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

Yuri Vanzine

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements for the degree
Master of Sciences in the Department of Computer and Information Sciences
and the Department of Mathematical Sciences,
Indiana University South Bend,
November 2007

Accepted by the Graduate Faculty, Indiana University South Bend, in partial fulfillment of the requirements for the degree of Master of Science.

Master's Thesis Committee

_____
Chairperson, Dana Vrajitoru, Ph.D.

_____
Zhong Guan, Ph.D.

_____
Michael R. Scheessele, Ph.D.

Date of Oral Defense:

_____

This is dedicated to Kristy, Jack and Parker,
Who helped to keep the fire burning.

AKNOWLEDGMENTS

I would like to acknowledge the mathematical and computer science faculty of IUSB for their support and interest to my thesis.

In particular, I would like to thank Dr. Dana Vrajitoru for giving me direction and encouragement, defining the end goal of this research and generally helping through difficult parts of this thesis.

I would like to thank Dr. Zhong Guan for stimulating conversations and helping me understand that big things can come from PRNs.

I would also like to thank Dr. Michael Schesssele for reviewing this thesis from start to finish and ensuring its quality.

"…Has your child ever set a fire on purpose? Does your child show extreme curiosity about fire? Has your child dreamed about fire? Has your child played with fire? Does your child sometimes stare at fire for long periods of time?"

From "Is Your Child a Fire Risk?", an article about Fire Safety in the state of Colorado.

"…Fire has this hypnotic, magnetic effect where you can keep staring at it for long periods of time, simply because one is drawn to a presence of maximum change, maximum release of energy. A piece of wood disappears, using its internal energy to give birth to heat and light… …Fire is a miniature model of the process of life in the universe."

From "Life", Michael Weller

"…Look what I have created! I have made FIRE!"

Chuck Noland, Movie "Cast Away"

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 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.

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 are collected, the concepts presented in previous work on volumetric fire rendering are tested and the feasibility of achieving interactive frame rates within a 3D engine framework is 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 [14].

These systems consist of a particle emitter and the 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 [26], 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 the 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. 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.

Figure 1.1: PC game Half Life 2: Episode 1

Consider Figure 1.1, a series of screenshots from a PC game Half Life 2: Episode 1, by Valve [39]. 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 in a flipbook animation fashion. 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 [41] by Blizzard Entertainment, Company of Heroes [36] by Relic Entertainment or Guild Wars [38] by ArenaNet. Particle-based fire suffers from appearing atomistic, as opposed to the natural holistic look of fire. One

way to deal with it is to enlarge the size of and reduce number of the particles. Figure 1.2 shows a blob torch fire and a campfire from World of Warcraft.

Figure 1.2: A blob torch fire and a campfire from World of Warcraft.

Several games feature more sophisticated models of fire, consisting of at least two subsystems. For instance, F.E.A.R. [37], 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. See example of F.E.A.R. fire in Figure 1.3 below.

Figure 1.3. A rigid shader fire from F.E.A.R.

In the course of my investigation into real-time volumetric rendering of fire, I researched 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 improved the performance of some algorithms that have been proposed. Simplifications of previous models were made to allow for a sufficiently fast real-time rendering. For example, I proposed to use only one curve instead of four in the inverse transformation/free-form deformation step and to approximate the position of volume boundaries instead of solving the knot equation as it does not address the macro-curve discontinuity problem when the volume is distorted more than a visually acceptable amount. Also, in the model used in [13] and in my model, I observed discontinuities still present when neighboring subvolumes overlap and vary in size wildly (due to, for instance, high temperature of fire in the physics simulation). Fire turbulence can only be efficiently represented by local detail rather than by the global character of the fire volume. Invalid regions of macro flame movement must be carefully avoided as a curve with curvature

larger than a certain threshold will produce unacceptable visual results. It must be merely approximated by eliminating the invalid sections.

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 in this thesis, I 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.

2. Literature Review

Movie industry has for a long time utilized quite realistic fire rendering, starting as early as 1985 with the first example of particle emitters [26]. Various such techniques exist in abundance in the available literature. 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 be 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* [8].

Physics-based algorithms of rendering fire are based on the laws of fluid dynamics and represent fire as *hot and turbulent gas* [10,11,12]. 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 [10,11,12]. Several authors utilize incompressible Navier-Stokes to model vaporized fuel and hot gas with voxels [21]. 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* [29]. 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.

In the original particle system algorithm [26], 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. [17] 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.[30] successfully used them in combination with the Lattice Boltzmann Model [15] 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 [37, 40], but, in my 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 [25] 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. 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 equation (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. Usually, such a function outputs a value between 0 and 1 and denotes opacity of a color value. A Gaussian volumetric fallout function, commonly used, would exponentially blend the underlying pixel color and the flame's

color inside and close to the volume, using the position of the flame, position of the blended pixel and the distance from the center of the volume.

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 (expensive stochastic model of turbulent flow used for linear advection equations), 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 [10,11,12]. 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 [10] 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:

$$w(u) = v\nabla \cdot (\nabla u) - (u \cdot \nabla)u - \nabla p \tag{1}$$

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 $v\nabla \cdot (\nabla u)$. The v coefficient is the kinematic viscosity. Smaller viscosity leads to more rotation in the gas.

I draw inspiration for realistic volumetric rendering from research done in [6] and [13]. The methods describe generating procedural volumetric fire in real time using *filtered back projection* [6]. Filtered back projection is a nuclear medicine method to reconstruct a volume image based on information collected from several two-dimensional image projections. The mathematical description of this method is known as the Radon Transform and its opposite is the Inverse Radon Transform. These transforms perform an integral projection of a 3D function *f(x,y,z)* onto a plane. By combining curve-based volumetric free-form, inversely parameterized deformation, hardware-accelerated volumetric rendering and Improved Perlin Noise or M-Noise, the authors [13] 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 [25] 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 [23] 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 [13] 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. Modeling Fire

A robust parameterized system of rendering fire has been 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

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 3.1.

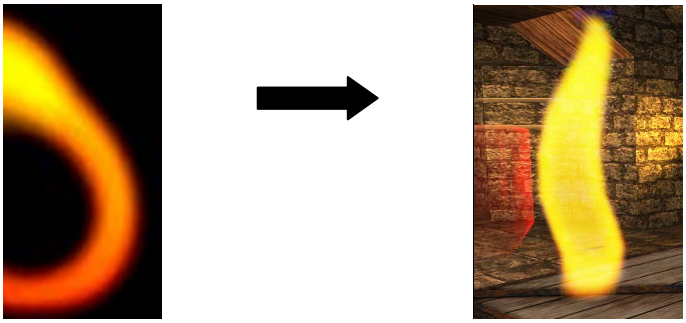


Figure 3.1. Mapping Color from 2D texture to 3D volume

The particles are transformed into the parametric space of the flame texture using inverse parameterization. M-noise or Improved Perlin noise provides 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, I define a number of procedural controls to govern placement, intensity, lifespan, and evolution in shape, color, size, and behavior of the flames.

In the next section I introduce the mathematical definitions of the components of the model. Figure 3.2 shows a set of points interpolated as a B-spline. This curve represents the basic model for the spine of the fire.

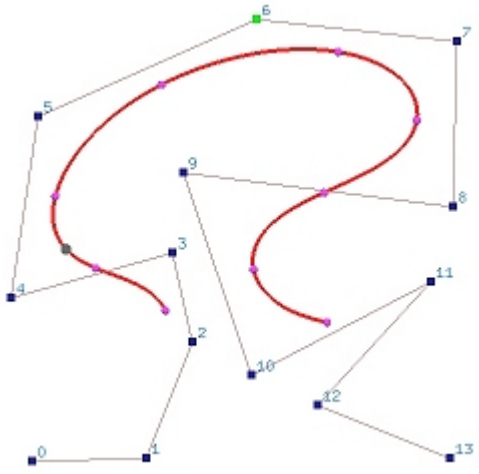


Figure 3.2. B-Spline Curve

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. Such a curve can be constructed in a variety of ways. The common name for such a parametric curve which interpolates points and provides second-order continuity is a Bezier curve. I am interested in a specific type of a polynomial curve that fulfills certain requirements that other types of smoothly interpolating curves do not. My curve must lie in the [0,1] domain, so that the global volume can be divided into discrete sub-volumes, and such that the value of the arc-length can be easily computed for each point. The arc-length is essential in the modeling process because it allows me to easily map volume world coordinates to local texture coordinates. Additionally, the curve must have *local* control only (only a limited set of neighbor points affects the curve at the given point) and not be at the mercy of the entire set of control points.

Initially I considered the Hermite polynomial as the first available algorithm to interpolate a set of points. A simple algorithm worked intuitively by interpolating through the control points. It did not, however, provide a method to re-distribute the points uniformly and without an efficient way to determine arc-length, it was not very useful. Upon review of other curves, B-splines were singled out, because of their knot definition on any given interval.

A B-spline (or Basis Spline) is a spline function (special function defined piecewise by polynomials) that has minimal support with respect to a given degree, smoothness, and domain partition. DeBoor's *fundamental* (or main) *theorem* [45] states that every spline function of a given degree, smoothness, and domain partition can be represented as a linear combination of B-splines of that same degree and smoothness, and over that same partition.

Consultation of [9] revealed that Hermite polynomial basis functions can be negative leading to numerically unstable non-convex combinations. Also, Hermite polynomials are not invariant under affine parameter transformations; the fact that leads to an error faster than if the parameters were affinely invariant. Hermite cubic splines only provide first order derivative

continuity, where uniform B-splines guarantee second-order continuity. Second-order continuity is required for the curvature to remain continuous and for the curve space to be prevented from twisting randomly.

I must define barycentric combinations and affine maps before proceeding. These constitute the basic elements of B-splines.

Barycentric combinations are addition-like operations defined for points. They are weighted sums of points where the weights sum to one:

$$\mathbf{b} = \sum_{j=0}^{n} \alpha_j \mathbf{b}_j \; ; \; \mathbf{b}_j \in \mathrm{E}^3, \; \alpha_0 + \ldots + \alpha_n = 1 \tag{2}$$

In the notation of [9] E^3 is the 3-Dimensional Euclidean space which *points* are elements of. R^3 is the 3-Dimensional vector space which *vectors* are elements of. This formula can be rewritten to be the sum of a point and a vector:

$$\mathbf{b} = \mathbf{b}_0 + \sum_{j=0}^{n} \alpha_j (\mathbf{b}_j - \mathbf{b}_0)$$

An example of a barycentric combination is the centroid **g** of a triangle with vertices **a,b,c**, given by

$$\mathbf{g} = \frac{1}{3}\mathbf{a} + \frac{1}{3}\mathbf{b} + \frac{1}{3}\mathbf{c}$$

The term barycentric combination is derived from “barycenter”, meaning “center of gravity”. The origin of the formulation is in physics: if the $\mathbf{b}_j$ are centers of gravity of objects with masses m_j, then their center of gravity **b** is located at

$$\mathbf{b} = \frac{\sum m_j \mathbf{b}_j}{\sum \mathrm{m}_j}$$

What is an affine map? A map Φ that maps E^3 into itself is called an affine map if it leaves barycentric combinations invariant. So, algebraically, if

$$\mathbf{x} = \sum \alpha_j \mathbf{a}_j \;\; ; \sum \alpha_j = 1, \; \mathbf{x}, \; \mathbf{a}_j \in E^3$$

and Φ is an affine map, then the following must be true:

$$\mathbf{\Phi x} = \sum \alpha_j \mathbf{\Phi} a_j \;\; ; \;\; \mathbf{\Phi x}, \; \mathbf{\Phi a}_j \in E^3 \tag{3}$$

This can also take on a more familiar form:

$$\mathbf{\Phi x} = A\mathbf{x} + \mathbf{v} \tag{4}$$

Some examples of affine maps are simple operations like scaling, rotation and translation.

Numerically stable construction of Basis splines by means of blossoms was independently developed by de Casteljau and Ramshaw [9]. A blossom is a function that can be applied to any polynomial, but mostly is used for Bezier and spline curves and surfaces, because it has the following properties:

- It is a symmetric function of its arguments
- It is affine in each argument
- It satisfies the diagonal property, which states that passing several identical points will result in the same point as their blossom

Consider the following interpolation of points involving ratios *t* and *s* as shown in Figure 3.3.

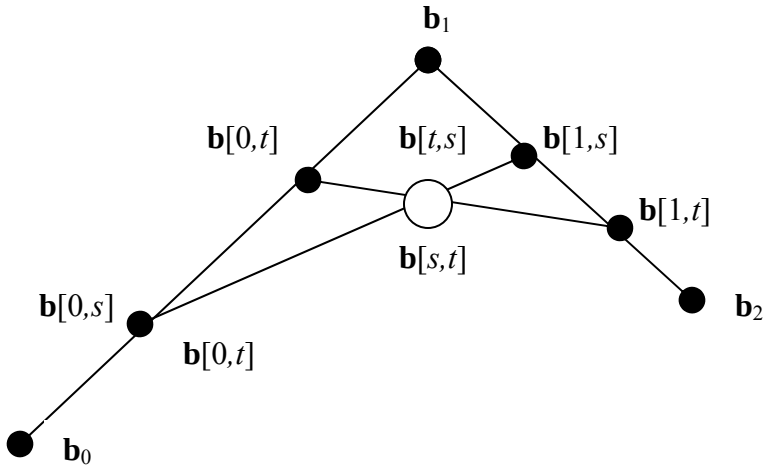


Figure 3.3. The point **b**[s,t] may be obtained from linear interpolation at *t* or *s*.

Referring to figure 3.3, I define

$$\mathbf{b}[0,t] = (1-t)\mathbf{b}_0+t\mathbf{b}_1,$$
$$\mathbf{b}[0,s] = (1-s)\mathbf{b}_0+s\mathbf{b}_1,$$
$$\mathbf{b}[1,t] = (1-t)\mathbf{b}_1+t\mathbf{b}_2,$$
$$\mathbf{b}[s,1] = (1-s)\mathbf{b}_1+s\mathbf{b}_2$$

By Menelaos' theorem [43], the following two points are identical:

$$\mathbf{b}[s,t] = (1-t)\mathbf{b}[s,0]+t\mathbf{b}[s,1]$$
and
$$\mathbf{b}[t,s] = (1-s)\mathbf{b}[0,t]+s\mathbf{b}[t,1]$$

These functions **b**[*s*,*t*], **b**[*t*,*s*] are examples of blossoms.

B-Splines possess several important properties which make them fit for the purpose of volumetric rendering:

1. Affine invariance. For example, one can first compute a point *b(t)* and then apply an affine map to it; or one can apply an affine map to the control polygon and then evaluate the mapped polygon at *t* and obtain the same value.
2. Invariance under affine parameter transformations. Geometrically, it does not matter if the curve is defined over [0,1] with parameter *t* or over [*a,b*] with parameter *u*, because it uses *ratios only* to interpolate control points. Algebraically, the parameter invariance property is defined as follows, where $B_i^n(t)$ is the so-called control point (de Boor point) or knot of a B-spline:

$$\sum_{i=0}^{n} \mathbf{b}_i B_i^n(t) = \sum_{i=0}^{n} \mathbf{b}_i B_i^n\left(\frac{u-a}{b-a}\right) \tag{5}$$

This property is true, because the ratios are grounded in the affine invariance of the linear interpolation.

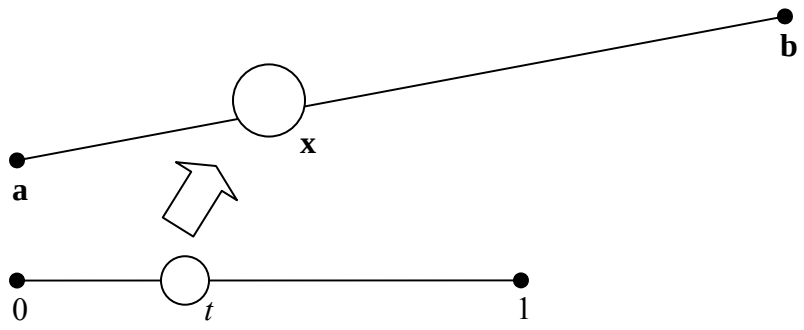


Figure 3.4 Linear Interpolation. Two points **a**,**b** define a straight line between them. The point *t* in the 1D domain is mapped to the point **x** in the range.

By the definition of affine maps, the function computing the point ***x*** based on the parameter *t*, defined as x=(1-*t*)**a**+*t***b** is an affine map of the three 1D points 0,*t*,1! This linear interpolation is an affine map of the real line onto a straight line in E^3.

3. Convex Hull Property. For $t \in [a, b]$, $\mathbf{b}^n(t)$ (a set of blossomed, interpolated values) lies in the convex hull of the control polygon.

The notation <n> or <r> in the exponent stands for how many repeated knots the blossom has, also known as the multiplicity of the knot. The multiplicity of the knot is also equivalent to the continuity of the curve. For example, C^3 continuity of the cubic curve is ensured by three knot values of the blossom notation.

This fact that the blossomed values are always found inside the convex hull of the control polygon is clear from the definition of blossom earlier, since every intermediate point is obtained as a convex barycentric combination of previous points. An interpolation algorithm that uses this property never produces points outside of the convex hull of the total points at any step.

B-spline curves consist of a sequence of polynomial curve segments. Let's consider one such segment and define a point interpolated between u_0, u_1, u_2, u_3 using blossoms. The quadratic blossom $\mathbf{b}[u,u]$ (linear interpolation of these four points) may be written as

$$\mathbf{b}[u, u] = \frac{u_2 - u}{u_2 - u_1}\mathbf{b}[u_1, u] + \frac{u - u_1}{u_2 - u_1}\mathbf{b}[u, u_2]$$

$$= \frac{u_2 - u}{u_2 - u_1}\left(\frac{u_2 - u}{u_2 - u_0}\mathbf{b}[u_0, u_1] + \frac{u - u_0}{u_2 - u_0}\mathbf{b}[u_1, u_2] \right)$$

$$+ \frac{u - u_1}{u_2 - u_1}\left(\frac{u_3 - u}{u_3 - u_1}\mathbf{b}[u_1, u_2] + \frac{u - u_1}{u_3 - u_1}\mathbf{b}[u_2, u_3] \right)$$

I successively express u in terms of intervals of growing size. Starting with the $\mathbf{b}[u_i,u_{i+1}]$ as input control points, I may express $\mathbf{b}[u,u]$ (in the blossom notation) as points further and further refined (quadratic in this case):

$\mathbf{b}[u_0,u_1]$

$\mathbf{b}[u_1,u_2]$ $\mathbf{b}[u_1,u]$

$\mathbf{b}[u_2,u_3]$ $\mathbf{b}[u,u_2]$ $\mathbf{b}[u,u]$

The blossom notation used here may require explanation. The above expression of $\mathbf{b}[u,u]$ should be read as a progression of interpolations from left to right. So $\mathbf{b}[u_1,u]$ in the second column is a result of interpolation between $\mathbf{b}[u_0,u_1]$ and $\mathbf{b}[u_1,u_2]$ where u_1 is kept as a common blossom element and u is a newly determined point. The third column contains the final blossom value $\mathbf{b}[u,u]$. It is an interpolation of $\mathbf{b}[u_1,u]$ and $\mathbf{b}[u,u_2]$ which, this time, share u and participate in obtaining the second interpolated u. The ratio part is omitted from this notation for sake of notation's brevity.

B-spline curves consist of a sequence of polynomial curve segments. Let U be an interval $[u_I, u_{I+1}]$ in a sequence $\{u_i\}$ of knots. There are ordered sets U_i^r of successive knots, each containing u_I and u_{I+1}. The set U_i^r is defined as follows:

U_i^r consists of $r + 1$ successive knots.

U_I is the $(r - i)$ th element of U_i^r, with $i = 0$ denoting the first element of U_i^r.

A degree n curve segment corresponding to the given interval is given by $n + 1$ control points $\mathbf{d}_i$ which are defined by

$$\mathbf{d}_i = \mathbf{b}\left[U_i^{n-1} \right] \; ; \; i = 0,..., n.$$

The point $\boldsymbol{x}(u) = \mathbf{b}[u^{<n>}]$ on the curve is recursively computed as in the following equation:

$$\mathbf{d}_i^r(u) = \mathbf{b}\left[u^{<r>}, U_i^{n-1-r} \right]; \; r = 1,..., n; \; i = 0,..., n - r$$

with $\boldsymbol{x}(u) = \mathbf{d}_0^n(u)$.

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 [9].

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

$$\boldsymbol{s}(x) = \sum_{i=0}^{p-1} \boldsymbol{d}_i \cdot N_i^n(x) \tag{6}$$

where

$$N_i^n(x) = \frac{x - u_i}{u_{i+n} - u_i} \cdot N_i^{n-1}(x) - \frac{x - u_{i+n+1}}{u_{i+n+1} - u_{i+1}} \cdot N_{i+1}^{n-1}(x) \tag{7}$$

and

$$N_i^0(x) = \begin{cases} 1 & if \quad x \in [u_l, u_{l+1}] \\ 0 & otherwise \end{cases} \tag{8}$$

Due to the spline locality property,

$$\boldsymbol{s}(x) = \sum_{i=l-n}^{l} \mathbf{d}_i \cdot N_i^n(x) \tag{9}$$

So the value **s** (x) is determined by the control points $\mathbf{d}_{l-n}$, $\mathbf{d}_{l-n+1}$, ..., $\mathbf{d}_l$; the other control points $\mathbf{d}_i$ have no influence. DeBoor's algorithm is a procedure which efficiently evaluates the expression for $\mathbf{s}_x$. Here p is the number of control points, the $\mathbf{u}_i$ 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 $\mathbf{d}_i$ is a control point.

To define, the local frame of reference, the next step after building the curve is defining a bounding volume around the curve. This volume is composed of hexahedrons defined continuously at regular intervals of the curve. These hexahedrons are determined by a special local coordinate reference frame that must evolve smoothly along the knot points to give impression of volumetric continuity. I use the tangent at each uniformly sampled knot of the basis curve (green arrow), as shown in Figure 3.5 below. I take the cross product of the tangent of the current and next knot (blue arrow) and obtain the x-axis of the local space (the red dot represents the arrow pointing at the camera). The tangent is the y vector of the local space at the control point (green arrow). The curvature, obtained via the cross product of vectors x and

y is the z vector of the local space at the control point (yellow arrow). This special local coordinate system, linked to each control point $\boldsymbol{x}(t)$ is known as the Frenet Frame [9]. I use this local coordinate system to define each subvolume.

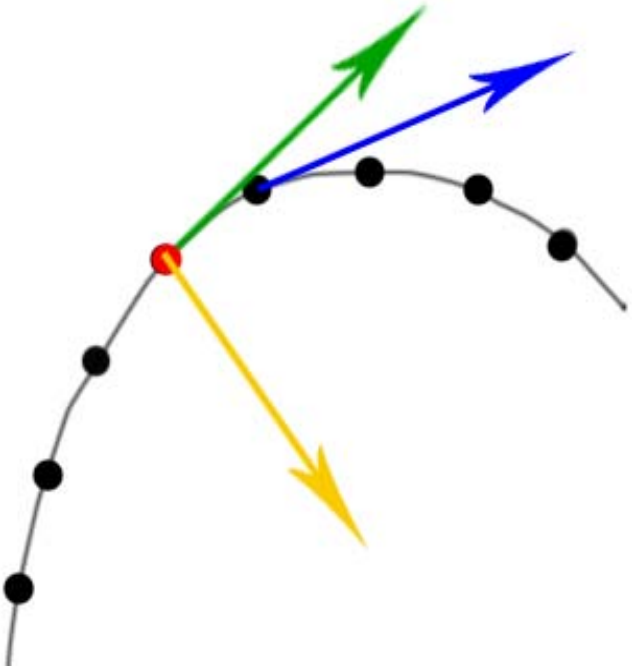


Figure 3.5. Unit vectors of the local subvolume space.

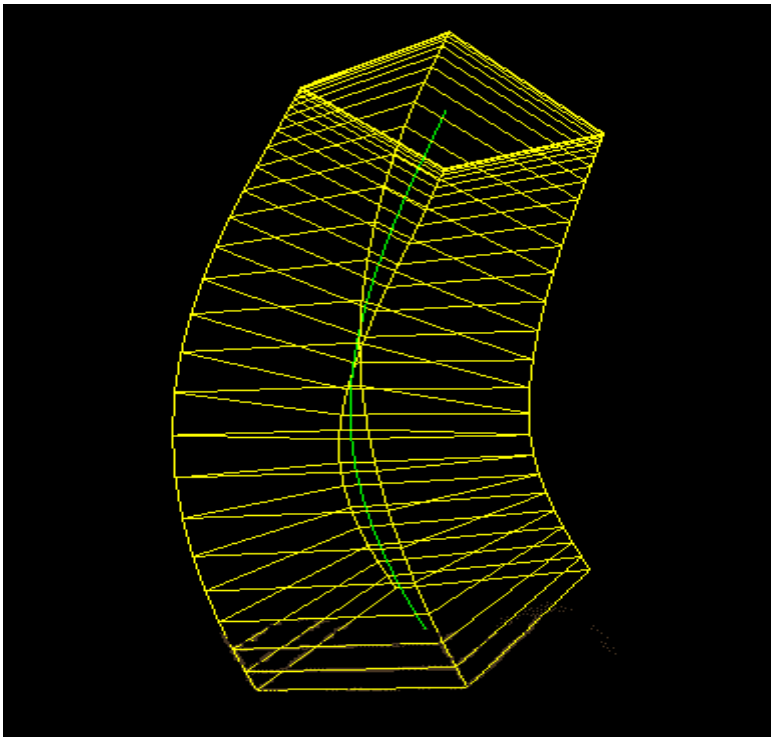


Figure 3.6. B-spline appearing in green represents the flame spine. Deformed hexahedrons wrapped around the curve define object subvolumes.

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 [15]. The primary equation of motion is

$$\frac{dx_p}{dt} = w(x_p, t) + d(T_p) + V_p + c(T_p, t) \tag{10}$$

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:

$$c(T_p, t) = -\beta g(T_o - T_p)t_p^2 \tag{11}$$

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 [10]. In many mathematical models that rely on Navier-Stokes, solutions exist for highly simplified gas flow situations where certain terms in the equations have been eliminated through a rational process. In the above formulation, it is assumed that a) motion due to molecular diffusion is negligible relative to other effects. Also, in some extreme gaseous phenomena there may also be motion caused by shock and pressure waves that arise because gas can be locally compressed. If so, the class of effects is restricted

to common effects such as smoke, fire, or steam under the simplifying assumption that locally, b) 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 I use a lattice and volume-slicing technique first described in [6] and perfected for shader hardware in [13]. An example of such a lattice can be seen in Figure 3.7, 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 bottom perpendicular to the tangent of the curve at the current control point.

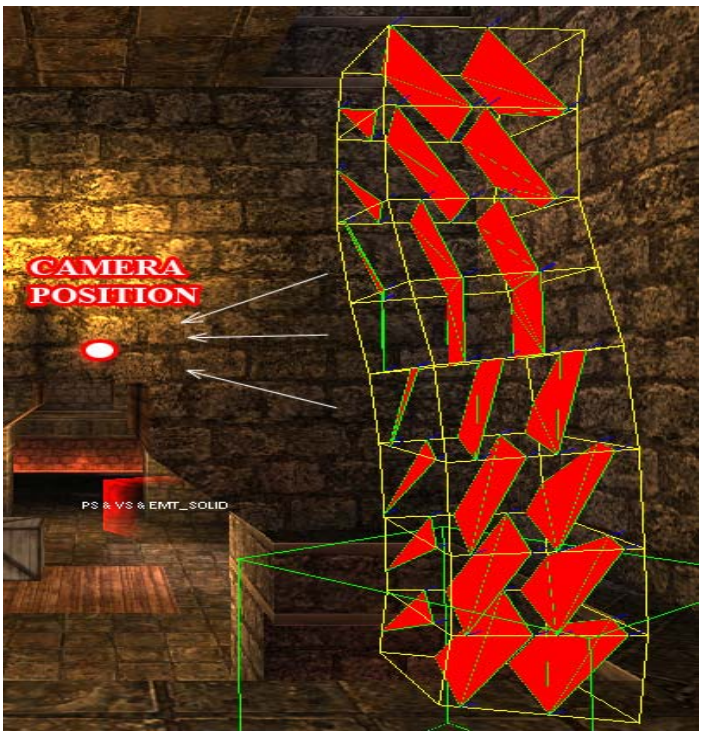


Figure 3.7. View-aligned slices of the lattice facing camera position.

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. This number is controlled in

my simulation and is 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 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 on the y-axis, I extract the local coordinates of the points of the 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:

$$\boldsymbol{x}_t \boldsymbol{y}_t = \left(\sqrt{x^2 + z^2}, y\right) \tag{12}$$

The values x_t and y_t represent coordinates in the texture and they provide the color value that the pixel shader must display for the point (*x, y, z*). Pixel colors in the fire slices are added as they are displayed by a method similar to a ray tracer that uses evenly spaced samples. Additive blending is useful here where translucency is handled to account for the fire's emissive character. These algorithms are explained in detail in the next chapter.

This model has been implemented in the Irrlicht 3-D engine [35] and rendered in both OpenGL [27] and DirectX [5]. For shader support I use Open GL Shader Language (GLSL) [16] and High Level Shader Language (HLSL) [5]. Microsoft Visual Studio 2005 has been used to create and debug the program.

4. Algorithms

This chapter presents a more detailed account of the miscellaneous algorithms that have been mentioned throughout chapter 3, and now require a more thorough explanation. Sections 4.1 to 4.5 detail how the individual sub-volumes of the complete deformed fire space are prepared for rendering via the graphics pipeline. Sections 4.6 to 4.9 delve into types of pseudo-random noise used to render turbulent flow of fire.

4.1 Slicing of the Volume

As it is impossible to render the 3D volume source data continuously, I concern myself with discretization and sampling of the data at equal intervals and sparsely enough in order to achieve interactive frame rates. Soon one will be able to efficiently map a 3D volume to the screen coordinates and render it all volumetrically in the GPU hardware, but currently I must use CPU-processed 2D approximations of such space, i.e. slices of volume.

As described in [6], volumetric objects can be efficiently rendered with 2D or 3D textures with a variable degree of precision. Before images can be applied to textures, a volume of space occupied by the object must be sliced into object-aligned or view-aligned 2D planes.

When visualizing medical or scientific data object-aligned slicing of the 3D space is considered precise and appropriate. The number of slices is maximized and the speed of rendering is sacrificed in favor of a more exact representation of volume data. Oftentimes, the discrete nature of such slicing does not interfere with the visualization task as the high level of detail and not the realism is the primary goal.

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.

With such a method of volume slicing, I 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.

The blending technique that I use to account for translucency of the fire texture is widely used in 3D computer graphics and is known as *depth queuing with the over operator* [44]. A brief description of the algorithm follows.

Let us consider a pixel color resulting from the blending of color values of k points belonging to k translucent surfaces. For each point i on surface i, its color is $f_i(\alpha_i, r_i, g_i, b_i)$, where α_i is degree of transparency, and r_i, g_i, b_i are the red, green and blue components of the color value. Then, the color of the blended pixel is defined as

$$f = \sum_{i=0}^{k} c_i \cdot f_i$$

where c_i is a weighting factor defined as follows:

$$\begin{aligned} c_0 &= \alpha_0 \\ c_1 &= \alpha_0(1-\alpha_1) \\ c_2 &= c_1(1-\alpha_2) \\ &\vdots \\ c_k &= c_{k-1}(1-\alpha_k) \end{aligned}$$

In OpenGL the over operator is accomplished with

gl_BlendFunc(GL_ONE, GL_ONE_MINUS_SRC_COLOR);

and in DirectX, the equivalent function calls read:

lpD3Device->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCCOLOR);
lpD3Device->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCCOLOR);

Since c_k depends on all the previous values of c, the rendering algorithm performs this calculation iteratively back to front for all k superimposed objects (slices) with α_k being their level of translucency, as the color of each pixel is obtained.

When slice planes are determined, they must be broken down into rendering pipeline primitives before texturing can occur. This intermediate step is known as p*roxy geometry* generation, explained in the next section.

4.2 Proxy Geometry

The existing graphics pipelines (OpenGL and DirectX) that render 3D polygons operate in graphical primitives which include points, lines, triangles and lists of points, lines and triangles (lists, strips or fans) [31].

A triangle list is a list of isolated triangles, like the one shown in Figure 4.1. They might or might not be near each other. A triangle list must have at least three vertices and the total number of vertices must be divisible by three.

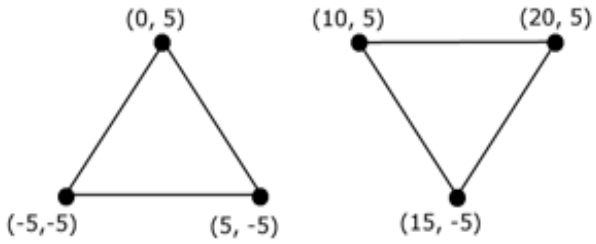


Figure 4.1. Triangle List

A triangle strip is a series of connected triangles. Because the triangles are connected, the application does not need to repeatedly specify all three vertices for each triangle. For example, you need only seven vertices to define the triangle strip in Figure 4.2:

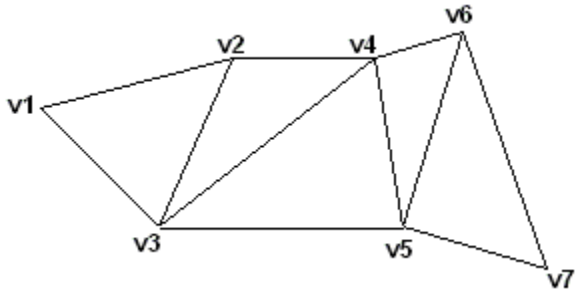


Figure 4.2. Triangle Strip.

A rendering framework uses vertices v1, v2, and v3 to draw the first triangle; v2, v3, and v4 to draw the second triangle; v3, v4, and v5 to draw the third; v4, v5, and v6 to draw the fourth; and so on.

A triangle fan is similar to a triangle strip, except that all the triangles share one vertex. A rendering framework uses vertices v2, v3, and v1 to draw the first triangle; v3, v4, and v1 to draw the second triangle; v4, v5, and v1 to draw the third triangle; and so on.

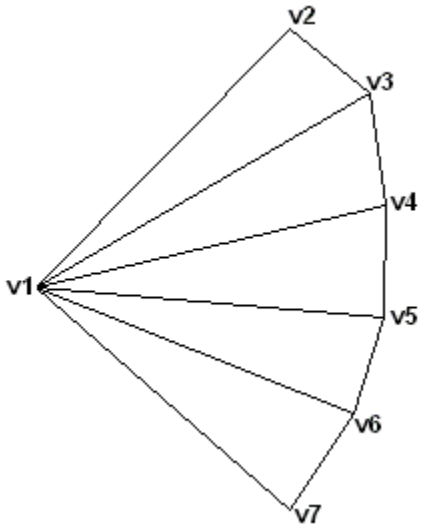


Figure 4.3. Triangle Fan

OpenGL also provides polygons as a surface primitive which enumerates a front-facing set of points clockwise. Any 3D object is represented structurally by a 3D mesh consisting of 3D polygons, which, in turn, are composed of triangles. Because DirectX does not support polygons as graphic primitives, I considered a strip of triangles and settled on a fan of triangles as the common denominator between the two rendering 3D graphics libraries.

4.3 Triangulation

When the slicing plane cuts through the volume, I obtain their intersection points and I am faced with the task of triangulating these points into a triangle fan for the graphics pipeline. This process is called appropriately polygon triangulation and though it appears simple on cursory examination, it is, as a general solution to the problem, 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). Figure 4.4 shows an example of these cases and the decomposition of the resulting polygon in a triangle fan.

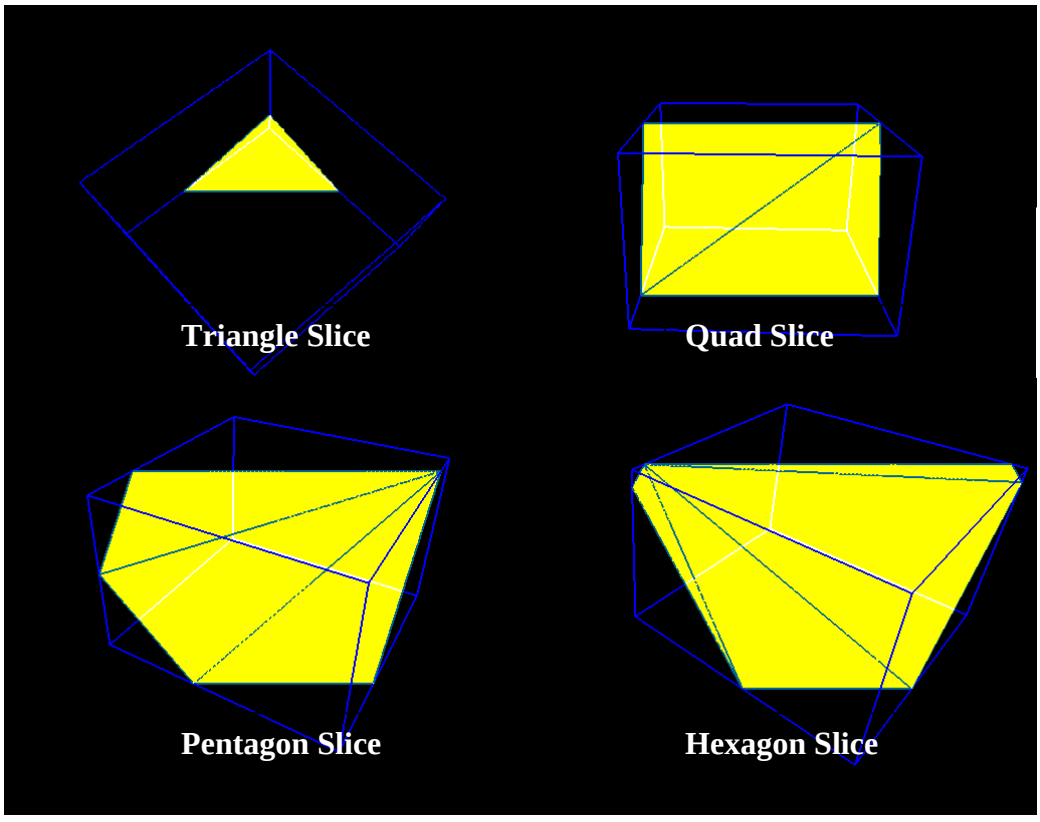


Figure 4.4. Possible intersections of a sub-volume of fire and their triangulations

When one looks at our specific triangulation more closely, one must make the following observations:

- There are always 3, 4, 5 or 6 vertices and between 1 and 4 triangles in a given polygon.
- The polygon is always *convex*, without any holes, *monotone* and *simple*
- The polygon vertices are always found on the edges of the common convex hexahedron (where we identified the intersections of the plane and the bounds of the volume).

Let us define the above polygon types. A simple polygon is a polygon whose sides do not intersect. A convex polygon is a simple polygon whose interior is a convex set. Polygon convexity requires that every internal angle is at most 180 degrees and that every line segment between two vertices of the polygon does not go outside of the polygon. A monotone polygon is one with a boundary that consists of two parts, each of which consists of points that have incrementing coordinates in one dimension.

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*.

Triangulations of polygons can be performed with a variety of algorithms, ranging from simple and slow to the most complex and the most efficient and fastest. One can draw diagonals between non-adjacent vertices (corners) and retain those that don't intersect any other diagonal. The remaining ones compose a triangulation of the polygon, but not necessarily a triangle fan that I chose as my geometry primitive. This algorithm is $O(n^4)$, where n is the number of points. The second possible algorithm is called ear-clipping. An ear of a polygon consists of three consecutive vertices for which no other vertices of the polygon are inside the triangle.

Upon identifying such first ear in the polygon of n vertices, one can clip it off and obtain a polygon of n-1 vertices and repeat the procedure until only the last triangle remains. This lends itself well to a recursive algorithm which can execute in $O(n^2)$ and has a relatively straightforward implementation [28]. A polygon can be partitioned into monotone polygons in O(n log n) time. There exist several randomized algorithms [1,28] which compute trapezoid decompositions of a simple polygon, leading to $O(n)$ complexity of the triangulation; as well as Chazelle's celebrated first-ever linear-time algorithm [7].

4.4 Convex Hull

I decided, for the most part, to ignore these more sophisticated algorithms and focus on the observations that I made above, when I obtained the polygon. The fact that my polygon is a simple, monotone and convex one must play a crucial role in constructing an efficient triangulation algorithm.

I start with the implementation of a convex hull algorithm. The algorithm has the following steps:

- Compute the centroid of the proxy polygon as the average of all the points.
- Consider a vector from world origin to the centroid and add to it the double in length of the vector from centroid to one of the points. The point obtained this way will be located outside of the polygon. This point becomes the new origin. Figure 4.5 illustrates the procedure.

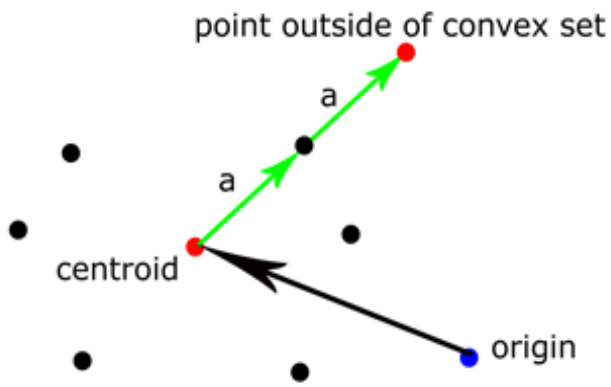


Figure 4.5. Calculating the outside origin of the convex set of points.

- Let $\mathbf{v_0}$ be the origin and $\mathbf{v_1}$ the closest point. Let the remaining points be $\mathbf{v_i}$, $i=2..n$. They are sorted by the angle between $\mathbf{v}_0$ $\mathbf{v}_1$ and $\mathbf{v}_0$ $\mathbf{v}_i$ calculated by means of the inner product between the vectors. Figure 4.6 shows a few steps of this algorithm.

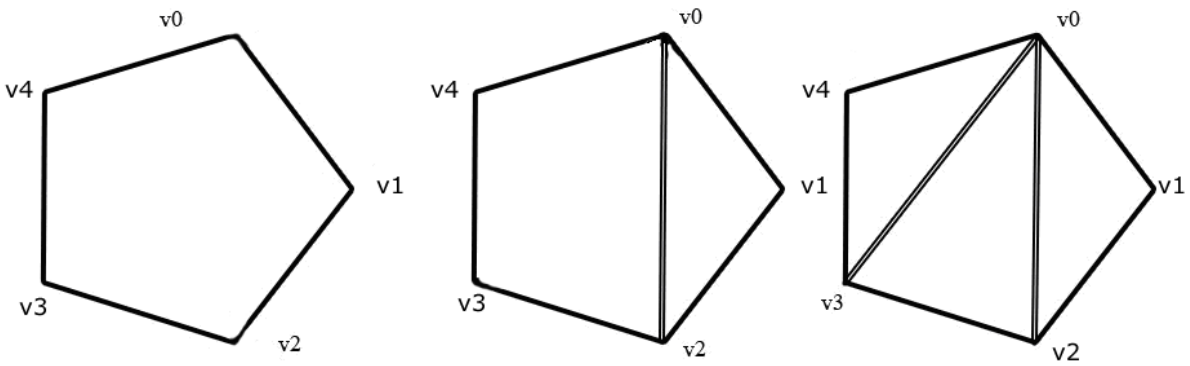


Figure 4.6. Convex Hull Dot Product Comparison

- Tesselate the proxy polygon into triangle fans and add the resulting vertices to the output vertex array in the order described for the triangle fan.

This convex hull algorithm is quite fast, compared to the other convex hull algorithms mentioned above. It is $O(n^2)$ in its re-ordering stage and the unit operation is a dot-product

which consists of three multiplications. It can be further optimized to include the following heuristic. The intersections are predictably repeated for each slice. They result in new coordinates each time, but because the slices are parallel planes at regular intervals, these coordinates are found at constant intervals. I apply this optimization for the convex hull which improves the speed of the algorithm. I will describe in this optimization in more detail in the implementation section.

4.5 MC Slicing

I have earlier observed the following property of our 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. The character of these intersections is also very limited. 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. Classifying the point's relation to the plane consists of a single dot product. Since a plane is commonly defined with a point (origin) on the plane and a normal to the plane, I calculate a vector from the origin of the plane to the point in question and compute the dot product between the resulting vector and the normal of the plane. 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 (the point with respect to the plane), I 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 [3] and is based on the classic Marching Cube algorithm [20].

The Marching Cubes algorithm (MC) 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. Figure 4.7 shows an example of MC Slicing. Table 1 shows an excerpt of the pre-calculated lookup table of the intersected edge sequence. Entry 175 lists all the edges intersected the slicing plane, when vertices 0, 1, 2, 3, 5, 7 are *behind* the plane. The number 175 is the hashing value created by the summation of bits, uniquely assigned to the vertices by means of a bitmask. Table 2 shows the bits, together with the bitmask, for which the sum is 175 .

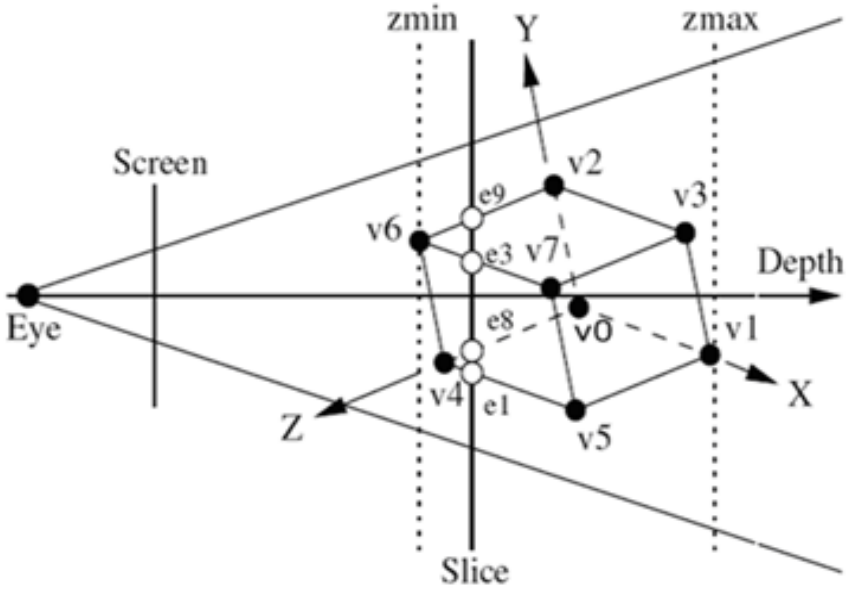


Figure 4.7. Example of MC slicing. Vertices 0, 1, 2, 3, 5 and 7 are deeper than the slicing plane.

$\mathbb{T}$	*intersected edge sequence*
0	-1, -1, -1, -1, -1, -1, -1, -1
1	4, 8, 0, -1, -1, -1, -1, -1
⋮	⋮
174	1, 3, 9, 4, 0, -1, -1, -1
175	3, 9, 8, 1, -1, -1, -1, -1
176	3, 11, 10, 8, 5, -1, -1, -1
⋮	⋮
254	4, 0, 8, -1, -1, -1, -1, -1
255	-1, -1, -1, -1, -1, -1, -1, -1

Table 1. Excerpt from MC Slicing Algorithm table. Each entry is an ordered sequence of edges hit by the surface.

Bit Mask	1	2	4	8	16	32	64	128
Vertex	**0**	**1**	**2**	**3**		**5**		**7**
Sum of bit values is equal to **175**								

Table 2. Calculation of the bit value when vertices 0, 1, 2, 3, 5 and 7 are behind the slicing plane.

The original Marching Cubes algorithm is employed for approximation of the boundaries of the volumetric object. The MC Slicing algorithm variant applies the pre-calculated table lookup idea to the volume slicing. Instead of approximating boundaries, I must figure out exact intersections but the benefit is immense from performing less intersection tests (6 instead of 12) and the implicit ordering of vertices, made ready for triangle fan generation. With the Convex Hull, it is necessary to perform 12 intersection tests (without an optimization suggested by the thesis advisor, Dr. Dana Vrajitoru, and described below). If I use MC Slicing, only at most 6 intersections are tested for. Because I have to classify the relation of the point to the slicing plane, one dot product operation is performed per point classification, with a total of 8 points. When the intersection test is carried out, two dot products are calculated per intersection test and replacing intersections tests with point classifications is highly desirable. This qualifies MC Slicing as a very efficient algorithm for our specific triangulation task.

The Convex Hull optimization also significantly improves the speed of the algorithm and is based on the following. The slicing planes appear at constant intervals along the view frustum near plane's normal as shown in Figure 4.8. The edges of the hexahedron are intersected by the slicing plane also at constant intervals. I take advantage of this fact in the Convex Hull optimization. I omit the calculation of intersection between the plane and the edge, if, after

adding the vector that describes the constant interval along the edge to the previous intersection, the resulting point is still between the edge boundaries (corner vertices). This replaces 2 dot products during the intersection with 3 vector additions. However, if the point has left the edge, we must still test for intersections.

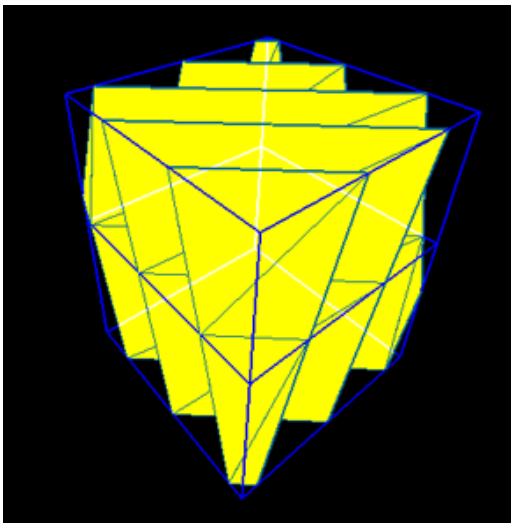


Figure 4.8. Optimization of Convex Hull Algorithm.

4.6 Pseudorandom noise

Computer Scientists have always looked to pseudorandom numbers (PRNs) to create randomness, a level of unpredictability, in behaviors of agents or appearances of objects. Generally, pseudorandom number generators (programs which are called to produce predictably random output) take the so called random seed (input value), use modulo operations on a few prime numbers and as a result generate a random number (output value). Unavoidably, the result values are never truly random, i.e. for every unique input, there is always one unique output. Conveniently, this fact is used to repeat any sequence of random numbers as needed. There are examples of computer games which very compactly describe whole worlds with seeded PRNs and clever generation of resources [42]. When memory is a scarce resource or extensively varied art assets are expensive to make, PRNs play a very crucial role.

3D Computer generated objects always appear perfect, primarily because they were created with a few simple inputs and rules, unlike objects found in nature, where there are an infinite number of factors (inputs) affecting the object! Knowing this, there is a necessity to make these perfect computer generated objects look like their natural counterparts. It is possible to apply an appearance of randomness with PRNs and the function responsible for it is called the noise function.

If one looks at objects in nature, one notices that many of them present a fractal nature, i.e. they are self-similar (larger parts consist of smaller parts similar to them). They have various levels of detail. A common example is the outline of a mountain range. It contains large variations in height (mountains), medium variations (hills), small variations (boulders), tiny variations (stones) and so on. We can look at almost anything: the distribution of patchy grass on a field, waves in the sea, the movements of an ant, the movement of branches of a tree, patterns in marble, turbulence of winds. All these phenomena exhibit the same pattern of large and small variations. Noise functions recreate this by simply adding up noisy values of different scales.

A noise function takes an integer as a parameter, and returns a random number based on that parameter. If the same parameter is used twice, the function produces the same number twice. Figure 4.9 illustrates a typical PRN-based noise function.

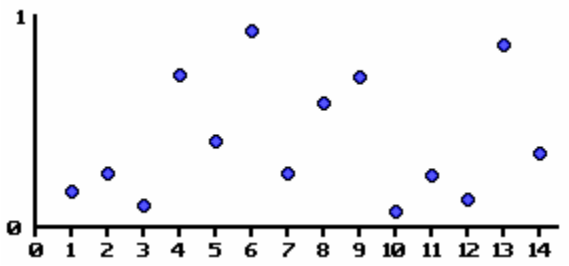


Figure 4.9. A typical noise function.
A random value between 0 and 1 is assigned to every point on the X axis.

Because these x-values are so discrete and almost nothing is discrete in nature, one must connect or *interpolate*, possibly smoothly, between these values to form a continuous set of values. Thus, one can define a continuous function that takes a non-integer as a parameter and allow the definition of an output value everywhere throughout the input variable domain. Figure 4.10 shows a function that interpolates between the points found in Figure 4.9.

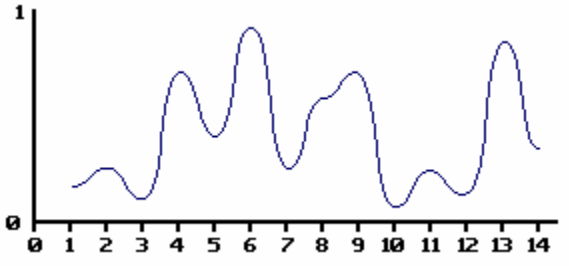


Figure 4.10. Interpolation of discrete inputs and outputs.

The interpolation can be done either linearly, or using a trigonometric function, or a polynomial.

The linear interpolation requires the least amount of calculation but also has the most inferior appearance, as can be seen in Figure 4.11.

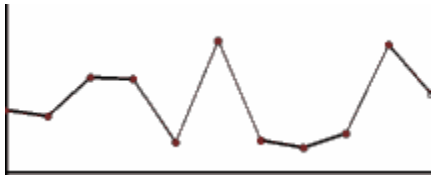


Figure 4.11. Linear Interpolation

It can be represented by the following pseudocode:

```
float lerp(a, b, x){
        return a*(1-x) + b*x;
}
```

Where **a** and **b** are vectors of any dimension, representing the given points between which the interpolation is computed and x is a float value between 0 and 1.

The linear interpolation does not work well, because it introduces discontinuities in the derivatives of any order and the resulting curve is only C^0.

Using a trigonometric function for interpolation involves calculation of a square root and is generally inferior to polynomial interpolation (Figure 4.12). The exception to the rule is hardware that implements trigonometric functions as single-operator procedure.

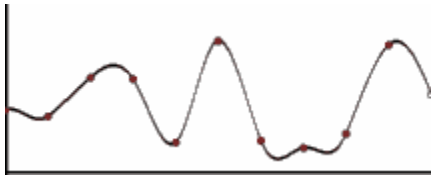


Figure 4.12. Polynomial interpolation

Different noises use different polynomial interpolators. The ones used by Perlin are expressed by a Cubic Hermite Equation,

$$f(t) = 3t^2 - 2t^3 \tag{13}$$

and by a Quintic Hermite Equation,

$$f(t) = 6t^5 - 15t^4 + 30t^3 \tag{14}$$

A polynomial interpolator is built on top of the linear interpolation:

```
float polynomial(a, b, x){
        return a*(1-f(x)) + b*f(x);
}
```

Where the f(x) is one of the equations above.

To properly describe different scale of natural objects, one must be able to provide a mechanism to control the size of the variation of noise in computer generated objects. To this end, I manipulate the amplitude and the frequency of the noise signal as shown in Figure 4.13.

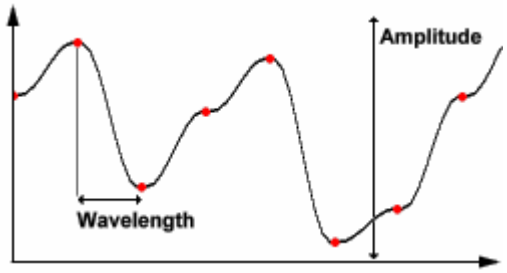


Figure 4.13. Signal Amplitude and Wavelength ($1/_{\text{frequency}}$ of signal).

In Figure 4.13 the red spots indicate the random values defined along the dimension of the function. In this case, the amplitude is the difference between the minima and maxima of the function. The wavelength is the distance from one red spot to the next. Again the frequency is defined to be $1/_{\text{wavelength}}$.

By combining many such functions, with various frequencies and amplitudes and adding them all together, one can create a new noisy function. Such process is demonstrated in Figures 4.14 and 4.15. Each successively added noise function is known as an *octave*. Thus, any specific noise function is created via addition of several basic noise functions.

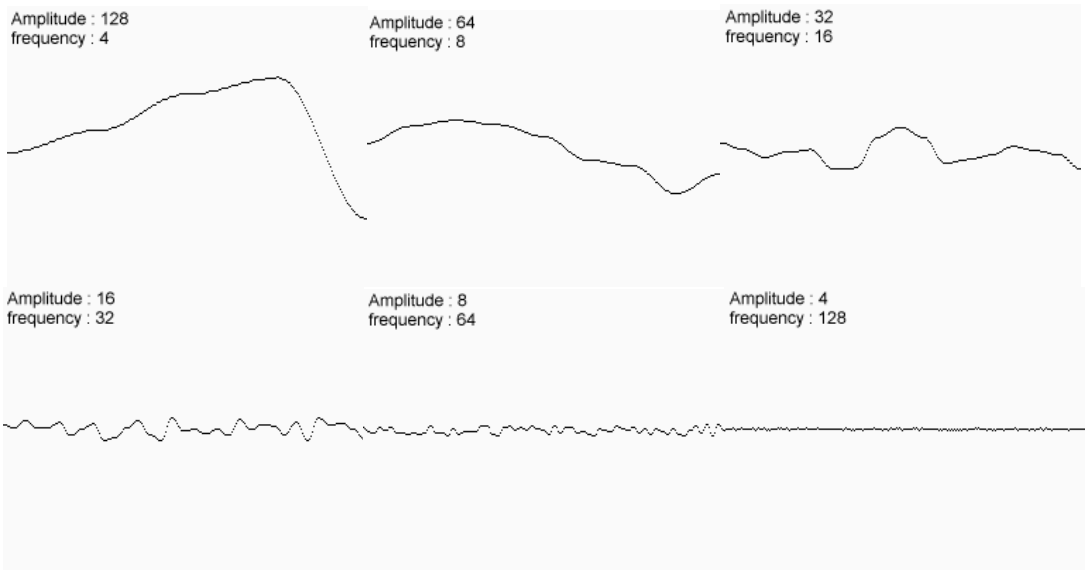


Figure 4.14. Individual Noise Octaves

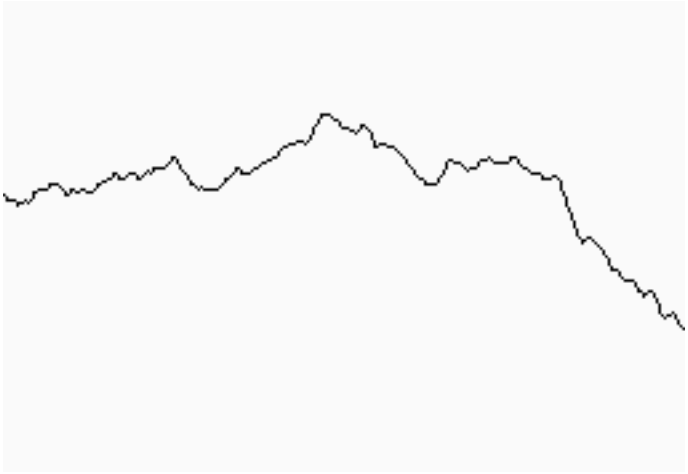


Figure 4.15. Resulting Sum of noise octaves

When these noise functions are added together, one must choose the amplitude and frequency for each function call. The one dimensional example in Figures 4.14 and 4.15 used twice the frequency and half the amplitude for each successive noise function added. This is done quite commonly. However, one can create noise functions with different characteristics by using other frequencies and amplitudes at each step. For example, to create smooth rolling hills, one could use a noise function with large amplitudes for the low frequencies, and very small amplitudes for the higher frequencies. Or alternatively one could create a flat, but very rocky plane by choosing low amplitudes for low frequencies.

A single number can be used to specify the amplitude of each frequency. This value is known as *Persistence*. Noise with a lot of high frequency has low persistence:

$$frequency = 2^i$$
$$amplitude = persistence^i$$

where **i** is the **i**th noise function being added. Table 3 below illustrates the effect of persistence on the output of the noise. It shows the component noise functions that are added, the effect of the persistence value, and the resultant noise function.

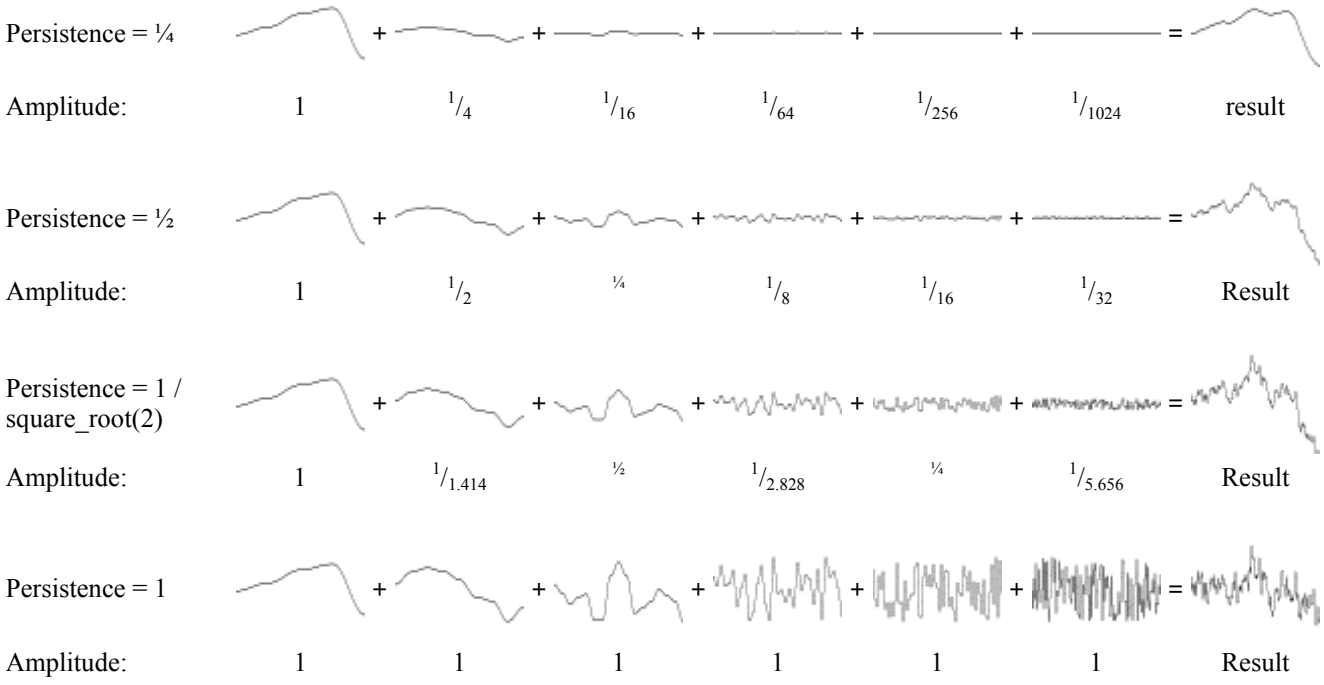


Table 3. Persistence of the noise.

Oftentimes, the persistence is broken out into *lacunarity* and *gain*. Lacunarity controls the frequency changes between octaves of noise. It is effectively the ratio between the sizes of successive octave scales. Lacunarity describes the texture of the noise fractal. It is related to the size distribution of the holes. Roughly speaking, if a fractal has large gaps or holes, it has high lacunarity. Gain controls the amplitude changes between octaves of noise. Gain gives the edges

of the noise fractal their definition. If a fractal has high detail in the self-affine structure of the edges, it has low gain.

Typically, the lacunarity is set to 2, and the gain is equal to 0.5. Any time when gain is equal to $1/_{\text{lacunarity}}$, it is commonly called "1/f" noise. It is helpful to use non-power-of-two lacunarity values which prevents *alignment* of noise octaves (subsequent octave repeats exactly twice on top of the larger octave), helps reduce artifacts and breaks the noise periodicity. Using complex ratio numbers (for example 1.93485736 or 2.18387276 instead of 2.0) for lacunarity helps as well. In general, breaking persistence out into lacunarity and gain aids in keeping the object noise appear without noticeable patterns.

4.7 Perlin Classic and Improved Noise

Perlin noise is a so-called gradient noise, which means that a pseudo-random gradient is set at regularly spaced points in space, and a smooth function between those points is interpolated. To generate Perlin noise in one dimension, a pseudo-random gradient (or slope) is associated with each integer coordinate, and the function value at each integer coordinate is set to zero.

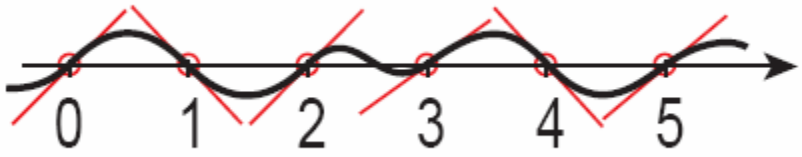


Figure 4.16. 1-D noise gradients interpolated using polynomials.

For a given point x situated somewhere between two integer points, the value is interpolated between two values, namely the values that would have been the result if the closest linear slopes from the left and from the right had been extrapolated to the point in question. This interpolation is not linear with distance, because that would not satisfy the constraint that the derivative of the noise function should be also continuous at the integer points. Instead, a

blending function is used that has the derivative equal to zero at its endpoints. The original Perlin noise uses equation (13), where improved Perlin noise uses equation (14).

In two dimensions, the integer coordinate points form a regular square grid. At each grid point a pseudo-random 2D gradient is picked as shown in Figure 4.17. For an arbitrary point *P* on the surface, the noise value is computed from the four closest grid points. Just like for the 1D case, the contribution from each of the four corners of the square grid cell is an extrapolation of a linear ramp with a constant gradient, with the value zero at its associated grid point.

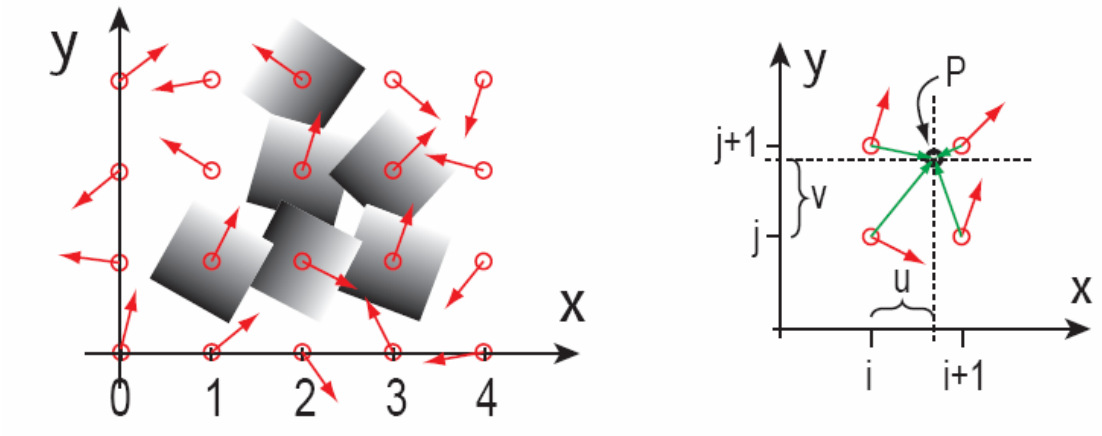


Figure 4.17. Left: Regular grid points with interpolated pseudo-random gradient values. Right: Interpolation of the noise value at a given grid point.

The value of each gradient ramp (see below, n_{00}, n_{10},n_{01},n_{11}) is computed by means of a scalar product (dot product) between the gradient vectors of each grid point and the vectors from the grid points (i,j), $(i,j+1)$, $(i+1,j)$, $(i+1,j+1)$, to the point P being evaluated. Equations (15) below illustrate this process.

Equation Group (15):

$$P=(x,y)\ i=floor(x)\ j=floor(y)$$
$$g_{00} \text{ is the gradient at } (i,j) \tag{15}$$
$$g_{10} \text{ is the gradient at } (i+1,j)$$
$$g_{01} \text{ is the gradient at } (i,j+1)$$
$$g_{11} \text{ is the gradient at } (i+1,j+1)$$

$u=x-i$, $v=y-j$ in the Figure 4.17, Right.

$$n_{00} = g_{00} \cdot \begin{bmatrix} u \\ v \end{bmatrix}, \quad n_{10} = g_{10} \cdot \begin{bmatrix} u-1 \\ v \end{bmatrix}, \quad n_{01} = g_{01} \cdot \begin{bmatrix} u \\ v-1 \end{bmatrix}, \quad n_{11} = g_{11} \cdot \begin{bmatrix} u-1 \\ v-1 \end{bmatrix} \quad (16)$$

The blending of the noise contribution from the four corners is performed as follows. Let the interpolation function be *f(t)=fade(t)* where *fade(t)* is either Equation (13) or (14) or any other function which ensures C^0, C^1 or C^2. The final interpolation value at (x,y) is n_{xy}
which is calculated as

$$n_{x0} = n_{00} f(u) + n_{10}(1 - f(u))$$

$$n_{x1} = n_{01} f(u) + n_{11}(1 - f(u))$$

$$n_{xy} = n_{x0} f(v) + n_{x1}(1 - f(v))$$

The above equations describe gradient interpolation in 2D where it is best understood. The algorithm is easily extended to describe other dimensions.

For the noise function to be repeatable, i.e. to always yield the same value for a given input point, gradients need to be pseudo-random. They need to have enough variation to conceal the fact that the function is not truly random, but too much variation will cause unpredictable behavior for the noise function. A good choice for 2D and higher is to pick gradients of unit length but different directions. For 2D, 8 or 16 gradients distributed around the unit circle is acceptable. For 3D, Ken Perlin's recommended set of gradients is the midpoints of each of the 12 edges of a cube centered on the origin as shown in Figure 4.18.

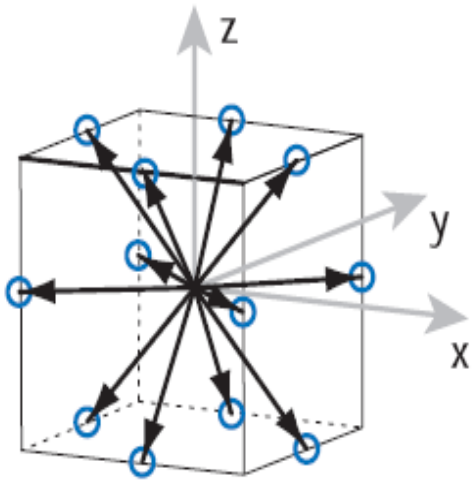


Figure 4.18. Pseudorandom vectors in 3D Perlin Noise are selected from midpoints of the edges of a cube centered at the origin.

Gradients with values of only -1, 0, 1 for each component are chosen because computing the dot product with such a vector does not require any multiplications, only additions and subtractions.

To associate each grid point with exactly one gradient, the integer coordinates of the point can be used to compute a hash value, which in turn can be used as the index into a look-up table of the gradients. Figure 4.19 shows a 3-Dimensional single-octave Perlin noise.

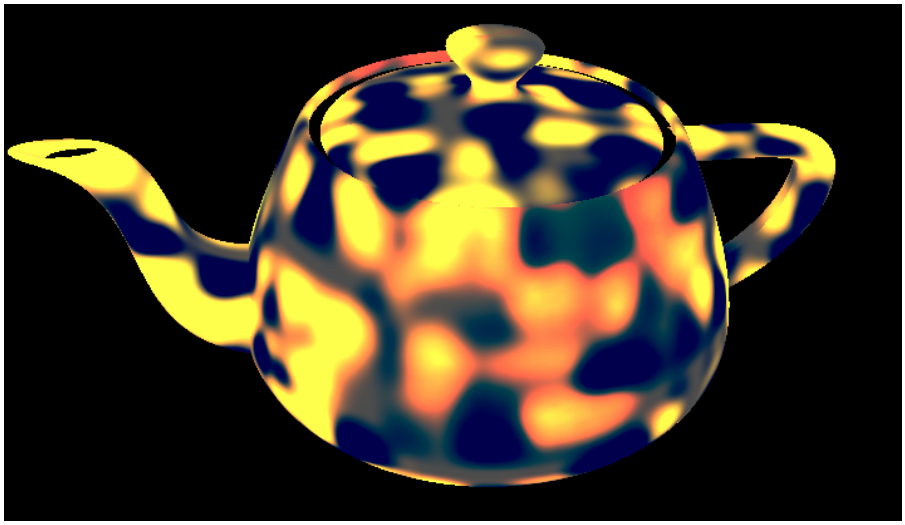


Figure 4.19. 3-Dimensional Perlin noise applied to a teapot.

4.8 Simplex Noise

4.8.1 *Simplex instead of Hypercube*

For simplex noise, simplex grids are used. A simplex is the simplest and most compact shape that can be repeated to fill the entire *N*-dimensional space. For a one-dimensional space, the simplest space-filling shape is intervals of equal length placed one after another, head to tail. In two dimensions, the common choice for a space-filling shape is a square, but that shape has more corners than what is necessary. The simplest shape that tiles a 2D plane is a triangle, and the formal simplex shape in 2D is an equilateral triangle. Two of these make a rhombus as shown in Figure 4.20.

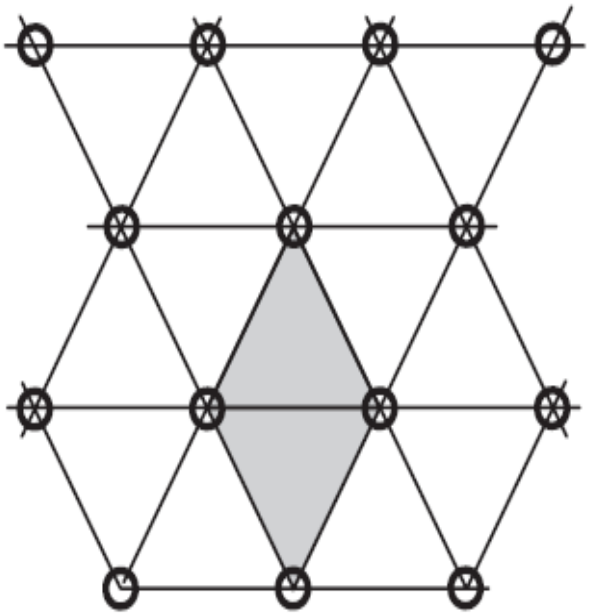


Figure 4.20. Simplex grid covering 2D space.

In three dimensions, the simplex shape is a slightly skewed tetrahedron, six of which make a cube that has been squashed along its main diagonal, shown in Figure 4.21.

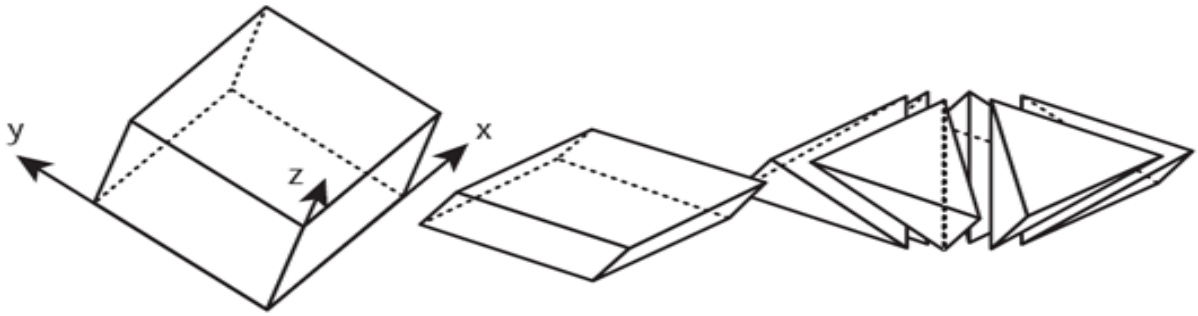


Figure 4.21. Tetrahedron, simplex shape for 3D space.

In four dimensions, the simplex shape is hard to visualize, but it has five corners, and 24 of these shapes make a 4D hypercube that has been squashed along its main diagonal. Generally speaking, the simplex shape for dimensions is a shape with *N+1* corners, and *N!* of these shapes can fill an *N*-dimensional hypercube that has been squashed along its main diagonal.

The definite advantage of a simplex shape is that it has as few corners as possible, much fewer corners than a hypercube. This makes it easier and faster to interpolate values inside the simplex based on the values at its corners. While a hypercube in *N* dimensions has 2^N corners, a simplex in *N* dimensions has only *N+1* corners. When the noise is calculated in higher dimensions, the complexity of evaluating a function at each corner of a hypercube and interpolating along each principal axis is an $O(2^N)$ problem that more quickly becomes intractable than a similar evaluation for each corner of a simplex shape $O(N)$.

4.22. *Summation instead of Interpolation*

A fundamental problem of classic noise is that it involves sequential interpolations along each dimension. Apart from the rapid increase in computational complexity as one moves to higher dimensions, it becomes more and more of a problem to compute the analytic derivative of the interpolated function. Simplex noise instead uses a straight summation of contributions from each corner, where the contribution is a multiplication of the extrapolation of the gradient ramp and a radially symmetric attenuation function. Figure 4.8.2 demonstrates the summation of contributions.

A point P inside a simplex gets contributions to its value only from the three (in case of 2D) kernels centered on the surrounding corners (shaded, red circles). Kernels at corners farther away (green circles) decay to zero before they cross the boundary to the simplex containing P. Thus, the noise value at each point can always be calculated as a sum of three (in case of 2D) terms.

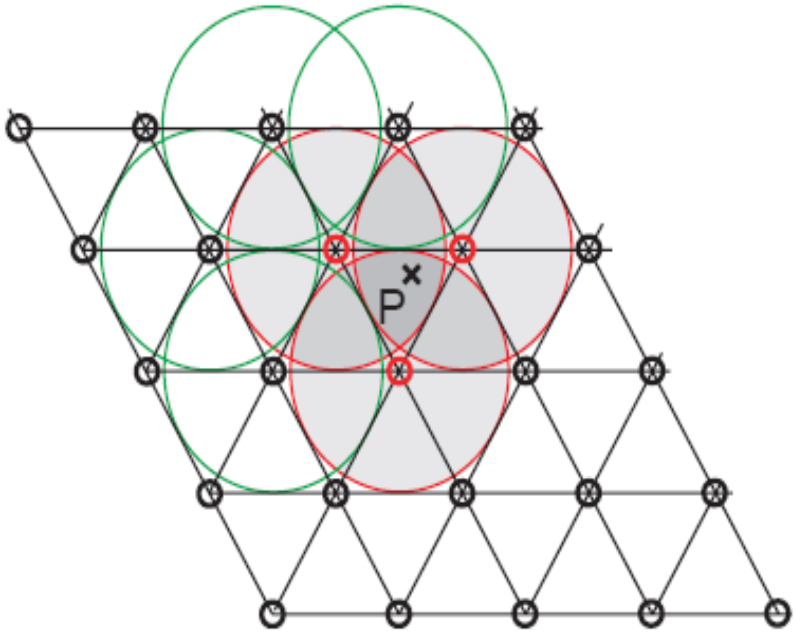


Figure 4.22. Noise Value Contributions

The radial attenuation is carefully chosen so that the influence from each corner reaches zero before crossing the boundary to the next simplex. This means that points inside a simplex will only be influenced by the contributions from the corners of that particular simplex.

4.8.3. *Selecting the Simplex*

For any point in the simplex space one must decide which simplex the point is in. This is performed in two steps. First, the input coordinate space is skewed along the main diagonal so that each squashed hypercube of *N!* simplices transforms into a regular, axis-aligned *N*-dimensional hypercube. Then it is easy to decide which hypercube the point is by looking at the integer part of the coordinates for each dimension (floor), similarly to classic Perlin noise. Then, the further decision on which particular simplex the point is in can be made by

comparing the magnitudes of the distances in each dimension from the hypercube origin to the point in question. The process is illustrated in Figure 4.23.

A 2D simplex grid of triangles can be skewed by a non-uniform scaling to a grid of right-angle isosceles triangles, two of which form a square with sides of length 1. By looking at the integer parts of the transformed coordinates (*x,y*) for the point one wants to evaluate, one can quickly determine which cell of two simplices contains the point. By also comparing the magnitudes of *x* and *y*, one can determine whether the point is in the upper or the lower simplex, and traverse the correct three corner points.

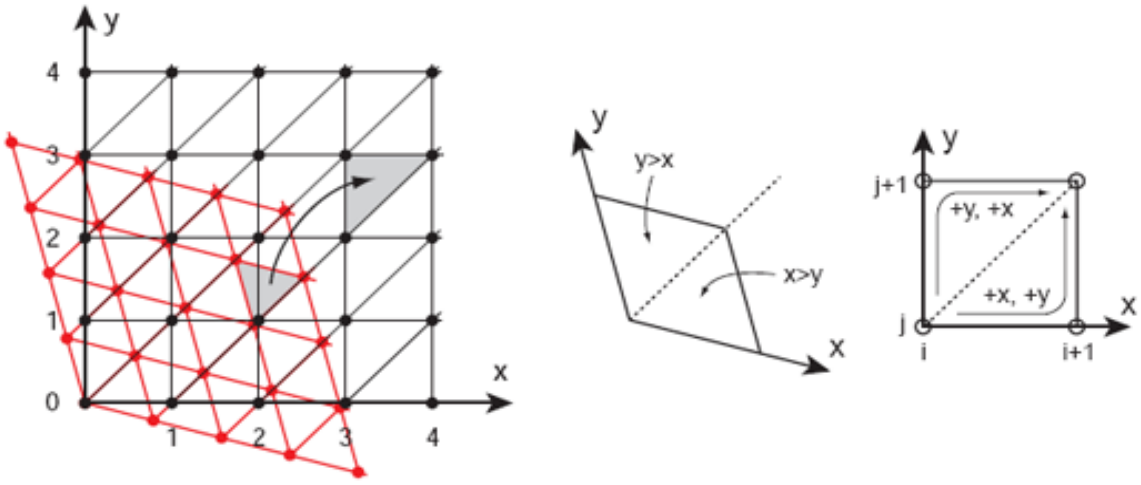


Figure 4.23. Skewing the simplex grid and determining the exact simplex of the point.

Just like a 2D simplex grid can be skewed to a regular square grid, a 3D simplex grid can be skewed to a regular cubical grid by a scaling along the main diagonal, and the integer parts of the coordinates for the transformed point can be used to determine which cell of 6 simplices the point is in. For this, one looks at the magnitude of the coordinates relative to the cell origin and compares them. Figure 4.24 is a view of a cube cell along its main diagonal where $x = y = z$. Points in the six simplices obey quite simple rules expressed as inequalities.

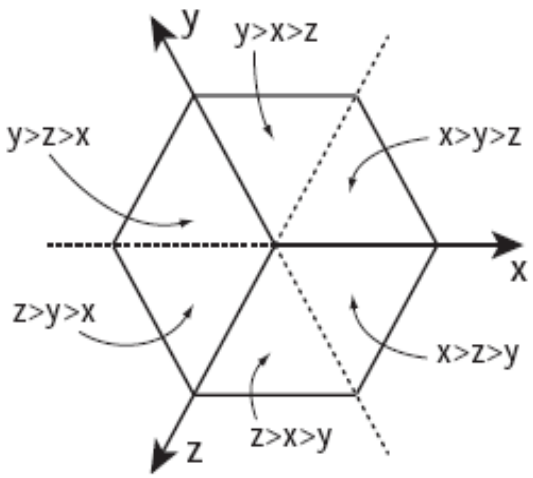


Figure 4.24. Using magnitudes of the coordinates relative to the cell origin to determine which tetrahedron the point is in.

In 2D, if $x>y$ the simplex corners are [0.0], [1,0] and [1,1], else the simplex corners are [0.0], [0,1] and [1,1]. The simplex traversal always takes one unit step in x and one unit step in y, but in different order for each of the simplices.

The traversal scheme for 2D generalizes straight off to 3D and further to an arbitrary number of dimensions: to traverse each corner of a simplex in N dimensions, one should move from the origin of the hypercube [0,0,…,0] to the opposite corner [1,1,…,1], and move unit steps along each principal axis in turn, from the coordinate with the highest magnitude to the coordinate with the lowest magnitude.

Figure 4.25 shows a 3-Dimensional single-octave simplex noise.

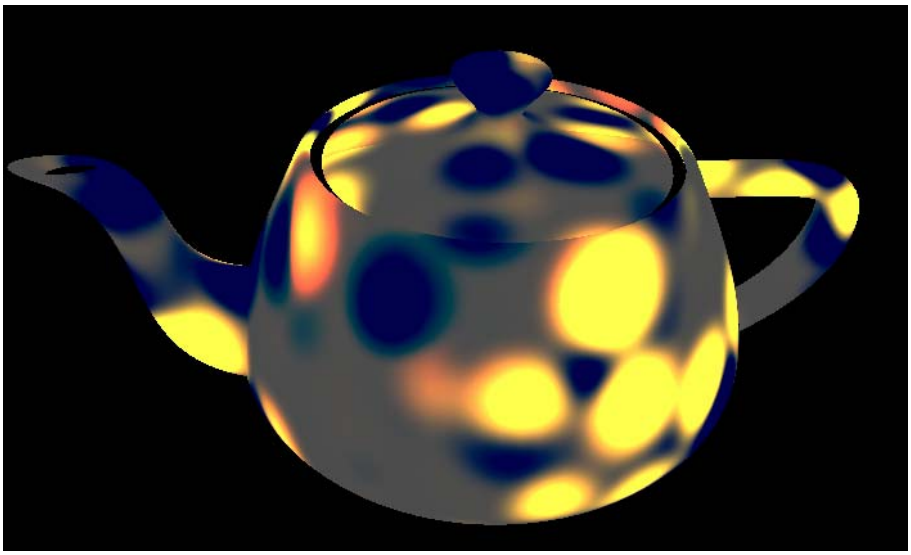


Figure 4.25. 3-Dimensional simplex noise applied to a teapot.

4.9 Modified Noise

Both Perlin's classic and improved noise were designed to run efficiently on a CPU. Modified Noise [23] includes two modifications to Perlin's improved noise that make it much more suitable for GPU implementation, allowing faster direct computation.

Modified noise does not take advantage of the modifications proposed in simplex noise and calculates its gradient based on 2^N lattice points in a hypercube around the point that must obtain noise function outputs. In Modified 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.

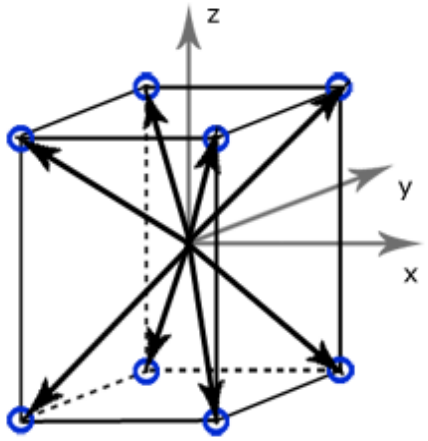


Figure 4.26. Modified noise gradients are selected from hypercube corners.

4.9.1 Selecting the Gradient

The core work of calculating gradients in Modified noise is done with the *grad* function which calculates the gradient value at each integer input point of the lattice. Similar to improved noise, gradients are limited to simple values of only -1, 1 for each component and a dot product with such a vector only requires additions and subtractions.

$$\boldsymbol{p}_{i;\,jkl} = [\ \lfloor \boldsymbol{p}_x \rfloor + j,\ \lfloor \boldsymbol{p}_y \rfloor + k,\ \lfloor \boldsymbol{p}_z \rfloor + l\ \ ];\ j, k, l \in \{0, 1\}$$

Similar to u,v in equation group (15) on page :

$$\boldsymbol{p}_f = \boldsymbol{p} - \boldsymbol{p}_i$$

The *grad* function in general is:

$$grad(\boldsymbol{p}_i, \boldsymbol{p}_f) = gradient(\boldsymbol{p}_i) \cdot \boldsymbol{p}_f$$

Because of the choice of gradient positions, the *grad* function in 3D space is simply this:

$$grad3D(\boldsymbol{p}_f) = \boldsymbol{p}_f^x + \boldsymbol{p}_f^y + \boldsymbol{p}_f^z$$

For lower dimensions the *grad* function simplifies nicely:

$$grad2D(\boldsymbol{p}_f) = \boldsymbol{p}_f^x + \boldsymbol{p}_f^y + 0$$

$$grad1D(\boldsymbol{p}_f) = \boldsymbol{p}_f^x + 0 + 0$$

Therefore the *grad* function produces results in lower dimensions identical to *slices* of output in higher dimensions. This property of modified noise is called the *dimension reducibility* property. The number of dependent texture lookups on a GPU, as a potential bottleneck, can be reduced using this property of Modified noise.

There is a potential for axis-aligned clumping when picking gradients this way, which was noted in the improved noise by Perlin, but unavoidably, in improved noise, 1D and 2D gradients are still picked from corner points of the hypercube. When one observes the visual results of the 2D noise, Modified noise does manifest axis-aligned clumping. Figure 4.27 shows such clumping in an image of 3-Dimensional single-octave modified noise. However, in 3-D and 4-Dimensional turbulence based on Modified noise these defects are not noticeable.

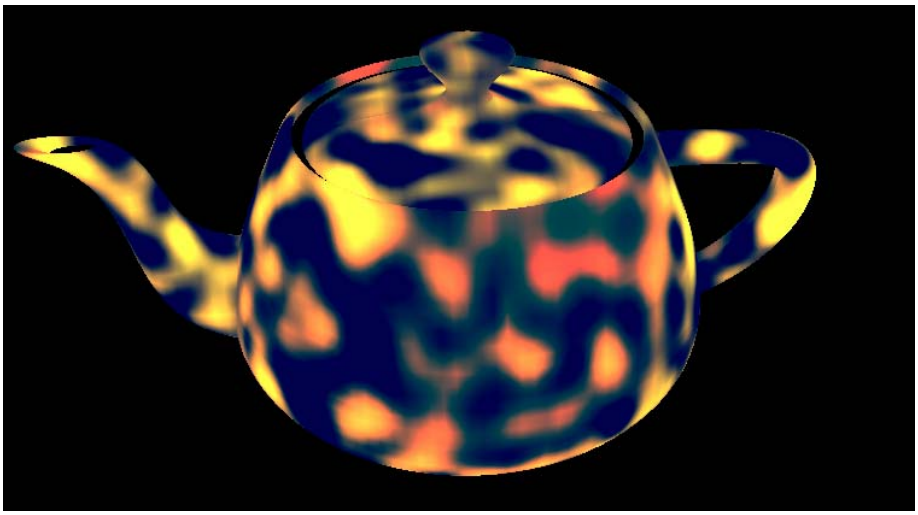


Figure 4.27. 3-Dimensional Modified noise applied to a teapot.

4.9.2 Hashing

Perlin's formulation for classic and improved noise uses several chained table lookups, an operation that is relatively fast on a CPU, but can be a bottleneck on a GPU due to higher texture (memory) fetch latency.

The gradient table/texture used to calculate a pseudorandom gradient at any given point in Perlin noise is precomputed to map each integer in the interval [0,255] to a new unique new integer from the interval [0, 255]. An integer representing the address of another integer in the

permutation table is passed to the *permute* function (17) and the integer stored at that memory location is returned. The final single hash value is

$$hash3(\mathbf{p}_i) = permute\left(permute\left(permute\left(\mathbf{p}_i^x\right) + \mathbf{p}_i^y\right) + \mathbf{p}_i^z\right) \tag{17}$$

The lower-dimensional hash functions are just a slice from the higher dimensional hash, offset by permute^{-1}(0).

$$hash2(\mathbf{p}_i) = permute\left(permute^{-1}(0), \mathbf{p}_i^x, \mathbf{p}_i^y\right)$$

$$hash1(\mathbf{p}_i) = permute\left(permute^{-1}(0), \mathbf{p}_i^x\right) = permute\left(\mathbf{p}_i^x\right)$$

Modified Noise replaces precomputed hash with purely computable or, if necessary, its own version of pre-computed hash based on the Blum-Blum Shub (BBS) pseudorandom generator [4].

The BBS generator computes

$$x_{i+1} = x_i^2 \bmod M \tag{18}$$

Where $M = pq$, for large primes p and q. The low order bits from the random output are used to generate gradients. There are two different implementations of modulo operator: a five instruction version,

$$i \bmod M = i - floor(i / M) * M$$

and a four instruction version,

$$i \bmod M = frac(i / M) * M$$

The latter suffers from precision problems. The results are visually acceptable, but the noise does not repeat at M as it should, which becomes evident in 2D and lower noise. The *exact* five instruction version is used to compute 2D noise where the *inexact* four instruction version can be applied in 3D and higher, where the noise repeat factor is less of an issue.

A multidimensional hash function is constructed in a way similar to the permutation table lookups in Perlin noise:

$$hash3(\mathbf{p}_i) = hash\left(hash\left(hash\left(\mathbf{p}_i^x\right) + \mathbf{p}_i^y\right) + \mathbf{p}_i^z\right)$$

$$hash(x) = x^2 \bmod M$$

Modified noise is a *purely computable* noise but it can be and should be accelerated with texture lookups if they do not present a performance bottleneck. The low level OpenGL shader code for 2D Modified noise is 45 instructions long which is too many for a function used several times for the same pixel, and one can save time by pre-baking the gradient texture using the hash procedure described.

The 3D noise is still too large to store as a texture. Modified noise uses its dimension reducibility property and optimizes calculation of 3D noise by storing the 2D noise and the z-gradient in separate textures or a single 2-channel texture. 2D noise and *z*-gradient are accessed in the texture(s) using:

$$\boldsymbol{c}_0 = \left(\boldsymbol{p}^x, \boldsymbol{p}^y + hash\left(\boldsymbol{p}_i^z\right)\right)$$

$$\boldsymbol{c}_1 = \left(\boldsymbol{p}^x, \boldsymbol{p}^y + hash\left(\boldsymbol{p}_i^z + 1\right)\right)$$

where $\boldsymbol{c}_0$ and $\boldsymbol{c}_1$ are the two x-y terms for the integer z value below and above the noise argument.

The final 3D noise using separate 2D noise and *z*-gradient textures is

$$flerp\left(\boldsymbol{p}_f^z, mNoise2(\boldsymbol{c}_0) + \boldsymbol{p}_f^z \cdot zgrad(\boldsymbol{c}_0), mNoise2(\boldsymbol{c}_1) + \boldsymbol{p}_f^z \cdot zgrad(\boldsymbol{c}_1)\right) \tag{19}$$

Using a single 2-channel texture, the computation is

$$flerp\left(\boldsymbol{p}_f^z, \left(1, \boldsymbol{p}_f^z\right) \cdot tex(\boldsymbol{c}_0), \left(1, \boldsymbol{p}_f^z\right) \cdot tex(\boldsymbol{c}_1)\right) \tag{20}$$

where *tex* is a texture lookup, returning two floats (2-channel texture) in the case of optimized 3D noise, and *flerp* is a polynomial interpolation function which uses either Cubic Hermite (Eq. 13) or Quintic Hermite (Eq. 14) polynomials from page 40:

$$flerp(t,a,b) = (1 - polynomial(t)) \cdot a + polynomial(t) \cdot b$$

4D noise is similar to precomputed 2D noise, using a 3-channel 2D texture containing noise, *z*-gradient, *w*-gradient.

$$flerp\left(\boldsymbol{p}_f^w, flerp\left(\boldsymbol{p}_f^z,\ (1, z, w)\cdot\ tex(\boldsymbol{c}_{00}),\ (1, z, w)\cdot\ tex(\boldsymbol{c}_{01})\right)\right. \\ \left., flerp\left(\boldsymbol{p}_f^z,\ (1, z, w)\cdot\ tex(\boldsymbol{c}_{10}),\ (1, z, w)\cdot\ tex(\boldsymbol{c}_{11})\right)\right) \tag{21}$$

where

$$\boldsymbol{c}_{jk} = (\boldsymbol{p}^x, \boldsymbol{p}^y + hash(\boldsymbol{p}^z + k + hash(\boldsymbol{p}^w + j)))$$
$$j, k \in \{0, 1\} \tag{22}$$

4.10 Turbulence

As described in 4.6, 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* [13],

$$turb\ = \sum_{i=0}^{n} gain^{\ i} \cdot abs(noise(position \cdot lacunarity^i)) \tag{23}$$

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 Modified 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.

Figures 4.28-4.30 show 3-Dimensional turbulence of each of the three noises discussed in the previous sections.

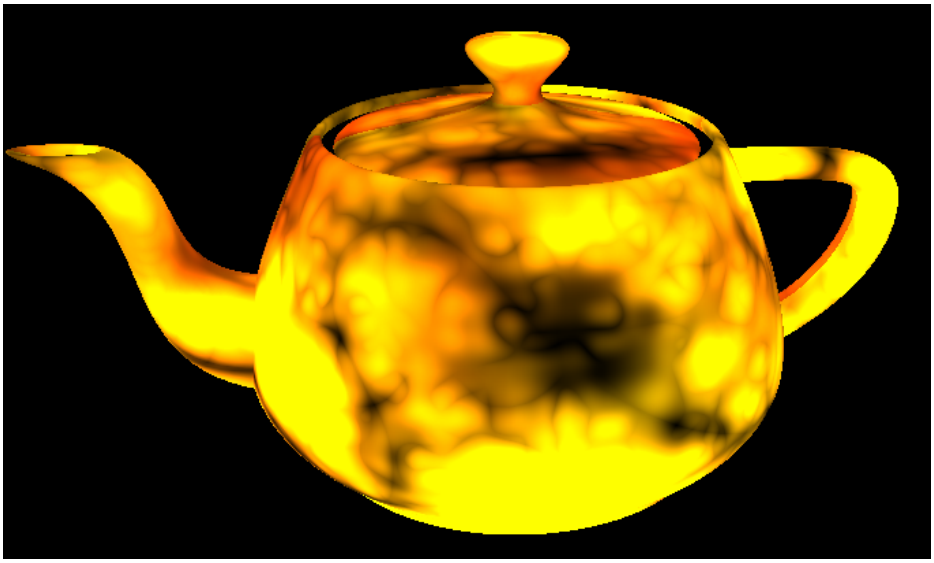


Figure 4.28 3-Dimensional turbulence of Perlin noise applied to a teapot.

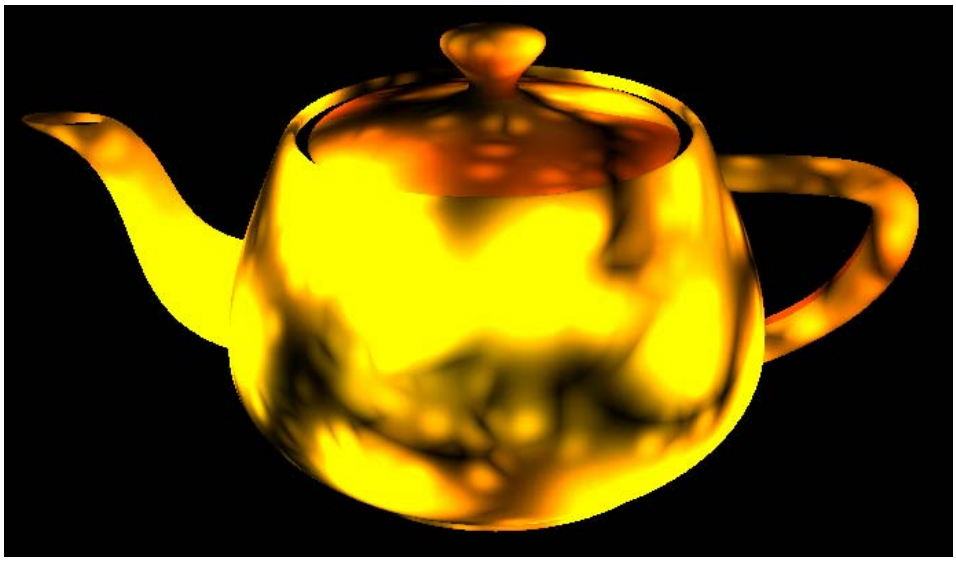


Figure 4.29. 3-Dimensional turbulence of Simplex noise applied to a teapot.

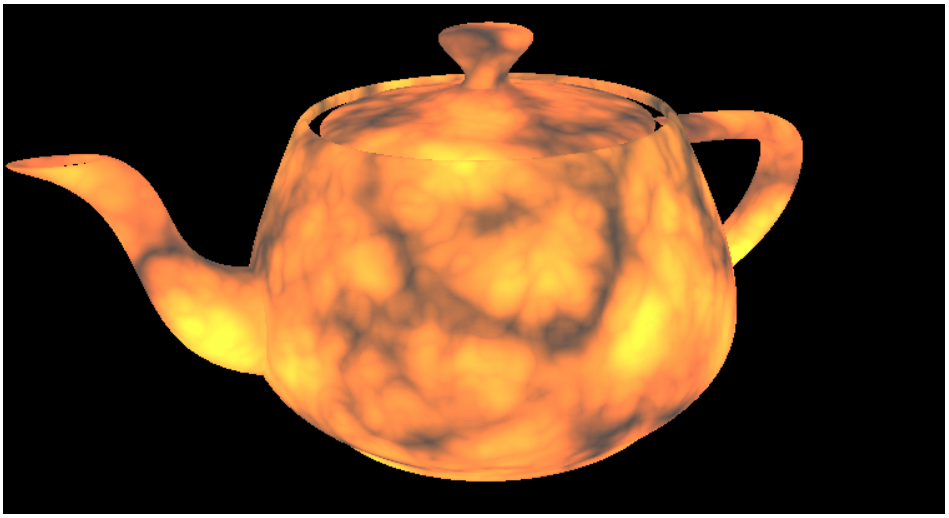


Figure 4.30. 3-Dimensional turbulence of Modified noise applied to a teapot.

Each type of noise has a distinct visual appearance. Let's examine Figures 4.19, 4.25 and 4.27 and, since turbulence is calculated in exactly the same manner for all three types of noise, consider just noise types.

Simplex noise stands out in its appearance. Round shapes dominate the surface (because the surface is 'carved' out of 3D volumetric noise, think of these shapes as spheres rather than circles). This is the visual result of moving from the time-consuming interpolation one dimension at a time to a fast, direct summation. Simplex noise also has a different statistical character: it has a higher peak range than the other noise types. It is however possible to generate a Simplex noise-based turbulence resembling a Perlin noise-based turbulence, via careful manipulation of lacunarity and gain and noise scale.

Modified noise and Perlin improved noise in 2D and 3D look almost identical, and it is not surprising, considering that the method of construction of these types of noise is very similar. *Modification* of Modified 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 method of computing random hashes completely on the fly 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. Modified noise picks its gradient from the corners of a unit-n cube, because it is required by the dimension reducibility property. The visual difference becomes evident when 4-D Modified noise is compared to others.

Figure 4.31. 4-D Modified noise in a scene. Fire Separation.

Modified noise in Figure 4.31 has an almost ‘fluid’ character. It is the only noise among the three types, whose large, solid parts separate and float away from the main body of fire very similarly to how natural fire separation occurs.

5. 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 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.

In the 3D engine these modifiable parameters include the underlying rendering engine(s) used to render 3D primitives, i.e. OpenGL/GLSL [27] or DirectX/HLSL [5] 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 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.

All of these factors influence the system's performance. I now explore the feasibility of the model and describe how to efficiently leverage combinations of these parameters. Testing was done on a Pentium 4 3.0 GHz PC, equipped with ATI Radeon X1650 video card with 256 Megabytes of memory. Total rendering space (Viewport) is 800 by 600 pixels.

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 fps indicates a better graphics performance.

Please refer to Addendum 1 on pp. 74-75 to see all the tables of the feasibility study.

I start by figuring out the impact of a standard configuration of static fire on the production system. Such a configuration (Table A1) consists of a single volume of fire with 2 subvolumes

(flame knots), 20 slices (view-aligned surfaces that cut through the volume), 1 *flamespine* attached to the *firescenenode*. Only 4% of screen space is occupied by fire. During any given test, 1 type of gradient noise is used with varying FPS (frames per second) results. The method of triangulation is Marching Cube Slicing. The physics simulation is turned off. The total count of scene triangles in the viewport is 3174. The rendering API is OpenGL with GLSL Shader Engine. The number of triangles that belong to the fire model alone is 70.

Gradient Noise	Triangulation	Scene Triangles	Renderer	Fire Triangles	Frames Per Second
Perlin's Improved Noise	MC	3174	OpenGL/GLSL	70	30
Simplex Noise	MC	3174	OpenGL/GLSL	70	53
Modified Noise	MC	3174	OpenGL/GLSL	70	119

Table A1. Without Octree Environment, Density:2, Slices:20, Flame Volumes:1, Screen Space: 4%, Noise Octaves: 4, No Physics.

Review Tables A1 (rendered without octree environment) and A2 (rendered with octree environment) and compare the performance between matching gradient noise models. The octree represents the scene set up for the fire.

Gradient Noise	Triangulation	Scene Triangles	Renderer	Fire Triangles	Frames Per Second
Perlin's Improved Noise	MC	3174	OpenGL/GLSL	70	22
Simplex Noise	MC	3174	OpenGL/GLSL	70	45
Modified Noise	MC	3174	OpenGL/GLSL	70	101

Table A2. With Octree Environment, Density:2, Slices:20, Flame Volumes:1, Screen Space: 4%, Noise Octaves: 4, No Physics.

The difference is a constant 8 frames per second. This means that the cost of rendering the octree level map in Irrlicht is only 8 frames per second. For comparison, Table A7 shows the performance of the graphics engine, when only the octree environment is rendered.

The OpenGL version of the environment runs at 252 frames per second. The fire model running on Modified noise only runs at 101 frames per second. The fire procedure costs this particular system 151 frames per second in the most optimal configuration. The performance of Modified noise is impressive (Table A1), relative to Perlin noise, as my initial goal was to optimize the system to run at above 60 frames per second.

Table A3 compares three different triangulation techniques.

Gradient Noise	Triangulation	Scene Triangles	Renderer	Fire Triangles	Frames Per Second
Modified Noise	**MC**	3885	OpenGL/GLSL	300	68
Modified Noise	**Convex Optimized**	3836	OpenGL/GLSL	303	53
Modified Noise	**Convex**	3740	OpenGL/GLSL	301	56

Table A3. Triangulation Test, Density:8, Slices:20, Flame Volumes:1, Screen Space: 7%, Noise Octaves: 4, No Physics.

Notice that I increased the density of the lattice to 8 subvolumes as the difference in performance on a sparse lattice is negligent. The Convex and Convex Optimized Methods perform about the same, while Marching Cube Slicing performs significantly better.

Table A4 is meant to show that the complexity of rendering fire is independent of the lattice density. The lattice density defines the number of knots of the flame spine and thus, the number of hexahedrons composing the fire.

Lattice Density	Gradient Noise	Scene Triangles	Renderer	Fire Triangles	Frames Per Second
3	Modified Noise	3816	OpenGL/GLSL	100	87
8	Modified Noise	3816	OpenGL/GLSL	258	69
13	Modified Noise	3816	OpenGL/GLSL	407	59
19	Modified Noise	3816	OpenGL/GLSL	521	54
25	Modified Noise	3816	OpenGL/GLSL	644	50

Table A4. Density Test, Slices:20, Flame Volumes:1, Screen Space: 7.5%, Noise Octaves: 4, Triangulation: MC, No Physics.

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 [13].

Table A5 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.

Slices	Gradient Noise	Scene Triangles	Renderer	Fire Triangles	Frames Per Second
10	Modified Noise	3174	OpenGL/GLSL	34	168
20	Modified Noise	3174	OpenGL/GLSL	70	101
30	Modified Noise	3174	OpenGL/GLSL	106	73
40	Modified Noise	3174	OpenGL/GLSL	145	56
50	Modified Noise	3174	OpenGL/GLSL	145	46

Table A5. Slice Test, Density:2, Flame Volumes:1, Screen Space: 4%, Noise Octaves: 4, Triangulation: MC, No Physics.

Table A6 is meant to validate the theory that the complexity increases linearly with the increase of screen space occupied by the fire model. As tests reveal, the complexity is logarithmic.

Screen Space Occupied	Gradient Noise	Scene Triangles	Renderer	Fire Triangles	Frames Per Second
4.0%	Modified Noise	3174	OpenGL/GLSL	70	101
9.0%	Modified Noise	3174	OpenGL/GLSL	70	51
30.0%	Modified Noise	3174	OpenGL/GLSL	70	28
40.0%	Modified Noise	3174	OpenGL/GLSL	70	15
50.0%	Modified Noise	3174	OpenGL/GLSL	70	12

Table A6. Screen Space Test, Density:2, Slices:20, Flame Volumes:1, Noise Octaves: 4, Triangulation: MC, No Physics.

Table A7 gives us the base performance of the Irrlicht engine when rendering a scene using different rendering APIs.

Gradient Noise	Triangulation	Scene Triangles	Renderer	Fire Triangles	Frames Per Second
-	-	3174	OpenGL/GLSL	-	252
-	-	3174	DirectX 9/HLSL	-	232

Table A7. Renderer - Environment Only.

It is of note that Irrlicht version 1.3.1 renders the same scene 20 frames per second faster using OpenGL than DirectX. This version of the engine apparently runs better on OpenGL. It is common that multi-platform engines vary in performance. This happens because of code

branching that occurs in low-level APIs and because of code optimizations that do not take place uniformly across all libraries.

I have to take this fact into consideration when comparing noise performance in different renderers. Table A8 compares the performance of Simplex noise in OpenGL/GLSL vs. DirectX/HLSL.

Gradient Noise	Scene Triangles	Renderer	Fire Triangles	Frames Per Second
Simplex Noise	3174	**OpenGL/GLSL**	70	51
Simplex Noise	3174	**DirectX 9/HLSL**	70	72

Table A8. Simplex Noise - Renderer, Density:2, Slices:20, Flame Volumes:1, Screen Space: 4%, Noise Octaves: 4, Triangulation:MC, No Physics.

Table A9 compares the performance of Perlin noise in OpenGL/GLSL vs. DirectX/HLSL.

Gradient Noise	Scene Triangles	Renderer	Fire Triangles	Frames Per Second
Perlin Noise	3174	**OpenGL/GLSL**	70	53
Perlin Noise	3174	**DirectX 9/HLSL**	70	29

Table A9. Perlin Noise - Renderer, Density:2, Slices:20, Flame Volumes:1, Screen Space: 4%, Noise Octaves: 4, Triangulation:MC, No Physics.

Table A10 compares performance of Modified noise in OpenGL/GLSL vs. DirectX/HLSL.

Gradient Noise	Scene Triangles	Renderer	Fire Triangles	Frames Per Second
Modified Noise	3174	**OpenGL/GLSL**	70	101
Modified Noise	3174	**DirectX 9/HLSL**	70	79

Table A10. Modified Noise - Renderer, Density:2, Slices:20, Flame Volumes:1, Screen Space: 4%, Noise Octaves: 4, Triangulation:MC, No Physics.

It is revealed in tables A8-A10 that there is no additional cost of running the fire model in OpenGL/GLSL vs. DirectX/HLSL which confirmed my prior knowledge about what happens to a high-level shader program when it is compiled into low-level shader assembly. It is optimized equally well by both OpenGL and DirectX drivers on the GPU.

Table A11 compares performance of the fire system with varying number of fire volumes.

Flame Volumes	Screen Space Occupied	Triangulation	Scene Triangles	Renderer	Fire Triangles	Frames Per Second
1	4.0%	MC	10	OpenGL/GLSL	70	119
2	8.0%	MC	10	OpenGL/GLSL	140	62
3	12.0%	MC	10	OpenGL/GLSL	210	42

Table A11. Varying Fire Volumes, Density:2, Slices:20, Gradient Noise:Modified Noise, Screen Space: 4%, Noise Octaves: 4, Triangulation:MC, No Physics.

Additional fire volumes add to the complexity of the rendering algorithm logarithmically as the GPU parallelizes the shader program calls.

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 4-D noise is expensive is confirmed when we compare the expense of rendering a 4% - screen space volumetric fire (150 frames per second drop) to the expense of rendering the octree environment (a medieval castle) (only 8 frames per second 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 thesis 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 increase in use of procedural noise (among fluid solvers and other physics-based techniques) for volumetric effects.

6. Implementation

The program written to test volumetric fire rendering is called *Prometheus*, after one of the titans of Greek Mythology. Legend has it that crafty Prometheus stole fire from Zeus and gave it to men for their use, suffering a terrible fate afterwards. Without this sacrifice of Prometheus, mankind would not have civilization, and without this code this thesis would merely be, well, noise.

Prometheus is written in C++ and compiled using Microsoft Visual Studio.NET 2005. Prometheus builds on top of the Irrlicht Engine [35] and borrows from it the facilities of 3D mathematical computation, memory management, basic graphics rendering and shader management.

The Irrlicht Engine is an open source high performance real-time 3D engine written and usable in C++ and also available for .NET languages. It is completely cross-platform, using Direct3D, OpenGL and its own software renderer, and has all of the state of the art features which can be found in commercial 3D engines. The subset of features used in this thesis includes high performance real-time 3D rendering using Direct3D and OpenGL, built-in and extensible material library with vertex and pixel shader support, powerful, customizable and easy to use 2D GUI System with buttons, lists, edit boxes, 2D drawing functions like alpha blending, color key based blitting, font drawing and mixing 3D with 2D graphics, direct import of common mesh file format Quake 3 levels (.bsp), optimized fast 3D math and container template libraries. Most importantly, the engine is open source and totally free which allowed me to use it in an academic setting.

Custom code creates the geometry of the scene and makes API calls to Irrlicht, which, once the scene is merged from map files and custom geometry, represented by TriangleFans, passes rendering control to either OpenGL or DirectX. The GPU renders the geometry of the scene with appropriate colors by executing custom shader programs via the ShaderCallback facility (Figure 6.1).

The vertex and the pixel shader programs are initially written in GLSL and later converted into HLSL. All of the shader programs were prototyped, debugged and tested in the shader environment ATI RenderMonkey [34], because shaders are notoriously hard to debug inside the main program, but significantly easier to troubleshoot as an isolated piece of code. Vertex shaders are compiled under shader model 1.1 as they are fairly basic programs and do not require many instruction slots or complex instruction sets. The pixel shaders in this implementation, on the contrary, are very demanding programs, requiring many instruction slots, looping constructs and several dependable texture lookups. Therefore shader model 2_b was used to compile pixel shaders used in this thesis. Prometheus includes Stefan Gustavson's GLSL implementation of Simplex and Perlin noise, and Marc Olano's GLSL implementation of Modified noise.

The code has the organization described in Figure 6.1 below. All objects are rendered in the Irrlicht pipeline by inheriting from class *irr::scene::ISceneNode* and implementing the virtual functions declared in it.

An Irrlicht scene node is a node in the hierarchical scene graph. Every scene node may have *n*-children, which are other scene nodes. There is no limitation on the height or the size of the scene graph tree structure, but new nodes or levels are added with discretion, only as needed. Children move relative to their parent's position. In this way, it is possible to attach a light to a moving car (height of 2), or to place a walking character on a moving platform on a moving ship (height of 3). Commonly, the transformation of scene nodes is done via concatenation of the transformation matrices and Irrlicht scene nodes are no exception to the rule. It is also easy to occlude parts of a scene not visible to the camera via the hierarchy of the scene graph.

The two most prominent methods that one must implement when integrating code into Irrlicht are *OnAnimate()* and *OnRegisterSceneNode()*. *OnAnimate()* is called just before rendering the whole scene. Nodes may calculate or store animations here, and may do other useful things, dependent on what they are. Also, *OnAnimate()* should be called for all child scene nodes. This method will be called once per frame, notwithstanding whether the scene node is visible or not.

OnRegisterSceneNode() is also called just before the rendering process of the whole scene. Nodes may register themselves in the render pipeline during this call, precalculate the geometry which should be rendered, and prevent their children from being able to register them if they are clipped by simply not calling their *OnRegisterSceneNode()* Method.

The *FireSceneNode* represents all fires in a particular location in the scene. It is an invisible node that controls the placement of the origin of fire object space. *FireSceneNode* is capable of attaching to any other node in the scene which allows me to express moving sources of fire such as torches, etc. *FlameSpine* and *FlameSpineStatic* are the rendering classes for fire objects. *FlameSpine* class uses physics simulation to advance its knots in space and can be attached to a parent *SceneNode*. *FlameSpineStatic* has no physics logic and simply renders a static volume with fire noise. Usually one object of either class is enough to represent a single fire. Even though a single *FlameSpine* renders as a single entity, it consists of multiple logical *FlameKnots*, which provide the mapping between local knot space and world space. It is possible to render multiple *FlameSpines* in the same scene and there is benefit to rendering multiple overlapping volumes of fire in the same scene but it is prohibitively expensive, as *n* fires will be rendered in *n*-time.

Figure 6.1 shows the Prometheus program structure.

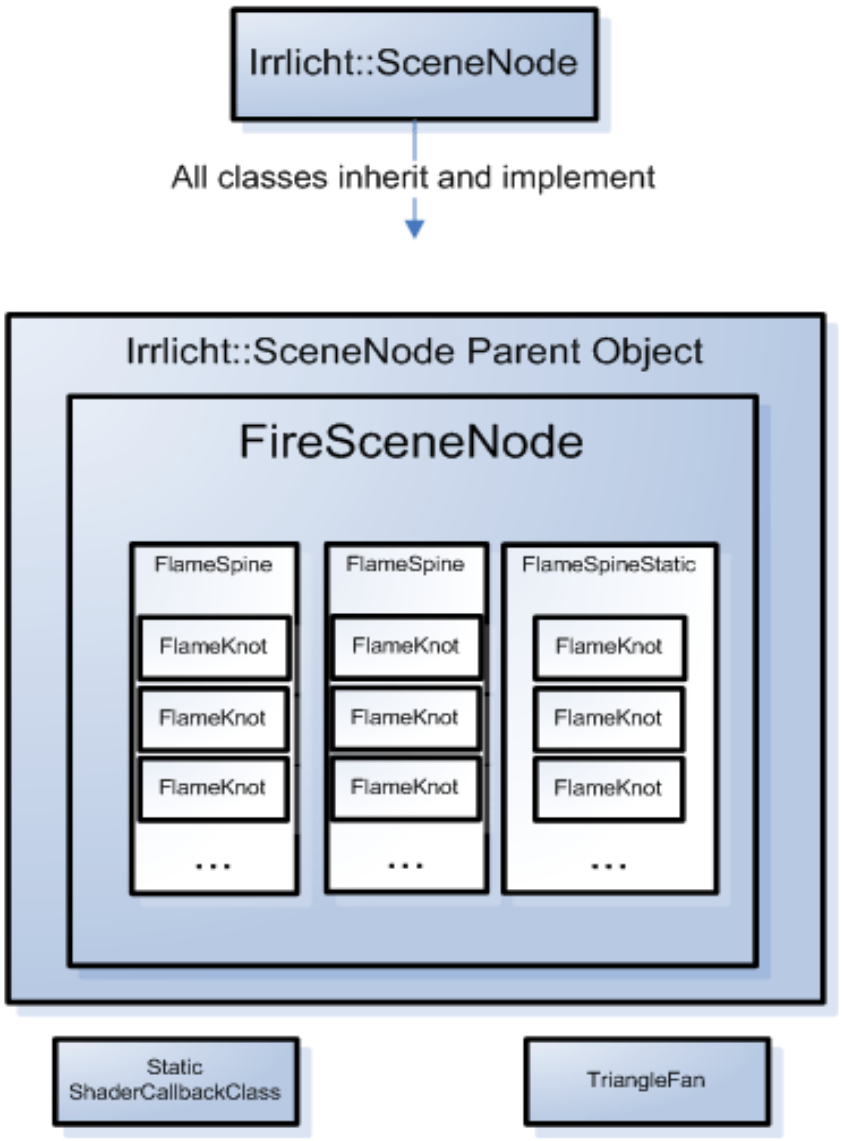


Figure 6.1. Prometheus Program Structure.

Figure 6.2 shows the Prometheus framework. Parameters are adjusted with the sliders on the left.

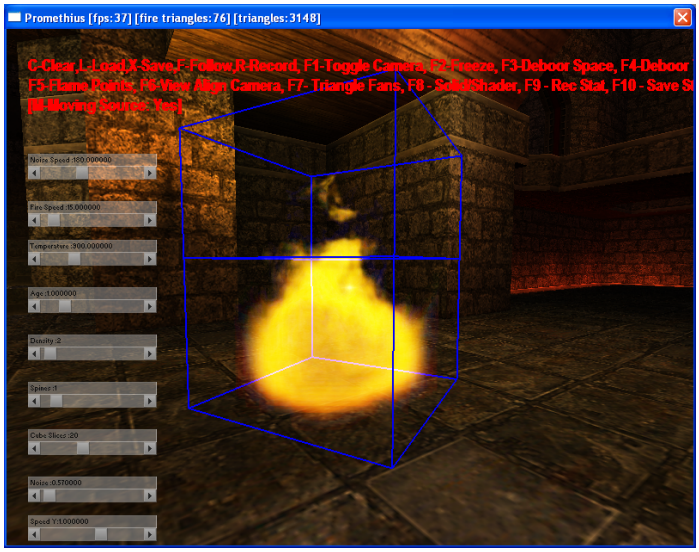


Figure 6.2: Prometheus: Fire Framework for Feasibility Study. Final rendering of fire.

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 use this octree map as the necessary attribute of a true production system. One individual fire volume is displayed in this image. It represents a standalone fire or a flame, depending on the character of the fire.

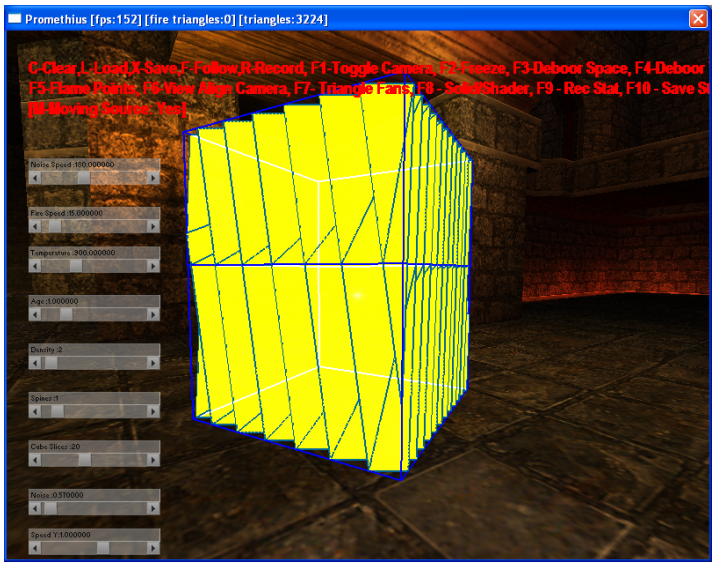


Figure 6.3: Prometheus: Fire Framework for Feasibility Study. Fire volume view-aligned surfaces.

The volume is made up of two sub-volumes, displayed in blue (Figure 6.2). 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 the 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 holistic, volumetric look.

7. Conclusion

As a result of this thesis, a fire-rendering framework has been written using the Irrlicht 3-D engine [35], utilizing established and novel algorithms. Improved algorithms have been described. Physical and procedural controls are leveraged to explore performance of the fire model.

Some examples of fire produced with Prometheus can be seen in Addendum 2 on Page 83. Figure A1 is a candle flame with detail enlarged at the top of the flame. Figure A2 displays a simulation of a fire significantly hotter than a wood fire, i.e. a natural gas fire. Figure A3 contains a fire model with electrical discharges on top. Figure A4 gives an example of several fires blended in a scene. Figure A5 is a flame that simulates a fire which appears distorted as if observed through a stained glass. A6 to A10 represent the evolution of the single flame in time.

Statistics have been collected to determine system bottlenecks and to test the feasibility of the model. I 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

1. Nancy M. Amato, Michael T. Goodrich, Edgar Ramos, "Linear-Time Triangulation of a Simple Polygon Made Easier Via Randomization," *Discrete and Computational Geometry*, 26:245-265, 2001.
2. P. Beaudoin, S. Paquet, P. Poulin, Realistic and Controllable Fire Simulation, *Proceedings of Graphics Interface* 2001, June 2001.
3. A. Benassarou, E. Bittar, N.W. John, L. Lucas, MC Slicing for Volume Rendering Applications, *In International Conference on Computational Science* (2), 2005.
4. Blum L., Blum M., Shub M., A simple unpredictable pseudo-random number generator. *SIAM Journal on Computing* 15,2 (May, 1986), pp. 364-383
5. D. Blythe, *DirectX*, The Direct3D 10 System, Microsoft Corporation, 2006.
6. 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, pp. 91-98.
7. B, Chazelle, "Triangulating a Simple Polygon in Linear Time." *Discreet and Computational Geometry,* Volume 6, pp. 485-524, 1991.
8. 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.
9. G. Farin, *Curves and Surfaces for Computer-Aided Geometric Design,* fifth ed. ISBN-10: 1558607374, Academic Press, San Diego, CA, USA, 2002.
10. 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.
11. N. Foster and D. Metaxis, Controlling Fluid Animation, *Proceedings of the 1997 Conference on Computer Graphics International*, 1997**.**
12. N. Foster and D. Metaxas, Realistic Animation of Liquids, *Graphical Models and Image Processing*, 58(5), 1996, pp. 471–483.

13. 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.
14. K. Hawkins, D. Astle, *OpenGL Game Programming*. ISBN 0-7615-3330-3, Chapter 15 Special Effects. Using Particle Systems.
15. B. D. Kandhai. *Large Scale Lattice-Boltzmann Simulations*, PhD thesis, University of Amsterdam, December 1999.
16. J. Kessenich, D. Baldwin and R. Rost, *The OpenGL Shading Language*. Version 1.10.59. 3Dlabs, Inc. Ltd.
17. S. A. King, R. A. Crawfis, and W. Reid. Fast Volume Rendering and Animation of Amorphous Phenomena. *Volume Graphics*, pages 229–242, 2000.
18. J. Kniss, G. Kindlmann, C. Hansen: Interactive volume rendering using multidimensional transfer functions and direct manipulation widgets. *In Proceedings of the conference on Visualization '01.* pp. 255-262, 2001
19. A. Lamorlette, N. Foster, *Structural Modeling of Flames for a Production Environment*, DreamWorks, Association for Computing Machinery, Inc. (ACM), 2002.
20. W. Lorensen, H. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics (SIGGRAPH 87 Proceedings)* 21(4) July 1987, pp. 163-170
21. 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.
22. D. Nguyen, R. Fedkew, H. Jensen, *Physically Based Modeling and Animation of Fire*, Association for Computing Machinery, Inc. (ACM), 2002.
23. M. Olano, *Modified Noise for Evaluation on Graphics Hardware*, ACM SIGGRAPH/Eurographics Graphics Hardware, 2006.
24. J. O'Rourke. *Computational geometry in C*. Cambridge University Press, 1994. 4 R., Chapter 1, 2
25. 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.

26. 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
27. R. J. Rost, *OpenGL 2.0 Overview*, 3Dlabs, Inc., February 2002
28. R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry: Theory and Applications*, 1(1):51-64, 1991.
29. 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.
30. 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.
31. *Primitives (Direct3D 9)*, Microsoft Developer Network, http://msdn2.microsoft.com/en-us/library/bb147291.aspx
32. Stefan Gustavson, *Simplex noise demystified*, Linköping University, 03/22/2005, http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf
33. Hugo Elias, *Perlin Noise*, http://freespace.virgin.net/hugo.elias/
34. *ATI ShaderMonkey, ATI Research Inc.*, http://ati.amd.com/developer/rendermonkey/index.html
35. *Irrlicht 3-D engine,* http://irrlicht.sourceforge.net/.
36. *Company of Heroes*, Relic Entertainment, PC, September 14, 2006.
37. *F.E.A.R.*, Monolith Productions, PC, October 18, 2005
38. *Guild Wars*, ArenaNet, PC, April 28, 2005.
39. *Half Life 2*, Episode 1, Valve Corporation, PC, June 2006.
40. *Call of Cthulhu*: Dark Corners of the Earth, Headfirst Productions, October 24, 2005 (Xbox), March 27, 2006 (PC).
41. *World of Warcraft and World of Warcraft: The Burning Crusade*, Blizzard Entertainment, PC, November 23, 2004.
42. *Elite*, Acornsoft, Firebird, PC 1984

43. Menelaus of Alexandria, Encyclopædia Britannica, 2007. Encyclopædia Britannica Online. 11/11/2007 <http://www.britannica.com/eb/article-9052003>.
44. T. Porter and T. Duff. Compositing digital images. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 253-259, July 1984.
45. Carl de Boor, *A Practical Guide to Splines*. (1978) Springer-Verlag, 113-114.

Addendum 1: Feasibility Analysis Tables

Table A1. Without Octree Environment

Lattice Density	Slices	Flame Volumes	Screen Space Occupied	Gradient Noise	Noise Octaves	Triangulation	Physics Simulation	Scene Triangles	Renderer	Fire Triangles	Frames Per Second
2	20	1	4.0%	**Perlin's Improved Noise**	4	MC	No	3174	OpenGL/GLSL	70	30
2	20	1	4.0%	**Simplex Noise**	4	MC	No	3174	OpenGL/GLSL	70	53
2	20	1	4.0%	**Modified Noise**	4	MC	No	3174	OpenGL/GLSL	70	119

Table A2. With Octree Environment

Lattice Density	Slices	Flame Volumes	Screen Space Occupied	Gradient Noise	Noise Octaves	Triangulation	Physics Simulation	Scene Triangles	Renderer	Fire Triangles	Frames Per Second
2	20	1	4.0%	**Perlin's Improved Noise**	4	MC	No	3174	OpenGL/GLSL	70	22
2	20	1	4.0%	**Simplex Noise**	4	MC	No	3174	OpenGL/GLSL	70	45
2	20	1	4.0%	**Modified Noise**	4	MC	No	3174	OpenGL/GLSL	70	101

Table A3. Triangulation Test

Lattice Density	Slices	Flame Volumes	Screen Space Occupied	Gradient Noise	Noise Octaves	Triangulation	Physics Simulation	Scene Triangles	Renderer	Fire Triangles	Frames Per Second
8	20	1	7.0%	Modified Noise	4	**MC**	No	3885	OpenGL/GLSL	300	68
8	20	1	7.0%	Modified Noise	4	**Convex Optimized**	No	3836	OpenGL/GLSL	303	53
8	20	1	7.0%	Modified Noise	4	**Convex**	No	3740	OpenGL/GLSL	301	56

Table A4. Density Test

Lattice Density	Slices	Flame Volumes	Screen Space Occupied	Gradient Noise	Noise Octaves	Triangulation	Physics Simulation	Scene Triangles	Renderer	Fire Triangles	Frames Per Second
3	20	1	7.5%	Modified Noise	4	MC	No	3816	OpenGL/GLSL	100	87
8	20	1	7.5%	Modified Noise	4	MC	No	3816	OpenGL/GLSL	258	69
13	20	1	7.5%	Modified Noise	4	MC	No	3816	OpenGL/GLSL	407	59
19	20	1	7.5%	Modified Noise	4	MC	No	3816	OpenGL/GLSL	521	54
25	20	1	7.5%	Modified Noise	4	MC	No	3816	OpenGL/GLSL	644	50

Table A5. Slice Test

Lattice Density	Slices	Flame Volumes	Screen Space Occupied	Gradient Noise	Noise Octaves	Triangulation	Physics Simulation	Scene Triangles	Renderer	Fire Triangles	Frames Per Second
2	**10**	1	4.0%	Modified Noise	4	MC	No	3174	OpenGL/GLSL	34	168
2	**20**	1	4.0%	Modified Noise	4	MC	No	3174	OpenGL/GLSL	70	101
2	**30**	1	4.0%	Modified Noise	4	MC	No	3174	OpenGL/GLSL	106	73
2	**40**	1	4.0%	Modified Noise	4	MC	No	3174	OpenGL/GLSL	145	56
2	**50**	1	4.0%	Modified Noise	4	MC	No	3174	OpenGL/GLSL	145	46

Table A6. Screen Space

Lattice Density	Slices	Flame Volumes	Screen Space Occupied	Gradient Noise	Noise Octaves	Triangulation	Physics Simulation	Scene Triangles	Renderer	Fire Triangles	Frames Per Second
2	20	1	**4.0%**	Modified Noise	4	MC	No	3174	OpenGL/GLSL	70	101
2	20	1	**9.0%**	Modified Noise	4	MC	No	3174	OpenGL/GLSL	70	51
2	20	1	**30.0%**	Modified Noise	4	MC	No	3174	OpenGL/GLSL	70	28
2	20	1	**40.0%**	Modified Noise	4	MC	No	3174	OpenGL/GLSL	70	15
2	20	1	**50.0%**	Modified Noise	4	MC	No	3174	OpenGL/GLSL	70	12

Table A7. Renderer -Environment Only

Lattice Density	Slices	Flame Volumes	Screen Space Occupied	Gradient Noise	Noise Octaves	Triangulation	Physics Simulation	Scene Triangles	Renderer	Fire Triangles	Frames Per Second
2	20	1	4.0%	-	-	-	No	3174	OpenGL/GLSL	-	252
2	20	1	4.0%	-	-	-	No	3174	DirectX 9/HLSL	-	231

Table A8. Renderer - Simplex Noise

Lattice Density	Slices	Flame Volumes	Screen Space Occupied	Gradient Noise	Noise Octaves	Triangulation	Physics Simulation	Scene Triangles	Renderer	Fire Triangles	Frames Per Second
2	20	1	4.0%	Simplex Noise	4	MC	No	3174	OpenGL/GLSL	70	51
2	20	1	4.0%	Simplex Noise	4	MC	No	3174	DirectX 9/HLSL	70	72

Table A9. Renderer - Perlin Noise

Lattice Density	Slices	Flame Volumes	Screen Space Occupied	Gradient Noise	Noise Octaves	Triangulation	Physics Simulation	Scene Triangles	Renderer	Fire Triangles	Frames Per Second
2	20	1	4.0%	Perlin Noise	4	MC	No	3174	OpenGL/GLSL	70	53
2	20	1	4.0%	Perlin Noise	4	MC	No	3174	DirectX 9/HLSL	70	29

Table A10. Renderer - Modified Noise

Lattice Density	Slices	Flame Volumes	Screen Space Occupied	Gradient Noise	Noise Octaves	Triangulation	Physics Simulation	Scene Triangles	Renderer	Fire Triangles	Frames Per Second
2	20	1	4.0%	Modified Noise	4	MC	No	3174	OpenGL/GLSL	70	101
2	20	1	4.0%	Modified Noise	4	MC	No	3174	DirectX 9/HLSL	70	79

Table A11. Varying Fire Volumes

Lattice Density	Slices	Flame Volumes	Screen Space Occupied	Gradient Noise	Noise Octaves	Triangulation	Physics Simulation	Scene Triangles	Renderer	Fire Triangles	Frames Per Second
2	20	**1**	4.0%	Modified Noise	4	MC	No	10	OpenGL/GLSL	70	119
2	20	**2**	8.0%	Modified Noise	4	MC	No	10	OpenGL/GLSL	140	62
2	20	**3**	12.0%	Modified Noise	4	MC	No	10	OpenGL/GLSL	210	42

Addendum 2: Prometheus Fires

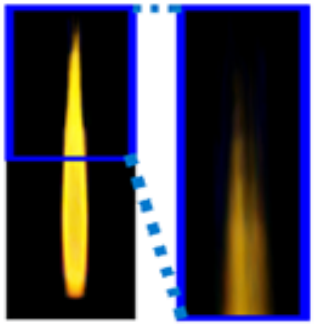


Figure A1. Candle Flame

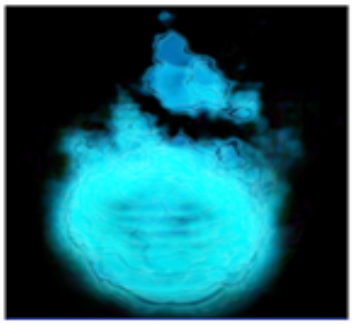


Figure A2. Blue Gas

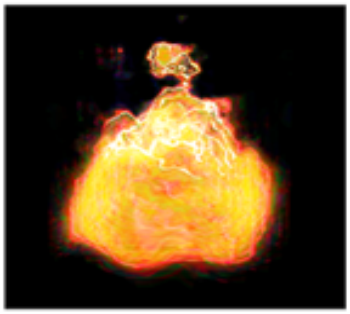


Figure A3. Electrical Charge

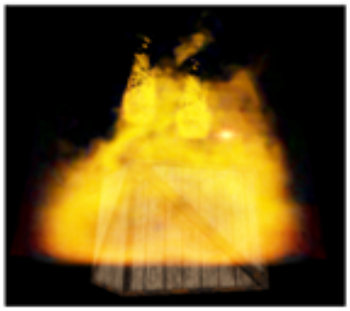


Figure A4. 3 Fires in a Scene

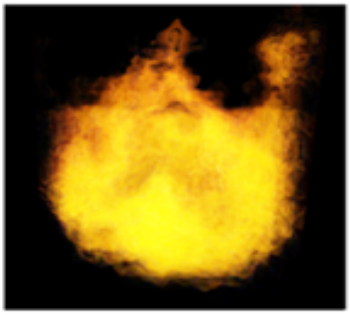
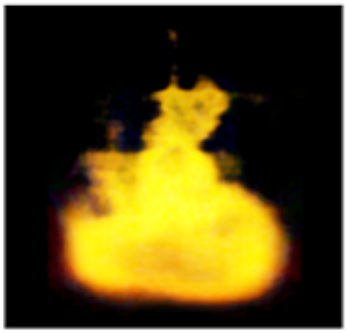
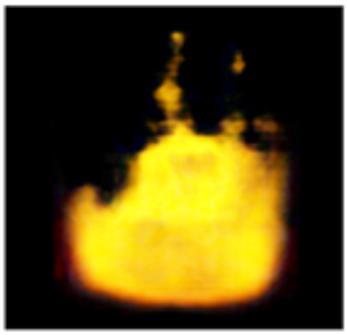
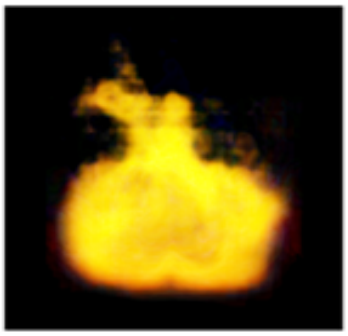
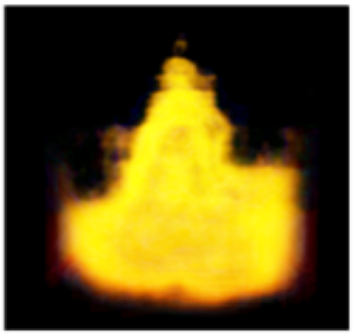


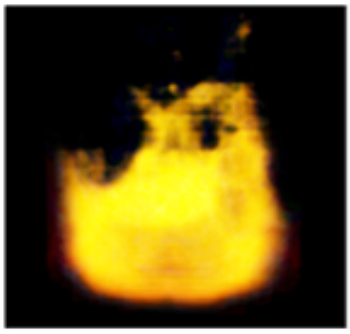
Figure A5. Artistically Rendered Fire











Figures A6 - A10. Evolution of Flame

Screenshots, movies and demo of the volumetric fire at **http://vanzine.org/fire**