Abstract—This paper describes the SpelunkBots API, a proposed benchmark for computational and artificial intelligence algorithms. The benchmark, developed by the authors, is an API designed to allow for AI controllers to be written for the game Spelunky: a challenging 2D platforming game that requires players to commit quick reactive behaviours as well as long-term deliberation in order to maximise reward and minimise completion time. In this paper we highlight the features of the Spelunky game and a rationale for its relevance as an AI benchmark. Followed by a description of how the original Spelunky source code has been modified to provide a testing environment for bots. As well as some examples of simple hand-written bots, we discuss the relevance of this benchmark in the context of current AI methods.

I. INTRODUCTION

Looking back on the last 10 years of research in computational and artificial intelligence research in games, the community has often sought new problem domains that challenge the state of the art. Often the most richest and most challenging problem domains are presented to the research community with the intent to encourage, if not challenge, others to improve upon that which has been done already. It is a healthy practice that not only marshalls research activity to address specific problem areas that the community has not yet resolved, but rallies their efforts behind a benchmark that has some recognition and merit.

The most engaging and challenging of these problem domains are often elevated to competition status: allowing researchers from across the world to find varied means of addressing the challenges the domain provides, acting as a benchmark. If we were to consider the problem domains explored solely at the IEEE CIG conference, we see a gradual maturity of these benchmarks. The Simulated Car Racing Competition [1] is a strong example given the rich tool sets provided by the TORCS engine. Similarly, the original Ms. Pac-Man competition changed significantly since its original run at CEC 2007 [2], with its successor in the Ms. Pac-Man Vs Ghosts competition [3] which adopted a new set of tools that streamlined the development process.

Furthermore, research goals continue to shift as the state-of-the-art progresses, with new challenges being pursued across a variety of domains. A notable example being the 2K BotPrize [4]: the game research community’s equivalent of the Turing Test built in Unreal Tournament 2004. In addition, the Mario AI competition [5] introduced more tracks over time to reflect the needs and challenges of the community: not only adopting its own Turing track [6], but introducing a Level Generation track [7] to provide an accessible and relatable benchmark for procedural generation techniques.

Looking at the range of benchmarks that have been developed and sustained, they typically exhibit the following:

- A domain that provides rich and interesting problems for which a simple or uniform solution is not evident.
- A domain that is extensible and customisable, such that more complex problems can be introduced in time.
- A domain is recognisable due to its place in gaming culture.
- An interesting benchmark that challenges current AI methodologies.

In this paper, we introduce an API the authors believe satisfies these traits. Our work, the SpelunkBots API is an AI framework for developing and testing bots in the popular 2009 video game Spelunky shown in Figure 1: a 2D platforming game that challenges players to traverse hazardous environments in order to collect treasures as quickly as possible. We discuss the mechanics of Spelunky and the challenges it presents both to humans and AI methods, followed by a detailed description of how this API has been developed by
extending from the original source code. After exploring a number of basic behaviour bots crafted by the authors, we discuss the relevance of this domain in the context of AI research and offer the work to the community for consideration.

II. SPELUNKY

Spelunky [8] is a 2D platforming game originally released as freeware for Windows PC in 2009 by Derek Yu. Since then, Spelunky has been remade and released on Xbox 360 in 2012, followed by PC, Playstation 3 and Playstation Vita in 2013. The game’s protagonist, the spelunker, is tasked with navigating a series of underground labyrinths and gathering as much gold and treasures as possible whilst avoiding nearby enemies and traps.

A. Gameplay Mechanics

The game is reliant upon platforming mechanics that are similar in nature to Super Mario Bros. [9], in that the player can run and jump to cross gaps in the terrain or reach nearby platforms. Jumping can also be used as a method to attack nearby enemies. In addition, the spelunker has two other key behaviours: hanging from or grabbing local geometry and attacking enemies using his whip. These abilities can be manipulated with various items in the game and there is no limit on how many times these actions can be performed. The jump and whip attacks can be used against a wide range of monsters shown in Figure 2 that vary in size and strength.

Outside of the core abilities, the player has the option of using additional tools. Arguably the most interesting of these tool is the bomb, given it allows the player to destroy terrain as shown in Figure 3. In many instances treasures and even the exit door of the level cannot be accessed directly without the player destroying the local geometry. As a result, bombs are a precious resource given that the player can become trapped or stranded in an area of the level. Players only start with four bombs, more can only be accrued by finding them in crates or purchasing them in local stores.

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 4. Along with being used to navigate players can also use them to attack monsters which can prove useful despite being rather costly.

As previously mentioned, items are acquired either within crates or can be purchased through shops shown in Figure 5. 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 to steal them. The price for items increases with the number of completed levels, meaning paying for items becomes increasingly resource intensive. However, should the player opt to steal items from the shop, the player has become an enemy to all shopkeepers for the duration of the round. In addition, extra enemies will spawned at the exit of each level should the player steal items; this significantly increases the risks faced when stealing, but is often balanced by the items you collect and the additional score accrued for not paying for items.

B. Environment

Environments in Spelunky are procedurally generated which ensures a unique experience every time the game is played. Each level is split into nine random, pre-built segments which ensures that every level generated is enjoyable to play by offering layouts to compliment the games design. This procedural approach is responsible for the aforementioned blocked areas in the terrain that require the use of bombs as part of gameplay. The environments in Spelunky are also littered with a number of traps, each with a unique trigger and behaviour. Traps are introduced in the level generation
to impede the players progress. It is important to learn the behaviour of these traps to ensure safe navigation.

Each level of the original Spelunky also contains damsels: one of whom can be seen next to the spelunker in Figure 1. Damsels the player with extra health when escorted to the levels exit. Damsels can also be used tactically however; if the player is in a tough spot they can be used to trigger traps or thrown as a weapon. These decisions can drastically influence players tactics and play-style.

C. Game Structure

The goal of Spelunky is to reach the end of the game having accrued as much treasure as possible, while minimising time taken and items purchased. This challenge is increased given that Spelunky is considered a ‘roguelike’: a genre of games whose style is reticent of the 1980 game Rogue [10]. The implication of rouge-like gameplay is not only the aforementioned procedural-generation, but the player has only one life: meaning they must start from the beginning of the game again with no items or score transferred from one run to another. An added impetus for players to complete levels as quickly as possible, is that after 150 seconds a ghost will appear on all levels with the exception of the rst and last stage. The ghost moves slowly in the environment and kills on impact making it a great threat. However, if the ghost passes over jewels they are upgraded to incredibly valuable diamonds that increase the score. Awaiting the ghosts arrival such that it allows the player to grab diamonds is a tactic players refer to as ‘ghost running’.

III. The SpelunkBots API

The Spelunkbots API was developed using a combination of GameMaker 8.1 Pro and C++ DLLs (Dynamic Linked Libraries). The original or ‘classic’ version of Spelunky was developed using GameMaker and the original source code is freely available from the games website [11].

Despite the full source being available to the community, the authors recognised that the implementation in GameMaker may not be ideal from the standpoint of creating an AI toolset, as it introduces the hurdle of becoming accustom to the GameMaker engine as well as utilising the GameMaker language (GML). While GML is intended to provide means to develop games without the need to learn a programming language, it would be best to provide an option suited for experienced developers. SpelunkBots API allows users to choose between C++ and GML: ensuring the project is accessible to both experienced programmers and newcomers alike as they are able to choose the toolset which is most suited for them.

The aim of the API is to make the information the bot can receive as close to that of a human player as possible. To ensure that the Spelunkbots API was useable and effective, developers need to be able to gather information regarding all of the elements discussed in the previous section. The API provides sufficient information such that inuence maps and pathnding are feasible, in addition to information that would be more applicable for machine learning approaches. Developers are expected to utilise the PlayerChoice script as the starting point for implementing their bot.

A. Controls

Compared to the control sets found in API’s such as Platformer AI, Ms. Pac-Man and Physical Travelling Salesman Problem, Spelunky is rather complex. 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 PlayerChoice script for the effect of the button press to be applied to the bot. A collection of control variables for the bot are shown in Table I.

The control variables are subsequently reset after being applied each frame and are only applied after the entire player script has been run so any pre-computation can be conducted as much as the developer feels necessary. The bot behaviour is dependant on what items have been employed at this juncture. For example, pressing jump while wearing the jetpack item allows the player to travel further. Querying what items the player has equipped is achieved using the srcClearGlobals script. An example that utilises these variables for the purposes of jetpack control is shown in Figure 6.

B. Pathfinding

As discussed in Section II, one of the biggest problems introduced is destructible terrain. With the world not being persistent, it was very important that the data the bot receives reectives the current state. For the implementation of the static, yet dynamic environment the game map is iterated over, storing
```javascript
if (global.hasJetpack) {
    if (platformCharacterIs(ON_GROUND)) {
        global.playerJump = true;
        global.spJumpPressedPreviously = true;
    }
    global.running = false;
}
```

Fig. 6. An example of utilising global variables to identify the state of the spelunker to dictate subsequent actions.

the value of the type of terrain or liquid within a 2D array which can be queried via the API tools provided. Each of these static objects remove themselves from the data once the Destroy action is called in GameMaker; automatically setting that nodes value to ‘0’ as shown in Figure 7.

Moving objects have a different implementation, given that the objects location is updated every frame to ensure the data available to the bot was accurate. Each of these objects retains an ID within GameMaker. These ID’s remains constant throughout the duration of its lifetime. As a result, we store the object type and ID in a C++ DLL. This data is stored in a list and each GameMaker object makes use of three actions to ensure the data is up to date:

- Create - Adds a new object with ID and location to the C++ list
- Step - Passes the objects ID and location, searches the C++ list for the object with matching ID and updates location
- Destroy - Searches for an object with matching ID and removes from the C++ list

To ensure that the toolset remained both a challenge for more experienced developers yet accessible to novices, there are two options for path finding with the Spelunkbots API. Firstly, there is a built-in A* path nding algorithm within theSpelunkbots DLL. The A* implementation is reliant upon a 2D tile grid model, adopting the information previously discussed. Each tile is 16x16 pixels and is mapped directly to the tile method used to develop Spelunky.

While there are variants of A* that would prove more optimal, these are not provided in the API. However all information that our pathfinding is reliant upon is available. As a result, implementations of techniques such as D* Lite and Jump Point Search are certainly feasible.

C. Fog of War

One key feature of the Spelunkbots API that adds to the original game is the ‘Fog of War’ (FoW) shown in Figure 8. FoW reduces the amount of information the bot is able to receive from the environment. As explained in Section II, players are provided with limited information about each level and it was important to ensure that the information that the bot could receive would give them no advantage over players. Traditionally, FoW has been implemented in top down strategy games as a tool to limit AI opponents knowledge of the world. In Spelunky, as exploration and discovery are a large part of the gameplay it was key that AI bots were not able to have any advantage over the player in the information they can access.

As Spelunky has a wide range of both static and dynamic objects, two key behaviours had to be established and implemented. For static objects, once the bots camera had viewed a location they are able to receive information about this location permanently. This would enable the bot to effectively path nd to previous locations, and to navigate to the exit in a more complex level. To view areas out of reach, players in Spelunky have the ability to look up or down increasing their knowledge of the environment as shown in Table ??.

Any location that had not been explored in the game world will return a ‘1’ when queried, which implies that the node is either solid or fog. This would typically result in a path being impossible to nd if it was outside the viewed area and was implemented using the information provided in the API.

Fig. 8. Diagram to visualise the Fog of War Implementation, demonstrating the difference in the data returned within the players view area and outside of it.
Meanwhile, enemies and dynamic objects act differently to static objects since information about their location and state is not available unless they are within the current view. Methods that can be called by a bot which query this information can be found in Table II.

D. Debug Info

Another additional tool included is means for developers to gain feedback on-screen while testing. This include data about the bots state, its path, or the environment. A screenshot showing some of the debug information at runtime can be found in Figure 9.

a) Path Feedback: Shows possible directions the bot could travel in, a small about of debug info regarding paths the bot could take is displayed on the screen. Inspired by the debug info available on screen in the Mario AI API, the data shows possible locations the player could go if they walked, jumped or fell. The paths displayed react to a few of the items provided though items like the jet-pack do not change the paths displayed as it creates too many situations to display clearly to the developer.

b) Text Feedback: Similar to the Mario AI API, Spelunky offers a range of inputs the effects of which sometimes may not be noticeable on the screen. To ensure that users can clearly see the bots intentions, the state of each button is displayed on the screen. Along with this, Spelunky also features a range of level types. It was important that on observation of the screen it is easy to see which type of level it is. Although this can be obvious to experienced players of the game, it may not be so clear to newcomers that could be trying to implement different behaviours depending on the Level Feel.

IV. TESTING TOOLS

For the purposes of assessing the current state of the SpelunkBots API, as well as providing some basic tools that would enable others to build and test simple bot behaviour, we have focussed on creating a number of basic scripts in addition to a level editor. In addition, the game has been modified to allow shortcuts to be made in standard game mode: allowing users to skip early levels and test against the more challenging environments without the need to play the entire game each time. Furthermore, options have been made available in the SpelunkBots project to change the game speed at runtime. This is particularly useful should developers opt for a machine learning approach that requires repeated sampling of the problem in order to converge on an optimal solution. Developers can increase the game speed to four times faster than usual speed. Conversely, there is also the option to slow the game to half speed, allowing users to observe the behaviour of their bot in greater detail.

A. Level Editor

As mentioned in Section II, each labyrinth is built using a procedural generation algorithm. However, developers may want to assess how well their bots handle specific tasks before particular situations arise in the randomly generated environments. Spelunky has been modied to allow developers to build their own Spelunky maps. A collection of example maps can be seen in Figures 10, 11, 12 and 13.

For example, in Figure 10 we have a simple environment designed to test a simple bots ability to path-find, utilise a cape and dodge traps when falling.
TABLE II. A list of useful methods built within the API that aids the player in exploring the environment.

<table>
<thead>
<tr>
<th>Method to Call</th>
<th>Parameters</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetNodeState(x,y)</td>
<td>Location X &amp; Y Position</td>
<td>State of Terrain</td>
</tr>
<tr>
<td>GetNumberOfCollectableTypeInNodeXY(Type,x,y)</td>
<td>Collectable Type &amp; Location X &amp; Y</td>
<td>Amount of Collectable Type in Node.</td>
</tr>
<tr>
<td>GetNumberOfEnemyTypeInNodeXY(Type,X,Y)</td>
<td>Enemy Type &amp; Location X &amp; Y</td>
<td>Amount of Enemy Type in Node.</td>
</tr>
<tr>
<td>GetPushBlock(x,y)</td>
<td>Location X &amp; Y</td>
<td>Whether Push Block Exists</td>
</tr>
</tbody>
</table>

Fig. 11. Screen capture to demonstrate some of the challenges that Spelunky Challenge test-bed introduces.

Fig. 12. Screen capture to display the challenge set by the Shotgun test-bed.

are available within the game to enable the spelunker to focus on particular elements of the game such as attacking enemies or focussing on resource allocation. For example, the ‘Gold Digger’ bot searches for the first gold bar it discovers by scrolling across the tile data from left to right, it will then move towards that gold bar. Meanwhile more complex bots such as ‘Discovery Dan’ search for the level exit by travelling to the nearest node on the same axis as the player which is still covered in fog should the goal not already be visible. A full listing of the source code of ‘Discovery Dan’ can be found in the Appendices. In addition, a video of different controllers is available for the reader to observe on YouTube¹. These sample bots allow us to highlight the range of unique behaviours that can be achieved by the spelunker, many of which are equally valid at solving particular problems.

V. DISCUSSION

Given the range of game mechanics previously discussed in Section II, in addition to the overall design and focus of the game, we are confident that Spelunky presents an interesting challenge to AI research in games. However, while it introduces a number of interesting challenges, we feel it presents an evolution from recent competition domains such as Ms. Pac-Man and Super Mario Bros.

If we consider the success of Ms. Pac-Man, much of the challenge found in the game is the ability to not only react to local threats, such as nearby ghosts, but also strategically navigate the maze such that pills are consumed as quickly as possible, as well as utilising power pills to full effect to maximise score. Spelunky shares these traits in that the player is required to evade nearby enemies with an array of unique behaviours: some passive in nature, others focussed on attacking the spelunker. In addition, players are expected to strategise their navigation and also the destruction they inflict upon the environment, allowing them to reach more desirable treasures and potentially the exit itself.

Similarly, Super Mario Bros. shares traits with Spelunky given that it is a 2D platformer: exhibiting the same mechanics of running and jumping to navigate the environment. Many

¹http://www.youtube.com/watch?v=N7XVGN8TqJg
enemies can be killed quickly by jumping on them, while others require the use of weapons such as the whip or the shotgun. One major deviation from Mario is the need to backtrack and traverse particular areas of the environment. Whereas Mario relied on the player simply moving forward towards the goal, in Spelunky we are focussed on navigating to particular areas of the map to acquire resources and treasures. In addition, the player may need to actual conduct a search of the environment find the exit as it may be concealed in a particular area of the map.

The need to manage resources effectively, maneuver in the environment, avoid and attack enemies and ultimately devise a plan of action that both maximises score gained while minimising time taken is the real challenge of Spelunky. Given the range of problems that can already be defined with the basic mechanics, the added use of a variety of enemy types and items increases the problem-space significantly.

While the problem space is significant, the ability to utilise the built-in procedural generation in the standard game as well as define our own problem maps allows for this challenge to be scaleable. Researchers can focus on particular elements of the problem space as they iterate and improve upon existing solutions to handle more complex problems over time, with the goal of being able to play the complete Spelunky game over time.

VI. CONCLUSION

The SpelunkBots API is currently available and ready for use by fellow researchers and developers. We are currently investigating methods how to utilise computational intelligence methods for the purposes of solving small problems in the Spelunky game. While we are developing these bots, we are aware that there are areas of the application that need further improvement. These improvements are based not only on our own expectations, but hoping to predict requests we may receive from within the community.

From an implementation perspective, the one area we wish to address is that at present, it would be difficult to implement the likes of Monte-Carlo Tree Search (MCTS) in Spelunky given its reliance on a forward model. This model allows the main deliberation phase to take place as the algorithm runs a series of simulations to assess the validity of a given node in the search space [12]. At present researchers would need to build their own means of simulating the forward model, which is unreasonable. We are currently investigating how to integrate this within the Spelunky source code.

In addition, one key area for future experimentation is the procedural generation algorithms being adopted within the game. We are aware that the games is reliant upon a particular hand-crafted method and we are keen to experiment with extending the API to allow for new procedural generation methods to be deployed much akin to the level generation software provided in the Platformer AI framework.

From a gameplay perspective, the 2012 release of the game entitled Spelunky HD carries two additional features: the ability to play multiplayer in the main portion of the game, as well as ‘battle arenas’, where players could face off against one another in small skirmishes. Given that the tactics required in either case are different from those of the main game, it would be interesting to try and apply these within the confines of SpelunkBots.

We openly encourage fellow researchers to download and experiment with the SpelunkBots tools. It is hoped that this can introduce further challenges for AI researchers.

ACKNOWLEDGMENTS

The authors would like to thank Dan Pearce, Andrew Roper, Matthew Smith and James Hobson for their contributions in testing the SpelunkBots API.

SPELUNKBOTS API

The SpelunkBots tools are available at: http://www.t2thompson.com/spelunkbots

REFERENCES

APPENDIX

‘DISCOVERY DAN’ BOT CODE

This is the source code written for the SpelunkBot ‘Discovery Dan’.

```java
if (!global.hasGoal)
{
   for (i = 2; i < 40; i+=1)
   {
      if (GetFogState(i,y/16) == 1)
      {
         global.targetX = i * 16;
         global.targetY = y;
         global.hasGoal = true;
         global.itemGoal = true;
         global.fogSearch = true;
         CreateAStarPathFromXYtoXY(x/16,y/16,
         global.targetX/16, global.targetY/16);
         return 0;
      }
   }
   for (i = 0; i < 42; i+=1)
   {
      for (j = 0; j < 34; j+=1)
      {
         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;
   }
}
```