GPU-Accelerated Iterated Function Systems

Simon G. Green
NVIDIA Corporation

We describe GPUflame, an application that generates high quality renderings of iterated function system (IFS) fractals using graphics processing (GPU) hardware to accelerate both the computation and display of the final images. The application is inspired by Scott Drave's "Fractal Flame" animations [Draves 2003]. By using the computational performance of the GPU our system allows real-time rendering of IFS patterns and interactive exploration of the parameter space. We also extend the usual two dimensional IFS functions to three dimensions to produce solid 3D patterns that can be viewed from any angle, and use the floating point blending capability of recent GPUs to render high dynamic range IFS images.

1 Algorithm

Iterated function systems are a well known class of fractal algorithm. They consist of a finite set of functions that map points from one point in space to another. (Traditionally the points are represented in the two-dimensional image plane, but we also allow points in three dimensions.) The solution to the system is the set of points S that is the convergence point of the recursive set equation:

$$S = \bigcup_{0 \le i < n} F_i(S)$$

The functions are traditionally affine transformations such as rotations, scales and translations, but can also be non-linear functions such as sin or cos. For a stable solution to be found, the functions should be contractive, so that on average they bring points closer together.

The easiest way to produce an image of an IFS is using the "chaos game" [Barnsley 1988]. This consists of choosing a random point, choosing one of the functions at random, applying it to the point, and then repeating this process for a number of iterations until the point settles on a stable position. If this is repeated for a large number of points, a pattern will emerge. One of the best known examples of an IFS is the Sierpinski gasket (Figure 1).

2 GPU Implementation

We implement the IFS algorithm on the GPU using standard GPGPU [Pharr 2005] techniques in OpenGL. The positions of the points are stored in a 2D floating point texture. A single quadrilateral is drawn for each iteration. A fragment program reads the position of the point from the texture, applies one of the functions chosen at random to move the point to its new position, and then writes the result back to the frame buffer. Since current graphics hardware doesn't include random number functions, a texture is pre-computed that contains pseudo-random numbers, and the fragment program looks up in this based on the fragment position and current time value. By modifying the parameters of the functions over time animated sequences can be generated.

Once we have the texture containing the positions of the points, they can be rendered using vertex texturing. The vertex program reads from the position texture and applies the correct transformations to display the point on the screen.

Rendering the points as solid pixels does not produce very compelling images. Instead, we use additive blending so that the higher the point density in a particular region, the brighter it is. Since the distribution of point densities is very non-linear, we perform the blending to a floating point frame buffer, which is then displayed using a final tone mapping step. We use point sprites to allow arbitrary images to be used for the points.

The points are colored based on a value that tracks the number of the function that was applied at each step, thus revealing the structure of the pattern. This value is stored in the alpha channel and then used in the shader to index into a color table stored in one dimensional texture. Future work will investigate more complex coloring schemes based on 2D or 3D color lookup tables, as well as the addition of physical simulation to add secondary motion.

Since the point data remains resident on the graphics card, and the GPU hardware can evaluate the IFS functions and render the points very quickly, we achieve performance of up to 20 frames per second with a million points. This allows interactive exploration of a complex mathematical system that would not be possible with offline rendering.

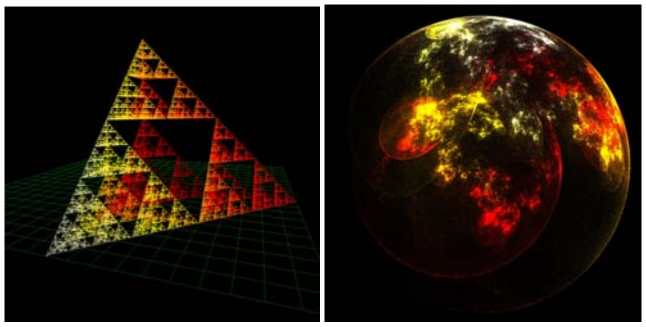

Figure 1. 3D Sierpinski and spherical IFS patterns

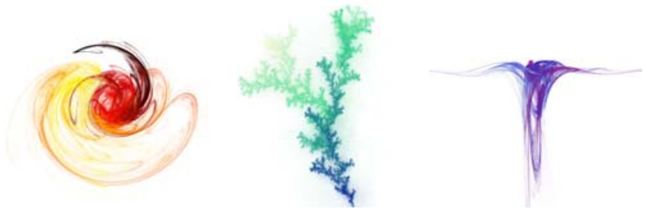

Figure 2. Example non-linear 2D IFS patterns

References

DRAVES, S. 2003 The Fractal Flame Algorithm. http://flam3.com
BARNSLEY, M. 1988 *Fractals Everywhere,* Academic Press.
PHARR M., Editor. 2005. *GPU Gems 2: Programming Techniques for High Performance Graphics and General Purpose Computation*, Chapters 29 – 36, Addison Wesley.