## CONTENTS

1 **Introduction** 5

2 **Problem Definition and Analysis** 6
   2.1 Problem Definition 6
   2.2 Problem Analysis 6
   2.3 Skeleton Game Description 7
   2.4 Requirements 7

3 **Design and Implementation** 8
   3.1 LibraRynth 8
      3.1.1 Maze Generator 8
      3.1.2 PlayerObserver 19
   3.2 PortalBug-LibraRynth Interactions 21
      3.2.1 Maze Generator 21
      3.2.2 PlayerObserver 21
      3.2.3 antagonist 22
   3.3 LevelGrid Generator 22
   3.4 Antagonist Behaviour 23
      3.4.1 RainAI 23
      3.4.2 Navigation 24
      3.4.3 Player Detection 24
      3.4.4 Behaviour Tree 24
   3.5 Power-ups 26
   3.6 Further Modifications to Skeleton product 26
      3.6.1 Revamp of code smells 26
      3.6.2 Preparation for LibraRynth extension 27
      3.6.3 Audio 27
      3.6.4 Health bars 27
      3.6.5 Graphics 27

4 **Final Product** 29
   4.1 Description 29
      4.1.1 LibraRynth 29
      4.1.2 PortalBug Extended 29
   4.2 Evaluation Requirements 30
      4.2.1 Evaluation Requirements: LibraRynth 30
      4.2.2 Evaluation Requirements: Antagonist Behaviour 30
      4.2.3 Evaluation Requirements: MoSCoW 31

5 **Process** 32
   5.1 Organisation 32
      5.1.1 Division of Labour 32
      5.1.2 Communication 32
      5.1.3 Collaboration Tools 32
   5.2 Software Methodology 33
      5.2.1 Scrum 33
      5.2.2 Software Development Tools 34
   5.3 Quality Assurance 34
      5.3.1 Software Improvement Group 34
      5.3.2 Testing 35
## 5.4 Process Evaluation

- **5.4.1 Organisation** ........................................... 35
- **5.4.2 Collaboration** ......................................... 36
- **5.4.3 Scrum** .................................................... 36

## 6 Conclusion

- **6.1 Conclusion** .................................................. 37
- **6.2 Challenges** .................................................. 37
  - **6.2.1 What constitutes good AI?** .......................... 37
  - **6.2.2 Unity prefabs** .......................................... 37
  - **6.2.3 Limited documentation for RainAI** ................. 38
- **6.3 Discussion and Recommendations** ....................... 38
  - **6.3.1 Improvement in maze generation speed** ............ 38
  - **6.3.2 Improvements in the antagonist behaviour** ........ 38
  - **6.3.3 Improvement of PortalBug** ............................ 38
- **6.4 Future Work and Extensions** ............................. 38

### Bibliography

- **A Glossary** ..................................................... 40

### B Maze Generator API

- **B.1 LibraRynth API** .......................................... 41
- **B.2 LibraRynthData API** ...................................... 42

### C Software Improvement Group Feedback

- **C.1 First software analysis.** ................................. 43
- **C.2 Second software analysis** ............................... 43

### D Infosheet

- **E Team contract** ............................................. 46

### F How to fix: RainAI

- **G Original project description** .............................. 57

### H Research Report

- **39**
- **40**
- **41**
- **42**
- **43**
- **44**
- **46**
- **49**
- **57**
- **68**
This is the final report for the Bachelor’s Thesis executed by Bas Boellaard, Tom Peeters, Valentine Mairet and Wendy Bolier as part of the Bachelor Computer Science program at Delft University of Technology. For the past ten weeks, we have been working on the Personalised Gaming Using Machine Learning project, commissioned by Divireo. This report contains all the information about the project.

Various people helped us in realizing this project. We would like to thank two of them in specific. First of all, Mr. Reijm, thank you for being an outstanding client. You were clear in what you expected from us and you were always available for questions and meetings. Secondly, Dr. Bulling, thank you for guiding us through this project and giving us excellent feedback. We were always able to reach you when we needed help, even while travelling or in the weekends you took made time to read and judge our work.

Bas Boellaard
Tom Peeters
Valentine Mairet
Wendy Bolier
Delft, June 2015
This Bachelor Thesis is part of the Bachelor Computer Science and Engineering program at Delft University of Technology. During this project the team had to create an extension to an existing game framework. This extension consisted of improved artificial intelligence for the antagonist and a maze generator that could be re-used as a library.

The original framework consisted of a game that was implemented in Unity. The game featured a bug that had to traverse several mazes that were interconnected by portals. These portals would allow it to travel between mazes. The mazes also featured an antagonist that will pursue the player and attempt to kill it. The intention of the game is to be challenging and difficult, and allow the framework to easily modify the difficulty settings. The game has an atmosphere that resembles a horror game and should be scary to a certain extent.

The implementation of the artificial intelligence was done with Rain AI, a freely distributed library that is used in combination with Unity. This library allows for the creation and modification of the behaviour of the antagonist through use of a behaviour tree. This behaviour tree allows the antagonist to react differently depending on its state.

The implementation of the maze generator was created separately from the main framework. This allows for it to be easily taken out as a whole, thus effectively acting as a library. It depends on various parameters to generate maze data, which the framework can provide. The framework can also specify a specific difficulty, making the maze generator use default parameters for that difficulty. The maze data is converted to actual Unity 3D objects in the framework.

In order to provide the parameters for the antagonist AI and the maze generator, a player-observer was made. This observer tracks the decisions made by the player and utilises this to generate statistics regarding the player. These are then converted to parameters that are used by the antagonist and the maze generator. This conversion depends on the difficulty settings set for the player.

In addition there have been improvements to the graphical design and gameplay. The gameplay now features more items that can be used by the player as well as improved graphics and the usage of sound.
PortalBug, the skeleton game that was provided by Divireo, contained a collection of mazes generated by an adapted version of Prim's algorithm, mentioned in the Research Report (Appendix H). Its mazes contained portals that were linearly distributed. It also contained a playable character and an antagonist, whose purpose was to find and kill the player. In this version, the intelligence for the antagonist was poorly implemented and lacked any logic. It would merely walk around at random and ignore the player entirely.

Our task was to improve the intelligence of the antagonist in order to create a much more difficult gaming experience. We were also instructed to write an extension library to customise the maze generation according to difficulty parameters, which would be configured by the players' own personality. This aspect is determined by an entity that records and processes data about the players' actions.

In this final report, different aspects of the project are discussed, starting with a more detailed problem definition and an analysis of the tasks at hand in Chapter 2, as well as a list of requirements and our intended approach to the project. In Chapter 3, an extensive explanation of the design and implementation of this project is given, describing how we implemented this extension library and how we turned the antagonist into an actual artificial intelligence. In Chapter 4, a description of the final product is given and requirements are evaluated. Which requirements were met and which were not is discussed here. Chapter 5 focuses on the evaluation and reflection of the process over the duration of the project. Finally, this report gives a conclusion and looks into recommendations and possibilities for future work or extensions in chapter 6.
PROBLEM DEFINITION AND ANALYSIS

In this section we start with the problem definition, then we will analyse this problem and give a short description of the skeleton game we received in the beginning of this project. Finally we will list the clients requirements in order of priority.

2.1. PROBLEM DEFINITION
Divireo’s mission is to continuously look for novel ways to improve the technology currently powering the arts and entertainment industries. Eventually, they would like to achieve this goal by using Artificial Intelligence (AI) in the entertainment engineering field, in order to create personalised products that are accessible to everyone. For that reason they asked us to develop an intelligent game personalisation library for the Unity game development framework. This library will be used for a specific class of games, specifically games involving decision making and strategies to solve a large variety of puzzles. The main problem addressed in this thesis is: How can one use AI to create a personalized game that gets more challenging as time progresses?

In addition to the AI, the environment in which the player is situated should contribute to this challenge as well. Dependent on the environment, the player could spend significantly more time traversing it than he would otherwise, thus prolonging his game. This environment can be a challenge in itself to the player. The addition of variables for both the AI as well as for the environment creates a system which can be controlled better, allowing for experimentation of the difficulty it poses to the player.

2.2. PROBLEM ANALYSIS
It is important for a good game to be challenging, but at the same time it should not be too hard as the player might give up. Mihaly Csikszentmihalyi [1] described a mental state called the "Flow" in 1990. When a person is in this mental state while performing an activity, then they are fully immersed in a feeling of energised focus, involvement, and enjoyment of the activity. To achieve this, the balance between their skills and the challenge should be exactly right; tough but not too difficult to perform it successfully. Research has shown [2, 3] that the learning process is the most effective in this mental state. This insight can be used to optimise game-based learning. Therefore, this project not only has value for the entertainment industry, but can also contribute to create effective, personalised learning games for schools in the future.

For the game itself this means that, although challenging, it cannot be impossible. However, to determine which parameters make the game difficult and which make it impossible would require a more in-depth study. Not only is it difficult to determine how challenging a task is, it is also dependent on the player itself. Certain players will require a more difficult game in order to face the same challenge as others would. All in all, it is required that the parameters can not only easily be accessed and altered by the framework, but the framework also has to allow varying difficulties to accommodate for different players.

RESEARCH REPORT
In order to gain a better understanding of the problem at hand, we started this project with a two week research phase. During this research phase we looked into possible solutions for the main problem and its
subproblems. Additionally, background research was done on additional game elements and graphics. Our findings have been reported inside a research report, which can be found in Appendix H.

2.3. SKELETON GAME DESCRIPTION

At the beginning of the project we received the skeleton game PortalBug from Divireo. The given skeleton game is made in Unity, a cross-platform game engine developed by Unity Technologies, and contains a collection of mazes generated by an adapted version of Prim’s algorithm. Prim’s algorithm [4] is a greedy algorithm that constructs a minimum spanning tree for a connected weighted undirected graph. The player starts in the beginning of the first maze, and can wander through the maze as an explorer. The goal is to get to the end cell, located in the last maze. To travel between mazes the player has to go through portals which connect mazes to one another. Portals have been designed to connect all mazes within the collection, and both the player and antagonist can make use of them. All portals have the same colour. Additionally, potions and special cells are distributed among the mazes, and the player has the possibility to interact with them by picking up the potions or absorbing the special cells. Three different types of potions exist: Potions that temporarily increase movement speed, potions which heal the players’ wounds, and potions that render the player invisible. Special cells exist which can increase the players’ maximum health or paralyse the player for a few seconds. The skeleton game also stars an antagonist, which currently spawns near the end of the last maze. The implemented AI is very basic, as the antagonist moves through the maze randomly, or in the direction of the player when spotted. When the antagonist collides with the player, damage is dealt to each character.

2.4. REQUIREMENTS

The requirements for the final product were given to us by the client in MoSCoW prioritisation format [5]. We verified that the MUST requirements were attainable within the time frame given for the project and assessed the viability of the requirements. The final requirements were as follows:

MUST

• The antagonist makes intelligent decisions using the available data
• The maze changes based on the player’s strategy at one or more of the following:
  – At creation time
  – Dynamically while the game is being played
• The maze contains multiple portals on levels with more than two connected mazes
• Documentation must be accurate and extensive
• Tested to the highest extent possible
• Modular and extensible
• Loosely coupled to the Skeleton Game

SHOULD

• The user can view a log of the decisions made by the antagonist
• The user can influence the decisions made by the antagonist
• Extra game play elements, for example:
  – Moving fog, smoke, or poison clouds
  – Player strategies such as a sword or key
  – Bonus level
  – Collectible items

COULD

• The game has high quality graphics
• The game has a menu and other game play related UI element
3.1. **LibraRynth**

In this section the extension library will be discussed. This consists of both the maze generator as well as the player observer.

The extension library that was created for PortalBug was dubbed LibraRynth, a wordplay on library and labyrinth. This extension itself contains two separate modules. The first module is the maze generator. This module generates the maze according to the parameters given. The second module is directly related to the first module and allows for observing a player that traverses a maze generated by the first module.

### 3.1.1. Maze Generator

In this section, the generic guidelines for the maze generator will be discussed first. After this, the API and maze object will be discussed as well as the parameters that the API requires. Next the recursive strategy that is used to generate the maze will be discussed. The usage of preliminary work on the maze, the shape of the maze as well as the rooms and vortexes it can contain will be talked about. Finally, looping in mazes and the theory behind it will be explained.

The maze generator contains the extension-ready functionality to generate a random maze. This random maze can be influenced by specifying the parameters that the maze generator should use during the generation.

#### Guidelines for the maze generator

According to the product requirements and the expected requirements for the LibraRynth extension, the maze generator is expected to comply with the following requirements.

- **Loosely coupled**
  
  The entire LibraRynth library, and by extension the maze generator, must be loosely coupled. The goal of this kind of coupling is that the entire extension could be taken out of the project and used in any other project. Proper loose coupling would allow for the extension to be exported as a library and used by itself without being dependent on the rest of the code in PortalBug.
  
  All of the functionality within the maze generator relies either on itself or on the default C# libraries. Any project that will incorporate our code is expected to have access to the default C# library and therefore the maze generator can be exported as stand-alone functionality. The public API of the maze generator is restrictive and simple and allows for as little error as possible.

- **Seeded**
  
  The result of the maze generator should be deterministic if necessary. This means that, given specific parameters, if the exact same parameters are fed to the generator again then the generator should return the exact same maze. Since the maze generator must still be random, this means that the generator should allow for the framework to provide a seed which it will use for its decisions that rely on chance. Should the framework decide not to use a seed, then the maze generator should pick one itself at random.
Additional advantages of a seeded maze are that it is a lot easier to debug the maze and to see the impact of different parameters when a similar maze is produced.

**Randomization of choices**
Since the maze-generator is expected to generate unique results, it is expected to strive for near-fair randomness while generating a maze. In some cases, this tends to be more difficult and unreliable than in other cases. The importance and difficulties of this near-randomness is apparent while generating the rooms for the mazes.

It is important to note that the discussed randomness is not dependent on the random number generator that is used, but the way in which it is used to generate the results in the maze.

**Restricting parameters**
Since the library must be fit to be used as an extension, it is important that the errors that can be made by an outside party are as limited as possible.

One of these is the restriction of the API to a single class. The API that the maze generator uses should be as simple as possible. In this case, the actual function calls to the maze generator is limited to a single class. In this class there can be several methods that correspond with different means of generating the maze, but these methods must be overloaded.

The second is the parameter class that these public API methods take as parameter. These parameters must be instantiated by a possibly overloaded static method, each of which produce a complete and stable parameter object that is to be fed to the actual maze generator. This parameter object must also allow for easy modification of its values, so that the user can modify generated parameters at any point.

**Easily extended**
Should the maze generator need to be extended it should be easy to do so. The maze generator consists at current of several different parts that can act independently on the maze data. Certain parts will always be required (i.e. the initial generation of the maze), however these parts should also be easily swapped by other functionality.

Aside from the actual implementation, the parameter class would have to reflect this change as well.

**Extension API**
The public API that is used for the maze generator consists, in compliance with the guidelines, of a single class that contains the method call to the generator. This method is located in the LibraRynthMazeBuilder class. This method takes a LibraRynthParameters parameter object. As result a LibraRynth data object is returned.

**Maze builder class**
The extension API, complying with the guidelines, consists of a single class. This class contains two overloaded methods. Both these methods require the maze parameters and one of these allows for the user to submit a seed for the maze.

- static LibraRynth BuildMaze(LibraRynthParameters)
- static LibraRynth BuildMaze(LibraRynthParameters, Int32)

These are the only public methods to return an actual LibraRynth object.

**Maze parameter class**
The maze parameter class is instantiated by either of the public static methods defined below.

- LibraRynthParameters CreateParameters(MazeDifficulty, MazeBounds, MazeBuilderValues)
- LibraRynthParameters CreateParameters(MazeDifficultyHandling.DifficultyParameters, MazeBounds, MazeBuilderValues)

The first method will generate parameters with values according to the default values for the specified difficulty. This allows for quick and easy generating of parameters while still allowing the user to customise specific values in the parameter object. The second method allows for the framework to specify the values for the maze generator itself, rather than relying on default values. This allows for the framework to determine
the difficulty itself and to alter the parameters dynamically should the need arise. The MazeBuilderValues can, in both cases, be omitted completely. If this is done, the default values will be used. In most cases, this means that additional editing to the maze is skipped entirely.

**MAZE DATA OBJECT**
The LibraRynth data object that is returned allows for the user to request cell and wall placement. The class itself is abstract and cannot be instantiated directly.

To view the LibraRynth API, please refer to Appendix B.1.

**LibraRynthData**
A LibraRynth object is too restrictive to be properly used for the maze generator. The maze generator itself uses a subclass of LibraRynth to accommodate for its functionality. It was decided to use separate classes to ensure that the maze data provided to the player is immutable. The subclass mainly consists of various setter methods for the data that will be supplied to the framework and several helper methods that assist in the generation of the maze. Additionally, LibraRynthData contains the rooms and vortices that have been set for a maze. This class, like the LibraRynth class itself, is abstract to ensure that it is not instantiated directly.

To view the LibraRynthData API, please refer to Appendix B.2.

**MAZE GENERATOR PARAMETERS**
There are several parameters that must be provided to the maze generator, as well as some that are not obligated. The maze generator itself can take an integer seed value that it will use to generate the maze.

**Maze bounds**
The maze bounds is an obligated parameter that the framework must provide. The bounds given an indication as to how large the maze is. Since the stored base maze is always rectangular, the original bounds would always be provided as a combination of the width and height of the maze. This, however, held up badly when different shapes were introduced. Since all shapes decreased the total surface of the maze by making certain locations inaccessible, this could potentially cause a maze solution to be significantly smaller. An alternative to explicit bounds is to provide the implicit surface that should be strived to be achieved. This would require the actual rectangular bounds to be calculated depending on the shapes, but it would give every maze with a different shape a comparable solution size.

**Maze strategy parameters**
The following parameters are present and editable for the generation of the maze.

**Difficulty**
This indicates the default difficulty that was used to generate the maze. This is set to be unspecified when the parameters were not generated using a default difficulty.

**DifficultyParameters and MazeBuilderValues**
The DifficultyParameters are mostly based on preliminary chances for the maze that is generated, whereas the MazeBuilderValues are used for the actual construction for the maze. It consists of the following parameters.

**BasicFormProbabilities**
This item indicates the chance for specific shapes (forms) to be used for a maze.

**BasicRoomProbabilities**
This indicates the amount of rooms and vortices to be used in the maze and the percentage of space they should use in the maze.

**Momentum**
This is purely for the recursive maze generation strategy, although it should not be too difficult to be
used for other maze generation strategies. It embodies the chance that the pivot in the maze generator goes in the same direction as it used to. Any value larger than 1.0 indicates that a pivot prefers to go in the same direction (i.e. go straight), whereas any value smaller than 1.0 indicates that the pivot would prefer to go in a different direction (i.e. turn left or right). A value of exactly 1.0 means that the pivot in the generator will choose any direction with equal chance.

**Looping parameters**
The looping parameters indicate the parameters that should be used for the looping behaviour. If this parameter is not set (i.e. is null), then no looping will occur.

**Imperfection**
Imperfection indicates the amount of walls that should be removed. A value of 1.0 will remove all walls that are allowed to be removed, whereas an imperfection value of 0.0 will not remove any walls at all, effectively disabling the looping behaviour.

**Minimal amount of adjacent walls**
An integer indication of how many walls must be adjacent to any random wall for it to be considered removable.

**Minimal adjacency is bilateral**
An indication whether the minimal amount of adjacent walls has to be present in both directions of the wall.

**Corner counts as additional wall**
An indication whether 90-degree corners would count as an additional wall.

**Parameter restrictions**
At current there are certain restrictions for each parameter. Most of these restrictions are checked on initialisation of the parameters and an appropriate exception is thrown should the parameters be erroneous. Most of these restrictions are forced upon the framework by only allowing it to use the `LibraRynthParameters` class to generate a maze.

The following special restrictions exist for these parameters:

**Must be between 0.0 and 1.0**
This restriction exists for the `Looping Parameters::Momentum` parameter.

**Recursive strategy design**
The maze we use is generated using the recursive strategy design. This strategy acts much like a random walk algorithm. Contrary to what the name suggests, the actual implementation is iterative to lessen the memory footprint. This is a well known design that by default introduces very little bias in the maze.

**Recursive strategy for generating a perfect maze**
The classical recursive strategy starts by picking a random point in the maze. This location will be marked, whereas all other locations will start off unmarked. Every marked location is considered to be part of the maze solution.

The recursive strategy algorithm keeps track of the path it has taken so far. This path contains all locations in the order in which they were marked and will from here on be referred to as PATH. Initially, PATH contains

![](image.png)

Figure 3.1: Initial random point
only the first location which was chosen at random.
At every iterations in the algorithm, the last location in the PATH is requested. We will refer to this location as LAST. The LAST entry is indicated with a filled blue dot. The algorithm then lists all the available neighbours of LAST that have not yet been marked. We will refer to these as NEIGHBOURS. Except for the initial location in the first loop, any LAST can have 3 NEIGHBOURS at most. When the NEIGHBOURS for the LAST location have been determined, a random neighbour is selected to continue the PATH.

This continues for a number of iterations until LAST does not have any NEIGHBOURS left.

Once the algorithm is stuck, it will backtrack step by step until its LAST entry has NEIGHBOURS again. It will then choose one of these NEIGHBOURS and continue the path from this point.

The recursive generation strategy allows for easy usage of momentum while generating. Momentum refers to the additional chance that the generator will prefer to take a path in the same direction instead of going left or right. If the value for the momentum is positive, there will be a higher chance for the maze to continue straight. It is also possible to set the momentum to a value below 1.0, thus making the generator prefer to deviate from the straight path. In Figure 3.5 the green dot indicates the direction of the momentum, whereas the blue dot indicates a 90-degree corner to the right of the path. The momentum depends on the relation between the LAST entry in the path and the entry before that.
Eventually, the path will backtrack all the way to the initial location again and the length of PATH will be 0. This means that the maze can not be extended any further. If it could have extended any further, there must be any marked location that has an unmarked neighbour. If this would be the case, it would have been picked up by the PATH. Therefore, all marked locations in the generated maze are part of the maze solution, since they have been visited.

The completed maze is considered to be a perfect maze. “A perfect maze is defined as a maze which has one and only one path from any point in the maze to any other point. This means that the maze has no inaccessible sections, no circular paths, no open areas” [6]. This is desirable since it allows for post-processing to introduce circular paths in a controlled fashion.

**Our choice for this strategy**

Since it was preferred to incorporate a momentum in the maze, this was deemed to be one of the most fit generators. The recursive strategy does not have a notable bias in any direction, but still allows for the maze to be as random as possible. Its ease of programming, relative low strain on the system and versatility have made this the prime strategy to generate the maze.

Initially it was discussed that several strategies could be used to generate different mazes, and therefore it is relatively easy to replace the existing generator with another generator. In the end, however, the priority of these generators became too low and additional implementations never took place. Regardless, the design to allow easy addition of new generators still exists.

**Problems with incorporated rooms**

The recursive generator seemed like a perfect choice, except for a small issue in the classical design in regards to additional features that we wanted to add. Namely, the implementation of rooms and vortexes meant that they required to be connected to the maze solution. This meant they had to be placed in the maze before the generation strategy started, because otherwise it would possibly cut off locations in the maze solution and make them inaccessible.

One solution would be to take several starting points for the recursive strategy. These starting points are incidental to the rooms and vortexes that have been generated. Should no rooms or vortexes be instantiated, these starting positions would default to a single random location in the maze.

Each step would expand on these starting points, and the separate paths may merge only once. This strategy would result in a semi-perfect maze, where the imperfection is only existent in the rooms.
3.1. LiBrARyth

**Rooms and vortexes**
To better influence the difficulty of the mazes, the concept of rooms and vortexes were introduced to the maze.

**The use of rooms and vortexes**
One of the best ways to decrease the difficulty of the maze is to make the size of the maze solution smaller. This logically means that there are less locations that can be traversed and as such, make the path towards the end shorter. Instead of making the maze itself smaller, another way to achieve this is by creating open spaces surrounded by walls. These open spaces would only have 2 openings and its walls are clearly visible by the player. Not only does this reduce the maze solution size, since the player can oversee a lot of locations at once, but it also acts as an anchor point for the user. Since the room will have a certain degree of uniqueness, the player should be able to relate to it and find his bearings in the maze again.

Increasing the difficulty can be done with the use of a vortex. A user will try to remember his path by the direction he has been going. A vortex will spin from the outside to the inside, where a three-way split is present. The direction the user takes in this three-way usually does not relate to the direction where the user will be leaving the vortex, thus confusing the player[7]. A prominent problem with vortices is the amount of size they take. Whereas a room would suffice with 2x2 room, a vortex needs at least 3x3 just to exist as an actual vortex. In order to create a proper, challenging vortex its size should be even larger. This, however, proves problematic, especially with the relatively small mazes that have been used. Because of this, vortices are only recommended on larger, more difficult mazes that do not contain any other rooms.

**Placement of rooms and vortexes**
Since we want to emulate randomness in the maze, the placement of the rooms needs to be random as well. Vortexes must be near-square (can have at most a deviation of 1 between width and height), but a room can be any size as long as its width and height are larger than 2.

At first, the total available surface area is used to calculated the rooms and vortexes in the maze. Next, the size for each room is taken squared (which equals one side if the room would be square) and a certain degree of deviation that is determined at random is applied to the width and inversely to the height. Since a vortex must be near-square, its squared surface area is taken for the width and height.

A problem that occurs when placing multiple rooms is that once a room has been placed, the space where the other room can be placed has become more limited. This could cause certain rooms to not be able to be placed any longer, because they lack the space to do so. Since a random solution was preferred that would still allow for all the rooms to be housed, at current a brute-force method is in place. This method finds all combinations where the rooms can be placed in the maze and then picks a random one from this collection. This however requires more computation than initially expected. Please see Recommendation for improvement on the following page for a run-down of possible solutions to this issue.

Another problem is that placement of rooms could create inaccessible areas, as depicted in Figure 3.7.

![Figure 3.7: Room placement problem](image)

A simple solution to this problem is to require rooms to have at least 1 space both between each other and the outer wall of the maze. This way, there will always be space available for the maze and locations will not
become inaccessible.

**Implementation of rooms**
The implementation of the rooms is about as simple as the implementation of the original maze. All of the cells within are instantiated and only the outer walls are created. After this, two unique random outer positions are chosen for the two entrances to the room. Other than these entrances, a room will not have any other entrances or exits.

![Maze with room](image)

**Implementation of vortexes**
Vortexes are slightly more tricky, because they must to loop around. The vortexes that are created in the application always have 3 entrances. This is because having only 2 entrances does not significantly increase the difficulty because it does not present the player with a choice. Furthermore, when a vortex has 4 entrances it resembles a swastika, which was deemed unprofessional.
The vortex starts from the middle in a 2x2 area. It has three paths, whose lengths are respectively 1, 2 and 3 lines long. Each iteration, the shortest line is wrapped along the rest of the body of the vortex, thus extending it by looping around it. This continues until the entire vortex has been created in the assigned space.

![Maze with vortex](image)

**Recommendation for improvement**
As hinted at before, the current performance of the maze builder takes a major hit in this section. This is because of the brute force algorithm that searches for all the possible combinations of the rooms in the maze and then picks a random combination to use. It does so by matching it with all the locations that are available in the maze. These locations are stored in a separate HashSet. Although requesting the locations from the HashSet is relatively fast, the combination of having to ask for all of them each time is tedious and takes a toll on performance.

A solution to this bottleneck is to place rooms at random and retrying placement a certain number of times before conceding. This would be the fastest solution but it would not produce complete solutions at certain times, because it might be unable to find a valid placement for the multiple rooms.

Another more sophisticated solution is to calculate squares in the maze and use a collection of squares for the placement of the rooms. If all squares are calculated that are at least 2x2, these can be used to find proper placement. This solution would still rely on brute-force search of all combinations, but would make the algorithm approximately $\ln(n)$ faster by not requiring every single location to be looked up when checking for valid placement. In several cases, for example a rectangular maze, this method would speed it up by about $n$. This would be slower than the previously discussed solution, but would still retain the randomness that is desired as well as a far better guarantee in being able to fit all the rooms in the maze.
SHAPE OF THE MAZE
In this section, the shape of a maze and its implications on the difficulty will be discussed.

Organic shapes in natural mazes
One of the most important things in solving mazes is orientation. If a player knows his approximate location, he will be able to estimate where he needs to go to solve the maze. This has already been discussed for the rooms in a maze that act as an anchor point for the player. For a square maze, the player can approximate its position better when it encounters one of the outer walls. If he cannot go further, the player knows that the rest of the maze must be perpendicular to this wall.

If we introduce different shapes, this would not necessarily be the case. The outer walls would not be smoothed out, thus not revealing to the player that it is in fact one of the outer walls. Furthermore, odd shapes can throw the player off because it expects the surroundings to be different. A more experienced player may partially see past these obfuscations, but the added difficulty in navigation remains.

Square maze
The default maze that most player will expect is the rectangular maze. This maze is the easiest to traverse. Because of its nature, the path near the outer walls will usually be more or less smoothed out. This is because it will only allow an incoming path to go left or right, the path can't go outside of the maze. Because of this higher likelihood of a path hugging the outer wall, combined with a possible momentum in the generator, the outer walls will most of the time be easily recognizable.

Circular maze
At current, there only exists one additional shape, namely the circular maze. As discussed earlier, its more irregular shape than the rectangular maze combined with the expectations of most players will cause a certain degree of confusion for players traversing it. Player may have difficulties recognising the maze corners, as well as erroneously take a random straight wall for the outer bounds. Its added confusion and obfuscation will make traversing the maze more challenging to the player.

Ease of extension
The extension contains several subclasses for LibraRynthData that correspond to different shapes of the maze. Addition of extra maze shapes is easily done by creating new classes and altering the chances for the specific shape in the parameters. These classes are the actual maze objects that are instantiated.

The shape of a maze is determined by the overridden abstract methods.

LibraRynthData generic subclass API description:

bool CellLocationIsAccessible(Int32Point coordinates)
Method that must be overridden from the LibraRynth superclass. This method determines whether a cell location is accessible and as such valid for maze traversal. The maze will not be generated through cells that are inaccessible.

By defining the proper methods and making them scale with the maze, it becomes possible to incorporate additional shapes into the maze generator. Though the underlying base maze will always be rectangular, the accessibility of locations can force the generator to skip these, effectively creating the desired shape. An ill-defined subclass may leave cells unfit for the maze path. This happens when there are different groups of accessible tiles that are entirely separated by inaccessible tiles. It should be noted that even if a location is accessible, it is not necessarily instantiated or part of the maze solution. To check whether a location is part of the solution, the IMazeCell GetCellAt(Int32Point coordinates) method should be relied on.

IMPLEMENTATION OF RECURSIVE STRATEGY
This section will discuss the order of the implementation and the implications that are caused by the different sections of the generator.

General order of implementation
When the maze is generated, it must first choose one of the basic shapes. The shape it chooses is dependent on the parameters that have been supplied by the parameters and will default to a square maze if no proper parameters have been supplied. After the shape has been chosen, the rooms and vortexes must be initialised and placed inside the maze. These rooms and vortexes will result in a collection of starting points. Each of
these starting points will have a unique identifier that corresponds to the room. This will allow the rest of
the generator to link these together while still allowing for a perfect maze outside the rooms. Now, the actual
maze generator strategy, the recursive strategy in our case, can actually start expanding. The starting points
don't necessarily have to be used as actual starting points, but they must be connected for all the rooms and
vortexes to be accessible and interconnected.
After these steps have been completed, the raw maze is ready to be returned to the framework. Additional
actions may be performed on the maze. One of these actions which has been implemented is the generation
of loops in the maze.

Change in original design
The original design included the rooms and vortexes, but did not go into enough depth to realise the prob-
lem of them having to be connected while still generating a perfect maze outside of the rooms. This problem
becomes most apparent for rooms that share a common starting point. These points are interconnected by
default and this must be taken into account as well during the generation of the maze.
In the original design it was the intention to create several strategies to generate the maze. The choice be-
tween these strategies would supply an additional parameter to the maze, thus making it more customisable
and allowing for more possibilities. Although the generator has been build to support such an implementa-
tion, there was not enough time and priority to implement this functionality.

Reflection on complexity and performance
The performance on the generation of rooms has already been reflected on, therefore we will now focus on the
performance of the recursive strategy. The big-O notation will be used to signify the worst case performance
of the algorithm.
At the start of the algorithm, we loop through all starting positions, marking duplicate starting positions to
become a dead end.
\[ O(o) \text{ where } o \in \text{starting positions} \]
After this, we keep looping through all the paths, either extending them if possible or stepping back if they
cannot be extended. For convenience, we will assume that only 1 random starting point and no rooms or
vortexes are used to generate the maze. Each tile in the maze will be visited once, so for a \( m \times n \) maze this
means \( O(m \times n) \). The additional computing time comes from retracing the path to former points that are
already present in \( \text{PATH} \). Every single location in the maze is expected to be retraced at most three times.

Proof:
Take the last location from a random path during the calculation of maze via means of the recursive strategy.
This random location can have 2 states:
- State 1: It has unmarked neighbours that it can extend to
- State 2: It has no unmarked neighbours

We assert that the specific location can only be at the end of the path for only 1 very specific path. This is
because each tile is only marked once and not visited again by any other path after it has been marked. Fur-
thermore, the premise of the maze being a perfect maze already states that there is only a single path between
any two random locations, which in this case are the starting tile and the current location that is being con-
sidered. As such we can be sure that only one specific path exists with this location at the end.
In the case of state 1, it still has neighbours and as such can still expand into it. Except for the initial lo-
cation that has four neighbours, there must be at least one marked location adjacent to the current location.
This location is part of the path from the starting location to the current location. This means that the cur-
rent location can expand into at most three other adjacent locations. As such, it will only be retraced at most
three times, once for each direction. In reality, the amount it is retraced will on average be significantly less,
because existing paths usually disallow a location to expand into all directions.
In the case of state 2, the random location has no neighbours left and will be removed from the path. After
it is removed, it cannot be made part of another path since its location has been marked. Therefore, it will
only be the last location at most four times in any path. It will be the last location when it is initially added
to the path and it can be the last location three additional times when each of the adjacent locations it has
expanded into is being retraced back to the current location.
In conclusion, each location is retraced at most three times when the maze is calculated via means of the recursive strategy. This is proved by the fact that any location can only be the last location in any PATH at most four times.

Q.E.D.

This means that not only will every single tile be added once to any path, it will also be removed only once. Since the algorithm ends when all tiles have been retraced back to the very start, the final complexity of the algorithm is $O(3 \times m \times n) \times O(m \times n) = O(4 \times m \times n)$.

All of the data sets that are used in the algorithm are expected to return a call in $O(1)$ time (i.e. HashSet, Dictionary) or are iterated through, giving no additional penalty to the theory explained above (i.e. List). Retrieving the neighbours for any given point happens in $O(1)$ time as well. Therefore, the expected complexity of the recursive maze generator is expected to be:

$$O(m \times n) + O(o)$$

Since $o << m \times n$, the final expected complexity would be

$$O(m \times n)$$

**GENERATION OF LOOPS**

**Wall removal from the original maze**

![Figure 3.10: Completed](image)

Since the completed maze is perfect, as depicted in Figure 3.10, there will always be only 1 path from any random point to any other random point in the maze. This means that if you were to remove a wall between two marked cells, you would create a short-cut between these two locations. As such, this short-cut will always result in at least 1 additional path (namely the short-cut itself), thus making the maze imperfect and introducing loops.

**Proof:**
Remove a random wall anywhere in the maze between two marked locations. We will name these locations A and B.

There will be at least one unique path between the two points. This path has to exist since both locations were marked. This path also has to be longer than 2 tiles, since there is a wall between these tiles, disallowing direct traversal between them.

Once the wall is removed, it will generate an additional path between these two points. This will result in the creation of a loop, since there are now two ways to travel from location A to location B.

Q.E.D.
3.1. LIBRARY

![Diagram](image)

(a) Removing a wall  
(b) Creation of a cycle

Figure 3.11: Representation of how removing a wall guarantees the creation of a cycle

Expected impact of parameters

An old trick to solving ‘any’ maze is to always touch the wall of a maze with your left hand and keep walking. This will ensure you find your way to the exit eventually. However handy this ‘rule of thumb’ appears to be, this only always works for perfect mazes. It could work for non-perfect mazes, but only if you are not touching one of the walls that are not directly connected to the wall that is next to the exit.

In the case of wall removal, we create ‘islands’ in the maze that make it imperfect. These can cause the game to both become easier as well as more difficult depending on placement and size.

There is an increase in difficulty for the maze when a limited amount of walls are removed. If too many walls were to be removed the maze would become too transparent as it would allow the player to possibly look past several layers that are now open, thus possibly speeding up his solving time far too much. Since there are loops, this trick will not work in all cases any more and the maze would become more challenging in general.

At the same time, if the player is in close proximity to the antagonist of the game, the alternative routes could allow the player to avoid the antagonist altogether. Simultaneously, the likelihood of getting stuck in a dead end when the antagonist is chasing the player become less likely as well. For this reason, some imperfection is recommended for any difficulty.

When the direction of the adjacent walls are not bilateral, this would make the removal of walls very random. The randomness of this action is expected to leave the maze a lot messier and with fewer dead ends that could be used as reference. It is therefore recommended to only use this on higher difficulty settings.

The amount of adjacent walls that are used are very dependent on the difficulty setting. In general, a smaller value will amount to more randomness in the maze, whereas a larger amount will appear to be better structured. The amount of adjacent walls and the imperfection of the loops rely on one another, and their combination must be considered to give a proper reflection of the difficulty it is trying to represent. In general, a higher adjacency combined with a higher imperfection is advised for easier difficulties, whereas a smaller adjacency with a sparser imperfection would be more fit for a higher difficulty.

Whether corner walls count as additional walls is a setting that can be used to make the maze less see-through. It is advised to keep this turned on for larger adjacency values and to turn this off when the adjacency is quite low.

3.1.2. PLAYER OBSERVER

The PlayerObserver is one of the most important features the client desired, and its completion was critical to the success of our project.

The task of the PlayerObserver is to monitor the players actions when moving through the maze and gather statistics on the players’ behaviour. These statistics should then be stored for future usage and retrieved by other parts of the code for decision making purposes.

The PlayerObserver monitors the following statistics, which can be requested:

CellsVisited

This monitors the number of cells visited. Specifically, this number increments whenever the player moves between cells.

CellsRevisited

This monitors the number of cells revisited. This number only increments whenever the revisited cell was not among the last 4 cells visited. This is because this metric is intended to be used to determine how
‘lost’ the player is, as the player is likely to visit the same area numerous times when lost. Hence, minor backtracking or circling the same spot should be ignored.

**Surface**

Though not an actual statistic, knowing the surface area of a maze is important for a great deal of calculations, and hence is included in the metrics returned.

**SolutionTime**

This tracks the number of ticks have occurred while the player has been present in this maze.

**PercentageExplored**

This tracks the percentage of cells that have been explored.

**DirectionalBias**

This contains three values, a left, right and straight bias, which track the user's preference in their respective directions when faced with a junction which features that direction as an option. For example, if the player has a bias towards the left of 0.75 then the player has a 75% to go left when a junction features a left corner. Only unexplored directions will be taken into consideration, as otherwise the bias values would naturally trend towards equal values.

Additional PortalBug-specific statistics exist, however considering the number of statistics that were applicable to any maze we decided to implement the majority of the PlayerObserver's functionality inside the LibraRynth extension. This was not included as a requirement of the client, however we decided to implement it like this so that it may be used freely in other projects like the Maze Generator. For the PortalBug-specific statistics, such as the interactions with the antagonist, there would be an additional class in PortalBug which extends the PlayerObserver inside LibraRynth.

**Decision: Receiving Data**

One of the major points when developing the PlayerObserver was how player data should be received. Initially, it was decided that this would be done through a delegate method. The way this would work is that the PlayerObserver would use Unity's built-in `Update()` function to update its information every game tick using the given delegate to retrieve required information. This means that the user effectively only has to initialise the PlayerObserver with the correct arguments and it would run on its own afterwards. This approach would result in fewer possible errors by incorrect usage on the user-side.

However, using Unity's built-in `Update()` function required the PlayerObserver to extend `MonoBehaviour`, Unity's standard 'Game Object' class. This, however, caused various errors to appear during standard Unit Testing. The solutions to these problems were complex, such as overwriting `MonoBehaviour` or changing our existing Unit Tests into Unity Unit Tests.

We instead decided to drop Unity's built-in `Update()` function and require the user to call the PlayerObserver's `Update()` function. While this requires some more user-interaction, it allows LibraRynth to be used outside of Unity projects, as well as give more control to the user on how often and when LibraRynth updates. Given these benefits, it was clear to us that this was a better option.

**Challenge: Directional Bias**

One of the features the client was very interested in was the idea of tracking player actions in regards to turns at junctions. This knowledge could be used by the antagonist to cut corners or devise short-cuts, and could also be used by the maze to strategically place powerups. Though both the antagonist and powerups are unique to PortalBug, tracking directional bias by itself is not. Therefore, we decided to implement this feature in the external LibraRynth library.

Detecting junctions isn't as straightforward as it may seem, however. At first glance it may seem sufficient to simply determine if the player can move in more than 1 direction at any given cell (ignoring the direction which the player originated from), but this would cause rooms to also be regarded as a junction. Likely, the movements of a player in a room will be based on what is inside the room or where the exit is, we therefore need to filter out these cases.

No existing research or other sources of information exist on this subject, and therefore we needed to come up with our own solution. To that end, various junctions were made in order to determine how to best recognise junctions. Figure 3.12 shows a number of different junctions that were created for this very purpose. Similarly,
these cases are all covered in our Unit Tests. Furthermore, note that Figure 3.12f is not considered a valid junction as the room should be ignored for directional bias.

(a) Standard-X junction  
(b) Standard-T junction  
(c) Corner-integrated room door  
(d) X-junction without corner walls  
(e) Shortened-T junction  
(f) Room junction

Figure 3.12: Various possible junctions. Note that 3.12f is NOT considered a valid junction for data collection.

The way that this problem was solved was by determining for each corner-point of a cell if there was a wall connected to it, as displayed in Figure 3.13. If this is true, then we know it has the minimum amount of walls required to be considered a junction. We then determine the amount of directions (North, East, South, West) the player can head towards, and if this number is greater than or equal to 3 then the current cell must be a valid junction for determining directional bias. All examples in Figure 3.12 can be correctly identified using this method. Minimalistic cases can also be identified correctly, such as X-junctions composed of only four walls or T-junctions composed of a mere three walls.

3.2. PORTALBUG-LIBRARYNTHERY INTERACTIONS

3.2.1. MAZE GENERATOR
PortalBug’s interaction with the Maze Generator is clean and simple. When a game is started PortalBug requests the required amount of mazes from the Maze Generator, and then converts these to actual game objects. During this conversion it instantiates the proper cells, walls and other game objects (such as portals) as Unity 3D objects. Furthermore, the received maze data object is then forwarded to the PlayerObserver by PortalBug so that it is aware of the structure of the mazes.

3.2.2. PLAYEROBSERVER
The PlayerObserver is extended inside PortalBug, tracking additional PortalBug-specific statistics. This extension also manages the forwarding of data to the base PlayerObserver. Furthermore, the extension is in charge of reading, writing and loading stored statistics files. Once the data has been loaded other classes can make use of this data. For example, this data is used inside the PlayerParameterParser to adjust difficulty settings according to the data collected. This is done by determining how much the collected statistics differ from the expected norm, and based on this returns a deviation $d$ where $-1 \leq d \leq 1$. Currently this deviation is determined based on how long the player took and how many already visited cells were revisited. If these numbers are high, it is likely the player got lost frequently, and the difficulty should be adjusted.

Because the adjustments to difficulty are made purely based on deviation it is a simply matter to expand this in the future. Additional statistics can simply be added to the weighted deviation calculation without having to touch any other code.
3.2.3. **ANTAGONIST**
The antagonist is largely separate from the LibraRynth library. All interactions in terms of maze generation and construction are handled solely inside the Maze Generator. No interaction is required on the side of the antagonist.

The antagonist does, however, have (partial) access to the statistics collected by the PlayerObserver. It is aware of the directional bias the player possesses, and will use this to predict the players path. For example, in the event that the player becomes invisible, the antagonist will attempt to predict the players’ movement using the directional bias that the player possesses.

3.3. **LEVELGRID GENERATOR**
The mazes in the skeleton game were comprised of a pre-determined number of mazes in a linear layout. One of the larger tasks of this project was to make the maze layout non-linear. In order to achieve this, we needed to generate a graph of mazes, or simply nodes, and connect them accordingly with edges. A start and end node should then be selected based on which nodes are furthest apart, to ensure that the player must traverse a minimum number of mazes before reaching the end.

Figure 3.14 shows an example of the layout that we desired to achieve. Algorithms to generate random graphs exist already, however a problem with these existing algorithms quickly became apparent. It is hard to control the minimum distance from start to end node using standard graph generating algorithms. Even in randomly generated graphs with as much as 12 nodes, which is roughly the maximum number of mazes that would ever be desired in a single level, it was entirely possible that the farthest points were only 2 or 3 edges away from one another. As such, existing algorithms proved to be ineffective, and a custom approach was required for the gameplay that was desired.

Figure 3.15 shows how Figure 3.14 can be represented in this grid structure.

**ADDITIONAL METHODS**
Figure 3.14 and 3.15 depict a start and end node, colored green and red respectively. LevelGrid has a method to determine these start and end nodes using two Breadth-First algorithms.

This starts by selecting an initial node at random and determining the node(s) farthest away from this initial node. A random node is selected from these farthest node(s) and afterwards it is determined which node is farthest away from it. This ensures sufficient distance exists between the start and end nodes.

Finally, the LevelGrid also features a method that can be used to set the desired level of sparseness. A sparseness value of 1 means that all possible edges have been created, while a sparseness of 0.5 means half of all edges have been created. By setting a fairly high target sparseness value non-linearity is ensured.

**OPTIONS**
The LevelGrid features a number of additional options that can be set to change the characteristics of the produced grid.

**Structuring element**
Originating from mathematical morphology. Currently, only N4 and N8 are supported, but the system has...
been designed that expansion to larger structuring elements is feasible. Typically, N4 is used only on lower
difficulties, as it is prone to creating simpler layouts.

![Structuring elements](image)

**Figure 3.16: Structuring elements currently available for use in LevelGrid**

**Grid size**
By limiting the height and width of the maze rare occurrences where the random walker continues heading
into the same direction can be avoided. This is a useful setting for further ensuring non-linearity.

**Maximum number of edges**
This setting controls the maximum number of edges allowed for each node. This maximum number of
edges is taken into account accordingly when setting a target sparseness value.

**Auto connect**
During the Random Walk algorithm it frequently occurs that the random walker walks to already visited
nodes. Using the auto connect option it can be controlled whether an edge should be created when this
occurs or not. It is recommended to leave this option disabled for numerous reasons:

- Leaving this option disabled results in LevelGrid generating a Minimum Spanning Tree.
- Enabling this option can create very densely connected nodes in the corners of the grid due to the
  nature of the Random Walk approach, while leaving other areas at a relatively low sparseness.
- In rare cases, in combination with the 'Maximum number of edges option', it can cause the node
  creations to fail. This is because it can create a situation where each node has already reached its
  maximum number of edges before the desired number of nodes has been reached, as depicted in
  Figure 3.17. Though the LevelGrid will handle this event accordingly and will not cause a crash or
  infinite loop, it is recommended to avoid it for simplicity, in addition to the previously named reasons.

![Grid size example](image)

**Figure 3.17: Given 'Maximum number of edges'=3 it is impossible to add any additional nodes to this existing grid.**

### 3.4. Antagonist Behaviour

#### 3.4.1. RainAI
RainAI [10] (Rain) is the library that was used to design the antagonist’s behaviour. This library was designed
for Unity, to create interactive characters. In order to incorporate Rain into PortalBug, we needed to attach an
AI Rig to the antagonist prefab. This component ensures the ability to set up motor functions, intelligence,
memory, and basic senses.

The antagonist makes use of a basic motor in order to move. It was also given a basic sensor, namely "eyes",
to be able to see the player. To determine which actions to take, we created a behaviour tree using the Rain
behaviour tree editor.
3.4.2. **Navigation**
The motor functions from the antagonist make use of a game object provided by Rain, called a navigation mesh, in order to determine which path can be walked. Normally speaking, the generation of the navigation mesh is done automatically by Rain. The algorithm Rain uses to generate a navigation mesh makes use of flat surface detection, to determine which areas in the world can be walked. However, this navigation mesh does not update itself when the terrain changes at runtime. As our mazes were being generated at runtime this meant that this automatic generation did not work for us. As a result of that, we had to force Rain to generate the navigation mesh during runtime.

This was achieved by accessing Rain’s methods for generation of a navigation graph, which determines where the navigation mesh will be. Prior requirements needed to be met before these methods could be successfully executed, however, which was not specified in the Rain API. In Appendix F, we describe the problems we encountered with this navigation mesh generation and how we solved them.

3.4.3. **Player Detection**
To detect the player, the antagonist makes use of its basic senses. We attached a visual sensor which acts as a pair of eyes, from which a range and angle of vision can be set. The possibility to see through walls was disabled, to make it more realistic.

To notify Rain that the player is a detectable character, an Entity Rig component needed to be attached to the player prefab. This way, when this Entity Rig comes into range of the sensor, Rain automatically knows to save it into the antagonist’s memory.

Saving the player in the antagonist’s memory upon detection allows for easy manipulation of this data, because the player then becomes easily accessible.

3.4.4. **Behaviour Tree**
The behaviour tree we created is comprised of four main elements. The antagonist can explore the maze, pursue or flee the player if in sight, it can choose to go through portals, and react upon certain variables in the game.

In order to set up this behaviour tree, action and decision nodes are available. Action nodes tell the AI Rig to execute a certain task, while decision nodes force a choice of action, based on given variables. What we made the most use of were custom action and decision nodes, which are actions and decisions determined by code.

Rain automatically reads in the behaviour tree and execute its content for us. But the hard work resides in implementing the custom actions and decisions. Each action and decision had to be programmed through code and required logical thinking in order to be done properly.

The antagonist’s basic behaviour was configured through the following behaviour tree.
Walking through the tree with a breadth-first approach, two nodes are first executed sequentially. First, the antagonist waits a second before jumping to the next node, which is a custom decision that determines whether the player is ‘spraying pheromones’ or not, and whether the two characters are in the same maze. If so, the antagonist will go towards the player else, it will adopt the normal behaviour.

The detect node at the bottom uses the antagonist’s visual sensor to detect the player “Entity Rig” component. The two ‘PlayerDetected’ and ‘PlayerNotDetected’ nodes are constraints that are executed in parallel. If the player is detected, the antagonist goes into pursue or flee mode, where it will decide if it should flee, or pursue the player. If in pursue mode and the player goes invisible, it will attempt to predict the player’s direction. If the player is not detected and the antagonist’s health is still high, it will normally explore the mazes. However, if the health is low, it will first heal itself using its healing abilities.

**EXPLORATION**
The antagonist’s exploration feature was implemented through code, and occurs when the latter has nothing else to do. To optimise this exploring behaviour, we added a list of maze cells to visit in the antagonist class. This list is composed (at the start) of all the maze cells contained in the maze which the antagonist currently resides in.

The antagonist will move towards the closest nearby unexplored cell, or a random one if multiple are equally close. Rain then uses the basic motor to move towards it. By making use of the already available script to determine the antagonist’s current cell, each visited cell is removed from this list.

Once the antagonist has visited all the cells on its list, it will decide to go through a portal.

**PURSUE OR FLEE**
When the player is in range, the antagonist has two possibilities: pursue or flee. Its choice is determined by its chance of winning, which is calculated by taking into account and comparing its health and damage quantity, to the player’s health and damage quantity. It hence calculates which of the two actors has the highest chance of winning.

If the antagonist has a higher chance of winning the battle, it will go into Pursue mode, which tells the basic motor to move towards the player.

If the antagonist has a higher chance of losing, it will go into Flee mode. Whenever the antagonist is exploring,
it will record intersections (maze cells with less than two walls) and save them into a list in its memory. When in fleeing mode, it will run back towards an intersection. This way, it can make a choice to take a turn at this point and lose the player, just in case the player is pursuing it.

PORTALS
During the exploration phase the antagonist also records where portals have been seen. It maps these portals to their respective maze cells for each maze using a Dictionary, just like the other Dictionaries. When there are no more cells left to explore, the antagonist will choose to go through the nearest portal. If the antagonist is pursuing the player, and the player jumps through a portal, it will follow it through.

OTHER ELEMENTS
There are three other variables in PortalBug which the antagonist can react to.
The first thing it will react to is when the player uses ‘spray pheromones’. This ability allows the player to summon a little green cloud which serves as a marker. The antagonist can detect that cloud, however. If a cloud is being sprayed in the same maze as where the antagonist is, the latter will move towards the cell where the pheromones are located.
Secondly, when the player picks up an invisibility potion, it will turn invisible. If the antagonist is pursuing the player at this point, it will make use of the PlayerObserver, as mentioned earlier, to determine the most likely direction in which the player went.
Potions can also be picked up by the antagonist. A basic potion behaviour was implemented, making use of a pick-up-when-seen strategy. When the antagonist is low on health, it will pick up a health potion if it sees one. When the antagonist is in Fleeing mode, it will pick up the closest invisibility potion it can find, and when in Pursue mode, it will pick up a speed potion, to move faster and catch the player.

3.5. Power-ups
In order to offer the player more game strategies, two power-ups for the player were added: a sword and a shield. One of each will be placed randomly in the level.
When the player possesses the sword, their attacks will do more damage to the antagonist. The shield will, while possessed by the player, protect the player and lower the damage of the attacks done by the antagonist.
If the power-ups would only benefit the player, the power-ups would not result in additional player strategies as the player would simply pick up everything that crosses their path. To make sure that the player has to think about whether or not to pick up a power-up, downsides were added as well. Both the sword and the shield will slow the player down, which can be inconvenient when the player still has to walk a great distance to the end or, at times, when pursued by the antagonist.
The option to drop the power-ups back on the floor was also implemented. This way the player can drop the power-ups when not wanted any more, or place them somewhere with the intention to pick them up later. A power-up for the antagonist was also designed, but due to time restrictions it has not been implemented. The idea was to give the antagonist the ability to place spiderwebs in the maze which would serve as a sort of alarm. When the player would walk into such a spiderweb, the antagonist would immediately know the player’s location. This power-up would have been a good opportunity to show the intelligence of the antagonist, as it should place them more and more efficient.

3.6. Further Modifications to Skeleton Product
Besides the above-mentioned, some other modifications to the skeleton product were made in order to improve it. These modifications will be discussed in this section.

3.6.1. Revamp of Code Smells
The original product code has been partially rewritten to better fit the theme of the C# style and to help prevent possible bugs in the future. Many of these changes were done because the original code was written in a more Java-style approach due to the original creators’ programming experience. Things such as not using auto-properties, usage of the wrong or inefficient data structures and usage of public variables were modified. The initial revamp did not change anything to the code behaviour and would merely prevent errors related to
these code sections in the future.

### 3.6.2. Preparation for LibraRynth Extension

In order to prepare the original framework for the extension, the original implementation of the maze generator had to be taken out of the framework itself and placed separately into the extension. This would first require to have the proper extension API prepared, so the maze could immediately be called and converted to the proper format for the framework.

A similar extraction was done for the artificial intelligence for the antagonist. However, some time into the project it was decided that the RAIN AI was to be used for the implementation of the artificial intelligence. Because of this, the extension for the artificial intelligence was made obsolete and as such, removed from the extension entirely.

### 3.6.3. Audio

Since the theme of the game a horror-esque game, we needed to have some proper horror characteristics in it as well. Horror is most apparent in the atmosphere of the game. A tense or desolate atmosphere aids towards the user feeling alone and in danger. One of the prime ways of creating an eerie atmosphere is to use music and sound effects at the right moment. The right sound effect at the right time will effectively act as a jump scare and a creepy sound track will make the user more tense while playing the game.

For the main track an ambient sound track was used that is designed to make the player feel uncanny as he is navigating through the mazes. There are several sound effects that are used, for example when the antagonist sees the player and when the antagonist is in close proximity to add to the horror atmosphere.

### 3.6.4. Health Bars

To give the player more insight as to what is happening during the game, it was decided to display the health of the player and the antagonist via health bars on the screen. As shown in Figure 3.19, the colour of the bar gradually changes from green to red as the player loses health.

![Health Bar](image)

(a) Full health  
(b) Damaged  
(c) Almost dead

Figure 3.19: Health bar of the player

### 3.6.5. Graphics

Additionally, a number of improved graphics were added. The basic models for the sword and shield were obtained from the 3D model download website http://tf3dm.com. The textures were made by us and added with Blender. The models for the player and the antagonist were created using the tablet application ‘123D Sculpt+’ and modified and coloured with Blender.
3.6. **Further Modifications to Skeleton Product**

(a) Antagonist

(b) Player

Figure 3.20: Characters

(a) Sword

(b) Shield front

(c) Shield back

Figure 3.21: Powerups
Final Product

4.1. Description
Having come to the end of this project, the product has been finalised. The final product is comprised of a complete maze generator and an extensible PlayerObserver, which form the LibraRynth extension; as well as an advanced behaviour for the main Antagonist of the game, and finally additional gameplay elements that were added by the team.

4.1.1. LibraRynth
The LibraRynth extension exists of two parts: the Maze Generator and the PlayerObserver.

Maze Generator
The maze generator generates mazes at the start of the game, based on values from the PlayerObserver and difficulty parameters. If the player appears to be having a tough time, or easily solving the maze, the game will adjust accordingly. An example of an easy maze is a maze with wide open spaces, which can be used as a point of reference to help the player navigate the maze. A harder maze can contain vortices, which rotate the direction the player is heading in to throw the players sense of direction off.

PlayerObserver
The PlayerObserver is the learning component of LibraRynth. It records how long the player spends in a maze, how much of a maze is explored by the player, how frequently the player revisits a maze cell (which indicates how lost the player is), which direction is preferred by the player, and how the latter interacts with the Antagonist.
These observations are saved onto the user's disk, and are used the next time the game is (re)started.

4.1.2. PortalBug Extended
LevelGrid Generator
The LevelGrid generator generates a grid of mazes, interconnected with portals. It can create both linear and non-linear paths across mazes, and is also capable of creating dead ends; portal that lead to a maze that is not connected to any other. It also determines the start and end mazes by determining the mazes that are farthest apart from one another. At harder difficulties additional mazes are created and a dense grid of portals is generated, creating a complex network that is difficult to navigate.

Antagonist Behaviour
The Antagonist now has a more complex behaviour. It will explore the mazes and decide when to enter portals. It can also use potions to its advantage and decide whether or not it is a good idea to attack the player. When the Antagonist believes that it is going to lose the fight, it will pull back and heal itself before attacking the player again. It can also access the PlayerObserver to learn more about the player. The behaviour tree that was built relies upon this knowledge to evaluate and take the most appropriate decisions.
Powerups have been added to PortalBug. The player can make use of a sword, which increases damage, and a shield, increasing the players’ resilience to damage. These power-ups appear once in each game, and are scattered throughout the mazes. Furthermore, the code of the skeleton game has been partially rewritten and audio, health bars and graphics have been added as well.

4.2. Evaluation Requirements

In this section, we will look at the requirements from section 2.4 and discuss whether we have met these requirements.

4.2.1. Evaluation Requirements: LibraRynth

For the LibraRynth extension the client required it to be loosely coupled, and thus reusable as a library. It should be possible to remove it from of the project entirely and use it in a different project. The extension is implemented as if it was a library. It has a public and simplistic API and it only relies on its own code and the default C# libraries. It does not require the framework or download any additional libraries in order to function. The mazes should be generated and be dependent on the behaviour of the player. The requirements stated that this should be done either at creation time or dynamically while the game was being played. Due to lack of time it was eventually decided to store the player data and load this at the start of the game again. This player data would then be used at the start of the game to generate all the mazes in the game. A future update could make the framework generate the mazes dynamically per maze, since both the storage of mazes as well as the maze generator allow for such an extension. A maze can contain multiple portals on levels with more than two connected mazes. The portals in a maze are reflected by the connectivity between mazes. This connectivity is generated and managed by the Level-Grid functionality that is located inside the framework. Since the LevelGrid functionality allows for multiple connections from any maze to other mazes, the portals are present in these mazes as well, allowing for multiple portals in each maze. The documentation had to be accurate and extensive. This is especially important in the extension, since it will possibly be used for different projects as well. The documentation for the public API is complete and extensive in both cases, and the documentation for the internal methods are, for the majority, accounted for. The LibraRynth extension is tested as well as time would allow. Although not a complete 100% was tested, the most crucial and bug-prone code has good test coverage which will detect the introduction of any bugs in future development. The LibraRynth implementation should be modular and extensible. In the case of the maze generator, the different components are loosely coupled and easily replaceable by different but similar functionality.

4.2.2. Evaluation Requirements: Antagonist Behaviour

An important part of this project was to develop an intelligent Antagonist. Part of the client’s requirements was that the Antagonist could make intelligent decisions using the available data, and that the user is able to influence the Antagonist’s behaviour. In the delivered product, we have set up a basic behaviour for the Antagonist which can be easily extended. Using the RainAI library, it can switch behaviour according to variables in the game. At the start of this project, we wanted to make PortalBug more realistic by ensuring that the Antagonist only knew what it could observe at run-time, and learn from these observations. However, we isolated this learning component to within the PlayerObserver and connected it to the Antagonist. This way, the Antagonist could acquire more information and make more intelligent decisions. A basic potion behaviour was implemented with a pick-up-when-seen strategy. But it would be possible to extend on this by making use of a Dictionary that maps potions to their respective locations for each maze. The Dictionary to store these potions has already been implemented, however the antagonist currently does not make use of this. Example usages of this Dictionary are, for example: when the Antagonist is low on health, it could go to the nearest health potion it has come across, recorded in this Dictionary. If no health potions have been detected yet, it could continue on exploring to find one. When the Antagonist is in Fleeing mode, it could pick up the closest invisibility potion it can find, and when in Pursue mode, it could pick up a known speed potion, to go even faster and catch the player. The existing strategies can be extended with additional strategies. For now, however, the basic behaviour is
complete.

4.2.3. Evaluation Requirements: MoSCoW
Using the MoSCoW prioritisation format that was given to us by the client, we can establish that all MUSTs have been completed. The only part that was not completed relates to the dynamic generation of mazes, namely while the game is being played. Originally, we wanted to generate mazes based on the PlayerObserver at the time the player enters a new maze through a portal. However, because of the fact that the Antagonist also travels through mazes, all mazes need to be generated at the start. This means mazes would have to be destroyed and recreated at run-time when the player steps through a portal. In the end, due to the complexity of this task, time shortage, and other priorities, we decided to focus on more important requirements. When looking at the SHOULDs, we can declare to have completed a great portion of these. Because we prioritised other tasks, we did not implement the functionality of logging the decisions made by the antagonist. However, the user can indeed influence the decisions made by the antagonist, and we have added extra game play elements: the sword and shield powerups for the player.
This chapter describes the process of this project. The way we organised the project in general is discussed in Section 5.1. The software methodology we used is discussed in Section 5.2. Section 5.3 discusses the ways in which we ensured our code quality, and finally Section 5.4 evaluates our process.

5.1. ORGANISATION
First we discuss the division of labour in subsection 5.1.1, afterwards we discuss our ways of communication in subsection 5.1.2 and finally we name the collaboration tools we used in subsection 5.1.3.

5.1.1. DIVISION OF LABOUR
Concerning the way tasks were divided among team members, we prioritised skill and preference. We were aware that each of us had different abilities and used this to our advantage. There were multiple main tasks that could be handled separately, namely the Maze Generator, the Level-Grid Generator, the PlayerObserver, the antagonist’s behavior, and the modifications/extensions to the game (powerups, etc). Each team member was interested and/or skilled in one of these areas, so this made division of labour an easy task. As every team member mainly worked in their own area of the code, we were almost never in each others way. We regularly worked together in the same room, but by keeping each other up to date, we were also able to make considerable progress while working on our own on separate tasks. Each week, we would merge everyone’s work together and fix any possible issues that would emerge.

5.1.2. COMMUNICATION
When we were not working together in the same room, we would use mostly Skype to communicate. Video calls were used for meetings and big decisions while the chat was used for small, simple questions and keeping each other up to date. Besides Skype we also used WhatsApp for communication, mostly when someone was needed but not available on Skype. Communication with the client and coach was done mainly via email. Furthermore, a Team Contract has been put together and signed at the beginning of this project. A number of basic rules for communication and how to treat your team members have been composed in this Team Contract. The contract can be found in Appendix E.

5.1.3. COLLABORATION TOOLS
To support the teamwork process and ensure that we worked together as effectively and efficiently as possible, we made use of a number of tools. In this subsection we name these and briefly describe how and for what purpose we used them.

Trello
Trello is an organisation tool that allows for the creation of cards. These cards were used to indicate tasks that needed to be completed for the project. This could be any task, and did not necessarily involve programming. In these cards we referred to issues on BitBucket as well. Each card could be assigned to one or more team members. Once created, these cards were placed under any of the following categories:
• **Backlog** - As according to the Scrum definition of the backlog, this category contains tasks that still need to be handled for future weeks.

• **To Do** - The task has been assigned to (a) team member(s) and will be processed once they were available.

• **In Progress** - A team member is actively working on the current task.

• **Needs To Be Tested** - A code-related task is considered done, but the code itself is not properly tested yet. Either the Test Master or original card owner still has to write tests.

• **Done** - Tasks that are considered completed.

Trello easily allowed the team to see which tasks were currently being worked on and by whom. It removed the problem of not knowing who was working on what issue. It also provided good insight into the division of labour and allowed us to properly assign tasks to people who were available. It ensured that team members always had something to work on, and allowed them to pick a task from the backlog if they happened to complete all their tasks with time to spare. Trello also helped in communicating what we expected one another to do at that moment.

**Google Drive**

Google Drive was our common file folder. All our project-related files, except for the code and **\LaTeX** reports, went here: administration, ideas, meeting agendas and summaries, research etc. Thanks to Google Drive, everyone always had access to the latest version of all relevant files and multiple people could work on a single document simultaneously.

**Time tracking table**

In our project we used several sheets to keep track of the time we spent on different tasks. This allowed us to see the number of hours which we were still supposed to spend in a week and how we had been doing so far. At the same time we could see the individual hours spent by each member in their own table. This acted as an extension to Trello, since it allowed us to couple the tasks to the actual amount of hours spent on it.

**Overleaf**

Overleaf was used to write our research and final report. Overleaf allowed everyone to work on the report simultaneously using **\LaTeX**.

**TimeTable**

Our timetable was the planning we had for the current week. This did not include exceptions for individual team members, but gave a general idea of how our week was composed. The timetable included meetings we planned to have together and with supervisors alike. The timetable also contained certain specifics about the meeting, such as the Scrum review at the beginning of the week. The timetable was a general guideline for the team members and gave a good indication of our progress.

### 5.2. **SOFTWARE METHODOLOGY**

First it is discussed why and how the Software Engineering methodology of Scrum was used in Section 5.2.1 and afterwards the Software Development Tools that were used are detailed in Section 5.2.2.

#### 5.2.1. **SCRUN**

For the development of this project, the Software Engineering methodology of Scrum was used. This agile development method was chosen because we all had experience with it and learned from our Bachelor Computer Science program that it is a very efficient development method for these sorts of projects. We used an iterative approach (sprints) for the tasks at hand, and organised daily stand-ups, where the team meets to discuss recent and upcoming progress.

Each sprint started with a meeting where the team would decide upon a sprint plan, where tasks were distributed to each team member and quantified in terms of Scrum units. Units determine the amount of work a specific task is to accomplish. The value of one unit is determined by how much work the most basic task requires to be completed.

Each sprint ended with a Scrum review, where the team met with the client to show what had been accomplished in the previous sprint and receive feedback on the current product.
One of the requirements of the product shown during Scrum reviews was that it had to be an actual, complete, and compiling product (in terms of software). This meant that if the current version we had was incomplete and did not compile, we showed a previous, working version.

At the end of each scrum review, the team met and reviewed the previous sprint plan. Which tasks were completed and how much time it took to achieve them was reported. This allowed for progress in terms of determining the workload of tasks, and for the team to evolve positively. The team then decided upon a new sprint plan.

Part of the Scrum methodology is dividing the team into specific roles. We elected Valentine Mairet as the Product Owner, who was responsible for the communication with the client and knew exactly what the product was supposed to be. The Product Owner has the task to lead the Scrum reviews and stir the team towards the right path. In case the Product Owner was unavailable, we decided that the Scrum Master should temporarily take on that role. The Scrum Master is mainly responsible for the team and is in charge of leading the product development. Wendy Bolier was our Scrum Master, and she led the daily stand-ups and made sure the team worked properly.

### 5.2.2. SOFTWARE DEVELOPMENT TOOLS

During the project the following software development tools were used:

**Unity**

Unity is a cross-engine game engine and game development tool. PortalBug has been created in Unity, and therefore it is a requirement that everyone works with Unity. Unity allows working in a number of programming languages, but in this project we exclusively made use of C#. Unity comes built-in with MonoDevelop, an text editor for writing code. Group members were free to make use of this editor, though the majority used Microsoft Visual Studio.

**Microsoft Visual Studio**

Microsoft Visual Studio is what most group members used for writing code over the duration of this project. It possesses numerous features that MonoDevelop lacks, notably integrated Unit Testing support, which was made extensive use of throughout the project.

**BitBucket**

BitBucket has been supplied to us by the client. Hence, we use Git to manage our code and programming workflow. The Git repository features a master branch which contains a stable version of the game at all times, as desired by the process requirements and Scrum methodology. For the most part our tasks were split up, allowing each team member to work in their own branch during Scrum sprints. These branches are merged into the master branch once they are stable with the master branch.

**Blender**

Blender is the 3D modelling application we used for creating, modifying and colouring our 3D models for the graphics. We chose for Blender because it is one of the most widely-used 3D modelling applications and is free of charge. There is a lot of documentation on Blender available online and, furthermore, Unity natively imports Blender files.

**123D Sculpt+**

Besides Blender, we also used the tablet application 123D Sculpt+ to create some of the models. We chose for 123D Sculpt+ because it is a very self-explanatory application that barely requires any experience to create a 3D model.

### 5.3. QUALITY ASSURANCE

To ensure the quality of our product, several measures were taken during the project. The code has been validated twice by the Software Improvement Group (SIG) and the LibraRynth extension has been thoroughly tested with unit tests. Furthermore, the stability of the system as a whole has been proven by extensive stress testing.

#### 5.3.1. SOFTWARE IMPROVEMENT GROUP

An important requirement of this project was that our code needed to be reviewed by the Software Improvement Group (SIG) entity. SIG grades code quality based on certain rubrics such as Unit Complexity, Component Independence, etc. We were quite satisfied with the first software analysis, where we scored a 3.5 out
of 5, but we still took in the feedback and improved the code’s maintainability. Both the first and second SIG feedback can be found (in Dutch) in Appendix C.1 and C.2 respectively.

Addressing analysis results
We were happy with the results of the software analysis. The results were largely positive, and confirmed that we had been making good progress. Hence, we continued our thorough unit testing as we had been doing, and also addressed the large or complex methods which were referenced in the analysis. Furthermore, a sweep was done over our code in order to find more such cases. Furthermore, before the deadline of the second software analysis we did a thorough search to adjust inefficient data types/algorithms and fix warnings that existed within our code. In particular, over three quarters of our warnings originated from variables that were assigned by Unity, which resulted in Visual Studio marking them as being never assigned. These warnings clogged up the warning log. Through the use of `#pragma warning disable/restore` we suppressed these warnings. By suppressing these numerous warnings it became much easier to see if newly written code created a warning somewhere.

5.3.2. Testing
Testing is a very important part of software development. That is why we have dedicated quite some time to testing. This section describes shortly the two different types of testing that we used.

Unit testing
The LibraRynth extension has been designed to serve as an external library and easily be incorporated into other projects. Because of this thorough testing is critical, users should not need to worry about bugs in external libraries. In order to achieve this a C# test project was set up immediately for LibraRynth and thorough testing was required before code was accepted onto the master branch. At the end of this project the library possesses a respectable code coverage of just barely under 75%. Furthermore, the LevelGrid possesses thorough unit testing as well, attaining a code coverage of roughly 85%.

Stress testing
Many features that were added (RainAI, powerups, etc) require testing at runtime, which was not practical to accomplish with just unit tests. To fix this issue, it was decided to extensively stress test the game, which means running it and trying out different moves in an attempt to trigger an error. Using stress testing, many other issues were found and resolved. Issues that we were not able to resolve are described in the Bug Report that we handed to the client at the end of the project. It was possible to test elements at runtime using the Unity Test Tools [11] framework, but because other requirements had a higher priority and time was short, the decision to leave it at stress testing was taken.

5.4. Process evaluation
In this section our process during the project is evaluated. Our organisation is evaluated in Subsection 5.4.1, followed by our collaboration in Subsection 5.4.2. Finally, our usage of the Scrum methodology is evaluated in Subsection 5.4.3.

5.4.1. Organisation
The way we organised our project has greatly improved over the course of these ten weeks. Some things went well from the start, for example the division of labour and the usage of the collaboration tools such as Trello. Other aspects, such as communication, did not go so well in the beginning. Some of us were sometimes unreachable for a number of hours because they had forgotten to either turn on Skype, or notify the team of their absence in advance. Keeping each other up to date did not go smoothly in the beginning, because everyone was busy with independent tasks. Fortunately, these problems were quickly resolved. We quickly recognized the problem and an agreement was made as to how we should deal with communication and other issues. For the remainder of the project communication advanced smoothly.
5.4.2. COLLABORATION
We are content with the way our collaboration went. Not all team members knew each other before the start of this project, so none of us were entirely sure what to expect from one another.
In the end, getting along with each other was an easy task, and we never got into major arguments.
Regarding topics of interests and skills, we complemented each other. The ambiance of the group was satisfying: we could give concise feedback and critiques on each other's work without causing trouble or offending each other, which is the way it is supposed to be.

5.4.3. SCRUM
As we had already expected, Scrum turned out to be a good choice as development methodology for this project.
Thanks to the daily stand-ups, each team member was always up to date on the project's progress and the sprints incited us to think about what we wanted to achieve and work hard to complete it before the deadlines. However some things turned out to be more challenging than expected. Assigning Scrum units was one of them, especially at the start, where our average experience with Unity and Rain AI was close to none. It was nearly impossible for us to estimate the amount of workload certain tasks would take due to this inexperience.
Some things seemed easy until more information was obtained, and vice versa.
Another thing that did not work out as expected was the sprint planning. Every week we set up a sprint plan, and almost every week during the sprint review it appeared out-dated. Many constraints had appeared during the week: extra tasks with higher priority came up, tasks were exchanged between team members, tasks turned out to be twice as much work, tasks were put back on the backlog or even completely discarded. As we dived further into the project, we got accustomed to taking these issues into account during the sprint planning, but it never went completely flawless.
This chapter contains the conclusion of this project in section 6.1, some of the challenges that we faced in section 6.2 and section 6.3 contains discussion and recommendations.

6.1. CONCLUSION
Back in April, Divireo instructed us to extend upon a product of theirs. This product was a game called PortalBug, made in Unity, which was comprised of multiple mazes, each interconnected through portals, and a main antagonist that was there to prevent the player from reaching the end cell in the final maze.
Our task was to create a library that would take care of the maze generation, using different difficulty levels. We were also instructed to extend upon the antagonist’s behaviour. Both maze generator and antagonist were to adapt according to the user’s way of playing the game.
We expected to use a player observer entity extensively, especially for the antagonist’s behaviour. However, due to other, more time consuming than expected tasks, we were not able to make full use of this entity to that end.
However, within the course of ten weeks, the maze generation library was successfully implemented. We can now create mazes of different shapes and difficulty levels. Using this player observer, we can ‘observe’ certain useful aspects of the players’ gameplay, such as the time spent in a maze, most preferred direction, average distance between player and antagonist, etc. We make use of these recorded values to adapt the maze generation, and also the antagonist’s behaviour to some extent.

6.2. CHALLENGES
6.2.1. WHAT CONSTITUTES GOOD AI?
It was quite a challenge to figure out what constitutes a good AI. Several methods are well known: decision trees and tables, or rule-based approaches, but most techniques are not available for the public, due to patenting.
In the end, we opted for Rain AI, because it is an exceptionally popular AI library used in Unity with a great deal of positive feedback from both professional and amateur users alike.

6.2.2. UNITY PREFABS
Part of the requirements of this project was using Git. However, when transmitting a new prefab to Git, things never went right. When pulling another branch, all prefabs would disappear and all connections to the corresponding script were lost. This was the result of prefabs being binary files, which easily led to conflicts when combined with Git.
Fortunately, one of our team members had experience with Unity and managed to find a way to resolve this issue. We always uploaded the newest versions of the Unity Assets (which contains scripts and prefabs) to our Google Drive repository such that everyone in the team would have access to it.
6.3. **Discussion and Recommendations**

Though we are finished with this project, there is still much left to improve upon. Although we did complete the requirements, there are still quite some things that can be improved. In this section we will discuss some of these improvements.

6.3.1. **Improvement in Maze Generation Speed**

At current, the maze generation incorporates a brute force search for the placement of the rooms and vertices. This brute force search finds all possible combinations for placement of the rooms and then picks a random combination from this collection. This is expected to take the most amount of time, and if improved could speed up the maze generator significantly.

6.3.2. **Improvements in the Antagonist Behaviour**

In regards to the behaviour of the the antagonist a number of improvements can be made as well. Right now a basic potion behaviour has been implemented which makes use of a pick-up-when-seen strategy. This can be improved by making the spider consciously go back to a potion which it has seen before. The antagonist already saves all potions it encounters during its explorations in a Dictionary, it just does not use this Dictionary yet. Another improvement regarding the potions could be making the antagonist steal potions away from the player. Currently the antagonist will walk right past a potion if it has no immediate need for it, but it would be more intelligent if it would also consider whether the player needs it and whether to pick it up before the player can.

6.3.3. **Improvement of PortalBug**

In comparison with the skeleton game, where the bug and spider were represented by standard spheres, the graphics have already improved a lot. But they are not 'high quality graphics', which is a given as we had almost no experience with creating graphics. Our recommendation is to outsource the graphics to real graphic designers if higher quality graphics are desired. Some other things we wanted to add to PortalBug, but unfortunately could not due to time restrictions, were a menu screen and keeping score of the player's accomplishments.

6.4. **Future Work and Extensions**

The current delivered product is highly extensible, and Divireo is keen on improving it. The PlayerObserver for instance, could be used to also track how many potions the player picks up, how often the player uses the sword and/or shield, and many other strategies which can be observed. The Antagonist could also make more use of the PlayerObserver, and could also have its own powerups (such as spiderwebs, possible vision through walls, etc). Combining the data it sees, data it knows, and the features it can make use of, a more complex behaviour tree can be created to increase the intelligence of the Antagonist, which will serve to make PortalBug even more challenging.

Generating mazes at run-time was a task we wished to complete, but refrained from in the end, because other tasks had a higher priority, and because of time shortage. The maze generation at run-time will be implemented in the future, but will be done by Divireo.

There are many more possibilities for extension. PortalBug could be a multiplayer game, or even contain multiple Antagonists, to further increase the difficulty.


API
Application Programmable Interface; The methods that are available to the framework that uses the specified interface.

Maze solution
All tiles within a maze that are accessible and have been instanced. Every tile in the maze solution can be reached from any other tile in the maze solution.

Unity Prefab
A Unity Prefab is a pre-built GameObject which already has all of its components and properties specified in advance. This allows for easy setup in advance and easy fetching through Unity's built-in GetComponent methods.

AI
Artificial Intelligence; The generic description of some object or class that is able to make 'intelligent' decisions based on its knowledge.

Navigation Mesh
A GameObject that is used by Rain AI to determine a walkable area. It can be configured according to size and slope angle.

AI Rig
A GameObject that is used by Rain AI to setup an AI for a character rig. It consists of a memory, a mind, motion, animation, navigation, perception (basic senses), and custom AI elements that can be created by the developer.

Entity Rig
A GameObject that can be detected by an AI Rig.
Maze Generator API

B.1. LibraRynth API

Int32 GetWidth()
Retrieve the width of the maze grid.

Int32 GetHeight()
Retrieve the height of the maze grid.

IMazeCell GetCellAt(Int32Point coordinates)
Retrieve the IMazeCell object that corresponds to the specified coordinate. Will throw an exception if out of bounds. If the location contains a cell that is not null, the location will be part of the maze solution.

Boolean HasWallAt(Int32Point coordinates, WallData.WallDirection direction)
Retrieve whether there exists a wall at the specified coordinate in the specified direction. Walls on the opposite side are automatically taken into account, so coordinates outside of the maze are allowed.

Boolean CellLocationIsAccessible(Int32Point coordinates)
This method allows for the framework to check whether a cell at the specified coordinate would allow for a cell to be housed.

Boolean GetRandomValidCellPoint(Random random, out Int32Point cellPoint)
Retrieve a random cell in the maze that has been instantiated and is part of the maze solution. Will return false if no valid points are available and true otherwise.

Int32Point[] GetXRandomValidCellPoints(Int32 amount)
Retrieve several unique random cell points in the maze. If the requested amount exceeds the available amount of valid tiles, the method will throw an exception.
B.2. **LIBRANYTHDATA API**

```csharp
void SetCell(Int32Point coordinates, IMazeCell cellData)
    Set the cell at the specified location. Can set the cell to null to indicate cell removal. Will throw an exception if the coordinates are inaccessible or out of bounds.

void SetWall(Int32Point coordinates, WallData.WallDirection direction, bool hasWall)
    Set the wall at the specified location and direction. A true value for hasWall indicates the presence of a wall. This method will automatically set the wall for the opposing cell as well. This method will not throw an exception and will replace a wall completely unchecked.

Boolean TrySetWall(Int32Point coordinates, WallData.WallDirection direction, bool hasWall)
    Much like the SetWall functionality, except that this method will not replace a wall if it is deemed a static wall. A static wall is a wall that is either part of the surrounding bounds of the maze (i.e. the opposing cell is not accessible) or if the wall is a part of a room (i.e. this or opposing cell is part of a room). This method will return true if the wall was replaced and false otherwise.

void ParseRoomBounds(MazeDifficultyHandling.RoomBoundsResult result, Random random)
    Initialise the calculated room for this maze to the maze itself. This will make the supplied room draw themselves to the maze. The supplied room will be added to the collection of rooms that are stored in the maze, thus making its walls static walls.

List<IdentifiedPoint> GetStartingPoints(Random random)
    Retrieve the listing of all starting points that the maze contains. These starting points are either dependent on the rooms that are contained in the maze or, if there are no rooms, a singular random point in the maze.

HashSet<Int32Point> GetValidRoomPoints()
    Retrieve all the points in the maze that are accessible and as such fit for room placement.

void FillAllWalls()
    Initialise the maze by setting up all walls inside of it.

List<Int32Point> GetNeighbours(Int32Point mazeCell)
    Retrieve all neighbors directly surrounding the indicated cell (North, East, South, West) that are accessible by the maze.

void RemoveWallsBetween(Int32Point newPoint, Int32Point lastPoint)
    Remove the walls between the indicated points to ensure access between. Will throw an exception if the wall is deemed to be a static wall.

Boolean IsStaticWall(Int32Point cell, WallData.WallDirection direction)
    Retrieve a Boolean indication whether the specified wall is a static wall. A static wall is a wall that is either part of the rooms of a maze or the bounds of the maze and should not be removed unless part of the initialisation of the rooms.
```
SOFTWARE IMPROVEMENT GROUP
FEEDBACK

C.1. FIRST SOFTWARE ANALYSIS
“De code van het systeem scoort 3,5 ster op ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is. De hoogste score is niet behaald door een lagere scores voor Unit Complexity en Component Independence.
Voor Unit Complexity wordt er gekeken naar het percentage code dat bovengemiddeld complex is. Het op-splitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, makkelijker te testen is en daardoor eenvoudiger te onderhouden wordt. Een voorbeeld van een methode met onnodige complexiteit is GridDirections.GetDirection. Zowel het gebruik van complexe boolean-condities als het herhalen van if-statements zorgt ervoor dat je snel een foutje maakt. Daarnaast is het nu erg vervelend om voor deze methode een unit test te moeten schrijven, aangezien de code nu 21 mogelijke paden telt.
In andere gevallen, zoals bijvoorbeeld LevelGrid.SetMinimumSparseness, wordt de complexiteit veroorzaakt door de lengte van de methode. Deze code wordt nu met commentaar in stukken verdeeld, en dit soort commentaarregels zijn een goede indicatie dat er een autonoom stuk functionaliteit te ontdekken is. Door elk van deze functionaliteiten onder te brengen in een aparte methode met een descriptieve naam kan elk van de onderdelen apart getest worden en wordt de overall flow van de methode makkelijker te begrijpen.
Tot slot is het goed om te zien dat jullie unit test-code hebben geschreven. Hopelijk zal het volume van de test-code ook groeien op het moment dat er nieuwe functionaliteit toegevoegd wordt.
Over het algemeen scoort de code bovengemiddeld, hopelijk lukt het om dit niveau te behouden tijdens de rest van de ontwikkelfase.”

C.2. SECOND SOFTWARE ANALYSIS
“In de tweede upload zien we dat zowel de omvang van het systeem als de score voor onderhoudbaarheid zijn gestegen.
De stijging in score is echter niet toe te schrijven aan een verbetering op het gebied van Unit Complexity, dat tijdens de analyse van de eerste upload als verbeterpunt werd aangemerkt. Jullie code bevat nog steeds methodes die te veel beslispunten bevatten en daardoor moeilijk te testen zijn, zie bijvoorbeeld MazeGenRecursiveBacktracking.BuildMazePath.
Uit deze observaties kunnen we concluderen dat de aanbevelingen van de vorige evaluatie deels zijn meegenomen in het ontwikkeltraject.”
GENERAL INFORMATION

**Project Title:** Personalized Gaming Using Machine Learning

**Client Organisation:** Divireo

**Final Presentation Date:** June 26, 2015

DESCRIPTION

The entertainment industry is one of the biggest, fastest growing, and most popular fields in the world of today. Artificial Intelligence has also been a very popular subject these past decades, and is often found in games. The more intelligent and complex a game is, the more challenging it becomes, and that is an important aspect of a good game. Divireo has hence instructed us to extend upon their own skeleton game, with the purpose of creating a scary, close to unbeatable game.

The most challenging aspects of this project were having to work with a framework that only one of our team members had experience with, and working with a library that was rather poorly documented. We had several issues with aspects of this library that impeded on our work.

During the research phase, we learned much about which algorithms can be used to render difficult mazes and which techniques are applied when dealing with Artificial Intelligence in games; combined, these elements make the game difficult.

To ensure a good development process, we established a team contract and made use of the Scrum methodology of software development. We also made extensive use of communication tools such as Skype and WhatsApp to ensure we stayed in touch. There were some problems as each member had different working hours that they preferred, but we managed to overcome this problem through frequent communication. We also had weekly meetings with both coach and client to discuss the product’s progress.

The product we delivered now contains a maze generator, a LevelGrid generator which creates a grid of mazes, interconnected by portals, an antagonist with an intelligent behaviour that was implemented, and extra elements such as powerups for the player.

There is great potential for extensions to the game: adding more powerups, refining the Antagonist’s behaviour, generating mazes at run-time instead of at start-up, etc. Divireo is planning on taking the product we have created and extend it for future releases.
**TEAM MEMBERS**

Name: Tom Peeters  
*Interests:* Artificial Intelligence, Game Design  
*Role and contribution:* Developer, LaTeX Master

Name: Wendy Bolier  
*Interests:* Game Design, Graphics  
*Role and contribution:* Scrum Master, Developer, Stress Tester, Graphics Designer

Name: Bas Boellaard  
*Interests:* Algorithms and Algorithm Logic  
*Role and contribution:* Developer, Git Master

Name: Valentine Mairet  
*Interests:* Artificial Intelligence, Game Design, Unity, Management of Technology  
*Role and contribution:* Product Owner, Developer, Unity Maintenance

All team members contributed to preparing the research report, the final report, and the presentation.

**CLIENT**

Name: Bastiaan Reijm  
*Affiliation:* Divireo

**COACH**

Name: PD Dr. Nils Bulling  
*Affiliation:* Department of Intelligent Systems, TU Delft

**CONTACT**

Bas Boellaard  bas.boe@gmail.com  
Tom Peeters  tompeeters368@gmail.com  
Valentine Mairet  val.mairet@gmail.com  
Wendy Bolier  wendybolier@hotmail.com
TEAM CONTRACT
Team Contract

Team Name: PortalBug

These are the terms of group conduct and cooperation that we agree on as a team.

Participation: We agree to...

1. Be honest and open during meetings.
2. Encourage a diversity of opinions on all topics.
3. Give everyone the opportunity for equal participation.
4. Be open to new approaches and listen to new ideas.
5. Avoid placing blame when things go wrong. Instead, we will discuss the process and explore how it can be improved.

Communication: We agree to...

1. Seek first to understand, and then to be understood.
2. Be clear and to the point.
3. Practice active, effective listening skills.
4. Keep discussion on track.
5. Use visual means such as drawings, charts, and tables to facilitate discussion.
6. We will do our best to be always reachable via Whatsapp or Skype.
7. Update our time tracking sheet every day.
8. Update Trello immediately when starting or finishing a task.
9. Give useful feedback to our teammates.
10. Call for help immediately when needed.

Meetings: We agree to...

1. Attend all the meetings (daily scrum, scrum reviews, coach meetings and client meetings).
2. Encourage everyone to participate.
3. Encourage all ideas (no criticism).
4. Build on each other’s ideas.
5. Get input from the entire team before a decision is made.
6. Be there in time.
7. Come to meetings prepared.
8. Unresolved issues will be added to the Issues list.

Conflict: We agree to...
1. Regard conflict as normal and as an opportunity for growth.
2. Seek to understand the interests and desires for each party involved before arriving at answers or solutions.
3. Choose an appropriate time and place to discuss and explore the conflict.
4. Listen openly to other points of view.
5. Repeat back to the other person what we understand and ask if it is correct.
6. Acknowledge valid points that the other person has made.
7. State our points of view and our interests in a non-judgemental and non-attacking manner.
8. Seek to find some common ground for agreement.

**Deadlines:** We agree to...

1. Finish our work way in advance of the hard deadlines to prevent missing them.
2. Let our teammates know as soon as possible when realizing that you might not be able to complete your task in time.

<table>
<thead>
<tr>
<th>Team Member's Name</th>
<th>Team Member's Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bas Boellaard</td>
<td></td>
</tr>
<tr>
<td>Tom Peeters</td>
<td></td>
</tr>
<tr>
<td>Valentine Mairet</td>
<td></td>
</tr>
<tr>
<td>Wendy Bolier</td>
<td></td>
</tr>
</tbody>
</table>
HOW TO FIX: RAINAI
# Table of Contents

Introduction  
Adding RainAI Components at Runtime  
Generating a Navigation Mesh at Runtime  
  Prior Requirements  
  Hierarchy  
  Using Layers  
Fixing Concurrency Problems
Introduction

As we get more familiar with RainAI we realise that there is barely any documentation on the things we are attempting to do. Objects that RainAI should influence are generated at runtime, and this seems to have never been done before. In this document, we explain the issues we have encountered when prefab generation happens at runtime, and we elaborate on the solutions we have discovered.
Adding RainAI Components at Runtime

RainAI was designed for preexisting objects in the Unity world. In our project, we add objects (prefabs) at runtime. This was the first challenge we encountered: how do we add an AI component to a prefab that does not even exist yet? Fortunately, it was possible to manually add an AI component to a prefab in the editor, and to save that component. At runtime, the prefab would spawn and the AI component would function appropriately.
Generating a Navigation Mesh at Runtime

As mentioned earlier, RainAI was designed for preexisting objects in the world. As we generate cells and walls at runtime, generating a navigation mesh straight from the RainAI editor was not an option. The Rival Theory website provides with a code sample to generate a navigation mesh straight from a script. Unfortunately, little information was made available as to how to use this code sample.

Prior Requirements
Looking at the given code sample (see Figure 1), we immediately notice that we need a preexisting navigation mesh in the world.

![Figure 1: RainAI Navigation Mesh sample code](image)
To be able to use this code, we chose to create a GameObject corresponding to this class. This GameObject, along with a preexisting navigation mesh, was placed as a child of the DungeonMaster object (see Figure 2).

![Figure 2: Hierarchy 1](image)

The DungeonMaster then instantiates the generator and calls the GenerateNavmesh() method. In Figure 3, a capture of our current code is displayed.

```csharp
private void InstantiateNavMeshGenerator()
{
    meshGenerator = GetComponentInChildren<NavMeshGenerator>();

    meshGenerator.GenerateNavmesh();
}
```

![Figure 3: InstantiateNavMeshGenerator()](image)

**Hierarchy**

Not only did the navigation mesh generator and navigation mesh itself need to be children of the Dungeon Master, but it seemed that every object the navigation mesh applies to also needs to be a child as well. We hence decided to make everything in the world a child of the Dungeon Master (see Figure 4), to keep things logical and simple.
Using Layers

One final tweak that needed to happen in order for the navigation mesh to work was dealing with layers. The navigation mesh needs to know which layer it needs to act upon. In our code, maze cells and maze walls gets set as children of corresponding mazes, which get set as children of a Wrapper object. We recursively go through all the children of the Wrapper object (see Figure 5) in order to change their layers to one common layer, which the navigation mesh will be applied to. Prior to starting the game, we set up the right layer in the navigation mesh editor (see Figure 6).

```csharp
//Depth search of a Transform hierarchy to change every child's layer
public static void makeLayer (Transform trans, string name)
{
    foreach (Transform child in trans) {
        child.gameObject.layer = LayerMask.NameToLayer (name);
        makeLayer (child, name);
    }
}
```

Figure 5: makeLayer(Wrapper, "Mazes")

Figure 6: Navigation Mesh Editor
Fixing Concurrency Problems

Another problem we came across was that the AI component of the Antagonist gets activated as the Antagonist is instantiated. While we position the enemy on the board after instantiation, two concurring threads come to be. On one thread, the AI forces the enemy to move, on the other thread, the enemy is forcibly positioned at a specific location. We annihilated this problem by setting up a timer (see Figure 7) in the behaviour tree: the enemy waits a few second before starting to move. This way, we can position the enemy properly before it even starts moving.

![Figure 7: Antagonist Behaviour Tree](image)
ORIGINAL PROJECT DESCRIPTION
Bachelor Thesis Description
# Table of Contents

- **Introduction**
- **General Overview**
- **Skeleton Game**
  - Maze
  - Walls
  - Cells
  - Player
  - Antagonist
  - Portals
  - Potions
  - Miscellaneous
- **Product Requirements**
  - Must
  - Should
  - Could
- **Process Requirements**
  - Must
  - Should
  - Could
- **Glossary**
- **Contact Information**
Introduction

Welcome to the Divireo Bachelor Project 2015! We are very pleased to be hosting our first Bachelor Project. We expect that this be a mutually beneficial experience, both for the company and the students from the TU Delft.

This document is the official project description of the Divireo Bachelor Project 2015. All the necessary information about the project, the provided material, the expectations, and the requirements of this project are contained in the appropriate sections. The General Overview section gives a high level description of the project and while the Skeleton Project section details the given testing game. The Product Requirements outline what is expected as a deliverable and complete product at the end of this project. Next the Process Requirements outline what is expected of the way the students work on the project. Finally, a Glossary is provided for reference.
General Overview

The goal of this project is to develop an intelligent game personalisation library for the Unity game development framework; this library will be used for a specific class of games, specifically games involving decision making and strategies to solve a large variety of puzzles. The library will then be applied to a pre-existing game skeleton for testing and verification purposes.

The provided skeleton game is a first-person 3D maze game where the player can choose out of multiple available strategies to beat the level. Traps and power-ups are scattered around the maze, while the main antagonist roams around the area, with the goal of finding and destroying the player.

The main antagonist of the game needs to be intelligent in order to make the game more challenging to play over time. The library must provide the antagonist with a ‘virtual brain’ that can learn the player’s strategies and preferences to change and adapt its behaviour accordingly.

To show the full capabilities of the library, the students will also be expected to extend the game with an adaptive maze generator as well as transforming the provided skeleton into a working prototype for a personalised experience.

The students may use machine learning techniques, AI techniques, and search algorithms if well developed arguments are supplied. The library is expected to follow industry software engineering principles, thus it must be open to future extensions and must be tested as much as reasonably possible. Additionally, all code is expected to be well-formatted, easily readable, and well documented.
Skeleton Game

PortalBug is a game skeleton built in Unity3D and is written in C#. The features below have been implemented and are sorted by type.

**Maze**

The maze is the main playing space in the game. It consists of walls and cells. The following features are included in the skeleton game:

- Mazes are randomly generated (using a modified Prim’s algorithm)
- Mazes are guaranteed to be solvable
- Maze size is editable from the **Inspector**
- Cell size is editable from the **Inspector**
- Multiple mazes can be generated behind each other
- Cells can be retrieved by coordinates
- Walls can be retrieved by coordinates of the cell and a direction id

**Walls**

The walls of the maze are static Game Objects that prevent the player and antagonist from moving in certain directions. The following features is included in the skeleton game:

- Wall size is editable from the **Inspector**
- Wall texture is editable from the **Drawers**

**Cells**

The floor of the maze consists of Cells. Cells have a rigid body that prevents the player from falling through and can have other properties. The following Cell types have been implemented in the skeleton game.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>No special properties</td>
</tr>
<tr>
<td>Start</td>
<td>Unique to an entire level, location at which the player starts</td>
</tr>
<tr>
<td>End</td>
<td>Unique to an entire level, location at which the antagonist starts and the player must finish</td>
</tr>
<tr>
<td>Max Health Increase</td>
<td>Increases the maximum health capacity of the player (not the current health)</td>
</tr>
<tr>
<td>Paralysis</td>
<td>Paralyses the player for 5 seconds if the player is stationary on the tile for more than 3 seconds (values can be edited via the <strong>Inspector</strong></td>
</tr>
</tbody>
</table>

Divireo 2015
Player

The player is represented as bug that roams around the maze. The following features have been implemented in the skeleton game:

- Small sphere representing the player
- Sphere size is editable from the Inspector
- The player can be controlled with key presses
- The player knows which cell it's currently on
- The player has 200 health
- The player can deal 50 damage
- The player can be attacked and if killed, loses the game
- The player starts on the Start Cell
- The player must reach the End Cell to win the game
- The player can leave a trail of bug juice behind

Antagonist

This is the main enemy of the player and is represented in the game as a spider. The following features have been implemented:

- Large sphere representing the player
- Sphere size is editable from the Inspector
- The antagonist knows which cell it's currently on
- The antagonist has 1000 health
- The antagonist can deal 50 damage
- The antagonist has a head direction (the current direction it’s facing in)
- The antagonist has a maximum visibility which is editable from the Inspector
- The antagonist cannot see or move through walls
- The antagonist pursues the player if the antagonist sees the player
- The antagonist randomly roams around if not pursuing the player
- The antagonist can be attacked
- The antagonist starts on the End Cell

Portals

A portal is how any object travels from one maze to another inside a level. The following features have been implemented:

- If a level contains 1 maze, no portals are generated
- The first maze contains 1 portal to the next maze
- The last maze contains 1 portal to the previous maze
- The other mazes contain both a portal to the next and previous maze
- The player can travel through a portal
- The antagonist can travel through a portal
- A portal will only be generated on a random basic cell
# Potions

Potions give the player certain boosts or abilities. Potions are stored in a potion bag and can be used at anytime by the player. The following Potion types have been implemented in the skeleton game.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Health</td>
<td>The player's current health is increased (only if current health &lt; max health)</td>
</tr>
<tr>
<td>Speed</td>
<td>The player's speed is increased</td>
</tr>
<tr>
<td>Invisibility</td>
<td>The player is not visible to the antagonist</td>
</tr>
</tbody>
</table>

# Miscellaneous

The following features have also been added to the skeleton game:
- The game can be restarted using the space bar
- The camera follows the player to create a first person point of view
- The camera can be rotated independent from the player
- The camera snaps back to the player's view if the player moves or rotates
Product Requirements

Must

• The antagonist makes intelligent decisions using the available data
• The maze changes based on the player’s strategy at one or more of the following:
  • At creation time
  • Dynamically while the game is being played
• The maze contains multiple portals on levels with more than two connected mazes
• Documentation must be accurate and extensive
• Tested to the highest extent possible
• Modular and extensible
• Loosely compelled to the Skeleton Game

Should

• The user can view a log of the decisions made by the antagonist
• The user can influence the decisions made by the antagonist
• Extra game play elements, for example:
  • Moving fog, smoke, or poison clouds
  • Player strategies such as a sword or key
  • Bonus level
  • Collectable items

Could

• The game has high quality graphics
• The game has a menu and other game play related UI elements

Any additions to the skeleton game (such as graphics and game play elements) or too these requirements must be discussed with the company mentor as well as all members of the group.

Divireo 2015

BEP Description

Page 8 of 10
Process Requirements

Must

• The Bachelor Thesis is recognised to be 15 ECTS thus **each** student is expected to log a minimum of **420 hours**
• A team and individual log must be kept up to date to justify the hours spent
• The team must use an agile development methodology (Scrum, Extreme Programming, etc)
• The team must have regular meetings with the TU Coach and the company mentor
• The team must keep notes for each iteration which include:
  • Accomplished Work
  • Meeting Notes
  • Problems and Solutions (if applicable)
  • Reflection (at the end of the iteration)
• During the implementation phase of the project there must **always** be a stable version of the product (aside from the start of the project)

Should

• There must be an up-to-date Product Backlog
• Common email address
• Common file folder
• Agile Development Tool (digital or analog)
• Keep Pair Programming to a minimum
• Work in the same physical location as much as possible

Could

• Role or feature specialisation per person
Glossary

<table>
<thead>
<tr>
<th>Drawer</th>
<th>A prefab that is provided in the Skeleton Game project which contains multiple prefabs; is used to change the appearance of the walls and cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>Game Object</td>
<td>Any object in Unity that supports some level of interaction</td>
</tr>
<tr>
<td>Inspector</td>
<td>The view in Unity3D that allows inspection of objects as well as editing of public variables of classes</td>
</tr>
<tr>
<td>Prefab</td>
<td>Pre-Fabricated Game Object</td>
</tr>
</tbody>
</table>

Contact Information

For questions and comments please send an email to reijm@divireo.com
# Contents

1 Introduction .................................................. 1

2 Project Description ........................................... 1
   2.1 The Skeleton Game ........................................ 1
   2.2 The Extension ............................................ 1
   2.3 Design Goals ............................................. 2

3 Project Research ............................................. 2
   3.1 Intelligent Antagonist .................................... 2
      3.1.1 Decision trees ..................................... 3
      3.1.2 State machines .................................... 3
      3.1.3 Behavior trees .................................... 3
      3.1.4 Goal-oriented systems ......................... 3
      3.1.5 Existing examples of AI in Unity ....... 3
      3.1.6 Final decision .................................. 4
   3.2 Intelligent Maze ....................................... 4
      3.2.1 Maze generation strategy ....................... 4
      3.2.2 Maze patterns ................................... 5
      3.2.3 Maze adaptability: player variables ........ 5
      3.2.4 Maze adaptability: tailored maze generation 6
      3.2.5 Path through the maze ............................. 6
      3.2.6 Perfect and non-perfect mazes .............. 6
      3.2.7 Maze adaptability: power-ups ................. 7
   3.3 Modular Design .................................... 7
      3.3.1 Modular dependencies .............................. 7
      3.3.2 Modular maze generator ......................... 8
      3.3.3 Modular enemy AI .................................. 8
   3.4 Extra Gameplay Elements .............................. 8
      3.4.1 Challenging ........................................ 8
      3.4.2 Rewarding ......................................... 9
      3.4.3 Flexibility .......................................... 9
      3.4.4 Horror techniques ............................. 9

4 Development Method ...................................... 9
   4.1 Scrum .................................................. 10
   4.2 Testing ................................................ 10
   4.3 Team Roles ............................................ 10

5 Conclusion ................................................ 10

Bibliography ................................................... 11

A Appendix ..................................................... 13
   A.1 High Quality Graphics .................................. 13
      A.1.1 Polygon meshes .................................. 13
      A.1.2 Blender .............................................. 13
      A.1.3 Textures ............................................ 13
      A.1.4 Optimization of 3D models ................... 13
      A.1.5 Lights and shadows ............................ 13
1. INTRODUCTION
For the 2015 Bachelor Thesis, we have chosen, as a team, to work on a project hosted by the Divireo company. Divireo is a small, recently founded company in its startup phase. They have provided us with the skeleton of a game called PortalBug, that we have to modify and extend in order to meet their requirements. In this report, we first describe in section 2, the actual project and what is expected from us. Then in section 3, we go deep into an analysis of various possibilities for the development of the new game. We consider various implementation techniques in order to find the most suitable one for this project. We also give our own analysis of a modular design technique for the extension we need to implement, and we inquire upon extra gameplay elements that could be added to the original game. In section 4, we describe our method of development: which software engineering methods we will use and how we will plan and report iterative development sessions. Finally, we extend towards the Game Design field, giving suggestion upon what design techniques we should use to render an enjoyable and appealing video game.

In the Appendix, we elaborate on techniques to create high quality graphics.

2. PROJECT DESCRIPTION
The first thing that Divireo noticed when entering the entertainment industry is that Artificial Intelligence (AI) is a highly praised and discussed topic. It is thrilling and alarming at the same time. Divireo wants to use AI, combined with a strategic gameplay, in order to render a challenging game. Why challenging? Well, who does not like a little competition from time to time? Now, what happens when the game itself is the competition? By combining Artificial Intelligence, strategic gameplay, and elements from the field of horror, Divireo wants to create a game that will both frighten and entertain players as much as possible.

2.1. THE SKELETON GAME
The given skeleton game is made in Unity, a cross-platform game engine developed by Unity Technologies, and contains a collection of mazes generated by an adapted version of Prim's algorithm. Prim's algorithm [1] is a greedy algorithm that finds a minimum spanning tree for a connected weighted undirected graph. The player starts in the beginning of the first maze, and can wander through the maze as an explorer. The goal is to get to the end cell, located in the last maze.

To explore new mazes, the player has to go through a portal, which takes them to a portal connected to the latter, in the next maze. Portals currently have the same colour. Potions and special cells are distributed among the mazes, and the player has the possibility to interact with them by picking up the potions or absorbing the special cells. Potions that can increase speed, health or render the player invisible are scattered across. Special cells that increase maximum health or paralyze the player for a few seconds are laid out and hidden deep within the maze.

The skeleton game also stars an antagonist, currently located near the end of the last maze. The implemented AI is very basic, as the antagonist moves through the maze randomly, or in the direction of the player when spotted. When the antagonist collides with the player, damage is dealt to each character.

Portals have been designed to connect each maze within the collection, and both player and antagonist can make use of them.

2.2. THE EXTENSION
Divireo has instructed us to extend the skeleton game by wrapping a library that will ensure the intelligence inside the game.

The library must grant the antagonist a logical basis, and a certain intelligence that will allow it to make rational decisions according to the players’ moves. The point is that the antagonist should predict and/or surpass the players’ strategy, and even come up with its own to hunt and kill the player. Multiple ways can be used for this and Divireo has advised us to look into Agent programming in Unity.

In addition to an intelligent antagonist, the company has requested an intelligent game itself. As mentioned before, the mazes are generated according to an adaptation of Prim’s algorithm, which uses randomization to generate walls and cells. What we need to accomplish is that the maze should generate itself according to the players’ “personality”: the way they play the game. This means that the players’ moves and preferences need to be recorded and taken into account when generating a maze, either as the game starts or dynamically as the game is being played.
2.3. **Design Goals**

One of the goals Divireo has in mind is to be able to reuse our extension for similar games that will be produced. Therefore, the AI library must be as loosely coupled as possible from the skeleton game. Furthermore, as both the antagonist and game itself learn from the players’ mindset, Divireo wants to derive a multiplayer platform from this game, where players can challenge others to solve their maze, with their antagonist inside it.

The extension has a number of parameters which the company wants to be able to easily access and modify. We should therefore design the extension in such a way that Divireo can reach the internal settings to fine-tune the result.

Certain aspects of the game, concerning gameplay and graphics, should be similar to games in the field of Horror. *Amnesia* (*Frictional Games, 2010*), *Slender* (*Parsec Productions, 2012*) and *Resident Evil* (*Capcom*) are games that make use of shadowing and music in order to frighten their players. Divireo wants *PortalBug* to not only be a difficult game, but also lean towards the horror side. Having a spider as antagonist already contributes to this atmosphere of horror.

3. **Project Research**

Two distinctions can be made from the project description. On the one hand, we have an intelligent antagonist. On the other hand, an intelligent game itself. Moreover, we have established that the extension should be as loosely coupled as possible from the original game. Finally, we extend on another subject: extra gameplay elements that could be added to the game, to make it more appealing and entertaining to the audience.

In this section, we execute a research into the best strategies to accomplish the four requirements listed above.

3.1. **Intelligent Antagonist**

AIs can be created in numerous ways, each with their own distinct strengths and weaknesses. Decision trees, neural networks, goal-orientated systems, fuzzy logic and state machines are just some of the many methods which have been employed in the past to create AI.

Depending on the tasks that an AI is required to complete different methods should be used, therefore it is important to be clearly aware of what tasks you want your AI to perform. Hence, detailed requirements as to what tasks the antagonist should be capable of are required.

1. The antagonist should be realistically aware of its environments. This means that it should not possess knowledge it cannot determine from its senses.
2. The antagonist should map the maze as he explores it.
3. The antagonist should intelligently attempt to track or locate the player using its knowledge.
4. The antagonist should be able to employ tactics.
5. The antagonist should be able to use hearing to locate the player when nearby, but out of sight.
6. The antagonist should be able to use smell to locate the player when the player uses bug spray.

One of the things that we can immediately determine from these requirements is that a knowledge base is required. In particular, the antagonist will be dealing with a lot of information, which makes some systems, such as fuzzy logic, a poor choice, as they provide little scaleability. Furthermore, systems such as neural networks, which rely on large training sets, are also unfeasible.

Based on these requirements, we narrowed down the list of befitting AI systems down to the following collection of systems:

- Decision Trees
- Behavior Trees
- State Machines
- Goal-oriented Systems

These systems will be researched in further detail to determine the optimal system to be used.
**Decision Trees**

Decision trees are a popular method of implementing AI. They have the advantage of being fast, easy to implement and simple to comprehend [2]. Because of their speed they are often used in games that feature potentially thousands of agents active at once.

However, their simplicity also makes AIs created using them basic and predictable. This issue can be resolved to quite an extent by adding random variables to decisions made, but doesn’t resolve the issue entirely. Furthermore, in the case of our antagonist the decision tree may grow to excessive sizes, making it hard to oversee.

For these reasons we recommend against using decision trees.

**State Machines**

The way state machines typically work is by the character being in a state, such as *search* until a certain objective is met, such as *target found*.

Using a variation of the standard state machines, called hierarchical state machines, it becomes possible to create many states within one encompassing state [3]. For example, in the state *search mode* we can include the states *explore* when it doesn’t have any leads on the player, *follow scent* when it catches the scent of the bug spray and *chase* when it lost the player after locating him. As each of these are contained within the mode *search mode* they will all automatically respond to an event such as *player sighted* which would cause it to enter *battle mode*. Using such a design, depth can be created by creating many states, while retaining responsiveness and overview through the use of hierarchical states.

For these reasons we recommend the use of hierarchical state machines.

**Behavior Trees**

Behavior trees can be regarded as a combination of decision trees and hierarchical state machines, in particular, the simplicity of decision trees and depth of hierarchical state machines. Instead of a single decision being at the end of the tree, a sequence of actions is there instead, collectively called a task. Because of this structure, they can be used to perform complex actions.

Additionally, Unity’s asset store offers Rain AI [4], a free, well-documented and highly regarded AI system that allows for the creation of behavior trees.

For these reasons we recommend the use of behavior trees.

**Goal-oriented Systems**

Goal-oriented systems work by using their knowledge base to determine what action is currently most important to complete [2]. For example, if the antagonist possesses the knowledge *player sighted* it will increase the importance of *chase player*, whereas the knowledge *low on health* would decrease this importance and make *flee* more important.

Because goal-oriented systems can determine high priority objectives they excel when there are a large amount of available actions. Determining the action to take would require a massive decision tree for example, but a goal-oriented system can remain clear and compact even when a large amount of actions are available.

One of the weaknesses of a goal-oriented system lies in that it tends to not take the result of its actions into account, or plan for future actions, making it have a hard time combining multiple tasks. This can in part be resolved by using goal-oriented action planning, an extension of the system. However, this planning assumes the situation goes exactly as the character expects it to go, which can easily be disrupted by actions of the player. As a result, there is no effective way of resolving this issue in a real-time, hostile and unpredictable environment such as PortalBug.

For these reasons we recommend against using goal-oriented systems.

**Existing Examples of AI in Unity**

The engine the game runs on, Unity, is a game engine which has exploded in popularity during the past 5 years. In particular, we will be writing the code using C#, which is powerful enough to support any of the AI systems previously mentioned. More than enough examples of these AI systems exist in other games created using Unity.

For example, recently released Cities: Skyline (*Colossal Order, 2015*) and upcoming RollerCoaster Tycoon World (*Area 52 Games, 2015*) both feature thousands of agents traversing the game simultaneously. These games most likely make use of a behavior or decision tree because their speed would be required for smooth gameplay.
Tactical AIs exist as well, such as in Hearthstone *(Blizzard Entertainment, 2014)* and Scrolls *(Mojang, 2014)* where player and AI battle each other in a card game, likely achieved via a goal-oriented system, to allow the AI to prioritize events.

One of the most similar AIs is featured in Dungeonland *(Critical Studio, 2013)*, which features three players battling monsters in a dungeon. During gameplay, an additional player can take control of the Dungeon-master, capable of setting traps and summoning enemies to thwart the other players. The Dungeonmaster can also be controlled by an AI, making it quite similar to our antagonist, although unlike our antagonist the Dungeonmaster does not navigate the maze himself.

Several AI frameworks for Unity can also be found in its asset store. This includes free options such as the earlier mentioned Rain AI [4], which features a framework for behavior trees, and AI Rule Engine [5], a framework for rule-based AI. It also features paid options, such as Advanced AI Pro [6], priced at $125, which also includes some basic behaviors as well as hearing. Most features of Advanced AI Pro would have to be cut, however, as they don't fit PortalBug. Instead a framework such as SimpleFSM [7] would be a better option, which features a Finite State Machine framework, priced at $10.

The asset that matches our intended use the most is coAdjoint Orbit [8], priced at $100, which allows the creation of an AI that learns the opponents playstyle and adjusts difficulty based on performance. However, coAdjoint is a recent addition to the store and mostly aimed at RPGs and other skill-tree-based games.

**Final decision**

In the end it was decided to use Rain AI. The primary reason for this is that Rain AI is easily integrated into a Unity project and saves a great deal of time spent implementing an own solution or other generic C# AI frameworks such as Stateless [9] or Simple Rule Engine [10]. Furthermore, Rain AI is well documented and highly regarded by developers that have worked with it.

### 3.2. Intelligent Maze

The generation of a proper maze is one of the more important aspects of the game. A good maze will have the player stroll about looking for the exit while still feeling like he is making progress. A bad maze will either be far too short or make the player too disorientated. This means that we will want to tailor mazes for the player that is playing it and the difficulty he has selected. If the player chooses a hard maze, it should be hard, but not impossible for him. However, a more skilled player might prefer an even more difficult maze.

**Maze generation strategy**

When the maze is generated for the player, it will contain either a start or an end node (or both, in very simple cases) and one or more portals. The rest of the maze is filled with normal tiles that might contain a powerup of some kind.

**Recursive strategy and straight paths**

One of the things that has the most impact on whether or not a maze is difficult is the disorientation of the user. The easiest maze will be one with lots of straight corridors, because the user will be more easily able to orient himself on the paths he has already taken and where to go next. A difficult maze will hardly contain any straight corridors at all. The algorithm we consider best suited for the task is a recursive algorithm.

A recursive maze generation algorithm will start at a random point and at each step, go to a point in the maze that was adjacent to the last point and in territory that hasn't been marked yet. Each time a new tile is chosen, that tile is marked. When the path can't be continued (because all adjacent tiles are already marked), the algorithm will backtrack to the last point that had at least one unmarked adjacent tile and continue in that direction.

The advantage about this algorithm is that we can assign a 'momentum' to the random movement. This momentum will make it more likely that the random generator will choose a path in the same direction as the last path, thus creating more straight corners. In an easy maze, the momentum will be high, whereas in a difficult maze the momentum will be low.

**Irregular maze bounds**

Most of the time, parks that contain a maze tend to have a lot of round and slightly irregular forms. This is partially for the aesthetics of viewing the maze from above, but this also has an ulterior goal. Mazes with irregular or unexpected bounds tend to be more difficult.
If a player can keep his bearing in a square maze, he will know roughly where in the maze he is. He can judge this by having visited the presumed walls of the bounds of the maze and will thus have less difficulty navigating about. However, if the presumed walls differ as the user progresses in the maze he will not be able to rely on them for positioning.

For the easiest setting of the maze, it is recommended to take the square maze as a base. For harder settings it is possible to take any basic geometric shape that is not exactly square\cite{11}. For the hardest setting, a series of geometric shapes can be connected to each other with a bottleneck between each shape. Unless the player can identify the bottleneck and simultaneously keep his bearing in the series of irregular sub-mazes, this should be the most difficult setting to navigate in.

In the case for the PortalBug game, it is not necessary to define specific bottlenecks. Since PortalBug incorporates portals to transport the player from one place to another, we can instead use these portals to allow players to navigate around.

**Maze patterns**

Having only the recursive algorithm is a good start and will suffice for moderately difficult mazes. However, to be able to tweak the maze difficulty in terms of form as much as possible, there are some other strategies that need to be kept in mind.

**Rooms**

Some ‘casual’ players will want to play the game for their enjoyment without wanting too much of a challenge. For these players it is important to create a number of landmarks where they can find their bearings again and try other paths of the maze. These open rooms will also allow for the player to try and shake off the antagonist if it is in pursuit of the player. Depending on the difficulty and the size of the maze, there can be multiple rooms and of varying size.

**Vortex**

A vortex is a kind of spiral that leads the player to its center. In this center, the vortex will have a junction of at least 3 paths that can be chosen. A spiral itself is by nature disorienting to the player, so presenting a choice of paths at the center will be tough for players to keep track of\cite{12}. This is mostly due to the fact that the vortex looks mostly the same on the inside.

At the start of the algorithm, before the maze itself is generated, we can instantiate several rooms and vortexes, dependent on the difficulty level that the user chose. These vortexes will be 2-7 lanes deep, depending on location and difficulty. Rooms and vortexes cannot be directly adjacent to one another, because otherwise a path between might not exist.

**Maze adaptability: player variables**

In order to keep our game challenging enough for skilled players and doable enough for lesser ones, we need to generate mazes depending on the behavior of the player. This means we will have to monitor the movement of the player and collect data regarding his decisions. The following list gives a rough estimate of the variables that we will want to collect from the user.

- Time spent in the maze. More time means an easier maze is required.
- The frequency of the direction chosen on a junction. This can be used to tweak the spacial distribution of the powerups.
- The frequency of powerups used. This can be used to tweak the chance that powerups are instantiated.
- The average proximity to the antagonist (not euclidean distance). A closer proximity means the player is likely to require an easier maze.
- The average amount of times tiles are revisited. If a player revisits tiles too often, an easier maze is required.
Maze adaptability: tailored maze generation

For this section it will be assumed that the game consists of a maze with a start node, a maze with an end node and an arbitrary amount of custom mazes between those two. The mazes between will generally not be linear and will contain multiple portals.

One of the intrinsic problems with custom tailored maze generation is that it is difficult to change the difficulty of a maze on the fly.

Intermediate maze adaptation

Intermediate maze adaptation means that whenever a player steps through a portal, the new maze on the other side of the portal will be custom generated for the user. Any former visited mazes will retain their original form whenever they are revisited, and only newly entered mazes will be generated. This however creates a couple of issues as well. In the original project description, it is stated that the antagonist (the spider) spawns at the very end of the mazes, while the player spawns at the start. If mazes are generated as the player progresses through the maze, the antagonist would not be restricted by walls. This would create an initial unfair advantage for the antagonist and possibly unwanted behavior.

Recalculate a tailored maze for the player

Another strategy is to calculate all mazes in advance. This means that the antagonist can start at the very end and can start moving around. As soon as a player goes through a portal, a maze is re-calculated depending on the statistics on the player. Should the antagonist be in the maze that is recalculated, it is important that we ensure that the distance between the antagonist and all the other portals in the maze remain the same. This means that the portals must be relocated in the maze as soon as it is recalculated. Once generated a second time, a maze will retain its form and will not change again if the player goes back and forth again.

Path through the maze

Once the recursive algorithm is done and the maze has been generated, every tile is accessible from any other tile. This means that it does not matter where the start-node is situated. We can choose a completely random point to start the maze.

After the start node has been selected, we can do a breadth-first search of all adjacent tiles that are accumulated when we start off with the start node. This will leave us with a collection of tiles and their respective distance to the start node. The distance is the amount of tiles on the path to get to a tile through the maze and not the euclidean distance. This collection is used to determine either the end-node (if that is located on the same maze) or the portals in the maze. Dependent on the difficulty of the maze, the easier approach to the maze will be to have the portals nearer to the start node than on a more difficult setting[13]. On a difficult setting we would prefer to have one of the tiles the farthest away to house the portals[14]. Ideally, we want to ensure that the portals are distanced from one another as well. This check is easily performed, as we only need to confirm that the space between the portals is obscured by walls.

Perfect and non-perfect mazes

The description of a perfect maze is a maze that contains no cycles. Such a maze can basically be seen as a tree. One very intrinsic property of such a maze is that it is very easy to visit every single tile in the maze without memorizing any part of it at all. By always following the wall to the left or right, one will eventually visit all the tiles in the maze and return to the start again.

Because of this intrinsic property, usually a perfect maze would be considered far easier than a maze that contains cycles in it. However, this is not always the case in PortalBug. Because of the fact that there is an antagonist chasing the player, a perfect maze would leave no way out if the player were to be trapped by the antagonist in a dead end. For this reason, the easier mazes will have more circular paths and less dead ends, thus giving the player a better chance of survival should they be followed by the antagonist. The increase in difficulty because of the circular paths will be negated by the landmarks that the easy maze incorporates. The harder mazes will have fewer circular paths and an encounter with the antagonist will generally be deadlier. It is better to have at least several circular paths in the maze instead of none at all. Even though fewer circular paths make antagonist encounters deadlier, too few will make the maze too easy to solve.

Circular paths can easily be created by removing walls at certain places in the maze. The following things need to be kept in mind while doing this.
No removal of walls on the main path
When the main path is being chosen, it will be one of the furthest away points dependent on the start-position of the maze. If any wall is removed in the main path, this will create a path from the start to the endpoint that may be shorter or of equal length to the original path. To avoid suddenly creating shortcuts to the portals, we must not remove any walls from the main paths that lead the player to any of the portals. Ideally, this situation should also be implemented for the paths between the end-nodes (portals), however, doing this may result in too few circular paths being available. Unless this proves problematic in the amount of walls removed, all paths between all portals should be retained with their original walls.

Prevent creating wall islands
When there are too few walls adjacent to each other, removing a wall could create an obvious island, namely one or two singled out walls that are distinctly separate from the rest of the maze. For example, if a wall is removed that has only 1 wall adjacent on one of the sides, this single wall will stand out because it is now completely isolated. If possible at all, walls that are removed should have at least 3 straight walls or 2 perpendicular walls adjacent to each side to prevent obvious wall islands.

Maze adaptability: power-ups
Another way to adapt the maze to the player is not to change the layout, but the contents of the maze. The maze will contain several powerups for the player. Certain players will prefer different strategies and different powerups that correspond with said strategy. There are certain ways in which we can influence the contents of the maze.

Likelihood of content
If a player is using a specific powerup noticeably more than other powerups, we can tweak the likelihood of that powerup spawning in a maze. The amount of times a player has actually come across a powerup needs to be taken into account as well. For easier settings, a slight increase in frequency should be implemented. In case the user has chosen a harder difficulty setting, the other items should receive more priority than the players’ favorite.

Content positioning bias
When the user has a tendency to choose a specific direction when encountering a junction, this will result in a noticeable bias. At easier difficulty settings, this should not influence the locations of the items spawned. However, at harder difficulty settings, we could use this knowledge to make items more likely to spawn on the opposite direction. This will create a bias in powerups and increase the difficulty for the user. The amount of bias can easily be tweaked.
It is our recommendation to keep the bias to a small number. If the bias is too noticeable, the player will try to adapt and might even use it to his advantage by choosing opposing junction direction in each successive maze. This bias is also considered far more unfair to the player than any former methods of increasing difficulty, since it forces the player to play unbiased.

3.3 Modular Design
A requirement for the product extension is loose coupling of the code. This means that the code must be easily re-usable by different applications that possibly use different frameworks or strategies.

Modular dependencies
The extension must be usable by all other applications that want to incorporate it. In order to do this, we can only rely on the default framework of C#. We cannot use methods from Unity, because there is no guarantee that whichever application implements the extension will be using Unity.
For the most part, we will exchange data with the host application that is formatted according to our custom designed data classes. These classes will implement an interface, so that the host application can easily browse the possibilities for data retrieval.
For events we will use the event-system that C# uses. Users will be able to link methods to our event handlers, after which these event handlers will be coupled and providing the data. Classes that incorporate event-handlers and need data in order to operate will require a Delegate method in the constructor that will provide them with this.
**3.4 EXTRA GAMEPLAY ELEMENTS**

In this section, we are researching which gameplay elements PortalBug could really benefit from. "What makes a game good?" is the main question to start with. There are a lot of different answers to this question as the definition of a "good game" differs for everybody [15]. But there are some keywords that come back every time: challenging, rewarding and flexible [16][17]. We have divided this section into three parts, in each part we get further into one of these keywords. Additionally, horror techniques will also be covered, as the client expressed his wish to make PortalBug more of a horror game.

**CHALLENGING**

A game should be challenging so that the player does not get bored, but it should not be too hard as the player might give up. Mihaly Csikszentmihalyi [18] described in 1990 a mental state called the "Flow". When a person is performing an activity and he is in this mental state, he is fully immersed in a feeling of energized focus, involvement, and enjoyment of the activity. To achieve this, the balance between their skills and the challenge should be exactly right; tough but not too difficult to perform it with success.

Above we discussed how we plan to give the spider and maze artificial intelligence, this will make the game a lot more challenging than it is now. But to make the game even more challenging, we would like to add collectible items as well. We considered two different functions for them:

- All items need to be collected to be able to win the game (for example to activate the portal)
- Collectible items are just for bonus (extra points, power-ups etc.)

The first function would really add to the challenging part of the game as the player cannot only focus on getting through the maze anymore, but he also has to search for these items and he cannot go on without them. On the other hand, we should be aware of making the game not too challenging. After discussing it with the client, we decided that the collectible items would function as a gateway to the bonus level; when all items are collected, the player gets access to the bonus level where he can collect points and power-ups. This way, players can choose for themselves whether to collect the items or not, and thus if they want to make the game more challenging.
REWARDING
The collectible items will already make the game more rewarding as it gives a feeling of accomplishment every time an item is found. But to make the game even more rewarding, we would like to add the following features:

- **Scoreboard** - the game keeps score of the players accomplishments; for example time, points for defeating the spider, etc
- **Music** - for example victory music when defeating the spider
- **Sounds** - happy, rewarding sounds when the player does something good, for example picking up a potion
- **High quality graphics and animations** – we will discuss this in appendix A.1.
- **Bonus levels**

FLEXIBILITY
The third keyword is flexibility. By flexibility we mean giving the player more options or strategies with which to complete the game. Currently the player is only able to walk around the maze, pick up some potions and flee when he encounters the spider. By adding items or power-ups with which the player can defeat the spider, we already offer the player two strategies: avoid or fight the spider. In combination with the opportunity to collect items in order to get access to the bonus level and the chance to collect power-ups, there are even more different ways to finish the game.

We are still not sure what item or power-up we want to give the player to defeat the spider with, but while brainstorming about it we came to the conclusion that we want it to have something to do with the portals.

Another benefit from extra gameplay elements is that it creates extra input variables from the player on which the artificial intelligence extension can respond to.

HORROR TECHNIQUES
Finally, we did some research on what makes games, and other things, scary. During an experiment of Vachiratamporn et al. [19] players indicated that shock sound was the scariest game element in the horror game Slender: The Eight Pages, followed by music. But why is hearings things scarier than actually seeing things? [20] According to evolutionary biologist Daniel Blumstein [21], the answer lies in our animal instincts. Because sound is processed faster by our brain than the information we receive from sight, humans most likely evolved to use sound as a first defense against predators. These predators often use so-called "nonlinear sounds", which include rapid frequencies, nonstandard harmonies and noise. This is why horror movies and games often make use of these nonlinear sounds, as they evoke the biggest emotional reaction. Using this knowledge, we would make the game scarier by adding:

- Scary, nonlinear music
- Sounds that indicate that the spider is close, for example hearing his footsteps
- Shock sound when the spider comes into view

Furthermore, if we look at the 10 most common phobias [22], we are already well on the way with our antagonist being a spider as the number 1 phobia is arachnophobia a.k.a. the fear of spiders. If we have some time left, we would also like to add thunder and lightening to the environment as the fear of thunder and lightening, astraphobia, is the number 6 on the list. Lastly, darkness is a frequently used element in horror movies and games as many people are afraid of the dark. Like with the non-linear sounds, this may also be explained by evolution as most predators hunt at night [23]. To use this in PortalBug, we would remove some of the light points to make the environment darker.

4. DEVELOPMENT METHOD
For the development of this project, we have decided to use the Software Engineering methodology of Scrum [24]. We will use an iterative approach (sprints) for the tasks at hand, and will regularly organize daily stand-ups, where the team meets to discuss recent and near-future progress. Scrum is a developed and stable methodology, and it is widely accepted in the professional community.
4.1. **Scrum**

Each sprint should start with a meeting where the team decides upon a sprint plan, where tasks are distributed to each team member and quantified in terms of Scrum units. Scrum units determine the amount of work a specific task will take to complete. The value of one unit is determined by how much work the most basic task requires to be completed.

Each sprint should end with a Scrum review, where the team meets with the client to show what has been accomplished in the previous sprint and receive feedback on the current product. One of the requirements of the process is that there is always a stable version during the implementation phase. This version of the product must compile and be complete. During the Scrum Review a demonstration of the stable version will be given and during the backlog ordering phase the experimental version can also be demonstrated if beneficial to the discussion.

At the end of each scrum review, the team will have another meeting to review the previous sprint plan. Which tasks were completed and how long it took to complete them will be reported. This allows for improvement in terms of determining the workload of tasks, and for the team to evolve positively. The team will then decide upon a new sprint plan.

4.2. **Testing**

The final product should be thoroughly tested. The code should be loosely coupled, hence the code written by us should be able to compile outside of Unity, such as in Microsoft Visual Studio. As such we will be using Visual Studio to perform tests whenever possible. This should be done using Unit Tests whenever possible. More difficult portions of the code should be tested using mocks, for which we will be using Moq[25]. Sometimes it is considered unfeasible to implement tests in games, and that standard Test Driven Development (TDD) does not work. Various sources however do tell us that this is false [26] [27] [28]; though gameplay itself may not be able to be tested, TDD ensures that large sections of the code work properly. This makes it possible to narrow down the cause of issues as well as ensure that any kind of refactoring does not break existing code. All of this contributes to a stabler and faster game development cycle.

In the end, though, not all parts of games are lend themselves well to testing. Hence, our aim will be to achieve a code coverage of 60%. Furthermore, code is only considered complete when tests have been written for it as well. Therefore, our aim will be to maintain this desired level of code coverage throughout the projects lifetime.

4.3. **Team Roles**

Part of the Scrum methodology is dividing the team into specific roles. We have decided to elect Valentine Mairet as the Product Owner, who is responsible for the communication with the client and determining what the product is supposed to be. The Product Owner has the task to lead the Scrum reviews and lead the team in the right direction. In case the Product Owner is unavailable, we have decided that the Scrum Master should temporarily take on that role. The Scrum Master is mainly responsible for the team and is in charge of leading the product development. Wendy Bolier is our Scrum Master, and she will lead the daily stand-ups and ensure the team works properly.

We have assigned our own roles (independent from Scrum) to the rest of the team. Bas Boellaard is responsible for BitBucket. He manages branches and decides upon when to merge with the master branch. Tom Peeters is responsible for organizing the testing phase, where code and functionalities will be tested.

5. **Conclusion**

At the end of this research phase, we have organised our project the following way.

We will begin by incorporating Rain AI into the loosely coupled extension, which we have already partially developed. We plan on first implementing a basic intelligence for our Antagonist, so it can explore, pass through portals, pick up objects, and pursue the player when in sight. Then, we will proceed on making the game itself intelligent, by creating an algorithm that will generate different mazes, depending on the parameters we feed it.

In the meantime, we want to experiment with 3D modelling using Blender, so when the time comes to focus more on higher quality graphics for PortalBug, we will have the proper experience required.

We will accomplish the tasks given to us using the Scrum methodology, in order to have a strict and structured development process.
BIBLIOGRAPHY


A. APPENDIX

A.1. HIGH QUALITY GRAPHICS

The skeleton game of PortalBug is still very simplistic. The client wants more detailed, higher quality graphics. In this section, we are researching the best way to achieve this.

POLYGON MESHES

Scenes in Unity are created by using polygon meshes [29]. A polygon mesh is a collection of vertices, edges and faces that define the shape of a polyhedral object. In Unity, the faces consist of triangles. Except from some basic shapes such as spheres, cubes, etc., Unity does not provide the ability to create meshes, therefore we will need an external mesh modeling application.

BLENDER

There are many 3D modeling applications available (SketchUp, AutoCAD, Rhinoceros, and many more [30] [31]) and after some research we decided to use Blender 3D. Blender is a free, open-source 3D creation suite [32]. It is used for creating 3D models, animated films, visual effects and even complete games. We decided to use Blender because of the following reasons:

- It is one of the best 3D modeling applications that is free of charge
- The use of Blender for any purpose, including commercially, is permitted [32]
- Blender is a popular program; documentation and help can easily be found on the internet
- Unity natively imports Blender files; When modifying a .blend file, Unity will automatically update the model in the game [33]

TEXTURES

Besides meshes, we will also need textures. Textures are image files that are used to give meshes a colored and/or structured surface [34]. For creating or manipulating these textures we will use image editors that we are already familiar with in order to save time. These image editors are Paint.NET, GIMP, InkScape and Photoshop. It is also possible to download high quality textures that are free for personal and commercial use. For example on www.cgtextures.com.

OPTIMIZATION OF 3D MODELS

In order to keep the file size and rendering time of the game to a minimum, it is important to optimize the 3D models. The optimization of polygon meshes is done by reducing the number of polygons the mesh consists of [35]. Many algorithms exist to select the polygons to be removed while keeping the shape of the model intact. Blender provides several tools [36] to easily reduce the number of vertex. Unity assists in this matter as well, as the import pipeline will compress and optimize our 3D models.

LIGHTS AND SHADOWS

In order to add realism, it is a good idea to add light and shadow effects. Lighting defines the color and mood of a 3D environment [37], making them an essential tool when creating scary environments. There are two types of lighting in Unity: dynamic and baked lighting [38]. Baked lighting is calculated beforehand and saved to the textures, while dynamic lighting is calculated real time when the game is running. Dynamic lighting is more realistic as the lights and shadows can move and change during gameplay. However, dynamic lighting can severely impact performance. To reduce the calculation time of dynamic lighting, there is an option for rendering per vertex instead of rendering per pixel. This method is not as high on performance requirements.

In order to ensure the most realistic graphics while still keeping a good performance, we will decide per object which type of lighting and rendering of the light we will use, dependent on the importance of the object and the ability of the player to walk around it.