This space is reserved for the EPiC Series header, do not use it

Developing a Grammar for Procedural Generated Video Games

Rowan Murphy and Isaac D. Griffith

Idaho State University
Computer Science Department
Pocatello, Idaho, United States
rowanmurphy@isu.edu
isaacgriffith@isu.edu

Abstract

Procedural Content Generation is a widely used strategy for creating content in video games and other simulations. In this paper, we are proposing to create a DSL that makes it easier to achieve the main aspects of PCG. And allows new developers a tool to implement PCG more easily. This is done by creating an internal DSL and evaluating whether our DSL allows us to improve on the characteristics of PCG or not. We will evaluate this by comparing a program written with our DSL and one without. This will be a small DSL that will focus on the minor aspects of game development and PCG to get accurate and consistent results.

1 Introduction

"Procedural Content Generation(PCG) is an ever-growing technique used mainly in the field of game development and computational creativity to generate content without or with a reduced amount of human input."[10] It is used in a variety of applications from simple randomization techniques in AAA titles to tile placement and so much more. But PCG doesn't just appear out of thin air, we have to constrict it in certain ways to validate the quality and useability of this technique. PCG is a very complex and confusing field at times, that is why we have observable characteristics that can help us evaluate the success of the implementation of PCG. These characteristics are speed, controllability, reliability, and expressivity/creativity. By using these guidelines we can try to determine if our implementation of PCG was a success, and what revisions are needed to better fulfill these requirements.

The main issue we are trying to address is making it so developers new to PCG can implement it easier while still maintaining the core concepts of PCG. We are aiming to address if a DSL will affect the core concepts, IE: will our DSL make it easier for us to achieve the characteristics we want? Or will it complicate things and make it harder to stick to these goals. How can we ensure simplicity for developers while retaining readable and implementable code? The goal of our research is to address all of these questions, and hopefully, implement a DSL that will improve the attainability of the primary aspects of PCG and make it so new developers can do this in an easy and controllable way.

We are planning to Develop a grammar to be used inside of the procedural generation of content for video games. This type of approach for content generation in games has not had much research. Domain-specific languages and grammar could have a very beneficial impact on PCG in games. "The field of research regarding procedural generation in games is growing rapidly, and the applications of them are very useful with games and other content generation practices."[9] This will be done by studying PCG algorithms and techniques as well as the creation of an internal DSL. PCG has a significant amount of complexity and having a fundamental understanding of it is essential in creating a functional DSL. This DSL will be created in C++ and use a small game to validate the success of this research. The evaluation of our DSL will be closely related to the main aspects of PCG. We will measure the main characteristics of PCG by comparing the speed, controllability, reliability, and expressivity/creativity with two versions of a PCG implementation, one using the DSL and one using more traditional techniques. By doing this we can check many of the characteristics of PCG adequately.

Procedural Content Generation in games is a vast field spanning multiple areas of computer science,[10] but it can often be complex and unforgiving to people pursuing research in the field. "A prior knowledge of AI and machine learning is useful regarding this topic, as well as extensive knowledge in algorithm creation." [13] Creating a DSL specific to PCG would allow us to use these types of algorithms more easily and effectively to hopefully increase research interest in this field. There is some useful background information on DSls and Procedural content generation needed to understand the problem and solution we are proposing.

The papers format is as follows: Background, Methods, Initial Findings, and Timeline. Background covers all needed information to follow what we are doing and why. Methods will describe how we are going to do this. Initial findings go over what we have found so far with our research as of 3/20/22(mostly PCG content techniques). And Timeline and conclusion expresses why we are doing this problem as well as our time constraints.

2 Background

"PCG involves the use of algorithms for the generation of game content with limited or no human contribution." [4] There are many different types of PCG techniques from fractal generation, level/map generation, forest generation, and hundreds more, each with their own unique uses in generating dynamic content. [11] By using this, unique game content can be generated dynamically and allow for reduced workload and increased quality for commercial video games. A recent study created a way to chain Wikipedia links together and create them as NPC's and game objects. What this accomplishes, is creating an entirely unique "game" experience that can be dynamically changed as it is played." [2] This is very promising as it shows that the applications of procedural generation are endless, and research into this topic is highly valuable. The main desirable aspects of PCG are speed, controllability, reliability, and expressivity/creativity." PCG [10] These aspects were created outlined in "Procedural Content Generation in Games" and outline the most important results of using a technique like this. When using PCG one of the most important things is variability, and having control over the variability; However, in many games that implement this strategy, this is not the case.

Creating a grammar for this application requires the use of multiple types of software and Languages. A simple game with basic objectives, rewards, and layouts will be needed to test the validity of the PCG content; We are still deciding whether to adopt an existing game to use these techniques or to create one of our own. There has not been much work regarding adaptations of PCG in existing titles, so we are leaning towards creating a simple 2D game to work with this research. C# or C++ would most likely be the preferred language to use

because of their compatibility with industry-standard game engines such as the Unreal Engine or the Unity Engine, but depending on the game size, the use of these may not be necessary. An internal DSL would be preferable as dealing with parsers and the other semantics of external DSLs would put a large constraint on the time and success of this research project. Currently, we are investigating DSLs with the book "Software Languages"[6] A simple implementation would be "using PCG to generate dungeon environments with simple goals, rewards, and potential layouts."[12] Other students have done similar research regarding this topic and should be a fantastic resource to develop initial skills with PCG. we think that this will be a potential idea for implementing the grammar set, as it will be easy to evaluate and somewhat simple to create. "There has also been significant research into 3D terrain generation using PCG by using real-world terrain data to simulate "real" environments."[8] If possible, we would love to explore this as well within the bounds of software language engineering, but this may not be possible.

3 Methods

So how can we create a grammar that will solve the problem of the complexity in PCG? The best fit for this research will require us to create an embedded DSL inside of C++. An embedded DSL will be much better for the purposes of our research due to the simplicity of them compared to external DSLs. "This is done by creating expression objects, with node objects underneath. Those node objects contain the necessary code for the interpreter." [7] We will then take some popular procedural generation algorithms (such as fractal generation) and use them to create simple grammars that will hopefully, allow us to implement procedural generation in small scale programs written in C++ easily. As for the applications of this research, we hope that add an element of simplicity to implementing these techniques and improve the overall quality and readability of the code and programs.

4 Initial Findings

The initial findings of this research has been minimal, as time and a steep learning curve have been major factors. Almost all of the research so far has been relating to PCG, how it is done, and ways to implement it that are feasible in the scope of this project. The next few paragraphs will go into detail on some of what has been learned, and what will happen next within the semester.

"Procedural Generation begins with one thing, Noise. Computer generated noise known as Perlin noise is used to generate seemingly "random" patterns as opposed to the pseudo randomness found in things such as random number generators on computers." [1] With these random patterns we can generate content that is actually random (for the most part). A simple implementation of this would be, Taking all white pixels in this noise and assigning it to a certain texture, and taking all black pixels and making them a different texture or empty space. [5]. One developer who worked on the game The Sims, Spore, and many others by the name of Kate Compton implemented just what we are trying to do here but with different applications. Her language, "Tracery, was used to recursively generate content out of generated content. A good example is her making alien names procedurally by using specific rules to ensure that while they are random, they were all controllable and similar to the main templates."[3] This has been a fantastic resource to learn about PCG and will continue to be used for the duration of the research. Our research problem still seems feasible and we are hopeful that the results

will be of some benefit. Due to the complexity of PCG that has taken a lot of time, we are just now hitten a stride of progress on the research. This section and others will be updated as research takes place.

The next part of our research entails studying specific games or creating one for this research, as well as developing the initial DSL and to begin testing. Most of the time so far has been spent learning about PCG and implementing them in small examples; Sadly, these examples are not applicable to this research. A 2d game with procedural terrain will most likely be used because of the standard practice of Perlin noise and how our DSL may end up working. A simple implementation of terrain generation will work well to address our problem because it should be easily controllable and give consistent results. Adding more PCG in this evaluation would make it too hard to evaluate and take too much time. In the next few weeks the plan is to show some code that will hopefully be able to express minimal aspects of PCG. After this, a small terrain generator or game will be made and the majority of testing will be done.

5 Project Setup

Initially, in our research, we were devoted to showing the main characteristics of PCG and how they were affected by the implementation of a DSL and whether they were preserved and improved upon or negatively affected. We quickly realized that this would be a tricky thing to analyze. To analyze this, we had a few options in mind; Our first idea was to use a 2D game to show PCG techniques and then use a DSL to see if those were affected. Unfortunately, that was outside of the scope of this research, as developing a game and DSL would take too much time. After realizing this, we understood that a game wasn't needed if all we were testing the implementations of PCG techniques.

The first step was to develop a simple graphical DSL to create an overview of the embedded DSL we would make. This was done with Visual Studio 2019 DSL modeling tools and demonstrated that a traditional DSL would not work well within simple indie development or small game development. We then decided to look at PCG as a whole to determine our next steps.

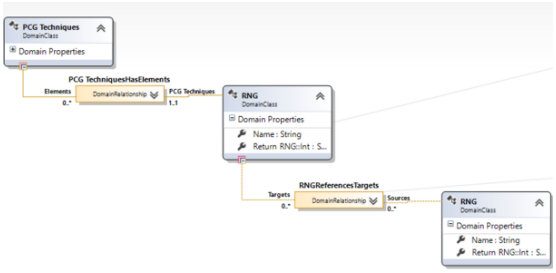

Figure 1: Very simple Graphical DSL Diagram for RNG

There are many different PCG techniques, from fractal patterns to cave generation [11], but one thing is shared between almost all of them. They are highly specialized and created specifically for the project being made. So how can we test a DSL that is supposed to make PCG more accessible if each PCG technique is tailor-made to the project being created? We came across this issue as the PCG in games was much more complex than initially anticipated. So to get some results out of this, we looked at the simplest form of PCG, Random Number Generation. Linear Congruency, C++ rand, and Lehmer Generator.

By using Random Number Generation(RNG), we could test a simple embedded DSL and see if the characteristics of PCG were being affected in the slightest way possible. We took a few different kinds of RNG. We implemented them in C++ with and without the simple embedded DSL to determine if having the DSL put significant constraints on the main characteristics of PCG. By doing this, we could analytically decide whether or not it is feasible to use a DSL for these purposes in a small game.

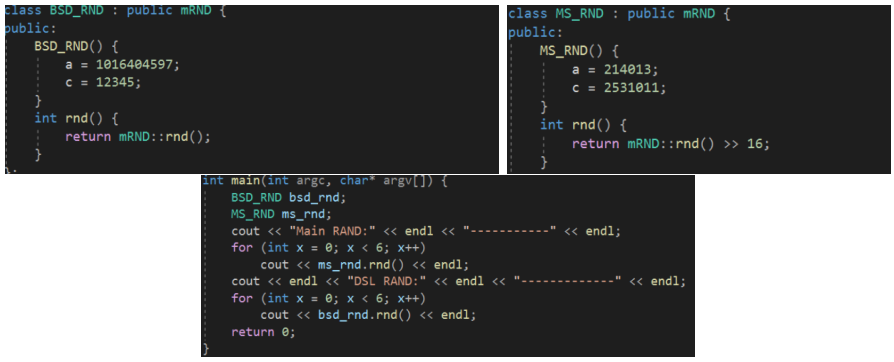

Figure 2: Simple test code in C++ for RNG Using Seeds

6 Results

Creating an embedded DSL was more straightforward than we thought for this application, but it did take some time to implement and a lot of research. The embedded DSL consists of C++ code that allows the use of RNG. We tested our simple embedded DSL with a few different types of RNG, and the processes results were surprising.

When testing these random number generators with and without the DSL, the results were clear. It was significantly more challenging to implement them with the DSL and to keep the adaptability, creativity, and controllability of our PCG. Once We implemented the DSL, it was simple to use these different RNGs, but they were extremely limited in what they could do. We found ourselves adjusting the DSL multiple times to fit our testing goals which was difficult to justify doing. And by doing so, any new developers trying to use these techniques would become frustrated with the lack of adaptability in this solution.

The central aspect of PCG that we are focusing on is adaptability, controllability, and creativity. These are essential as, without them, PCG becomes a cookie-cutter solution and loses the things that are good about it. When applying PCG to games, in particular, the quality and uniqueness need to be there for the project to be worthwhile. Our goal was to determine if the characteristics of PCG were affected positively or negatively, and the result from our small tests showed a negative change. Creativity and Adaptability were affected by the lack of flexibility inside this small embedded DSL. "Without flexibility, these two main characteristics cannot be achieved." [10] The quality of PCG is determined by the main characteristics listed above. Three out of the five main attributes of PCG were negatively affected within our results, showing that it is not a good fit for applications in small-scale game development.

7 Conclusion

Game development is such a broad category of software engineering. Before starting this research, we thought it might greatly benefit from software language engineering. Still, with our results, at least for indie or solo developers, a DSL only complicates things and does not significantly improve PCG implementation in games. There is promise for this idea in a large setting with many years to develop a sizeable commercial AAA game. The applications of software language engineering are exciting and have great potential. For small applications, we are not quite there yet as far as implementing this for minor projects such as indie development; The time needed to get something like this to work for a small and unique product is just too much and takes away from the overall experience and goal of indie game development.

References

[1] Khan Academy. Perlin noise. https://www.khanacademy.org/computing/computer-programming/programming-natural-simulations/programming-noise/a/perlin-noise, 2020.

[2] Gabriella A. B. Barros, Antonios Liapis, and Julian Togelius. Playing with data: Procedural generation of adventures from open data. http://www.digra.org/digital-library/publications/playing-with-data-procedural-generation-of-adventures-from-open-data/, 2016.

[3] Kate Compton. Practical procedural generation for everyone. https://www.youtube.com/watch?v=WumyfLEa6bU&list=LL&index=1&t=730s, 2017.

[4] Edirlei Soares de Lima and Antonio L. Furtado Bruno Feijó. Procedural generation of quests for games using genetic algorithms and automated planning. https://ieeexplore.ieee.org/document/8924855, 2019.

[5] Digit. Procedural generation, how does it work? https://www.youtube.com/watch?v=-POwgollFeY, 2020.

[6] Ralf Lammel. Software languages. digital, 2018.

[7] Arne Mertz. Domain specific languages in c++. https://arne-mertz.de/2015/06/domain-specific-languages-in-c/, last viewed March 2021, 2015.

[8] Journal of Computer Graphics Techniques. Designer worlds: Procedural generation of infinite terrain from real-world elevation data. [digital], 2014.

[9] Christian Schenk and MiKTEX Contributors. Search-based procedural generation of maze-like levels. https://ieeexplore.ieee.org/document/5742785, 2011.

[10] Noor Shaker, Julian Togelius, and Mark J. Nelson. Procedural content generation in games. , 2016.

[11] Open Source. Algorithms for pcg. http://pcg.wikidot.com/category-pcg-algorithms, last viewed March 2022, 2022.

[12] Roland van der Linden, Ricardo Lopes, and Rafael Bidarra. Procedural generation of dungeons. https://www.researchgate.net/publication/260800341_Procedural_Generation_of_Dungeons, 2013.

[13] Ryan J. Vitacion and Li Liu. Procedural generation of 3d planetary-scale terrains. https://ieeexplore.ieee.org/document/8863868, 2019.