University of Derby

School of Computing and Mathematics

A project completed as part of the requirements for the BSc (Hons) Computer Games Programming

entitled

Create a set of AI Tools for Spelunky that will allow users to program their own player bots

by

Daniel Robert Scales
In the years 2013 - 2014
This project builds upon the original open source Spelunky source code for GameMaker 8. (Derek Yu & Mossmouth, 2009).

Abstract
This report examines various Artificial Intelligence (AI) Application Program Interfaces (APIs) created for competitions, such as Infinite Mario Bros API (Togelius, 2009) and Ms Pac-Man (University of Essex, 2009), and the benefits that running these competitions can provide. We then continue to explore the process of creating an AI API for Spelunky, a much more complex game, using GameMaker. In this paper we examine the extensive gameplay elements offered by Spelunky, and discuss various solutions to creating a toolset which along with providing the bot with data also enforces suitable limitations to ensure the challenge provided is as similar to that a player would face as possible. To evaluate, we analyse various implementations created by developers to date using the Spelunky API and how they tackle the challenges created by the test-beds provided. This is followed by a user guide that provides an overview of the toolkit. This user guide provides developers with details on how to query Spelunky to receive the information they need, guidelines on how they should access information and brief instructions on how to use the tools provided such a pathfinding to create a simple bot implementation.
Acknowledgements

Tommy Thompson for supervising this dissertation.

Special Thanks to Andrew Roper, Dan Pearce, James Hobson, Ryan Simpson, Matthew Smith and Tim Rukowski.
Table of contents:

1.1 Introduction........................................................................................................................................10
1.2 Project Aim.........................................................................................................................................10
1.3 Project Objectives ..........................................................................................................................11
2.0 Spelunky Overview........................................................................................................................12
2.1 Introducing Spelunky Dude ............................................................................................................13
2.2 Items and Monsters .........................................................................................................................14
2.3 Environments ..................................................................................................................................16
2.4 Traps ..................................................................................................................................................17
2.5 Items in Depth ..................................................................................................................................18
2.6 Key Locations ..................................................................................................................................19
  2.6.1 Altars ..........................................................................................................................................19
  2.6.2 Shrines ........................................................................................................................................20
  2.6.3 Secret Exits ...............................................................................................................................20
  2.6.4 Dice House ...............................................................................................................................21
2.7 Dark Levels .....................................................................................................................................21
2.8 Level Feel .........................................................................................................................................22
3.0 Literature Review ............................................................................................................................23
  3.1 Existing AI Frameworks ...............................................................................................................23
    3.1.1 Implementation .......................................................................................................................25
    3.1.2 Examples of Implementations using the APIs .......................................................................27
  3.2 Debugging and Visualisation .........................................................................................................28
    3.2.1 Path Drawing ..........................................................................................................................28
  3.3 Path Finding ..................................................................................................................................29
    3.3.1 A* Path Finding .....................................................................................................................30
    3.3.2 Adaptive A* Pathfinding ..........................................................................................................32
    3.3.3 D* & D* Lite Pathfinding .........................................................................................................33
    3.3.2 Rapid Symmetry Reduction & Jump Point Search Path Finding .........................................34
      3.3.2.1 Rectangular Symmetry Reduction ......................................................................................34
      3.3.2.2 Jump Point Search: ............................................................................................................35
  3.4 Influence Maps ..............................................................................................................................37
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5</td>
<td>Considering Moving Objects</td>
<td>38</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Destructible Terrain</td>
<td>39</td>
</tr>
<tr>
<td>3.6</td>
<td>Fog of War</td>
<td>41</td>
</tr>
<tr>
<td>4.0</td>
<td>Methodology and implementation</td>
<td>42</td>
</tr>
<tr>
<td>4.1</td>
<td>Considerations</td>
<td>42</td>
</tr>
<tr>
<td>4.1.1</td>
<td>GameMaker</td>
<td>42</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Input Considerations</td>
<td>42</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Path Finding</td>
<td>43</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Available Data when Pathfinding</td>
<td>44</td>
</tr>
<tr>
<td>4.1.5</td>
<td>Level Landmarks</td>
<td>45</td>
</tr>
<tr>
<td>3.3.6</td>
<td>Traps</td>
<td>45</td>
</tr>
<tr>
<td>4.2</td>
<td>General Threats</td>
<td>45</td>
</tr>
<tr>
<td>4.3</td>
<td>Hidden Objects</td>
<td>46</td>
</tr>
<tr>
<td>4.4</td>
<td>Debugging and Visualisation</td>
<td>47</td>
</tr>
<tr>
<td>4.5</td>
<td>Implementation of SpelunkBots API</td>
<td>47</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Creating a C++ DLL for GameMaker</td>
<td>47</td>
</tr>
<tr>
<td>4.6</td>
<td>Player Input</td>
<td>48</td>
</tr>
<tr>
<td>4.7</td>
<td>Level Data</td>
<td>49</td>
</tr>
<tr>
<td>4.7.1</td>
<td>Implementing &amp; Testing Level Data</td>
<td>49</td>
</tr>
<tr>
<td>4.7.2</td>
<td>Implementing Fog</td>
<td>52</td>
</tr>
<tr>
<td>4.7.3</td>
<td>Implementing Pathfinding</td>
<td>54</td>
</tr>
<tr>
<td>4.7.4</td>
<td>Dynamic Objects</td>
<td>55</td>
</tr>
<tr>
<td>4.7.5</td>
<td>Implementing Moveable Blocks, Enemies and Collectable Objects</td>
<td>56</td>
</tr>
<tr>
<td>4.7.6</td>
<td>Enemies &amp; Moving traps</td>
<td>57</td>
</tr>
<tr>
<td>4.7.7</td>
<td>Collectables, Items and other Dynamic Objects</td>
<td>58</td>
</tr>
<tr>
<td>4.8</td>
<td>Analysing bots</td>
<td>59</td>
</tr>
<tr>
<td>4.8.1</td>
<td>Implementing debugging visualisation</td>
<td>59</td>
</tr>
<tr>
<td>4.8.2</td>
<td>Explored Area</td>
<td>60</td>
</tr>
<tr>
<td>4.8.3</td>
<td>Game speed</td>
<td>60</td>
</tr>
<tr>
<td>4.8.4</td>
<td>Bot Statistics</td>
<td>61</td>
</tr>
<tr>
<td>4.8.5</td>
<td>Level Creation</td>
<td>61</td>
</tr>
<tr>
<td>5.0</td>
<td>Testing &amp; Analysis</td>
<td>64</td>
</tr>
<tr>
<td>5.1</td>
<td>Behaviour Implementation</td>
<td>64</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>5.2 Testing Spelunkbots with Developers</td>
<td>66</td>
<td></td>
</tr>
<tr>
<td>5.3 Summary of Results</td>
<td>67</td>
<td></td>
</tr>
<tr>
<td>5.1 Spelunkbots as a Challenging Domain</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>5.2 Recommendation for future research in Spelunky AI</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>5.3 Summary</td>
<td>74</td>
<td></td>
</tr>
<tr>
<td>References</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>Bibliography</td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>Appendix A: Software User Guide</td>
<td>79</td>
<td></td>
</tr>
<tr>
<td>A.1: Creating your bot</td>
<td>79</td>
<td></td>
</tr>
<tr>
<td>A.1.1: Character input</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>A.1.2: level information</td>
<td>81</td>
<td></td>
</tr>
<tr>
<td>A.2: Useful Calls</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td>A.2.1: Player Queries</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td>A.3.2: Environment Queries</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>A.3.3 State Queries</td>
<td>88</td>
<td></td>
</tr>
<tr>
<td>A.4: Utilising Pathfinding</td>
<td>89</td>
<td></td>
</tr>
<tr>
<td>A.5: Creating &amp; using levels</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>A.6: Using C++ in GameMaker</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>Appendix B: Spelunkbots Behaviour Video</td>
<td>92</td>
<td></td>
</tr>
<tr>
<td>Appendix C: Spelunkbots Project</td>
<td>92</td>
<td></td>
</tr>
<tr>
<td>Appendix D: Spelunkbots Bot Source</td>
<td>93</td>
<td></td>
</tr>
<tr>
<td>Appendix E: Spelunkbots Bot Source from Developers</td>
<td>103</td>
<td></td>
</tr>
</tbody>
</table>
Table of Figures:

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Spelunky Dude’s various core abilities. (Spelunky Wiki, 2014)</td>
<td>13</td>
</tr>
<tr>
<td>2.8</td>
<td>The 4 Standard Environments from Spelunky (Spelunky Wiki, 2014)</td>
<td>16</td>
</tr>
<tr>
<td>2.9</td>
<td>Spelunky Altar with Yeti placed (Spelunky Wiki, 2014)</td>
<td>19</td>
</tr>
<tr>
<td>2.10</td>
<td>Golden Idol Head (Spelunky Wiki, 2014)</td>
<td>20</td>
</tr>
<tr>
<td>2.11</td>
<td>Dice House in Spelunky (Spelunky Wiki, 2014)</td>
<td>21</td>
</tr>
<tr>
<td>2.12</td>
<td>Example of a Dar Level in Spelunky (Spelunky Wiki, 2014)</td>
<td>21</td>
</tr>
<tr>
<td>3.1</td>
<td>Mario AI displaying the debugging visualisation (Lovett, 2009)</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>Best first search and A-Star comparison (Patel, 2010)</td>
<td>30</td>
</tr>
<tr>
<td>3.3</td>
<td>Simplified map of Romania (Russel &amp; Norvig, 2010, PAGE)</td>
<td>31</td>
</tr>
<tr>
<td>3.4</td>
<td>Comparison of A*, RSR and JPS (Harabor, 2012)</td>
<td>34</td>
</tr>
<tr>
<td>3.5</td>
<td>Jump Point Search Neighbour Pruning (Harabor, 2012)</td>
<td>35</td>
</tr>
<tr>
<td>3.6</td>
<td>Jump Point Search Forced Neighbours (Harabor, 2012)</td>
<td>36</td>
</tr>
<tr>
<td>3.7</td>
<td>Jump Point Search, Jump Points (Harabor, 2012)</td>
<td>36</td>
</tr>
<tr>
<td>3.8</td>
<td>Influence Map displaying Strategic Reasoning (Champandard, 2011)</td>
<td>38</td>
</tr>
<tr>
<td>3.9</td>
<td>Super Bomberman (Hudson Soft, 1993)</td>
<td>39</td>
</tr>
<tr>
<td>4.1</td>
<td>Pathfinding Fog</td>
<td>44</td>
</tr>
<tr>
<td>4.2</td>
<td>Udjat Eye, Spelunky (Spelunky Wiki, 2014)</td>
<td>46</td>
</tr>
<tr>
<td>4.3</td>
<td>Spectacles, Spelunky (Spelunky Wiki, 2014)</td>
<td>46</td>
</tr>
<tr>
<td>4.4</td>
<td>Querying for jetpack in GameMaker Script</td>
<td>48</td>
</tr>
<tr>
<td>4.5</td>
<td>Spelunky Level Data</td>
<td>50</td>
</tr>
<tr>
<td>4.6</td>
<td>Spelunky level containing water (Spelunky Wiki, 2014)</td>
<td>51</td>
</tr>
<tr>
<td>4.7</td>
<td>C++ Fog of War Implementation</td>
<td>52</td>
</tr>
<tr>
<td>4.8</td>
<td>Spelunky level data with fog</td>
<td>53</td>
</tr>
<tr>
<td>4.9</td>
<td>Spelunky Dude tackling spider webs (Spelunky Wiki, 2014)</td>
<td>55</td>
</tr>
<tr>
<td>4.10</td>
<td>Spelunkbots Debug Display</td>
<td>59</td>
</tr>
<tr>
<td>4.11</td>
<td>Cape level</td>
<td>61</td>
</tr>
<tr>
<td>4.12</td>
<td>Level selection menu in Spelunky</td>
<td>63</td>
</tr>
<tr>
<td>5.1</td>
<td>Level with pathfinding around corner</td>
<td>65</td>
</tr>
<tr>
<td>A1</td>
<td>Getting the cost of an item</td>
<td>88</td>
</tr>
<tr>
<td>D.1</td>
<td>Gold Digger bot source</td>
<td>93</td>
</tr>
<tr>
<td>D.2</td>
<td>Sean Bean bot source</td>
<td>94</td>
</tr>
<tr>
<td>D.3</td>
<td>Jetpack Joyride bot source pt.1</td>
<td>95</td>
</tr>
</tbody>
</table>
Figure D.4: Jetpack Joyride bot source pt.2 .................................................................96
Figure D.5: Shotgun Total Warrior bot source pt.2 .......................................................97
Figure D.6: Shotgun Total Warrior bot source pt.2 .......................................................98
Figure D.7: Discovery Dan bot source ...........................................................................99
Figure D.8: Not So Solid Snake bot source ...................................................................100
Figure D.9: Indie bot source ........................................................................................101
Figure D.10: Jar Jar bot source .....................................................................................102
Figure E.1 Dan Pearce bot source ................................................................................103
Figure E.2: Andrew Roper bot source ...........................................................................104
Figure E.3: Matthew Smith bot source ..........................................................................105
Figure E.4 James Hobson bot source ...........................................................................106
Table of Tables:

Table 2.1: Trap behaviour........................................................................................................17
Table 2.2: Environment Manipulating Items in Spelunky.........................................................18
Table 3.1: AI Framework implementations, rules and restrictions ........................................24
Table 3.2: Mario AI Competition Implementations (Tolgeius, 2009).....................................27
Table 4.1: Example Enemy Types and Behaviour .................................................................57
Table 4.2: Provided Test-Beds and Descriptions ....................................................................62
Table 5.1: Spelunkbot behaviour implementations ..................................................................64
Table 4.3: Spelunky bot implementations ................................................................................66
Table A.1: Control variables.....................................................................................................80
Table A.2: Spelunkbots API useful methods ............................................................................81
Table A.4: liquid nodes..............................................................................................................82
Table A.5: Collectable nodes....................................................................................................83
Table A.6: Enemy Variables.....................................................................................................83
Table A.7: Player Resource Variables .....................................................................................84
Table A.9: Inventory Query Variables .....................................................................................85
Table A.10: Player Resource Variables ....................................................................................86
Table A.11: Spelunkbots Room Queries ...................................................................................86
Table A.12: Receiving object state ..........................................................................................88
Table A.13: Spelunkbots Pathfinding Methods .........................................................................89
Chapter 1

1.1 Introduction

AI APIs and competitions have proven quite popular in recent years, providing developers with varied sets of tools to face an array of challenges. These competitions benefit AI development in a couple of key ways. Firstly, these competitions can set fresh challenges which may not have been explored previously; this means that new solutions could be found to problems which could later be applied in practice to other problems. Along with this, these competitions can be used by newcomers to AI development as an accessible entry point; as many of them provide tools to assist with creating AI implementations, this is often a great place to begin.

This dissertation will examine existing APIs and toolsets, focusing on the challenges they face, the tools they provide to tackle them and how they limit the information that bots are able to receive from the game. The tools and techniques that are discussed will cover a lot of different aspects of AI development and implementation including pathfinding, debug data, bot control and data limitation. Each of these tools and implementations are discussed whilst taking in mind how they could benefit tackling the challenges introduced by Spelunky (Mossmouth, 2009), a game with incredibly rich and complex gameplay which is discussed in Chapter 2, Introducing Spelunky.

1.2 Project Aim

To create an artificial intelligence API which programmers can use to create bots that will attempt to play Spelunky including the ability to utilise all of the games features, and path find adapting to the destructible environment. Although there have been AI toolsets created for other games such as Ms Pac-man (Namco, 1982), Spelunky offers some new challenges in regard to destructible, randomly generated terrain. The challenges that Spelunky introduce for players and AI are rather complex as there are a lot of different elements to consider, such as dynamic environments, enemies with varying behaviour and collectables. I aim to provide a framework that can be used to effectively tackle problems that Spelunky introduces in a range of different ways.
1.3 Project Objectives

- To create an AI Framework using GameMaker 8.1 (YoYo Games, 2009), as this was the platform that Spelunky Classic was created with. This AI Framework should enable programmers to create their own Spelunky bots.
- To explore adaptive path finding, with 2D destructible terrain.
- To provide tools that can be used to measure bots performance using statistics provided with Spelunky.
Chapter 2 - Introducing Spelunky

2.0 Spelunky Overview

This chapter aims to provide an overview of Spelunky (Mossmouth, 2009) to explain all of the core elements of the game. Spelunky was originally released in 2009, developed by Derek Yu, founder of the Mossmouth studio, and offers players a unique and incredibly challenging gameplay experience.

There is a large amount of elements to Spelunky which is part of what makes it so interesting; this includes randomly generated levels, rogue-like gameplay and a huge array of tools, environment types and enemies for players to learn.

The gameplay in Spelunky combines platforming gameplay which most players are familiar with from games such as Super Mario Bros (Nintendo, 1985) with incredibly challenging ‘rogue-like’ gameplay. The implication of rouge-like gameplay on the player is that once they have died they must start from the beginning of the game again and no objects or perks are transferred from one run to another. Although there are shortcuts that allow players to start from various points in the game, the real challenge and attraction of Spelunky is being able to complete the game in a single run whilst achieving the highest score possible.
2.1 Introducing Spelunky Dude

In Spelunky, the player takes control of ‘The Spelunky Dude’. The Spelunky Dude has three core abilities: jumping, holding and attacking. Although these abilities can be manipulated with various items in the game there is no limit to how many times these actions can be performed; an example of these actions being performed can be seen in Figure 2.1.

![Figure 2.1: Image demonstrating Spelunky Dude’s various core abilities. (Spelunky Wiki, 2014)](image)

One of the most interesting tools at the player’s disposal is the ability to destroy terrain with limited capacity; this can be achieved with a range of tools, though typically bombs are used. The player starts with four bombs, and although more can be purchased and found in crates they can be quite a precious resource, so their placement must be very carefully considered; not having enough bombs could make certain areas unreachable or could even cause the player to become trapped.

![Figure 2.2: Screen-capture displaying a bomb exploding and destroying terrain. (Spelunky Wiki, 2014)](image)

The other common resource that players are able to use is ropes. Limited in the same way as bombs, the ropes can be used to access areas which were previously out of reach, as displayed in Figure 2.2. Along with being used to navigate, players can also use them to attack monsters which can prove useful despite being rather costly.
2.2 Items and Monsters

As displayed in Figure 2.3, Spelunky offers a wide array of tools and monsters, all of which alter the gameplay in some way. With almost 100 different types of monsters and tools to consider, this creates thousands of different possible combinations.

![Figure 2.3: five of the items available in Spelunky (Spelunky Wiki, 2014)](image)

There is a range of ways to acquire items in Spelunky, they can often be found in crates around levels, though they are also available by completing certain tasks, sacrificing items, but they are most commonly available through shops. These items can modify the behaviour of almost all aspects of the player, the items the player carries and grant access to new areas.

Shops can hold a range of different items from bombs and ropes to jetpacks and teleportation devices. To acquire these items from a shop the player has two choices; pay for the items or steal them. The price for items increases with the number of levels passed, so paying for items becomes increasingly costly; but stealing items from the shop has large implications on the game. A typical shop layout can be seen in Figure 2.4.

![Figure 2.4: Shop and shopkeeper within the mines in Spelunky (Spelunky Wiki, 2014)](image)

Once the player has become a thief by stealing an item from a shop all shopkeepers become enemies for the duration of the round. In addition to this, extra enemies are spawned at the exit of each level; this makes stealing from shops a huge risk, but this is often balanced out by the items collected and the additional score achieved by not paying for items.
Each level of Spelunky also contains Damsels, shown in Figure 2.5, which provide the player with extra health when escorted to the levels exit. Damsels can also be used tactically, for example, if the player is in a tough spot they can be used to trigger traps, to be thrown as a weapon or even be made as a sacrifice to Kali on a sacrificial altar. These decisions can drastically influence the player’s tactics and play-style.

![Damsels in three states – standing, running and dazed.](Spelunky Wiki, 2014)

Finally, the range of monsters poses a huge threat to players of Spelunky, some of which can be seen in Figure 2.6. Each of the monsters has a unique behaviour which the player must learn and adapt to if they hope to be successful in reaching the end goal.

![Four of the monsters in Spelunky](Spelunky Wiki, 2014)

The combination of these complex elements makes Spelunky an incredibly interesting game; it becomes more of a learning experience for the player discovering new types of enemies, environments and tools as they progress through the game. This makes it a great challenge for AI development as designing the implementation would be a largely iterative experience, tweaking and adapting the bots behaviour for different situations and play styles as new traps, tools and enemies are discovered by the developer.
2.3 Environments

Each of the environments in Spelunky is randomly generated which ensures a unique experience every time the game is played. The level is split into 9 random, pre-built segments which ensures that every level generated is enjoyable to play by offering layouts to compliment the games design.

One of the most interesting problems that Spelunky’s environments introduce, especially when implementing AI, is the ability to destroy terrain with resource. This is achieved with bombs, as explained in section 2.1 Introducing Spelunky Dude.

The environments in Spelunky are also littered with a number of traps, each having a different trigger and behaviour; one of which can be seen in Figure 2.7. More traps are introduced the further the player progresses and it is important that they learn the behaviour of these traps to ensure they can traverse the levels safely.

![Image displaying the ‘Arrow Trap’ being triggered by Spelunky Dude.](Spelunky Wiki, 2014)

From start to finish, Spelunky offers six unique destinations; though some of these are secret locations which can only be reached by achieving certain criteria. The four key locations can be seen in Figure 2.8. Each of these new settings introduces new challenges in the form of traps, items and monsters for the player to consider.

![Image displaying the 4 Standard Environments from Spelunky](Spelunky Wiki, 2014)
2.4 Traps
There are a range of traps with differing behaviour throughout Spelunky, as discussed previously in section 2.3. Table 2.1 explains the different behaviours of the traps available in Spelunky.

Table 2.1: Trap behaviour

<table>
<thead>
<tr>
<th>Trap</th>
<th>Behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Box</td>
<td>• Static</td>
</tr>
<tr>
<td></td>
<td>• Explodes on impact from thrown object or bomb</td>
</tr>
<tr>
<td>Spikes</td>
<td>• Static</td>
</tr>
<tr>
<td></td>
<td>• Player is killed if jumped on, but can walk through unharmed</td>
</tr>
<tr>
<td>Tiki Traps</td>
<td>• Static</td>
</tr>
<tr>
<td></td>
<td>• Spikes harm anything directly to the left or right of the trap which is two blocks tall – spikes can only be triggered in intervals</td>
</tr>
<tr>
<td>Arrow Traps</td>
<td>• Static</td>
</tr>
<tr>
<td></td>
<td>• Emits a single projectile when an object passes within a certain distance in-front of it</td>
</tr>
<tr>
<td>Boulder</td>
<td>• Rolls out of the idle and destroys terrain in its path slowly losing velocity</td>
</tr>
<tr>
<td></td>
<td>• Can be moved with an explosion</td>
</tr>
<tr>
<td>Lava</td>
<td>• Fills a space in the terrain, instant death</td>
</tr>
<tr>
<td>Acid</td>
<td>• Fills a space in the terrain</td>
</tr>
<tr>
<td></td>
<td>• Can be reduced or destroyed by removing terrain directly below or to the side</td>
</tr>
<tr>
<td>Spring</td>
<td>• Static</td>
</tr>
<tr>
<td></td>
<td>• Propels bots and players in to the air when stood upon</td>
</tr>
<tr>
<td>Turret</td>
<td>• Moveable</td>
</tr>
<tr>
<td>Forcefield</td>
<td>• Prevents any objects attempting to pass between two points</td>
</tr>
<tr>
<td>Crush Trap</td>
<td>• Dynamic</td>
</tr>
<tr>
<td></td>
<td>• Moves the direction of the player if the player is reachable within a straight line and a certain distance</td>
</tr>
</tbody>
</table>
2.5 Items in Depth

Within Spelunky there are a range of items available for the player to use that manipulate the world, and sometimes the characteristics of the player. A list of some of the most important and game changing items can be found in Table 2.2.

Table 2.2: Environment Manipulating Items in Spelunky

<table>
<thead>
<tr>
<th>Object</th>
<th>Impact on environment</th>
<th>Player can move when placed</th>
<th>Placed by player</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bomb</td>
<td>Destroys two blocks in each direction</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Ropes</td>
<td>Gives the player the ability to climb higher than they are able to jump – up to the max of approx. eight blocks.</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>Gloves</td>
<td>Gives the player the ability to hold on to walls in the environment</td>
<td>N/A</td>
<td>False</td>
</tr>
<tr>
<td>Exploding Frogs</td>
<td>Once the player has jumped on the exploding frogs, they share the same properties as the bombs. The frog loses this ability forever when entering water.</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>Mattock</td>
<td>Allows the player to remove blocks one at a time – that they are directly above or beside</td>
<td>N/A</td>
<td>True</td>
</tr>
<tr>
<td>Ladders</td>
<td>Allows the player to climb places which they would have previously not been able to reach</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Jetpack</td>
<td>Allows the bot to fly for a variable distance</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Jump Shoes</td>
<td>Doubles the jump height of player</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

All images of objects taken from Spelunky Wiki (Spelunky Wiki, 2014)
2.6 Key Locations

There are a few key locations within the environments in Spelunky with unique behaviours, this includes: altars, shrines, hidden doors, casinos and, as already discussed in 2.2, shops.

2.6.1 Altars

Within Spelunky, there is also a reward & punishment system for players to consider with altars, displayed in Figure 2.9. At altars the player can choose to sacrifice items for rewards, or to destroy the altar, though there is no real benefit in doing so. Players can sacrifice enemies, shopkeepers, damsels and other items on these altars which can provide the player with random objects and eventually ‘Kapala’, allowing players to gather health from killing enemies.

Figure 2.9: Spelunky Altar with Yeti placed (Spelunky Wiki, 2014)
2.6.2 Shrines

Shrines are another recurring location that can be randomly spawned within levels in Spelunky. These shrines are always built around a Golden Idol as seen in Figure 2.10, which is an incredibly valuable treasure which must be carried to the exit of the level; however, when the idol is taken this triggers elaborate traps such as the boulder in the mines.

Figure 2.10: Shrine within the Mines with Golden Idol Head placed above. (Spelunky Wiki, 2014)

2.6.3 Secret Exits

Throughout Spelunky there are some secret additional levels which can be reached by destroying terrain in certain locations or fulfilling certain criteria. One example of this is the ‘Black Market’, which can be found in any one of the four forest levels with the ‘Udjat Eye’ in the players’ inventory. Once the player has this special item equipped, it begins to blink once the market is nearby; with the speed of the blinking increasing the closer the player becomes. These secret levels offer both additional challenges to players, but also great rewards such as special items and large numbers of gems.
2.6.4 Dice House

Similar to the shops discussed in Section 2.2, Dice Houses contain items and a shopkeeper, but the way they are obtained differs to the traditional shop, providing the player doesn’t choose to steal the items. The player is able to bet on the dice roll for a price, and if they are successful the wall between them and the item, displayed in Figure 2.11 will be removed.

Figure 2.11: Dice House in Spelunky (Spelunky Wiki, 2014)

2.7 Dark Levels

‘Dark Levels’ appear randomly throughout Spelunky and can happen on any standard level, but not on special levels such as the final boss. Dark levels in Spelunky restrict the players view to a small radius around the player, which can be expanded by lit objects such as traps and crates. This makes the game a lot more difficult to play as the amount of information that the player has about the environment and enemies is greatly limited, this is demonstrated by Figure 2.12.

Figure 2.12: Example of a Dar Level in Spelunky (Spelunky Wiki, 2014)
2.8 Level Feel

Levels in Spelunky are often accompanied by a Level Feel; this is a message that is displayed at the beginning of a level that provides information about the characteristics of that specific level. Some examples of Level Feels include:

- “I hear snakes… I hate Snakes”: This informs the player that a snake pit has been created, which always contains a Mattock buried underground at the bottom.
- “It smells like wet fur in here”: This informs the player that they have just entered a ‘Yeti Kingdom, where the Yeti King enemy can be found.
- “You hear rushing water”: This informs the player that there is a large expanse of water at the bottom of the level full of Piranha enemies, and one giant Piranha enemy.
Chapter 3

3.0 Literature Review

This chapter is going to investigate various AI API implementations in different games, and AI techniques that can be used for developing AI tools, and eventually be used in a toolset designed especially for Spelunky (Mossmouth, 2009).

Looking at existing APIs and their implementations, tools and techniques used in AI development, this chapter will investigate the design considerations and technical implementations of the components of an AI API and how a similar system could then be constructed for Spelunky. The challenges that Spelunky introduces are quite vast, down to the number of tools at the players disposal, but one of the biggest challenges is adapting to a fully destructible environment whilst considering the players constraints as discussed in Chapter 2, *Introducing Spelunky*.

3.1 Existing AI Frameworks

There are a few examples of existing AI Frameworks that are often used to host competitions where programmers can submit their own bots to compete against others. The competitions that are held are important for a number of reasons; firstly they can provide an accessible learning environment for new developers to learn AI techniques of which the skills and knowledge gained from this can later be transferred to other projects and problems. Secondly, AI competitions can provide new interesting challenges to experienced developers, such as the case of Spelunky with the destructible terrain and resource management, or the research and knowledge gained from this could again be applied to other situations.
Two key examples of AI APIs with such competitions are the Ms Pac-Man (University of Essex, 2009) and Infinite Mario APIs (Togelius, 2009). Each of these APIs have held many competitions in which developers submit their own AI created using the tools provided to see who can create the bot that achieves the highest score. These competitions aim to deliver different problems for developers to solve with their bots, providing different situations, environments and rules to consider – some examples of which can be seen in Table 3.1.

Table 3.1: AI Framework implementations, rules and restrictions

<table>
<thead>
<tr>
<th>Framework</th>
<th>Type</th>
<th>Rules &amp; Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ms Pac-Man AI</td>
<td>Pac-Man</td>
<td>• Create a Ms Pac-Man that aims to avoid ghosts, survive for the highest time and achieve the highest score</td>
</tr>
<tr>
<td></td>
<td>Ghost</td>
<td>• Create a team of ghosts that will attempt to catch an AI controlled Ms Pac-Man</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Ghosts are not allowed to reverse – can only choose a direction at corners</td>
</tr>
<tr>
<td></td>
<td>Combined</td>
<td>• You can utilise a maximum of 512MB of Random Access Memory (RAM)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Your bot may not take more than 5 seconds to initialise</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Multithreading is not allowed</td>
</tr>
<tr>
<td>Mario AI (Togelius, 2009)</td>
<td>Infinite Mario Bros</td>
<td>• Controllers need to run in real time – cannot cause the competition computer to slow down</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• All information about the environment is provided by Environment interface</td>
</tr>
</tbody>
</table>
3.1.1 Implementation

In the case of the Ms Pac-Man API (University of Essex, 2012), the tools that you are provided with are quite small, but that is largely down to the fact that it is a much simpler game. All of the data provided would be available to a human player, simply by looking at the screen. The Ms Pac-Man API provides a ‘getMove()’ method where the player must calculate their move, which returns a direction. This method also adds an extra limitation of the amount of time the calculations can take, everything must be calculated within 40ms; this restricts the amount of calculations that have to be done and ensures that the bot does not slow down the game (University of Essex, 2012).

The information that is available through the API that can help in making an informed decision about the move includes the locations of the dots, whether there is a special object, and ghosts’ positions and previous moves taken.

The Mario API however is radically different to the Ms Pac-man toolset, but is much more relevant to Spelunky. The gameplay shares a lot of functionality seeing as they are both 2D platforming games; this means that the two titles share a large amount of mechanics, especially in regards to the way the player navigates the environment. Each game has the player control a side-scrolling character which is affected by gravity that must navigate whilst avoiding hazards by landing on platforms.

The Mario AI API is a lot more complex than the Ms Pac-Man API as there are lots more options to consider. There are a total of $2^5 \ (32)$ button combinations in Mario (Togelius, 2009) as opposed to the five possible button combinations in Ms Pac-Man.

Along with having a lot more possible button combinations than Pac-Man, the Mario API is also built for use with an infinite, procedurally generated level. The information available from the Mario API is still essentially equivalent to what is being displayed on the screen at the time; it does not provide information about upcoming segments.
The Mario API is based on using “Agents” for each individual bot. Most AI Systems use agents, even if only at a very basic level, as Russel & Norvig explained “An agent is anything that can be viewed as perceiving its environment through sensors and acting upon the environment through actuators” (Russel & Norvig, 2010, p34).

All of the decisions made for the bot in the Mario API would be made by the agent. In an effective agent, these decisions would be relative to the data that the agent has received through it’s sensors about the environment, and the tools that the agent has at its disposal.
3.1.2 Examples of Implementations using the APIs

Over the duration of the competitions, there has been a range of different challenges set and each time there have been hundreds of different implementations of bots all utilising the tools provided for each competition. Looking at just a few submissions from a single year of the Infinite Mario AI Competition from 2009 it is clear that the techniques used by developers in their bots are quite varied: (Tolgeius, 2009).

Table 3.2: Mario AI Competition Implementations (Tolgeius, 2009)

<table>
<thead>
<tr>
<th>Developer Name</th>
<th>Technique Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Robin Baumgarten</td>
<td>• Analyse Mario’s physics engine to obtain movement equations for all objects</td>
</tr>
<tr>
<td></td>
<td>• Create our own physics engine that can predict next world state</td>
</tr>
<tr>
<td></td>
<td>• Plug engine into an A* algorithm to evaluate fitness of each node</td>
</tr>
<tr>
<td></td>
<td>• Heuristic: How long before Mario reaches goal?</td>
</tr>
<tr>
<td></td>
<td>• Penalty for falling into gaps or being hurt</td>
</tr>
<tr>
<td></td>
<td>• Ignore coins, enemies, power-ups</td>
</tr>
<tr>
<td>Glenn Hartmann</td>
<td>• Modified version of one of the heuristic agents that came with the software</td>
</tr>
<tr>
<td></td>
<td>• Move forward</td>
</tr>
<tr>
<td></td>
<td>• Jump if in danger of falling</td>
</tr>
<tr>
<td></td>
<td>• Jump over enemies if safe</td>
</tr>
<tr>
<td></td>
<td>• Shoot continuously</td>
</tr>
<tr>
<td>Peter Lawford</td>
<td>• An A* search to increase the position of X as much as possible</td>
</tr>
<tr>
<td></td>
<td>• Partial simulation to anticipate future positions</td>
</tr>
<tr>
<td></td>
<td>• Some pruning of the search tree</td>
</tr>
<tr>
<td>Sergio Lopez</td>
<td>• Rule based system to answer 2 questions “Should I jump” and “Which kind of jump”</td>
</tr>
<tr>
<td></td>
<td>• Evaluates possible landing points</td>
</tr>
<tr>
<td></td>
<td>• Calculates a danger value for each action</td>
</tr>
<tr>
<td>Andy Sloane, Caleb Anderson and Peter Burns</td>
<td>• Based on A-Star Path Finding</td>
</tr>
<tr>
<td></td>
<td>• Separate physics simulation (not using the game engine)</td>
</tr>
<tr>
<td></td>
<td>• Prediction of enemies movements</td>
</tr>
</tbody>
</table>
The amount of different implementations that are possible using the toolsets is clearly demonstrated by just a handful of implementations of the 2009 competition displayed in Table 3.2, each using different techniques to create an AI for the same game.

### 3.2 Debugging and Visualisation

During the development of an AI with the use of an API, debugging information is very useful to developers to provide them with the ability to visually see the bots thought process. The additional information displayed on the screen usually displays a wide array of information relative to the game and the bots planning process as displayed in both Ms Pac-Man (University of Essex, 2009) and the Infinite Mario Bros API (Togelius, 2009).

#### 3.2.1 Path Drawing

One of the most common visual debugging tools provided with APIs is to display on the screen the possible moves that the bot could make. This is very clearly demonstrated in the Infinite Mario Bros API by drawing red lines that demonstrate where all of the possible moves would send the bot.

![Screenshot of Mario AI displaying the debugging visualisation](image)

**Figure 3.1:Screenshot of Mario AI displaying the debugging visualisation (Lovett, 2009)**

All of the lines on Figure 3.1 above show the possible moves towards the desired location considered by the agent through simulating the possible physics outcomes. As Mario only has three speeds - running, standing and walking - this is feasible to produce and display on the screen. The Ms Pac-Man Vs Ghosts API offers a lot less information, simply showing the amount of time the bot has been running, and the amount of points that have been collected; this however isn’t a major problem for Ms Pac-Man as there is not as much information being displayed on the screen.
Being able to visualise the decision making process and what options were considered by a bot is very important when developing a more complex implementation as it provides a faster, more accessible way of analysing the bots behaviour.

This information is displayed in a way that enables the developer to get an understanding of the choices being made at a glance. Although the option still remains for the developer to debug the process step by step, this can be useful for quickly finding areas that the agent has difficulty handling.

3.3 Path Finding

Path finding solutions are often provided as part of an AI API, providing users with paths to their destinations, allowing them to set their desired location. The most common technique implemented for 2D APIs and games in general is A* path finding, although there a lot of other possible path finding implementations which each provide different benefits and drawbacks. Path finding algorithms are very important for AI with a moveable character and an environment to ensure that they can navigate levels effectively, without becoming trapped, lost, or navigating to areas unnecessarily. The aim of this section is to evaluate the different pathfinding methods available, how they are implemented and the pros and cons of using each system.
3.3.1 A* Path Finding

Like most of the different options that will be explored in this chapter, the A* pathfinding algorithm is a 2D grid based algorithm. A* is likely the most commonly implemented path finding algorithm as it is very flexible, can be adapted to work in a range of different situations, and also quite simple to implement. As with most pathfinding algorithms, the aim of the A* algorithms is to return the shortest path for a given goal.

![Best-First Search vs A* Algorithm](image)

Figure 3.2: Best first search and A-Star comparison (Patel, 2010)

The example in figure 3.2 above (Patel, 2010) demonstrates the difference between an AI using the A* algorithm, to a best first search algorithm. This shows that the A* algorithm takes much more of a direct route to the destination when compared to the more greedy Best-First-Search algorithm. This demonstrates that when using pathfinding with forward planning that the AI is likely to take a much more sensible, shorter route.

A* path finding is built upon using nodes with neighbours. How this is implemented would be down to the user, but neighbouring nodes would typically have a direct path between themselves, with no obstructions. This is best explained with the equation $f(x) = g(x) + h(x)$, where $g$ is the cost of movement from point A to point B on a square grid, and $h$ is the estimated cost to move from the current point to the final destination (Lester, 2005).

There are two key things that must be considered with A* pathfinding, the path cost and the step cost.
3.3.1.1 Path Cost

A path cost is a function that assigns a cost to each path. This is different depending on the problem, but when using Figure 3.3 as an example a sensible measure would be kilometres (Russel & Norvig, 2010). In a simpler example, such as pathfinding in a 2D grid, the path cost could simply be the amount of moves taken.

![Figure 3.3: Simplified map of Romania (Russel & Norvig, 2010, PAGE)](image)

3.3.1.2 Step Cost

The step cost is the cost of each individual move in an environment – moving from state A to state B (Russel & Norvig, 2010). Using Figure 3.3 as an example, moving from Neamt to Lasi would always have a step cost of 87 kilometres.

A* works by predicting the cost of the cheapest solution to the goal, and continues to search until it reaches the closest possible value to this number. This is achieved by attempting to reach the destination by travelling to the node with the lowest value of the estimated distance from the goal and distance from the current node.

To demonstrate, looking at Figure 3.3 with Bucharest as the goal and Sibiu being the current node, there would be four neighbouring nodes to consider travelling to. The first node to be travelled to would be Rimnicu Vilcea, as the combined total of distance from Sibiu and itself, and itself and Bucharest would be 178kms - compared to the 310kms of Fagaras.
A* pathfinding can often cause performance issues as it is quite intensive. Despite this, it would only be relevant to software with a combination of large areas to navigate, and a vast number of paths to calculate. There are also versions of the A* algorithm that do not require re-evaluating the entire path, which could greatly improve performance.

### 3.3.2 Adaptive A* Pathfinding

A* path finding algorithms are often manipulated to consider more than simply a static environment as game environments are normally dynamic with threats, moveable objects and destructible terrain that should be considered. Each implementation must take into account the bot's possible movements and physics restrictions.

There is a range of adapted A* algorithms that take note of moving objects and manipulating the environment, some examples of these include:

- D*
- D* Lite
- Rapid Symmetry Reduction (RSR)
- Jump Point Search (JPS)
3.3.3 D* & D* Lite Pathfinding

The D* algorithm is an adaptation of the grid based A* path finding algorithm which was found by A. Stentz in 1994 during research for The Robotics Institute. The main objective of the D* algorithm was to provide a path finding solution which would not have to be entirely re-calculated to adapt to changes. The initial objective was that D* would prove to be more efficient than A*, but as Antony Stentz explains, “D*, however, has a reputation for being extremely complex, and has been completely obsoleted by the much simpler D*-Lite.” (Stentz, 1994).

D*-Lite achieves pathfinding without recalculating the entire path each time using ‘Lifelong Planning’ (Koenig, 2002). This maintains an estimate step cost of each node, so rather than recalculating these each time the values are stored in memory which results in a lot fewer checks each time pathfinding is required. This does have other implications however, as the data having to remain in memory the whole time could be unnecessary if pathfinding isn’t calculated a large amount of times. This method may also become much more complex in environments where the terrain is being manipulated and changed frequently, as it would mean that a lot of the values would have to be recalculated anyway.
3.3.2 • Rapid Symmetry Reduction (RSR) & Jump Point Search (JPS) Path Finding

Two other alternatives to A* pathfinding are Rectangular Symmetry Reduction (RSR) & Jump Point Search (JPS) pathfinding. Each of these algorithms are adaptations of the A* algorithm, and are implemented on top of the existing system.

As Figure 3.4 demonstrates, the A* algorithm covers a lot more of the area in comparison to both RSR and JPS, meaning that it is a lot more expensive to execute. Each of these has different implications as to what information will be returned and how they could be used as a method of pathfinding.

3.3.2.1 Rectangular Symmetry Reduction

RSR search is a lot more efficient than using A*, as shown by Figure 3.4, and requires a lot less computation. RSR has a few stages, some of which could become quite inefficient when working with a large map with many dynamic objects. This works differently in comparison to JPS as there is pre-processing required for this system to work (Harabor, 2012).

The first step for RSR is to divide the game map into sets of empty rectangles that adapt to the location of objects within the game map (Harabor, 2012). Once this process has finished, the nodes from the interior of the empty triangle are removed; at this stage it has already been established that there will be no obstacles within the bounds of the rectangle.

Once this step is complete, the second stage of the process is to add macro edges to the set of empty rectangles that have been created. Macro edges are simply the areas that connect each rectangle to adjacent areas, showing the possible entrance and exit points for each space.
The last stage of the process of RSR is Online Insertion (*Harabor, 2012*). This stage is only performed if the starting position of the pathfinder is within one of the empty rectangles that were created previously, where deleted nodes would have been placed prior to their deletion. If true, a temporary node is created and connected to the four closest perimeter nodes around the outside of the empty rectangle that the start of the path is within. Once all of the nodes have been created, traditional A* is run using the new set of nodes, which results in a much quicker search.

### 3.3.2.2 Jump Point Search:

Jump point search, as shown in Figure 3.4 has the lowest performance overhead of all 3 methods, and is another example of how A* can be manipulated to become more efficient. Like RSR Search, JPS is implemented by making collections of nodes, reducing the amount of redundant checks that need to be made. Once this has been done, applying the A* algorithm to the remaining nodes requires a lot fewer queries and no pre-processing, and can be up to 10 times faster than just using A* (*Harabor, 2012*).

There are again a few stages to the Jump Point Search, starting with ‘Neighbour Pruning’.

Neighbour pruning is the first step of reducing the amount of nodes that has to be searched through once it is time to use pathfinding; this is demonstrated by Figure 3.5. In this figure, the node labelled X is the node which is being expanded and the arrow displays the direction of travel from the parent. In both of these images, the greyed out nodes can be removed as it is possible to reach them in an optimal fashion without ever passing through Node X (*Harabor, 2012*).

![Figure 3.5: Jump Point Search Neighbour Pruning (*Harabor, 2012*)](image-url)
This method is then applied recursively to prune each and every set of neighbours in the map, where the aim is to remove symmetries by jumping over all nodes which can be reached optimally by a path that does not visit the current node (Harabor, 2012).

The next step of JPS is forced neighbours. Forced neighbours is a method of preventing key nodes from being removed as displayed by Figure 3.6 below. These two images display node X expanding in different directions but the two circled nodes become ‘Forced Neighbours’ as a result of being adjacent to X, as well as an obstacle.

The last step of JPS is to establish jump points, both in straight lines and diagonally as displayed in Figure 3.7. The lines are travelled until a node with an obstacle is reached, such as Y in Figure 3.7. The nodes between the jump points are then no longer evaluated, or even created, as it has already been established that it is possible to travel from X to Y optimally; this reduces the search space by a great amount.

Despite Jump Point Search being up to 10 times faster than A* in environments with a lot of symmetrical elements, the benefits that you could expect from maps with less symmetry would be minimal (Harabor, 2012).
3.4 Influence Maps

As Spelunky contains a lot of obstacles, collectibles and key areas there is a lot of information to consider both in constructing a path and navigating it. One technique that could help with this is ‘Influence Maps’.

Influence maps are a tool which allows the AI to analyse the entire game environment, considering different elements such as enemies, the layout of the terrain and items and adapt to this information (Champandard, 2011).

As explained by A. Chapmandard, influence maps have been implemented in AI systems since the very early stages, and the first implementations can be traced back to strategy games over 10 years ago. Influence maps are starting to become more commonly used in all genres – for example Killzone 2 (Sony Computer Entertainment, 2009) which Chapmandard designed the AI system for.

There are situations in which influence maps are not very useful, for example worlds with simple maps or large open terrains. Influence maps are intended for use in games with rich maps with a lot of information are where they would start to benefit the AI system (Champandard, 2011). This information can be split into different layers such as collectibles, enemies and obstacles and then combined in to one influence map which would give a very rich yet easy to assess collection of data (Roper, 2013).

There is a range of different ways to implement an influence map, and to structure the data. For simple 2D games the most sensible technique is to use 2D grids as this is often similar to the layout of the map. The data for these maps could be held in either a 2D array or a list.

Storing the data in a 2D array would usually take more memory – as there doesn’t tend to be 1 or more object in each node, so a lot of empty nodes would be stored. Contrary to this a list could simply hold the nodes which contained data and the gaps would be assumed by the user if the list contains no data for it. Each node would contain a number relative to how many enemies, coins, or whichever other object was specified there are nearby, relative to the data which the particular influence map is holding. Usually, each influence map holds information on one set of data, such as enemies or coins, and it would be down to the user to analyse the data held in each map to assess where they would travel to next.
A visual example of an influence map being represented visually is shown in Figure 3.8, where the red areas are less likely to be travelled to, and the blue are the most likely.

Figure 3.8: Influence Map displaying Strategic Reasoning (Champandard, 2011)

3.5 Considering Moving Objects

One of the most common variables that pathfinding algorithms have to adapt to is moving objects. The navigation, survival or even the goal of the player often relies on adapting to moving items in case an object has moved and obstructed the path, or could pose a threat to the player. This is especially relevant in Spelunky as the game contains lots of moving enemies, and also objects that can move through terrain.
3.5.1 Destructible Terrain

AI in games often has extra tools at its disposal which can have a large effect on the pathfinding algorithm and the way the bot behaves; this is one of the key challenges that Spelunky offers as explained in Chapter 2.5. This section will investigate retail games that offer similar challenges that the AI consider as part of their toolset, focusing on the ability to manipulate the environment.

One key example of a game where the bots consider their limited ability to destroy terrain is Bomberman (Hudson soft, 1993), a strategic arcade game. The aim of Bomberman is to destroy your opponents with bombs; this all takes place in a destructible environment which is removed on impact of an explosion from bombs, displayed in Figure 3.9. Although there is no public information as to how the Bomberman AI was implemented, it would theoretically have to consider the target position, possible paths with amount of bombs, and the strength/thickness of walls between themselves and other bots/bombs. This would be key to the effectiveness of the AI, however it is also possible that the bots simply choose the closest possible location to their target enemy and destroy terrain without actually considering it at all. This is highly likely, as the amount of memory available on the systems that Super Bomberman was launched on would have been relatively small.

One other example of a game with ‘AI’ that can manipulate the terrain is Terraria (Re-logic, 2011). This game takes place in a fully destructible world with bots that are able to manipulate the terrain as part of their toolset. The behaviour of the bots in Terraria (Re-logic, 2011) is heavily scripted, and there is no limitation to the bots with the ability to destroy terrain, the worm for example, so their path can act almost as if the terrain was never there in the first place.
Another key example of a game where the AI has tools to destroy terrain at its disposal is Worms (Team 17, 1995). Worms is a turn based strategy game which takes place in an entirely destructible environment, but after examining the bots behaviour they do not usually consider destructible terrain as part of their toolset, as it doesn’t give them any advantage. When considering their turn, if there is no clear path to the enemy they are trying to destroy the worm will attempt to shoot them through the terrain.

Two other games with destructible terrain include Red Faction (THQ, 2003) and Mercenaries 2 (Electronic Arts, 2008). Each of these games provides destructible environments, and after observing the behaviour of the bots it is clear that they do not consider their ability to destroy terrain/buildings as part of their path finding toolset yet simply adapt to the changes the player has made to the environment. This is likely a design decision as without this limitation the simplest way for the bots to reach the player would often involve destroying the environment which would be very difficult to control. Along with this, allowing bots to destroy terrain in games would often lessen the destruction experience for players, which could be rather disappointing from a gameplay standpoint.
3.6 Fog of War
One of the problems that offering a Spelunky bot API introduces is the knowledge given to the bot, especially in relation to the map layout. If the bots are able to access information about the entire map before having explored it this would give the bots a distinct advantage over a human player. This has been found to be an issue in many video games, especially in a lot of strategy games such as ‘Rise of Nations’ (*Microsoft Studios, 2003*), but can be an issue for all genres, as AI having more information than the player gives them a distinct advantage and this can often impact gameplay.

A solution that is commonly used for this is a system that is often referred to as ‘Fog of War’, which has also been implemented in to other AI toolsets, such as the Real Time Strategy (RTS) Game AI competition set up by the University of Alberta (*Buro et al. 2006*). In this particular implementation the AI was only able to obtain information from the environment where they had troops deployed; once the troops are no longer in the area the information that is available is revoked. The information is available to the AI mimics the information that is available to the player through the screen, making the battle in an RTS situation more balanced as the information that the AI had was less compromising to the player. Although in this situation it has been implemented to be beneficial to the player by improving gameplay, the *Fog of War* technique could potentially lend itself well to making fair AI competitions.

As exploring is a key part of the gameplay of Spelunky, it is more relevant to this particular problem than both Ms Pac-Man and Infinite Mario Bros, but the concept and restrictions are still mirrored in these APIs. Firstly, in Ms Pac-Man the bot can see everything, as can the player, putting them on relatively equal level information wise. The same can be said for Mario AI competition but path finding and exploring is not key yet both of these examples adhere to giving the same information to the bot as the player. Fog of war is a way of simulating this for bots, as anything under the fog of war is effectively the same as being off screen for the player (*Schwab, 2009*).
Chapter 4

4.0 Methodology and implementation

This chapter will cover the process of the development of the Spelunkbots API, detailing which tools and techniques were considered and implemented including why the set of tools provided were chosen.

4.1 Considerations

There were a lot of different factors to consider when planning the development of the Spelunkbots toolset; this included the tools being used to create it, the tools provided to developers, and the constraints applied to these tools. This section will investigate the considerations made in these areas.

4.1.1 GameMaker

As Spelunky was originally developed using GameMaker (YOYO Games, 2009), this also offered a new set of challenges and considerations; most APIs are usually created in a programming language which offers a range of benefits that are lost when developing a set of tools for GameMaker. All of the examples of AI APIs within Chapter 3 were all developed using various traditional programming languages, but GameMaker has certain limitations which meant that the approach to this project had to be slightly different. Traditionally, APIs are provided as a plugin for existing software, provided in packages which cannot be easily changed but GameMaker does not provide this type of functionality; instead, the Spelunky AI toolset that has been developed will have to be downloaded as an entire project for users to be able to get the additional functionality that the tools provide.

4.1.2 Input Considerations

The amount of button combinations in Spelunky is quite large; there are a possible 7 different buttons that could be pressed at any one time, in any combination. Although some of the combinations would have no effect it would still count as a valid input from a controller, so they are still options for the bot, too. Developers must have full control over each input, each frame; and a solution must be found to provide this.
4.1.3 Path Finding

Creating path finding for Spelunky offers a range of problems. Despite it being possible for the bot to get through each level without using bombs and destroying the environment (as each level is always created with one clear path), the way for the bot to achieve the highest score would normally be to deviate from this path. The way that the bot is able to destroy terrain is measurable as it is possible to count the amount of bombs they have and each bomb is able to destroy two land tiles in any given direction. This information must be taken into consideration for the most effective path finding. Along with bombs to complicate the path finding process, there are an array of items and enemies that the bot can manipulate to change the way that the player is able to navigate the world as discussed in section 2.3.

Along with the items that change the way the player moves, and the way they are able to manipulate the environment as discussed in section 2.5; there are also enemies to either navigate around or actually be used as a tool during navigation. Even though there could be a clear path through the terrain, it isn’t certain that the bot would actually be able to navigate this without dying as a result of the enemies.

The Spelunky map is already grid based as it is built on a map that is split in to 16x16 pixel sized segments; it lends itself perfectly to implementing a grid based pathfinding algorithm such as A* discussed in section 3.3.1.
4.1.4 Available Data when Pathfinding

For pathfinding, to ensure that there is actually challenge in making the Spelunky bots and that there is variation in their implementation, it is important to make sure that users/bots are not provided with too much information. To prevent bots from being able to see the entire level, yet effectively navigate the areas that they can see – this would be achieved by implementing a ‘fog’ as discussed in Chapter 3.6. Initially, areas that haven’t been explored would be considered as a solid node in the pathfinding algorithm but as the player progresses through the level the nodes are updated to their correct state.

Figure 4.1: Pathfinding Fog

Figure 4.1 displays the progression of the data available to the bot as it navigates throughout the level, showing that the pathfinding would be able to go greater distances once the bot has seen these areas.

Another important action when playing Spelunky is the ability to look up and down to view more of the environment that may have been missed. This can be performed only when the player is stood still and moves the camera to reveal more of the environment. This functionality is vital to finding areas that contain hidden objects and items that are hard for the bot to reach.

One other constraint that has to be considered by the fog of war is ‘dark levels’, as explained in section 2.7. To ensure that the information remains balanced with that which a real player would receive; it is important that fog is only cleared once light has passed over that node.
4.1.5 Level Landmarks
Another component that plays a key role in the game of Spelunky is the level landmarks that the player can interact with; the game will change depending on how the player chooses to interact with these items. These landmarks include shops, dice houses and altars, all of which are discussed in Chapter 2. It is important that the bot is able to analyse data about these landmarks so that it can respond in an appropriate manner; this data could range from simply the location of an Altar, to the price of items in a shop. The way in which the bot responds to each landmark is entirely down to the user implementing the bot.

3.3.6 Traps
Throughout the levels in Spelunky there are a set of traps that must be considered whilst pathfinding. The bot should be able to gather information about nearby traps, their types and the state they are currently in. Each of the traps behaves in different ways that must be considered for the implementation of the API, and for bots created with the tools. An evaluation of these different traps can be found in section 2.4. The information that the bot should be able to access for each of these traps can include:

- Location: The bot should be able to access the location of each of the traps; however, as spring traps are supposed to be hidden their location will only be updated once they have been activated.
- State: The bot should be able to access the state of some of the traps. Tiki-traps are the most important, as timing is key. The state of most other traps including boulders and crush blocks can be monitored by their movement.

4.2 General Threats
Throughout the world of Spelunky there is also a lot of general threats, such as projectiles and enemies, and it is important that the bot can get the relevant information about each threat so that it can adapt its behaviour accordingly. For example, the way that a player might want to deal with the hazard of a bat could greatly differ to the way it would handle a walking threat such as a caveman or yeti.
4.3 Hidden Objects

Often during Spelunky there are objects hidden under the terrain such as gold, gems or even items, and although they can be discovered at any time they can only be seen within the terrain when the player has certain items equipped. It is important that the bot can only access the same information as the player, meaning that to see these items it would require the Udjat Eye (Figure 4.2) or the spectacles (Figure 4.3) to be able to see the location of the items. This is an important rule to maintain as it ensures that the items that are often crucial to success for a real player are still just as important to the implementation of the bot.

Figure 4.2: Udjat Eye, Spelunky
Figure 4.3: Spectacles, Spelunky

(Spelunky Wiki, 2014) (Spelunky Wiki, 2014)
4.4 Debugging and Visualisation
As Spelunky has a lot of information within each level, and to gather from the player – the debugging and visualisation was something that had to be greatly considered when implementing the toolset for the Spelunky API. As shown in section 3.2.1, each of the existing AI APIs for both Mario and Ms Pac-Man that were explored display different information relative to the level and the player. There is a range of different information that is important to display at runtime to developers attempting to create their own Spelunky AI implementation including data regarding the level, the bots input and the bots path; all of these items must be considered during the APIs implementation.

4.5 Implementation of SpelunkBots API
This section will discuss and reflect upon how the Spelunky bots API was implemented including what tools were used, and how the problems and considerations discussed earlier in this chapter were addressed.

4.5.1 Creating a C++ DLL for GameMaker
As GameMaker Script is quite basic, and doesn’t allow storage of variables that are not publically available, an alternative had to be found which would make the workings of the API a little less accessible to users making sure that implementing a bot was still challenging and provided varied results. The best, and only solution to using GameMaker script (YoYo Games, 2008) was the ability to use Dynamic Linked Libraries written in C++. For consistency in implementation and in making the safest way possible to query the environment, as much of the framework as possible would have to be written in C++.

4.5.2 Restrictions
Combining GameMaker script with C++ Dynamic Linked Libraries did provide a lot of issues and restrictions during development of the toolset which meant that a lot of workarounds had to be found for tasks which would usually be simple. One of the biggest issues that this caused was the ability to use a wide range of variable types. Using the GameMaker runtime meant that communication between the software and the DLL was limited to sending only Strings and Doubles; however utilising this meant that tools which were only available in C++ within the DLL were able to be used with the GameMaker project.
4.6 Player Input
One of the most obvious functions that an AI API should provide is the ability to control the character. Compared to other AI API offerings in Chapter 3, such as Mario (Togelius, 2009) and Ms Pac-Man (University of Essex, 2009), the controls on Spelunky are rather complex as there are a large number of inputs and their effect changes depending on the state of the player.

For a developer to send instructions to the bot, the global variable for the button press has to be set to TRUE in the player choice script for the effect of the button press to be applied to the bot.

These variables are reset after being applied each frame, and are only applied after the entire player script has been run so that within the script the variables can be changed as often as the developer feels necessary.

The way the bot controls will change as different items are applied; this includes the way the player jumps, the behaviour of bombs the player fires and the way the player attacks. To query the items the player has equipped the global variables within the srcClearGlobals script provided by the original Spelunky source can be used. For example, the bot can query whether or not it has a jetpack equipped by using ‘global.hasJetpack’, as displayed in Figure 4.4.

Figure 4.4: Querying for jetpack in GameMaker Script

```javascript
if (global.hasJetpack)
{
    if (platformCharacterIs(ON_GROUND))
    {
        global.playerJump = true;
        global.spJumpPressedPreviously = true;
    }
    else if (!global.spJumpPressedPreviously)
    {
        global.playerJump = true;
    } else
    {
        global.spJumpPressedPreviously = false;
    }
    global.running = false;
}
```
4.7 Level Data

Data regarding the environment in Spelunky is one of the biggest pieces of information that developers would utilise when implementing a bot. All of the data for the levels environment needed to be held within a data structure in the C++ DLL as this way constraints could be easily applied to the data to prevent users from accessing too much information.

The environment data in Spelunky includes everything from standard terrain to static Traps (section 2.4) and key locations such as Altars, Dice Houses and Hidden Exits (section 2.6). Players must be able to access data about each piece of terrain, and this data must be reflective of the current state of the map; this is incredibly important, especially as bots harness the ability to destroy terrain as discussed in Section 2.1.

4.7.1 Implementing & Testing Level Data

There were a few steps to ensuring that the level data was stored safely within the C++ DLL, and as a typical Spelunky map is 32 by 43 tiles in size, it had to be done in an efficient way.

Firstly, a GameMaker script iterates through each of the terrain tiles in the environment and sets their value within the DLL. This creates a static snapshot of the environment within the DLL which can be accessed from inside the GameMaker project. The only exception here is Spring Traps, as they are meant to be hidden from the player; as a result, their location data was only added once they had been activated.

Next, the data regarding the environment had to be updated to reflect changes such as destruction, traps and hidden exits being discovered. Initially this was achieved by repeating the first step each frame, iterating over the whole environment and setting the state of every node every frame; but this was incredibly inefficient and caused the game to run slowly. To combat this, an alternative method was put in place which only updated tiles whenever their state changed using the Destroy event in GameMaker; a script was added within the event for each environment object which would cause the terrain at that node to be updated upon destruction, ensuring that the data would be correct each frame.
To ensure that using this data was easy for developers to utilise, a global variable for each type of terrain node was created, preventing users from having to memorise the value of each type of node. A table of the variables and values can be found in Appendix A.1.2.

Figure 4.5: Spelunky Level Data

To test that this data was working correctly the level layouts stored in the C++ DLL were saved to text files where they could be compared to the layout at run time, similar to the method seen in Figure 4.5. The dynamic objects in the image such as the grey blocks are empty, as the dynamic objects were yet to be implemented.
Another element that had to be considered when implementing pathfinding tools and information was liquids, as shown in Figure 4.6. Throughout Spelunky there are two different substances that can fill areas of the map which are water and lava. It was decided to make this data available in a different data set, as liquid tiles can often overlap terrain so separating them was the simplest way of ensuring that the data remained consistent with the map. The liquids were implemented in the same way as the rest of the environment with scripts in the Destroy event in GameMaker being used to ensure nodes are up to date.
4.7.2 Implementing Fog

As discussed in Chapter 4.1.4, it was important to ensure that the data that bots could receive about the environment was limited to areas which had been seen by the game camera. To ensure that the bots challenge was as similar to that of a real player as possible, Fog of War (section 3.6) was implemented.

Figure 4.7 demonstrates the implementation of the fog, including both methods that would have to be used in to the level creation stage of Spelunky and that developers can choose to access from their bot to receive information about the environment.

There are three methods in this code sample that are already implemented in to the GameMaker scripts, and only one that should be utilised by developers using this API. These three methods, ResetFogForNewLeve(), SetMapCoord() and ClearFogFromSqure() are all available from the modified Spelunky GameMaker project.

```c++
GMEXPORT void ResetFogForNewLevel()
{
  for (int i = 0; i < 32; i++)
  {
    map[i][i] = false;
    fog[i][i] = false;
  }
}

/* Call this each time a new level is loaded in a loop in GameMaker
Setting the state of the block as to whether there is a terrain
block in that coordinate.
Should also be called when a terrain block is destroyed by
*anything*
*/
GMEXPORT void SetMapCoord(int x, int y, bool state)
{
  map[x][y] = state;
}

/// Call this each time a new square is discovered in the map
GMEXPORT void ClearFogFromSquare(int x, int y)
{
  fog[x][y] = true;
}

// returns whether a node has terrain or not
// if the section has not been discovered then it returns false
GMEXPORT bool GetNodeState(int x, int y)
{
  if (fog[x][y] = true)
  {
    return map[x][y];
  }
  return false;
}
```
These methods are used to set the original state for the level, update the map nodes when terrain is destroyed and clear fog from the areas that the bot has seen. The only method in this section that the developer really needs to use is GetNodeState; the value this method returns is dependent on the state of fog in the queried location, which is demonstrated in Figure 4.8. In this figure, normal ground tiles are represented by the number 1, ladders as 2 and blank space is 0. Everything outside of the area that the bot has seen returns 1 as a result of the fog so is treated as solid. The moveable blocks in Figure 4.8 are displayed as zeroes as they are dynamic, and are treated differently as a result.

Another consideration that has to be made with the fog implementation is ‘Dark Levels’. These levels greatly restrict the player’s view, making the information displayed on the screen at any one time greatly limited; although they can also increase this by performing a number of actions. Players are able to carry torches and light designated points around the level to increase their view distance. To ensure this worked correctly an additional stage was added to the Fog of War during ‘Dark Levels’; this iterated through each of the objects which emit light, checked if they were within viewing distance and if so cleared the fog within a radius around them. This meant that the data bots would receive in these levels were reduced significantly, as it is for a real player.
Lastly, to ensure it would be possible to calculate which areas had not been explored, access was also given to the data held in the 2D array which held the value of the fog. This would allow bots to analyse which areas were unvisited, which could often be necessary in finding the level exit.

**4.7.3 Implementing Pathfinding**

Despite pathfinding solutions not being provided by any of the other APIs examined in 3.3, it was decided that a simple pathfinding solution would be provided with the Spelunkbots project. The solution was developed for two reasons; firstly to test that a bot could be created which could effectively navigate the map and secondly to provide as part of the project. The aim of providing this Path-Finding solution as part of the toolset was not to give developers the best solution to the problems which Spelunky offers, but to offer a starting point which developers could use when getting used to the Spelunkbots toolset.

After considering all of the pathfinding solutions considered in Chapter 3.3 it was decided that A-Star (3.3.1) would be the solution provided as it was simple to implement and modify, and as paths would not be created each frame there was no noticeable impact on performance.

The A-Star implementation provided was all calculated within the Spelunkbots DLL and the data used was restricted by the Fog of War. Three methods were created for use within the Spelunkbots project: CreateAStarPathFromXYtoXY(), GetNearestXPos() and GetNearestYPos(). The first method, CreateAStarPathFromXYtoXY, allows developers to pass two coordinates and in the DLL a path would be calculated and stored. The other two methods would return the closest X or Y coordinate along the path from the position passed, which presumably would be the bots’ position.
4.7.4 Dynamic Objects
Along with being able to navigate through the level, it is also imperative that bots are also able to receive information about dynamic objects in the environment such as traps and enemies (section 3.5). To ensure that the toolset was easy to manage on my side and remained consistent for users, a system needed to be created to provide information on the dynamic objects in a similar way to the map data, using C++ and a DLL. Along with it being important that the toolset is consistent, implementing ‘Fog of War’ (section 3.6) to the dynamic objects was necessary to prevent bots from gaining additional information about the level, outside of what they had explored. Seeing as this was already implemented for the pathfinding using the DLL was the most sensible way to utilise this.

Dynamic objects could not be implemented in the same way as the static objects for a number of reasons. Firstly, the dynamic objects have the ability to move; this is not considered for static terrain. Secondly, there can be more than one dynamic object in one node; using the same method as terrain for these objects would result in inaccuracies in the data relative to the amount of objects in each node, and of what type.

Figure 4.9: Spelunky Dude tackling spider webs (Spelunky Wiki, 2014)

One of the unique dynamic objects that are implemented in to the API is spider webs as displayed in Figure 4.9. Although the webs are static, it is possible that there would be more than one spider web on a tile, and they also disappear after the player has moved through them for a certain amount of time. Before the web has disappeared any item that attempts to pass through will be slowed down significantly. Webs are placed by spiders as well as being generated randomly on the levels creation.

After creating methods in the DLL to manage the creation and removal of webs (and a script in the GameMaker source to handle this), a script was provided that users could call which would return the number of webs in each tile of the map. Anywhere that the bot had not explored would return 0 as a result of the fog of war. Restricting the information to the players view was unnecessary for this particular item, as their state is highly unlikely to change as a result of anything out of the players view.
4.7.5 Implementing Moveable Blocks, Enemies and Collectable Objects

Other objects in Spelunky tend to have more variables than the spider webs and function in a way which is much different to terrain; therefore their implementation was more carefully considered. These objects include a range of enemies, moveable blocks and important NPCs such as the Damsel. As the amount of dynamic objects in the map can be quite large, iterating through each tile on the map and checking for over 100 different types of object per frame would have been incredibly inefficient and risked inaccuracies in the data if there was more than one object of a single type in a tile.
4.7.6 Enemies & Moving traps

Enemies are one of the most complex and varied entities that users will have to consider when implementing their bots, especially as the types of enemies in Spelunky are quite diverse and their abilities and behaviour differs between each type.

Table 4.1: Example Enemy Types and Behaviour

<table>
<thead>
<tr>
<th>Enemy Type</th>
<th>Behaviour</th>
</tr>
</thead>
</table>
| Bat        | - Floats in the direction of the player  
- Moves once the player is within a certain radius  
- Killed when jumped upon  
- 1 health |
| Snake      | - Travels left and right along the environment until met by an obstruction  
- Killed when jumped upon |
| Spider     | - Falls from ceiling when the player stands directly beneath its position  
- Jumps in the direction of the player along the X axis  
- Killed when jumped upon |
| Caveman    | - Walks if the player is in sight  
- Walks along the terrain  
- Stunned if jumped upon |

As displayed in Table 4.1 by listing just a range of enemy behaviours, it is easy to see that the challenges that each different enemy introduces is unique. Not all enemy types are listed in this table, as a large part of the challenge of Spelunky is learning the behaviour of the game and enemies as discussed in Chapter 2. It is important for the bot to be able to differentiate between each enemy type, and the state of each enemy so that it can adjust its behaviour accordingly.

As a result of the new information that was going to have to be stored, a new data structure was created in the DLL which would hold a list of all of the enemies in the map, with a reference to their position, type and their object ID in GameMaker. The object ID is a unique identifier for each object in a GameMaker project, which cannot be changed.
To ensure that the data held in this list was updated correctly, a similar method was used to that which was implemented for updating terrain (section 4.7.1). Scripts were added to each of the enemies Create, Step and Destroy actions which would perform different tasks:

- **Create**: Passing the object type, ID and position; the create script adds a reference to the bot to the data structure in the DLL.
- **Step**: Passing just the ID and position; the Step script would ensure that the object reference with a matching ID was up to date.
- **Destroy**: Passing just the ID, the object reference with matching ID would be removed from the list in the DLL.

Once this data was being correctly added, methods were then created which would query the amount of objects in a certain node, and also to retrieve the ID of the first object of matching type within a node. This was provided to ensure there was a way which allowed developers to query the status and direction of an enemy; this would be done using a combination of the methods provided by the Spelunkbots DLL and tools provided by GameMaker Script, a full explanation of which can be found in Appendix A.

### 4.7.7 Collectables, Items and other Dynamic Objects

Seeing that the data which bots would need to collect about enemies was quite complex, it was decided that information about other dynamic objects would be held in a separate list; although the data held regarding each object would be the same. The ID was still made accessible to these objects so that it was possible to query the state or the price of objects.

The data is managed in exactly the same way as the enemy data, using the Create, Step and Destroy actions to manage the data as described in section 4.7.6.

It was considered that *Influence Maps* (3.4) could have worked well as a structure for the data, providing a map of areas which would be most lucrative to explore, but it was decided that it would be down to developers to implement it as part of their own solution. The way in which influence maps could be implemented was considered during the implementation of collectables however, which were structured in a way which wouldn’t cause complications when attempting to create influence maps.
4.8 Analysing bots

Being able to effectively test bots is very important, especially with Spelunky as a result of the large amount of actions and choices that are available to the bots. Provided are a few different features and tools designed for testing the bots ability to handle different situations, allowing developers the ability to easily monitor their bots’ behaviour in specific situations.

4.8.1 Implementing debugging visualisation

As discussed in section 4.4, there is a lot of data to consider when playing Spelunky and it is important that developers are able to easily assess the state of their bot and the actions they are performing. Firstly, the actions which the bot is performing, or not performing, are displayed on the screen, this provides the status of the states of each button the bot has access to, similar to the information displayed as part of the Mario API (3.2.1); an example of this can be seen in Figure 4.10.

Figure 4.10: Spelunkbots Debug Display

Secondly, it was also important that data displaying the bots intended positions were displayed on screen. This would display to the user bot the bots intended direction and goal, along with possible moves that the bot could take. This is also demonstrated by Figure 4.10; the red line pointing at the gold shows the bots destination, the blue line displays its nearest path node, and the other red lines show possible moves the bot could take. This is again similar to the visual representation of bot data seen in the Mario API in Chapter 3.2.1, but was adapted to show more information about the bots final goal.
4.8.2 Explored Area

Despite the bot only being able to see the areas that it has explored it would be useful for the user to be able to see a version of the map, showing the shape of the level and demonstrating what the bot has and has not explored. As this information would take up a lot of screen space, it was decided that it would not be displayed at run time, however, the debug method SaveLevelToFile() which is accessible in the GameMaker project saves the layout of the level to a text file which the developer can examine.

4.8.3 Game speed

As games of Spelunky can be quite lengthy, running up to 30 minutes and beyond, one of the tools it was important to include was the ability to increase the running speed of the game. GameMaker already provides the ability to change the running speed of a game room but to take full advantage of this and to offer the most functionality to users to control of the game speed, providing users with the ability to increase and decrease it at run time. Allowing users to slow down the game speed also allows them to better observe the bots behaviour at points, as often Spelunky is quite fast paced. Neither the Ms Pac-Man (University of Essex, 2009) nor Mario API (Togelius, 2009) provide functionality similar to this, though in theory it would be quite simple for the user to implement.

How well increasing the speed of the game will work is entirely reliant on the capability of the system the project is running on. Some older machines may not be able to increase the speed of the game, as it requires all of the computation to be performed a lot faster. The fastest the speed was able to be increased to on the computer that it was developed on was 80 frames per second, which is just over double the regular speed Spelunky typically runs at.
4.8.4 Bot Statistics
Along with being able to measure the performance of bots at run time over one run, it is also important that a bots overall performance can be monitored. Statistics were already included as part of the original Spelunky project that can be used to track a bots performance which includes the following data:

- Number of Plays
- Number of Deaths
- Number of Wins
- Top number of Kills in one run
- Top number of Damsels (section 2.2) saved in one run
- Top amount of Money collected in one run

4.8.5 Level Creation
When developing an AI implementation, a good way to test its ability to handle situations is to provide tools that allow the user to create levels designed to stress test the bot, often referred to as test-beds. These test-beds allow users to effectively test the bots functionality. This is especially important in a game as complex as Spelunky where there are a lot of different situations for the bot to handle, with the likelihood of some of them appearing in the game being relatively low.

Figure 4.11: Cape level
Spelunky classic provides a level editor which can be accessed by pressing “F2” on the title screen. This tool can be used to create single levels, or a full Spelunky adventure containing multiple levels as demonstrated by Figure 4.11.
Within the Spelunkbots project there are a few simple levels which will allow users to test their bots to ensure they can perform various tasks, which can be seen in Table 4.2.

Table 4.2: Provided Test-Beds and Descriptions

<table>
<thead>
<tr>
<th>Level Title</th>
<th>Description</th>
<th>Starting Equipment</th>
</tr>
</thead>
</table>
| **Testbed 1** | Designed to test the basic skills of the bot – including pathfinding, using bombs and ropes, collecting items and tackling basic enemies (Spiders). | Life: 1  
Bomb: 0  
Ropes: 0 |
| **Jetpack** | Designed to test the bots ability to navigate using a jetpack – containing a simple level with lots of floating spikes and a room full of spiders. | Life: 4  
Bomb: 0  
Ropes: 0 |
| **Teleport** | Designed to test the bots ability with the teleportation device – contains a number of rooms to teleport between, and some enemies to test teleporting as a form of combat. | Life: 4  
Bomb: 0  
Ropes: 0 |
| **Web Gun** | Designed to test the bots ability to manipulate the web gun – simply a large line of spikes too long to jump across and a web gun. | Life: 4  
Bomb: 0  
Ropes: 0 |
| **Shotgun** | Simple level filled with enemies in small tunnels which prevent the player from jumping to test use of the shotgun. Very difficult due to large amount of enemies. | Life: 8  
Bomb: 0  
Ropes: 0 |
| **Cloak** | A small obstacle course designed for the cloak – contains spike and arrow traps. | Life: 4  
Bomb: 0  
Ropes: 0 |
| **Enemy Spikes** | A small level filled with spikes with enemies patrolling within – testing the bots ability to jump between enemies. | Life: 4  
Bomb: 0  
Ropes: 0 |
| **Fall Damage** | A tall level containing one long drop enough to cause ‘fall damage’ to the player. Contains patrolling enemies beneath which would soften blow – designed to test the bots ability to control movement whilst falling. | Life: 1  
Bomb: 0  
Ropes: 0 |
| **Maze** | A labyrinth of ladders and tunnels, designed to test the bots pathfinding, use of looking up/down and ability to gather gold. | Life: 4  
Bomb: 0  
Ropes: 0 |
| **Climbing Gloves** | A very simple level to test the bots ability to climb with green climbing gloves equipped. | Life: 4  
Bomb: 0  
Ropes: 0 |
| **Knife** | A small level containing a knife and a tunnel full of enemies. Designed to test the bots knife combat. | Life: 4  
Bomb: 0  
Ropes: 0 |
| **Rocks** | A level that simply contains rocks and bats – designed to test combat with thrown objects and jumping. | Life: 4  
Bomb: 0  
Ropes: 0 |
| **Traps** | A level filled with arrow traps to test the bots ability to use objects to protect them from damage | Life: 4  
Bomb: 0  
Ropes: 0 |
As well as making these levels linear, with simple tasks to complete, the levels are also littered with collectables that bots could only receive by carefully managing their resources. For the bots to effectively clear the levels they would have to utilise all of their abilities, including the ability to pan the camera by looking up and down to clear fog. This was a very important test for bots, as these kinds of techniques are often the key to achieving the highest possible score in Spelunky.

![Figure 4.12: Level selection menu in Spelunky](image)

Testing a bot implementation on these levels is also very simple; once the main screen has been reached press ‘F3’, and you are presented with a list of levels as displayed above in Figure 4.12. Once the desired level has been selected the implemented bot will be added automatically.
Chapter 5 – Testing & Analysis

5.0 Testing & Analysis

To ensure that the project had effectively achieved the objectives set in Chapter 1, two core items needed to be tested. Firstly, bots with different behaviours had to be implemented to demonstrate different information being accessed and actions being performed. To follow this, the solution would have to be provided to a handful of developers to ensure they are able to utilise the tools provided to create simple bots.

5.1 Behaviour Implementation

As a result of Spelunky having such a wide range of situations to be encountered, various bots were created with a range of behaviours to demonstrate the toolset in action; explanations of each can be seen in Table 5.1.

Table 5.1: Spelunkbot behaviour implementations

<table>
<thead>
<tr>
<th>Bot Title</th>
<th>Behaviour</th>
<th>Level Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘Gold Digger’</td>
<td>Searches for gold bars, travels in the direction of the earliest gold bar seen closest to the left of the map. Travels using the provided A-Star solution.</td>
<td>“Gold”</td>
</tr>
<tr>
<td>‘Discovery Dan’</td>
<td>Searches the level for the exit based on fog by travelling to the nearest node on the same X axis which has not been uncovered. Travels using the provided A-Star.</td>
<td>“Fog”</td>
</tr>
<tr>
<td>‘Not So Solid Snake’</td>
<td>Sets destination to above the first snake discovered and jumps to the location, and repeats until all of the snakes have been destroyed. Travels using the provided A-Star solution.</td>
<td>“Snakes”</td>
</tr>
<tr>
<td>‘Jetpack Joyride’</td>
<td>Searches for a jetpack and travels to its location; once equipped, the bot travels to the nearest gold bar whilst flying. Travels using the provided A-Star solution.</td>
<td>“JetpackC”</td>
</tr>
<tr>
<td>‘Indie’</td>
<td>Searches for gold bars and golden heads; once in contact with a gold head it picks it up and carries it to the exit. Travels using the provided A-Star solution. Demonstrates travelling around a corner.</td>
<td>“Idol”</td>
</tr>
<tr>
<td>‘Shotgun total warrior’</td>
<td>Finds a shotgun, equips it, and then searches the level for gold whilst shooting continuously. Travels using the provided A-Star solution.</td>
<td>“Shotgold”</td>
</tr>
<tr>
<td>‘Sean Bean’</td>
<td>Jumps to the nearest position with spikes just below it. Travels using the provided A-Star solution.</td>
<td>“Spikes”</td>
</tr>
<tr>
<td>‘JarJar’</td>
<td>Attacks Jar objects until there is no more. Travels using the provided A-Star solution.</td>
<td>“Rocks”</td>
</tr>
</tbody>
</table>
The aim of creating these bots was not to beat all of the problems that Spelunky introduces, but to complete a specific goal. Each of the scripts for these bots are provided as part of the project to demonstrate how different actions could be performed in Spelunky, the first of which is provided as an example in the user guide in A.1. Each of the levels provided were designed to create different problems for bots to face, an example of which can be seen in Figure 5.1; the “Idol” level which the Indie bot faces tests the bots ability to create a path around a corner. A video demonstrating all of these different behaviours can be found in Appendix B, and the source for each bot can be found in Appendix D.

Figure 5.1: Level with pathfinding around corner
5.2 Testing Spelunkbots with Developers

To ensure that the tools provided in the Spelunkbots API were usable and effective the project was provided to developers, along with the User Guide found in Appendix A. Each developer was given a simple task to accomplish with their bot using the tools provided, and the suggested constraints; these mostly involved finding and using certain objects. Descriptions of the implementations provided can be found in Table 4.3.

Table 4.3: Spelunky bot implementations

<table>
<thead>
<tr>
<th>Developer Name</th>
<th>Technique Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan Pearce (Appendix E.1)</td>
<td>• Heavily based off the example provided in the User Guide</td>
</tr>
<tr>
<td></td>
<td>• Utilises the provided A-Star algorithm</td>
</tr>
<tr>
<td></td>
<td>• Searches the viewed area for a jet-pack, and once this is equipped searches for gold</td>
</tr>
<tr>
<td>Andrew Roper (Appendix E.2)</td>
<td>• Searches for golden idols</td>
</tr>
<tr>
<td></td>
<td>• Walks to idols</td>
</tr>
<tr>
<td></td>
<td>• Picks up idol when in the same location</td>
</tr>
<tr>
<td></td>
<td>• Jumps and throws the idol in a random location and repeats</td>
</tr>
<tr>
<td>Matthew Smith (Appendix E.3)</td>
<td>• Walks towards the direction of a mattock, picks it up when in the same node and attacks if there is terrain in the node ahead once it is held.</td>
</tr>
<tr>
<td></td>
<td>• Does not use A-Star or pathfinding</td>
</tr>
<tr>
<td>James Hobson (Appendix E.4)</td>
<td>• Searches for Idols</td>
</tr>
<tr>
<td></td>
<td>• Utilises the A-Star provided</td>
</tr>
<tr>
<td></td>
<td>• Jumps if it is below the goal</td>
</tr>
<tr>
<td></td>
<td>• Swims to the exit when it has an idol</td>
</tr>
</tbody>
</table>

Despite the fact that these solutions did not differ a great deal from the examples provided, it demonstrates that developers are able to understand the tools being used and manipulate them to fit different needs by assessing the example and referring to the user guide found in Appendix A.
5.3 Summary of Results

Observing the range of different behaviours implemented in Section 5.2 and Section 5.3, it is clear that the tools provided provide developers with useable, yet flexible tools that allow an effective set of data to be queried. Although some developers chose to hard code bots using the tools, it displayed that the tools can suit different approaches to implementations and still work effectively for their desired purpose.
Chapter 6 - Conclusion

In this dissertation, the main aim was to objectively examine existing AI APIs and to create a new toolset for Spelunky which extended from what was discovered. The paper began by introducing two of the most popular APIs, Ms Pac-Man (University of Essex, 2009) and Infinite Mario Bros (Togelius, 2009).

To begin, we evaluated the rules and restrictions of each of these toolsets, the challenges they introduce to developers and the tools they offered to help solve the problem that each game introduced. Each of these APIs only provided information as to what was on the screen – meaning that they have very much the same information as a human player. Each of these toolsets also provided additional information on the screen for developers to assess their bots behaviour and performance at runtime without stopping the game to examine the debugger. This was a useful tool for developers as it meant that bots behaviour could better evaluated just through observing the data on the screen.

To follow this, we then examined various techniques used for implementing AI in both AI competitions and games in general; this meant that we could assess what features would be needed for the toolset and how to structure the tools in a way which could make implementing these techniques less difficult. We began by examining various methods of path-finding in games, as this would be one of the key components of creating a successful AI for Spelunky. This included discussing and evaluating A-Star, D-Star, D-Star Lite, Rapid Symmetry Reduction and Jump Point Search. Despite other algorithms having great advantages in performance, and providing different sets of information, A-Star seemed to be the easiest to implement along with the most commonly implemented in both games and as solutions to problems presented by API competitions. The performance issues presented by A-Star were only relevant when calculating a large number of paths, or in expansive environments.
To continue from pathfinding, we then investigated ‘Fog of War’, looking at situations where it had been implemented previously in other games and APIs. The only examples found of this being implemented were in Real Time Strategy games, but the concept lent itself perfectly to what was needed to be achieved in Spelunky. Limiting the data of the bot dependant on the areas seen was incredibly important to a game where exploration and map analysis is so core to the gameplay, as the bot should have no distinct advantage over a real player.

In the literature review we also investigated influence maps and how they can be utilised in different ways to benefit different play strategies in various games. Influence maps could prove an incredibly effective strategy when applied to Spelunky, as they could allow bots to evaluate which areas are more important to access and thus justify spend resources reaching.

In conclusion of the literature review we explored games with destructible environments, and how AI takes them into consideration with planning and path-finding. As many of the examples of games which offered destructible environments did not offer source code for the bots, this section was largely based on observation of their behaviour. Most bots did not seem to consider destructible environments as part of their toolset; this often seemed to be a choice to benefit the games’ design in games such as Red Faction (THQ, 2003) and Mercenaries (Electronic Arts, 2008), ensuring that the bots did not destroy everything before the player had a chance.

Moving on to the implementation, the aim was that the API would provide features which allow developers to create effective bots yet limit the information the bots are able to receive in order to ensure they are given no advantage over a human player. Within this section we considered problems that players face within Spelunky and the information that would be needed to tackle them effectively. The tools were written in a combination of C++ and GameMaker Language.

The main features that we investigated throughout the methodology included control, data structure and pathfinding. Firstly, developers needed to be able to control the bot dependant on its current state. To allow this, a flag was provided for each button press that a real player could use, if they were set to true during the bots calculations these buttons would be pressed.
The state of each of the buttons were reset each frame, prior to the bots next move. A range of tools such as whether the player is in the air, what items are equipped and how many resources the bot had remaining were also provided to developers, as explained in the user guide found in Appendix A. These three tools were already part of the Spelunky source, but can safely be used to query by the bot.

The most important stage of the implementation was how the data of enemies, items and the environment were structured and updated. Firstly, the environment was iterated over at the beginning of each level; this meant that the entire environment was then held in a 2D array in C++, holding the value of each grid piece. Once this had been implemented, a method call was added to each type of terrain which would empty the node in the C++ array once it had been destroyed; this ensured that the data held was consistently correct. This was tested by saving the data to a text file and comparing the data held to screenshots of the map.

Next, dynamic objects were implemented. Dynamic objects had to work in a different way to the static environment as scanning the entire map each frame was too much work, yet the data needed to be correct each frame to display their position. To tackle this, the ID of each enemy was stored along with its position in a list in the C++ DLL. To ensure that this data was correct each frame, each object in GameMaker was given a script to update the object with matching ID in the list to the correct position. Scripts were also added in the create and destroy actions in GameMaker to ensure that enemies were added on creation, and always removed when needed, otherwise bots would be responding to enemies that aren’t actually present.

To ensure that the data provided to the bot was not too extensive, fog of war (3.X) was then implemented to these two sets of data. The bots’ view is updated each frame and only when querying areas that it had seen would it be returned with sensible data. If the bot attempts to query an area it has not discovered the data returned would always return as if it was a solid wall. As a result, the fog data was also made available to the bot; this meant that it is possible to analyse which areas had not been explored, which could assist in reaching the level exit.
Once all of the data access and limitation had been implemented, a simple A-Star pathfinding solution was provided. Although it is really up to developers to create their own pathfinding for Spelunky, providing a simple solution could be used as a starting point for developers, and to also test my own implementations. The path-finding provided is all contained within the Spelunkbots DLL.

I feel that the project performs the desired task well, providing tools which allows developers of varying skill sets to create bots which handle the wide range of challenges that Spelunky offers. Along with providing tools to developers, I feel that the information available to bots was suitably limited, especially considering the Fog of War implementation which effectively limits the data the bot can access. Despite not implementing all of the methods considered in the literature review, the data was structured in a way which would complement their implementation.
5.1 Spelunkbots as a Challenging Domain

One of the hopes with this project was that it would prove that Spelunky was a challenging domain which would prove an interesting problem for developers to solve. When comparing to the Mario (Togelius, 2009) and Ms Pac-Man (University of Essex, 2009) APIs discussed in Chapter 3, and when observing the wide range of challenges which Spelunky introduces in Chapter 2 this is certainly the case.

Where other APIs which were explored such as Mario and Pac-Man contained a set of quite rigid rules and mechanics, Spelunky offers a much richer gameplay experience and a range of problems that can be solved in thousands of different ways.

One of the largest problems which Spelunky introduces is resource management; as bots are given limited resource to destroy the environment to gather more resources or points. This is incredibly interesting as there is no guarantee that spending resource will pay off, and the order in which items are acquired could greatly impact the performance of a bot in a level. Along with this there are a huge collection of items which can be equipped in almost any combination which greatly alter the way the game plays, but no combination of tools is guaranteed in any one play through.

Similarly to Mario AI, the levels within Spelunky are all randomly generated providing a fresh set of challenges each time the game is played.

As there is no explicit solution to success in Spelunky, the variety in the techniques used in attempting to solve the expansive problems which it introduces would be vast, making it a very interesting challenge for developers.

5.2 Recommendation for future research in Spelunky AI

Even though I feel that the implementation meets the target criteria successfully, the toolkit provided does leave a lot of room for expansion. As displayed by both the Mario and Ms Pac-Man AI toolsets discussed in Chapter 3.1, there are a range of different implementations and competitions that can be made from the same games (section 3.1). Spelunky offers an extensive range of possibilities for creating competitions which would offer very different challenges for AI. These competitions could include progressing as far as possible without collecting gold, completing the game in under a certain time or completing the game ‘the hard way’ (taking an alternative route through the game, considered to be the greatest challenge of them all).
If anyone desired to continue research in to AI in Spelunky some of the interesting situations that could be implemented by manipulating the tools provided in this dissertation could include:

- **Team based Spelunky**: Adding additional bots to levels and having the bots able to communicate, as a large portion of playing Spelunky HD (Mossmouth, 2012) is the multiplayer capability of the game.

- **Enemy AI**: As all of the enemies in Spelunky are currently running using incredibly basic scripts, adding bots with AI could be an interesting project to run either against players or bots.

- **Arenas**: Another addition in Spelunky HD (Mossmouth, 2012) was multiplayer battles. If battles were implemented into Spelunky Classic it could be very interesting to write competitive bots as the tactics for this are completely different to those which would be used to progress through levels. The tools provided with this project could be quite effectively manipulated to work for battles.

- **Level challenges**: Competitions where entire levels are underwater, or some kind of environment change is made, perhaps endless, procedurally generated levels. This would result in bots behaviour being optimised to face certain problems.

- **Gamemaker Studio**: Upgrading the project to work with GameMaker: Studio (GameMaker: Studio, 2013) would mean that it is using more recent software, making the project more accessible.

- **Implementing Forward Modelling**: One tool that could be interesting to implement when considering the challenges presented by Spelunky and the API toolset created for this dissertation is ‘Forward Modelling’. This would involve creating a way to simulate steps taken in the game to predict the future state of the Spelunky world, which is a technique used when implementing ‘Monte-Carlo Tree Search’, a complex algorithm for calculating decisions which is often applied to games.
5.3 Summary

Overall, I consider the implementation of the Spelunkbots API to be successful in achieving its desired goal. Reasonable data is available to developers creating AI for Spelunky, including limited data about the environment, and data about the bots actions displayed on the screen. Although the pathfinding provided does not evaluate resources available such as bombs, I feel the tools provided are effective enough to implement this effectively.
References


GameMaker (2009), software, version 8.1, YoYo Games

GameMaker: Studio (2013) software, version 1.3, YoYo Games


Microsoft Game Studios 2003, ‘Rise of Nations’, computer game: PC, Timonium, Big Huge Games

Mossmouth 2009, ‘Spelunky’, computer game: PC, Pasadena, Mossmouth

Mossmouth 2012, ‘Spelunky HD’, computer game: PC, Pasadena, Mossmouth

Namco 1982, ‘Ms Pac-Man’, arcade game: Chicago, Midway

Nintendo 1985, ‘Super Mario Bros’, console game: NES, Kyoto, Midway

Re-Logic 2011, Terraria, computer game: PC, Indiana, Re-Logic

Roper, A (2013) Implementing a high level planner into a real-time strategy game using the iThink planning library


Team 17 1995, ‘Worms’, console game: Sony PlayStation, Wakefield, Team 17


Bibliography

Yu, D; MossMouth, ‘Spelunky Classic & Source’, software: PC, Pasadena, Mossmouth
Available: http://spelunkyworld.com/original.html


Togelius, J (2009). Papers Related to Mario AI. Available:

YoYo Games (2008). Creating A DLL in C++ for use with GameMaker 8.1 Available:
Appendix

Appendix A: Software User Guide

To run this project you will need to be running GameMaker Professional 8.1. This project builds upon the original open source Spelunky source code. (Derek Yu & Mossmouth, 2009).

As discussed throughout Chapter 4, there are a range of tools available at your bots disposal to get information about the environment, your character and the general state of the game. This user guide will provide information about all of the different tools available, and how they could be utilised. This guide uses the GameMaker project found in Appendix C.

A.1: Creating your bot

To create your own bot you must use the scripts provided in project titled PlayerChoice, SetupPlayerVariables and ClearPlayerVariables. These scripts can be found in the AI Toolset folder under BOTSCRIPTS as displayed in Figure A.1.

Figure A.1: Bot script locations

All of your bots actions should be performed in the “PlayerChoice” script, but the other two scripts can be used to setup variables which your implementation may depend on.
A.1.1: Character input

There are a range of decisions which can be made by your bot implementation during each frame, all of which are controlled with global variables which essentially control the button that are being pressed in that frame. To use any of the variables, you simply have to set its value to true – when implementing you don’t have to worry about resetting the variables as this is done for you in a separate script.

<table>
<thead>
<tr>
<th>Global Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>global.goLeft</td>
</tr>
<tr>
<td>global.goRight</td>
</tr>
<tr>
<td>global.jump</td>
</tr>
<tr>
<td>global.duck</td>
</tr>
<tr>
<td>global.lookUp</td>
</tr>
<tr>
<td>global.attack</td>
</tr>
<tr>
<td>global.running</td>
</tr>
<tr>
<td>global.payp (pay in shop)</td>
</tr>
<tr>
<td>global.ropep (release rope)</td>
</tr>
<tr>
<td>global.bombp (release bomb)</td>
</tr>
</tbody>
</table>
A.1.2: level information

There is a large amount of data that can be gathered using the tools provided – all of which should be used when implementing a bot, as they also apply restrictions to the data that the bot is able to receive – using tools such as Fog of War (section 3.6).

Table A.2: Spelunkbots API useful methods

<table>
<thead>
<tr>
<th>Script to call</th>
<th>Data to pass</th>
<th>Information returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetNodeState(x,y)</td>
<td>The X and Y position of the node in question – though the actual position passed should be divided by 16</td>
<td>The state of the terrain in this block – returns the static object in this position. Global variables holding each type are displayed in Table 10.</td>
</tr>
<tr>
<td>GetNumberOfCollectableTypeInNodeXY(Type,x,y)</td>
<td>The X and Y position of the node you want to query, and the item you want to search for</td>
<td>The amount of the collectable type in the node. When passing in a world coordinate, divide by 16.</td>
</tr>
<tr>
<td>GetNumberOfEnemyTypeInNodeXY(Type,X,Y)</td>
<td>The X and Y position of the node you want to query and the enemy you want to search for</td>
<td>The amount of the enemy type in that node. When passing in a world coordinate, divide by 16.</td>
</tr>
<tr>
<td>NumberOfWebsAtNode(X,Y)</td>
<td>The X and Y position of the node you want to query</td>
<td>The number of spider webs in the queried node. When passing in a world coordinate, divide by 16.</td>
</tr>
</tbody>
</table>
To prevent users from having to remember the values of the data being held in the nodes, and what they are equivalent to, some global variables were created in GameMaker that the user can use to prevent them from having to reference the values. This method helps developers keep their code more maintainable and readable. Tables of the variables available can be found in Table A.3, Table A.4, Table A.5 and Table A.6.

Table A.3: Terrain Nodes

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>global.spEmptyNode</td>
<td>0</td>
</tr>
<tr>
<td>global.spStandardTerrain</td>
<td>1</td>
</tr>
<tr>
<td>global.spLadder</td>
<td>2</td>
</tr>
<tr>
<td>global.spEntrance</td>
<td>3</td>
</tr>
<tr>
<td>global.spExit</td>
<td>4</td>
</tr>
<tr>
<td>global.spSacAltar</td>
<td>5</td>
</tr>
<tr>
<td>global.spArrowTrapRight</td>
<td>6</td>
</tr>
<tr>
<td>global.spArrowTrapLeft</td>
<td>7</td>
</tr>
<tr>
<td>global.spIsInShop</td>
<td>8</td>
</tr>
<tr>
<td>global.spIce</td>
<td>9</td>
</tr>
<tr>
<td>global.spSpike</td>
<td>10</td>
</tr>
<tr>
<td>global.spSpearTrap</td>
<td>11</td>
</tr>
</tbody>
</table>

Table A.4: liquid nodes

<table>
<thead>
<tr>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>global.spEmptyNode</td>
</tr>
<tr>
<td>global.spLiquidWater</td>
</tr>
<tr>
<td>global.spLiquidLava</td>
</tr>
</tbody>
</table>
### Table A.5: Collectable nodes

<table>
<thead>
<tr>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>global.spGoldBar</td>
</tr>
<tr>
<td>global.spGoldBars</td>
</tr>
<tr>
<td>global.spEmerald</td>
</tr>
<tr>
<td>global.spEmeraldBig</td>
</tr>
<tr>
<td>global.spSapphire</td>
</tr>
<tr>
<td>global.spSapphireBig</td>
</tr>
<tr>
<td>global.spRuby</td>
</tr>
<tr>
<td>global.spRubyBig</td>
</tr>
<tr>
<td>global.spDiamond</td>
</tr>
<tr>
<td>global.spGoldNugget</td>
</tr>
<tr>
<td>global.spGoldChunk</td>
</tr>
<tr>
<td>global.spChest</td>
</tr>
<tr>
<td>global.spLockedChest</td>
</tr>
<tr>
<td>global.spKey</td>
</tr>
<tr>
<td>global.spCrate</td>
</tr>
<tr>
<td>global.spFlareCrate</td>
</tr>
<tr>
<td>global.spBombBag</td>
</tr>
<tr>
<td>global.spBombBox</td>
</tr>
<tr>
<td>global.spPaste</td>
</tr>
<tr>
<td>global.spRopePile</td>
</tr>
<tr>
<td>global.spParachute</td>
</tr>
<tr>
<td>global.spCompass</td>
</tr>
<tr>
<td>global.spSpringShoes</td>
</tr>
<tr>
<td>global.spSpikeShoes</td>
</tr>
<tr>
<td>global.spJordans</td>
</tr>
<tr>
<td>global.spSpecs</td>
</tr>
<tr>
<td>global.spUdjat</td>
</tr>
<tr>
<td>global.spCrown</td>
</tr>
<tr>
<td>global.spKapala</td>
</tr>
<tr>
<td>global.spAnkh</td>
</tr>
<tr>
<td>global.spGloves</td>
</tr>
<tr>
<td>global.spMitt</td>
</tr>
<tr>
<td>global.spJetpack</td>
</tr>
<tr>
<td>global.spCape</td>
</tr>
</tbody>
</table>

### Table A.6: Enemy Variables

<table>
<thead>
<tr>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>global.spGhost</td>
</tr>
<tr>
<td>global.spBat</td>
</tr>
<tr>
<td>global.spScarab</td>
</tr>
<tr>
<td>global.spSpider</td>
</tr>
<tr>
<td>global.spGiantSpiderHang</td>
</tr>
<tr>
<td>global.spGiantSpider</td>
</tr>
<tr>
<td>global.spFrog</td>
</tr>
<tr>
<td>global.spFireFrog</td>
</tr>
<tr>
<td>global.spZombie</td>
</tr>
<tr>
<td>global.spVampire</td>
</tr>
<tr>
<td>global.spPiranha</td>
</tr>
<tr>
<td>global.spJaws</td>
</tr>
<tr>
<td>global.spDeadFish</td>
</tr>
<tr>
<td>global.spManTrap</td>
</tr>
<tr>
<td>global.spMonkey</td>
</tr>
<tr>
<td>global.spYeti</td>
</tr>
<tr>
<td>global.spYetiKing</td>
</tr>
<tr>
<td>global.spUFO = 18</td>
</tr>
<tr>
<td>global.spUFOCrash</td>
</tr>
<tr>
<td>global.spAlienEject</td>
</tr>
<tr>
<td>global.spAlien</td>
</tr>
<tr>
<td>global.spAlienBoss</td>
</tr>
<tr>
<td>global.spBarrierEmitter</td>
</tr>
<tr>
<td>global.spBarrier</td>
</tr>
<tr>
<td>global.spCaveman</td>
</tr>
<tr>
<td>global.spHawkman</td>
</tr>
<tr>
<td>global.spMagma</td>
</tr>
<tr>
<td>global.spMagmaTrail</td>
</tr>
<tr>
<td>global.spMagmaMan</td>
</tr>
<tr>
<td>global.spTombLord</td>
</tr>
<tr>
<td>global.spOlmec</td>
</tr>
<tr>
<td>global.spOlmecDebris</td>
</tr>
<tr>
<td>global.spSnake</td>
</tr>
<tr>
<td>global.spSpiderHang</td>
</tr>
<tr>
<td>global.spMagmaMan</td>
</tr>
<tr>
<td>global.spShopKeeper</td>
</tr>
<tr>
<td>global.spSmashTrap</td>
</tr>
<tr>
<td>global.spCeilingTrap</td>
</tr>
<tr>
<td>global.spBoulder</td>
</tr>
<tr>
<td>global.spBones</td>
</tr>
<tr>
<td>global.spSpringTrap</td>
</tr>
</tbody>
</table>
A.2: Useful Calls

The original Spelunky (Mossmouth, 2009) project offers a range of useful tools, originally intended for use within the game, which can be used to retrieve information about the environment and the state of the player. Some of the tools which can be useful for developing a bot can be found below, with explanations as to how they could be utilised.

A.2.1: Player Queries

One of the most crucial items within Spelunky (Mossmouth, 2009) that the developer will need to get information about is the player itself. As the player has a wide range of states and varying abilities it is important that the developer is able to adapt to different situations because bots with different tools act completely differently to others. Some of these tools and methods have been made possible or more accessible as a result of changes made to the Spunkbots project.

- **platformCharacterIs(ON_GROUND/IN_AIR):** Using either of these calls, it is possible to check whether the player is grounded or not. This is especially useful when trying to use tools such as the jetpack or cape as these require actions to be performed whilst the bot is in the air.

- **Common Player Resource:** Similarly to querying the inventory, there are global variables provided that allows developers to query the amount of common resource they have available; this includes ropes, bombs and health. These variables can be seen in Table A.7. Each of these variables stores a number reflecting the amount of that resource available.

<table>
<thead>
<tr>
<th>Variable Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>global.pLife</td>
</tr>
<tr>
<td>global.bombs</td>
</tr>
<tr>
<td>global.rope</td>
</tr>
</tbody>
</table>

Table A.7: Player Resource Variables
• **Inventory Querying:** As a result of the large amount of items it is possible for the bot to have equipped, and the different behaviours that can come of this, it is important that the developer is able to adapt to the state of the bot’s inventory. There are a range of global variables provided within the original Spelunky source code that developers can utilise such as `global.hasStickyBombs` and `global.hasGloves`. Although some of the variables available should not be utilised as it would give the bots a significant advantage over the player, a list of the variables that could be used can be found in Table A.9.

Table A.9: Inventory Query Variables

<table>
<thead>
<tr>
<th>Variable Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>global.hasUdjatEye</code></td>
</tr>
<tr>
<td><code>global.hasAnkh</code></td>
</tr>
<tr>
<td><code>global.hasCrown</code></td>
</tr>
<tr>
<td><code>global.hasKapala</code></td>
</tr>
<tr>
<td><code>global.hasStickyBombs</code></td>
</tr>
<tr>
<td><code>global.hasSpectacles</code></td>
</tr>
<tr>
<td><code>global.hasCompass</code></td>
</tr>
<tr>
<td><code>global.hasParachute</code></td>
</tr>
<tr>
<td><code>global.hasSpringShoes</code></td>
</tr>
<tr>
<td><code>global.hasCape</code></td>
</tr>
<tr>
<td><code>global.hasJetpack</code></td>
</tr>
<tr>
<td><code>global.hasJordans</code></td>
</tr>
<tr>
<td><code>global.hasGloves</code></td>
</tr>
<tr>
<td><code>global.hasMitt</code></td>
</tr>
<tr>
<td><code>global.udjatBlink</code></td>
</tr>
</tbody>
</table>

• **Shopkeepers Murdered:** One of the choices the bot might make in the game is to kill shopkeepers – this would result in shopkeepers being aggressive. To keep track of the state of the shopkeepers the variable `global.murderer` can be used – this returns a Boolean dependant on whether you have murdered a shopkeeper or not.
• **Pickup Item Info:** As Spelunky players and bots have the ability to pick up items; it is important that it is possible to query whether the bot has any items equipped. To do so, the variables found in Table A.10 can be used.

<table>
<thead>
<tr>
<th>Method</th>
<th>Stores</th>
</tr>
</thead>
<tbody>
<tr>
<td>holdItem</td>
<td>Boolean. True if holding item.</td>
</tr>
<tr>
<td>holdItem.type</td>
<td>String holding the type of object, for example: “Shotgun”.</td>
</tr>
</tbody>
</table>

Table A.10: Player Resource Variables

A.3.2: Environment Queries

• **Room differentiation:** As there are a range of different locations within Spelunky, the bot may need to perform differently in different rooms – such as the title room, a standard level or the final boss level. There are a range of tools available from both GameMaker, the original Spelunky source and the modified source which make this easier; these tools are listed in Table A.11.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsRealLevel()</td>
<td>Provided by the original Spelunky implementation, this returns a Boolean dependant on whether the current room is a ‘real’ level.</td>
</tr>
<tr>
<td>isRoom(&quot;ROOMNAME&quot;)</td>
<td>Provided with GameMaker, this allows developers to establish the exact type of room, allowing different implementations for different room types. ROOMNAME would be replaced with the title of the room expected, such as “rTitle”.</td>
</tr>
</tbody>
</table>

Table A.11: Spelunkbots Room Queries
• **Location data:** Although it is possible to get some data regarding the environment around the player from ‘platformCharacterIs(ON_GROUND/IN_AIR)’, sometimes more detail may be needed. To find out of type of terrain the player is standing on – or whether the bot is in a shop or not – GetNodeState(x,y) can be used. The values passed to this method should be world coordinates divided by 16.

• **Compass:** If the bot is holding a compass, it is possible to get the location of the level exit using GetGoalPositionX() and GetGoalPositionY(). If the bot doesn’t have a compass and attempts to use these scripts 0 will be returned.

• **Level Feel:** The level feel can often give a good indication of what the level may contain, which may change the way you want your bot to behave. To get the level feel, global.message and global.message2 can be accessed; each of these variables contains a string holding the message.
A.3.3 State Queries

It is often important to know more information about an object, such as an enemies’ state or an objects’ price. The methods and variables which can be used to access this information can be found in Table A.12.

Table A.12: Receiving object state

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Methods/Variables</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enemy</td>
<td>GetFirstEnemyOfTypeInNode(Type,X,Y)</td>
<td>Returns the ID of the first enemy found in the questioned node with a matching type.</td>
</tr>
<tr>
<td></td>
<td>Status</td>
<td>Holds a number relative to the state it is in such as 0 for IDLE, and 99 for Dead. The status list for each enemy can be found in their create action within GameMaker.</td>
</tr>
<tr>
<td>Item/Collectable</td>
<td>GetFirstCollectableOfTypeInNode(Type,X,Y)</td>
<td>Returns the ID of the first item found in the questioned node with a matching type.</td>
</tr>
<tr>
<td></td>
<td>Cost</td>
<td>Only items which can be found in a shop have a cost value – this is 0 unless it is owned by a shopkeeper.</td>
</tr>
</tbody>
</table>

The variables and methods available to query states can be used as displayed in Figure A.1 – the ‘with’ method allows you to get additional information of an object with an ID which is passed. In this example, a message is displayed in GameMaker stating the price of a Jetpack.

Figure A1: Getting the cost of an item

```plaintext
with (GetIDOfFirstCollectableOfTypeInNode(global.spJetpack,1,j))
{
    show_message(cost);
}
```
A.4: Utilising Pathfinding

Despite it being recommended that developers implement their own pathfinding to ensure the most amount of control, the A-Star provided can be used to find routes to objects or destinations of the developer’s choice; all whilst searching with the Fog of War (Section 3.6) considered.

There are 3 different methods available which are used to utilise the pathfinding provided, these are explained in Table A.13.

Table A.13: Spelunkbots Pathfinding Methods

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateAStarPathFromXYtoXY(x,y,x,y)</td>
<td>This creates a path from one location to another using A-Star. It does not allow the user to define what is considered in the pathfinding, as it is only provided as a simple option for pathfinding – more complex implementations would have to be handled by the developer. The two variables passed should be world coordinates divided by 16.</td>
</tr>
<tr>
<td>GetNearestXPos(x,y)</td>
<td>Returns the X position for the nearest node, the x and y passed should be the coordinates of the player divided by 16.</td>
</tr>
<tr>
<td>GetNearestYPos(x,y)</td>
<td>Returns the Y position for the nearest node, the x and y passed should be the coordinates of the player divided by 16.</td>
</tr>
<tr>
<td>GetFogState(x,y)</td>
<td>Returns the state of the fog in this position – used for finding where in the map is undiscovered. The x and y passed should be world coordinates divided by 16.</td>
</tr>
</tbody>
</table>
An example bot implementation which implements very basic A-Star using the methods provided can be seen in A.1. This implementation was designed specific for the test-bed provided titled “Gold” (Section 4.8.5). There are a number of steps performed by this bot; firstly, if there is no goal it searches the environment for a gold bar. This bot only considers gold bars, so other items are completely ignored by it. Once the bot has found a gold bar a path is created. The bot then proceeds to travel towards the nearest possible node at all points until the gold has been collected. Once the bot has collected the gold, the process is repeated until it can find no more – at which point it becomes stationary.

```csharp
if ('global.hasGold')
{
    for (i = 0; i < 42; i++)
    {
        for (j = 0; j < 54; j++)
        {
            if (GetNumberOfCollectableTypeInNodeXY('global.goldBar', i, j))
            {
                global.targetX = i * 16;
                global.targetY = j * 16;
                global.hasGold = true;
                global.itemSet = true;
                CreateAStarPathFromXYtoXY(x/16,y/16, global.targetX/16, global.targetY/16);
                return 0;
            }
        }
    }
}
else
{
    // Check if the item we're looking for is still there - or did we collect it?
    if (GetNumberOfCollectableTypeInNodeXY('global.goldBar', global.targetX / 16,global.targetY / 16) == 0)
    {
        global.hasGold = false;
        global.itemSet = false;
    }
    // Go towards the x point of the closest node on the path
    if (x < GetNearestXPos(global.playerPositionX / 16) * 16)
    {
        global.goRight = true;
    }
    else
    {
        global.goLeft = true;
    }
    // Jump if below the nearest y point.
    if (y > GetNearestYPos(global.playerPositionY / 16) * 16)
    {
        global.playerJump = true;
    }
    else
    {
        global.duck = true;
    }
}
```

Figure A.2: Example bot utilising A* pathfinding
A.5: Creating & using levels

As discussed in Section 4.8.3, there are a range of levels for you to test your bot that have been included in the project – to access these levels, simply press F3 at the main menu and select the desired level. To create your own levels press F2 at the same point. A guide to creating your own levels can be found at the Spelunky Wiki (Spelunky Wiki, 2014).

A.6: Using C++ in GameMaker

It is possible to create a DLL for your calculations to be made if you prefer writing in C++; more information on this can be seen in Chapter 4.2.
Appendix B: Spelunkbots Behaviour Video

Included with this dissertation is a video demonstrating the bot behaviours described in Chapter 5.2 entitled “AppendixB.wmv”.

Appendix C: Spelunkbots Project

Included with this project is the Spelunkbots project, this includes the GameMaker project, Spelunkbots DLL and Source, and GameMaker 8.1 to run the project. To access the Spelunkbots project, open Game_Maker.exe in the “GameMaker_8” folder, and click yes when it prompts to enter Advanced Mode. Finally, click file and open, and select the file titled “Spelunkbots.gmk” in the folder Source\Spelunky_1_1.
Appendix D: Spelunkbots Bot Source

Figure D.1: Gold Digger bot source

```java
// GOLD DIGGER
if (!global.hasGoal)
{
    for (int i = 0; i < 42; i++)
    {
        for (int j = 0; j < 36; j++)
        {
            if (GetNumberOfCollectableTypeInNodeXY(global.spGoldBars, i, j) > 0)
            {
                global.targetX = i * 16;
                global.targetY = j * 16;
                global.hasGoal = true;
                global.itemGoal = true;
                CreateStarPathFromXYtoXY(x/16, y/16, global.targetX/16, global.targetY/16);
                return 0;
            } else if (GetNumberOfCollectableTypeInNodeXY(global.spGoldBars, i, j) > 0)
            {
                global.targetX = i * 16;
                global.targetY = j * 16;
                global.hasGoal = true;
                global.itemGoal = true;
                CreateStarPathFromXYtoXY(x/16, y/16, global.targetX/16, global.targetY/16);
                return 0;
            }
        }
    }
    else
    {
        if (global.pathCount > 60)
        {
            global.pathCount = 0;
            CreateStarPathFromXYtoXY(x/16, y/16, global.targetX/16, global.targetY/16);
        }
        global.pathCount += 1;

        // Check if the item we're looking for is still there - or did we collect it?
        if (GetNumberOfCollectableTypeInNodeXY(global.spGoldBars, global.targetX / 16, global.targetY / 16) == 0) //
        {
            global.hasGoal = false;
            global.itemGoal = false;
        }

        // Go towards the x point of the closest node on the path
        if (x < GetNearestXToTarget(x/16, global.playerPositionX / 16, global.playerPositionY / 16) + 16)
        {
            global.goRight = true;
        } else
        {
            global.goLeft = true;
        }

        // Jump if below the nearest y point.
        if (y > GetNearestYToTarget(y/16, global.playerPositionX / 16, global.playerPositionY / 16) + 8)
        {
            global.playerJump = true;
        } else
        {
            global.playerJump = true;
            global.duck = true;
        }
```
// SEAN BEAN
if (!global.hasGoal)
{
    for (i = 0; i < 42; i++)
    {
        for (j = 0; j < 34; j++)
        {
            if (GetNodeState(i,j) == global.spikes)
            {
                global.targetX = i * 16;
                global.targetY = j * 16;
                global.hasGoal = true;
                global.itemGoal = true;
                CreateASparseFromXYtoXY(x/16, y/16, global.targetX/16, global.targetY/16);
                return 0;
            }
        }
    }
    else
    {
        if (global.pathCount > 60)
        {
            global.pathCount = 0;
            global.hasGoal = false;
            global.itemGoal = false;
            global.pathCount += 1;
        }
        // go towards the x point of the closest node on the path
        if (x < (GetNearestXPos(global.playerPositionX / 16, global.playerPositionY / 16) * 16))
        {
            global.goRight = true;
        }
        else
        {
            global.goLeft = true;
        }
        // Jump if below the nearest y point.
        if (y > (GetNearestYPos(global.playerPositionX / 16, global.playerPositionY / 16) / 16 + 8))
        {
            global.playerJump = true;
        }
        else
        {
            global.playerJump = true;
            //global.suck = true;
        }
    }
}
```javascript

// JETPACK JOYRIDE
if (global.hasJetpack) {
    if (platformCharacterIsOnGround()) {
        global.playerJump = true;
        global.spJumpPressedPreviously = true;
    } else if (!global.spJumpPressedPreviously) {
        global.playerJump = true;
    } else {
        global.spJumpPressedPreviously = false;
    }
    global.running = false;
    if (!global.hasGoal) {
        for (i = 0; i < 42; i++) {
            for (j = 0; j < 36; j++) {
                if (getNumberOfCollectableTypesInNodeXY(global.spGoldBar, i, j) > 0) {
                    global.targetX = i * 16;
                    global.targetY = j * 16;
                    global.hasGoal = true;
                    global.itemGoal = true;
                    createStarPathFromXYtoXY(x/16, y/16, global.targetX/16, global.targetY/16);
                    return 0;
                }
            }
        }
    } else {
        if (getNumberOfCollectableTypesInNodeXY(global.spGoldBar, global.targetX / 16, global.targetY / 16) == 0) {
            global.hasGoal = false;
            global.itemGoal = false;
        }
        if (global.pathCount > 60) {
            global.pathCount = 0;
            createStarPathFromXYtoXY(x/16, y/16, global.targetX/16, global.targetY/16);
        }
        global.pathCount += 1;
        // go towards the x point of the closest node on the path
        if (x < (getNearestNodeX(x/16, global.playerPositionX / 16) * 16)) {
            global.gxright = true;
        } else {
            global.gxleft = true;
        }
    }
}
```
else
{
    if (!global.hasGoal)
    {
        for (i = 0; i < 42; i++)
        {
            for (j = 0; j < 54; j++)
            {
                if (GetNumberOfCollectableTypesInNodeXY(global.spJetpack, i, j))
                {
                    with (GetIDOfFirstCollectableOffTypeInNode(global.spJetpack, i, j))
                    {
                        show_message(cost);
                    }
                    global.targetX = i * 16;
                    global.targetY = j * 16;
                    global.hasGoal = true;
                    global.itemGoal = true;
                    return 0;
                }
            }
        }
    }
    else
    {
        if (global.itemGoal)
        {
            if (x < global.targetX)
            {
                global.goRight = true;
            }
            else
            {
                global.duck = true;
                global.attack = true;
            }
        }
        else
        {
            if (x < (GetNearestXPos(global.playerPositionX / 16, global.playerPositionY / 16) * 16))
            {
                global.goRight = true;
            }
            else
            {
                global.goLeft = true;
            }
        }
    }
}
```c
// SHOTGUN TOTAL WARRIOR
if (holdItem.type == "Shotgun")
{
    if (platformCharerals(ON_GROUNDED))
    {
        global.playerJump = true;
        global.spJumpPressedPreviously = true;
    }
    else if (!global.spJumpPressedPreviously)
    {
        global.playerJump = true;
    }
    else
    {
        global.spJumpPressedPreviously = false;
    }

global.running = true;
global.attack = true;

if (global.hasGoal)
{
    for (i = 0; i < 42; i++)
    {
        for (j = 0; j < 34; j++)
        {
            if (GetNumberOfCollectableTypeInNodeXY(global.spGoldBar, i, j) > 0)
            {
                global.targetX = i * 16;
                global.targetY = j * 16;
                global.hasGoal = true;
                global.itemGoal = true;
                CreateAStarPathfromXYtoXyXY(x/16,y/16, global.targetX/16, global.targetY/16);
                return 0;
            }
        }
    }
}
else
{
    if (GetNumberofCollectableTypeInNodeXY(global.spGoldBar, global.targetX / 16, global.targetY / 16) == 0)
    {
        global.hasGoal = false;
        global.itemGoal = false;
    }

    if (global.pathCount > 60)
    {
        global.pathCount = 0;
        CreateAStarPathfromXYtoXYXY(x/16,y/16, global.targetX/16, global.targetY/16);
    }

    global.pathCount ++ = 1;
    // go towards the x point of the closest node on the path
    if (x < (GetNearestXYXY(global.playerPositionX / 16, global.playerPositionY / 16) * 16))
    {
        global.goRight = true;
    }
    else
    {
        global.goLeft = true;
    }
}
```
else
{
    if (!global.hasGoal)
    {
        for (i = 0; i < 42; i++)
        {
            for (j = 0; j < 34; j++)
            {
                if (GetNumberOfCollectableTypeInNodeXY(global.spShotgun, i, j))
                {
                    global.targetX = i * 16;
                    global.targetY = j * 16;
                    global.hasGoal = true;
                    global.itemGoal = true;
                    return 0;
                }
            }
        }
    }
    else
    {
        if (global.itemGoal)
        {
            if (x < global.targetX)
            {
                global.goRight = true;
            }
            else
            {
                global.duck = true;
                global.attack = true;
                global.hasGoal = false;
            }
        }
        else
        {
            if (x < global.targetX)
            {
                global.goRight = true;
            }
            else if (x > global.targetX)
            {
                global.goLeft = true;
            }
        }
    }
}
Figure D.7: Discovery Dan bot source

```c
// DISCOVERY DAN
if (!global.hasGoal)
{
    for (i = 2; i < 40; i++)
    {
        if (GetFogState(i, y/16) == 1)
        {
            global.targetX = i * 16;
            global.targetY = y;
            global.hasGoal = true;
            global.itemGoal = true;
            global.fogSearch = false;
            CreateAStarPathFromXYtoXY(x/16, y/16, global.targetX/16, global.targetY/16);
            return 0;
        }
    }
    for (i = 0; i < 42; i++)
    {
        for (j = 0; j < 34; j++)
        {
            if (GetNodeState(i, j) == global.spEntrance)
            {
                global.targetX = i * 16;
                global.targetY = j * 16;
                global.hasGoal = true;
                global.itemGoal = true;
                global.fogSearch = false;
                CreateAStarPathFromXYtoXY(x/16, y/16, global.targetX/16, global.targetY/16);
                return 0;
            }
        }
    }
    else
    {
        if (global.pathCount > 60)
        {
            global.pathCount = 0;
            global.hasGoal = false;
            global.itemGoal = false;
        }
        global.pathCount += 1;
        // go towards the x point of the closest node on the path
        if (x < (GetNearestXPos(global.playerPositionX / 16, global.playerPositionY / 16) * 16))
        {
            global.goRight = true;
        }
        else
        {
            global.goLeft = true;
        }
        // Jump if below the nearest y point.
        if (y > (GetNearestYPos(global.playerPositionX / 16, global.playerPositionY / 16) * 16) + 8)
        {
            global.playerJump = true;
        }
        else
        {
            global.playerJump = true;
            // global.duck = true;
        }
    }

```
// NOT SO SOLID SNAKE
if ('global.hasGoal')
{
    for (i = 0; i < 42; i++)
    {
        for (j = 0; j < 34; j++)
        {
            if (GetNumberOfEnemyTypesInNodeXY(global.snake, i, j))
            {
                global.targetX = i * 16;
                global.targetY = (j-1) * 16;
                global.hasGoal = true;
                global.itemGoal = true;
                CreateAStarPathFromXYtoXY(x/16, y/16, global.targetX/16, global.targetY/16);
                return 0;
            }
        }
    }
}
else
{
    if (global.pathCount > 60)
    {
        global.pathCount = 0;
        global.hasGoal = false;
        global.itemGoal = false;
    }
    global.pathCount += 1;
    // go towards the X point of the closest node on the path
    if (x < (GetNearestXPos(global.playerPositionX / 16, global.playerPositionY / 16) * 16))
    {
        global.goRight = true;
    }
    else
    {
        global.goLeft = true;
    }
    // Jump if below the nearest Y point.
    if (y < (GetNearestYPos(global.playerPositionX / 16, global.playerPositionY / 16) * 16) + 8)
    {
        global.playerJump = true;
    }
    else
    {
        global.playerJump = true;
        //global.duck = true;
    }
}
Figure D.9: Indie bot source

```c
// Indie
if (holdItem)
{
    global.goRight = true;
    global.lockUp = true;
}
else
{
    if (!global.hasGoal)
    {
        for (i = 0; i < 42; i++)
        {
            for (j = 0; j < 34; j++)
            {
                if (GetNumberOfCollectableTypeInNodeXY(global.spGoldBar, i, j))
                {
                    global.targetX = i * 16;
                    global.targetY = j * 16;
                    global.hasGoal = true;
                    global.itemGoal = false;
                    CreateAStarPathFromXYcoXY(x/16, y/16, global.targetX/16, global.targetY/16);
                    return 0;
                }
            }
        }
        for (i = 0; i < 42; i++)
        {
            for (j = 0; j < 34; j++)
            {
                if (GetNumberOfCollectableTypeInNodeXY(global.spGoldSisor, 1, j))
                {
                    global.targetX = i * 16;
                    global.targetY = j * 16;
                    global.hasGoal = true;
                    global.itemGoal = true;
                    if (x/16 != 1 || y/16 != j)
                    CreateAStarPathFromXYcoXY(x/16, y/16, global.targetX/16, global.targetY/16);
                    return 0;
                }
            }
        }
    }
    else
    {
        if (global.pathCount > 60 && !global.itemGoal)
        {
            global.pathCount = 0;
            global.hasGoal = false;
            global.itemGoal = false;
        }
        global.pathCount += 1;
        if (x < (GetNearestXPos(global.playerPositionX / 16, global.playerPositionX / 16) * 16))
        {
            global.goRight = true;
        }
        else
        {
            global.goLeft = true;
        }

        // Jump if below the nearest y point.
        if (y > (GetNearestYPos(global.playerPositionX / 16, global.playerPositionY / 16) * 16) + 8)
        {
            global.playerJump = true;
        }
        else
        {
            // global.duck = true;
        }
        if (x < global.targetX + 16 && x > global.targetX -16 && y < global.targetY + 16 && y > global.targetY -16)
        {
            if (global.itemGoal)
            global.duck = true;
            global.attack = true;
        }
    }
}
```
// JAR JAR
if (holdItem)
{
    if (holdItem.type == "J")
    {
        if (platformCharacterIsOnGround())
        {
            global.playerJump = true;
            global.spJumpPressedPreviously = true;
        }
        else if (!global.spJumpPressedPreviously)
        {
            global.playerJump = true;
        }
        else
        {
            global.spJumpPressedPreviously = false;
        }
        global.running = true;
        global.attack = true;
    }
    else
    {
        if (!global.hasGoal)
        {
            for (i = 0; i < 42; i++)
            {
                for (j = 0; j < 34; j++)
                {
                    if (GetNumberOfCollectableTypeInRectXY(global.spJar, i, j))
                    {
                        global.targetX = i * 16;
                        global.targetY = j * 16;
                        global.hasGoal = true;
                        global.itemGoal = true;
                        CreateStarPathFromXYtoXY(x/16, y/16, global.targetX/16, global.targetY/16);
                        return 0;
                    }
                }
            }
        }
        else
        {
            if (global.pathCount > 60)
            {
                global.pathCount = 0;
                global.hasGoal = false;
                global.itemGoal = false;
            }
            global.pathCount += 1;
            if (x < global.targetX + 8 && x > global.targetX -8 && y < global.targetY + 8 && y > global.targetY -8)
            {
                global.playerJump = false;
                global.duck = true;
                global.attack = true;
            }
            else
            {
                if (x < (GetNearestXPos(global.playerPositionX / 16, global.playerPositionY / 16) / 16))
                {
                    global.goRight = true;
                }
                else
                {
                    global.goLeft = true;
                }
                // Jump if below the nearest y point.
                if (y > (GetNearestYPos(global.playerPositionX / 16, global.playerPositionY / 16) / 16) + 8)
                {
                    global.playerJump = true;
                }
                else
                {
                    global.playerJump = true;
                    //(global.duck = true;
                }
            }
        }
    }
}
Appendix E: Spelunkbots Bot Source from Developers

Figure E.1 Dan Pearce bot source

```c
if (!global.hasGoal)
{
    for (i = 0; i < 42; i++)
    {
        for (j = 0; j < 54; j++)
        {
            if (GetNumberOfCollectableTypeInNodeXY(global.spGoldBar, i, j))
            {
                global.targetX = i * 16;
                global.targetY = j * 16;
                global.hasGoal = true;
                global.itemGoal = true;
                CreateStarPathFromXYtoXY(x/16, y/16, global.targetX/16, global.targetY/16);
                return 0;
            }
        }
    }
    else
    {
        if (global.pathCount > 60)
        {
            global.pathCount = 0;
            CreateStarPathFromXYtoXY(x/16, y/16, global.targetX/16, global.targetY/16);
        }
        global.pathCount += 1;
        // Check if the item we're looking for is still there - or did we collect it?
        if (GetNumberOfCollectableTypeInNodeXY(global.spGoldBar, global.targetX / 16, global.targetY / 16) == 0)
        {
            global.hasGoal = false;
            global.itemGoal = false;
        }
        // go towards the x point of the closest node on the path
        if (x < GetNearestXYtoXY(global.playerPositionX / 16, global.playerPositionY / 16) * 16)
        {
            global.goRight = true;
        }
        else
        {
            global.goLeft = true;
        }
        // Jump if below the nearest y point.
        if (y > GetNearestYToXY(global.playerPositionX / 16, global.playerPositionY / 16) * 16 + 8)
        {
            global.playerJump = true;
        }
        else
        {
            global.playerJump = true;
            global.duck = true;
        }
    }
}
```
Figure E.2: Andrew Roper bot source

```java
// ANDREW_ROPER
global.waitTimer -= 1;
if (global.waitTimer > 0)
{
    return 0;
}
if (holdItem)
{
    //This is my rock, there are no others like it but this is mine.
    //global.attack = true;
    //global.playerJump = true;

    //global.waitTimer = 10 + random(30);
    global.attack = true;
    global.playerJump = true;
    if (random(10) < 5)
    {
        global.goLeft = true;
        global.goRight = false;
    }
    else
    {
        global.goRight = true;
        global.goLeft = false;
    }
    waitTimer = 10 + random(30);
}
else
{
    if (!global.hasGoal)
    {
        //Let's search for our... rock?
        for (i = 0; i < 12; i++)
        {
            for (j = 0; j < 36; j++)
            {
                if (GetNumberOfCollectableTypesInNodeXY(global.myGoldIdol, i, j))
                {
                    global.targetX = i * 16;
                    global.targetY = j * 16;
                    if (global.targetX == x && global.targetY == y)
                    {
                        global.hasGoal = true;
                        global.itemGoal = true;
                        CreateMachinePathFromXY(x/16, y/16, 1, 1);
                    }
                    return 0;
                }
            }
        }
    }
    else
    {
        if (global.pathCount > 60)
        {
            global.pathCount = 0;
            global.hasGoal = false;
            global.itemGoal = false;
            global.pathCount = 1;
            if (x < global.targetX + 10 && x > global.targetX && y < global.targetY + 16 && y > global.targetY -16)
            {
                global.playerJump = false;
                global.duck = true;
                global.attack = true;
            }
            else
            {
                if (x < (GetSmallestXYPos(global.playerPositionX / 16, global.playerPositionY / 16) * 16))
                {
                    global.goRight = true;
                }
                else
                {
                    global.goLeft = true;
                }
            }
        }
```
// Matthew Smith
if (holdItem)
{
    global.goRight = true;
    if (GetNodeState((x+18)/16,y/16))
    {
        global.attack = true;
    }
}
else {
    if (GetNumberOfCollectableTypeInNodeXY(global.smMattock,x/16,y/16))
    {
        global.duck = true;
        global.attack = true;
    }
    else {
        global.goRight = true;
    }
}
// JAMES HOBSON
if (booldm) {
    if (global.hasGoal == false) {
        for (i = 0; i < 42; i++) {
            for (j = 0; j < 44; j++) {
                if (GetNodeState(i, j) == global.spEntrance) {
                    global.targetX = i * 16;
                    global.targetY = j * 16;
                    global.hasGoal = true;
                    global.itemGoal = true;
                    global.fogSearch = true;
                    CreateAStarPathFromXYtoXY(x/16, y/16, targetX/16, targetY/16);
                    return 0;
                }
            }
        }
    } else {
        if (global.pathCount > 60) {
            global.pathCount = 0;
            global.hasGoal = false;
            global.itemGoal = false;
        }
        global.pathCount += 1;
        if (x > (GetNearestXPos(global.playerPositionX / 16, global.playerPositionY / 16) * 16)) {
            global.goRight = true;
        } else {
            global.goLeft = true;
        }
        if (y > (GetNearestYPos(global.playerPositionX / 16, global.playerPositionY / 16) * 16)) {
            global.playerJump = true;
        }
    } else {
        if (global.hasGoal == false) {
            for (i = 0; i < 42; i++) {
                for (j = 0; j < 44; j++) {
                    if (GetNumberOfCollectibleTypesInNodeXY(global.spGoldIdol, i, j)) {
                        global.targetX = i * 16;
                        global.targetY = j * 16;
                        if (global.targetX == x && global.targetY == y) {
                            global.hasGoal = true;
                            global.itemGoal = true;
                            CreateAStarPathFromXYtoXY(x/16, y/16, i, j);
                            return 0;
                        }
                    }
                }
            }
        } else {
            if (global.pathCount > 60) {
                global.pathCount = 0;
                global.hasGoal = false;
                global.itemGoal = false;
            }
            global.pathCount += 1;
            if (x == global.targetX + 13 && x > global.targetX && y == global.targetY + 16 && y > global.targetY - 16) {
                global.playerJump = false;
                global.back = true;
                global.targetX = true;
            } else {
                if (x < (GetNearestXPos(global.playerPositionX / 16, global.playerPositionY / 16) * 16)) {
                    global.goRight = true;
                } else {
                    global.goLeft = true;
                }
                if (y > (GetNearestYPos(global.playerPositionX / 16, global.playerPositionY / 16) * 16)) {
                    global.playerJump = true;
                }
            }
        }
    }
}

Figure E.4 James Hobson bot source