Perlin noise in Real-time Computer Graphics

Andrei Tatarinov
Department of Computational Mathematics and Cybernetics
Moscow State University, Moscow, Russia
atatarinov@graphics.cs.msu.ru

Abstract

In general case noise means unwanted signal of different nature. In computer graphics and image synthesis term “noise” is used to call a pseudo-random function which is generally used to generate procedural textures. Perlin noise is one of the most well-known noise functions[1].

Noise is generally used to create procedural textures, such as marble, wood, cloud textures. This requires noise to be a function of two parameters, which are texture coordinates.

However, if we extend noise function to three or four dimensions, we would get a powerful function which would allow us to modify geometry, generate procedural volumetric effects and vortex fields for particle simulations.

This paper shows how to compute noise function on GPU and how to use it in advanced real-time visual effects.

Keywords: *Noise, Perlin noise, GPU, landscape, fire, volumetric effects, ray marching, particle simulation, vortex fields.*

1. INTRODUCTION

This paper shows how GPU can be used for noise computation – a problem which is known to be very complex and thus expensive. Noise is commonly used to generate procedural textures and is rarely used to generate textures in real time due to complexity of computations.

However, due to its massively parallel architecture, GPU allows to compute several instances of noise function simultaneously, making it possible to compute noise functions in real-time. And not only two-dimensional functions can be computed, but three-dimensional and four-dimensional, opening up new possibilities for noise usage.

Originally noise (in that sense in which we consider it) was invented by Ken Perlin[2], who was searching for a procedural way of creating three-dimensional textures (i.e. volumes filled with texture). His goal was to be able to apply textures onto arbitrary objects without projecting these objects onto two-dimensional surfaces (traditional textures). His idea was to use object’s coordinates as texture coordinates into a volume filled with texture. However, Perlin realized that such texture would require a significant amount of data, so he was thinking of a possibility to generate such a texture procedurally. This is how he came to a function which is now commonly called Perlin noise.

One strong restriction for the desired function was that, in distinction to a natural noise, noise function should always return the same value if sampled with same coordinates. It means that it should be not truly random, but pseudorandom instead.

After some research Perlin came to a function known as a lattice noise with gradient variety. This function used a hypercubic grid with each grid knot being assigned its own pseudorandom gradient vector (Figure 1, a).

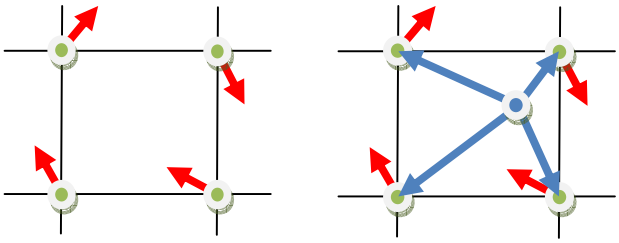

Figure 1: a. Grid points with assigned gradient vectors,

b. Vectors to hypercube vertices

Given coordinates *x*, *y*, *z* this function computes vectors from point defined by these coordinates to vertices of the hypercube where this point is situated (Figure 1, b).

Finally Perlin noise value in point x, y, z is computed as an interpolation of gradients – dot products between gradient vectors in hypercube vertices and vectors from given point to these vertices. Generally this cubic polynomial is used:

$$3x^2 - 2x^3$$

Turbulence function is used to sum up several frequencies of noise with corresponding weights to achieve desired look of noise function (Figure 3).

Turbulence function in its turn can be used in more complicated expressions to further modify noise look. This allows to create textures which would look like clouds, marble, wood or flame.

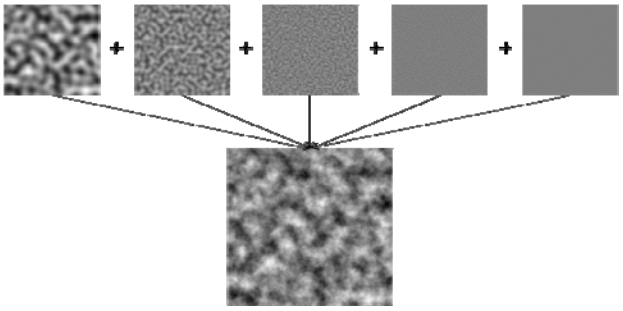

Figure 3: Putting together different noise frequencies

Later Perlin’s function was used to generate not only volumetric textures, but also two-dimensional images – fractal structure of noise and possibility to mix different noise frequencies allowed to create natural-looking patterns (wood, marble, different stones, clouds)[5]. A bunch of noise functions were obtained from Perlin noise. Ken Perlin himself suggested a number of valuable improvements to his function. First improvement was to use

simplex grid[3] instead of hypercube grid (it allowed a linear growth of computations instead of quadric growth when going to higher noise dimensions), and the second one was to use an interpolation curve of degree five instead of interpolation curve of degree three (this guaranteed noise smoothness).

However, even with these improvements, Perlin noise remained to be very expensive to compute. For a single noise sample, some amount of pseudorandom numbers should be generated, gradients computed and interpolated. Don't forget that you commonly need to compute several noise frequencies to generate a desired look of noise function.

However, with modern GPUs it becomes possible to compute noise functions in real-time. This gives a lot of advantages and allows to use full power of noise functions. Now it is possible to sample three-dimensional volumetric textures instead of precomputing and storing them somewhere, it is possible to compute vector fields using noise (this means three times more computations per sample, since you want to compute a three-dimensional vector), it is even possible to compute noise which dynamically changes (four-dimensional noise, where fourth dimension stands for time).

This paper is divided into four chapters. First chapter covers the details of noise computation implementation on GPU, talking about general idea of noise, noise computation and GPU-specific optimization tricks. Second part talks about using noise to modify geometry, which allows dynamically adding details to low-poly meshes, such as landscapes. Finally, third and fourth chapters talk about fire effect as an example of using noise to generate volumetric effects.

2. COMPUTING NOISE

Let's consider implementing simplex noise on GPU. Simplex noise (Perlin noise which is computed on a simplex grid instead of hypercube grid) is the most suitable for us since its computational complexity scales linearly with growth of number of dimensions. In general simplex is a figure which has minimum vertices in a given space, can't be represented in space with lesser number of dimensions, and can be replicated to pitch the entire space. In two-dimensional space it is a triangle, in three-dimensional space it is a tetrahedron. It is hard to imagine what it would look like in four-dimensional space, but at least it is guaranteed that this figure would have only 5 vertices instead of 16 vertices in a case of 4D-hypercube.

Translating simplex Perlin noise computation algorithm to GPU is pretty straightforward, since modern shader model (shader model 4.0 which is supported in DirectX 10) is very similar to what traditional programming languages have in terms of available operations.

Let's consider two basic routines. First one is a routine which computes a simplex in a hypercube in which a given point is situated. This routine uses magnitude sorting. This is how this routine can look like in three-dimensional case (the code is written using HLSL 4.0):

```
void Simplex3D( const in float3 P, out
float3 simplex[4] )
{
    float3 T = P.xzy >= P.yxz;
    simplex[0] = 0;
    simplex[1] = T.xzy > T.yxz;
    simplex[2] = T.yxz <= T.xzy;
    simplex[3] = 1;
}
```

As an output this routine computes four vertices which are vertices of a simplex (tetrahedron) in which given point is situated. These vertices are later used to compute gradients and interpolate them.

But first gradient vectors should be associated with vertices of this simplex. This is commonly done with a hash function which computes an index to a table of precomputed gradient vectors. On CPU this hash function is implemented with a number of multiplications and taking a remainder of division on some big prime number.

However, on GPU this approach is working quite slow due to a big number of "taking remainder" operations which are quite slow even on modern GPUs. The idea is to use so-called permutation textures - sets of precomputed hash values. You just need to come up with a reasonable function of picking a hash value from this texture. Possible solution can look like this:

```
int Hash( float3 P )
{
    return PermTexture.Load(
       int3(P.xy, 0) ).r ^ PermTexture.Load(
              int3( P.z, 0, 0 ) ).r;
}
```

Note that Load()'s are used instead of regular Sample() instructions. This is done in order not to normalize simplex coordinates and thus simplify computations. *PermTexture* in this example is a *UINT* texture, which means that it contains unsigned integer values, which are indices to a look-up table of gradient vectors.

Look-up table of gradient vectors can contain an arbitrary number of different vectors, however, in one of his researches Perlin found that 12 different vectors (in 3D case) would be enough to generate good noise patterns. He suggested to take a set of vectors directed to middles of the edges of hypercube (three-dimensional cube in 3D case). These vectors can be stored in a constant buffer for a shader to have the fastest access to this look-up table.

All we need to do after finding gradient vectors for a given simplex is to compute dot products and sum them up with a selected polynomial function (for his simplex noise Perlin used a degree of five function).

Turbulence function commonly looks like this:

```
float Turbulence3D( float3 p )
{
    float res = 0;

    for ( int i = 0; i<5; i++, p *= 2 )
       res += FrequencyWeights[i] *
              Snoise3D( p );

    return res;
}
```

This function is later used in all algorithms which are working with noise.

Similar routines can be written for 4D noise and for 3D flow noise (which is a simple 3D noise but with gradient vectors rotated with time)

3. ADDING DETAIL TO GEOMETRY

One of new usecases for noise is adding detail to geometry. This task perfectly fits to landscape rendering. In most cases landscapes are represented as big regular meshes which are made of big triangles. Different techniques are used to hide the lack of detail (multitexturing with a small detail texture, relief mapping), but these techniques can do nothing without the fact that the mesh is actually coarse – user still sees edgy silhouettes.

The cleanest solution would be to modify geometry itself – to tessellate it[6] to certain level and displace tessellated polygons with some function which would look very similar to landscape microrelief (Figure 4).

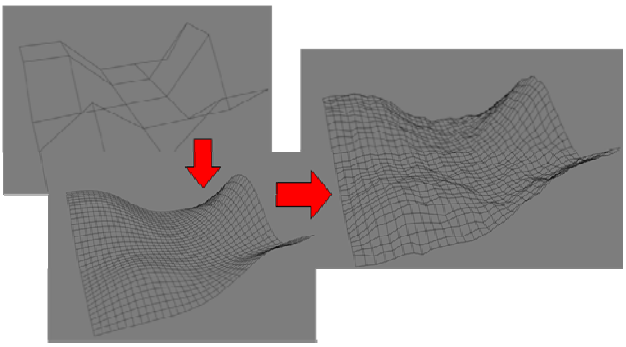

Figure 4: Adding detail to landscape

Why using Perlin noise function as such function? Using precomputed detail heightmaps would be good alternative. However, with noise you can generate procedural textures with much higher resolution than hardware is capable of storing, so you are saving memory and bandwidth. One may argue that using noise saves memory but increases computations, but the common rule is that in modern GPUs computational power grows much faster than bandwidth, so it much more favorable for an application to be computations-bound than to be memory-bound.

The second reason is that you can tweak Perlin noise (playing with frequency weights in turbulence function and with expressions in which turbulence function is used) to achieve a desired look and to make your microrelief look really natural. This nice property of Perlin noise is used in other effects as well.

And the third advantage is an ability to throw away high-frequency parts of turbulence function when you don't need them. That is, when you are rendering a landscape near the viewer, you can tessellate with higher tessellation factor and use more frequencies in turbulence function. But when you are rendering triangles which are far away from the viewer, you don't need to add that much of details and you can throw away those frequencies which viewer cannot see from a given distance.

Described approach was implemented and, given the possibility to tweak tessellation parameters, showed acceptable speed on a modern GPU – approximately 100 frames per second (GeForce 8800GT was used for testing).

4. DYNAMIC VOLUMETRIC EFFECTS

Commonly in real-time graphics effects as fire, explosions and smoke are represented as sets of slices blended together. Each slide contains its portion of represented effect. Picture on each slide can be animated to create a look of dynamically changing effect.

This approach is quite easy to implement, however, when such effect starts to interact with other objects in a scene, such artifacts as banding appear, unveiling the flaky nature of the effect.

Solution to this problem is representing such effects as truly volumetric effects, which can be evaluated in every point inside the volume, without being bound to any slices. To create dynamic truly volumetric effects we need either to store a four-dimensional texture in memory (which would require enormous amount of space) or compute this texture in real-time, for which Perlin noise is most suitable.

Basic idea for a volumetric fire effect is to use noise to generate vertical axis-aligned vertex field and distort some basic shape (Figure 5) with this field. This field can dynamically change if we use four-dimensional noise and treat fourth component as time.

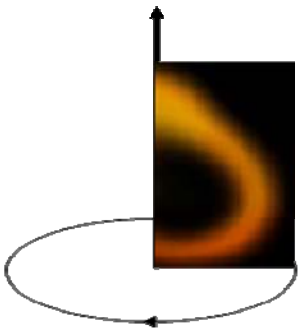

Figure 5: Basic fire shape

Basic fire shape is the main instrument of artist control for the described effect. Shape and color can be changed to achieve desired fire look.

Basic fire unit (Figure 6) is generated by revolving a fire shape around vertical axis.

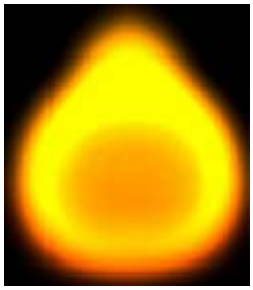

Figure 6: Basic fire unit

This shape is than distorted with vector field (Figure 7) generated using a 4D simplex noise (where fourth component stands for time). Rendering is done by ray marching a unit which contains fire shape and offsetting vertical position of each step by noise value. Displaced position is used to fetch a color from a revolved fire shape. All colors along a single ray are integrated to compute a final pixel color.

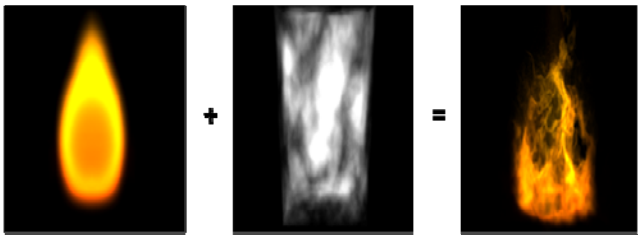

Figure 7: Distorting fire unit with noise

Varying weights of different frequencies in a turbulence function is another way of artist control. Tweaking these weights allows achieving different flame looks, from burning match to burning house.

Rendering speed of this approach depends only on the number of pixels occupied by described effect (this number equals to a number of rays traced). For different cases implementation of this effect showed from 20 to 40 frames per second.

4.1 Avoiding banding artifacts

Ray marching is performed with equal steps, so it is good to apply some jittering to initial positions of rays to avoid flaky look of the effect.

The main advantage of described effect is that it is defined in the whole entire volume and is not bound to any slices. That means that this effect can be evaluated at every point inside the unit volume. This is very useful when this effect is intersected with some obstacle (a case when traditional effects show up their flaky nature). Adaptive ray marching (Figure 8) can be used to avoid banding artifacts.

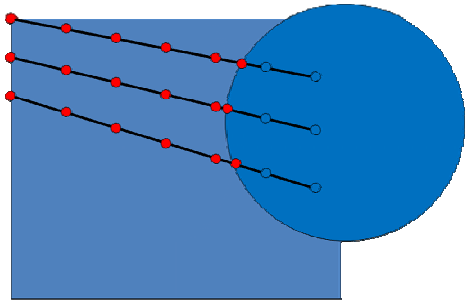

Figure 8: Adaptive ray marching

The idea of this approach is, when a next ray marching step is detected to be inside the occlude, place it to the occluder's border and integrate this sample with the appropriate weight. This would allow avoiding banding artifacts when fire unit is intersected with an arbitrary shape.

However, this would not allow making fire behave as if something was put inside of it (fire would not flow around the obstacle). The next chapter describes how to add physics simulations to this volumetric effect.

5. PARTICLE SYSTEMS

The idea of this approach is to represent a fire effect as a particle system. Each particle is simulated using fluid dynamics (this can be done on GPU using a stream-out which is available in DirectX 10) and occluders are taken into account when moving particles (Figure 9).

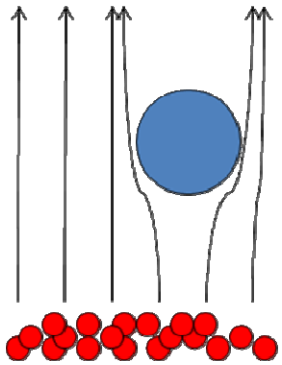

Figure 9: Particle trajectories

Particle system can be further voxelized in order to use ray marching to render this particle system (which commonly appears to be faster than directly rendering the entire particle system). Voxel grid can be frustum-aligned in order to make ray marching even more fast.

This approach is good for modeling a rough fire shape – it guarantees a physical correctness of the resulting effect. However, this approach would lack a distinctive whirligig nature of particles' motion.

The idea is to add this motion using Perlin noise. Noise can be used to generate vortex field, which is combined with particles' velocities computed during fluid simulations stage (Figure 10).

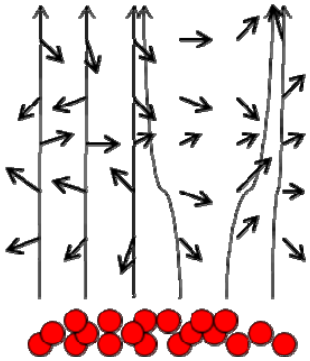

Figure 10: Combining trajectories with procedurally generated vortex field

Three samples of noise function are used to generate a single velocity vector in three-dimensional case. At each time step for each particle such vector is computed, and particle is advanced

ingenerated direction. Flow noise can be used to modify this vortex field by rotating gradient vectors.

Rendering speed of this approach doesn't depend on screen size of the effect and is only limited by a number of particles being simulated. For ~200 000 particles implementation of this effect showed 30 frames per second (GeForce 8800GT was used for testing).

6. CONCLUSION

Noise is a powerful function which allows generating not only procedural textures, but different patterns used in different applications. With modern GPUs it became possible to evaluate noise functions at real-time and use procedural patterns to generate new and improve existing effects (make them look more realistic and full of details).

Implementing noise in real time opens up a huge space for experimenting with using noise for different applications. Some of experiments performed by authors were described in this article, but there are still a lot of possibilities for applying noise in different fields of computer graphics. Keep thinking and experimenting on it!

7. REFERENCES

[1] Ebert, Musgrave, Peachey, Perlin, Worley, "Texturing & Modeling: A Procedural Approach", Third Edition, Morgan Kaufman, 2003

[2] Ken Perlin, "An Image Synthesizer", Computer Graphics magazine, Volume 19, Issue 3 (July 1985)

[3] Ken Perlin, "Improved Noise", International Conference on Computer Graphics and Interactive Techniques, Proceedings, Pages: 681-682, 2002

[4] Robert Cook, Tony DeRose, "Wavelet Noise", Pixar Animation Studios

[5] Ken Perlin, "Making Noise", GDC Talk, 1999

[6] Tamy Boubekeur, Christophe Schlick, "Generic Adaptive Mesh Refinement", GPU Gems 3, 2008

About the authors

Andrei Tatarinov is a student at Moscow State University, Department of Computational Mathematics and Cybernetics. His contact email is xroft86@mail.ru.