Realtime Fire Simulation

Jeremy Spitz
Matthew Lindsay
Advanced Graphics Spring 2015
`https://github.com/DiegoArmando/FlameSim`

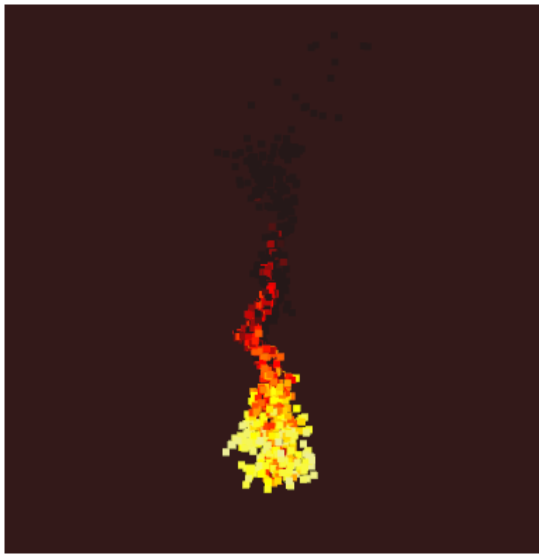

Figure 1: Base case of the project, showing a single source point flame's behavior.
`http://gfycat.com/PaltryHideousGroundhog`

1 Introduction

Flame simulation is nothing new to computer graphics. As early as 1996, researchers Nick Foster and Dimitri Metaxas were presenting the algorithms and techniques needed for fluid simulation [3]. Three years later, researcher Joe Stam was proposing modified techniques for use in computer graphics that would later become the basis for many iterations of calculating the behavior of fire [6] [5]. Many later attempts would get realistic results, but at the cost of being far from real time, needing minutes to calculate the state of the fluid after many physics iterations and performing the costly render. Other attempts, like Marc de Kruijf's Firestarter program whose is in real time, but may not behave in believable ways [2]. Many early game applications would use to run either pre-baked calculations or a series of flat, animated textures rotated for their normals to face the player to provide the illusion of flame. Nvidia was one of the only groups to have a flame simulation that was both real time and fairly believable. Their algorithm ended up very similar to our own [1], and was announced less than one year before the beginning of our project. Our aims in the project were to make a flame simulation that was both truly real time and exhibited believable behavior with a particle system by using a simplified and optimized version of traditional fluid simulations.

2 Previous Work

The intuition of how to perform a simulation of something like fire or water, which can be represented as a series of discrete particles, would be to model the system as points with forces which act upon them and rules dictating how they interact with each other. This sort of representation would immediately present a problem of n-squared complexity, as each particle would have to calculate its force contributions with every other particle in the system. Instead, a volumetric optimization is calculated and stored to allow for volumes of space to interact locally. To begin, the scene in question is divided into a series of identical cells, each representing a volume of space. These cells have stored a location, pressure, and velocity of fluid passing through a face in each of the three dimensions. In each time step, the pressures and velocities of each cell are calculated by iterating through the Navier-Stokes Equations to simulate liquid motion. With this information calculated, any arbitrary point in space (for example, the position of a particle at some point in time) can be interpolated. Each point finds the four closest cell's X, Y and Z velocities, and uses a weighted average of these velocities based on distance. This is the method described in the paper by Foster and Metaxas, and was iterated upon in further fluid simulations. In Nvidia's Flameworks software, each cell stores only one three dimensional velocity in its center, which cuts down required calculations threefold. This reduction allows Flameworks to be faster and more extensible, with many smaller cells used for the macro calculation of the flame-smoke fluid simulation. It is also worth noting that Nvidia is doing most of this on the graphics card; while this is a much faster solution than CPU based algorithms, this sort of programming was out of scope for the project, adding a layer of difficulty to the real time nature of our solution. Both of these solutions still have problems with scalability, allowing for only moderately sized flames for good performance. Rendering is another problem entirely. Foster and Metaxas's method is extensible to include how to render, but that is largely out of scope of the paper. Nvidia's method allows for particle, iso-surface rendering, or volumetric rendering (which they recommend). Both iso-surface and volumetric proved to be out of scope for the project, so our method used particle rendering to allow for visual representation of the believable behaviors we had calculated.

3 Our Solution

Our fire simulation algorithm has five main parts which will be gone over in detail in this section: Particle generation, Velocity Interpolation, Fuel Simulation, Heat Simulation, and Color Interpolation. The idea of our algorithm is to use these parts which represent the main some of the main properties that drive fire, and use them to generate a real time fire simulation in a non physically accurate way. Particle Generation defines the source of the flame, Velocity Interpolation represents the movement of the flame, Fuel Simulation represents the lifespan of the flame, Heat Simulation represents the the heat of the flame, and Color Interpolation determines the visual color of the flame.

3.1 Particle Generation

This algorithm serves as the spawn point for the particles, or what would be realistically known as the flame source. We generate particles randomly within specified region. Any specified region for a spawn point should work. Our examples are all spawning particles within 1 or more hemispheres. The particles are responsible for keeping track of their velocity, their heat, their fuel, and their position. To begin, the particles are given random initial velocities, and random heat and fuel to start with and are updated throughout the simulation. Further implementation details of this will be discussed later on.

3.2 Velocity Interpolation

Our velocity interpolations is very similar to Fluid Cell Based velocity interpolation mentioned in previous works, with one major difference. Instead of calculating the nearest points to use as the weighted values we used fixed cells relations from the current cell leading to slightly less realism, but also less computation. Furthermore, we have allowed the particles to be compressible as this requires less computations, and removes the necessity of emulating air particles. Lastly, we have set gravity to be going upwards, capped max velocity, and assigned specific velocities to the cells to start with. Our system allows customization of fire to one's desired needs. There are many options, changing the fuel/heat specific values set, changing the viscosity of the particles, changing the force of gravity, and changing the spawn points and initial velocities.

3.3 Fuel Simulation

Fuel is used in fire simulation in order to determine how long a particular particle lasts. Our fuel is randomly generated on a per particle basis by using a random value throughout a set number range which determine the life of the particle, how it changes color, how varied the flame is, length of the flame, as well as giving fire a more chaotic feel. When a particle runs out of fuel it is reset to the initial position and given a new random fuel value and correlating heat value. This range could be altered in order to give a more or less chaotic flame in some regards.

3.4 Heat Simulation

Heat is used in fire simulation in order to determine, the color at a specific point. In order to determine this, you need to calculate the heat for a particle at certain points. Our heat simulation algorithm is calculated on a per particle basis, and results in a value between 0 and 1. This is calculated by Heat=(current fuel)/(Initial fuel)-epsilon. Where epsilon is used to change what color it spawns at. This is used since the particles spawn at a random hemisphere, which would have been alive for a certain time. Additionally, it also allows some stylistic freedom of fire simulation, as following the goal of this algorithm; to make what we believe is a more believable flame, rather than emulate what is going on with the physical properties of a flame. Additionally, the heat also has a minimum value to emulate when a fire particle has turned into smoke, and should be set at a non-changing smoke color. The resulting heat value from all this is then passed onto the fragment shader and used in our color interpretation algorithm.

3.5 Color Interpolation

This is the algorithm we used to interpolate the heat value given for each particle into a color which determines how a particle of fire appears. This was implemented in the fragment shader. Our algorithm passes in the heat value (0-1) into the fragment shader which then uses that value to choose between mixes of 4 different colors based on the heat from the hottest 1(white), to .7(yellow), to .3(red), and 0(gray). It uses the Mix function to blend between the nearest 2 colors. These colors and values could easily be changed for different types of fires, or fantasy non-realistic fires, as shown in our examples.

4 Results and Discussion

Our results were fairly close to our vision for the project. The flames behave in convincing ways, and have interesting and dynamic emergent behaviors. Fires would flicker or particles of heated ash would drift away from the main flame before sputtering out and extinguishing. The system in place accounted for multiple flame sources, and allowed for both smaller, candle like flames, and more moderately sized campfire like scenes. There were unprecedented difficulties that occurred, and while they did lead to a slightly smaller scoped project with no blur as was originally intended, the main focus of the project, real time believable flame behavior, was still met. The simulation was not quite as fast as we had hoped for, and there are still optimizations that could be made. However, some additional uses we had not initially intended are now possible too, such as the use of our algorithm to create non-realistic flames, such as our fantasy flame example, as well as the ability to have customization of the flame to allow for more artistic freedom than what is just physically accurate.

5 Difficulties

There were a number of unanticipated problems which ended up slowing down development time significantly. The implementation of textures took up an entire week of development time before being abandoned for a better, faster alternative. Not only that, but our code base was fairly inflexible when it came to changing what information was sent to the shaders, which led to us discarding our original implementation of heat and using the method described above instead. Of such the main ones were difficulty implementing textures, difficulty passing variables into the fragment shader, and implementing blur which we did not have time to do. The problem we ran into with textures, is that it was surprisingly difficult to pass a texture or pass a variable to the fragment shader from the base code we were using. What we ended up doing instead is hijacking the red value of an already passed in color vec3 to represent our heat, which suited our needs, but was far beyond the original plan. The biggest problem by far though, was attempting to implement blur. Blur, as it turns out, can be implemented in any number of ways, all of which turned out to be either beyond the scope we could accomplish in our time frame or provided unsatisfactory results. There is the simplest method, which is to add a blur post-effect to the entire screen by linearly interpolating the color of the pixels around each particle to have the color's of each flame particle bleed out onto the screen. This would, however, render the simulation unusable in anything other than backgrounds of exactly one color, which is more restrictive than we wanted. There's the polygonal method, where spherical shells of decreasing opacity are placed around each particle depending on each particle's heat, but this was both difficult to implement in our time frame and very computationally costly. Figuring out a way to use the shells without slowing down the simulation would have taken double the time we had. And lastly, there is the method wherein a quad is placed in front of the camera to have the scene rendered onto. The simulation would then project each point onto the quad and modify its texture to include localized particle blurring, repeated for each particle in the scene. This turned out to have the same problem as the shells, being both too computationally expensive and too complicated to implement.

6 Further work

There are many different directions this project could be extended to include. First, changing initial conditions to lead to immediately believable state instead of several cycles of strangely behaving particles would be implemented. As it stands now, it takes a few resets of the particles to get a consistent looking flame. After that, testing and implementing the blurring algorithms discussed in Section 4 would be the second order of business, and would provide the most immediate improvement in rendering accuracy and verisimilitude. Once a blurring algorithm has been chosen and put into effect, further optimizing the fluid simulation components, both in algorithm used and in moving them onto the GPU as much as possible, would be the next notable improvement. One easy move would be to move the heat and/or fuel calculation onto the GPU simply by passing the variables, and doing the calculations there. Moving all code onto the graphics card to make it optimally efficient is a difficult and time consuming task, and would take as long or longer to develop as all work up to that point. However, once it has been moved, it becomes possible to further complicate the simulation with less significant impact on performance. Changing the simulation to be one of two fluids, fire in various states of combustion, and air, would allow for more emergent behavior, such as vortices. From there, depending on the desired uses for the simulation, collisions with meshes or the spreading of flames could be added, but that is beyond the initial scope of the project. At that point, it becomes adapting the smaller project into a larger, completely different one.

7 Conclusion

The project was successful in that we were able to do real-time fire simulation. Our solution provides what we believe to be a believable fire behavior, as well as allows for artistic choice and freedom if desired. However, a major shortcoming that holds the simulation back is that we did not have time to implement blur, which would allow for a more realistic and believable rendering of fire.

8 Contribution Summary

Jeremy Spitz worked on modifying the base code, working with the velocities, heat, point generation, and fuel algorithms, as well as helping with debugging how to transfer variables to fragment shader for a total of approximately 50 hours of work. Matthew Lindsay worked on the majority of the shader code, the color to heat algorithms, helped with modifying the base code, wrote much of the modified particle class, and added in all references to textures (now obsolete) for a total of approximately 50 hours of work.

References

[1] Nvidia flameworks. `https://developer.nvidia.com/flameworks`. Apr. 2014.

[2] Marc de Kruijf. firestarter – a real-time fire simulator. `http://pages.cs.wisc.edu/~dekruijf/docs/capstone.pdf`.

[3] Nick Foster and Dimitri Metaxas. Realistic animation of liquids. *Graphical Models and Image Processing*, 1996.

[4] FEDKIW R. NGUYEN, D. and H. JENSEN. Physically based modeling and animation of fire. *ACM Trans. Graph. (SIGGRAPH Proc.), vol. 29, 721–728*, 2002.

[5] Joe Stam. Stable fluids. *SIGGRAPH*, 1999.

[6] Joe Stam. Interacting with smoke and fire in real time. *Communications of the ACM Volume 43 issuer 7*, 2000.

9 Examples

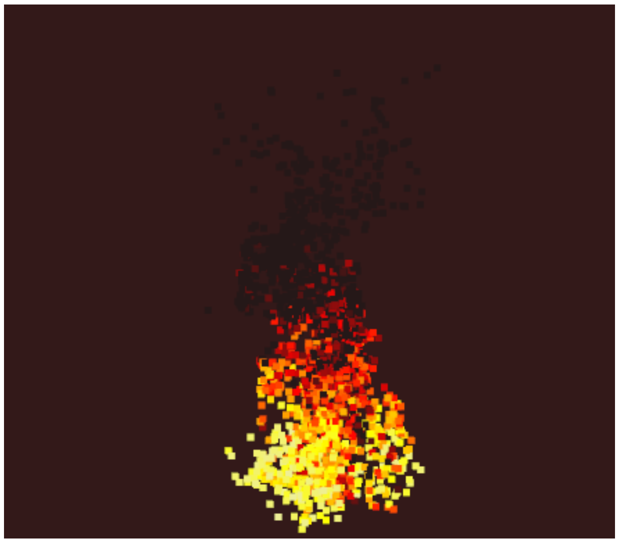

Figure 2: A more complex scene with three separate source points, demonstrating the ways in which the flames interact with each other in believable ways.
http://gfycat.com/BonySomeGorilla

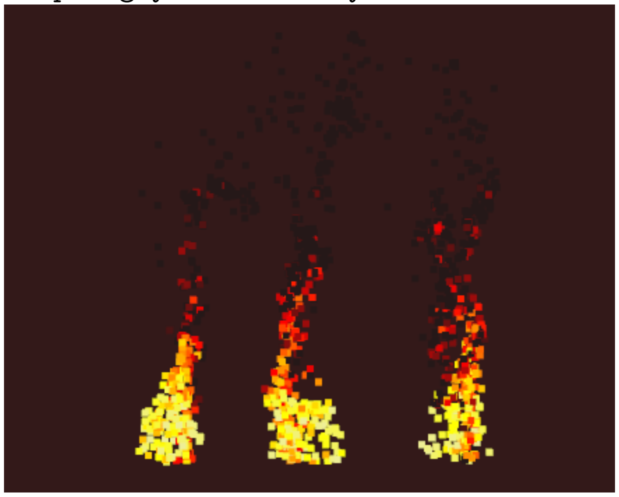

Figure 3: Case of three seperate source points placed further apart to give a better sense of how far apart flames influence each other.
http://goo.gl/Dvpimf

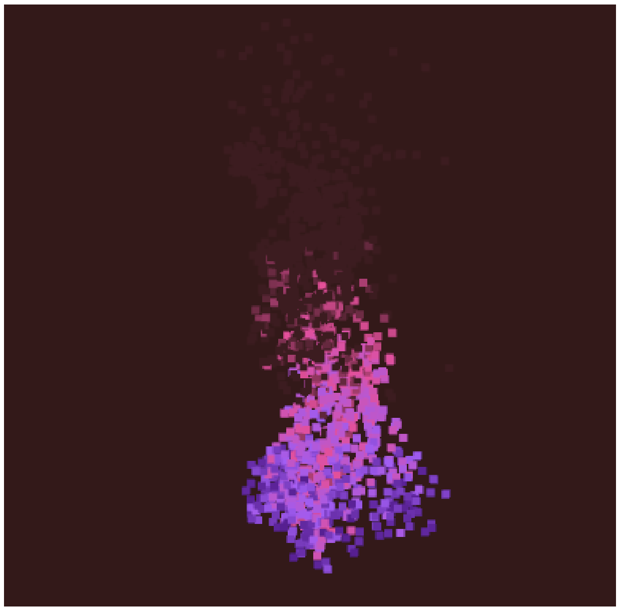

Figure 4: Our project supports arbitrary coloration to the artist's specifications in the shader.
http://gfycat.com/AnotherHorribleHaddock