Animating Fire

Spencer Van Leeuwen
University of Waterloo

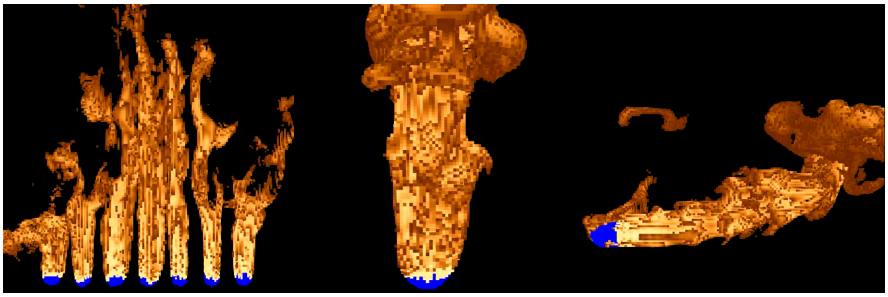

Figure 1: *3D fire animations*

Abstract

This is a report for my final project in the course *Physically-Based Animation*. The goal of my project was to implement a fire simulator. In this report, I discuss how I implemented the simulator and the results.

1 Introduction

From an explorer carrying a torch to a dragon wreaking havoc on a village, the ability to animate fire is extremely important for artistic expression. Physically-based animation and rendering of fire was introduced to the computer graphics community by [Nguyen et al. 2002]. When modelling fire, they model the fuel and flames as a two-phase flow. They also allow their fire to produce smoke as it cools, using concepts from [Fedkiw et al. 2001]. The model has also been extended by [Hong et al. 2007] to introduce wrinkled and cellular patterns into the flames. For this project, I have worked towards implementing these papers. In the following sections, I will describe the concepts and implementation details of my project, followed by a short discussion.

2 Overview

2.1 Basic Fluid Simulation Framework

When building any semi-Lagrangian fluid simulator, whether we want to simulate a liquid or a gas, there are several common elements that must be implemented. For this section, I followed the instruction of [Bridson 2015], which is the standard textbook for fluid simulation in computer graphics.

First, we want to store our velocities and pressures on a staggered grid, where the pressures are stored at grid centres and velocities are stored at the centre of the faces between elements, see Figure 2. We will store the pressures in p and the velocities for the x-, y-, and z-axis in u, v and w, respectively. We can think of this as separate grids when storing the data, but we must relate these structures back to the staggered grid structure. For example, the location of $u_{i,j,k}$ on the staggered grid is the same as the location of $p_{i-0.5,j,k}$. Note that if p has dimensions $n_x \times n_y \times n_z$, then u will have dimensions $n_x + 1 \times n_y \times n_z$, and similarly for v and w.

Our goal in a basic fluid simulator is to solve the Euler equations

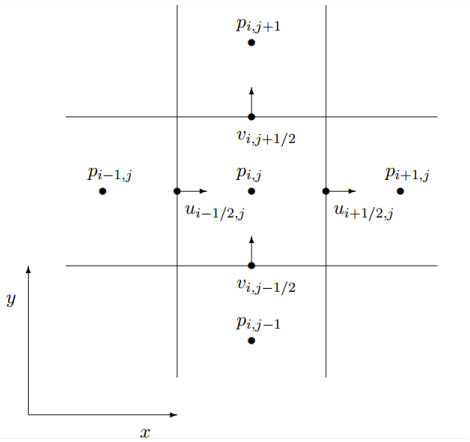

Figure 2: *2D representation of a staggered grid, taken from [Bridson 2015]*

for describing inviscid, incompressible fluid flow:

$$\begin{aligned} \frac{D\vec{u}}{Dt} + \frac{1}{\rho}\nabla p &= \vec{g}, \\ \nabla \cdot \vec{u} &= 0, \end{aligned} \tag{1}$$

where $\vec{u}$ the velocity of the fluid; $\frac{D\vec{u}}{Dt}$ is the material derivative of the velocity; ρ is the density; p is the pressure; and $\vec{g}$ is gravity.

2.1.1 Advection

To calculate $\frac{D\vec{u}}{Dt}$, we perform semi-Lagrangian advection using a time integration scheme; in particular, I used Runge-Kutta 3:

$$
\begin{aligned}
k_1 &= f(q^n), \\
k_2 &= f(q^n + \frac{1}{2}\Delta t k_1), \\
k_3 &= f(q^n + \frac{3}{4}\Delta t k_2), \\
q^{n+1} &= q^n + \frac{2}{9}\Delta t k_1 + \frac{3}{9}\Delta t k_2 + \frac{4}{9}\Delta t k_3,
\end{aligned} \tag{2}
$$

where q^n is a quantity q at the nth time step and f is some function. When advecting, q corresponds to what we are advecting, in this case velocity, and f is a function that gets the velocity at the point passed to it. During advection, we often need velocity values at points that do not lie on the grid, so we need to perform some sort of interpolation. For interpolation, I used a function for monotonic cubic interpolation introduced by [Fedkiw et al. 2001]:

$$
f(t) = a_3(t - t_k)^3 + a_2(t - t_k)^2 + a_1(t - t_k) + a_0, \tag{3}
$$

where

$$
\begin{aligned}
a_3 &= d_k + d_{k+1} - \Delta_k, \\
a_2 &= 2\Delta_k - 2d_k - d_{k+1}, \\
a_1 &= d_k, \\
a_0 &= f_k
\end{aligned}
$$

and

$$
d_k = \frac{(f_{k+1} - f_{k-1})}{2}, \Delta_k = f_{k+1} - f_k,
$$

which can be easily extended to 2 or 3 dimensions. The reason we want cubic interpolation is because linear interpolation is too dissipative and can cause artifacts. We will see later why we want a monotonic interpolator. After close inspection, it becomes obvious that cubic interpolation near the edges will sample outside the staggered grid. To handle this case, I padded the outside of the grid with default values, e.g. velocity was set to 0.

2.1.2 Forces

After advection, we apply forces. For now, this just means adding gravity to each element in v.

2.1.3 Pressure

Next, we want to update the pressure with values that enforce incompressibility. First, we solve the equation

$$
Ap = b, \tag{4}
$$

where A is the Laplacian matrix for the velocities and b is the divergence of the velocity at each point on the pressure grid. This is the main reason that we use a staggered grid, it makes it easy to perform central differences to get the divergence. We solve for p using conjugate gradient, I used the ViennaCL library for this. It is also possible to precondition the matrix for faster solves, but I did not. After solving for p, we update the velocities:

$$
\vec{u}^{n+1} = \vec{u} - \Delta t \frac{1}{\rho} \nabla p. \tag{5}
$$

2.2 Smoke

To simulate smoke, we need to add new values for advection, add two new kinds of forces, and render the smoke. For forces, we replace gravity with the following:

$$
f = f_{buoy} + f_{conf}, \tag{6}
$$

where f_{buoy} is buoyancy and f_{conf} is vorticity confinement.

2.2.1 Advection

When simulating smoke, we add values for soot concentration, s, and temperature, T, to the grid centres. We advect the soot and temperature using (2) with q as the value for soot or temperature, respectively, and f is the velocity as before. Note that the soot concentration must be between 0 and 1, which is why we are using a monotonic cubic interpolator; the monotonicity ensures that we do not overshoot the data. Since I used the concentration for rendering, I still found it necessary to clamp the concentrations because it is still possible to get values above 1 due to numerical error.

2.2.2 Buoyancy

The equation for buoyancy is

$$
[\alpha s - \beta(T - T_{amb})]\vec{g}, \tag{7}
$$

where α and β are positive constants, and T_{amb} is ambient temperature, $273^\circ K$. As suggested by [Bridson 2015], I used

$$
\begin{aligned}
\alpha &= \frac{\rho_{soot} - \rho_{air}}{\rho_{air}}, \\
\beta &= \frac{1}{T_{amb}},
\end{aligned}
$$

where I set $\rho_{air} = 1.3$ and $\rho_{soot} = \rho_{flames}$, which I will define later. The reason we choose these values of α and β is that we are making a Boussinesq approximation, where we assume that

$$
|\alpha s - \beta \Delta T| << 1.
$$

2.2.3 Vorticity Confinement

When advecting velocities at lower grid resolutions, the curl velocity is unnaturally dissipated. To correct this error, [Fedkiw et al. 2001] introduced vorticity confinement to reintroduce the curl. The force for vorticity confinement is

$$
f_{conf} = \epsilon \Delta x (\vec{N} \times \vec{\omega}), \tag{8}
$$

where ϵ is a scaling factor to vary how much curl is added in. Since this is not physically correct, it is difficult to choose a value that is correct for all situations. For example, a column of smoke may require higher vorticity to look real, while a smoke ring would not even retain shape with high vorticity. I demonstrate the visual difference between varying constants in the accompanying video. We scale by Δx since the error disappears as the resolution is refined, so we want vorticity confinement to disappear at high resolutions. To calculate $\vec{N} \times \vec{\omega}$, we use

$$\vec{\omega} = \nabla \times \vec{u}, \tag{9}$$

$$\vec{N} = \frac{\nabla \|\vec{\omega}\|}{\|\nabla \vec{\omega}\|}, \tag{10}$$

both of which are calculated using central differences. However, to avoid dividing by 0, it is better to use

$$\vec{N} = \frac{\nabla \|\vec{\omega}\|}{\|\nabla \vec{\omega}\| - 10^{-20} M}, \tag{11}$$

where

$$M = \frac{1}{\Delta x \Delta t}.$$

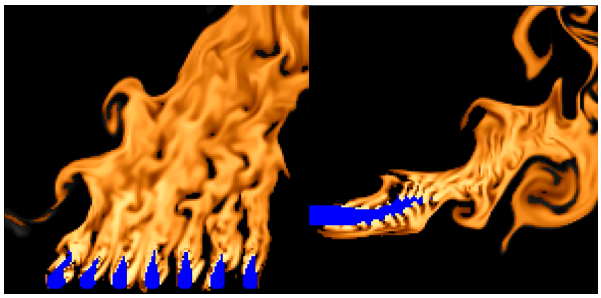

Figure 3: *2D fire animations*

2.3 Fire

When simulating fire, we simulate two fluids, fuel and flames, separated by a level set. This section covers general concepts needed to simulate fire and specific implementation details from my project.

2.3.1 Level Set

We define values for the level set, ϕ, at the centre of grid elements as the distance from the zero isocontour in the flames, and the negative distance in the fuel region. For evolving the level set, we use the following equation:

$$\frac{\partial \phi}{\partial t} + \vec{u}_{fuel} \cdot \nabla \phi = S, \tag{12}$$

where S is the burn rate of the fuel. I set $S = 0.08$ in 2D and $S = 0.2$ in 3D. So, to update the level set, we advect it using semi-Lagrangian advection, then add $\Delta t S$ to the advected value. It is also necessary to redistance the level set occasionally; I redistance the level set at every time step using the fast sweeping method [Zhao 2005].

2.3.2 Jump Conditions

In this section, we describe what happens near the zero isocontour or when we cross it during advection. Near the zero isocontour, we have

$$\begin{aligned} \vec{u}_{flames} &= \vec{u}_{fuel} + \Delta V \hat{n} \\ &= \vec{u}_{fuel} + \left(\frac{\rho_{fuel}}{\rho_{flames}} - 1 \right) S \hat{n}. \end{aligned} \tag{13}$$

Reducing ρ_{fuel} results in fuller flames. In my implementation, I set $\rho_{fuel} = 1$ and adjusted ρ_{fuel}, e.g. $\rho_{flames} = 0.01$ in 3D examples, to get different effects. When advecting from one fluid to the other, we add or subtract $\Delta V \hat{n}$ to get the correct velocity.

It is also necessary to correct the velocities when calculating the divergence for the values in b from (4). This is important because it causes the fuel to expand. For each grid element with $\phi > 0$ at the centre, we increment each adjacent grid element with $\phi \leq 0$ at the centre by $\Delta V \hat{n}$. Additionally, it possible to account for variable density in A from (4), but I did not.

2.3.3 Temperature

[Nguyen et al. 2002] discusses several methods for advecting temperature, but I chose the simplest method. First, inside the fuel region, we set the temperature to always be $T_{ignition}$, which I chose to be $1000^\circ K$. Then, when we are advecting a value in the flame region and we need to get a temperature from the fuel region, we set the temperature to T_{max}, which I chose to be $2000^\circ K$.

When advecting within the flame region, it is also necessary for the temperature to decay over time. For the decay, we use the equation

$$\frac{DT}{Dt} = -c \left(\frac{T - T_{amb}}{T_{max} - T_{ambient}} \right)^4. \tag{14}$$

[Bridson 2015] converts (14) to the following update:

$$T^{n+1} = T_{amb} + \left[\frac{1}{(\tilde{T} - T_{amb})^3} + \frac{3c\Delta t}{(T_{max} - T_{amb})^4} \right]^{-\frac{1}{3}}, \tag{15}$$

where $\tilde{T}$ is the advected temperature and c is a cooling constant. When I was trying to find a good value for c, it was hard to find an actual "constant" that worked, so I set $c = 10^{-5} T_{max}^2$.

2.3.4 Advecting Soot

When animating fire, we advect soot along with the flames so that the fire can produce smoke as it cools. However, if we add soot to the fuel region then it causes an upward draft in the fuel due to buoyancy which I did not like. To resolve the issue, I set the fuel to zero in the fuel region at each time step, and when advecting from the flames into the fuel region, I set the soot in the flames to 0.5.

2.3.5 Adding Fuel

For adding fuel, I added a boolean at the grid centres to indicate whether the grid element was a source of fuel. At each time step, I would manually set the velocity at sources. I took two different approaches to update the level set. In Figure 3 (right), I set each of the level set values at the sources to -1. In Figure 1 and Figure 3 (left), I used a function to add spheres, or circles in 2D, of fuel given a centre and a radius. When setting the source values, the grid element was a source if the centre of the grid element was in the sphere. To update the level set, I iterated over the grid and for each point in the sphere, I set the level set value to the negative distance to the surface of the sphere. When using either level surface update technique, I redistanced the level set afterwards.

2.3.6 Detonation Shock Dynamics

In [Hong et al. 2007], the authors introduced the use of detonation shock dynamics (DSD) equations to evolve the burn rate, S, of the level set. To accomplish this, we replace S in (12) with the result of evaluating third order DSD equations. I will not go into depth how this is done specifically since it is not particularly enlightening. The most important detail to note is that the burn rate will be dependent on the mean curvature of the surface. After calculating the mean curvature, it is necessary to extrapolate the values for the mean curvature in one direction away from the zero isocontour; the authors did not specify which direction, so I extrapolated outwards into the flames. Additionally, there are many tunable parameters included in the model which affect the overall appearance of the flames. [Hong et al. 2007] also includes an additional jump condition for the temperature that they did not explain well enough for me to include.

2.4 Rendering

2.4.1 Smoke Colour

In 2D, I rendered smoke when the flames dropped below $T_{ignition}$. The colour of the smoke was the concentration multiplied by each of the colour channels of `0xFFFFFF` to get varying shades of grey. In 3D, I did not render smoke because it would have been too much work to account for translucency and I wanted to be able to see the fire completely.

2.4.2 Fire Colour

In the animations, I made the fuel blue, but in physically-based rendering the fuel would be invisible. To render the flames, I used Planck's formula for blackbody radiation,

$$L(T, \lambda) = \frac{2C_1}{\lambda^5(e^{C_2/(\lambda T)} - 1)}, \tag{16}$$

where T is the temperature, λ is the wavelength, $C_1 \approx 3.7418 \cdot 10^{-16} Wm^2$, and $C_2 \approx 1.4388 \cdot 10^{-2} m^\circ K$ [Nguyen et al. 2002]. We sample wavelengths between $400nm$ and $700nm$ at $10nm$ intervals. Then we integrate the samples to CIE XYZ and convert from CIE XYZ to RGB. I used functions from PBRT [Pharr and Humphreys 2004] to perform these calculations. After converting to RGB, I performed a gamma correction with $\gamma = 2.2$. Since I did not want to scale the RGB values and CIE XYZ contains colours outside the RGB colour space, I clamped each of the RGB values. I also passed twice the temperature to Planck's formula to get correct variation in the flames. I did this instead of adjusting the temperature because I did not want the flames to rise faster due to buoyancy. As mentioned in the previous section, when the temperature of the flames drop below $T_{ignition}$, I render smoke instead.

2.4.3 Ray Tracing

In 2D, I performed rendering by dividing the pixels into a grid and setting the value of each pixel to the corresponding colour. In 3D, I implemented a basic ray tracer and placed the grid in the scene. When a ray of light hit the grid, I traversed it in a similar way one would traverse a uniform grid. If the ray intersected an element that contained either fuel or flames, I would return the corresponding colour.

3 Discussion

All of the figures and clips in the accompanying video are 600×600 pixels. The grids were 150×150 in 2D and $150 \times 150 \times 150$ in 3D, not including padding, with the exception of Figure 1 (middle), which had a grid of $100 \times 100 \times 100$. The 2D clips were rendered at varying framerates and steps per frame, however the 3D animations were all rendered at 30 FPS with one step per frame. The 3D animations were rendered overnight (8-12 hours) on a desktop with two Intel Xeon hyperthreaded 12-core CPUs at 2.7GHz, i.e. a total of 24 cores.

I implemented DSD from [Hong et al. 2007] when my fire simulation was still 2D, but it was not working correctly. I realized that mean curvature did not mean the same thing in 2D as 3D, so I extended my simulation to 3D. When I ran the fire simulator in 3D with DSD, the level set became unstable and exploded, and there were no visible cellular patterns in the fire. Tweaking the constants stopped in from blowing up as fast, but it still blew up. As you can see, it takes a long time to render at higher resolutions in 3D, and I could not test DSD at lower resolutions for a couple reasons: the constants have different effects at different resolutions and I do not think that the cellular patterns would even appear at lower resolutions. Since I could not quickly iterate through different parameters, I could not tell if there was a bug in my implementation or if I was just choosing poor parameters. If there is a problem with the implementation, and it is not a bug, then I have a couple suspicions about the cause. I could have extrapolated the mean curvature in the wrong direction, or maybe I should have accounted for variable density when building A from (4). If I had more time, I would try to set up my simulation so that I could use the constants directly from [Hong et al. 2007]. Then if it still did not work, I would try experimenting with the things I might be missing. I see no reason why there would be a bug, since the implementation details are pretty simple, but I would also probably double check that as well.

References

BRIDSON, R. 2015. *Fluid simulation for computer graphics*. CRC Press.

FEDKIW, R., STAM, J., AND JENSEN, H. W. 2001. Visual simulation of smoke. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ACM, 15–22.

HONG, J.-M., SHINAR, T., AND FEDKIW, R. 2007. Wrinkled flames and cellular patterns. In *ACM Transactions on Graphics (TOG)*, vol. 26, ACM, 47.

NGUYEN, D. Q., FEDKIW, R., AND JENSEN, H. W. 2002. Physically based modeling and animation of fire. *ACM Transactions on Graphics (TOG) 21*, 3, 721–728.

PHARR, M., AND HUMPHREYS, G. 2004. *Physically based rendering: From theory to implementation*. Morgan Kaufmann.

ZHAO, H. 2005. A fast sweeping method for eikonal equations. *Mathematics of computation 74*, 250, 603–627.