eludamos
Journal for Computer Game Culture
Vol. 10, No. 1 (2019)

http://www.eludamos.org

Study of Artificial Intelligent Algorithms Applied in Procedural Content Generation in Video Games

Martín González-Hermida, Enrique Costa-Montenegro, Beatriz Legerén-Lago, Antonio Pena-Giménez

Eludamos. Journal for Computer Game Culture. 2019; 10 (1), pp. 39–54

Study of Artificial Intelligent Algorithms Applied in Procedural Content Generation in Video Games

MARTÍN GONZÁLEZ-HERMIDA, ENRIQUE COSTA-MONTENEGRO, BEATRIZ LEGERÉN-LAGO, AND ANTONIO PENA-GIMÉNEZ

I. Introduction

During the last decade, the video game industry has undergone unprecedented changes. These changes have mainly affected the design process. Originally, video games were perceived only as a product, but during the last years this vision has been distorted, approaching more and more the concept of video games as a service. Due to this change of perspective, video game developers are forced to generate a lot of content for the one single game, which implies a considerable increase in production costs. This content must be generated by a designer and, in most cases, its production also implies that more complex programming tasks are necessary for its implementation.

So, as the production of content for video games is currently a problem, Procedural Content Generation (PCG) is presented as a solution to it, automating the generation process. Using these techniques, content that guarantee hours of playtime can be algorithmically generated without further work by designers. In other words, the fundamental idea of the procedural content generation is that the video game, or part of it, is generated computationally through a well-defined procedure, instead of being developed manually.

The use of these techniques has experienced an important growth in the last decade and according to (GamingBolt 2018) is likely to do so at a higher rate. However, its origin is not so recent. The first important title using procedural generation dates back to 1980 with the Rogue (A.I. Design 1980) video game. In this video game for terminal, the player was represented by the ASCII character "@" and had to go through a series of dungeons that were generated in a pseudo-random way in each game. The next great milestone within PCG would be Minecraft (Mojang 2011), being the most popular video game to date that makes use of these techniques, with a total of 122 million copies sold (Polygon 2017). Since then, the amount of video games using procedural generation has increased significantly, and has been a very important differentiating factor for video games produced by low-medium budget companies, such as Minecraft (Mojang 2011). More recent titles are No Man's Sky (Hello Games 2016) and Starbound (Chucklefish 2016), which implement procedural generation of galaxies, with fully explorable planets, in 3D and 2D environments respectively.

The procedural content generation is not a trivial problem, since the techniques used can be applied to many of the elements that make up a video game (from the lowest level of how to create sounds or textures, to a more abstract level as the rules that govern the game). The absence of a general methodology (Hendrikx and Meijer

Eludamos Vol. 10, No. 1 (2019)
pp. 39–54

2013) that can be applied to each of these components and the scarcity of practical information on the subject are then challenging, as the generator design and constraints will affect considerably the final result.

Our work will consist on the development of four demos[1] where we use different procedural generation techniques. Each demo will use one or more specific artificial intelligent techniques to generate a specific type of content. We have used Unity, a well-known IDE (Integrated Development System) specialized for the development of video games, to create these demos. This work can also serve as a starting point to know how to deal with a problem that can take advantage of the use of procedural generation techniques.

In this paper, we will study different artificial intelligent algorithms, mostly rule based, used for procedural content generation. These algorithms will be applied to: (i) labyrinth generation, (ii) dungeon generation, (iii) 2D terrain generation and (iv) 3D terrain generation. We will evaluate the performance of all these algorithms and apply them in four video games demos created in Unity. This performance tests are only an approximation to the complexity of the algorithms, since their results will strongly depend on the hardware used, the operating system.

II. Labyrinth Generation

Labyrinths are resources that currently are not used very often in the design of video games. In the past, and more specifically, in the early days of video games, the use of labyrinths as the main element of entertainment had a more considerable role. In recent years, its use is normally reduced to specific levels within a video game where the player is forced to complete a maze in order to progress of his game. Some examples can be any title of the saga The Legend of Zelda (Nintendo 1986) or Diablo (Blizzard North 1996).

It may be interesting to use a labyrinth generator for several reasons. One of them may be the fact that the manual design of labyrinths is not an easy task and, in many cases, its implementation could be even more tedious than the use of an algorithm for its construction. Another compelling reason is predictability from the player's point of view. Once the labyrinth has been solved, it is trivial to solve when it is faced again, causing these moments to become uninteresting. These two problems could be solved using procedural generation to create the labyrinth.

We have developed MazeGen (Figure 1), a playable demo whose objective is the resolution of a labyrinth that has been generated using an algorithm. It allows to user to select the next options: 1) rows: the height of the labyrinth; 2) columns: the width; 3) Algorithm: algorithm used in the generation, can choose between Depth First Search, Prim’s Algorithm or Recursive Division; 4) Seed: to change the seed of the random number generator; 5) View: allowing the player to use a first person view or an aerial view; and 6) Fog Mode: to visualize only the areas of the labyrinth that have already been visited by the player.

Algorithms generating labyrinths is a subject extensively treated, and not necessarily by its application in video games. We will briefly explain three algorithms (Kozlova, A.

et al. 2015) widely used to generate labyrinths. The labyrinths generated by these algorithms are modeled as two-dimensional matrixes whose elements represent the cells of the labyrinth, each with its four walls. The labyrinths generated always meet the definition of a perfect labyrinth:

- The labyrinth must have a single entrance and a single exit (these are usually located at the corners).
- All the cells that compose it must be reachable.
- The labyrinth cannot contain loops (there is only one way to get from one box to another).

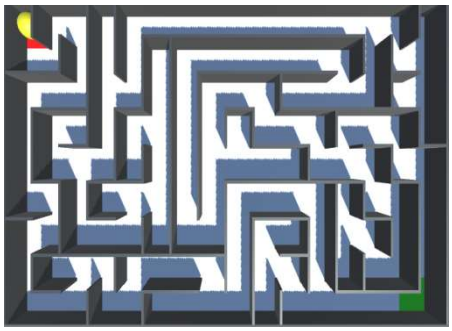
The three used algorithms are:

- Depth First Search (DFS): an agent (Nwana 1996) is used that randomly wanders through the matrix, breaking walls between the cells through which it advances. First, a starting point is chosen for the agent and it can never pass through the same cell twice as it advances breaking walls. In case of not being able to advance more, it goes back following its steps until finding a neighboring cell by which to be able to advance. The process ends when it has passed through all the cells. This algorithm generates labyrinths with very long corridors, with very small trees of choice and with short paths with no exit. Therefore it reduces the chances of getting lost and it is usually necessary to travel long distances to reach the end.
- Prim's Algorithm: it begins with the random selection of a cell in the matrix. Subsequently, one of its neighbor cells is chosen randomly, the wall that exists between them is broken and added to a list of cells already visited. In each iteration, one of the cells in the list is chosen randomly, one of its neighbors is chosen randomly, if it is not in the list of cells already visited, the wall is broken between them and it is added to the list. The process is repeated until all the cells are in the list of cells already visited. This algorithm creates labyrinths with very short corridors, multitude of crosses and multiple roads with no exit. It is very easy to get lost during the course of it, but the minimum distance necessary to travel it is usually very small.
- Recursive Division: it starts with a cell matrix without walls. The generation process consists of placing a wall (horizontal or vertical and not necessarily in the center) that divides the matrix into two parts. Then, decides to open a hole in a cell along the wall, to allow access between the two sections. Of the two parts generated from this division, it keeps the largest one in a stack and chooses the smallest one to continue working. The previous process is repeated until the part with which we are working has one of its dimensions with a length equal to one cell. Then, the algorithm continues with a part extracted from the stack. The entire process ends when the stack is empty. The best adjective to define this algorithm is chaotic. It generates very unpredictable mazes, since these can be excessively simple or exaggeratedly entangled and complex.

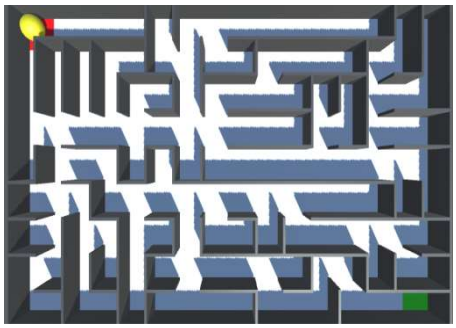
To test the performance of these algorithms, we generated labyrinths of dimensions 5*5, 16*16, and 50*50 in an Android smartphone device (Xiaomi Redmi 3). We can see the results in Table I. In the case of 5*5 small labyrinths, the results are identical for practical purposes. With 16*16, Prim's algorithm gives better results and would be the safest option to avoid maximum generation times. For the 50*50 case, Prim is still the algorithm with best results, DFS clearly gets the worst result, most likely because it is necessary to go back over the previous steps too often, and during this time it does not dig walls. Recursive Division remain practically identical in average performance as Prim but with worst maximum time. Anyway, all of them give results in an acceptable range for gaming in a smartphone.

	5*5			16*16			50*50		
	Min.	**Average**	**Max.**	**Min.**	**Average**	**Max.**	**Min.**	**Average**	**Max.**
Depth First Search	55.37	61.95	68.05	198.9	212.4	229.8	1497	1581	1652
Prim's Algorithm	56.79	62.85	68.66	199.2	208.7	219.9	1175	1206	1247
Recursive Division	58.23	64.21	67.81	198.8	209.2	231.3	1162	1213	1280

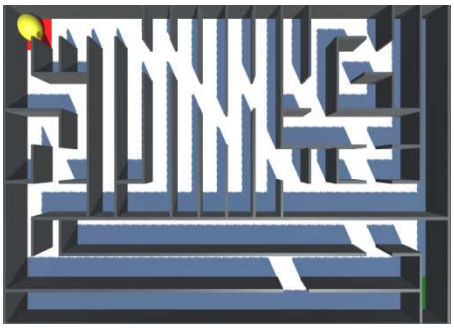
Table I: MazeGen algorithms performance measures in miliseconds



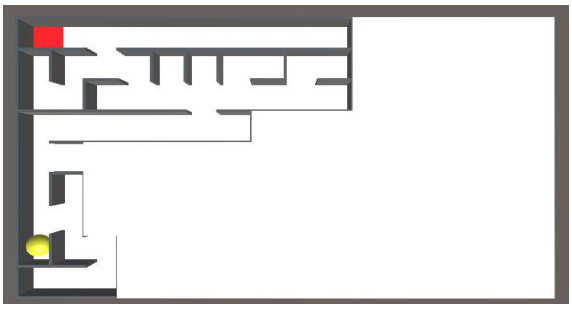
(a) Depth First Search



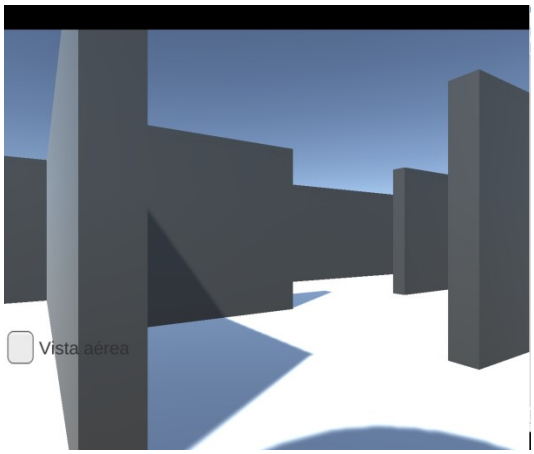
(b) Prim's Algorithm



(c) Recursive Division



(d) Fog mode



(e) First person view

Figure 1: Snapshots of the MazeGen demo

III. Dungeon Generation

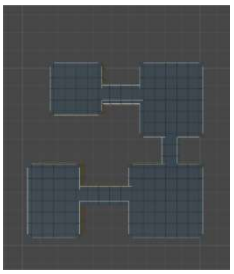
We define a dungeon as a set of rooms linked together through several corridors arranged, generally, in a labyrinthine way. Dungeons are an habitual resource in the development of video games, some famous examples can be Diablo II (Blizzard North 2000) or Torchlight II (Runic Games 2012) which have used dungeons recurrently as part of their level design. In fact, the games based on crossing dungeons (dungeon-crawler) were one of the pioneer genres to make use of procedural content generation techniques.

We have developed DungeonGen (Figure 2), a playable demo of a first-person shooter that uses dungeons generated in a procedural way, but also has a number of additional details, including the generation of locks (closed doors that need a key to open them located in another room in the dungeon) and some simple methods for the placement of enemies and treasures. Every time the player starts a new game, a completely different dungeon is generated, increasing its replay value.

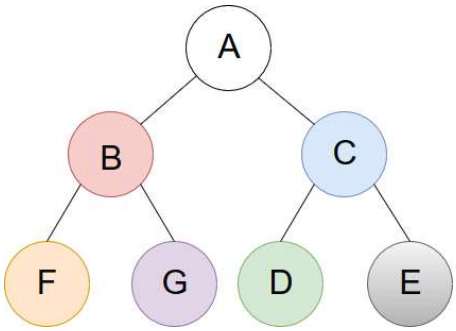
Figure 2: Snapshot of the DungeonGen demo

The algorithm used in this demo to generate the structure of the dungeon is called the space binary partition (Shaker et al. 2016, pp.31-55). Consists in the recursive

division of an initial space into sections, which can be used to create dungeon rooms and obtain a representative graph of its structure. Using this algorithm, we obtain as output a dungeon formed only by halls and corridors (Figure 3a) and a graph that represents the structure of it (Figure 3b). But this content is of no use by itself, it is necessary to add more details or improvements so that it can be used as content ready to play, such as treasures, enemies or a system of closed doors that require a key to open them (Figure 3d). So we added a method of generating dynamic locks, proportional to the size of the dungeon, which makes the content generated more interesting, by forcing the player to visit more rooms than necessary in search of a needed key. Other algorithms were used to place enemies and treasures in the rooms of the dungeon.

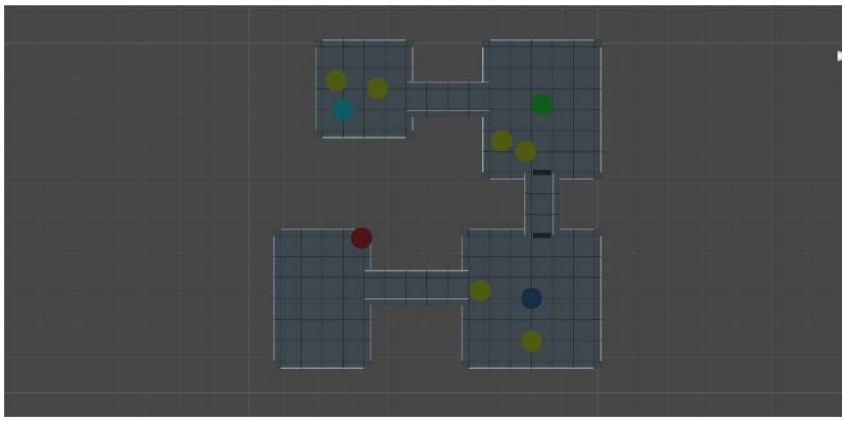


(a) Rooms with corridors



(b) Dungeon final graph

(c) Representation



(d) Final dungeon

Figure 3: Dungeon generation progress

The Space Binary Partition works as follows. Initially, that space is divided into two parts by a horizontal or vertical line at a random point of it. Of the fragments generated, the largest is stored in a stack, while the small one is chosen to continue working. The chosen fragment is further divided by applying the previous process until the size of one of the fragments generated is less than a threshold. In the latter case, a fragment is extracted from the stack to continue working. The process ends when the stack is empty, because all the fragments that remain are smaller than the threshold. The final fragments will be where the rooms of the dungeon will be placed.

Its dimensions are chosen by randomizing the position of the lower left corner and the upper right corner within the limits of a fragment. Through this method, we can draw a graph of the dungeon based on the fragments that have been generated. Rooms will correspond to the lowest nodes in the graph. This abstract model provides a series of advantages, among them is the fact that it provides a superior point of view about the structure of the dungeon and also the possibility of creating corridors between brother nodes, which reduces the possible intersections of these with other corridors or rooms, shortening its length.

The generation of dynamic locks is achieved through a seven steps algorithm developed from the dungeon graph, that ensures that all the rooms of the dungeon are accessible and therefore, to prevent a key from a door being placed behind it, preventing its opening. This method guarantees that the dungeon generated will always have a solution, and that the blockages that are generated will try to maximize the number of rooms that are on each side of them. It also works independently with respect to the position of the end of the dungeon, since it ensures that all rooms can be reached.

The number of enemies in a room is calculated randomly given a maximum number. Rooms with objects (such as keys or objects for combat) may contain a greater number of enemies than usual. The treasures are placed in rooms that are only accessible through a single corridor to force the exploration of the player.

To test the performance of our algorithms, we obtained the average measures for some generated dungeons. The number of dungeons increase as the player passes different levels, also the number of enemies and the difficulty of the demo. In Table II we can see the generation times of the different algorithms in a laptop (Intel Core i7-7700HQ CPU, 16 GB RAM, Geforce GTX 1050 GPU with Windows 10 OS). As we can see the times are quite small and will not affect the experience of the game.

Level vs Algorithms generation time (ms)	**Dungeon level 1: 4 - 6 rooms**	**Dungeon level 5: 8 – 12 rooms**	**Dungeon level 9: 12 - 16 rooms**
Rooms	9.153	15.34	26.57
Corridors	76.88	299.1	587.2
Blocks	8.251	17.56	28.16
Enemies and Treasures	4.421	5.859	8.992
Total	98.71	337.9	650.9

Table II: DungeonGen algorithms performance measures

IV. 2D Terrain Generation

In this case we are dealing with the shape of the terrain of the world or the regions where the events in the game occur. In The Legend of Zelda (Nintendo 1986), all events take place in the kingdom of *Hyrule,* which has a specific terrain shape, in which different elements are found such as forests, rivers, mountains, coastal areas, etc. where different creatures exist and where civilizations have also appeared, including the changes in the terrain associated with them, such as the construction of cities, towns, roads, etc. The challenge of procedurally generating civilizations and changes to the terrain according to historical facts, generated also in an algorithmic way, has already been faced by Dwarf Fortress (Tarn Adams 2006), where when the game starts, several generations of terrain are made, simulations of the passage of time and changes caused by possible civilizations until reaching a result that meets minimum requirements.

Although 2D terrain, or land, is a widely used resource in the video game industry, it does not involve playable content on its own. This implies that it is necessary to make several modifications or additions to the process of generating land according to the genre of the game (strategy, shooter, adventure, etc.) in order to use the results in the final product. The design of realistic terrains manually can be a very tedious job, so although the procedural generation of land is not used as a characteristic of the final product, it can be useful to help or suggest to the designers where the game could be set during the development phase.

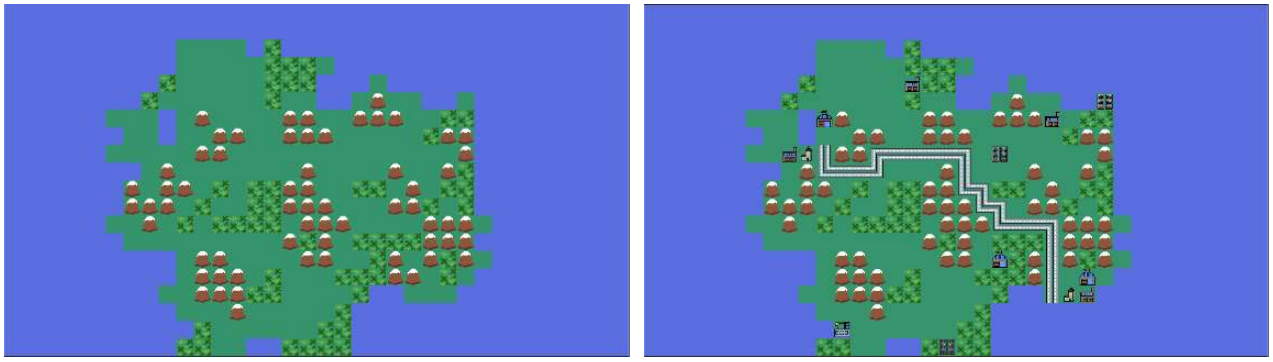
We have developed MapGen (Figure 4), a 2D maps generator based on grid and focused mainly on turn-based strategy games, initially for two players, inspired by the title Advance Wars (Intelligent Systems 2001). It is an interactive tool that allows the user to specify some of the parameters of the generated maps. The applied algorithms can be divided into two sections: those algorithms that are used to generate the 2D terrain and those that serve to adapt their result to the genre of turn-based strategy games.

Figure 4: Snapshot of the MapGen demo

To generate the land, the map is composed of the following kinds of terrain each one occupying a cell: water, plain, mountain and forest. The different combinations of them in a bidimensional matrix will create the generated land. We have developed

our own method, inspired by techniques such as Cellular Automata (Schweitzer et al. 2003), which aims to imitate, in a simplified way, the process of creation of volcanic islands. So the grid will initially be formed only by water cells. Later, we will choose some grid points in a random way, from which the plain type cells will begin to expand. In each iteration, the probability of expansion to adjacent cells decreases. At the end of the process of expansion of the plain we will have a primitive formation of the land. The placement of the forest and mountain type cells follows a similar process, but with certain restrictions. These cell types can only be expanded over other plain cells. With this we avoid generating isolated mountain formations by water and we also provide a certain coherence to the generated terrain, since the mountains and forests will not be randomly distributed over the surface of the islands. An example can be seen in Figure 5a.

In strategy games, it is common to face two players and each of them have initial establishments. In addition, resources also tend to exist and they are spread across the map symmetrically, so that both players have the same possibilities to take control. In our demo, the initial settlements are placed trying to maximize the distance between them in plain or forest type cells. Subsequently, a road is drawn between them, which will be the fastest way to reach the base of the opponent using the A* algorithm (Hart et al. 1968), which derives from the Dijkstra algorithm (Dijkstra 1959), but specializes to quickly find the shortest path between two points. The resources are placed symmetrically using the Dijkstra algorithm, assigning a cost to each cell according to its type. The result can be seen in Figure 5b.



(a) Terrain generated *(b) Adapted final map*

Figure 5: MapGen demo after different algorithms

The land generation algorithm places different cell types (water, plain, mountain and forest) in the map, inspired by the Cellular Automata technique, follows the next steps:

1. First, the dimensions of the map are chosen, filling it all with water cells.
2. Afterwards, a number of seeds of plain type cells are chosen and placed randomly.

3. This planted seeds influence the cells around them, causing nearby cells to become, with a certain probability, more plain cells. This process is repeated during a series of iterations and with each of them the probabilities of expansion are reduced.

4. Steps 2 and 3 are repeated for the mountain and forest type cells, but these must follow stricter expansion rules as they can only be extended over plain type cells.

The function chosen for the reduction of probability according to the iteration (starting with $i = 0$) is the following:

$$P(i) = \frac{2 * P_{initial}}{1 + i}$$

After some tests, we have selected the values in Table III for each of the types of cells for the number of seeds, the number of total iterations and the initial probability values of expansion.

Terrain	Seeds	Iterations	Initial probability
Plain	8	6	0.85
Mountain	18	1	0.2
Forest	20	1	0.2

Table III: MapGen terrain generation values chosen for different types of cells

The strategy games adaptation algorithm, given a previous map with a defined terrain, places the player's bases and the resources that players can capture trying to put them in locations where both players have the same opportunity to get them. The steps that the algorithm follows are:

1. The bases of the players are placed trying to maximize the distance in a straight line between them, in cells of type plain or forest. A little randomness id added to avoid monotony in locations.

2. A road that connects the bases of the two players is drawn. To select the layout of the road we use the algorithm A*, which derives from the Dijkstra algorithm. We give different costs to the different cells: 1 for plain, 2 for forest, 6 for mountain and 8 for water. So the roads will try to avoid crossing mountain or water cells.

3. The placement of resources by the map is done using the Dijkstra algorithm, only on the plain or forest type cells. This algorithm requires the calculation of two cost maps. We consider two types of resources: 1) Equidistant resources: those that are approximately at the same distance (respecting the cell's cost

system) of the initial establishments of the two players; and 2) Symmetric resources: those that are easier to capture by one of the players, because they are closer to the base of any of them, but for the existence of one of these, there must always be another equivalent for the opposing player.

To test the performance of our algorithms, we evaluated them, finding the main problems in the generation algorithm and the placement of the bases as they consume a high number of resources. We reduce the complexity of the generation algorithm by only taking into account in the current iteration the cells generated in the previous iteration, obtaining more uniform values in the iterations. With the placement of the bases, finding the longest possible distance between two valid cells of the entire map leads to a complexity problem $O(n \cdot (n-1))$ where n is the total number of cells in the map.

In Table IV we can see the generation times of the different algorithms in a laptop (Intel Core i7-7700HQ CPU, 16 GB RAM, Geforce GTX 1050 GPU with Windows 10 OS). The time used to find the optimal placement of the bases increases enormously as does the total number of cells on the map. From certain dimensions, suboptimal solutions might have to be considered to avoid long generation times, or performing these algorithms in background while the player is in the previous level, as doing it in real time will impact in the gameplay experience.

Map size vs Algorithms generation time (ms)	**10*10**	**32*32**	**60*60**
Terrain	3.628	6.300	297.33
Plain	2.584	31.80	212.33
Mountain	0.655	6.000	44.5
Forest	0.389	6.300	40.5
Settlements	11.08	1280.3	18,310
Bases of the players	9.735	1220	18,160
Layout of roads	1.345	30.30	253.8
Resources	5.451	101.2	665.7
Equidistant	5.115	84.8	490.5
Symmetric	0.336	16.4	175.2
Total	20.16	1,395	19,380

Table IV: MapGen algorithms performance measures

V. 3D Terrain Generation

We have developed MyCraft (Figure 6), a demo that uses 3D terrain generation as part of its content. The main peculiarity of these lands is that they are formed only by blocks, inspired by the game Minecraft (Mojang 2011) that also makes use of them. This demo consists of a small navigation simulator for terrains based on 3D blocks. The terrain through which the player can move is potentially infinite (not really, mainly due to memory limitations) since it is generated as the player navigates through it.

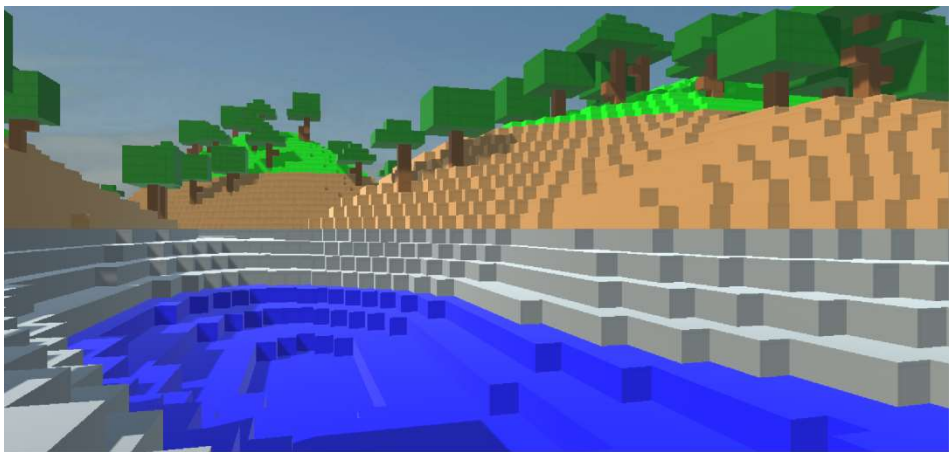
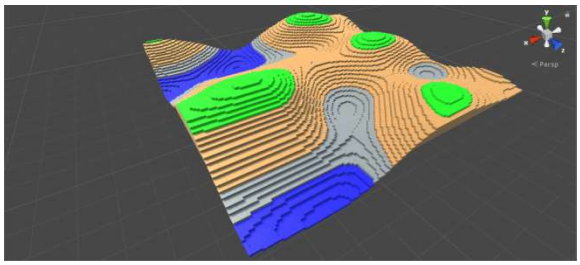


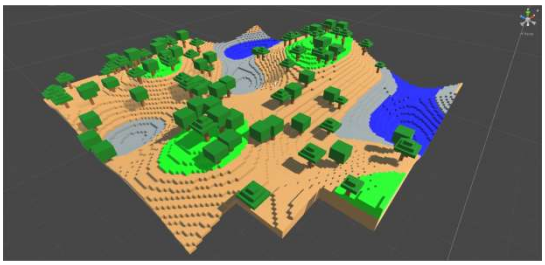
Figure 6: Snapshot of the MyCraft demo

For the generation of 3D terrain based on blocks, the best option is to model the terrain as a bidimensional matrix (because the coordinates to the axes corresponding to depth and length take discrete values) and save the corresponding height value for each point, as if it were a map of heights. Filling in the values of this matrix is not as simple as using a conventional random number generator, since the values obtained would be independent of each other and the generated terrain would lack coherence. Therefore, we must use other types of random number generators, whose output values have a minimal correlation, such as Perlin noise (Shaker et al. 2016, pp.57-72) (Figure 7a).

To get the terrain generated continuously as the player moves in one direction, some adaptations need to be applied. First, the terrain generated is divided into portions, commonly called chunks, of 16*16*16 blocks. This division is useful because we can generate the terrain on demand in a simple way and improve the performance of the game during navigation. When starting the demo, a certain number of chunks are generated around the player. When the player advances to another chunk, those that have been left behind will not longer be drawn, as can be seen in Figure 7b. Simultaneously, a number of chunks equivalent to the hidden ones are generated, with which the number of chunks shown is always the same and with it, the performance during navigation is more uniform.



(a) Terrain generated by Perlin noise.



(b) Division of the terrain into chunks.

Figure 7: MyCraft terrain

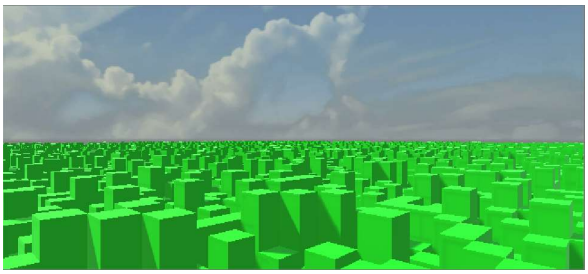
With the Perlin's Noise the gradient-based terrain looks more natural, it still retains a bit of artificiality because the terrain undulates at a constant frequency. A real terrain has variations at multiple scales, but in all of them a similar structure is preserved. The simplest way to produce land with these properties is to repeat the same process but at multiple scales, adapting its amplitude to the chosen frequency. Using the function:

```
heightMap [x, z] = Mathf.RoundToInt (Mathf.PerlinNoise (x, z) * maxHeight);
```

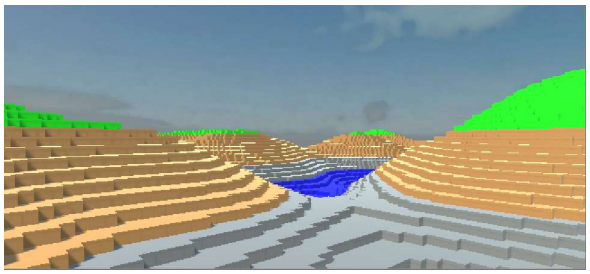
the results are not very satisfactory as we obtain a chaotic terrain (Figure 8a) due to the samples that we are taking are too far apart in space. In order to improve these results, the best option will be to use a scaling factor for the coordinates received by the noise generator. The scaling factor should be a decimal number between 0 and 1, with values close to 1 generate more chaotic terrain, while values close to 0 generate terrains with much smoother transitions between changes in height. In our tests we found that the most recommended values are in the range of 0.01 to 0.05. Finally we selected as scaling factor value 0.0175, obtaining the results that can be seen in Figure 8b. Another interesting improvement that can be used is a seed to allow generating different terrains. Finally the function used is:

```
heightMap [x, z] = Mathf.RoundToInt (Mathf.PerlinNoise ((x * scaler + seed), (z *
scaler + seed)) * maxHeight);
```

To fill these lands with a greater degree of reality, a good starting point could be to add details to the terrain as those that can be elements of an ecosystem, such as fauna or flora. So, we decided to add trees to the generated lands. The simplest option, but which also achieves a reasonable visual result, would be the distribution of trees according to the type of block, only grass or earth, and using a random number generator. To avoid monotony we can also randomly choose the height of tree trunks within a reasonable range and have at least two types of leaf layout. With this, the results obtained will be much more visually attractive, as can be seen in Figure 8c.



(a) Chaotic terrain



(b) Result with scaling factor

(c) Terrain with trees

Figure 8: Different stages of the terrain generated

To improve the generation of the terrain, and to take advantage of the calls made to the GPU by Unity, we decided to group the terrain blocks generated in chunks of size 16*16*16. With this a stable navigation performance was achieved, but it still entails a computational cost that is too high. In Table V we can see the generation times of a chunk and the initial navigable terrain, composed of 64 chunks. This data has been obtained using a laptop (Intel Core i7-7700HQ CPU, 16 GB RAM, Geforce GTX 1050 GPU with Windows 10 OS).

Time (s)	***Minimum***	***Average***	***Maximum***
Chunk generation	3.03	4.62	6.49
Initial terrain generation	10.23	10.84	11.32

Table V: Generation time of a chunk and the initial terrain

VI. Conclusions

There is no general method to produce content for video games automatically. In (Hendrikx and Meijer 2013), the authors compile the different types of content that can be produced through procedural generation and which techniques can be used to produce each of them. In (Shaker et al. 2016), the authors provide different ways to deal with the problem of procedural generation and deepens the operation of the techniques that can be used, citing as examples some commercial video games that use them. In our work, we have put in practice some of the different procedural content generation techniques that already exist, even more, we have added improvements on some of them and developed some created by us.

As a result, we have developed four demo games, all freely available. These demos have served to demonstrate the use of different artificial intelligent techniques applicable to the world of video games for procedural content generation.

Given this, the procedural generation of content can be considered as a diamond in the rough, whose polishing will be accelerated by the need for content production by large video game companies and by its potential combination with other fields of engineering, like Artificial Intelligence.

Games Cited

A.I. Design (1980) Rogue. Epyx (Unix).

Mojang (2011) Minecraft. Mojang (PC).

Hello Games (2016) No man's sky. Hello Games (PC).

Chucklefish (2016) Starbound. Chucklefish (PC).

Nintendo (1986) The Legend of Zelda. Nintendo (NES).

Blizzard North (1996) Diablo. Blizzard Entertainment (PC).

Blizzard North (2000) Diablo II. Blizzard Entertainment (PC).

Runic Games (2012) Torchlight II. Runic Games (PC).

Tarn Adams (2006) Dwarf Fortress. Bay 12 Games (PC).

Intelligent Systems (2001) Advance Wars. Nintendo (Game Boy Advance).

References

Ariyurek, S., Betin-Can, A., and Surer, E. (2019) Automated Video Game Testing Using Synthetic and Human-Like Agents. Available from: https://arxiv.org/abs/1906.00317v1 [Accessed 13 Jan. 2020].

Engemann, Ch., Sudmann, A., eds. (2018) *Machine Learning. Medien, Infrastrukturen und Technologien der Künstlichen Intelligenz*. Bielefeld: Transcript.

Ernst, Ch., Kaldrack, I., Schröter. J., and Sudmann, A. (2019) Künstliche Intelligenzen. Introduction. Special Issue „Künstliche Intelligenzen", *Zeitschrift für Medienwissenschaft* 21, pp. 10-19.

Escribano (2012) Gamification as the Post-Modern Phalanstère - Is the Gamification Playing With Us or Are We Playing With Gamification? In Zachariasson, P. & Wilson, T. (eds.) *The Video Game Industry: Formation, Present State, and Future*. Routledge.

gamesindustry.biz (2018) Global games market value rising to $134.9bn in 2018. Available from: https://www.gamesindustry.biz/articles/2018-12-18-global-games-market-value-rose-to-usd134-9bn-in-2018 [Accessed 13 Jan. 2020].

Goertzel, B. and Pennachin, C. (2007) *Artificial General Intelligence*. Springer.

LeCun, Y., Bengio, Y and Hinton, G. (2015) Deep Learning. *Nature*, 521 (May), pp. 436–444.

Newell, A. and Simon, H. A. (1961) GPS, a program that simulates human thought. In Billing, H. (ed.) *Lernende Automaten*. Oldenbourg, pp. 109–124.

NVIDIA (2019). Deep learning and AI Startup Incubator. Available from: https://www.nvidia.com/en-us/deep-learning-ai/startups/ [Accessed 13 Jan. 2020].

Russell, S. J. and Norvig, P. (1995) *Artificial Intelligence. A Modern Approach*. Prentice Hall.

Samuel, A. L. (1959) Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development* 3, pp. 211–29.

statistica (2019) Worldwide artificial intelligence market revenue. Available from: https://www.statista.com/statistics/621035/worldwide-artificial-intelligence-market-revenue/ [Accessed 13 Jan. 2020].

tractica (2019) Artificial Intelligence Software Market to Reach 105.8 Billion in Annual Worldwide Revenue by 2025. Available from: https://www.tractica.com/newsroom/press-releases/artificial-intelligence-software-market-to-reach-105-8-billion-in-annual-worldwide-revenue-by-2025/ [Accessed 13 Jan. 2020].

Turing, A. (1950) Computing machinery and intelligence. *Mind*, Vol. LIX (236), 1 October 1950, pp. 433–460.

Uber (2019) Uber AI. Available from: https://www.uber.com/de/de/uberai/ [Accessed 13 Jan. 2020].

Unity (2018) Introducing Unity's Guiding Principles for Ethical AI. Available from: https://blogs.unity3d.com/2018/11/28/introducing-unitys-guiding-principles-for-ethical-ai/?_ga=2.147125054.881379825.1564238426-901192280.1564238426 [Accessed 13 Jan. 2020].

Unity blog (2019) Available from: https://blogs.unity3d.com/2019/01/18/fostering-ai-research-meet-us-at-aaai-19/ [Accessed 13 Jan. 2020].

Notes

[1] All demos are freely available at https://defu.itch.io/