IEEE/ EG Symposium on Volume and Point-Based Graphics (2008)
H.- C. Hege, D. Laidlaw, R. Pajarola, O. Staadt (Editors)

Pseudorandom Noise for Real-Time Volumetric Rendering of Fire in a Production System

Y. Vanzine[1] and D. Vrajitoru[1]

[1]Indiana University South Bend, South Bend, USA

Abstract

This paper presents an effort at developing a robust, interactive framework for rendering 3D fire in real-time in a production environment. Many techniques of rendering fire in non real-time exist and are constantly employed by the movie industry and have directly influenced and inspired real-time fire rendering, including this paper. Macro-level behavior of fire is characterized by wind fields, temperature and moving sources and is currently processed on the CPU while micro-level behavior like turbulence, flickering, separation and shape is created on the graphics hardware. This framework provides a set of tools for level designers to wield artistic and behavioral control over fire as part of the scene. The resulting system is able to scale well, to use as few processor cycles as possible, and to efficiently integrate into an existing production environment. We present performance statistics and assess the feasibility of achieving interactive frame rates within a 3D engine framework. The framerates we obtained vary from 42 to 168 depending on the rendering conditions, and indicate that the real-time procedural fire might not be far away.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

1. Introduction

There are a multitude of fires that exist in the physical reality and have a wide range of visual representations. The algorithms generally used to create 3D fires in contemporary video games use primitive particle emitter systems [HA02]. Particle emitter systems are still the most efficient and well understood way to render fuzzy phenomena in real time. Volumetric rendering, on the other hand, though more appropriate, is considered prohibitively expensive. For our model of fire, we explore volumetric rendering and attempt to take advantage of modern shader hardware.

Few real-time production systems go the extra length to model fire outside of particle emitters because it is difficult to simulate and computationally expensive. Usually, a game features a single way of rendering fire, where several peculiar appearances are required. This results in an environment which denies the participant not only the realism but also the suspension of disbelief. Some games feature expensive fire models which allow them to use fire in pre-rendered sequences or inside their environment but very sparingly. The problem remains unsolved, even though most production systems have provided realistic but incomplete models.

For example, the PC game Half Life 2: Episode 1, by Valve, represents one of the contemporary real-time fire models. The technique uses a flat surface and alpha-blending of the resulting colors into a pseudorandom fire in order to achieve the effect of emissive lighting. The geometry of the fire primitives consists of a flat view-aligned plane and fire size scales poorly. The texture of fire often has a 5-second cycle after which the moving image is repeated in a flip-book animation fashion. This production fire is not designed to allow dynamic change of LOD (level of detail) very easily for different levels of performance.

Other methods of fire implementation in production traditionally include particle-based blob fire [SF95]. It is found in many games: World of Warcraft by Blizzard Entertainment, Company of Heroes by Relic Entertainment or Guild Wars by ArenaNet. Particle-based fire suffers from appear-

ing atomic, as opposed to the natural holistic look of fire. One way to deal with it is to enlarge the size and reduce number of the particles.

More sophisticated models of fire can also be encountered consisting of at least two subsystems. For instance, F.E.A.R. by Monolith Productions features fire with the shader-based flame system and the particle smoke system. Adding external detail to the fire model does help conceal the fact that the core fire system in F.E.A.R. consists of a rigid surface shader fire which does not evolve in shape over time.

Previous work in hardware-accelerated volumetric fire reported frame rates outside of any engine environment based solely on the productivity of the fire system itself. We chose to test the fire frame rates within a game engine, because one can only properly evaluate feasibility of these methods when the experiment is conducted in a real, production setting.

An alternative rendering method would be raycasting, but to obtain results of similar quality it would require a density of voxels that would be prohibitively expensive in terms of rendering cost for a real-time production system. Ray casting is still not used in such rendering engines.

2. Previous Work

Movie industry has for a long time utilized quite realistic fire rendering, starting as early as 1985 with the movie Star Trek II: The Wrath of Khan made by Pixar Studios [Ree85]. Various such techniques exist in the literature and they require off-line rendering that takes an average of two or three minutes of rendering per frame. Off-line rendering algorithms of fire are divided into three distinct categories: physics-based, particle or texture-based, and mixed [Eym04].

Physics-based algorithms of rendering fire are based on the laws of fluid dynamics and represent fire as hot and turbulent gas [FM97, FM96]. Several authors utilize incompressible Navier-Stokes to model vaporized fuel and hot gas with voxels [SRF05, HSF02].

In the original particle system algorithm [Ree85], Reeves was the first to describe and use particle systems to model fire, as used in the film Star Trek II: The Wrath of Khan. To avoid the computational complexity of large particle systems, King et al. [KCR00] have used textured splats for fire animation. Texture splats were so promising that Wei et al. [XWK02] successfully used them to render fire in real time with 15ms per frame. Texture splats were used in a number of games, but, in our opinion, their appearance does not have the holistic look fire must have and they are better suited to smoke or steam simulation.

Lamorlette and Foster [LF02] provide a very solid mixed framework for macro-movement of fire that was used for the motion picture Shrek by DreamWorks. This model is the closest to the system we chose to implement. It uses parametric space curves to define the spine of the flame that evolve over time and particles are point-sampled close to the visible part of the flame using a volumetric falloff function. Procedural noise is applied to the particles which are then rendered using either a volumetric, or a fast painterly method.

Wind fields are an important aspect of making the fire look realistic. A good description can be found in [FM97, FM96]. They are influenced by the velocity and temperature of the gas in its interaction with the surrounding air. Turbulent motion is exaggerated if the gas flows around solid objects. At first the gas flows smoothly along the surface, but it eventually becomes chaotic as it mixes with the still air behind the object.

A reduced form of Navier-Stokes equations [FM97] is appropriate for modeling of the wind field and is described below. Let u be a four-dimensional vector consisting of three spatial dimensions and time as a fourth dimension. Thus, $u = (x_p, t)$, where x_p is the displacement vector of the particle. $w(u)$ represents the change of velocity of gas in an arbitrary wind field and is expressed as:

$$w(u) = \nu\nabla \cdot (\nabla u) - (u \cdot \nabla)u - \nabla p \qquad (1)$$

Equation 1 describes how the velocity of the gas changes over time depending on convection $(u \cdot \nabla)u$, its pressure gradient ∇p, and drag $\nu\nabla \cdot (\nabla u)$. The ν coefficient is the kinematic viscosity. Smaller viscosity leads to more rotation in the gas.

We draw inspiration for realistic volumetric rendering from research done in [FKM*07], and present a more comprehensive study of the integration of this model within a game engine and taking advantage of shader software.

Perlin Noise [Per02] or Improved Perlin Noise can be used as a procedural shader algorithm which is used to increase the level of realism in surface texture. It is implemented as a function of (x, y, z) or $(x, y, z, time)$ which uses interpolation between a set of pre-calculated gradient vectors to construct a value that varies pseudorandomly over space and time.

M-Noise [Ola05] or Modified Noise is a more recent alternative to Improved Perlin Noise, specifically tailored for execution on GPUs. It is especially useful for for 3D or 4D noise not easily stored in reasonably sized textures. Perlin Noise uses several chained table look-ups, the operations that can lead to a bottleneck on GPUs. It is largely a faster and better, and although more complex adaptation of Perlin Noise to GPU hardware.

3. Modeling Fire

A more detailed description of the system and its implementation can be found in [Van07]. Our system uses a robust parametrized system of rendering fire consisting of macro-behavior and micro-detail. For macro-movement, each fire

source is represented by a fire skeleton which is influenced by forces such as direct diffusion, movement of the fire source, thermal expansion and arbitrary wind fields. Micro fire effects, *e.g.*, fire flame shape, location, flickering and turbulence are rendered by a high-level shader. While the CPU is freed up by rendering local fire phenomena on the GPU, such pipeline separation provides the necessary animator or simulation control at the high level and allows for real-time rendering of detail-rich, realistic fire at the local level.

This model has been implemented in the Irrlicht 3D engine and rendered in both OpenGL and DirectX. For shader support we use Open GL Shader Language (GLSL) and High Level Shader Language (HLSL).

3.1. Model Outline

The model as a general fire animation tool has 5 stages:

1. Individual flame elements are modeled as parametric space curves. Each curve interpolates a set of points that define the spine of the flame.

2. The curves evolve over time according to a combination of physics-based, procedural, and hand-defined wind fields. The curves are frequently re-sampled to ensure continuity, and to provide mechanisms to model flames generated from a moving source.

3. A texture based, cylindrical profile is used to build a color volume representing the oxidation region for the shader rendering step, *i.e.*, the visible part of the flame as shown in Figure 1. The particles are transformed into the parametric space of the flame texture using inverse parametrization. M-noise or Improved Perlin noise provides local turbulent detail by means of pseudorandom gradients applied to each pixel.

4. The particles are rendered as shader fragments using shader hardware acceleration. The color of each pixel is trilinearly interpolated according to color properties of its neighbors, allowing flame elements to visually merge in a realistic way.

5. To complete the system, we define a number of procedural controls to govern placement, intensity, lifespan, and evolution in shape, color, size, and behavior of the flames.

3.2. Curve-based Spline Modeling

The structure of the fire volume is created by interpolating a smooth curve through the points that a) evolve when affected by various forces of the physics simulation or b) stay constant as a result of being positioned manually. After considering Hermite interpolation, we decided that uniform B-splines would be the best approach because they guarantee second order continuity, present affine invariance, and have the convex hull property. We employed the de Boor algorithm [Far02] for the interpolation.

To define a rendering volume around the curves, a local reference frame is necessary. For this purpose, after building the curve the next step consists in defining a bounding volume around the curve. This volume is composed of hexahedrons defined continuously at regular intervals of the curve, as shown in Figure 1. These hexahedrons are determined by the Frenet Frame [Far02].

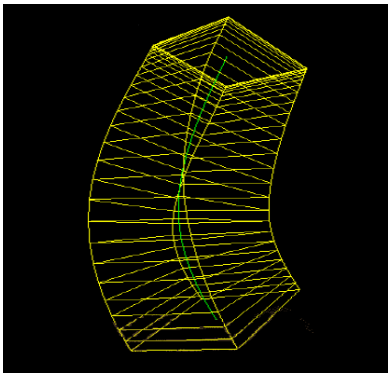

Figure 1: *Flame spine curve and bounding volume composed of hexahedrons*

3.3. Macro Animation of the Flame

Physics-based controls in Step 2 govern macro-movement of flame spines according to the system of equations in [Kan99]. The primary equation of motion is

$$\frac{dx_p}{dt} = w(x_p,t) + d(T_p) + V_p + c(T_p,t) \qquad (2)$$

where$w(x_p,t)$ is an arbitrary controlling wind field, $d(T_p)$, the motion due to diffusion of particles modeled as temperature-scaled Brownian motion, V_p , motion due to movement of the source and $c(T_p,t)$, the motion due to thermal buoyancy. T_p is the temperature of the particle. Thermal buoyancy is constant over the lifetime of the particle:

$$c(T_p,t) = -\beta g(T_o - T_p)t_p^2 \qquad (3)$$

where β is the thermal coefficient, g is gravity, T_o is the ambient temperature and t_p is the age of the particle given when the particle is created.

In order to create an arbitrary wind field, simplified Navier-Stokes equations may be used to simulate convection and macro drag [FM97]. In the above formulation, it is assumed that motion due to molecular diffusion is negligible relative to other effects and that the gas is incompressible. When these assumptions are applied to the Navier-Stokes equations, which fully describe the forces acting within a gas, the reduced form is derived.

3.4. Volumetric Rendering

To render volumetric fire we use a lattice and volume-slicing technique first described in [CCF94] and perfected for shader hardware in [FKM*07]. An example of such a lattice can be seen in Figure 2, where red triangles that slice through the volumes on the right are view-aligned to the camera location on the left. The complete volume of fire is approximated by a set of hexahedrons, each with constant height and with the bottom perpendicular to the tangent of the curve at the current control point.

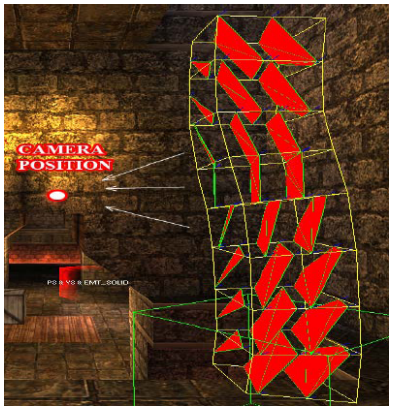

Figure 2: *Fire volume and view-aligned slicing*

Every time the volume of fire is rendered, it is broken up into an configurable arbitrary number of sub-volumes corresponding to knots in the de Boor interpolation, defined as the density of the complete volume of fire. Each sub-volume is then sampled into evenly spaced view-aligned slices using an optimized cube-slicing algorithm. Each slice is modeled efficiently as a triangle fan for processing in the shader hardware. We selected a triangle fan as it is common to both OpenGL and DirectX. Each triangle is rendered through a pixel shader after obtaining the color from the fire texture.

View-aligned slicing is created with a number of planes perpendicular to the near edge of the view frustum. These planes are then clipped by their intersection with the volume bounding box of the current sub-volume hexahedron. The texture coordinates in the parametric object space are mapped to each vertex of the clipped polygons taking advantage of the fact that the edges of the volume bounding box have fixed texture coordinates. During rasterization, fragments in the slice are trilinearly interpolated from the 3D texture and projected on the image planes using adequate blending operations. We used the depth queuing with the over operator [PD84] to account for translucency of the fire texture. Figure 3 shows the 2D texture applied to the resulting triangles.

With such a method of volume slicing, we can minimize the number of planes intersecting the volume and maximize the speed of rendering without creating noticeably discrete gaps between the planes when examining the object from different viewpoints.

Figure 3: *Fire texture*

When the slicing plane cuts through the volume, we obtain their intersection points and we must organize them into a triangle fan for the graphics pipeline. This process is called appropriately polygon triangulation and it is a complex, fundamental algorithm of computational geometry.

When a plane intersects the volume of fire, the possible intersections are a triangle (3 vertices), a quad (4), a pentagon (5) or a hexagon (6). The intersection calculation algorithm provides a cloud of points or vertices as output. The problem that must be solved is determining the order of the vertices in the convex polygon, which is equivalent to finding out which vertices are connected by the edges of the polygon. This is essentially the problem of triangulation.

The polygon vertices are always found on the edges of the common convex hexahedron. There are always a limited number of intersection combinations when the plane cuts through the hexahedron. Because there are 8 vertices in a hexahedron, there exist 2^8 cases, where any particular group of vertices is found behind, in front of, or on the slicing plane. For every point we calculate the dot product between the normal to the plane and a vector from the origin of the plane to the point in question. If the resulting number is positive, the point is in front of the plane; if it is negative, the point is behind the plane. The point is on the plane if the result is equal to zero.

It is beneficial to establish the plane intersection combinations in order to be able to perform less than the 12 intersections between the cutting plane and the edges of the hexahedron each time. Also, if such combinations are established, with the help of the above heuristic we can also pre-define the correct ordering of vertices in preparation for submission of the vertices to the rendering pipeline. The clockwise ordering of vertices will be sufficient for creation of triangle fans in linear time.

This very original and simple idea of pre-calculating intersections between a plane and a volume is described in [BPP01] and in [BBJL05] based on the classic Marching Cube (MC) algorithm [LC87]. The original MC algorithm is employed for approximation of the boundaries of the volumetric object. The MC Slicing algorithm variant applies the pre-calculated table look-up idea to the volume slicing.

The MC algorithm allows one to efficiently polygonize an approximation of the intersection between a surface and a cube. The approximation is achieved by evaluating a predicate (condition) at the eight corners of the cube. The 256 possible combinations are known and stored in a pre-calculated table. Each entry of the table is a sequence which indicates which edges were hit by the surface and allows us to interpolate the intersection triangles.

With MC Slicing, only at most 6 intersections between the slicing plane and the hexahedron are tested for. To determine the relation of the point to the slicing plane, one dot product operation is performed per point classification, with a total of 8 points. This qualifies MC Slicing as a very efficient algorithm for our specific triangulation task.

3.5. Micro-Movement and Noise

Noise functions are used to simulate an appearance of randomness and in computer graphics in particular, to make objects appear more like their counterparts in nature. These noise functions often have a fractal component resulting from adding noisy values of different scales, also called noise octaves. The noise can be configured using the property of persistence, representing the amplitude of each noise frequency, that can be further decomposed into lacunarity and gain.

Perlin noise is a so-called gradient noise, which means that a pseudorandom gradient is set at regularly spaced points in space, and a smooth function between those points is interpolated. *Simplex noise* uses simplex grids dividing the space. Both Perlin's classic and improved noise were designed to run efficiently on a CPU. Modified Noise includes two modifications to Perlin's improved noise that make it much more suitable for GPU implementation, allowing faster direct computation.

Modified noise or M-Noise calculates its gradient based on 2N lattice points in a hypercube around the point to which the noise function outputs are applied. In M-Noise gradient values are chosen from the corners of the hypercube instead of its edge centers like in improved noise and instead of the unit n-sphere like in classic noise.

Several octaves of noise can be combined to produce a random function with a more complex spectrum. The combination commonly used and the function with which things like smoke, fog, fire are made, is called turbulence [FKM*07], and is defined by

$$turb = \sum_{i=0}^{n} gain^i abs(noise(position \cdot lacunarity^i)) \qquad (4)$$

where n is the number of octaves and, generally, *gain* is equal to $1/lacunarity$.

While turbulence in Perlin's classic and improved noise make independent calls to the noise function, the common computation of M-Noise allows for a more efficient turbulence function. For 3D turbulence, the hash function can be computed up to two octaves together and the flerp function up to four octaves together. For 4D turbulence, the hash function can be computed up to three octaves together and the flerp function up to four octaves together.

Each type of noise has a distinct visual appearance. Simplex noise stands out in its appearance with round shapes dominate the surface. Simplex noise has a higher peak range than the other noise types but it is possible to generate a Simplex noise-based turbulence resembling a Perlin noise-based turbulence, via careful manipulation of lacunarity and gain and noise scale.

M-Noise and Perlin improved noise in 2D and 3D look almost identical, and it is because the method of construction of these types of noise is very similar. Modification of M-Noise only addresses the bottlenecks of noise creation on GPU rather than CPU via the dimension reducibility property for improvements in memory requirements and minimizing number of texture-dependent lookups. It also takes advantage of GPU parallelism and provides a faster method of computing random hashes which allows for purely computable noise (*i.e.*, noise without texture-dependent lookups). The only difference between these two types of noise is in the manner the gradients are picked. Perlin noise selects gradient vectors from the centers of the edges of a unit-n cube. M-Noise picks its gradient from the corners of a unit-n cube, because it requires dimension reducibility. The visual difference becomes evident when 4D M-Noise is compared to others.

Figures 4 and 5 (left) show the three types of noise as applied to rendering fire. Figures 6 to 8 show a variety of artistic effects that can be easily achieved by varying the simulation parameters.

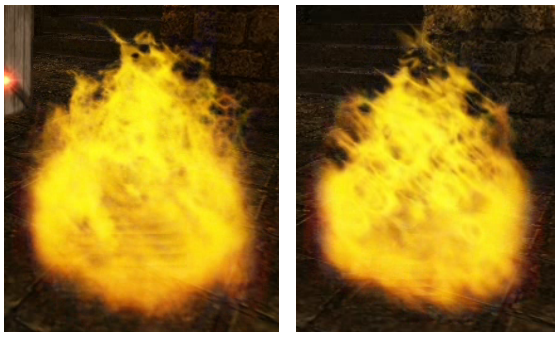

Figure 4: *Fire rendered with Perlin noise (left) and Simplex noise (right)*

Modified noise in Figure 5 right has an almost 'fluid' character. It is the only noise whose large, solid parts separate and float away from the main body of fire very similarly to how natural fire separation occurs.

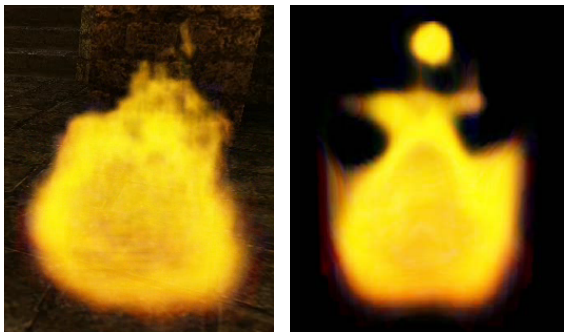

Figure 5: *Fire rendered with M-Noise (left) and 4D M-Noise showing fire separation (right)*

Figure 6: *Simulation of a simple candle (left) and of fire seen through a stained glass (right)*

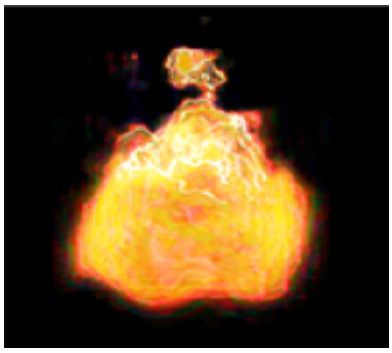

Figure 7: *Simulation of an electrical charge*

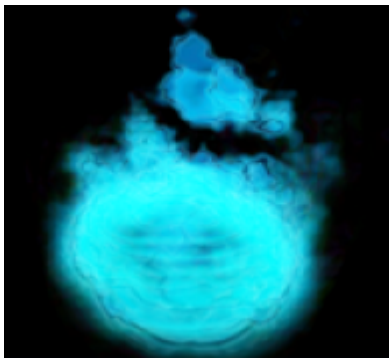

Figure 8: *Simulation of a blue gas*

4. Feasibility Study

As we focus on testing the possibility of rendering fire volumetrically within the constraints of a real 3D engine production system, we must assess the feasibility of achieving interactive frame rates within such framework. As a goal of our feasibility study we set out to collect statistics and measure performance of the volumetrically rendered fire, depending on the degrees of freedom of our framework.

For the 3D engine framework, we have studied the effect of the underlying APIs used to render 3D primitives, *i.e.*, OpenGL/GLSL or DirectX/HLSL and of the number of geometric primitives already present in the scene beside the fire. For the chosen fire model, the physical parameters include time, flame temperature, flame velocity, gas thermal coefficient β, and fire particle's age. The procedural degrees of freedom include the number of flame volumes in the scene, lattice resolution (density and slice spacing) of the volume, screen size of fire, the number of noise octaves and most importantly, the graphics hardware.

Testing was done on a Pentium 4 3.0 GHz PC, equipped with ATI Radeon X1650 video card with 256 Mb of memory. The total rendering space (Viewport) is 800 by 600 pixels. The physics engine is turned off for these tests. The number of flame volumes is 1, the slicing is MC, and the engine is OpenGL/GLSL unless otherwise specified.

The measure of performance is the number of frames per second (FPS), for which the standard number in video games is either 60 FPS without blur or 30 FPS with blur for smooth believable animation. Higher values of the FPS indicates a better graphics performance.

We start by figuring out the impact of a standard configuration of static fire on the production system. Such a configuration consists of a single volume of fire with 2 subvolumes of 64 flame knots, 20 slices (view-aligned surfaces that cut through the volume), 1 flamespine attached to the firescenenode. The total count of scene triangles in the viewport is 3174. The number of triangles that belong to the fire model alone is 70. Table 1 compares the performance between matching gradient noise models. The octree represents the scene set up for the fire.

Table 2 compares three different triangulation techniques, the MC Slicing, the classic Convex Hull (CH) algorithm [CLRS01] and an optimization (CH Opt) taking advantage of the fact that the slices are placed at regular intervals. We increased the density of the lattice to 8 subvolumes as the difference in performance on a sparse lattice is negligent. CH

Table 1: *Noise Test, Density:2, Slices:20, Screen Space: 4%, Noise Octaves: 4, 174 triangles, 70 fire triangles*

Octree	Perlin	Simplex	M-Noise
FPS without	30	53	119
FPS with	22	45	101

Table 2: *Triangulation Test, Density:8, Slices:20, Screen Space: 7%, Noise Octaves: 4, M-Noise*

Slicing method	Scene triangles	Fire triangles	FPS
MC	3885	300	68
CH Opt	3836	303	53
CH	3740	301	56

and CH Opt perform about the same, while MC performs significantly better.

Table 3 is meant to show the dependency of complexity of rendering the fire on the lattice density. The lattice density defines the number of knots of the flame spine and thus, the number of hexahedrons composing the fire. The complexity increases logarithmically with the increase of subvolumes. Potentially, there exist more significant optimizations that would lead to lessening the performance hit caused by additional lattice sections [FKM*07].

Table 4 is meant to help calculate the rate of the complexity increase with the higher number of view-aligned surfaces (slices) intersecting the hexahedrons. The complexity increases with logarithmic time.

Table 5 compares performance of the fire system with varying number of fire volumes. Additional fire volumes add to the complexity of the rendering algorithm logarithmically as the GPU parallelizes the shader program calls.

Last, Table 6 compares the performance of the three types of noise using OpenGL and DirectX. As the difference of 20 FPS seems to be persistent regardless of other graphic settings, we concluded that it was inherent to the version 1.3.1 of the Irrlicht engine.

Table 4: *Slice Test, Density:2, Screen Space: 4%, Noise Octaves: 4, M-Noise, Scene triangles: 3174*

Slices	Fire Triangles	FPS
10	34	168
20	70	101
30	106	73
40	145	56
50	145	46

Table 5: *Varying Fire Volumes, Density:2, Slices: 20, M-Noise, Noise Octaves: 4, Scene triangles: 10*

Flame Volumes	Screen Space	Fire Triangles	FPS
1	4.0%	70	119
2	8.0%	140	62
3	12.0%	210	42

The statistics collected show that pseudorandom noise (Perlin noise and its derivatives), even when thoroughly optimized for the GPU, is still a very expensive tool for a production system. The fact that 4D noise is expensive is confirmed when we compare the expense of rendering a 4% - screen space volumetric fire (150 FPS drop) to the expense of rendering the octree environment (a medieval castle) (only 8 FPS drop). But even as is, the developed component can be and should be used within a production system, albeit with lower lattice density and minimum slicing of the volume. The objective of this study to run realistic volumetric fire at above 30 or 60 frames per second was successfully completed.

With volumetric rendering already being used as a viable production tool, we must look onward and anticipate

Table 3: *Density Test, Slices:20, Screen Space: 7.5%, Noise Octaves: 4, M-Noise, 3816 scene triangles*

Lattice density	Fire Triangles	FPS
3	100	87
8	258	69
13	407	59
19	521	54
25	644	50

Table 6: *Slice Test, Density:2, Screen Space: 4%, Noise Octaves: 4, M-Noise, Scene triangles: 3174.*

Gradient Noise	Fire Triangles	FPS DirectX/HLSL	FPS OpenGL/GLSL
-	0	232	252
Perlin	70	29	53
Simplex	70	51	72
M-Noise	70	79	101

increase in use of procedural noise (among fluid solvers and other physics-based techniques) for volumetric effects.

5. Conclusion

In the study presented in this paper, a fire-rendering framework has been written using the Irrlicht 3D engine, utilizing established and novel algorithms. Improved algorithms have been described. Physical and procedural controls are leveraged to explore performance of the fire model. We showcased the influence of the noise type on the aspect of the rendered fire and presented various artistic effects that can be obtained within the framework.

Statistics have been collected to determine system bottlenecks and to test the feasibility of the model. We determined that the volumetric fire model can be integrated with the 3D engine framework and still not let the overall system performance deteriorate below the acceptable frame rate of 30-60 frames per second. However, because of the expensive nature of pseudorandom gradient noise, even scaled production use of volumetric rendering of fire is still a year or two away.

References

[BBJL05] BENASSAROU A., BITTAR E., JOHN N., LUCAS L.: Mc slicing for volume rendering applications. In *5th International Conference on Computational Science* (2005), Springer, pp. 314–321.

[BPP01] BEAUDOIN P., PAQUET S., POULIN P.: Realitic and controllable fire simulation. In *Proceedings of Graphics Interface* (2001), pp. 159–166.

[CCF94] CABRAL B., CAM N., FORAN J.: Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings of the 1994 Symposium on Volume Visualization* (1994), ACM Press, pp. 91–98.

[CLRS01] CORMEN T., LEISERSON C., RIVEST R., STEIN C.: *Introduction to Algorithms*, second ed. MIT Press and McGraw-Hill, 2001.

[Eym04] EYMAN Y.: *Rediscovering Fire: A Survey of Current Fire Models and Applications to 3-D Studio Max*. Tech. rep., University of Maryland, 2004.

[Far02] FARIN G.: *Curves and Surfaces for Computer-Aided Geometric Design*, fifth ed. Academic Press, 2002.

[FKM*07] FULLER A. R., KRISHNAN H., MAHROUS K., HAMANN B., JOY K. I.: Real-time procedural volumetric fire. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games* (2007), pp. 175 – 180.

[FM96] FOSTER N., METAXAS D.: Realistic animation of liquids. *Graphical Models and Image Processing 58*, 5 (1996), 471–483.

[FM97] FOSTER N., METAXAS D.: Modeling the motion of a hot, turbulent gas. In *Proceedings of the ACM SIGGRAPH Conference* (1997), pp. 181–188.

[HA02] HAWKINS K., ASTLE D.: *OpenGL Game Programming*. Course Technology PTR, 2002.

[HSF02] HONG J.-M., SHINAR T., FEDKIW R.: Wrinkled flames and cellular patterns. *ACM Transactions on Graphics 26*, 3 (2002), 47.

[Kan99] KANDHAI B. D.: *Large Scale Lattice-Boltzmann Simulations*. PhD thesis, University of Amsterdam, 1999.

[KCR00] KING S. A., CRAWFIS R. A., REID W.: *Volume Graphics*. Springer, 2000, ch. Fast Volume Rendering and Animation of Amorphous Phenomena, pp. 229–242.

[LC87] LORENSEN W., CLINE H.: Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics 21*, 4 (1987), 163–170.

[LF02] LAMORLETTE A., FOSTER N.: Structural modeling of flames for a production environment. In *Proceedings of the 29th annual conference on Computer Graphics and Interactive Techniques* (2002), pp. 729–735.

[Ola05] OLANO M.: Modified noise for evaluation on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware* (2005), pp. 105–110.

[PD84] PORTER T., DUFF T.: Compositing digital images. *Computer Graphics 18*, 3 (1984), 253–259.

[Per02] PERLIN K.: Improving noise. In *International Conference on Computer Graphics and Interactive Techniques* (2002), pp. 681–682.

[Ree85] REEVES W. T.: Approximate and probabilistic algorithms for shading and rendering structured particle systems. *Computer Graphics 19*, 3 (1985), 313–322.

[SF95] STAM J., FIUME E.: Depicting fire and gaseous phenomena using diffusion processes. *Proceedings of the ACM SIGGRAPH Conference* (1995), 129–136.

[SRF05] SELLE A., RASMUSSEN N., FEDKIW R.: A vortex particle method for smoke, water and explosions. *ACM Transactions on Graphics 24*, 3 (2005), 910–914.

[Van07] VANZINE Y.: *Real-Time Volumetric Rendering of Fire in a Production System: Feasability Study*. Master's thesis, Indiana University South Bend, 2007. http://www.cs.iusb.edu/thesis/YVanzine_thesis.pdf.

[XWK02] X. WEI W. LI K. M., KAUFMAN A.: Simulating fire with texture splats. In *IEEE Visualization* (2002), pp. 227–237.