“Level Design and Gameplay through Procedural Generation”

Phillip Anthony Law

2016-17

BSc Games Development

Implementation Project Dissertation

Department of Computer Science

Disclaimer

This work is original and has not been previously submitted in support of any other course or qualification.

Name: Phillip Law

Signed: Date:

ABSTRACT

The aim of this dissertation was to research the history of Procedural Content Generation (PCG) and its various uses within the video games industry including its recent rise and fall in popularity. Methods of implementing PCG systems were investigated and evaluated based on their advantages and disadvantages and an original design was created based on said research.

The original design was implemented within the Unity Engine using Microsoft Visual Studio and the C# programming language; original 3D models were created using 3DS max and the project was saved using GitLab as a source control solution.

The implementation was then evaluated using a playtest session consisting of volunteers to make sure it was functional and achieved the goals set for it.

A final evaluation of the dissertation looking at the strengths and weaknesses of each aspect of the project was conducted.

ACKNOWLEDGEMENTS

I would like to take a moment to thank my dissertation supervisor Ralph Ferneyhough for his support and guidance throughout this project, without his expertise this project would not have been possible.

I would also like to thank Gabriela Ruiz-Leonard for being a constant source of encouragement and advice, you managed to make a mountain of work feel like a molehill and kept me going when I was ready to give in.

Finally, I would like to thank Ben Satchell for all the times he offered his limited free time to offer feedback, constructive criticism and bounce ideas around.

CONTENTS

LIST OF FIGURES

1 INTRODUCTION

1.1 AIMS

The primary aim of this project was to create a computer game prototype using the Unity Engine that implemented Procedural Content Generation (PCG) to enhance the gameplay in a meaningful way for the player in areas such as level design and item creation. A secondary aim was to attempt to answer the question of how effective PCG is as a tool for games development.

The question of PCG's effectiveness has recently been asked due to the increased belief among members of the games industry that PCG is often overused and poorly implemented, crediting several high-profile releases as proof with the recent game *No Man's Sky* (Hello Games, 2016) being held as the epitome of all the current failings of PCG implementation. The more lenient reviews of the game were complimentary of the technology powering the universe generation and complemented the technical accomplishments of the small development team (Baines, 2016). Despite this, the consensus was that the product was "an impressive set of tools grafted onto a game with very little going on" (Kollar, 2016). Jim Sterling, a prominent, independent games journalist with was perhaps the most critical, citing the game as indicative of a larger problem in the industry, that of prioritising scale over quality (Sterling, 2016).

1.2 WHAT IS PROCEDURAL CONTENT GENERATION?

Procedural Content Generation (PCG) in terms of games development is the process of creating content for a game using algorithms and rule sets to direct the process (Nelson, Shaker, & Togelieus, 2016). PCG has no real limitations when it comes to its potential uses within games development; in the past it has been used simply to generate unique trees in pre-determined locations (Bidarra, Linden, & Lopez, 2014), meanwhile the currently accepted default use of PCG is as a level creation tool (Bidarra, Linden, & Lopez, 2014). An important word to remember when considering the scope of PCG is 'content'; although PCG has several tried and tested functions it can be used to create all kinds of content from levels to game rules and even narrative aspects (Nelson, Shaker, & Togelieus, 2016). A prime example of this was the nemesis system in *Middle-Earth: Shadow of Mordor* (Monolith Productions, 2014) which created enemies with unique appearances, personalities and traits. The player interacting with these enemies would alter them further by altering their appearance with battle wounds or by having them reference past encounters.

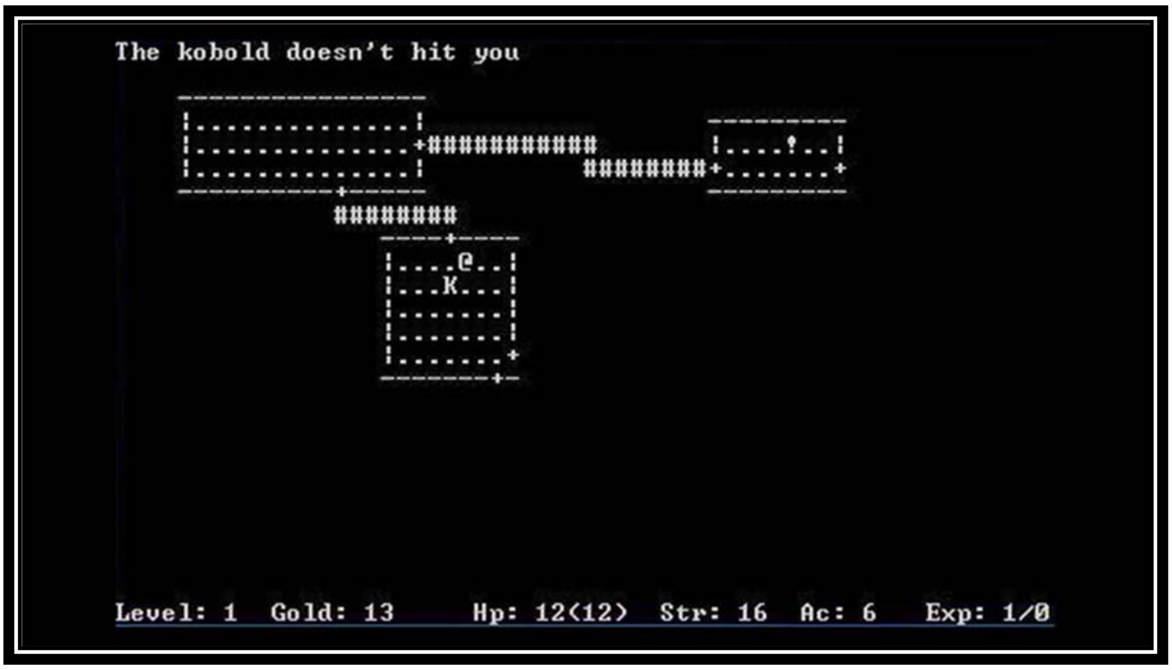
1.3 OBJECTIVES

This project had three major objectives:

1. Investigate the various applications for PCG in video games such as item creation, level design and narrative elements
2. Investigate and evaluate the various methods used to implement PCG in a video game
3. Create a game prototype using the Unity Engine that uses PCG as part of the level design and gameplay elements in response to the research

1.4 BACKGROUND

1.4.1 Historical Overview



```
The kobold doesn't hit you

Level: 1  Gold: 13     Hp: 12(12)  Str: 16  Ac: 6   Exp: 1/0
```

Figure 1 - Rogue (A.I. Design, 1980) using character-set graphics to create a level (Barton & Loguidice, 2009)

PCG was an idea born of the constraints of early computing technology; one of the earliest games to contain PCG was a UNIX based game known as *Rogue* (A.I. Design, 1980). According to Barton and Loguidice (2009) UNIX based terminals were only able to produce simplistic audio and visual experiences, often having to rely on images created out of text/characters known as "character-set graphics" (As can be seen in *Figure 1*). Memory was also a major limitation at the time (Lee J. , 2014) which for *Rogue* (A.I. Design, 1980) meant that saving hundreds of predetermined dungeon layouts was an impossibility. Instead the game used an algorithm to generate a new 'dungeon' for the player to explore following a set of pre-determined rules to make sure it was playable (Barton & Loguidice, 2009). While *Rogue* (A.I. Design, 1980) may not have been the first game to use PCG it is certainly one of the most well-known, having been the namesake for the genre "Rogue-Like" (Barton & Loguidice, 2009).

An early focus of PCG that has seen continuous use throughout the games industry was in non-interactive elements of video games such as foliage and trees (Smith, 2014). The primary focus of this implementation was to increase player immersion with seemingly realistic vegetation (as well as other background aspects such as rubble) and to decrease the workload of artists.

Many other video games utilised PCG to enhance the experience in various ways, however it was not until the release of *Minecraft* (Mojang, 2011) that PCG was fully cemented in the public consciousness and started to be adopted as a common tool for developers to use (Lee J. , 2014). Initially PCG was utilised by smaller 'indie' studios; games such as *The Binding of Isaac* (McMillen, 2011a) and *Rogue Legacy* (Cellar Door Games, 2013) were critically well received and profitable, attracting the attention of bigger developers and publishers.

1.4.2 How did the problem arise?

After the huge success of games such as *Minecraft* (Mojang, 2011) many developers from both 'Indie' and 'Triple A' began to look at PCG as a potential development tool and with good reason; PCG has many potential benefits it can bring to the development of a game the primary one being the potential cost it can save. Games Development has proven to be an expensive undertaking, especially for 'AAA' titles (a term used to describe blockbuster titles) with recent games such as *Grand Theft Auto V* (Rockstar North, 2013) and *Destiny* (Bungie, 2014) costing $265 million (not including marketing) and $500 million (including marketing) respectively (T.C., 2014). Although those are two extreme examples, the cost of a console project with around 100 developers reportedly doubled to $20 million in 2013 and was expected to rise (Sherr, 2013).

There were many reasons given for the rising cost of game development, the primary one being the constant improvement in computer graphics requiring increasingly large teams of full-time artists to meet the demands of their audience (T.C., 2014). Furthermore, the equipment required for developers to perform their jobs is also costly as each employee contributing to development would require a workstation which required among many things a high spec computer and the many software licenses for development tools (Boucher-Vidal, 2014).

Using PCG allows developers of all sizes to create a vast amount of content quickly and for a fraction of the cost it would take to make all the content manually (Nelson, Shaker, & Togelieus, 2016). Furthermore, PCG has the potential to increase the longevity of a game by adding near limitless variety and gameplay options, generating a unique experience each time the game is played (Couture, 2016). Moreover, the scope for PCG in the future has yet to be fully explored, the potential for 'never-ending games' or games that can generate content tailored to their players have become very real possibilities in the future (Nelson, Shaker, & Togelieus, 2016).

PCG also has many potential negatives, one of which surprisingly stems from the promise of near infinite content. Couture (2016) commented on this stating that while PCG has the potential to create almost limitless possibilities, if mishandled it can also create a large amount of uninteresting, directionless or disjointed content, "it can just be a pile of random parts with little means of engaging the player" (Couture, 2016).

Unfortunately, many high-profile games containing PCG such as *No Man's Sky* (Hello Games, 2016) and *We Happy Few* (Compulsion Games, 2016) released to general negative reviews with many outlets citing the poor implementation of PCG as the primary reason for the criticism. Due to the exposure these games received and the backlash from fans who also cited the poor implementation of PCG as a reason for their anger, PCG has started to develop a stigma among the gaming audience as overused and under utilised.

It can be noted that the games that received the most backlash from the gaming audience are the ones that utilised PCG primarily as a level design tool. It can be argued that the popularity of *Minecraft* (Mojang, 2011) and it's implementation of PCG cemented the technique purely as a level generation tool in the eyes of the audience which led to many developers who attempted to imitate the success of *Minecraft* (Mojang, 2011) to focus either solely or primarily on the aspect they knew the audience enjoyed, the procedural level design instead of attempting new and interesting methods of using it.

1.4.3 Personal Involvement

I felt a personal interest in PCG as a topic for several reasons; my interest in the games industry outside of the academic world meant I witnessed the rise in popularity and subsequent fall from grace that PCG suffered and I believed the majority of the backlash against it was mostly unfounded. The accusations that PCG caused the problems in major games such as *No Man's Sky* (Hello Games, 2016) hurt the reputation of PCG as a development tool when the truth of the matter is the tool is only as effective as those who wield it will allow. This led to my primary reason for interest in the topic; I saw PCG as a tool with a lot of potential that had just been mishandled in the industry in recent years and I wished to explore its uses myself.

A secondary reason was simply that I was interested in several aspects of game design, especially programming and saw this as an opportunity to challenge myself with a highly intensive project that enabled me to expand my knowledge of games development in multiple areas.

1.5 The Future of PCG

1.5.1 Potential Outside of Level Design

As previously mentioned, PCG has been used to varied degrees of success in many aspects of games development from foliage to entire levels; perhaps the most critically praised implementation in recent years is the previously discussed Nemesis System from *Middle-Earth: Shadow of Mordor* (Monolith Productions, 2014). The recent negative reviews surround high-profile games using PCG may be a blessing in disguise for PCG within the games industry as the current audience resistance to procedurally generated levels could pave the way to innovations such as the Nemesis System as developers begin exploring new ways to utilise PCG.

1.5.2 Potential in Multiplayer Games

Although recent attempts at creating PCG driven Massive Multiplayer Online (MMO) games such as *Landmark* (Daybreak Game Company, 2016) met very little success with several of them being cancelled. It is possible that the scope of these projects was simply too great compared to what is currently possible however the potential remains for a player driven MMO that reacts using PCG techniques to what each player does in that world, creating unique and infinite content (Smith, 2014).

1.5.3 Potential to Create New Genres

Smith (2014) argues that many memorable moments and innovations in games come from the unexpected, and that PCG has the potential to create these unexpected moments and twists on modern gameplay due to its ability to create nearly infinite content and variations on a scenario.

1.6 Review of Hardware and Software

1.6.1 Hardware

This dissertation utilised a ‘high-end’ computer for both the written and implementation aspects. The implementation was also created to utilise both the Xbox 360 controller and the Xbox One controller.

1.6.2 Software

This dissertation was written using Microsoft Word 2016 (a part of the Microsoft Office Suite 2016) on the Windows 10 Operating System. The practical implementation was created using Microsoft Visual Studio to write the C# scripts and the Unity Engine as the chosen game engine; 3D models were created within 3DS Max. GitLab was utilised as this projects source control (as is expected within the software development industry) to prevent the loss of any work/progress.

2 Literature Review

2.1 Advantages of using Procedural Generation

2.1.1 Efficiency and Cost Saving

The most widely agreed upon benefit of PCG is the potential to “greatly reduce the increasing workload” (Bidarra, Linden, & Lopez, 2014) for developers. The average cost of games development has risen over the years due to increases in a variety of factors such as development team size and the demands and expectations of the audience (Nelson, Shaker, & Togelieus, 2016; Couture, 2016). PCG, when implemented effectively, can allow even small teams to create a large amount of content within a very short amount of time. From a developer’s point of view this will also save them from any potential monotony from creating each piece of content individually (Ra, 2016).

During a presentation at the Game Developers Conference (GDC) in 2014, the developers of *Rogue Legacy* (Cellar Door Games, 2013), brothers Kenny and Teddy Lee, discussed their use of PCG during development and how it saved them a lot of time; the brothers spent two weeks creating a custom editor that allowed them to create and test a room within five minutes (Lee & Lee, 2014). Although some may see those two weeks as better spent manually creating content, the brothers agreed that “The time that [they] saved making content more than made up for the time [they] spent on the code” (Lee & Lee, 2014).

2.1.2 Dynamically adapt to the player

PCG has been predominantly used within the games industry as a means to create environments for a player to explore (Bidarra, Linden, & Lopez, 2014) with games such as *Minecraft* (Mojang, 2011) and *No Man’s Sky* (Hello Games, 2016) creating entire worlds. PCG does not have to be limited to just level creation however, as Rogers (2014) points out, just because a “game uses procedural design doesn’t mean everything in the game has to be generated procedurally”.

In recent years PCG has been used in more subtle ways to enhance the gameplay of a title by adapting to the player. *Left 4 Dead 2* (Valve Corporation, 2009) used handmade levels which many would consider an odd choice for a game that advertised itself as “highly replayable” (Booth, n.d.), however each playthrough was kept fresh using ‘The Director 2.0’, an artificial intelligence with the ability to track player metrics such as health and position and react accordingly. The Director 2.0 was in control of various factors including enemy placement and population, item placement and even had the ability to add and remove parts of the level to create new paths for the player to use guaranteeing a new experience each playthrough.

A more recent example can be found in *Middle-Earth: Shadow of Mordor* (Monolith Productions, 2014); most the game is handmade except for the lauded 'Nemesis System' which procedurally generated unique enemies with names, personalities and traits for the player to interact with. How the player interacted with them would alter their behaviour and occasionally their appearance; an enemy defeated in battle may return with scars reflecting how they were beaten for example. In this respect, the Nemesis System utilised PCG to create dynamic storytelling for the player to great effect.

2.1.3 Potentially Infinite Content

Recent research has also touched on the idea of dynamic difficulty (Chu, Harada, Kaidan, & Thawonmas, 2015) in which the game can begin to create levels based on the skill of the person playing which in turn has created interest in the idea of an 'infinite game' (Extra Credits, 2015). The idea is that if a game can start creating personalized content for each player, the game could potentially be played forever.

2.2 Disadvantages of using Procedural Generation

2.2.1 Efficiency and Cost Saving

Ironically a major drawback of PCG can also be found in the efficiency and cost of development. Extra Credits have stated that "procedural generation is…an economy of scale" (2015) referencing the potential expense and time it could take to develop a fully functional and effective PCG framework/engine in relation to the amount of content that needed to be produced. If PCG was implemented into a game that only required a handful of levels to be created then the potential benefits will be outweighed by the costs and the resources used would have been better spent creating the content by hand (Extra Credits, 2015). Others such as Ra (2016) confirm this view but also stress that cost can be influenced by the choice to either develop a PCG system from scratch or to take advantage of third party software. Licensed software, while possibly expensive financially, has the potential to save a lot of development time in the long run (Ra, 2016).

Overall it seems that PCG can be both a help or a hindrance in this area depending on the situation into which it is implemented; ultimately the choice to utilise it must be made after the requirements for the project have been reviewed and it has been determined that it falls into the right place on the "economy of scale" (Extra Credits, 2015).

2.2.2 Generic Content

Once again, a perceived benefit of PCG can also be a major drawback; although Couture (2016) praises the potential for limitless content and emergent scenarios, it is just that – potential. If implemented incorrectly PCG can "create stages that are too hard, stages that are too easy or plain dull, stages that feel like repeats of each other, or just otherwise create an array of flavorless or frustrating experiences" (Couture, 2016). Others such as Schier (2015) confirm this view stating that creating visually appealling or generally interesting levels for a player to explore can be a difficult task even when they are designed manually; by offloading the design almost exclusively to a computer, the risk of creating a wealth of bland content increased dramatically (Schier, 2015).

In order to avoid this Couture (2016) recommends introducing constraints and rules that a PCG system would adhere to as it generated content to ensure that as a minimum the level was functional and hopefully contained elements of interest for the player. Rogers (2014) confirmed this view by stating that when utilising PCG it was important to be aware of the metrics such as player movement speed in order for the system to create a playable level. Furthermore, although the 'randomness' PCG offers can in itself be appealing, players need structure and consistency in order to learn how to 'navigate' and play a game (Rogers, 2014);

2.2.3 Lack of Control

A concern that is often brought up is the control that developers are sacrificing when using PCG. Depending on the chosen method of implementation it is possible that developers would be completely unable to alter the level that has been generated or that it would be difficult to create the same kind of engaging levels that are possible through manual creation (Extra Credits, 2015; Ra, 2016).

2.3 Guidelines for using Procedural Generation

2.3.1 Don't Overuse Randomness

The consensus seemed to be that although random elements can be of interest to the player, it is important to keep them to a minimum or at least to carefully selected areas to avoid the pitfall of bland content (Rogers, 2014). Many games that have been praised for their use of PCG such as *Enter the Gungeon* (Dodge Roll, 2016) and *The Binding of Isaac* (McMillen, 2011a) which used handmade rooms that were simply stitched together in a new order by the PCG system to generate levels, thus providing both a well-crafted and dynamic experience instead of the 'bland' experience cited by those who reviewed *No Man's Sky* (Hello Games, 2016). Extra Credits (2015) take this a step further recommending a mixed approach; in their example a PCG system would create a level around one manually created area for a boss encounter, allowing the journey to be different each time but the destination to remain consistent.

2.3.2 Know the Game's Metrics

As mentioned by Rogers (2014) and Couture (2016) it is imperative to know the rules and metrics of the game and to apply them to any PCG system that is implemented. This will at the very least limit the chance of a 'broken' level being produced where the player has no hope of completing it and has the potential to create consistent and interesting levels for the player to explore. Furthermore, it can give back some control to the developers that they lost by allowing them to guide the generation and alter the metrics/rules if any problems arise.

2.3.3 Create 'Subsets'

A guideline that ties into limiting the random elements of PCG, Couture (2016) describes a problem that can arise where a PCG system may have multiple elements it can use to create a level but the end result is "a mish-mash of varied parts" (Couture, 2016) where none feel tonally consistent or they are elements that just do not work effectively together. To avoid this situation a solution that was used in *Spelunky* (Mossmouth, 2013) was recommended whereby elements of the game are grouped into subsets and tied together (Couture, 2016). By grouping elements together in this manner, you can assure that they will only be deployed together and that they will always be relevant/tonally consistent (Ra, 2016). Furthermore, it allows developers to implement a feeling of progression by only allowing certain elements of the game to be deployed once a player reaches a certain point or when certain conditions are met.

2.4 Methods of implementing Procedural Generation

2.4.1 Cellular Automata

According to Shiffman (2012), cellular automata (CA) and the systems associated with it were attributed to both Stanisław Ulam and John von Neumann, two researchers at the New Mexico based Los Alamos National Laboratory in the 1940s. In the simplest terms, von Neumann was attempting to create a model for self-reproduction, focusing first on biology and then robotics before a suggestion from Ulam lead him to simplify his attempts into a 2D cell based structure (Wolfram, 2002). From this structure, CA could mimic “behavior similar to biological reproduction and evolution” (Shiffman, 2012, p. 325).

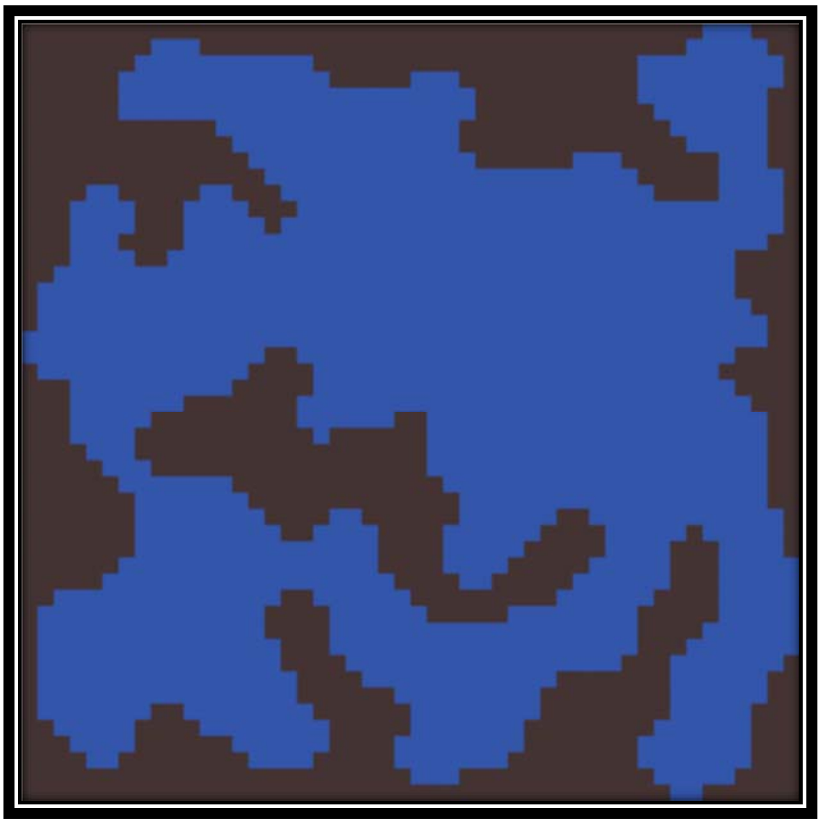


Figure 2 - A 2D cavern created using cellular automata (Cook, 2013)

Further development occurred in the 1970s when John Conway introduced “The Game Of Life” (Cook, 2013) a simplified version of CA that kept the cells in a 2D grid but reduced the potential twenty-nine states available to cells under van Neumann to only two states – either alive or dead (Shiffman, 2012). The simulation would begin by randomly assigning a state to each cell (based on some initial rules) and would then apply some simple rules to each cell to determine if it should change its state or not, usually based on its current state and the state of the cells that surround it, often called its neighbourhood (Cook, 2013).

When it comes to implementing the basics of CA the primary components are a 2D array of integers (whole numbers) which allows the creation of a 2D grid and stores the state of each cell, a seed value which allows the generation to be pseudo-random each time and multiple for loops in order to populate the 2D array and to check and apply rules to each cell that determine what state it should take (Shiffman, 2012)

Within the context of games development, the ‘natural’ shapes and formations that are created by CA are highly suitable for level generation for any areas based in caves/cavernous locations (Figure 2), especially when considering the alternative would be to create these environments by hand. The sheer variation available and organic feel of the generated maps is not something that could be easily replicated manually.

2.4.2 Perlin Noise

Perlin Noise (PN) is an algorithm that is used “for generating coherent noise over a space” (Zucker, 2001); put simply this means that “for any two points in the space, the value of the noise function changes smoothly” (Zucker, 2001). Whereas a typical random number generator produces results without an obvious pattern, Perlin Noise produces naturally ordered sequences of numbers as a result, creating a more ‘natural’ final product (Khan Academy, n.d.).

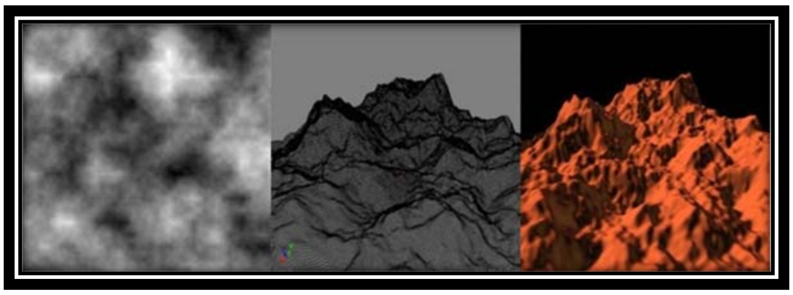


Figure 3 - The three stages of creating a landscape mesh from perlin noise; height map, mesh, rendered mesh (Tulleken, 2009)

The algorithm has many applications in both games development and other visual media such as films (Tulleken, 2009); in the past is was primarily used to procedurally generate and blend textures for items such as fire and clouds (Tulleken, 2009), however, Figure 9 shows it has also been used to procedurally generate a landmass by converting the noise into a height map and then generating a mesh from that data (Tulleken, 2009).

2.4.3 Procedural Generation and Manual Editing

PCG and manual editing is the process of taking a procedurally generated content as a basis and then manually editing it to make it suitable for use, for example, a developer could use a technique such as Perlin Noise to create a landmass and then, using the initial generation as a template, alter the terrain and manually add trees and cities etc. *Star Citizen* (Cloud Imperium Games, n.d.), an upcoming PC game, utilises this technique for generating the planets that players can explore. Whereas *No Man's Sky* (Hello Games, 2016) used PCG to create every planet without any input from a developer, *Star Citizen* (Cloud Imperium Games, n.d.) creates each planet using PCG and then adds fully designed missions and manually designs locations within the initial generation that the player will be able to complete and explore (Prescott, 2016). This technique offers a compromise by allowing multiple locations to be generated quickly during development without potentially compromising the content but it does mean that content cannot be created during runtime, meaning the content will always be the same for the player.

2.4.4 Modular Level and Component Design

Jaroslawsky (2013) described the core idea of modular design as being able to "produce many different results using high quality content generated only once", putting it simply, to create a lot from a little. As a method for content generation within games development, modular design has been utilised in several forms since early console games, typically through the creation of 2D tile sets that, when

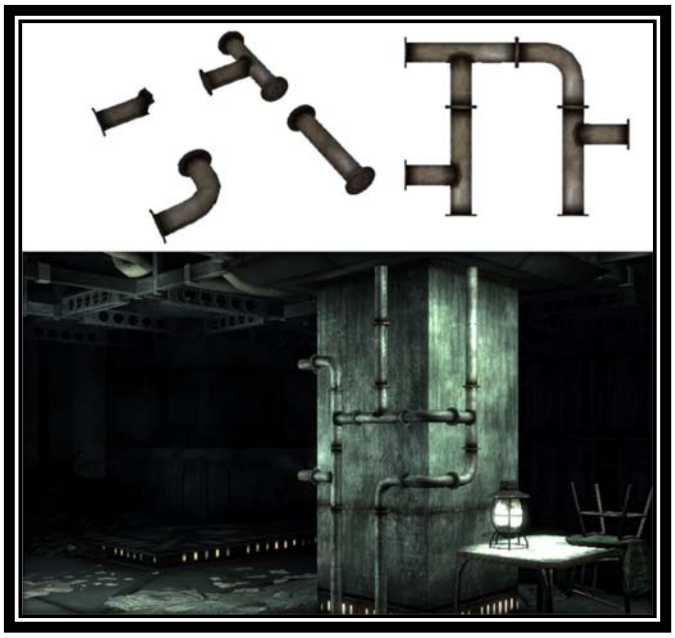


Figure 4 - A simple pipe kit, four modular pieces can create intricate designs (Burgess, 2013)

applied to a grid, would be used to construct varied levels from the same building blocks (Perry, 2002). Both *Super Mario Bros.* (Nintendo Research & Development 4, 1985) and *The Legend of Zelda* (Nintendo Research and Development 4, 1986) are early examples of such a method, using modular tiles in order to create their levels and obstacles (Jaroslawsky, 2013).

Modular design has also been used extensively within 3D games in recent years, mostly in response to demands from the audience for both an increased visual fidelity and an abundance of content (Perry, 2002). As previously mentioned, increased visual fidelity typically requires increased time and more artists which in turn raises development costs (Sherr, 2013). Furthermore, the creation of highly detailed, non-modular assets can potentially cause issues when changes are required due to a lack of flexibility in editing said assets (Perry, 2002).

The foundation of effective modular components can be divided into three areas; “consistent module size, optimal use of the grid and a precise concept” (Jaroslawsky, 2013). Establishing a ruleset that each component must adhere to so that it satisfies each area requires a developer to invest heavily in the design/conceptual phase with the benefits being reaped further down the line (Jaroslawsky, 2013). Adherence to all three areas usually produces components that are correctly scaled, easily and seamlessly placed on a grid and artistically/tonally similar (Jaroslawsky, 2013).

Burgess (2013) discussed how his team utilised modular design when developing both *Fallout 3* (Bethesda Game Studios, 2008) and *The Elder Scrolls V: Skyrim* (Bethesda Game Studios, 2011); the developers created multiple components following a set of design rules that ensured each asset would adhere to the three foundation areas of modular design (Jaroslawsky, 2013), allowing both consistency and seamless placements. Components would then be grouped into ‘kits’, allowing designers to easily pick the relevant components with ease and create a huge variety of levels by altering the placement and configuration of items from within the kit (Burgess, 2013). Figure 4 shows how a simple pipe kit consisting of only four art assets can be used to create multiple configurations within a level.

The modular nature of kits lends them readily to PCG as a properly implemented PCG system could generate nearly infinite configurations for a player to explore.

2.4.5 Room-Based Generation

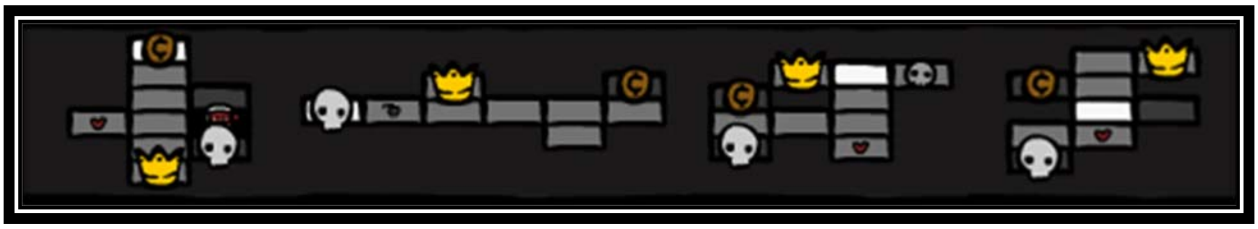


Figure 5 - Examples of layouts generated by The Binding of Isaac's PCG system (McMillan, 2011b)

A technique made popular in recent years due to several popular indie titles, most notably *The Binding of Isaac* (McMillen, 2011a) and *Enter the Gungeon* (Dodge Roll, 2016); Room-Based Generation (RBG) is another technique that attempts to remedy the potential in PCG for generic and repetitive content by combining pre-made rooms with an algorithm that 'stiches' them together (W., 2016). Figure 5 shows a selection of layouts created by the games level creation system. The primary difference between RBG and Manual Editing is that RBG can create new combinations of levels for the player to explore at runtime, potentially creating infinite content while Manual Editing is used during development. In essence, RBG is a form of modular design with each room being part of a level generation kit.

McMillen (2011b) described his motivation for using RBG as wanting to create a game that felt like a fresh experience with every playthrough but still wanted it to feel like it followed consistent rules and was mostly fair for the player (the use of random elements makes it near impossible to be entirely fair but values could be tweaked to make it as fair as possible). To do this McMillen (2011b) created a set of rules for his level creation system to follow: each level had to include four core rooms and then add 5-20 non-core rooms that are drawn from a pool of over 1000 pre-designed rooms. These pre-designed rooms were divided into subsets as per Couture (2016) based on their difficulty and what level of the basement the player was currently on.

PCG was not just limited to the level design, and was extended to influence the enemies the player would fight and the items they could find; McMillen (2011b) briefly described that each enemy had minor random elements, with a small percentage chance that a stronger variant would appear. The same could be said of the collectible, weapon and leveling items which were grouped into common, uncommon and rare with each assigned a percentage chance that they would spawn based on their group (McMillen, 2011b).

3 Design

3.1 Concept

As previously mentioned, the creation of a PCG system can be both an expensive and time consuming undertaking (Extra Credits, 2015). Furthermore, there was always the risk of creating generic/inferior content when compared to what could have been made manually (Couture, 2016; Rogers, 2014). The initial release of *The Binding of Issac* (McMillen, 2011a) was created by a team of two and started life in a week long game jam before being developed further for an additional three months (McMillen, 2012). Moreover, the games approach to PCG was highly praised when reviewed; Teti (2011) stated that every playthrough felt distinct, even after ten hours of playing, and praised the quality and quantity of the designed content. It appeared that the game had found the elusive balance of randomness and predesigned content.

Due to the time constraints of the project, the use of a method that had proven it could be implemented effectively by a small team in a short timeframe was highly desirable. As such, the concept of this project was established; the aim was to create a game in a similar vein to *The Binding of Isaac* (McMillen, 2011a) and other roguelikes such as *Enter the Gungeon* (Dodge Roll, 2016) using modular assets in a room-based generation system.

3.2 Metrics/Rulesets

A common approach adopted by several developers who used PCG was to establish a series of rules and goals that would be considered by their PCG system when generating content (Couture, 2016; Short, 2014). These metrics/rules allow the developer to take some control over the generation process, creating a barrier to creating generic content (Short, 2014). Furthermore, establishing metrics/rulesets early keeps development of the PCG system focused, preventing lost development time (Couture, 2016). With this in mind, the initial task set for this project was to create a set of goals that the PCG system would achieve to be considered successful.

1. Per Bidarra, Linden & Lopez (2014) the level created must reflect the design and asthetic of a dungeon or cavern as they lend themselves well to PCG.
2. The levels should be generated within an acceptable timeframe.
3. Each level must, within reason, create a new layout for the player to explore.

4. There should be a start point and a goal for the player to reach.
5. The goal would be placed as far from the start point as possible.
6. The player must be able to reach the goal from the start point (the PCG system will not make the goal inaccessible and render the game unwinnable/unplayable).
7. Following the recommendation of Short (2014), the level must contain optional areas and objectives for the player to explore and complete if they desire; this would encourage exploration and give the player further rewards.

3.3 Proposed Design

3.3.1 Required Components

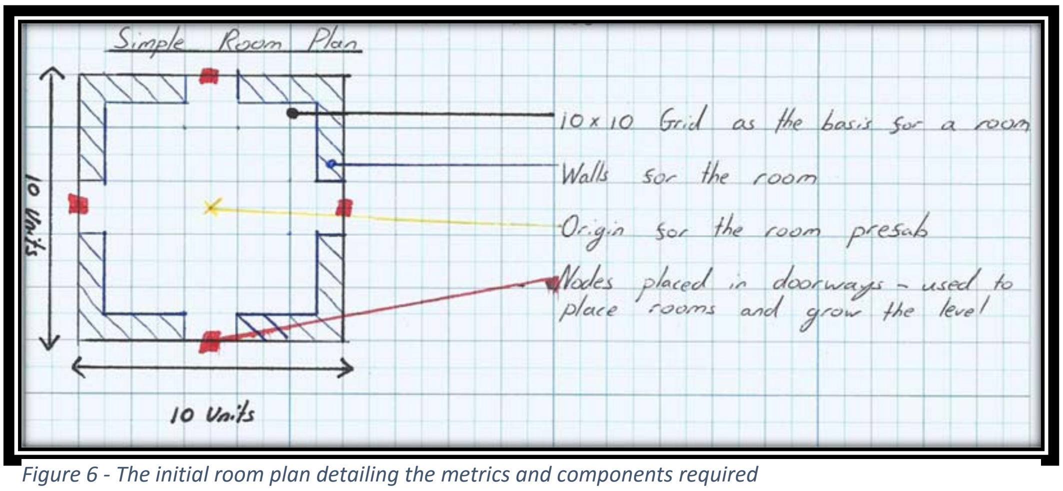
The creation of a level would require several elements:

- Level Manager – A globally accessible game object responsible for keeping track of the generation and altering the phase as required.
- Rooms – Premade, modularly designed rooms that would make up the level. The parent object of any nodes it contains.
- Nodes – A child object placed at the doorways of a room where new rooms could be spawned; responsible for all the initial room placements.

3.3.2 Room Designs

3.3.2.1 Overview

As the desired result was a level created from modular pieces, it was critical that time was spent designing each room so that they would fit together (Jaroslawsky, 2013). Figure 6 shows the initial room design including metrics for the room size and the components that each room would require. Each room was designed to be ten units by ten units in size; the room would be surrounded by walls of one unit thickness with any doorways being two units wide to allow enough space for the player to fit through. Each room would contain nodes at its doorway five units away from the central point; these nodes would be responsible for spawning additional rooms five units away from their current position.


Simple Room Plan
10x10 Grid as the basis for a room
Walls for the room
Origin for the room prefab
Nodes placed in doorways - used to place rooms and grow the level
10 Units
10 Units
Figure 6 - The initial room plan detailing the metrics and components required

3.3.2.2 All Room Types

Inspired by the modular approach of Burgess (2013), in particular the pipe kit seen in Figure 4, several room types were designed based on the template seen in Figure 6. Just like the pipe kit, rooms were designed to act as either a straight connection with two doorways and a single node, an elbow with two doorways and a single node, a tee connection with three doorways and two nodes or a blank connection with one doorway and no nodes, as seen in Figure 7.

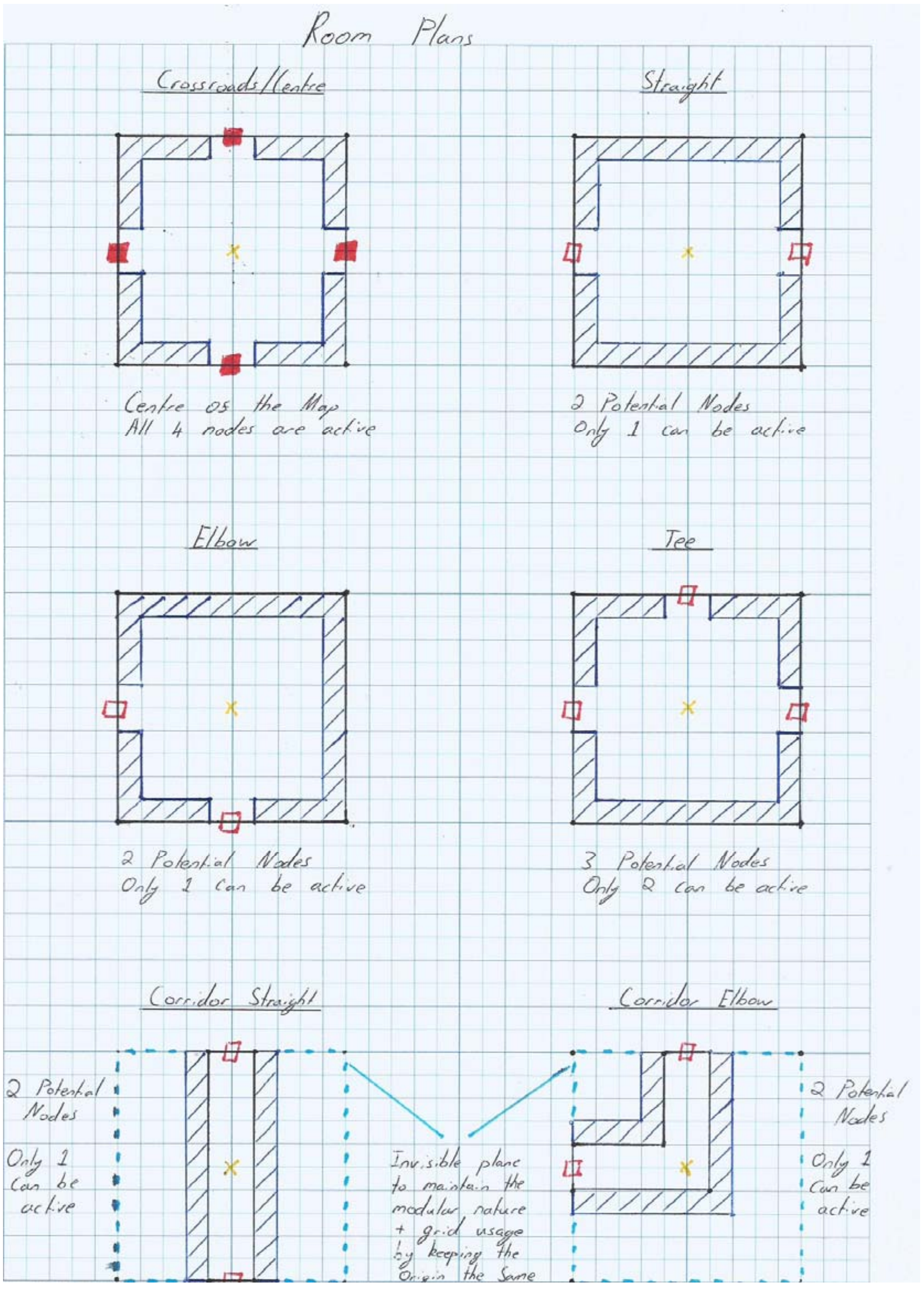


Figure 7 - Design for all room types

3.3.3 Level Generation

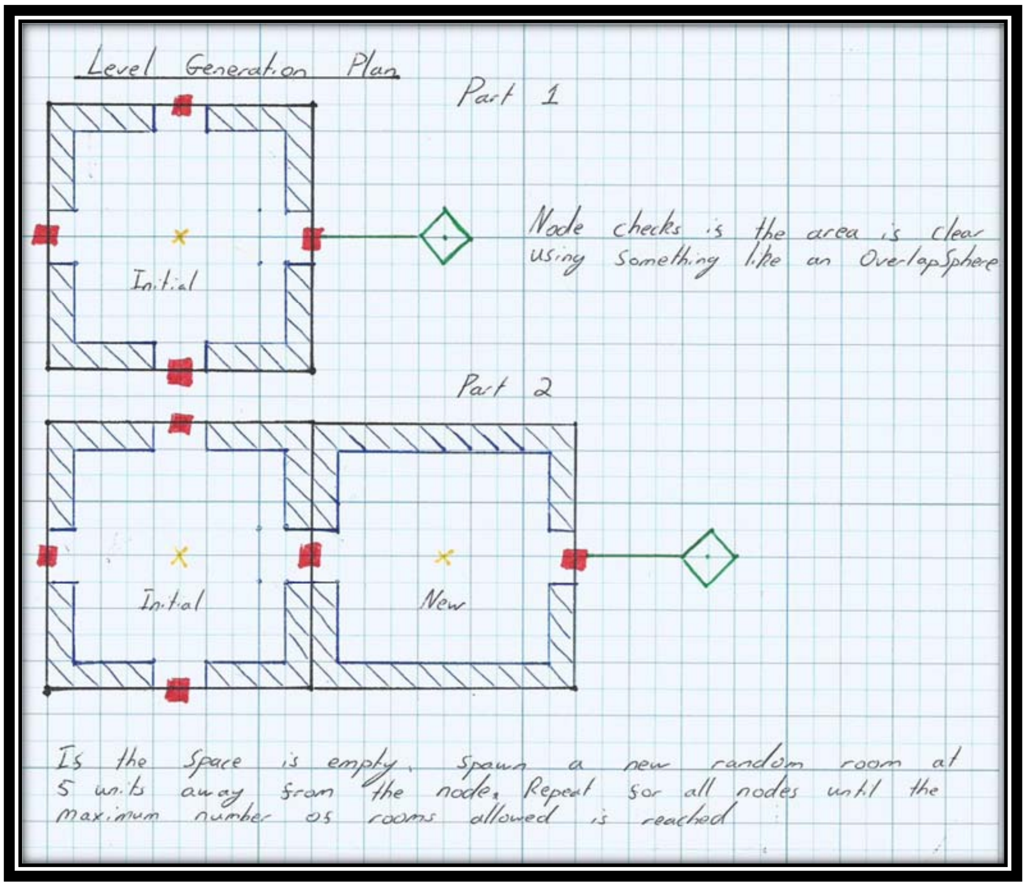

Level Generation Plan
Part 1
Node checks is the area is clear using Something like an OverlapSphere
Initial
Part 2
Initial
New
Is the Space is empty, Spawn a new random room at 5 units away from the node. Repeat for all nodes until the maximum number of rooms allowed is reached

Figure 8 - Level Generation Plan

3.3.3.1 Step One – Initial Room Placement

The PCG system places a starting room, selected from a list of potential starting rooms, at the centre of the map (within Unity this would be position 0, 0, 0). The centre room would contain a minimum of three doorways to allow the level to grow in multiple directions and encourage player exploration.

3.3.3.2 Step Two – Populating the map

Figure 8 shows how each room would contain a 'node' at each doorway with the ability to spawn a new room five units away, from a list of potential rooms, providing the following conditions have been met:

1. The desired space to create a room must be empty.
2. The node must not have spawned a room before.
3. The maximum number of rooms must not have been reached.

Once a room has been spawned it and the nodes that it contains would announce themselves to the level manager so that the total number of rooms and the status of each node could be tracked.

3.3.3.3 Step Three – Capping the map

Once the maximum number of rooms was reached, special rooms designated as 'Caps' would be placed by any nodes that indicate they have the space for a room and that have not placed a room. Cap rooms would contain only the one entrance from the previous room, producing a dead end. This would prevent creating incomplete levels with areas that a player could fall off.

3.3.3.4 Step Four – Safety Checks

Once the level has been capped, a check would be run to ensure that no rooms are overlapping; if any are, the level manager will remove one of the two. In theory, the check would be unnecessary as nodes would check for an empty space before placing a room and the planned modular design of the rooms means there would not be an overlap on a properly utilised grid, however, the importance of preventing errors that would prevent the player from completing the level makes safety checks a necessity.

4 IMPLEMENTATION

The following chapter discusses the implementation of the project; due to the limited word count only the parts of the implementation related to the PCG system are discussed as they were the primary focus of the dissertation.

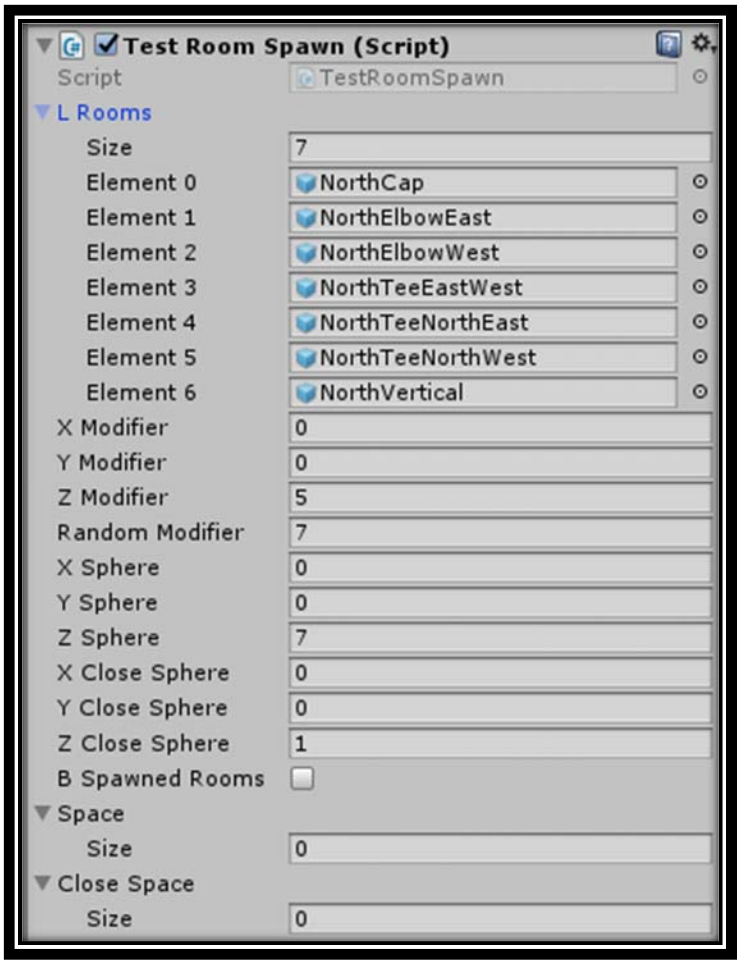
4.1 CREATING THE PREFABS

As previously mentioned, an important aspect of modular design is utilising a grid system and using consistent metrics across your assets (Jaroslawsky, 2013). The majority of modern game engines contain an inbuilt grid system by default (Jaroslawsky, 2013) and the Unity engine was no different. Each room was designed to be 10 units x 10 units (units being a generic word simply used to express the values that would be used in the engine) with any nodes placed in the doorway 5 units away from the origin (central point) of the room. This was easily implemented within the Unity engine by using the inbuilt shapes, creating a 10 x 10, 2D plane to be used as the floor and then simply manipulating 3D cubes into the appropriate shapes for walls. The nodes were created by making a small 3D cube and placing it in the appropriate location; while creating the PCG system the nodes were left as visible for easy access when troubleshooting however they were all made invisible for the final project.

To create the multiple variations of rooms required, the first room created was the central room with four exits which was then saved as what Unity calls a prefab asset type; when saved as a prefab multiple instance of the object can be created easily and any changes made to the prefab itself are applied to all instances (Unity Technologies, 2017). The central room prefab could be easily manipulated to create new rooms by creating an instance of it in the scene and then breaking its link to the prefab by deleting, moving and resizing elements such as the walls and nodes as required. Once finished, the new instance would be saved as a new prefab. The central room prefab was used as the basis for every room simply because it already had all the required elements for each variation and was far faster than creating each room entirely from nothing.

4.2 MODULAR SCRIPTS

Each node performed the same behaviour, spawning a new room, but each required certain elements to be tweaked depending on where they would be spawning said room; a node positioned on the east of the room would only need to spawn rooms with an east connection for example. In keeping with the modular approach and to deal with this issue, many important variables in the TestRoomSpawn script were made public so that they would be available in the Unity Editor (Figure 9). This allowed variables to be set in the editor rather than in the script, making the script itself a modular asset; each node utilised the same script but used different variables for values such as `xModifier` and `zModifier` to make sure it would spawn the rooms in the correct location. Each node would also have its list of potential rooms (`lRooms`) set in the editor to make sure that only compatible rooms would be spawned.



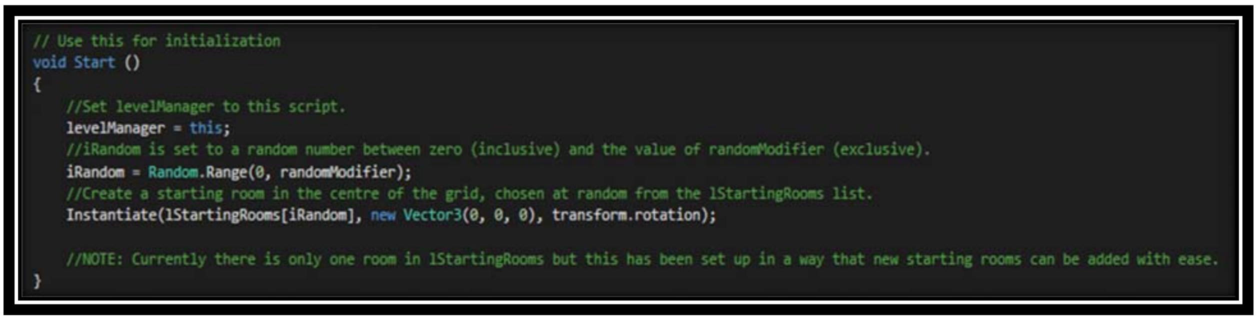
Test Room Spawn (Script)	
Script	TestRoomSpawn
L Rooms	
Size	7
Element 0	NorthCap
Element 1	NorthElbowEast
Element 2	NorthElbowWest
Element 3	NorthTeeEastWest
Element 4	NorthTeeNorthEast
Element 5	NorthTeeNorthWest
Element 6	NorthVertical
X Modifier	0
Y Modifier	0
Z Modifier	5
Random Modifier	7
X Sphere	0
Y Sphere	0
Z Sphere	7
X Close Sphere	0
Y Close Sphere	0
Z Close Sphere	1
B Spawned Rooms	☐
Space	
Size	0
Close Space	
Size	0

Figure 9 - Script found on a node as seen in the editor, values could be set here making the script modular

The alternative approach would have been to create four separate scripts, one for each node type and while this would have saved time spent setting each element in the editor for each node, it would have also meant changes to the script would not immediately be applied to all prefabs, making repeated tweaking and testing an issue.

4.3 Spawning the first room

Although the final project only included one starting room, the PCG system was designed to allow the inclusion of multiple starting rooms, adding additional variety to the level generation each time from the different number of and position of starting nodes. With this in mind, the obvious solution of simply placing the central room into the scene before runtime was not viable and so the responsibility of initial room placement was passed on to the LevelManager.

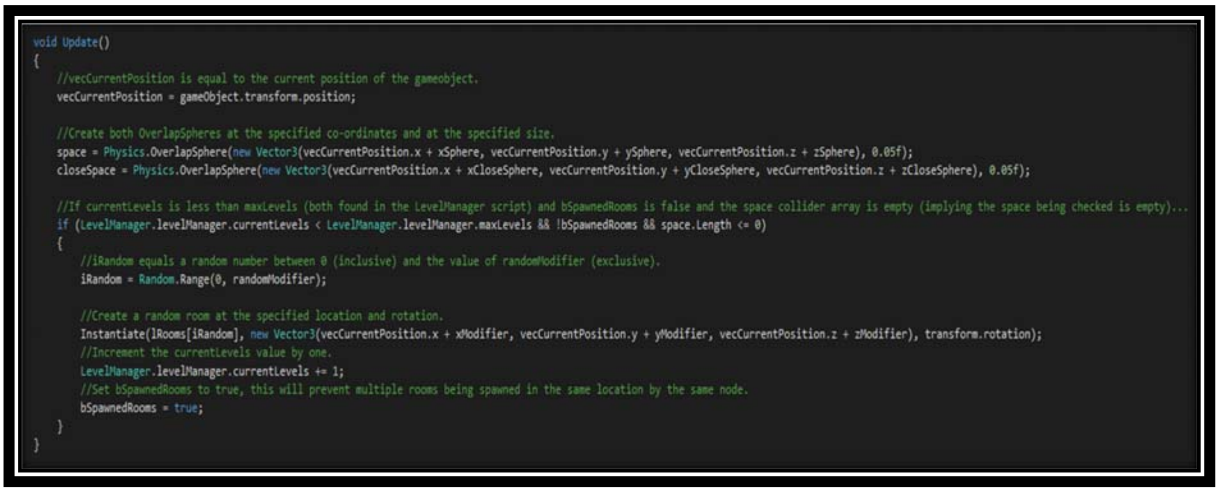


```
// Use this for initialization
void Start ()
{
    //Set levelManager to this script.
    levelManager = this;
    //iRandom is set to a random number between zero (inclusive) and the value of randomModifier (exclusive).
    iRandom = Random.Range(0, randomModifier);
    //Create a starting room in the centre of the grid, chosen at random from the lStartingRooms list.
    Instantiate(lStartingRooms[iRandom], new Vector3(0, 0, 0), transform.rotation);

    //NOTE: Currently there is only one room in lStartingRooms but this has been set up in a way that new starting rooms can be added with ease.
}
```

Figure 10 - LevelManager Start function containing the command to spawn the initial room

Spawning the initial room was accomplished within the Start function of the LevelManager script using two integer variables, `iRandom` and `randomModifier`, and a list of rooms called `lStartingRooms`. As can be seen in Figure 10, `iRandom` is first set to a random integer value between zero and whatever `randomModifier` has been set to in the editor; this is then used to instantiate (create/spawn) a room from the `lStartingRooms` list at the centre of the scene. The start function was chosen as the code only needed to be executed once.

4.4 GENERATING THE MAP



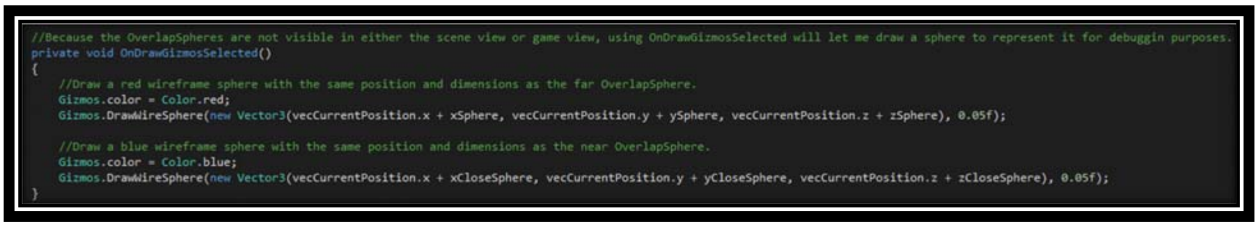
```
void Update()
{
    //vecCurrentPosition is equal to the current position of the gameobject.
    vecCurrentPosition = gameObject.transform.position;

    //Create both OverlapSpheres at the specified co-ordinates and at the specified size.
    space = Physics.OverlapSphere(new Vector3(vecCurrentPosition.x + xSphere, vecCurrentPosition.y + ySphere, vecCurrentPosition.z + zSphere), 0.05f);
    closeSpace = Physics.OverlapSphere(new Vector3(vecCurrentPosition.x + xCloseSphere, vecCurrentPosition.y + yCloseSphere, vecCurrentPosition.z + zCloseSphere), 0.05f);

    //If currentLevels is less than maxLevels (both found in the LevelManager script) and bSpawnedRooms is false and the space collider array is empty (implying the space being checked is empty)...
    if (LevelManager.levelManager.currentLevels < LevelManager.levelManager.maxLevels && !bSpawnedRooms && space.Length <= 0)
    {
        //iRandom equals a random number between 0 (inclusive) and the value of randomModifier (exclusive).
        iRandom = Random.Range(0, randomModifier);

        //Create a random room at the specified location and rotation.
        Instantiate(lRooms[iRandom], new Vector3(vecCurrentPosition.x + xModifier, vecCurrentPosition.y + yModifier, vecCurrentPosition.z + zModifier), transform.rotation);
        //Increment the currentLevels value by one.
        LevelManager.levelManager.currentLevels += 1;
        //Set bSpawnedRooms to true, this will prevent multiple rooms being spawned in the same location by the same node.
        bSpawnedRooms = true;
    }
}
```

Figure 11 - TestRoomSpawn Update function that spawns new rooms

Once the first room was created by the LevelManager, the remaining rooms were created using the nodes attached to any rooms that were already spawned. The principle remained the same as when spawning the initial room, a random value was assigned and then used to spawn a room from a list of potential rooms assigned to that node in the editor. The primary difference between the two was the number of checks required before a room could be spawned; whereas the initial room could just be spawned at the central point without any issues, all subsequent rooms required a variety of conditions be met before allowing a room to spawn, to prevent errors such as rooms spawning on top of each other or an infinite generation.

The first of three conditions checked was to compare the integer variable `currentLevels` (found in the LevelManager script) with `maxLevels` (also found in the LevelManager script). As previously mentioned, each room would 'announce' itself to the LevelManager by adding itself to the `lCurrentRooms` list and incrementing `currentLevels` by one. Comparing `currentRooms` with `maxRooms` before spawning additional prefabs added a cap in the level generation which prevented an infinite number of rooms from spawning and added a simple way to control the size of the level.

The final two conditions focused entirely on preventing rooms from spawning on top of each other and potentially causing issues such as inaccessible areas. The first was simply to use a Boolean value called `bSpawnedRooms` which was used to check if the node had already spawned a room and if so, it would not be allowed to spawn any more. The second check involved using an OverlapSphere to verify that there was space for a room to spawn at the proposed location. The OverlapSphere created an invisible sphere at the desired location and then returned an array of all the colliders that it touched or overlapped with. Checking that the area was empty could be accomplished by checking the size of the returned collider array; if it was empty then the space was empty and if it wasn't then the space was already occupied by another room.



```
//Because the OverlapSpheres are not visible in either the scene view or game view, using OnDrawGizmosSelected will let me draw a sphere to represent it for debuggin purposes.
private void OnDrawGizmosSelected()
{
    //Draw a red wireframe sphere with the same position and dimensions as the far OverlapSphere.
    Gizmos.color = Color.red;
    Gizmos.DrawWireSphere(new Vector3(vecCurrentPosition.x + xSphere, vecCurrentPosition.y + ySphere, vecCurrentPosition.z + zSphere), 0.05f);

    //Draw a blue wireframe sphere with the same position and dimensions as the near OverlapSphere.
    Gizmos.color = Color.blue;
    Gizmos.DrawWireSphere(new Vector3(vecCurrentPosition.x + xCloseSphere, vecCurrentPosition.y + yCloseSphere, vecCurrentPosition.z + zCloseSphere), 0.05f);
}
```

Figure 12 - OnDrawGizmosSelected was responsible for creating the OverlapSphere wireframe representation

As the OverlapSpheres were invisible in both the scene and game view, testing could have been made quite difficult; although it could be assumed they were in the right place it was preferential to know the exact location and size of them. This problem was overcome by using the `OnDrawGizmosSelected` function, shown in Figure 12, which allowed the creation of a wireframe sphere of the same size and in the same location as the OverlapSpheres to represent them. Only when all three conditions were met (checked using an if statement) would a new room be allowed to spawn using the same techniques described when spawning the initial room and shown in Figure 11.

4.5 SPAWNING THE GOAL ROOM AND CAPPING THE EDGES

When the desired number of rooms had been created (indicated by `currentLevels` being equal to, or greater than `maxLevels`) the system would add a special room designated the 'goal room'; this was the primary objective for the player to find in order to complete the level. Afterwards the system would cap the remaining, incomplete rooms that still had a doorway leading to nothing where the player could literally walk off the map as seen in Figure 13. Both tasks required very similar solutions and so were both implemented in the same function called `CapEndsOfMap().`

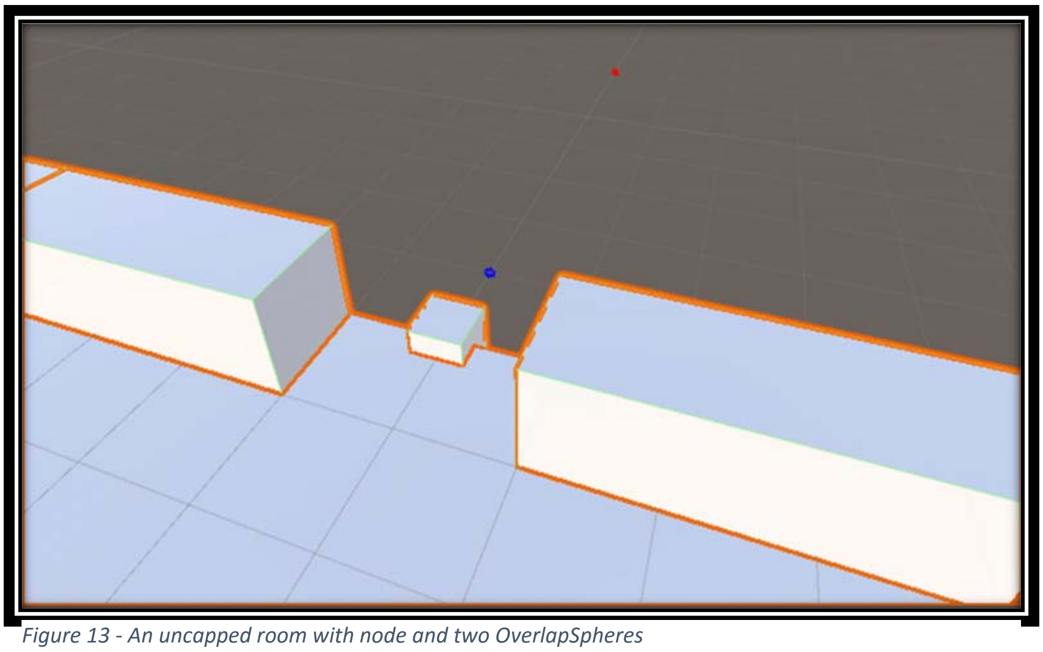


Figure 13 - An uncapped room with node and two OverlapSpheres

Placing the goal room required the use of another list within the LevelManager that contained all of the current nodes within the level called `lCurrentNodes`. In the same way a room would announce itself to the LevelManager when it was created, each node would add itself to `lCurrentNodes` upon spawning. Using this list in the `CapEndsOfMap()` function, the system would check for any uncapped nodes (nodes that had not spawned a room due to `maxLevels` being reached but did have the space available to spawn one) by using a for loop to iterate through every element of the list and an if statement to check the status of that elements version of `bSpawnedRooms`. If it was false then the system would assign `nodeTag`, `nodeObjectPosition` and `nodeScript` to the current elements tag, position and instance of the TestRoomSpawn script respectively. Each of these variables were used multiple times throughout the function and it was less expensive computationally (and also far easier to read) to create a reference to all the elements than to constantly use the expensive `GetComponent` function every time I needed to access one of the previously mentioned elements.

After each variable was assigned, an if statement would check the Boolean variable `bGoalRoomSpawned`; if that was currently false then it meant the goal room had not been created and so one would be spawned at the current element in the appropriate location. `bGoalRoomSpawned` would then be set to true to avoid multiple instances of the goal room from being created and the current elements instance of `bSpawnedRooms` would also be set to true, indicating that it should not be allowed to spawn any more rooms.

Spawning the level caps was done in a very similar way; `lCurrentNodes` would be checked for any uncapped nodes and then the relevant variables would be set as before. The script would then use a switch statement to compare `nodeTag` with a set of predefined constants and spawn the relevant cap room from the list of room caps called `lRoomCaps`. For example, if the `nodeTag` was "NorthConnection" then the north cap would be spawned.

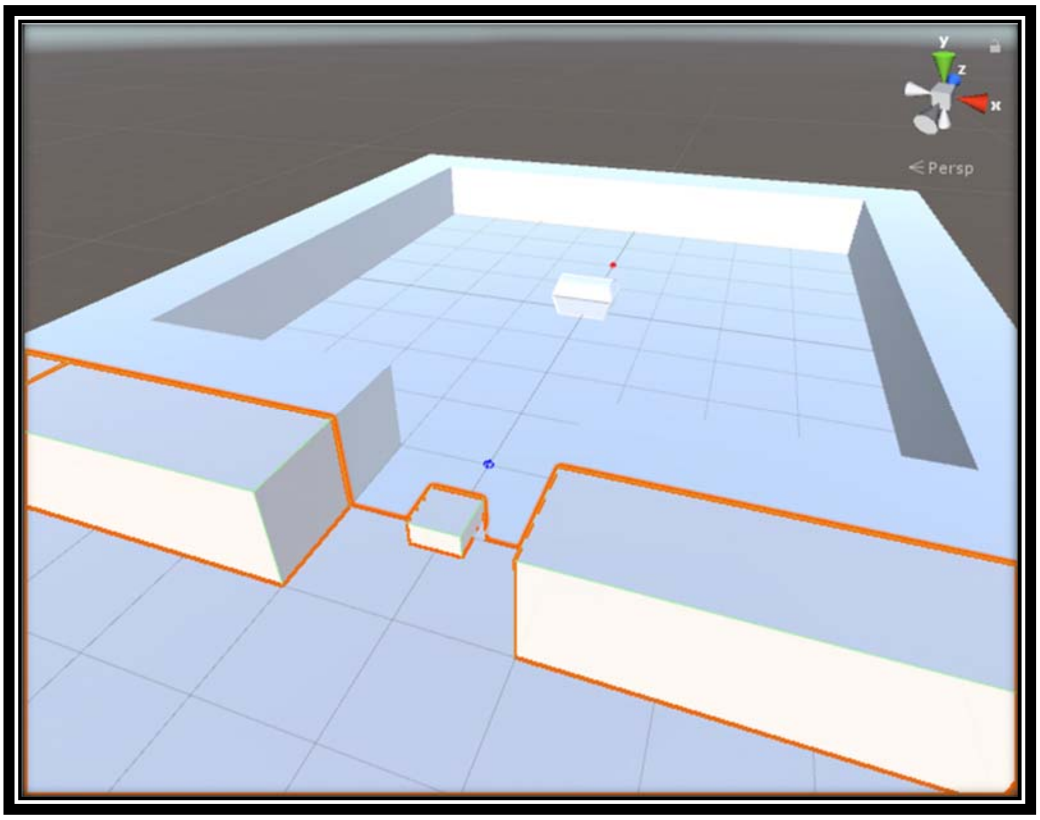
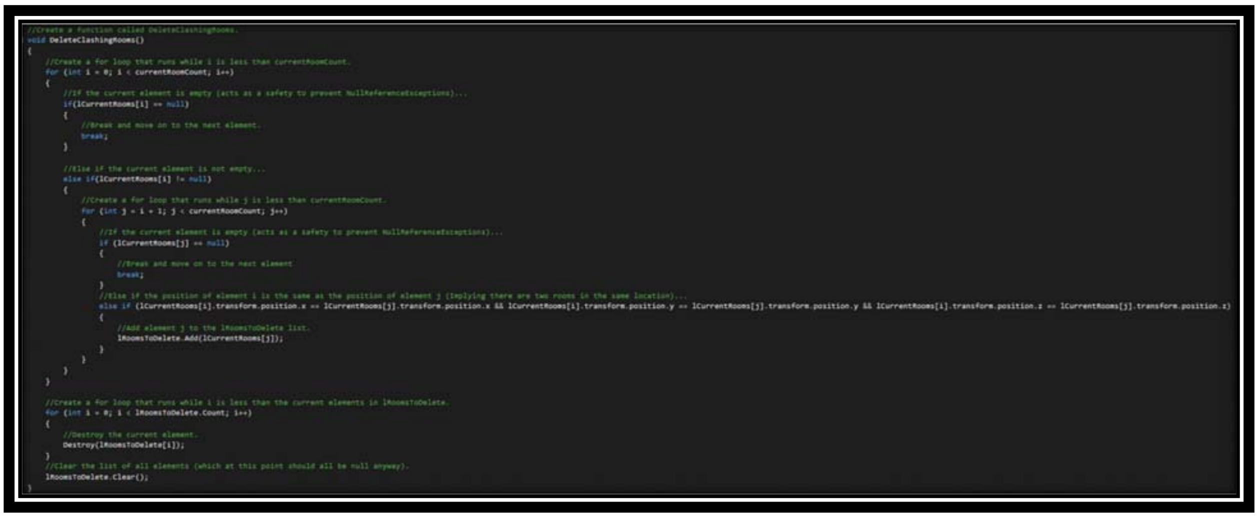


Figure 14 - The same room once CapEndsOfMap has been called

4.6 Deleting any clashing rooms

Despite using multiple checks to prevent rooms from spawning on top of each other it was still possible for it to occur. Furthermore, it is vital in any software development work to anticipate problems wherever possible and create a contingency for it (CITATION NEEDED). Due to modular nature of this PCG system and the fact that each room would always be placed with its origin at intervals of five, it was possible to detect clashing rooms by checking the current position of one room and comparing it with the positions of the remaining rooms; if the positions were exactly the same then they were clashing and one needed to be removed.



```
//Create a function called DeleteClashingRooms.
void DeleteClashingRooms()
{
    //Create a for loop that runs while i is less than currentRoomCount.
    for (int i = 0; i < currentRoomCount; i++)
    {
        //If the current element is empty (acts as a safety to prevent NullReferenceExceptions)...
        if(lCurrentRooms[i] == null)
        {
            //Break and move on to the next element.
            break;
        }

        //Else if the current element is not empty...
        else if(lCurrentRooms[i] != null)
        {
            //Create a for loop that runs while j is less than currentRoomCount.
            for (int j = i + 1; j < currentRoomCount; j++)
            {
                //If the current element is empty (acts as a safety to prevent NullReferenceExceptions)...
                if (lCurrentRooms[j] == null)
                {
                    //Break and move on to the next element
                    break;
                }
                //Else if the position of element i is the same as the position of element j (Implying there are two rooms in the same location)...
                else if (lCurrentRooms[i].transform.position.x == lCurrentRooms[j].transform.position.x && lCurrentRooms[i].transform.position.y == lCurrentRooms[j].transform.position.y && lCurrentRooms[i].transform.position.z == lCurrentRooms[j].transform.position.z)
                {
                    //Add element j to the lRoomsToDelete list.
                    lRoomsToDelete.Add(lCurrentRooms[j]);
                }
            }
        }
    }

    //Create a for loop that runs while i is less than the current elements in lRoomsToDelete.
    for (int i = 0; i < lRoomsToDelete.Count; i++)
    {
        //Destroy the current element.
        Destroy(lRoomsToDelete[i]);
    }
    //Clear the list of all elements (which at this point should all be null anyway).
    lRoomsToDelete.Clear();
}
```

Figure 15 - The DeleteClashingRooms function

A function was created called `DeleteClashingRooms`() that used a nested for loop to compare the position of each element of `lCurrentRooms` with every other element in the list; if the positions were the same then the second element would be added to a new list called `lRoomsToDelete`. Once every element had been checked, a new for loop would iterate through each element of `lRoomsToDelete` and delete every room it contained using the `Destroy()` function. Finally, `lRoomsToDelete` would be cleared, partially for neatness but primarily to avoid any errors if the list was accessed while it had null elements.

Although functional, this was an imperfect solution to the problem; ideally the system would have checked the rooms that were clashing and then compared them to the rooms surrounding them, keeping the one that was most suitable. Several attempts were made to implement such a system; originally a large OverlapSphere would be created at the centre of the clashing rooms and create a list of those plus the nine surrounding rooms. This proved ineffectual for two reasons, the first being that the OverlapSphere would also include all the walls and nodes in its list too, making it very difficult to compare just the rooms. While it was possible to either remove the excess elements using a for loop or to make the OverlapSphere ignore them entirely by placing them on a different physics layer, the second problem still prevented this being an effective solution; there were so many combinations of rooms that it would have been a massive undertaking to account for every possibility. In the end, although not ideal, the simplest solution was implemented as it was deemed that a simple if imperfect one was preferable to the risk of creating an imperfect level.

4.7 Adding chests

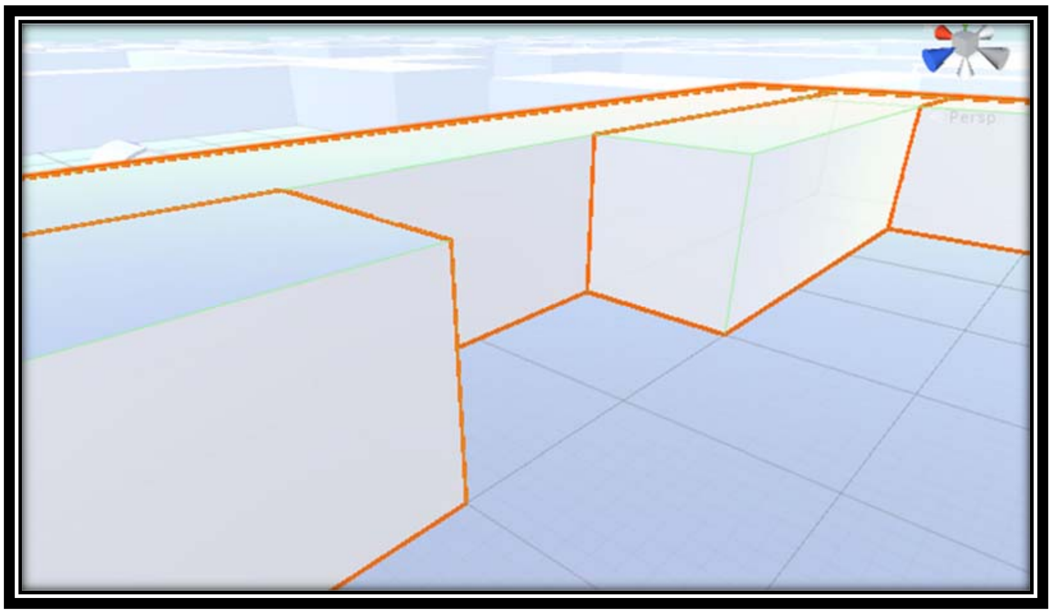
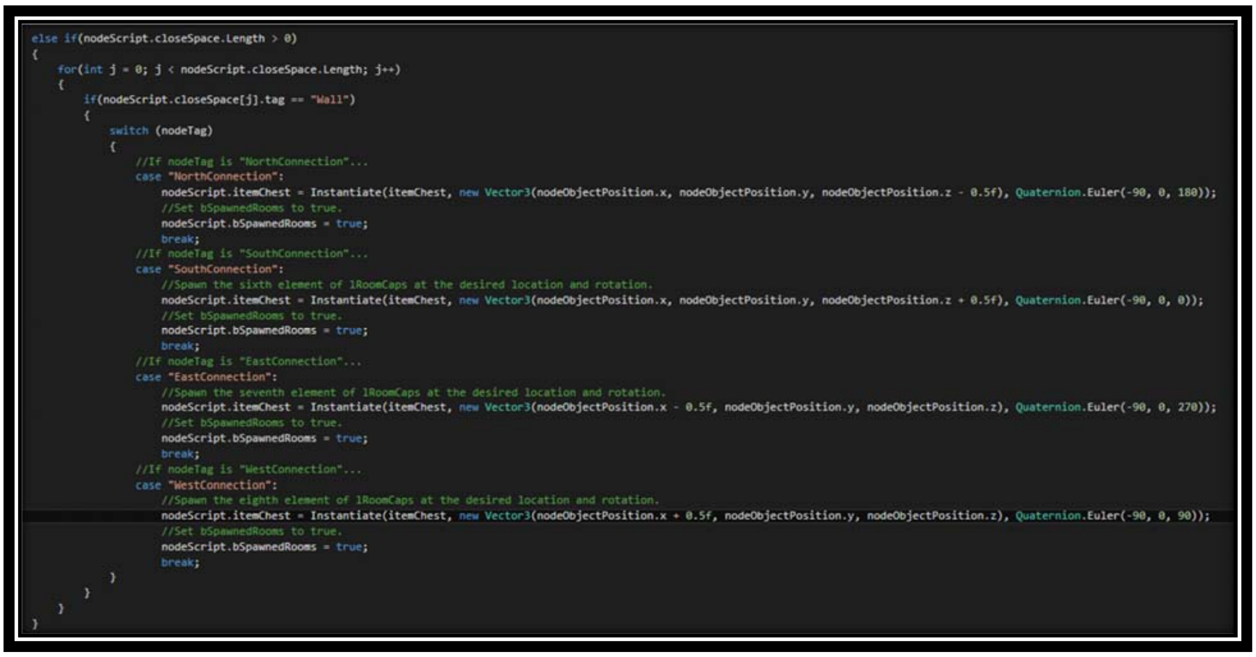


Figure 16 - A 'dead-end doorway'

To encourage exploration, chests containing items would be place throughout the level; to avoid making the player feel like a dead-end was a waste of their time, each room cap would contain a chest by default. Another opportunity for placing additional chests was discovered when searching for a solution to what was dubbed a ‘dead-end doorway’; once the level generation was completed there were rooms that contained a doorway which was blocked by a wall of the room next to it as seen in Figure 16. These areas could have simply been filled in with a wall prefab but instead were used as additional areas for a chest to be placed.



```
else if(nodeScript.closeSpace.Length > 0)
{
    for(int j = 0; j < nodeScript.closeSpace.Length; j++)
    {
        if(nodeScript.closeSpace[j].tag == "Wall")
        {
            switch (nodeTag)
            {
                //If nodeTag is "NorthConnection"...
                case "NorthConnection":
                    nodeScript.itemChest = Instantiate(itemChest, new Vector3(nodeObjectPosition.x, nodeObjectPosition.y, nodeObjectPosition.z - 0.5f), Quaternion.Euler(-90, 0, 180));
                    //Set bSpawnedRooms to true.
                    nodeScript.bSpawnedRooms = true;
                    break;
                //If nodeTag is "SouthConnection"...
                case "SouthConnection":
                    //Spawn the sixth element of lRoomCaps at the desired location and rotation.
                    nodeScript.itemChest = Instantiate(itemChest, new Vector3(nodeObjectPosition.x, nodeObjectPosition.y, nodeObjectPosition.z + 0.5f), Quaternion.Euler(-90, 0, 0));
                    //Set bSpawnedRooms to true.
                    nodeScript.bSpawnedRooms = true;
                    break;
                //If nodeTag is "EastConnection"...
                case "EastConnection":
                    //Spawn the seventh element of lRoomCaps at the desired location and rotation.
                    nodeScript.itemChest = Instantiate(itemChest, new Vector3(nodeObjectPosition.x - 0.5f, nodeObjectPosition.y, nodeObjectPosition.z), Quaternion.Euler(-90, 0, 270));
                    //Set bSpawnedRooms to true.
                    nodeScript.bSpawnedRooms = true;
                    break;
                //If nodeTag is "WestConnection"...
                case "WestConnection":
                    //Spawn the eighth element of lRoomCaps at the desired location and rotation.
                    nodeScript.itemChest = Instantiate(itemChest, new Vector3(nodeObjectPosition.x + 0.5f, nodeObjectPosition.y, nodeObjectPosition.z), Quaternion.Euler(-90, 0, 90));
                    //Set bSpawnedRooms to true.
                    nodeScript.bSpawnedRooms = true;
                    break;
            }
        }
    }
}
```

Figure 17 - The code to place additional chests in the CapEndsOfMap function

Figure 17 shows the code that was added to the `CapEndsOfMap()` function to handle placing additional chests as the behaviour was very similar to the behaviour already used in this function; the `lCurrentNodes` list was checked for uncapped nodes (`bSpawnedRooms` set to false but with space for a room), in this instance however, the list would be checked for nodes where `bSpawnedRooms` was set to false and did not have space for a room according to both OverlapSpheres. When these nodes were found, an instance of the chest prefab would be spawned at the location of the node, as seen in Figure 18, and `bSpawnedRooms` would be set to true. This was accomplished by using a for loop to iterate through the list and an if statement to check for the correct conditions.

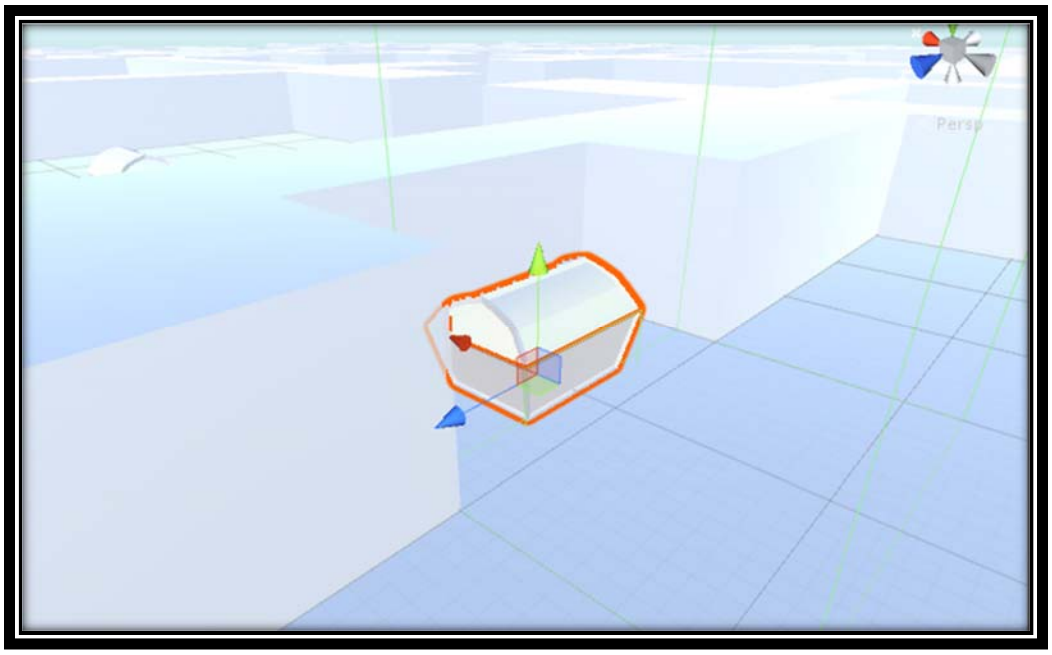


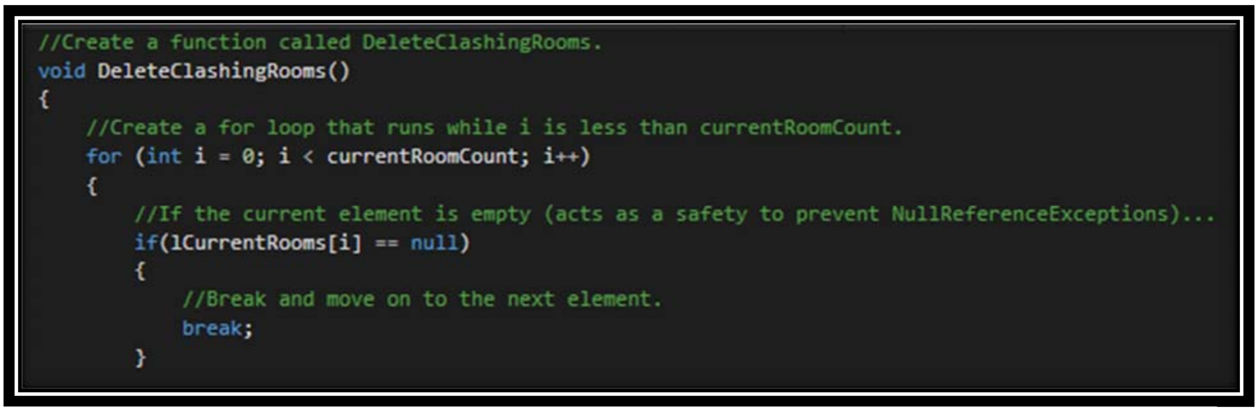
Figure 18 - the 'dead-end doorway' once a chest was spawned

5 Testing and Evaluation

5.1 Testing Log

Early in development, a testing log (Appendix D) was created to document any testing that took place, the bugs that were found and the actions taken to fix them. As there was a substantial random element to the project each test was repeated multiple times to make sure that the results were consistent across multiple generations. If a bug occurred the details were noted but due to the way the system had been designed it was impossible to replicate the conditions that caused it exactly, the best that could be achieved was either to continue creating generations and hoping a similar situation would occur or manipulating the metrics available to force a similar situation. This added some time into testing that could have been better spent further developing the project.

5.1.1 Safety Checks



```
//Create a function called DeleteClashingRooms.
void DeleteClashingRooms()
{
    //Create a for loop that runs while i is less than currentRoomCount.
    for (int i = 0; i < currentRoomCount; i++)
    {
        //If the current element is empty (acts as a safety to prevent NullReferenceExceptions)...
        if(lCurrentRooms[i] == null)
        {
            //Break and move on to the next element.
            break;
        }
```

Figure 19 - An example of a safety to prevent NullReferenceException errors

While performing tests it became apparent that a solution was needed for NullReferenceExceptions. This error became a common occurrence throughout development as the PCG system utilised lists regularly. Initial attempts to address the issue were imperfect and unreliable, only reducing the amount of errors that occurred instead of eliminating them entirely. The final solution was far simpler than previous attempts and highlighted an important oversight in the programming methods of the project, there were no safety checks to act as preventative measures. In the case of NullReferenceExceptions, simply checking if the chosen element was currently null (and if so, move on to the next element) was the best solution, as seen in Figure 19. This method of identifying potential errors before they happened and creating a simple preventative check was adopted for the remainder of the project.

5.2 User Testing

In order to obtain feedback and increase the number of generations that were tested, two playtest sessions were held with four participants. Each participant was given a questionnaire to complete once they finished their session. Both playtest sessions involved the same five participants in order to gauge their responses once changes had been made based on their feedback.

5.2.1 Adding Corridors

An area of feedback that all testers were unanimous on was the appeal of each level generation which confirmed previous suspicions seen in the testing log. Although the level generation was unique in the sense that different rooms would spawn in different configurations each time, the consensus was that the shape of the level was generally the same, a large rectangle. Corridors had been considered in the design but had not yet been implemented. When revisiting the design to find a solution to this problem the corridors seemed to be an appropriate approach as the difference in size when compared to a room would offer a distinct visual difference when generated together.

Corridor prefabs were created for each direction and then added into the level generation system with relative ease. Upon completing a second round of testing with the same participants, the responses were far more positive as the corridors added a great deal more variety to the overall shape of the level.

5.2.2 Adding Small Caps

Adding corridors added an unforeseen challenge for the `CapEndsOfMap()` function. The OverlapSphere used to determine if there was space for a cap room to be placed was at the centre point of where said room would be placed. Due to the difference in size between a corridor prefab and a room prefab it was entirely possible for a node to think it was already capped when it wasn't as it was detecting the corridor in the OverlapSphere instead of an empty space. A normal cap room would clash with the corridor if placed in these locations so a new, smaller cap prefab was created and a second OverlapSphere was added to each node. The second OverlapSphere was positioned closer to the node and was checked in the event that the first OverlapSphere was not empty. If the second OverlapSphere was empty then it indicated that there was space for a small cap to be placed.

5.2.3 Bugs

One participant brought a rather serious bug to light, stating that they encountered a level that could not be completed as the goal room had not spawned. Further investigation revealed that in rare situations the level would produce too many cap rooms and stunt the growth of the level. This in turn prevented `currentLevels` from becoming greater than (or equal to) `maxLevels` which was the requirement for all the LevelManager functions to run.

A simple fix was implemented that would force `currentLevels` to become greater than `maxLevels` after five seconds passed which would then trigger the relevant functions. While a brute force solution it severely reduced the number of unwinnable levels that were produced. There was still the potential for a level to be produced that did not have any space for a goal room to spawn even with the fix. After another investigation, it was concluded that the best way to prevent this issue was to either remove the nodes ability to produce level caps or to reduce the chance of a cap room from being chosen.

Ultimately it was concluded that removing the caps from the initial generation was undesirable as it would limit the potential of the initial layout, instead the chances of one appearing were simply reduced; it was still possible for one to appear but it was more likely another room or corridor would be picked instead.

6 CONCLUSION

6.1 LIMITATIONS OF THE DISSERTATION

6.1.1 Scope

Extra Credits (2015) described PCG as an economy of scale in reference to the costs of development verses the potential benefits. They expanded on this by recommending that before any attempt was made to create a PCG system, the developers should first set goals for the system to accomplish and review if the development time and cost would be worth the investment. This advice was taken to heart and the scope of the project was designed to be accomplishable within the set timeframe. Unfortunately, the difficulty of implementing a PCG system was underestimated and it became clear partway through development that the scope of the project was too large. Due to this, some aspects of the initial design, primarily aspects of the gameplay, were removed to allow further development of the PCG system, the primary focus of the project.

6.1.2 Gameplay

Several aspects of the gameplay were reduced in scope or removed due to the time required to develop the PCG system. It was deemed appropriate to scale these elements back as they were not the primary focus of the project.

6.1.3 User Testing

The number of participants in the user testing was far from perfect; ideally there would have been a minimum of twenty participants to allow for multiple generations for bug testing and a large amount of feedback. Despite the small sample size the feedback gained was invaluable; for example, without it, it is entirely possible that unwinnable generations would have been far more common.

6.2 STRENGTHS OF THE DISSERTATION

6.2.1 Lessons learned

During the development of the project, several lessons were learned in various areas whether it was PCG techniques or a new feature of the Unity engine that could be utilised in future projects:

- Utilising planning documentation is vital for any software development, keeping development focused by providing clear goals/milestones and helping to avoid additional problems/errors.

- Establishing a ruleset/metrics for the PCG system to follow made testing easier as there was always a clear standard to compare the results of the test to.
- Anticipating any potential problems within a script and adding a 'safety check' to prevent them saves time and is good practice; several errors encountered during development, particularly the NullReferenceException when accessing a list could have been prevented by adding a simple check before attempting to access the element, as seen in Figure 19.
- The use of modular/reusable code was both time and resource efficient and is a technique that will be utilised in the future.
- Source Control should always be used without any exceptions to prevent loss of progress and to maintain the ability to roll back any mistakes with ease.
- Expanded knowledge of the Unity Engine in various areas such as OverlapSpheres.

6.3 Evaluation

An initial goal of the dissertation was to research PCG systems, their uses within the games industry and the various methods of implementation available. It is felt that this was mostly successful as shown in Chapter 2 with multiple uses and methods of implementation being explored and evaluated. Despite this, it is also felt that an argument can be made suggesting the research was generally too broad, only providing a summary of each method instead of digging into the details. Perhaps a more suitable approach would have been to select fewer methods to focus on in greater detail.

The primary aim of the dissertation was to create a functional prototype that utilised PCG within the level design and for gameplay elements. In this respect, the dissertation was a mixed success; a level generation system was successfully implemented and was deemed suitable for purpose; however, gameplay elements were not highly prioritised, the majority of development time was spent designing, creating and iterating on the PCG system (itself a large undertaking) as this was the primary focus of the project. Unfortunately, this left the primary aim only partially fulfilled (although it could be argued that the fulfilled aspect was the most important of the two in relation to the purpose of the dissertation).

The secondary aim was to evaluate the effectiveness of PCG as a tool for games development; based on the research, it has been concluded that while PCG has suffered a hit to its reputation in recent years (Baines, 2016; Sterling, 2016), it is not the fault of the tool but rather those who wield it. When used effectively PCG has the potential to create critically acclaimed games such as *The Binding of Isaac*

(McMillan, 2011a) or create dynamic experiences as seen in the nemesis system (Monolith Productions, 2014). PCG still has the potential to be a revolutionary tool within the games industry (Extra Credits, 2015) and it seems its potential has yet to be fully explored.

6.4 Future Work

If work was to be continued on this project without any time constraints then the main focus would be to expand the prototype into a full game, further expanding and developing both the PCG system and the gameplay elements. McMillan (2011b) attributed the success of *The Binding of Isaac* (McMillan 2011a) partially to the sheer variety and quality of the content it provided; multiple room types and enemy types ensured that no one would play the same way twice. Due to this, further rooms and corridors of various types would be added to increase the variety in the level generation and the gameplay would be expanded by adding additional items, enemy types and weapons for the player to interact with. A further addition would be to implement audio (both SFX and music) into the project as this was omitted to make more time to refine the PCG system.

7 REFERENCES

A.I. Design. (1980). Rogue. San Francisco, CA: Epyx.

Baines, T. (2016, August 31). *Maths, No Man's Sky, and the problem with procedural generation.* Retrieved from Thumsticks: http://www.thumbsticks.com/no-mans-sky-problem-procedural-generation/

Barton, M., & Loguidice, B. (2009, May 5). *The History of Rogue: Have @ You, You Deadly Zs.* Retrieved from Gamasutra: http://www.gamasutra.com/view/feature/132404/the_history_of_rogue_have__you_.php

Bethesda Game Studios. (2008). Fallout 3. Rockville, MD: Bethesda Softworks.

Bethesda Game Studios. (2011). The Elder Scrolls V: Skyrim. Rockville, MD: Bethesda Softworks.

Bidarra, R., Linden, R. v., & Lopez, R. (2014). Procedural Generation of Dungeons. *IEEE Transactions on Computational Intelligence and AI in Games, Vol.6, No. 1*, 78-89.

Booth, M. (n.d.). *The AI Systems of Left 4 Dead.* Retrieved from Valve Software: http://www.valvesoftware.com/publications/2009/ai_systems_of_l4d_mike_booth.pdf

Boucher-Vidal, G. (2014, July 31). *What does it really cost to open an indie studio? All your money, most of your life.* Retrieved from Polygon: http://www.polygon.com/2014/7/31/5949433/the-cost-of-a-game-studio

Bungie. (2014). Destiny. Bellevue, Washington: Activision.

Burgess, J. (2013, May 1). *Skyrim's Modular Approach to Level Design.* Retrieved from Gamasutra: http://www.gamasutra.com/blogs/JoelBurgess/20130501/191514/Skyrims_Modular_Approach_to_Level_Design.php

Cellar Door Games. (2013). Rogue Legacy. Toronto: Cellar Door Games.

Chu, C. Y., Harada, T., Kaidan, M., & Thawonmas, R. (2015). Procedural Generation of Angry Birds Levels That Adapt to the Player's Skills Using Genetic Algorithm. *IEEE 4th Global Conference on Comsumer Electronics (GCCE)*, 535-536.

Cloud Imperium Games. (n.d.). Star Citizen. Los Angeles, CA: Cloud Imperium Games.

Compulsion Games. (2016). We Happy Few. Montreal: Compulsion Games.

Cook, M. (2013, July 23). *Generate Random Cave Levels Using Cellular Automata.* Retrieved from Envatotuts+: https://gamedevelopment.tutsplus.com/tutorials/generate-random-cave-levels-using-cellular-automata--gamedev-9664

Couture, J. (2016, May 3). *The pros and cons of procedural generation in Overland.* Retrieved from Gamasutra: http://www.gamasutra.com/view/news/271814/The_pros_and_cons_of_procedural_generation_in_Overland.php

Daybreak Game Company. (2016). Landmark. San Diego, CA.

Dodge Roll. (2016). Enter the Gungeon. Austin, Texas: Devolver Digital.

Extra Credits. (2015, July 22). *Procedural Generation - How Games Create Infinite Worlds - Extra Credits [Online Video].* Retrieved from Youtube: https://www.youtube.com/watch?v=TgbuWfGeG2o

Flafla2. (2014, August 9). *Understanding Perlin Noise.* Retrieved from adrian's soapbox: http://flafla2.github.io/2014/08/09/perlinnoise.html

Hello Games. (2016). No Man's Sky. Guildford: Hello Games.

Jaroslawsky, H. (2013, February 8). *Modular level design: A round up of the basics for budding level designers.* Retrieved from SAE Alumni: http://alumni.sae.edu/2013/02/08/modular-level-design-a-round-up-of-the-basics-for-budding-level-designers/

Khan Academy. (n.d.). *Perlin noise.* Retrieved from Khan Academy: https://www.khanacademy.org/computing/computer-programming/programming-natural-simulations/programming-noise/a/perlin-noise

Kollar, P. (2016, August 12). *No Man's Sky Review.* Retrieved from Polygon: http://www.polygon.com/2016/8/12/12461520/no-mans-sky-review-ps4-playstation-4-pc-windows-hello-games-sony

Lee, J. (2014, November 26). *How Procedural Generation Took Over The Gaming Industry.* Retrieved from MakeUseOf: http://www.makeuseof.com/tag/procedural-generation-took-gaming-industry/

Lee, K., & Lee, T. (2014, March 17). *Rogue Legacy Design Postmortem: Budget Development [Online Video].* Retrieved from GDC Vault: http://gdcvault.com/play/1020541/Rogue-Legacy-Design-Postmortem-Budget

Livingston, C. (2016, August 18). *No Man's Sky Review.* Retrieved from PC Gamer: http://www.pcgamer.com/no-mans-sky-review/

MathSphere. (2014). *MathSphere Graph & Line Paper.* Retrieved from MathSphere: http://www.mathsphere.co.uk/resources/MathSphereFreeGraphPaper.htm

McMillen, E. (2011a). The Binding of Isaac. Santa Cruz, CA: Edmund McMillen.

McMillen, E. (2011b, September 11). *The Binding of Isaac Gameplay explained.* Retrieved from Edmund McMillen: http://edmundmcmillen.blogspot.co.uk/2011/09/binding-of-isaac-gameplay-explained.html

McMillen, E. (2012, November 28). *Postmortem: McMillen and Himsl's The Binding of Isaac.* Retrieved from Gamasutra: http://www.gamasutra.com/view/feature/182380/postmortem_mcmillen_and_himsls_.php

Mojang. (2011). Minecraft. Stockholm: Mojang.

Monolith Productions. (2014). Middle-Earth: Shadow of Mordor. Kirkland, WA: Warner Bros. Interactive Entertainment.

Mossmouth. (2013). Spelunky. San Francisco, CA: Mossmouth.

Nelson, M. J., Shaker, N., & Togelieus, J. (2016). *Procedural Content Generation in Games: A Textbook and an Overview of Current Research.* Cham: Springer. Retrieved from Procedural Content Generation in Games.

Nintendo Research & Development 4. (1985). Super Mario Bros. Kyoto, Japan: Nintendo.

Nintendo Research and Development 4. (1986). The Legend of Zelda. Kyoto, Japan: Nintendo.

Perry, L. (2002, November). *Modular Level and Component Design.* Retrieved from Unreal Engine: https://docs.unrealengine.com/udk/Three/rsrc/Three/ModularLevelDesign/ModularLevelDesign.pdf

Placzek, M. (2016, November 23). *Object Pooling in Unity.* Retrieved from raywenderlich.com: https://www.raywenderlich.com/136091/object-pooling-unity

Prescott, S. (2016, October 11). *Star Citizen's procedurally generated planets showcased at CitizenCon [Updated].* Retrieved from PC Gamer: http://www.pcgamer.com/star-citizens-procedurally-generated-planets-showcased-at-citizencon/

Ra, F. (2016, June 08). *Procedural Generation: Pros and Cons.* Retrieved from gamedev.net: http://www.gamedev.net/page/resources/_/technical/graphics-programming-and-theory/procedural-generation-pros-and-cons-r4362

Rockstar North. (2013). Grand Theft Auto V. Edinburgh, Scotland: Rockstar North.

Rogers, S. (2014). *Level Up! The Guide to Great Video Game Design* (2nd ed.). Chichester: Wiley.

Schier, G. (2015, October 23). *Pros and Cons of Procedural Level Generation.* Retrieved from schier: https://schier.co/blog/2015/10/23/pros-and-cons-of-procedural-level-generation.html

Sherr, I. (2013, June 11). *Developers Defeat Rising Game Costs.* Retrieved from Wall Street Journal: http://search.proquest.com.voyager.chester.ac.uk/docview/1366282725?pq-origsite=summon&accountid=14620

Shiffman, D. (2012). *The Nature of Code.* Daniel Shiffman. Retrieved from The Nature of Code.

Short, T. (2014, April 02). *Level Design in Procedural Generation.* Retrieved from Gamasutra: http://www.gamasutra.com/blogs/TanyaShort/20140204/209176/Level_Design_in_Procedural_Generation.php

Smith, G. (2014). The Future of Procedural Content Generation in Games. *Experimental Artificial Intelligence in Games*, 53-57.

Sterling, J. (2016, August 10). *No Man's Sky Review - Falling Skies.* Retrieved from The Jimquisition: http://www.thejimquisition.com/no-mans-sky-review/

T.C. (2014, September 25). *Why video games are so expensive to develop.* Retrieved from The Economist: http://www.economist.com/blogs/economist-explains/2014/09/economist-explains-15

Teti, J. (2011, October 7). *The Binding of Isaac.* Retrieved from Eurogamer.net: http://www.eurogamer.net/articles/2011-10-07-the-binding-of-isaac-review

Tulleken, H. (2009, April 25). *How to Use Perlin Noise in Your Games.* Retrieved from Dev.Mag: http://devmag.org.za/2009/04/25/perlin-noise/

Unity Technologies. (2017). *Prefabs.* Retrieved from Unity Documentation: https://docs.unity3d.com/Manual/Prefabs.html

Valve Corporation. (2009). Left 4 Dead 2. Kirkland, Washington: Valve Corporation.

W., T. (2016, April 19). *Q&A: The guns and dungeons of Enter the Gungeon.* Retrieved from Gamasutra: http://www.gamasutra.com/view/news/270719/QA_The_guns_and_dungeons_of_Enter_the_Gungeon.php

Wolfram, S. (2002). *A New Kind of Science.* Champaign, IL.: Wolfram Media.

Zucker, M. (2001, February). *The Perlin noise math FAQ.* Retrieved from MZucker: https://mzucker.github.io/html/perlin-noise-math-faq.html#toc-whatsnoise

8 APPENDICES

The following items can be found on the supplied USB Stick:

- Appendix A – Dissertation Proposal
- Appendix B – Ethics Form
- Appendix C – Initial Plan
- Appendix D – Testing Document
- Appendix E – Participant Information
- Appendix F – Participant Consent Form
- Appendix G – Blank Questionnaire
- Appendix H – Participant One Questionnaire
- Appendix I – Participant Two Questionnaire
- Appendix J – Participant Three Questionnaire
- Appendix K – Participant Four Questionnaire
- Appendix L – Final Unity Project
- Appendix M – Final Window Executable
- Appendix N – Screenshots