International Journal of Multimedia and Ubiquitous Engineering
Vol.12, No.9 (2017), pp.41-52
http://dx.doi.org/10.14257/ijmue.2017.12.9.04

Procedural Content Generation for Dynamic Level Design and Difficulty in a 2D Game Using UNITY

Gilbert Nwankwo, Sabah Mohammed and Jinan Fiaidhi

Department of Computer Science, Lakehead University, ON, Canada
{gnwankwo, mohammed, jfiaidhi}@lakeheadu.ca

Abstract

The gaming industry has quickly turned into a multi-billion dollars a year industry, even now rivaling the movie industry, recent games like Unchartered a thief's end amongst many has seen movie producers play a huge role in the story line for the game. Daily statics from steam online store will show over 12 million active players during peak period of the day. This large active player numbers translates into huge profit for the gaming Industry. With gamers demanding more engaging content and quality, researches like ours into game content generation, and many others in game design and game theory help to address some of the current problems faced in the industry. Procedural content generation (PCG) is the algorithmic generation of game level and content with little or no human input. Level design is an art which consists of creating the combination of challenge, competition, and interaction that players call fun and involves a careful and deliberate development of the game space. When working with PCG it is important to review how the game designer sets the change in difficulty throughout the different levels. Procedural content generation promises to be a powerful tool for independent game developers, since they do not have to a big budget and a large team of designers, artists and programmers, the automatic creation of content can make them more competitive. In this paper, we propose and introduce a generic PCG generator for a 2D tile-game using UNITY. The generator is a main component of the game and without it the game will not function. We also implemented a dynamic difficulty adjustment (DDA) component which uses some criteria to dynamically adjust the difficulty of the game after each level, thereby maximizing the level of satisfaction the gamer gets from the game. This project addresses the following problem in the gaming industry (1) Improves Replay-ability feature (2) Personalized to the gamer (3) Reduce disk space (4) PCG generator as the main component

Keywords*: Games, Level Design, Procedural Content Generation, Dynamic Difficulty Adjustment, Platformer, Game Design Pattern, Game analysis, Genetic Algorithm, Content creation, Player satisfaction, Adaptive games, UNITY*

1. Introduction

At this very moment, across the world millions of people are engaged in game play on various platforms (Console, PC, Mobile) either online via collaborative or cooperation games or local content single or two player games. [1]Daily statistics from steam online store will show over 12 million active players during peak periods of the day. This large active player numbers translates into huge profit for the gaming Industry. Researches like ours, and many others in game design, game theory and game content generation help make even better quality and engaging games. [2]Game design is the art of applying design

Received (May 5, 2017), Review Result (July 23, 2017), Accepted (September 10, 2017)

[1] http://store.steampowered.com/stats/

[2] https://en.wikipedia.org/wiki/Game_design

and aesthetics to create a game to facilitate interaction between players for entertainment or for medical, educational, or experimental purposes. Game level is the area/space of the game where the gamer moves around the game world and interacts with the content of the game. For many game genre, the blend of the space and the content helps to tell a story and holds the distinct set of challenges gamer's experience. Considering these factors, we can conclude the game level can make or break a game. Procedural Content Generation is the algorithmic creation of game content, with little or no human input. In recent years, games like Battlefield 3, Borderlands series, Diablo series and Minecraft has shown PCG can be applied to commercial games. However, except for Minecraft, PCG was applied to minor features like aesthetics, rather than to produce entire game levels. One of the very first games with procedural level generation is ROGUE, designed in 1980 and ported to the IBM PC in 1984. Rogue is an ASCII adventure dungeon crawling video game which levels are generated every time the gamer starts a new game. Though generated levels were not complex because the design is based on ASCII text and symbols, the innovation was not in the design of the levels but in the ability to generate infinite numbers of levels without human intervention.

In this article, we address a few of the current research challenges in the gaming industry (1) Replay-ability feature of a game (2) Personalized game (a game designed with the gamer in mind) (3) Storage space (4) PCG as the main component of the game. This research is parallel by a significant increase in PCG research in academia, we will be aiming to achieve an online runtime level generation of certain types of content relevant to the game. No matter how good a game is or well received when it was released, games do not stay relevant beyond 12 months, this is simply because the content of the game is static. Playing the game, a second time or third time becomes boring because all the plot and twist is already known. With a dynamically generated level content, the replay-ability feature of the game is increased exponentially because no matter how many times the gamer plays the same game, the chances of the playing the same level again is very slim. Research has shown a gamer gets bored playing a very easy game while at the same time gets frustrated playing a very difficult game, in our project we propose to strike a balance by using the principles of dynamic difficulty adjustment to alter the difficulty level of the game based on the performance of the gamer. The difficulty level of the subsequent level is determined by how the player performed in the current level.

We focus on generating entire game levels, both the spaces in which the player can maneuver, and the content with which he can interact. The set of game levels in incredibly large and diverse, typically presenting distinct PCG challenges. In this project, we narrow our focus on a tile game. A tile-based game is a type of game where the playing area consists of small rectangular square, parallelogram or hexagonal graphical images, referred to as tiles. When Sony PS4 game console was released with 500GB disk size, gamers were thrill. The consensus was now we can download endless number of games to our console. That did not work out as expected, the improvement in console hardware directly led to demand in high quality game which also takes up huge disk space.

2. Related Research

What does it really mean to design a game? [1] In the study of games, Brian Sutton-Smith writes, "Each person defines game in his own way (the anthropologists and folklorists in terms of historical origins; the military men, businessmen and educators in terms of usages; the social scientists in terms of psychological and social functions)". It can be deduced that the idea of designing a game is largely influenced by the game designer's background and experience. However, regardless of each designer's individuality, all games must have some certain basic game design principles and practice. [2] In design practice the user's ability to act is conveyed through affordance and the user's inability to act is conveyed through constraints. To be able to procedurally generate suitable content for a digital game the algorithm should be able to produce output that is meaningful to the player. Cellular automata, is one of the techniques for procedural level generation in dungeon games. [3] This self-organizing structure consists of a grid of cells in any finite number of dimensions. Each cell has a reference to a set of cells that make up its neighborhood, and an initial state at zero time (t = 0). To calculate the state of a cell at the next generation (t + 1), a rule set is applied to the current state of the cell and the neighboring ones. The representational model of a cellular automaton is a grid of cells and their states. Johnson, Yannakakis *et al.*, [4] use the self-organization capabilities of cellular automata to generate cave levels. They define the neighborhood of a cell as its eight surrounding cells, where its possible states are {floor, rock, and wall}. After an initial cell conversion, the rule set is iteratively applied in multiple generations.

In 2012, Manuel Kerssemakers *et al.*, [5]in their paper introduce a procedural level generator generator for platform game. The system is an interactive evolutionary algorithm that uses agent based level generator, these agents draw and erase blocks by walking on the canvas of the level. The evolutionary algorithm presents candidate artifacts to a human user, who responds by assigning a fitness value or selecting which ones of the presented artifacts he/she likes best. Fausto Mourato *et al.* [6] proposed a framework for automatic level generation, using genetic algorithm (GA). This algorithm is based on Darwin’s theory of Natural selection, it simulates the process of evolution by sorting a set of entities, then use a fitness function to evaluate each entity with a score and making the most scored more willing to continue to the next generation. Areas of the game level are represented in grids, grouped in screens of 10 by 3 cells. Thus, making it theoretically possible to generate all conceivable levels for this game by generating all possible combination of cells. [7] Ferreira, Luca, and Claudio Toledo also presented a GA for the procedural generation of levels in Angry Birds game. The GA evaluates the levels based on a simulation which measures the elements’ movement during a period. The algorithm's objective is to minimize this metric to generate stable structures. The level evaluation also considers some restrictions, leading the levels to have certain characteristics. Markus Persson’s Infinite Mario Bros! [8] Generates levels by probabilistically choosing a few idiomatic pieces of Platformer levels and fitting them together. While this technique produces levels that, on the surface at least, look and feels a great deal like the popular Mario game, over time it becomes apparent that there is very little variety between levels due to the repetition of idioms.

Jim Whitehead *et al.* in 2009 proposed a Rhythm-based approach to 2D Platform content generation. This approach first generates rhythms, and then the geometry of the platforms is generated based on those rhythms. They identified that, because platform games are based on leaping over obstacles and jumping gaps created by separating platforms, the key behind a 2D platform level design is the notion of the rhythms and the timing and repletion of distinct user actions. In 2011, Jim Whitehead *et al.* [9] completed implementation of *Launchpad*, a rhythm-based level generator for 2D Platformer games. Levels are built out of small segments called “rhythm-groups” which are generated using a two-tiered, grammar-based approach. Polymorph is a dynamic difficulty adjustment and level generation tool developed by Martin Jennings-Teats *et al.* of University of California. Based of varying skill levels gamers might find games to be uninterestingly easy or frustratingly difficult for others. A solution to this problem is dynamic difficulty adjustment, an approach that alters the game difficulty based on the player’s performance. Polymorph [10] employs techniques from level generation and machine learning to understand game component difficulty and player skill, dynamically constructing a 2D platformer game with continually appropriate challenge. Van der Linden *et al.* in 2013 [11] proposed a method which empowers game designers to author and control level generators, by expressing gameplay-related design constraints. They designed a dungeon crawler game which method uses graph grammars, the result of the designer-expressed constraints, to generate sequences of desired player actions. These action graphs are used as the basis for the spatial structure and content of game levels; they guide the layout process and indicate the required content related to such actions.

Many Unity 5 developers have in past Game Developers Conference (GDC) demoed and argued the pros for switching to procedural content generation but also cautioned restraint, basically not every entity in a game needs to be procedurally generated. [3]Daniil Basmanov created a free Unit Procedural Toolkit that acts as a template for any developer to download and use in his game project. Though his methodology was not explained in his documentation, he had various generators that include character generators, maze generators, primitive generators amongst others.

2.1. Problem Statement

Almost all existing approaches to PCG focuses on generating content for an existing game, where the core game itself could exist without the PCG mechanism. PCG is merely used to facilitate design and adaptation. In 2D Platformer level generation tools like Tanagra and Launchpad, the game level is generated by the tools and then applied to the game engine or platform by the game designer. Even in hugely successful games like Rogue, Spelunky and Diablo, where a key feature of the game is the endless variation in game content, all the levels could in principle have been generated offline and presented to the player without taking player choice into consideration. This research paper intends to develop a game where procedural content generation is a central mechanic of the game, without which the game could not exist at all. A true PCG-based game where the content of the game is generated in real-time, choices and skill level of the gamer affects the level of difficulty of the game, to enhance player satisfaction.

3. Methodological Approach

Agile software development methodology is the new approach in developing software products. It provides a refreshing and new ways of managing IT development team and projects, different from waterfall or sequential software development lifecycle approach. Likewise called iterative software development. Because this research work involves designing and building a PCG game, it is very important to go with a methodology that is adaptive rather than predictive. Game story development is highly subjective, and usually a change in any aspect of the story creates a ripple effect that affects most of the game design and logic, hence using an adaptive software development approach such as agile allows for a seamless reactive approach to such changes.

A sprint is a set period during which specific work/iteration should be complete and made ready for review. In this project, we have identified 4 sprints

- Mobile-First Approach
- Procedural Content Generation
- Game Story development
- Dynamic Difficulty Adjustment

3.1. Game Design Patterns and Principles

Most of the basic game design patterns and principles used in this implementation originates from the free book [4]Game programming patterns by Bob Nystrom. The implemented patterns include:

[3] https://github.com/Syomus/ProceduralToolkit
[4] http://gameprogrammingpatterns.com/contents.html

- **Command Pattern**. This is used for button controller callbacks, it creates an easy way to rebind keys, undo old movements and make a cool replay function.
- **Flyweight Pattern**. Also, sometimes referred to as the thousand instances pattern, it shows how to save memory by reusing what objects have in common to minimize the amount of data is pushed to the GPU.
- **Observer Pattern**. Involves setting up event listeners on game objects, to react in a certain way when the event is received. *e.g.* Door opens when player stands in front of door.
- **Singleton Pattern**. This is this pattern one can create only one instance of a class. It is one of the most popular patterns, often used to create persistent music across levels and keeping player scores.
- **Update Method**. The idea is that the game world has a collection of objects whose behavior must be updated each frame. Each of these objects must have an update method, and each frame the game updates every object in the collection. It is interesting to note that frame differs from one device to another, I fast moving game content one on device might be reduced to crawling on anther device.
- **Affordance**. This refers to the clues sent by a game object to the player about its interactions with other objects in the game world. In this case of our game story, the player needs to know which farm animal is likely to die at what time by seeing the health bar on each animal.

3.2. Evaluating Content Generators

I have created a content generator, now what? It is not just enough for a content generator to generate content and auto design game levels, it should be evaluated in some form. But what is a good content generator, is the interesting level I created a fluke, a result of a bug, or genuine result? That depends very much on what the generator was designed to create and why. An important consideration in PCG is understanding how well the generators can be controlled to produce different kind of outputs, and especially how small changes in rule systems or priorities alter the expressivity of the system. If a designer requests that the system create shorter levels, will it still produce a broad range of content?

We are going to evaluate the proposed generator in terms of their capabilities as generative systems and in performing evaluations of the content that they create.

4. System Analysis, Design and Implementation

Based on the system requirement, we can start to model the system. Agile methodology tells us that content is more important than representation. In this section, we will be describing the architectural design model and the detailed design model using both text and comprehensive design methods, ideally expressed in UML static and dynamic views.

Deciding on a game engine, we had to decide between Unity 5 or Unreal Engine 4. [5]After researching both platforms, their pros and cons, we decided to go with the Unity Game Engine. Unity was launched in 2005 at Apple's Worldwide Developers Conference and it has gone on to be the most popular and widely used game engine in the world. [6]It is estimated there are over 2 million registered Unity developers including brand giants like Coca-Cola, Disney, Nasa among others, and more than 350,000 active users. Another

[5] https://unity3d.com/public-relations
[6] http://www.cinemablend.com/games/Unity-Surpasses-2-Million-Registered-Developers-Bring-Games-57366.html

strong attractive feature of the platform is its flexibility with programming languages, unlike Unreal that is solely written in C++, Unity allows the developer to choose any one of three languages (two of those among the most popular in the world) JavaScript, C# and Boo. Our project will be developed using C# because of its strong support for OOP and because a large percentage of Unity developers use C#, making it easy to find solution to tough situations on forums.

Procedural Content generation and Dynamic Difficulty Adjustment techniques may take considerable time to be performed. Since generation occurs right before the output is required, (when a player starts a new level). We aim to keep the waiting time short, although this needs to be balanced with the quality of the generated levels. We aim to keep the game active between levels, by showing score progress as a visual distraction.

For this project, two of the main components is the PCG generator and the DDA criteria components. Figure 1 shows the sequential activity flow between components when the gamer clicks on the options button to Start Game, the PCG generator is called to instantiate and design the level from code, the level created is a 2-dimensional tile level, using some mathematical calculation to create X and Y number of equal size grids. The generator is divided into 5 main components depending on the requirement of the game story. The components are generated sequentially as depicted in Figure 8, this is because some of the components are dependent on the component on top of it. Generate Borders component determines the size of the screen aspect ratio and creates a set of wall boundaries to border the game level (Garden). This component keeps the Non-Playable Entities and the game protagonist within the boundaries of the game. Generate Maze, using pseudorandom generators this component generates tile coordinates based on a 2D dimension (x and y), which is then used to create bricks on the generated coordinates. Generate NPC, Generate Fox, Generate Rewards components, all these use the same pseudorandom generator method of the PCG Generator class to assign coordinates and then instantiate the game-object pass to the method on the game level coordinate. The Generate Rewards component implements the Risk and Rewards game pattern/theory, the reward is destroyed/disappears after 7 seconds if the protagonist doesn’t pick it up. DDA is not called for level 1, the system uses user defined settings on level 1. It should be noted that we only load the level once, the PCG Generator only recycles the game level after each level objective is complete. Hence allowing an endless possibility of game levels, without increasing the size of the game on disc.

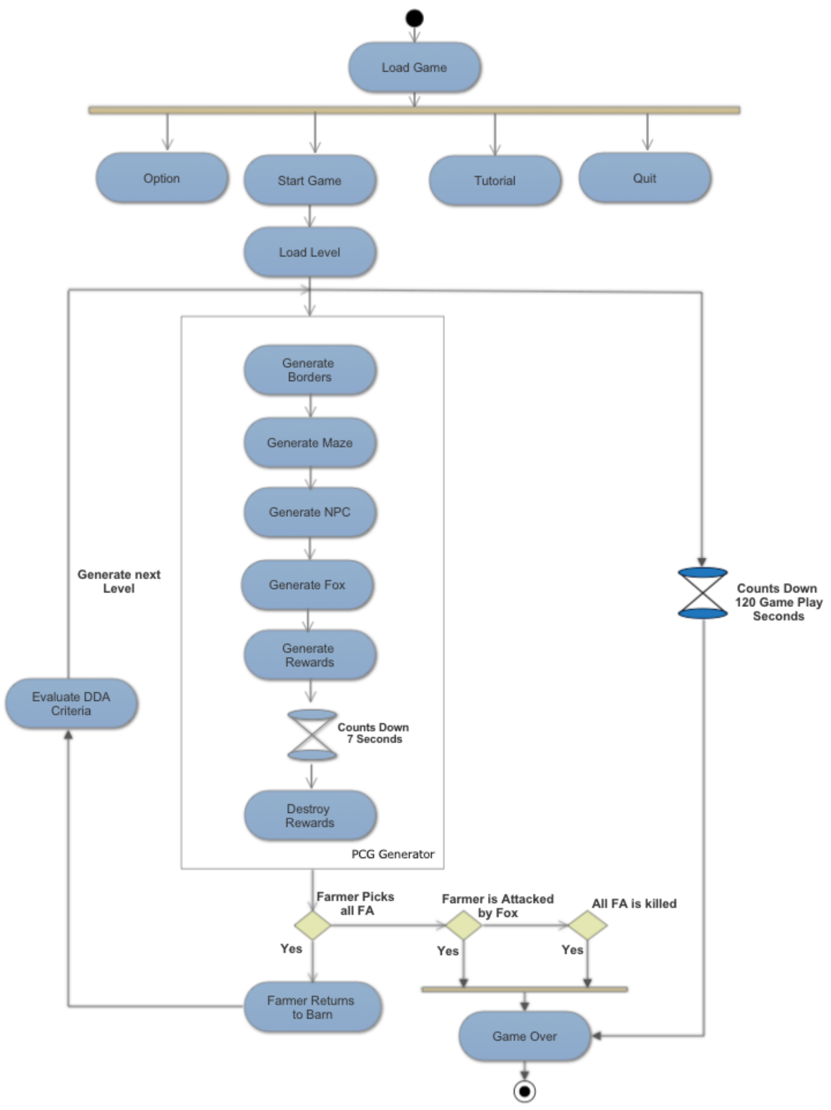

Figure 1. A Sequential Activity Diagram Representing the Flow of Activity from One Component to the Another

4.1. Implementation

The largest screen resolution on mobile devices, laptop/desktop screens and HDTV is 16:9 = 1.7777, therefore, we will target a game level implementation for 16:9. The game play level is expected to graciously scale on devices with lower screen resolution. We will be using the concept of quiet zones. A quiet zone refers to size of the viewport with reference to the screen size. In this case, the viewport is the smallest aspect ratio we want the game to be played on, and for this implementation we have chosen 4:3 = 1.3333. Figures 2a and 2b below shows the initial phase (No PCG) of the game stage. Figure 2a contains the quiet zones on the left and right and 2b have the quiet zones chopped off. To ensure we can always visualize the viewport, we have represented the viewport with a bright colored panel as shown on the scenes. We could achieve this by tweaking the canvas scaler script properties of Unity's canvas component.

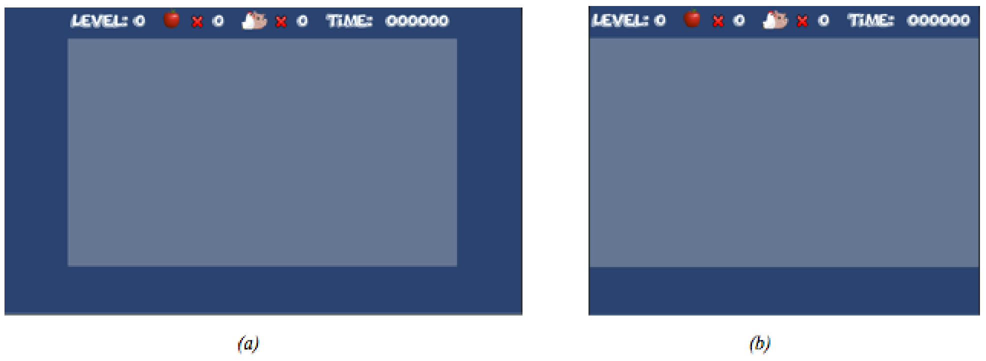

Figure 2. Screen Aspect Ratio 16:9 (a) Showing the Quiet Zones on Both Sides of the Screen, Aspect Ratio 4:3 (b) With Quiet Zones Chopped Off

UV Mapping

One of the main problem when designing a PCG generator is after the level game-objects is instantiated how do we position the game-objects in the game world to achieve Balance. An unbalanced system represents wasted development resources at the very least, and at worst can undermine the game's entire ruleset by making important roles or tasks impossible to perform. We need to keep track of what is created and its location on the game world. To solve this puzzle, we turned to [7]UV mapping. UV mapping is all about wrapping a 3D object with a 2D image. Because our game is a 2D game, we will UV map a grass image onto our game level. This will be done in such a way that a grass of tile represents 1-world unit, the bottom left corner of the viewport with represent coordinate (1,1) in 2-dimension space.

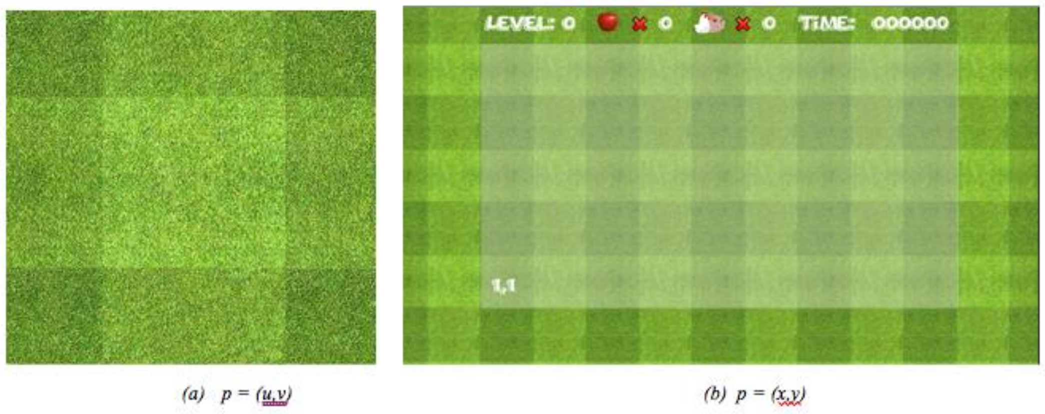

Figure 3. (a) 2D Image for the UV Wrapping. (b) Game Level after Uv Mapping is Applied and the Necessary Calculations Done

We converted the image in Figure 11a to a texture in Unity 5 Engine by changing its texture type to default. Second step, add a Raw image component to the level canvas component. Figure 3a is set to the texture property of the Raw image. We need to set the UV Rect width and height property of the Raw image. Our objective is to achieve 5 rows and 9 columns in 4:3 aspect and 5 rows and 12 columns in 16:9 aspect. Total number of tiles in 16:9, 16/4 * 3 = 12 tiles. There are 2 grass squares per tile in Figure 11a, therefore, for the UV Rect, we divide the number of tiles by 2. Width is equal to 6. Height, H = 6/16 * 9 = 3.375. Figure 11b, verifies our values. In aspect 16:9 we have 11 tiles with half tiles

[7] https://en.wikipedia.org/wiki/UV_mapping

on each edge adding up to 12 tiles wide in 16:9. The actual game viewport contains 9 columns and 5 rows with the bottom left and top right set to coordinates (1,1) and (9,5) respectively.

PCG Generator

- GenerateGameBorders: This class is responsible for generating the boundary of the game. Although, we know we want a 5 row and 9 col viewport, we have decided to generate the boundaries from code to make it generic. We used a 2-dimension array. While looping through the array, we create a border when the value is 1 and do nothing when the value is 0. The 2D position in the game world to place the objects is determined by the col and row values of the loop. See Figure 12a

```
private int [,] worldMap = new int[,] {
   {1,1,1,1,1,1,1,1,1},
   {1,0,0,0,0,0,0,0,1},
   {1,0,0,0,0,0,0,0,1},
   {1,0,0,0,0,0,0,0,1},
   {1,0,0,0,0,0,0,0,1},
   {1,1,1,1,1,1,1,1,1},
};
```

- PCG_MazeBricks: We declare a 2-dimensional array with the col and row size from GenerateGameBorders, we declare a static list array *mazePositions* to hold occupied positions on the tile. We use a pseudorandom generator to generate a number between 0 and 5 and then assign its value to the worldMap [col, row], we then check the value of worldMap [col, row] if its value is greater than or equal to 4, we instantiate a brick and position it on 2D plane [col, row] and finally add the position to *mazePositions.* See Figure 12b
- NonPlayableEntities: This class is responsible for generating the farm animals and the barn. We exposed a public game-object array, for this project, we set the size of the array to 2 and passed two farm animals (pig and Rooster) to the array, the array size can be increased to whatever number, if the number of animals match the size of the array. Because this script is called after the maze is generated, we have a method *DetermineEntityDropPosition* that checks and returns an empty tile position to place the generated farm animals and just like in the maze, the position is then added to *mazePositions.* It also contains a method *RemovePositionFromGrid*, that removes destroyed (killed) farm animal position from *mazePositions.* See Figure 12c
- GenerateFox: This class generates the skulk of foxes, it also uses the *DetermineEntityDropPosition and RemovePositionFromGrid* of the NonPlayableEntities class. The *mazePositions* is also used. The number of fox generated to hound the farm animals is determined by the difficulty set. See Figure 12d
- GenerateAppleRewards: This class implements the Risk and Rewards pattern. Using InvokeRepeating method of the MonoBehavior class, we set the method *ProbabilityOfApple* to execute after x seconds. The probability of generating an apple is Time.deltaTime * frequency, where frequency is a set value and Time.deltaTime is used to express frame independence, because devices run at different processing speed. See Figure 12d

- PcgGenerator: After a level is complete, this class calls all the above classes to recycle the level scene and generate all the necessary component to get the next level ready for play.

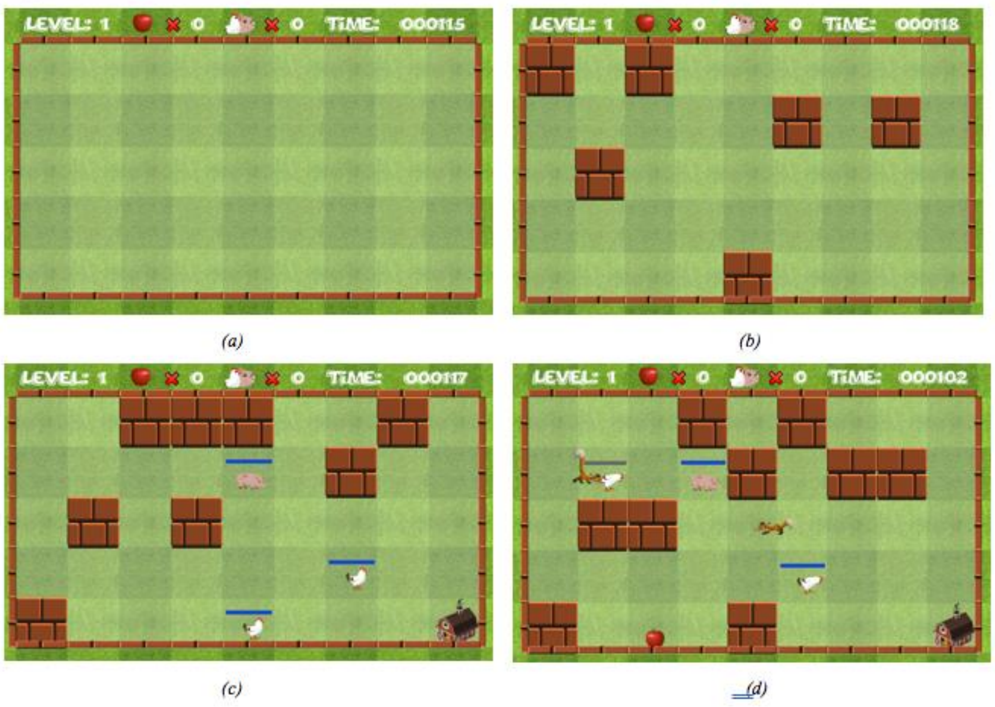

Figure 4. Different Stages of Pcg Classes after Execution

5. Conclusion and Future Work

The research objective of this research was to develop a true PCG-based game where the content of the game is generated in real-time. The PCG component is not merely a part of the game but the central mechanic of the game, without which the game could not exist at all. We proposed an approach that uses pseudorandom generators to assign coordinates on the game world and a form to determine if an entity is already placed on the coordinate and how and when to remove entities on the game world. With our approach of blending procedural content generation with dynamic difficulty adjustment, we have not only demonstrated we can develop a game with an infinite level capability using the smallest amount of disk space but we can also tailor it to the strength of the gamer by ensuring each level is not only unique in design but in player experience and satisfaction as well. We surveyed researches which showed a variety of PCG methods that are suitable for procedural content generation. From the analyzed papers, we concluded that a variety of different PCG method types can already effectively be controlled by designers. From this survey, we also noticed that the PCG method to implement also depends on the game’s genre, while pseudorandom generator and constraint checking worked for our tile game implementation it is highly unlikely that it will work on a 2D platform game, other widely implemented PCG methods include Automata and grammar. On a longer term, our generator can be extended between generation sessions. As such, the corresponding level will depend on the previous level thus incrementally generating tougher level maze for the gamer to navigate through.

References

[1] M. A. Elliott and B. S. Smith, “The study of games”, John Wiley & Sons, **(1971)**.
[2] D. A. Norman, “The design of everyday things”. Basic Books, New York, U.S.A., **(2002)**.
[3] V. der Linden, R. R. Lopes and R. Bidarra, “Procedural generation of dungeons”, IEEE Transactions on Computational Intelligence and AI in Games, vol. 6, no. 1, **(2014)**, pp. 78-89.
[4] J. Lawrence, G. N. Yannakakis and J. Togelius, “Cellular automata for real-time generation of infinite cave levels”, In Proceedings of the 2010 Workshop on Procedural Content Generation in Games, ACM, **(2010)**, pp. 10.
[5] K. Manuel, J. Tuxen, J. Togelius and G. N. Yannakakis, “A procedural level generator”, In 2012 IEEE Conference on Computational Intelligence and Games (CIG), IEEE, **(2012)**, pp. 335-341.
[6] M. Fausto, M. Próspero dos Santos and F. Birra, “Automatic level generation for platform videogames using genetic algorithms”, In Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology, ACM, **(2011)**, pp. 8.
[7] F. Lucas and C. Toledo, “A search-based approach for generating angry birds levels”, In Computational intelligence and games (cig), 2014 ieee conference on, IEEE, **(2014)**, pp. 1-8.
[8] M. Persson, “Infinite Mario Bros! (Online Game)”, Last Accessed: December 11, 2008. http://www.mojang.com/notch/mario/
[9] S. Gillian, J. Whitehead, M. Mateas, M. Treanor, J. March and M. Cha, “Launchpad: A rhythm-based level generator for 2-d platformers”, IEEE Transactions on computational intelligence and AI in games, vol. 3, no. 1, **(2011)**, pp. 1-16.
[10] J. Teats, M. G. Smith and N. W. Fruin, “Polymorph: dynamic difficulty adjustment through level generation”, In Proceedings of the 2010 Workshop on Procedural Content Generation in Games, ACM, **(2010)**, pp. 11.
[11] V. der Linden, R. R. Lopes and R. Bidarra, “Designing procedurally generated levels”, In Proceedings of the second workshop on Artificial Intelligence in the Game Design Process, **(2013)**.