Department of Informatics

University of Leicester

CO7201 Individual Project
Retro Game

Final Report

Author: Matthew Hall

Submission Date: 04/09/20

DECLARATION

Name: Matthew Hall

Date: 04/09/20

Table of Contents

1.0 Abstract

The creation of a functional 2D retro platforming game with procedural and infinite level generation through the application of Cellular Automata (CA), in addition to Artificial Intelligence (AI) guided automated testing, satisfied the main aims of the project. All objectives outlined within this report were achieved with the exception of adaptable difficulty which would require restrictions on level design and so was removed due to its potential to interfere with the interesting level variations provided by CA. The testing process highlighted two main rulesets 6 and 10, the most and least effective when attempting to generate levels that could be fully completed by an AI player. Rulesets with a greater number of rules (ruleset 6) each containing a lower cell count were found to produce less varied level characteristics and long flat platforms, increasing ease of completion for the AI. Rulesets with fewer rules (ruleset 10) due to rule duplication that contained higher cell counts, created more unpredictable and complex terrain often beyond the capabilities of the AI player, resulting in low level completion rates.

Analysis of the testing outcomes concluded that CA for level generation is effective in its ability to provide interesting and varied terrain while still maintaining a 55.51% average successful completion rate (with ruleset 6). While this average success rate is not particularly high, it is dependent on the capability of the AI, suggesting this is a much lower estimate of possible level completion for a human player. The perceived quality of the AI based on the test results obtained and the difficulty of a generated level both depend on the level quality in each test case, closely linking all three concepts. To further explore CA as a level generation technique, both AI and user testing should be carried out on the same set of generated levels to gain a more accurate estimation of completion rates.

2.0 Project Aims

The main aim of this project is to create a retro 2D platforming video game incorporating advanced technical concepts such as procedural generation for each level and the application of artificial intelligence. The game was created using the Unity game engine and the C# programming language.

3.0 Motivation

The motivation for completing this project is to explore the possibility of greater level variation in video games, with the application of Procedural Content Generation (PCG) to provide a new challenge to the player with each completed level. Through the creation of a functional game and the use of a PCG method such as Cellular Automata (CA), variation in level terrain can potentially be infinite, adding great replay value, a feature that is often sought after in many modern games. The exploration of methods less widely used for video game level generation may offer insight into interesting approaches for the creation of levels within the platforming genre.

The level design for the game was influenced by more traditional retro platformers such as Super Mario Bros. and Mega Man from the late 1980's. To achieve this design, certain core level features must be present including terrain with platforms of varying size or shape that the player can move across, separated by a series of gaps to fall down and fail the level. These level features are provided through the rules used to constrain the level generation. To provide a flat main section of the level similar to that seen in Figure 1 for example, CA rules must not place a platform cell on top of any other platform that has two background cells above it. Gaps can be created by preventing platforms being placed to the right or left of another with two adjacent background tiles for a two-tile wide gap or alternatively not placed in between two platforms for a single tile gap.

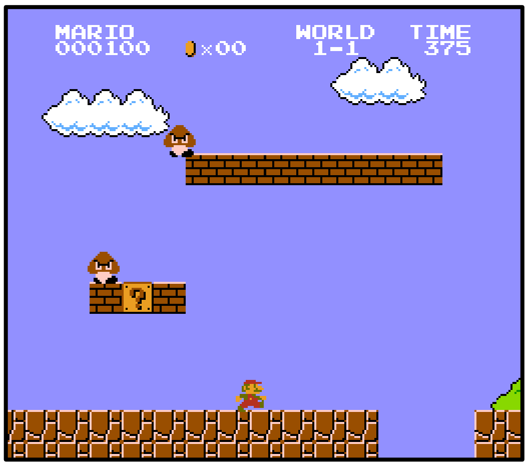


Figure 1: Super Mario Bros. level example displaying a flat main walkable platform with no descending platforms and an example of a gap separating two platforms (World 1-1 (Super Mario Bros.), 2020).

The terrain formed as a result of adjacent rules is likely to be unpredictable and could create more interesting features as seen in Figure 2. This could also be achieved more precisely with rules that consider the placement of platform tiles underneath pre-existing platforms. The addition of a rule ensuring that a platform is not placed under the left most cell would create an overhanging edge or placing a platform tile above the middle cell (on a platform 3 cells in length) would create a step. These rules however would only have the desired effect at positions that matched the exact rule configuration.

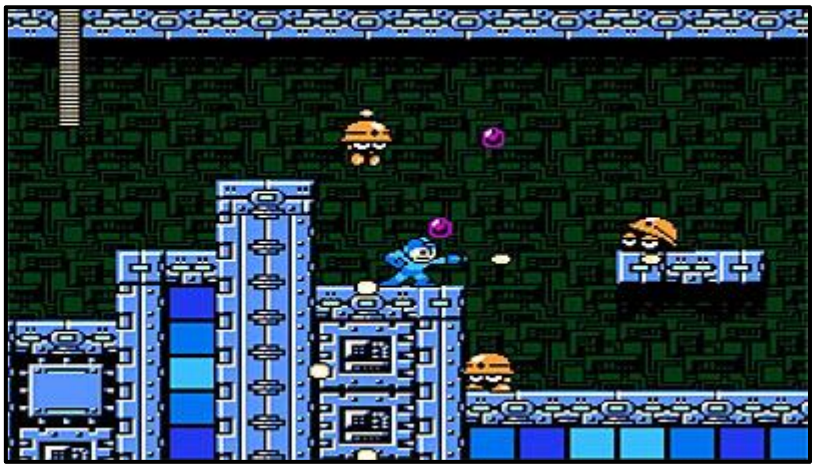


Figure 2: Greater variation in level terrain with more verticality and drops to lower platforms shown in Mega Man (Brown, 2010).

In addition to this, a further understanding of the tools and concepts used within video games development can be achieved as the Unity game engine and C# language are frequently used within industry. The 2D development capabilities of Unity offer a more traditional flat retro game appearance more suited to this project, in comparison to a 3D game viewed from a fixed angle. As the Unity engine deals internally with more routine systems such as physics and animation, this gives the opportunity for more interesting and complex practices such as procedural generation or AI to form the largest part of the project.

4.0 Objectives

The list of Essential, Recommended and Optional requirements in the preliminary report were grouped into objectives relating to the use of PCG, AI and adaptable difficulty, the three main concepts that were the planned focus of this project. Other objectives within this project include more basic concepts such as character movement and general understanding of the C# language and Unity engine. The completion of these objectives marks the full completion of the project.

Objectives
Learn C# game development concepts in the Unity engine.
Develop 2D platformer levels utilising methods for Procedural Content Generation.
Create player character movement and animation.
Develop AI capable of completing the generated levels.
Develop levels that adapt to the players ability.

Table 1: The table above lists the main objectives that incorporate the features of the project requirements detailed within the preliminary report for this project.

5.0 Background Literature

The following section details the background reading throughout the project and the considerations made when selecting methods for PCG, AI and game engine selection.

5.1 Procedural Level Generation

Two possible methods considered for Procedural Content Generation (PCG) within this project were Cellular Automata (CA), typically used for the creation of levels such as cave systems and the method of noise functions to produce random terrain data from pseudo-random noise, which can then be rendered directly or stored as colour coded elevation in a heightmap image (Rose and Bakaoukas, 2016).

The more complex Cellular Automata method requires a set of rules to govern the transition of states for each cell from either on or off, a platform or gap in this case. While heightmaps may be more easily implemented to create basic terrain, CA offers the ability for cells to determine their own state based on their neighbouring cells and the transition rules (Macedo and Chaimowicz, 2017). The comparison between purely random generation and the use of CA by (Johnson et al, 2010) shows the potential for CA to generate playable levels with greater variation beyond simply varying the height of a series of platform columns to form the terrain. In addition, the ruleset utilised by CA offers some level of control while still maintaining an element of randomness.

5.2 Artificial Intelligence

Search algorithms for the creation of AI such as Minimax or the similar method Monte Carlo Tree Search (MCTS) were initially considered as suitable options for platforming game AI. Upon further research however, both methods were more suited to complex games with a large number of possible moves such as chess or checkers and were far more in depth than required for basic traversal of a 2D platformer (Chaslot et al, 2008). As the creation of an AI was beyond the scope of this project, a package was needed to incorporate AI into the game. Therefore, the availability of packages with adequate documentation that could be integrated with the Unity engine proved to be a deciding factor when selecting the ideal AI method.

After further development of gameplay elements and discussion regarding AI that utilises sensor input to inform decision making behaviour, the use of Astar pathfinding was considered the most appropriate option. The Astar algorithm provides a high level of accuracy when determining the shortest path to the target, which is a useful feature in the event that the AI drops down into an area of more complex terrain. To find the shortest path the following function is utilised:

$$f_{(n)} = g_{(n)} + h_{(n)}$$

Where n is the next node in the path, $g_{(n)}$ is path cost from the start node to n and $h_{(n)}$ is the cost of the cheapest path from n to the end target node (Rafiq et al, 2020). A path is created by following the set of nodes with the lowest $f_{(n)}$ value in each case.

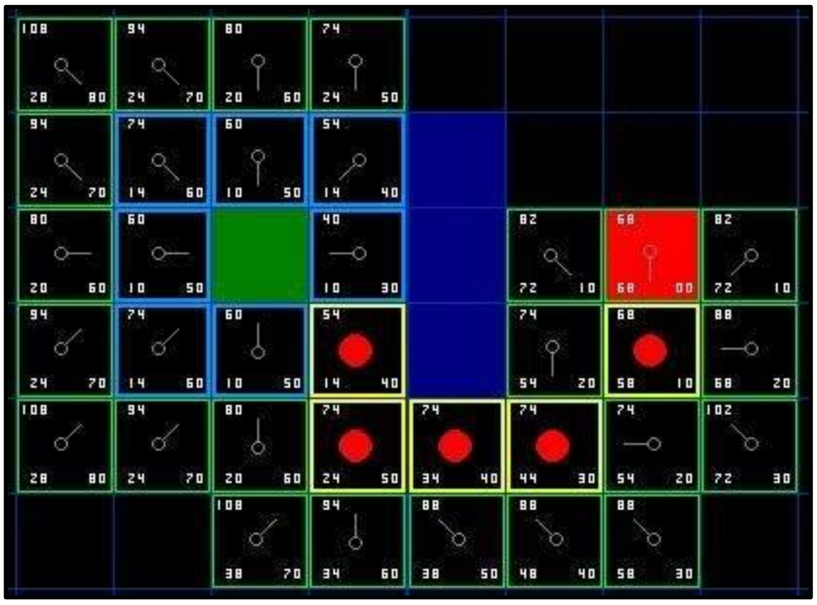


Figure 3: Visual representation of A* pathfinding with the function values shown in each cell, $f_{(n)}$ (top left), $g_{(n)}$ (bottom left) and $h_{(n)}$ (bottom right) and the shortest path marked to the target node (Lester, 2003).

Once a node with the lowest $f_{(n)}$ value has been selected, the $g_{(n)}$ and $h_{(n)}$ values for all nodes surrounding the new node are calculated again and the process repeats until the most direct route to the target has been found as shown in Figure 3 (Sebastian Lague, 2014). The implementation of this function is contained within the scripts offered by the A* Pathfinding Project package (Granberg, 2009) used within this project.

5.3 Unity Development

The Unity engine offers the flexibility of professional game engine software and is a platform both widely recognised and used within industry. The analysis of game engines (Christopoulou and Xinogalos, 2017) shows that Unity provides similar features to other leading industry engines such as Unreal but offers a greater range of accessible tutorials for C# coding and a simplified user interface. It was concluded that Unity is more suited to beginners but can still produce professional quality video games. The level of guidance offered with Unity is particularly useful due to the requirement of suitable familiarisation with the software early on in the project to continue developing more technically advanced concepts throughout.

6.0 Game Design in Unity

The platforming game developed during this project adopts a typical side scrolling design from left to right across the screen with terrain that occupies the lower half of the level scene giving the player character room to jump and manoeuvre in the air. Gaps in the terrain offer a challenge to the player requiring timing and skill to overcome these obstacles and progress through the level reaching the end. Upon falling from a platform, the level and game timer will reset, restarting the game but the completion of the level will cause the timer to persist, in both cases new level terrain will be generated. This gameplay process can continue infinitely while the game is running and the aim of the player is to complete as many levels as possible in the shortest amount of time before inevitably failing.

Where possible within development, C# scripting was used to provide all major functionality, including levels generated at runtime and therefore not visible within the Unity editor window until the game is running. As a result, other objects such as the player were positioned correctly at the start of the level within the PlayerController.cs script rather than the more basic drag and drop approach for placing a game object inside the editor.

Each created C# script must first be added to its respective created game object as a component to give functionality, for example the PlayerController.cs script is attached to the PlayerCharacter object. Basic components such as Transforms to monitor the position of an object and Renderers allowing sprites or other graphics to be visible are often added by default when an object of a specified type is created in the Unity editor. This process forms the basis for connecting the implementation of code to the more visual design aspects of the game engine. The diagram below (Figure 4) details the dependencies and interactions between each of the C# scripts included within the project.

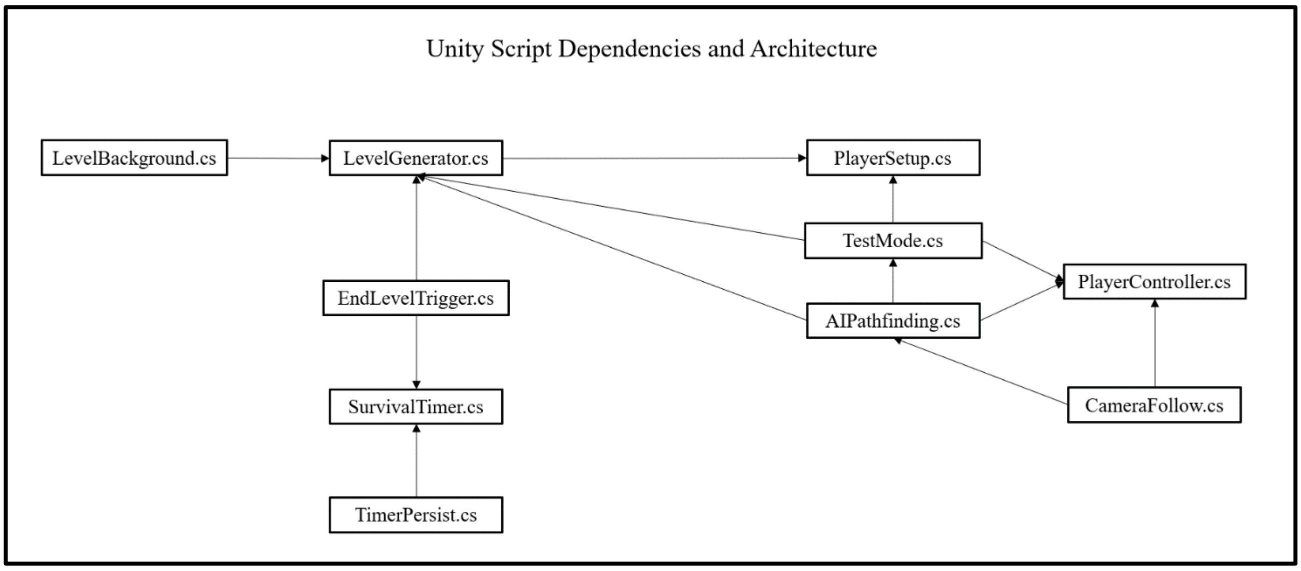


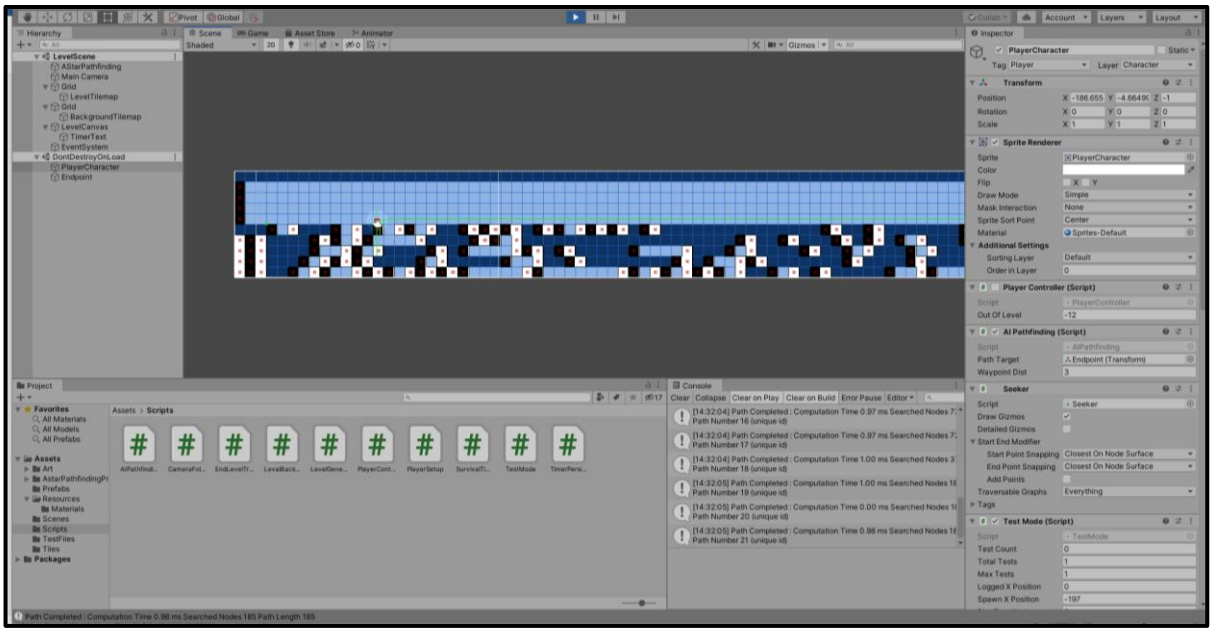
Figure 4: Displays the directional dependencies represented by arrows for all 10 C# project scripts.

Many elements of the functionality within the game require the existence of specific scripts to access methods and variables or to produce a combined function as in the case of running the test mode. The execution of LevelGenerator.cs is dependent on the existence of the PlayerSetup.cs script as before the level ruleset can be saved successfully, the testRuleList array within PlayerSetup.cs must be initialised. The activation of the TestMode.cs script is dependent on the ActivateTest method within PlayerSetup.cs and the activation of testing also enables the AIPathfinding.cs script. Both TestMode.cs and AIPathfinding.cs are dependent on LevelGenerator.cs as they both require a generated level for the character to move across to generate a route for pathfinding or begin logging the distance travelled during the testing process. Furthermore, TestMode.cs and AIPathfinding.cs utilise the RestartLevel method and as a result share a dependency with the PlayerController.cs script. The cameraFollow.cs script attached to the character object uses character movement to dictate the motion of the camera and so is dependent on the movement of either a human player (PlayerController.cs) or an AI (AIPathfinding.cs).

While the SurvivalTimer.cs has no dependencies of its own, both EndLevelTrigger.cs and TimerPersist.cs require the output from SurvivalTimer.cs to visually represent their functionality. EndLevelTrigger.cs uses the DontDestroyOnLoad method available in Unity to cause the current instance of the timer to persist when the character enters the collider trigger on the Endpoint object at the end of the level. As a new instance of the LevelCanvas game object holding the timer is created with each new level, the TimerPersist.cs script removes the new instance immediately. The EndLevelTrigger.cs and LevelBackground.cs scripts also have a dependency on LevelGenerator.cs as without a level providing a valid path the endpoint cannot be reached. To provide the white background that fits the dimensions of the level, the height and width initialised within LevelGenerator.cs need to be accessible.

Game assets such as the sprites for the player, platform and background cells were created outside of Unity using basic drawing software Paint.net as three coloured cubes red, black and white respectively. These sprites were imported to Unity and Tile assets were created for the platform and background cells to allow the SetTile() Unity method to place these blocks within the scene to form an interactable level. The option of including a more visually interesting player sprite was available by importing a spritemap from the Unity website but would have required further attention for elements such as basic animation using editing tools provided within Unity and checking that the sprite changed dependent on the movement direction. As the project focuses on more technical aspects a simple cube character was deemed sufficient and is in keeping with the minimalist style of the level design.

Building a gameplay scene inside the Unity editor utilises a series of created game objects as seen in the Hierarchy on the left side of Figure 5 below. Each of these objects are assigned to layers that determine their order within the scene. The PlayerCharacter example shown exists in the Character layer (labelled in the top right corner in the Inspector panel), which appears in front of the Background layer (for the white cells on the BackgroundTilemap) and allows the character to always be visible when playing the game. This applies to other objects such as the LevelTilemap (for platforms) which also remains at the forefront of the scene. In certain cases where specified layers need to be accessed, for example when selecting which layers can be detected by a raycast or the layers to search for walkable and non-walkable nodes for pathfinding (shown in the scene view at the centre of Figure 5) a layer-mask variable can be created to isolate single or multiple layer instances.



```
[14:32:04] Path Completed : Computation Time 0.97 ms Searched Nodes 7:
Path Number 16 (unique id)
[14:32:04] Path Completed : Computation Time 0.97 ms Searched Nodes 7:
Path Number 17 (unique id)
[14:32:04] Path Completed : Computation Time 1.00 ms Searched Nodes 3'
Path Number 18 (unique id)
[14:32:05] Path Completed : Computation Time 1.00 ms Searched Nodes 1
Path Number 19 (unique id)
[14:32:05] Path Completed : Computation Time 0.00 ms Searched Nodes 1
Path Number 20 (unique id)
[14:32:05] Path Completed : Computation Time 0.98 ms Searched Nodes 1
Path Number 21 (unique id)

Path Completed : Computation Time 0.98 ms Searched Nodes 185 Path Length 185
```

```
PlayerCharacter                       Static
Tag Player                Layer Character
Transform
Position        X -186.655  Y -4.6649  Z -1
Rotation        X 0         Y 0        Z 0
Scale           X 1         Y 1        Z 1
Sprite Renderer
Sprite                PlayerCharacter
Color
Flip                  X  Y
Draw Mode             Simple
Mask Interaction      None
Sprite Sort Point     Center
Material              Sprites-Default
Additional Settings
  Sorting Layer       Default
  Order in Layer      0
Player Controller (Script)
Script                PlayerController
Out Of Level          -12
AI Pathfinding (Script)
Script                AIPathfinding
Path Target           Endpoint (Transform)
Waypoint Dist         3
Seeker
Script                Seeker
Draw Gizmos           ✓
Detailed Gizmos
Start End Modifier
  Start Point Snapping  Closest On Node Surface
  End Point Snapping    Closest On Node Surface
  Add Points
Traversable Graphs    Everything
Tags
Test Mode (Script)
Script                TestMode
Test Count            0
Total Tests           1
Max Tests             1
Logged X Position     0
Spawn X Position      -197
```

Figure 5: The Unity editor software displaying the Hierarchy (left), Inspector panel (right) and the scene view window (centre).

6.1 Level Design

Initial level design ideas intended to provide more generic flat platforms at varying heights, with the use of Cellular Automata (CA) however, level generation provided much more varied and visually interesting terrain due to the unpredictable nature of CA. While the configuration of surrounding cells in a Moore neighbourhood (a 3x3 grid with the target cell at its centre) impact the possible change in state of the target cell, this only considers a single cell in each direction surrounding the cell at the centre and does not consider cell configurations beyond this. As a result, the outcome of an applied rule may connect with an adjacent rule in an unexpected or problematic way that cannot be controlled without heavy constraints, which would negate the benefits of interesting terrain created by CA. This in turn decreases the level of certainty that each platform in a generated level will be reachable by the player.

To ensure a level format that has suitable space for the character when jumping and moving through the air, generation with CA was confined to the lower half of the scene to prevent platforms being placed randomly across the entire playable area. In addition, a border of platform cells was placed along the top and sides of the level to reduce the possibility that the player would unintentionally move outside of the level bounds. Collider components are attached to the platform tiles to allow for a solid surface that the player can move across while white background tiles contain no colliders and do not impede player movement.

The connection of adjacent CA rules produces common level characteristics and features often seen across multiple rulesets. The series of features shown below (Figure 6) are all valid terrain configurations with the exception of test 24 which exceeds the maximum horizontal jump distance with no alternative routes. With a maximum jump height and distance of two cells, test 5 shows a three cell width gap but only a two cell high jump from the lower platform, plausible for both an AI and human player. Test 6 is simple for a human player but an AI would likely become stuck under the overhanging platform with only a single cell space to return to the surface. Test 8 offers two possibilities for completion along the surface or through the middle, the middle route would likely be too confined for an AI and cause issues. Falling to the left of the three cell high column provides a failure condition for both a human and AI as there are no alternate paths. The gap in test 9 is an extremely challenging manoeuvre for a human player and impossible for the AI used within this project due to high level of accuracy and timing required for a diagonal jump two cells high and two cells wide, the absolute limit of the game character. Test 10 provides a valid route across the surface requiring landing accuracy for the single cell platform, which is within the ability of the AI in this case but falling into the gap provides a failure condition similar to test 8.

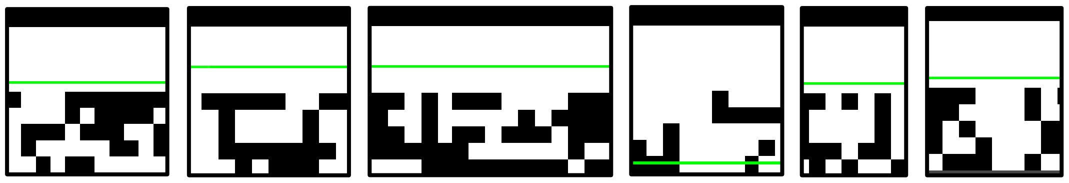


Figure 6: A selection of terrain characteristics from ruleset 10, tests 5, 6, 8, 9, 10 and 24.

6.1.1 Ruleset Development

Initially rulesets were coded directly into the LevelGenerator.cs script, this made for cumbersome and repetitive code but provided a quick platform to test different ruleset concepts, with the idea of unobstructed vertical columns in rules generating gaps and horizontal rows generating optional pathways or platforms depending on the position of the rows. This allowed for a quick and effective understanding of rule application and a visual result within the level to be gained.

For the purpose of efficiency, rulesets were instead generated automatically at random through a process of selecting a number of cells for a given rule and placing them randomly within a Moore neighbourhood grid, checking for each created rule at every cell. The target cell was changed at random to either a platform or a blank background cell in each case as in the example below (Figure 7). The automated random approach reduces the level of control over the rulesets and therefore increases unpredictability but allows for efficient testing with larger sample sizes to more accurately determine the suitability of CA for level creation.

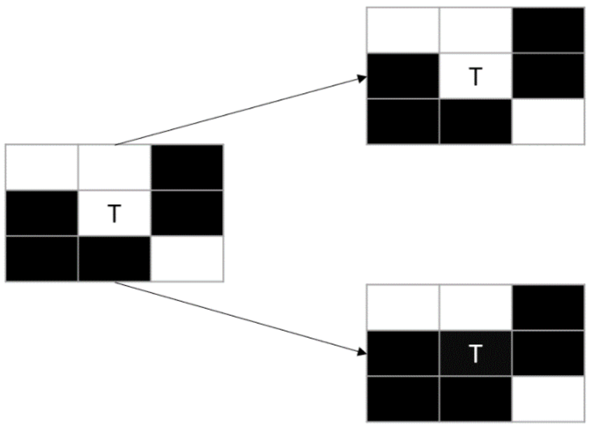


Figure 7: An example Cellular Automata rule within a 3x3 Moore neighbourhood grid with the two possible configurations resulting from a change in the target cell.

6.2 Character Design and Movement

During the early stages of development, the player character game object was instantiated at the beginning of each new level in the PlayerController.cs script, rather than created as a permanent object within Unity and all relevant components were added at the point of instantiation. This caused issues with the CameraFollow.cs script as both scripts started at the same time, the camera could not detect a player object to follow. To overcome this, the player was created as a permanent object and positioned outside of the camera bounds and moved into position immediately upon starting the game. Player object components such as the Rigidbody and Boxcollider were added inside the Unity editor rather than as part of a script to prevent an issue causing duplicate components on level restart.

The player can move in both the left (negative) and right (positive) x-axis directions while in contact with a platform and with directional in-air control while jumping, which allows the player some margin of error to correct a poorly timed jump, preventing frustration and constant level failure. This directional control is only possible within the trajectory of the player character (the time taken for the jump to be completed) so a limit on the possible horizontal distance travelled is automatically imposed. The automated AI guided test character has the same capabilities as the player with regards to maximum jump height and the requirement to be grounded to initiate a jump. The execution of movement while testing was active was determined by a series of sensors shown in Figure 8 below.

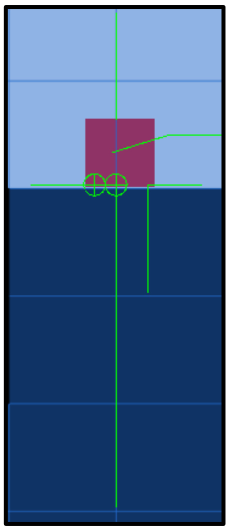


Figure 8: The visual representation of terrain raycast sensors attached to the character object, checkR (right), checkFR (front-right positioned downward), checkLand (downward-middle), checkBM (bottom-middle circle), checkBL (bottom-left circle), checkL(left) and checkT (top).

For positive x-axis direction movement, checkR detects any incoming obstacles and the sensor is shorter to navigate more confined spaces when needed. The checkFR sensor detects gaps, jumping in an attempt to reach another platform and the checkLand raycast checks for suitable landing platforms below while in the air, dropping vertically if a platform is detected. The circle colliders checkBM and checkBL detect contact with the platform to enable jumping, while checkL detects obstructions to the rear if moving backward and jumps attempting to return to the surface. The checkT sensor is responsible for detecting overhanging platforms signalling that the character should return to the surface.

6.3 AI Testing

To create an AI capable of attempting to complete the procedurally generated levels and test the quality of each level, the A* Pathfinding Project (Granberg, 2009) was integrated into the project. Only two scripts from the package were utilised, the Seeker and Pathfinder. The Pathfinder script was attached to a new game object and provided a grid graph overlay (Figures 9, 10 and 11) to denote reachable (blue) and non-reachable (red) cells of the level by placing colliders to detect the character at each point. The collider diameter was adjusted to achieve the closest possible correct grid layout, a diameter of 0.97 gave the most suitable fit (Figure 11) with an immediate change to Figure 10 at 0.98 and Figure 9 from 0.99 – 1.0. Diameters lower than 0.97 would not recognise any walkable areas.

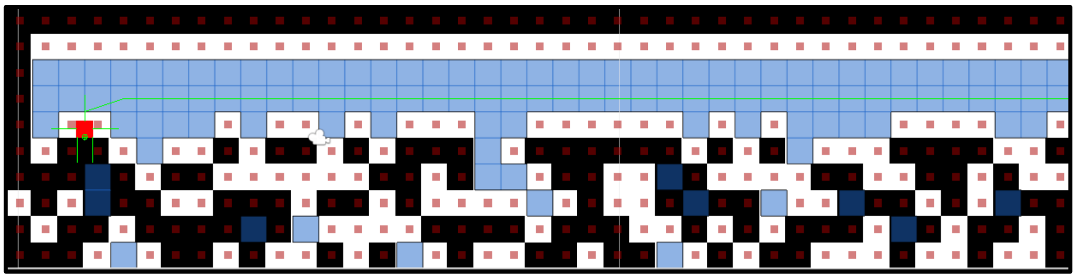


Figure 9: Collider diameter 1.0 producing non-reachable cells above the platforms, this is incorrect as the path should follow along the platforms where the AI will move.

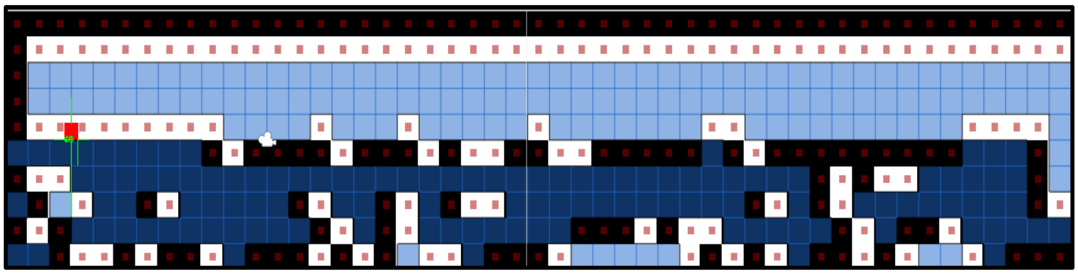


Figure 10: Collider diameter 0.98 produces a generally better fit than 0.97 but for levels generated with a long flat starting platform the unwalkable cells prevent a path from being generated.

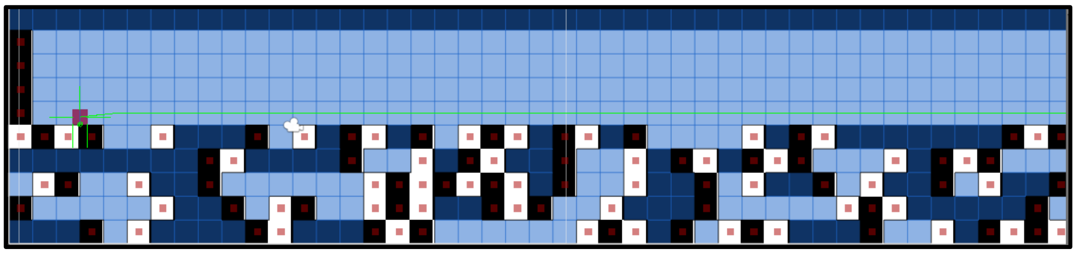


Figure 11: Collider diameter 0.97, a small change in diameter produces drastic change in reachable cell placement. This is still not ideal as some platform cells are marked as reachable but this does provide the expected path.

For the grid graph to detect the reachable and non-reachable cells the Pathfinder script relies on a layer mask to indicate which layer contains the obstructions. As a result, the white background cells were generated on a new tilemap on a separate layer to the platforms so as not to interfere with graph generation. The poor graph fit in some cases is likely due to the complexity of the generated levels as more simplistic or uniform levels with more concentrated white or black cells would provide a better fit.

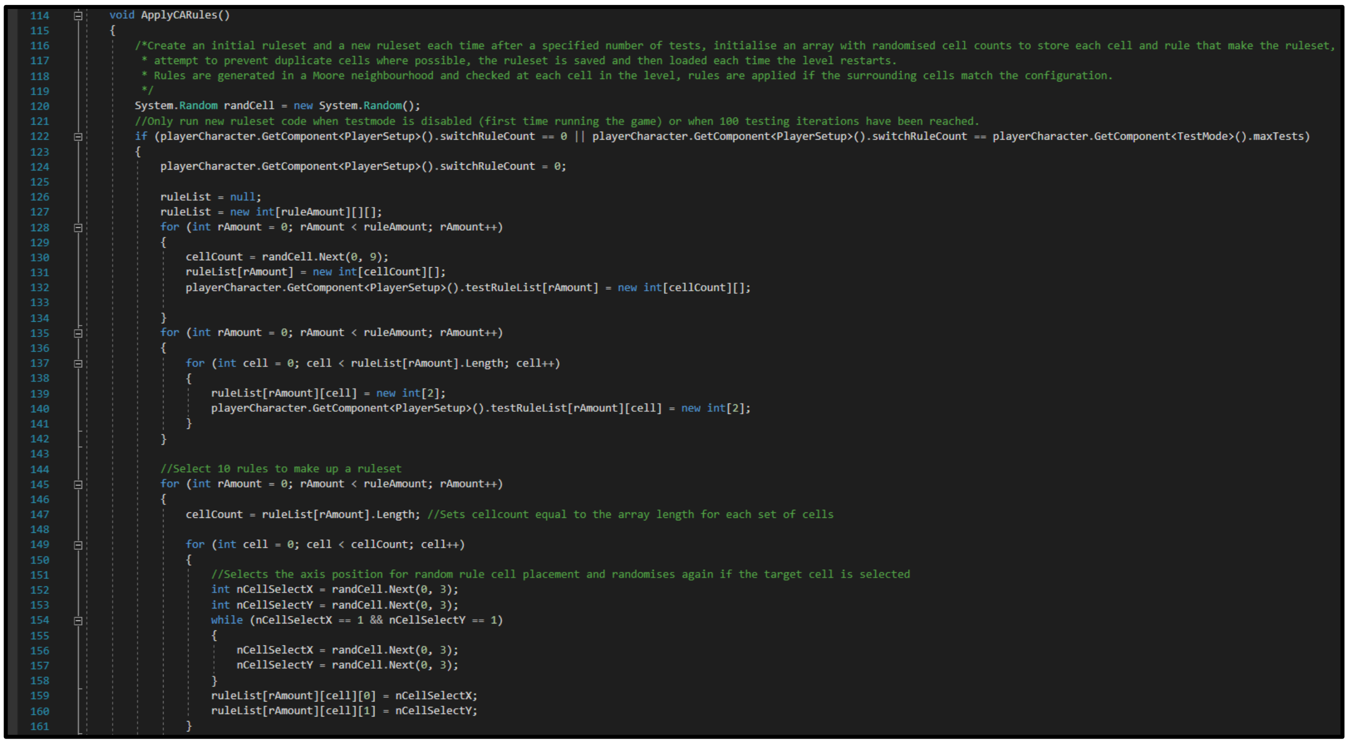
The Seeker script attached to the PlayerCharacter game object generates a path to the target object and is responsible for drawing the visual representation of the path through the use of gizmos (the green path line seen in Figures 9, 10 and 11). A custom AIPathfinding.cs script based on the Brackeys 2D pathfinding tutorial (Brackeys, 2019) was created in place of the default package script to allow the pathfinding and AI movement to be included in a single script for convenience and access to common variables. A game object representing the endpoint of the level was added as the pathfinding target and the AI character moved in the direction of the generated path with the aid of sensors to detect obstructions or platform gaps in front (when moving right) or behind (when moving left).

7.0 Implementation of Features

The following sections will discuss in detail the function of C# coded scripts created for each of the main project features, Cellular Automata, AI and testing, presented with code snippets for illustration.

7.1 Cellular Automata

The implementation of Cellular Automata (CA) is divided into several sections within the LevelGenerator.cs script, first the cell positions are defined by simply iterating through each cell of the levelGrid array representing the size of the playable level and assigning 1's (denoting platform cells) and 0's (for blank cells), this provides an initial random starting state for the level. To create a surface that exists within the game that the player can walk along and collide with, each position assigned as 1 is added as a tile that holds a collider on a TileMap object inside the Unity editor. The CA rules are then created using the code below (Figures 12, 13, 14, 15 and 16).



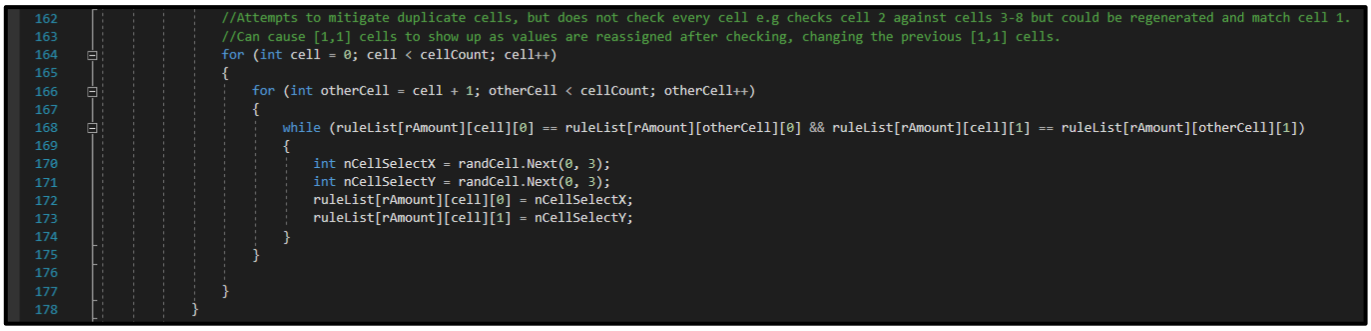
```
114  void ApplyCARules()
115  {
116      /*Create an initial ruleset and a new ruleset each time after a specified number of tests, initialise an array with randomised cell counts to store each cell and rule that make the ruleset,
117       * attempt to prevent duplicate cells where possible, the ruleset is saved and then loaded each time the level restarts.
118       * Rules are generated in a Moore neighbourhood and checked at each cell in the level, rules are applied if the surrounding cells match the configuration.
119       */
120      System.Random randCell = new System.Random();
121      //Only run new ruleset code when testmode is disabled (first time running the game) or when 100 testing iterations have been reached.
122      if (playerCharacter.GetComponent<PlayerSetup>().switchRuleCount == 0 || playerCharacter.GetComponent<PlayerSetup>().switchRuleCount == playerCharacter.GetComponent<TestMode>().maxTests)
123      {
124          playerCharacter.GetComponent<PlayerSetup>().switchRuleCount = 0;
125
126          ruleList = null;
127          ruleList = new int[ruleAmount][][];
128          for (int rAmount = 0; rAmount < ruleAmount; rAmount++)
129          {
130              cellCount = randCell.Next(0, 9);
131              ruleList[rAmount] = new int[cellCount][];
132              playerCharacter.GetComponent<PlayerSetup>().testRuleList[rAmount] = new int[cellCount][];
133
134          }
135          for (int rAmount = 0; rAmount < ruleAmount; rAmount++)
136          {
137              for (int cell = 0; cell < ruleList[rAmount].Length; cell++)
138              {
139                  ruleList[rAmount][cell] = new int[2];
140                  playerCharacter.GetComponent<PlayerSetup>().testRuleList[rAmount][cell] = new int[2];
141              }
142          }
143
144          //Select 10 rules to make up a ruleset
145          for (int rAmount = 0; rAmount < ruleAmount; rAmount++)
146          {
147              cellCount = ruleList[rAmount].Length; //Sets cellcount equal to the array length for each set of cells
148
149              for (int cell = 0; cell < cellCount; cell++)
150              {
151                  //Selects the axis position for random rule cell placement and randomises again if the target cell is selected
152                  int nCellSelectX = randCell.Next(0, 3);
153                  int nCellSelectY = randCell.Next(0, 3);
154                  while (nCellSelectX == 1 && nCellSelectY == 1)
155                  {
156                      nCellSelectX = randCell.Next(0, 3);
157                      nCellSelectY = randCell.Next(0, 3);
158                  }
159                  ruleList[rAmount][cell][0] = nCellSelectX;
160                  ruleList[rAmount][cell][1] = nCellSelectY;
161              }
```

Figure 12: Lines 114 to 161 of the LevelGenerator.cs script detailing the creation of the ruleList array and the generation of each cell within a rule.

The three dimensional "Jagged" array ruleList is used to represent a full ruleset, as each ruleset contains 10 rules but each rule can include any number of cells from 0-8 (where 8 is the maximum possible surrounding cells within a Moore neighbourhood), an array containing arrays of varying sizes is required. The fixed ruleAmount variable initialises the first array to size 10 and then iterates through the array adding a new integer array of a size based on the number of cells for each rule. Each cell was then assigned as an array of size 2 to represent the X and Y axis coordinates of surrounding cells. As the same ruleset was needed to generate 100 level iterations for the purpose of testing, an array of the same dimensions was created within the PlayerSetup.cs script for the purpose of saving and loading the same ruleset until the required number of tests was complete (dependent on the condition on line 122). (Lines 126 to 142).

Each of the rules were produced randomly by creating coordinates ranging from [0,0] to [2,2], denoting a surrounding 3x3 grid of cells with a target cell at its centre. Checks are carried out to attempt to prevent the occurrences of target cell positions ([1,1]) within rules by randomizing the number again within the while loop (lines 145 – 161).

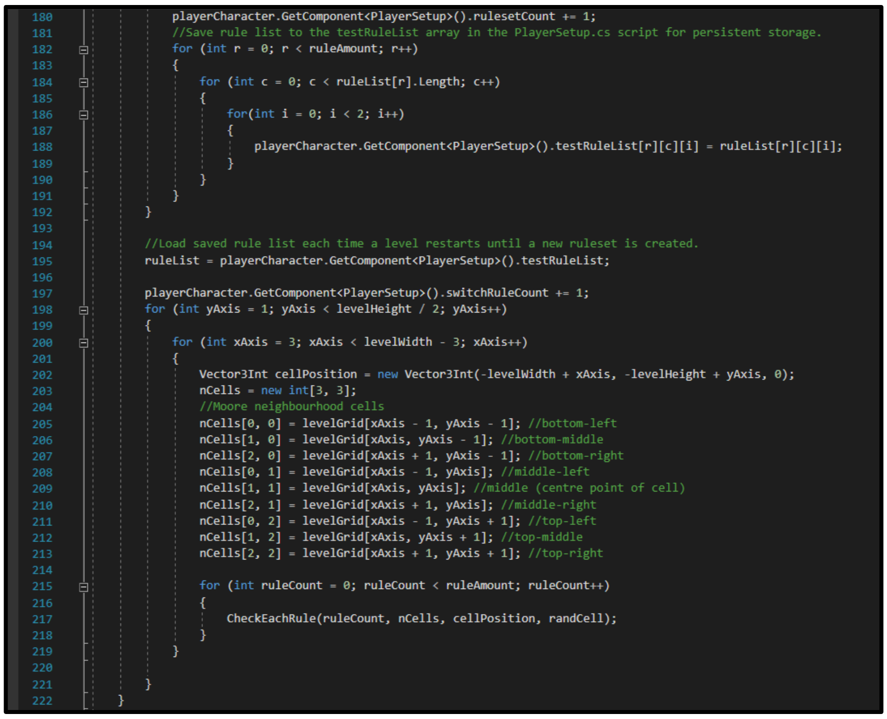
The same concept is applied in Figure 13 below to check each cell against the others within a given rule. Iterating through each rule checks every cell, however, in the event cell 1 does not match any other cells such as 2-8 but cell 2 matches a cell from 3-8 and is randomised again there is a chance that the newly generated cell could match cell 1, as cells are not checked against the previous entries in the list (lines 164 – 177). In the analysis section of this report duplicate cells are omitted and rules are considered to have smaller cell counts where applicable.



```
162                 //Attempts to mitigate duplicate cells, but does not check every cell e.g checks cell 2 against cells 3-8 but could be regenerated and match cell 1.
163                 //Can cause [1,1] cells to show up as values are reassigned after checking, changing the previous [1,1] cells.
164                 for (int cell = 0; cell < cellCount; cell++)
165                 {
166                     for (int otherCell = cell + 1; otherCell < cellCount; otherCell++)
167                     {
168                         while (ruleList[rAmount][cell][0] == ruleList[rAmount][otherCell][0] && ruleList[rAmount][cell][1] == ruleList[rAmount][otherCell][1])
169                         {
170                             int nCellSelectX = randCell.Next(0, 3);
171                             int nCellSelectY = randCell.Next(0, 3);
172                             ruleList[rAmount][cell][0] = nCellSelectX;
173                             ruleList[rAmount][cell][1] = nCellSelectY;
174                         }
175                     }
176 
177                 }
178             }
```

Figure 13: Lines 162 -178 of the LevelGenerator.cs script continuing the checks from lines 145-161.

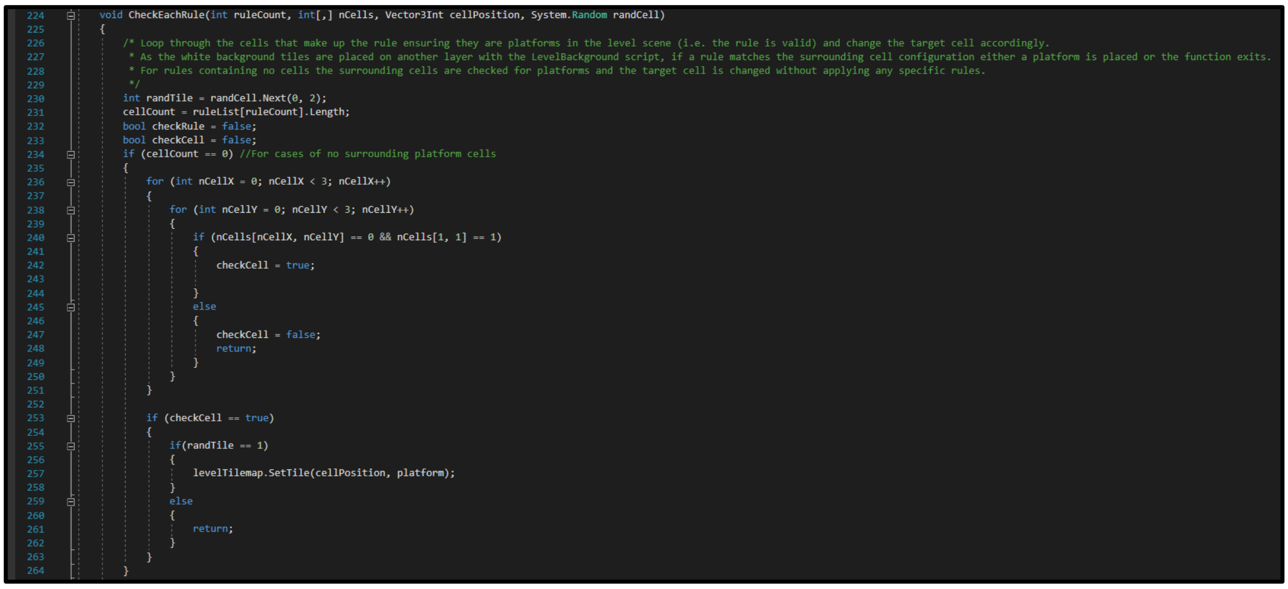
Each element of ruleList is saved to each position in the testRuleList array in the PlayerSetup.cs script (lines 182 to 192) shown in Figure 14 below and is still within the main IF condition for ruleset creation which runs dependent on the number of tests completed. As the ruleset creation will run every time when outside of test mode, a ruleset is always available to load immediately after the start-up of the game. When test mode is activated the ruleset is loaded and then applied (lines 195 to 221). For each iteration through each cell within the bottom half of the levelGrid array leaving space for movement in the top half of the level, a Moore neighbourhood is created around the target cell position. The configuration of the cells surrounding the target are then checked against every rule within the ruleset with the CheckEachRule method.



```
180                 playerCharacter.GetComponent<PlayerSetup>().rulesetCount += 1;
181                 //Save rule list to the testRuleList array in the PlayerSetup.cs script for persistent storage.
182                 for (int r = 0; r < ruleAmount; r++)
183                 {
184                     for (int c = 0; c < ruleList[r].Length; c++)
185                     {
186                         for(int i = 0; i < 2; i++)
187                         {
188                             playerCharacter.GetComponent<PlayerSetup>().testRuleList[r][c][i] = ruleList[r][c][i];
189                         }
190                     }
191                 }
192             }
193 
194             //Load saved rule list each time a level restarts until a new ruleset is created.
195             ruleList = playerCharacter.GetComponent<PlayerSetup>().testRuleList;
196 
197             playerCharacter.GetComponent<PlayerSetup>().switchRuleCount += 1;
198             for (int yAxis = 1; yAxis < levelHeight / 2; yAxis++)
199             {
200                 for (int xAxis = 3; xAxis < levelWidth - 3; xAxis++)
201                 {
202                     Vector3Int cellPosition = new Vector3Int(-levelWidth + xAxis, -levelHeight + yAxis, 0);
203                     nCells = new int[3, 3];
204                     //Moore neighbourhood cells
205                     nCells[0, 0] = levelGrid[xAxis - 1, yAxis - 1]; //bottom-left
206                     nCells[1, 0] = levelGrid[xAxis, yAxis - 1]; //bottom-middle
207                     nCells[2, 0] = levelGrid[xAxis + 1, yAxis - 1]; //bottom-right
208                     nCells[0, 1] = levelGrid[xAxis - 1, yAxis]; //middle-left
209                     nCells[1, 1] = levelGrid[xAxis, yAxis]; //middle (centre point of cell)
210                     nCells[2, 1] = levelGrid[xAxis + 1, yAxis]; //middle-right
211                     nCells[0, 2] = levelGrid[xAxis - 1, yAxis + 1]; //top-left
212                     nCells[1, 2] = levelGrid[xAxis, yAxis + 1]; //top-middle
213                     nCells[2, 2] = levelGrid[xAxis + 1, yAxis + 1]; //top-right
214 
215                     for (int ruleCount = 0; ruleCount < ruleAmount; ruleCount++)
216                     {
217                         CheckEachRule(ruleCount, nCells, cellPosition, randCell);
218                     }
219                 }
220 
221             }
222         }
```

Figure 14: Lines 180 to 222 of LevelGenerator.cs, including saving and loading of rulesets and the generation of the Moore neighbourhood.

The CheckEachRule method illustrated in Figure 15 below, checks a given cell for both rules containing no cells and each configuration of every rule. In the event that each position within the Moore neighbourhood is a blank cell, the target cell is randomly allocated as either a blank cell or platform giving two possible outcomes. The check for surrounding cells is determined by marking a checking variable as true for each blank position if a platform cell is not found and returning to the loop (on line 215 of Figure 14) to check the next rule if the checking code detects a platform or the target cell is successfully changed.

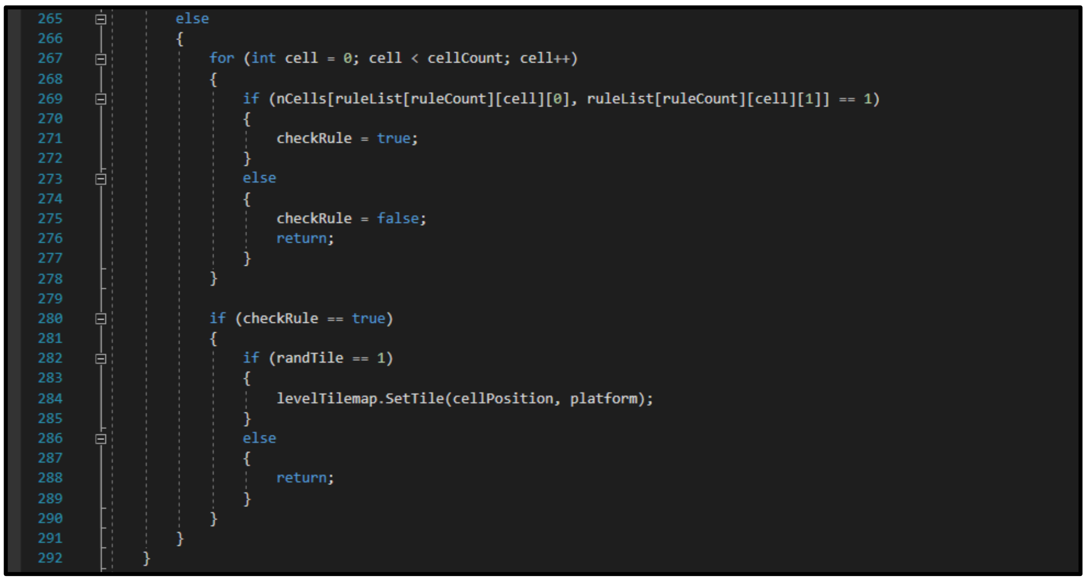


```
224     void CheckEachRule(int ruleCount, int[,] nCells, Vector3Int cellPosition, System.Random randCell)
225     {
226         /* Loop through the cells that make up the rule ensuring they are platforms in the level scene (i.e. the rule is valid) and change the target cell accordingly.
227          * As the white background tiles are placed on another layer with the LevelBackground script, if a rule matches the surrounding cell configuration either a platform is placed or the function exits.
228          * For rules containing no cells the surrounding cells are checked for platforms and the target cell is changed without applying any specific rules.
229          */
230         int randTile = randCell.Next(0, 2);
231         cellCount = ruleList[ruleCount].Length;
232         bool checkRule = false;
233         bool checkCell = false;
234         if (cellCount == 0) //For cases of no surrounding platform cells
235         {
236             for (int nCellX = 0; nCellX < 3; nCellX++)
237             {
238                 for (int nCellY = 0; nCellY < 3; nCellY++)
239                 {
240                     if (nCells[nCellX, nCellY] == 0 && nCells[1, 1] == 1)
241                     {
242                         checkCell = true;
243 
244                     }
245                     else
246                     {
247                         checkCell = false;
248                         return;
249                     }
250                 }
251             }
252 
253             if (checkCell == true)
254             {
255                 if(randTile == 1)
256                 {
257                     levelTilemap.SetTile(cellPosition, platform);
258                 }
259                 else
260                 {
261                     return;
262                 }
263             }
264         }
```

Figure 15: Lines 224 to 264 of LevelGenerator.cs detailing the check for rules containing no cells within the CheckEachRule method.

The code below (Figure 16) is a continuation of the method discussed in Figure 15, if the cellCount value of a rule is greater than zero this section executes, checking if the neighbourhood coordinates for each cell match platform positions. This section utilises the same checking marker technique as the previous section and again randomly assigns a target cell tile.

Difficulties were encountered particularly when finding a suitable technique for storing a ruleset with varying cell counts and ensuring that the array was initialised correctly both within LevelGenerator.cs and PlayerSetup.cs simultaneously when starting the game.



```
265         else
266         {
267             for (int cell = 0; cell < cellCount; cell++)
268             {
269                 if (nCells[ruleList[ruleCount][cell][0], ruleList[ruleCount][cell][1]] == 1)
270                 {
271                     checkRule = true;
272                 }
273                 else
274                 {
275                     checkRule = false;
276                     return;
277                 }
278             }
279 
280             if (checkRule == true)
281             {
282                 if (randTile == 1)
283                 {
284                     levelTilemap.SetTile(cellPosition, platform);
285                 }
286                 else
287                 {
288                     return;
289                 }
290             }
291         }
292     }
```

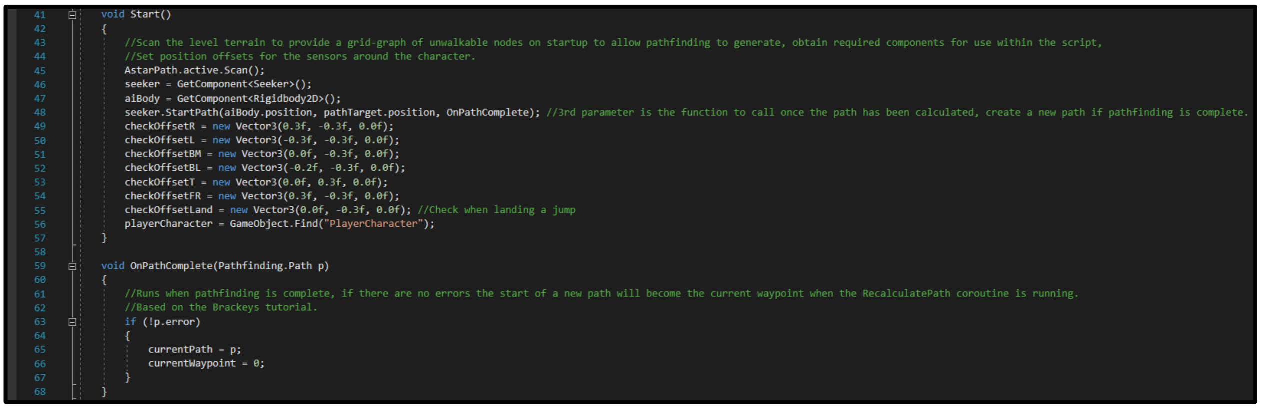
Figure 16: Line 265 to 292 of LevelGenerator.cs, a check to iterate through each rule as part of the CheckEachRule method.

7.2 AI for Testing

The AI for testing utilised the A* Pathfinding Project package (Granberg, 2009) to provide scripts required to visualise and interpret the level terrain for a pathfinding route (AstarPath.cs and Seeker.cs). In addition to this a custom script based partly on the Brackeys 2D Pathfinding YouTube tutorial (Brackeys, 2019) with large sections of independently created code provided path tracking and the terrain sensors responsible for character movement.

To successfully determine a pathfinding route, the AstarPath.cs responsible for calculating and storing paths and generating a walkable node grid-graph, checks the distance to each of the nearest accessible nodes and plots the most direct route. The AstarPath.cs script interacts with the Seeker.cs script which produces a visualisation of the generated path using gizmos that are only visible within the Unity editor scene view. All other functionality is provided by the code shown below from the custom AIPathfinding.cs script.

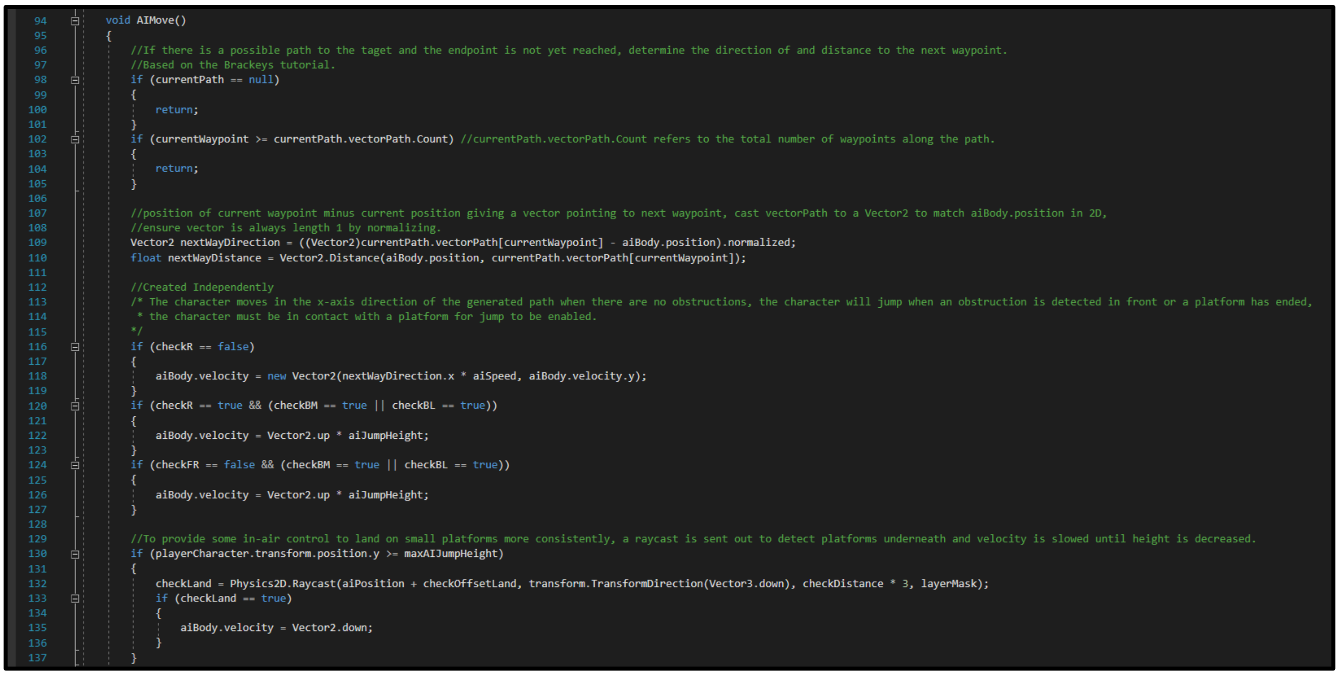
The AIPathfinding.cs script is enabled upon activation of the test mode causing the Start method (Figure 17 below) to scan the grid-graph, calculate the path and begin visualising a route from the position of the character Rigidbody (the component of the character that moves) to the target position assigned as the endpoint of the level. Once the path is calculated the current waypoint is set to 0, the beginning of the path. Other variables such as the positions of the terrain sensors around the character are also defined.



```
41    void Start()
42    {
43        //Scan the level terrain to provide a grid-graph of unwalkable nodes on startup to allow pathfinding to generate, obtain required components for use within the script,
44        //Set position offsets for the sensors around the character.
45        AstarPath.active.Scan();
46        seeker = GetComponent<Seeker>();
47        aiBody = GetComponent<Rigidbody2D>();
48        seeker.StartPath(aiBody.position, pathTarget.position, OnPathComplete); //3rd parameter is the function to call once the path has been calculated, create a new path if pathfinding is complete.
49        checkOffsetR = new Vector3(0.3f, -0.3f, 0.0f);
50        checkOffsetL = new Vector3(-0.3f, -0.3f, 0.0f);
51        checkOffsetBM = new Vector3(0.0f, -0.3f, 0.0f);
52        checkOffsetBL = new Vector3(-0.2f, -0.3f, 0.0f);
53        checkOffsetT = new Vector3(0.0f, 0.3f, 0.0f);
54        checkOffsetFR = new Vector3(0.3f, -0.3f, 0.0f);
55        checkOffsetLand = new Vector3(0.0f, -0.3f, 0.0f); //Check when landing a jump
56        playerCharacter = GameObject.Find("PlayerCharacter");
57    }
58
59    void OnPathComplete(Pathfinding.Path p)
60    {
61        //Runs when pathfinding is complete, if there are no errors the start of a new path will become the current waypoint when the RecalculatePath coroutine is running.
62        //Based on the Brackeys tutorial.
63        if (!p.error)
64        {
65            currentPath = p;
66            currentWaypoint = 0;
67        }
68    }
```

Figure 17: Lines 41 to 68 of AIPathfinding.cs, including the Start method run when the script is enabled and the OnPathComplete method.

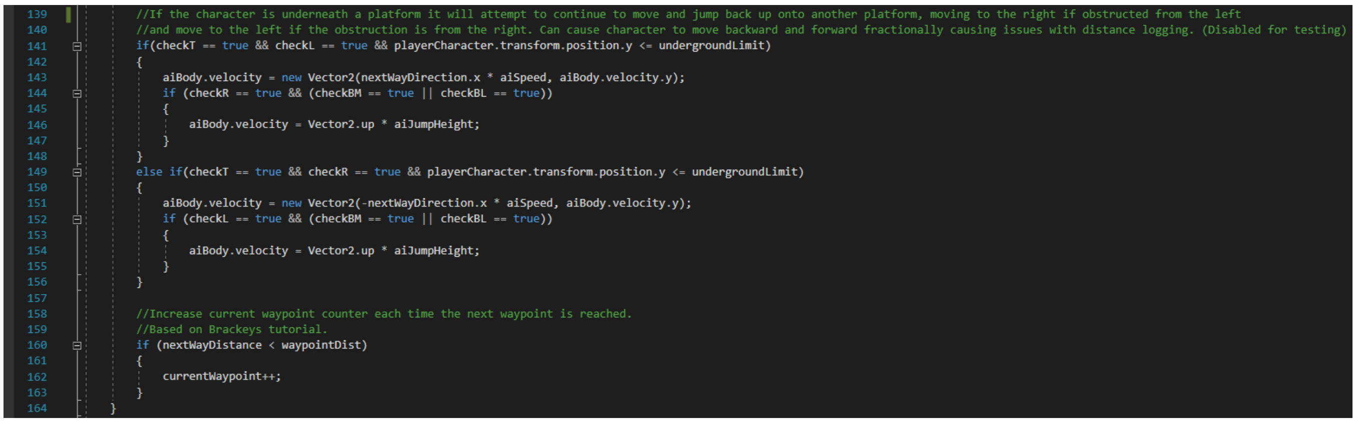
The AIMove method (Figure 18) only executes when a path exists or the end waypoint of the path is not yet reached. The direction of the next waypoint is calculated as a Vector2 in 2D space and the x-axis component of this vector is used to move the character in the direction of the pathfinding route. A series of sensors use raycasts to detect colliders on obstacles in the forward movement direction. The checkR (right) and checkFR (front-right, orientated downwards to detect incoming gaps) boolean variables allow the character to continue forward if no obstruction is detected and jump up steps in the terrain or over gaps. Forward movement will stop while checkR is true until the jump height moves the sensor above the platform, enabling forward movement onto the top of the platform. Both checkBM (bottom middle) and checkBL (bottom left) detect contact with the platform and the bottom of the character and either must be true for jump to be enabled, checkBL allows jumps to be made while only a small portion of the character is in contact with the ground. To enable more accurate landing, checkLand casts a ray when the character reaches approximately maximum jump height and sets the velocity to (0, -1) in the negative y-axis direction causing the character to fall towards the platform directly below.



```csharp
void AIMove()
{
    //If there is a possible path to the taget and the endpoint is not yet reached, determine the direction of and distance to the next waypoint.
    //Based on the Brackeys tutorial.
    if (currentPath == null)
    {
        return;
    }
    if (currentWaypoint >= currentPath.vectorPath.Count) //currentPath.vectorPath.Count refers to the total number of waypoints along the path.
    {
        return;
    }

    //position of current waypoint minus current position giving a vector pointing to next waypoint, cast vectorPath to a Vector2 to match aiBody.position in 2D,
    //ensure vector is always length 1 by normalizing.
    Vector2 nextWayDirection = ((Vector2)currentPath.vectorPath[currentWaypoint] - aiBody.position).normalized;
    float nextWayDistance = Vector2.Distance(aiBody.position, currentPath.vectorPath[currentWaypoint]);

    //Created Independently
    /* The character moves in the x-axis direction of the generated path when there are no obstructions, the character will jump when an obstruction is detected in front or a platform has ended,
     * the character must be in contact with a platform for jump to be enabled.
    */
    if (checkR == false)
    {
        aiBody.velocity = new Vector2(nextWayDirection.x * aiSpeed, aiBody.velocity.y);
    }
    if (checkR == true && (checkBM == true || checkBL == true))
    {
        aiBody.velocity = Vector2.up * aiJumpHeight;
    }
    if (checkFR == false && (checkBM == true || checkBL == true))
    {
        aiBody.velocity = Vector2.up * aiJumpHeight;
    }

    //To provide some in-air control to land on small platforms more consistently, a raycast is sent out to detect platforms underneath and velocity is slowed until height is decreased.
    if (playerCharacter.transform.position.y >= maxAIJumpHeight)
    {
        checkLand = Physics2D.Raycast(aiPosition + checkOffsetLand, transform.TransformDirection(Vector3.down), checkDistance * 3, layerMask);
        if (checkLand == true)
        {
            aiBody.velocity = Vector2.down;
        }
    }
```

Figure 18: Lines 94 to 137 of AIPathfinding.cs, containing the code for forward movement of the character when TestMode.cs is active.

The code in Figure 19 below utilises the sensors checkT (top) to detect platforms above the character and checkL (left) to detect obstructions in the negative x-axis direction when moving backwards, in addition to the same sensors used for forward movement. In the event that the AI falls down a gap and moves under an overhanging platform an attempt will be made to return to the surface when the y-axis is at a depth of at least one cell below the surface level. At this point the checkT raycast length is extended to three times its initial length (a total of 3 cells) to detect the possibility that the player is still underground but with a pocket of two blank cells above. An extension at a specified depth prevents the raycast colliding with the border along the top of the level causing the underground code to execute. For obstructions on the left or right of the character, movement will be applied in the opposite direction of the obstacle until an object is detected, initiating a jump. Once the character has reached a waypoint along the route the currentWaypoint is increased by 1 to be used when calculating the direction and distance to the next waypoint when the AIMove method runs again.



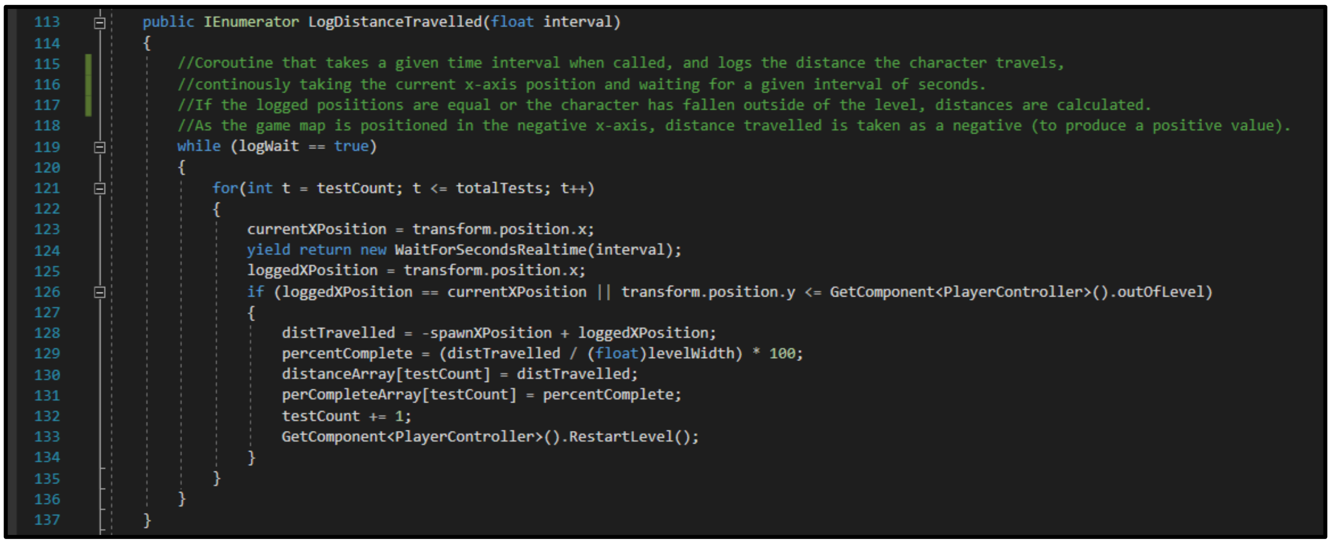
```csharp
    //If the character is underneath a platform it will attempt to continue to move and jump back up onto another platform, moving to the right if obstructed from the left
    //and move to the left if the obstruction is from the right. Can cause character to move backward and forward fractionally causing issues with distance logging. (Disabled for testing)
    if(checkT == true && checkL == true && playerCharacter.transform.position.y <= undergroundLimit)
    {
        aiBody.velocity = new Vector2(nextWayDirection.x * aiSpeed, aiBody.velocity.y);
        if (checkR == true && (checkBM == true || checkBL == true))
        {
            aiBody.velocity = Vector2.up * aiJumpHeight;
        }
    }
    else if(checkT == true && checkR == true && playerCharacter.transform.position.y <= undergroundLimit)
    {
        aiBody.velocity = new Vector2(-nextWayDirection.x * aiSpeed, aiBody.velocity.y);
        if (checkL == true && (checkBM == true || checkBL == true))
        {
            aiBody.velocity = Vector2.up * aiJumpHeight;
        }
    }

    //Increase current waypoint counter each time the next waypoint is reached.
    //Based on Brackeys tutorial.
    if (nextWayDistance < waypointDist)
    {
        currentWaypoint++;
    }
}
```

Figure 19: Lines 139 to 164 of AIPathfinding.cs, movement for attempting to navigate out of gaps and from underneath overhanging terrain (part of the AIMove method).

Due to the continued activation of the test mode throughout multiple tests the AIPathfinding.cs script remains active and therefore only runs the scan to identify the pathfinding route in the Start method once. To overcome this issue the scan method was also added to the LevelGenerator.cs script ensuring that once the testing mode was enabled, a scan would execute with each new generated level.

7.3 Testing Mode

The TestMode.cs script provides the functionality for both logging the results of the distances traversed for each level during each automated test and outputting the results to a CSV file format. The saved ruleset for each batch of tests is also output to a CSV file by iterating through each entry in the testRuleList array from the PlayerSetup.cs script. Both of these output files are provided in a more readable format within the TestResults.xls and Rulesets.xls files. The coroutine responsible for distance logging is shown below (Figure 20).



```
113    public IEnumerator LogDistanceTravelled(float interval)
114    {
115        //Coroutine that takes a given time interval when called, and logs the distance the character travels,
116        //continously taking the current x-axis position and waiting for a given interval of seconds.
117        //If the logged posiitions are equal or the character has fallen outside of the level, distances are calculated.
118        //As the game map is positioned in the negative x-axis, distance travelled is taken as a negative (to produce a positive value).
119        while (logWait == true)
120        {
121            for(int t = testCount; t <= totalTests; t++)
122            {
123                currentXPosition = transform.position.x;
124                yield return new WaitForSecondsRealtime(interval);
125                loggedXPosition = transform.position.x;
126                if (loggedXPosition == currentXPosition || transform.position.y <= GetComponent<PlayerController>().outOfLevel)
127                {
128                    distTravelled = -spawnXPosition + loggedXPosition;
129                    percentComplete = (distTravelled / (float)levelWidth) * 100;
130                    distanceArray[testCount] = distTravelled;
131                    perCompleteArray[testCount] = percentComplete;
132                    testCount += 1;
133                    GetComponent<PlayerController>().RestartLevel();
134                }
135            }
136        }
137    }
```

Figure 20: Lines 113 to 137 of TestMode.cs, showing the LogDistanceTravelled coroutine.

A coroutine runs continuously from the point it is first called, unlike traditional methods however, its execution can be delayed as in the case of LogDistanceTravelled. This delay interval specified when calling the coroutine allows the loggedXPosition variable to be obtained approximately 10 seconds after the currentXPosition variable. As a result, these two values can be compared to determine whether the character has moved in the x-axis direction. If the two values are equal or the y-axis position of the character is greater than a specified outOfLevel value, the total distance travelled through the level is calculated. Due to the position of the playable level being generated in the negative region of the x-axis, the spawnXPosition is taken as a negative to produce a positive distance value. A total percentage of the level completed is then calculated from this distance and both values are added to their respective arrays ready to be written to the output file. As falling outside of the level bounds or being stuck in a single position for 10 seconds is considered a point of failure for a test, after the logging process is complete the test counter is increased by 1 and the RestartLevel method from the PlayerController.cs script is executed to begin the next level.

The code from Figure 19 for moving back to the surface occasionally interfered with distance logging during the testing process as the AI would attempt to move out from under an overhanging platform and if no other platforms were detected above the character would move back to the right. At the edge of a platform the constant change in checkT from true to false would cause the character to move fractionally, changing the x-axis direction several times within a 10 second period. This issue was highly dependent on the complexity of the terrain but caused no distance to be logged and prevented the level from restarting. The section of code concerned was commented out for the duration of testing as navigating out of complex gaps was often beyond the scope of the AI for this project.

8.0 Testing Process

To determine the quality of generated levels, the AI character progressed through a series of 100 level iterations per ruleset for a total of 10 rulesets (1000 iterations in total with each ruleset containing 10 rules). The x-axis distance covered by the test character was recorded in each instance and the averages and ranges of these results were calculated and examined. For cases varying from the largest to shortest distances traversed, the levels were examined further, both in terms of the rules generated to create the levels and how these rules were represented visually within the levels themselves.

Specific terrain features of levels with approximately 33%, 66% and 100% completion were analysed to determine if these instances contained platforms that were unreachable, had larger gaps, more drops to lower platforms or shorter, less frequent platforms. These characteristics suggest the validity and therefore the quality of generated levels, in addition to how this translates to general difficulty for a human or AI player. The behaviour of AI when presented with certain features and its ability to complete levels provided insight into AI quality. The results of this testing process will inform future improvements within each area.

The distance travelled through a given level acted as a measurement of level quality and difficulty, in addition to the effectiveness and ability of the AI character.

8.1 Outcome and Results

The automated testing yielded positive results with 8 of the 10 rulesets producing one or more valid levels based on the number of completions and the distance data obtained. The average percentage of the levels completed, the number of full completions for each ruleset and the maximum and minimum percentage reached is shown below in Table 2.

Ruleset	Average Level Percentage Complete (%)	Number of Full Completions	Maximum Percentage Reached (%)	Minimum Percentage Reached (%)
1	39.87	9	100	0.58
2	18.26	0	82.08	0.13
3	24.66	1	100	0.30
4	44.20	20	100	0.57
5	46.96	21	100	0.07
6	55.51	32	100	0.06
7	32.18	6	100	0.58
8	26.97	2	100	0.30
9	36.09	13	100	0.30
10	17.37	0	75.60	0.30

Table 2: An overview of level completion data for each of the 10 rulesets from the TestResults.xls file produced from the automated testing process.

While the average level percentages completed do not exceed approximately 55.51% for any of the rulesets, this average is heavily dependent on the level quality and difficulty in addition to the ability of the AI guided character to move through the often-complex level terrain. These factors result in a larger number of low percentage completions in many cases reducing the average. This can be seen with ruleset 6 which achieved 32 full completions, approximately 1/3 of the 100 tests executed while only obtaining an average of 55.51%. The minimum percentages reached for each ruleset were very low in each case, suggesting that some levels were generated with starting conditions too advanced for the test AI or that potentially erroneous conditions impeded character movement.

Despite the low averages of rulesets 2 and 10 and no full completions of the generated levels, the maximum percentages reached in each case were 82.08% and 75.60% respectively. As the distance data relies on both level quality and effectiveness of the AI this indicates that a level may still provide a valid completion path for a human player. Table 3 below provides a more in-depth analysis of level completion in the ranges of 0-33%, 33-66% and 66-100%.

Ruleset	Number of Results 0-33%	Number of Results 33-66%	Number of Results 66-100%
1	51	27	22
2	86	10	4
3	73	20	7
4	50	21	29
5	44	24	32
6	39	16	45
7	69	16	15
8	71	18	11
9	60	20	20
10	86	10	4

Table 3: A breakdown of the number of results obtained within three evenly spaced completion ranges from the TestResults.xls file.

From the data provided by both Table 2 and 3, rulesets 4, 5 and 6 provided the best results across all categories with the lowest number of low percentage results and the highest rate of results in the 66-100% range as well as the most completions. Rulesets 2, 3 and 10 provided the worst results with 73 or more of the results recorded falling within the 0-33% range, with ruleset 3 successfully producing one full completion.

8.2 Analysis

The number of rules in each ruleset and the number of cells in each rule were investigated for correlations between the quantity of rules and the effectiveness of level generation seen in Table 4 below. In some cases, duplicate cells or the target centre cell are generated as part of a rule, therefore these are omitted from the cell counts resulting in smaller rules where applicable. Rulesets may also contain rules with no cells which are a valid option if all surrounding cells are blank, some rulesets contain duplicate rules which have been removed in the rule count.

Rulesets	Number of Rules	Number of Cells Per Rule	Average Cells Per Rule
1	10	4, 5, 5, 1, 6, 5, 6, 4, 3, 2	4.10
2	10	0, 7, 6, 4, 3, 4, 2, 5, 5, 4	4.00
3	9	6, 0, 4, 6, 6, 2, 2, 5, 6	4.11
4	10	2, 0, 1, 5, 6, 4, 2, 7, 2, 3	3.20
5	10	3, 5, 2, 6, 2, 5, 1, 6, 3, 0	3.30
6	10	4, 6, 6, 4, 1, 2, 0, 5, 5, 1	3.40
7	10	3, 2, 3, 2, 4, 4, 3, 6, 2, 3	3.20
8	9	6, 2, 2, 0, 4, 7, 6, 5, 3	3.89
9	10	3, 4, 4, 3, 5, 7, 6, 5, 6, 1	4.40
10	8	5, 7, 7, 0, 5, 4, 6, 5	4.88

Table 4: An overview of the number of rules contained within each ruleset and the number of cells within each rule, including the average cells per rule, taken from the Rulesets.xls file.

Rulesets 4, 5 and 6 had the lowest average cells per rule with a high number of rules (with the exception of ruleset 7), suggesting that a larger number of rules containing fewer cells may be beneficial to level generation. However, ruleset 7 contains fewer cells on average but achieved a lesser average level completion across the 100 test iterations. This exception may be a result of the randomised starting state of the cells and the number of rule matches that were applied to each level. The least effective rulesets 2, 3 and 10 had relatively high averages for cells per rule and generally smaller rulesets, particularly in the case of ruleset 10 with only 8 rules and the highest average of 4.88. This further implies that for maximum level generation quality and validity, large rulesets with fewer cells are preferable.

The rulesets for the most effective level generation (ruleset 6) and the least effective (ruleset 10) were examined further to determine how the generated cells for each rule are visually represented as terrain features within a level and how these features impact difficulty and level validity. The visual representations of each rule are shown in Figure 21 and 22 below, where 'T' denotes the target cell at the centre of a Moore neighbourhood. The target cell is randomly generated as a platform or blank cell upon matching the surrounding cells to a given rule. For rules containing no cells the target cell is changed if all surrounding cells are blank.

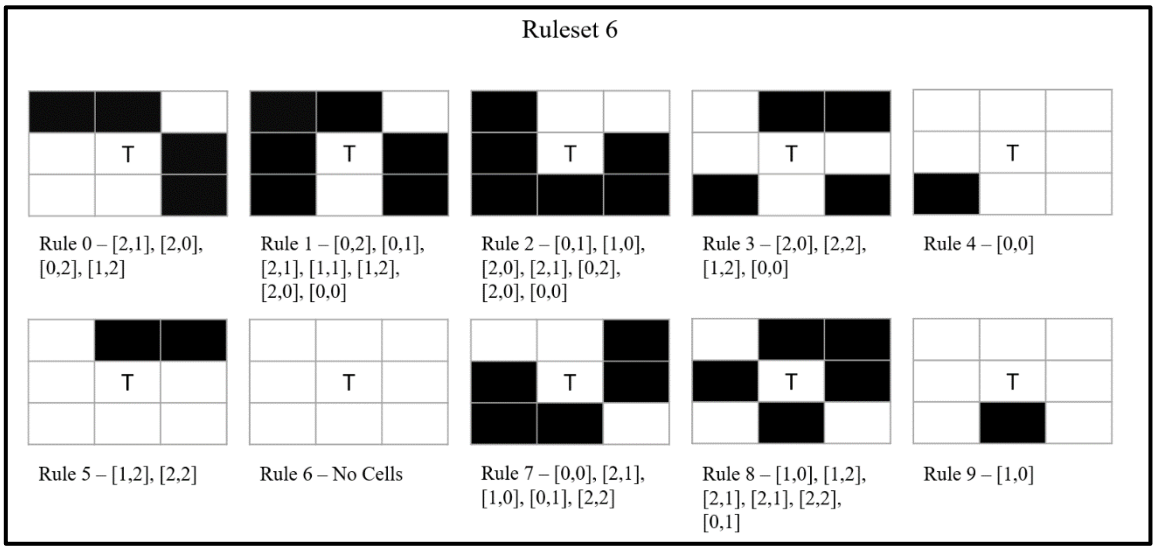

Ruleset 6
T
Rule 0 – [2,1], [2,0], [0,2], [1,2]
Rule 1 – [0,2], [0,1], [2,1], [1,1], [1,2], [2,0], [0,0]
Rule 2 – [0,1], [1,0], [2,0], [2,1], [0,2], [2,0], [0,0]
Rule 3 – [2,0], [2,2], [1,2], [0,0]
Rule 4 – [0,0]
Rule 5 – [1,2], [2,2]
Rule 6 – No Cells
Rule 7 – [0,0], [2,1], [1,0], [0,1], [2,2]
Rule 8 – [1,0], [1,2], [2,1], [2,1], [2,2], [0,1]
Rule 9 – [1,0]

Figure 21: A visual representation for ruleset 6 displaying the surrounding cell patterns that match each rule.

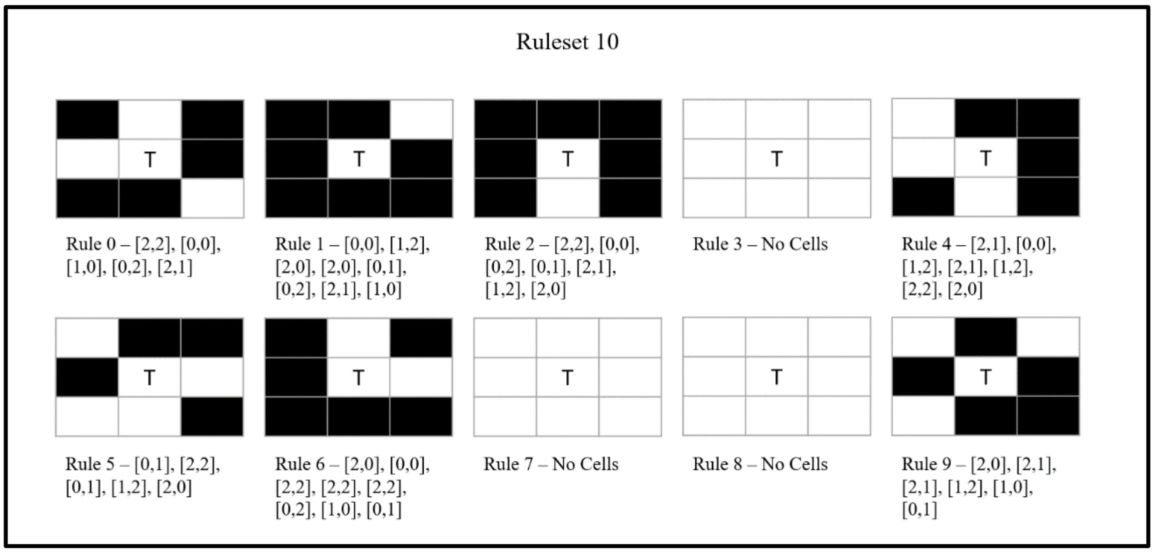

Ruleset 10
T
Rule 0 – [2,2], [0,0], [1,0], [0,2], [2,1]
Rule 1 – [0,0], [1,2], [2,0], [2,0], [0,1], [0,2], [2,1], [1,0]
Rule 2 – [2,2], [0,0], [0,2], [0,1], [2,1], [1,2], [2,0]
Rule 3 – No Cells
Rule 4 – [2,1], [0,0], [1,2], [2,1], [1,2], [2,2], [2,0]
Rule 5 – [0,1], [2,2], [0,1], [1,2], [2,0]
Rule 6 – [2,0], [0,0], [2,2], [2,2], [2,2], [0,2], [1,0], [0,1]
Rule 7 – No Cells
Rule 8 – No Cells
Rule 9 – [2,0], [2,1], [2,1], [1,2], [1,0], [0,1]

Figure 22: The representation of ruleset 10 and the cell patterns for each rule.

The lower number of cells per rule is clearly visible within ruleset 6 (Figure 21) compared to ruleset 10 (Figure 22) which contains fewer rules overall (due to the duplicate rules with no cells). A lower cell count per rule could cause more rules to be applied across the level, depending on the initial randomised starting state and would tailor the level more closely to the ruleset reducing the amount of randomly placed cells. Rules where the target cell is confined by surrounding cells and is therefore not accessible by the player (such as rules 1 and 8 in ruleset 6 or rules 1, 2 and 9 in ruleset 10) provide purely visual level characteristics such as more solid block shaped platforms that do not affect the level terrain difficulty or gameplay. The following screenshots (Figures 23, 24, 25 and 26) obtained throughout the testing process were examined to determine which rules affect level generation.

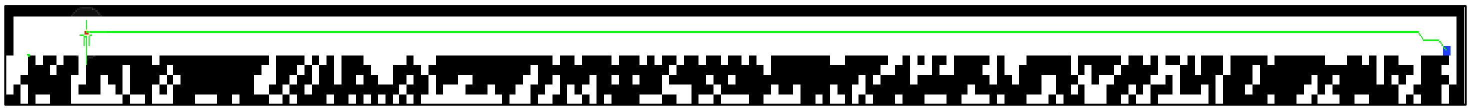


Figure 23: Ruleset 6 test 20 with 100% completion.

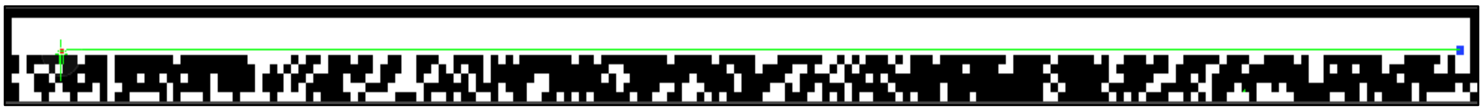


Figure 24: Ruleset 6 test 90 with 100% completion.

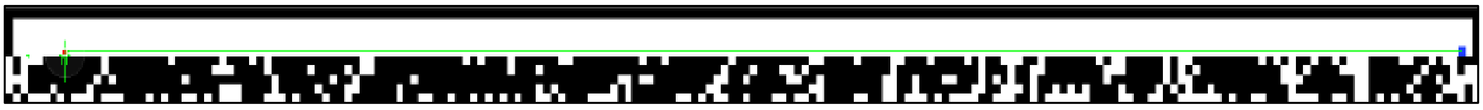


Figure 25: Ruleset 6 test 52 with 67.07% completion.

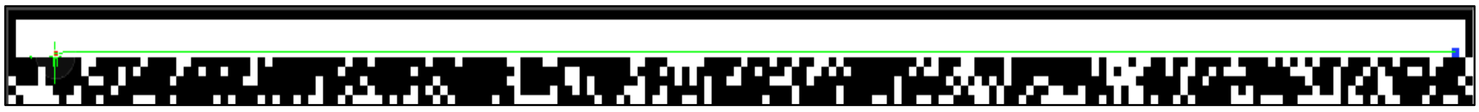


Figure 26: Ruleset 6 test 60 with 33.65% completion.

The levels generated for tests above share several common characteristics in the form of longer flat platforms both at the player spawn position and throughout the level. A low number of gaps with a depth greater than the maximum jump height of the character (two cells) and no flat-sided gaps with a width greater than two cells are present in tests 20 and 90 increasing the possibility of 100% completion. In addition to this there are no areas below the surface level with openings facing the character which may cause the AI to become stuck when entering, reducing level completion percentage.

With tests 52 and 60 there is a subtle increase in gap width, and frequency with the addition of several openings below the surface level terrain. In the case of ruleset 6, rules 0 and 5 would likely create overhanging areas the character could move under, rule 3 could create overhangs or steps, in addition to steps created by rules 2 and 7. Rules 4 and 9 could produce platform edges or single column platforms with gaps and rule 6 would add single cell floating platforms to the level. Despite the unpredictable nature of Cellular Automata and the result of applying rules being dependent on cell configurations both inside and outside of the Moore neighbourhood, ruleset characteristics are clearly observable. As ruleset 10 did not achieve any full completions, the tests from 33%, 66% and 75.60%, the maximum completion percentage reached, were analysed using Figures 27, 28 and 29 below.

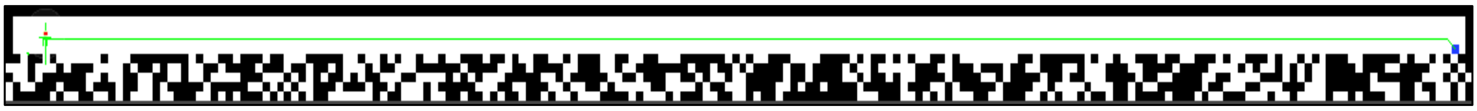


Figure 27: Ruleset 10 test 77 with 75.60% completion.

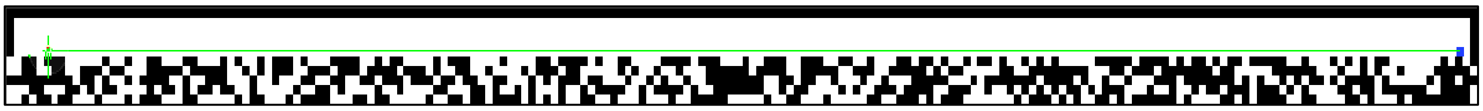


Figure 28: Ruleset 10 test 36 with 67.57% completion.

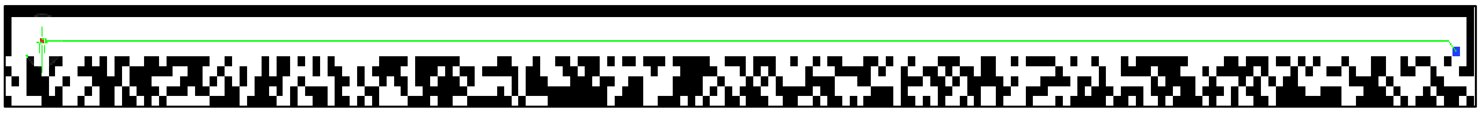


Figure 29: Ruleset 10 test 29 with 33.80% completion.

In comparison to the test screenshots from ruleset 6, the platforms generated with ruleset 10 are far more fragmented with more frequent gaps and greater terrain variation resulting in increased level difficulty for the AI. However, there appear to be no instances of invalid terrain in these cases, suggesting that similarly to the low completion tests for ruleset 6, the ability of the AI to reliably complete levels can have equal impact on the test results. A greater number of single cell platforms can be seen throughout the ruleset 10 tests potentially due to rules 0,4 and 6 with a single detached cell depending on the state of the target cell. The rules containing no cells may also produce floating single cell platforms. The rules 0, 4, 5 and 6 of ruleset 10 could all generate either steps in the terrain or overhanging areas the character could move under.

While all of these characteristics increase level difficulty for an AI, a human player can more easily navigate these obstacles unless they require a jump height which is beyond the functionality of the game. The completion rates for AI when testing are therefore a lower estimate of level completion in practice.

For the cases of minimum completion percentages (Table 2), many of these are due to poor initial AI manoeuvrability as with test 85 (Figure 31) and in some rare instances, immediate level terrain issues at the character spawn position such as small or potentially non-existent starting platforms or invalid gap sizes (Figure 30).

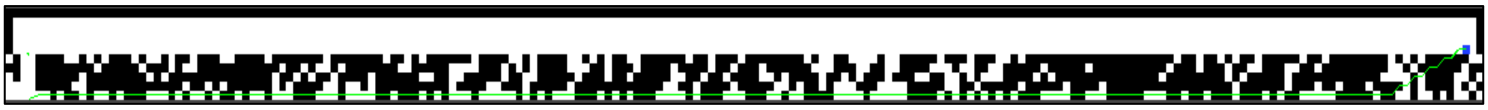


Figure 30: Ruleset 6 test 28 with 0.06% completion.

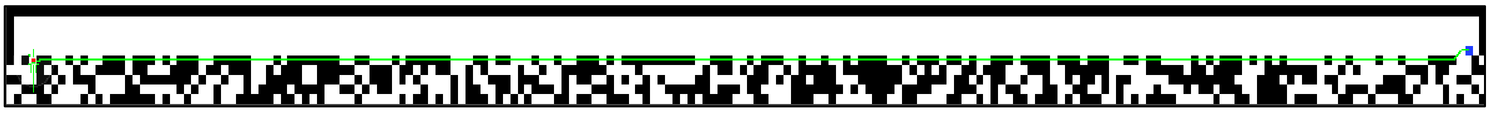


Figure 31: Ruleset 10 test 85 with 0.30% completion.

To prevent these issues a larger fixed platform could be generated at the start of each level to ensure the character has more space to move forward before jumping, likely timing the jump more accurately. This would however impose pre-authored content on a purely CA generated level, detracting from the random variation of the initial section of each level.

9.0 Evaluation

The evaluation section discusses the main conclusions drawn from the testing regarding AI and level quality and how the interpretation of level difficulty can change as a result. Each of these concepts is highly dependent on the quality of the others when generating levels.

9.1 Level Quality

As a result of the random nature of CA and the unpredictability of how each applied ruleset will connect with cells outside of the immediate Moore neighbourhood, a definitive analysis of whether each generated level supports a valid pathway to the endpoint is not possible within the scope of this project. The presence of a valid pathway is highly dependent on the complexity of the AI testing the level, more complex terrain requires an AI of greater ability for accurate testing. This can be seen when a human player attempts to complete a level, fully aware of all possible reachable platforms within the view of the in-game camera, providing better accuracy than an AI only aware of obstacles immediately approaching.

The testing process shows that all rulesets with the exception of ruleset 10 are capable of producing valid pathways even with a relatively simplistic AI and that ruleset 10 achieved 75.60% in maximum completion percentage. Close inspection of Figure 27 suggests that there were no invalid terrain areas and that the inability of the AI to finish the level was based on the terrain complexity compared to the AI complexity. To fully determine if every level generated would be considered valid, a method for automatic examination of every distance between each cell in a given level could be created or a far more complex AI system could be utilised throughout testing.

During general user testing completed while developing the project, very few instances of levels with impassable terrain characteristics were generated. The unpredictability of CA is likely both the cause and prevention of invalid characteristics as only two main terrain features are considered invalid, steps requiring a vertical jump height greater than 2 cells or horizontal jump distances across gaps of width greater than 2 cells with no lower platforms. The application of rules on each individual cell from a randomised initial starting state drastically reduces the occurrences of these features that do not have an alternate route. Therefore, level quality is generally much higher from the perspective of a human than that of an AI and is generally sufficient for the requirement of a functional and playable game.

9.2 AI Quality

The AI generally performed effectively when moving along surface level platforms and navigating any steps in terrain up to the maximum jump height. The ability of the AI was sufficient in most cases for levels with longer flat platforms resulting in higher rates of completion but struggled with more fragmented platform placement. The combination of frequent gaps between single cell platforms or movement requiring precise timing and accuracy such as diagonal jumps across a gap from a lower platform to a platform at maximum jump height proved to be too difficult for the current AI. Overhanging terrain also caused issues when attempting to return back to the surface level as seen in Figure 32 below.

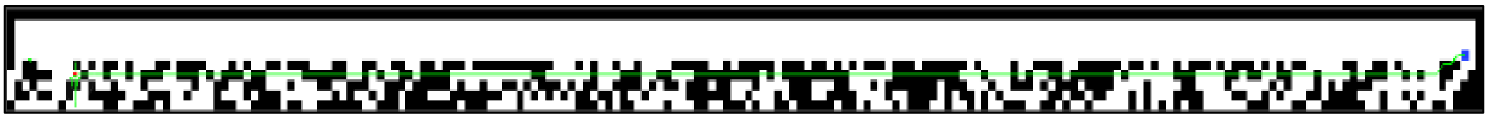


Figure 32: Ruleset 10 test 13 with 3.08% completion.

The generated level in test 13 shows many instances of difficult terrain ranging from overhanging areas (where the AI fails at the start of the level), single cell platforms and several occurrences of diagonal jumps to maximum reachable height. A particular issue with detecting overhanging areas is that the AI is not obstructed by any obstacles immediately in the direction of movement, the checkR sensor detects no colliders and the character proceeds in the current direction. The code within the AIPathfinding.cs script attempts to overcome this by checking if both checkT and checkR detect platforms and moving the character in the opposite direction if this condition is met. At this point however, the character is already underneath the overhang and will encounter the transition issue at the edge of the platform with a rapidly changing checkT value causing the character to move both left and right.

For the instance shown above (Figure 32) an additional check of checkL as an OR condition when at a specified depth below the surface could continue the movement in the negative x-direction until checkT is false. This would only be suitable for overhangs that do not lead to lower areas of terrain as this would cause obstructions within the detection area of checkL.

Further improvements to the AI in the form of manoeuvrability, landing accuracy and awareness of the surrounding terrain for jump timing would greatly improve AI quality and level quality testing as a result.

While the testing of level quality is dependent on the quality of AI, the perceived effectiveness of AI is also dependent on level quality. In the rare cases of invalid level features the effectiveness of the AI is not sufficiently tested and as a result the calculated average completion rates become less accurate.

9.3 Interpretation of Level Difficulty

The evaluation of level difficulty is subjective and therefore different for both a human and AI player. As adaptable level difficulty was not incorporated into this project due to adaptability constraining aspects such as platform frequency, gap width and other level features, generated levels may contain varying difficulty throughout, based on the ruleset used. Rulesets with a greater number of rules each with a lower cell count produce easier more simplistic levels for both humans and AI, with a ruleset of fewer rules with high cell counts drastically increasing difficulty. This increase in difficulty is considerably less challenging for a human player than an AI. Based on the AI testing results alone, this change in difficulty could be perceived as lower quality level generation although in most cases has simply increased the difficulty beyond the ability of the AI.

To more accurately examine level difficulty beyond the scope of this project, similarly to level quality, a more competent AI is required and thorough user testing of player ability levels with a range of rulesets should be completed.

10.0 Conclusion

The incorporation of Cellular Automata (CA) as a technique for procedural level generation within a platforming game proved effective for creating a near infinite number of possible terrain configurations, within the height and width constraints of the level. However, due to often unpredictable outcomes when applying CA rules, the highest average percentage of level completion achieved across all 1000 test iterations for 10 different rulesets was only 55.51% (reached within the 100 tests executed with ruleset 6 seen in Figure 21). Ruleset 6 contained no duplicate rules and therefore included the maximum of 10 rules within the ruleset, with a generally lower cell count in each, producing long flat platforms with fewer complex obstructions or variations. While the average percentage completion did not exceed 55.51%, ruleset 6 obtained the highest number of completions within the 66-100% range and the lowest for 0-33% (Table 3). In comparison to this, ruleset 10 was the least effective for producing high completion rates, obtaining the lowest average of only 17.37% with the highest number of 0-33% completions with a smaller ruleset of 8 rules due to duplication. The cell count was higher per rule and created more varied and fragmented level terrain.

This suggests ruleset 6 provides greater assurance of a valid path for completion of each level but provides less interesting gameplay possibilities and as a result lower difficulty for both AI and a human player. Ruleset 10 however, offers greater challenge and difficulty maintaining the interest of the player but with a higher potential need for restarts due to invalid terrain. With the large difference in ability of the AI used for the testing process compared to a human player it is not possible to definitively state that a level is invalid without extensive user testing. Test 29 (Figure 29) and test 85 (Figure 31) for ruleset 10 at 33.80% and 0.30% completion respectively, did not appear to contain invalid terrain on closer inspection and therefore could be completed by a human player.

While CA can provide excellent variation in some cases and seemingly only rare occurrences of invalid paths, games of a sufficient standard require reliable systems for core components such as level generation to prevent player frustration. Improving the reliability and in turn the predictability of CA would only be possible with heavy restrictions, detracting from the inherent variation of the CA process. For the purpose of interesting and engaging gameplay for a human player, ruleset 10 may provide a more suitable level of challenge but without sufficient user testing, a definitive value for the frequency of invalid levels cannot be obtained for comparison against ruleset 6.

To further explore the capabilities of CA and provide a more conclusive argument for its suitability for a human player, both AI and user testing could be carried out and compared or alternatively a far more competent AI could be used when testing to achieve results closer to that of a human. Some initial project requirements such as adaptable level difficulty and AI opponents were not incorporated into the final project. The decision to not include adaptable difficulty was based on the concept that an increase in predictability as a result of steady intervals of higher difficulty, would interfere with level variation and in some cases make the use of CA obsolete. The idea of AI opponents was not implemented in part due to time constraints and the interpretation of the role of an opponent changing several times throughout the project timeline. As a future addition a single AI controlled opponent could simply seek the player throughout each level in an attempt to reset the character on contact, rather than attempting to race the player to the endpoint of the level.

The suitability of CA rule application for level generation without the aid of heavy constraints is highly subjective due to its random nature and often unexpected outcomes but even in its most basic form interesting, playable and varied game level creation was achieved with a competent AI guided testing process.

11.0 Bibliography

Brackeys, (2019) *2D PATHFINDING – Enemy AI in Unity.* Available at: https://www.youtube.com/watch?v=jvtFUfJ6CP8 (Accessed: 16 August 2020).

Brown, A., (2010) *Mega Man 10 Review.* Available at: https://www.cheatcc.com/wii/rev/megaman10review.html (Accessed: 19 August 2020).

Chaslot, G., Sander, B., Istvan, S., Pieter, S., (2008) 'Monte-Carlo Tree Search: A New Framework for Game AI', *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, pp. 216-217.

Christopoulou, E., Xinogalos, S., (2017) 'Overview and Comparative Analysis of Game Engines for Desktop and Mobile Devices', *International Journal of Serious Games*, 4(4), pp. 30-34.

Granberg, A., (2009) *A* Pathfinding Project.* Available at: https://arongranberg.com/astar/# (Accessed: 10 August 2020).

Johnson, L., Yannakakis, G. N., Togelius, J. (2010) 'Cellular Automata for Real-Time Generation of Cave Levels', *PCGames 10: Proceedings of the 2010 Workshop on Procedural Content Generation in Games,* (10), pp. 2, 3.

Lester, P., (2003) *A* Pathfinding for Beginners.* Available at: https://www.gamedev.net/tutorials/programming/artificial-intelligence/a-pathfinding-for-beginners-r2003/ (Accessed: 20 August 2020).

Macedo, Y. P. A., Chaimowcz, L. (2017) 'Improving Procedural 2D Map Generation Based on Multi-Layered Cellular Automata and Hilbert Curves', *2017 Brazilian Symposium on Computer Games and Digital Entertainment*, pp. 118-120.

Rafiq, A., Kadir, T. A. A., Ihsan, S. N., (2020) 'Pathfinding Algorithms in Game Development', *IOP Conference Series Materials Science and Engineering*, pp. 2-3.

Rose, T. J., Bakaoukas, A. G. (2016) 'Algorithms and Approaches for Procedural Terrain Generation: A brief review of current techniques', *2016 8th International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES)*, pp. 1, 2.

Sebastian Lague, (2014) *A* Pathfinding (E01: algorithm explanation).* Available at: https://www.youtube.com/watch?v=-L-WgKMFuhE&list=PLFt_AvWsXl0cq5Umv3pMC9SPnKjfp9eGW&index=1 (Accessed: 21 August 2020)

World 1-1 (Super Mario Bros.) (2020), Available at: https://www.mariowiki.com/World_1-1_(Super_Mario_Bros.) (Accessed: 20 August 2020).