A Computer Graphics Simulation of Fire

Using Particle Systems and Hypertextures

Christian Baekkelund
Jessica Laszlo
Christa Starr

Abstract

Creating realistic images of non-rigid, fluid phenomena such as clouds, smoke, and fire is one of the most interesting and challenging problems in computer graphics. This project focuses on modeling and rendering the natural phenomenon of fire in an attempt to create a visually acceptable model of its motion and appearance.

Introduction

When William Reeves published his paper *Particle Systems - A Technique for Modeling a Class of Fuzzy Objects* in 1983, he opened up a range of new possibilities for the field of computer graphics. Previously, graphics had been primarily successful when applied to the modeling of solid objects that could be easily represented by geometrical primitives such as spheres, rectangles, and polygonal objects (e.g. cars, furniture, buildings, and similar structures). Reeves's paper presented a method for depicting other types of objects, not as a set of primitive surface elements, but as a cloud of particle primitives that defined the object's volume. In his system, particles are created, given a certain motion, then eventually die off. The particles can also undergo changes in form, size, color, location, transparency, mass, and other similar characteristics. The modifications made to these characteristics depend on the type of natural phenomenon being represented.

There are many possible approaches to creating realistic natural motion in a particle system. Jos Stam has written a number of papers detailing methods with which to move particles using turbulent wind fields, fluid dynamics Navier-Stokes equation solvers, and diffusion processes (Stam 93, 95). Some others methods rely on using noise and fractal functions to create a ’hypertexture’ lattice through which the particles move (Ebert, et al). Other work has been done to move the particles in a mock frequency domain using spectral synthesis before translating them back to a world space domain (Sakas 93).

Similarly, there are many approaches to rendering the particles once a motion has been created. Simple raytracing is often used, however it frequently cannot accurately reflect the gas-like nature of the particle systems people are trying to render. Another method is to volumetrically render the particles using a raymarching technique (Musgrave, in Ebert et al.). Other methods include stochastic rendering based on random variables (Shinya 92). One reason we picked this project was the opportunity it afforded us to look into these different issues and weigh their respective challenges, benefits, and disadvantages.

Another motivating factor for this project came from the fact that our class would not be covering this topic in much depth. Since their introduction, particle systems have grown steadily in usage to the point where they are now a popular topic for computer graphics courses, as well as a common feature of commercial graphics packages (Dynamation and Maya, for example) . Although these systems are becoming more available in commercial packages and (in a smaller sense) in the public domain of shareware, we felt it would benefit us greatly to look into this rapidly growing field, as there is still a lot to be learned in creating a particle system and applying it to a simulation of some real world ’fuzzy’ modeling problem.

Goals

For our 6.837 Final Project, we set out to create a powerful and easy to use program that would create a continuous fire animation. We also wanted the program to be modular and robust enough to allow extensions for modeling various other natural phenomena using particle systems.

The goal for our fire program was twofold. First, we intended to write a particle simulator to model a realistic approximation of fire: its motion and general coloring. Second, we aimed to create a rendering mechanism to draw the particles more accurately, and with more detail. Additionally, to facilitate interaction with the system, we wanted to provide a front-end GUI. The GUI would allow a user to customize a set of options to render a number of frames to create a short, unique, and dynamic animated clip of the fire.

Achievements

We were able to accomplish most of our goals. We coded a rather powerful and easy to use GUI which interfaces with both our particle system simulator and our renderer. A user can run our program, set a number of options in the GUI detailing how they want the fire to act and look, then the particle system and renderer can each run according to their specifications, writing a series of frames to disk (as TIFF files).

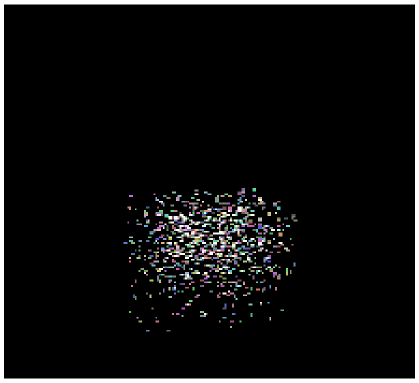
Currently, due to time constraints, the renderer and particle system simulator we have implemented are fire-specific. However, our program has been designed with highly modular object-based methods, such that one could easily implement particle systems to simulate smoke, steam, or other similar effects. A corresponding renderer could be easily implemented as well.

Approach

Our project was essentially divided into two sections: particle simulation and rendering. For our simulation, we intended to create an approximation of the Navier-Stokes fluid dynamics equations for our fire motion. For the rendering, we intended to modify a raytracer to handle "warped blobbies" as described by Jos Stam in his paper *Depicting Fire and Other Gaseous Phenomena Using Diffusion Processes*. However, in doing research we realized that these approaches were far more complex than we could realistically handle given the time frame of the project. We then researched the use of hypertextures in *Texturing and Modeling: A Procedural Approach* by Ebert et al., which described creating approximations of physical phenomena using specialized spatial noise and rendering them with a volumetric raymarching technique. This approach seemed much more realizable in the project's time frame.

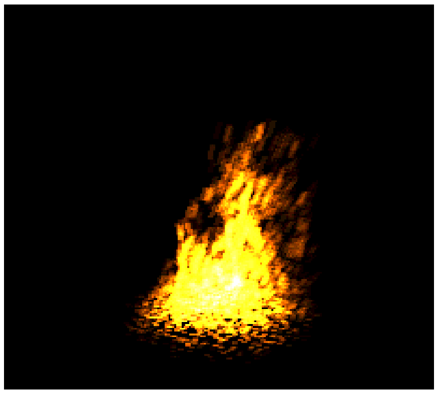
Implementation Overview

We started by building a particle system similar to the one described in Reeves' paper (Reeves 83). Since his system was used to create a fire effect in *Star Trek II: The Wrath of Khan*, it seemed well-suited to our task. The Reeves method emits a number of particles from an emission area, assigns each particle initial positions, velocities, colors and lifespans, moves them in some fashion at each time step, then extinguishes them when their lifespan is completed. Though other (more recent) particle system implementations allow for particle-to-object collisions and particle-to-particle interactions, for the purposes of modeling fire we felt that these would slow down our processing considerably, and they would not be as important as in, for example, depicting flocking behaviors (another popular application of particle systems).



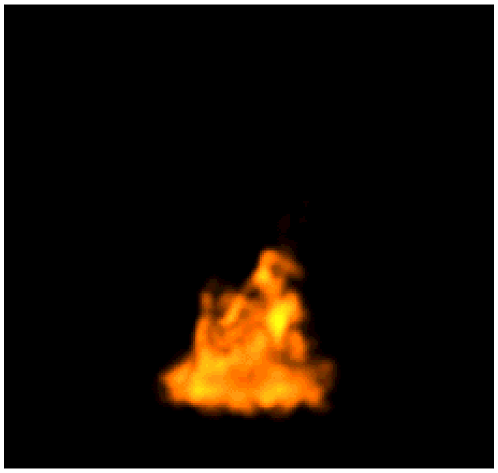
Our first particle system utilized our Point and Particle classes (described below) to represent the individual particles, and the ParticleSystem class to give them motion and draw them. We also created a simple display mechanism using OpenGL (specifically, the GLUT) such that the particles could be rapidly drawn to the screen as short lines (one vertex at position, the other at position + velocity*timestep). We also created a rough version of our GUI to be using simple OpenGL-GLUT event handling. We gave our particles randomized colors and simple upward motions. The result is shown to the left.

Once we had a simple system running, we turned our attention to motion simulation. It would not be enough to apply random forces to our particles, as we needed to create a repeatable motion (each particle should be affected the same way when the simulation is started over). The book *Texturing and Modeling* describes various applications of pseudo-random noise functions in creating a ’hypertexture’ grid through which objects would move. The pseudo-random functions guarantee that repeated calls to the noise functions with the same inputs would produce the same results. For a more in-depth discussion of the noise, noise/gradient, and fractal functions we used, see the sections under ’Various Math Functions’.



We chose to represent the motion of our particles as a result of summation of several forces. After observing fire in all its splendor, we decided that there were three prevailing forces acting on fire. The first was an upward component, which we called a rising function. The second was a prevailing wind function causing a "gusty" sideways motion of the fire. Finally, there seemed to be some randomized turbulence affecting the fire as a whole. We used our noise and fractal functions to moderate the effects of these primary forces. We also determined that the temperature of fire (as opposed to the mass of the particles in traditional systems) most closely determined the effect of these forces on its motion. As fire gets cooler, it is more effected by wind, and turbulence, and rises more slowly. Therefore, we added a temperature attribute to our particles and made their motion functions and colors dependent on the temperature. An initial test of our particle fire is shown above. For drawing the particles, we simply represented them as GL lines with vertices dependent on the current position and velocity of the particles, and colors dependent on their temperature.

The final part of our project was the rendering mechanism. We built a simple raycaster/raymarcher to do volumetric rendering of particles. We cast a ray from each pixel point in our screen. Our particles were represented as spheres with a density falloff from the center of the sphere to the edges, determined by the particle’s temperature. As the ray passed through our space, it checked against the particles to see which of them intersected the ray. That particle’s influence on the ray was calculated according to its density along the span of the ray intersecting the sphere. Each span was added to the ray, then once all particles had been checked and the spans sorted according to t values from the eyepoint to the far clipping plane, we marched down the ray. Calculating contributions of the density spans to the color and opacity of the ray, we combined all these influences into the total color of the pixel point we cast the ray from. An example of this rendering is shown below.



Implementation Problems

As we see it, there are two major problems with our fire rendering. First, there is no accounting for the velocity of the particle in the rendering. Second, there is no motion blur. The first could be solved by representing our particles as ellipses instead of spheres, with the focal points determined by particle position and velocity (similar to the way in which we draw the GL line representation). As for motion blur, we would have to calculate two successive frames simultaneously then combine the position and velocity of a point in each frame to come up with a blurred image.

A number of additions could also be made to our current fire rendering methods. Most importantly, if the volumetric sphere objects were back-traced through previous steps and correspondingly warped according to previous forces, such as Jos Stam describes in his work (Stam 93, 95), our fire could have a more tensile nature and look less solid.

Implementation Specifics

Mathematical Functions

In creating the particle motion simulations, we used several mathematical techniques described in *Texturing & Modeling* to create pseudo-randomized forces on our particles.

Noise

Our noise functions were based on the Perlin noise functions described in *Texturing & Modeling*. The first type of noise described involved using a permutation table to create a pseudo-randomized sequence. The table depicts a ’lattice’ of numbers - feeding sequential integers into this lattice gives back the pseudo-random, repeatable sequence. However, simply using this type of noise function creates grid artifacts. To counteract the grid-pattern inherent in lattice noise, we also implemented a gradient mechanism, which calculates a gradient at the cell corners of the lattice, then interpolates between them. When combined, noise and gradient functions have far less artifacts than either one alone. A more complete discussion of these issues is can be found in *Texturing & Modeling*, pp. 66-74.

Fractal

A fractal is defined by F. Musgrave as "a geometrically complex object, the complexity of which arises through the repetition of form over some range of scale." (Ebert et al., 277) We implemented a fractal class to simulate complex motion of our particles. Our fractal function is similar to the one described by Musgrave in *Texturing & Modeling* (pg. 282) but also allows for the calculation of a gradient at the fractal point for smoother motion interpolation. The Fractal class relies heavily on the Noise class, and provides standard variable control features such as dimension, lacunarity, and cutoff (these features are referred to as H, lacunarity, and octaves in the Musgrave text). The dimension of a fractal defines its underlying shape, lacunarity the frequency, cutoff the amount of repetitive processing done of the fractal to create the final shape. Since the fractal function is implemented to give back numbers in the range (-1, 1), we also implemented helper functions to return a range from (0, 1).

Particle System Functions

Point

Our Point class is a data abstraction with 3 data members: x, y, and z coordinates. We used this class to represent any quantity that has 3 components; vectors, positions, colors, etc. All 3-d vectors we utilize in our system (velocities and wind forces, for example) make use of the Point class for their representation. The Point class provides supplemental mathematical functions to add, subtract, multiply and divide Points against each other and against floats. Finally, simple vector arithemetic functions were added to compute dot products, cross products, and to handle vector normalization.

Particle

Our Particle class is a container for particle attributes. It holds the particle position, velocity, color, temperature, age, lifespan and particle id number. Position and velocity were used to store the motion of the particle. Color and temperature were used to draw the particles as colored lines in the simple GL drawing mode. Temperature was used in the volumetric rendering scheme as well, to compute particle sizes and densities. Age, lifespan, and temperature were used to determine how long each particle would live. This class could be potentially expanded to include such other characteristics as mass and electromagnetic charge.

ParticleSystem and FireSystem

Our ParticleSystem is a general class that handles particle emission, movement, deletion, and simple line drawing. ParticleSystem includes a step() function that upon every step will create particles, call an appropriate function to update (modify) particles as necessary, and finally delete any old particles that have outlived their usefulness. A ParticleSystem also has another major function, drawScene, which also acts on a time step and draws all of our active particles at that time step as GL lines on the screen. FireSystem is a subclass of ParticleSystem that treats the particles specifically as fire particles. While particle creation and deletion are inherited from ParticleSystem, the updating of its particles is different. The creation of a FireSystem requires a number of arguments, several directly applicable to the particle system (such as emitter and fractal characteristics), as well as fire-specific variables such as coolRate (how quickly the particles lose their temperature), rise (the strength of the rise force on the particles), wind (a vector describing the direction of the overall wind), turbAmp and turbScale (to define the strength of the turbulence force on the particles.

Other functions of interest in FireSystem include the functions to calculate rise, wind, and turbulence forces. RiseForce returns a vector that calculates a linear function of the rise variable designed to have less effect on the particle as it cools. WindForce returns a vector that uses a fractal function of time to create a varied wind force, and calculated to have more effect on the particle as it cools. TurbForce also returns a vector, calculated by a fractal and gradient function based on the particle's position and the current time, having more effect on the particle as it cools.

Volumetric Renderer

The volumetric renderer was an extension of the raymarcher idea put forth by Musgrave in Chapter 9 of *Texturing & Modeling*. The general theory was to cast rays from each pixel on the screen, then 'march' down the ray checking for intersections the particles. At each march step, particles were checked to see if they influenced that section of the ray. Each particle was represented as a Sphere with a range of influence which could be used to determine how much that particle contributed to a color at that step of the ray. Sphere influence was determined by the radius and weight of the sphere, calculated according to the particles’ temperature. After the entire ray had been marched down, we collected all the influences on the ray into a final color and opacity.

Render

Our main rendering handler (encapsulated in Render.c) contains the functions needed for calculating Ray origins and directions, creating the VolSpheres from the particles, then intersecting these VolSpheres with each ray. Ray origins here are taken as the eye point dictated into the GUI (and also used in setting the corresponding projection matrix in OpenGL). The direction is then found by using gluUnProject twice to unProject screen-space onto world-space coordinates at two different z-depths and interpolating them to find a direction.

The renderer creates rays for every screen space pixel, casts them to find their intersections with any of the current Volumetric objects in the scene, then shades the pixel as is necessary (see Ray).

Ray

Our ray class was based on the ray caster from Problem Set 6. The Ray class holds an eye point and direction for a ray through each pixel in screen space. However, our ray was also responsible for collecting particle influences onto itself. We chose to represent these as ’spans’ that would accumulate along the ray through intersecting the ray with VolSpheres. VolSpan is a structure internal to ray that has a tMin and tMax to mark the extent of the span on the ray, and a pointer to the object that the span came from to be used in density calculations. So, when we started the ray-particle intersection calculations, we would add a VolSpan to a vector of VolSpans internal to the ray. These VolSpans were then sorted so that we could march down the ray and calculate the color of the pixel by calculating the contributions of the spans to the overall color.

Box

Our Box class provides a simple bounding box in space. Two points are enough to specify a unique box in space, hence that is our data representation for the box. The class provides methods that test whether a given point intersects the box, and whether a given ray intersects the box. Box is used by our other volumetric object classes.

VolObj

VolObj is a generic supertype, extended by VolSphere (and potentially VolEllipse in future implementations). Its constructor takes a Box as an argument. VolObj provides virtual methods to support density calculations of its subclasses and to Ray-object intersections. The intersectRay method of VolObj differs from Box’s method of the same name in that its subclasses will add the intersecting span (if existent) to the Ray.

Our VolObj class is also used in our data abstraction to efficiently store and sort the particles (a kdtree that we call Jtree) in that VolObj is the data representation for the Jtree’s partitions of space.

VolSphere

In the volumetric rendering scheme, our particles were represented by VolSpheres instead of points in space. The VolSphere was centered at the particle’s position, then given a radius and a weight. The weight and radius allowed us to calculate a density for each sphere that fell off in a smoothstep from the center of the particle to the edges. VolSphere also implements the intersectRay function which will add a VolSpan to the Ray according to the intersection length.

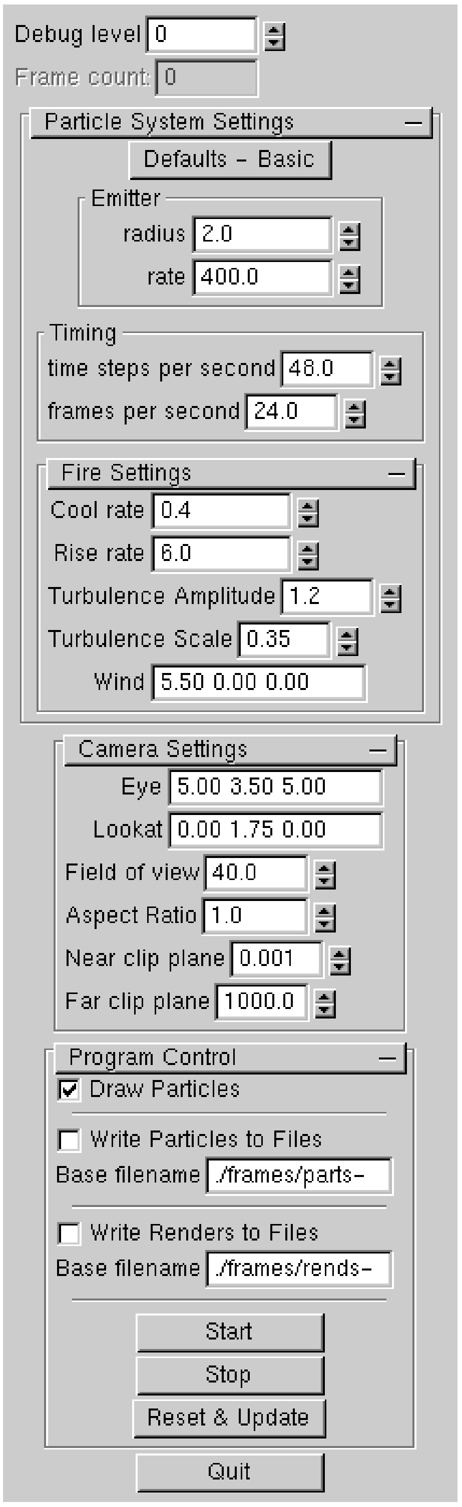
Jtree

Jtree is our implementation of a k-d tree class. A k-d tree provides more efficient storage for a large collection of particles. Storing the particles in a vector may be appropriate for a small number of particles, but as the collection grows larger, sorting becomes exorbitantly expensive (in terms of cpu time) and it is impossible to render frames in a timely fashion.

K-d trees partition space into buckets containing a pre specified number of items (in our case, 8). Jtree contains a constructor that makes a k-d tree from an array of VolObj, and a method to intersect a ray with the Jtree of particles. The intersect method checks first if the ray intersects the bounding box of the entire tree. If it does not, it can return immediately. If it does intersect, it recurses through the tree, calling the intersect method on the right and left child of the current node. The base case of recursion is reaching a leaf of the tree, the partition of VolSpheres that the ray intersects with. The routine then calls the intersectRay method for each of the VolSpheres in that VolObj leaf. (see the intersectRay method described in VolSphere and VolObj). Jtree reduces the computation time from O(n^2) to O(n lg n).

Interface

main.c



Debug level 0
Frame count: 0

Particle System Settings
Defaults - Basic

Emitter
radius 2.0
rate 400.0

Timing
time steps per second 48.0
frames per second 24.0

Fire Settings
Cool rate 0.4
Rise rate 6.0
Turbulence Amplitude 1.2
Turbulence Scale 0.35
Wind 5.50 0.00 0.00

Camera Settings
Eye 5.00 3.50 5.00
Lookat 0.00 1.75 0.00
Field of view 40.0
Aspect Ratio 1.0
Near clip plane 0.001
Far clip plane 1000.0

Program Control
[✓] Draw Particles
[ ] Write Particles to Files
Base filename ./frames/parts-
[ ] Write Renders to Files
Base filename ./frames/rends-

Start
Stop
Reset & Update
Quit

The Graphical User Interface (GUI) is the mechanism for the user to interact with our system by updating many system variables. We chose to implement our gui using the GLUI API. It was a convenient choice because it was designed to be compatible with OpenGL, and the API provides all the interface components necessary to create an intuitive and attractive user interface.

The GUI is made up of several rollout panels, grouping the system variables by their function: Particle System Settings, Camera Settings, and Program Control.

Particle System Settings This rollout contains all the variables having to do with the Particle System, in general. There are three panels: for the particle emitter, timing, and variables unique to the Fire System, such as cooling rate of particles, rise rate, wind forces and turbulence. In the future, should we implement a Water System, or Steam System, additional panels could be added to the Particle System rollout for the variables specific to each of those systems. There is also a button that, when pressed, will return all the settings to a pre-defined default set of system variables. When any Particle System settings are changed, in order for the changes to take effect, the user must click the "Reset & Update". This will destroy the current Particle System, and create a new one with the modified settings.

Camera Settings This rollout contains textfields to change the location of the eye point, the lookat point, field of view, aspect ratio, and near and far clip planes. These can all be updated in realtime. These work through accepting the necessary variables, and then creating the necessary projection matrices in OpenGL using primarily gluPerspective and gluLookat.

Program Control This rollout contains the global program control functions (Start, Stop, and Reset&Update), and checkboxes for the user to specify whether or not to write the particles and/or renders to file. Textfields are also provided for the user to specify the basename and location for the written frames.

At the bottom of the GUI is a "quit" button, to exit the program.

Individual Contributions

Christian Baekkelund:

- Parts of the GUI, including all event and process handlers
- All the OpenGL GLUT, GLU, and GL routines
- Some of the Particle and Point classes
- Most of the ParticleSystem class and some of the FireSystem class
- The eye-ray creation and main rendering loop
- The TIFF file saving mechanisms for particle systems and renderings

Jesscia Laszlo:

- Parts of the GUI
- The KD-tree
- Some of the Particle and Point classes
- Testing of final rendering look

Christa Starr:

- The pixel shading functionality (across many classes)
- Most of the FireSystem class and some of the ParticleSystem class
- The VolObject and VolSphere classes
- The Bounding Box
- The Ray Marching mechanism (the Ray class)

Lessons Learned

Large software projects present many difficulties not immediately obvious from the outset, even when the problem to be solved appears straightforward. Coordinating the implementation of the different segments of the project present logistical difficulties. Debugging can be extremely frustrating and tedious, as a small bug can result in catastrophic errors hindering the progress of all the members of a group. We discovered, over the term of the project, that particle systems, while widely used, are not trivial, and that 6 weeks was not enough time to implement the long laundry list of items we initially set forth. One of the major discoveries we made was the complexity of the math involved in modeling complicated volumetric objects discussed in the Stam/Fiume paper, as well as the difficulty of implementing solutions to fluid dynamics equations. If the opportunity existed for us to continue our work, in order to make a more realistic representation of fire, we would want to implement the more complicated volumetric object (VolEllipse), and compare the quality and realism of the rendered final images.

We also discovered the glories of noise and fractal functions. We used them in many parts of our particle system and renderer. It seems like there's nothing they can't do.

Acknowledgements

Christa would like to thank:

- "The Dread Gypsy" Christopher Horvath of ILM, for teaching me that fractals and gradients are like chocolate and peanut butter (and for donating his work on Noise and Fractal to our cause), as well as for invaluable advice in all areas.

- Mike Root of MatteWorld, for making me ponder the philosophical ramifications of the term "good idea."

Christian would like to thank:

- Sam Leffler, author of the SGI Tiff library

Bibliography

- Ebert, David, Musgrave, F. Kenton, Peachey, Darwyn, Perlin, Ken, and Worley, Steven, *Texturing & Modeling: A Procedural Approach.*, Academic Press, 1998.
- Reeves, W.T. *Particle Systems -- A Technique for Modeling a Class of Fuzzy Objects*, SIGGRAPH 83, 359-376.
- Rademacher, Paul. GLUI User Interface Library, v1.01. http://www.cs.unc.edu/~rademach/glui/
- Sakas, G. "Modeling and Animating Turbulent Gaseous Phenomena Using Spectral Synthesis." *The Visual Computer*, 9:200-212, 1993.
- Stam, Jos. *Stable Fluids*, Proceedings of SIGGRAPH'99. In Computer Graphics Proceedings, ACM
SIGGRAPH, 1999, 121-128.
- Stam, Jos, and Eugene Fiume. *Depicting Fire and Other Gaseous Phenomena Using Diffusion Processes.*
Proceedings of SIGGRAPH '95: 129-136.
- Stam, Jos, and Eugene Fiume. *Turbulent Wind Fields for Gaseous Phenomena*. Proceedings of SIGGRAPH
'93: 369-376.
- Stam, Jos. *Multi-Scale Stochastic Modeling of Complex Natural Phenomena*, PhD Thesis, Dept. of Computer
Science, University of Toronto, 1995.
- Stam, Jos. *A General Animation Framework for Gaseous Phenomena*. ERCIM Research Report R047.
- Stam, Jos. Multi-Scale Stochastic Modelling of Complex Natural Phenomena , PhD Thesis, Dept. of Computer Science, University of Toronto, 1995.
- Stam, Jos. A Multi-Scale Stochastic Modelling Primitive for Computer Graphics , Masters Thesis, Dept. of Computer Science, University of Toronto, 1991.
- Woo, Andrew. "Fast Ray-Box Intersection", *Graphics Gems #1* pg. 395.

Appendix

- makemovie -- a utility provided on SGI machines that we used to turn the Tiff files generated by our Renderer into a QuickTime or MPEG movie.
- our program is dependent upon OpenGL, the GLUT, the GLUI, and the SGI Tiff Library

---