6CCS3PRJ Final Year Individual Project Report Title

Final Project Report

Author: Zishan Rahman

Supervisor: Senir Dinar

Student ID: 20071291

March 8, 2023

Abstract

Procedural generation refers to content in a medium that is produced algorithmically in lieu of by hand. Most notably, procedural generation algorithms are implemented in video games, for generating levels, terrain and other game contents programmatically. This project takes some of the more prominent algorithms for procedural generation- Lindenmayer Systems, Voronoi Points, Poisson Disk Generation and Simplex Noise- and implements them in a 3D walking simulator in the open-source Godot game engine, and compares their workings and performance. My aim with this project is to (1) increase my knowledge of procedural generation in games beyond the surface level, by going in-depth into some of the algorithms that are used, and (2) use this knowledge to implement said algorithms in a 3D walking simulator scenario in Godot, then compare how each algorithm works and performs.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Zishan Rahman

March 8, 2023

Acknowledgements

It is usual to thank those individuals who have provided particularly useful assistance, technical or otherwise, during
your project. Your supervisor will obviously be pleased to be acknowledged as he or she will have invested quite a lot of time overseeing your progress.
Thanks to my supervisor Senir Dinar, for providing me the guidance I so badly needed
to make this project not only the best for my mark, but also one that I enjoy.
Thanks also to Kevin Lano, who helped me up when I was down.

Contents

1 Introduction **2**
1.1 Report Structure 2

2 Background **3**
2.1 Procedural Generation: Background 3
2.2 Justifying My Choice of Engine: Godot 5

3 Report Body **7**
3.1 Algorithms 7

4 Design & Specification **10**
4.1 Section Heading 10

5 Implementation **11**
5.1 Section Heading 11

6 Legal, Social, Ethical and Professional Issues **12**
6.1 Section Heading 12

7 Results/Evaluation **13**
7.1 Software Testing 13
7.2 Section Heading 13

8 Conclusion and Future Work **14**
Bibliography 15

A Extra Information **16**
A.1 Tables, proofs, graphs, test cases, ... 16

B User Guide **17**
B.1 Instructions 17

C Source Code **18**
C.1 Instructions 18

Chapter 1

Introduction

Procedural Content Generation, or PCG, refers to the use of algorithms and programming in lieu of human handiwork to design and implement various contents in video games, such as levels, terrains, trees and cities. A PCG algorithm is ontogenetic when it tries to produce a foreseeable end result as it goes along. For this project, I will be implementing several well-known ontogenetic algorithms in a basic 3D walking simulator game, using the open-source Godot game engine, and then comparing how each algorithm carries out the creation of levels in said game.

1.1 Report Structure

Chapter 2

Background

For my BSc individual project, I will be researching procedural content generation (PCG) algorithms and then implementing them each in a small 3D game made with the Godot Engine (and its domain-specific GDScript language).

2.1 Procedural Generation: Background

Procedural content generation (usually referred to as simply "procedural generation") refers to the creation of levels and other game objects programmatically and algorithmically, in lieu of a human being doing all the work. While procedural generation algorithms can be used to generate a myriad of things, from textures (for things like trees and clouds) to music ("generative music," as coined by legendary musician Brian Eno), by far its most common context is in automated level design, generating level layouts algorithmically in lieu of work from level designers. Game developers may opt to use procedural generation to save time and money designing levels or show off technical prowess in their games.

Procedural generation in video games has a rich history. Pioneering games such as Rogue (1980) took direct influence from tabletop role-playing games such as Dungeons and Dragons, and thus had a player navigate a randomly-generated world that expanded further as they went on. Such games spawned the *roguelike* and *roguelite* genres, which experienced immense popularity in the last decade. In the realm of first-person shooters, 2004's .kkrieger, as seen in Figure 2.1, used procedural generation to create intricate 3D levels and fit them all into a game that takes up just 96 kilobytes of space.

Figure 2.1: The game .kkrieger, which uses procedural generation to design maps while keeping the game at a 96 kilobyte file size. Source: https://www.researchgate.net/figure/The-game-kkrieger-has-a-file-size-of-96-kb_fig1_320722498

Other games that use procedural generation in its levels include Elite (originally published in 1984), Elite: Dangerous (2012), Minecraft (2009), No Man's Sky (2012) and Spelunky (2013). The latter game's use of procedural generation has notably been covered by video games journalist Mark Brown in a YouTube video.

Figure 2.2: The roguelike game Spelunky, which uses procedural generation to build intricate levels for the player character to explore. Source: https://store.steampowered.com/app/239350/Spelunky/

In many cases, these games end up having a **large** number of different environments that each game could generate for its players. However, by procedurally generating them upon the *loading* of the game level, in lieu of loading a layout from disk, they can save a lot of space (albeit with a considerable need for processing power, depending on the game's and algorithms' performance), as seen with .kkrieger.

Using one or some different procedural generation algorithms, such as the use of Perlin,

Simplex or other noise, Voronoï disks and also poisson disk generation, among others, games can load a seed to randomly generate a level every time it is played, meaning no two playthroughs of a game with procedurally generated content are ever the same.

2.2 Justifying My Choice of Engine: Godot

While a myriad of resources exist for procedurally generated game contents exist for Unity and Unreal, I want to implement them in Godot, for several reasons:

- It's the engine I have the most experience with, having already developed 2 published web games with it.
- It's not got as many resources on procedural generation compared to Unity, Unreal and some other popular game engines, particularly on the side of academic research (that is, there aren't as many papers on procedural generation that pertain to Godot as they do to Unity, Unreal and other engines).
  - However, it is still very powerful and feature-rich (it has its own Open Simplex noise class, for example) and I'm sure I can make procedural generation algorithms work on it.
- Compared to Unity and Unreal, Godot is a very light engine with a feature-rich editor, clocking in at under 100MB, with editors for Windows, macOS, Linux and even the web browser.

By the end of my allotted time, I plan to have implemented several procedurally generated environments in small Godot games, using a myriad of methods (such as Voronoï cell and poisson disk generation) in a myriad of contexts (anything from platformers to first-person games). With these games, I plan for the final report to be the centrepiece of my project, with it containing my research on how each environment was implemented, as well as my findings on the algorithms themselves and how they work.

This is more a research-oriented project than an implementation-oriented project, but the implementations will nonetheless prove that Godot is just as adept at procedural content generation as the other major players in the game engine space, and I will have gained immense knowledge on PCG in the process.

2.2.1 Note on Differing Versions of Godot

Godot currently is at version 3, but concurrently there is also Godot 4, which is nearing its release and is in working condition. The latter version of Godot contains several new features and breaking changes, so any project made in Godot 3 won't readily be compatible with Godot 4 (and vice-versa) without making the necessary changes and conversions. I have access to both versions of Godot and, for all the Godot projects I create, which version I'm using will be clearly denoted and clarified.

Chapter 3

Report Body

3.1 Algorithms

3.1.1 Lindenmayer Systems

Hungarian academic Aristid Lindenmayer devised a mathematical model for the reproduction of fungi in 1967.[2] His model involved a string of symbols, each unique symbol denoting a specific action and/or branch. Essentially, running that initial string, called the *axiom*, through a set of rules (called a *grammar*) gives us an ever-expanding string that is then taken as instructions to draw something from. Lindenmayer Systems, or L-Systems, have since been used in several scenarios beyond its initial purpose of modelling fungi, from trees to fractals. In video games, they are frequently used to aid in the creation of foliage in several environments, as well as buildings and, here, level layouts.

3.1.2 Types of Lindenmayer System

The most basic form of L-System is a *0L*-System, 0 in this case referring to the fact that the grammar is *context-free*.

For this example[1], consider an alphabet V, which consists of the following symbols:

$$\mathrm{F}, +, -$$

where F means "to go forward", and + and − denote turning right or left (respectively) a set number of degrees .

Take an axiom ω, for example:

$$F + F + F + F$$

And a set of rules P which, in this case, is of size 1:

$$F \rightarrow F + F - F - FF + F + F - F$$

We can represent this *parametric* L-system in the following form:[3]

$$G = (V, \omega, P)$$

To implement G in Godot, we can take each rule and replace each string in accordance to our one rule, using the replace method, like so:

A Simple String Replacement for an L-System with 1 rule

```
1    string = string.replace(rule["from"], rule["to"]) #Here the
         rules were stored in dictionaries.
```

Code 3.1: Simple String Replacement for an L-System with 1 rule

The first 3 iterations of this operation are shown here:

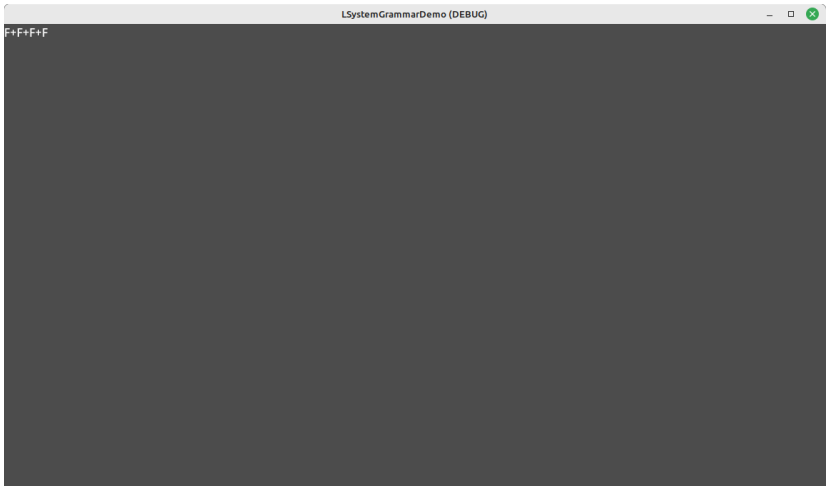

Figure 3.1: The axiom of the aforementioned simple L-System with just one rule. String size: 8. Source: Own work.

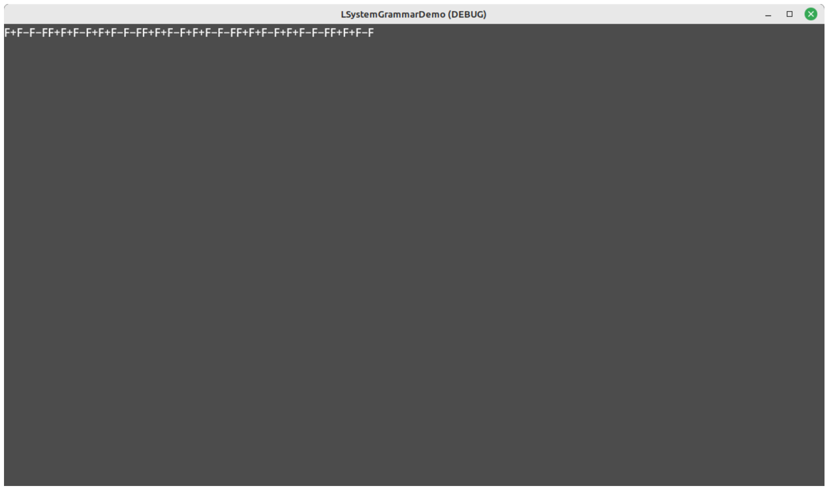

Figure 3.2: The first iteration of the aforementioned simple L-System with just one rule. String size: 59. Source: Own work.

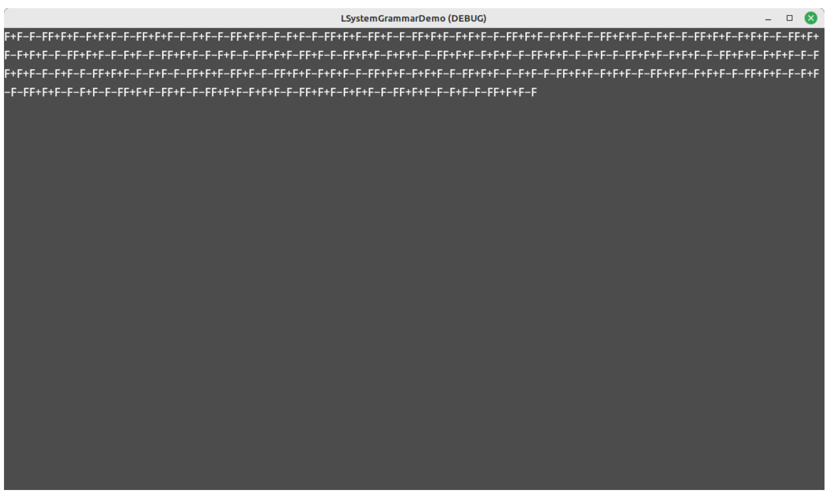

Figure 3.3: The second iteration of the aforementioned simple L-System with just one rule. String size: 475. Source: Own work.

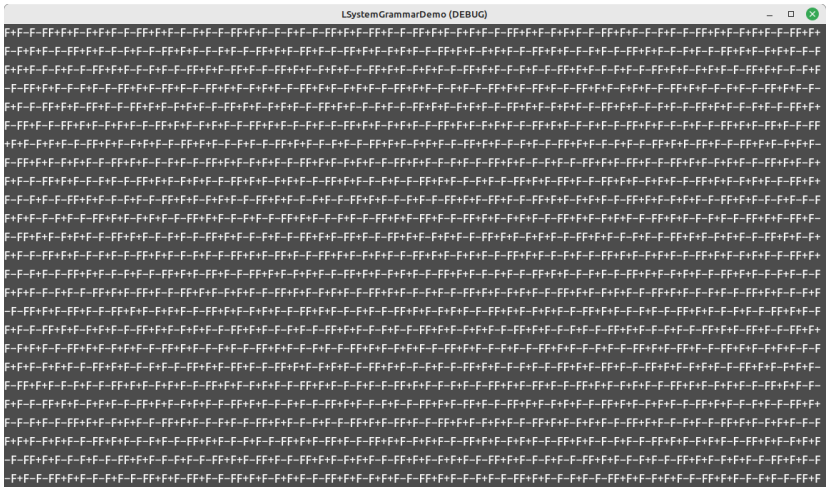

Figure 3.4: The third iteration of the aforementioned simple L-System with just one rule. String size: 3803. The string is too large to show in the window, as you can see here. Source: Own work.

Chapter 4

Design & Specification

4.1 Section Heading

Chapter 5

Implementation

5.1 Section Heading

Chapter 6

Legal, Social, Ethical and Professional Issues

Your report should include a chapter with a reasoned discussion about legal, social ethical and professional issues within the context of your project problem. You should also demonstrate that you are aware of the regulations governing your project area and the Code of Conduct & Code of Good Practice issued by the British Computer Society, and that you have applied their principles, where appropriate, as you carried out your project.

6.1 Section Heading

Chapter 7

Results/Evaluation

7.1 Software Testing

7.2 Section Heading

Chapter 8

Conclusion and Future Work

The project's conclusions should list the key things that have been learnt as a consequence of engaging in your project work. For example, "The use of overloading in C++ provides a very elegant mechanism for transparent parallelisation of sequential programs", or "The overheads of linear-time n-body algorithms makes them computationally less efficient than $O(n \log n)$ algorithms for systems with less than 100000 particles". Avoid tedious personal reflections like "I learned a lot about C++ programming...", or "Simulating colliding galaxies can be real fun...". It is common to finish the report by listing ways in which the project can be taken further. This might, for example, be a plan for turning a piece of software or hardware into a marketable product, or a set of ideas for possibly turning your project into an MPhil or PhD.

References

[1] Paul Bourke. L-system user notes. July 1991.

[2] Aristid Lindenmayer. Mathematical models for cellular interactions in development ii. simple and branching filaments with two-sided inputs. *Journal of Theoretical Biology*, 18(3):300–315, 1968.

[3] Wikipedia contributors. L-system — Wikipedia, the free encyclopedia, 2022. [Online; accessed 23-February-2023].

Appendix A

Extra Information

A.1 Tables, proofs, graphs, test cases, ...

The appendices contain information that is peripheral to the main body of the report. Information typically included in the Appendix are things like tables, proofs, graphs, test cases or any other material that would break up the theme of the text if it appeared in the body of the report. It is necessary to include your source code listings in an appendix that is separate from the body of your written report (see the information on Program Listings below).

Appendix B

User Guide

B.1 Instructions

You must provide an adequate user guide for your software. The guide should provide easily understood instructions on how to use your software. A particularly useful approach is to treat the user guide as a walk-through of a typical session, or set of sessions, which collectively display all of the features of your package. Technical details of how the package works are rarely required. Keep the guide concise and simple. The extensive use of diagrams, illustrating the package in action, can often be particularly helpful. The user guide is sometimes included as a chapter in the main body of the report, but is often better included in an appendix to the main report.

Appendix C

Source Code

C.1 Instructions

Complete source code listings must be submitted as an appendix to the report. The project source codes are usually spread out over several files/units. You should try to help the reader to navigate through your source code by providing a "table of contents" (titles of these files/units and one line descriptions). The first page of the program listings folder must contain the following statement certifying the work as your own: "I verify that I am the sole author of the programs contained in this folder, except where explicitly stated to the contrary". Your (typed) signature and the date should follow this statement.

All work on programs must stop once the code is submitted to KEATS. You are required to keep safely several copies of this version of the program and you must use one of these copies in the project examination. Your examiners may ask to see the last-modified dates of your program files, and may ask you to demonstrate that the program files you use in the project examination are identical to the program files you have uploaded to KEATS. Any attempt to demonstrate code that is not included in your submitted source listings is an attempt to cheat; any such attempt will be reported to the KCL Misconduct Committee.

You may find it easier to firstly generate a PDF of your source code using a text editor and then merge it to the end of your report. There are many free tools available that allow you to merge PDF files.