMenu class

The end menu (Table 23) has the following two buttons: **Play Again** (which restarts the game) and the same **Main Menu** button as the pause menu.

EndMenu
EndMenu(GameMain game, BitmapFont font, final GameplayScreen screen)

Table 23: The EndMenu class

The **GameMenu class** (Table 24) holds two CustomButton variables. A GameMenu object has a background, making the buttons easily distinguishable from the rest of the screen. The Main Menu button is defined inside this class, as it is shared by both the PauseMenu (Table 22) and the EndMenu (Table 23).

GameMenu
GameMenu(final GameMain game, String button1Text, BitmapFont font)

Table 24: The GameMenu class

An explanation is owed for some of the variables of the Gameplay class.

The **Texture class** is provided by the game engine. It is the engine's way of loading an image. The **Sprite class** is provided by the game engine and it is build using a Texture object. Basically, it is a Texture with more features, such as updating its position, rotation and much more. Extending this class into a new one, called **CustomSprite** (Table 25), reduces the number of code lines needed for setting up a Sprite. Extending this class even further, creating the **CustomSpriteWithBody class** (Table 26), adds a body (Bounding Volume), with mass and shape. Now forces and impulses can be applied to game objects.

CustomSprite
CustomSprite(String filepath) void dispose()

Table 25: The CustomSprite class

CustomSpriteWithBody
Body body Shape shape
CustomSpriteWithBody(String filepath, Body body) Body getBody() void generatePolygonShape() void generateCircleShape(float radius) Shape getShape() void disposeShape() void attachFixture(FixtureDef fixtureDef)
void updatePositionToBody()

Table 26: The CustomSpriteWithBody class

The class **Atom** (Table 27) represents the base class for game objects that must be drawn on a screen, have a body and can be detected by the collision detection system.

Atom
CustomSpriteWithBody sprite GameMain game
Atom(GameMain game, String filepath, World world, BodyDef.BodyType bodyType, float x, float y, float density, boolean isSensor, float radius) void updatePosition() void draw() CustomSpriteWithBody getSprite()

Table 27: The Atom class

The **Spaceship class** (Table 28) is build on top of the Atom class, using specific parameters for initializing an instance of it.

Spaceship
Spaceship(GameUtils utils, float x, float y) void updatePosition() void dispose()

Table 28: The Spaceship class

The **Satellite class** (Table 29) is build on top of the Atom class, using specific parameters for initializing an instance of it.

Satellite
Satellite(Vector2 coordinates, GameUtils utils) void updatePosition()

Table 29: The Satellite class

The **CustomAnimation class** (Table 30) is the base class for an animation. It is similar to the Texture class. Extending it to a new class **CustomAnimationWithBody** (Table 31), allows it to be affected by interactions with other objects.

CustomAnimation
GameUtils utils Animation<TextureRegion>animation Texture texture float stateTime float x float y
CustomAnimation(String filepath, float frameDuration, GameUtils utils, int FRAME_COLS, int FRAME_ROWS) void draw() void updatePosition(float x, float y) void dispose() float getWidth() float getHeight()

Table 30: The CustomAnimation class

CustomAnimationWithBody
Body body
CustomAnimationWithBody(String filepath, float frameDuration, Body body, GameUtils utils, int FRAME_COLS, int FRAME_ROWS) void attachFixture(FixtureDef fixtureDef) void updatePosition() Body getBody()

Table 31: The CustomAnimationWithBody class

The **Player class** (Table 32) represents the blueprint for the game's main character. It extends the Atom class, however, it also contains a CustomAnimationWithBody variable. This allows switching between the rest texture and the walking animation, based on the movement state. The movement state is set by an instance of the **PlayerInputProcessor** class (Table 33), which detects what input comes from the player and applies a force to the character's body such that it moves in the desired direction.

Player
Constants constants int movementMultiplier PlayerInputProcessor playerInputProcessor CustomAnimationWithBody animation boolean isMoving
Player(GameUtils utils) void move(Directions direction) void stopMoving(Directions direction) void dispose() void pauseMovement() void setInputProcessorToPlayer() void draw() void updatePosition()

Table 32: The Player class

PlayerInputProcessor
Player player
PlayerInputProcessor(Player player) boolean keyDown(int keycode) boolean keyUp(int keycode) boolean keyTyped(char character) boolean touchDown(int screenX, int screenY, int pointer, int button) boolean touchUp(int screenX, int screenY, int pointer, int button) boolean touchDragged(int screenX, int screenY, int pointer) boolean mouseMoved(int screenX, int screenY) boolean scrolled(float amountX, float amountY)

Table 33: The PlayerInputProcessor class

The **Terrain class** (Table 34) is the blueprint for building the map. When an instance of this class in initialized, it must receive a **Vector2** parameter, representing the start coordinates from where the map should start being drawn. It has three variables: a start platfom, the maze and an end platform. The platforms are instances of the **Platform class** (Table 35).

Based on the start coordinates, the Terrain class calculates the initial coordinates for all of the above variables and passes them accordingly.

The **Cube class** (Table 36) is a matrix of 16 blocks. Based on a list of allowed directions for movement, it draws or not the edge and/or the center blocks.

Terrain
Platform startPlatform CubeMaze maze Platform endPlatform
Terrain(GameUtils utils, Vector2 coordinates) Vector2 getEndPlatformMiddleCoordinates() void updatePosition() void draw() void dispose()

Table 34: The Terrain class

Platform
GameUtils utils Array<Cube>cubes float endX float startX float startY
Platform(Vector2 coordinates, int cubesCount, Directions directions, GameUtils utils) void draw() void dispose() void updatePosition() float getEndX() Vector2 getMiddleCoordinates()

Table 35: The Platform class

Cube
GameUtils utils Array<CustomSprite>sprites Array<Block>blocks
Cube(BlockTypes[][] arrangement, Vector2 coordinates, GameUtils utils) float startY Platform(Vector2 coordinates, int cubesCount, Directions directions, GameUtils utils) void updatePosition() void dispose()

Table 36: The Cube class

The maze is build using the **CubeMaze class** (Table 37). It creates a Maze instance and generates a graph, then maps each Vertex to an instance of the Cube class, and then draws them.

CubeMaze
Array<Cube>cubes float endX float exitY
CubeMaze(int mazeSize, Vector2 coordinates, GameUtils utils, int entranceRow, int exitRow) void updatePosition() void draw() void dispose() void getEndX() void getExitY()

Table 37: The CubeMaze class

The Platform class (Table 35) places two rows of Cube objects, as shown in Figure 42:

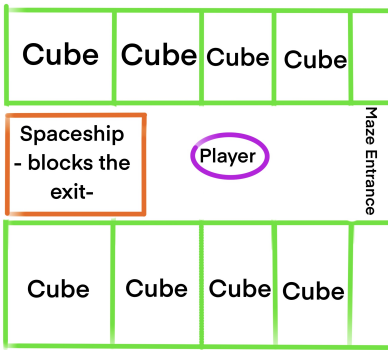

Figure 42: The positioning of the start platform

The end platform is the same as the start platform, except that the maze entrance and the object blocking the exit switch places.

The World class does all the math behind the physics interaction between the game objects. This class is provided by the Box2d library.

The **FollowingCamera class** (Table 38) consists of two OrthographicCamera variables, one following the character's image and the other following the character's bounding volume. If the debug mode is turned on, the two cameras are combined, the player being able to see both images overlapped.

FollowingCamera
GameMain game World world OrthographicCamera mainCamera OrthographicCamera debugCamera Box2DDebugRenderer debugRenderer boolean debugModeIsOn
FollowingCamera(GameUtils utils) void setProjection() void followSpriteAndBodyOf(CustomSpriteWithBody sprite) void followBody(Body body) void followSprite(Sprite sprite) void resize(int width, int height)

Table 38: The FollowingCamera class

5 Results

Considering the instructions in this report have been correctly followed, the result will be a **game with content generated procedually**. In this case, the content comes in the form of a maze, which is different each time a new gameplay session is started. Here is some live footage:

- The Main Menu screen (Figure 43)

Figure 43: The Main Menu screen

- The Settings screen (Figure 44)

Figure 44: The Settings screen

- The Settings screen with the DEBUG MODE checkbox ticked (Figure 45)

Figure 45: The Settings screen

- The Credits screen (Figure 46)

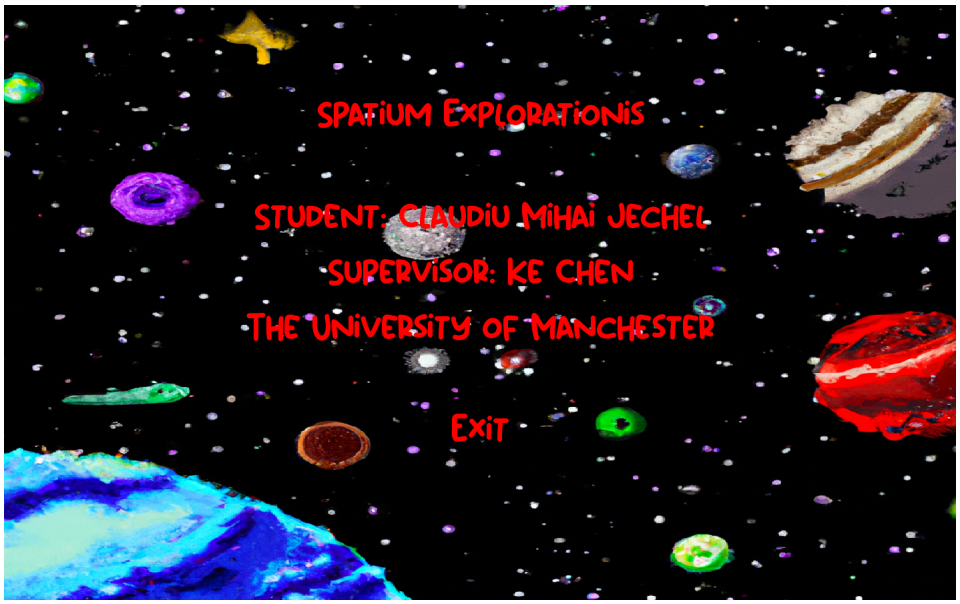

Figure 46: The Credits screen

- The Gameplay screen - at start (Figure 47)

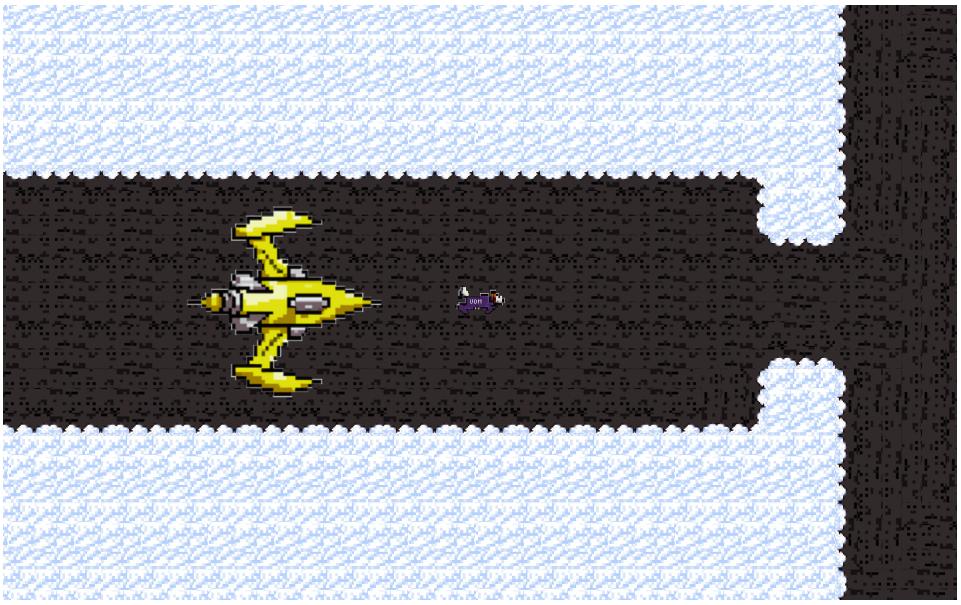

Figure 47: The Gameplay screen

- The Gameplay screen - trying to find the exit (Figure 48)

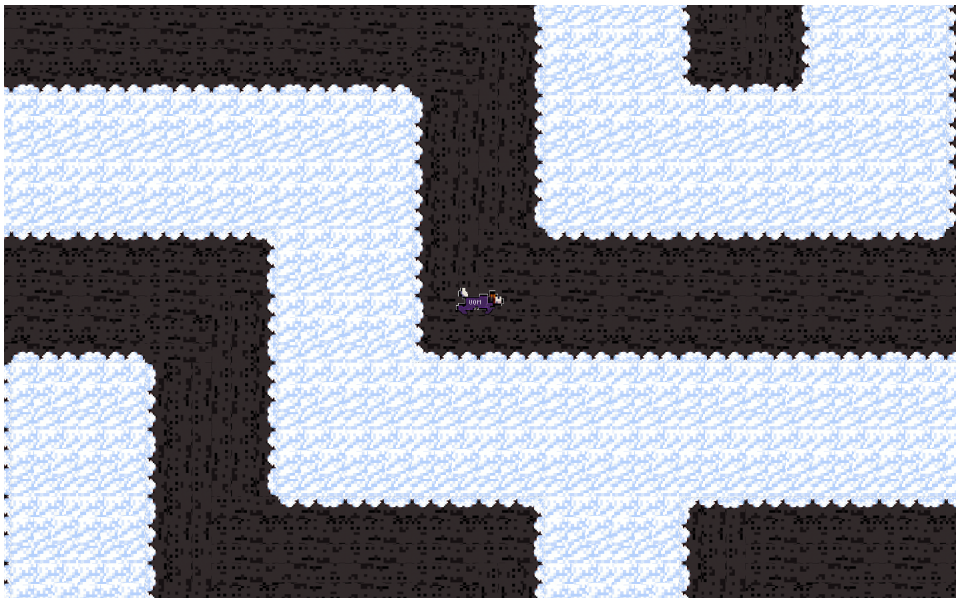

Figure 48: The Gameplay screen

- The Gameplay screen - reaching the exit (Figure 49)

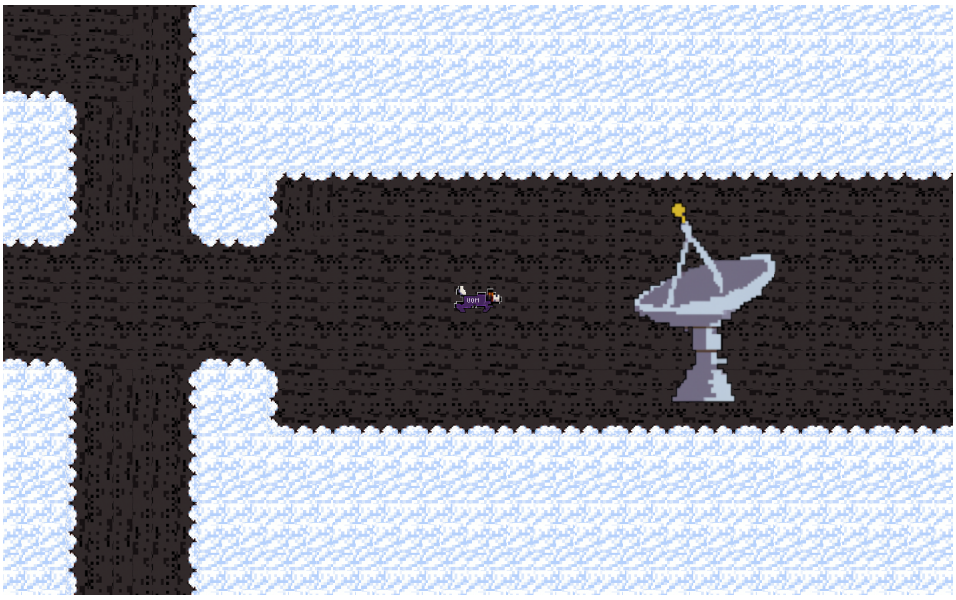

Figure 49: The Gameplay screen

- The Gameplay screen - the pause menu (Figure 50)

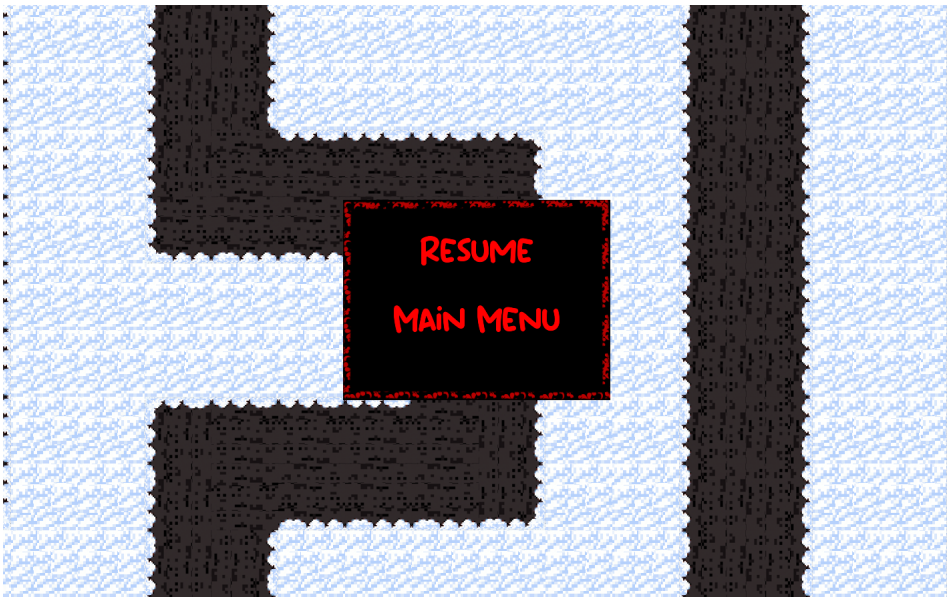

Figure 50: The Gameplay screen

- The Gameplay screen - the end menu (Figure 51)

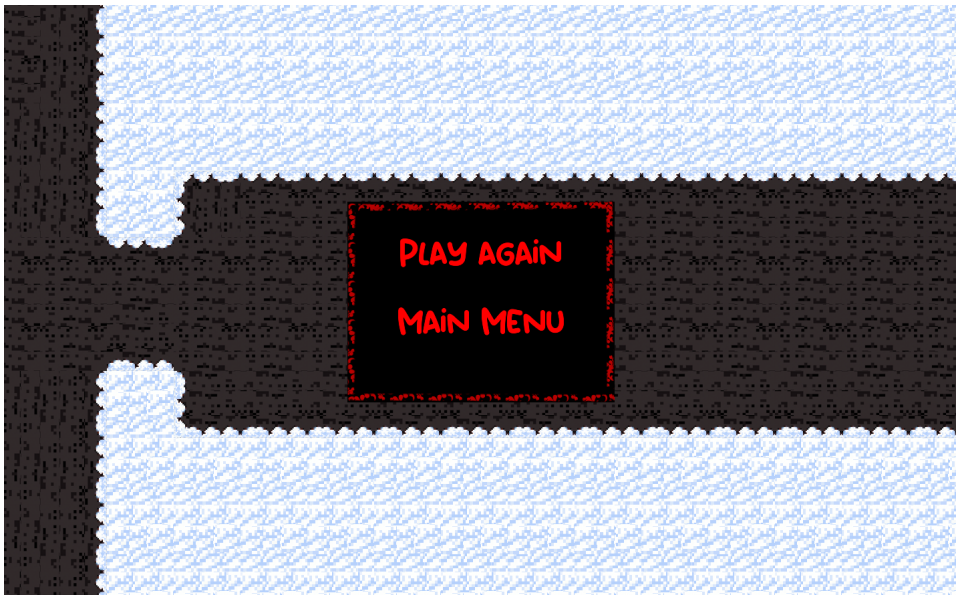

Figure 51: The Gameplay screen

- The Gameplay screen when the game has the debug mode turned on (Figure 52)

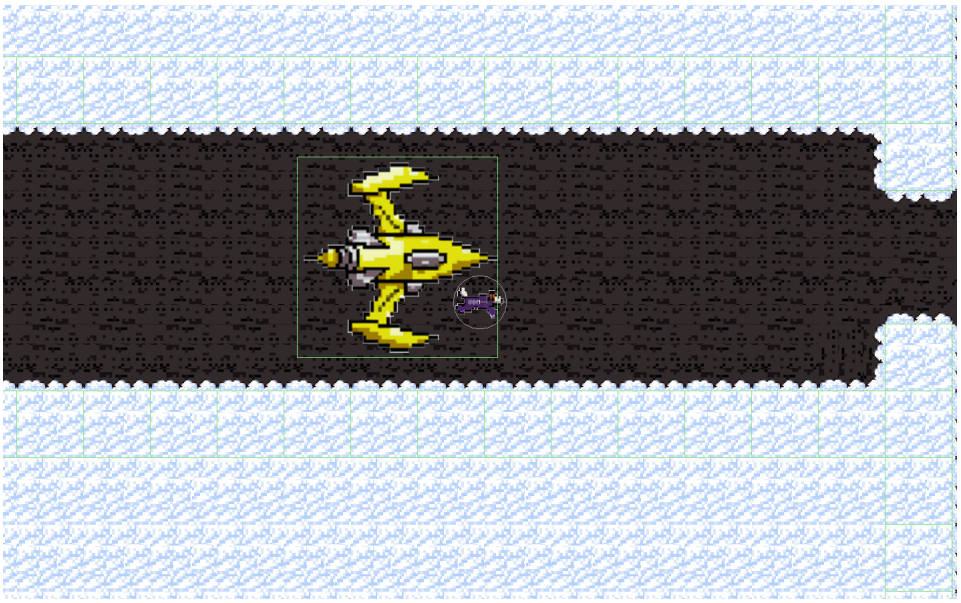

Figure 52: The Gameplay screen

6 Evaluation

6.1 Performance

In terms of correctness, the algorithm works as expected. Every time a new maze is being generated. However, it has its limitations:

- Since it is using a recursive implementation, the generated mazes might contain many long corridors. That is due to the DFS's nature of exploring as far as possible before backtracking.
- Depending on the operating system and the available memory, the algorithm's running time is different. For example, if the game is build and run on a computer using macOS, a graph of size 50x50 will take a few good minutes to be generated. Moreover, displaying all the images and calculating how objects interact with each other lags the game. This is not a problem on a modern Windows operated computer.

6.2 Comparison

There are plenty of choices (Wikipedia Contributors, 2019b) for replacing the recursive implementation of the DFS algorithm:

- **DFS (the iterative implementation)**
- **Randomized Kruskal's algorithm** - As the name suggest, this is randomized version of the original version. While there are multiple data structures that can be use to represent the cells, an efficient implementation is using a disjoint-set, that can perform each union and find operation on two sets in nearly constant amortized time. Making the running time of this algorithm is proportional to the number of walls available to the maze. Its implementation is easy to code. The disadvantage of this algorithm is that usually produces regular patterns, which are easy to solve.
- **Randomized Prim's algorithm** - This one introduces a stylistic variation compared to the previous one, because the edges closer to the starting point have a lower effective weight. If it was just the original version (not randomized), then the maze's style would be the same as Kruskal's.
- **Randomized Prim's algorithm (modified)** - Using a list of adjacent cells. If a randomly chosen cell has multiple edges that connect it to the existing maze, it would randomly select one of these edges. This tends to branch slightly more than the edge-based version.
- **Randomized Prim's algorithm (simplified)** - A further improvement can be done for the modified version, by randomly selecting cells that neighbour visited cells, rather than keeping track of the weights of all cells or edges. This leads to an easy way of finding the path to the starting cell, but difficult to find a path anywhere else.

- **Wilson's algorithm** - Generates an unbiased sample from the uniform distribution over all mazes, using loop-erased random walks. This is better than the DFS algorithm, which is biased towards long corridors.
- **Aldous-Broder algorithm** - Produces uniform spanning trees, but it is one of the least efficient maze algorithms.
- **Recursive division method** - This algorithm works dividing a room with a wall at a randomly chosen point. Then it randomly chooses a place in the wall to make gap. Then the process of division is repeated for a number of times.
- **Eller's algorithm** - Prevents loops by storing which cells in the current line are connected through cells in the previous lines, and does not remove walls between any two connected cells.
- **The Sidewinder algorithm** - It starts with an open passage along the entire top row, and subsequent rows consist of shorter horizontal passages with one connection to the passage above.
- **Maze** - A Life-like cellular automaton in which cells survive from one generation to the next if they have at least 1 and at most 5 neighbours, resembling Conway's Game of Life in a few ways ways.
- **Mazectric** - It is the same as the above except that cells don't survive if they have 5 neighbours.
- **Variational Autoencoders** - Autoencoders are a kind of neural network trained to produce an output that wants to be as similar as possible to the input. It is composed of an encode and a decoder. More details available on Medium.

While there are other alternatives for procedually generating a maze, as mentioned above, when it comes to building a large application that combines many areas, the developer is often left with a simple approach: build a simple system, and then improve each feature, one at the time. Therefore, no one can argue against the simplicity of the DFS algorithm. In the end, it got the job well done.

When it comes to the game engine, libGDX, a few remarks can be made about it. It is indeed a friendly engine for the more skilled programmer, as there is no GUI to help. It is obvious that it should not be the first choice for a person without much programming experience. On top of it, it is build with Java, which offers a pleasant OOP experience. On top of everything, it is free, for both commercial and non-commercial use, and has a lot of resources available online. The fact that it offers cross-platform deployment is one of its biggest sell points.

Although it is arguable, its main competitor is the Unity engine. While the learning curve is stiffer than in the case of libGDX, sometimes a GUI is usefull, especially when dealing with animations. Unity uses C# as its main programming language, which is very similar to Java.

Where Unity excels, compared to libGDX, is its massive and active community full of tutorials. It also has the Unity Asset Store, which is full of useful tools and assets to help the developer.

The tools used in this report for creating the visual files are some of the best, especially the Pixilart editor, because it is open-source. The same goes for the Garage Band application. Due to its simple GUI, no more than two hours are needed for creating a simple, rhythmic soundtrack.

6.3 UX aspects

When it comes to designing an UI, one must take into consideration UX tips and tricks. The minimum amount of knowledge for this are the Eight Golden Rules of Interface Design (Wong, 2018), writen by Ben Shneiderman. The game presented in this report abides by these rules as follows:

- **Strive for consistency**: The main menu, the credits, the settings, the end and pause menus are built using the same positioning style, font size and colours. The back button on from the Credits and Settings screen are placed at the exact same position on the screen, regardless of the window's size, making it easier for the player to memorize it location. The end and pause menus are fairly simple in their design, appearing at the same location on the screen and having similar buttons. Both menus share the go to Main Menu button. One background image is shared across the Settings, Credits and Main Menu screens.

- **Enable frequent users to use shortcuts**: In case the player wants to exit the Gameplay screen and go back to the main menu, for example, to change the size of the maze, he has the option of pressing the ESCAPE key and then press the Go to Main Menu button, as a shortcut. At the same time, when the player has finished a game, he can opt for playing again straight from the end menu, rather than going to the Main menu and then hit the Play button again.

- **Offer informative feedback**: The player is greeted with the end menu when he reaches the exit of the maze, instead of the screen disappearing and going straight to the Main Menu screen.

- **Design dialogue to yield closure**: The buttons are very intuitive, as the action that they trigger are described in their own labels.

- **Offer simple error handling**: In this state of the game, there should be no bugs. In a qualitative assessment (QA) session that took care with people from the University of Manchester, no bugs were found in the artefact built for this report.

- **Permit easy reversal of actions**: Every action in this game is reversable. If the player went into the Gameplay Screen with the wrong value set for the size of the maze, he can easily go back and change that. If by mistake he did enter one of the Settings or Credits screen without wanting to, he has a simple Back button that he can press to go back to the Main Menu.

- **Support internal locus of control**: Due to the lack of complicated actions, the player will not be stressed for triggering any of the available action in the game, thus giving him the impression of having full control over the system. Of course, he does not have control over finding the path towards the exit, he just has to play.

- **Reduce short-term memory load**: Due to the simplicity of the UI, it should be the case for the average individual that chooses to play this game to learn all the commands in a short period of time.

6.4 Improvements

Have a look of some possible improvements that can be made on top of the current game:

- **AI generated images**: With the recent appearance of AI models trained to generate images, such as DALL-E 2, one could integrate one through the use of an Application Programming Interface (API), to generate different images, based on keywords. This would allow the player to explore different planets, other than Pluto. After all, the game is called Spatium Exploratio.

- **AI generated sounds**: There are AI models that can generate music and sound effects, same as the previous example. So why not integrate this as well.

- **Curtain-like effects**: One issue of the current state of the game is that the player can see other corridors in the proximity of his location. While this does not give lots of clues about the exit's location, it can be considered a cheating system. To avoid this, a curtain-like effect could be placed on the screen to hide other paths. For example, if the player is on Pluto, a snow storm can be simulated to hide the rest of the map. If the environment was placed in a system of caves, one a shadow effect could be placed.

- **Resources**: Additional game objects could be added to the game, such as minerals: copper and iron; and other chemical elements. Thus, an extra mission for the player: to discover all the resources in the game. The progress could be saved inside a game journal, which should show the total number of items and the number of discovered and undiscovered items.

- **Monetary system**: This one builds on top of the above feature: Since the player can find and collect resources, why not stack them up and exchange them for in-game money. A digital currency could be created.

- **Utility store**: This feature is dependent on the monetary system. Since there is a currency, why not make a shop where the player can buy different item that can aid the process of finding the exit of a maze, such as torches - for improved visibility, signs - to be placed around so that the player knows what paths have already been visited, defense tools and others.

- **Enemies**: The player can purchase defense tools from the shop, but what for? The answer is to defend himself from the enemies that spawn at random locations on the map. Different classes of enemies, with different levels of health and attack, could cross the player's path, if implemented.

- **Random events**: Talking about enemies spawning at random locations, the player could be further challanged by events that are randomly triggered. For example, there might be small asteroids falling, which would require a prompt response from the player.

- **Multiplayer**: Make the game even more fun by playing it with friends. As an example, a race between two player, the winner being the one that reaches the end before the other one.
- **Cross-platform support**: After a final product has been created for the Desktop, with a small amount of time, cross-platform support could be added. Thus, enabling players to play the game on Android/IOS devices and gaming consoles.

7 Conclusion

In conclusion, this report discussed the game development process using a PCG technique for maze generation. Details about how one can use the recursive implementation of the DFS algorithm to generate a maze, how a game engine works, the process of building game assets, setting up the working environment, and designing the game logic have been described throughout the sections of this report. Ultimately, the reader should have a strong background about how a 2D game with PCG can be developed. Finally, it presents the results of applying the topics mentioned above and evaluates them in terms of algorithm performance, comparing the chosen techniques and similar algorithms. The evaluation also contains a UX analysis of the UI and proposes further improvements.

Bibliography

1. Balasubramanian, K. (2022) Game Engines: All You Need to Know, Gameopedia. Available at: https://www.gameopedia.com/game-engines-all-you-need-to-know-about/.

2. Cathode-ray tube amusement device (2022) Wikipedia. Available at: https://en.wikipedia.org/wiki/Cathode-ray_tube_amusement_device.

3. De, A., Cruz, L. and Ryan, J. (2015) Tennis For Two. Available at: https://web.mit.edu/6.101/www/s2016/projects/angeld19_Project_Final_Report.pdf.

4. Glaiel, T. (2021) How to make your own game engine (and why), Geek Culture. Available at: https://medium.com/geekculture/how-to-make-your-own-game-engine-and-why-ddf0acbc5f

5. Harper, S. (no date) Read UX from 30,000ft | Leanpub, leanpub.com. Available at: https://leanpub.com/UX/read#c-input (Accessed: 18 April 2023).

6. History.COM Editors (2017) Video Game History, HISTORY. A&E Television Networks. Available at: https://www.history.com/topics/inventions/history-of-video-games.

7. Krimgen, M. (2020) Algorithm to Generate a Maze | Baeldung on Computer Science, www.baeldung.com. Available at: https://www.baeldung.com/cs/maze-generation.

8. Mary Bellis (2019) Spacewar: The First Computer Game, ThoughtCo. Available at: https://www.thoughtco.com/history-of-spacewar-1992412.

9. Museum, A. (no date) Pong Arcade Schematics. Available at: https://www.arcade-museum.com/manuals-videogames/P/PongSchematics.pdf (Accessed: 18 May 2023).

10. Nim (2020) Wikipedia. Available at: https://en.wikipedia.org/wiki/Nim.

11. Nimrod (computer) (2022) Wikipedia. Available at: https://en.wikipedia.org/wiki/Nimrod_(computer).

12. Noor Shaker et al. (2018) Procedural Content Generation in Games. Cham Springer International Publishing Springer. Available at: https://www.academia.edu/1578545/Procedural_content_generation_in_games.

13. OXO (video game) (2021) Wikipedia. Available at: https://en.wikipedia.org/wiki/OXO_(video_game).

14. Rockstar Advanced Game Engine (no date) GTA Wiki. Available at: https://gta.fandom.com/wiki/Rockstar_Advanced_Game_Engine.

15. Salviati, J. (2022) Maze generation with Variational Autoencoders, Medium. Available at: https://1littleendian.medium.com/maze-generation-with-variational-autoencoders-8a4
(Accessed: 21 April 2023).

16. Wikipedia Contributors (2019a) Grand Theft Auto V, Wikipedia. Wikimedia Foundation. Available at: https://en.wikipedia.org/wiki/Grand_Theft_Auto_V.

17. Wikipedia Contributors (2019b) Maze generation algorithm, Wikipedia. Wikimedia Foundation. Available at: https://en.wikipedia.org/wiki/Maze_generation_algorithm.

18. Wikipedia Contributors (2019c) Minecraft, Wikipedia. Wikimedia Foundation. Available at: https://en.wikipedia.org/wiki/Minecraft.

19. Wikipedia Contributors (2019d) Pong, Wikipedia. Wikimedia Foundation. Available at: https://en.wikipedia.org/wiki/Pong.

20. Wikipedia Contributors (2019e) Tennis for Two, Wikipedia. Wikimedia Foundation. Available at: https://en.wikipedia.org/wiki/Tennis_for_Two.

21. Wong, E. (2018) Shneiderman's Eight Golden Rules Will Help You Design Better Interfaces, The Interaction Design Foundation. UX courses. Available at: https://www.interaction-design.org/literature/article/shneiderman-s-eight-golden-ru