Procedural Generation of Terrain 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

Noise, Perlin noise, Three,js, functions, red-green-blue (RGB), gradient, height map, terrain-generation image

ACM Reference format:

Gorge Gubbiotti, Pam Malagò, Steve Fin, Steve Ram, Som Tacchi, Lamp Giovannini, David Bisero, Mike Madami and Glen Carlotti. 2017. Magnetic Normal Modes of Bi-Component Permalloy Structures. *ACM Trans. Graph.* 4, XXXX. 2 (October 2017), 10 pages.

DOI: 124564

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. However, this challenge 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 landscape in application with other mathematics concepts.

**1**

2 METHOD

The project follows the structure:

1. Testing slight altercations 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 from the pseudo-random generation

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 target towards runt-time or pre-rendered scenery. 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. 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.



Fig. 1. Black and white image made with Perlin noise values for each pixel

(Perlin (x/2, y/2) \* 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. 1. Images created with zoom levels of 2, 8, 16, and 64, respectively

(Perlin (x/zoom, y/zoom) \* 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”. 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.

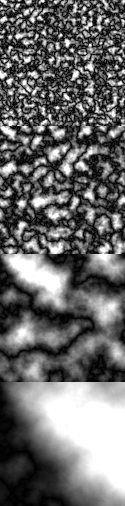


Fig. 2. Images created adding turbulence with Perlin noise with zoom levels 8, 16, 64, 256

**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 iterated and 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 none, so it has a max height of approximately 1.5.

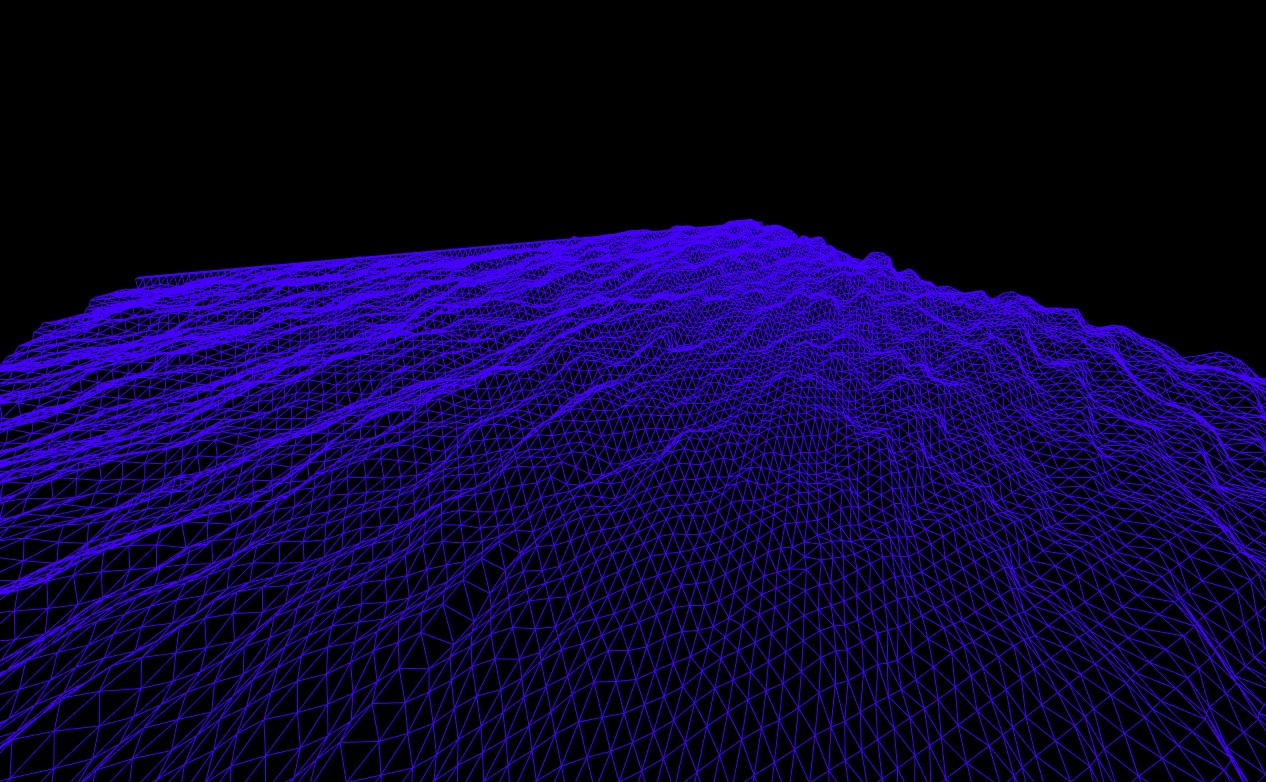


Fig. 3. A rendered terrain using Turbulence values for each vertex

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;

}

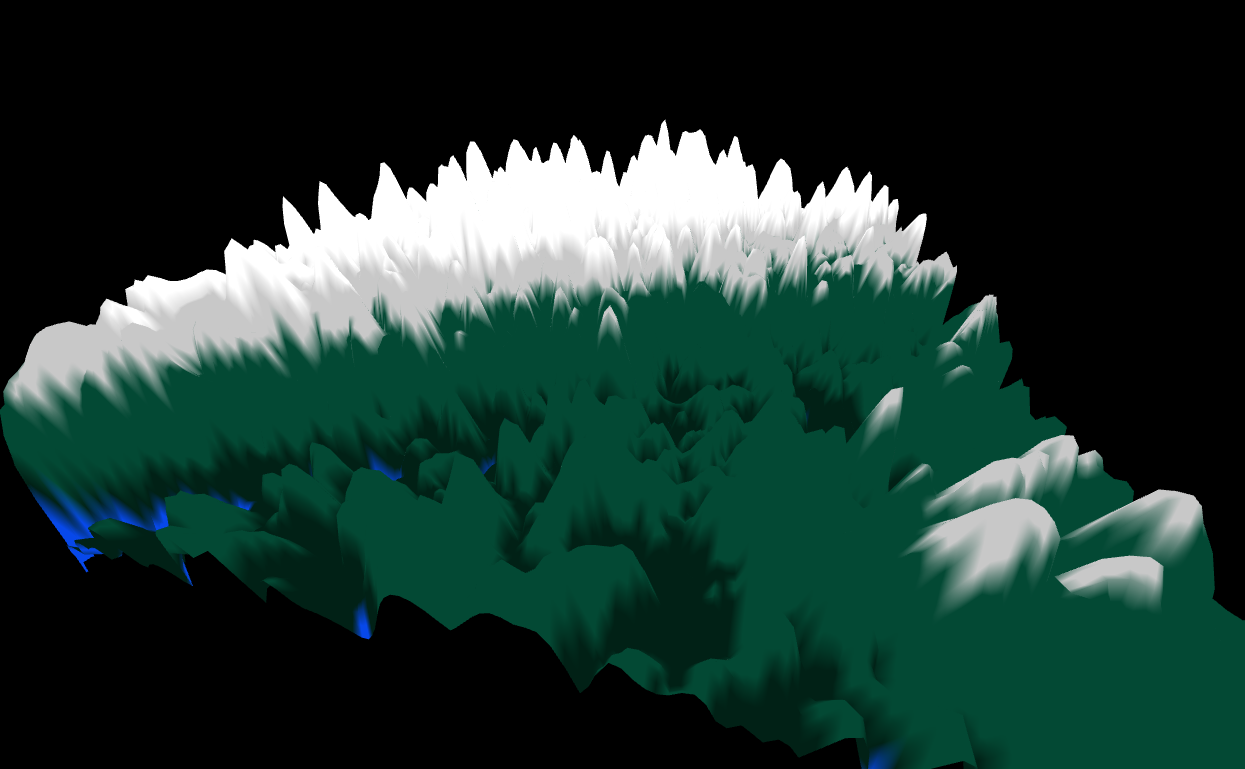


Fig. 4. Applying texture created from Turbulence values

To make the texturing have a subtle look, we can apply a gradient. Again, we can use the same Turbulence value received, called Tv from here. 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 or lightening.

Fig. 5. 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 larger models. 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 is 0.7, Tv\* 255 is 178.5 and falls in normal grass color range. Normal grass has RGB (3, 73, 52), and when multiplied by Tv, the new color is RGB(2,51,36) with each value being floored. In other words, we get approximately 0.7% of the original color’s value.

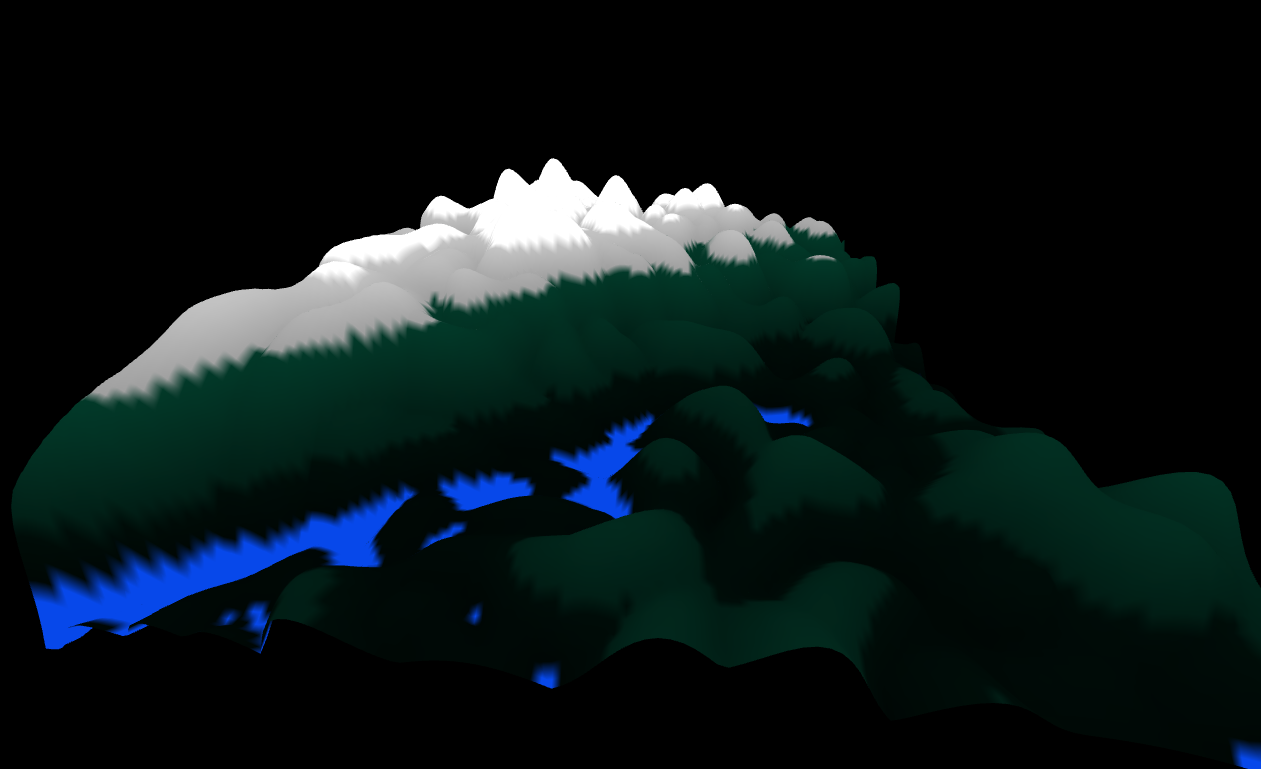


Fig. 6. Multiplying pixel colors by Tv for linear gradient

6. DEFINING MOUNTAIN CHARACTERISTICS

Even though points are pseudo-random, we can still have some degree of control with the shape of the land. If we use the function Tv^2, then we get similarities with the original x^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 that Tv falling within a range. For example, by looking at Fig. 7 we can see that if Tv is between [0,0.2], we will get a roughly flat result. If above 0.2, then we get points that follow the curve upwards. This is caveat is trickier when working with piecewise functions.

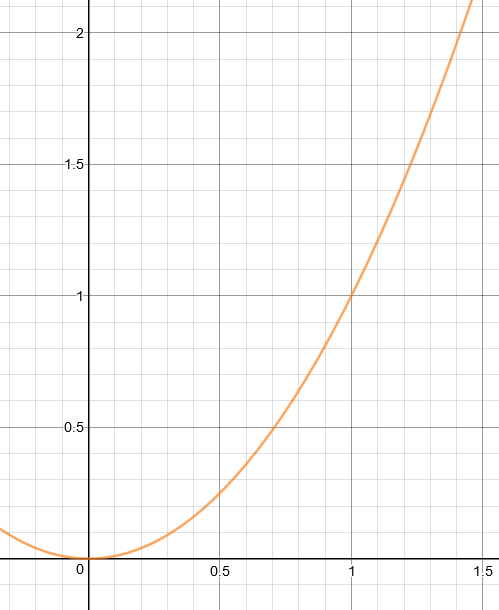


Fig. 7. Original x^2 curve

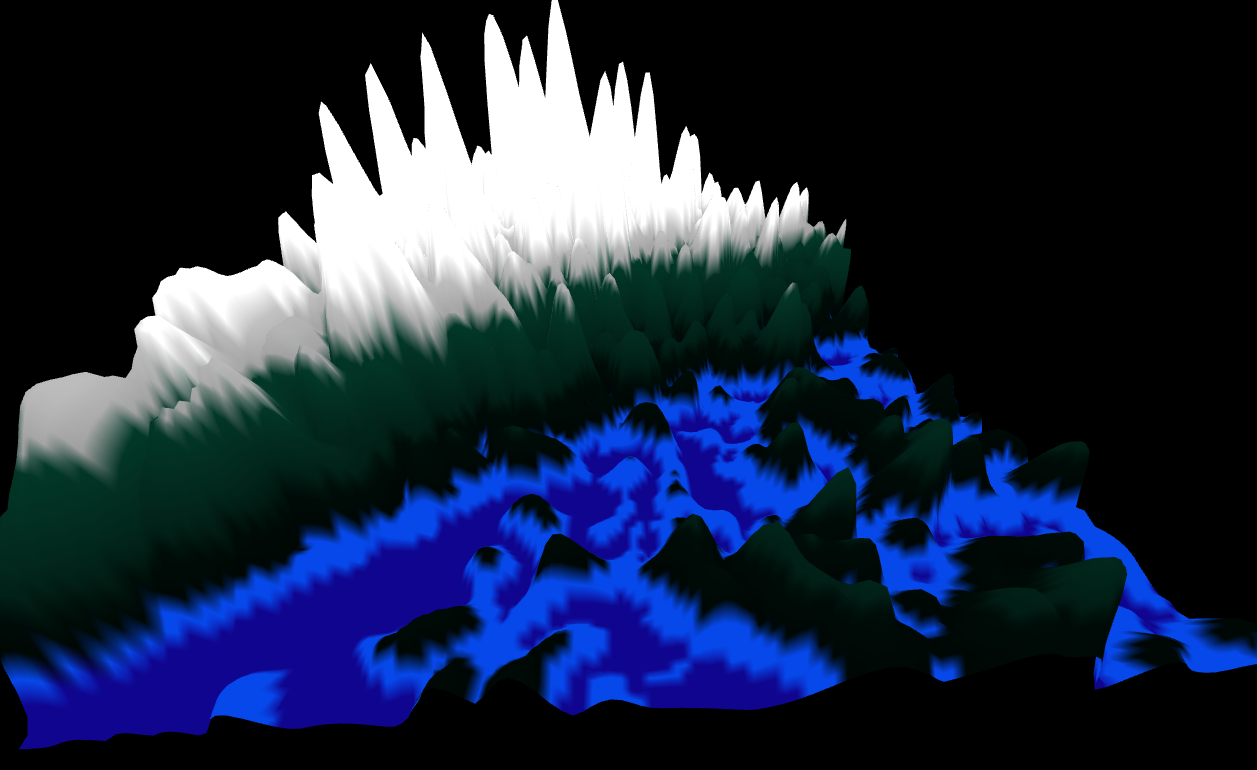


Fig. 8. Results of Tv^2

Observing application of the piecewise function:

Pow(y,4) if y <= 0.4

Sin(y) if y > 0.4

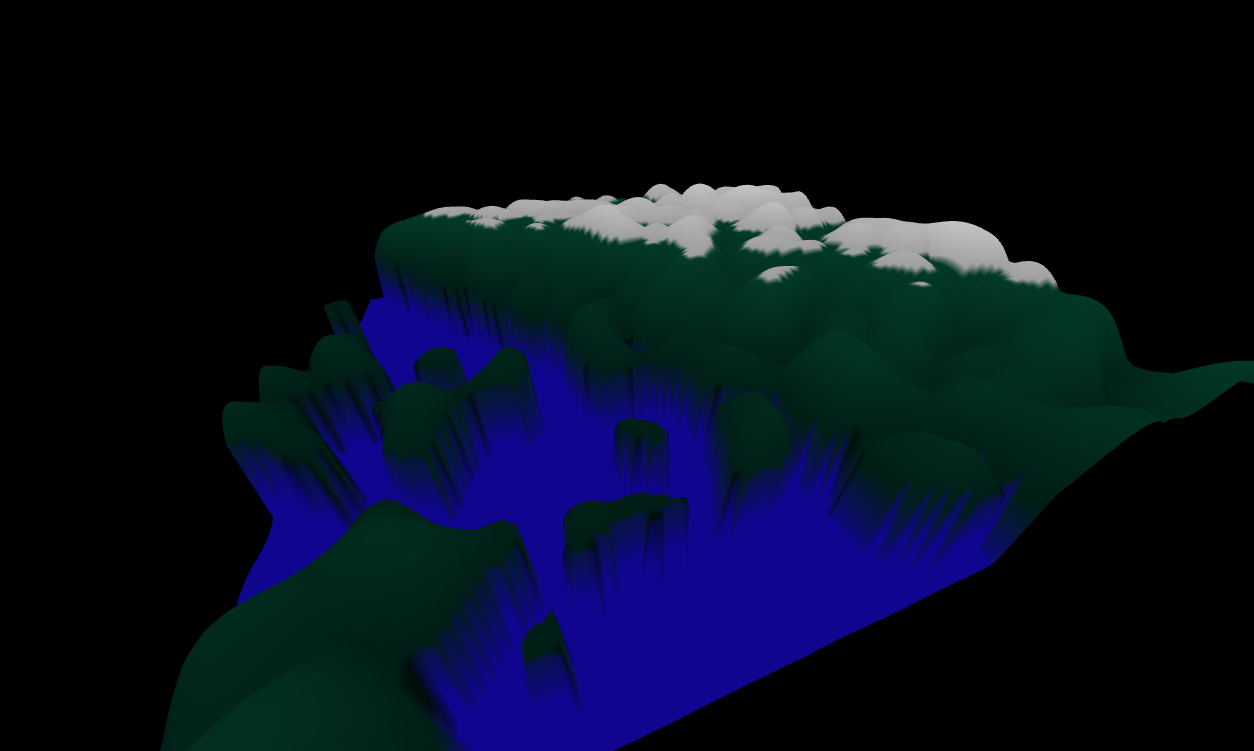


Fig. 9. Running Tv through piecewise function

Notice that sudden vertical change in height is due to no smooth transition between each function at 0.4? In other words, if Tv is 0.4 then you get 0.25, if Tv is 0.41 you get \approx 0.41. So, going from 0.16 to 0.41 is quite a high height difference in change with two adjacent vertices.

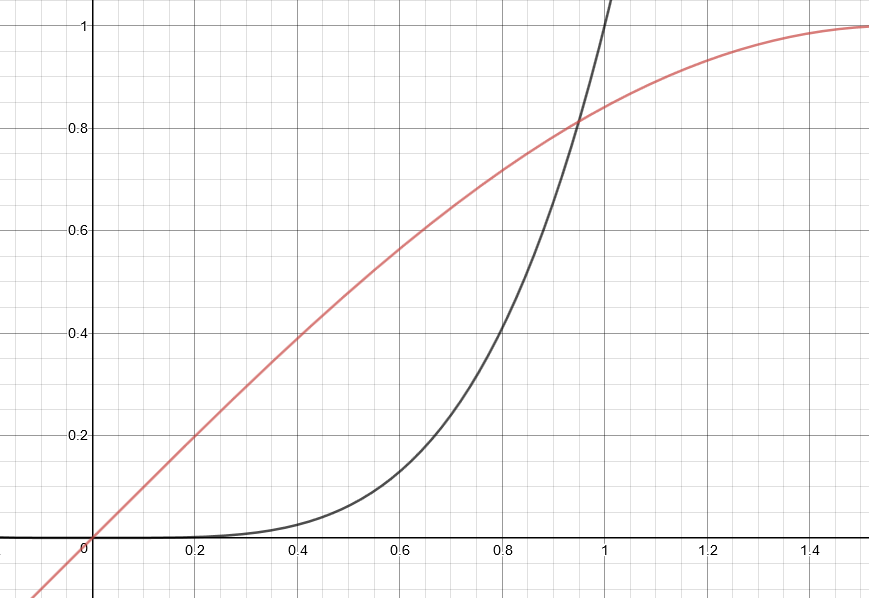


Fig. 10. Sin(Tv), red curve, and Tv^4, black curve

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 many more effects as well, such as applying randomly disperse dead grass, moisture droplets, water ice, and so on (not covered in this paper).

ACKNOWLEDGMENTS

This work was partially supported by the MIUR-PRIN 2010--11 Project 2010ECA8P3 “DyNanoMag” and by the National Research Foundation, Prime Minister's office, Singapore under its Competitive Research Programme (CRP Award No. NRF-CRP 10-2012-03).

A HEADINGS IN APPENDICES

The rules about hierarchical headings discussed above for the body of the article are di.erent in the appendices. In the appendix environment, the command section is used to indicate the start of each Appendix, with alphabetic order designation (i.e., the first is A, the second B, etc.) and a title (if you include one). So, if you need hierarchical structure within an Appendix, start with subsection as the highest level. Here is an outline of the body of this document in Appendix-appropriate form:

REFERENCES

|  |  |
| --- | --- |
| [1] | Patricia S. Abril and Robert Plant. 2007. The patent holder’s dilemma: Buy, sell, or troll? *Commun. ACM* 50, 1 (Jan. 2007), 36–44. DOI: http://dx.doi.org/10.1145/1188913.1188915 |
| [2] | I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. 2002. Wireless Sensor Networks: A Survey. *Comm. ACM* 38, 4 (2002), 393–422. |
| [3] | David A. Anisi. 2003. *Optimal Motion Control of a Ground Vehicle*. Master’s thesis. Royal Institute of Technology (KTH), Stockholm, Sweden. |
| [4] | P. Bahl, R. Chancre, and J. Dungeon. 2004. SSCH: Slo.ed Seeded Channel Hopping for Capacity Improvement in IEEE 802.11 Ad-Hoc Wireless Networks. In *Proceeding of the 10th International Conference on Mobile Computing and Networking* (MobiCom’04). ACM, New York, NY, 112–117. |
| [5] | Kenneth L. Clarkson. 1985. *Algorithms for Closest-Point Problems (Computational Geometry)*. Ph.D. Dissertation. Stanford University, Palo Alto, CA. UMI Order Number: AAT 8506171. |
| [6] | Jacques Cohen (Ed.). 1996. Special Issue: Digital Libraries. *Commun. ACM* 39, 11 (Nov. 1996). |
| [7] | Bruce P. Douglass. 1998. Statecarts in use: structured analysis and object-orientation. In *Lectures on Embedded Systems*, Grzegorz Rozenberg and Frits W. Vaandrager (Eds.). Lecture Notes in Computer Science, Vol. 1494. Springer-Verlag, London, 368–394. DOI:http://dx.doi.org/10.1145/3-540-65193-429 |
| [8] | Ian Editor (Ed.). 2008. *The title of book two* (2nd. ed.). University of Chicago Press, Chicago, Chapter 100. DOI: http://dx.doi.org/10.1145/3-540-09237-4 |

Received February 2007; revised March 2009; accepted June 2009

Journal Name:ACM Trans. Graph.

Volume:4

Issue:2

Year:2017

Month:October

Article Seq:XXXX

DOI:124564

Copyright Year:2017

Copyright Statement:rightsretained

ISSN:0730-0301

Price:$15.00