Parallel Terrain Generation

By Petros Emmanouilidis and Victor Zayakov

Summary

For our 15-418 Final Project we explored parallelism in terrain generation with Perlin Noise. Specifically, we developed 2 separate data-parallel implementations of Perlin Noise in CUDA and compared their speed under different input conditions. We ran our program on both Nvidia RTX2080 and Nvidia V100 GPUs, on the GHC servers and PSC respectively. Moreover, we implemented a data-parallel Voronoi Noise generator using CUDA to partition Perlin heightmaps into distinct biome regions. Our final biome-divided procedural terrain was visualized in 3D using the open-source software Blender.

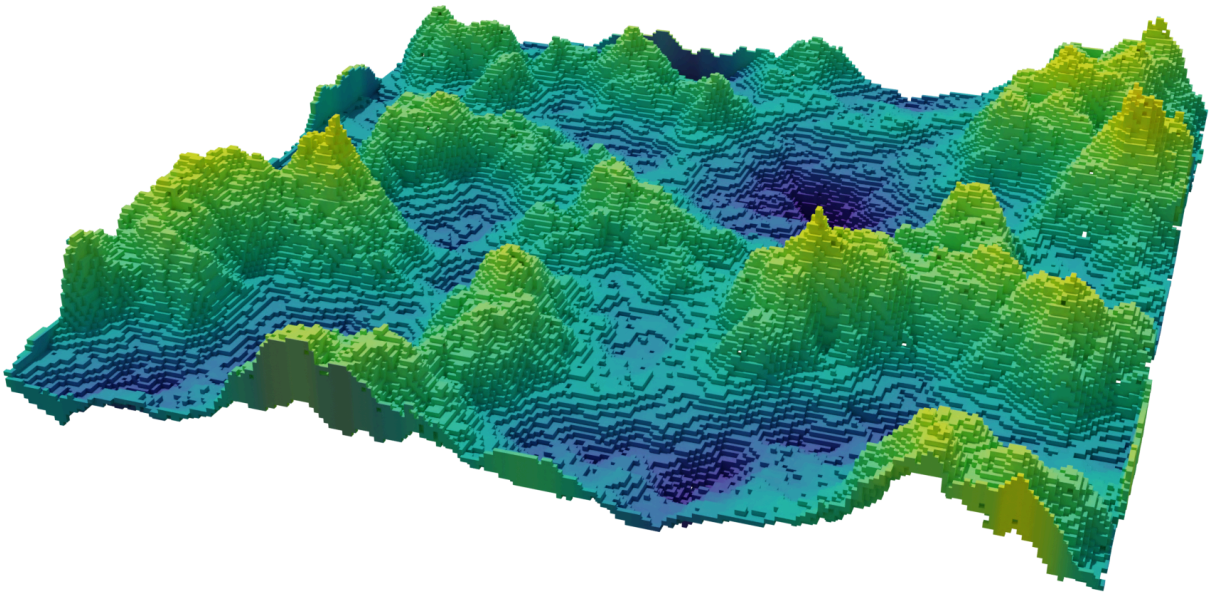


Figure 1: 2D Perlin Noise heightmap generated by us, visualized in 3D in Blender

GitHub Repository: https://github.com/vzayakov/15418-FinalProject

Background

Perlin Noise

Perlin Noise is a popular type of gradient noise first developed in 1983 by computer scientist Ken Perlin. It is often used in procedural terrain generation due to its smooth value transitions and is usually implemented in 2 or 3 dimensions. In 2 dimensions, a Perlin Noise implementation operates on an image of pixels, usually over multiple iterations, and involves the following steps:

1. Divide the map into a square grid and assign a random unit-length gradient vector to each intersection in the grid as seen in Figure 1. Note that each grid square may enclose multiple pixels
2. For each pixel, calculate dot products between the gradient vectors at each of its grid cell's corners and an offset vector from the pixel center to each corner
3. For each pixel, interpolate between the 4 dot products and increment the pixel's value by the result
4. Resize the grid, calculate new gradients, and repeat the previous 3 steps for a fixed number of iterations

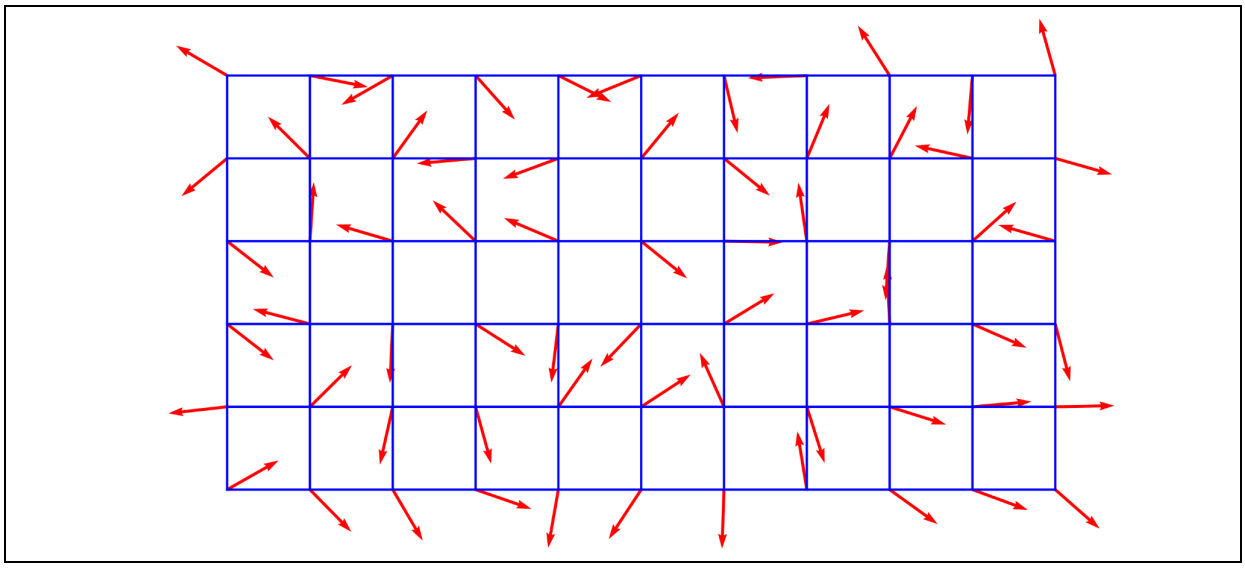


Figure 2: A Perlin grid with random gradient vectors at each intersection ("Perlin noise", 2024)

Generating random gradients in each iteration can be done in many ways. The original paper written by Ken Perlin involves a **global permutation table**: an array of pseudorandom values of size 256 that is used as a seed to produce gradients in each iteration. Generating a gradient vector out of the permutation table involves accessing the array multiple times using the gradient's location and iteration as indices. This is the approach we followed in our own parallel implementations of Perlin Noise.

In that sense, Perlin Noise can be thought of as a function that takes in an image of pixels and returns a noise image of the same size. Moreover, there are typically 4 hyperparameters associated with the algorithm: **Scale**, **Octaves**, **Persistence**, and **Lacunarity**.

Scale refers to the initial size of the grid at the first iteration; as a rule of thumb, smaller grids produce more detailed features. **Octaves** refers to the number of iterations in the

algorithm. **Persistence** describes the amount by which each iteration contributes to the final image. Similarly, **Lacunarity** refers to the amount by which the grid changes shape over each iteration.

For example, a persistence of 0.5 implies that the contribution of each dot product is halved every iteration while a lacunarity of 2 suggests that the grid size is halved every iteration. For best results, it is common practice to decrease the contribution of the dot products and reduce the size of the grid over each iteration: effectively each iteration provides the image with more detailed features that have, however, less of an impact on the overall result.

Perlin Noise can significantly benefit from parallelism in 2 distinct ways. Firstly, the same operations are performed on each pixel in the image, meaning that the performance of the algorithm can be significantly improved by parallelizing across segments of the image (an implementation that works well with a data-parallel approach). Parallelizing over the image can, therefore, allow for the quick generation of large procedural terrain maps which are commonly used as background assets in the entertainment industry.

Secondly, a large number of octaves can place a significant sequential strain on generating Perlin-based terrain. As a result, a data-parallel approach can also be deployed to partition the problem into partial sums for each iteration and a global reduction to generate the final result. Admittedly, this approach introduces an additional challenge of handling synchronization between partial sums and the global reduction. Despite that, many applications of procedural terrain generation require very detailed contours that are only

achievable with large numbers of octaves, motivating a parallel “temporal” approach to Perlin Noise.

For both “spatial” and “temporal” partitionings of the Perlin Noise problem, generating and maintaining consistent random gradients can prove a challenging task within the context of data parallelism. Handling this challenge was a primary focus in our CUDA parallel implementations of Perlin Noise.

Voronoi Noise

Voronoi Noise, also known as Worley Noise or Cellular Noise, is another type of noise texture that we use to partition Perlin heightmaps into biomes. It involves dividing an image into a square grid, generating random coordinates for a point (also known as a seed point) inside each grid square, and finally assigning each pixel a tag corresponding to the closest seed point. The value of each pixel can either be an integer associated with the closest seed or a float corresponding to the pixel’s distance from the closest seed. Both approaches are demonstrated in Figures 2 and 3.

Figure 3: Voronoi Noise with integer seed tags

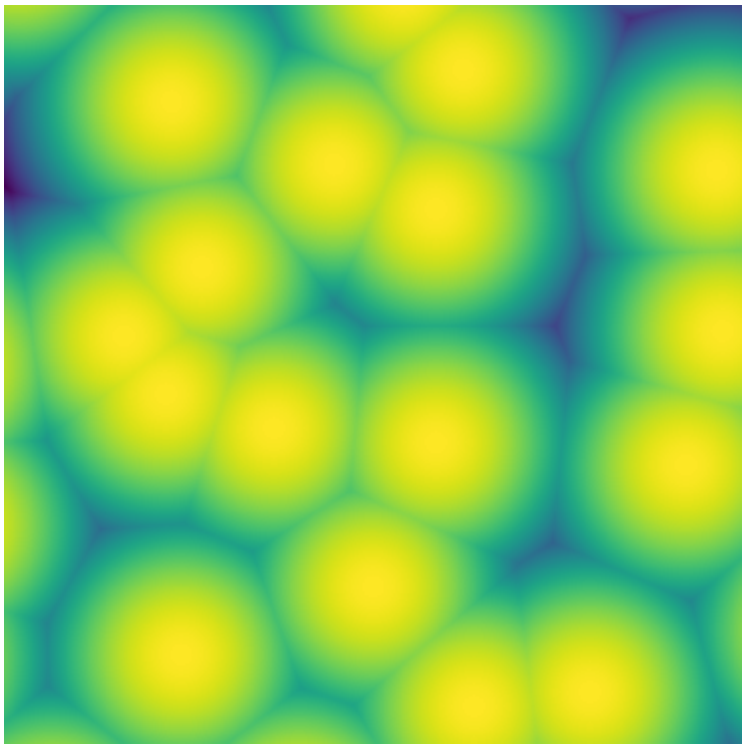


Figure 4: Voronoi Noise with distance from closest seed

Approach

We implemented the Perlin Noise algorithm in CUDA, targeting Nvidia RTX and Volta GPUs. We chose this hardware for two reasons: Perlin Noise is naturally most suited for data-parallel execution, and we wanted to use hardware that is commonly used in the entertainment industry. We wrote two different CUDA implementations of the Perlin Noise algorithm: “spatially partitioned" and “temporally partitioned”, as described below. We also wrote a serial implementation for reference, that computes the noise value at each pixel of the image serially. We did not end up using the serial implementation in any of our experiments since it performed roughly 3000x worse than the parallel implementations. We instead drew comparisons between the two parallel implementations.

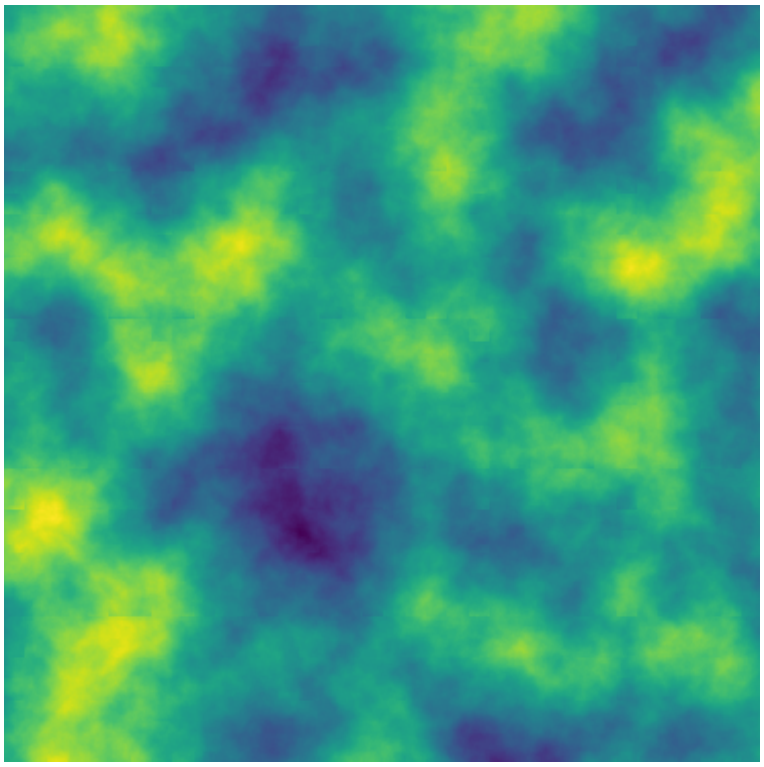


Figure 5: 2D Perlin Noise heightmap generated by us

Spatial Implementation

Our “spatial” implementation of Perlin Noise only parallelizes across pixels of the image and performs all octaves/iterations of the algorithm sequentially within each block. Specifically, we use CUDA blocks to divide the image into contiguous 32x32 square pixel regions. Inside each block, each of the 1024 CUDA threads traces a single pixel throughout all iterations.

A primary challenge in implementing Perlin Noise in CUDA is keeping track of the random gradients in each octave. Since different blocks may enclose pixels that “share” the same gradient, the values of these gradients in each iteration must be consistent across different blocks.

To deal with this problem, we considered different approaches: First, we considered generating and managing pseudorandom gradients in global memory and having each block grab the gradients it encloses in the first (or on each) iteration. This idea was quickly scrapped as to optimize performance we needed to minimize the use of global memory by blocks as much as possible.

Our next approach involved generating a copy of all pseudorandom gradients in the shared memory of each CUDA block and having each block handle updating its copy of those gradients locally. This approach was also scrapped because, for large enough images, we simply could not fit all of the gradients in shared memory.

After some research, we stumbled upon Ken Perlin's original research paper on the algorithm. We noticed that they did not use a uniformly random distribution to generate gradients, as we had originally assumed. Instead, they made use of a 256-element **permutation table**. Effectively, in Perlin's implementation, random gradients are limited to only 4 possible values: 45°, 135°, 225°, and 315°. The idea is to index into the permutation table multiple times using the gradient's grid coordinates and octave and use the resulting value (modulo by 4) to determine that gradient's direction. The pseudocode for this process looks like this:

```
gradient_dir = dirs[ p[ p[ p[octave] + x_grad] + y_grad] % 4]
```

This allowed us to generate any gradient at any iteration within a block as long as we had access to the permutation table. As a result, we had the threads of each block collaborate to copy the permutation table from global memory to shared memory. On each iteration, the block would determine which gradients affect its enclosed pixels (given the dimensions of the gradient grid and the block's position in the image) and would use its threads to collaboratively generate those gradients using the permutation table and store their values in shared memory. Given that each block only enclosed 32x32 pixels, we had enough shared memory to store gradients for all possible grid sizes (with the maximum number of gradients occurring when each grid square encloses only one pixel).

As a result, our final algorithm performed the following steps:

1. Threads collaboratively push the permutation table from global memory to shared memory

2. At the beginning of each iteration, the block determines the number of gradients that affect its enclosed pixels. The threads collaborate to generate the targeted gradients and store them in shared memory
3. Each thread grabs 4 gradients from shared memory for its corresponding pixel and locally accumulates its interpolated dot product sum
4. At the end of all iterations, each thread writes its locally accumulated sum to global memory

Temporal Implementation

Our “temporal” implementation of Perlin Noise divides the work by both octaves and regions in the image. It is implemented as a single CUDA kernel, where each block has 1024 threads. We launch the kernel with one block per image region per octave, where each image segment is a square of size 32x32 pixels. Originally, we wanted to have one block per octave that covers the entirety of the output image, but we realized that the 48KB of shared memory per block wouldn’t be enough to fit all of the necessary gradients. While it is possible to load only some of the necessary gradients at a time, such an approach would’ve been difficult to implement, and also significantly slower than our “spatial” implementation. Hence, we opted for a “hybrid temporal” approach, where each block is assigned one 32x32 pixel image segment within one octave.

Each “temporal” CUDA block performs similar operations to those in the “spatial” implementation. Initially, the necessary gradients – that correspond to the block’s image segment and octave – are computed in constant time by repeatedly accessing the

permutation table, as previously described. They are then stored in the block's shared memory, for use throughout the block's execution. The number of gradients is different for different octaves/iterations since the grid size changes, but there is always enough shared memory to fit all of them.

The main difference from the "spatial" implementation is that "temporal" blocks only execute over one octave, and do not accumulate a sum of pixel values in the block's shared memory. Each thread in each block writes its computed pixel value into an array in global memory. Note that there is one value per octave for each pixel in the image. Then, a separate CUDA kernel is launched that reads the values from the array in global memory and performs a sum reduction. Specifically, each thread in this kernel sums up the values of each octave for a particular pixel and writes that sum into the output pixel array (in global memory).

The main advantage of this approach is that it has a finer task granularity since each block gets assigned one image region over a single octave. This may improve dynamic block/task scheduling on the GPU. However, the sum reduction of values in global memory adds a significant overhead, since global memory is quite slow, which may lead to poorer performance for large input sizes. We analyze this tradeoff further in later sections of this paper.

Voronoi Implementation

We also wrote a parallel implementation of Voronoi Noise in CUDA, which we used for generating biome maps. The idea is that the biome map can be applied to the Perlin Noise heightmap to accentuate certain features and add colors to the rendered image, depending on what "biome" a particular pixel is in. We got this idea from the video game Minecraft, where the map is divided into distinct biomes like deserts, grasslands, plains, mountains, etc.

We implemented Voronoi Noise as a single CUDA kernel. The kernel is launched with blocks mapping to 32x32 pixel regions in the image, similar to our Perlin Noise kernel. The image is divided into a square grid, where each grid square contains one "seed point" whose location is chosen at random. We used the cuRAND library to compute seed point coordinates, using the same seeds across blocks to ensure consistent coordinate values. Each block loads the seed point – with all of its coordinates and attributes – of its corresponding grid square in shared memory, as well as the seed points of the 8 neighboring grid squares. Then each thread computes the nearest seed point to each pixel in the image (one thread per pixel), and assigns color and distance attributes to that pixel, as determined by the nearest seed point.

We did not focus on optimizing this kernel or trying different implementations in CUDA. It was originally a "far-reach" goal and not the main focus of our project.

Rendering

As mentioned in the previous section, to repurpose our Perlin Noise images for procedural terrain generation, we interpreted the generated noise images as height maps. That means that the value of each pixel in the image corresponds to the height of the surface level at that point in the generated terrain. Figure 1 demonstrates a 3D visualization of the Perlin height map in Figure 5. Note that peaks in the 3D landscape have a vibrant yellow color while troughs are dark blue, directly mapping to the float values of the 2D image in Fig. 5.

To partition our generated terrain into distinct biomes, we overlaid the Voronoi noise map on top of the Perlin map, each Voronoi cell/region representing a distinct biome in our terrain. Specifically, we developed 7 distinct biomes: mountains, ocean, desert, grasslands, hills, swamp, and mesa. The idea was to further accentuate, scale, dampen, and recolor each point in the Perlin Noise heightmap, depending on the Voronoi biome it belonged to.

For example, mountain biomes were given a height offset and scale to accentuate any hill-like bumps as well as a significant dropoff with distance from the center of the biome to grant the region a characteristic conic shape. Similarly, mesa regions were given a large scale paired with a height cap to give the generated hills a flat top. Using a small Python script, we used the Perlin and Voronoi maps to generate a biome height map and a biome color map. The resulting images are seen in Figures 6 and 7.

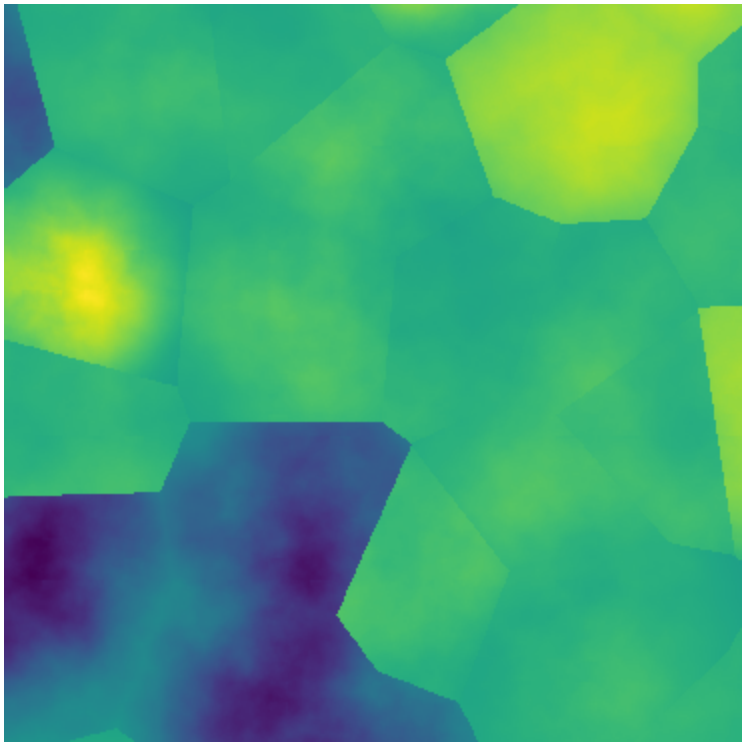


Figure 6: A biome height map with small (blue) and large (yellow) altitudes

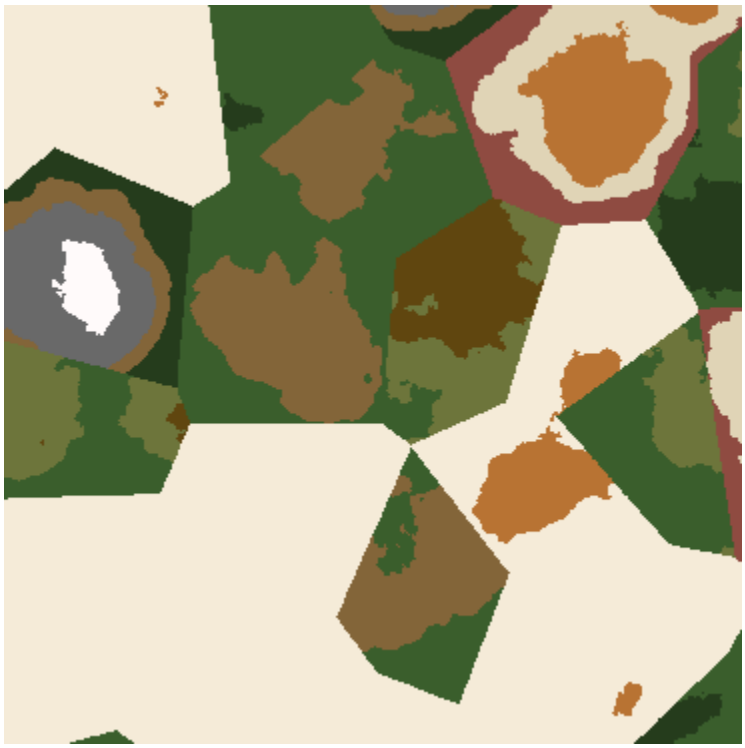


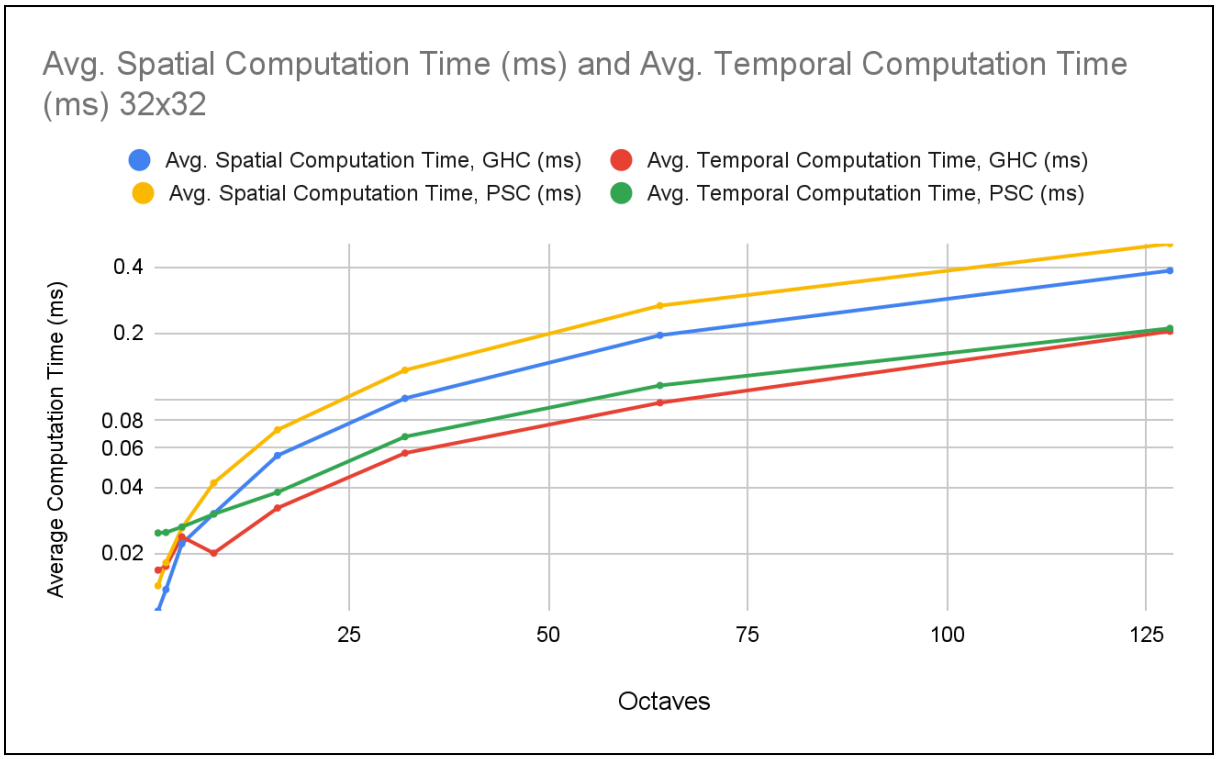
Figure 7: A corresponding color map

To visualize the height and color maps in 3D, we used Blender. In short, we applied the height map image as distortions on a subdivided plane and then overlaid the color map as a texture. To achieve a “voxel look” reminiscent of the game Minecraft, we used Blender’s “Remesh” functionality. Final renders are found in the “Rendered Results” subsection in “Results”.

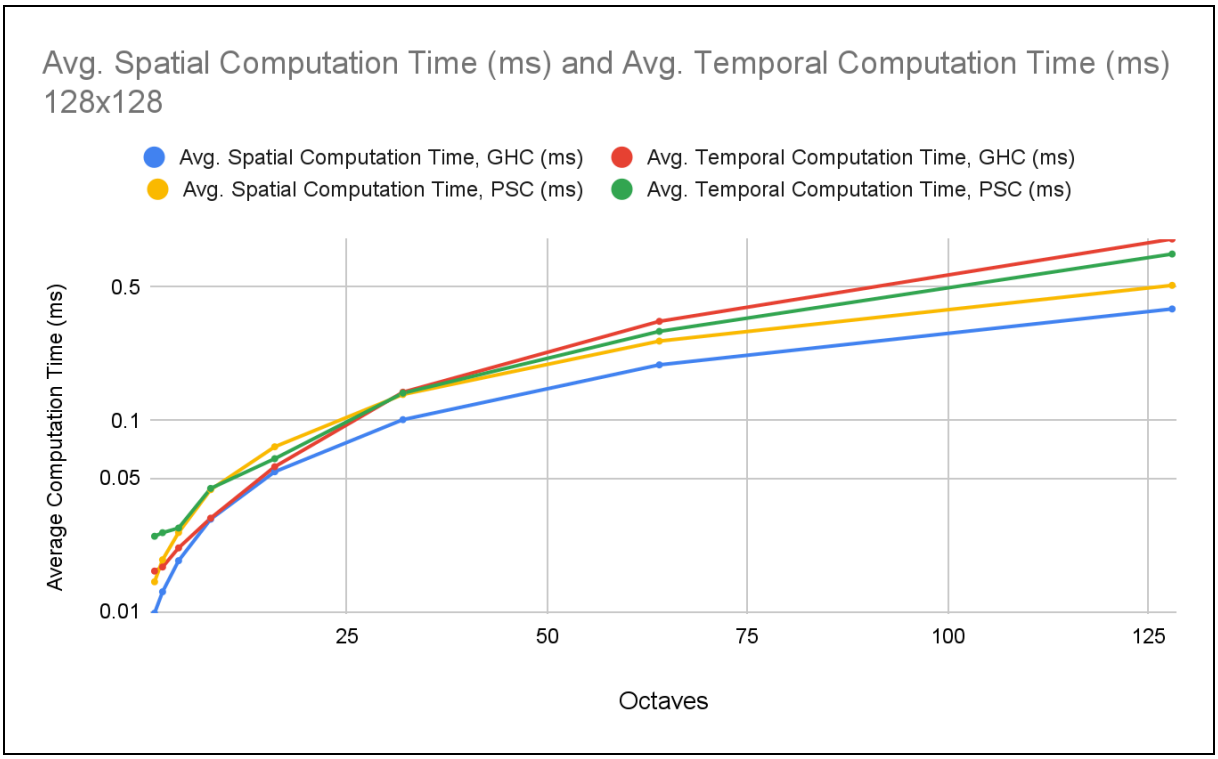
Results

Below we present our experimental results and graphs, as well as the final renders of our generated terrain. We compare the computation times of our two different Perlin Noise implementations – spatial and temporal – while varying the various hyperparameters to determine how they perform under different input conditions.

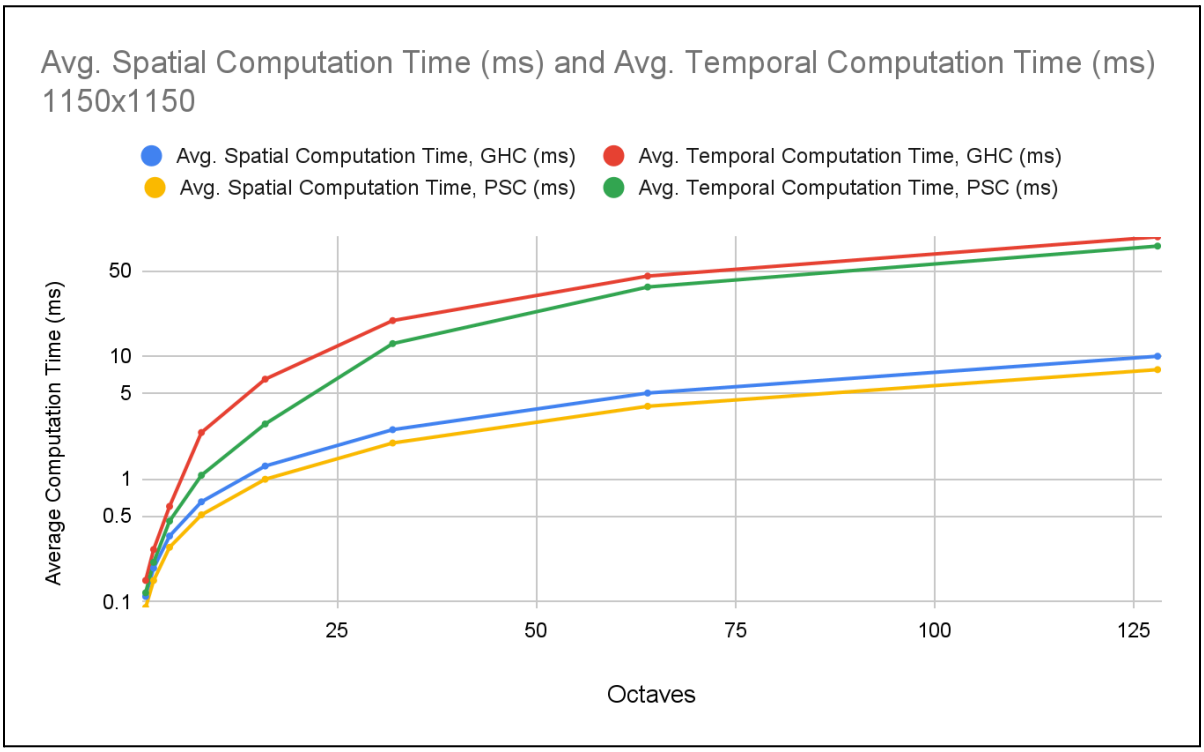
Graphs



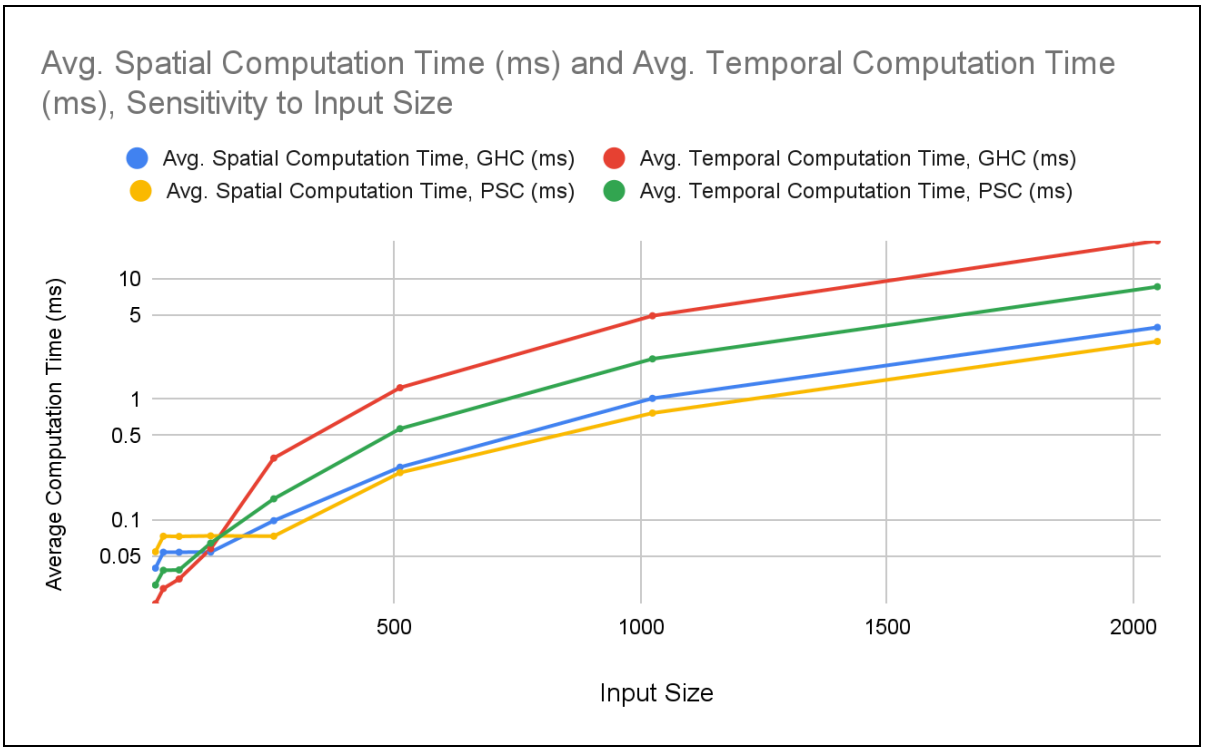
Graph 1: Average Computation Time (ms) vs. Octaves for 32x32 input size, logarithmic scale



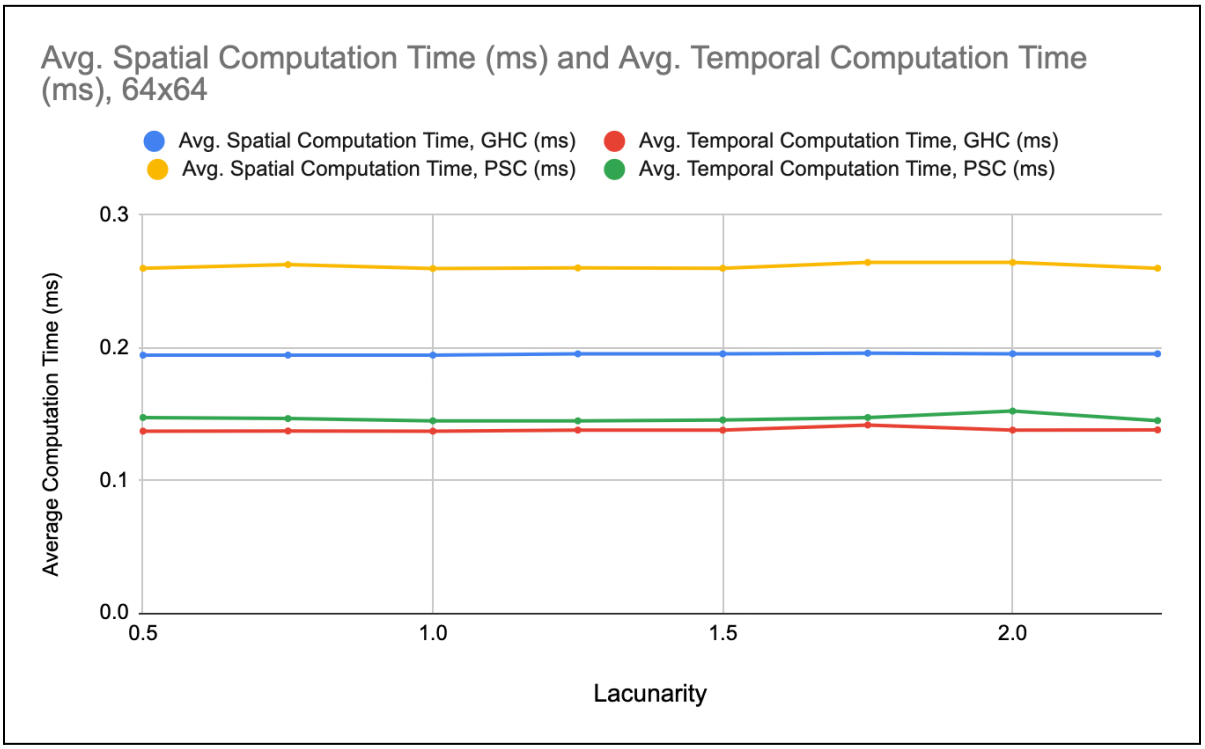
Graph 2: Average Computation Time (ms) vs. Octaves for 128x128 input size, logarithmic scale



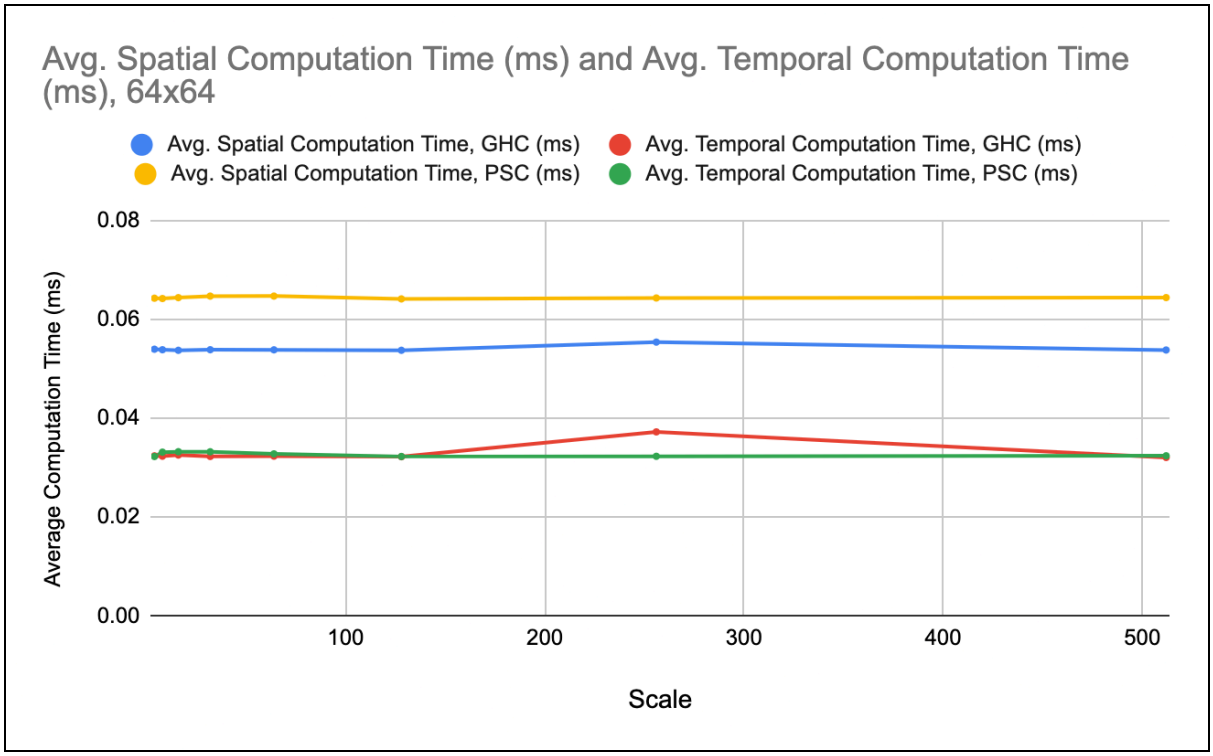
Graph 3: Average Computation Time (ms) vs. Octaves for 1150x1150 input size, logarithmic scale



Graph 4: Average Computation Time (ms) vs. Input Size, logarithmic scale



Graph 5: Average Computation Time (ms) vs. Lacunarity for a 64 x 64 input size



Graph 6: Average Computation Time (ms) vs. Scale for a 64 x 64 input size

Algorithm Comparison

We compiled our spatial and temporal code into an executable binary which takes in various command line arguments, including height, width, scale, octaves, persistence, and lacunarity. We executed our spatial and temporal code on two different GPUs: an Nvidia RTX2080 on the GHC machines, and an Nvidia V100 32GB on the PSC GPU nodes. We evaluated their performance by varying the four different hyperparameters (octaves, scale, persistence, lacunarity) as well as the input size and observed tradeoffs in the computation times of the two implementations. We also wrote a serial implementation of Perlin Noise for reference, but we chose not to use it for our analysis because it was more than 3000x slower than the parallel implementation. We compared the computation times of the two parallel implementations instead. We believe we made the right choice by implementing

the algorithm on a GPU since Perlin Noise is most suitable for data-parallel computation because the same steps are applied for each pixel.

Varying the number of octaves had the greatest effect on the computation times of the two implementations. We investigated this by performing 3 experiments where we varied the number of octaves from 1 to 128, with input sizes of 32x32, 128x128, and 1150x1150, while keeping scale, persistence, and lacunarity fixed (see Graphs 1, 2, 3). The temporal implementation performed better than the spatial implementation for smaller input sizes and a large amount of octaves (see Graph 1). The spatial implementation performed better than the temporal implementation for large input sizes (see Graph 3). The two performed about equally for an input size of 128x128 (see Graph 2).

The spatial implementation performed better for large input sizes due to its efficient use of shared memory: each block computes the pixel values for all octaves within its region of the image, stores each pixel value in shared memory, adds them together, and only writes to global memory once per-pixel at the very end. By contrast, the temporal implementation has to perform a sum reduction of the pixel values for each octave in global memory, which is inefficient for large input sizes. The temporal implementation performed better than the sequential one for small input sizes, due to its finer task granularity, since it launches one block for each octave and spatial region of the image. The global memory sum reduction is not a bottleneck for small input sizes, and the finer task granularity allows for more efficient scheduling on the GPU. The benefit of finer task granularity was more pronounced

on the V100 GPUs on the PSC machines, since those have almost double the amount of SMs compared to the RTX2080s, and can therefore execute more blocks at a time.

We also ran an experiment where we varied the input size from 16x16 to 2048x2048 pixels while keeping the other hyperparameters fixed (see Graph 4). We observed quadratic growth relative to the image side length, which makes sense given that the image is two-dimensional. We also observed that the temporal implementation outperforms spatial for small input sizes, while spatial outperforms temporal for large input sizes, similar to previous experiments.

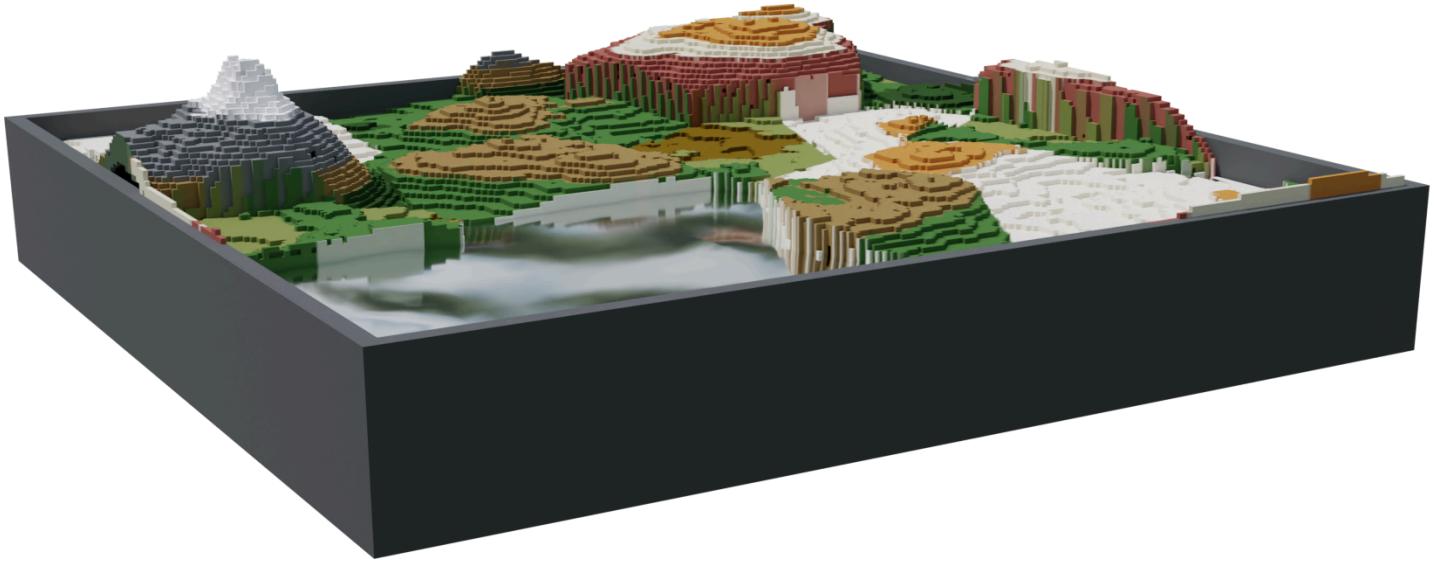
Finally, we ran experiments where we varied the lacunarity (see Graph 5) and scale (see Graph 6) with an input size of 64 x 64 pixels while keeping the other hyperparameters fixed. Neither varying the lacunarity nor varying the scale led to any noticeable changes in the computation times of the spatial or temporal algorithms. This is likely due to the way our code stores gradients in shared memory. Each block uses up to 1024 threads to load the gradients into shared memory at the beginning of its execution, and we have capped the grid size to be 3 pixels wide at minimum. Hence, regardless of lacunarity or scale values, each block will load the gradients in a single instruction or loop iteration.

We did not observe persistence affecting the computation time, since persistence does not alter the flow of the algorithm in any way. There was no reason to investigate this dependence further, so we haven't included any graphs related to that.

Overall, the computation times of the two implementations were strongly dependent on the number of octaves and the input size and were not dependent on the lacunarity, scale, or persistence. The temporal implementation performed better for small input sizes and a large number of octaves, and is suitable for scenarios where small chunks of highly-detailed terrain are being generated. One such application is the video game Minecraft, where highly detailed chunks of 16x16 blocks are generated at a time. By contrast, the spatial implementation performed better for large input sizes, and is suitable for scenarios where large chunks of terrain need to be generated. One such application is in video production and special effects, where large chunks of procedurally generated terrain need to be generated for movie scenes. Ultimately, the two implementations are suited for different input conditions, and both of them have potential real-life applications. Both implementations achieved excellent speedup relative to the sequential implementation. Admittedly, some optimizations could be made, particularly regarding the use of global memory in the temporal implementation. Figuring out how to further minimize the amount of reads/writes to global memory would improve the computation time even further.

Rendered Results

Below are the rendered results of our terrain generation code, rendered using Blender.



Distribution of Work

The workload was split 50%-50% between the two of us. Both of us were present during any work session, and we split the work equally amongst ourselves. Here's a general breakup of who did what:

- Petros: Wrote most of the Blender rendering scripts
- Victor: Wrote most of the Voronoi Noise CUDA kernel
- Worked together on: Spatial and Temporal Perlin Noise CUDA kernels, supporting code for the CUDA implementation (main.cpp, header files, etc)

References

Himite, B. (2021, November 20). *Replicating minecraft world generation in python*. Medium. https://towardsdatascience.com/replicating-minecraft-world-generation-in-python-1b491bc9b9a4

Perlin, K. (2002a). Improved Noise reference implementation. https://mrl.cs.nyu.edu/~perlin/noise/

Perlin, K. (2002b). Improving Noise. *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*. https://doi.org/10.1145/566570.566636

Quilez, I. (2019). *fBM – 2019*. Inigo Quilez. https://iquilezles.org/articles/fbm/

Wikipedia contributors. (2024, November 18). Perlin noise. In *Wikipedia, The Free Encyclopedia*. Retrieved 20:52, December 16, 2024, from https://en.wikipedia.org/w/index.php?title=Perlin_noise&oldid=1258165795

Wikipedia contributors. (2024, June 2). Worley noise. In *Wikipedia, The Free Encyclopedia*. Retrieved 01:53, December 16, 2024, from https://en.wikipedia.org/w/index.php?title=Worley_noise&oldid=1226912510

Zipped. (2023, July 29). *C++: Perlin Noise Tutorial* [Video]. YouTube. https://www.youtube.com/watch?v=kCIaHqb60Cw