Realistic Real-Time Rendering of Fire in a Production System: Feasibility Study

Thesis Proposal

Yuri Vanzine
Department of Computer Science
Indiana University of South Bend

Email: yuri@vanzine.org

Advisor
Dr. Dana Vrajitoru
Department of Computer Science

Committee:
Dr. Zhong Guan
Dr Michael R. Scheessele

Date: May 23, 2007

Abstract

The thesis presents an effort at developing a robust, interactive framework for rendering 3-D 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. Many of the systems developed for rendering fire in offline mode directly influenced and inspired real-time fire rendering, including this thesis.

Macro-level behavior of fire is characterized by wind fields, temperature and moving sources and is currently processed on CPU while micro-level behavior like turbulence, flickering, shape is created on graphics hardware.

This framework must support two-way interaction with fire, i.e. not only fire is influenced by the environment but also the environment can receive input from fire via collision detection. It must provide a set of tools for level designers to wield complete artistic and behavioral control over fire as part of the scene. The resulting system must be able to scale well, to use as few processor cycles as possible, and efficiently integrate into an existing production environment.

The focus in this work is on testing the feasibility of rendering fire volumetrically within the constraints of a real 3D engine production system. Performance statistics must be collected and the concepts presented in previous work on volumetric fire rendering must be tested and the feasibility of achieving interactive frame rates within a 3D engine framework must be assessed.

Table of Contents

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 3-D fires in contemporary video games use primitive particle emitter systems [10].

They consist of a particle emitter and particles themselves. The emitter, the routine which regulates the change and variations in the behavior and appearance of the particles, releases and evolves the particles. Particles are usually simulated with graphical primitives like pixels or polygons. A particle system is basically an algorithm for simulating a class of fuzzy objects [10], including fire, explosions, smoke, flowing water, sparks, falling leaves, etc. 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 my model of fire, I explore volumetric rendering and attempt to take advantage of modern shader hardware, as it has been the state of the art of rendering surfaces since 2001.

Shader instructions are usually small programs which run on Graphics Processing Units (GPU). They calculate geometry and color of objects in the scene. Vertex shaders are typically responsible for position of vertices of triangles where fragment or pixel shaders produce the color of each individual pixel on the surface that is being rendered.

Few production systems go the extra length to model it outside of particle emitters. Fire is difficult to simulate and is computationally expensive. Usually a production system 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. A lot of 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 somewhat realistic but incomplete models.

Figure 1: PC game Half Life 2: Episode 1, by Valve Corporation [27]

Consider Figure 1, a series of screenshots from a PC game Half Life 2: Episode 1, by Valve [27]. The same fire displayed in these three images represents one of the contemporary real-time fire models. The technique uses a flat surface and alpha-blending of the resulting colors into a pseudo-random 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. Notice that the system is incapable of view-aligning to the camera along the y-axis and the third image reveals that. The texture of fire also has a 5-second cycle after which the moving image is repeated. This is an example of production fire that due to its flaws must be used sparingly and 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, i.e. fire modeled with, most frequently, spherical primitives with various levels of transparency. It is found in many games: World of Warcraft [29] by Blizzard Entertainment, Company of Heroes [24] by Relic Entertainment or Guild Wars [26] by ArenaNet. Particle-based fire suffers from appearing atomistic, as opposed to the natural holistic look of fire. One way to deal with it [29] is to enlarge the size of and reduce number of the particles.

Several games feature more sophisticated models of fire, consisting of at least two subsystems. For instance, F.E.A.R. [25] , a horror-themed first-person shooter computer and video game developed 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 flaws within the core system,

however, the core fire system in F.E.A.R. consists of a rigid surface shader fire which does not evolve in shape as time goes by.

In the course of my investigation into real-time volumetric rendering of fire, I would like to research better algorithms of curve-based, macro-micro movement, hardware-accelerated real-time rendering of fire such as have been taking place in the last several years and I also wish to improve performance of some algorithms that have been proposed. For instance, improvements in inverse transformation/free-form deformation are expected, where I propose to use only one curve instead of four and approximate the position of volume boundaries instead of solving the knot equation as neither does it address the macro-curve discontinuity problem when the volume is distorted more than a visually acceptable amount (because with overlapping regions discontinuities will still be visible), nor does it need it to, because I feel that fire turbulence must be represented by local detail rather than the shape of the flame curve.

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. While theoretical research must be done in isolation, in order to succeed, I feel that in order to achieve a successful application of these methods, they must be tested in a real setting.

2. Literature

Movie industry has for a long time utilized quite realistic fire rendering, starting as early as 1985 with the first example of particle emitters [20]. Various such techniques exist in abundance in the literature available. All these techniques require off-line rendering and on average take up as long as two or three minutes of rendering time where real-time fire must be rendered and displayed in 60 frames per second on average in order to look natural. Many of the algorithms in these papers can be either implemented directly in new shader-enabled hardware or broken up into steps for more efficient hardware pipeline execution. Off-line rendering algorithms of fire are divided into three distinct categories: *physics-based*, *particle or texture-based,* and *mixed* [4].

Physics-based algorithms of rendering fire are based on laws of fluid dynamics and represent fire as *hot and turbulent gas* [6,7,8]. A very efficient method of rendering fire this way is by solving volumetric differential equations at low resolutions, known as full 3D finite difference solution of Navier-Stokes equations [6,7,8]. Several authors utilize incompressible Navier-Stokes to model vaporized fuel and hot gas with voxels [17]. Navier-Stokes describes how the velocity of gas changes over time depending on its physical properties, i.e. convection, pressure and drag. It explains how gas convects due to Newton's Laws of Motion and rotates because of drag and thermal buoyancy.

Another approach of volumetric rendering is *Blob-warping* [22]. The algorithm consists of modeling non-uniformly modified density blobs as they are convected (i.e. moved) by a wind field and diffusion processes over time. Blobs, in this case, are spherical particles, whose volume is partially occupied by gas which is normally distributed and rendered using diffusion equations [22].

In the original particle system algorithm [20], William T. Reeves was the first to describe and use particle systems to model fire. In the film *Star Trek II: The Wrath of Khan*, the wall-of-fire element was generated using a two-level hierarchy of particle systems. Due to the discrete nature of particles, a huge amount of them were required to achieve good results. To avoid the computational complexity of large particle systems, King et al. [14] have used textured splats to achieve fire animation. These splat primitives are based on simple and local dynamics, and only a small number of them are required for an animation with a sufficient amount of complexity. In fact, texture splats were so promising that Wei et al.[23] successfully used them in combination with Lattice Boltzmann Model[12] as a less computationally intensive alternative to Navier-Stokes equations, to render fire in real time. This research resulted in being able to display around 100 texture splats and render the entire image in 15ms. Texture Splats were used in a number of games [28], but, in my opinion, that their appearance does not have the holistic look fire must have and that they are better suited to smoke or steam simulation.

Lamorlette and Foster [15] provide a very solid *mixed* framework for macro-movement of fire in a production system. The paper describes the effort behind the mathematical modeling of fire for the motion picture “Shrek” by DreamWorks [15]. Since this model is the best fit for the system I chose to implement, I shall describe it in more detail. The model as a general fire animation tool has eight distinct 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, as described in [1].
2. The curves evolve over time according to a combination of physics-based, procedural, and hand-defined wind fields. Physical properties are based on statistical measurements of natural diffusion flames. The curves are frequently re-sampled to ensure continuity, and to provide mechanisms to model flames generated from a moving source.
3. The curves can break, generating independently evolving flames with a limited lifespan. Engineering observations and stochastic assumptions provide heuristics for both processes.
4. A cylindrical profile is used to build an implicit surface representing the oxidization region, i.e. the visible part of the flame. Particles are point-sampled close to this region using a volumetric falloff function.
5. Procedural noise is applied to the particles in the parameter space of the profile. This noise is animated to follow thermal buoyancy.

6. The particles are transformed into the parametric space of the flame's structural curve. A second level of noise, with a Kolmogorov frequency spectrum (another application of Navier-Stokes equations to turbulence), provides turbulent detail.
7. The particles are rendered using either a volumetric, or a fast painterly method. The fast painterly method is a way to render an image to simulate a painting-like quality with height and opacity maps where brush strokes are visible and appear three-dimensional. The color of each particle is adjusted according to color properties of its neighbors, allowing flame elements to visually merge in a realistic way.
8. To enable control, a number of procedural controls are defined to govern placement, intensity, lifespan, and evolution in shape, color, size, and behavior of the flames.

Wind fields are described in [6,7,8]. The main factor in the motion of the gas is the velocity it has when rushing into the surrounding air. As it mixes with the slower moving air, the hot gas experiences drag (shearing forces), and starts to rotate in some places. This rotation causes more mixing with the air, and results in the characteristic turbulent swirling that we see when gases mix. A second important factor that governs gas motion is temperature. 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. The model presented in the papers is a customized one, because it incorporates only the physical elements of gaseous flow that correspond to interesting visual effects, not those elements necessary for more scientific accuracy.

The model is built around a physics-based framework, and achieves speed without sacrificing realism as follows. A volume of gas is represented as a combination of a scalar temperature field, a scalar pressure field, and a vector velocity field. The motion of the gas is then broken down into two components: 1) convection due to Newton's laws of motion, and 2) rotation and swirling due to drag and thermal buoyancy. The rotational, buoyant, and convective components of gaseous motion are modeled by coupling a reduced form of the Navier-Stokes equations with an equation for turbulent mixing due to temperature differences in a gas. This coupling provides realistic rotational and chaotic motion for a hot gaseous volume.

A reduced form of Navier-Stokes equations [6] is appropriate for modeling of the wind field, where $w(u)$ represents change of velocity of gas in an arbitrary wind field and is expressed as:

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

Where u is 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 position of the particle.

Equation (1) describes how the velocity of gas changes over time depending on convection $(u \cdot \nabla)u$, its pressure gradient ∇p, and drag $v\nabla \cdot (\nabla u)$.

The v coefficient is the kinematic viscosity. Smaller viscosity leads to more rotation in gas.

I draw inspiration for realistic volumetric rendering from research done in [9] and [3]. The methods describe generating procedural volumetric fire in real time using *filtered back projection* [3]. Filtered back projection is a nuclear medicine method to reconstruct a volume image based on information collected from several two-dimensional image projections. Mathematics, describing this method, is known as Radon Transform and Inverse Radon Transform. These transforms perform an integral projection of a 3D function f(x,y,z) onto a plane [3]. By combining curve-based volumetric free-form, inversely parameterized deformation, hardware-accelerated volumetric rendering and Improved Perlin Noise or M-Noise, the authors [9] are able to render a vibrant and uniquely animated volumetric fire. Their system is easily customizable by content artists. The fire is animated both on the macro and micro levels, although I am particularly interested in the authors' approach to micro movement. Micro fire effects such as individual flame shape, location, and flicker are generated in a pixel shader using three- to four-dimensional Improved Perlin Noise or M-Noise (depending on hardware limitations and performance requirements). Their method supports efficient collision detection, which, when combined with a sufficiently intelligent particle simulation, enables real-time bi-directional interaction between the fire and its environment. The result is a three-dimensional procedural fire that is easily designed and animated by content artists, supports dynamic interaction, and can be rendered in real time.

Perlin Noise [19] or Improved Perlin Noise is 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 pseudo-randomly over space and time.

M-Noise [18] 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 lookups, 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.

Inverse Parameterization is used when it is impossible to use forward free-form volumetric deformation. It is a well-known problem that cannot be solved analytically. Its numerical solution requires significant computation and exhibits robustness problems. One cannot, in real time, deform the entire volume of the fire, which has almost infinite amount of detail, but one can deform a number of discrete sub-volumes. Also, shader engine renders onto triangle fragments, thus triangles must cover every point of the transformed fire. I use the lattice method described in [9] to construct a grid around the deformed volume of the fire in world space coordinates and associate each point of the lattice with texture coordinates in the unit fire volume. I then use a 3-D shader flame texture to implicitly interpolate all the points of the fire surfaces.

3. Proposed Solution

A robust parameterized system of rendering fire must be developed that would serve as a generic and flexible way of depicting fire in production systems. Depiction of Fire in this model consists of macro-behavior and micro-detail. For macro-movement, each fire source is represented by a fire skeleton which is either defined by a script or influenced by a number of forces participating in the simulation. These forces include: 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 (as opposed to assembly) 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.

3.1 General Model Outline

My 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. Physical properties are based on statistical measurements of natural diffusion flames. 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. See figure 2.

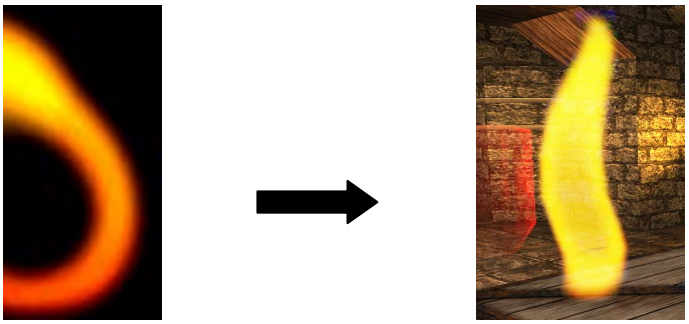

Figure 2. Mapping Color from 2D texture to 3D volume

The particles are transformed into the parametric space of the flame texture using inverse parameterization. M-noise or Improved Perlin noise provides turbulent detail.

4. The particles are rendered as shader fragments using shader hardware acceleration. The color of each triangle is implicitly interpolated according to color properties of its neighbors, allowing flame elements to visually merge in a realistic way.
5. To complete the system, I 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

Curve-based animation in step 1 will be based on Farin [5]. The initial research was done using Hermite Polynomials, which, like many types of Bezier curves, construct a smooth curve and satisfy given endpoint (position and tangent) conditions. Upon reading of sources [5,9], I discovered that other curves, specifically Base Splines or B-Splines lend themselves better to correct mapping of the curve space coordinates to the texture space coordinates. Hermite cubic splines only provide first order derivative continuity, where uniform B-splines guarantee second-order continuity. Second-order continuity is required for curvature to remain continuous and for the curve space to be prevented from twisting randomly. Support of B-splines is known to be defined on domain over [0, 1] regardless of arc-length, which allows to map world coordinates to local texture coordinates. B-Splines are also invariant to affine transformations which include scaling and rotation. The de Boor's algorithm was chosen as a fast and numerically stable B-spline algorithm. It is a generalization of the de Casteljau's algorithm for Bezier curves. The algorithm was devised by Carl R. de Boor. It is also known as Cox-deBoor algorithm [5].

One wants to evaluate the spline curve for a parameter value $x \in [u_l, u_{l+1}]$.

We can express the curve as

$$\boldsymbol{s}(x) = \sum_{i=0}^{p-1} \boldsymbol{d}_i \cdot N_i^n(x) \tag{2}$$

where

$$N_i^n(x) = \frac{x - u_i}{u_{i+n} - u_i} \cdot N_i^{n-1}(x) - \frac{x - u_{i+n+1}}{u_{i+n+1} - u_{i+1}} \cdot N_{i+1}^{n-1}(x) \tag{3}$$

And

$$N_i^0(x) = \begin{cases} 1 & if \quad x \in [u_l, u_{l+1}] \\ 0 & otherwise \end{cases} \tag{4}$$

Due to the spline locality property,

$$\boldsymbol{s}(x) = \sum_{i=l-n}^{l} \mathbf{d}_i \cdot N_i^n(x) \tag{5}$$

So the value $\boldsymbol{s}(x)$ is determined by the control points $\boldsymbol{d}_{l-n}, \boldsymbol{d}_{l-n+1}, \ldots, \boldsymbol{d}_l$; the other control points $\boldsymbol{d}_i$ have no influence. DeBoor's algorithm, described in the next section, is a procedure which efficiently evaluates the expression for $\boldsymbol{s}_x$. Here *p* is the number of control points, the $\boldsymbol{u}_i$ values are the knot values which are uniformly distributed between 0 and 1, $N_i^n(x)$ is the recursive B-spline basis function and $\boldsymbol{d}_i$ is a control point.

3.3 Micro and Macro Animation of the Flame

Physics-based controls in Step 2 govern macro-movement of flame spines according to the system of equations in [15]. The primary equation of motion is

$$\frac{dx_p}{dt} = w(x_p, t) + d(T_p) + V_p + c(T_p, t) \tag{6}$$

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:

$$c(T_p, t) = -\beta g_y (T_o - T_p) t_p^2 \quad (7)$$

Where β is the thermal coefficient, g is gravity, T_o is the ambient temperature and t_p is the age of the particle.

In order to create an arbitrary wind field, I intend on using simplified Navier-Stokes equations to simulate convection and macro drag from [6].

3.4 Volumetric Rendering

To render volumetric fire I use a lattice and volume-slicing technique first described in [3] and perfected for shader hardware in [9]. See figure 3.

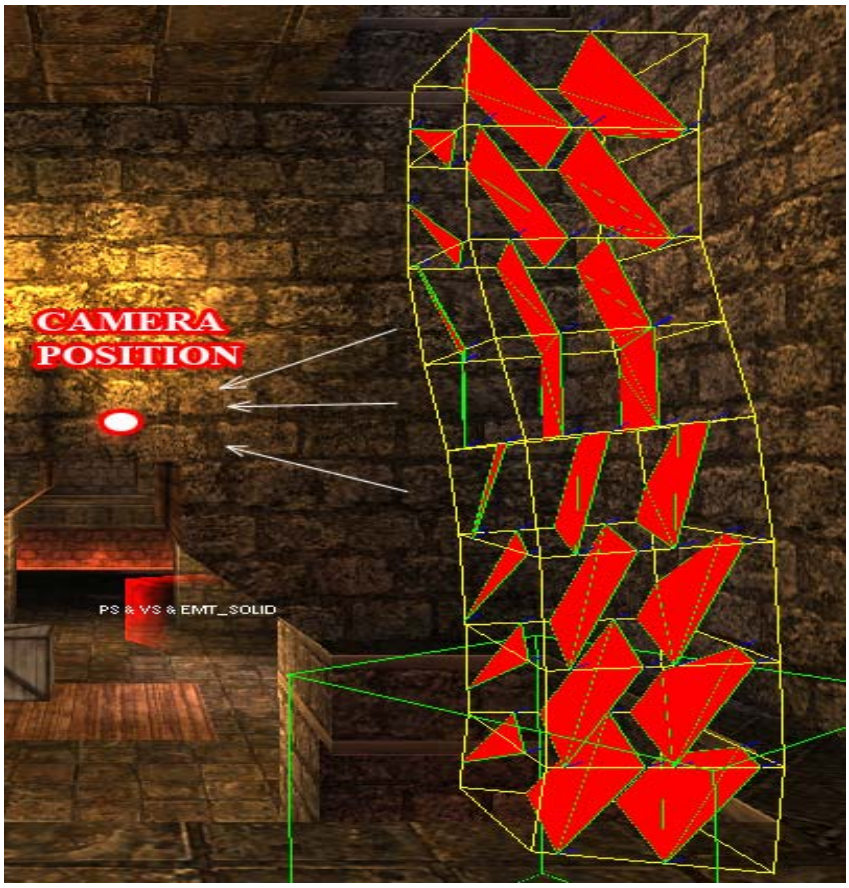

Figure 3. View-aligned slices of the lattice facing camera position

Every time the volume of fire is rendered it is broken up into an arbitrary number of sub-volumes corresponding to knots in the Cox-DeBoor interpolation. Each sub-volume is then sampled into evenly spaced view-aligned slices using a highly optimized cube-slicing algorithm. Each slice is then triangularized efficiently for processing in the shader hardware. Each triangle is rendered through a pixel shader after obtaining color from the fire texture. Because the unit of fire exists in the xyz-space between [-1,1] on the x and z axis and between the current knot-value and the next knot-value, I extract the local coordinates of the points of triangle primitives making up the volume slices and pass this vertex information to the shader.

I then perform a texture lookup from the xyz-coordinates provided as follows:

$$\boldsymbol{xy} = \left(\sqrt{x^2 + z^2}, y\right) \tag{8}$$

Looking up X and Y coordinates in the texture gives us the color value which the pixel shader must display. Pixel colors in the fire slices are added as they are displayed similarly to a ray tracer that uses evenly spaced samples. Additive blending comes in handy here where translucency is handled to account for fire's emissive character.

This model will be implemented in the Irrlicht 3-D engine [10] and rendered in both OpenGL [21] and DirectX [2]. For shader support I will use Open GL Shader Language (GLSL) [21] and High Level Shader Language (HLSL) [2]. Microsoft Visual Studio 2005 will be used to create and debug the program.

4. Feasibility Study

As I focus on testing the possibility of rendering fire volumetrically within the constraints of a real 3D engine production system, I must assess the feasibility of achieving interactive frame rates within such framework. I also provide interactive controls for physical and procedural parameters of the simulation. As a goal of my feasibility study I set out to collect substantial statistics and measure performance of the volumetrically rendered fire. Such performance depends on the various degrees of freedom that exist in my framework as well as in the 3D engine framework.

For the 3D engine they include the underlying rendering engine(s) used to render 3D primitives, i.e. OpenGL/GLSL [21], DirectX/HLSL [2] or the *Burning Video* Software Rasterizer [11] and the number of geometric primitives already present in the scene, representing the complexity of the environment in which fire is displayed. 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 number of flame volumes in the scene, lattice resolution (density and slice spacing) of the volume, screen size of fire, number of noise octaves and most importantly, graphics hardware.

All of these factors influence the system performance. I will explore the feasibility of the model and describe how to efficiently leverage combinations of these parameters.

Conclusion

As a result of this thesis, an easily expandable fire-rendering framework must be written using the Irrlicht 3-D engine [11], utilizing established and novel algorithms. Improved algorithms will be described. Physical and procedural controls will be leveraged to explore performance of the fire model. Statistics will be collected to determine system bottlenecks and to test the feasibility of the model. I will determine if the 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.

Figure 4 shows the framework for rendering fire using the methods I outlined.

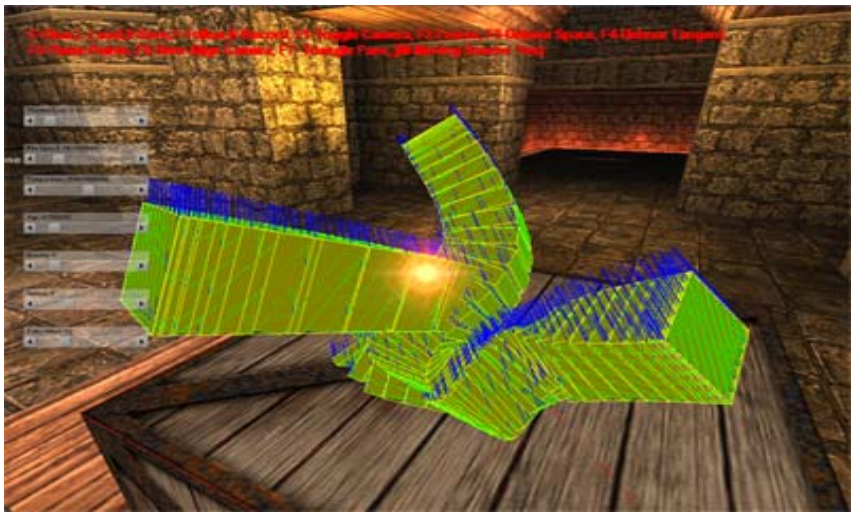

Figure 4: Prometheus: Fire Framework for Feasibility Study

The background of this image is an octree map (tree data structure where each node has up to eight children, which is a common model for 3D environments) of a medieval castle. I will be using this octree map as the necessary attribute of a true production system. I will collect the count of frames per second in the context of a production system.

3 individual curved volumes are displayed in this image. Each represents a standalone fire or a flame, depending on the character of the fire. Each volume is made up of sub-volumes, displayed in yellow. Each sub-volume is sliced into view-aligned triangles, which are surfaces onto which the pixel shader engine renders color pixels, corresponding to the areas of the flame texture. These surfaces are rendered using additive blending, in which color of each back surface is added to the color of the front surface as surfaces are rendered, traditionally in 3D rendering environments, back to front, creating a volumetric look we strive for.

References

1. P. Beaudoin, S. Paquet, P. Poulin, Realistic and Controllable Fire Simulation, *Proceedings of Graphics Interface* 2001, June 2001.
2. D. Blythe, *DirectX*, The Direct3D 10 System, Microsoft Corporation, 2006.
3. B. Cabral, N. Cam, J. Foran, Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. *Proceedings of the 1994 Symposium on Volume Visualization,* ACM Press, New York, NY, USA, 91-98.
4. Y. Eyman, Rediscovering Fire: A Survey of Current Fire Models and Applications to 3-D Studio Max. *Independent Study*, University of Maryland, Fall 2003 - Spring 2004.
5. G. Farin, *Curves and Surfaces for Computer-Aided Geometric Design,* fifth ed. Academic Press, San Diego, CA, USA, 2002.
6. N. Foster and D. Metaxis, Modeling the Motion of a Hot, Turbulent Gas. Center for Human Modeling and Simulation University of Pennsylvania, Philadelphia, *Proceedings of Special Interest Group in Graphics Conference (SIGGraph),* Association for Computing Machinery, Inc. (ACM), 1997.
7. N. Foster and D. Metaxis, Controlling Fluid Animation, *Proceedings of the 1997 Conference on Computer Graphics International*, 1997**.**
8. N. Foster and D. Metaxas, Realistic Animation of Liquids, *Graphical Models and Image Proc*., 58(5), 1996, pp. 471–483.
9. A. R. Fuller, H. Krishnan, K. Mahrous, et al., *Real-time Procedural Volumetric Fire*, Institute of Data Analysis and Visualization and Department of Computer Science, University of California, Davis, Davis, CA 95616, 2006.
10. K. Hawkins, D. Astle, *OpenGL Game Programming*. ISBN 0-7615-3330-3 Chapter 15 Special Effects. Using Particle Systems.
11. *Irrlicht 3-D engine* (http://irrlicht.sourceforge.net/).
12. B. D. Kandhai. *Large Scale Lattice-Boltzmann Simulations*, PhD thesis, University of Amsterdam, December 1999.
13. J. Kessenich, D. Baldwin and R. Rost, *The OpenGL Shading Language*. Version 1.10.59. 3Dlabs, Inc. Ltd.
14. S. A. King, R. A. Crawfis, and W. Reid. Fast Volume Rendering and Animation of Amorphous Phenomena. *Volume Graphics*, pages 229–242, 2000.
15. A. Lamorlette, N. Foster, *Structural Modeling of Flames for a Production Environment*, DreamWorks, Association for Computing Machinery, Inc. (ACM), 2002.
16. D. Nguyen, D. Enright and R. Fedkiw, *Simulation and Animation of Fire and Other Natural Phenomena in the Visual Effects Industry*. Western States Section, Combustion Institute, Fall Meeting, UCLA, 2003.
17. D. Nguyen, R. Fedkew, H. Jensen, *Physically Based Modeling and Animation of Fire*, Association for Computing Machinery, Inc. (ACM), 2002.

18. M. Olano, *Modified Noise for Evaluation on Graphics Hardware*, Graphics Hardware 2005.
19. K. Perlin, Improving Noise, *International Conference on Computer Graphics and Interactive Techniques*, Proceedings of the 29th annual conference on Computer graphics and interactive techniques, Pages: 681–682, San Antonio, Texas, 2002.
20. W. T. Reeves, *Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems*. Association for Computing Machinery, Inc. (ACM), Volume 19,Number 3, Pages: 313 – 322,1985
21. R. J. Rost, *OpenGL 2.0 Overview*, 3Dlabs, Inc., February 2002
22. J. Stam, E. Fiume, *Depicting Fire and Gaseous Phenomena Using Diffusion Processes*, Department of Computer Science, University of Toronto, Association for Computing Machinery, Inc. (ACM), 1995.
23. X. Wei, W. Li, K. Mueller and A. Kaufman, *Simulating Fire With Texture Splats*, Center For Visual Computing (CVC) And Department Of Computer Science State University Of New York At Stony Brook, NY 11794-4400.
24. *Company of Heroes*, Relic Entertainment, PC, September 14, 2006.
25. *F.E.A.R.*, Monolith Productions, PC, October 18, 2005
26. *Guild Wars*, ArenaNet, PC, April 28, 2005.
27. *Half Life 2*, Episode 1, Valve Corporation, PC, June 2006.
28. *Call of Cthulhu*: Dark Corners of the Earth, Headfirst Productions, October 24, 2005 (Xbox), March 27, 2006 (PC).
29. *World of Warcraft and World of Warcraft: The Burning Crusade*, Blizzard Entertainment, PC, November 23, 2004.