COMP 4905 Honours Project Report

Procedural Pixel Art Generation

Ashton Woollard (101011084)

Supervisor: Dr. David Mould

Carleton University

School of Computer Science

April 17, 2020

Abstract

The area of procedural generation within games is an ever-expanding field. The number of any kind of asset that can be crafted by hand for any game is finite. Most games resort to reusing a small subset of assets many hundreds of times over to populate the game world. Following the principle that 1000s of unique assets do not increase the core enjoyment of a game. However, this will not always be the case. With procedural generation algorithms, a nearly infinite number of assets are able to be created instantaneously for any scenario. In this project, I investigate and develop robust algorithms capable of generating many variations of pixel-art styled art assets. Through the modification of starting variables while using the same general algorithm different assets are generated. I also investigate interpolation algorithms to transition a distinct game area into a separate, distinct game area.

Acknowledgements

This project is the final project of my undergraduate studies for the Bachelor of Computer Science, within the Computer Game Development Stream, at Carleton University. I wish to thank the Computer Science Undergraduate Advisor Edina Storfer, who helped make this all possible years ago. I would like to thank my supervisor, Dr. David Mould, for overseeing this project and providing guidance and help when necessary. I would like to give additional thanks to individual professors I encountered during my studies including, Donald Bailey, Dr. Jason Hinek, Dr. Chistrine Laurendeau, Dr. Pat Morin, Dr. David Mould, Dr. Oliver Van Kaick, whose teaching helped excel my education. Finally, I would like to thank the Carleton Co-op office and the Carleton University Computer Science Department as a whole.

Table of Contents

List of Figures

1. Introduction

Procedural Generation is an incredibly useful asset when generating 'Tier 2' content[1]. Assets that fall into this category include both 2D and 3D models as well as textures, scene compositions, level layouts, and music. It is the process of algorithmically creating content instead of creating content by hand. It can fail to create quality assets occasionally, so it is not usually suitable for creating the core content and objectives within a game, but when creating background content, it is advantageous. In this project, I focus on developing robust algorithms capable of generating specific 2D Art Assets such as trees, rocks, and mountains in the Pixel Art art style[2], and also developing algorithms capable of generating and interpolating discrete game areas (such as a generic biome), to separate distinct game areas.

1.1 Problem

A developer can only create a finite amount of assets, which are then often repeated to flesh out a virtual world. Usually, players will take these repeated assets at face value and consider the repetition to be 'good enough,' and a limit of developing games in this medium.. Additionally, playing through repeated identical levels, quests, and scenarios can limit the enjoyment of a player and reduce the replayability of a game.

[1] Content that is not hand crafted. Follows the mantra of quantity over quality.
[2] A style of art/games with limited resolution.

By utilizing procedural generation, a developer can instead create an algorithm to generate any manner of assets on the fly, creating a virtually infinite number of assets instead. Thus not only decreases the developer's reliance on reusing assets but also can drastically increase the replayability of a game.

1.2 Motivation

The original motivation for this project was born from a personal project I planned to work on regardless of the requirement to complete an Honours Project. This personal project's purpose was to retain coding behaviours and practices I acquired during my Co-op placements, many of which do not get adequately used and tested in a school environment. By retaining these behaviours, it will assist me when I reacclimate myself into full-time work after graduation.

The original vision for my game was a fantasy inspired, procedurally generated, dungeon crawler similar to games such as Pixel Dungeon (Dolya, 2015) and the Pokemon Mystery Dungeon series (Nintendo, 2020). This original vision morphed into what it is today. The project I completed was focused on the aspect of procedural generation, so the game inspiration transitioned from a top-down dungeon crawler into a side-scrolling dungeon crawler, taking inspiration from games such as Darkest Dungeon (Red Hook Studios, 2016) and Slay The Spire (Mega Crit Games, 2019). By shifting the perspective from top-down, to a side view perspective allowed the focus of the procedural generation to move from level generation to the development of new and varied 2D art assets. By choosing to set the scenery of this project above ground, across biomes also allows the ability to procedurally interpolate between discrete game areas.

1.3 Project Overview

For this project, I split the requirements into two main components, the Asset Generation and the Scene Generation. On top of this, I plan to create a game using these two components. However, this game is not part of this Honours Project.

The Asset Generation component is responsible for generating specific 2D art assets using robust, reusable algorithms. All assets within this project are generated from three distinct algorithms. The Tree Generation Algorithm, the Rock Generation Algorithm, and the Mountain Generation Algorithm. From there, the initial variables (such as colour ramps[3]) are altered to achieve different assets using the same algorithm. The Tree Generator is capable of generating numerous styles of arborescent objects. In this project I use it to generate deciduous trees, coniferous trees, snowy coniferous trees, and cacti. The Rock Generator is capable of generating a variety of rock shaped assets (but is not limited to strictly rocks). In this project I demonstrate it's capabilities by generating objects such as igneous rocks, bushes, and clouds. The final asset generator is the Mountain Generation Algorithm. This algorithm was created to solve a specific problem discussed in Section 3.3: Mountain Generation. Ideally, the generation of mountains and hills are handled by the Scene Generator.

The Scene Generation Algorithm is responsible for building the individual scenes within a discrete game area[4], interpolating between two discrete game areas, while creating a sense of depth in the world. To build a scene, the Scene Generator must first generate the ground from a discrete colour ramp as provided by the developer for a specific area within the game. Next, it generates the background

[3] A discrete set of colours increasing, typically increasing in intensity, hue, brightness, or opacity
[4] An area of the game with unique descriptors, such as a desert.

of the scene in following the idea of a persistent game world (such to say this background may be developed from a separate discrete game area dependent on not only player proximity but alto orientation to nearby separate areas. Finally, the algorithms fill the scene with assets which are either created by hand or generated by the Asset Generator. The Scene Generator also interpolates between discrete game areas, and the algorithm accomplishes this in the same way that it builds a generic scene, except in this case wherein the algorithm receives a ratio value, and two seperate colour ramps for each discrete game area. Finally, the Scene Generator simulates depth of field by utilizing techniques such as parallax scrolling[5] (Stahl, 2013) with several generated bands.

1.4 Outline

As previously introduced in this paper, Section 1: Introduction provides the context for this project. Within this Section is Section 1.1: The Problem, which describes the need for procedural generation, Section 1.2 Motivation, which describes the motivation behind the origination of this project, Section 1.3: Project Overview, which describes a summary of the project and its major components, and Section 1.3: Outline, which describes a summary of each section within this report.

In Section 2: Background, basic concepts necessary for understanding this project are explored. Within this section is Section 2.1: Pixel Art, which describes the pixel-art art style in which this project operates within and why this project utilizes this art style, Section 2.2: Procedural Generation, which describes the basic concepts of procedural generation, and Section 2.3 Engine, which describes the engine and language required for development in this project and why these choices were made.

[5] Placing 2D objects on discrete planes and manipulating each of those planes differently from one another.

In Section 3: Asset Generation, specific algorithms, problems, and solutions regarding the Asset Generation algorithms are described. Within this section is Section 3.1: Tree Generation, which discusses the Tree Generation algorithm in detail, Section 3.2: Rock Generation, which discusses the Rock Generation algorithm in detail, and Section 3.3: Mountain Generation, which discusses the Mountain Generation algorithm in detail, as well as why the need for this algorithm arose.

In Section 4: Scene Generation, specific algorithms, problems, and solutions regarding the Scene Generation algorithms are discussed. Within this section is Section 4.1: Biome Generation, which describes the algorithms and processes utilized to synthesize scenes.Section 4.2 Persistent Discrete World Areas, discusses the requirement and problems associated with generating far from player discrete game areas.Section 4.3: Biome Interpolation, discusses the algorithms used to synthesize interpolated scenes between discrete game areas. Section 4.4: Simulated Depth of Field, discusses the need, use, and application of parallax scrolling (Stahl, 2013) in regards to this project.

In Section 5: Optimization, specific problems that arose within this project in regards to optimization and performance are addressed. Within this section is, Section 5.1: Optimization in Asset Generation, which discusses the optimization techniques used within the Asset Generation component to increase performance, and Section 5.2: Optimization in Scene Generation, which discusses the optimization techniques required by the Scene Generator component to reach an acceptable performance level.

In Section 6: Conclusion, reflects on specific aspects of this project. Within Section 6.1 Review Goals, the completed work as well as what intended to be completed within the original proposal are addressed as well as compared., Section 6.2: Unsolved Problems, discusses specific problems that still exist within the project.Section 6.3: Future Improvements, details specific improvements to the Asset Generator and Scene Generator components that may be done in the future. Section 6.4: Future Work, discusses future work to be implemented to turn this project from a simulation into a real game.

Finally, Section 7: References, notesthe resources and inspiration used for this project.

2. Background

Within the context of this project, a small amount of background is necessary to understand certain design choices, as well as the implementation and significance of certain decisions. Procedural Generation is a large area of study and practice. However, it is not typically used to generate pixel art style assets. It is typically used to generate 3D models and textures.

2.1 Pixel Art

Pixel Art is an art that uses the precise placement of pixels to create a larger image. The resolution of pixel art is usually very low, and so certain techniques and styles must be used to achieve pleasant effects. Usually, pixel art is created by hand with very deliberate design choices in regards to shape, colour, and shading being made because of the medium.

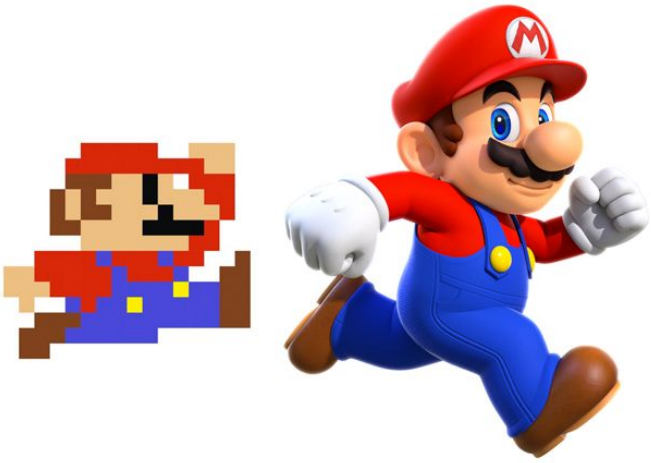

FIGURE 1: Shape language and representation in low-resolution images.
(Solarski, 2018) (Image courtesy of Nintendo)

Pixel art uses exaggerated shape language to convey different shapes with only a few pixels. Fig.1 above demonstrates an example of the successful application of shape to represent a much different object. Within the pixel art variation, proportions and limb positions are vastly exaggerated compared to the high-resolution counterpart. It is also apparent inFig.1 how limited the colour canvas becomes when working in low resolution. Generally, pixel art uses a discrete set of colours to simplify images. At such extremely low resolutions such as Fig.2, shading is not typically used, but techniques such as dithering or blurring are often used in higher resolution pixel art, which can be observed in Fig.2.

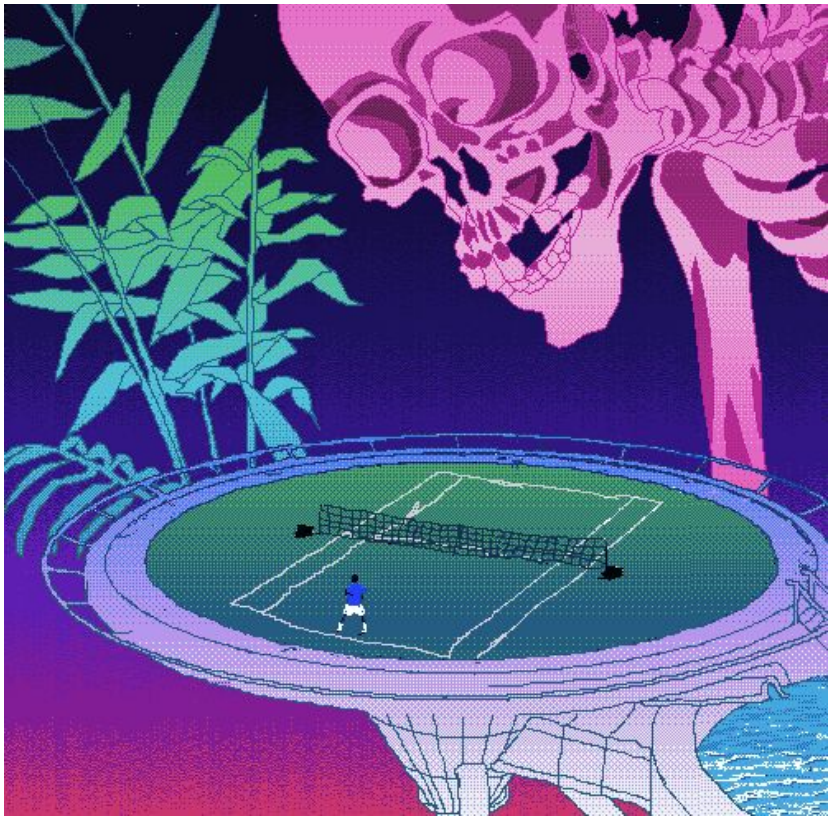

FIGURE 2: Example of shading and blending using dithering within Pixel Art (Cameos, 2014)

2.2 Procedural Generation

Typically, procedural generation is used for the creation of 3D models, 2D textures, and gameplay elements within games (such as level design). It is the process in which an algorithm can synthesize a chunk of data. An excellent example of procedural generation is present in Fig.3, which demonstrates a game called No Man's Sky (Hello Games, 2016), which famously generates almost all 3D assets and textures within the game using procedural generation.

FIGURE 3: Procedurally Generated Landscapes in No Man's Sky
(Khatchadourian, 2015) (Image courtesy of Hello Games)

Typically, the concept of procedural generation conflicts with the general design philosophy of pixel art. Pixel art is about the careful placement of pixels by hand, while procedural generation is focused on the larger picture, and often breaks down when dealing within the fine details of an extensive system.

2.3 Engine Choice

This project has been written in C++[6] and uses the Godot Engine (3.1.2) (Verschelde, 2019). A few options were available when choosing what engine to write this project with, including using a custom engine developed by myself, Unity (Unity, 2005), Unreal (Unreal, 2004), and Godot (Godot, 2007). Godot Engine 3 is a relatively young engine that was chosen for a few reasons. Firstly, it is entirely open-source, which means that if necessary, I can modify the engine myself if the engine is not capable of performing to expectations. Secondly, it is a practical choice for pixel art style games, making the process of developing these styles of games very streamlined. And finally, Godot is capable of using a variety of programming languages with the appropriate library, including C++, C#[7], Rust[8], and it's custom language GDScript[9]. Other mainstream engines require the use of a specific language, and a requirement of mine was to be able to write in C++. Unreal also uses C++ as its core language; however, Unreal is not as useful for pixel art games and is used more often when developing 3D realistic games.

[6] Object orientated language often used in video game development
[7] Object orientated language, used by game engines such as Unity
[8] Object orientated language similar to C++ but provides memory safety and it thus less error prone
[9] Custom object orientated language developed by the Godot team, semantically similar to python

3. Asset Generation

The Asset Generator component of the project utilizes general-purpose and robust algorithms to develop 2D pixel art assets. For this project, I implemented two such general purpose algorithms, called the Tree Generation algorithm and the Rock Generation algorithm. These algorithms are not limited to generating just trees or rocks, but those are the terms I use to describe the shape of objects it does generate. The algorithms are capable of taking a set of variables (such as discrete colour ramps) and using those variables to generate different objects. For example, the same algorithm that generates coniferous trees also generates cacti.

3.1 Tree Generation

For this project, tree-like structures are categorized as objects such as deciduous trees, coniferous trees, and cacti. Other possible tree structures could be as far as to include objects such as skyscrapers and statues. The tree generation algorithm works in four distinct steps, not including the final 'blur step,' all of which can be observed in Fig.4.

FIGURE 4: Tree Generation Algorithm for Deciduous, Coniferous, and Cactus Trees

The first step calculated the location and shape of the trunk using various parameters. The first thing that gets determined is the width of the trunk at its base, the height of the trunk, and then the number of nodes within the trunk (all are determined based on input parameters). Nodes are distinct

sections of the trunk with varying widths and growth directions (determined based on input parameters). Branches also connect to the tree at the location nodes that occur. Nodes are precisely and evenly spaced throughout the trunk. All the nodes are then connected, and the inside of the trunk is filled (using colour from input colour ramp).

The second step colours the trunk by adding extended highlights and shadows to the trunk; these lines are drawn between nodes. For the cactus above, it can be observed in Fig.4 that shadows are not added to the trunk, this is because a cactus' trunk is not grooved like other tree trunks are. To increase the variability of both the highlights and lowlights, several random pixels have their colour intensity index along the given colour ramp increased or decreased by one.

Branches are now added, each branch is made of two sections. Each section is connected end to end and rotated relative to the trunk based on input parameters. The number of branches is determined randomly (between two input parameters), while branch length is based on height and whether branches are being mirrored. If branches are mirrored on both sides of the tree, then the length is proportional to the distance from the top of the trunk. If the branches are not mirrored, then the length is inversely proportional to the distance from the top of the trunk. Calculating branch length like this is a tweak on the algorithm that is unique to the pixel art medium, and allows the creation of exaggerated shape language in low resolution. At the same time, these branches are coloured in the same way the trunk is coloured.

The fourth step is adding leaves, accomplished by using a spray function at the end of every branch and the top of the trunk. The spray function selects a random number of pixels (based on input parameter intensity) within a radius from the center, and these pixels are set to the colour determined in

the input colour ramp for leaves. The radius is determined in the same way branch length is determined. When branches are mirrored, it is proportional to distance from top of the tree; otherwise, it is inversely proportional to the distance from the top of the trunk. Similar to above, this is done to exaggerate the shape language of the object. This spray function is then repeated twice, each time with a slightly smaller radius, a randomly shifted center point (shifted up and to the left), and the next colour within the input colour ramp. Certain artifacts such as snow are then added. To add snow to the tree, every time a leaf cluster is added, the top-most couple of exposed[10] pixels within the sprite are modified to a new given input colour.

Finally, if performing the optional blurring step, each pixel is set to the average colour between it and its neighbours. If adding other artifacts such as fruit or flowers, a random number of points are chosen (between two input parameters) to be modified, along with their neighbours, to have their colour changed to the artifact colour. An example of the different input parameters that develop the assets shown in Fig.4 can be seen in Appendix A: Input Parameters of Deciduous, Coniferous, and Cactus Trees.

[10] A pixel is exposed if all pixels above it are not set

3.2 Rock Generation

This project categorizes rocks as in sort of broad, bulbous shaped objects. In this project, graphite rocks, bushes, and clouds use the rock generation algorithm to be synthesized. The rock generation algorithm works three distinct steps, with a 4th optional ‘blur step.’. Fig.5 demonstrates this process for a graphite rock. Note that Fig.5 also displays the process for generating a bush and a cloud, however, these shapes do not render anything to screen for the first two steps of the algorithm.

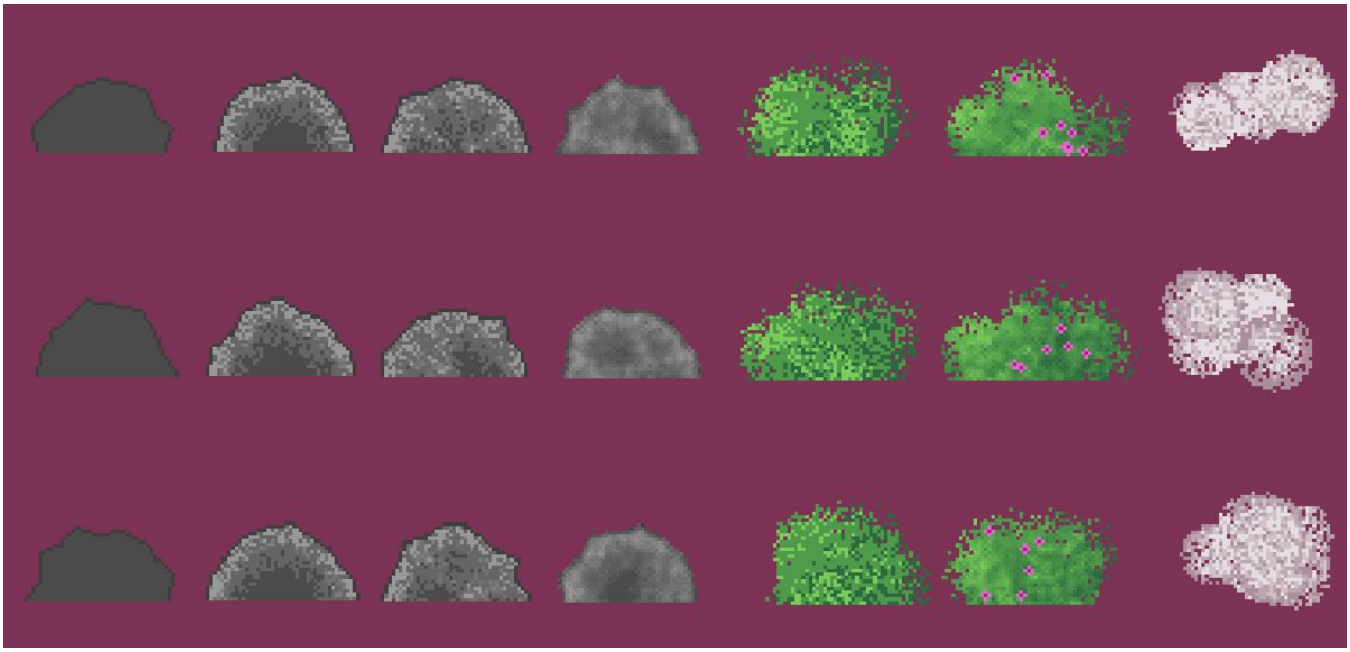

FIGURE 5: Rock Generation Algorithm for Graphite Rock, Bush, and Cloud

The first step is to develop an outline of the object. This process is accomplished by first determining a random number of points (between two input parameters) that will follow along the edge of the rock. These points will all be placed an equal distance from the center of mass with a constant angle between each point: such that the first point lies along the ground on the left, and the last point lies

along the ground on the right. Each point is then assigned a variance score (between two input parameters), which is a random deviation from its original position. The point is moved closer or farther from the center of mass, depending on the variance score. Now each point is connected to form the outline of the object, and the object is filled.

The second step is to add faces[11] along the edge of the rock by using a spray function at edge points along the rock's edge. The spray function selects a number of pixels (based on input parameter intensity) within a radius (another input parameter). These selected pixels will have their intensity index increased by one along the colour ramp of the object. This process is repeated with twice the original intensity and half the original radius.

The third step adds faces throughout the object. For bushes and clouds, this is the first step that takes place. A similar process to above takes place; however, instead of spraying along the edge of the outline, a line is created that intersects two points along the edge of the object, the above process of using the spray function twice takes place on points along this new line. The process of finding a line intersecting two points and spraying is repeated a random number of times between two input parameters.

Finally, if performing the optional blurring step, each pixel is modified to be the average between it and all of its neighbours. If adding artifacts such as flowers or gem deposits, a random number of points are now chosen (between two input parameters) to be modified, along with their neighbours, to have their colour changed to the input artifact colour.

[11] A relatively flat area of a rock that reflects light similarly. In the context of this project, it is a patch of area similarly coloured.

3.3 Mountain Generation

The creation of the Mountain Generation algorithm was a failure of the Scene Generation component. Ideally, mountains would be created in the background and displayed the same way the ground and sky textures are displayed using the Scene Generator. When developing the mountain generation algorithm within the Scene Generator, I reached a limitation within the architecture I previously created that prevented me from creating pleasant-looking mountains. The issue is that it was not possible to draw shapes within a texture. It was possible to draw the silhouette of the mountains, but it was not possible to draw mountains on top of other mountains. The prototype final result as shown in Fig.6, was considered unsatisfactory due to the 'flat' appearance of these mountains. A method to add depth to these mountains was not found within the constraints of Scene Generator. The previous mountain silhouette algorithm generated a weighted graph, then used a weighted A* Search (Ebendt & Drechsler, 2009) to calculate the shape of the mountains.

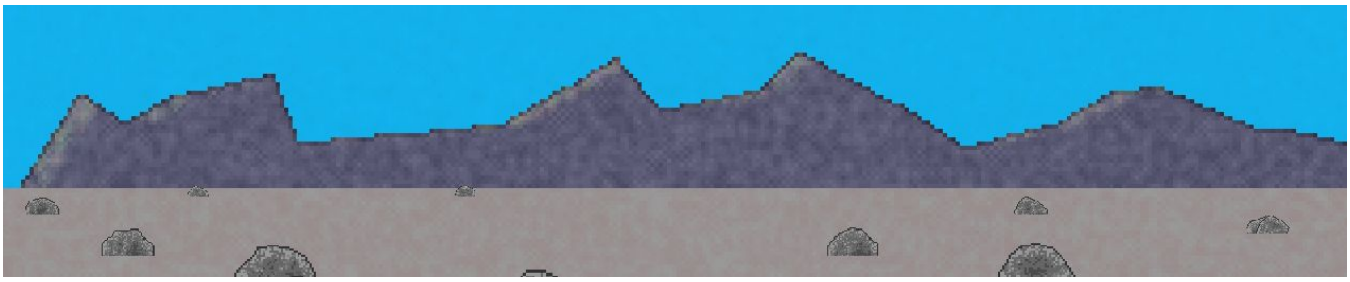

FIGURE 6: Prototype mountain range generated using Scene Generator

Unfortunately, the best method for creating mountains without significant rework of the Scene Generator was to use the Asset Generator; however, the Asset Generator had limitations of its own. The Asset Generator assumed that sprites would always be square, which means that in the case of this

project, all sprites fit within a 64x64 image. It is possible to modify the size of the sprite, but a core assumption was that sprites would be square. As such, it was not possible to create a single long sprite within the constraints of the Asset Generator. The solution used was to create numerous small mountains and overlay them horizontally to create the appearance of a mountain range. An example of such can be seen in Fig.7.

The current algorithm works by first defining a maximum peak of the mountain based on the input parameter height. The second step determines the number of inflection points, which is proportionate to the max height of the mountain. The location of these inflection points is defined by calculating a variance amount, then deviating a point from along the line defined by the average slope (maximum peak to the ground). With these inflection points defined, the points are connected to define the outline of the mountain, then the shape is filled with the input colours, and the sprite is blurred. Overlaying multiple mountains next to each other is done to create the mountain range. There do exist some benefits to using the Asset Generator to synthesize mountains, mountains can exist on multiple parallax layers, and can overlap with clouds to create depth. However, neither option is considered satisfactory, and in the future, this feature will be revisited.

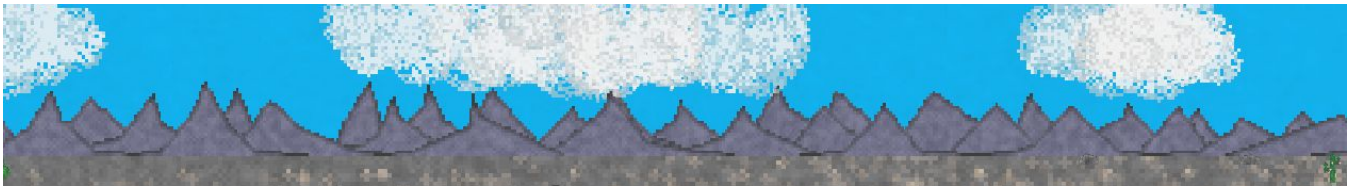

FIGURE 7: Mountain Range generated using Asset Generator

4. Scene Generation

The Scene Generator component of the projects houses four important algorithms conducive to the creation and management of the discrete game areas. Each component on its own is relatively simple. The first component is the Biome Generation, which takes simple parameters such as a discrete colour ramp, and what objects belong in the biome, and synthesizes a scene of the biome. The second component is the Persistent Discrete World Areas, which is just a way of saying that what gets generated in the distance is not what necessarily gets generated in the foreground. The third component is Biome Interpolation, which handles the blending of a discrete game area into a separate discrete game area. The final component is the Simulated Depth of Field, which simulates the appearance of depth in the world.

4.1 Biome Generation

The Biome Generation component synthesizes a scene using only a couple of input parameters. The first input parameter is the set of colours that is used to create the scene. Each pixel of the ground texture is set to one of these input colours. The Asset Generator can also be given this colour ramp to create assets that match the colour scheme of the scene. The sky is similarly textured with a different colour ramp. Each pixel of the ground texture is then averaged with its neighbours to find the blended/average colour value. Assets are then generated if necessary, using the Asset Generator and randomly placed throughout the scene. An example scene composition can be seen in Fig.8. The assets generated in this scene are clouds, mountains, cacti, and graphite rocks.

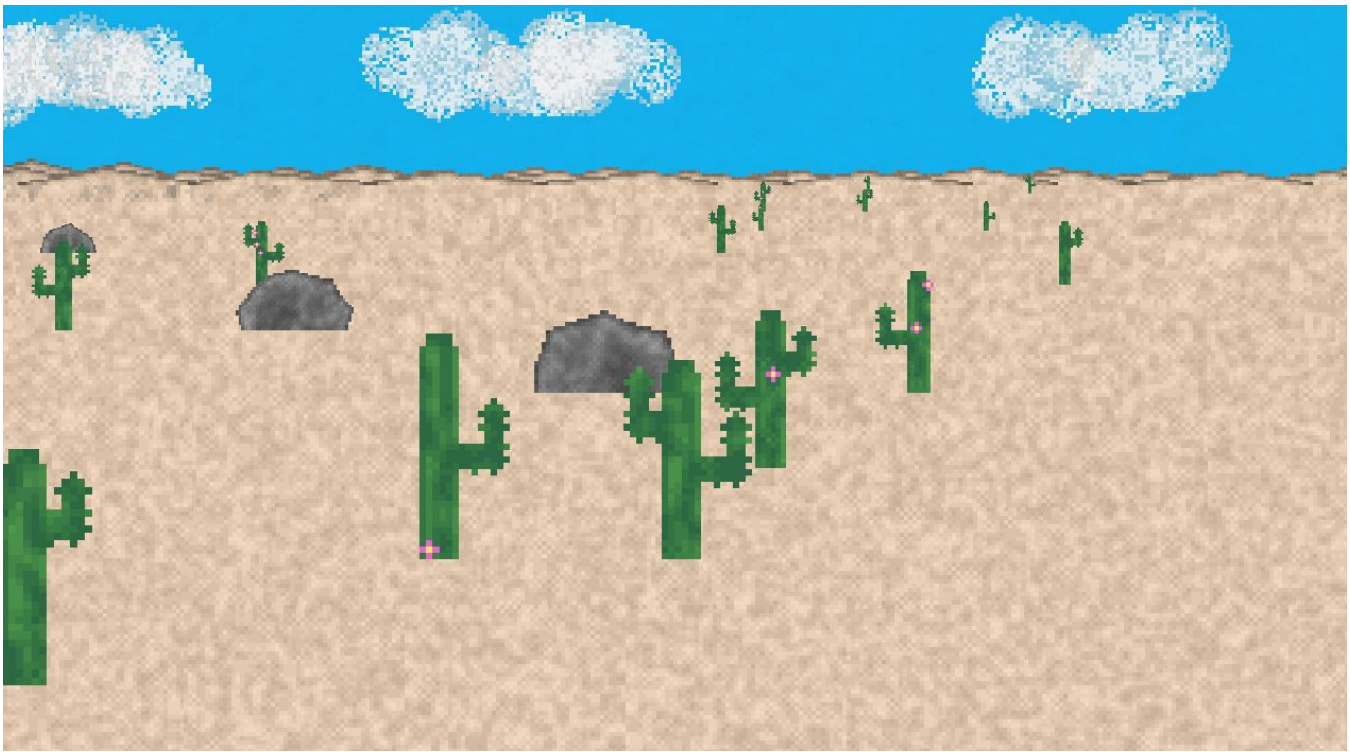

FIGURE 8: Desert Scene Generated by Scene Generator

4.2 Persistent Discrete World Areas

The inspiration between Persistent Discrete World Areas is that the world should exist outside the infinitesimal point the player stands on. For 3D games, this concept is trivial, you can always see what is around you, but many 2D games forsake this concept and only show you a tiny chunk of the world. In this context, what the Scene Generator builds into the background is not always relevant to the biome you are currently in, but if you are walking by a mountain range, the player can expect to see the mountain range in the distance. An example of this can be seen below in Fig.9. The process to implement this feature is simple in concept, textures and assets far from the player (in the background) are generated using input parameters from neighbouring world areas.

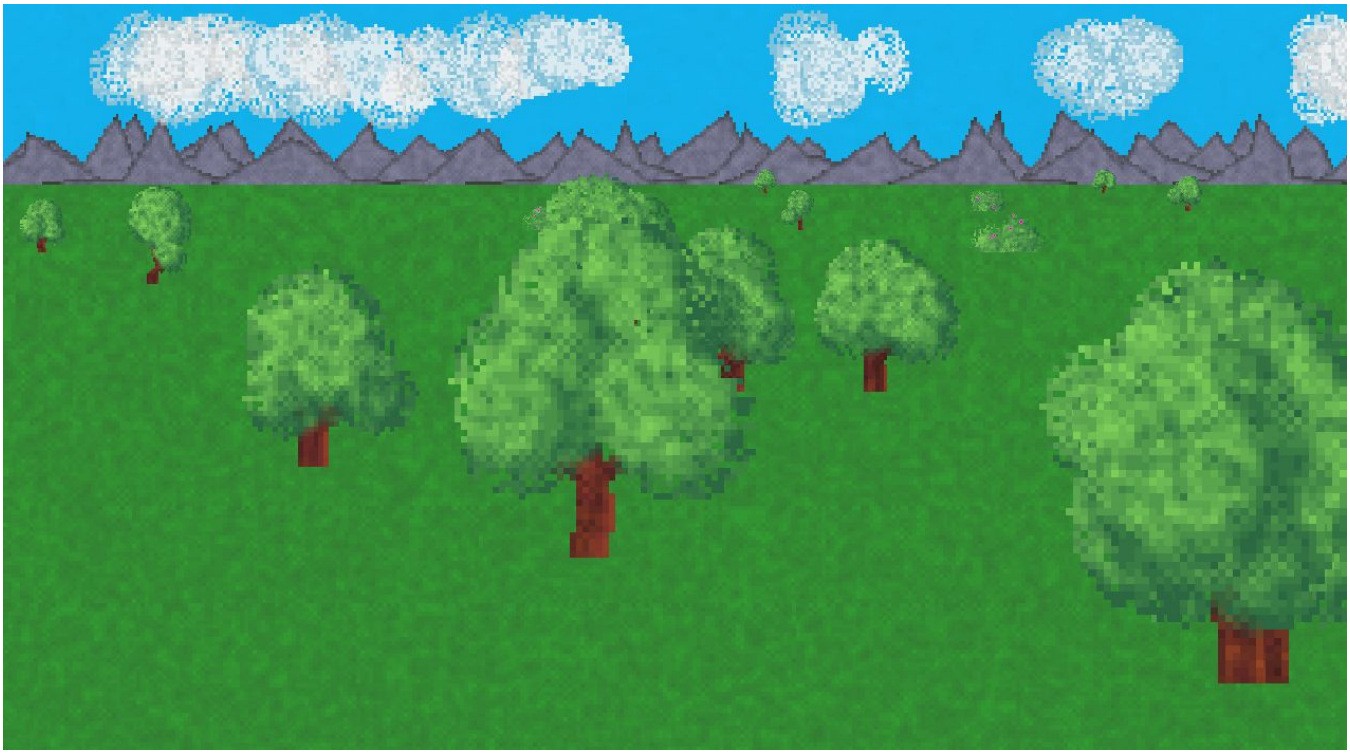

FIGURE 9: Nearby Mountain Features, Visible from Local Meadow

4.3 Biome Interpolation

This component is responsible for blending multiple discrete world areas. It takes a set of input parameters, each biome's unique features (colour ramps and specific asset input parameters), and then a ratio value corresponding to the ratio of one area to the other. It then generates a scene using the Biome Generation component, albeit with a mixture of two biome's input parameters. An example of this biome interpolation is displayed in Fig.10.

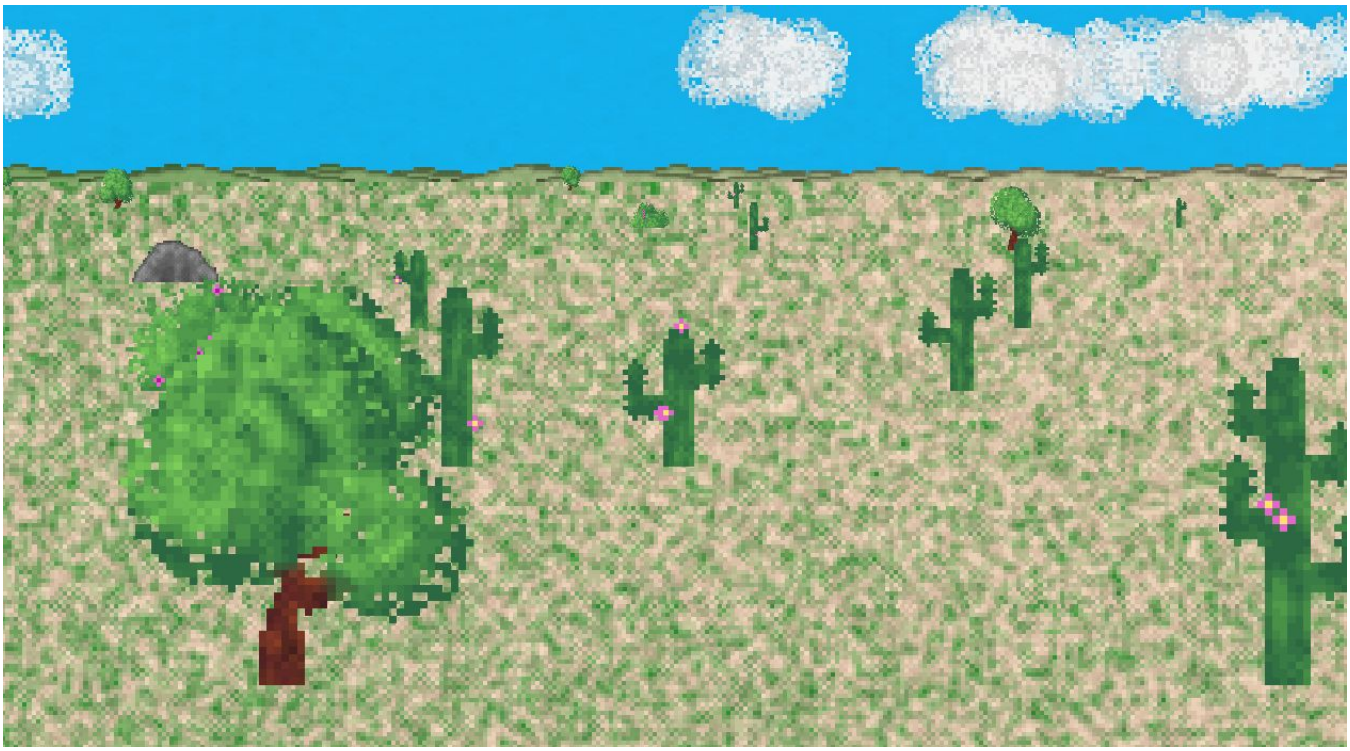

FIGURE 10: A Meadow Biome Blending into a Desert Biome with Biome Interpolation

4.4 Simulated Depth of Field

Typical 2D pixel art games are drawn in an isometric style, which is naturally conducive to the art medium; however, for this project, depth of field is essential to simulate the feeling of existing in a massive open world. This feature works by splitting the scene up into a series of layers, each of different distances from the camera. Each layer scales objects differently, and most importantly, the Scene Generation takes advantage of a technique called parallax scrolling (Stahl, 2013). Which functions by scrolling each of these layers at different speeds respective to the player. The effect that occurs is the closer layers scroll faster while farther layers scroll slower, which simulates how a landscape is perceived as a 3D environment.

A concession was made here when optimization efforts were implemented. Once frame rate increased, 'jitters'[12] appeared when scrolling parallax layers that were all locked to the same pixel grid. These 'jitters' appeared depending on the exact distance scrolled between specific frames. A jitter is described as occurring when a far parallax layer would scroll a single pixel, while a close parallax layer would not scroll at all. The consequences would break the effect of Simulated Depth of Field, and so the concession to lock each parallax layer to its own pixel grid was made.

[12] An irregular animation pattern that breaks game immersion

5. Optimization

Optimization issues were prevalent enough within this project that they deserve a section dedicated to discussing issues that arose and their eventual solution. Before the specifics of how the project was optimized can be discussed, the custom architecture that allows the project to operate must be discussed. In the original prototype, the Asset Generator and Scene Generator handled and calculated every pixel before every draw call; therefores allowing the manipulation of every pixel on an individual basis, asevery pixel can be carefully calculated and placed to ensure the highest level of accuracy. However, such accuracy came at a steep cost in performance.With this cost in mind, the architecture was modified to process an approximation of the calculated images, rather than a perfectly calculated composition.

5.1 Optimizations in Scene Generation

The most substantial obstruction to optimization was the Scene Generator. The prototype Scene Generator calculated every pixel carefully, placing each pixel before drawing each pixel. The total number of pixels that were stored, calculated, manipulated, and then individually rendered to the screen by the Scene Generator was approximately 60,000 pixels.

The solution to this optimization problem was to modify the architecture to create, calculate, and then cache the pixels as a chunk of pixels in a single texture image. The size of this chunk is determined based on each parallax layer's scroll speed, but on average, each chunk contained approximately 700 pixels. Each chunk of pixels could then be stored, manipulated, and rendered to the screen as a chunk.

Using this new architecture does have some overhead costs, which can mostly be ignored. Firstly, there is an additional memory overhead when manipulating an entire image texture compared to manipulating the individual chunks, but this memory overhead is mostly negligible within the context of this project. Secondly, when generating a chunk using Biome Interpolation, biome colour density must be estimated through the chunk instead of accurately calculated. Thirdly, finding blended colours between parallax layers becomes impossible. Because the adjacent pixels between parallax layers is not constant, a calculated average between pixels at every frame is not possible. Instead, this Scene Generator takes advantage of alpha blending[13] and overlapping parallax layers. Finally, the last overhead cost is that each parallax layer is no longer locked to a specific grid. In most pixel art games, all pixels on screen fit within a grid, such that two seperate objects will end up aligned, even if they are rendered separately. A concession was made here to instead enforce the pixel grid only within parallax layers and instead render and scrolling each layer independently of each other. The problem that is caused when this concession is not made is described in Section 4.4: Simulated Depth of Field.

[13] Overlaying textures and images with reduced opacity on top of other images and textures

5.2 Optimizations in Asset Generation

Similar optimizations to the Scene Generation were also made within the Asset Generator, but for different reasons. In the prototype Asset Generator, each asset generated stored, calculated, and rendered to screen approximately 4000 individual pixels. While each pixel was only calculated once, each pixel was needing to be individually rendered to screen each frame. Following the same optimizations made within the Scene Generator, the architecture was modified to calculate every pixel of a sprite, store the sprite inside an image texture, and draw to the screen with a single render call. The overheads of such an optimization are limited to using a more significant; but still negligible amount of memory.

6. Conclusion

Overall, this project was successful and completed every required goal in some capacity. More work is required to flesh out the algorithms and to find new robust algorithms. Additionally, the possibility of creating a game on top of this architecture is still mostly possible. Originally, a game was intended to be implemented and submitted along with this project, but that quickly proved infeasible with the scale of AssetGenerator and SceneGenerator.

6.1 Review Goals

In the original proposal, a series of goals for the Asset Generator and the Scene Generator were created. For the Asset Generator, the creation of an algorithm capable of spawning rocks/boulders, coniferous trees, deciduous trees, dead trees, and cacti was the goal. For this goal, the original algorithm was split into two seperate robust algorithms. The first was the rock generation algorithm, the second was the tree generation algorithm. I did not dedicate resources to finding a set of input parameters to create dead trees, but within Fig.4, a good approximation for what these might look like can be seen. Additionally, input parameters for bushes, clouds, and snowy trees were also found and demonstrated within this project.

For the Scene Generator, some features were cut when most of the actual game elements of the game were separated from the rest of the project. The first feature not implemented was the inclusion of assets not generated by the Asset Generator. To implement such a feature would be trivial, hand created assets would require storing the image texture within an Asset object identically to how the Asset Generator stores assets after they are created. The hurdle with demonstrating this feature was creating

the art assets by hand to implement. The second feature that was not implemented was the concept of interactables within the scene. Interactables would be better implemented within the game architecture rather than the generation algorithms because interactables are not relevant to the Scene Generator as a whole.

6.2 Unsolved Problems

With the current state of the project, problems still exist within the Mountain Generation algorithms and the method of synthesizing mountain ranges. A reasonable solution has been implemented, but a truly effective solution still needs to be found. An active Mountain Range generation algorithm would require changing some core assumptions the architecture makes within the Asset and Scene Generators.

6.3 Future Improvements

As mentioned at length throughout this report, additional work is required to improve the Mountain Generation Algorithms. Two approaches are currently considered. The first would be modifying the Scene Generator component and changing some core assumptions the architecture makes. Expressly, the architecture assumes that everything it generates will be drawn as a flat object. As such, it is not currently possible to add 3D shading to mountains or have them interact and overlap other assets within a scene, such as clouds. The second approach would be modifying the Asset Generator component and modifying the architecture to no longer make a core assumption that every asset can be drawn within a square sprite, allowing the generation of differently shaped assets. Doing so would allow the creation of a vast, multi-screen spanning asset, in which a mountain range could be generated within.

More optimizations would also have to be made to generate an image of this size fast. As a complement to the Mountain Generation Algorithm development, it would be beneficial to increase the capabilities and influence of the Persistent Discrete Game Area algorithms. Theoretically, it would be beneficial to incorporate the Biome Interpolation algorithm into the Persistent Discrete Game Area algorithm, thus game areas could be interpolated horizontally, as well as vertically. Doing so would create the basis of being able to move in two dimensions.

Additional improvements can also be made to the Asset Generator component. Firstly, continual tuning of the generation algorithms is always beneficial. Secondly, finding more input parameter sets to create various other objects using the Rock Generation Algorithm and Tree Generation Algorithm. It is believed that with more tuning and a new set of parameters, these algorithms can become even more robust and be used to generate a plethora of assets, including buildings, lakes, rivers, mountains, creatures, and so on. When expanded, these algorithms would be coalesced into a single algorithm with a huge number of input parameters.

6.4 Future Work

In the future, should this project be revisited, improvements detailed in Section 6.3: Future Improvements would be addressed and rebuilt. Additionally, as mentioned throughout this report, the creation of a game that utilizes the procedural art generation as a means of filling the game world with assets was initially planned, but cut from the project. It was originally cut due to it being feasible to implement a game as well as the generation algorithms within the time frame of one semester while attending school and finishing my degree. And secondly, game elements were not the core aspect of what this project was exploring. This project was exploring the way procedural generation can be used with the pixel art medium, a game on top of this was simply a means to package the generation algorithms up in such a way to present them.

7. References

Mega Crit Games. (2019, January). Slay the Spire. Retrieved April 2020, from https://store.steampowered.com/app/646570/Slay_the_Spire/

Cameos. (2014). Hey, What are you Doing? Retrieved April 2020, from https://66.media.tumblr.com/b68aaf02b88e102effd968218bdbeca0/tumblr_mzzgasop4W1qmf2zwo1_500.png

Dolya, O. (2015). Pixel Dungeon. Retrieved April 2020, from http://pixeldungeon.watabou.ru/

Ebendt, R., & Drechsler, R. (2009, June 16). Weighted A∗ search – unifying view and application. Retrieved April 2020, from https://www.sciencedirect.com/science/article/pii/S000437020900068X

Godot. (2007). Godot Engine. Retrieved April 2020, from https://godotengine.org/

Hello Games. (2016, August). No Man's Sky. Retrieved April 2020, from https://www.nomanssky.com/

Khatchadourian, R. (2015, May). The Galaxy-Sized Video Game. Retrieved April 2020, from https://www.newyorker.com/magazine/2015/05/18/world-without-end-raffi-khatchadourian

Nintendo. (2020, March). Pokémon Mystery Dungeon. Retrieved April 2020, from https://mysterydungeon.pokemon.com/en-us/gameplay/

Red Hook Studios. (2016, January). Darkest Dungeon. Retrieved April 2020, from https://store.steampowered.com/app/262060/Darkest_Dungeon/

Solarski . (2018, August). ADAPTIVE GAMEPLAY AESTHETICS (PART 2): A Disruptive Game Design Framework. Retrieved from https://www.gamasutra.com/blogs/ChrisSolarski/20180820/324832/ADAPTIVE_GAMEPLAY_AESTHETICS_PART_2_A_Disruptive_Game_Design_Framework.php

Stahl, T. (2013, March). Chronology of the History of Video Games. Retrieved April 2020, from https://www.thocp.net/software/games/golden_age.htm

Unity. (2015, June). Unity Real-Time Engine. Retrieved April 2020, from https://unity.com/

Unreal. (2004). Unreal Engine: The most powerful real-time 3D creation platform. Retrieved April 2020, from https://www.unrealengine.com/en-US/

Verschelde. (2019, November). Maintenance release: Godot 3.1.2. Retrieved April 2020, from https://godotengine.org/article/maintenance-release-godot-3-1-2

Appendix A: Input Parameters of Deciduous, Coniferous, and Cactus Trees

```
***Deciduous Trees***

mTrunkColorRamp = [Color(0.275, 0.125, 0.125), Color(0.450, 0.175, 0.175), Color(0.600,
0.275, 0.155)];
mLeafColorRamp= [Color(0.175, 0.410, 0.255), Color(0.315, 0.610, 0.295), Color(0.490,
0.805, 0.355)];

mMinHeight = 32;
mMaxHeight = 44;
mMinNodes = 6;
mMaxNodes = 10;
mMinDistBetweenNodes = 5;
mMaxDistBetweenNodes = 7;
mMaxOffsetFromCenter = 2;
mMinBranch = 2;
mMaxBranch = 5;
BranchAngle1 = Vector2(6,2);
mBranchAngle2 = Vector2(3,1);
mLeafDensity = 1.0;
mMirrorBranches = false;
mSnow = false;
mTaperTrunk = true;
mNumFlowers = 0;
mBlur = true;


*** Snow Covered Coniferous Trees***

mTrunkColorRamp = [Color(0.275, 0.125, 0.125), Color(0.450, 0.175, 0.175), Color(0.600,
0.275, 0.155)];
mLeafColorRamp= [Color(0.100, 0.200, 0.120), Color(0.175, 0.410, 0.255), Color(0.250,
0.510, 0.295)];

mMinHeight = 32;
mMaxHeight = 44;
mMinNodes = 6;
mMaxNodes = 10;
mMinDistBetweenNodes = 5;
mMaxDistBetweenNodes = 7;
mMaxOffsetFromCenter = 0;
mMinBranch = 20;
mMaxBranch = 30;
mBranchAngle1 = Vector2(6, 0);
mBranchAngle2 = Vector2(3, 0);
mLeafDensity = 1.0;
mMirrorBranches = true;
mSnow = true;
mTaperTrunk = true;
mNumFlowers = 0;
mBlur = true;
```

```
*** Cacti ***

mTrunkColorRamp = [Color(0.175, 0.410, 0.255), Color(0.175, 0.410, 0.255), Color(0.315,
0.610, 0.295), Color(0.490, 0.805, 0.355)];
mLeafColorRamp = [];

mMinHeight = 32;
mMaxHeight = 44;
mMinNodes = 5;
mMaxNodes = 6;
mMinDistBetweenNodes = 3;
mMaxDistBetweenNodes = 7;
mMaxOffsetFromCenter = 0;
mMinBranch = 1;
mMaxBranch = 2;
mBranchAngle1 = 0;
mBranchAngle2 = 90;
mLeafDensity = 0;
mMirrorBranches = false;
mSnow = false;
mTaperTrunk = false;
mNumFlowers = rand() % 5 - 1;
mBlur = true;
```