Procedural Game Environment Generation in Independent Game Development: Case Study and Simulation Approach

by

Yihua He

A thesis submitted to

Auckland University of Technology

in partial fulfilment of the requirements for the degree of

Master of Computer and Information Sciences

2015

School of Computer and Mathematical Sciences

Primary Supervisor: Associate Professor Nurul I Sarkar

Abstract

The evolution of game development techniques and digital distribution platforms has not only boosted the power of mainstream game developers, but also provided a global opportunity for independent game developers. Various indie titles (e.g. Minecraft, Terraria and Starbound) show reliable sales records encourages more and more independent game developers to use procedural game environment generation (PGEG) techniques in their own game products. One of the key factors in improving a game product's popularity and life cycle is to provide a unique gameplay experience for each player. Procedural game environment generation has the ability to create unique worlds while greatly reducing the resource requirement during the development process. Although the characteristics of this technique appeal to millions of indie developers who have a low level of available resources, previous studies have focused only on procedural solutions for generating specific game environments; the study of flexible and fast PGEG frameworks is still a not well-explored area. According to previous research and practical case study, one of the most important factors in a PGEG system is performance. A PGEG framework is proposed and implemented along with a game prototype simulation in this thesis. The simulation approach uses the core features of the framework and a combination of Perlin noise and other procedural algorithms to generate a 2D town as the game environment. Players can control a small character to explore a real-time generated world with a smooth experience. Although the elements and structure of the prototype world are simple, a series of evaluation tests prove the viability and flexibility of the framework. Many objects could be rendered in a single frame using the optimized render system in the framework. The multi-threading procedural system and block system provide a simple but effective way of using various combinations of procedural methods to generate virtual worlds while maintaining good control of visual issues and system consumption. The stability and effectiveness of the framework mean that there is more computational power for practical game development. Throughout the evaluation and discussion, the framework is proven to be suitable and valuable for independent game developers who intend to build games based on PGEG. Additionally, recommendations for future developments are pointed out. They include developing more simulations among various procedural algorithms and generating a three-dimensional environment for further evaluation.

Acknowledgements

I would like to thank

- My supervisor Associate Professor Nurul I Sarkar for his guidance, discussions, patience and encouragement throughout this research work.
- My parents for their support in every aspect even they are thousands of miles away from New Zealand.
- Julie McMeikan for her professional proofreading advices on English grammar correction of this thesis.
- The authorities at AUT University for providing the opportunity for me to finish this research work.

Table of Contents

List of Abbreviations and Acronyms

AAA	Triple A quality
AI	Artificial intelligence
API	Application programming interface
ASP	Answer set programming
CPU	Central processing unit
CS	Complex systems
DAT	Data amplification technique
EDPCG	Experience-driven procedural content generation
FPS	Frames per second
GG	Generative grammars
GPU	Graphics processing unit
HCI	Human-computer interaction
IDE	Integrated development environment
IVG	Independent video game
KB	Kilobytes
LOD	Level of detail
MS	Millisecond
PC	Personal computer
PG	Procedural generation
PGCG	Procedural game content generation
PGEG	Procedural game environment generation
PRNG	Pseudo-random number generation
RMD	Random midpoint displacement
RPG	Role-playing game
SFML	Simple and Fast Multimedia Library
STL	Standard template library

List of Figures

List of Tables

Chapter 1

Introduction

Since the very first electronic game was created in 1947 [1], the video game industry has grown into a multi-billion dollar-valued market in the last few decades. Different from the movie industry, the interactions between users and virtual environments that are provided by video games have attracted more and more people to explore and spend time in the virtual world. The gaming population has grown into a huge group globally in recent years [20]. Today, large-scale video game publishers and developers are dominating the market. Numbers of AAA (Triple A means the highest level in the industry) titled video games are released by them every year and huge profit is generated by these creations. As the various software development techniques have advanced, more and more individuals and small studios have found out that they can actually create high quality games to earn profit from this huge market. Multiplatform game engines (such as Unity, Cry Engine, Unreal) have proved to be great frameworks for independent game developers [31]. However, within those top selling indie games such as Minecraft, Terraria and Don't Starve, one special technique has been used as the main technique for unique gameplay: Procedural Game Content Generation (PGCG). The most highlighted characteristic of this technique is to generate a unique game environment or game map for each new gameplay [20]. As a result, every new gameplay is a fresh adventure for players, as the environment is completely different from the last gameplay. As a matter of fact, the procedural generation technique was applied in early video games. Table 1.1 shows two famous 90s video games that used PGCG.

Table 1.1: Two early video games that used PGCG

Title	Release year	PG content	Developer
Diablo	**1996**	**Dungeons & Equipment**	**Blizzard Entertainment**
Civilization	**1991**	**World map**	**MicroProse**

Table 1.1 shows the types of game content that two 90s video games procedurally generated. Both of these titles were famous and top-selling during that period. However, PGCG was never the highlighted feature or selling point among these titles. PGCG was used as a supportive

technique for the gameplay. Due to the performance limitations of personal computers, PGCG did not bring much benefit to video game development at that time. Though PGCG itself is not a newly developed technique, using it in large open world generation leading it to great success has just happened in the past few years. In terms of the development of computer hardware and game development techniques, PGCG brings lots of possibilities for gameplay in real-time. But on the other hand, the high cost for AAA titles makes large-scale game developers and publishers avoid using PGCG as the main feature in their products, as PGCG cannot be completely controlled [20]. For independent developers, creative and unique gameplay is the most important feature in order to earn a place in the market, as most indie developers could not afford or get investment for developing cinematic games. After the great success of the independent game "Minecraft", which sold 14.3 million copies on PC and 54 million copies on consoles, PGCG has become a hot technique among indie developers, as "Minecraft" is mainly uses this technique to generate its unlimited world in real-time. Instead of handcrafting every inch of the game environment, PGCG provides the possibility of letting the application generate the environment following specified rules. In this case, the developers only have to create basic environment elements for the application to combine them procedurally to generate new content. It reduces the cost of development and provides unique gameplay. More and more indie game titles use PGCG to generate their game world, and many of them achieve great sales results. Investigating PGCG, and simulating a PGCG based framework for the game environment generation, is therefore valuable as a topic for study.

In this thesis, firstly a literature review of previous PGEG-related research was performed to find out the possibilities for creating a PGEG framework for the indie game industry. Secondly, a case study of the independent PGCG-based video game "Minecraft" was executed from a technical perspective. According to the results from literature review and case study, a simulation application was designed and implemented in order to investigate and evaluate the impact of PGCG in independent game development. The thesis has the following three main objectives:

1. To identify appropriate algorithms that has been used by most PGCG indie games.
2. To investigate current PGEG framework solutions through a literature review and case study.
3. To design and evaluate a PGEG framework for independent game development using a

simulation approach.

Understanding, implementing and evaluating the key algorithms was the most important factor in achieving the objectives above. Support for popular procedural methods could prove the availability of the framework. During indie game development, time consumption and cost are sensitive factors that affect the success of the final product. Studying whether PGCG would reduce these factors was therefore important. In addition to affecting time consumption and investment cost, it was also important to know if using PGCG would downgrade the gameplay experience. Finally, finding out what types of indie video game would benefit from this technique is necessary. As a result, during the case study section, the algorithms that "Minecraft" adopted to procedurally generate its environment are explored in detail. In the simulation implementation section, a prototype game is developed using the same key algorithms from the case study. Evaluation is based on theory and practical data, including speed of execution, system memory consumption and size of the game world created. By combing through the findings from the literature review, the disadvantages and advantages of PGCG in independent video game development can be investigated.

1.1 Contribution and the structure of this thesis

Figure 1.2 shows the overview of the thesis structure. The main introduction section includes three chapters, which provide the basic and necessary information as preparation before analysing and implementing the PGCG algorithms. Chapters 4 to 7 combine to make the main contribution. The literature review and case study chapters describe the supporting theoretical and practical evidence, which is referenced within the implementation and discussion chapters. The final chapter gives the conclusion to this research.

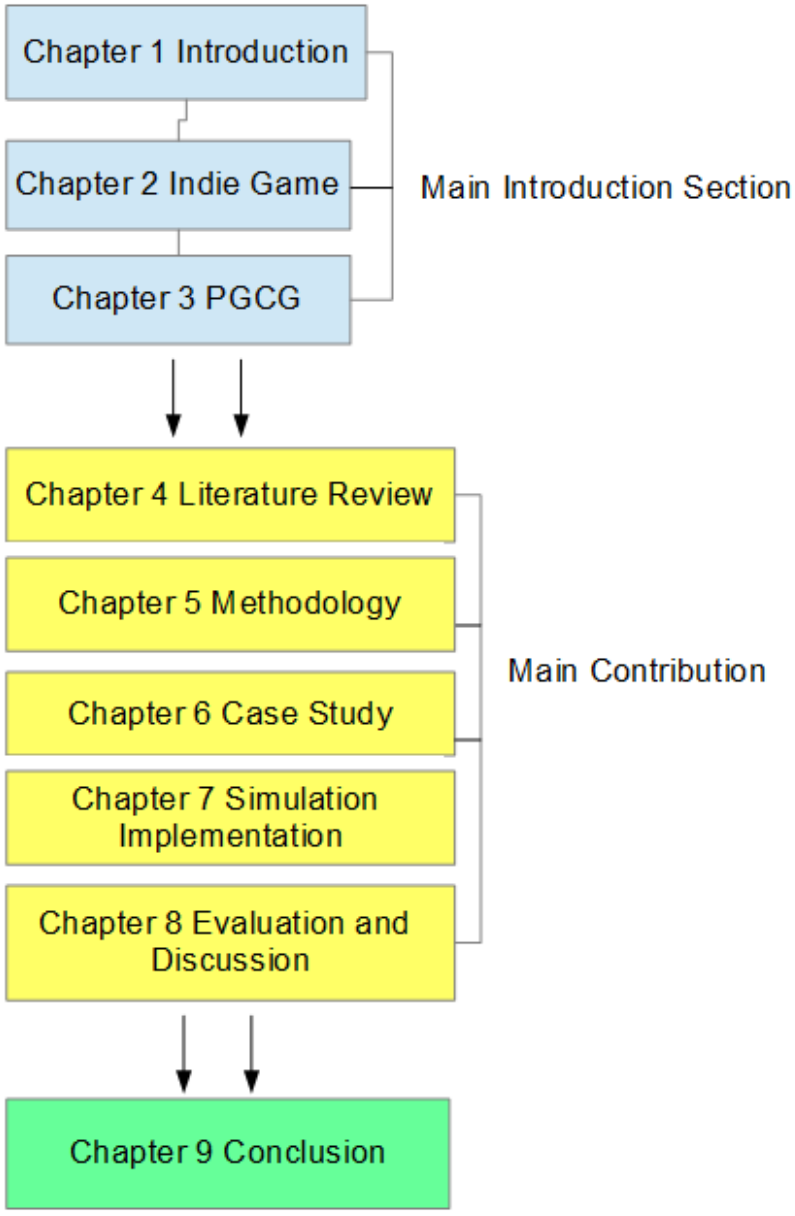

Figure 1.2: Overview of the research structure

Chapter 1 gives an overall description of the thesis. Chapter 2 provides an introduction to independent video games by illustrating their concept, history and recent titles which have used PGCG as the main technique. Differences between commercial video games and independent video games are compared. The development of indie games is provided as well as their characteristics, such as their low investment requirement and small development team. Popular independent game tiles using PGCG are listed and introduced within this chapter.

Concepts and background illustrations are helpful in understanding the literature review section and methodology section. Chapter 3 shows the concept and history of procedural

generation technique in video game development. Different PGCC algorithms are introduced. However, due to hardware limitations, only a few algorithms are used in real-time for game content generation. In particular, Perlin Noise is one of the most frequently used algorithms in the procedural generated game world. A detailed introduction to this algorithm is illustrated with appropriate diagrams and examples in Chapter 3.

These concepts and background illustrations in turn are helpful in understanding the literature review, case study and methodology sections. Chapters 4 to 7 form the main contribution sections of this thesis. Within the literature review, several papers and articles from relevant industry experts are reviewed. Hendrikx described the history of procedural generation techniques and possible procedural generated elements in a video game [20]. After understanding the possibility of PG in game development, the benefits that PG techniques bring to video games needed to be studied. Yannakakis and Togelius provided detailed research about how PG affects the other aspects of gameplay [63]. Before actually diving into demonstrating programme development, having the knowledge of how to generate various game environments with different algorithms is necessary. Many articles provide information about various different methods of achieving procedural generated environments or other elements [2, 8, 9 and 10]. Although not all of them related to video games, the algorithms that are applied in them have inspired this development approach.

Chapter 5 gives detail discussion about the use of methodology in this research. The reasons to use combination of case study and simulation approach will be presented.

Chapter 6 contains the case study. In this chapter, detailed analyses of technical aspects are based on the most successful independent game “Minecraft”, which uses PGCG techniques for most of its game content. The study expands on how PGCG is used in Minecraft and how PGCG affects Minecraft. The primary algorithms that “Minecraft” used to achieve a procedural generated world are explored. Outcomes from this section support the development progress and discussion sections.

Chapter 7 is the methodology section. The goal of this section was to design and implement a 2D game demonstration which could generate a town as a game environment using PGCG techniques. The game prototype is based on the PGEG framework, which is the main product of this thesis. The whole implementation progress from scratch is illustrated and explained. The

main architecture of the framework is shown first. The functions and modules that use the procedural generation technique are explained with text and diagrams. Explanation of the algorithm implementation is provided with charts and text in order to show how a procedural generated world is built. The execution results are shown with screenshots and numerical data. Data for evaluation is collected and displayed using text and diagrams.

In Chapter 8, various performance evaluations focus on different aspects of the system. Using the evaluation results, the advantages and disadvantages of the PGEG framework among independent development is discussed. While discussing the evaluation data from the simulation, the theoretical and practical findings from chapters 4 and 5 are referenced in order to provide supportive evidence for the outcomes.

Finally, a conclusion covers the important discoveries of the research. It includes the disadvantages and benefits from PGEG in independent video game development in particular situations. The possibility and functionalities brought by the PGEG framework are discussed as well. Also, recommendations for future developments are pointed out.

Chapter 2

Independent Video Games

2.1 Introduction

Independent video games (IVG), also known as indie games, have become a popular category in recent years. However, IVG has existed throughout most of video game development history. In association with on-line publishing platforms such as App Store, Google Play and Steam, more and more indie developers have had the opportunity to display and sell their products to millions of users. How does one define "indie" as a game tile? To classify a game title as "indie", "mainstream" or "AAA" is always difficult. The definition of an independent game can be made according to many aspects, such as style, popularity, cost, production scale and so on. Video games, as a new form of electronic art, can be defined using similar rules as for "indie movies" and "indie music". Newman [33] proposed that the key factor in defining indie media is whether it is different from mainstream media. Compared to mainstream media development and published structures, indie media always sits on the opposite side to the mainstream production and distribution processes. For video games, the same judgments can be followed. Flexible production and publication structures partly define "Indie" games compared to mainstream game companies [25]. Small game studios may have less access to professional tools such as Visual Studio, Maya and Beast, which have a high subscription price. As a result, indie game companies choose to use open source tools and coding languages, which bring disadvantages compared to AAA developers.

Another characteristic of independent games is that their titles are released mainly on digital distribution platforms. Mainstream game companies work with publishers and put their products on shelves in retail stores. The rise of digital application stores like Steam, App Store and Google Play offers a great opportunity to indie game developers. More than 80% of the purchase price goes to the developer in digital distribution, while only 17% to the retailer [15]. The communities that these digital platforms create also bind the users and indie developers. Independent developers adapt feedback from the community and implement popular features in the next update. In the documentary film "Indie Game: The Movie" [35], the developer of "Braid",

Jonathan Blow, points out the spirit of indie games compared to mainstream companies: “Part of it, is about not trying to be professional….What those companies do is create highly polished things that serve as large of an audience as possible. The way you do that is by filing off all the bumps on something …. The creation of this highly glossy, commercial product is the opposite of making something personal”. The culture of independent game production is more about focusing on player communities than the development labours [11]. The indie development circle seems to have its very own characteristics [61].

2.2 History

With the rise of digital distribution platforms, there has been significant growth in the popularity of indie games in the last 10 years [12]. However, before the growth of video games in the 1990s, there was no large scale publication industry for video games. Game developers in that era could be considered independent. “Shareware” or “demo ware” is the name for those independent game developments [38]. During the explosive growth of video games after the 1990s, more and more publishers and investors took part in the video game industry and the established industry of video game was created. Facing the disadvantages of techniques and investments, indie games were not able to take their place in the market and gained only a small audience. In 2003, the release of the largest PC game digital distribution platform Steam [51] gave a strong boost to the indie games industry. Seventy-five percent of the income from purchasing games in Steam goes directly to the developers [35]. Later, the Greenlight system of Steam allowed indie developers to upload their games and let the community decide if the games should be on the shelf by voting. Meanwhile, the rise of smartphones and mobile application stores has strengthened the indie game industry. In recent years, digital distribution platforms allow another benefit for indie game developers in that they can sell their games in beta or alpha phase. Similar to the mode of online crowd-funding games, the developer updates features constantly while earning money from the players in support of development. Since the success of Minecraft [32], indie games have become a hot spot in the video game industry. The growth of indie games, which has been explosive following the success of Minecraft and various other platforms [17], motivated this research thesis.

2.3 Indie Games using PGCG

Since the procedural generated world brought great success to Minecraft, many independent

developers have tried to adapt the technique in their titles. Table 2.1 shows some recently released popular independent games that used PGCG to generated various game content. There are many more PGCG based indie games that are not listed in the table. Although these games have different gameplay mechanisms, many of them use PGCG to generate the game environment. From the style of the game environment, most of them are based on blocks that are generated by similar algorithms. It proves that procedural generation techniques are popular in generating game environments.

Table 2.1: Recent popular games that have used PGCG

Titles	Release Date	PGCG Content
Kerbal Space Program	27 Apr, 2015	Missions/Contracts
Project Explore	20 Mar, 2015	Environment
Dig or Die	4 Mar, 2015	Environment
Stranded Deep	23 Jan, 2015	Environment
Instant Dungeonl	25 Nov, 2014	Environment
Dungeon of the Endless	27 Oct, 2014	Environment
Banished	18 Feb, 2014	Environment
Starbound	4 Dec, 2013	Environment
Legend of Dungeon	12 Sep, 2013	Environment
The Binding of Isaac	28 Sep, 2011	Environment
Terraria	16 May, 2011	Environment

Source: Steam [51]

There are reasons that indie developers choose PGCG to generate the game world in their products. Greatly reducing human resources during development progress can be considered the most important feature of PGEG. However, a usable PGEG framework is highly demanded within the indie game industry. This is also the motivation of this research thesis. The effect of PGCG on independent game development require further explored. The algorithm used to generate the game environment will be introduced and discussed in the following chapters in this thesis.

2.4 Summary

This chapter introduced the basic concepts and definitions and the history of independent video games. Similar to other forms of indie art, indie game developers are usually small in scale and lack investment money compared to mainstream developers like Valve, Bioware and DICE. Different from mainstream titles, the content within indie games usually represents the

perspectives and personality of their developers. Unique and creative gameplay mechanisms and stories are often found in indie games. Although this form of game development appeared earlier, the rise of the digital distribution platform actually has given the opportunity to millions of indie developers to release their titles to global audiences. List of recent popular indie games which used PGCG is illustrated.

Chapter 3

Procedural Game Content Generation

3.1 Introduction

Since the invention of computers, finishing tasks effectively and automatically is what computers are programmed to do. However, in many cases, users have to create content manually. For example, if an artist is going to create a 3D model of an apartment, he has to manually build every wall and material texture for the building. But on the other hand, if the computer can automatically generate an apartment following the artist's style rules, the productiveness of development process can be greatly improved. Moreover, each apartment could be made unique by adding randomized content. Generating content according to particular algorithms rather than manually is procedural generation [39 and 54]. Procedural techniques are already used in many fields [62]. For video games, procedural content generation means elements of the game are generated after the application is executed, instead of loading pre-made resources. As creating video games involves much more experience than programming, the complexity involved in creating a game is high. Many video game productions require more resources in their game content, such as art, music and story, than actual programming [55]. Developers would benefit if procedural content generation could be involved during the development processes. With the development of the video game industry, developers and researchers have created many methods, including AI, mathematic and more to generate different kinds of content procedurally. Procedural game content generation (PGCG) covers a wide range of problems and methods relevant to the scenarios above. The increasing size and detail of the virtual worlds in the video game industry pushes developers to adapt PGCG during the development processes [44].

3.2 PGCG Algorithms

To achieve generating game content procedurally, many different algorithms are created by both practical developers and academic researchers to suit different situations. In generating the maze-like levels for video games, Ashlock, Lee and McGuinness [3] proposed "search-based" algorithms. In their research, comparison of multiple representations for mazes based on evolution algorithms was illustrated. The basic structure of the maze is generated by binary girds

or chromatic girds, which provide numerical values for the program to decide whether there is an open area or a block. A “fitness function” is responsible for checking connectivity or path length within the maze based on several pre-set checkpoints. Optimization for real time content generation had not yet been presented for this algorithm. Other successful algorithms that procedurally generated maze-like maps for games are recorded in a previous survey [56].

There are more algorithms for generating indoor environments (e.g. mazes and dungeons) for video games [18, 30] but, on the other hand, the capability to generate outdoor environments procedurally is more attractive and valuable. As mentioned in previous sections, the advancing game development techniques brought more complexity to the virtual world. While players can have a better gameplay experience, it takes more effort for artists and developers to create a virtual environment. For large scale outdoor environments, more detail means more human power is involved during the development process. Prachyabrued, Roden and Benton [41] proposed algorithms to generate a stylized 2D outdoor map which could be used in video games. In their research, they first used a data amplification technique (DAT) and random midpoint displacement (RMD) to generate a stylized 2D map with ocean, river, mountain and other landscape features. A basic map would be generated at the beginning showing the boundaries of the different landscape features. With DAT and RMD techniques, smooth coastlines and rivers can be created. Trees and mountains are randomly placed on the solid ground.

However, to gain smoother and more various biomes in a procedurally generated outdoor environment, a more effective algorithm is required. Perlin Noise [40] has been proved and used in many simulations and in practical game development [61, 13, 10, 23, 7, 2 and 46]. Perlin Noise is a procedural noise which was first used for procedurally generating the textures of fire, smoke and clouds. The function provides smooth and continuous results for two or three dimensions, which are frequently used in development. Researchers found that this algorithm could generate a two-dimensional array which is filled with smooth interpolated values. Each index in the array could represent a height value of height map in terrain generation. For example, the game Minecraft, which the case study section targets, uses multiple maps that are generated by Perlin Noise to build terrain and different biomes. Further information about how to use this algorithm in virtual world generation will be presented in the case study and simulation implementation chapters.

Normally, a single algorithm is not enough to generate the various elements in the virtual world. A combination of different procedural algorithms leads to a virtual world full of detail [13]. As mentioned in previous sections, PGCG was adopted in video games in earlier years. However, limited by hardware performance levels and game development techniques, not every aspect of a video game could be procedurally generated. Many games used PGCG to generate dungeons or maze-like levels which could be combined with simple rooms and paths. Moreover, most of them were pre-generated with no procedurally generated features in real-time. In recent years, the usage of PGCG in game development, especially among independent game developers, has become more and more heavy. Developers use PGCG to generate infinite outdoor environments in real-time in order to provide unique gameplay experiences for players. Moreover, adopting PGCG seems to reduce the cost for game developers, especially for independent game developers who are more sensitive regarding resources. Will PGEG bring benefits, disadvantages or both for indie game developers? The question will be answered throughout this research.

3.3 Summary

This chapter has given basic definitions of procedural game content generation techniques, which are based on specific algorithms to automatic generate game content after the execution of the application. Although PGCG has been used in the video game industry for many years, independent developers became popular on using this technique after the successful PGCG game Minecraft was introduced. Various algorithms have been proposed to suit different requirements in generating game content for particular game mechanisms. Perlin Noise has proved popular in procedurally generating open world environments in indie games. However, a combination of various procedural algorithms is the new popular way to create a virtual world full of detail. Information from this chapter provides an overview of the concepts used in the following chapters.

Chapter 4

Literature Review

4.1 Independent Games: Developments and Opportunities

Although the video game industry has existed for a long time, independent games only became popular a decade ago. Through the rise of the digital distribution platform, indie developers gained opportunities to compete with mainstream developers with their own titles. However, since the very beginning of the video game industry, all game developers were considered independent.

Parker [38] mentioned that, in the early years, the established industry and economic frameworks were not ready for the video game industry. Shareware and freeware were common forms of delivering game titles; "amateur game development" describes the situation. Before the distribution process became digitalized, amateur game development was limited by geographical location, coding languages and development tools. Ito [16] found that early Japanese amateur game development expanded a community around the tools that were used for a specific game type – role-playing games (RPG). Because of a lack of published information, the audience and type of early "independent" games were limited. Over time, after digital distribution platforms such as Steam and App Store proved to be commercially viable, great opportunities opened for independent developers. Guevara-Villalobos [1] and Westecott [58] pointed out in their research that the globalized mass market brought about by digital distribution platforms greatly affected the rise of independent games. However, without providing players with an attractive gameplay experience, having access to the global digital market would not necessarily bring advantages as any developer could release their products to the new market through digital publication. Compared to mainstream game titles that always include war stories and a cinematic experience, independent developers avoided investing large amounts of resources in developing the same experience but created unique and creative gameplay according to their personality. Being different from mainstream games and providing a unique game experience are two featured characteristics of modern independent games [21, 25 and 29] However, the truly explosive increase in independent games can be considered to have happened after the game Minecraft's [32]

release in 2009. Figure 4.1 shows the approximate number of indie games released on the biggest digital PC games distribution platform, Steam [51], before 2009 and after 2009.

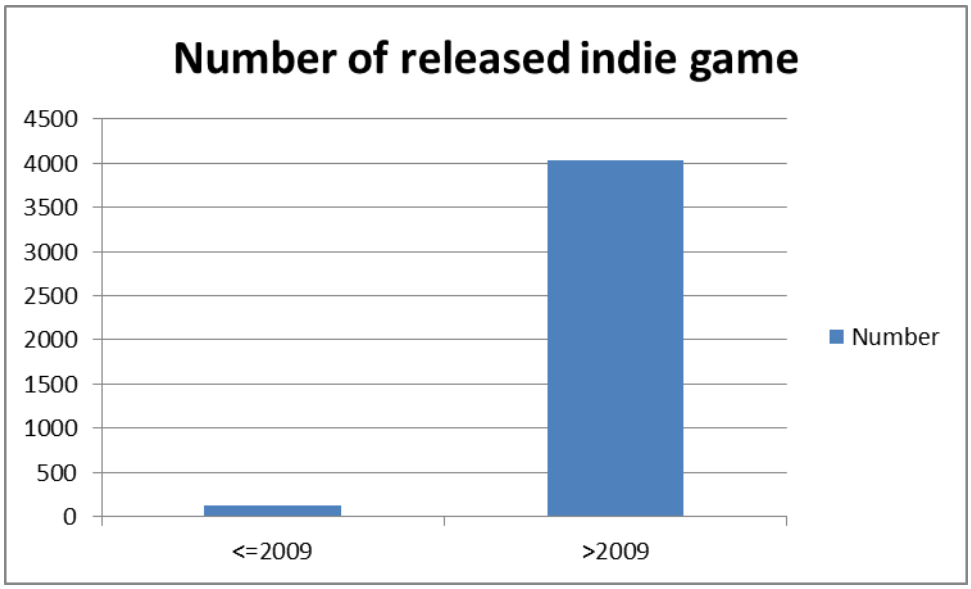

Figure 4.1: Number of indie games released on Steam before and after 2009

There was significant growth after 2009. More and more indie developers adopted similar techniques and tried to duplicate the same level of success as Minecraft [24]. This fact illustrates that with the association of digital distribution platforms, more and more indie games would be released to the global market in the future. With the help of light weight development tools and platforms, indie game developers will meet great opportunities that never exit before. But in the other hand, as video games have grown for decades, user requirements for both mainstream and independent titles will become more critical. Competitions between mainstream developers and indie developers would become more frequently and closer. As players demanded a larger world scale and higher production costs, PGCG seemed to be a good solution [14]. Many Indie games used PGCG as a main feature. The primary objective that PGCG could achieve in these games was to generate a virtual environment that was suitable for particular sorts of gameplay. However, there are many different algorithms for generating different types of environment for virtual space. How to choose or combine various methods is a problem among developers. Effectiveness and controllability within the development iterations become more and more important. Several studies about procedural environment generation will be reviewed in the following section.

4.2 Procedural Game Environment Generation (PGEG)

Before reviewing the various procedural algorithms needed to generate a virtual game environment, the game elements that can be created by procedural methods are discussed by previous researchers. Hendrikx, Meijer, Van Der Velden & Iosup [13] presented a survey of PGCG. By automating game content generation, PGCG could address challenges such as scalability and production costs in the modern video game industry. Game content is the most important factor affecting the gameplay experience for players. Generating content that is suitable for a particular art style and interesting features is difficult. In the research by Henrikx et.al, game content that could be generated procedurally was divided into six classes: bits, space, systems, scenarios, design and derived. Factors such as textures, buildings, terrain, road networks, and indoor and outdoor maps could be created by PGCG technique according to the research. Fortunately, these factors are key elements in a game environment. Textures could be used for colouring and detailing almost every element in the virtual world. Buildings as well as road networks have to be generated while a procedural urban city is required. The survey proved that it is possible to generate a virtual environment procedurally for video games based on current technology level. Particularly, for texture generation, a range of mathematical algorithms could be useful in developing various geometric patterns [59]. Under a limited set of rules, buildings could be procedurally generated. Generative Grammars (GG) were widely used in building generation [62, 27]. In road network generation, Smelik, De Kraker, Tutenel, Bidarra & Groenewegen found that L-system, agent simulations and tensor fields are used during the procedural generation process [45]. When creating indoor maps, dungeons and mazes were common types of indoor game environments. Togelius et.al [56] found out that a wide range of procedural algorithms used for creating dungeons and mazes were based on Pseudo-Random Number Generation (PRNG) and Complex Systems (CS) techniques. In order to generate realistic indoor spaces, some researchers proposed using a combination of PRNG and GG techniques [8 and 50]. Grid-based structures like height maps are usually used for outdoor environment generation. Semlik [44] and de Carpentier and Bidarra [5] found that terrain could be procedurally generated by various techniques based on PRNG and CS. The algorithms above are well researched standalone, but discussions of combined usage of various PG are very limited. These researches show their advantages in generating different types of elements. However, to generate a virtual world for video games, many of the techniques above would have to be used in

cooperation. Only one well designed procedural element cannot form an interesting world for video game player. The usage of combination of various procedural algorithms in virtual environment generation needs a further research.

In practical PGCG scenarios, various researchers proposed algorithms and designs for different situations. Smith and Mateas [48] used answer set programming (ASP) for procedural content generation when focusing on design space. In their research study, a demonstration application was developed to prove that ASP was suitable for creating reliable rich content generator rapidly. Similarly, regarding design aspects, experience-driven procedural content generation (EDPCG) was proposed in order to provide better performance in human-computer interaction (HCI) [63 and 6]. In each loop of the content generator, the player's experience would be modelled in order to provide parameters to the content generator to change the game level.

Several experiments based on various classic game maps proved that EDPCG could provide a personalized experience while saving labour spent on designing levels. Similarly, Raffe explored methods used in creating personalized procedural maps with evolutionary algorithms [42]. Smith et.al [49] explored the possibility of enabling new gameplay experiences through implementing a prototype 2D platformer game with procedural content generation. The highlighted feature in the research was that the player's decisions could affect future procedural generation in the game levels. Different from other forms of PCG based games, the study showed the potential benefits of combining player behaviours and procedural generative systems. These outcomes present that the generation of game level can be user-driven. It is smart that the application would change the level structure based on the player's behaviours. In this way, each player could face a unique gameplay experience as he is partly building the upcoming environments. However, the positive affect only appears in generating classic game types such as platformer or maze. The outputs of applying this technique in games with large open-world are not been researched yet. What could be predicted is that in generating large scale world, the process of user-driven generation would become much more complex. More parameters and tests need to be developed in order to form the world. Performance and gameplay experience would be main challenges.

Early PGCG research [40] mainly focused on what could be generated by particular methods and how the methods could work. However, nowadays, how to achieve a particular result and

have more control has become a hotspot for PGCG researchers [43]. Because PGCG grew in complexity, more and more advanced methods have been adapted to improve the interaction between developers and machines [47]. Moreover, user experience has been proved to be useful for providing various variables to control PGCG process to generate player-based results [26]. Providing more control for developers and receiving feedback from gameplay as generative parameters are the main objectives of modern PGCG. Moreover, in generating large scale virtual environment, the performance of the generation process is a critical criterion that affects the whole gameplay experience. Different from the previous research that generated small levels for playing, modern PGEG requires to generate a large open-world in real-time.

Regarding large scale virtual environments, Steinberger et.al [53] created approaches that could generate an "infinite city" by using parallel generation of architecture on the GPU [52]. In traditional methods, the CPU would generate different city shapes based on generative grammar, and the generated data would be stored and streamed to GPU for rendering. In their approaches, generation steps were put on the GPU, which could only generate and render a current view. Additionally, by combining a derivation tree and level of detail (LOD), the render speed was faster than in all previous work. The whole generation process was divided into several steps. The first step was to generate the layout of the city such as roads and districts by L-system algorithms. The second step was to generate hulls as 3D boundaries for buildings. The next step was viewing frustum pruning which ignored building hulls that were outside of the view frustum. The fourth step was to create specific various building types. The occlusion pruning step will removed the entry building if it was completely blocked visually by other buildings. The final step was to construct the remaining buildings.

Greuter et.al [10] presented an approach to procedurally generate "pseudo infinite" city in real-time. Similarly, instead of using a CPU to generate procedural content, a GPU based approach could provide better performance and controllability. In their research, a city layout was heavily simplified to be represented by square grids. The main objective was to generate a large number of various buildings while keeping performance high. Before generating the buildings, the algorithm would ignore buildings that were outside of the view frustum. After the view frustum filling phase, the form and appearance of each building would be determined by seeds that were generated by the pseudo random number generator (PRNG). The seeds would be stored

in a hash map. Combining position coordinate values and hashed seed values, a unique seed for building generation was provided. Buildings were generated from the top to the ground. Pillars of various heights would be generated, and a combination of pillars represented a building. Generated buildings would be stored in OpenGL's display lists for faster future redraw. These algorithms were also recorded in Kelly's survey [22] of procedural techniques for city generation, which introduced further interesting methods.

Other research on the procedural modelling of cities, by Parish & Müller [37], focused on generating a realistic transportation network that was affected by population density and environmental elements. The whole procedural system in the research was based on L-systems, which were famous in generating plants and networks using generative grammar. According to the population density and land-water boundary maps, the system generated different types of transportation networks and divided the terrain into sections. Moreover, buildings were also generated by the L-system combined with basic texture elements. Two modified L-systems were used for two different generation steps: one for street creation, another for building creation. In the street creation phase, the L-system was modified to dynamically change parameters while encountering various situations in the environment. In generating buildings, five consecutive steps were designed. It firstly generated the bounding box for the building, and more detailed structures were generated by the L-system at each step. The whole generation process took minutes to complete. In the procedural generation of roads, Galin et.al [9] proposed a way to generate complex and realistic road networks based on current terrain. The automatic method was based on an algorithm that found the shortest path between weighted nodes. While connecting the initial and final points, different parameters from the terrain and natural obstacles affected the trajectory of the road in order to give realistic road patterns. Similar algorithms were also proposed by Marteck [28]. However, though these methods guaranteed the realistic of the procedural content, the complexity of calculations made them impossible to gain enough performance in real-time generation.

Smelik et.al [44] proposed that cooperation between procedural generation and manual editing would improve development effectiveness. Generating virtual worlds completely based on procedural algorithms is fast and simple, but there is a lack of user control and it is difficult to integrate. Instead of procedurally creating the whole world, they provided tools for developers to

decide which areas of the world should be generated. Another feature of the workflow in their modelling framework was that the virtual world was divided into several layers: earth layer, water layer, vegetation layer, road layer and urban layer [46]. After creating each layer following a particular order, all the layers would be combined together to provide information for actual rendering. Multi-layer structure not only simplifies the structure of generation process, but also creates spaces for further performance improvements.

In procedural terrain generation, valued noise maps are commonly used for providing information for the generator. Lagae et.al did a survey of various procedural noise functions [23]. Perlin Noise was found to be reliable in many procedural terrain generation studies [34]. Parberry [36] proposed a method that combined traditional algorithms and real-world elevation data to procedurally generate infinite terrain. The approach used Perlin Noise as a basic procedural algorithm to generate seamless terrain data. After describing the basic mechanisms of Perlin Noise, the research introduced methods for analysing geospatial data. Based on the results from the geographical data analysis, height distribution could be scaled and applied to various parameters in Perlin Noise to generate natural and interesting terrain characteristics. Instead of letting the system control the generation results automatically, the approach allowed designers to apply interesting geographical characteristics into the virtual world during the procedural generation process. Further, Perlin Noise was used to generate boundaries for various virtual environments such as cities in Wijgerse's research [60].

It could be found that recent researchers are not focusing on how to design an algorithm to generate specific environment, but focusing on how to improve existing algorithms' performance to gain better results. As mentioned by previous researchers [53], a real-time usage such as video games, performance is a critical factor that affects the user experience from the PGEG system. Visual latency and application freeze would cause disasters to real-time applications. Three researches [10, 52 & 53] mainly focus on improving performance of PGEG process by transferring some workflow from CPU to GPU. The reason is that the growing scale of virtual world makes performance become the primary challenge in real-time application. How to synchronize the procedural generation process and render process is great challenge for indie game developers who intends to use PGEG is primary feature in their games. Not matter generate environment by using noise maps or L-system, large amount of data needs to be

created and converted in real-time. As a matter of fact, performance could be considered one of the most important criteria in modern PGEG game development.

4.3 Summary

This chapter has organized and introduced some previous research relating to independent game development and PGEG. However, there are many more studies in this field that could be further reviewed. Table 4.1 summaries key features from the several previous studies that have been reviewed in this chapter. Many related works have proved the possibility of using PGCG to create virtual game worlds, and the topic has become more and more popular in recent years with the rise of indie games. Though various algorithms and methods have been developed for generating game environments procedurally, proposing a suitable PGEG based framework in independent game development is rare. Moreover, regardless of what types of PGEG techniques are implemented, performance is considered to be the most important factor throughout many previous studies. As most researchers have focused on proposing specific algorithms for particular usage, it is beneficial for this research to design a PGEG framework that makes use of various procedural methods while providing high performance.

Table 4.1 Summary of related work on PGEG

Author	Year	Main Algorithm/Technique	Output content
Steinberger et.al	2014	Using GPU to calculate procedural process	Effective rendering of large urban city
Greuter et.al	2003	Using GPU to calculate procedural process	Generation of infinite city
Parish & Müller	2001	Modified L-Systems	Generation of road networks and buildings
Smelik et.al	2010	Combining procedural methods and manual layering	Generation of large scale terrain with user controllability
Smelik et.al	2008	Multi-layer procedural generation	Large scale outdoor environments
Parberry	2014	Tweaked Perlin Noise parameters with real-world geographical data	Infinite terrain with realisitc characteristics

Chapter 5

Research Methodology

5.1 Introduction

The research is carried out using the case study and simulation approaches. Throughout these two methodologies, the effectiveness of PGCG techniques in independent game development is investigated. The case study focuses on the technical aspects of the famous PGCG-based indie game “Minecraft”. The game environment content based on PGCG is divided into four main classes in the case study: terrain, biomes, physiognomies and special structures. The algorithms that “Minecraft” used for generating these factors are introduced and discussed. Clear diagrams are provided to gain better descriptions. Information from the case study can become supportive evidence for the research question. Also, the case study provides inspiration for simulation development. Figure 5.1 illustrates the methodologies adopted in carrying out the research work.

In the development phase, the target of the implementation was to develop a game prototype based on the structure of the PGEG framework. The application was developed in C++ programming language. For providing graphics and operation system interfaces, Simple Fast Media Library (SFML) and OpenGL were used as an external development library. The game world is under two-dimensional coordinates in order to gain a clearer illustration style. Several milestones are set up to achieve the simulation target. The first step is generating different terrain tiles. The second step is generating a road network. The third step is procedurally placing buildings according to the first and second steps. In this case, as the application is a prototype and the main objective is the implementation of the PGCG algorithms, elements such as terrain tiles, roads and buildings are represented by simple coloured blocks and images. Potential advantages of procedural content generation can found throughout the development and evaluation of the simulation program. To evaluate the game simulation prototype, various aspects including render performance, generation performance, visual issues and system memory consumption are evaluated throughout a series of experiments.

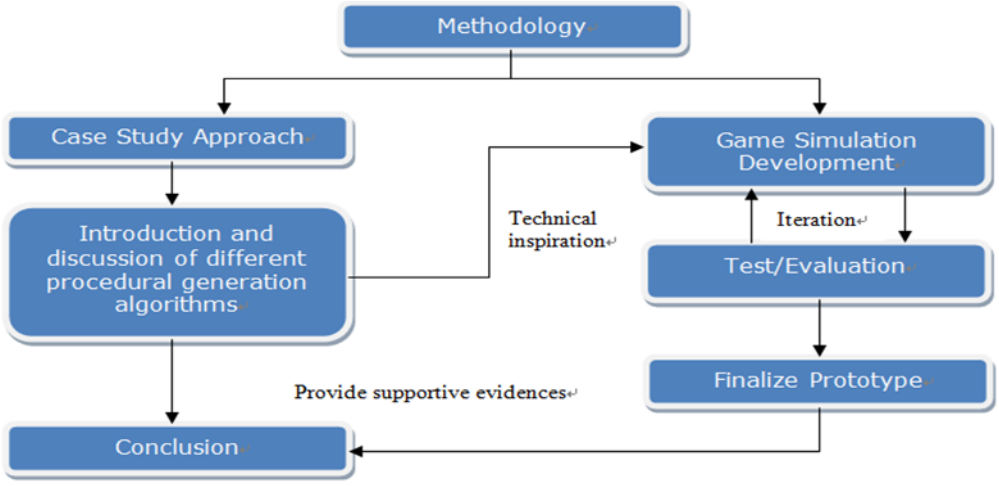

Figure 5.1: Methodology structure

In chapter 4, we found out that many previous researchers proposed various designs and algorithms to achieve different PGEG tasks. However, few of them are aiming to provide a PGEG based system for game developers especially for independent developers. This fact supports that it is valuable to have a research focus on designing a PGEG system for indie game development. Moreover, some citied articles showed efforts on improving performance of generating different procedural content as performance can be considered a critical factors during process of PGEG. This fact provides important information of the evaluation and discussion of the simulation approach in this research. Most of the evaluation test cases can be designed around the criteria 'performance' to justify the viability of the system more accurately.

As mentioned in the previous chapters, the game 'Minecraft' leads a technical heat within indie game circle: more and more developers are willing to use PGEG in their titles as 'Minecraft' produces an attractive sales record. Later, large numbers of indie games could be considered partly duplicators of 'Minecraft'. Selling 54 million copies made 'Minecraft' the most successful PGEG independent game. As a result, it is reasonable to perform an in-depth case study of this game among technical aspects. The case study will mainly focus on how 'Minecraft' achieve its large outdoor environment generations. System structure and relevant algorithms will be investigated according to developers' papers. The findings from case study could be used as supportive evidences during development process. However, it is impossible to investigate the implementation details of 'Minecraft'. Many indie developers intended to merge similar mechanism in their games. Proving the framework is capable to handle similar gameplay mechanism is important.

Additionally, this research is to purpose a framework for PGEG based games and indie game developers. However, the framework itself could not prove its availability or performance advantages. Theory and architectural designs are not persuasive enough to illustrate the value of the system. As the purpose of the framework is to provide an effective way to develop PGEG games, building a PGEG game prototype based on this framework is useful to demonstrate the features of the framework. Practically, the game prototype is 2D exploration game which allows player to run around a procedural generated town. Layers of terrain, road and buildings would be generated separately in order to prove that the framework is capable to use multiple procedural methods to form the final virtual environment. Lager scale of environment also challenges the framework's performance of rendering and resource management. Also, with the simulation approach, evaluations could be implemented easily and precisely by adjusting the various parameters of the game prototype. For example, switching different render modes in the simulation provides a way to test the render performance of the framework.

After gathering data from evaluation sections, discussion could be performed regarding to factors from literature reviews, case study, simulation development and evaluation results. Whether the framework can use various algorithms from literature reviews will be discussed. Moreover, as the implementation of 'Minecraft' is undiscovered. By this methodology, it could find out if the framework is capable to handle similar game mechanism for indie game developers. The technical investigation during case study can provide useful criteria to justify the results. During the discussion process, implications for independent game developers from this framework could be found. The reason the use such a methodology is to provide a more clear and reliable way to achieve the objectives of this research. For example, to design a tool, only theoretical justifications are not enough. It has to be performed well during field tests with different complexity settings. Also, it should be reliable in the most popular way of use: creating large scale outdoor procedural generated game environment in this case. The combination of case study and simulation approach would do great help to justify the features of the system.

For more justification, if case study and simulation approach are used as standalone elements in this research, there will be not enough supportive evidences to achieve the research target. For standalone case study of 'Minecraft', though the game itself could be considered a milestone of PGEG based indie game development process, the technical details of the game are

limited. It is impossible to clearly investigate the structure of ‘Minecraft’ application and algorithms as the authors keep large part of the implementation process as secrets. In this condition, a standalone case study could not support the objective: design and implement a PGEG based framework for indie game developers. In another case, a standalone simulation approach could partly support the research target. However, to justify the framework only by building a game prototype simulation and evaluation tests are not enough. As the framework is targeting indie game developer in improving future PGEG game development process, it should be proved to support popular PGEG game styles developments. If there are in-depth analyses of current popular PGEG indie games, the information could do great help to prove the viability of the framework. Additionally, it is valuable to know how other released indie games achieve PGEG in their process. Inspirations of the framework development could be found from practical case study. In this case, the combination of case study ‘Minecraft’ and simulation development is a good methodology to complete the objectives of this research.

5.2 Summary

This chapter discussed the research methodology adopted to carry out the thesis work.. The reasons to use combination of case study and simulation approach are discussed. Whether to use combination methods or standalone method has been justified. A structural guidance of the research process is provided for better understanding of the following chapters: case study and simulation implementations.

Chapter 6

Case Study: Minecraft

6.1 Introduction

Minecraft [32] becomes extremely popular, as it has an infinite world that is created by procedural generation technique. While gamers are crazy about the game, many educational organizations also use this game as a teaching tool to build and activate their students' teamwork and creativity. The surprise success of Minecraft opened the gate for distributing independent games on digital platforms. Huge amounts of followers use procedurally generated environments in their games to achieve the same level of success as Minecraft. Exploring the super star of PGCG games is extremely valuable [5].

Minecraft, which is written in Java programming language, was first created as an Alpha version by Markus Persson in 2009 [24]. Before the released version was announced in 2011, the author delivered constant updates about adding new features to the game. Though the game sold millions of copies with little advertising, its inexpensive development cost could be sensed through its appearance (Figure 5.1). During the early versions of the game, only one developer created the core mechanisms and everything else in the game. As a result, the world in Minecraft is built in blocks and all the textures are drawn in "pixel style". But procedural generation techniques helped the developer create a unique infinite and detailed world for each player. The experience of exploration is what attracts millions of players to this game. Even among mainstream commercial games, the level of success could be considered one of the best. The techniques that the game adapted are the key features that led to its unique gameplay experience.

Figure 6.1: Screenshot of the world of Minecraft

6.2 Basic Mechanisms

This chapter will not explore much about the gameplay systems in Minecraft, but will introduce the procedural mechanisms under the surface. The core content of the game is to let players explore and modify the procedurally generated world with their creativity. Focusing on the PGEG technique, the system provides many interesting subsystems to support the core features. When the player first creates a new world, the environment around the player is generated. The generated parts are stored on hard disk and are loaded if the player enters the same area. Though the developer's blog has not given detail about the implementation of the read and write system, it is clear that the input and output systems are running on another CPU thread in real-time, as there are few latencies during gameplay. As each block of the world can be stored and loaded in real-time, the game allows players to destroy or create any blocks in the world. This is the core feature of the game that allows player builds what they want by combing various types of blocks. To improve the gameplay experience further, different types of "mines" are placed during the procedural generation process. Moreover, different biomes such as trees, grasses and groups of animals are distributed on the surface and in the underground caves. Players can collect material from the elements above to create more content. Though there are many technical aspects in Minecraft that could be studied, such as a diurnal cycle system and lighting system, this chapter will focus on the procedural environment generation aspect, as it is the most highlighted core feature.

6.3 Procedural Environment Generation

Generating environments automatically is the most important feature in Minecraft. Analysis of generating terrain, biomes and villages will be presented. However, deeper analysis is difficult as the developers keep their secret formulas and only illustrate basic information on their development blogs.

Firstly, the core algorithm used to generate infinite smooth terrain is Perlin Noise. Originally, Perlin Noise was developed by Ken Perlin in 1985 and was used as a source of smooth random noise for procedural texture generation [40]. The reason for using Perlin Noise as the noise value generator is that the distribution of normal generic random noise is too rough for terrain. Figure 5.2 shows the difference between normal noise distribution and Perlin Noise distribution.

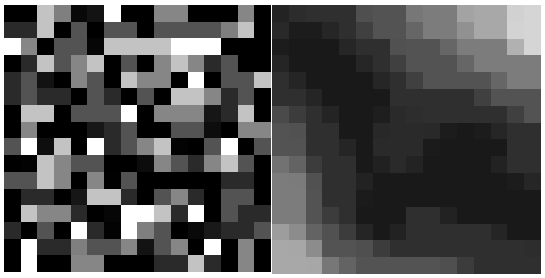

Figure 6.2: Comparison between generic random noise (left) and Perlin Noise (right)

The 2D Perlin Noise (Figure 5.2 right) is similar to the height map in geographical terrain generation. Normally, each pixel value in the noise map is used as a height value for the vertex in terrain generation. Formally, the result value R is within -1.0 and 1.0: $R \in [-1.0, 1.0]$. In 2D space, every coordinate (x, y) for which x and y are both integers, the function gives a random value and gradient value to make it a gird point. These chosen points divide the whole space into quads. The given values of these points are output to the final noise map. For those points within quads that are generated above, a grey scale value is interpolated according to the closest four gird points and their gradient value. Then a process called turbulence is used to add noise values at different frequencies and amplitudes [40]. Amplitude is multiplied by persistence and frequency is multiplied by lacunarity from one octave to another. “Octave” means noise at the same frequency. In terrain generation, noise value $z = f(x, y)$ is multiplied by a suitable value (based on the application’s needs) in order to scale the original value to the appropriate height value.

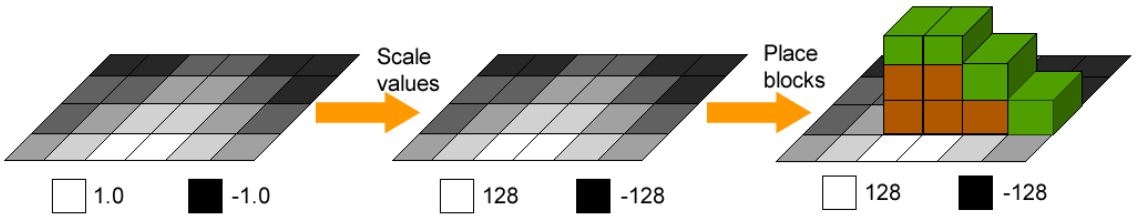

Figure 6.3: A simple example to explain terrain generation in Minecraft

To reduce complexity, Figure 5.3 shows how Minecraft creates its terrain with blocks according to the Perlin Noise map. In the first phase of generation, a noise value map where the value range is from -1.0 to 1.0 is generated. In order to gain controllability and better results, the noise value is scaled to [-128, 128] by multiplying a suitable variable. After the final height map is generated, the system places various stacks of blocks according to the value z = f(x, y) coordinated in the noise map. Each block type is represented by an integer ID and the world is stored in a three dimensional container in order to execute read and write operation easily. Within the container, positions that do not have terrain blocks are considered to be air. If an air block's noise value is smaller than 0 it is considered to be water. Under these mechanisms, basic formulation of a world is generated procedurally. Based on this basic algorithm, developers can also add many other modifications as secret formulas in their development to create more interesting features in the terrain. Elements like caves and valleys are implemented during the continuous update patches.

Having the basic terrain formation, many environmental features are added. Various physiognomies are distributed according to another layer of Perlin Noise map. The z = f(x, y) value from the noise map is used for representing different physiognomies. For example, $z \in [-1.0, -0.5]$ represents desert, while $z \in [0, 1.0]$ represents grassland. In particular, when point (x, y) in the second noise map represents desert, the surface block which positions (x, y) on the terrain is replaced by a sand block. After repeating the same operation for each surface block, an interesting physiognomy distribution is generated (Figure 5.4). As Perlin Noise is a pseudo random algorithm, by inputting the same seed at any time, the same terrain appearance is generated. Using the same techniques in generating physiognomy distributions, various biomes can be placed around the virtual world. Different types of plants are placed according to their corresponding noise value and the height value of the terrain. Also, animal groups, such as horse, pig, sheep and cow, can be placed by following the same rules above. Combining multiple

environment layers, an attractive virtual world is created for players to explore. However, detailed implementation and modifications are kept secret by the developers. But throughout the developers' information, it is promised that Perlin Noise can be used for procedurally generating virtual environments with rich elements for video games.

Figure 6.4: Screenshot from Minecraft showing different physiognomy distributions

Additionally, as a continuous infinite virtual world is generated, allocating the whole world in memory in initialization phase is impossible. Fortunately, the characteristics of the pseudo random generator let the system output the same result for the same position when the same seed is input. In Minecraft, a chunk system is used to store the world both on memory and hard disk. The overview mechanism of the chunk system is that the world can be divided into different chunks. Each chunk is represented by a three dimensional array. The current location of the player is the central chunk. The groups of chunks around the central chunk are generated at the beginning. While the player is moving, if a chunk exceeds the radius of the central chunk, it is released. At the same time, the new chunk that enters the valid radius is generated in real-time. An infinite world can therefore be explored on-the-fly when the performance of the application is guaranteed.

Besides generating natural elements for the gaming environment, the system also procedurally generates dungeons and villages for players to explore. Figure 5.5 shows how a regular village looks like in Minecraft. In village generation, the system first searches for flat

areas near the player's position. When a valid area is found, the system decides whether to build a village according to the pseudo seed and other information. If a village is approved to be created, a well is first created as the starting point of the village. From the position of the well, paths are extended and directed based on probability. Pre-designed buildings and farms are randomly placed along the path. However, each village must have a black smith and a church under the system's rules.

Figure 6.5: Screenshot showing a generated village in Minecraft; a well is located where the two paths cross

Though many environmental factors are procedurally generated in real-time, the application runs smoothly in normal situations. However, when the render distance is set to 32 chunks away, obvious visually latency can be observed and the frame rate drops to 7 frames per second when generating new chunks. Table 5.1 shows a performance test with different render distance settings. The data is captured by in-game debug tools. A multi-threading system probably optimizes the resource management in order to improve the performance. Writing and reading generated chunks to hard disk files can create bottle necks in performance. Regarding the richness of the procedurally generated virtual environment, the performance is acceptable to most players.

Table 6.1: Performance test in Minecraft with various render distance setting

Render Distance	Frame Rate (Frames per second)	System Memory Consumption	Visual Latency
8 Chunks	**60**	**1832MB**	**None**
16 Chunks	**33 - 60**	**2241MB**	**6 – 10 chunks**
32 Chunks	**28 - 60**	**2405MB**	**Larger numbers of chunks**

6.4 Impaction of PGCG

The most significant advantage of procedural environment generation in Minecraft is that this technique requires less human power during the development process while it produces these unique large virtual worlds. As the world is not pre-designed, each time a player starts a new world, the feelings of repetition are much less than in traditional games. Pseudo algorithms also provide the game the ability to let players share their worlds among communities by simply uploading their seeds. The sale numbers show the success of its core feature: over 3 million copies were sold by the summer of 2011 [24]. Though the positive aspects that PGCG brings to Minecraft are clear, there are also disadvantages that come with this technique. As the algorithms are based on pseudo random generators, it is difficult for developers to predict the results. In early versions of Minecraft, many visual issues such as floating blocks and unnatural terrains appeared in the game world. Developers spent a lot of time modifying and testing the original algorithms to gain acceptable results. It shows that having control over PGCG methods requires developers to have strong programming and mathematical skills. For independent developers, professional skills are an issue compared to commercial development teams. Open-world adventure game like Minecraft could greatly benefit from PGEG, as it does not require handcrafted scenes to fit the story's or gameplay mechanism's needs. For traditional level-based games, the low controllability of PGEG would bring disadvantages without careful research and design.

6.5 Summary

This chapter presents a PGEG example called Minecraft. PGEG has allowed the game to gain massive success on the market as an independent game. The core algorithm for generating virtual environments in this case is Perlin Noise. An overview of the terrain generation process and biomes generation was provided. The performance tests presented that this technique's performance is acceptable in terms of the richness it brings to the pseudo world. However, though

both advantages and disadvantages are introduced in using this technology. Many core mechanisms of the system are yet undiscovered. For independent developers, this technique would bring great benefits for open-world adventure games. However, increased usage of PGEG techniques among indie game circles following Minecraft makes it an urgent task to propose a workable PGEG framework. It is important to design and simulate a lightweight but flexible framework that is capable of handling PGEG while maintaining good performance.

Chapter 7

Simulation Design and Implementation

7.1 Overview

Because PGCG, especially PGEG, has become extremely popular within independent game developers in recent years, this research proposed to design and implement a framework that could be used for generating large scale game environments while keeping performance high. The system will generate a simple 2D town to represent the result of PGEG. Grass, dirt and concrete would be the terrain types that are painted on the world. Also, roads and buildings would be placed around the world. To explore the generated world, users have the ability to control a“person” to walk around. This chapter first introduces the hardware and software configuration of the whole development process. Then an overview architectural design is given before presenting the details of implementation. The third section is the actual implementation description. Design details of multi-threading system, block system and render system are presented. The fourth section presents the final results of the simulation by providing data and screenshots. Finally, a summary section concludes the content within this chapter.

7.2 Hardware and Software Configuration

The whole development phase was executed on a desktop personal computer. In order to provide a smooth environment for developing the simulation approach, the hardware specifications of the development computer needed to be high performance. Table 6.1 shows the hardware configuration of the development platform.

Table 7.1: Hardware specifications of development PC

CPU	I5-3570 3.4GHz * 4 cores
RAM	16GB DDR3
Video Card	GTX 660ti 2GB VRAM
Hard Disk	7200 rpm drive

For software configuration, the Windows 7 64bit operating system was used. The program was written in C++ 11 programming language. The integrated development environment was

Visual Studio 2013 professional, which provides full support for compiling C++11 features. Several libraries were used to help the development process. OpenGL was the graphics library that provided video interfaces for the development. Simple and Fast Multimedia Library (SFML) was used to gain interfaces to access the operating system events and basic 2D vector math. Libnoise was the library that provided the functions to generate various types of noise, including Perlin Noise. All third party libraries are free to use for any purpose under their licences. Table 6.2 shows the software configuration of the development phase.

Table 7.2: Software configuration in development phase

Operating system	Windows 7 64bit
IDE	Visual Studio 2013
Programming language	C++ 11
Third party libraries	OpenGL, SFML, Libnoise

7.3 Architecture design

7.3.1 Overview

The objective of this framework was to provide an effective and simple solution for independent game developers mainly using various noise functions to procedurally generate large scale virtual environments. As a result, the performance of the framework is a critical factor in determining whether the framework is useful. The overall workflow of the system was to first put seed values into the procedural system. Noise functions and other generative functions are responsible for outputting values to form a noise map. Based on the noise value map, a virtual world can be created and stored in the memory under certain rules. Finally, the data that stored in the memory were rendered by the render system. Figure 6.1 shows the basic workflow of the system.

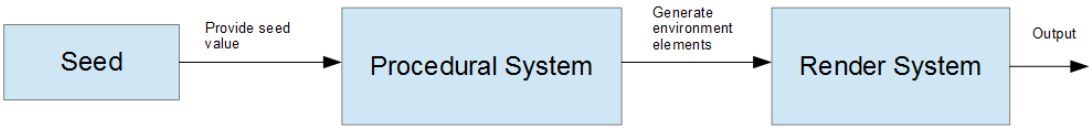

Figure 7.1: Basic workflow of simulation approach

Particularly, in the seed value generation phase, a seed value is inputted by the user or a default value is used. With a valid seed value, the procedural system will first generate a layer of the Perlin Noise map for terrain creation based on the original seed value. The second layer of Perlin Noise is generated after dividing the original seed value by an integer. The calculation is to make sure that the second layer of Perlin Noise is different from the first one when it is reproduced, as the second seed value is calculated from the original seed. The second layer of Perlin Noise is used to provide various building areas. Both layers of Perlin Noise values are scaled to gain more continuous results. Another procedural generation process is to generate roads. Similarly, seeds for generating roads are calculated from the original seed value. Finally, the three procedural generation results are combined and stored for future rendering. By using the original seed value as the starting point, the same environment can be recreated. This feature provides important support for structuring the render system. Figure 6.2 provides an overview of the procedural system.

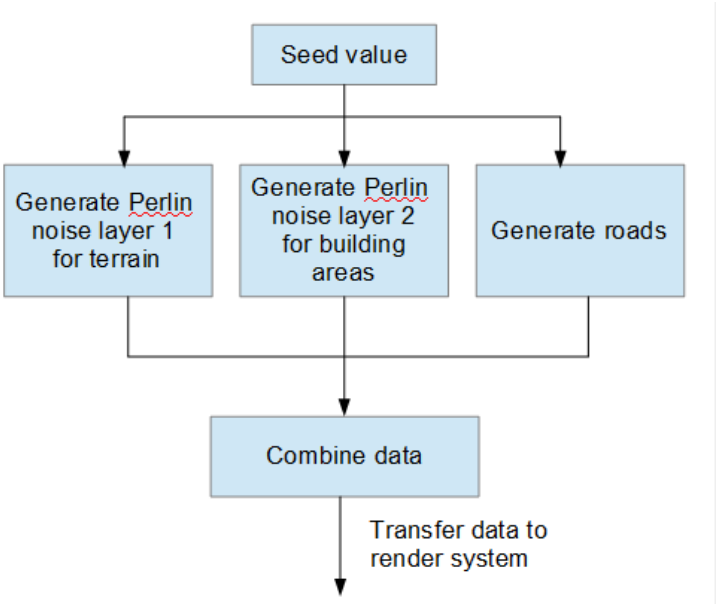

Figure 7.2: Overview of the procedural system

7.3.2 Block System

One of the purposes of this framework is to provide functions to generate large scale virtual environments for games. It is impossible to generate the whole world and store it in memory before rendering. The world has to be divided into sections and these sections are generated and rendered in real-time. In order to gain better performance, the render system contains a subsystem called the “Block System”. In the initialization phase, the block system allocates nine blocks to

store the upcoming world data. Figure 6.3 (a) shows the initial state of the allocated blocks. The red block represents the block that the player is located in. Each block contains a numbers of tiles in which to store environmental element data for rendering. Meanwhile, the player's view (border of the application window) is limited inside the red block or barely exceeds it. In order to maintain performance, the overall number of allocated blocks is nine in any situation. For example, the player is moving upward and exceeds the top border of the original red block in Figure 6.3 (b). The block in light red represents the previous player's location, while the red block represents the current player's location. As the player is going to see a new row of three blocks on the top in this case, three new blocks must be generated and rendered before the player's view reaches the area. From Figure 6.3 (b) and (c), we can see that the three light yellow blocks represent three blocks that are too far away from the player's current block location. The content of these three blocks can be erased and moved to the top as a container for three new environment blocks. All of these operations can be executed once the player enters a new block. No matter which direction the player is heading in, blocks that are considered too far away are erased and moved to new position. During the blocks' movement, at least three blocks must be moved, while five blocks are the maximum number if the player goes into one of the four corners. Certain rules and a coordination system are designed to make this system work.

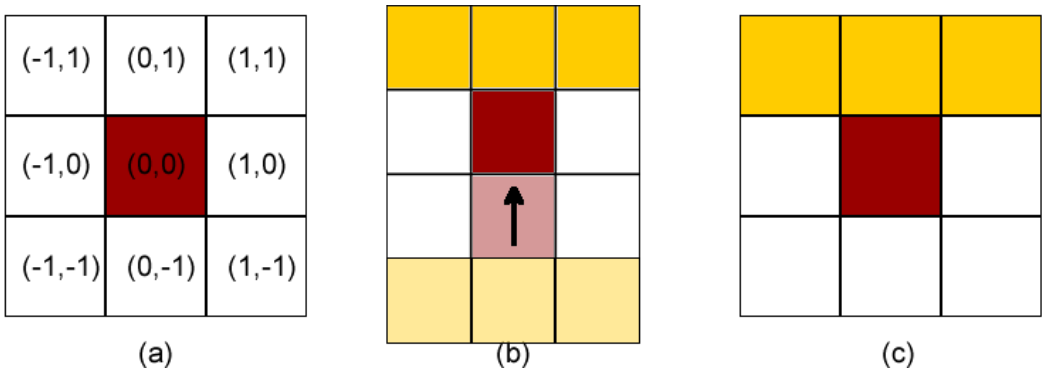

Figure 7.3: Working process of the "Block System"

Based on the original coordinate system in OpenGL, a new coordinate system called 'block coordination', which uses only integer values to represent the x and y axis, was designed to reduce operational complexity in the block system. Each block has its own block coordination. In Figure 6.3 (a), the initial state of the blocks is illustrated. The block which contains the player's current location is given the block coordinate (0, 0). Based on the original block, the eight blocks around

it are given corresponding coordination values. Once the player moves to another block, the system calculates which blocks are two units away from the player's block.

Let's take the current block's coordination (Xc, Yc) and one of the other block's coordination (X, Y). If the absolute value of (Xc – X) or (Yc – Y) is bigger than 1, the system marks block (X, Y) as available for regeneration. Additionally, each time the current block's coordination change, the system rebuilds a range that contains the maximum and minimum coordination of the blocks around the player's location. In order to discover which blocks are new and need to be generated, the system compares the blocks around the new central block to the range of the previous central block. If the blocks are not within the old range, the available blocks move to the new location and regenerate content from the procedural system. In this case, the number of blocks that needed to be calculated and rendered was nine during run-time. It provides a way to improve performance, while users will not notice the movement of the blocks and refreshment outside of their view.

7.3.3 Multi-threading Procedural System

Only reducing the calculations is not fast enough to generate environmental elements in real-time. The rendering system should not pause to wait for the procedural system to finish generation tasks. As a result, using multiple CPU threads is a good solution for rendering and generating elements at the same time [57]. The basic structure of the multi-threading generation and render system is presented in Figure 6.4. After the initialization phase, a loop called the "game loop" is entered and can only be exited after receiving a "game exit" event from the system. Within the game loop, two CPU threads are created to handle different tasks. The main thread first handles the system events such as user inputs. Then it changes the game data according to system inputs and game logic e.g. object movement and animation. After everything that would be seen by players is ready, the render system renders the screen. Each complete walkthrough of the game loop should be finished within a game frame. In order for each player to have a smooth experience for player, each frame should be finished in less than 33.3 milliseconds (30 frames per second). Meanwhile, in another CPU thread, the procedural system is waiting for the block system to update. Once the block system notices a player has entered another block and new blocks need to be generated, information is sent to procedural system in thread one and it regenerates blocks based on the corresponding coordination. As the erase and regeneration

operations are executed off-screen, users will not notice visual glitches during gameplay. The data race in multi-threading does not appear, as the procedural system is the only one editing the stored data, while the render system reads stored data and outputs to the screen.

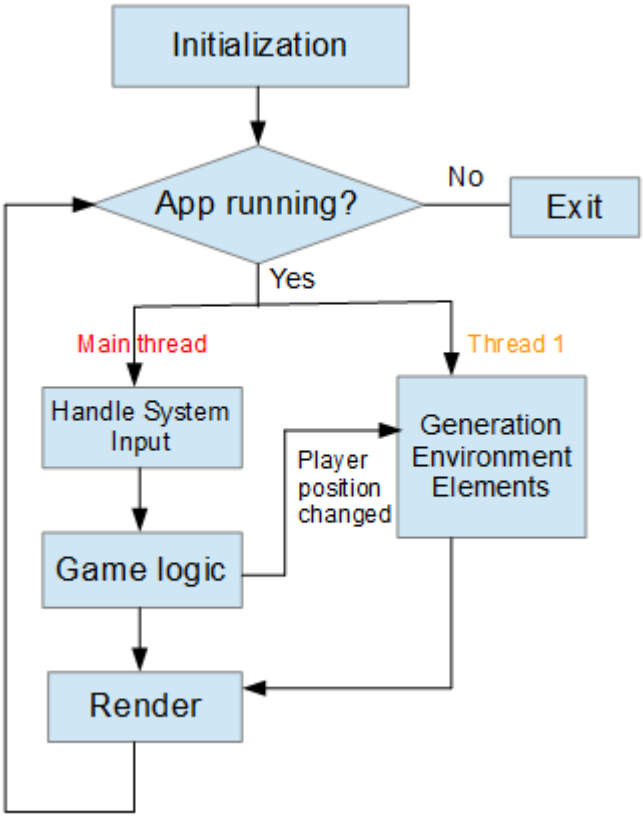

Figure 7.4: Basic workflow of multi-threading rendering in simulation approach

The architectural designs of core features in this framework are introduced below. However, there are more details about the mechanisms that should be described, so it would be clearer to introduce them along with actual implementation examples. In the next part of this chapter, practical implementation solutions are illustrated.

7.4 Implementation

7.4.1 Overview

Figure 6.5 shows the class diagram of the application that was generated in Visual Studio. “Block class” represents a block in Figure 6.3 and it stores a list of Tile classes in a container called a vector. Also, the Block class contains the procedural generation method for updating its own data. The Block system is implemented as the BlockManager class. It handles all the updates

of each block. The Tile class inherits from the GameObject class which is directly rendered by the render system. The Player class is self-explanatory and it handles all operations of the game character that are affected by user inputs. The SceneGameplay and SceneMainMenu classes inherit from the State class, as the system uses various states to control resource loading and releasing. The SeedHolder class holds the seed value that was entered by the user and transfers it from the main menu to gameplay state. In addition, enumeration values are defined to work with the system. STATE_ENUM holds values representing various states. TileType holds values representing various types of tile.

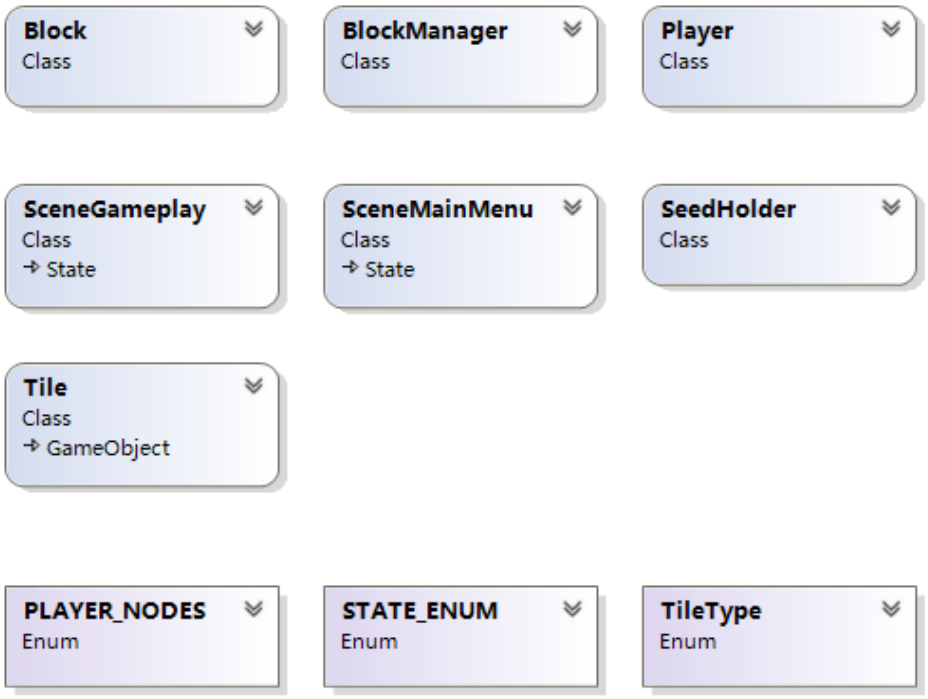

Figure 7.5: Class diagram from Visual Studio

Overall, the application is a finite state machine which changes to different states during run-time. Each state includes three basic steps: enter, run and leave. The "Enter" phase is similar to an initialization step; objects needed within the state are initialized and suitable resources are loaded. The "Run" phase could be considered to be a game loop, which is introduced above. The "Leave" step is executed before the system changes to another state. Useless resources are released during "leave" phase. Only two states are created in this simulation: the main menu state and gameplay state. The main menu state will change to the gameplay state if the "new game" button is pushed. The gameplay state will change back to the main menu state if "ESC" is pressed

by user. Figure 6.6 shows this mechanism. The system first enters the main menu state. If the "new game" button is pushed, the "leave" phase of the main menu is executed before changing to the "enter" phase of the gameplay state. When no change state events are triggered, the "run" state is executed as loop. The purpose of this system is to gain better control over structural and resource management. Each state manages its own resources and object creation and destruction. Any new state can be registered to the system by inheriting the root class "State".

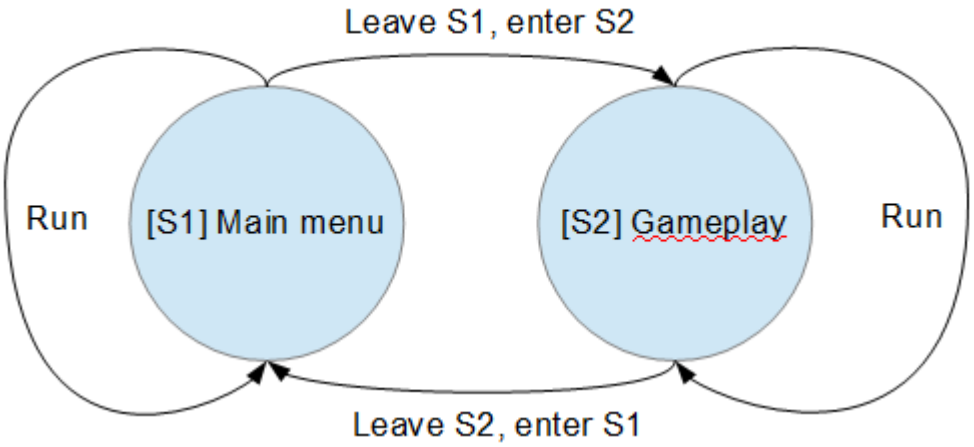

Figure 7.6: Two states in the simulation

7.4.2 Implementation of the Block System

The most important feature of this framework is implemented within the gameplay state. As mentioned at the beginning of this section, the Tile class represents the smallest unit of the procedurally generated environment. Each tile is rendered as a square textured object on the screen. It could be a tile for grass, dirt, concrete, roads or buildings which are based on the results from the procedural process. The Block class contains x by x tiles which form a larger square area. The variable x is and integer and is defined by BLOCK_SIZE while the size of each tile is defined by TILE_WIDTH. Both values cannot be changed during run-time and can be edited before compilation. Each "Block" defined here is equal to a little square in Figure 6.3. Tiles within a block are stored in a "vector" container from the C++ Standard Template Library (STL) in order to provide simple and powerful data operations. Moreover, the Block class contains methods for procedural environment generation, which is the core of the procedural system. Implementation of this system will be discussed later. According to the architectural design, a manager to handle all the blocks is created. The BlockManager class is responsible for initializing and updating each block in the world. Three "vector" containers are allocated in this class to store blocks, available

index of blocks for regeneration and block coordination of new blocks that need to be generated. Within the initialization phase of “BlockManager”, an additional CPU thread is created and attached to the function which is responsible for procedural generation. This CPU thread stands by in the background till the manager sends it the signal to begin generation. In order to make the block system work, in each game loop of the gameplay state, each player’s world coordination must be transferred to block coordination. A player’s world coordination is defined as “playerPos” and the player’s block coordination is defined as “playerBC”. The formula to get a player’s block coordination is: playerBC(x, y) = (playerPos.x / TILE_WIDTH * BLOCK_SIZE, playerPos.y / TILE_WIDTH * BLOCK_SIZE). Once the player enters a new block, the “extend” function of block manager is called to perform erase and regeneration operations, which are explained as the “block system” in the architectural design section. As the blocks are stored in a “vector” container, the index of each available block (the distance to the player’s block coordination is larger than one) is stored in another container. In the meantime, block coordination of new blocks that need to be filled is stored in a container as well. If the sizes of these two containers are equal, the system will have enough available blocks for erasing and regenerating new blocks. At this point, the manager begins to signal to the standing-by CPU thread by setting the Boolean value “_updateBlock” to true. Each available block receives its new block coordination and moves all the tiles it owns to the new position. World coordinates are calculated by reversing the formula used for getting the player’s block coordination. It uses the tile’s block coordination to multiply the product of TILE_WIDTH and BLOCK_SIZE to gain the tile’s world coordination. When all tiles are moved to the new position, procedural methods are executed. Figure 6.7 shows the structural relationships between the various classes that were introduced in this paragraph. Good management of environment containers helps improve the performance of the framework. However, lower level optimization is necessary to achieve performance when rendering large numbers of objects.

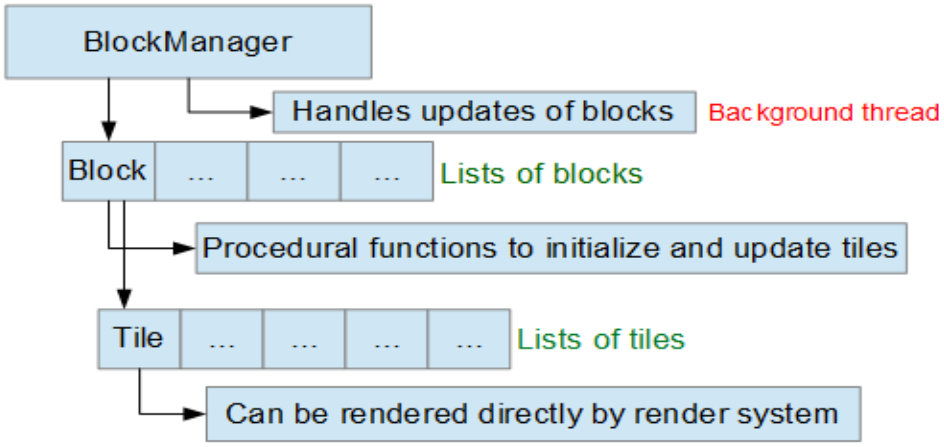

Figure 7.7: Structural relationships in implementation of the block system

7.4.3 Optimization of Render System

In traditional object orientation programming, each object has its own render function and can be called by the manager. When there are few objects on the scene, this method works fine. However, in this case, what the system does is to generate a large scale environment and future game logic calculations are considered as well. Calling rendering functions from system graphics API multiple times has hugely negative effects on performance. The performance of the framework can be greatly improved if the number of calling render functions is limited. As a result, a feature of OpenGL is used in this framework: Vertex Array. Figure 6.8 presents the basic mechanism of using vertex array to store the position attributes of each vertex in this framework. As a tile is a rectangle, it can be represented by four vertices. Each vertex is defined by three coordinate values: x, y, and z. As the simulation approach is displayed in 2D style, the z axis is used for representing different layers of objects. When a tile is allocated, the coordinate values of its vertices are stored in a vertex array. Instead of self-rendering, each tile knows the position of its vertices in the vertex array in order to perform a transformation in the future. According to this design, a structure called the “vertex attribute” is created to store the three core attributes of the vertex from an object. Table 6.1 shows members of the “vertex attribute” structure and corresponding OpenGL render functions. Position is defined by three floating point values x, y and z. UV coordination is used for mapping texture on the face of the rectangle. It is defined by two floating point values s and t. Colour is defined by four floating point values r, g, b and a. The vertex array stores lists of vertex attributes and the pointers of each attribute list are transferred to

corresponding OpenGL render functions. If a new object is created during gameplay, its vertices values can simply be inserted into the vertex array in order to be rendered. Once a vertex array is filled, the render system calls only four render functions from OpenGL in each frame: glVertexPointer(), glTexCoordPointer(), glColorPointer() and glDrawArray(). Compared to traditional method, if there are N objects are being rendered, the number of render functions called is four; while the one in traditional method is N. In this way, many objects can be rendered on the screen while limited function calls are made. The performance evaluation is illustrated in the next chapter.

Table 7.1: Vertex attributes and corresponding OpenGL render functions

Vertex Attributes	OpenGL Render Functions
Position (Coordination)	glVertexPointer
UV coordination	glTexCoordPointer
Colour values	glColorPointer

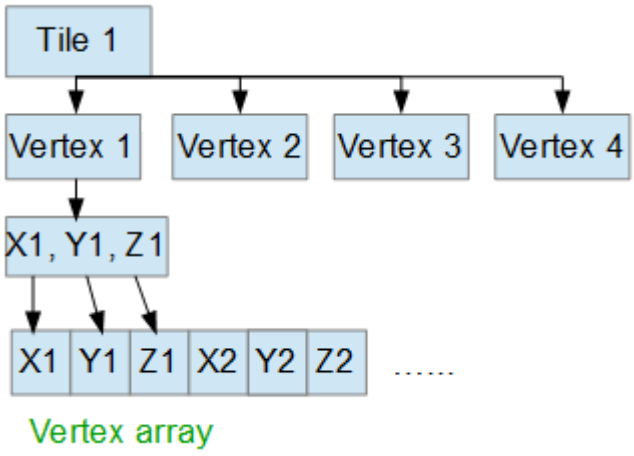

Figure 7.8: Usage of vertex array in this framework

7.4.4 Texture Management

To further improve the performance of the framework, another technique is used in texture resource management. In traditional real-time rendering, textures are stored independently. A texture must be bound before being mapped to corresponding faces. If there are 100 textures to be used on different surfaces, there will be many function calls to bind and release textures. One of

the methods to improve performance is to render objects that use the same texture together after binding the texture. As the framework batches geometry vertices before rendering, it is reasonable to use a texture atlas to manage the same types of textures. Figure 6.9 explains the structure of the texture atlas. From the meaning of "atlas", many pieces of textures (usually in the same category) are contained in one large texture. In this simulation approach, textures related to the environment are packed into one large texture. The system binds the large texture before rendering lists of vertex attributes, and then various textures can be looked up in the texture atlas by setting different UV coordinates. Though few textures are used in this simulation development, in practical development a texture atlas could greatly improve performance if hundreds of textures were going to be loaded and released.

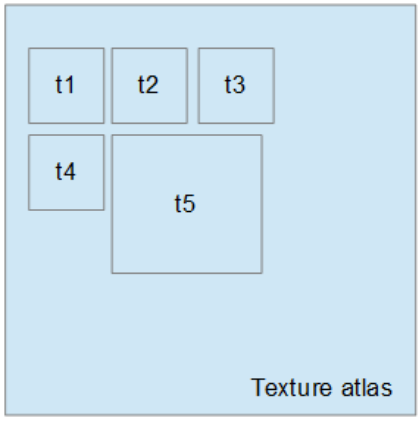

Figure 7.9: Example of texture atlas

7.4.5 Implementation of Procedural System

When the other systems are ready and optimized, the most important part can take place: implementation of procedural system. The purpose of this framework is to provide a way to generate large scale virtual environments procedurally. It can be considered that most of the virtual environment is static. Therefore, relevant environment tiles are only calculated when a block needs to be regenerated. Going back to figure 6.1, which shows the workflow of the procedural system; the seed value for pseudo random generators is taken by a seed holder class from the main menu and transferred to the procedural system during the initialization phase of creating the world. There are two phases in the procedural system: the initialization phase and the update phase. The workflows of these two phases are similar, except in the initialization step the

numbers of objects are allocated. In the first phase, two instances of Perlin Noise generator are created. One is used for terrain generation and the other is used for building area generation. Perlin Noise generators are provided by the Libnoise library, which wraps various noise functions for convenience usage while having the ability to tweak various parameters of the generators. After a tile is allocated, the world coordination of a tile is divided by the width of the tile to gain a coordinate for looking up the noise value from the noise generators. For smoother results, instead of getting the values directly from the noise generator, the system downscales the lookup coordinate by 10.0 before finding any noise values. The downscale value can be changed for best results in other situations. Once the noise values for terrain generation and building area generation are bonded to a tile, its parent block begins to select the type of terrains and buildings for the tile. Usage of these noise values is shown in Figure 6.9. Values in a Perlin Noise map are within the range from -1.0 to 1.0, including two boundaries. However, there is little chance that the noise value will exceed the range limitation. In the terrain noise map, values within -1.0 to 0.0 are considered concrete. The range from 0.0 to 0.5 is for dirt, while 0.5 to 1.0 is for grass. Values smaller than -1.0 represent concrete. Values bigger than 1.0 represent grass. Similarly, in the building area noise map, the range from 0.0 to 0.5 is available for residential buildings. The range from 0.5 to 1.0 is available for commercial buildings. Once the environment types are set for each tile, the corresponding textures are bonded to the tile and prepared for future rendering. In order to gain the best results, several parameters of Perlin Noise are tweaked based on visual adjustments. For the terrain noise map, persistence is set to 0.25 for smoother value interpolations. Smaller persistence results in flatter amplitudes in curves within the noise function. For the building area noise map, the parameter frequency is set to 40 to make building distribution more independent and realistic. A larger frequency value results in more value disruption in fixed units. More parameters can be adjusted to gain acceptable results for different purposes.

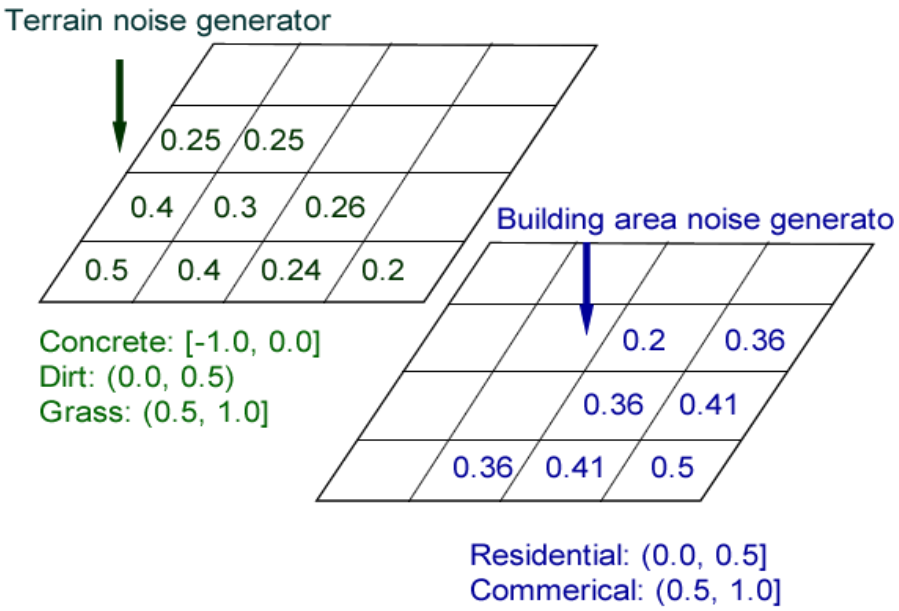

Figure 7.10: Usage of two Perlin Noise maps

Before combining both layers, another environment element needs to be generated. Many great researchers have purposed various algorithms for generating road networks. Some of them are simple and fast simulations, while others push the system to the limit for realistic simulations. In this research a simple and fast method of road network procedural generation is introduced. The objective is to show the possibility of using various procedural methods for generating a virtual environment within this framework. Also, seed values are needed for the generator in this stage. By combining the original seed from the seed holder and a different tile's world coordinates, unique seeds are created. For generating roads, the system first finds four nodes as starting points for the road networks for each block. More nodes would be found if more tiles were contained in a single block. The new seed values mentioned above are suitable for generating block coordinates. In order to keep this coordinate inside the block, a reminder from the generated value divided by BLOCK_SIZE is used as the final result for the block coordinate. Using the created nodes as a central point, roads will extend in four directions: left, right, upward and downward, based on a 50% probability. If the extended distance is greater than a specific number (e.g. four with 21 by 21 tiles for each block), there is a 50% chance that the road will grow to the left side or right side of its current direction. The road then grows to the edge of current block. In this way, simple but fast road networks are created.

Finally, the terrain map, building map and road networks are combined as one layer which is rendered as the final result. As all three maps are the same size and each tile is well aligned, combining maps is a simple task. In this simulation, road tiles have top priority, which means they will overlay any other type of tile. The building tile will overlay the terrain tile. While all content from the pseudo generator is based on the various transformations of the original seed, if the same seed is provided a whole world with the same elements can be reproduced. Using this framework, a large scale 2D virtual environment was procedurally generated. Screenshots from the simulation approach in the next section illustrate the various results from the different systems.

7.5 Results

This section shows various screenshots from the simulation to present the results from the actual execution of the simulation approach. The left diagram in Figure 6.11 shows console output after initialization of the framework. Every module is initialized successfully. The image on the right in Figure 6.11 shows the main menu state in the simulation. Two buttons are displayed on the menu. The “New Game” button is used to begin procedural world generation, while “End Game” is used for exiting the application. In the input box, 12345 is the default seed value.

Figure 7.11: Left: console output of initialization; Right: Main menu state

Figure 7.12: A world generated by seed “20152015”

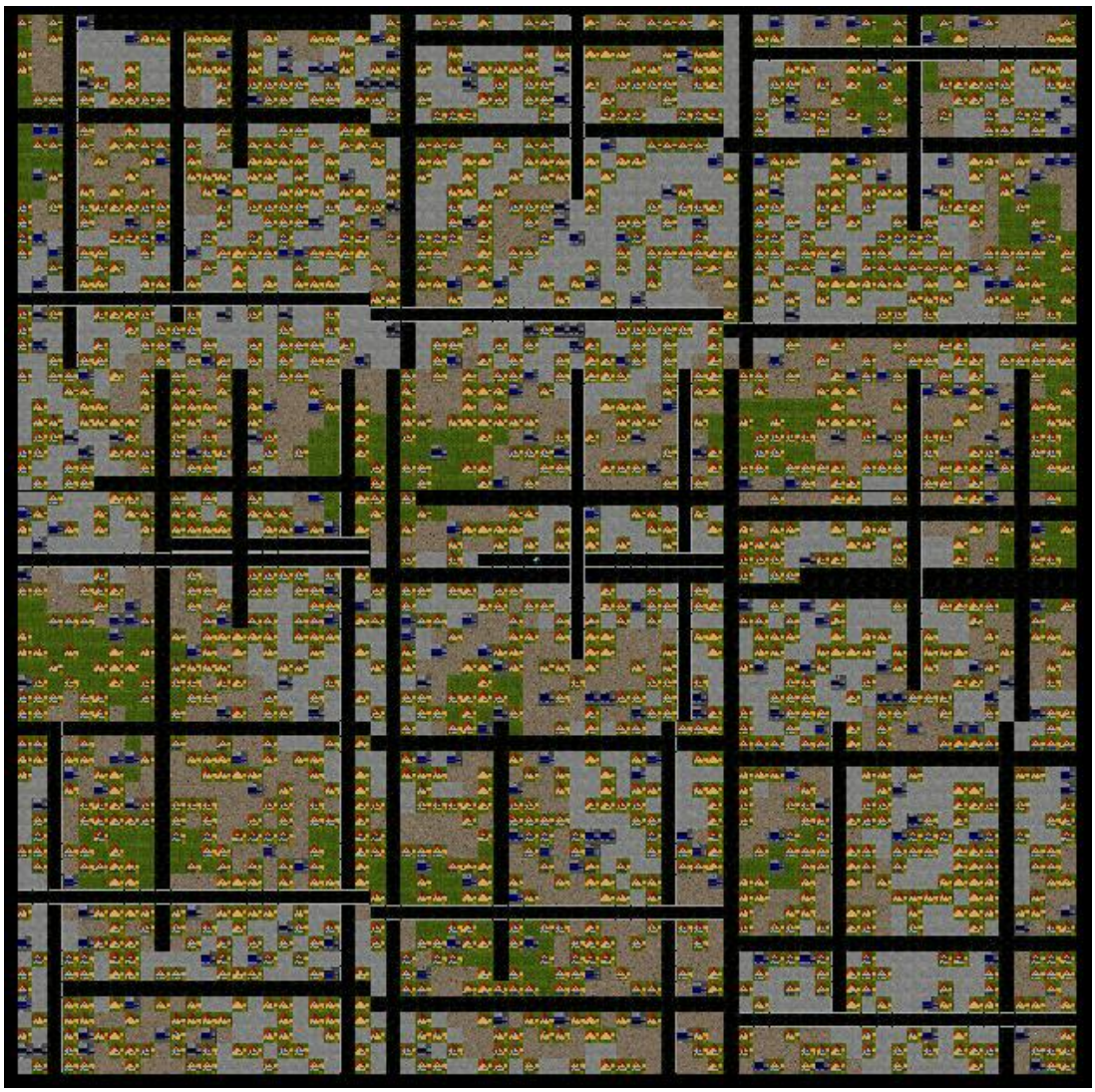

Figure 7.13: Overview of block system of the world created by seed “20152015”

Seed value 20152015 was selected to procedurally generate a world in the simulation. Figure 6.12 shows the player's view. Roads that lead in different directions are generated. Both residential and commercial buildings are placed around the world. Moreover, terrain types like concrete, grass and dirt are mapped to the environment. From Figure 6.12 we can see that nine blocks in the block system procedurally generate the environment elements. Terrain tiles between blocks are continuously distributed. As a player moves further, new areas of the world are procedurally generated in real-time. In association with the pseudo generator, if a player goes back to the starting point, the same views are generated.

7.6 Summary

This chapter has highlighted features in the design and implementation of the simulation. It first described the architectural design of the block system and multi-threading procedural system. Workflows and algorithms within these systems were introduced in text and diagrams. Both systems are designed to improve system performance from different aspects.

After going through the architectural design aspects of the system, the actual implementation was presented to further describe the functionalities of the various systems. The development of the simulation was based on C++ programming language in association with third party libraries e.g. OpenGL, SFML and Libnoise to gain interfaces and controllability in different areas. Following an overview of the classes within the system, a description of implementing a finite state machine was presented. The system can change to different states in run-time to handle particular events. The Block system is implemented by taking advantage of manager and class inheritance. With block system, the framework gives better memory control and optimization while generating a large scale virtual environment. Optimizations of the render system were illustrated. Vertex array render and texture atlas are two key techniques that were used in the framework to enhance performance. The implementation of the procedural system was presented in detail. Three layers of environment elements were procedurally generated. The terrain layer and building area layer were generated based on two different Perlin Noise value maps. The road network layer was based on a simple and fast algorithm. By combining layers of environment elements, a large scale procedural environment was generated. At the end of the chapter, the results of the simulation were presented in several screenshots.

Chapter 8

Simulation Approach Evaluation and Discussion

8.1 Introduction

PGCG has becomes more and more popular within the independent game industry, so proposing a simple but effective framework for PGEG is now necessary. In the previous chapter, the visual results of the simulation were presented. However, only providing visual results from the execution of the simulation is not enough to show the value of the framework. In this chapter, an evaluation of the performance and gameplay experience is illustrated by tables and discussions. The performance evaluation section contains four main test sections: render system, multi-threading, multi-threading procedural system and different sizes of blocks. Following the test sections, a discussion section presents the advantages and disadvantages according to the performance evaluation. A further evaluation section focuses on the effects of the development process and user experience (gameplay experience) followed by further discussion. In the next section, the limitations of the framework are pointed out in order to lead to aspects that could be considered during future development. Finally, a summary of this chapter is presented.

8.2 Performance Evaluation

When evaluating if an application is valuable, performance is usually considered to be the most important criteria. In real-time render based applications such as video games, how many pictures the application can provide per second decides the performance. The basic concept in video games is animation. The more pictures that are rendered per second leads to a smoother experience for players. Each picture is called a frame in the industry. Frames per second (FPS) is the unit to measure the speed of an application. A frame rate lower than 30 FPS produces discontinuous images to human eyes. As a result, having a frame rate lower than 30 FPS in a video game is not acceptable. In this test section, instead of measuring the number of frames per second in each case, the time taken to complete a frame will be measured to provide a more straight forward and accurate way of evaluating performance. Figure 7.1 shows the mechanism of recording the time consumed in completing single frame. At the beginning of the game loop, the system records the time that has elapsed since the execution of the application in milliseconds.

Then the system continues other tasks to finish the frame. Once the program runs back to the beginning of the loop, the system will also record the time that has elapsed. Then the system calculates the difference between the current elapsed time and the one recorded in the previous frame. The difference represents the time consumption for completing a frame. For improving the accuracy of the evaluation, each test data was recorded after the application ran for one minute. The average interval time was recorded in milliseconds for each test done. Also, the largest and smallest time intervals are presented as criteria as well.

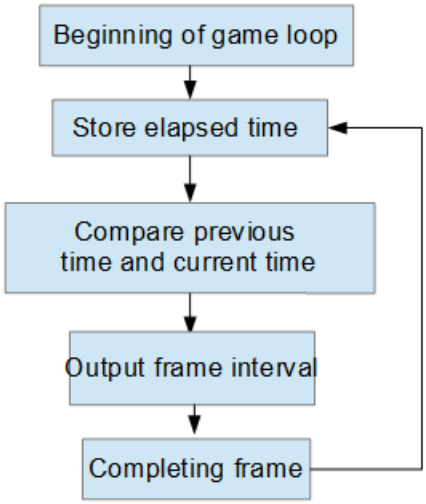

Figure 8.1: Methods to measure time intervals between each frame within simulation

Other than measuring frame intervals for performance evaluation, system memory consumption is another factor that should be considered in evaluating the performance of the framework. As the simulation generated various sizes of virtual world based on different demands, system memory consumption varied. System memory consumption was measured by recording data from Windows Task Manager.

8.2.1 Effects from Optimized Render System

The first test focused on the render system. As introduced in the previous chapter, vertex array was used to improve performance when rendering a large number of objects. In this test, the simulation used the traditional render method and batch geometry method to render the procedurally generated world. In the traditional method, every object has its own member render function and is called in order by the object manager. In order to gain more accurate data, the number of tiles in each block was the only changeable variable that defined the different test

complexities. The complexity levels are listed below in Table 7.1. Frame intervals were record for both render methods and static/dynamic modes. The total number of tile for each complexity level ranged from 1,089 to 815,409, which pushed the system to the limit. Moreover, the frame interval times were recorded in both static mode and dynamic mode. In static mode, the character in the simulation and the camera remains static during the data recording. In dynamic mode, the camera follows the character that is being controlled by the tester to explore the world. The purpose of using the dynamic mode was that animation is frequently appeared in video games. During the animation process, many objects can be moved and rendered. Therefore, using static and dynamic modes to evaluate performance was reasonable and valuable. In order to gain more accurate data, only one direction was allowed in a player's movement. Additionally, the player's travel distance was 100 tiles for every test case in dynamic mode. For every test case in this evaluation, the seed of the pseudo generator was set to 11111 to make sure the contents of the generated environment remained the same throughout the tests.

Table 8.1: Complexity settings in render system evaluation

Complexity Level	Tiles per Block
Level 1	**121 (1,089 total)**
Level 2	**441 (3,969 total)**
Level 3	**961 (8,649 total)**
Level 4	**2,601 (23,409 total)**
Level 5	**5,041 (45,369 total)**
Level 6	**10,201 (91,809 total)**
Level 7	**22,801 (205,209 total)**
Level 8	**90,601 (815,409 total)**

In this evaluation section, the static mode and dynamic mode had 16 test cases each. Table 7.2 shows the test results from all 32 test cases. In the render system mode tag, uppercase letters "S" and "D" represent static mode and dynamic mode. In level 8 complexity testing, the traditional render method was too slow for rendering 815,409 objects in real-time as it took a second to finish a frame.

Table 8.2: 16 test case results from evaluation of render system

Complexity Level	Render System Mode	Average Interval in 10 Frames	Highest Frame Interval	Lowest Frame Interval	Difference
Level 1	Traditional - S	1.4 MS	2 MS	1 MS	∓0.1MS
	Batch Geo - S	1.5 MS	2 MS	1 MS	
	Traditional - D	1.5 MS	2 MS	0 MS	∓0.1MS
	Batch Geo - D	1.4 MS	2 MS	0 MS	
Level 2	Traditional - S	10.5 MS	12 MS	11 MS	∓8.5MS
	Batch Geo - S	1.6 MS	2 MS	1 MS	
	Traditional - D	10.4 MS	13 MS	10 MS	∓8.7MS
	Batch Geo - D	1.7 MS	2 MS	0 MS	
Level 3	Traditional - S	20.1 MS	25 MS	20 MS	∓18.3MS
	Batch Geo - S	1.8 MS	2 MS	1 MS	
	Traditional - D	21.3 MS	24 MS	20 MS	∓19.4MS
	Batch Geo - D	1.9 MS	2 MS	1 MS	
Level 4	Traditional - S	47.1 MS	48 MS	45 MS	∓44.3MS
	Batch Geo - S	2.8 MS	3 MS	2 MS	
	Traditional - D	47.4 MS	49 MS	46 MS	∓44.5MS
	Batch Geo - D	2.9 MS	3 MS	2 MS	
Level 5	Traditional - S	83.1 MS	84 MS	81 MS	∓79.8MS
	Batch Geo - S	3.3 MS	4 MS	3 MS	
	Traditional - D	84.3 MS	85 MS	83 MS	∓80.9MS
	Batch Geo - D	3.4 MS	4 MS	3 MS	
Level 6	Traditional - S	170.5 MS	172 MS	168 MS	∓165.9MS
	Batch Geo - S	4.6 MS	6 MS	3 MS	
	Traditional - D	171.3 MS	176 MS	169 MS	∓166.6MS
	Batch Geo - D	4.7 MS	6 MS	4 MS	
Level 7	Traditional - S	361.4 MS	367 MS	354 MS	∓353.2MS
	Batch Geo - S	8.2 MS	11 MS	8MS	
	Traditional - D	361.8 MS	369 MS	356 MS	∓353.5MS
	Batch Geo - D	8.3 MS	10 MS	8 MS	
Level 8	Traditional - S	1236.3 MS	1248 MS	1229 MS	∓1207.6MS
	Batch Geo - S	28.7 MS	31 MS	25 MS	
	Traditional - D	1233.1 MS	1250 MS	1228 MS	∓1204.3MS
	Batch Geo - D	28.8 MS	33 MS	26 MS	

Overall, rendering in either the traditional way or using the batch geometry technique, frame interval times were similar between the static mode and dynamic mode. Among the tests at the

first complexity level, both render methods performed well enough. The average interval time difference (in 10 frames) between the two methods was around 0.1 milliseconds, which could be ignored when both methods rendered nearly 660 frames (1000MS/1.5MS) per second. At level 2 complexity, batch geometry rendering began to show its advantages, as the number of tiles increased rapidly. At level 2, batch geometry was approximately 80% faster than the traditional way. At level 3, the advantage grew to nearly 91%. As the complexity level increased, the advancing percentage of batch geometry rendering grew continuously to around 97%. On the other hand, at level 4, time consumption for traditional rendering to complete a frame was more than 33.33 MS, which created obviously discontinuous images. At the final level of complexity, the traditional method needed more than a second to render a scene with 815,409 tiles, while batch geometry rendering still had an acceptable result (around 28 MS). Comparing the highest interval data and lowest interval data, we can see that both methods were stable during run-time as no high curves appear on the frame interval time.

8.2.2 Effect of Multi-threading Procedural System

One of the purposes of the framework is to provide the possibility of generating large scale virtual environments. The actual generated world could be many times larger than the nine blocks that are allocated in the block system. As described in the previous chapter, the blocks would be updated during the player's movements. This section tests performance while the system was updating new blocks with and without multi-threading in the procedural system. Without multi-threading tasking enabled, the system waits for all the new blocks to be generated before rendering new frames. Levels 1 to 6 in Table 7.1 were used as the complexity level variables during the tests in this section. The performance of updating new blocks could be measured by recording the highest frame interval time and lowest frame interval time. Fluctuation of highest and lowest frame intervals represents the lag that players will suffer during gameplay. The rendering method was limited to batch geometry. As causing the blocks to update requires players to move in the simulation, in the next test cases the player was only allowed to move left to cause new blocks to update. Also, the seed value was set to 11111 to limit the generated contents. With these limitations, content of the same size (3 blocks on the left) was updated throughout each test case. Twelve test cases in total were performed in this section. Table 7.3 shows the test results of all evaluations in this section.

Table 8.3: 12 test cases from toggle multi-threading procedural system evaluation

Complexity Level	Multi-threading Procedural System	Average Interval in 10 Frames	Highest Frame Interval	Lowest Frame Interval	Fluctuation
Level 1	On	1.4 MS	2 MS	1 MS	1 MS
	Off	1.5 MS	5 MS	1 MS	4 MS
Level 2	On	1.5 MS	2 MS	1 MS	1 MS
	Off	1.6 MS	11 MS	1 MS	10 MS
Level 3	On	1.7 MS	2 MS	1 MS	1 MS
	Off	1.7 MS	21 MS	1 MS	20 MS
Level 4	On	2.6 MS	3 MS	2 MS	1 MS
	Off	2.5 MS	50 MS	2 MS	48 MS
Level 5	On	3.2 MS	4 MS	3 MS	1 MS
	Off	2.9 MS	96 MS	2 MS	94 MS
Level 6	On	4.5 MS	6 MS	4 MS	2 MS
	Off	4.3 MS	191 MS	4 MS	187 MS

According to Table 7.3, we can see that the multi-threading procedural system provided a stable performance when updating new blocks in twelve test cases. The fluctuation range is limited to 1 to 2 milliseconds. On the other hand, when multi-threading tasking is toggled off, the fluctuation value becomes larger and larger along with the increase in complexity level. At level 6, when 30,603 tiles needed to be updated on the player's movement, the system froze for 187 milliseconds while waiting for a frame to complete without multi-threading. In the meantime, with multi-threading toggled on, only a 2- millisecond fluctuation appeared during the generation of new blocks. Overall, when no blocks were updating, both methods gave similar results on the average interval time within 10 frames. From level 1 to level 6 in complexity, the average frame intervals grew from 1.5 milliseconds to 4.3 milliseconds approximately.

8.2.3 Visual Latency on Various Sizes of Blocks

Besides running tests on the render system and multi-threading procedural system, another important factor to be measured was the visual latency while updating new areas of blocks. As the render system and procedural system work in separate CPU threads, users can still explore the world with their character while new areas of blocks are updating. There is a chance that players might meet a void area because of latency as the procedural system is still on the way to that area.

The purpose of this test was to find out if visual latency became obvious during the increasing complexities. In this evaluation case, eight complexity levels in Table 7.1 were used as the only changeable variable in all the test cases. Still, the average interval time in 10 frames could be measured. Moreover, in order to quantify visual latency, the time consumption for completing new block generation was recorded in milliseconds. Also, the level of visual latency was divided into three levels: none, moderate and obvious. The level of visual latency was marked by a tester based on the visual effect in the simulation test cases. In order to cause the blocks to update in different situations, the tester controlled the in-game character to move horizontally to the left of the world. As a result, three blocks on the left border were updated constantly. Also, the seed value was 11111. Moreover, the camera in the simulation zoomed out in order to provide a full view of the upcoming blocks to the tester, similar to Figure 6.13. Eight test cases in total were examined in this section. Table 7.4 illustrates the test results of this visual latency evaluation.

Table 8.4: 8 test cases from visual latency evaluation with various sizes of blocks

Complexity Level	Number of tiles in one block	Average Interval in 10 frames	Time consumption for block updating	Visual Latency Level
Level 1	101	1.5 MS	3 MS	None
Level 2	441	1.6 MS	8 MS	None
Level 3	961	1.8 MS	18 MS	None
Level 4	2601	2.4 MS	50 MS	Moderate
Level 5	5041	3.1 MS	98 MS	Moderate
Level 6	10201	4.2 MS	195 MS	Moderate
Level 7	22801	7.8 MS	439 MS	Obvious
Level 8	90601	28.3 MS	1745 MS	Obvious

In Table 7.4, the data for the average interval time in 10 frames is close to the data in Table 7.2 for batch geometry rendering. For level 1 and 2 complexities, it took 3 milliseconds and 8 milliseconds for the procedural system to finish updating all the blocks. With higher complexity levels, the time consumption for block updating became larger. At level 6, this number grew to 195 milliseconds. At level 8, when 3 * 90601 tiles needed to be updated, the system spent 1745 milliseconds finishing the update task. From the marks given by the tester on visual latency level, there was no feeling of latency at level 1 to level 3. Moderate visual latency level is marked on the

level 4 to level 6 complexities settings. However, starting from level 7, the visual latency level was marked as “Obvious” for tester.

8.2.4 System Memory Consumption on Different World Sizes

In this case, the evaluations focus on the simulation’s system memory consumption after generating different sizes of worlds. To gain more data, various complexity levels were set and are presented in Table 7.5. Four levels of complexity were provided. At each level of complexity, using the same number of blocks, two numbers (27 and 57) of tiles would be used for generating a world. The purpose of the settings was to find out whether the total number of blocks or number of tiles per block was the factor affecting the framework’s system memory consumption. Both factors decided the size of the world. The listed factors were the only variables among all the test cases.

Table 8.5: Complexity settings for system memory consumption tests

Complexity Level	Total Number of Blocks	Number of Tiles per Block
Level 1	**5 * 5 = 25**	**27**
		57
Level 2	**10 * 10 = 100**	**27**
		57
Level 3	**50 * 50 = 2500**	**27**
		57
Level 4	**100 * 100 = 10000**	**27**
		57

Eight test cases in total were performed in this section. Table 7.6 records results from all the test cases for system memory consumption.

Table 8.6: 8 test cases from system memory consumption on different world sizes

Complexity Level	Total Number of Blocks	Number of Tiles per Block	System memory consumption
Level 1	**25**	**27**	**51,380 KB**
		57	**76,592 KB**
Level 2	**100**	**27**	**51,384 KB**
		57	**76,601 KB**
Level 3	**2,500**	**27**	**51,391 KB**
		57	**76,585 KB**
Level 4	**10,000**	**27**	**51,381 KB**
		57	**76,595 KB**

According to Table 7.6, we found that the system memory consumption only depends on how many tiles a block contains. Regardless of complexity level, if the number of tiles per block was the same, the system memory consumption results were very close. The more tiles per block that needed to be generated, the more the system memory was consumed in the simulation approach.

8.2.4 Performance Discussion

Four performance tests were performed and the results were recorded via different tables. In the first test, the objective was to compare the performance of two render methods in the render system. No matter what kind of virtual environments are going to be generated, high performance rendering is always the number one priority in the video game industry. Nowadays, there are more and more experienced players demanding larger and deeper virtual worlds in video games. Larger and deeper worlds mean more objects to be rendered during gameplay. The test results show that the framework can provide high performance rendering with the batch geometry technique. Over 800,000 tiles were rendered during the simulation, while the system still provided a frame rate of 30+ frames per second. Each tile is formed by two triangles. As a result, at the last complexity level, over 1.6 million triangles were rendered on the screen. With the first few levels of complexity, less than 3 milliseconds were needed for the system to complete a frame. On the other hand, nearly 330 frames could be rendered in a second. If 60 frames per second is the standard for a "smooth experience", the system can use the remaining computational power to create a more attractive virtual world. Moreover, through all of the test cases, the frame rates were stable. Similarly, the previous researches [10, 52 & 53] also tweak the process with GPU power to gain better performance. In this test case, it again shows that performance could be greatly improved by adjusting generation process. Although the performance was related to the computing tasks of the simulation and hardware configurations, the results show the framework has the ability to handle other computation tasks while procedurally generating a large scale virtual world for video games.

As introduced in the literature review, many great researchers have purposed various procedural algorithms to generate different types of virtual environments. If a small scale indoor environment or similar are needed in a game, the multi-threading procedural system will not show most of its advantages, as the game environment can be generated completely during the

initialization phase. However, the huge success of the game “Minecraft” has not only led independent developers to create large unique worlds, but more and more players are willing to have adventures in those unique worlds. From the test results for the multi-threading procedural system, the framework actually brings benefits in generating larger scales of game environments in real-time. At the level 6 complexity setting, once the player moves into a new area, three blocks must be updated before the player reaches them. Thus 91,809 tiles in total have to be filled with procedural algorithms. In association with the multi-threading procedural system, players can still have a smooth gameplay experience while a large part of the world is being updated in the background. The multi-threading system has become an important feature in modern game engine design [4]. Without this system, we can see that player would suffer from more and more serious application ‘freeze’ as the size of the world grows bigger. The performance evaluation of the multi-threading mechanism proved that it could handle other types of real-time procedural environment generation.

When using the multi-threading system to update a part of a world, the problem of visual latency should be considered. Many released games with large environments have visual latency when the render distance is set sufficiently far. In the case study chapter, the same problem was found in the game “Minecraft” when the render distance was set to 32 chunks away. According to the visual latency experiments, there was noticeable visual latency when the complexity level grew above 4. The time consumption for completing block updates provided a quantified way to measure visual latency. According to the simulation design, the normal size of the viewport for the player is equal to or smaller than the size of a block. The mechanism of the block system decides that new blocks are always generated one block away from the player’s current block. In other words, the player has to cross one block to reach those newly generated blocks. In the simulation tests, the player’s movement speed was 300 units in world coordinates. It took seconds for the player to cross a block. At complexity level 8, it took 1.7 seconds to generate all the new blocks. However, each block contained 90,601 tiles, so that it was impossible for the player to cross it in 1.7 seconds. Visual latency could only be seen when the camera zoomed out for an overview. Though visual latency existed in the simulation, it can be avoided by having proper viewport settings and a suitable world size.

Limited system memory consumption is enabled by block system. Among the data records

from the system memory tests, system memory consumption was controlled by the number of tiles per block. No matter how many blocks were contained in the whole world, only nine blocks would be allocated during run-time. Huge worlds could be built, while great control over system memory consumption was provided.

Overall, throughout the four performance evaluations of the different aspects, the high performance records for the simulation were acceptable and encouraging. However, usage of Perlin Noise and other procedural algorithms to generate large scale virtual towns as the game environments only represents one of many procedural game designs. But the performance of the simulation test cases proves the possibility of the framework supporting other types of PGEG methods with acceptable performance levels.

8.3 Implications for Indie Development

PGEG has become popular among independent game developers in recent years. Few studies provide possible framework solutions that are suitable for independent game developers to use in various procedural algorithms. Though more and more professional game engines are available for independent games, many indie developers don't have the resources to make use of those professional tools. The simplicity and performance provided by this framework is reasonable and accessible for many indie game developments. Finite state machine designs let developers manage their applications with various states. Each state can be registered easily and has its own resource management process. In the literature review chapter, different procedural algorithms were introduced. Many of those algorithms could be adapted or even boosted by this tile based framework. For example, Prachyabrued, Roden and Benton [41] proposed a method to generate a stylish 2D map. The process of their algorithm is based on a grid-like data container which could be reproduced using the tile-based system in this framework. With the help of the block system and render system, a larger world could be generated based on their algorithm. Moreover, from the Minecraft case study section, we can see that Minecraft's procedural environment is based on a grid container as well. From literature review section, it found out that most released indie game titles use noise maps as main feature in PGEG. The simulation of this research also uses the noise map to product virtual environment. Figure 7.13 illustrates that a large scale virtual town is generated in the simulation approach. Unique formation of the world could be found by using various seed values. Multiple layers of procedural generated content

provide the world different details: residential and commercial building distributions, changeable but continuous terrain and road networks. During the player's exploration, new areas are generated effectively in real-time. The presentations and evaluations show that the system is cable to use noise maps, which is a popular method among indie developers, to generate game environment effectively.

The tile-based map and block system provide a great foundation for 3D terrain generation, as each tile can represent a point with its own height value. A similar game mechanism from Minecraft could be developed within this framework. Moreover, Parberry [23] proposed to use noise maps to create large outdoor terrain procedurally. The simulation also uses noise maps during the process, though the presentation is in a two-dimensional form, 3D terrain similar to previous research could be created by extending the data according to Y-axis. The multi-threading procedural system provides a way for developers to combine various procedural algorithms to build a unique world that is suitable for their game designs. In the research of Smelik et.al [44], a multi-layer system is used to improve the effectiveness for level designers while hand crafting large scale outdoor environment. Taking the advantages that multi-layer system brings, a multi-layer procedural system is implemented within the framework. In the simulation design, three layers of environmental elements were generated before the procedural game environment (a town) was generated. Although the visual layout and structure of the environment was rather simple, it shows the ability to combine different procedural algorithms to produce more interesting results for gaming. Along with the successful outcomes from Smelik et.al [44], the procedural system with multi-layer and multi-threading enables is proved to be effective during game development. Simplified structure gives developers a more logical and structural way to divide their expected world into reasonable layers. In this way, both performance and controllability could be raised due to simplified data management. Moreover, the performance improvement enabled by the render system provided more calculation power for visual effects and game objects. More computational room means more tasks can be added and finished for gameplay during game development. A lot of human power is needed to create and polish a large game environment. Obviously, PGEG development time could be greatly reduced by automatic processes. With the flexibility and performance produced by this framework, independent game developers who are resource sensitive could focus their effort on their core

designs more effectively and accurately. Currently, the explosive growth of independent games is still continuing. However, lack of particular tools for game development makes PGEG not easy to show its advantages in many indie tiles. The evaluations proved that the framework proposed in this research could help indie game developers to create PGEG games in the future.

8.4 Limitations and Future Developments

Though evaluation of the simulation approach shows good performance, several limitations were found during the research process. Firstly, only one game design aspect was developed during the design and implementation process of the simulation. More procedural environment generation methods should be covered in order to further show the flexibility of the framework. Another limitation is that the evaluation process was limited to the development on a personal computer. Though the framework is based on cross platform libraries and low level programming languages, different configurations of hardware may cause different results in performance and visual effects. The focus of this framework was to provide support for PGEG; developers who demand high levels of controllability in the environment creation may not benefit from the framework. The Tile based system limited the form of virtual environment that could be created. A highly automated environment creation process may not suit game designs that have restrict specific objects placement and art styles.

A prototype of the framework was designed and implemented along with a game prototype in this research. There are many aspects for future researchers to improve. More procedural algorithms could be combined with the framework to create specific world for various gameplay designs. Moreover, an environment structured in three-dimensional space should be simulated to gain more information about the framework. Further optimization of the render system as well as system architecture could also be studied. With the increasing release of independent game titles, more and more indie game developers choose PGEG in their games to gain advantages in the market they share with professional mainstream developers. More research on the relationship between PGEG and indie game development should be pursued.

8.5 Summary

This chapter has presented evaluations and a discussion on the simulation approach that was designed and implemented in this research. It was necessary to provide evaluation after

describing the development process in Chapter 6. Four evaluation tests were performed to prove the ability of the framework. Performance test results of the render system and multi-threading procedural system have been presented. While rendering over 800,000 tiles, the framework still provided a smooth visual effect in association with the render system and multi-threading procedural system. Also, visual latency in the multi-threading system and the system memory consumption of the framework were explored. Both visual latency and system memory consumption can be controlled by tweaking various game settings such as viewport size and block size. The simulation data shows positive evidences that the framework could greatly support various PGEG methods with high performance.

Discussions on performance and the implications for independent game development have further proved the advantages of this system. The simplicity and flexibility of the framework are acceptable and useful for indie game developers who have limited resources. Even with its high rendering performance, the framework leaves enough computing power for developers to attach more tasks for improving the gameplay experience during game development. The application runs smoothly while updating large parts of the environment in the background. Usage of the multi-threading system provides a way to combine various procedural methods to generate interesting game environments. Also, the tile-based map structure is flexible enough for many types of game environments.

Additionally, the limitations and recommendations for future developments are found in this chapter. More procedural environment algorithms and 3D virtual procedural environments could be implemented within the framework to improve its flexibility and performance further. More optimization techniques could be researched and implemented into the framework. With the rise of the indie game industry, studies on the relationship between PGEG and indie games are valuable.

Chapter 9

Conclusion

The main objective of this thesis was to design and evaluate a framework for PGEG for independent game developers. Modern digital distribution platforms provide great opportunities for independent game developers to compete with traditional mainstream game developers. However, mainstream developers have dominated video game markets for decades: massive development tools and paid professionals make their games excellent. With limited resources, it is extremely difficult for independent developers to earn profits in the shared market. In 2009, the indie game Minecraft achieved great commercial success through its procedurally generated environment. Countless indie developers studied and implemented procedural methods in their games. Many of these tiles are achieved surprising success [19]. In recent years, more and more researchers have proposed studies about automatically generating virtual environments. However, very few studies have proposed a framework to adapt PGEG for game development. Studies of independent developments are even rarer. With the expectation of a continuous increase in indie games, it is worthwhile to propose a PGEG framework for indie developers.

In Chapter 2, the definition and development of independent games was explored. Similar to other forms of indie art, such music and films, indie games are considered different to "mainstream" games. If focusing on the development scale, indie game teams are represented by small studios or individuals who have limited resources. Chapter 3 introduced the concept of PGCG. Instead of manually crafting elements for video games, PGCG creates game content according to particular rules. Though PGCG is used in the creation of many game elements, PGEG could be considered the most popular.

Through the literature review in Chapter 4, various PGEG algorithms were reviewed. Some of them generate indoor levels while others create large scale outdoor environment automatically. Generative grammar or search-based algorithms are common in the procedural process. Usage of a combination of various procedural methods to create special virtual environments has been proposed by previous researchers. Most of the studies mentioned that high performance is a priority when adapting PGEG techniques. However, few researchers have focused on providing a

framework for using PGEG in game development. According to the methodology of this thesis, a case study on the most famous indie game Minecraft was presented. In order to support the massive procedurally generated world, the Minecraft developer designed a chunk system to gain controllability and performance on memory usage. In world generation, Perlin Noise is widely used to generate terrain, physiognomies and in biomes distribution. The most significant characteristic of Perlin Noise is that its pseudo random generator provides continuous noise values which are very suitable for outdoor terrain formation. Many previous studies have used Perlin Noise as the basic algorithm to create a virtual world. The information from previous studies inspired the use of Perlin Noise in generating the virtual environment in this thesis. However, the design and implementation of the system in Minecraft remains unknown, as the developer keeps them secret.

It is obvious that PGEG brings lots of advantages to indie developers, such as saving development resources and providing interesting results. A wide range of factors create obstacles for independent developers. The lack of PGEG frameworks available for the indie game industry is becoming an issue as the popularity of PGEG among indie circles continues to grow.

Chapter 6 presented the architectural design and implementation of the framework and simulation approach. Diagrams and text descriptions were provided to introduce the core systems of the framework. Adapting the finite state machine intended to simplify resource management in every game state. The Blocks system manages world updating and limits system memory usage. The multi-threading procedural system supports multiple layers of procedural content generation in real-time while not affecting the gameplay smoothness. The optimized render system uses the batch geometry technique to greatly improve performance in rendering nearly a million objects. The whole application was developed in C++ programming language. With the association of third party libraries such as OpenGL, SFML and Libnoise, all expected functionalities worked well. Results from actual execution of the simulation are presented by screenshots and explanations.

In order to find out the viability and values of the framework, 40 test cases in total were performed. Frames per second (FPS) proved to be the most suitable metric for measuring performance among real-time applications e.g. video games. Performance conditions were recorded using four different criteria. By adapting the batch geometry technique, the render

system could render large scale environments more than 90% faster than traditional methods. The multi-threading procedural system eliminated frame rate fluctuation while updating large areas of the game environment during gameplay. In the single threading tests, constantly updating of 30,603 tiles caused the simulation to freeze for 187 milliseconds. Only a 1 to 2 millisecond fluctuation appeared in the same complexity setting with the multi-threading procedural system. In the visual latency testing, nearly 1.7 seconds of visual latency appeared at the maximum complexity level. However, as the size of block grows much larger in such a complexity setting, it was difficult for players to see visual issues before the update tasks completed. Meanwhile, system memory consumption could be controlled by the mechanism of the block system.

Throughout discussions on the performance and implications for independent game development, the framework was found viable for indie developers with its simplicity and flexibility. High performance left enough computational power for more game content calculations in practical development. Also, the combination of various procedural methods proved to be successful within the framework. The framework is considered useful for independent game developments using PGEG.

Though the availability and functionalities of the framework were investigated through literature reviews, case study and evaluation of the simulation, more aspects of this topic need to be researched in the future. A wide range of procedural environment techniques could be implemented with the framework. Also, generating a 3D environment could be valuable for further proving the flexibility of the framework. As the popularity of procedural environment generation techniques is continually increasing within the independent game industry, more and more indie developers will use PGEG to gain advantage in their game titles. To resolve these limitations and future requirements, more theoretical and practical research is necessary. The designs and evaluations of the PGEG based framework proved that it can do great help in future indie game developments. Hopefully, more inspirations and improvement could be found by other developers based on this research.

Appendix A: Third party libraries used in development

A.1 OpenGL

OpenGL (Open Graphics Library) is a cross-platform, cross-language application programming interface (API) for providing controllability of various hardware pipelines while rendering 2D and 3D graphics. Hardware-accelerated rendering can be achieved through interactions with the graphics processing unit (GPU). During the development in this research work, OpenGL provides the possibility of accessing low level programming for optimization in the render system. The library was firstly developed by Silicon Graphics Inc. in 1991 and was used in a wide range of industries including video games, engineering and sciences. More discussion of the features of OpenGL can be found on the official website: www.opengl.org.

A.2 Simple and Fast Multimedia Library (SFML)

Simple and Fast Multimedia Library is a free and open-source software designed to provide simple cross-platform interfaces to multimedia components. The library is written in C++ programming language with fully object-oriented structures. In this research work, SFML provides fast operations in window creation and handling various system events. More information about SFML can be found on the official website: www.sfml-dev.org.

A.3 Libnoise

Libnoise is a lightweight portable open-source library designed to generate various types of coherent noise. It is written in C++ programming language. In the simulation approach of this research, Libnoise is responsible for providing the Perlin Noise generator with a list of configurable parameters. More information about Libnoise can be found on the official development website: libnoise.sourceforge.net.

References

[1] Arsenault, D. (2009). Video game genre, evolution and innovation. Eludamos. Journal for Computer Game Culture, 3(2), 149-176.

[2] Ashlock, D. (2010). Automatic generation of game elements via evolution, IEEE Symposium on Computational Intelligence and Games (CIG), 2010.

[3] Ashlock, D., Lee, C., & McGuinness, C. (2011). Search-based procedural generation of maze-Like Levels. *IEEE Transactions on Computational Intelligence and AI in Games, 4*(3), 260-273. doi:10.1109/TCIAIG.2011.2138707

[4] Carr, D. A. (2009). *Towards automatic parallel game engine architectures*. ProQuest, UMI Dissertations Publishing.

[5] de Carpentier, G. J., & Bidarra, R. (2009). Interactive GPU-based procedural heightfield brushes. In *the Proceedings of the 4th International Conference on Foundations of Digital Games*. ACM. doi:10.1145/1536513.1536532

[6] Dimovska, D., Jarnfelt, P., Selvig, S., & Yannakakis, G. (2010). Towards procedural level generation for rehabilitation. *ACM*. doi:10.1145/1814256.1814263

[7] Ebert, D. S. (2003). *Texturing & modeling: a procedural approach*.United States of America: Morgan Kaufmann.

[8] Frade, M., de Vega, F. F., & Cotta, C. (2010). Evolution of artificial terrains for video games based on accessibility. In *Applications of evolutionary computation* (pp. 90-99): Springer.

[9] Galin, E., Peytavie, A., Maréchal, N., & Guérin, E. (2010). Procedural generation of roads. *Computer Graphics Forum, 29*(2), 429-438. doi:10.1111/j.1467-8659.2009.01612.x

[10] Greuter, S., Parker, J., Stewart, N., & Leach, G. (2003). Real-time procedural generation of `pseudo infinite' cities. *ACM*. doi:10.1145/604471.604490

[11] Guevara-Villalobos, O. (2011). Cultures of independent game production: Examining the relationship between community and labour. Symposium conducted at the meeting of the Proceedings of DiGRA 2011 Conference: Think Design Play.

[12] Harvey, A., & Fisher, S. (2013). Intervention for inclusivity: gender politics and indie game development. *Loading*, 7(11).

[13] Hendrikx, M., Meijer, S., Van Der Velden, J., & Iosup, A. (2013). Procedural content generation for games: A survey. *Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP), 9*(1), 1-22. ACM doi:10.1145/2422956.2422957

[14] Iosup, A. (2011). POGGI: generating puzzle instances for online games on grid infrastructures. *Concurrency and Computation: Practice and Experience, 23*(2), 158-171.

[15] Irwin, M. J. (2008). Indie game developers rise up. Forbes. com. Retrieved from: http://www.forbes.com/2008/11/20/games-indie-developers-tech-ebiz-cx_mji_1120indiegames.html

[16] Ito, K. (2007). 10 Possibilities of non-commercial games: The case of amateur role-playing games designers in Japan. *Worlds in Play: international perspectives on digital games research, 21*, 129.

[17] James, K. (2005). Indie Gaming Evolution (pp. 17). Toronto: Broken Pencil.

[18] Jansson, B. (2014). Procedural Generation in Gravel. Thesis. Linköping University.

[19] Jiow, H. J., & Lim, S. S. (2012). The evolution of video game affordances and implications for parental mediation. *Bulletin of Science, Technology & Society, 32*(6), 455-462.

[20] Juul, J. (2012). *A casual revolution: Reinventing video games and their players*. United States of America: MIT press.

[21] Karray, S., & Sigué, S. P. (2015). A game-theoretic model for co-promotions: Choosing a complementary versus an independent product ally. *Omega, 54*, 84-100. doi:10.1016/j.omega.2015.01.008

[22] Kelly, G., & McCabe, H. (2006). A survey of procedural techniques for city generation. *ITB Journal, 14*, 87-130.

[23] Lagae, A., Lefebvre, S., Cook, R., DeRose, T., Drettakis, G., Ebert, D. S., . . . Zwicker, M. (2010). A survey of procedural noise functions. Symposium conducted at the meeting of the Computer Graphics Forum. Wiley Online Library.

[24] Lastowka, G. (2011). Minecraft as web 2.0: Amateur creativity & digital games. Retrieved from: http://papers.ssrn.com/sol3/papers.cfm?abstract_id=1939241

[25] Lipkin, N. (2012). Examining Indie's Independence: The meaning of "Indie" Games, the politics of production, and mainstream cooptation. *Loading 7*(11).

[26] Lopes, R., Tutenel, T., & Bidarra, R. (2012). Using gameplay semantics to procedurally

generate player-matching game worlds. In *Proceedings of the The third workshop on Procedural Content Generation in Games* (p. 3). ACM.

[27] Müller, P., Wonka, P., Haegler, S., Ulmer, A., & Van Gool, L. (2006). Procedural modeling of buildings. *Acm Transactions On Graphics (Tog), 25*(3), 614-623.

[28] Martek, C. (2012). *Procedural generation of road networks for large virtual environments*. ProQuest, UMI Dissertations Publishing.

[29] Martin, C. B., & Deuze, M. (2009). The independent production of culture: A digital games case study. *Games and Culture, 4*(3), 276-295.

[30] Mawhorter, P., & Mateas, M. (2010). Procedural level generation using occupancy-regulated extension *IEEE* Symposium conducted at the meeting of the Computational Intelligence and Games (CIG)..

[31] Michael, L., & Jeffrey, J. (2002). Game engines in scientific research (Vol. 45, pp. 27). New York: Association for Computing Machinery.

[32] Mojang. (2009). Minecraft [Computer Software]. Retrieved from: https://minecraft.net/

[33] Newman, M. Z. (2011). *Indie: an American film culture*. United States of America: Columbia University Press.

[34] Olsen, J. (2004). *Realtime procedural terrain generation*. Denmark: University of Souther Denmark

[35] Pajot, L., Swirsky, J., McMillen, E., Refenes, T., Fish, P., & Blow, J. (2012). *Indie Game: The Movie.* Canada: Filmakers Library.

[36] Parberry, I. (2014). Designer worlds: Procedural generation of infinite terrain from real-world elevation data. *Journal of Computer Graphics Techniques 3*(1).

[37] Parish, Y., & Müller, P. (2001). Procedural modeling of cities. *ACM.* doi:10.1145/383259.383292

[38] Parker, F. (2013). Indie Game Studies Year Eleven. *DiGRA 2013: DeFragging Game Studies*. Canada: York University

[39] Pereira, F. C. (2007). *Creativity and artificial intelligence: a conceptual blending approach* (Vol. 4). Berlin: Walter de Gruyter.

[40] Perlin, K. (2002). Improving noise Symposium conducted at the meeting of the ACM Transactions on Graphics (TOG). ACM.

[41] Prachyabrued, M., Roden, T., & Benton, R. (2007). Procedural generation of stylized 2D maps ACM doi:10.1145/1255047.1255077

[42] Raffe, W. (2014). Personalized procedural map generation in games via evolutionary algorithms. *SIGEVOlution U6 - U7 - Journal Article U8 - FETCH-acm_primary_26617393*, *7*(1), 27-28. ACM doi:10.1145/2661735.2661739

[43] Roland van der, L., Ricardo, L., & Rafael, B. (2014). Procedural generation of dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, *6*(1), 78. doi:10.1109/TCIAIG.2013.2290371

[44] Smelik, R., Tutenel, T., de Kraker, K. J., & Bidarra, R. (2010). Integrating procedural generation and manual editing of virtual worlds. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games* (p. 2). ACM.

[45] Smelik, R. M., De Kraker, K. J., Tutenel, T., Bidarra, R., & Groenewegen, S. A. (2009). A survey of procedural methods for terrain modelling. In *Proceedings of the CASA Workshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS)* (pp. 25-34). Netherlands.

[46] Smelik, R. M., Tutenel, T., de Kraker, K. J., & Bidarra, R. (2008). A proposal for a procedural terrain modelling framework. In *Poster Proceedings of the 14th Eurographics Symposium on Virtual Environments EGVE08* (pp. 39-42). Netherlands.

[47] Smelik, R. M., Tutenel, T., de Kraker, K. J., & Bidarra, R. (2011). A declarative approach to procedural modeling of virtual worlds. *Computers & Graphics*, *35*(2), 352-363.

[48] Smith, A. M., & Mateas, M. (2011). Answer set programming for procedural content generation: A design space approach. *Transactions on Computational Intelligence and AI in Games*, *3*(3), 187-200. *IEEE* doi:10.1109/TCIAIG.2011.2158545

[49] Smith, G., Gan, E., Othenin-Girard, A., & Whitehead, J. (2011). 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* (p. 7). ACM.

[50] Sorenson, N., & Pasquier, P. (2010). Towards a generic framework for automated video game level creation. In *Applications of Evolutionary Computation* (pp. 131-140). Berlin: Springer.

[51] Valve Corporation. (2015). Steam [Computer software]. Retrieved from:

http://store.steampowered.com/

[52] Steinberger, M., Kenzel, M., Kainz, B., Müller, J., Peter, W., & Schmalstieg, D. (2014, May). Parallel generation of architecture on the gpu. In *Computer graphics forum* (Vol. 33, No. 2, pp. 73-82). Steinberger, M., Kenzel, M., Kainz, B., Wonka, P., & Schmalstieg, D. (2014). On‐the‐fly generation and rendering of infinite cities on the GPU. *Computer Graphics Forum, 33*(2), 105-114. doi:10.1111/cgf.12315

[53] Togelius, J., Kastbjerg, E., Schedl, D., & Yannakakis, G. (2011). What is procedural content generation?: Mario on the borderline. ACM doi:10.1145/2000919.2000922

[54] Togelius, J., Whitehead, J., & Bidarra, R. (2011). Guest editorial: Procedural content generation in games. *IEEE Transactions on Computational Intelligence and AI in Games, 3*(3), 169-171. doi:10.1109/TCIAIG.2011.2166554

[55] Togelius, J., Yannakakis, G. N., Stanley, K. O., & Browne, C. (2011). Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games, 3*(3), 172-186.

[56] Tulip, J., Bekkema, J., & Nesbitt, K. (2006). Multi-threaded game engine design. Symposium conducted at the meeting of the Proceedings of the 3rd Australasian conference on Interactive entertainment, Murdoch University.

[57] Westecott, E. (2012). Independent game development as craft. *Loading... 7*(11).

[58] Whitehead, J. (2010, June). Toward procedural decorative ornamentation in games. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games* (p. 9). ACM.

[59] Wijgerse, S. (2007). Generating realistic city boundaries using two-dimensional Perlin noise. Univerity of Twente.

[60] Wilson, J. (2005). Indie rocks! Mapping independent video game design. Retrieved from: http://www.complex.com/pop-culture/2009/12/indie-rocks-a-guide-to-the-best-independent-video-games

[61] Wonka, P., Wimmer, M., Sillion, F., & Ribarsky, W. (2003). *Instant architecture* (Vol. 22): ACM.

[62] Yannakakis, G. N., & Togelius, J. (2011). Experience-driven procedural content generation. *IEEE Transactions on Affective Computing, 2*(3), 147-161. doi:10.1109/T-AFFC.2011.6