MERLIN'S MULTICORE MAPS

Parallelized World Generation

15-418 Spring 2020

Oscar Dadfar (odadfar)

Shuby Deshpande (shubhand)

GitHub: https://github.com/bhaprayan/m3

1. Summary

Our algorithm parallelizes O'Leary's map generation algorithm [1] originally implemented in JS. We ported both the generation and rendering code over to C++ and were able to get speedups using both SIMD and OpenMP parallel execution, as well as GPU hardware acceleration on the render side. We test the performance of our map generation algorithm on a fixed-size map as we increase threads, and also demonstrate that we are able to generate much larger maps than the sequential version in the same time.

2. Background

2.1 Data Structure

We use a custom map data-structure that records all of the voxels of our discretized maps, as well as the heightmap of each voxel. For each voxel, we record its centroid as a 2D coordinate, as well as a list of the 2D coordinates of all vertices around the voxel. These vertex locations are useful during the rendering stage when we rasterize each voxel, while the centroid locations are useful for computing the height of each voxel during the generation stage.

```
struct point_t {
  float x;
  float y;
};
```

```
struct edge_t {
  point_t v1;
  point_t v2;
};
```

```
typedef vector<edge_t> edges_t;
```

```
struct map_t {
  point_t *vcenter;     // size |v|
  edges_t *edges;       // size |v| * deg(v)
  float *heightmap;     // size |v|
  int nVoxels;
  int width;
  int height;
};
```

2.2 Operation Sequence

The key operations we perform on this data structure are Voronoi, Parsing, Slope, Mountain, Normalize, Mean, and Render. Their operations are described as follows:

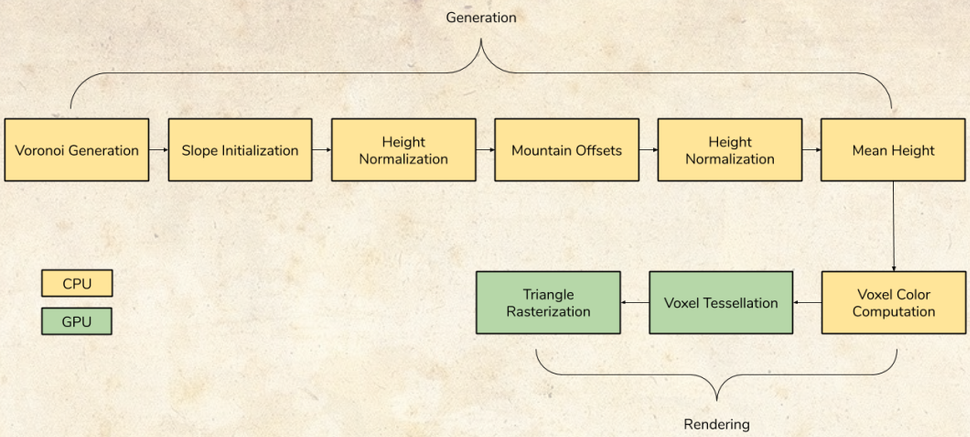

Figure 1: Block diagram of Generation and Rendering stages. Arrows show a linear set of dependencies in the program. CPU-executed components are in yellow, and GPU components are in green.

- **Voronoi** - Compute a random Voronoi map with N voxels using the Fortune Sweep Library [2].
- **Parsing** - Iterate over the centroids and edges of the returned Voronoi map and store them in the custom map data structure, making sure to clip any voxels outside the screen space. Allocate a zeroed out heightmap of the same size as the number of voxels.
- **Slope** - Generate a random 2D vector representation of a line and compute the dot product of each voxel centroid to the line. Store this distance in the voxel's heightmap.
- **Mountain** - Generate M 2D mountain locations. For each voxel, compute the distance between the voxel and mountain center. The height increase as a result of the mountain is inversely proportional to this distance (voxels closer to the mountain will be influenced more).
- **Normalize** - Retrieve the max and min value in the heightmap list. Subtract the min from each value in the heightmap, and divide by the difference between the max and min.
- **Mean** - Retrieve the mean of the heightmap list.
- **Render** - Compute the color for each voxel as a function of its height. Before rendering each voxel, tessellate the polygon using the EarCut algorithm [5], and render each triangle using Raylib [4] with the color computed from the voxel's height.

Figure 1 shows the sequential ordering of these components, and whether they are executed on the CPU or GPU. We specifically execute two Height Normalization commands since we want to evenly combine the heightmap computed from the Slope Initialization and the heightmap computed from the Mountain Offset without either heightmap dominating. In the appendix, there are multiple cases where having too few mountains will cause the slope's heightmap to dominate, and too many mountains will cause the mountain's heightmap to dominate.

From there, the algorithm has the option to either output a text file of all heightmap and polygon data or render the results directly to an image buffer. The text file is in the form of floats where each line contains the height of the voxel and a list of 2D coordinates for the vertices of the voxel that a separate program can later parse and render.

As input, the algorithm can take in the following arguments:

./mapgen -n {numVoxels} -m {numMountains} -t {numThreads}

The color for each voxel is a function of its centroid height. A voxel is considered a part of land if its height is greater than the mean of the heightmap, and is considered a part of sea otherwise. We use a color gradient where higher patches of land are shaded a lighter green denoting high grasslands and higher patches of the sea are shaded a lighter blue denoting shallower waters.

Specifically, we rescale the heightmap values from the original range $[h_{min}, h_{max}]$ to the range $[0.5, 0.25]$ (for land) and the range $[0.65, 0.25]$ (for sea), which we interpret to be the value component of HSV color scale. Whether a computed cell will be mapped into land or sea, is decided according to the mean value of the global heightmap (the logic of which has been described in an earlier section). The HSV values are then transformed into the RGB color space according to a well defined linear transformation, which is subsequently passed into the render pipeline to fuse with the rasterized polygons and synthesize the final colored map. Note, that the benefit of using a simple threshold and affine-based coloring scheme helps in doing SIMD operations since the same series of operations can be processed on a vector of voxels at once to compute their color.

3. Approach

Our map generation algorithm uses the C++ Fortune Sweep Voronoi generation library [2] to generate the initial voxels in our Voronoi grid. For rendering, we used both Python's Graphics.py library [3] built using Tkinter and C++'s Raylib rendering library [4] which operates on the tesselated voxels which we generate using the EarCut tesselation algorithm [5].

The original map generation algorithm we designed was inspired by a JavaScript implementation by Martin O'Leary [1]. We ended up porting sections of it over to C++ for lower-level access to data generation and parallelization. O'Leary's implementation goes further and creates additional features such as erosion, rivers, and even city names. Given the time constraints, we decided not to include these features in order to spend less time on the code porting stage, and more time on the parallelization design stage. In particular, we recognized there to be no need for land erosion since the discretized nature of the voxel cells made a very rigid, eroded-looking boundary between water and land cells. Of the features we implemented, such as Slope, Cone, Mountain, and Normalize, we did not include Cone in our execution since the performance of Cone did not provide any visually significant modifications to the heightmap. Removing the function ended up saving execution time.

3.1 Machine

Our target machine for benchmarking the generation stage was the 8-core GHC Cluster Machines. Since we were unable to install and correctly link Raylib on the GHC machines, and since X-11 forwarding would slow down the rendering process, we ended up benchmarking the rendering stage using a local 4-core Linux workstation due to easier configuration capabilities while setting up software dependencies for rendering.

3.2 Parallelization Strategies

Our code requires iterating over array elements when parsing and computing the heightmap for each voxel. We assign a static even workload to each core using OpenMP, and within each core, we use SIMD execution to process multiple array elements at once given that each element of the heightmap shares the same arithmetic commands. We provide comprehensive benchmarking tools using the additional -I flag when running the program, which logs the execution time of each major component in the block diagram in Figure 1 and helps us narrow down where to focus parallel efforts.

Each component of the CPU-implemented block diagram in Figure 1 had at least one for loop that would iterate over voxel data, and including an OpenMP pragma around these for loops helped to boost performance substantially. Since we were accessing hundreds of thousands of floats each for loop, we wanted to organize the map data structure in such a way that it would maximize cache hits when multithreaded. We had two potential options:

- Store a separate list of heights, voxels, and edges, so that all the heights are cached aligned with themselves, all the centroids are cache aligned with themselves, and all the edges are cache aligned with themselves.
- Create a voxel class that stores the voxel centroid, list of edges, and height so that these entries are cache aligned. Redefine the map data structure to be a list of voxels instead.

By analyzing the access patterns of each for loop, we found that most of the accesses to the map data structure are exclusive to the heightmap (when updating, normalizing, and computing the mean) or the edges (when tesselating and rasterizing). As a result, we went with the first strategy since it promoted cache locality when accessing the entire heightmap array, centroid array, or edges array on its own. Doing this assisted the performance of OpenMP parallelization.

In the original block diagram in Figure 1, we had the opportunity to compute the heightmaps of the Slope Initialization and Mountain Offsets separately in parallel and normalize each before combining the results since there is no sequential dependency between them. Such a strategy would require twice as many heightmap allocations and accesses, as well as an additional join step that would ultimately hurt performance, so we kept the ordering of operations the same and instead parallelized operations within function calls rather than between function calls.

To introduce additional parallelism, we had to rewrite portions of the code in sections where we used random number generation. When we attempted to add OpenMP pragmas to for loops in the Voronoi generation stage that generated random points, we were met with unusually higher execution times. This was because calling C++'s rand() function is not thread-safe, and would serialize all requests for random numbers, thus leading to slower parallelized times in this section than the sequential case. To solve this, we wrote a new rand() function where each thread would have their own random seed and call from their own pseudorandom distribution in order to create thread-safe parallel random number generation.

In the original serial algorithm, we used C++ vectors in place of arrays when we stored voxel and height map information. These vectors came with their own share of automatically parallel applications, such as min/max finding, and using SIMD on any for loop operations. Despite using these parallel operations, switching to pre-allocated lists in the sequential version showed to have a faster execution time. Thus, we changed all instances of vectors to pre-allocated arrays in order to get more control of the parallelism applied to the data rather than relying on C++ vectors. Doing pre-allocation reduced the Parsing stage's time since we only had to allocate once and store values rather than continuously pushing elements back into a growing vector. This also allowed us to parallelize the Parsing segment of code since pushBack operations on vectors are not thread-safe, and devising a lock mechanism for pushBacks would unnecessarily slow down execution compared to simply allocating and storing to a predefined index in a list.

For the rendering stage, we used Raylib, which was a wrapper for OpenGL that made it easy to rasterize thousands of triangles while still getting the benefits of accelerated hardware performance linked with OpenGL on the GPU. Since Raylib did not support the rasterization of polygons, we had to compute the tesselation of each voxel separately, which improved cache locality since the vertices of the voxel that were used for tesselation would be used immediately after by Raylib for rasterization.

3.3 Failed Parallelization Strategies

Originally, our intention was to port O'Leary's JavaScript code line by line into C++ but found after trying to implement Voronoi generation that the original code used D3.JS for generating and iterating upon spatially sparse points to create a Voronoi. Since we did not have access to the same library, we were able to find a different C++ library to help generate our Voronoi Map [2]. This saved us time in the code porting stage so that we could focus more on the parallelization stage.

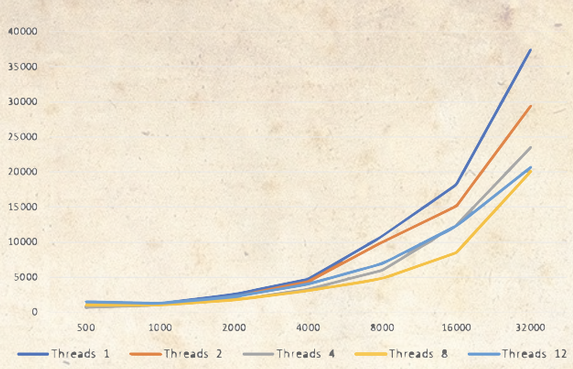

Figure 2: Generation time in microseconds [Vertical] varied over the number of voxels [Horizontal] with the number of mountains fixed to m=25.

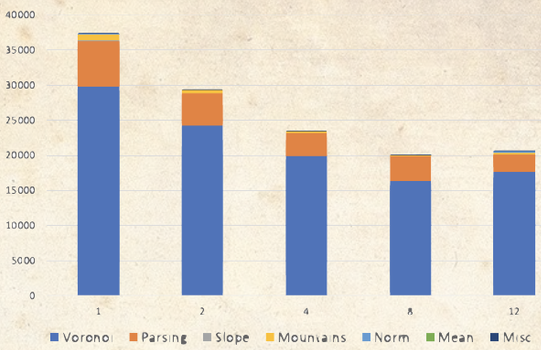

Figure 3: Generation time breakdown in microseconds [Vertical] per task for number of mountains m=25 and number of voxels n=32000 with varying number of threads [Horizontal]

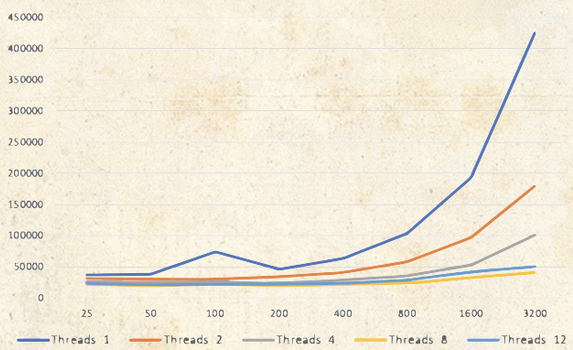

Figure 4: Generation time in microseconds [Vertical] varied over the number of mountains [Horizontal] with the number of voxels fixed to n=32000.

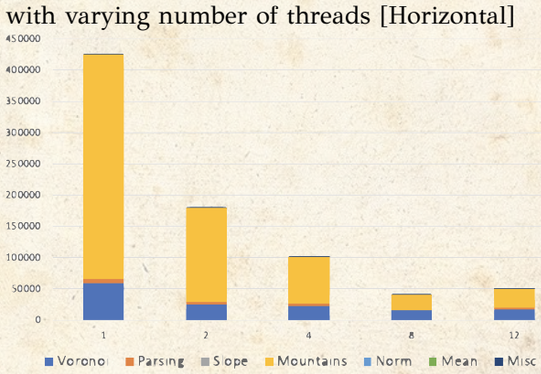

Figure 5: Generation time breakdown in microseconds [Vertical] per task for number of mountains m=3200 and number of voxels n=32000 with varying number of threads [Horizontal].

Another failed attempt was attempting to parallelize the Python Graphics.py visualizer. When trying to use Python's multithreading library and spawn threads to render polygons in an interleaved strategy, we found that Tkinter, what Graphics.py was built on, did not support multiple writes to the canvas, and as a result would crash. We ended up using the C++ Raylib rendering library instead, which had a much higher rasterization time given that we tesselated our own polygons before rendering. This, coupled with the fact that we did not have to export a separate text file to communicate with Python, but rather could keep the data in the same C++ file as we rendered made communication costs drop substantially (printf is an expensive operation when exporting many lines of geometric data).

In the Mountain Offset function, when adding additional height offsets for each mountain to nearby voxels, we use a double for loop and iterate over all the voxels in the outer for loop, and all the mountains generated on the inner for loop. Our initial strategy was to provide two OpenMP pragmas for each for loop to accelerate parallel computation. The issue was that the inner loop would then have multiple threads writing to the same voxel's height. Making the resulting heightmap an atomic structure slowed down performance, and attempting a strategy of pre-allocating a heightmap for each thread and storing the results before coalescing them all together into the main heightmap slowed down performance likely due to how quickly allocations would scale as the number of voxels would scale. Our simpler approach of just parallelizing the outer for loop where we statically assign each thread a section of the voxels to work with performs better since there is no contention for the same heightmap indices for any two threads, while still maintaining the benefit of cache locality in the mountain access pattern.

One other failed attempt was trying to precompute the color of each voxel before tesselating and rasterizing. Since the color computation was the same set of operations for each voxel, we thought of adding a new entry into the map data structure to save a color value for each voxel, and then in one for loop in the render stage, iterate to compute each color for each voxel, and in another for loop,

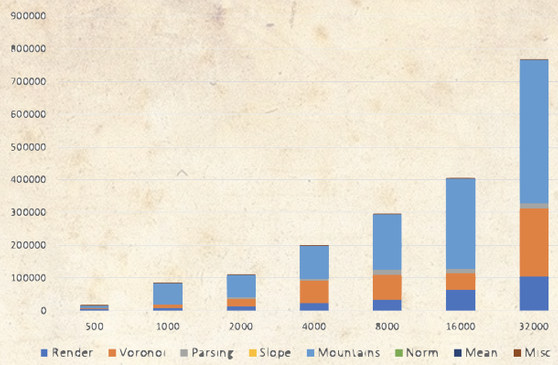

Figure 6: Render + Generation time in microseconds [Vertical] varied over the number of voxels [Horizontal] with the number of mountains fixed to n=3200.

iterate to tesselate each voxel and rasterize the resulting triangle with the precomputed color. We planned on doing SIMD paired with an OMP static partitioning on the color precomputation for-loop for a balanced workload per thread. Yet, doing this hurt performance, and we assume it is attributed to cache misalignment by storing a large array of color values in the map data structure.

4. Results

We define performance improvement in terms of speedup compared to the single thread execution. When benchmarking, we analyzed speedups in two regions: the voxel/heightmap generation stage, and the generation stage with rendering. In our benchmarking, we varied both the number of voxels generated when fixing the number of mountains, as well as the number of mountains while fixing the number of voxels over multiple threads. We used Chrono's high-resolution clock to measure the execution time of each major component in microseconds and provide a relative speedup graph above.

In Figure 2, we see as much as a 1.87x speedup between Threads 8 and Thread 1 when running with a large number of voxels on the GHC machines. Figure 3 goes more in-depth and provides the specific performance increases of each major execution component of the program when the number of voxels and mountains is fixed to 32000 and 25 respectively. We get peak performance with 8 threads given that the GHC machines have 8 cores, and any additional number of threads will cause unnecessary context switching.

Our biggest bottleneck when parallelizing is the Voronoi computation. In the single-threaded case, computing the Voronoi takes more than 3/4 of the execution time, and the speedup for Voronoi alone is less than 2x in the best case. The reason for this is that Voronoi is an iterative refinement method, and most of the code involved in this section cannot be parallelized due to the sequential dependency of trying to generate sparse points.

In Figure 4, we fix the number of voxels but increase the number of mountains and get a much substantial speedup of 10.51x between Threads 8 and 1. The reason for this superlinear speedup is due to the cache access patterns allowing for more cache hits in the parallel version than in the sequential. Figure 5 breaks down the performance of each major execution region when we fix the number of voxels and mountains are fixed to 32000 and 3200 respectively.

Of the breakdown, we get a much better speedup in the Mountains Offset region than in any other region, which helps significantly given that the Mountains execution takes more than 7/8 the total work in the single-threaded case, but we are able to reduce it down to almost 1/2 the total work in the 8-thread case.

Figure 6 shows the execution breakdown for all render and generation stages. Since we were not able to control the number of threads in the rendering stage, we ended up rendering with the

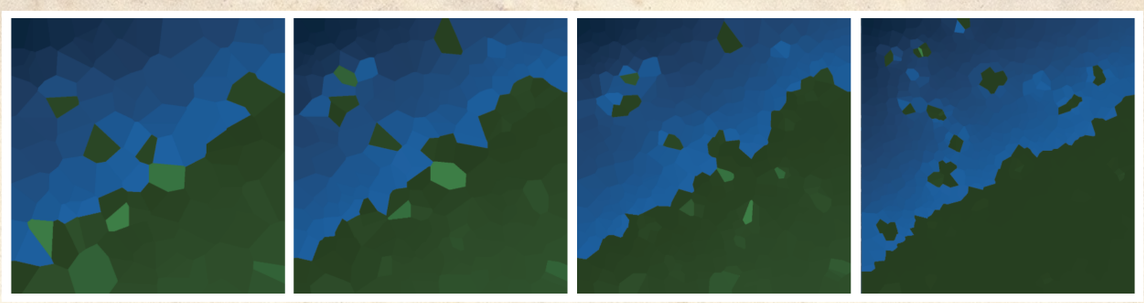

Figure 7: [Left to Right] Fixed number of mountains as voxel number is doubled.

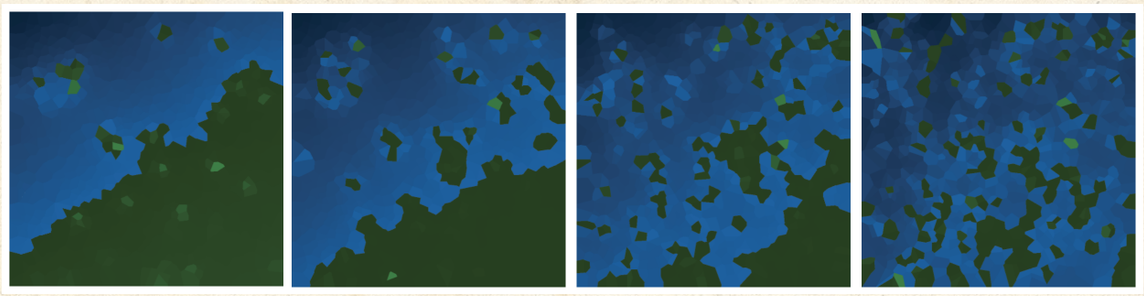

Figure 8: [Left to Right] Fixed number of voxels as mountain number is doubled.

max number of threads and compared execution times as we increased the map size. The time spent rendering is a little more than 1/8 of the execution time in most cases.

In Figures 7 and 8, we provide visual comparisons of the various outputs we get from our program. Figure 7 fixes the number of mountains while increasing the number of voxels to make a higher-resolution image with speed plots reflected in Figure 2. Figure 8 does the opposite and fixes the number of voxels while increasing the number of mountains to make a more eroded land structure with speed plots reflected in Figure 4. Figure 8 best represents the planetary evolution of landmasses.

Please see the GitHub Repo ReadMe for more results and cool visualizations. See the Appendix (last few pages) for additional map screenshots.

5. References

[1] O'Leary, Martin. "Generating Fantasy Maps." mewo2.Com, mewo2.com/notes/terrain/.

[2] Westerdahl, Voronoi, GitHub repository, https://github.com/JCash/voronoi

[3] Zelle, John M. Python Programming. Franklin, Beedle, Associates Inc, 2017.

[4] "RayLib." www.raylib.com/.

[5] Agafonkin, EarCut, GitHub repository, https://github.com/mapbox/earcut

6. Division of Work

Oscar:

- Generation code (slope, cone, mountain, mean, normalize)
- Generation code parallelization
- Python Rendering code writing

Shuby:

- Generation code (voronoi cell computation)
- Raylib Rendering code writing
- Raylib Rendering code parallelization

418 Staff:

- For being awesome and giving us a great semester.

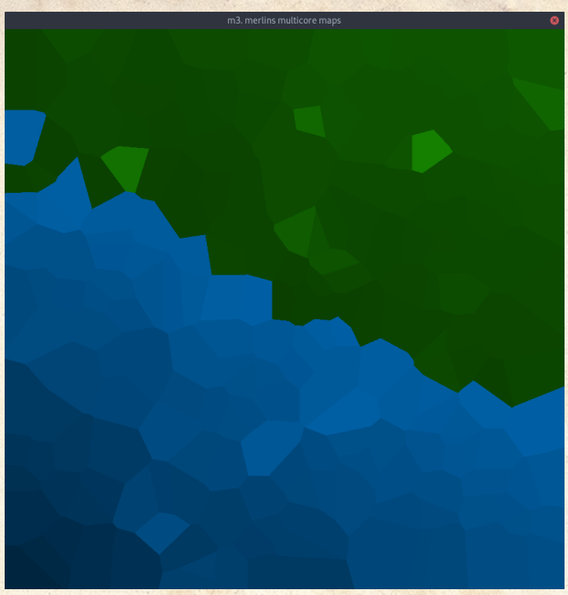

Figure 9: voxels 256, mounts 8

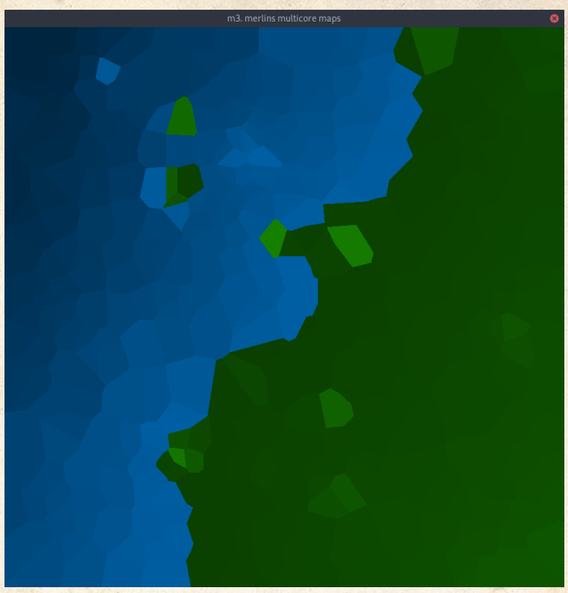

Figure 10: voxels 256, mounts 16

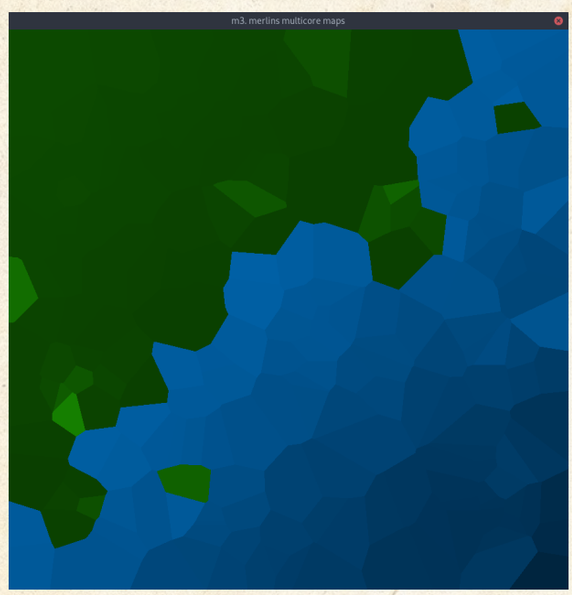

Figure 11: voxels 256, mounts 32

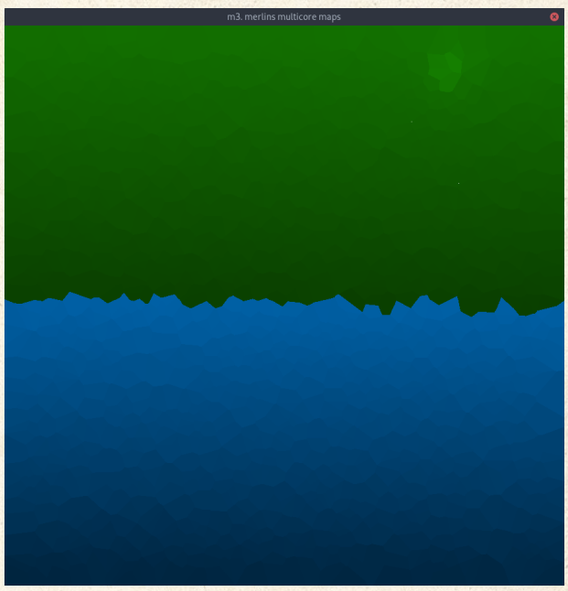

Figure 12: voxels 2048, mounts 1

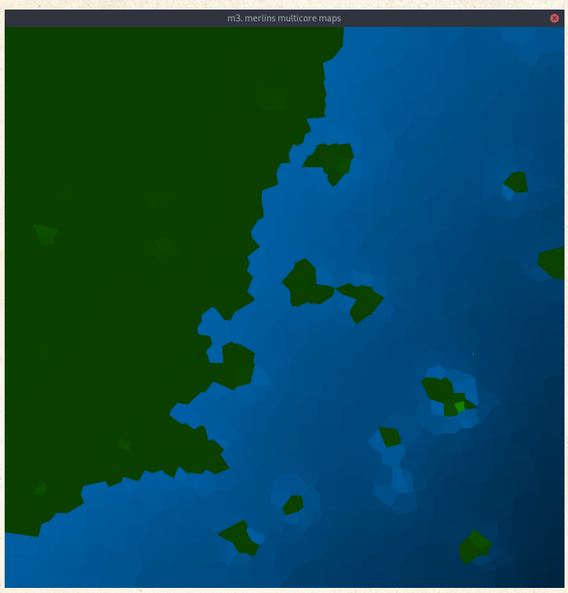

Figure 13: voxels 2048, mounts 32

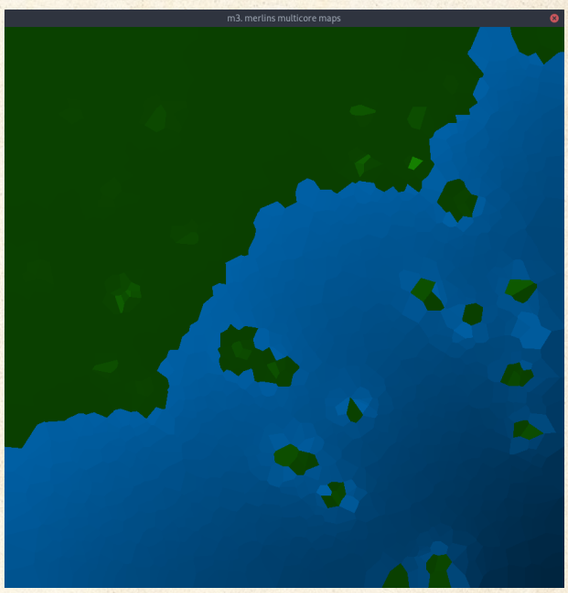

Figure 14: voxels 2048, mounts 32

Appendix(1/1)

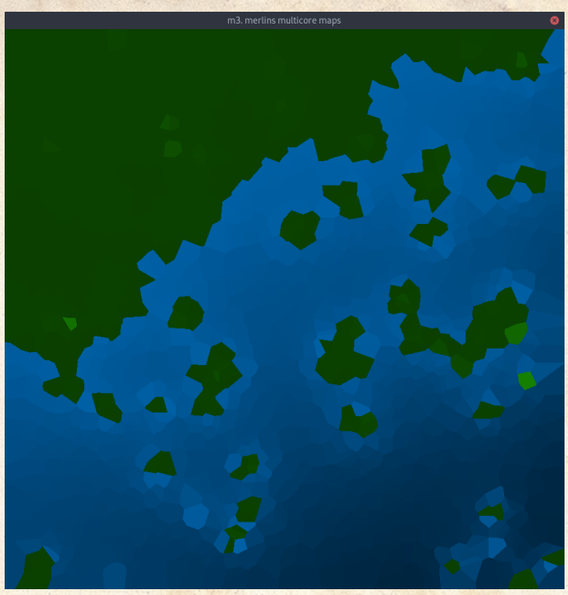

Figure 15: voxels 2048, mounts 64

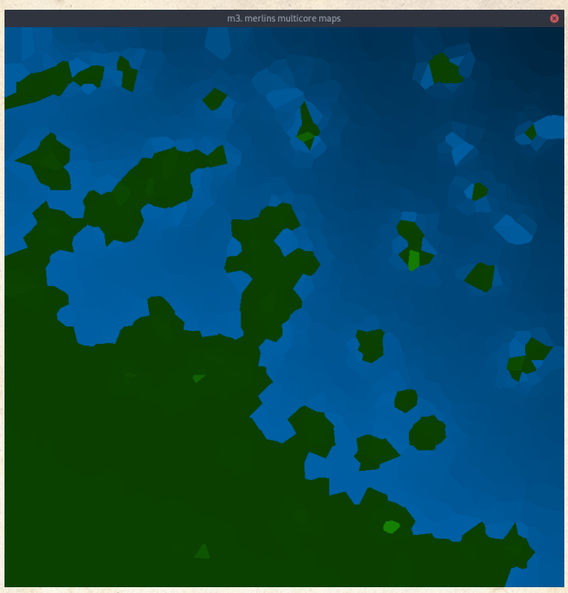

Figure 16: voxels 2048, mounts 128

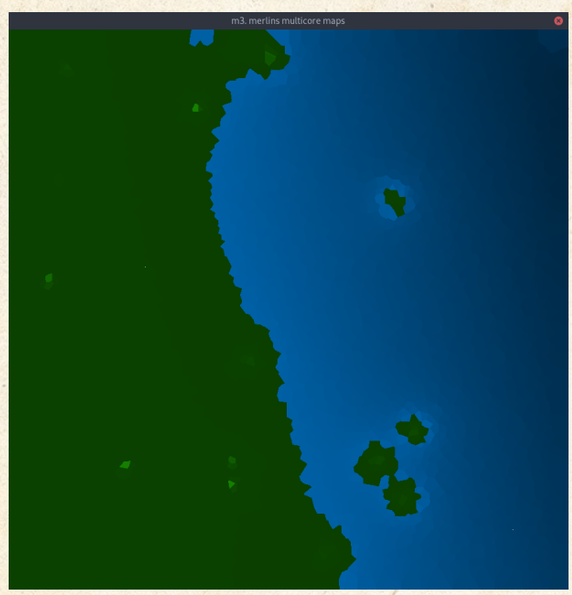

Figure 17: voxels 8192, mounts 16

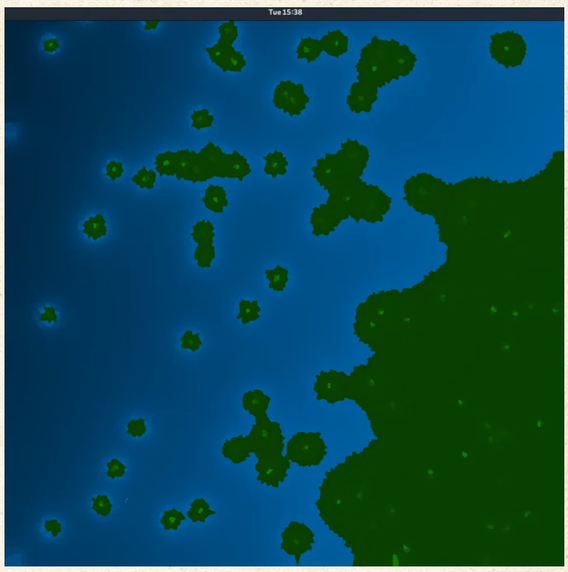

Figure 18: voxels 32000, mounts 1600

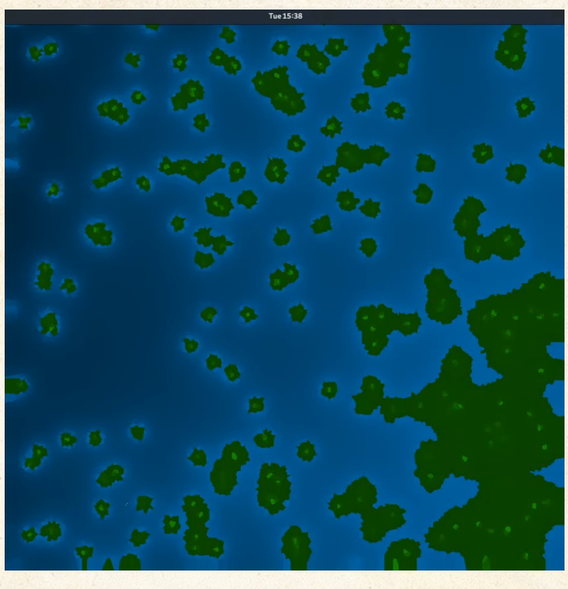

Figure 19: voxels 32000, mounts 3200

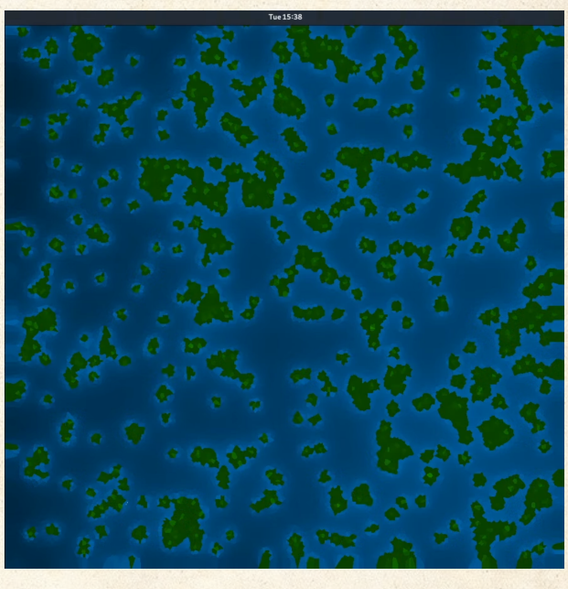

Figure 20: voxels 32000, mounts 6400

Appendix (2/2)