Salisbury UNIVERSITY

Honor College at Salisbury University

Honors Thesis

An Honors Thesis Titled

Using Artificial Intelligence and High Performance Computers in Video Games

Submitted in partial fulfillment of the requirements for the Honors Designation to the

Honors College

of

Salisbury University

in the Major Department of

Computer Science

by

Paul Fischer

Date and Place of Oral Presentation: SUSRC April 2018

Signatures of Honors Thesis Committee

Mentor:	[illegible]	Dr. Randall Cone
Reader 1:	[illegible]	Dr. Lance Garmon
Reader 2:	[illegible]	Dr. Timothy Stock
Dean:		Dr. James Buss
	Signature	Print

Abstract

In this paper, we explore using high performance computers (HPC) and artificial intelligence (AI) within video games. Historically, the use of HPCs has been difficult for the average person due to the high startup cost usually associated with them; however, with advancements in software, it is now easier than ever to make one yourself. Both video games and AI are fields that can see significant benefits when combined with higher computational power. Video game developers typically develop their games with using only the best hardware available, however this is not the case for many people that do not have the resources to upgrade their computers. HPCs are easy and inexpensive to make and could be used to help run a video game, or it could help expand the use of AI within video games for all people.

This paper outlines the creation of a simple video game, to test the feasibility that modern video games can effectively utilize AI and HPC. This test simulates an evolving environment that begins to grow vegetation once it becomes hospitable. Computations would be offloaded to a dedicated HPC, which updates the environment using a combination of a cellular automaton and a genetic algorithm, then update the video game using a client server application. The video game will only display the changes made by the HPC and allow the user to interact with it.

1. Introduction

In recent years, Artificial Intelligence (AI) within video games has seen only small improvements. A review of presented topics at the AI Summit during the Games Developer Conference, an annual conference to discuss advancements in video games, showed that the bulk of work only made small improvements to outdated techniques. In recent games released within the past couple years, like Middle Earth: Shadow of Mordor and Tomb Raider, an AI technique called Goal Oriented Action Planning (GOAP) is used [4]. This technique is a modern approach at an older idea called behavior trees. Some of these techniques date back to late 2004 with the release of *Halo 2*, a video game known as the best of its time because of the revolutionary AI techniques applied by developers. *Halo 2* used a technique referred to as behavior trees [1]. These trees consist of different actions the AI can make characters within the game perform. Figure 1 lays out a diagram of an example decision tree from *Halo 2*. The actions the enemies can take are shown in circles and the arrows lead to other actions it can take from its current action. An enemy cannot change from "Searching" to "Self-Preservation" without entering the "Idle" or "Combat" states first.

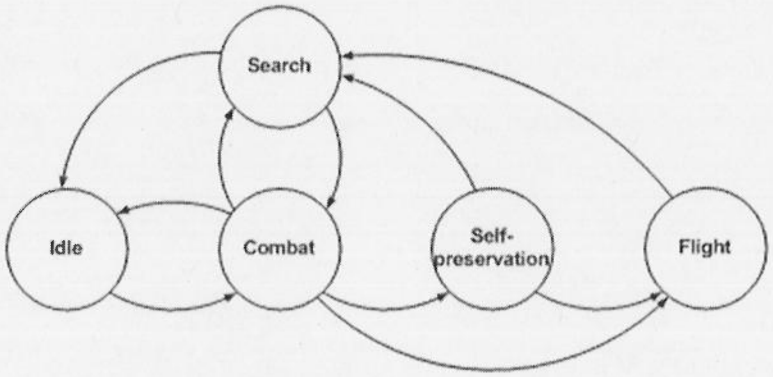

Fig. 1. Example of a behavior tree for enemies in *Halo 2*. [1]

As stated before, GOAP is a modern approach to behavior trees. With GOAP, a character will have the same states as seen in a behavioral tree, but the predefined ways to enter new states are not present. Instead, a character will see what actions it can perform based on what is currently happening within the game. Using the same example from *Halo 2*, if an enemy is "Searching" for the player but it has just run out of bullets while in the "Combat" state, it may decide to enter the "Flight" state directly versus jumping through other states first.

The AI in many modern games uses some form of both GOAP and behavior trees, leading to the feature being built directly into the Unity 3D game engine. The designers behind Tomb Raider and Middle Earth: Shadow of Mordor were able to modify the algorithm to allow more flexibility in what can be done in the games. This can be directly seen in Middle Earth: Shadow of Mordor with groups of enemies. They implemented a system that allows groups of enemies to make a collective group decision about a situation rather than every enemy acting individually. This gives the impression of enemies being smarter and working as a group, but it is an approach to reduce the footprint of AI within the game. It takes longer to find what each enemy might do in every frame during the game versus giving an entire group a collective task to take care of [4].

To make games more advanced, there must be new ways to implement AI. When viewing the work of K. Kunanusont et al. [5] and A. B. Moghadam and M. K. Rafsanjani [6], there has been a recent push for using evolutionary algorithms in game development. An evolutionary algorithm (EA) is an AI technique in which a program will simulate trials for a problem, where every trial the AI gets better at solving it. An EA uses a fitness function, which is a mathematical function that creates a score to rate how well the trial went. Depending on how the EA is created, a lower or higher score could indicate a more successful trial. This score can assess different variables like the length of time a particular trial took until completion or how complicated the solution had been. The goal of EAs are to find the best possible outcome in relation to the fitness function without having to explicitly tell the computer how to solve the problem. This greatly mimics the Theory of Evolution set forth by Charles Darwin, which is where it gets its name.

K. Kunanusont et al. made a game called Space Battle Evolved, which is based on a simple two player game called Space Battle where players battle head-to-head using space ships. They taught an AI agent to play the game through genetic algorithms, but it could not run as fast as modern games. It was limited to 40 milliseconds to plan an action and 1 millisecond to execute the plan [6], which corresponds to the game running at nearly 25 frames per second (fps). Since all modern games run at 60 fps, the AI in Space Battle Evolved would need to be much faster if it were to be included in a modern game. A. B. Moghadam and M. K. Rafsanjani made a sample side scroller game that used genetic algorithms to create an infinite level. This example side scroller would need to generate content while the player is playing the game [5]. The decision process they made for content is not complex, so a genetic algorithm on current computing devices would not have an issue deciding what content should be made. If this were expanded to have many decisions about multiple aspects of the game, it may start to slow the game down entirely to the point it would not be playable.

2. Background

This paper will introduce a new way to implement two existing AI concepts for video games, but it is important to understand these concepts and how they can be implemented first. The first is a genetic algorithm, which is a specific type of evolutionary algorithm that uses the basic biological principles of producing an offspring from two parents. With this algorithm, two parents will create an offspring that has shared traits from the parents, with the chance to have some mutation. The point of this algorithm is for the parents to create offspring that are better than themselves at completing a certain task in relation to the fitness function. For instance, searching genetic algorithms on the internet will return many guides and tutorials about how to teach a program how to play simple games. It does this by using a genetic algorithm to press buttons at different times to make an attempt to beat the game. Each attempt is rated against the fitness function to see how well it did. Every generation made in this algorithm is a step closer to mastering the controls. The second concept is cellular automata, which refers to a system where all units, called cells, on a specified grid or within a system use a predefined algorithm to create a simulation or outcome. Every cell within a grid will run the same algorithm and each cell will have a different result based upon its neighbors.

Some other useful concepts to understand are parallel versus serial processes and high performance computing. A parallel process refers to a process that can run at the same time across one or more computer processors to solve a problem. This differs from a serial process which only runs on a single computer processor. The boxes on the left-hand side are single threaded processes, while the boxes on the right-hand side are multithreaded processes. All processes can be run by a program in serial, but some processes will finish faster if they are made to run in parallel. The process of making a process parallel usually comes in the form of multithreading a program. A single thread will run a single process on a computer processor, where multiple threads will run multiple processes on the same computer processor. A single thread program is considered a serial program. High Performance Computing uses these parallel programs and will distribute the threads over multiple computers to speed up the time it takes to run a program.

Two other concepts worth mentioning are client-server applications and noise functions. Client-server applications are a well-known networking model where a computer, called a client,

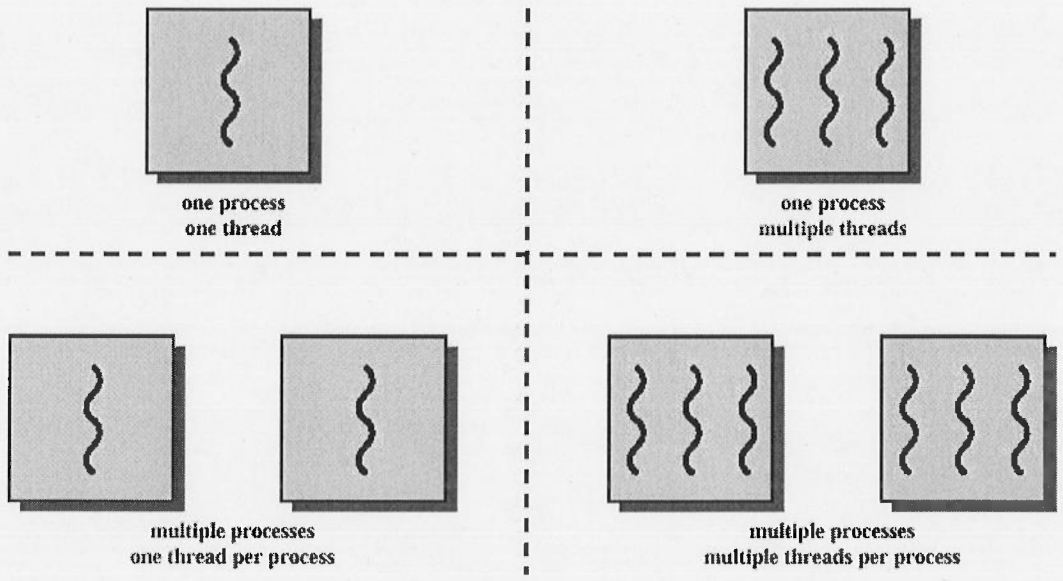

Fig. 2. Single thread versus multiple threads in a process. The more threads it has, the faster it will compute. [2]

will contact another computer on a network, called a server, to send and receive information. The server is run independently from the client and can usually service connections between more than one computer. The last concept is noise functions, which are random number generators, but the next number produced is influenced by the last number produced. This creates a natural variation of how numbers are created allowing for randomized curves or patterns to be made.

All the concepts discussed are normally not implemented into video games, with the exception of the client-server application. Using these concepts, a video game can be created that can support the modern AI techniques discussed. This will be done by using an HPC back-end server, meaning the AI algorithms will be run on a separate computer using a parallel process. The video game that will be created will have an environment that will evolve to become hospitable to support vegetation growth. This will be done using a voxel grid. Voxels are 3D units of space, which are generally represented as cubes. This voxel grid will use both a cellular automaton and a genetic algorithm to form biomes. Biomes within the game will be a collection of ground units that are very similar to one another. Ground units in one biome will be very similar to one another, but have large differences to other ground units in other biomes.

This paper presents the construction of a simple video game that will test the effectiveness of using both HPC and modern AI techniques. The video game uses both a cellular automaton and a genetic algorithm on a back-end HPC system. The back-end system will communicate with the video game front-end system through a client-server program. We will also discuss further ways to improve upon the methods used within the paper.

3. Implementation

This paper lays out how to create a video game that will use modern AI techniques, like genetic algorithms and cellular automata, in a video game while allowing an HPC back-end server to handle the AI. The back-end server and HPC will run solely in C++, and the video game will be created in Unity 3D (Unity) using C# for the front-end. Since the back-end and front-end are running in separate programming languages, and because it cannot be guaranteed all users of the video game will have their main computer also function as an HPC, it is best to use a client-server for the communication between the two.

As stated before, the video game will be made up of voxels that create an environment that will support multiple biomes. Simply put, voxels will be made that can be grouped together, which will be called biomes, and all voxels within the biome will be more similar to one another than the voxels outside the biome. All of these biomes will be a part of one single environment. For the purpose of this video game, these voxels will be called ground units, and each ground unit will have three attributes: moisture content, fertility, and average temperature. These three attributes will be represented by a decimal number between 0 and 1. For instance, a ground could have a moisture content of 0.51, fertility of 0.798, and an average temperature of 0.3387. Each ground will also know if it is in a biome, and if so, what biome it is a part of. Each biome has preferred attributes that it will accept as a part of the biome but will also accept within a random variation of the attributes specified by the game. This means that if the forest biome prefers 0.5 moisture content, 0.75 fertility, and 0.5 average temperature and there is a random variation of 0.1, then any ground within ±0.1 of each attribute will become a part of the biome.

4. Genetic Algorithm and Cellular Automaton

The Artificial Intelligence in this game will take two forms: the genetic algorithm and the cellular automaton. It is important to remember these two forms of AI within the game are closely related: the genetic algorithm will specify how the ground should evolve, and the cellular automaton updates the environment with the changes made by the genetic algorithm. This section will lay out how these two are implemented within the game.

The genetic algorithm will have two main parts: crossover and mutation. The crossover portion of the algorithm will use two ground units, which are the unit that will be replaced and a neighboring unit in the same biome, and a random variation to create an offspring. Biomes in this algorithm will act as the fitness function. If a ground unit is not in a biome, it can breed with any of its neighbors. For each of the parent's attributes x and of the neighbor's attributes y, a random number r between 0 and 1 will be used to combine parts of both parents into the offspring such that:

$$x = (x * r) + (y * (1 - r)).$$

Mutation works the same way as crossover, but it will allow a ground unit to breed with any neighbor.

The cellular automaton is the portion of the AI that runs the game. An overview of the cellular automaton algorithm is given in Algorithm 1. This is implemented within the game to allow ground units to interact with its neighbors. For each ground unit, the automaton will check which neighbors are in the same biome as it, then it will run the genetic algorithm. It will first run the crossover portion of the algorithm giving it a neighboring ground's attributes. The algorithm will compute the offspring and give that back to the automaton. Then, it will check to see if there is a mutation present by generating a random number. If this number is less than 0.1%, it will run the mutation portion of the algorithm giving it a different neighbor that is not in the same biome. If the ground unit is not in a biome, it will not check to see if there is a mutation. Once this is finished, for all ground units, it will replace all ground with their offspring in the same location.

Algorithm 1. Cellular Automaton

```
1:  input: node's groundArray
2:  tempGroundArray ← new ground array
3:  for each ground g in groundArray
4:      if g is in biome
5:          neighbor ← GET_RANDOM_NEIGHBOR_IN_BIOME(g)
6:          tempGroundArray ← g.breed(neighbor)
7:      else
8:          neighbor ← GET_RANDOM_NEIGHBOR(g)
9:          tempGroundArray ← g.breed(neighbor)
10:     end if
11:     if GET_RANDOM_NUMBER_BETWEEN_0_AND_1() < 0.001
12:         neighbor ← GET_RANDOM_NEIGHBOR(g)
13:         tempGroundArray ← g.breed(neighbor)
14:     end if
15: end for each
16: groundArray ← tempGroundArray
17: return groundArray
```

5. Front-End

The front-end will update to show what is happening with the AI from the back-end, which is described in Section 6. The game only allows users to view the changes made by the genetic algorithm as represented by the cellular automaton. The game displays a 20x20 grid, known as the environment, which makes up 400 total ground units. Figure 3 shows a sample view of the environment. Each ground unit is represented by a cube. Since each ground unit is comprised of three attributes, the attributes will form a specific color for the cube to have a visual representation of the attributes it has. Unity uses a slider for red, green, and blue to represent its colors. For each ground unit in the environment, average temperature corresponds to red, fertility corresponds to green, and moisture corresponds to blue. These three are combined to make each cube's specific color. To make the terrain more lifelike, a gradient noise function is used to allow smooth transitions of random numbers through the environment. This gives the effect of gradual changes within the environment. Each ground unit also has a specific height, which can be used to form different structures like mountains, hills, valleys, or even flat plains.

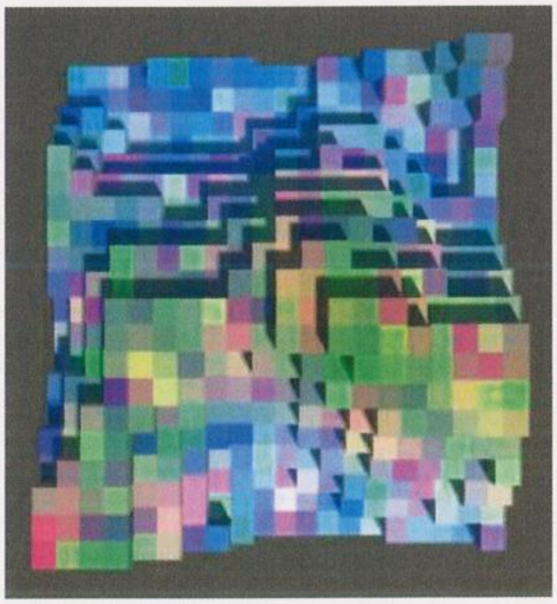

Fig. 3. A top-down view of the environment.

6. Back-End

The back-end system is made up of the client-server program and the two AI functions used in the game. This will be running on an HPC using a library known as MPI. The MPI library, or the Message Passing Interface, is a useful way to allow for communication between nodes in an HPC. For the back-end system, the 20x20 grid in the game will be broken down into smaller parts. Each node will have designated consecutive rows it will compute for. Depending on the amount of nodes present will depend on how the grid is broken down. If there are any extra rows, some nodes will receive an extra row. For example, if 8 compute nodes are present, then each node will receive 2 rows from the grid and the first through fourth nodes will receive an extra consecutive row. So, the first node receives the bottommost three rows and the second node receives the next three rows. Every node will have direct access to update the rows it has through the genetic algorithm, and a reference to the next consecutive row above and below for its neighbor calculations, if applicable.

Each generation will follow this basic order of events to update. For each ground unit, 1) Check if it is in a biome, 2) find its neighbors in the same biome, 3) run its crossover with another neighbor, then 4) apply mutation if needed. Each node will then update the consecutive node above and below itself with the new generation data for its neighboring rows. The master node will receive a copy of all nodes updated rows, then convert this updated grid into a string to send back to the server. The total compute time for this is very low due to the amount of parallelization it has to offer.

7. Results

Using both HPC and modern AI techniques within the test video game shows much about the current state of video games. Figure 4 demonstrates a progression of the game. Every generation produced gets closer to all ground units being within a biome. The game runs at 60 frames per second, but the environment does not update as fast as it was expected to. It seems to update once every few seconds versus instantly as expected. There are multiple reasons this may be happening. The largest reason is due to the game needing to parse through the string sent by

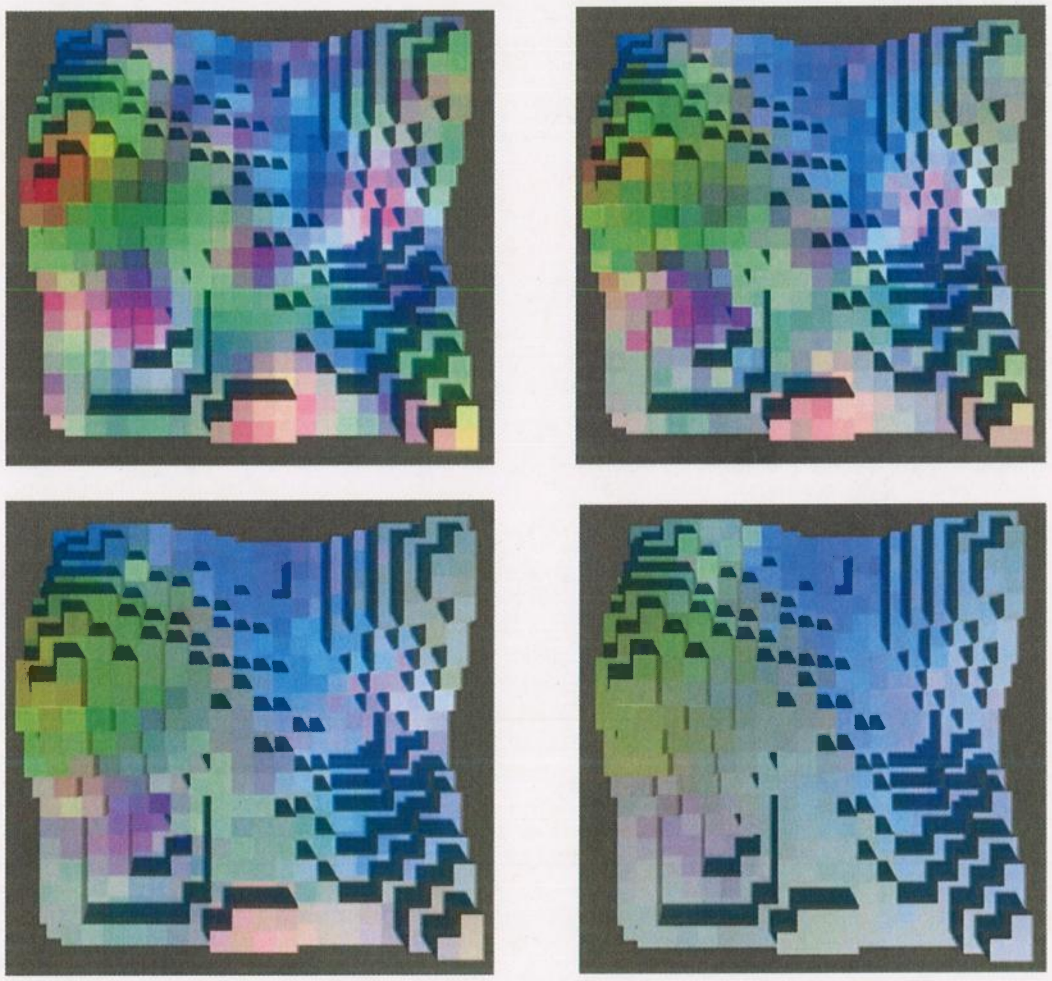

Fig. 4. A progression of the game over time. Biomes start to develop on top of the mountain and within the valley

the server for its information. The easiest method for sending an array over the network in standard C# is to first convert it to a string. Since the data is translated into a string before being sent over the network, the video game must parse the data it receives into data it can understand. This overhead from the video game makes any performance gain from using the HPC inconsequential. Since Unity cannot handle multiple threads within the game, it cannot parse the data sent by the server efficiently nor in a timely manner. If Unity allowed the use of more threads, it would help to alleviate this issue.

The environment itself was also the cause of some bad performance. The game must render 400 individual cubes, with each having six faces, all being simulated in 3D. If the ground rendering was handled in a better way, the game would have better performance. This can come in the form of not viewing all 400 cubes at the same time or using some other structure than just cubes.

8. Conclusion

In this paper, we proposed a new way to incorporate modern artificial intelligence into video games by using high performance computers. Modern AI techniques require many resources from computers in order to work effectively. Because of this, video games have a hard time using them. Some video games made, like the example platformer from A. B. Moghadam and M. K. Rafsanjani [6] or Space Battle Evolved from K. Kunanusont et al. [5], have used AI to either teach AI agents how to play the game or for level design, but none have used it to run a video game completely.

Using high performance computers to run artificial intelligence for video games does have many issues in the current state of video games, but it can be improved. The example game we made shows it is currently possible to make a game using high performance computers as a backend to a video game. With future advancements to video games, and the technology behind them, this could become a reality. With these advancements, video games could finally have the first actual improvements in artificial intelligence in over a decade.

Reference List

[1] D. Isla. "GDC 2005 Proceeding: Handling Complexity in the Halo 2 AI." Gamasutra, 2005. [Online]. Available: https://www.gamasutra.com/view/feature/130663/gdc_2005_proceeding_handling_.php.

[2] D. Marshall. "Threads: Basic Theory and Libraries." Cardiff School of Computer Science & Informatics, 1999. [Online]. Available: https://users.cs.cf.ac.uk/Dave.Marshall/C/node29.html.

[3] E. Long, "Enhanced NPC Behaviour using Goal Oriented Action Planning," Edmund Long, 2007. [Online]. Available: http://www.edmundlong.com/downloads/Masters_EnhancedBehav iourGOAP_EddieLong.pdf

[4] "Goal-Oriented Action Planning: Ten Years of AI Programming". YouTube video, Duration 1:01:05. Posted October 2017. https://www.youtube.com/watch?v=gm7K68663rA&t=2523s

[5] K. Kunanusont, R. D. Gaina, J. Liu, D. Perez-Liebana and S. M. Lucas, "The N-Tuple Bandit Evolutionary Algorithm for Automatic Game Improvement," ArXiv, 2017. [Online]. Available: https://arxiv.org/pdf/1705.01080.pdf

[6] A. B. Moghadam and M. K. Rafsanjani, "A Genetic Approach in Procedural Content Generation for Platformer Games Level Creation," Research Gate, 2017. [Online]. Available: https://www.researchgate.net/profile/Arman_Balali_Moghadam/publication/316165410_A_Genetic_Approach_in_Procedural_Content_Generation_for_Platformer_Games_Level

_Creation/links/59d34debaca2721f436ca289/A-Genetic-Approach-in-Procedural-Content-Generation-for-Platformer-Games-Level-Creation.pdf