Procedural Content Generation of Custom Tower Defense Game Using Genetic Algorithms

Vid Kraner, Iztok Fister Jr., and Lucija Brezočnik(✉)

Faculty of Electrical Engineering and Computer Science, University of Maribor, Koroška cesta 46 2000 Maribor, Slovenia
{iztok.fister1,lucija.brezocnik}@um.si

Abstract. In the present day, it is difficult to imagine the development of computer games without the use of artificial intelligence. We see it utilized for gameplay, players modeling, playtesting, or content generation. In this paper, we focused on the content generation of a custom Tower Defense game named Save the Sheep. The Tower defense game is a strategic game, which was, in our case, proposed in a non-violent way. We generated key building blocks of the game with a genetic algorithm, i.e., a game map, unit placement, and a waves system. The final Tower Defense inspired game was implemented in the Unity game engine. The results show that by applying genetic algorithms, it is possible not only to generate content that makes the game more complex, but also more challenging and interesting for players.

Keywords: Evolutionary algorithms · Genetic algorithm · Procedural content generation · Tower Defense

1 Introduction

One would think that Artificial Intelligence (AI) and its utilization in computer games (also known as Game AI) has a relatively long history, but that is not the case. Interestingly, researchers and the game industry have rarely collaborated in the past, although this started to change when AI was implemented in board games like Chess and Go [12]. Recently, more and more researchers have begun following the new movement [10, 11, 15] where they focus on using AI for the design and production of games. The latter also includes Procedural content generation (PCG), which is one of this paper's main topics.

PCG is an area of Game AI that has lately seen a tremendous growth in interest [16]. PCG comprises methods for generating game content like levels, maps, textures, stories, items, quests, music, weapons, vehicles, characters, and also game rules [16]. Such an approach can greatly affect the game's replay ability since it provides the user a new adventure every time [12]. PCG could be achieved via multiple methods, but in this paper, we will focus only on one subset of evolutionary algorithms named genetic algorithms (GA).

The Tower Defense game characterizes as a strategy computer game and was first introduced in the 1990s [1]. Until now, researchers applied PCG to the Tower Defense

I. Karabegović (Ed.): NT 2021, LNNS 233, pp. 493–503, 2021.
https://doi.org/10.1007/978-3-030-75275-0_54

Game genre, but they either used PCG to generate levels [13], generate paths and monster sequences [4], or generate road maps, tower locations, and monster sequences [9].

This work presents the design, implementation, and testing of a custom nonviolent Tower Defense inspired game named Save the Sheep. The game was first sketched with respect to its non-violent nature. The Blender tool was used to produce and animate proposed 3D models, which were later imported to the Unity game engine. To optimize the key building blocks of the game, we utilized GA [8]. Thus, the primary missions of this paper are:

- to present a custom non-violent Tower Defense inspired game named Save the Sheep,
- to demonstrate which building blocks of the custom Tower Defense game can be optimized using a genetic algorithm,
- to show that procedural content generation of the custom Tower Defense game creates solvable and more complex game levels, and
- to show that procedural content generation can provide more interesting game for players.

2 Fundamentals of Tower Defense Games

Every Tower Defense Game consists of the following main components: Map Generation, Unit Placement, the Waves System, and the Reward System. The map of the game includes restrictions for where the player can build towers and also determines the path for the units. The tower limitations and the complexity of the path influences the level's difficulty. Unit or Tower Placement is an essential mechanism in a Tower Defense Game, which requires a strategy for arranging the towers and using resources. Usually, towers have different attributes which define different types of towers and bring their own depth to the game. The Waves System is responsible for the deployment of enemy units on the map which closely relate to the types of towers and their attributes. Their mutual balance is crucial for the course of the game and for the game's difficulty, by defining both the time between waves and wave composition. Lastly the Reward System serves developers by creating more content for players to explore and at the same time making a more challenging game. A common implementation of the Reward System is where the player is, by completing levels, rewarded special game resources, which he/she can spend to purchase upgrades.

3 Design and Implementation of the Save the Sheep Game

The development of the Save the Sheep game consisted of the following steps:

- sketching the plot of the game,
- identification of the main game elements,
- game implementation in Blender and Unity,
- development of a modified genetic algorithm,

- optimized map creation,
- optimized unit placement,
- development of a waves system.

All steps are depicted in the following subsections.

3.1 Sketching the Plot of the Game

When coming up with an idea for the Save the Sheep game, and because of the general belief that games encourage violence, we decided to set the theme of the game a little differently [8]. The goal of our game, unlike in classic Tower Defense games, is to make sure that the opponent's units reach the end of the path as seen in the sketch in Fig. 1. In the game, we have the role of a farm manager who wants to bring his herd of sheep safely to the barn. The sheep slowly lose energy during their way back. The player's job is to protect the sheep by setting up towers. The towers are in the shape of a farmer feeding the sheep with hay, a farmerette giving the sheep water and a speed boost, and a dog barking at the sheep to herd them towards the barn.

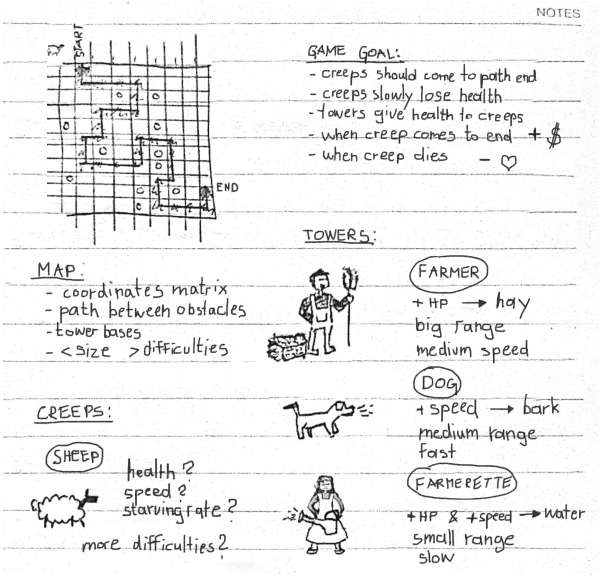

Fig. 1. Sketch of the first game idea.

3.2 Identification of the Game Elements

In our game, we identified three game elements to be optimized with GA. These elements are maps, unit placements, and waves. When implementing our game, we decided to exclude the Reward System, as its primary purpose is to increase the player's interest in

the game. In our case, we wanted to achieve this by using GA for generating building blocks.

In the following subsections, we present, in-detail, how GA utilization was carried out by providing technical descriptions and game examples.

3.3 Game Development in Blender and Unity

We used the Unity [14] Game Engine and the 3D modeling tool Blender [2] to implement the game. With the help of Blender, we first modeled 3D models for towers and the opponent's units. We decided for a low polygons graphic style, which is faster both to design and also later to run. For all 3D models, we first created a mesh and an armature and afterwards textured them. With the help of armature, we created the main animations and tested the models. Afterwards we imported all of the 3D models with textures and animations into Unity, which took care of assigning the correct material and recognized the animations of all models. Then, in Unity, we assigned an animation controller, which takes care of the animation and the transition between individual animations, to each of the models. The main mechanics of the game were implemented using C# scripts in the Unity Game Engine.

3.4 Development of a Modified Genetic Algorithm

The Genetic Algorithm is part of the largest subset of the evolutionary algorithms [5]. John Holland formulated the basic idea in 1975 [7], and later, with the help of Goldberg [6] and De Jong [3], this idea became a classic genetic algorithm. It is a metaheuristic and mimics biological evolution with operations like selection, crossover, and mutation. In our paper, the classic GA was modified to use it for the generation of the game key building blocks. The pseudocode of the modified GA is presented in Algorithm 1.

Algorithm 1 Pseudocode of the modified GA for the game elements formation

```
1: initpopulation();
2: evaluate population();
3: sort individuals in population();
4: while termination condition not met do
   5: elitist selection();
   6: roulette wheel selection();
   7: single point crossover();
   8: mutation();
9: end while
```

3.5 Optimized Map Generation

Map generation begins by defining a grid of coordinates matching the size of the map. First, the coordinates at the start and end of the map path are validated. If set, they must be on the edge of a map, otherwise they are chosen randomly. The map genotype consists of bit values of map size. Therefore, a map with a width of 5 and a height of 4 has 20 bit values. A bit value of 1 means that there is an obstacle at that coordinate. The genotype of such a random map can be 11000111100010010001.

It's phenotype is shown in Fig. 2, where the gray areas of the map are obstacles. If the example above had the beginning of the path at (2, 0) and the end of the path at (0, 2), then the phenotype of such a map in the game would look like the map in Fig. 3. There, bright green tiles represent the start of the path, brown tiles are the path itself and the house is the end of the path. Green raised platforms represent obstacles.

The parameters of the genetic algorithm for generating the map were adjusted through testing and set as shown in Table 1. The parameter that determines the obstacle probability was set at 62%. When the algorithm finds the best individual, the shortest path between the start and the end is found between obstacles. If the path does not exist, the algorithm is restarted. The path found is important later for both optimizing surfaces for building towers as well as for moving sheep.

(0, 0)	(0, 1)	(0, 2)	(0, 3)	(0, 4)
(1, 0)	(1, 1)	(1, 2)	(1, 3)	(1, 4)
(2, 0)	(2, 1)	(2, 2)	(2, 3)	(2, 4)
(3, 0)	(3, 1)	(3, 2)	(3, 3)	(3, 4)

Fig. 2. Abstract example of a 5 × 4 map.

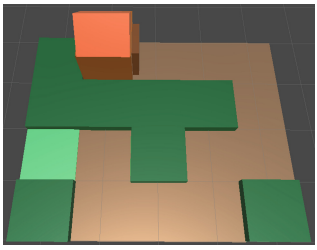

Fig. 3. Example of a 5 × 4 map.

Table 1. Parameters for map generation GA.

Population size	Elitism	Mutation probability	End condition 1	End condition 2
500	5	20%	Fitness repeats 5 times	Generations number over 50

Random Genes

Genes are randomly set for each individual of the GA population. The function that defines if genes are obstacles is passed to the algorithm as a parameter. The function

returns a value of 1 if a randomly generated number between 0 and 1 is less than the obstacle probability. For example, if the probability of the obstacle is 62% and the randomly generated number is 0.45, the function returns a 1 (0.45 is less than 0.62).

Fitness Function
The main goal of the fitness function for generating a map is to give a high score to the map which has a path between the start and the end through randomly placed obstacles. In the implementation of the fitness function, we first check if any of the obstacles are at the start or end of the path. In this case, we return a fitness of 0, otherwise we increase the fitness of the map. We then check to see if there is a path from start to finish. If the path exists, we give the map a high score. Otherwise, if the path does not exist, the map is assigned a proportional fitness of the closest distance to the end of path. To avoid an incorrect path, we reduce the fitness if the path is diagonally connected to the end of the path.

3.6 Optimized Unit Placement

Optimizing the locations for tower placement begins, when the map with an available path is generated. We name these locations tower bases. The genotype of tower bases consists of bit values whose size is the number of obstacles on the map. This basically means that towers can be built only on map obstacles. For example, a map with 8 obstacles would then have 8 bit values, where value 1 means that the tower can be built on that obstacle. The random genotype for the previously presented 5×4 map could be 00001011. Its phenotype is shown in Fig. 4, where tower bases are presented by gray circles.

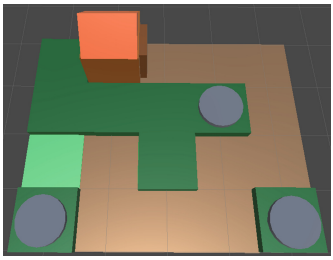

Fig. 4. Example of a 5×4 map with tower bases.

GA is also used for tower base optimization. Its parameters are presented in Table 2. The algorithm also requires a parameter for the number of tower bases we want to have on the map. For example, in Fig. 4 this would be 3. This parameter plays a role in calculating tower base possibility or with other words, a chance that the bit of genotype has a value of 1. The possibility is calculated by dividing a specified number of tower bases with a number of obstacles.

Table 2. Parameters for optimized unit placement.

Population size	Elitism	Mutation probability	End condition 1	End condition 2
100	4	5%	Fitness repeats 5 times	Generations number over 50

Fitness Function

The fitness value of a genotype is calculated by checking tower bases on a previously generated map. This is done by allocating tower bases specified in genotype to the obstacles on the map. For each tower base, we then inspect the surroundings within a range of two tiles. Each tower base has values for minimum distance to the path, index of the nearest path tile, and impact value. The impact value represents how many path tiles are in the 1 tile range. For a better understanding, the data we track is presented once again on the already known example of a 5 × 4 map in Fig. 5. From the acquired data, the fitness is then calculated. High fitness is determined by the number of tower bases near the path, the even distribution of tower bases over the path and high-impact tower bases.

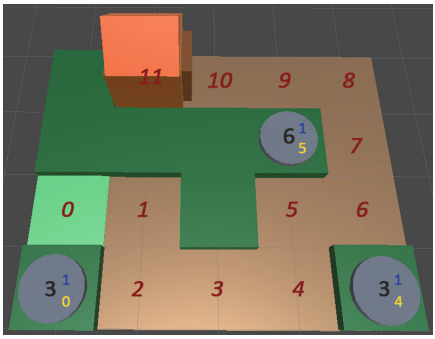

Fig. 5. Example of a 5 × 4 map with tower bases and their details.

3.7 Development of a Waves System

The wave system is optimized for each wave individually, according to the current number of waves and the number of total waves. With optimization, we are trying to achieve fewer units of easy difficulty in the first waves, and later more units with harder difficulty. The genotype consists of two real numbers. The first represents the difficulty value and the second the number of units in the wave. The parameters of the genetic algorithm are presented in Table 3. Furthermore, the optimization process requires additional difficulty level parameters. A level is given for each of the difficulty units, where the default value for the easy unit is 50%, for the medium unit 30%, and 20% for the hard unit.

Table 3. Parameters for optimized waves system.

Population size	Elitism	Mutation probability	End condition 1	End condition 2
50	2	10%	Fitness repeats 5 times	Generations number over 50

Fitness Function
Unlike previous fitness functions, in this algorithm, an individual first receives a certain fitness, which then decreases according to the deviation from the appropriate difficulty value. Fitness is calculated with Eq. (1).

$$f = \frac{|scr_m - scr_w|}{scr_m + (1 - wp)} \tag{1}$$

where:
f – fitness of the individual,
scrm – appropriate value of wave difficulty,
scrw – the actual value of wave difficulty,
wp – wave progress.

4 Experiments and Results

After a successful generation of the key building blocks of the game with GA, we conducted an experiment to demonstrate its effect on the game. The following subsections focus on a comparison of the game's fundamental building blocks creation with and without applied optimization. All experiments were performed on a 15 × 10 map with the same instance of the random number generator; therefore, we got the same optimization result, despite running the experiment multiple times. After the first round of the experiments, we gave the game to five gamers who graded it and provided feedback.

4.1 Map Creation With and Without Optimization

Generating a map without optimization leads to a map where obstacles are placed randomly, and the path between start and end almost certainly doesn't exist. In Fig. 6, we see a comparison between the map we took from the initial random GA population and a map that is a final optimization result. When testing different parameters of the algorithm, we found that the biggest problem for an incorrect solution is the obstacle rate parameter. The algorithm could be improved by choosing a more suitable version for the crossover which would mix the genetic code less drastically.

4.2 Unit Placement With and Without Optimization

Generating random positions for tower bases often leads to unsolvable game levels. This is mainly due to two reasons, the first is an uneven distribution of tower bases, and the

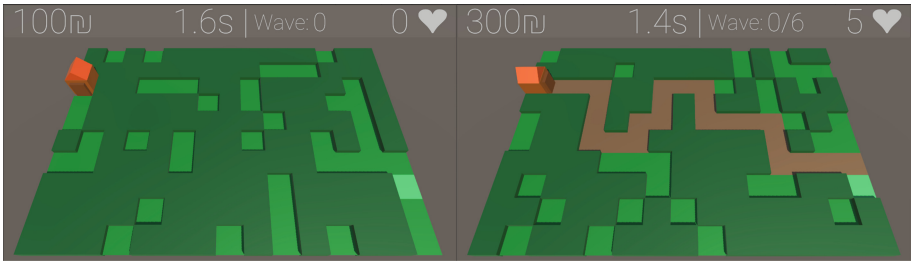

Fig. 6. Randomly generated map without optimization (left) and with optimization (right).

second is the tower base's distance from the path. In contrast to the classic implementation of Tower Defense games, where the uneven distribution of towers is not an obstacle, this is crucial in our game. The comparison between random and successfully optimized solution is presented in Fig. 7. With our optimization, we ensured that the vast majority of game levels are solvable.

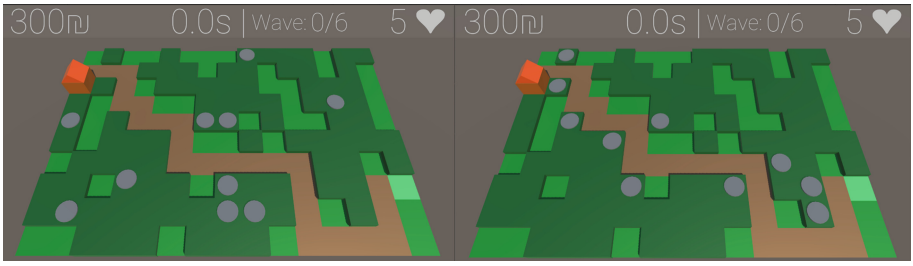

Fig. 7. Positions of tower bases without optimization (left) and with optimization (right).

4.3 Waves System With and Without Optimization

Among the implemented optimizations, the optimization of the wave system was the most difficult one. The main reason behind this lies primarily in the complexity of this system. The main goal of wave system optimization is the appropriate difficulty for the current state of the game. The difficulty of a wave is influenced by several factors, such as the attributes of the units, the number of units, the time/distance between the units, the time until the next wave, and the amount of a player's resources. We had challenges primarily with the implementation of the appropriate fitness function to satisfy the first four previously mentioned factors. In Fig. 8 we can observe both a non-optimized system, which is a predictable system where the difference in difficulty is only an increase in the number of units, and an optimized system, where each wave is optimized separately according to the current completed waves.

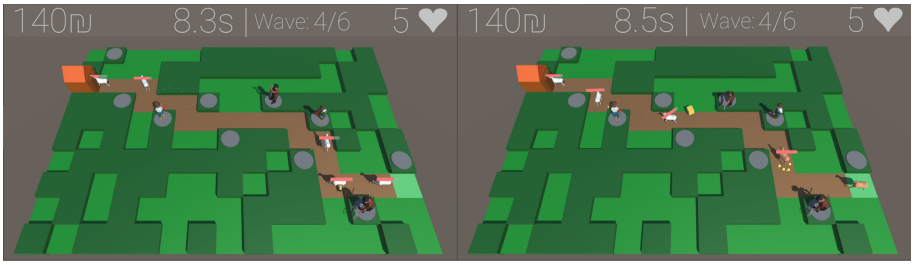

Fig. 8. Fourth wave of a game with a wave system without optimization (left) and with optimization (right).

4.4 Gamers Feedback

We gave the game to five independent gamers with the purpose of providing us with some feedback. The chosen gamers did not know each other, so they did not communicate during the testing, which served our goal of getting as accurate data as possible. They first had to play the game without the optimization, followed by the game where we optimized the game's key building blocks. In the end, they had to provide us with an overall opinion of the game, and judge if the optimized game seemed more interesting.

Out of five independent players, none of them said that the non-optimized game was better. Furthermore, all of them said that the optimized game was much more challenging and interesting to play, and would choose it over the non-optimized version.

5 Conclusion

This paper showed how we approached the design and implementation of a custom Tower Defense inspired game named Save the Sheep, which was developed in the Unity game engine. For an optimal generation of the game's key building blocks, i.e. a game map, unit placement, and waves system, we utilized a genetic algorithm. The results demonstrated that applying such an approach enhances the game's creation process and makes the final game more complex. The game was also tested by five independent gamers, who stated that the proposed optimized game was much more challenging and interesting to play than the non-optimized version.

We want to utilize other evolutionary algorithms to optimize the key building blocks of the game in the future. To tackle multiple targets in game development, i.e. various targets for map generation, we intend to use multi-objective evolutionary algorithms. Furthermore, the presented approach could also be extended to other games.

References

1. Avery, P., Togelius, J., Alistar, E., Van Leeuwen, R.P.: Computational intelligence and tower defence games. In: 2011 IEEE Congress of Evolutionary Computation (CEC), pp.1084–1091. IEEE (2011)
2. Blender.: Official Blender Website (2020). https://www.blender.org/. Accessed 05 Dec 2020

3. De Jong., K.A.: Analysis of the behavior of a class of genetic adaptive systems. Technical report, University of Michigan, USA (1975)
4. Du, Y., Li, J., Hou, X., Lu, H., Liu, S.C., Guo, X., Yang, K., Tang, Q.: Automatic level generation for Tower Defense games. In: 2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC), pp. 670–676. IEEE (2019)
5. Eiben, A.E., Smith, J.E.: Introduction to evolutionary computing. Springer, Berlin (2015)
6. Goldenberg, D.E.: Genetic algorithms in search, optimization and machine learning (1989)
7. Holland, J.H.: Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence (1975)
8. Kraner, V.: Using evolutionary algorithms for generation of key elements of Tower Defence game using Unity game engine. Master's thesis, University of Maribor, Faculty of Electrical Engineering and Computer Science (2020)
9. Liu, S., Chaoran, L., Yue, L., Heng, M., Xiao, H., Yiming, S., Licong, W., Ze, C., Xianghao, G., Hengtong, L., et al.: Automatic generation of Tower Defense levels using PCG. In: Proceedings of the 14th International Conference on the Foundations of Digital Games, pp. 1–9 (2019)
10. Mateas, M.: Expressive AI: games and artificial intelligence. In: DiGRA Conference (2003)
11. Nareyek, A.: Game AI is dead. Long live game AI! IEEE Intell. Syst. **22**(1), 9–11 (2007)
12. Risi, S., Preuss, M.: From chess and atari to starcraft and beyond: how game AI is driving the world of AI. KI-KünstlicheIntelligenz **34**(1), 7–17 (2020)
13. Sutoyo, R., Winata, D., Oliviani, K., Supriyadi, D.M.: Dynamic difficulty adjustment in tower defence. Procedia Comput. Sci. **59**, 435–444 (2015)
14. Unity: Official Unity Website (2020). https://unity.com/. Accessed 05 Dec 2020
15. Yannakakis, G.N., Togelius, J.: The 2010 IEEE conference on computational intelligence and games report [society briefs]. IEEE Comput. Intell. Mag. **6**(2), 10–14 (2011)
16. Yannakakis, G.N., Togelius, J.: Artificial Intelligence and Games, vol. 2. Springer, New York (2018)