Procedural Terrain

Computer Graphics and 3D (9 CFU)

Giovanni Burbi

Prof. Stefano Berretti

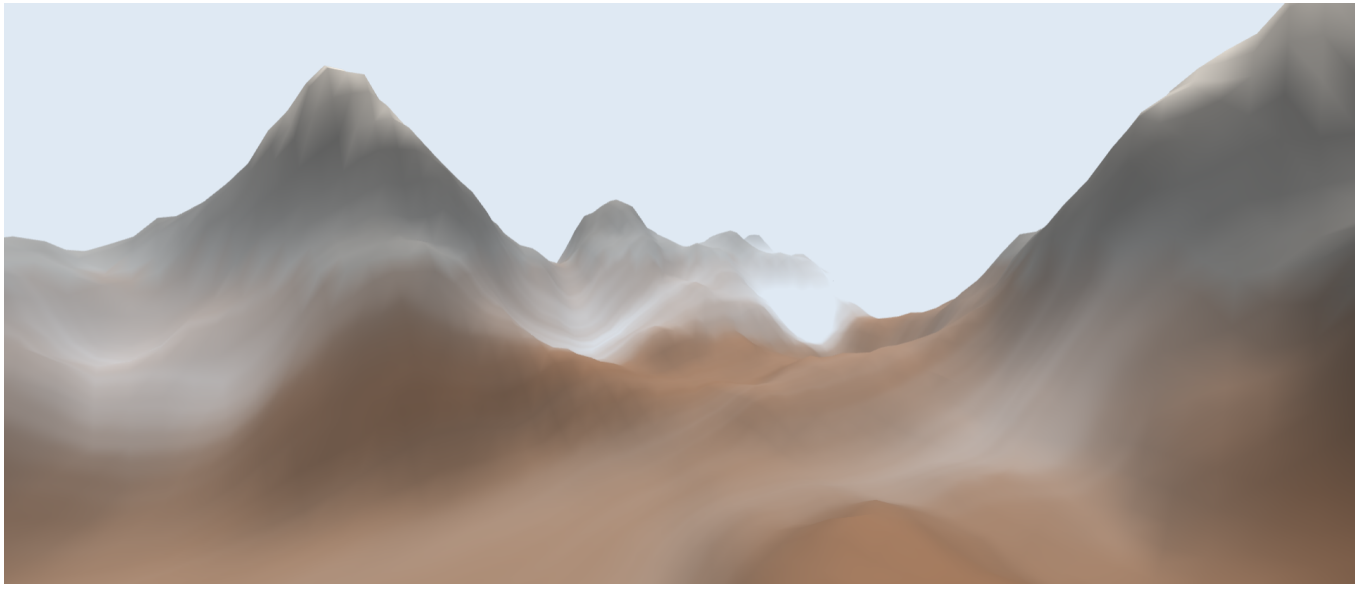
April 2023

Contents Overview

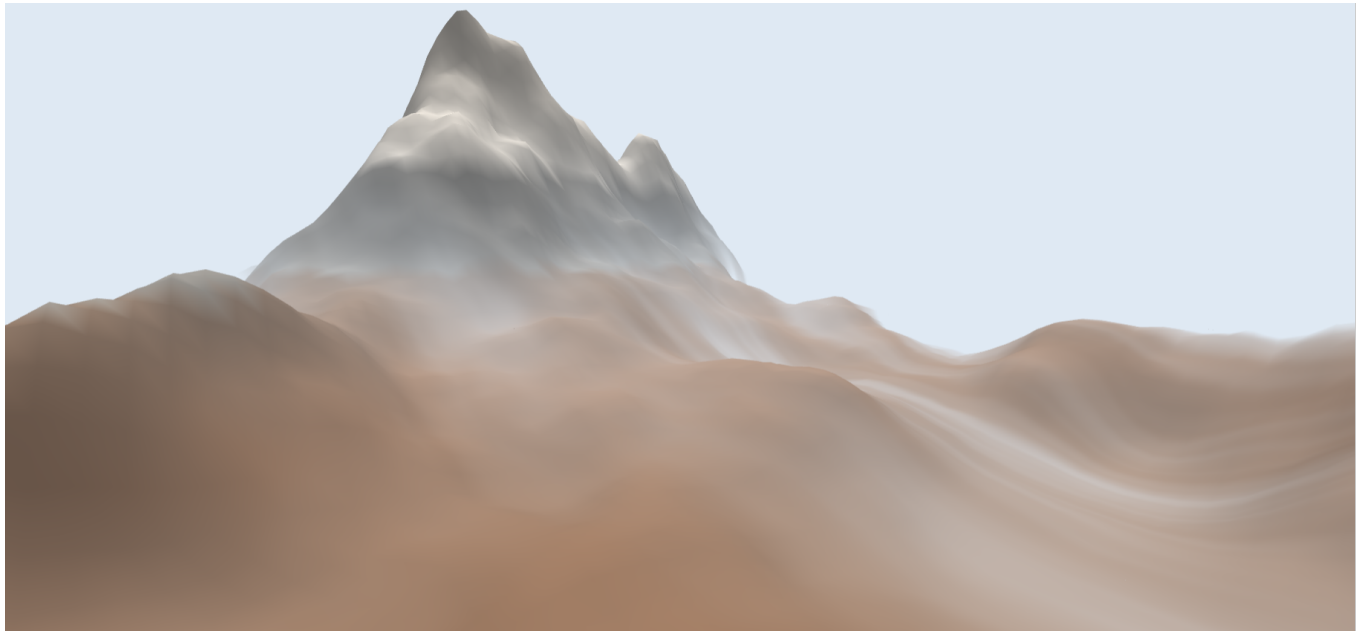
Introduction

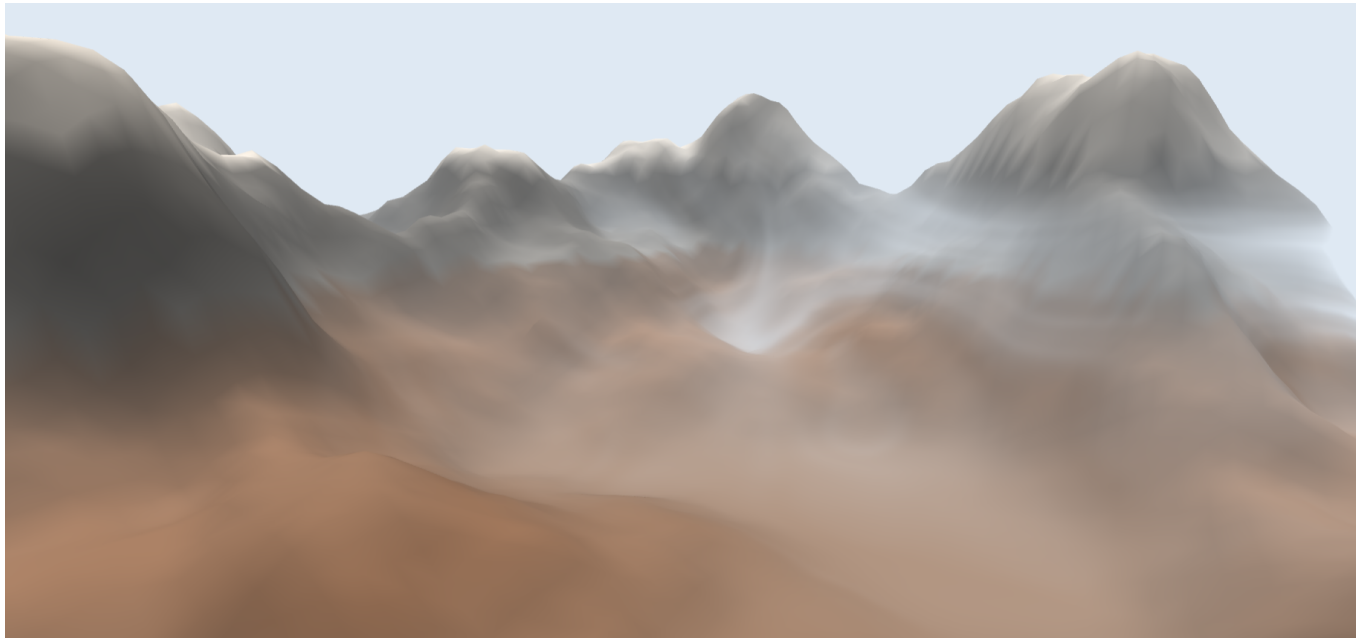
Procedural Content Generation

- **Procedural Content Generation** (PCG) refers to the creation of content by an automated system, rather than being produced manually
- It can be used to generate data, art, video game content, textures, 3d models, etc...
- In this project, PCG has been used to generate **procedural terrains**



Procedural Content Generation





ThreeJS

- **ThreeJS** is a cross-browser JavaScript library and API used to create and display animated 3D computer graphics in a web browser using WebGL
- ThreeJS has been used to implement all the elements described in this presentation

Procedural Terrain

- The main objective of terrain generation is mimicking the **natural look** of real terrains
  - Real terrains are not constructed from Euclid shapes
- Benoit Mandelbrot observed that Earth's rough surfaces can be described as **fractals**

Fractal Noise

Fractals

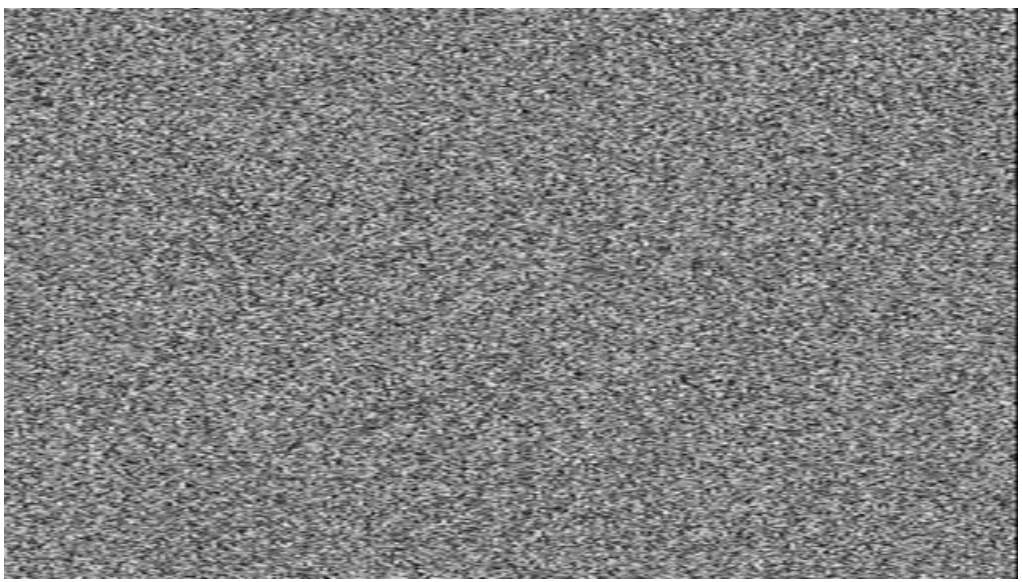
- Fractals can also be used to explain irregular patterns that can be found in aspects of the natural world
- Fractals have two fundamental traits, they are both:
  - **Self-similar**: a fractal can be subdivided into smaller versions of itself
  - **Chaotic**: fractal patterns are products of recursion and, as a result, can be viewed at an infinite number of scales. So they are described as chaotic due to their infinite complexity
- Fractals are **geometric patterns** which can be **generated procedurally**

Noise

- With computers, it is possible to produce seemingly random fractal terrain through the use of **noise functions**
- A noise function is a set of instructions that can be used to generate **pseudo-random noise**
  - It looks random but is **deterministic**: given the same input, it always returns the same value
  - It is based on a **seed** making it repeatable
- In mathematical terms, the noise function is a **mapping** from $\mathcal{R}^n \rightarrow \mathcal{R}$, where n is the dimension of the value passed to the noise function

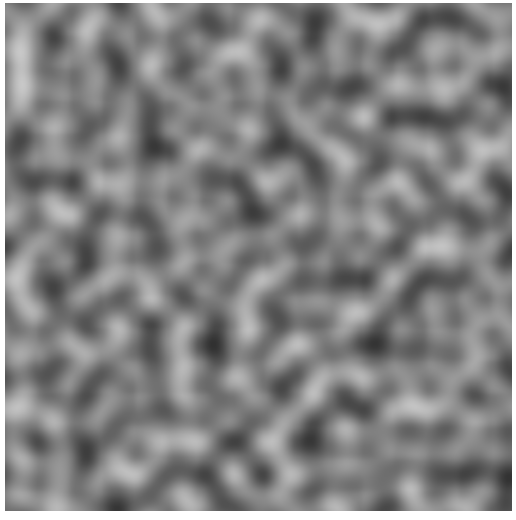
Noise

- There are different types of noise generated by different algorithms
- **Random value noise** is the simplest one
  - But unlike nature, it has **no consistency**, meaning each point is independent of others
  - They have no relationship and show **no discernible pattern**



Noise

- A little bit of randomness can be a good thing when programming organic forms
  - However, randomness as the only guiding principle is not necessarily natural
  - We want some **coherence** in the randomness, we want the noise produced by these functions to have **structure**
- **Perlin noise** can be used to generate various effects with natural qualities

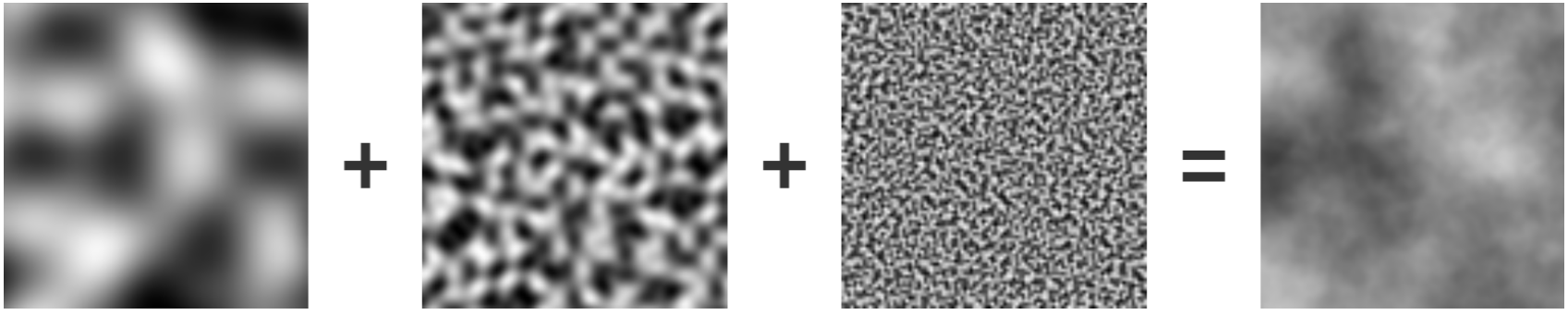


Perlin Noise

- Perlin noise is a type of gradient noise and it has a realistic organic appearance because it produces a **structured, or smooth, sequence of pseudo-random numbers**
  - Sampling the noise generator at two nearby points, will return two similar results
- There is also an improved implementation of the gradient noise called **Simplex noise**

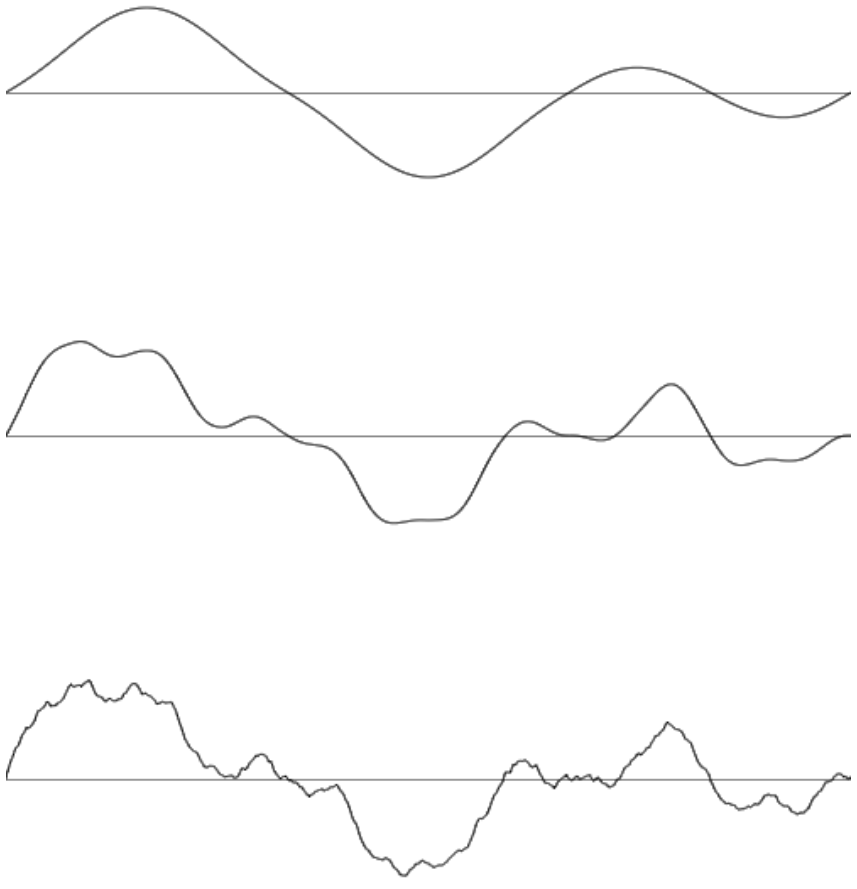
Fractal Brownian Motion

- Many noise functions, such as Perlin noise and Simplex noise, are **not inherently fractal**
- To produce fractal behavior that mimics the appearance of natural terrain, we can use a noise function in conjection with **Fractal Brownian Motion** (FBM)
- FBM is a technique that sums together multiple layers of noise varying some of their parameters in a coherent manner (**fractal sum**)
- The result is a finer, more detailed and controllable noise



Fractal Brownian Motion

- Looking at the noise in 1D we can better see the added details with the number of layers: from top, the noise is the result of 1, 3 and 8 layers added together



FBM Parameters

- **Scale**: it controls the scale of the generated noise
- **numLayers**: how many layers of noise will be added together
- **Persistence**: manage the rate by which the amplitude changes from layer to layer
  - It is a number between 0 and 1
  - It indicates **how much each layer contributes to the final noise value**
- **Lacunarity**: manage the rate by which the frequency changes from layer to layer
  - it determines **how much detail each layer adds to the final noise value**

FBM Algorithm

Algorithm Fractal Brownian Motion

procedure $\mathrm{FBM}(x, y, numLayers, lacunarity, persistence, scale)$

$amplitude \leftarrow 1.$

$frequency \leftarrow 1.$

$noiseSum \leftarrow 0.$

$norm \leftarrow 0.$

$xn = x/scale$

$yn = y/scale$

for $(i = 0; i < numLayers; i++)$ **do**

$noiseSum += noise(xn * frequency, yn * frequency) * amplitude$

$norm += amplitude$

$amplitude *= persistence$

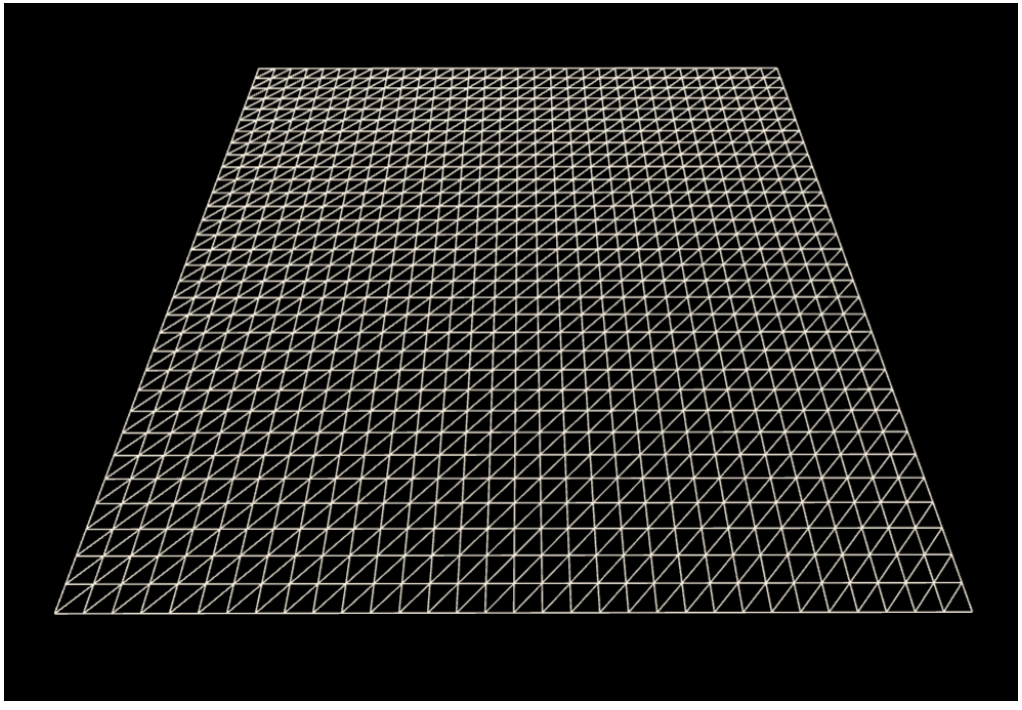
$frequency *= lacunarity$

return noiseSum / norm

Terrain Generation

Heightmap

- Now that we have a way to compute a fractal noise, we can use it to generate a **height map**
- The height map is a 2D grid of height values, each value $\in [0, 1]$, that will be used to **displace the elevation** of the corrispondent vertices of a plane



Heightmap

- To make the displacement visible we multiply the values of the height map by a constant value

---

Algorithm Heightmap

---

procedure HEIGHTMAP(*vertices*, *width*, *maxHeight*)

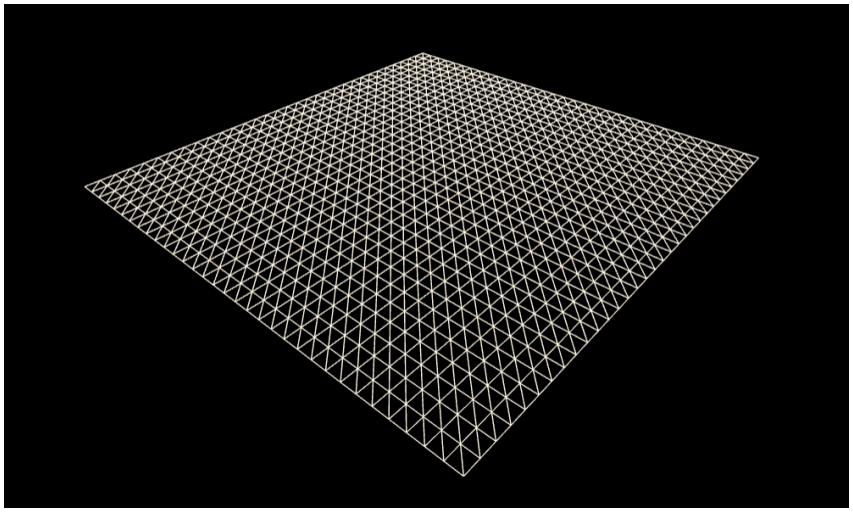
  for $(y = 0; y < width; y++)$ **do**

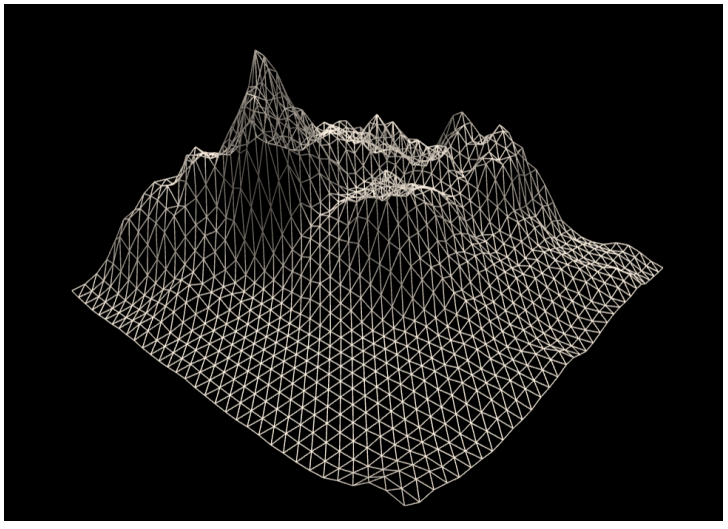
    for $(x = 0; x < width; x++)$ **do**

      $heightmap[y * width + x] = FBM(x, y, ...) * maxHeight$

  return heightmap

Heightmap





Heightmap

- Before the multiplication we can use a **power function** to push middle values down
  - Doing this results in the creation of **flat valleys**, which give a more natural effect to the terrain

---

Algorithm Heightmap

---

procedure HEIGHTMAP($vertices, width, maxHeight, expon$)

  for $(y = 0; y < width; y++)$ **do**

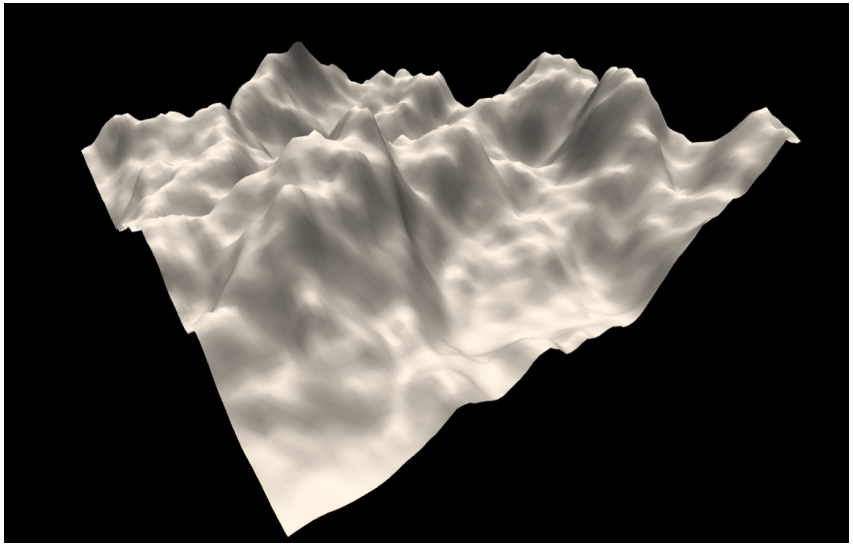
    for $(x = 0; x < width; x++)$ **do**

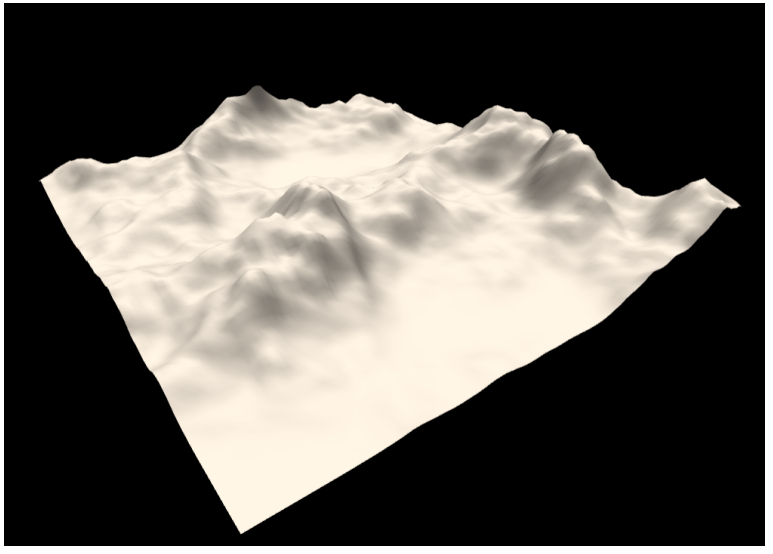
      $value \leftarrow pow(FBM(x, y, ...), expon)$

      $heightmap[y * width + x] = value * maxHeight$

  return heightmap

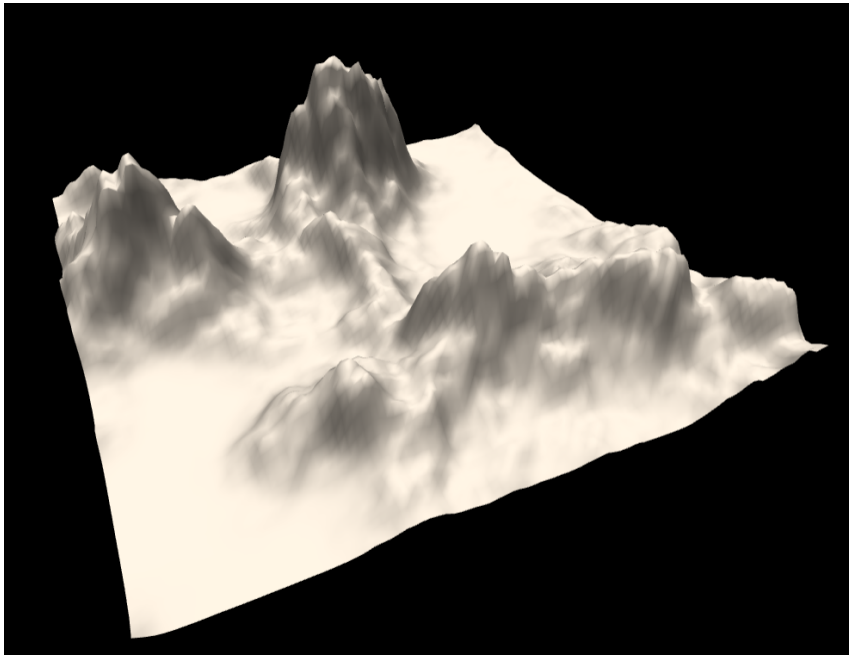
Heightmap

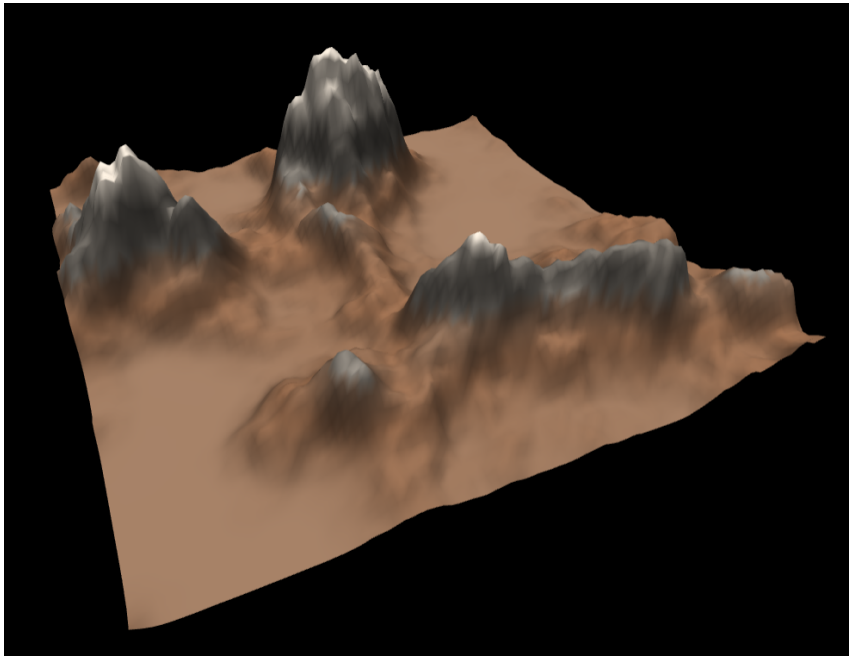




Colors

- We can also use the height map for **coloring the terrain based on the height value** of each vertex in the plane





Infinite Terrain

Infinite Terrain

- Up until now, we worked on a square plane, changing the height values and color of each vertex based on the fractal height map
- However, we can't move the camera too much without **exiting the plane** of our terrain
- We may think to simple increase the dimension of the plane, but this will cause a problem with the usage of resources, which are finite
- A solution to make the terrain appear infinite without using excessive computer resources involves the use of a technique called **Tiling**

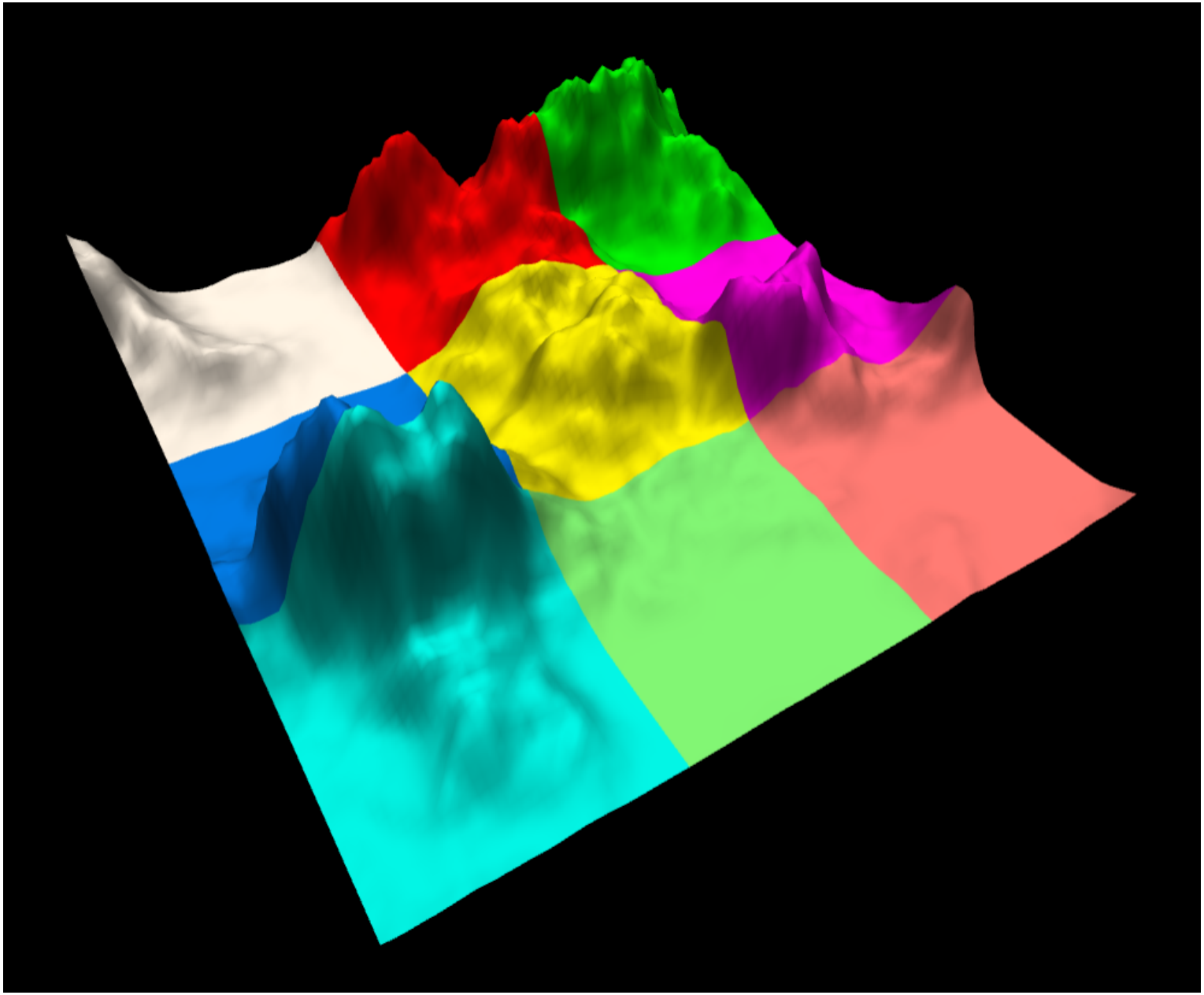
Tiling

- Instead of having only one tile for our terrain, we can generate more tiles and place them in a **3x3 grid**, positioning the camera in the center of the grid
- This will make the terrain appear bigger and it will allow us to move the tiles of the grid based on the moviments of the camera in a way that, at every time instant, the camera is on the center tile of the grid
  - This way we obtain a **seemingly infinite terrain** using only the resources needed for 9 tiles

Tiling

- Thanks to the proprieties of the noise function we used to generate the fractal on which our terrain is based on, we still have a **continuity** in the terrain between different tiles
  - Each tile will generate a fractal noise using the **positional offset** of the tile
  - Knowing that the noise function is deterministic, we can always **reproduce the same terrain even if the camera moves in and out of a certain region**

Tiling

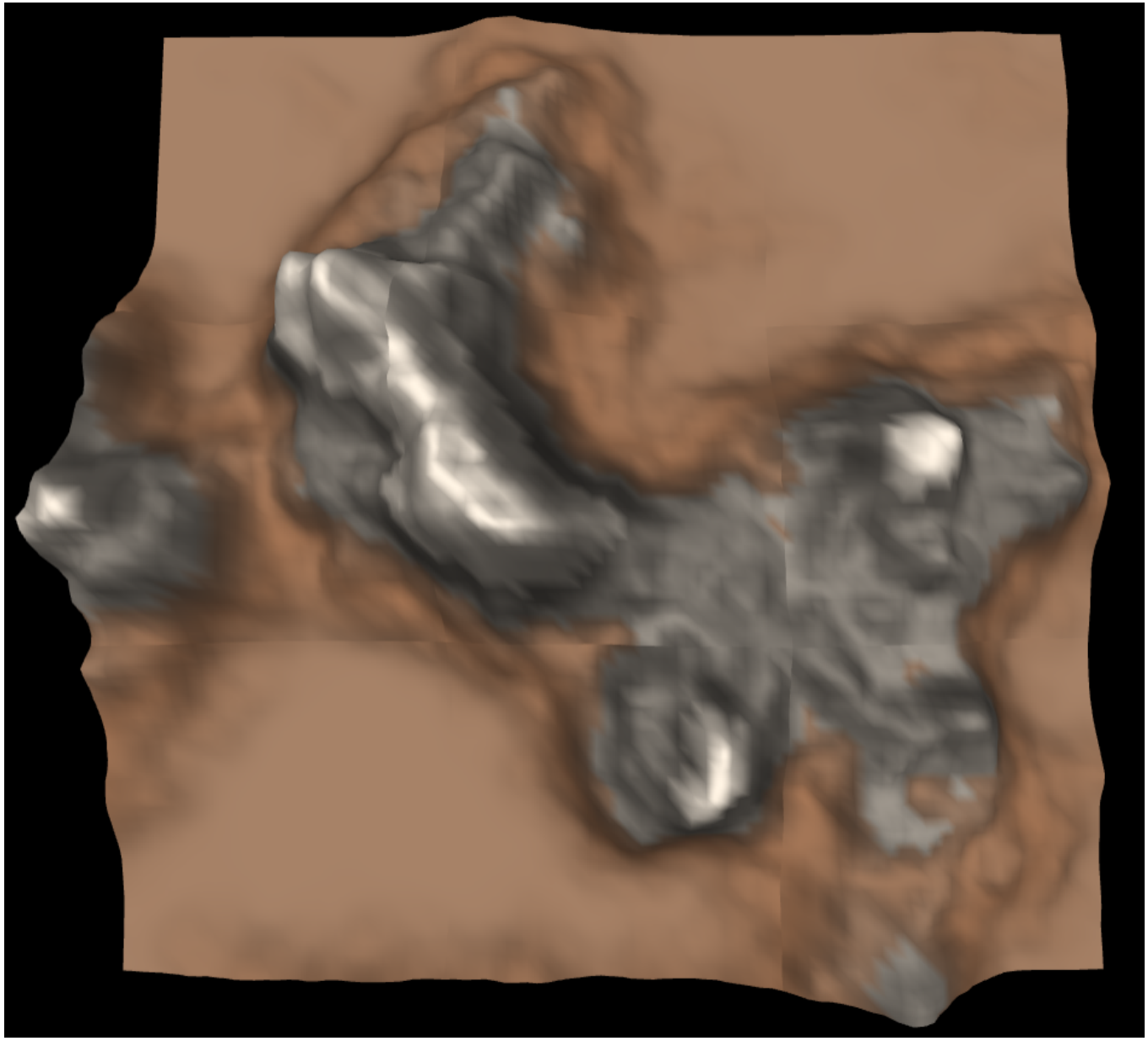


Tiles Normals

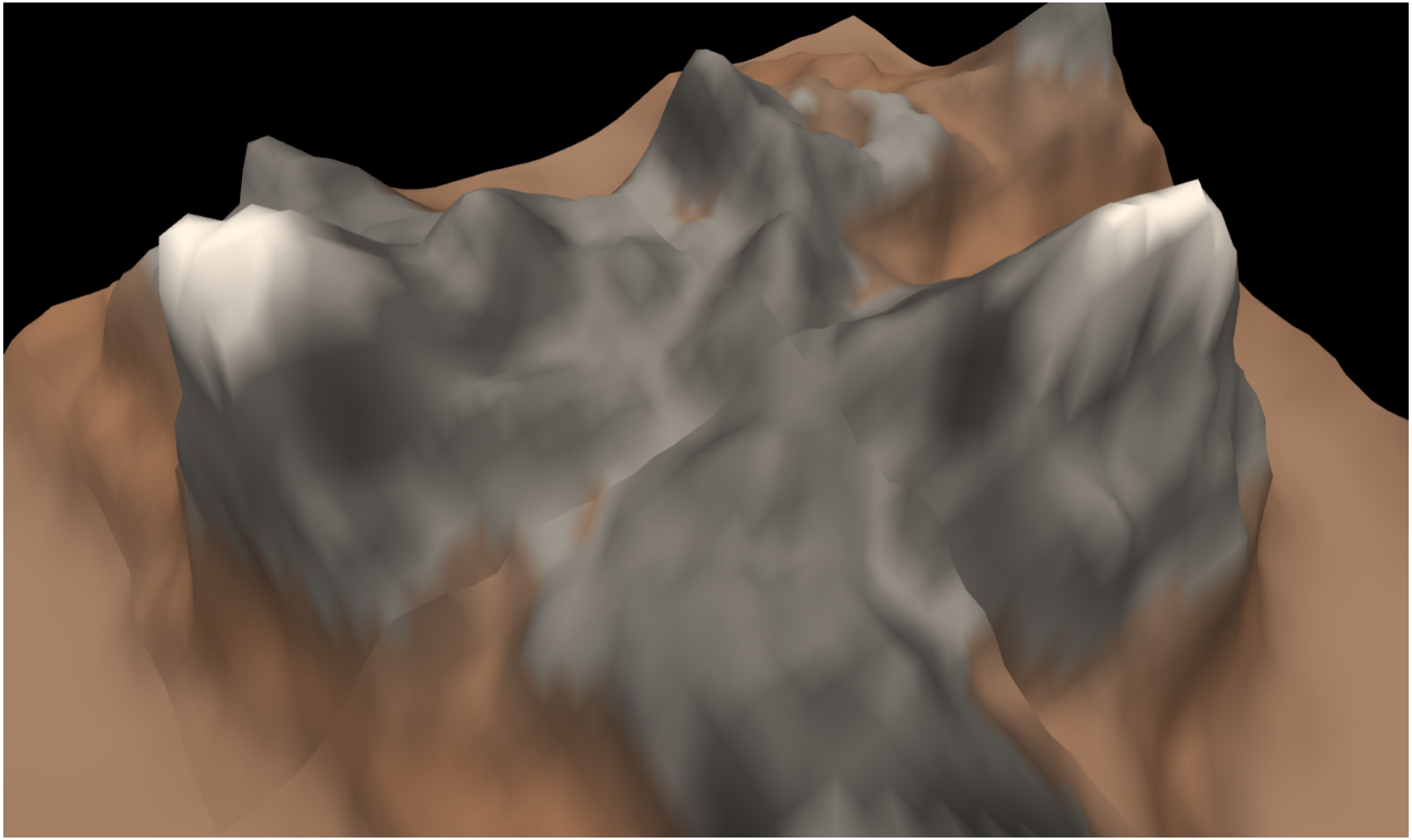
Normals

- Each tile of the grid is an independent plane
- Therefore, when we compute the normals, we compute them for each tile **independently**
  - The normals at the border between adjacent tiles will be computed only with the data available to those tiles, which is **half of what is needed** to compute the correct normal in that point of the terrain
  - Hence there will be vertices that will have **2 different normals in the same point of the terrain**
  - This results in **different shadows** at the edges of adjacent tiles, making the terrain not appear seamless

Normals

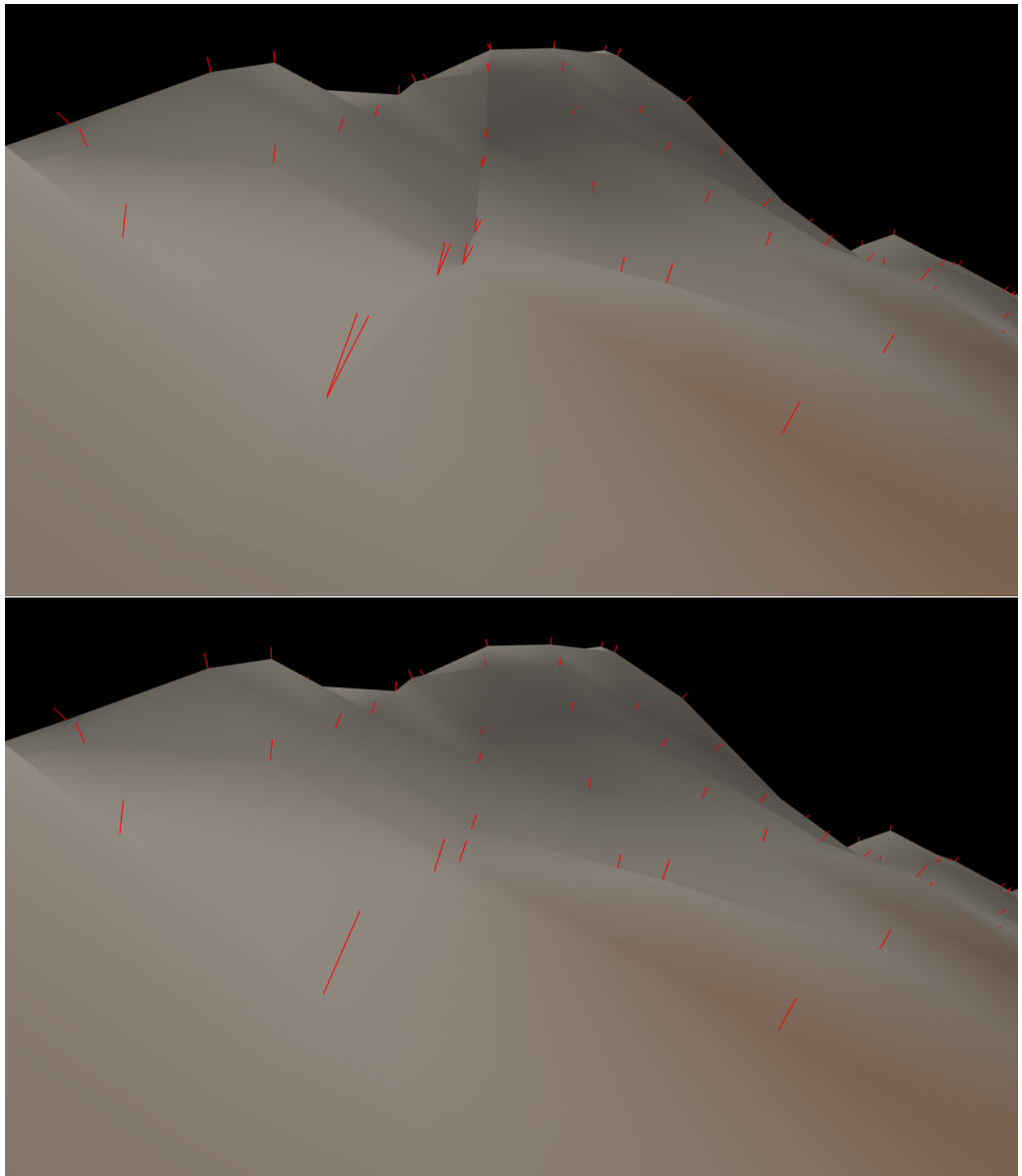


Normals

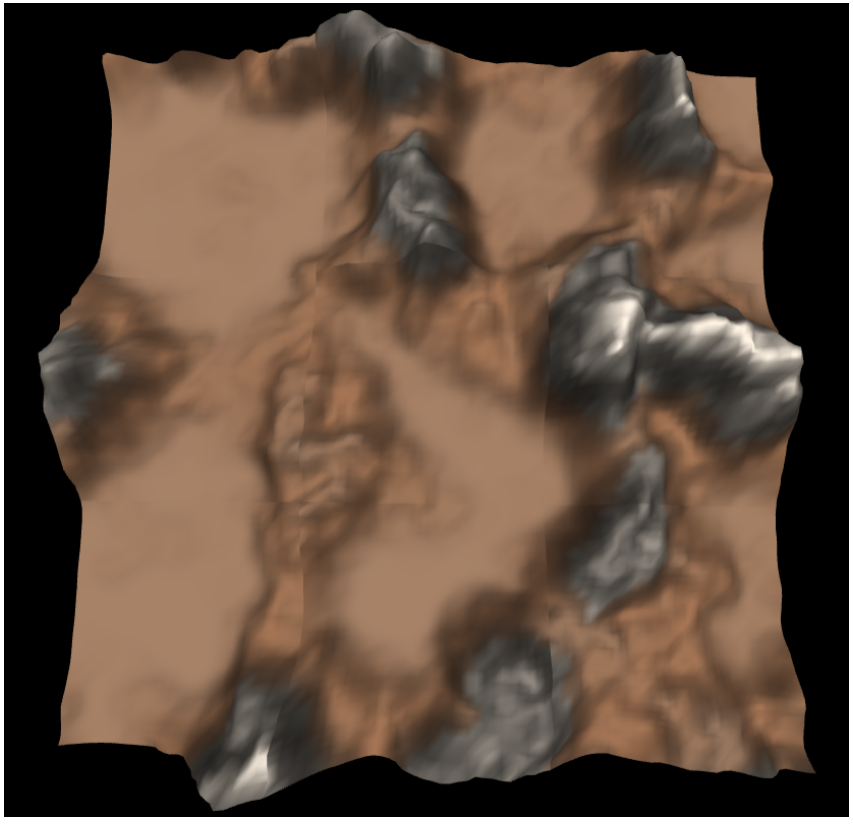


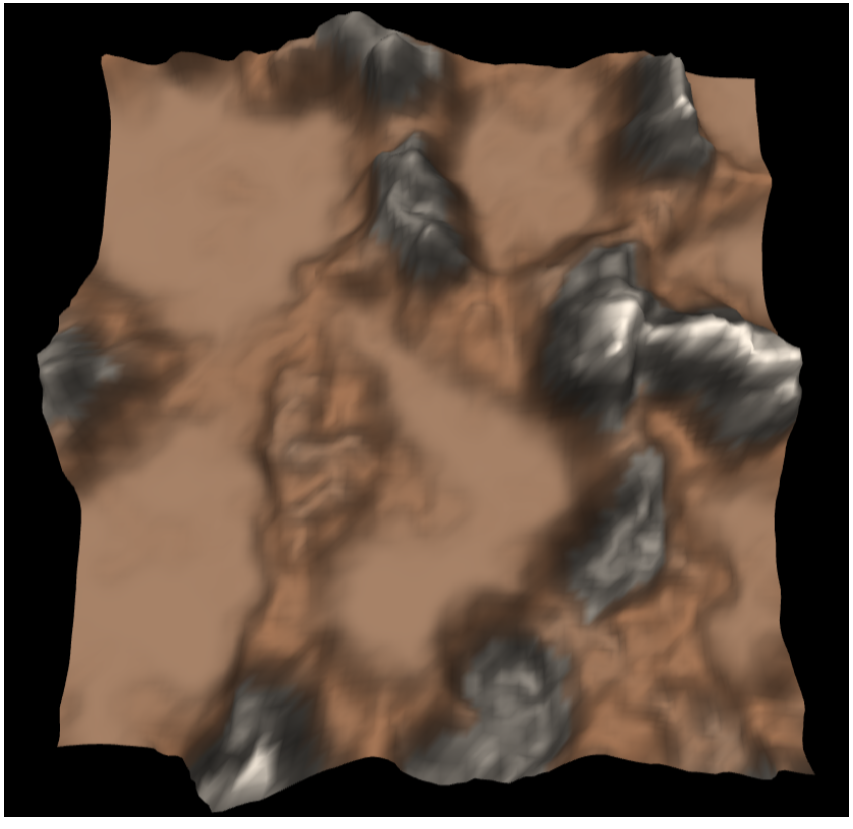
Normals

- To solve this problem we need to **average** the corrisponding normal vectors between adjacent tiles



Normals



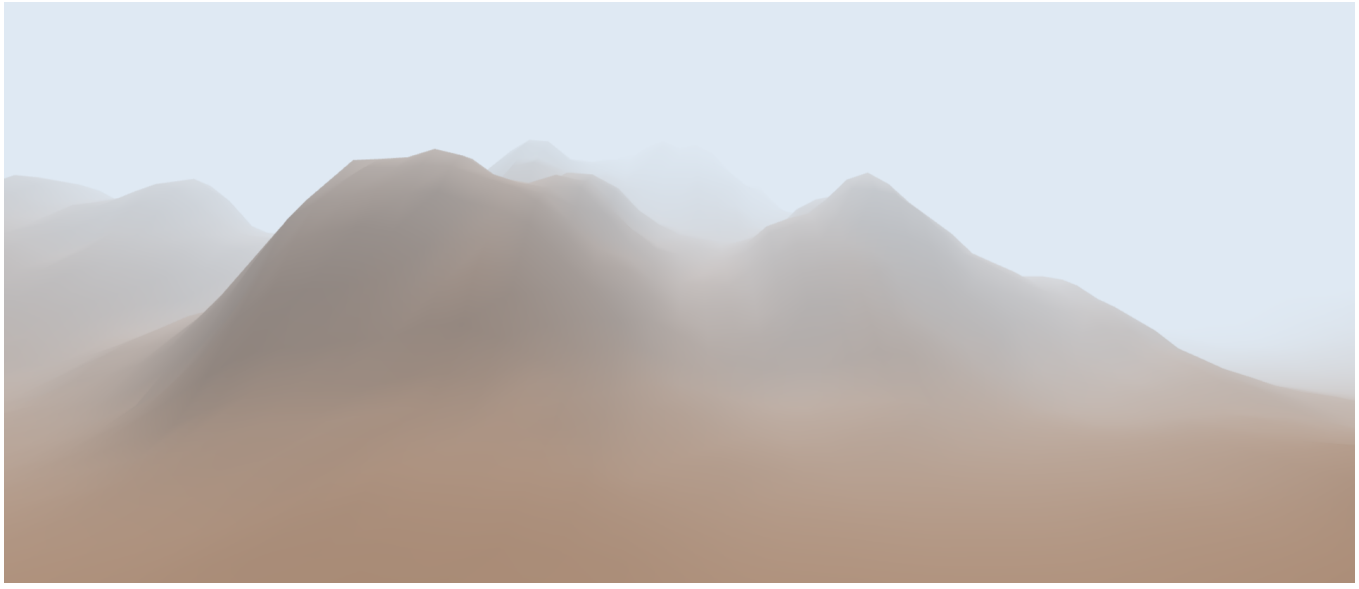


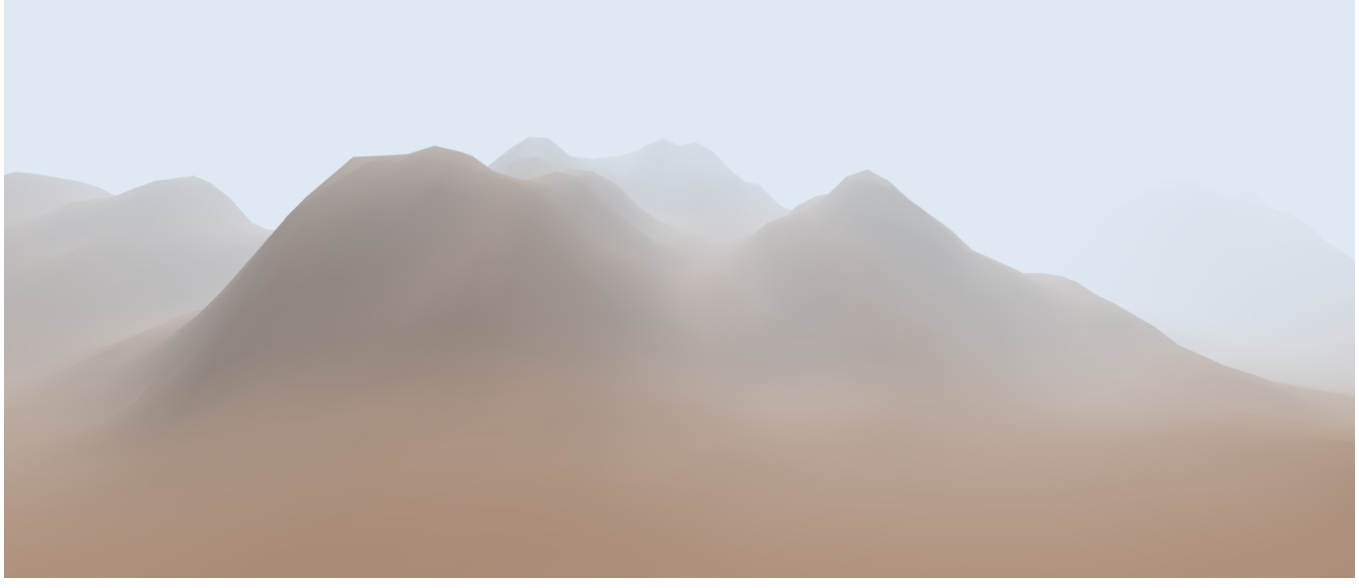
Fog

Fog

- Another element that we can add in our terrain is the **fog**
- Fog is used to give a sense of atmosphere and depth to a scene
- ThreeJS offers two types of fog:
  - **Linear fog**: Fog that grows linearly denser with the distance from the camera
  - **Exponential Squared Fog**: Fog that gives a clear view near the camera and a faster than exponential densening fog farther from the camera

Standard Fog





Volumetric Fog

- The standard fogs of ThreeJS are constant in respect to the height of the scene
- Realistic fog usually **falls off as the height increases**
  - We can model the density variation with an **exponential**
- We can improve the standard fog by making it **volumetric**

Volumetric Fog

- To use a custom shader for the volumetric fog with ThreeJS, we have to replace the standard fog shader implementation using the **shaders injection points** of the library

```
THREE.ShaderChunk.fog_pars_fragment = `
#ifdef USE_FOG
  uniform vec3 fogColor;
  varying float vFogDepth;
  #ifdef FOG_EXP2
    uniform float fogDensity;
  #else
    uniform float fogNear;
    uniform float fogFar;
  #endif
#endif`;

THREE.ShaderChunk.fog_pars_vertex = `
#ifdef USE_FOG
  varying float vFogDepth;
#endif`;
```

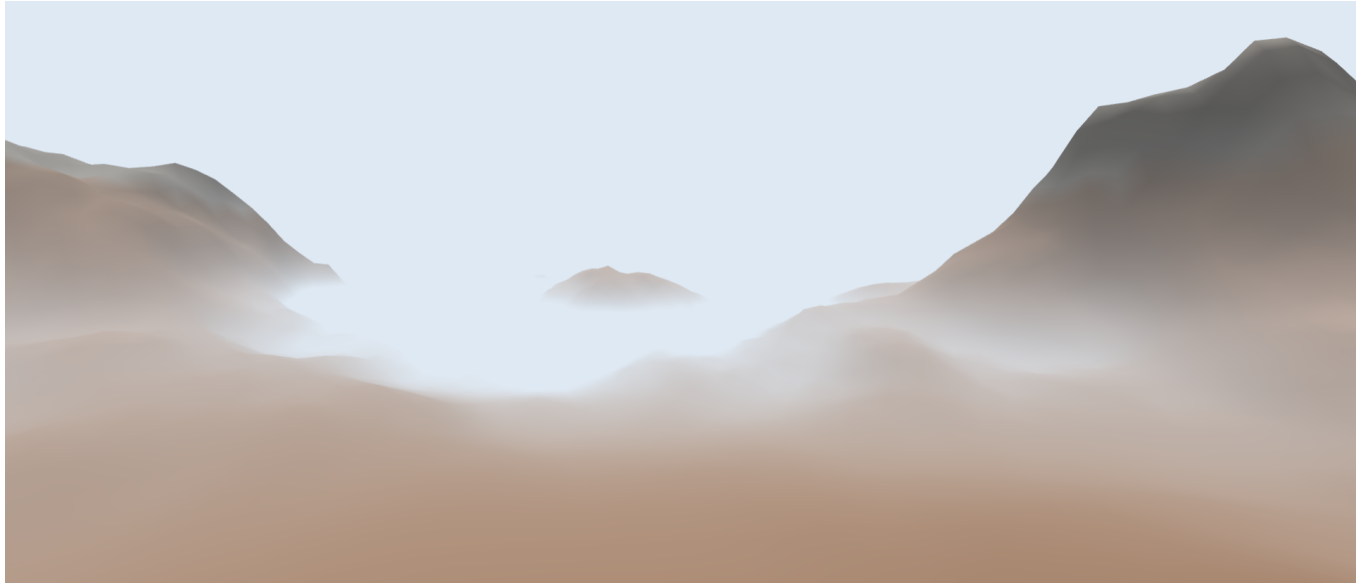
```
THREE.ShaderChunk.fog_pars_fragment = `
#ifdef USE_FOG
  uniform vec3 fogColor;
  varying vec3 vWorldPosition;
  #ifdef FOG_EXP2
    uniform float fogDensity;
  #else
    uniform float fogNear;
    uniform float fogFar;
  #endif
#endif`;

THREE.ShaderChunk.fog_pars_vertex = `
#ifdef USE_FOG
  varying vec3 vWorldPosition;
#endif`;
```

Volumetric Fog

```
THREE.ShaderChunk.fog_fragment = `
#ifdef USE_FOG
  #ifdef FOG_EXP2
    float fogFactor = 1.0 - exp( - fogDensity * fogDensity * vFogDepth * vFogDepth );
  #else
    float fogFactor = smoothstep( fogNear, fogFar, vFogDepth );
  #endif
  gl_FragColor.rgb = mix( gl_FragColor.rgb, fogColor, fogFactor );
#endif`;

THREE.ShaderChunk.fog_vertex = `
#ifdef USE_FOG
  vFogDepth = - mvPosition.z;
#endif`;
```

```
THREE.ShaderChunk.fog_fragment = `
#ifdef USE_FOG
  #ifdef FOG_EXP2
    vec3 fogOrigin = cameraPosition;
    vec3 fogDirection = normalize(vWorldPosition - fogOrigin);
    float fogDepth = distance(vWorldPosition, fogOrigin);
    fogDepth *= fogDepth;
    float heightFactor = 0.05;
    float fogFactor = heightFactor * exp(-fogOrigin.y * fogDensity) * (
        1.0 - exp(-fogDepth * fogDirection.y * fogDensity)) / fogDirection.y;
    fogFactor = saturate(fogFactor);
  #else
    float fogFactor = smoothstep( fogNear, fogFar, vFogDepth );
  #endif

  gl_FragColor.rgb = mix( gl_FragColor.rgb, fogColor, fogFactor );
#endif`;

// get camera position and direction
THREE.ShaderChunk.fog_vertex = `
#ifdef USE_FOG
  vWorldPosition = worldPosition.xyz;
#endif`;
```

Volumetric Fog

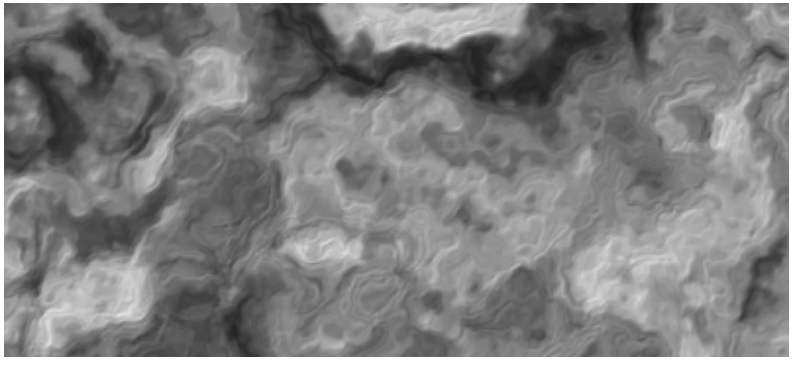


Further Improviments To The Fog

- Another thing about ThreeJS fog is that it is uniform in respect to the depth value
- Real fog has areas that are denser than others, and also a **shape** based on the wind currents
- We can further improve the fog giving it more complexity using a **fractal noise**

Domain Wrapping

- To obtain a realistic effect, we can **distort the fractal noise** of the fog
- Using a FBM to offset the coordinates of another FBM (creating a recursion) we can get a **flowing looking noise**
- This technique is called **Domain Wrapping**
  - Wrapping consists on **distorting the domain of a function with another function** before evaluating the former: $f(p) = f(g(p))$
  - In this case, we use a FBM to warp a space of a FBM: $g(p) = p + f(p)$, where f is the FBM function

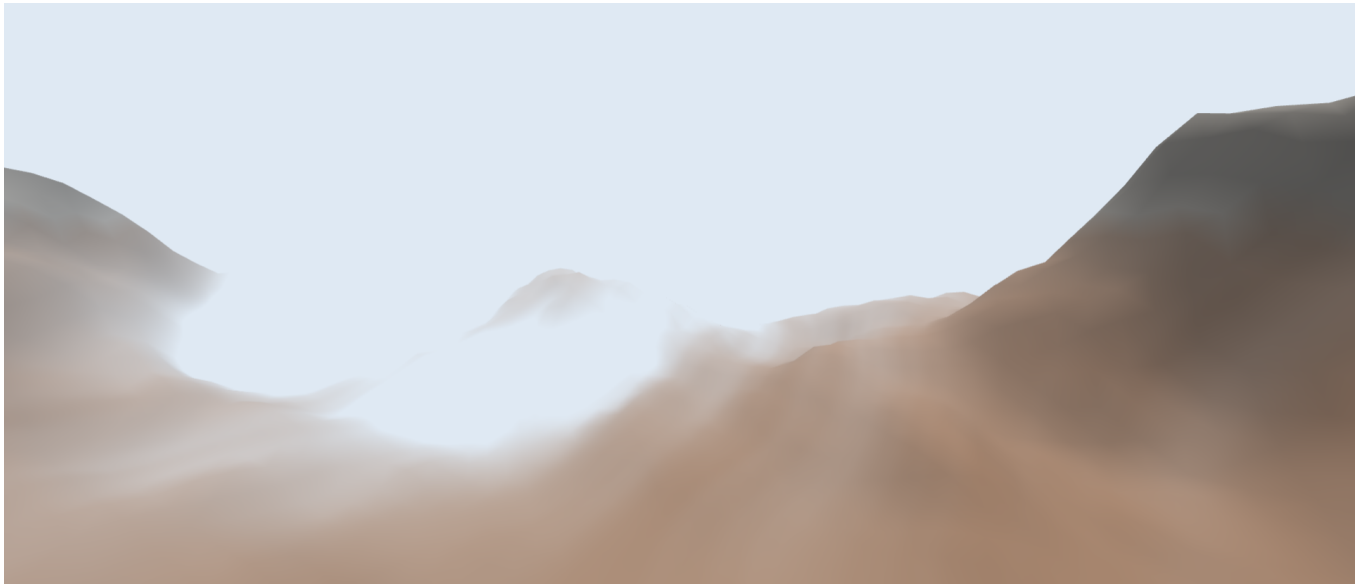


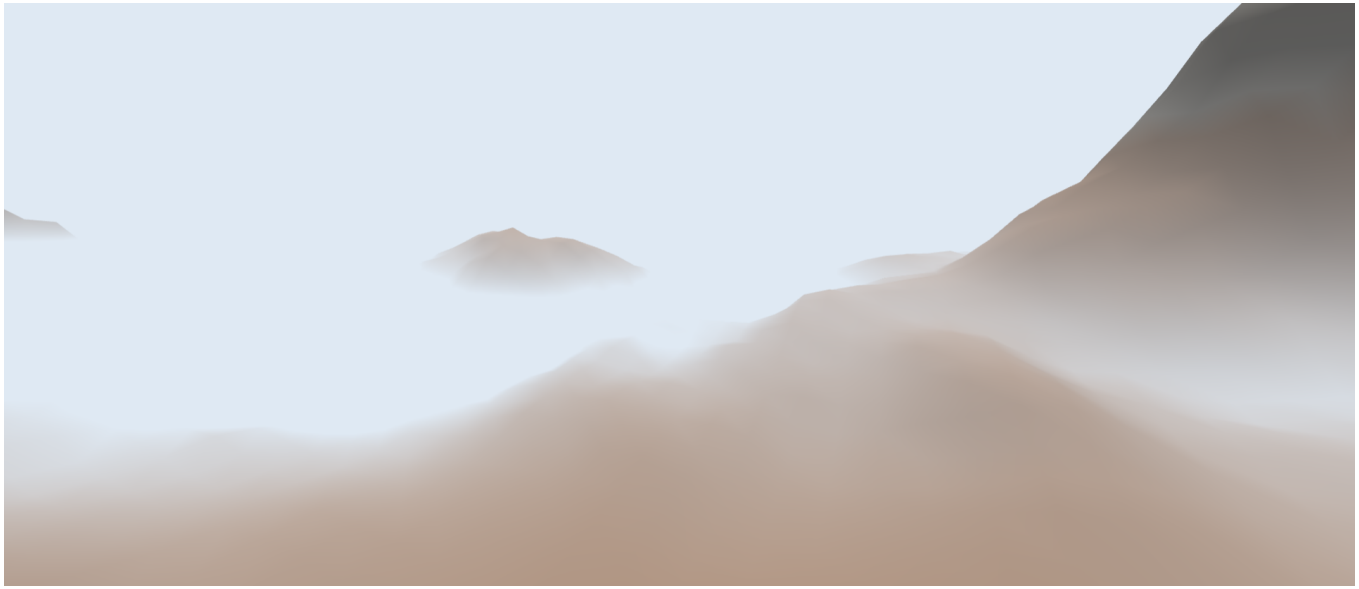
Wrapped Noise Based Fog

```
THREE.ShaderChunk.fog_fragment = `
#ifdef USE_FOG
  #ifdef FOG_EXP2
    vec3 fogOrigin = cameraPosition;
    vec3 fogDirection = normalize(vWorldPosition - fogOrigin);
    float fogDepth = distance(vWorldPosition, fogOrigin);
    fogDepth *= fogDepth;
    float heightFactor = 0.05;
    float fogFactor = heightFactor * exp(-fogOrigin.y * fogDensity) * (
        1.0 - exp(-fogDepth * fogDirection.y * fogDensity)) / fogDirection.y;
    fogFactor = saturate(fogFactor);
  #else
    float fogFactor = smoothstep( fogNear, fogFar, vFogDepth );
  #endif

  gl_FragColor.rgb = mix( gl_FragColor.rgb, fogColor, fogFactor );
#endif`;
```

```
THREE.ShaderChunk.fog_fragment = `
#ifdef USE_FOG
  #ifdef FOG_EXP2

    vec3 fogOrigin = cameraPosition;
    vec3 fogDirection = normalize(vWorldPosition - fogOrigin);
    float fogDepth = distance(vWorldPosition, fogOrigin);

    vec3 noiseSampleCoord = vWorldPosition * 0.00025;
    float noiseSample = FBM(noiseSampleCoord + FBM(noiseSampleCoord)) * 0.5 + 0.5;
    fogDepth *= mix(noiseSample, 1.0, saturate((fogDepth - 5000.0) / 5000.0));
    fogDepth *= fogDepth;
    float heightFactor = 0.05;
    float fogFactor = heightFactor * exp(-fogOrigin.y * fogDensity) * (
        1.0 - exp(-fogDepth * fogDirection.y * fogDensity)) / fogDirection.y;
    fogFactor = saturate(fogFactor);

  #else
    float fogFactor = smoothstep( fogNear, fogFar, vFogDepth );
  #endif

  gl_FragColor.rgb = mix( gl_FragColor.rgb, fogColor, fogFactor );
#endif`;
```

Wrapped Noise Based Fog





Thanks For The Attention