**Procedural Content Generation in Games**

AUSTIN ANDREWS, Harding University

Procedural content generation (PCG) is an older, yet increasingly common technique used by video game developers. While content created manually continues to lead in game development, procedural content generation offers a strategic answer to certain limitations unachievable by computers or human programmers and artists. Algorithms that dynamically create and render pseudo-infinite worlds or unpredictable outcomes to NPC (non-playable character) appearances or behavior have been allowed to grow more complex by the evolution of computer hardware capabilities. The number of individual procedurally generated assets which have multiple attributes can number in the millions with very little work by the creators.

Categories and Subject Descriptors: **D.2.2 [Software-Design]**: Methodologies; **I.2.1 [Applications and Expert Systems]**: Games

General Terms: Assets, Procedural, Algorithms, Perlin Noise, Voxels, Chunks

Additional Key Words and Phrases:

**ACM Reference Format**

Austin Andrews, 2015. Procedural Content Generation in Games.

# INTRODUCTION

Procedural content generation spawns from a common limitation software developers were forced to face in the early years of computers. Hardware limitations not nearly as problematic today, were issues in game development as well as any other software being created in the 1980s. Game developers sought after methods of achieving great depths of detail without requiring the use of computer power that not yet existed. Creating content ad hoc by the computer that could either be discarded after use or re-accessed without saving to a hard drive birthed the idea of procedural content generation.

“Procedural content generation (PCG) in games refers to the creation of game content automatically using algorithms” [Togelius. 2011].

While this definition may be slightly vague, it forms the basic essence of how developers describe PCG. There is no clear boundary for the concept of PCG; a game designer, and artist, and a developer may all have distinctly different definitions of what “content” refers to in games. For consistency, this paper refers to content in games as objects that have visual elements and attributes. Content can also be described as parts of the game unseen, such as storylines, dialogue, and audio, however, these are outside the scope of this paper.

Previously stated, PCG is created automatically using algorithms [Togelius, 2011]. These algorithms are built specifically for the creation of said content and are typically given some form of input to return a visual representation of the output. For example, if the content being realized is a map terrain, the input might be represented as the X, Y, and Z coordinates of the player’s position in the world, and the output might be terrain rendered around this position in a given radius. In this instance, a world can be represented as pseudo-infinite by creating new and unique terrain in every direction as the player travels, this can be continuous for as far as the computer can generate.

A common misconception about PCG in games is that procedural is equivalent to random. Procedural is not random; random creates chaos [Murray 2014].

“We are really picky about the word procedural. We say it’s not random. Random to me is like a chaotic kind of mess, potentially. But procedural is like creating mathematical formulas, and they create a result” [Murray 2014].

To clarify, an example of random would be choosing a number between one and one hundred and assigning an outcome to generate based off what number the computer chooses. Procedural is the use of mathematical formulas to generate content based off the input given to the formula; instead of choosing, the computer calculates. This could mean that the input are things surrounding an object, like flowers surrounding a tree. The number of flowers, color, height, and distance from the tree are all properties that can all be calculated based on what type of tree the flowers are surrounding, or the altitude of the tree on a mountain. Randomly generated flowers would have all these values chosen without any design whatsoever. The input could also be the position of the player’s X, Y, and Z coordinates in the world or perhaps the level or rank the player can yield the level and difficulty of the enemies a player must face during gameplay.

# HISTORY OF PCG

**2.1 Early Use of PCG in Games**

Early game developers were not considered professionals, rather, video games at that time were not necessarily professionally developed. There was no dire need for video games and those who developed them, did so on their own time. For this reason resources for creating high quality games were not readily available, if existing whatsoever. Memory and storage was limited as well as the processors on most every computer in the 1980s. The tradeoff for memory and storage and the processor was important and procedural generation was the answer [Lee-Urban 2013].

Designated as one of the very first procedurally generated video game, Rogue developed by Michael Toy and Glenn Wichman in 1980, is not only a pioneer for video games, it has also spawned its own genre, “Roguelike”. While Roguelike games are simple, they are known to generate dungeons on-the-fly, or procedurally, as can be seen in Figure 1. A dungeon has a series of rooms of varying sizes connected by passage ways. Generated are the rooms and connecting passages between.

Though there have been many inspired games from it, Rogue is the first dungeon-crawler to use the concept of PCG. Due to computer limitations at the time, content for games like Rogue had to be created on-the-fly by the computer instead of storing all the needed content in memory until used. Arguably the greatest contribution to video games from Rogue was the creation of essentially a new game each time played. Originally, Rogue was rendered completely using ASCII characters, then eventually was update to better graphics such as that in Figure 1.

“Rogue is generally credited with being the first ‘graphical’ adventure game, and it probably was at least one of the first (Wizardry could probably also make that claim). And its graphics have since been far surpassed by everything from Myst to Doom. But I think Rogue’s biggest contribution, and one that still stands out to this day, is that the computer itself generated the adventure in Rogue. Every time you played, you got a new adventure. That’s really what made it so popular for all those years in the early eighties” [Wichman 1997].

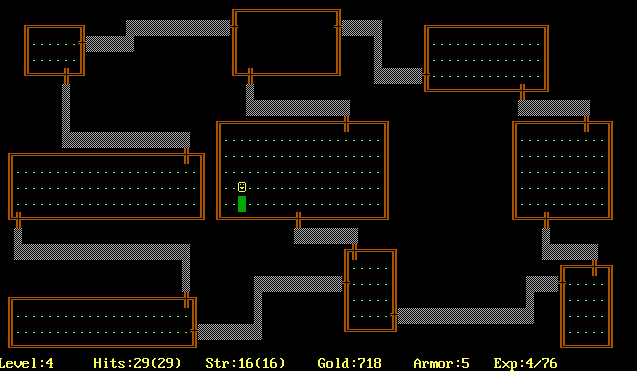


Fig. 1 Rogue [Michael Toy & Glenn Wichman, 1983]

Another game that was released shortly after Rogue was the infamous Elite developed by David Braben and Ian Bell in 1984. Unlike Rogue, Elite was a space trading game using revolutionary 3D wire-frame graphics with hidden line removal. Elite uses procedurally generated content in a significantly different way than Rogue. While Rogue uses PCG to dynamically create rooms inside of dungeons, Elite used a seed of 3 numbers to generate 2048 planetary systems spread over 8 different galaxies. Each planetary system had its own unique attributes that were dynamically assigned when needed based on the seed used so that no galaxies or planetary systems were required to be stored in memory. This paper discusses Elite in more depth later on in section 4.

**2.2 Hardware Limitations**

As previously stated, limitations of computer hardware in the early years of computers was an enormous contributor to the invention of PCG. Rogue and Elite are two very early games in specific which brought about this idea of creating worlds and levels algorithmically on very small sizes of RAM and processor power. For example, Elite only used around 32KB of memory [RABIDGREMLIN 2014]. Personal computers in the 1980s were very limited on what could be done in video games, thus requiring the developers to become creative in their use of algorithms to handle the game’s content.

# HOW IS PCG BEING USED IN GAMES?

**3.1 World Generation**

Generating worlds in games procedurally, whether it’s just the map terrain or the entire universe with planets and terrain for each planet, is the most used PCG technique. When discussing PCG, the topic usually will consist of generating some sort or terrain or world that the player is interacting with. A game that, in the recent years, has brought a common acceptance and understanding of procedurally generated worlds to the public is Minecraft. Minecraft is a game made up of pseudo-infinite maps represented by chunks made of voxels. Voxels are simply a method used for rendering meshes in 2D and 3D environments as demonstrated in Figure 3. The maps themselves are based on seeds and the terrain is rendered in chunks of 16x16x16x128 blocks [Notch, 2011]. The chunks are loaded based on where the player is currently in the world. The coordinates of the player determine what part of the terrain is loaded and rendered at a given chunk.

Minecraft is a great example of what procedurally generated terrain and world generation is capable of. Not only does the terrain give the illusion of infinity, it is also completely editable, meaning that individual block voxels in the chunks can be created or destroyed at any given location in the world. Generally, when an algorithm is written to create a world, the formula will realize the same result every time as long as the input is the same. For example, if a mountain is generated in Minecraft, the mountain will always be generated visually the same way in the same location as long as it is in the radius of the player’s rendering distance. When the mountain falls out of range of the player’s rendering distance, it will be discarded from memory until the next time in which the player enters that specific area again. If the player decides to alter the mountain, the edited chunks that make up the mountain will be saved to the hard drive, such a saved game. When the edited mountain’s chunks need to be re-rendered after they have been altered (when the player leaves and returns to its area), the computer checks to see if the chunks have been changed and applies the changes before they are rendered for the player, thus always having the same results as the last time the player visited that area.

This formula is not unique to Minecraft. With some variances, this is the common way that worlds are generated using PCG. The algorithm that determines a rendered terrain will provide the same results given the same input, however, anything outside the algorithm’s scope technically does not exist in memory. A good metaphor for understanding how worlds are generated using PCG is asking the question: if a tree falls in the forest, and no one is around to hear it, does it make a sound? Based on our observable universe, the answer would be yes, however in a procedurally generated environment inside of a game, the answer would be no. There is no tree because the tree only exists and is processed by the game if there is a player around for it to be rendered. If there is no player to generate the tree, there is no tree according the game. However, the tree in question will always be there when the player approaches it. How is that possible? Once the input required to render that tree (the player’s position in the world) is realized, the algorithm will proceed to process the information of that tree and the attributes associated with it. If it then decides to fall, the results can be observed [Robinson 2014; Murray 2014]. In a procedurally generated environment, nothing outside the algorithm’s output is stored in RAM or on a hard-drive (unless explicitly stored) meaning that it does not exist in the current context. It exists only in the output of the algorithm.

Are games that use this concept of “infinite terrain” truly infinite? While playing Minecraft, it is fairly obvious that there is no defined edge to the world. A player can travel in one direction for what seems like forever and never find a stopping point. The illusion of infinite terrain is simply that; an illusion. A computer scientist should know that there is always a limit to numbers and processes that can be computed using computer. First let us look at the most common method for creating procedural terrain; the use of Perlin noise. Perlin noise is a type of gradient noise developed by Ken Perlin in the 1980s as a result of his frustration of machine-like computer graphics at the time. As represented in Figure 2, the lighter parts of the image represent higher points on a map and the darker areas are the lower points. Notch, the nickname for Markus Persson the creator of Minecraft, explains how Minecraft uses Perlin noise to generate the height-maps as the hilly terrain seen in Figure 3, and what happens when a player tries to travel to the edge of the world.



Fig. 2 Perlin Noise

“I used a 2D Perlin noise heightmap to set the shape of the world. Or, rather, I used quite a few of them. One for overall elevation, one for terrain roughness, and one for local detail. For each column of blocks, the height was (elevation + (roughness\*detail))\*64+64. Both elevation and roughness were smooth, large scale noises, and detail was a more intricate one. This method had the great advantage of being very fast as there’s just 16\*16\*(noiseNum) samples per chunk to generate, but the disadvantage of being rather dull” [Notch 2011].



Fig. 3 Minecraft Terrain

“Let me clarify some things about “infinite” maps: They’re not infinite, but there’s no hard limit either. It’ll just get buggier and buggier the further out you are. These [chunks] have an offset value that is a 32bit integer roughly in the range of negative two billion to positive two billion. If you go outside that range (about 25% of the distance from where you are now to the sun), loading and saving chunks will start overwriting old chunks. At a 16th of that distance, things that use integers for block positions, such as using items and pathfinding, will start overflowing and acting weird” [Notch 2011].

For the average player, the illusion of infinite terrain is possible because no one can ultimately travel far enough to break the game.

**3.2 Other PCG Uses**

Even though procedural world generation is the most common use of PCG, it is not the only use. Another example of PCG in games aside from world generation is what this paper will refer to as “item drops”. Item drops in games are useful objects the player can use to progress in the game. Examples of item drops would be gold found from looting chests or combination codes for doors the player must find to move to a new level or area. Item drops can also be objects such as weapons or unlockable abilities unique to a player’s character.

In the game Borderlands, the guns a player can pick up are procedurally generated depending on certain parameters of the player. If two players played the game separately, the same chest would appear in the same place for both players, but the guns found inside would be drastically different both visually and statistically. Gun attributes such as rate of fire, damage, clip size, accuracy, and rarity are determined by the attributes of the player, specifically the level of the player at any given point. Figure 4 gives an in-game example of a gun with stats generated from a player around rank 61.

“There are over 17,750,000 different variations of weapons in Borderlands, as of its release” [Gearbox, 2009].

It would be impossible for the content creators at Gearbox to realistically create the art and stats of each of weapon in Borderlands individually. Instead, certain visual properties that *can* be applied to various weapon types are selected by the computer and the attributes are generated based on the input from the player’s stats to create a uniquely individual weapon each time the game is played. For example, an assault rifle type would tend to have a much higher rate of fire than a sniper rifle type, but its damage output would be a lot lower. However, the assault rifle found when the player is at rank 11 will be drastically different than the assault rifle found at rank 35. Borderlands dynamically changes the stats of weapons and enemies dependent on the stats and rank of the player at any given point in the game. These weapons are procedurally generated because until the very moment one is found, the attributes have not been determined. They are calculated upon discovery.



Fig. 4 Nasty Seeker from Borderlands [Gearbox, 2009]

The same idea can be applied to other forms of content as well such as creature creation. In games like Starbound developed by Chucklefish and Spore developed by EA, NPCs, or non-playable characters, are created by having a list of usable attributes by each species and selecting each attribute procedurally for each one. These selecting algorithms are usually in reference to attributes already chosen.

# Creating Games WITH Procedural content

**4.1 Advantages for Using PCG**

Some of the advantages have already been touched upon but not explicitly stated. One of the most influential reasons for a developer choosing to use PCG in a video game is the quantity of needed content. It is impossible for creators to manually create thousands to millions of individual components to a game when the ease of computationally generating the desired results are applicable. It is entirely possible to create a game such as Rogue by hand if so desired, however, a large advantage Rogue has is that it is a new experience each time it is played. Today, large video game companies create games with hundreds of developers who have the resources to make a game that is very large and in depth but still of a high quality while independent developers have trouble matching the same standard of development. Independent developers can choose to create larger open-world games simply by making the algorithms which do so with relative ease.

Another previously stated advantage to PCG is memory management. Games such as Rogue and Elite were required to make sure computers in the 1980s could handle such games. Today, hardware is rarely an issue, however, it is not feasible to render games in their entirety. PCG is flexible so that it the content essentially stays compressed until needed.

If we go outside the scope of this paper’s definition of content for a moment and look at content as more than just the visible and interactable objects of a game, we can see that storylines, dialogue, quests, rule sets, etc. are all in the realm of possible aspects of procedurally creatable content.

**4.2 Pitfalls**

A few of the dangers of creating content procedurally are as follows: In an open world discovery game, it can be difficult to always be sure of a “playable” world. Take Rogue for example, if one of the rooms was blocked because the algorithm did not generate a passage way to it, the player would not have access to the room. If the algorithm is not created with errors like this in mind, the player can be stuck with nowhere to go and no way to continue, thus halting gameplay entirely.

Another pitfall that developers have to worry about is rewarding players with too powerful of tools too quickly or too weak tools too late. While talking about item drops, it is important to not allow a player to pick up a weapon with too high of a damage output too early because they will be much to overpowered for the duration of the game. The same can be said for weak assets; if a player requires a certain level of damage to face a boss fight and it has not been available yet, there is no way for the player to continue forward in the game, halting gameplay once again.

Some developers see PCG as “lazy” development. PCG focuses on the quantity instead of the quality of content in a video game.

# The past and the future

**5.1 Elite (1984)**

As stated previously, one of the earliest games to adhere to the concept of PCG was Elite. Elite is a “space trading” game for the home computer and one of the first to use wire-frame graphics to represent 3 dimensions. Elite featured 8 full galaxies holding 2048 planetary systems total, each with a unique name, description, technology level, government type and economy. With all of these features, Elite only used 32KB of RAM. 32KB was large for the computers of the time, but was nowhere near large enough to hold the entire universe of 8 galaxies and 2048 planetary systems. In fact, not even a single galaxy could be stored in RAM at the time of Elite’s release in 1984. Because of these computational limitations, the developers were required to either shrink the universe to a size that computers could realistically handle or develop algorithms to obtain the desired result of which they were aiming [RABIDGREMLIN 2014].

The use of a seed in games using PCG is a common approach. Seeds are numbers that are given as the starting input for games. They can either be generated randomly or given explicitly to control the generation of the world slightly. Elite’s developers started with three numbers: 23114, 548, and 46931. These are used as the seed of the universe and are incremented and decremented algorithmically to represent the entire universe of Elite using an optimal amount of computing power. The basic outline of the process is as follows:

— Set the starting seed numbers (23114, 548, and 46931)

— Realize which of the 1-8 galaxies the player is current occupying and “twist” the seed, stepping through each galaxy to get the player’s current galaxy, as can be seen in Algorithm 2.

— Each galaxy contains 256 planetary systems that are iterated through, generating each of the details using the seed numbers.

Algorithm 1. Build Galaxy Function in Elite

void buildgalaxy( uint galaxynum )

{

uint syscount, galcount;

*/\* Initialize seed for galaxy 1 \*/*

seed.w0 = base0;

seed.w1 = base1;

seed.w2 = base2;

for( galcount = 1; galcount < galaxynum; ++galcount )

nextgalaxy( &seed );

*/\* Put galaxy data into array of structures \*/*

for( syscount = 0; syscount< galsize; ++syscount )

galaxy[ syscount ] = makesystem( &seed );

}

Algorithm 2. Twist Function in Elite

uint16 twist( uint16 x )

{

return ( uint16 )(( 256 \* rotate1( x << 8 )) + rotate1( x & 255 ));

}

Algorithm 3. Next Galaxy Function in Elite

void nextgalaxy( seedtype \*s ) */\* Apply to base seed; once for galaxy 2 \*/*

{

( \*s ).w0 = twist(( \*s ).w0; */\* twice for galaxy 3, etc. \*/*

( \*s ).w1 = twist(( \*s ).w1; */\* Eighth application gives galaxy 1 again \*/*

( \*s ).w2 = twist(( \*s ).w2;

}

[RABIDGREMLIN 2014]

**5.2 No Man’s Sky (2015)**

Like many forms of entertainment, video games are usually inspired by games that precede them. A game that has yet to be released at the time of this paper is No Man’s Sky, which is being developed by Sean Murray of Hello Games in Guildford, UK. Described as “a science-fiction game set in an infinite procedurally generated galaxy,” No Man’s Sky is going to be a giant step forward in the creation of games using PCG. Typically in open world games, a single player will create a server on which to host other players wanting to join. No Man’s Sky will be on one massive server with its current build at over 18 quintillion visitable planets [Murray 2014]. This means that everyone will be on a single, rather large server all playing in the same universe. Every planet can be flown to using custom ships and every planet is procedurally generated based on a 64bit seed.

In thirty years, game developers have been able to go from an impressive 2048 planetary systems over 8 galaxies in Elite to 18 quintillion planets in No Man’s Sky using the same techniques of procedurally generation planets, galaxies, and universes.

# Conclusion

While procedurally generated content is not a new programming concept, it is a lesser known technique outside of game developers that paves the way for diverse and interestingly new games. There is no other game development method that allows developers to control randomly created objects and still get nice, relatively unpredictable results. There is no other way to create quintillions of visitable worlds in space crafts like in No Man’s Sky or to build unbelievable structures in a completely customizable maps like in Minecraft than to use procedural content generation.

# References

Gearbox Software. 2009. Boarderlands Wiki. Retrieved from http://boarderlands.wikia.com/wiki/Weapons

Stephen M. Lee-Urban. 2013. Procedural Content Generation. *2013 Georgia Tech Lectures*. Retrieved February 4, 2015 from http://www.cc.gatech.edu/~surban6/cs4731/lectures/2013\_06\_18-ProceduralContentGeneration.pdf

Sean Murray and Cam Robinson. 2014. How Does No Man’s Sky Actually Work? – Reality Check. Video. Retrieved on January 2, 2015 from https://www.youtube.com/watch?v=ZVl1Hmth3HE

Markus Persson (Notch). 2011. The World of Notch, Rarely Updated, Rambling, Ranting, and Informing. Retrieved February 4, 2015 from http://notch.tumblr.com/post/3746989361/terrain-generation-part-1

Rabidgremlin. 2014. Rabidgremlin’s Soapbox – Procedural Content Generation: Creating a Universe. Retrieved February 5, 2015 from http://blog.rabidgremlin.com/2015/01/14/procedural-content-generation-creating-a-universe/

Julian Togelius. 2011. What is Procedural Content Generation? Mario on the Borderline. Retrieved February 17, 2015 from http://julian.togelius.com/Togelius2011What.pdf

Pcg.wikidot.com. 2014. Procedural Content Generation Wiki. Retrieved January 5, 2015 from http://pcg.wikidot.com