Desert Generation and Mapping using Improved Perlin Noise

A work by

NOGUEIRA Anthony

Inspired by the works of

Kenneth H. Perlin

and

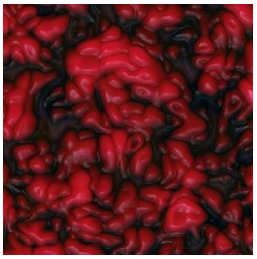
thatgamecompany

Introduction

Kenneth H. Perlin is a professor in the Department of Computer Science at New York University, founding director of the Media Research Lab at NYU, director of the Future Reality Lab at NYU, and the Director of the Games for Learning Institute.

In 1983, he developed the Perlin Noise, a procedural texture primitive, that is, a type of gradient noise, which main goal is to enhance the appearance of realism in computer graphics.

Two-dimensional slice through 3D Perlin noise at z=0



An organic surface generated with Perlin noise

Therefore, this technique is used for example to render biological, natural textures, such as lava, blood plasma, water, …, or even to generate natural-looking terrain, like in Minecraft.

On the other hand, thatgamecompany is an American independent video game development company founded by University of Southern California students Jenova Chen and Kellee Santiago in 2006. They are notorious for the games Flow, Flower and Journey.

In Journey, you control an anonymous character wandering through a huge and desertic world. The game developers actually made a talk in SIGGRAPH to explain how they rendered the sand, in order to recreate as much as possible, the feeling of an ocean of sand.

The desert in Journey (2012) by thatgamecompany

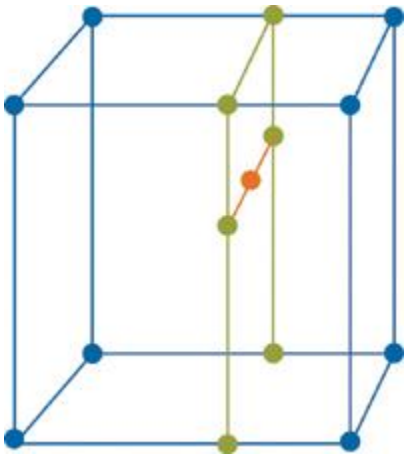
Therefore, I based the aforementioned work on those two incredible works to generate and render a realistic desert.

Improved Perlin Noise's algorithm

As said earlier, the purpose of the Perlin Noise function is to provide an efficiently implemented and repeatable, pseudo-random signal over a three-dimensional space that is band-limited (most of its energy is concentrated near one spatial frequency) and visually isotropic (statistically rotation-invariant).

The initial implementation, defined noise at any point (x, y, z) by using the following algorithm:

1. At each point in space (i, j, k) that has integer coordinates, assign a value of zero and a pseudo-random gradient that is hashed from (i, j, k).
2. Define the coordinates of (x, y, z) as an integer value plus a fractional remainder: (x, y, z) = (i + u, j + v, k + w). Consider the eight corners of the unit cube surrounding this point: (i, j, k), (i + 1, j, k), . . . (i + 1, j + 1, k + 1).
3. Fit a Hermite spline through these eight points, and evaluate this spline at (x, y, z), using u, v, and w as interpolants. If we use a table lookup to predefine the Hermite cubic blending function 3t 2 - 2t 2, then this interpolation requires only seven scalar linear interpolations: for example, four in x, followed by two in y, followed by one in z, as shown in Figure 5-2.



Steps Required to Interpolate a Value from Eight Samples in a Regular 3D Lattice

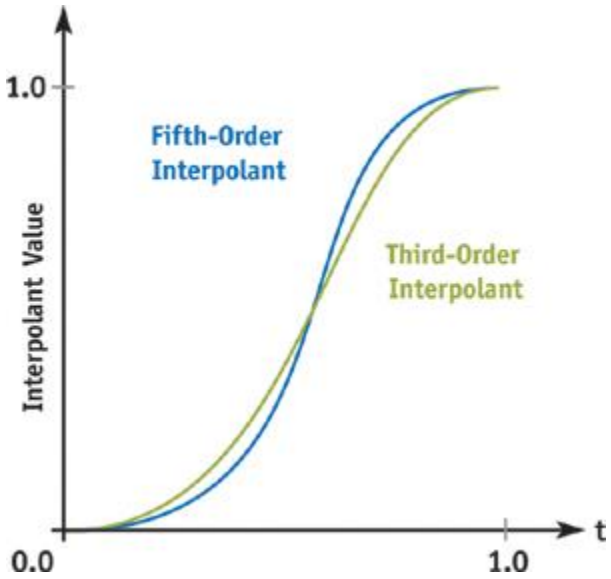
Unfortunately, the Perlin Noise in this had specific deficiencies:

- The choice of interpolating function in each dimension was $3t^2 - 2t^3$, which contains nonzero values in its second derivative. This can cause visual artifacts when the derivative of noise is taken, such as when doing bump mapping.

- The hashing of gradients from (i, j, k) produced a lookup into a pseudo-random set of 256 gradient vectors sown on a 3D sphere; irregularities in this distribution produce unwanted higher frequencies in the resulting Noise function.

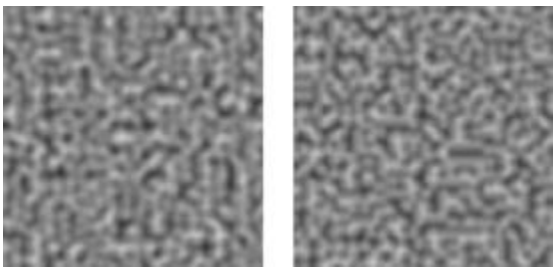
Therefore, Ken Perlin decided to improve his Perlin Noise in two specific areas: the nature of the interpolant and the nature of the field of pseudo-random gradients.

First of all, to improve the interpolant, he removed second-order discontinuities by switching to the fifth-degree interpolator: $6t^5 - 15t^4 + 10t^3$, which has both first and second derivatives of zero at both t = 0 and t = 1, removing the artifacts that were showing up when the noise was used in bump mapping. Because the effect of lighting upon a bump-mapped surface varies with the derivative of the corresponding height function, second-derivative discontinuities are visible.



Interpolation curves of the different interpolators used in the Perlin Noise

The other improvement was to replace the 256 pseudo-random gradients with just 12 pseudo-random gradients, consisting of the edge centers of a cube centered at the origin: (0, ±1, ±1), (±1, 0, ±1), and (±1, ±1, 0). This results in a much less "splotchy-looking" distribution.



Visualization of the Gradient Distribution Improvement

The reason for this improvement is that none of the 12 gradient directions is too near any others, so they will never bunch up too closely. It is this unwanted bunching up of adjacent gradients that causes the splotchy appearance in parts of the original Noise implementation. Furthermore, it makes it possible to avoid many of the multiplies associated with evaluating the gradient function.

In order to make the process of hashing into this set of gradients compatible with the fastest possible hardware implementation, 16 gradients are actually defined, so that the hashing function can simply return a random 4-bit value. The extra four gradients simply repeat (1, 1, 0), (-1, 1, 0), (0, -1, 1), and (0, -1, -1).

Now that Improved Perlin Noise's algorithm is explained, I will talk about the way it was implemented and used in the desert generation and mapping.

The algorithms and effects implemented

1. Desert generation using Improved Perlin Noise

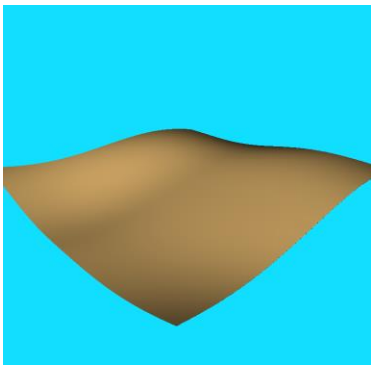
   Improved Perlin Noise allows us to generate terrains of different shapes. To do so, we first implement a PerlinNoise class, that is a C++ implementation of Kenneth H. Perlin's Java implementation, with the only difference that the permutation values used during the algorithm are generated randomly in order to increase the pseudo-random factor of the gradient noise.

```
public final class ImprovedNoise {
   static public double noise(double x, double y, double z) {
      int X = (int)Math.floor(x) & 255,                  // FIND UNIT CUBE THAT
          Y = (int)Math.floor(y) & 255,                  // CONTAINS POINT.
          Z = (int)Math.floor(z) & 255;
      x -= Math.floor(x);                                // FIND RELATIVE X,Y,Z
      y -= Math.floor(y);                                // OF POINT IN CUBE.
      z -= Math.floor(z);
      double u = fade(x),                                // COMPUTE FADE CURVES
             v = fade(y),                                // FOR EACH OF X,Y,Z.
             w = fade(z);
      int A = p[X  ]+Y, AA = p[A]+Z, AB = p[A+1]+Z,      // HASH COORDINATES OF
          B = p[X+1]+Y, BA = p[B]+Z, BB = p[B+1]+Z;      // THE 8 CUBE CORNERS,

      return lerp(w, lerp(v, lerp(u, grad(p[AA  ], x  , y  , z   ),  // AND ADD
                                     grad(p[BA  ], x-1, y  , z   )), // BLENDED
                             lerp(u, grad(p[AB  ], x  , y-1, z   ),  // RESULTS
                                     grad(p[BB  ], x-1, y-1, z   ))),// FROM  8
                     lerp(v, lerp(u, grad(p[AA+1], x  , y  , z-1 ),  // CORNERS
                                     grad(p[BA+1], x-1, y  , z-1 )), // OF CUBE
                             lerp(u, grad(p[AB+1], x  , y-1, z-1 ),
                                     grad(p[BB+1], x-1, y-1, z-1 ))));
   }
   static double fade(double t) { return t * t * t * (t * (t * 6 - 15) + 10); }
   static double lerp(double t, double a, double b) { return a + t * (b - a); }
   static double grad(int hash, double x, double y, double z) {
      int h = hash & 15;                      // CONVERT LO 4 BITS OF HASH CODE
      double u = h<8 ? x : y,                 // INTO 12 GRADIENT DIRECTIONS.
             v = h<4 ? y : h==12||h==14 ? x : z;
      return ((h&1) == 0 ? u : -u) + ((h&2) == 0 ? v : -v);
   }
   static final int p[] = new int[512], permutation[] = { 151,160,137,91,90,15,
   131,13,201,95,96,53,194,233,7,225,140,36,103,30,69,142,8,99,37,240,21,10,23,
   190, 6,148,247,120,234,75,0,26,197,62,94,252,219,203,117,35,11,32,57,177,33,
   88,237,149,56,87,174,20,125,136,171,168, 68,175,74,165,71,134,139,48,27,166,
   77,146,158,231,83,111,229,122,60,211,133,230,220,105,92,41,55,46,245,40,244,
   102,143,54, 65,25,63,161, 1,216,80,73,209,76,132,187,208, 89,18,169,200,196,
   135,130,116,188,159,86,164,100,109,198,173,186, 3,64,52,217,226,250,124,123,
   5,202,38,147,118,126,255,82,85,212,207,206,59,227,47,16,58,17,182,189,28,42,
   223,183,170,213,119,248,152, 2,44,154,163, 70,221,153,101,155,167, 43,172,9,
   129,22,39,253, 19,98,108,110,79,113,224,232,178,185, 112,104,218,246,97,228,
   251,34,242,193,238,210,144,12,191,179,162,241, 81,51,145,235,249,14,239,107,
   49,192,214, 31,181,199,106,157,184, 84,204,176,115,121,50,45,127, 4,150,254,
   138,236,205,93,222,114,67,29,24,72,243,141,128,195,78,66,215,61,156,180
   };
   static { for (int i=0; i < 256 ; i++) p[256+i] = p[i] = permutation[i]; }
}
```

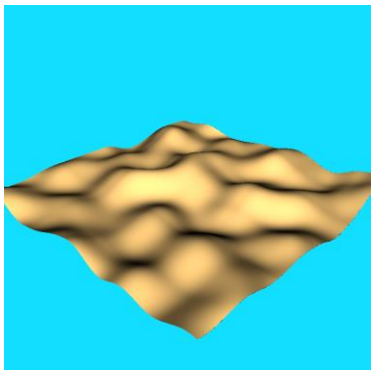
Java reference implementation of Improved Perlin Noise – Copyright 2002 Ken Perlin

At this point, for each computed point (in the order of width * height * offset), we send its x & y coordinates transformed at the x & z parameters of the Perlin Noise function. The given result will be between -1 and 1, and will be used to calculate the y-coordinate of the computed point. The coordinates transformation goes as follow: we take the world-based coordinate, we divide it by the desert's size and we multiply the result by a randomly selected scale between 2 and 3. That scale value influence the pseudo-randomly generated frequency waves of the noise. That is, the

bigger the scale is, the shorter and the clamper the dunes are. The chosen values are 2 and 3 because they both give the best-looking dunes.



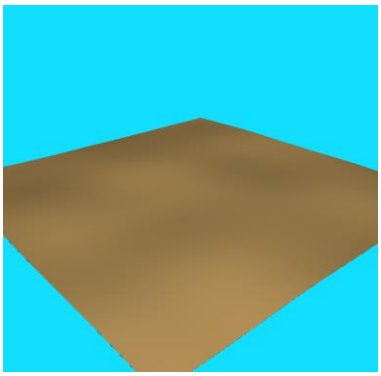
Desert generation with a scale of 1 (left) and 5 (right)



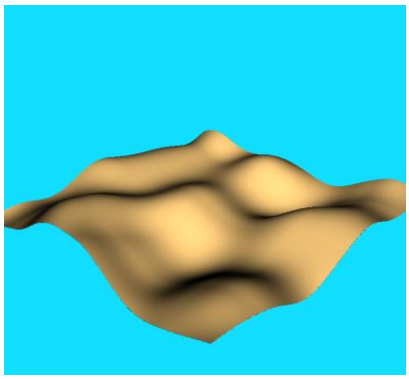
Perlin Noise's calculation

```
float noiseValue = noise((xCoord / static_cast<float>(size_)) * scale, 0, (yCoord / static_cast<float>(size_)) * scale) / scale;
```

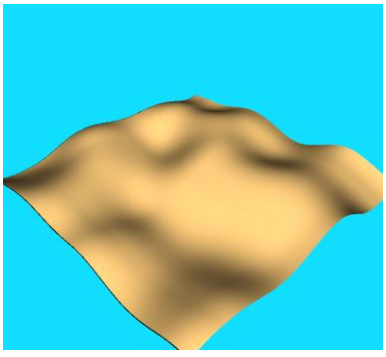
Now that we have our value given by the Perlin Noise, the desert is simulated by multiplying this value to the desert maximum height, given by the user in the console. The height I personally recommend for pretty and well-rounded dunes is around 75.



Desert generation with a size of 10 (left) and 100 (right)

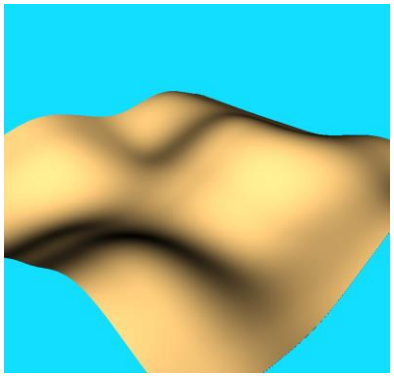


Lastly, the points are calculated based on the desert' size, picked randomly between 200 & 400 (a desert too big will saturate the memory), and an offset used to increase the precision of a unit, therefore the precision of the Perlin Noise application and terrain detailing afterwards.



Desert generation with an offset of 1 (left) and 10 (right)

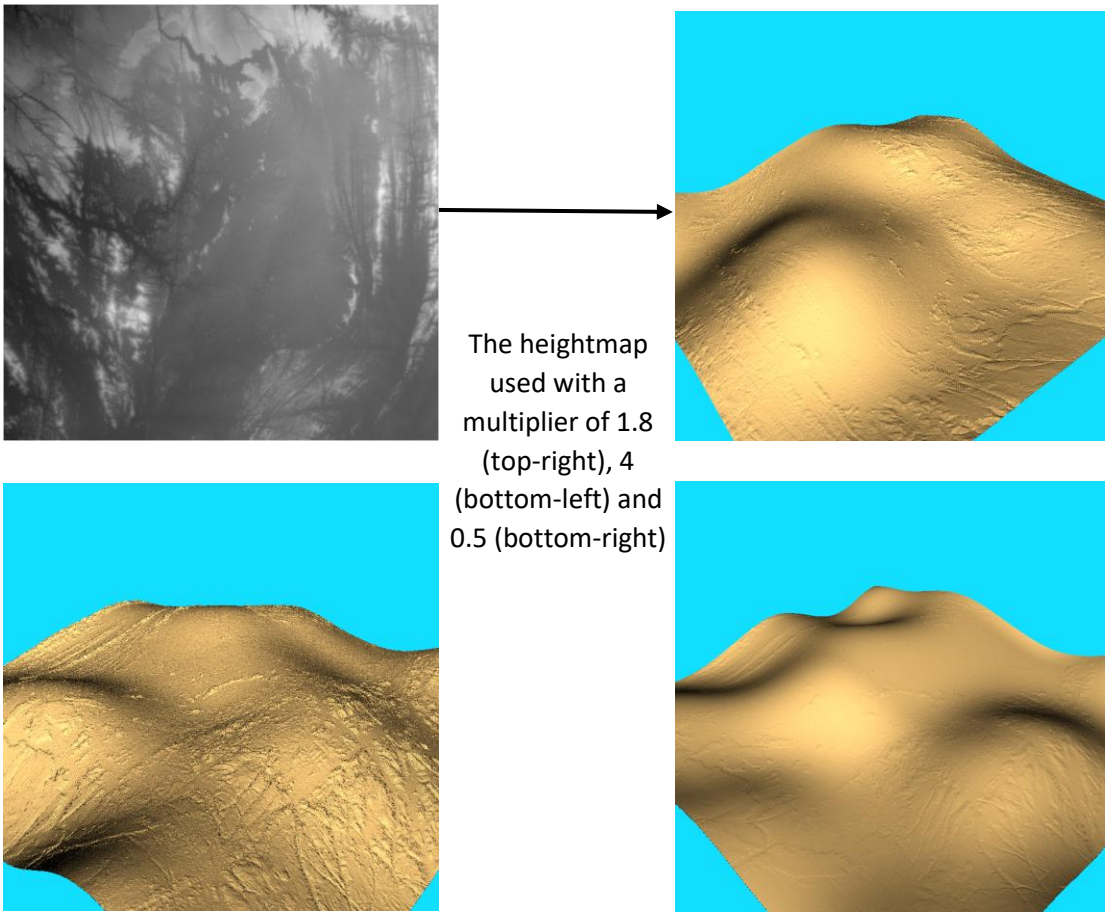
(height at 75)



2. Detailing the desert by using height mapping and simple noise

Now that our terrain is generated, we need to make it look like a real desert. To do so, we'll add a bump to each vertex by combining two different techniques, which are height mapping and another Perlin Noise.

Height mapping is a technique that uses a gray scaled texture to alter the height and normal of a material. It adds details by directly modifying the material and not just the normal. To do so, we'll use a height map taken by the blogger Rob Green using terrain.party, a website that allows us to generate a height map from a given area on Earth (here, an area on the Sahara's desert). Now that we have our height map, for each computed point in our terrain, we'll take the corresponding pixel in the loaded height map and use the red channel to gather the bump in the y-coordinate that need to be applied to our point. We then multiply that value between 0 and 1 by an arbitrary value (in the final result, this value is equal to 1.8, because it looks the best), and we add the resulting value to the previous y-coordinate calculated using Perlin Noise.



The heightmap used with a multiplier of 1.8 (top-right), 4 (bottom-left) and 0.5 (bottom-right)

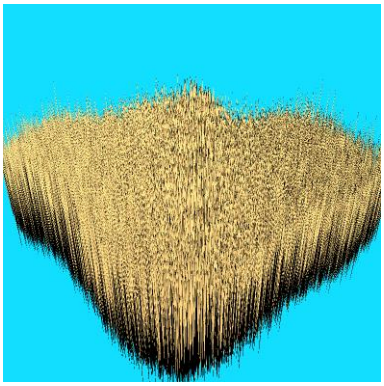
But we're not using the height map to calculate the normal of the computed point since we're adding to an already existing height. We'll have to compute the normal in another way that we'll see later on.

The second technique we use is to add a very slight and ultra-compressed Perlin Noise to the previous result to simulate the slight waves of an ocean of wave. To do so, we take the exact same

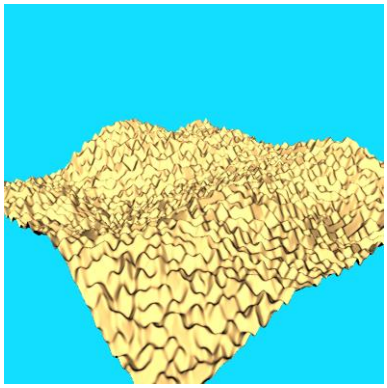
Perlin Noise calculation to generate the terrain, with the only difference that we multiply an arbitrary high number to the scale.

Compressed Perlin Noise's calculation

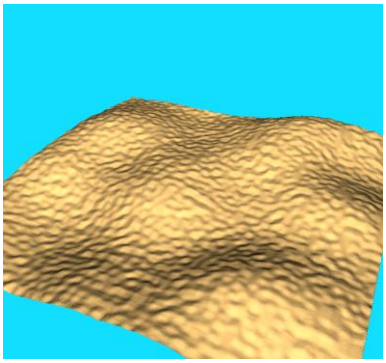
```
float noise_bump = noise((xCoord / static_cast<float>(size_)) * scale * 15, 1, (yCoord / static_cast<float>(size_)) * scale * 15) / scale * -15;
```



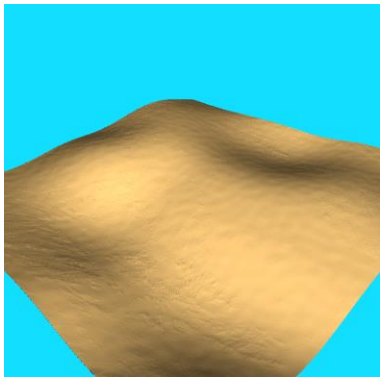
Desert generation with additional Perlin Noise's layer with a scale multiplier of 150 (left) and 15 (right)



Now that we have our new overly compressed Perlin Noise's value, we'll multiply it by a very small number to attenuate the variation and add it to the previous y-coordinate. Remember, we only want to simulate slight waves.



Desert generation after multiplying Perlin Noise's additional value by 0.25 (left) and 0.025 (right)



Therefore, in the final result, we'll keep an offset of 10, a height bump multiplier of 1.8, a scale multiplier of 15 and an additional multiplier of 0.025.

3. Desert's normal generation

The terrain's generation and mapping are completely finished. From now on, we'll talk about the different rendering techniques we use in our desert, but to start everything, we need to calculate the normal.

The normal calculation is very straightforward since we can access every points' coordinates: we calculate the dot product at the given point.

```
glPos3D vec1 = vertices_[(nb_pts * (!y ? (y + 1) : (y - 1)) + x)] - vertices_[(nb_pts * y + x)];
glPos3D vec2 = vertices_[(nb_pts * y + (!x ? (x + 1) : (x - 1)))] - vertices_[(nb_pts * y + x)];

normals_[y * nb_pts + x] = glPos3D{
    vec1.y * vec2.z - vec1.z * vec2.y,
    vec1.z * vec2.x - vec1.x * vec2.z,
    vec1.x * vec2.y - vec1.y * vec2.x
};
```

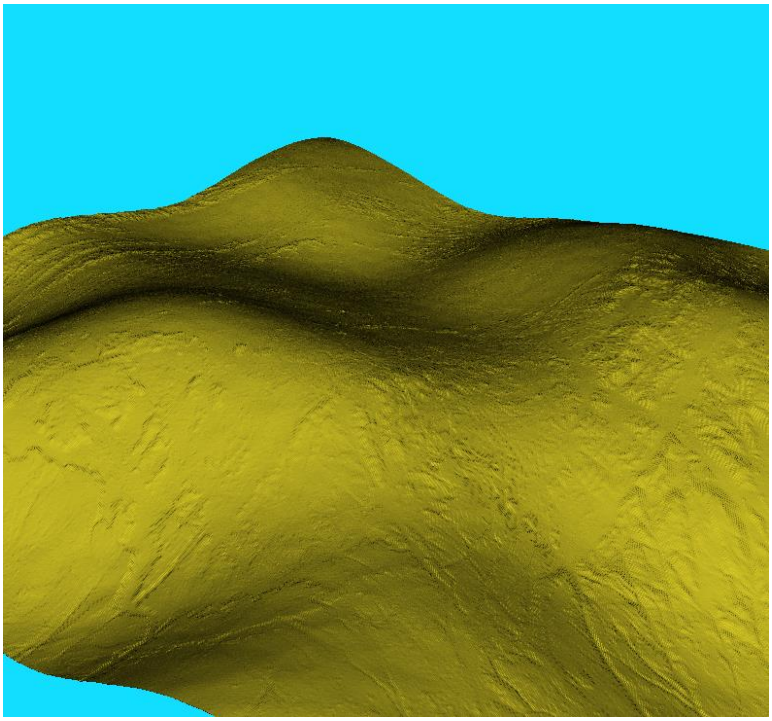
4. Lambert Shader and diffuse contrast

Now that we have the normal for each point, we can apply a simple shader at our starting point for our next rendering effects. Here, we opted for a Lambert shader, since Blinn-Phong shader pollutes the scene with too much specular.

Lambert shader is very simple; we just calculate the dot product of the normal and the position vectors, as follow:

```
vec3 Lambert(vec3 position, vec3 norm)
{
  vec3 nrmN = normalize(-norm);
  vec3 nrmL = normalize(position);
  float result = dot(nrmN, nrmL);

  return high_end_col * max(result, 0.0);
}
```

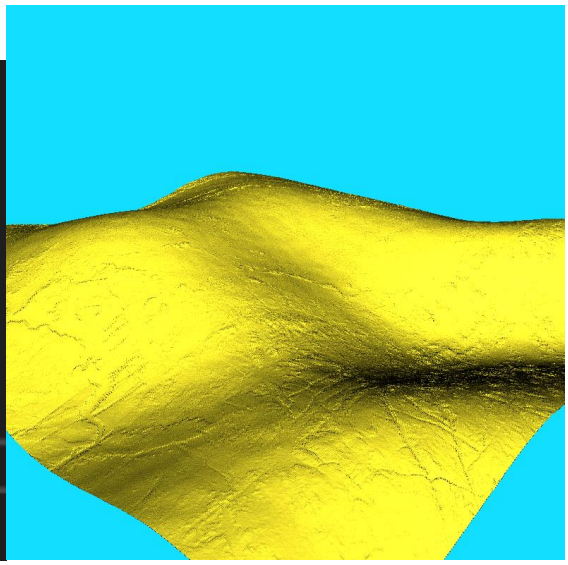
But, one huge downside of the Lambert shader is due to its simplicity, that is, it's way too plain.



To correct that, thatgamecompany tweaked some values by multiply the y-coordinate of the normal vector by a small arbitrary number and multiplying the dot product by another number, with the only purpose that it looks better. The purpose was to simulate parts of another shader called Oren-Nayar, and they called this hack Diffuse Contrast.

After a few tests on my side, we came with the following numbers:

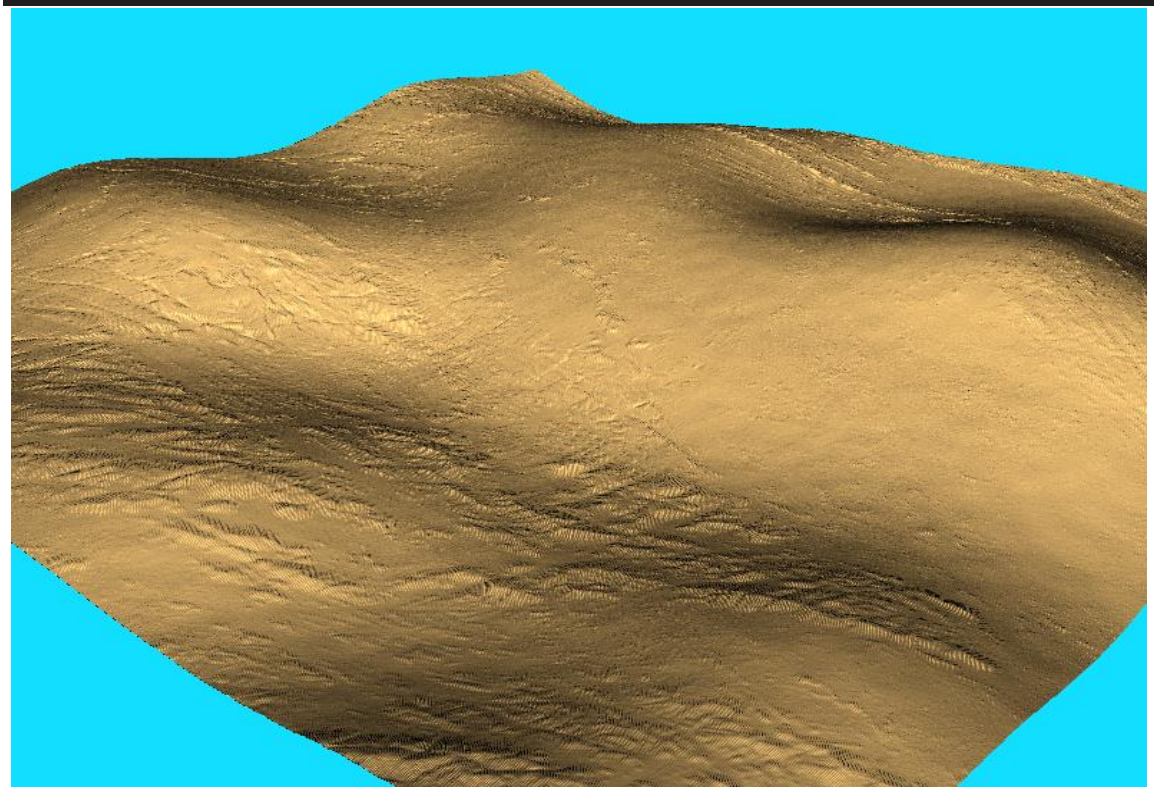
```
vec3 Lambert(vec3 position, vec3 norm)
{
  vec3 nrmN = normalize(-norm);
  nrmN.y *= 0.3;
  vec3 nrmL = normalize(position);
  float result = 2 * dot(nrmN, nrmL);

  return high_end_col * max(result, 0.0);
}
```



As you can see, it is way brighter, smoother and livelier.

On the other side, we decided to simulate the grainy aspect of the sand, not by applying a noise texture, just like thatgamecompany, but by altering the color modified by the Lambert shader by applying a mix between a low end color and a high end color, both chosen from their individual aspects, where the mix ratio is a simple noise value.

```
return mix(low_end_color, high_end_col, Noise) * max(result, 0.0);
```



We can see the small grains through the sand and the way the desert's color feels smoother and more harmonious.

5. Ocean specular

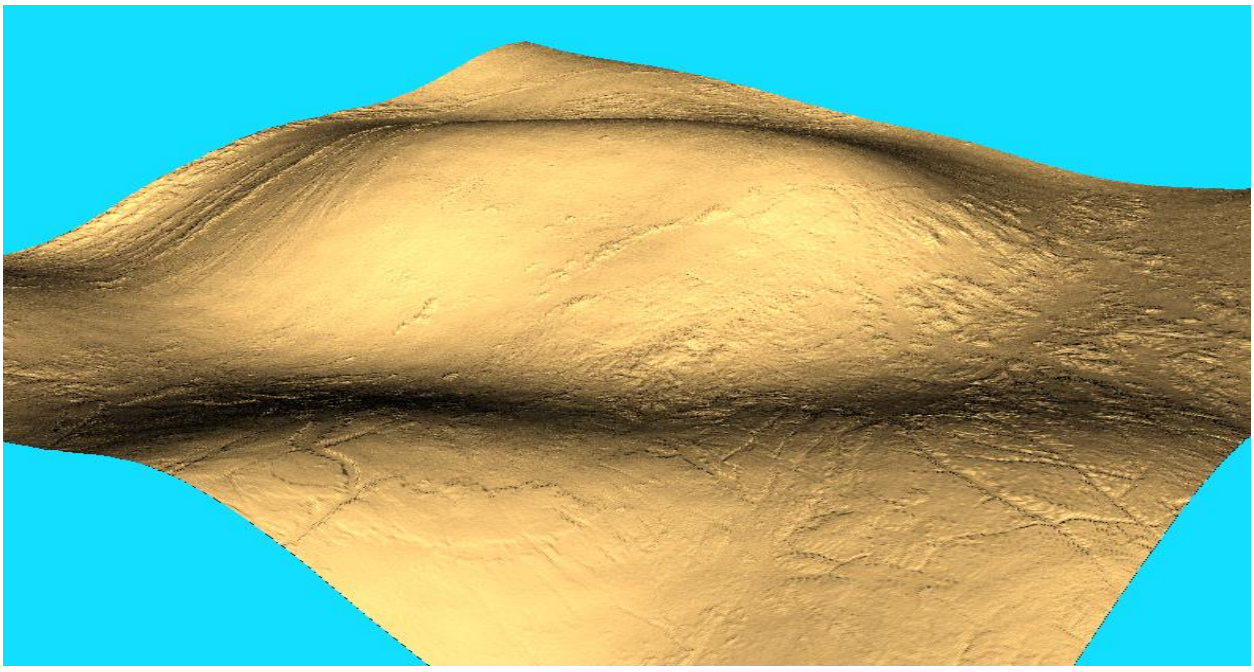
Now that we have our basic color rendered, we just have to add new computed values to it in order to add new rendering effects, and we'll start with ocean specular.

Adding ocean specular to the sand's rendering is an idea of thatgamecompany to really translate this feeling of ocean of sand. To do so, we based our work on a blog post from Yann_A, and the ocean specular goes as the following:

- First, we calculate the light reflection on the surface.
- Then, we calculate the specular by doing the dot product of the reflection and the camera's view direction.
- Finally, we return the specular to the power of the material's shininess.

```
float ocean_specular(vec3 position, vec3 normal)
{
    vec3 lightVec = normalize(position - Light.Position);
    vec3 reflection = reflect(lightVec, normal);

    float specular = dot(normalize(reflection), -normalize(Camera.ViewDirection));

    float specularShininess = 10.0f;

    if(specular > 0.0f)
        specular = pow(specular, specularShininess);

    return specular * 0.8f;
}
```

As you can see, we tweaked the shininess and checked the result, and the values above are what gave us the best result. We also decided to weaken the result because the wave effect was way too strong.



The result feel shinier and smoother, almost like an ocean wave stopped in time.

6. Glitter specular

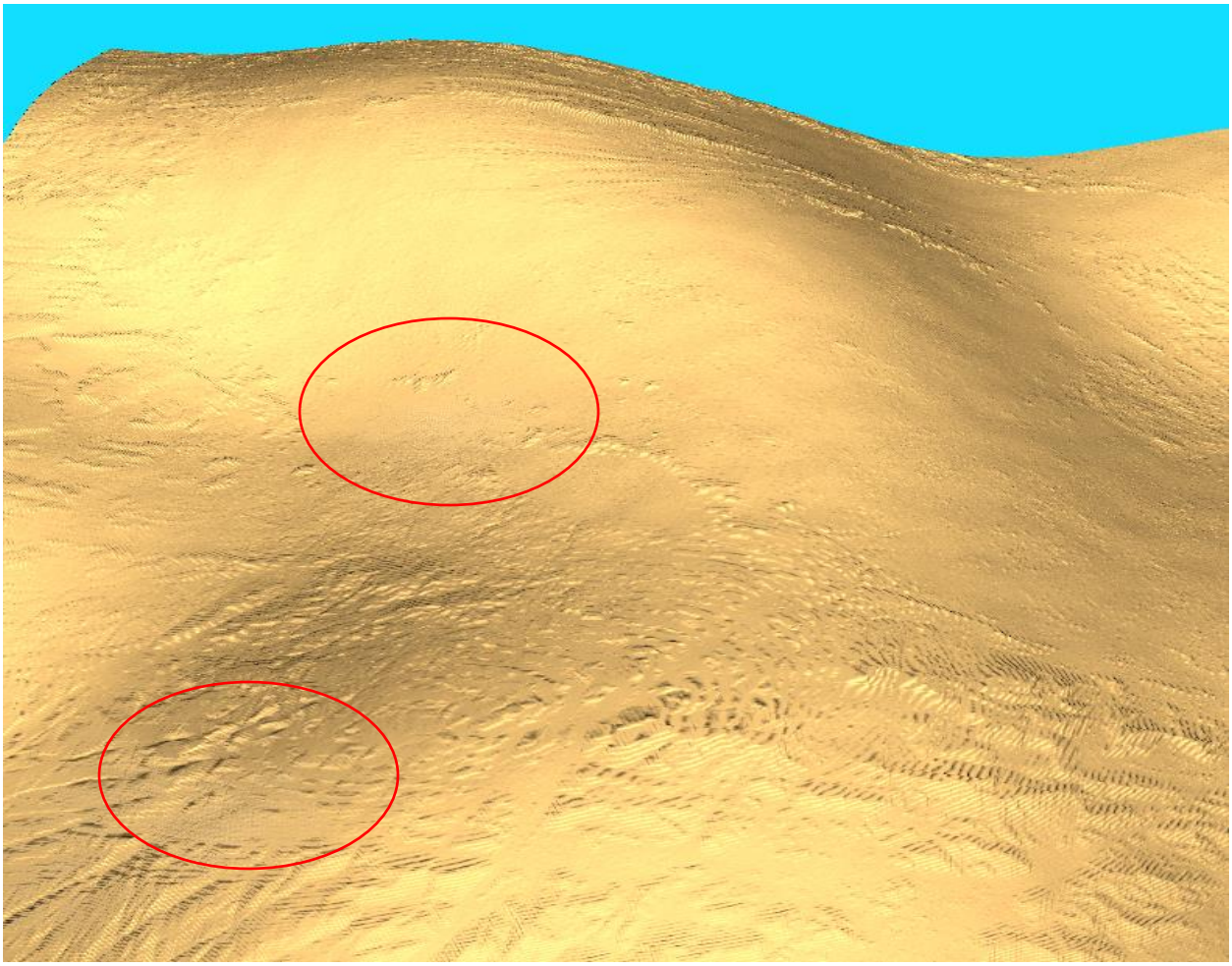
The final rendering effect we'll add is the glitter specular. It's used to add a sparkling effect to the rendered material, and we want to use it to simulate the sun waves hitting the multiple grains of the sand.

To implement it, we'll use a paper by AMD that uses the glitter effect to simulate the sparkling of the snow. It goes as follow:

```
float glitter_specular(vec3 position, float specular)
{
    vec3 fp = fract(0.7 * position + 3 * noise(position * 0.04).r + 0.1 * Camera.ViewDirection);
    fp *= (1 - fp);
    float glitter = 1 - (fp.x + fp.y + fp.z);
    return glitter * pow(specular, 7);
}
```

Here, AMD uses the 3D position of the point to index a 3D noise function (provided by Tom Minor in his GitHub), add the view vector to the noise value, and use fractals to further randomize the result. Here, the specular value given is the one returned by the ocean specular.

Since the values given by AMD generated a sparkling way too strong, since they used a basic specular to render snow, we tweaked some values and experimented until it gives us a nice and subtle sparkling effect.



You can take a glimpse of the glitter here, but it's more visible in movement.

7. Additional functionalities: Adding music

   As a very simple way to give our desert life and feeling, we decided to add a background music that reflected our state of mind when we think of deserts: serenity.

   To do so, we used a library called IrrKlang that allows us to easily trigger an audio file. The music used is Warm by A.L.I.S.O.N, that represents for me the serenity and the warmth of a desert or an oasis.

```
irrklang::ISoundEngine* engine = irrklang::createIrrKlangDevice();
if (!engine)
    return 1;

Window win{ (uint16_t)width, (uint16_t)height };

engine->play2D("../warm.ogg", true);
```

Sound initialization and trigger

The effects not implemented

Anisotropic highlight shading is a technique that allows us to stretch reflections and highlights in a direction that runs perpendicular to the grain or grooves in a surface.

Used for example in hair rendering, anisotropic highlight shading is enabled with the use of a specific anisotropic directional texture.

We then use the different color channels of the so-called texture to define the anisotropic direction, the specular and gloss level of the anisotropic shading, and the anisotropic mask, used to choose between anisotropic shading and simple Lambert shading for each point.

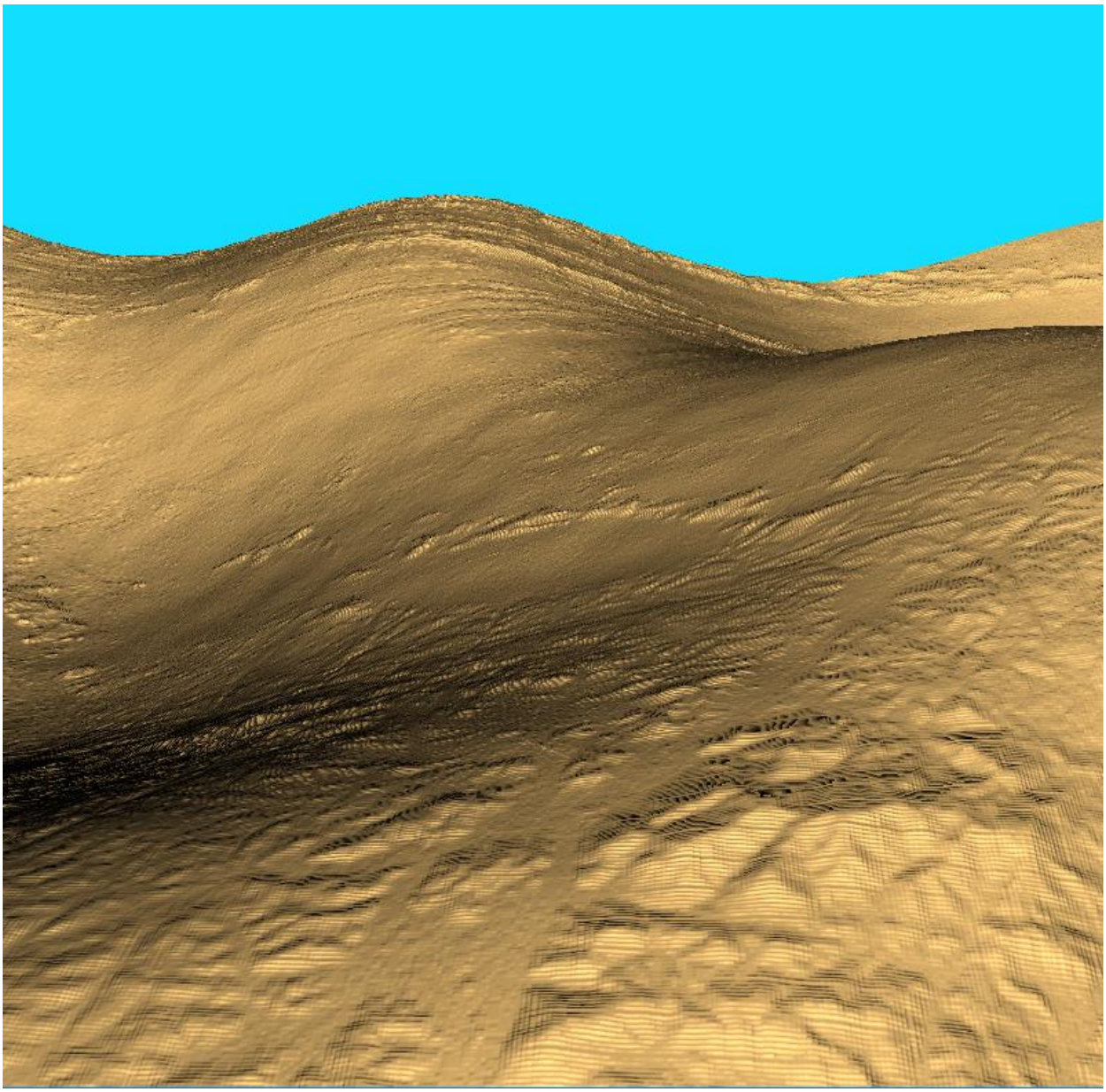
Anisotropic directional texture mimicking brushed metal

Sadly, the implementation of this feature requires a hand-made texture defining all the data necessary to apply to a procedurally generated desert.

Therefore, it was decided to withdraw this feature from the final project.

Conclusion

Improved Perlin Noise allows us to generate naturally-looking terrain of any type, while the awesome work of thatgamecompany transforms a plain procedurally generated terrain into a shiny, pseudo-realistic desert using different rendering techniques.



Ressources

Understanding Perlin Noise: https://flafla2.github.io/2014/08/09/perlinnoise.html

Implementing Improved Perlin Noise:
https://developer.nvidia.com/gpugems/GPUGems/gpugems_ch05.html

Java implementation for Improved Perlin Noise: https://mrl.nyu.edu/~perlin/noise/

Basic Procedural Desert Generation and Rendering in Unity:
https://www.wiseshards.com/blog/basic-procedural-desert-generation-and-rendering-in-unity/

Sand Rendering in Journey Course by ThatGameCompany:
http://advances.realtimerendering.com/s2012/thatgamecompany/SandRenderingInJourney_thatgamecompany.pptx

Lambert shader GLSL: https://gist.github.com/TomMinor/088766855a5fb161e236

Tutorial for implementing anisotropic highlight shader:
https://wiki.unity3d.com/index.php/Anisotropic_Highlight_Shader

GLSL functions to generate multiple dimensional noise and Perlin Noise:
https://gist.github.com/patriciogonzalezvivo/670c22f3966e662d2f83/

AMD's paper Getting' Procedural:
http://developer.amd.com/wordpress/media/2012/10/Shopf-Procedural.pdf

Good specular for water surface: https://www.gamedev.net/forums/topic/654663-good-specular-for-water-surface/

Modelling a desert ground plane using height mapping by Rob Green:
https://robgreenart.wordpress.com/2016/04/26/developments-modelling-desert-ground-plane/

Generating a normal map from a height map:
https://stackoverflow.com/questions/5281261/generating-a-normal-map-from-a-height-map

Graphic library used: **OpenGL4**

Music library used: **IrrKlang**