White Paper

Perlin Fire

February 2007
WP-03012-001_v01

Document Change History

Version	Date	Responsible	Reason for Change
01	February 15, 2007	AT, CK	Initial release

Go to sdkfeedback@nvidia.com to provide feedback on Perlin Fire.

Perlin Fire

Abstract

Volumetric effects such as fire and explosions play a significant role in modern interactive media applications. Typically, these applications use simple particle systems with varying textures to model these effects—but with unrealistic results.

The approach for fire described in this white paper uses an improved Perlin noise algorithm (suggested by Ken Perlin [1] and originally implemented by Yury Uralsky in an offline [3] algorithm). This method renders realistically animated fully procedural fire, with individual uniquely animated flame shapes generated in a pixel shader that uses three-dimensional simplex flow noise or four-dimensional simplex noise. The improved Perlin noise algorithm requires many complex scalar computations, making it a great match for the scalar architecture of the NVIDIA® GeForce® 8800 GPU. Coupled with the robust shader horsepower of the GeForce 8800, this algorithm can now be implemented for real-time effects.

Andrei Tatarinov, atatarinov@nvidia.com
NVIDIA Corporation

Motivation

Most fire effects in current interactive media applications are sprite-based or billboard-based. These effects use a set of two-dimensional animated images to create an illusion of real-life volumetric fire. When the images are placed well, the fire can be very believable.

Unfortunately, interacting with its environment can shatter the fire's impression. Unpleasant artifacts such as banding can appear and the sprite-based nature of the effect can show up. The screenshot in Figure 1 shows the possible artifacts of the sprite-based approach.

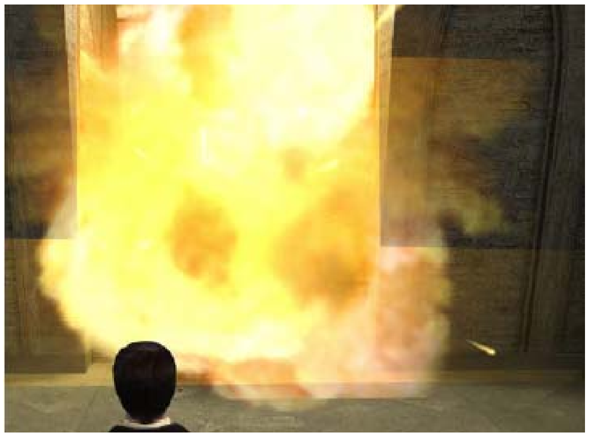

Figure 1. Sprite-Based Fire Effect Used in *Max Payne 2*

The improved Perlin noise algorithm discussed in this paper is fully volumetric, so it does away with many of the banding artifacts and does not have any predictable animation patterns. It is rendered using a ray-marching algorithm through a volume filled with fire, and it uses jittering to avoid banding.

How Does It Work?

Our suggested approach uses Perlin noise to perturb a basic fire shape created with a profile texture (Figure 2, Left), which is the only predefined texture in this algorithm. The profile texture controls the shape, color, and intensity of the fire. There are no restrictions on the size and shape of the profile texture and the artist can customize it to achieve the desired look and feel.

By sweeping this two-dimensional representation around the y-axis, we can create a cylindrical unit fire shape (Figure 2, Right). The dimensions of the unit volume are [-0.5; 0.5] for the horizontal axes x and z, and [0.1] for the vertical y-axis. Each rectangular volume, which contains a fire, is mapped into this unit volume.

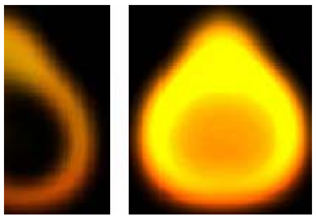

Figure 2. Left: Profile Texture. Right: Unit Fire Shape

However, calculating traditional Perlin noise in four dimensions is extremely expensive. Using simplex Perlin noise, we can significantly reduce the number of interpolations needed to compute a noise value. For a space with *N* dimensions we need to interpolate only (`N+1`) values for simplex Perlin noise—instead of `2N` values, as in the case of traditional Perlin noise.

Note: Simplex in *N*-dimensional space is the simplest and most compact shape that can be repeated to fill the entire space. It contains as few corners as possible, much fewer than a hypercube.

Calculating Perlin noise requires a large number of random numbers, which are generally too expensive to compute in real time. Instead, we use a permutation texture—a two-dimensional look-up texture filled with precomputed random numbers. To associate each simplex grid point with a gradient vector, we use the coordinates of a point as an index into the permutation texture to compute a hash value. The hash value is then used as an index into a look-up table of predefined gradient vectors.

For each pixel in the fire volume, a ray-march is performed. Each ray's initial position is shifted according to a value in a jittering pattern filled with noise. For each ray, a number of equal-length steps are performed. At each step, the obtained ray position is used as the texture coordinates for the unit fire volume and for the computation of the noise values.

Several octaves of Perlin noise are computed at each step and summed up with the given weights, and this sum is used to displace the unit volume texture coordinates along the y-axis. These coordinates in turn are used to obtain a color for a given point. The color of a pixel is considered the integrated sum of all the colors calculated during the ray-marching. The algorithm stops tracking a ray when it encounters an obstacle or when it reaches the unit fire volume boundaries.

Implementation Details

There are two files that implement the algorithm. `PerlinFire.cpp` prepares the scene and all the required resources. It then renders a fire volume, applying one of the techniques described in the effect file, `PerlinFire.x`.

The effect file, `PerlinFire.x`, contains the following routines:

- **`simplex3D`**: This routine computes four vertices of a 3D simplex, which contains a given point.
- **`simplex4D`**: This routine computes four vertices of a 4D simplex, which contains a given point.
- **`hash`**: This routine computes a hash value for a given grid point.
- **`snoise3D`**: This routine computes the 3D simplex Perlin noise value for a given point in three-dimensional space.
- **`snoise3Dflow`**: This routine computes the 3D simplex Perlin flow noise value for a given point in three-dimensional space. The flow noise gradient vectors are rotated around the y-axis to imitate the noise changing in time.
- **`snoise4D`**: This routine computes the 4D simplex Perlin noise value for a given point in four-dimensional space.
- **`turbulence3D`**, **`turbulence3Dflow`**, and **`turbulence4D`**: These routines compute turbulence noise values (a weighted sum of several octaves of noise).
- **`PerlinFire3DPS`**, **`PerlinFire3DFlowPS`**, and **`PerlinFire4D`**: These are the pixel shader routines, which perform the ray-marching algorithm.

References

[1] Ken Perlin
"Noise Hardware"

[2] Stefan Gustavson, Linkoping University, Sweden
"Simplex Noise Demystified"

[3] Yury Uralsky, NVIDIA Corporation
"Volumetric Fire"