Procedural Generation of Terrains Using Perlin Noise

SHANNON WILLIAMS, Harrisburg University of Science and Technology

Abstract: Over the years, noise algorithms have been used for various 2D/3D modeling and image processing techniques. This paper covers the use of Perlin noise to generate a height map and ultimately a terrain. Afterwards, this paper will introduce the notion of using parametric functions to generate the terrain in a more desirable fashion.

CS Concepts: • **Applied Mathematics → 3D Mathematical Modeling**; • **Web Programming→** Three.js (HTML, CSS, Javascript)

KEYWORDS

Simplex noise, Perlin noise, Three,js, functions, red-green-blue (RGB) mode, red-green-blue-alpha (RGBA) mode, gradient, height map,

1 INTRODUCTION

Creating terrains in CGI scenes or video game scenes are quite time consuming. Additionally, not everyone is skilled in environment modeling, such as game programmers, and as such will have difficulty doing so. Luckily, modeling can be done by using algorithms to procedurally create terrains during run time or in a pre-render. Although one may not get the terrain exactly as desired, much of the work can be done using one or several noise algorithms. In this project, Perlin noise is used. With this algorithm, I will show how we can achieve procedural generation of a mountainous landscapes in application with other mathematical concepts.

2 METHOD

The project's process is as follows:

1) Testing slight alterations to Perlin Noise algorithm to generate 2D images → Height Maps
2) Applying the proven mathematical model to generate the mesh for a plane → Terrain
3) Incorporating color schemes to the terrain based on height
4) Further defining the landscape

3 PERLIN AND 2D IMAGES (HEIGHT MAP)

3.1 Simplex vs. Perlin

Before using noise, it is important to realize the two most popularly used algorithms, Simplex and Perlin noise. Simplex noise was presented by Ken Perlin in 2001 as a replacement for the original (classical) Perlin noise. Simplex proves to have fewer computations, can scale to 4 or higher dimensions and is easier to implement in hardware. Nonetheless, this project simply focuses on generating terrain, not being specifically targeted towards performance efficiency. With that said, Simplex noise should be considered if performance is a major goal. Another thing to consider is that Perlin uses interpolation of gradient vectors of its surround grid point, while Simplex uses summation instead [1]. Hence, the output slightly differs between the two. Simplex also provides a more contrasting image, which is not necessarily needed for this purpose. For this reason, Perlin was chosen as the main algorithm to use.

3.2 Generating Height Map Images

Using a Perlin noise function, you can create a black and white image. This image ($RGBA$ mode, 0-255 scale) will allow for a good visual of the terrain if the algorithm was applied. To start, a simple image was generated, then followed by slight modifications to the input of the function. When generating an image of say 128 by 128 pixels, you can think of the image as a grid of pixels. This way we can loop through each row and column – x and y, and use those as inputs for the Perlin noise function. Additionally, each pixel is represented with 4 adjacent indices that indicates the $RGBA$ values – which will be the current index, called pxi.

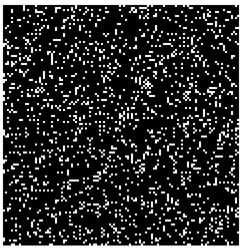

Fig. 1. Black and white image made with Perlin noise by setting each pixel to equal the rounded value of $Perlin(x/2, y/2) \times 255$

The output of the function gives a value from [-1, +1] and so needs to be multiplied by 255 to be a valid value for coloring of pixels.

3.3 Zoom

Furthermore, we can create a “zoom” like effect by dividing input values by larger numbers.

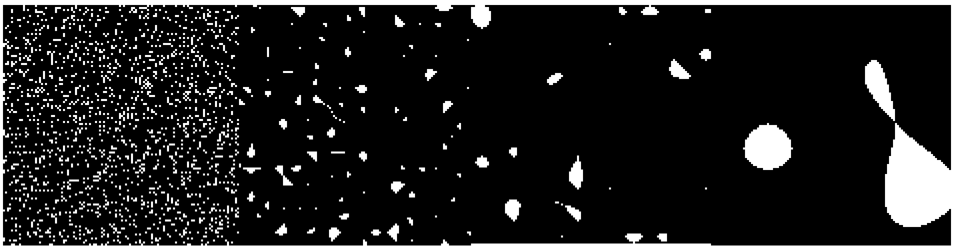

Fig. 2. Images created with zoom levels of 2, 8, 16, and 64, respectively $Perlin(x/zoom, y/zoom) \times 255$

3.4 Grayscale (Turbulence)

To get a smoother terrain, values of the height map cannot be either -1, 0, or +1. Instead, we need a grayscale image which serves to produce a smooth transition from high to low levels of land. Grayscale can be acquired by using the concept of “turbulence” [2]. This is the concept of adding the percentage of multiple zoom level values from the function. Rather than getting either black or white, we will get an interpolation essentially. The image result followed by the pseudocode below explains this further.

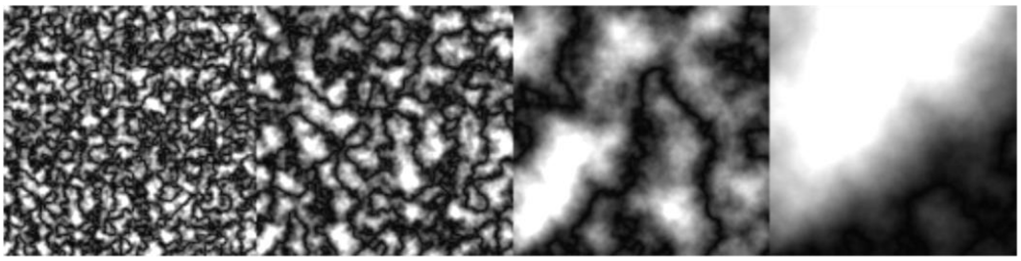
Fig. 3. Images created by adding turbulence with Perlin noise with zoom levels 8, 16, 64, 256

ALGORITHM 1: Turbulence Algorithm

```
// initialsize = zoom
Function turbulence(x, y, initialsize)

value = 0
size = initialsize

while size ≥ 1
    value += Perlin(x/size, y/size) * size
    size /= 2
end

Return | contrast * value / initialsize |
```

Note the return statement of the algorithm is slightly modified to get values with 0-255 scale. Dividing the *contrast * value* by the *initialsize* decreases the brightness.

4 GENERATING TERRAIN

Now that a nice blending result is achieved, this algorithm can now be applied in creating a terrain. Doing this is quite simple. When generating the terrain, create a grid (plane), preferably with the number of vertices as pixels for the height map, in this case 128x128. Having the same size allows for a nice mapping of the texture to the geometry. For the terrain to reflect the height map, each vertex needs to be set to the value of the output of the Turbulence function. You can increase the height factor by any multiplier you wish. This example has a max height of approximately 1.5.

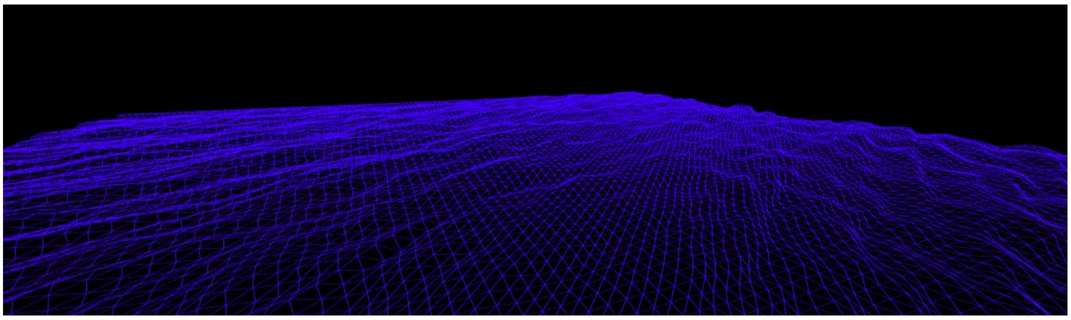
Fig. 4. A wireframe rendered terrain using Turbulence values for each vertex

5 TEXTURE CREATION

Landscape coloring or simultaneously generating a texture and applying the model is easy to do at this point. Below is a snippet of code used which is also quite self-explanatory:

```
function setTerrainTexturePixel(turbulenceval, image, pxi){
  let color = Math.round(turbulenceval * 255);

  if(color <= 50){ // give Water color

    image.data[pxi] = 7;
    image.data[pxi+ 1] = 72;
    image.data[pxi +2] = 234;
  }
  else if( color >= 50 && color <= 100){ // give DARKER grass color

    image.data[pxi] = 1 ;
    image.data[pxi+ 1] = 33;
    image.data[pxi +2] = 22;
  }
  else if(color > 100 && color <= 200){ // give grass color
    image.data[pxi] = 3 ;
    image.data[pxi+ 1] = 73;
    image.data[pxi +2] = 52;
  }
  else if (color >200 && color <= 250){ // DIRTY snow
    image.data[pxi] = 200 ;
    image.data[pxi+ 1] = 200;
    image.data[pxi +2] = 200;
  }
  else{ // pure white snow
    image.data[pxi] = image.data[pxi+1] = image.data[pxi+2] = 255;
  }

  image.data[pxi+3] = 255;

}
```

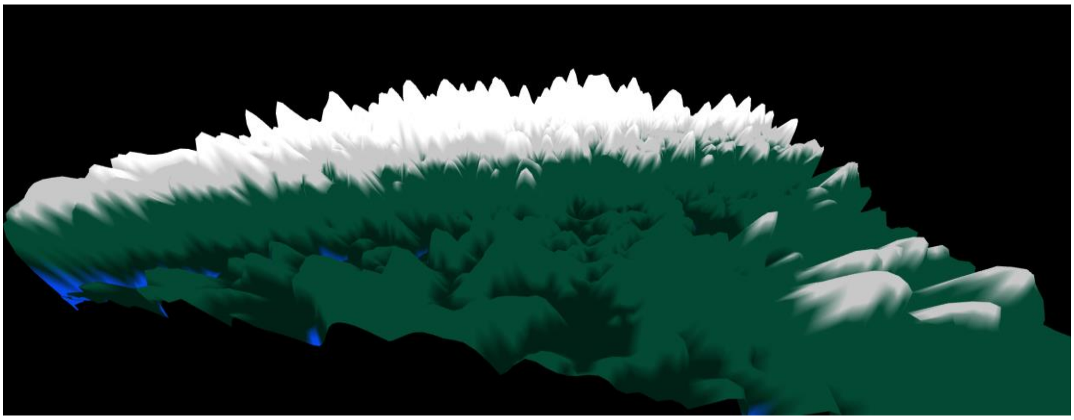

Fig. 5. Applying texture created from Turbulence values

To make the texturing have a subtle look, we can apply a gradient:

```
gradient = Math.clamp(turbulenceval,0,1);
    image.data[pxi] = 7 * gradient;
                    …
```

Again, we can use the same Turbulence value received - called Tv for the latter part of the paper. As we have created the color ranges (view code from section 5), we can multiply each RGB value by the gradient to achieve linear darkening [3].

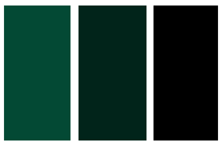

Fig. 6. Multiplying $RGB(3,73,52)$ by Tv 1, 0.5, and 0 respectively to achieve a linear change

Keep in mind that the linear change will be more visible with higher height ranges. As in this case, a very subtle change is seen. Also, with the mapping of color ranges between heights, the maximum darkness or lightness will differ. For example, if $Tv = gradient$ and is between [0.784, 0.396], then multiplied by 255, it falls in normal grass color range. This also means that with the gradient applied, we will get between $RGB(3 \times 0.784, 73 \times 0.784, 52 \times 0.784)$ to $RGB(3 \times 0.396, 73 \times 0.396, 52 \times 0.396)$. In other words, we won't get the original $RGB(3, 73, 52)$ color.

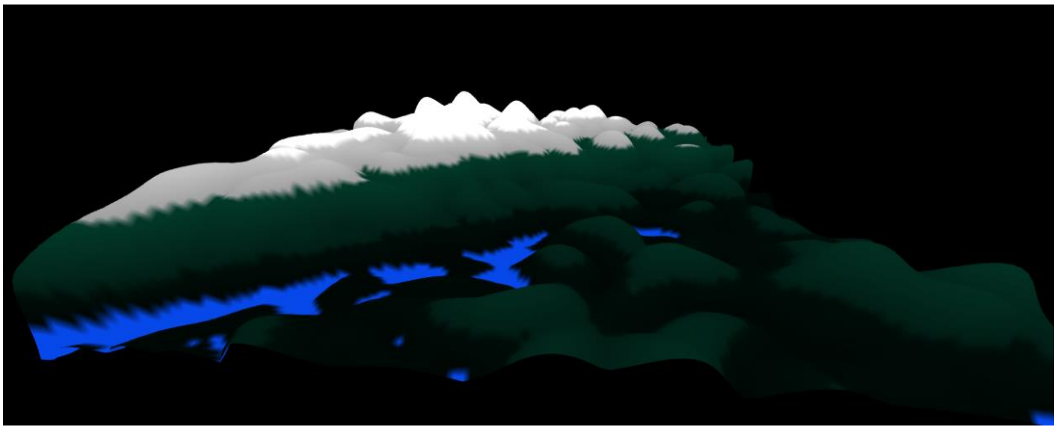

Fig. 7. Multiplying pixel colors by Tv for linear gradient

6. DEFINING MOUNTAIN CHARACTERISTICS

With points being pseudo-random, we can further fine tune the degree of control with the shape of the land. If we use the function Tv^2, then we get similarities with the original Tv^2 curve. Likewise, we can extend this to other functions and even piecewise functions! When observing the results, it is important to consider the probability of Tv falling within a range. For example, by looking at Fig. 8 we can see that if Tv is between [0,0.2], we will get a flat surface. If above 0.2, then we get points that follow the curve upwards. This caveat is trickier when working with piecewise functions.

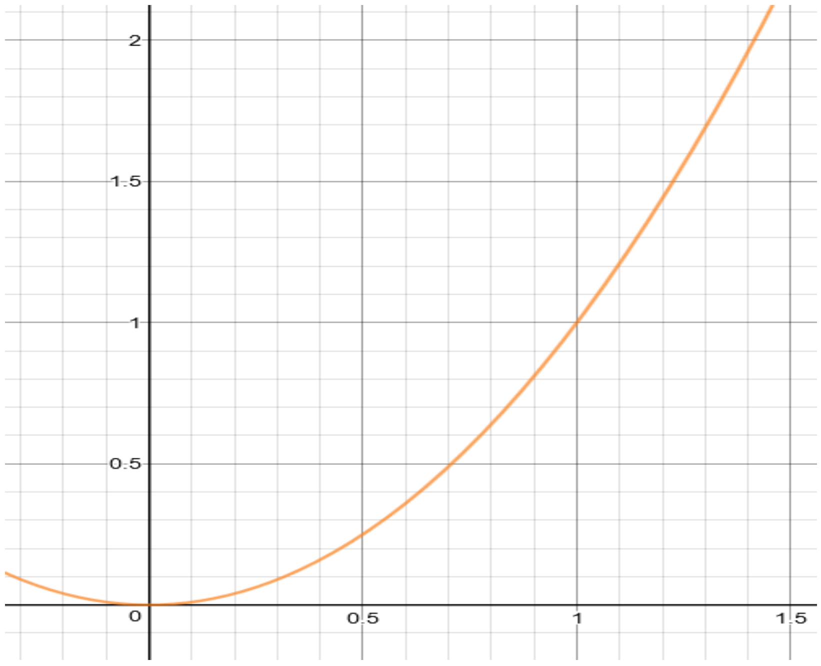

Fig. 8. Original x^2 curve

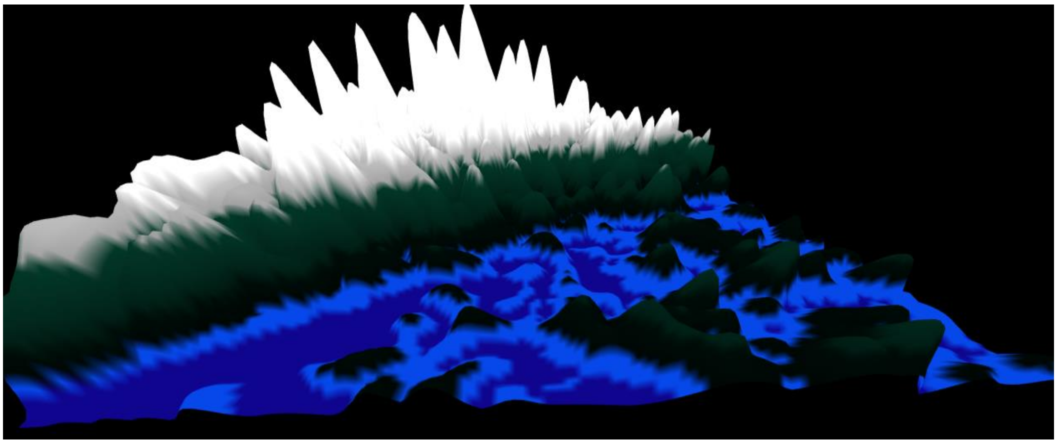

Fig. 9. Results of Tv^2

Extending with application of the piecewise function:

$$f(x) = \begin{cases} Tv^4, & Tv < 0.4 \\ \sin(Tv), & Tv \geq 0.4 \end{cases}$$

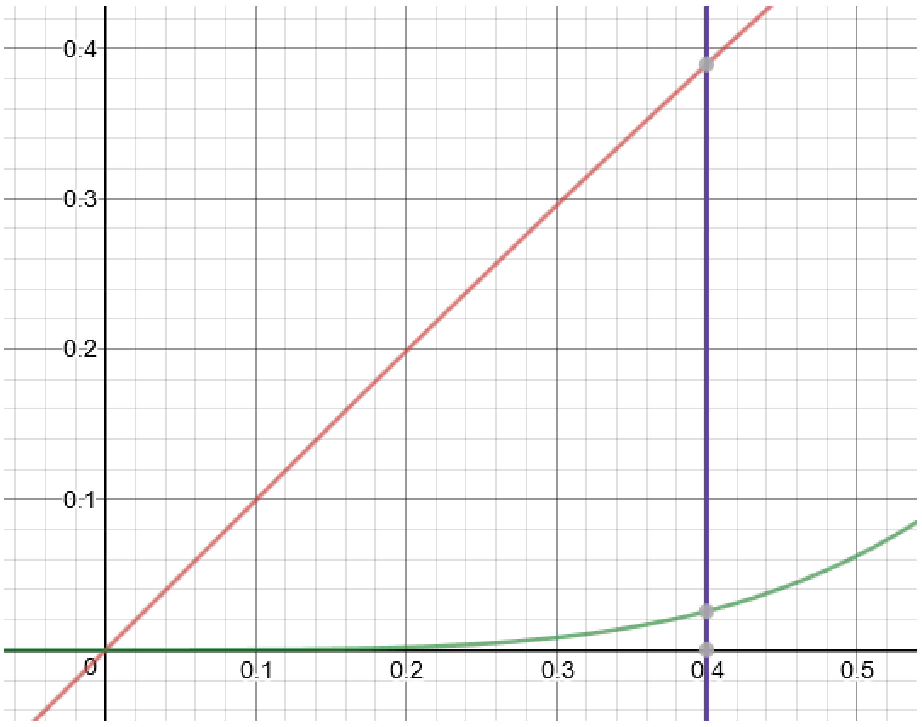

Fig. 10. $\sin(Tv)$, red curve, and Tv^4, green curve, $x = 0.4$, purple line

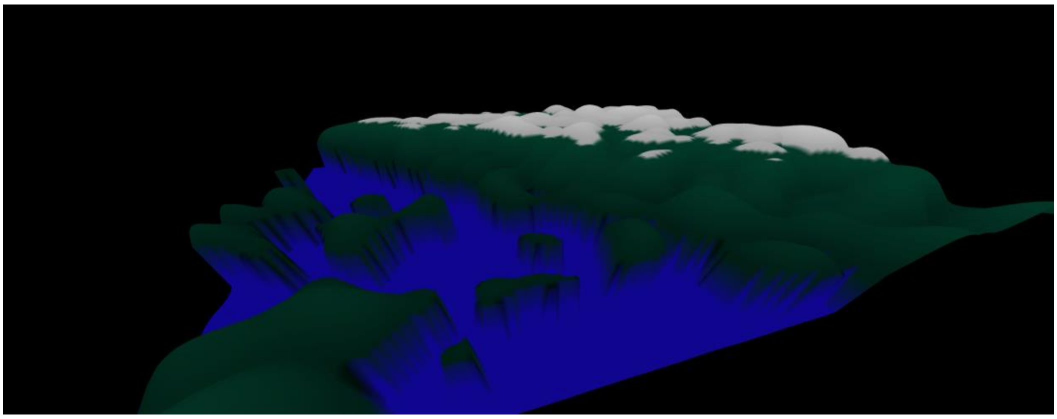

Fig. 11. Running Tv through piecewise function

Notice the sudden vertical change transitioning from one function to another. If Tv is 0.4 then you get 0.25, if Tv is 0.41 you get approximately 0.4. So, going from 0.16 to 0.4 is quite a sudden height change with any two vertices!

7. CONCLUSION

Hopefully at this point you can see the pros and cons of working with Perlin noise for procedural generation. Perlin noise can be used for achieving many more effects, such as randomly applying disperse dead grass, moisture droplets, ice in water, and so on.

REFERENCES

[1] L. Vandevenne, "Texture Generation using Random Noise," [Online]. Available: http://lodev.org/cgtutor/randomnoise.html.

[2] A. Patel, "Making maps with noise functions," [Online]. Available: https://www.redblobgames.com/maps/terrain-from-noise/.

[3] S. Gustavson, "Simplex noise demystified," [Online]. Available: http://weber.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf.