Parallelization of Procedural Content Generation

Rachel, Hunter, Cay, Ben, Ryan

I. Introduction

Our team is interested in taking parallelization into the game design field. Looking to games like Minecraft for inspiration we aim to investigate terrain and mesh generation broadly known as procedural content generation. Though we are programmers with different skill sets, we hope to reach a greater understanding of procedural content generation through sharing our knowledge and enhancing our individual work to eventually produce a unified conclusion. We will conduct our research through parallelized algorithms commonly used for such content generation.

II. Problem Statement

When creating video games, simulators, and other 3D world environments, a common problem arises with procedural content generation (PCG). Large landscapes, complex textures, and large-level assortments can be a taxing computation load on a computer and lead to difficulties that extend the time needed for game development. Hastening these procedures is a priority as software that utilizes content generation largely suffers from even minute delays. This can cause issues with simulation accuracy, hinder the player experience, and cause frustration on the side of both developer and player.

III. Challenges

Finding good metrics to cross-compare these algorithms and combinations of certain algorithms has been and will continue to be a challenge. Some like Perlin noise and Worley noise are better suited for generating terrain while the marching cubes algorithm is to make meshes. These algorithms suit various different purposes, and the efficiency of our written programs may not agree with the full potential they have to offer, causing the data we retrieve and compare to be subjective. Overall our goal is to see how these improve content generation as a whole so we will include these differences in our research.

IV. Tasks

Our tasks are to continue to refine our code to collect comparison points for the research. Each member wrote one of our implemented algorithms and will work to see how parallelization improves runtime, the complexity of generation, etc. We have these producing output images respectively for further data points of comparison.

V. Goals

Every week we set ourselves a goal. Moving forward, we each aim to write one page or so on the findings we specifically discover with our respective code. We also have a goal to conclude what the benefits and drawbacks of each content parallelization algorithm that we are working with.

VI. Abstract

Implementation

- Rachel - Perlin Noise
- Hunter- Worley Noise
- Cay - Marching Cubes
- Ben - Researching Parallelization and comparability, Project Manager
- Ryan - Collapsable Wave Functions

Each member of our team worked to implement an algorithm under the guidance and help of Ben, our project manager. We implemented these algorithms without parallelization first so that a comparison can later be made.

One challenge is that we produce output images based on the data the algorithms create. For Perlin noise this could simply be converted to a pixelated grayscale image. For Worley noise each sheet png was stitched together to create a stream of images or a nice video.

Since the marching cubes algorithm generates a mesh, OpenGL was used in order to have a visualization. Cay had some difficulty with this but ultimately overcame and was able to get it working.

It's worth noting some members were not very familiar with Java so deciding to use it meant they had to learn a little and get the environment set up. This was a bit confusing as Java packages always seem to cause a kerfuffle but we managed to all get setup.

VII. Technique

1) Perlin Noise: Parallelizing Perlin noise generation really helps for large datasets. The generation is a computationally intensive task, as it involves the calculation of noise values at each point in the grid. Parallelization can be achieved by dividing the grid into smaller chunks and assigning each chunk to a separate processing unit, such as a thread or a computing core. Each unit can then independently calculate the noise values for its assigned region, allowing for concurrent execution. This parallel approach can significantly reduce the overall computation time, especially on multi-core processors, by taking advantage of the available parallel processing power.

Additionally, parallelization of Perlin noise for three-dimensional or higher-dimensional spaces, where the computational load increases substantially, is really useful. By distributing the workload among multiple processors, the overall time required to generate the noise for the entire space is reduced. However, it's important to note that parallelization introduces challenges such as managing multiple threads and accessing data, which need to be carefully addressed to ensure correct and efficient execution. Clearly, this optimization will

help overall with generation of grid content such as terrain of mass size in quicker time.

2) Worley Noise: Parallelizing Worley noise generation, also known as cellular noise, is pretty similar to perlin noise. In Worley noise, each point in the grid corresponds to a cell, and the computation involves determining the distance to the nearest and second-nearest neighboring cells for every grid point. This process can be parallelized effectively by dividing the grid into segments and assigning each segment to a parallel processing unit. Each unit can independently calculate the Worley noise values for its assigned region, resulting in concurrent computation.

Parallelization becomes even more critical in scenarios where real-time or interactive applications, such as procedural terrain generation or texture synthesis, require Worley noise. By leveraging parallel processing capabilities, such as multi-core CPUs or GPU shaders, the generation of Worley noise can be significantly accelerated, contributing to smoother and more responsive user experiences. Synchronization and load balancing are key here to ensure the execution of Worley noise generation algorithms across multiple processing units.

3) Marching Cubes: The Marching Cubes algorithm generates a mesh based on a grid of voxels, or cubes. Each cube has eight points whose values are generated using one of the aforementioned noise generators or a scalar field. The algorithm "marches" through the grid cube by cube; for each cube, it determines a configuration of triangles by consulting a triangulation table based on which points are "above surface level". We chose to render these triangles using OpenGL. This algorithm has a runtime of O(n^3), and thus can take very long at high resolutions ($>$ 100 voxels wide). Parallelization is done partly by the GPU when rendering the mesh; however, parallelization could potentially improve runtimes at higher resolutions. Howabouts this may be done is still in the stages of experimentation; so far, we believe either chunkifying the grid into 9ths, 27ths, etc. and assigning a thread to each, or using shared variables (locks for mutual exclusion and atomic variables for the cube coordinates) would work.

4) Wave Collapse: The Wave Function Collapse is an algorithm for generating images based on existing data, using rules deciphered from input images as the basis to create a new image that follows those rules. The first part is the creation of rules from the original image, with the algorithm going through every pixel and determining the relation to surrounding pixels. In simple models, only the directly connected pixels are taken into account, while in overlapping models rules take into account pixels across the whole image for determining what can be connected to a specific pixel.

Then, the algorithm creates a blank canvas of a size determined by the user, and starts at the top left. The "wave function" exists at this point in a state where that pixel can be any of the colors from the original image, and it will then "collapse" down to one specific color in that pixel. This is typically chosen at random to begin with, among the colors that, by the rules, can be at the edge of the canvas. Then, the wave function algorithm propagates outward to pixels connected to that one, with the wave function collapsing to pixels that are able to be connected to it. This continues until either the canvas is filled, in which case the algorithm succeeds, or it reaches a point where it cannot possibly find a color to use that works within the established rules. At which point the algorithm restarts from that very first pixel, collapsing the wave function down to a different choice and propagating from that new starting point. The wave function algorithm would fail if no image of the size and parameters specified can be created from the established ruleset.

VIII. Evaluation

Evaluation is based on data size of generation, time of generation, and overall complexity of the generated content. Also of note is the difference between the original, unparalleled programs, and their more efficient counterparts. Each algorithm will be weighed on its own merit, and ultimately will be compared against each other in regards to the earlier mentioned criteria.

IX. Discussion

Prior to this project, several members of the group had taken CAP 4720 Computer Graphics, and had taken a passing interest in matters directly or tangentially related to procedural content generation. Parallelization seems particularly suited towards aiding in this process. As research progresses, it will be incredibly interesting to see just how much content generation can be made more efficient and which algorithms tend to benefit more from its aid.

X. Conclusion