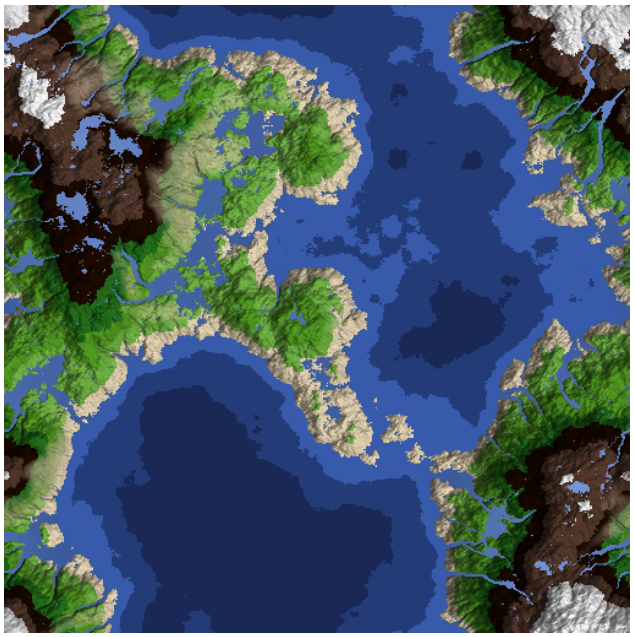
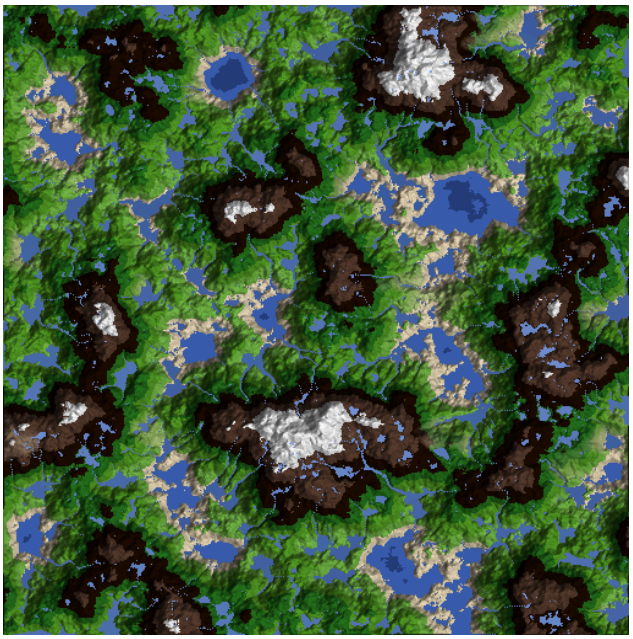
Procedural Terrain Generation

Victor Jiao - CMSC 23800

https://github.com/VjiaoBlack/terrain-gen





In this project, I explore and implement various algorithms for procedural terrain generation, and render my results on a top-down, pixel based map approach.

Topics include (* means not yet implemented):

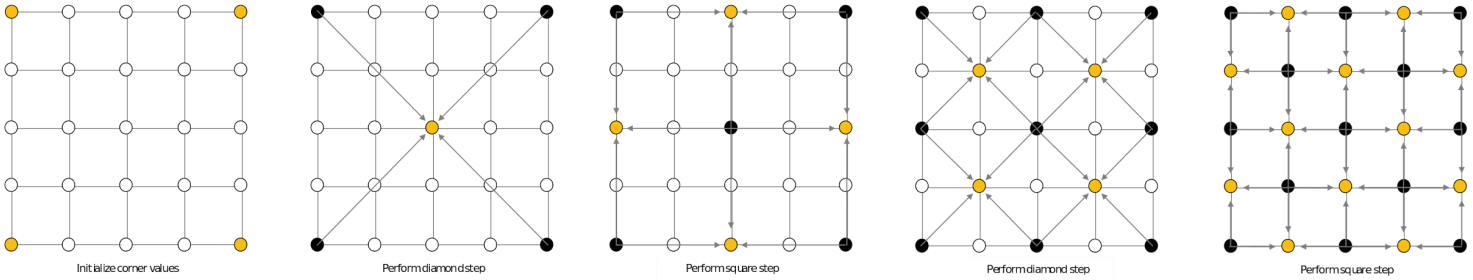
1. Heightmap Generation
    a. Diamond-Square
    b. Perlin Noise
    c. Modifications
2. Basic Biomes
3. Normal Map Generation
4. Water Simulation
    a. Basic
    b. *Graph Search
    c. With Erosion
5. Biome / Humidity Simulation
    a. Wind Calculation
    b. Basic Humidity Biome Calculations

Heightmap Generation

There are several ways to generate heightmap information. In this section, I explore two different common algorithms for randomly generating heightmaps, and techniques to polish them off (so to speak).

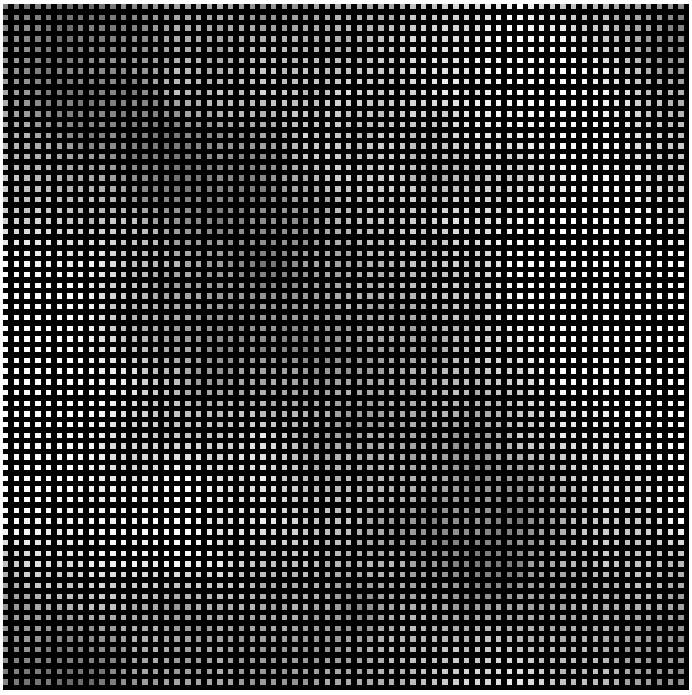
Diamond-Square Algorithm

The Diamond-Square algorithm was a terrain generation algorithm formally introduced at SIGGRAPH in 1982. Here's a picture from Wikipedia explaining how it works:

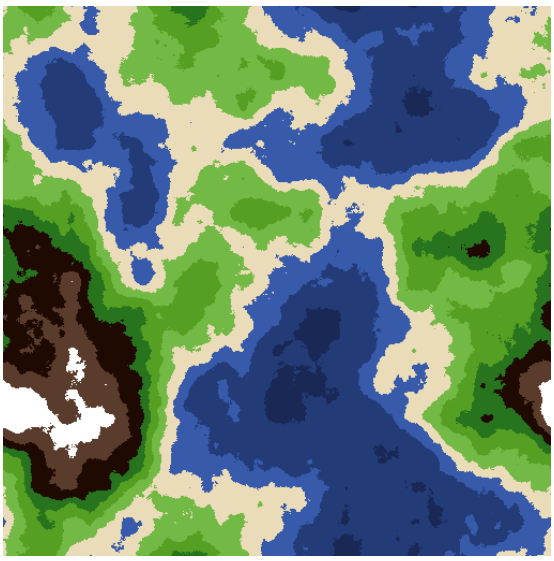


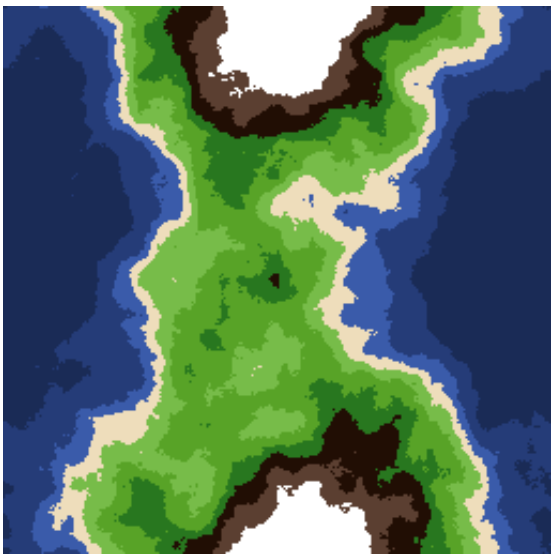
At every step of creating a new pixel value (yellow) from averaging nearby values (black), you add a random value to it (smaller values at the later stages of the algorithm), and thus create perturbations in your terrain.

Here is a picture of half-way done Diamond Square:



Here is are pictures of two Diamond Square terrains:

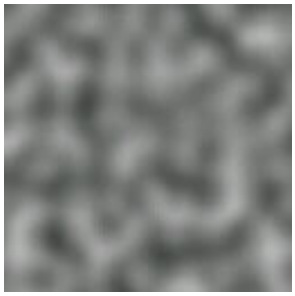




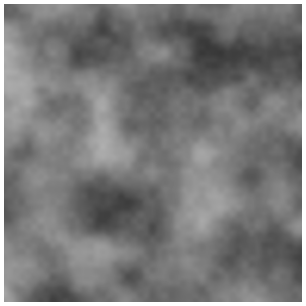
Initially, I had lots of bugs with this. Most of them (i.e. diagonal artifacts, terrain only generating a diamond shaped island centered around (0,0), etc.) turned out to be due to incorrect algorithm implementation. I was able to trivially create wrap-around terrains: instead of initializing an n+1 grid with 4 corner points, i just initialized a n grid at (0,0) and wrapped around.

Perlin Noise

Perlin noise was formally published by Ken Perlin in an 1985 SIGGRAPH paper (it was actually developed for Disney's 1982 movie *Tron*, and Perlin later won an Academy Award for his contribution). He originally intended for use in generating textures, such as wood or stone. One single layer of Perlin noise looks like this →



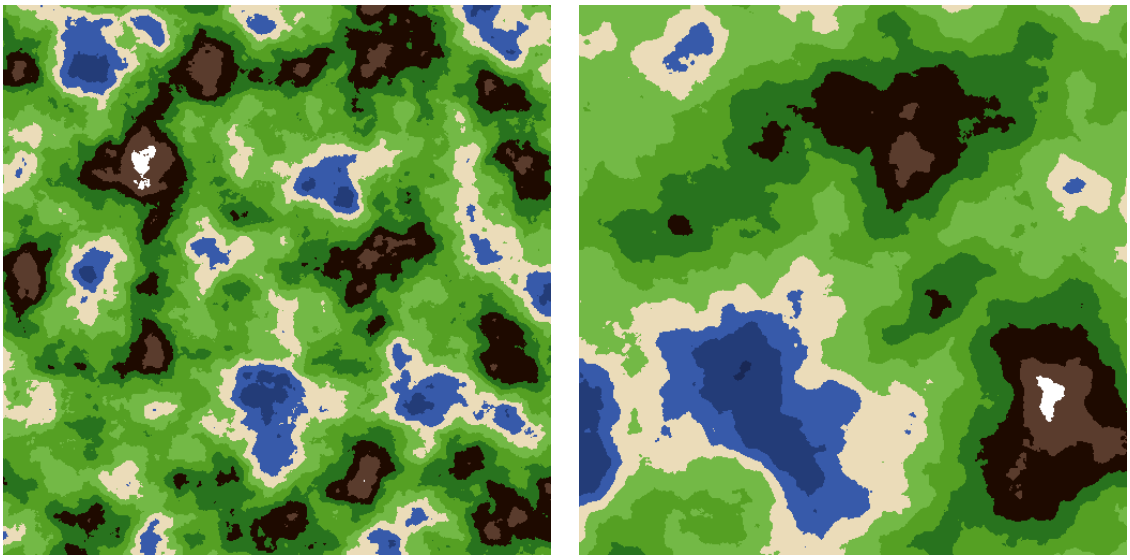
This is nice, but this will generate terrains that are FAR too blurry. We can improve that by adding different levels of noise so that each additional level of noise includes noise of exponentially less amplitude and higher frequency. Then, you get something like this ↓ (both images from Wikipedia.)



The way this works for 2D terrain gen is as follows: fill up a random cube with values (the bigger the cube, the more detailed the final noise). Then, at each point in the heightmap, index into that random cube and interpolate between values at a rate determined by the frequency parameter - i.e. highest-frequency noise would have each heightmap pixel correspond to 1 Perlin pixel, but the base-frequency

noise might correspond 100 heightmap pixels correspond to 1 Perlin pixel.

Examples of Perlin-noise heightmaps (with basic biomes, by altitude) are as follows:

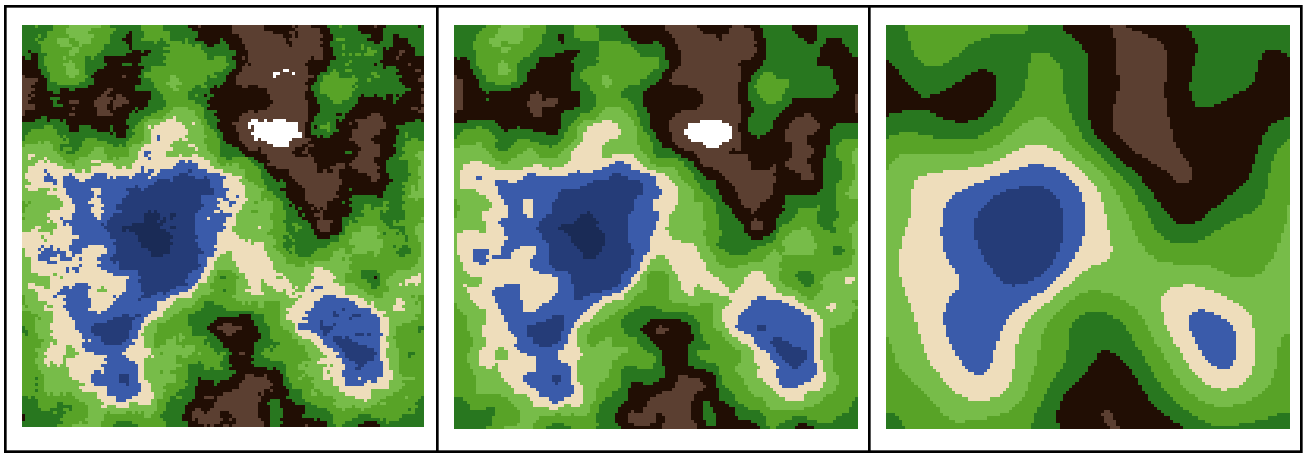


There are better versions of noise than Perlin noise (for one example, simplex noise kinda generates noise based on a hexagonal grid rather than a pixel grid, but not really; it's more complicated) but this is certainly good by its own right!

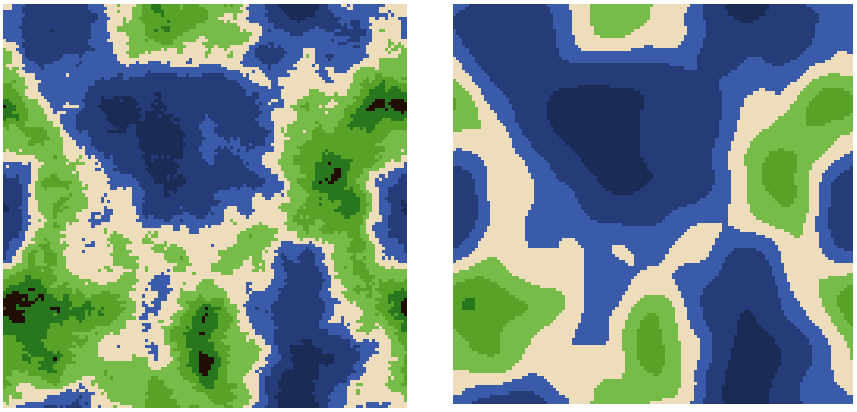
Modifications

Blurring can be useful. A repeated box blur is akin to a gaussian blur. I was able to just repeat a 3x3 blur and it works fine, without me having to write blur values into another buffer, and then copy them over. Normally, if you don't do that, significant artifacts will pop up as you go along the pixel matrix, since you're blurring (or re-using, in any case) new values with values that have already been blurred. In this case, the 3x3 blur is small enough that it doesn't happen very noticeably, but I will not formally prove this.

Pictures of box-blur on a terrain map: 0 iterations, 1 iteration, 30 iterations.



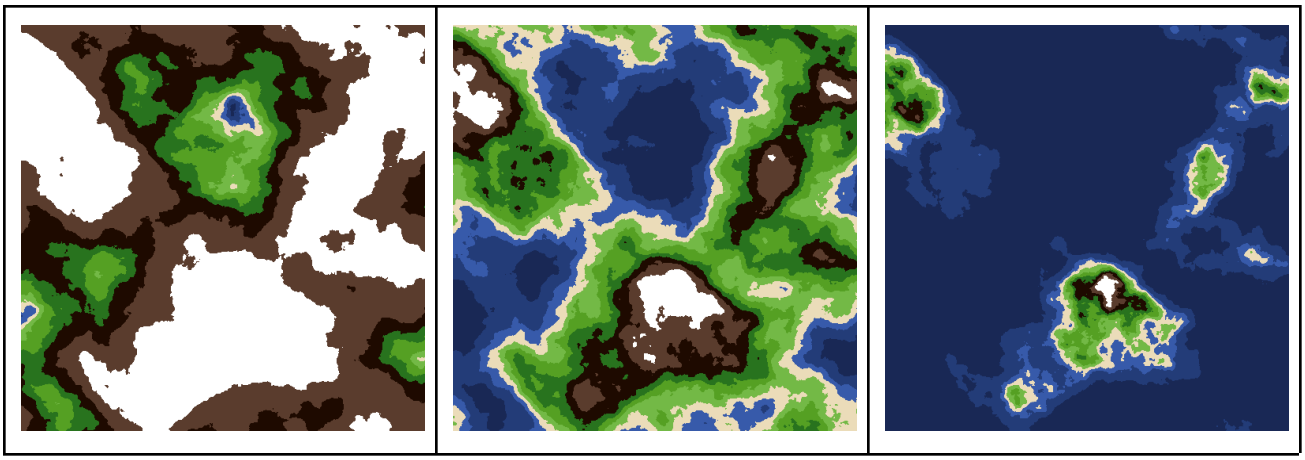
Integer blurring artefact:



You see too many horizontal lines and straight 1/1 diagonal lines. This was because I initially stored height as a uint_8, from 0 to 255. Averaging the 3x3 box of pixels around each pixel was both 1) decreasing the overall average height slightly due to truncation, and 2) limited in the amount of “blurring” if the answer happened to land /just/ on the right side of the rounding, creating these artefacts.

These artefacts thankfully disappeared easily when I switched to using doubles to represent height, and I could remove my code for manually adding in values for truncation correction.

Exponents are nice as well. They can create egg tops, or valleys and stuff.
I’ll just include a 3 by 3 view of a terrain, the same terrain with pow(.33) and pow(3).
(heights are adjusted to match the same “average” height)
Higher exponents create rounded valleys, and lower exponents create rounded hills.



Basic Biomes

You've already seen it, I just set the color of a pixel related to the height.
I'm doing it in an if case with these values. Some water, then sand, trees, rock.

```
int threshold[9] = {70, 100, 117, 132, 148, 165, 180, 198,
218};

uint32_t colors[9] = {0xFF1A2B56,0xFF253C78,0xFF3A5BAA,
                      0xFFEEDDBB,0xFF77BC49,0xFF58A327,
                      0xFF28771F,0xFF210E04,0xFF5B3F31};
```

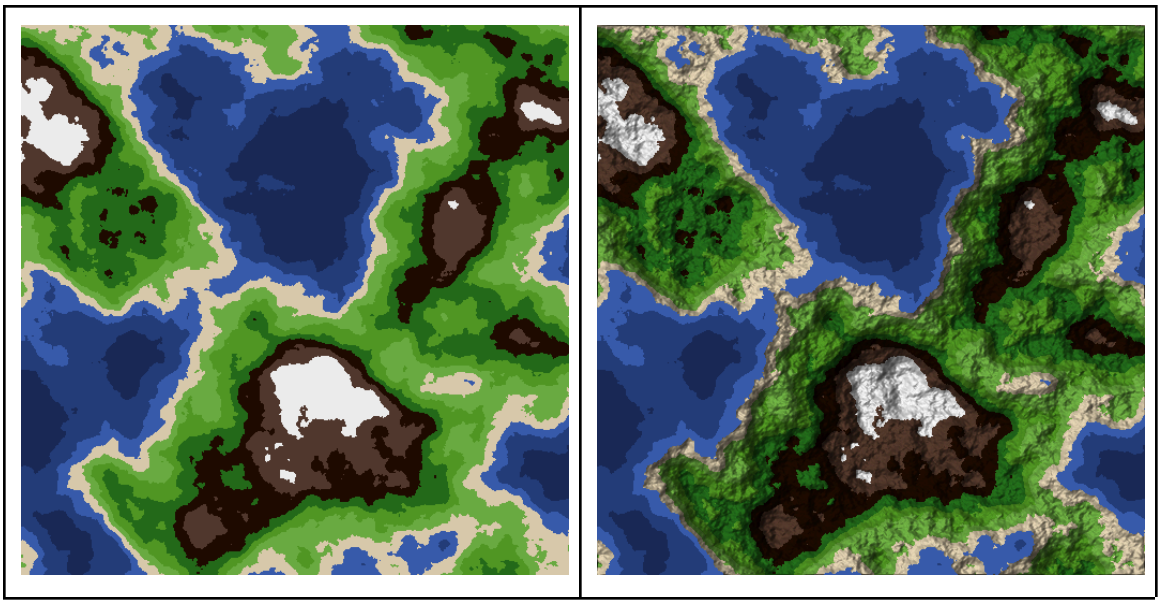
Normal Map

We can create a normal map from our heightmap with some simple math. For each pixel, generate a vector going from the pixel to its left to the pixel to its right, and a vector from the pixel from its top to the pixel to its bottom, as such:

```
vec3 lr(2.0, 0.0, 255.0 * (m_height->at(i,j+1) -
                           m_height->at(i,j-1)));
vec3 ud(0.0, 2.0, 255.0 * (m_height->at(i+1,j) -
                           m_height->at(i-1,j)));
```

We use a clear 1.0 = 1 pixel scale, so it's clear that the `lr` vector spans 2 "x" coordinates (note that vec3.x corresponds to columns, the second index in pixel matrix accesses), and the `ud` vector does similarly.
Then, simply take the cross product and normalize.

```
m_normal->at(i,j) = vec3::cross(lr,ud);
m_normal->at(i,j).normalize();
```



Water simulation

Wow, this was difficult to do right. Lots of bugs. Was it worth it? *shrugs*

Basic Water Physics

I create an additional pixel matrix of doubles (though, note, these are actually implemented as arrays for speed) to hold “water” value. Water value is the depth of water at a pixel; so a pixel with height 0.2 and water 0.4 has its water level at 0.6.

Here’s my code, and explanations:

For every pixel, we have to find which surrounding pixel is the lowest and thus gets water from us, so we create some values to help us do that.

```
double m = m_height->get(i,j) +
           m_water->get(i,j);
int mi = i;
int mj = j;
double mh = m;
```

Then we need to actually look at all surrounding pixels to see which has the lowest water level. I’ve shown you one check for a cardinal direction, and one check for a diagonal direction. Note that the diagonal direction has a artificially increased threshold - this is because diagonal pixels are farther away from directly adjacent pixels. Theoretically, what we want to do is diminish the apparent water level gap between those pixels and

our pixels by sqrt(2) - by multiplying the threshold by sqrt(2), we accomplish the same thing and save a division.

```
// cardinal etc...
if (m_height->get(i+1,j) + m_water->get(i+1,j) < mh) {
    mi = i+1; mj = j;
    mh = m_height->get(i+1,j) + m_water->get(i+1,j); }

// diagonal etc...
if (m - (m_height->get(i+1,j+1) + m_water->get(i+1,j+1)) >
         (m - mh) * 1.4142) {
    mi = i+1; mj = j+1;
    mh = m_height->get(i+1,j+1) + m_water->get(i+1,j+1); }
```

Next, we need to set up some basic checking to make sure our water doesn't do dumb things like move around until there's so little that floating point noise become significant, oscillate rapidly between two points (and then erode the ground infinitely). We do this simply by decreasing the amount of water that moves, depending on the depth of the water. If the water is deeper, less water gets to move, and at most, you can only move the amount of water you currently have.

We can have an additional logic block that makes water move less if it's deeper - this makes our water simulation less realistic, but allows for better erosion later, so that our rivers aren't too deep.

```
// if there's enough water worth moving
if (mh < m - 0.0001) {
    double diff = m-mh;
    diff *= 0.5; // takes care of oscillating water

    // prevents water from cutting too deep in rivers
    diff *= 1.0 - m_water->get(i,j);
    if (diff > m_water->get(i,j)) {
        diff = m_water->get(i,j);
    }

    m_water_temp->at(mi,mj) += diff;
    m_water_temp->at(i,j) -= diff;
}
```

To display this new water, simply modulate the color of the water (bluish) depending on height, and it'll also blend with existing fake height-based water-colors if you tune it.

Image will be shown after erosion!

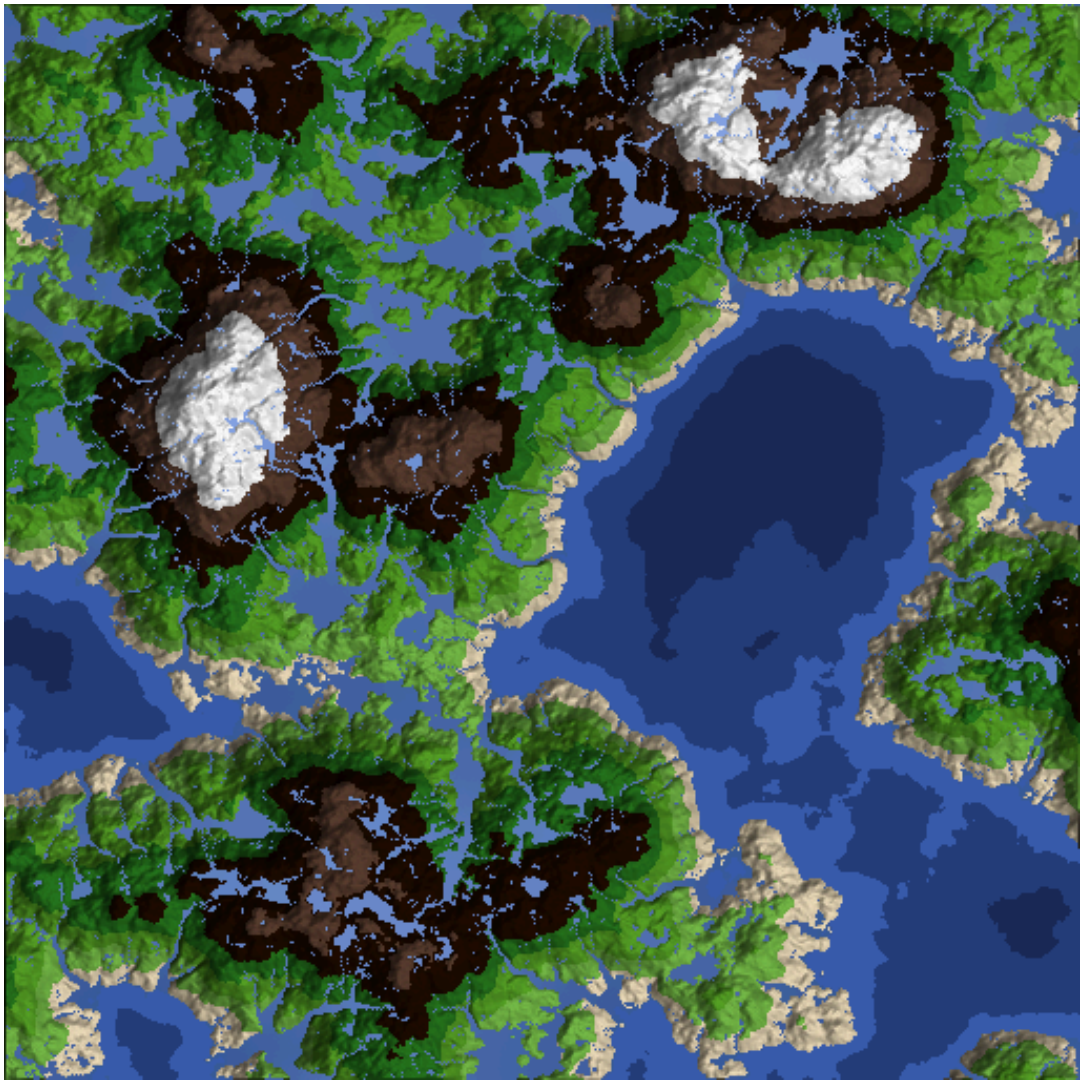
A* Water Physics

Not implemented. Currently, however, you see that the water physics is kinda unrealistic. One big limitation of it is that if we keep using this 3x3 window to check water, we'll always only be able to update water one tiny bit at a time. It'll take forever for oceans to settle down.

In theory, I can run a short A* on each grid point and move half of each pixel's water /through a contiguous body of water/ to reach its lowest point. This might be a better representation of flow, but it'd likely be a bit slower - should be still fast enough to run in , real time, though.

Erosion

We use the mysterious `water_temp` array in the previous section to modulate the heightmap as well. `water_temp` essentially stores the amount we need to change land by - if we decrease this value by a percentage, we'll essentially allow erosion. However, that's not all that we have to do - we also have to modulate the map height of adjacent pixels as well. Again, we use the sqrt(2) trick to make sure we don't over-erode pixels diagonally.



Advanced Biomes

Minecraft sucks, mostly. They generate (or at least they used to generate) biomes based on a random biome random noise generator along with a random moisture map. However, that's not how moisture works. We need to implement weather - or at least wind, and things like that.

Wind

We assume that the natural dominant wind of the terrain is going into the positive x direction (+ j direction), and we use that information along with the normal map to project the wind vector into the plane of the normal map. We end up with the wind vector!

Humidity

The current model of humidity involves running over the wind array, and then increasing values to the left of the current pixel in an “moisture” array based on the current moisture levels of the pixel.

Here is some pseudocode outline of what I did.

Do a separate pass to update ocean water

```
if (m_water->at(i,j) + m_height->at(i,j) < 0.458) {
    m_water->at(i,j) = 0.458 - m_height->at(i,j);
}
```

Do a separate pass to dry air a bit, add moisture if it’s above water

```
m_moisture->at(i,j) *= 0.9; // dry water
m_moisture->at(i,j) += m_water->at(i,j) * 3.0;
       // water makes air wet
if (m_moisture->at(i,j) < 0) m_moisture->at(i,j) = 0;
if (m_moisture->at(i,j) > 1.0) m_moisture->at(i,j) = 1.0;
```

Do a separate pass to run moisture physics

Note that the X dimension of the moisture pass is decreased when the hill is going down (i.e. wrong side of hill)

```
m_moisture->at(i  ,j+1) +=
    (m_moisture->at(i,j)*0.5) *(m_wind->at(i,j).x +
                                m_wind->at(i,j).z * 0.25 +
                                abs(m_wind->at(i,j).z) *
0.05)
                                * 0.98;
if (m_wind->at(i,j).y > 0.0) {
    m_moisture->at(i+1,j+1) += (m_moisture->at(i,j)*0.5) *
                                m_wind->at(i,j).y * 0.98;
} else {
    m_moisture->at(i-1,j+1) -= (m_moisture->at(i,j)*0.5) *
```

```
                                        m_wind->at(i,j).y * 0.98;
        }
        m_moisture->at(i,j) *= 0.5;
```

Blur everything

```
        m_moisture->boxBlur();
```

Update vegetation levels

```
        if (m_moisture->at(i,j) > 0.1) {
            m_vegetation->at(i,j) += 0.01;
        } else {
            m_vegetation->at(i,j) -= 0.01;
        }
```

Then, we can draw things.
Simply lerp the colors between normal and dry if the air is too dry.

```
if (m_height->at(i,j) * 255 > 117 && m_moisture->at(i,j) < 0.1) {
    m_diffuse->at(i,j) = lerpColor(
                            lerpColor(0xFFEEDDBB, 0xFF5B3F31,
                                      (m_height->at(i,j) - 0.458) * 0.8),
                            m_diffuse->at(i,j), 0.2 + 0.8 *
                            m_moisture->at(i,j) * 10.0);
}
```