A Procedurally Generated World for a Zombie Survival Game

Nikola Stankic, Bernhard Potuzak, Helmut Hlavacs

A Procedurally Generated World for a Zombie Survival Game

Nikola Stankic, Bernhard Potuzak, and Helmut Hlavacs

University of Vienna, Entertainment Computing, Vienna, Austria
helmut.hlavacs@univie.ac.at

Abstract. We present a method for procedurally creating game worlds, in our case levels for playing zombie survival games. The aim is to randomly create new levels that present new challenges to players, are fun, and "work" as game levels, i.e. look like levels that have been hand crafted. We create the topology, paths to follow, random houeses, and hordes of zombies to fight against. Players have to reach an end of the level, fight against zombies, and reach the final objective. The paper describes our approach, and presents an overall evaluation of players of the game.

Keywords: Procedural level generation · Zombie · Shooter

1 Introduction

Procedural Content Generation (PCG) in video games has been around for a long time, dating back to the 1980s [1]. A partial reason behind this were hardware restrictions, e.g. limited disc space, so some games turned to PCG in order to overcome these restrictions. A good example for this was a game called .kkrieger, which is only 96 KB in size [7]. Among the first games to have used procedural content generation are (i) Rogue (1980) [14], which is said to have started the trend of using PCG in video games also for the purpose of replayability [12], and (ii) Elite (1984) which was able to generate hundreds of unique star systems without having to explicitly store the data [19, 1]. Nowadays disc space and game size are not a problem anymore, but ther resources like budget, time and team size are. For example, the popular video game World of Warcraft was made over the course of five years, with an estimated budget between 20.000.000$ and 150.000.000$ [11]. All of this could be greatly reduced by generating at least some of the content, if not all, procedurally. But even when the aforementioned resources are not a problem, a lot of games today use procedural content generation, or at least some elements of it, in order to achieve a certain level of replayability [18].

Exactly that was the goal of this research: to produce a playable first-person shooter game utilizing procedural content generation, based on pseudo-random number generators, in order to create a small survival world from scratch and achieve at least some level of replayability. Pseudo-random Number Generators

(PRNG) are one of the simplest and most common techniques used in procedural content generation [11], with other common techniques being grammars, L-Systems, Fractals and Graphs [1, 4]. The game was made in Unity 3D, using assets from the Unity Asset Store. This paper describes our approach, as well as an evaluation of the result with several experimental participants.

2 Related Work

Almost all types of content for a video game can be procedurally generated. Starting with terrains and continents, rivers and roads [4], from buildings, characters and vegetation to whole virtual cities [8, 11, 4], worlds and even universes. Endless variety of non-playable characters and planets could be created, like in the game Spore [15], or billions of unique weapons like in Borderlands [9].

Nowadays, of course, there is a large amount of games that use procedural content generation in one way or another, but in this section, some of the more popular survival-based game titles will be mentioned. Popular choices for this are Minecraft [16] and No Man's Sky [10]. Minecraft uses procedural content generation to create an endless survival world with different thematic areas called biomes, and No Man's Sky procedurally generates a large amount of planets, and populates them with also procedurally created creatures and vegetation. Both of these games, however, have endless worlds and also a huge focus on exploration, where we wanted our game to have a smaller map instead, where the player can complete the objectives in short time.

Another conceptually similar game to ours is Left 4 Dead, which is a zombie survival first-person shooter game. The game uses something called the AI Director, which utilizes procedural content generation to place zombies and consumable objects and weapons around the map based on player performance [6]. It is also able to procedurally change the layout of the level by blocking paths, forcing the players to take a different routes in order to increase replayability [5, 6]. This approach offers great elements of surprise, however, it still requires a hand-made, predefined map, where our project is supposed to generate the whole level randomly.

The most conceptually similar tool is an engine called British Countryside Generator, used by a survival game called "Sir, you are being hunted" [13]. The engine uses the Voronoi diagram space partitioning method to create villages, then places buildings next to a previously generated road [2]. The goal of the engine is to create an open world level which resembles the British countryside. We wanted to implement a more random approach in our project, with more randomly-placed buildings and paths.

In our case, the goal was to make a first-person shooter survival game in Unity 3D. The game was supposed to have a completely procedurally created map from scratch, using pseudo-random number generators (PRNG).

The player controls and moves a first-person character. They have two weapons to choose from: a faster-shooting, but weaker pistol, and a stronger, but slower shotgun, which has a smaller magazine and takes longer to reload. By going into

buildings, the player is able to find consumables, like med-kits to heal if they have been hurt, or ammo boxes to reload their weapons. The objectives of the game would be the following: the players have to find the book necronomicon and pick it up; then they have to destroy four pillars which are scattered around the map; and finally, after completing the previous objectives, a portal opens, with which the player has to react in order to destroy the necronomicon. The player wins and the game is over after completing this final step.

3 Implementation

The first steps are creating a new Terrain Game Object, adding it to the game scene, and attaching a new script to it, which was named "World Generator". The default Terrain Game Object was used for this. All the other Terrain settings were left to default, as anything that had to be changed would be done on run-time with the script. Initially, a hexagonal grid to the terrain object was also included, in order to use the grid's cells as points for path and area generation, but it was later decided against it in order to include more placement options.

3.1 Path Generation

The algorithm for generating the path is fairly simple. All it needs are two points, a start point and an end point, with the start point being randomly chosen on one side of the map using a PRNG, and the end point then being calculated from the start point, so that it is on the opposite side of the map. The algorithm then starts from the start point, calculates the normalized direction towards the end point, and generates a new path point a fixed distance in this direction, with a possible offset of 60 degrees, 30 degrees in both positive and negative directions, which is also chosen by a PRNG. This is an iterative process, and ends when the algorithm reaches a very close proximity to the end point. All these path points are represented as spheres and are saved in a global container.

Figure 1 shows one of the possible outcomes of the path generation algorithm. As said before, at first, a hexagonal grid was used for this, with each grid cell representing one path point, but the algorithm was changed later in order to achieve a more "random" path pattern. Now that a path is ready, the algorithm can move on to other steps, starting with generating different areas around the path.

3.2 Area Generation

This part of the algorithm places different areas around the path built in the previous step. Each area is indicated by a sphere, and is defined by a type, like "Abandoned village" or "Trees" for example, which tells the algorithm what kind of objects are supposed to be found in that certain area, and a size, which indicates the radius of the area. An adaptation of a similar existing space filling algorithm [3] was used. Said algorithm is used to procedurally create two-dimensional geometry and textures by randomly placing shapes of decreasing

Fig. 1: Unity scene view of a random path possibility generated with the algorithm

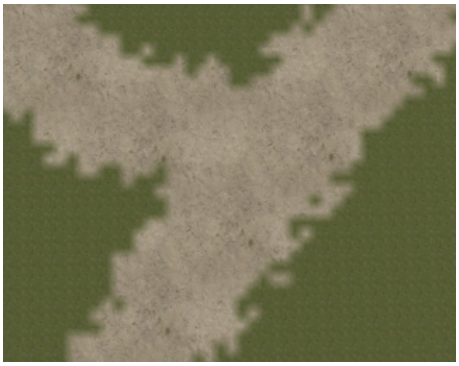

Fig. 2: A close-up look of a terrain alpha map offset between 'grass' and 'dirt'

size onto a two-dimensional plane. Our algorithm iteratively tries to place areas of decreasing radius onto random points, once again chosen by a PRNG. The key constraint here is that the areas are not overlapping with each other, as well as with any of the path points. The areas are placed in a decreasing size order, in order to make sure that smaller areas do not use up all the space and make it impossible for larger areas to be placed. The iterative process for placing areas is repeated until either a maximum number of iterations, or a maximum number of placed areas of current size, has been reached, both of which parameters are predefined, with the former being a fail-safe against an endless loop. After each area has been placed, its center is connected to a nearest existing path point, using the same algorithm previously described. After all areas have been placed, the remaining space, not already occupied by an area, is filled with trees.

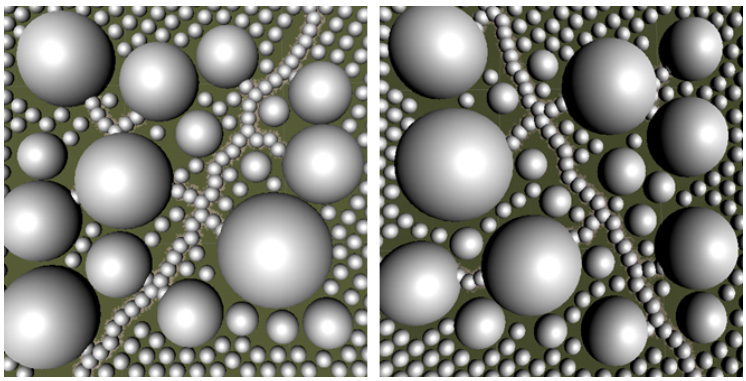

Fig. 3: Unity scene view of two possible area placements

Two possible outcomes of area generation can be seen in Figure 3, with the areas and path points denoted as spheres. Just like with path generation, a hexagonal grid was used here as well, but was replaced later for the same reasons. Now that the areas are ready, the algorithm can start populating them with buildings and objects.

3.3 Object Generation

Object generation is done in a very similar fashion to area generation. For each area generated in the previous step, objects are generated within it using the same random space filling algorithm. Objects, just like areas, are defined by a type, like "building", "tree" or a "prop", and a size. The type of the object generated depends on the type of the area which the object is associated with. Buildings are placed first, and "trees" and "props" can fill the remaining space. There are currently three thematically different building types that can be generated. The maximum number of objects generated in each area depends on the size of that area. If the type of the area is a "village", and the area is large enough to have two or more buildings in it, then each of the buildings generated is also connected to the nearest path point. In this step, however, only the information, about what kind of an object is to be spawned and where, is saved. The actual objects are not yet instantiated because there is no information about terrain heights yet.

3.4 Terrain Alphamaps and Heightmaps

After all the path points have been generated, the algorithm is now ready to generate alphamaps for the terrain. The terrain uses two textures - a "dirt" texture for the path and a "grass" texture for - well, everything else. Alphamaps in Unity are a three-dimensional float array, with the first two dimensions representing the x and y positions on the terrain, and the third dimension being splatmap texture to be applied (which, in our case, are the two "dirt" and "grass" textures). Here, the alphamaps are generated by going through all the terrain points, and, if a certain terrain point intersects with a path point-sphere, the texture for that terrain point is set to "dirt", otherwise it is set to "grass". A randomly generated offset is used here as well, when calculating the intersection, in order to achieve a more realistic dirt-grass pattern (Figure 2).

After that, the heights for each terrain point are set using perlin noise, which is one of the common pseudo-random number generator techniques. Heightmaps in Unity are a two-dimensional float array, each dimension representing the x and y positions on the terrain respectively, and having a float value between 0 and 1, which denotes the terrain height at that point.

3.5 Object Instantiation

Now that the information about the height of each terrain point is available, objects are ready to be instantiated. As mentioned before, there are currently

three different types of objects: building, tree and prop. Trees and props are placed in a same way: a random game object of that type is selected and instantiated at a position defined in step three, its height (the y-position) is set to the corresponding terrain height, and it is also rotated by a random amount around the y-axis. The same goes for buildings, they are, however, not rotated by a random amount, but are rather rotated so that they are facing the area center.

As mentioned before, we were limited by the building assets we could find in the Unity Asset Store, so at first we only had two different types of buildings. We realized that it would be too monotonous seeing the same buildings over and over again, so we decided to use some modular assets and combine them with our own script in order to procedurally generate different buildings as well. For this, a similar approach to something described as 'Building Blocks framework' in [17] was used. The script would use a previously man-made layouts (a template) for the floor, walls, ceiling and roof, and instantiate random corresponding components at the appropriate positions. The inside walls can be instantiated as well, which are chosen at random, with the number of placed walls also being random, up to a predefined maximum, potentially separating the inside of the building in two different rooms in the process. There are two versions of this script, one generating a warehouse-like buildings from six different layouts, and is also able to change the textures of inside walls, floors and the ceiling. The other version is not as advanced, as it generates a "ruined house" with only one layout and texture.

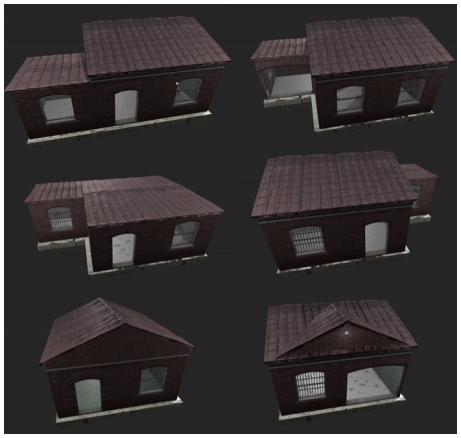

Fig. 4: Some of the warehouse buildings generated with the building generator

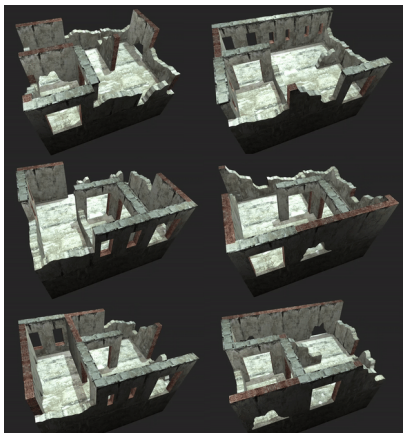

Fig. 5: Some of the ruined buildings generated with the building generator

Figures 4 and 5 show some of the possible outcomes of the building generator both for warehouse buildings and ruins. Each building is also able to contain other objects inside of it, like med-kits, zombies, or ammo boxes, and those things

are randomly instantiated in this step as well. The positions of potential items in buildings are also predefined with the same template-like system. After placing each building, the algorithm uses a Random Number Generator to determine if- and which object it is going to place at each position.

3.6 Finalization

At this point, the map can look like something in Figures 6 and 7. Now that the terrain is done and all the objects are finally placed, it is time to finalize the map and make the game playable. First, some grass details are added to the terrain. The engine uses a details map for grass, which is a similar mechanism to the alpha map. The algorithm basically ignores the path on the terrain, and paints the grass and potentially also flowers details everywhere else. It does this by going through all the terrain points and checking if the terrain texture there is set to "dirt" or "grass", which is how it distinguishes if it is a path or not.

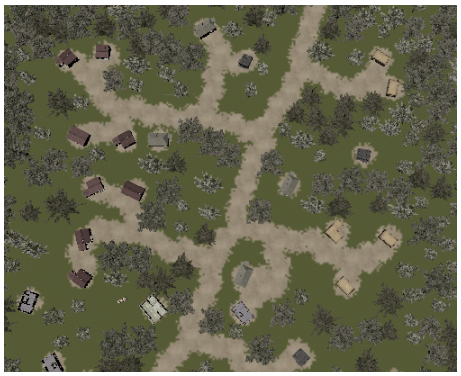

Fig. 6: Unity scene view of a possible world outcome

Fig. 7: Unity scene view of another possible world outcome

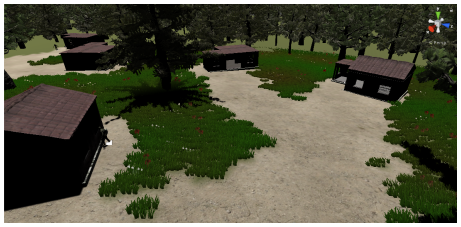

Fig. 8: Unity scene view of a generated village

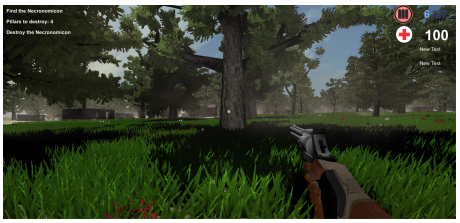

Fig. 9: Gameplay during daytime & sand storm

The final step includes placing the player at the beginning of the main path, placing the zombies around random path points, and placing the objectives. The pillars are placed along the path in order to be more visible and easier to find.

The necronomicon is placed in one of the houses in order to encourage the player to explore the buildings. Since everything is in place, the game is then ready and playable, see Figures 8 to 9.

4 Evaluation and Discussion

A total of 14 users were asked to play our game a few (at least 3 to 4) times and then fill out a questionnaire. Before starting the game, it was briefly explained to them what the game was about, how it works, what they had to do, and finally what the controls for the character are. We did not need any additional information about the users, so the questionnaire was anonymous. The questionnaire is composed of a total of twelve questions, which are split into two categories: the 'Gameplay' and the 'World', with each having six questions. The questions were formed as declarative statements, like for example "I enjoyed the game", and the answers were in a form of a classical Likert rating scale, allowing the users to rate the statements from 1 to 5, with 1 being "strongly disagree" and 5 being "strongly agree". This also gave us an opportunity to calculate the average rating of all the answers for an overall average rating of the user experience.

The Gameplay category of the questionnaire focused on the player's ability to understand the game mechanics and the UI, as well as their overall enjoyment. This part included the following questions: "I clearly understood what the objectives were", "I had no trouble controlling my character", "The User Interface was easy to understand", "I used the following game features: both weapons, flashlight, map, med-kits, ammo boxes", "I had not trouble completing the game", and "I enjoyed playing the game".

The World category of the questionnaire focused on the procedurally generated world and buildings component of the game, player's ability to navigate through the level and overall replayability of the game. It included the following questions: "I found it easy to navigate through the level", "The level was different/refreshing each time I played", "The buildings I encountered were different/refreshing each time I played", "I tried to explore the map and go into buildings", "I did not have any trouble finding med-kits or ammo", "I would describe the game as 'replayable' ".

Players were also given an option to leave their own comment at the end of the questionnaire.

4.1 Results and Discussion

After all the player sessions were done, we summarized the results of our questionnaire. Figure 10 shows the pie charts of the results for the Gameplay category of the questionnaire, and Figure 11 shows the pie charts of the results for the World category of the questionnaire. An average rating for all the answers was 4.21 out of 5. The results of both parts of the questionnaire will be discussed separately, starting with gameplay.

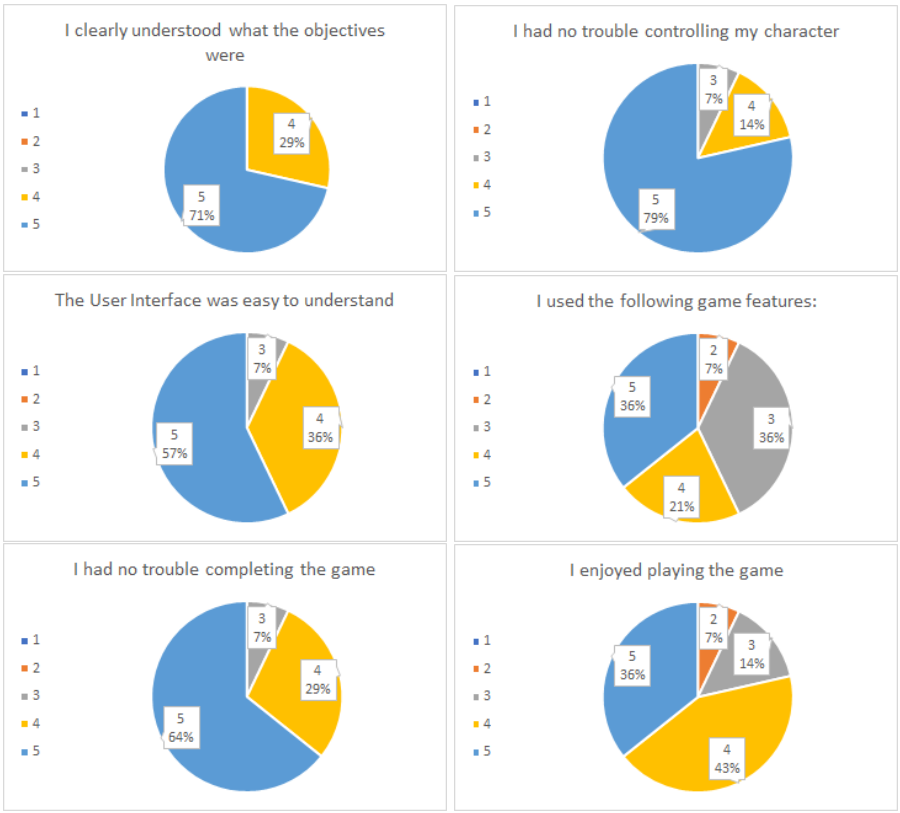

Fig. 10: Questionnaire results for the Gameplay category

All of the users agreed with "I clearly understood what the objectives were", where 10 out of 14 users gave it a rating of 5, and the rest 4 gave it a rating of 4. These are very good results, as they tell us that the instructions of the objectives were clear and the users knew from the start what they were supposed to do. Also none of the users disagreed with having trouble controlling the character. This is probably because we used the standard input commands for most first-person games (move the character with W/A/S/D keys and use the mouse to turn around and shoot) and most of the users were familiar with that. It could also be because the users did not have to use a lot of additional keys, besides the ones for turning on the flashlight, switching the weapons, interacting with objects, toggling the map and sprinting. Most users, with the exception of one, also agreed that the User Interface was easy to understand. This was an expected result, as the UI was not too crowded and only displayed things like objectives, current/max ammo and health, and additionally the map when the player decides to toggle it on. We got some mixed results from users when asked if they used all of the listed features. This was the case because some users did not bother to explore the buildings that much, so they were unable to find

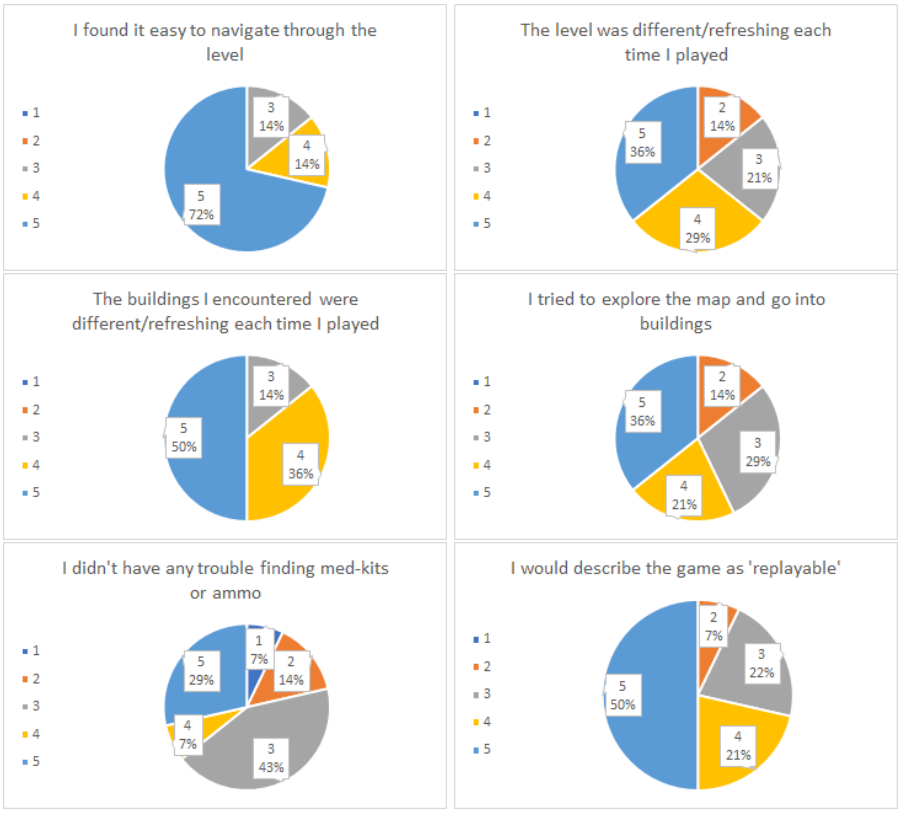

Fig. 11: Questionnaire results for the World category

any med-kits or ammo boxes. Some users even decided not to interact with the zombies at all, so they did not need any health or ammo pickups, nor did they use both weapons. Almost all users agreed on being able to complete the game, with the exception of one which gave this question a neutral rating. 11 out of 14 users also agreed when asked whether they enjoyed playing the game, where 2 users replied with "neutral" (rating of 3) and one person disagreed (rating of 2), giving this question an average rating of 4.07. Overall the gameplay part received good results, indicating that our goal for the game to be enjoyable was reached.

The questions from the world category of our questionnaire received some mixed results. When asked if they found it easy to navigate through the level, almost all users agreed that it was, while 2 answers were neutral. This indicates that the players were probably almost always aware of their position, and could tell at least which parts of the map they have already visited. The map feature also helped here. 9 out of 14 users agreed that they noticed the level to be different each time they played, with the remaining 3 feeling neutral about this statement, and only 2 disagreeing. This could be because the map was always

thematically similar, so they confused that with not being different. The majority of the users, however, thought the buildings did look different each time they played, which probably also helped them navigate through the world by recognizing the places they have already visited. Only 57.1 % of the users agreed that they tried to explore the map and the buildings, and 9 out of 14 users did not agree that they had no trouble finding med-kits and ammo boxes. This could be because they did not explore the buildings, but it could also be an indicator that a purely PRNG technique for placing consumable objects is not ideal, or simply that our algorithm had a way too low chance of generating such objects in those places. When asked if they would describe the game as "replayable", 10 out of 14 users agreed, 3 were neutral and 1 disagreed, giving this question an average rating of 4.14, which at least suggests that our goal for the game to be replayable was met.

5 Conclusion and Future Work

In conclusion, we were able to create a whole survival world procedurally and randomly, and fit it within a playable game. The user tests proved the game to be enjoyable and replayable.

We believe that the game has a lot of potential, and could be expanded to achieve more even more replayability. Different world themes, like a desert, or mountain village, could be added, along with more potential objectives and game modes. Other gameplay elements could be added, like more weapons, grenades, and even different types of zombies. The script for procedural generation of buildings could also be extended. It could include even more layouts, or be expanded to create more complex buildings.

Procedural content generation is a powerful tool, and even with its simplest technique such as PRNGs, a lot can be achieved. This research has shown us that a whole world could be successfully procedurally created using this simple, yet effective technique. Unfortunately, PCG has a strong limitation when it comes to re-usability. Most algorithms for procedural content generation are made specifically for a certain application, unlike, for example, some AI algorithms which could be reused in different games. Nevertheless, PCG has a lot of potential which should be, without a question, further explored.

References

[1] Nuno Barreto, Amilcar Cardoso, and Licinio Roque. "Computational Creativity in Procedural Content Generation: A State of the Art Survey". In: Nov. 2014. DOI: 10.13140/2.1.1477.0882.

[2] Tom Betts and James Carey. *Procedurally generated content in Sir, You Are Being Hunted*. 6-Nov-2013.

[3] Paul Bourke. "A Space Filling Algorithm for Generating Procedural Geometry and Texture". In: *GSTF Journal on Computing* 3 (July 2013). DOI: 10.7603/s40601-013-0004-2.

[4] D. M. D. Carli et al. “A Survey of Procedural Content Generation Techniques Suitable to Game Development”. In: *2011 Brazilian Symposium on Games and Digital Entertainment*. Nov. 2011, pp. 26–35. DOI: 10.1109/SBGAMES.2011.15.

[5] Alex J. Champandard. *Procedural Level Geometry from Left 4 Dead 2: Spying on the AI Director 2.0*. 6-Nov-2009.

[6] Travis Fort. “Controlling Randomness: Using Procedural Generation to Influence Player Uncertainty in Video Games”. In: 2015.

[7] Jonas Freiknecht and Wolfgang Effelsberg. “A Survey on the Procedural Generation of Virtual Worlds”. In: *Multimodal Technologies and Interaction* 1.4 (Oct. 2017), p. 27. ISSN: 2414-4088.

[8] Werner Gaisbauer and Helmut Hlavacs. “Procedural Attack! Procedural Generation for Populated Virtual Cities: A Survey”. In: *International Journal of Serious Games* 4.2 (June 2017).

[9] Gearbox Software. *Borderlands*. Computer Software. 2009.

[10] Hello Games. *No Man’s Sky*. 2016.

[11] Mark Hendrikx et al. “Procedural Content Generation for Games: A Survey”. In: *ACM Trans. Multimedia Comput. Commun. Appl.* 9.1 (Feb. 2013), 1:1–1:22. ISSN: 1551-6857.

[12] Rilla Khaled, Mark J. Nelson, and Pippin Barr. “Design Metaphors for Procedural Content Generation in Games”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’13. Paris, France: ACM, 2013, pp. 1509–1518. ISBN: 978-1-4503-1899-0.

[13] Geoff Manaugh. *British Countryside Generator*. 2013.

[14] Dagda Tanner Mattheus. *Rogue - Video Game*. Chapel Hill, NC, USA: Ventana Press, Inc., 2012. ISBN: 6138576403, 9786138576402.

[15] Maxis. *Spore*. Computer Software. 2008.

[16] Mojang. *Minecraft*. 2011.

[17] Gillian Smith. “Understanding Procedural Content Generation: A Design-centric Analysis of the Role of PCG in Games”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’14. Toronto, Ontario, Canada: ACM, 2014, pp. 917–926.

[18] Gillian Smith et al. “PCG-based Game Design: Enabling New Play Experiences Through Procedural Content Generation”. In: *Proceedings of the 2Nd International Workshop on Procedural Content Generation in Games*. PCGames ’11. Bordeaux, France: ACM, 2011, 7:1–7:4.

[19] Julian Togelius et al. “Procedural Content Generation: Goals, Challenges and Actionable Steps”. In: *Artificial and Computational Intelligence in Games*. Ed. by Simon M. Lucas et al. Vol. 6. Dagstuhl Follow-Ups. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, pp. 61–75.