Fluid Simulation for Video Games (part 11)

By Dr. Michael J. Gourlay

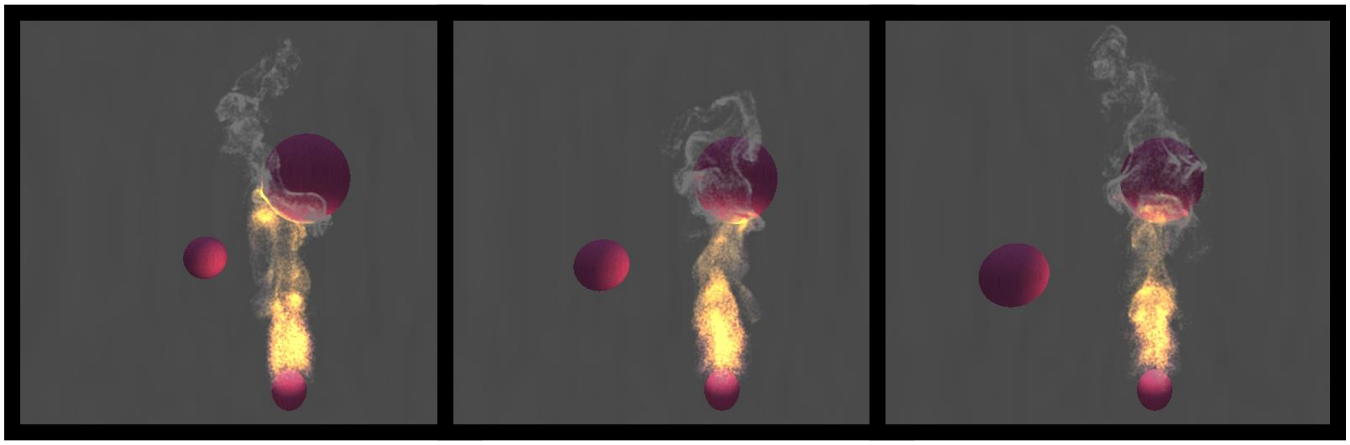

Figure 1. Combustion. A fuel-laden fluid emits from the lower ball, ignites, combusts, convects upward and heats the upper ball. The camera rotates around the scene.

Combustion: Fuel + oxygen + heat → fire.

Combustion is a chemical reaction where fuel oxidizes and releases heat. Although that might sound simple, the process is very complicated. Visual effects artists, however, neither want nor need all of that complexity. For visual effects, video games can simulate combustion as a fluid with a few components: fuel, flame, smoke, and, of course, heat.

Visual effects traditionally model flames and smoke as different layers in a particle effect. This article explains how to model them as a single continuous process. Doing so makes the simulation more emergent and interactive. For example, a user could squirt fuel onto a hot object which would cause the fuel to burn and smoke to rise. This could, in turn, heat a balloon above it, causing it to expand and float away.

This article describes how to modify the vortex fluid simulation presented in earlier articles, to include a fast and simple but effective model of combustion. Part 1 summarized fluid dynamics; part 2 surveyed fluid simulation techniques, and part 3 and part 4 presented a vortex–particle fluid simulation with two-way fluid–body interactions that runs in real time. Part 5 profiled and optimized that simulation code. Part 6 described a differential method for computing velocity from vorticity, and part 7 showed how to integrate a fluid simulation into a typical particle system. Part 9 described how to approximate buoyant and gravitational forces on a body immersed in a fluid with varying density. Part 10 described how density varies with temperature, how heat transfers throughout a fluid, and how heat transfers between bodies and fluid.

Combustion Theory

To understand combustion you need to understand how chemicals *move*, how they *interact* with each other and how they *use* and *produce heat*.

Whereas part 10 introduced the notion that a fluid can have variable density, this article introduces the notion that a fluid can have multiple constituents, also called chemical **species**. For example if a fluid contains methane (CH_4) and oxygen (O_2), it contains two species.

The following chemical equation models how a hydrocarbon fuel (C_xH_y) can combine with oxygen (O_2) to produce exhaust (carbon dioxide, CO_2, and water, H_2O):

$$C_xH_y \;+\; (x + {}^{y}/_{4})\, O_2 \;\rightarrow\; x\, CO_2 \;+\; {}^{y}/_{2}\; H_2O$$

Although the above equation indicates how much fuel makes how much exhaust, it does not tell us how fast the reaction occurs, nor how much energy is required to trigger the reaction, nor how much energy it releases. We need to know each of these things in order to model combustion. The sciences of **thermochemistry** and **chemical kinetics** endeavor to answer such questions.

Furthermore, the equation above assumes the reaction has just enough fuel to mix with available oxygen, and that the reaction will complete perfectly. (Such an equation is called **stoichiometric**.) In reality, the reaction might not complete and the resulting products might contain other compounds such as soot. Those "dirty" compounds constitute most of the visible parts of smoke.

The reaction-diffusion equation models how the concentration of a chemical species k can change under the influence of the processes of diffusion and reaction:

$$\underbrace{\frac{\partial \rho Y_k}{\partial t}}_{\text{Change in mass}} \;+\; \underbrace{\vec{u}\cdot\vec{\nabla}(\rho Y_k)}_{\text{Advection}} \;-\; \underbrace{\mu\nabla^2(\rho Y_k)}_{\text{Diffusion}} \;=\; \underbrace{R_k}_{\text{Reaction}}$$

where $Y_k = {}^{m_k}/_{m}$ represents the fraction of mass of species k in a fluid parcel with mass m, and R_k represents the rate at which a reaction produces species k.

Previous articles already explained how to model advection and diffusion. This article introduces a model for how chemical reactions can create and destroy species.

The **Arrhenius equation** models the rate R_k to produce species k from a reaction between species i and j:

$$R_k = A Y_i Y_j e^{-T_a/T}$$

Think of A as a **frequency factor** that relates to how often molecules of species i and j collide with each other (their **collision frequency**). The **activation temperature**, T_a, determines the number of molecules that have enough energy (the **activation energy**) to form the bonds to create species k, as Figure 2 depicts. The temperature, T, is measured in an absolute scale (such as *Kelvin*) where zero is the coldest temperature matter can become. The term $e^{-T_a/T}$ comes from a probability distribution. It basically tells you the number of molecules that have a certain energy, given that they collectively have a given temperature.

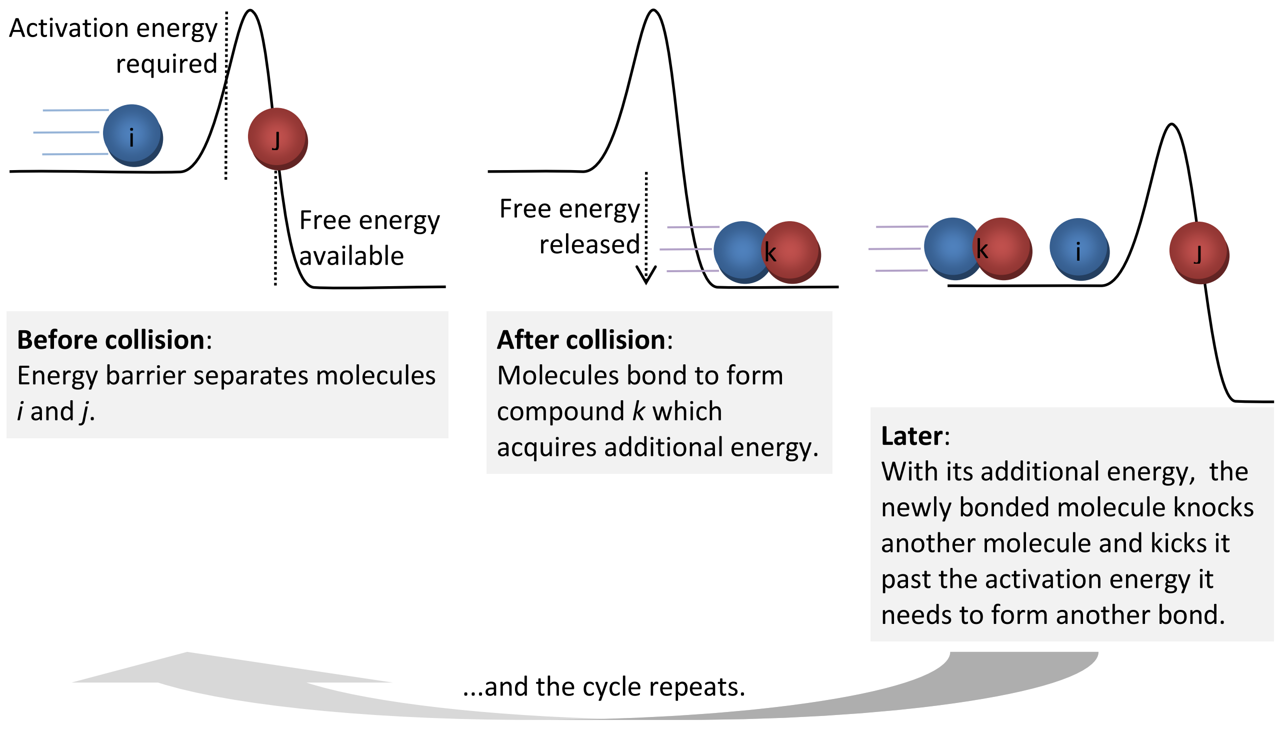
Activation energy required
i
J
Free energy available
Free energy released
k
k
i
J
Before collision:
Energy barrier separates molecules *i* and *j*.
After collision:
Molecules bond to form compound *k* which acquires additional energy.
Later:
With its additional energy, the newly bonded molecule knocks another molecule and kicks it past the activation energy it needs to form another bond.
...and the cycle repeats.

Figure 2. Activation Energy of Chemical Reactions.

Fuel combines with oxygen and releases heat. The amount of heat released is called the **enthalpy of formation**. Model that using this equation, where G_k is the amount of heat released per unit mass when forming species k:

$$H = \rho \Delta Y_k G_k$$

Simplifying assumptions

Combustion is an extremely complicated process, and video game hardware lacks the computational resources to simulate all of it in real time. Visual effects authors also do not always wish for perfectly realistic combustion; they also want magical effects, so they want more control than mere reality affords. This article therefore presents a toy model which simplifies the process.

In this simplified model, fuel, oxygen, flame and smoke all advect identically, and within the same spatial region, they have the same temperature. The fluid has an unlimited supply of oxygen, and it is everywhere; the reaction is never limited by a lack of oxygen. In this limited sense, the flame model could be said to be "**premixed**". (In contrast, in a **diffusion flame**, fuel and oxygen meet upon combustion.) Furthermore, this model has all components share fluid and thermodynamic properties such as viscosity, diffusivity and thermal conductivity. This model behaves as though those properties (such as viscosity) are independent of temperature, pressure and composition. As before, this model assumes that density differences are small except in buoyancy terms.

None of these assumptions are valid, which makes this model useless for engineering or scientific purposes.

This model also omits any notion of **lean** versus **rich** combustion, and provides no notion of **intermediate species**. Real fuel can have hundreds of components, real combustion can have hundreds of intermediate reactions and real exhaust can have hundreds products. This model only includes fuel, flame and exhaust as components, and treats each of them as a single species.

Technically, "flame" is not a chemical compound; in this model, it represents that smoke which is hot enough to glow. In reality, smoke is basically the same material as the orange part of the flame -- just cooler. The light in the blue part of flame comes from spectral lines of intermediate reactions. (Figure 3 shows a schematic of a flame.) This model does not explicitly model that portion, but the end of the article presents some ideas for possible ways to model the blue core without adding much complexity to the model.

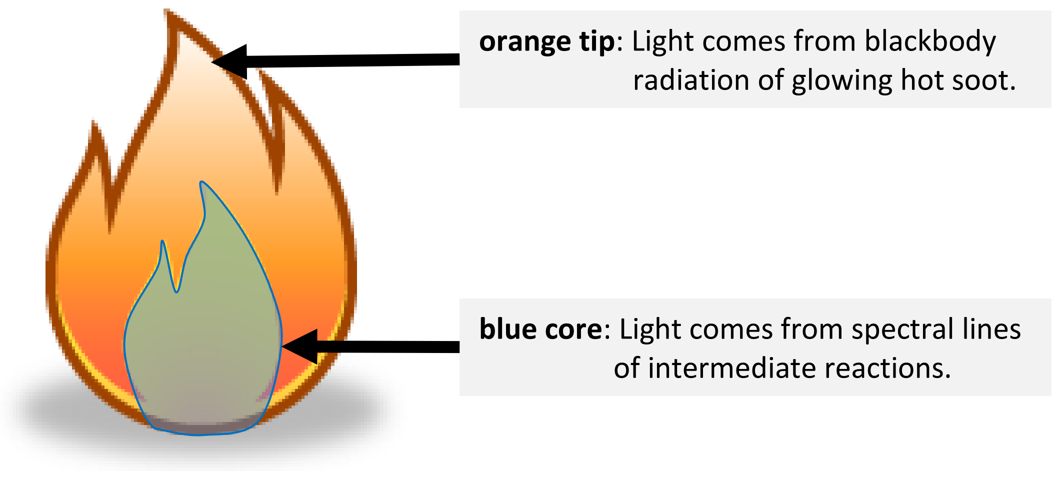
orange tip: Light comes from blackbody radiation of glowing hot soot.
blue core: Light comes from spectral lines of intermediate reactions.

Figure 3: Components of a flame.

In short, the model used here deviates far from reality and neglects numerous important effects. It retains only the most rudimentary aspects of a combustion model. Yet it looks nice and runs fast.

Note: The notion that the orange part of the flame comes strictly from blackbody radiation is popular in textbooks and scientific literature but is not strictly true. The orange part is not hot enough to match the spectrum that comes from it, if it radiated as a blackbody. But the important part of this story is that the blue and orange parts glow due to different processes.

In this model, heat only propagates by convection and diffusion. But heat should also propagate by **radiation**, which depends on temperature to the 4th power. Modeling this would imply tracing rays which travel at the speed of light -- effectively instantaneously. The absorption would depend on the capture cross section of each fluid parcel. It's immensely expensive to calculate. So this simplified model omits radiative effects (though the end of the article presents some ideas for how you might fake it).

Combustion has two regimes, called **deflagration** and **detonation**. In deflagration, heat propagates slower than the speed of sound by conducting to adjacent material. In detonation, a heat propagates faster than the speed of sound via shockwaves compressing adjacent material. Detonation is a more sudden and loud process than deflagration. This model intentionally lacks pressure waves -- since including them requires simulation time steps to be too tiny for real-time -- so this simulation does not model detonation.

Putting theory to use

While the details might interest some, visual effects authors can ignore most of details of combustion theory; they only need to deal with certain parameters summarized in Table 1: Y which determines how much fuel the fluid contains, A and T_a which control the rate of converting fuel to flame, and C_k which controls how much heat the combustion releases. The model also includes parameters that control how fast to convert flame to smoke.

Table 1: Combustion tuning parameters.

Parameter	Meaning	How to use it
$\boldsymbol{Y_{fuel}}$	Fraction of fluid which is fuel	Increase if you want more flames.
$\boldsymbol{A_{fuel}}$	Reaction rate factor for burning fuel	Increase if you want more intense flames.
$\boldsymbol{T_{a,fuel}}$	Activation temperature for burning fuel	Decrease if you want flames to occur at a lower temperature.
$\boldsymbol{G_{fuel}}$	Heat released by burning fuel	Increase if you want fluid to move (buoy) more due to burning.

Changes to the simulation code

To put this theory to use, the fluid simulation code presented in previous articles needs some data and code additions to particles and to the simulation.

Particle additions

To the Particle class, add a member variable for the mass fraction for each material component: fuel, flame and smoke. (Oxygen could also be a component, but this article omits it.) Each value must be between 0 and 1 (inclusive), and in this model their sum must remain below 1.

```
    // Each particle has 3 fractions: fuel, flame and exhaust (smoke).
    float mFuelFraction  ; ///< Fraction of mass which is fuel.
    float mFlameFraction ; ///< Fraction of mass which is flame.
    float mSmokeFraction ; ///< Fraction of mass which is smoke.
```

Figure 4 shows various components of flame rendered separately: fuel, flame and smoke.

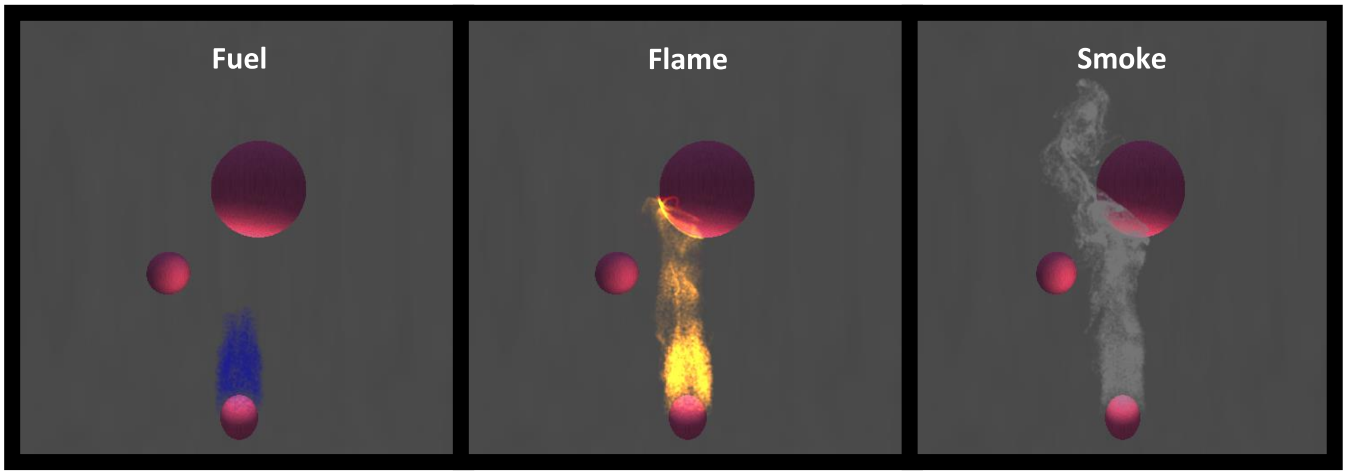

Figure 4: Components of Simulated Fluid Combustion.

Vorton simulation additions

To the VortonSim class, add the following members:

- mCombustionTemperature: Activation temperature (T_a) for reacting fuel into flame.
- mCombustionRateFactor: Pre-exponential factor (A) for combustion rate.
- mSpecificFreeEnergy: Amount of energy, per unit mass of fuel, released during combustion.

As mentioned above, the difference between flame and smoke in this simulation is basically a question of temperature: orange flame is glowing hot smoke. But (as explained below) the renderer treats flame and smoke as completely different kinds of material. So the simulation also treats them as separate kinds of matter.

This simulation models smoke as though it emerges from flame via a reaction. So, in this simulation, smoke also has a threshold temperature and a pre-exponential factor:

- mSmokeTemperature: Temperature below which flame begins to turn into smoke.
- mSmokeRateFactor: Pre-exponential factor for converting flame to smoke.

Remember that this combustion model lacks radiative effects. One artifact of this is that the simulated flame does not cool as fast as it should. This would otherwise lead to fluid remaining in a glowing-hot flame state for too long but because it models flame-to-smoke as a separate process, VFX authors can control how long flame persists independently of its radiative properties. This ability compensates for the lack of a radiative cooling model and lets VFX authors control the look of the flames and smoke with finer detail than reality would allow.

Transferring mass fractions from vortons to tracers

The whole point of this simulation is to visualize it. Remember from part 8 that the renderer depicted heavy fluid and light fluid differently. The renderer accomplished that by using tracer density to control how to render tracer particles. But the simulation evolves vorton (not tracer) density. To give tracers density that matches the simulation, the VortonSim transfers density from vortons to tracers via a grid. Now that "density" effectively has multiple components (fuel, flame and smoke), the simulation must now also transfer the mass fractions from vortons to tracers using the same approach. The highlighted code below is new:

```
    // Populate density and mass fraction grids.
    const size_t numParticles = particles.Size() ;
    for( size_t uParticle = 0 ; uParticle < numParticles ; ++ uParticle )
    {   // For each particle in the array...
        const Particle  &   rParticle   = particles[ uParticle ] ;
        const Vec3      &   rPosition   = rParticle.mPosition   ;
        const float volumeCorrectedDensity= rParticle.GetMassDeviation()* oneOverCellVolume ;
        densityGrid.Accumulate( rPosition , volumeCorrectedDensity ) ;
        mFuelFractionGrid.Accumulate ( rPosition , rParticle.mFuelFraction  ) ;
        mFlameFractionGrid.Accumulate( rPosition , rParticle.mFlameFraction ) ;
        mSmokeFractionGrid.Accumulate( rPosition , rParticle.mSmokeFraction ) ;
    }
```

Note: It is likely that the process of populating the grid from vortons is bound by memory bandwidth, in which case Accumulate would probably run faster if it copied all fields in a single pass, instead of copying a single field in each of multiple passes. That is because the source values all reside near each other in memory, so loading one likely loads others into the same cache line -- whether it is used or not (and it isn't). The code above could probably run 2 to 4 time faster by consolidating it into a single procedure. The drawback would be that Accumulate would then be very special-purpose and brittle instead of being generic and flexible as it is now. That's probably a valid trade-off to make in production code.

Combustion reactions

The main attraction for this article is, of course, the actual combustion of fuel into flame and smoke. The code to do this simply integrates the rate given by the Arrhenius equation:

```
    for( size_t iPcl = iPclStart ; iPcl < iPclEnd ; ++ iPcl )
    {   // For each particle in this slice...
        Vorton &    rVorton            = (*mVortons)[ iPcl ] ;
        const float vortonTemperature = rVorton.GetTemperature( mAmbientDensity ) ;
        if( rVorton.mFlameFraction > 0.0f )
        {   // This particle is on fire.
            const float arg1 = vortonTemperature / mSmokeTemperature ;
            const float arg2 = arg1 * arg1 ;
            const float temperatureDependence = exp( - arg2 ) ;
            const float smokeRate             =
                         mSmokeRateFactor * temperatureDependence * rVorton.mFlameFraction ;
            const float flameToSmokeChange      = smokeRate * timeStep ;
            // Decrease amount of flame.  (This implicitly increases amount of smoke.)
            rVorton.mFlameFraction -= flameToSmokeChange ;
        }
        if( rVorton.mFuelFraction > 0.0f )
        {   // This particle has some fuel.
            const float temperatureDependence =
                            exp( - mCombustionTemperature / vortonTemperature ) ;
            const float combustionRate   =
                     mCombustionRateFactor * temperatureDependence * rVorton.mFuelFraction ;
            const float fuelToFlameChange = combustionRate * timeStep ;
            // Decrease amount of fuel and increase amount of flame.
            rVorton.mFuelFraction  -= fuelToFlameChange ;
            rVorton.mFlameFraction += fuelToFlameChange ;
            // Change temperature due to heat released by combusting fuel:
            const float temperatureChange =
                            mSpecificFreeEnergy * fuelToFlameChange *
                            rVorton.GetDensity( mAmbientDensity ) * mSpecificHeatCapacity ;
            rVorton.SetTemperature( mAmbientDensity, vortonTemperature + temperatureChange );
        }
        rVorton.mSmokeFraction = 1.0f - ( rVorton.mFuelFraction + rVorton.mFlameFraction ) ;
    }
```

Those are the only changes to the simulation code: a few variables and a couple dozen lines of code. Now we want to see flames burn!

Rendering additions

This article series focuses on simulation dynamics, not on rendering. Fluid rendering is a complicated topic that warrants its own series of articles. Instead of delving into the details of how this particular demo renders flames, I will mention only a few aspects that are probably common to many fluid particle rendering techniques you would likely see in any video game.

Note: I will mention in passing, however, that the demo code accompanying this article uses pedestrian, nearly obsolete fixed-function rendering and no programmable shaders. Compared to the state-of-the-art, this rendering is laughable. Yet another exercise for the reader.

Even though the simulation treats fuel, flame and smoke in a unified way, the renderer uses different materials for each. If the only difference were texture, we could render all of those in a single pass and select texture based on composition. But alas there is a more fundamental difference between flame (which illuminates itself) and fuel or smoke (which do not) and this manifests as a change in render state, which therefore requires multiple draw calls. Also, we will often want to render both flame and smoke at the same location.

Particle Rendering Techniques

Although the demo has one particle system, it gets rendered three times, one for each component: fuel, flame and smoke (which Figure 2 shows). Fuel and smoke are rendered using the same *opacity-blended* technique as in previous articles. Fuel uses a blue and smoke uses a gray texture. The flames, however, use *additive* blending so they give the impression of self-illumination. Flames use an orange texture.

All particles are modulated according to their mass fraction. Opacity-blended particles (fuel and smoke) have their opacity modulated according to their mass fraction. Additive-blended particles (flame) have their color modulated according to their mass fraction. Effects authors can control the modulation coefficients to control brightness and opacity of each layer.

Lighting Trick

The demo sets up an orange point light somewhere within the flames and fluctuates its intensity to give a rough suggestion that light emanates from the flame. In the demo, you can see light flicker on the balls.

Results

Figure 5 shows a variety of burning effects you can achieve by tuning various combustion parameters.

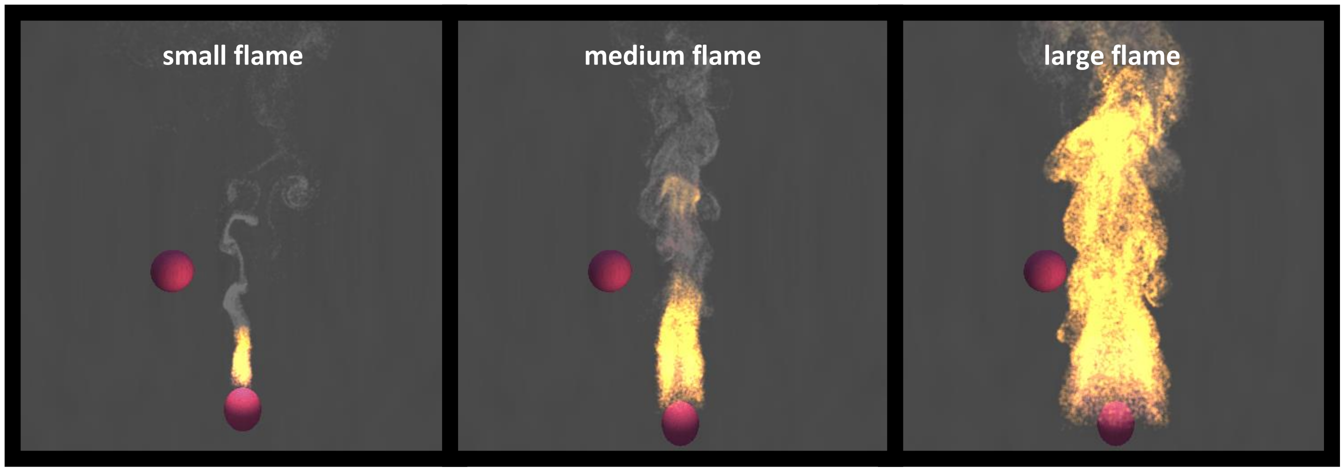

Figure 5: Various kinds of flames achieved by changing parameters.

Figure 6 shows a ball emitting fuel-laden fluid which, because its temperature is above the activation temperature, ignites and combusts. The heat it generates causes the fluid to convect upward. It makes contact with another ball and heats it. The camera rotates around the scene.

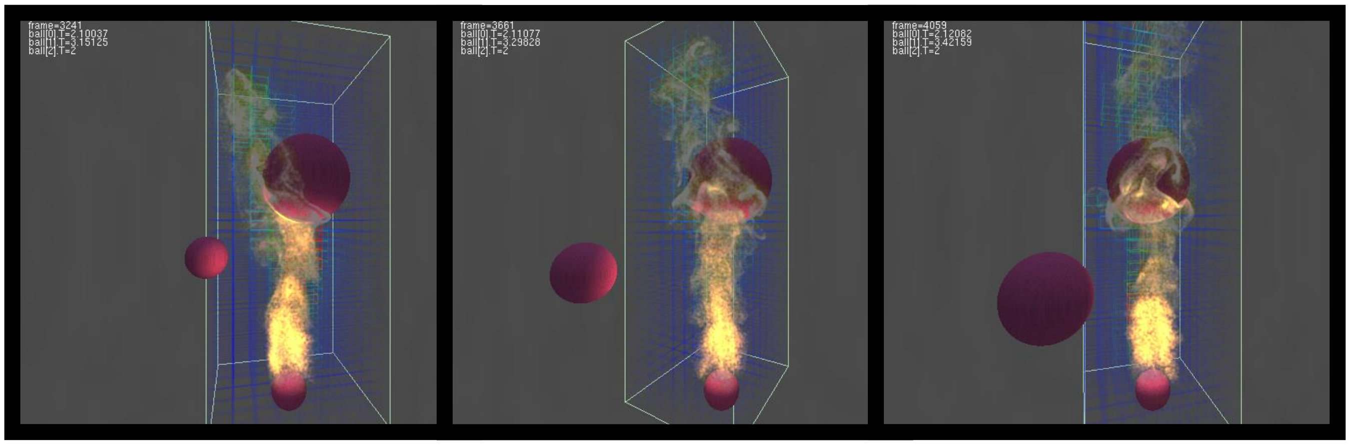

Figure 6: Combustion with diagnostic information.

In the demo, the upper ball is initially neutrally buoyant and free to move. Upon heating, it becomes positively buoyant and floats away.

Performance

As with previous articles, the demo spends most of its CPU time on rendering: 47% of the total time is spent filling vertex buffers, setting materials and drawing. In addition, another 16% of the time is spent transferring density and mass fraction values from vortons, via a grid, to tracers -- a process needed only to render the tracers with different materials.

The large portion of time spend on rendering matters especially in light of the fact that this demo does not use modern programmable shaders nor vertex stream frequency division or instancing, which would dramatically speed up particle rendering -- perhaps by a factor of 4 to 8.

Parallelization

Transferring density and mass fraction values was previously a serial process that consumed 38% of the CPU time. It is an embarrassingly parallel process so use Intel Threading Building Blocks to parallelize it.

Write a routine to perform assignment for a subset of its destination values:

```
static void AssignScalarFromGridSlice( Vector< Particle > & particles
  , size_t memberOffsetInBytes , const UniformGrid< float > & scalarGrid , size_t iPclStart
  , size_t iPclEnd )
{
    for( size_t iPcl = iPclStart ; iPcl < iPclEnd ; ++ iPcl )
    {   // For each particle in this slice...
        Particle &  rParticle = particles[ iPcl ] ;
        float value ;
        scalarGrid.Interpolate( value , rParticle.mPosition ) ;
        *(float*)(((char*)&rParticle) + memberOffsetInBytes) = value ;
    }
}
```

Write a functor to wrap that call and allow TBB to use it:

```
class Particles_AssignScalarFromGrid_TBB
{
        Vector< Particle >  &           mParticles              ;
        size_t                          mMemberOffsetInBytes    ;
        const UniformGrid< float > &    mScalarGrid             ;
    public:
        void operator() ( const tbb::blocked_range<size_t> & r ) const
        {   // Assign scalar from grid to a subset of particles.
            AssignScalarFromGridSlice( mParticles , mMemberOffsetInBytes , mScalarGrid
                                     , r.begin() , r.end() ) ;
        }
        Particles_AssignScalarFromGrid_TBB( Vector< Particle > & particles
                    , size_t memberOffsetInBytes , const UniformGrid< float > & scalarGrid )
            : mParticles( particles )
            , mMemberOffsetInBytes( memberOffsetInBytes )
            , mScalarGrid( scalarGrid )
        {}
} ;
```

Use TBB's parallel_for to invoke the functor:

```
void Particles::AssignScalarFromGrid( Vector< Particle > & particles
                    , size_t memberOffsetInBytes , const UniformGrid< float > & scalarGrid )
{
    if( scalarGrid.HasZeroExtent() )
    {   // Scalar grid is empty.  Probably first iteration.
        return ; // Nothing to do.
    }
    const size_t    numParticles = particles.Size() ;
        // Estimate grain size based on size of problem and number of processors.
        const size_t grainSize =  MAX2( 1 , numParticles / gNumberOfProcessors ) ;
        // Assign scalar from grid using threading building blocks.
        parallel_for( tbb::blocked_range<size_t>( 0 , numParticles , grainSize )
    , Particles_AssignScalarFromGrid_TBB( particles , memberOffsetInBytes , scalarGrid ) ) ;
}
```

Table 2 shows how runtimes scale with the number of threads on a 4-core CPU. Note that while AssignScalarFromGrid is parallelized with Threading Building Blocks, PartitionVortons is not, so its runtime should not significantly depend on the number of threads.

Table 2. Run Durations for Processes on an Intel® 3.4 GHz Core I7-2600® processor

Threads	AssignScalarFromGrid (ms)	PartitionVortons (ms)	Total (ms)
1	3.4	0.070	19.5
2	1.9	0.076	10.1
4	1.3	0.078	7.41
8*	0.90	0.078	5.61

*Note that the CPU on which this test ran has only 4 cores, but they are hyperthreaded.

Potential Improvements

As with all of the articles in this series, the accompanying code constitutes a mere prototype of a full particle system. These articles attempt to focus on the clarity of concepts. Production code is more robust, data-driven and, alas, more complicated. But that complication is necessary to make the code useful. So to manifest the ideas in these articles as useful code would require much more effort -- an exercise for the reader.

In the demo code that accompanies these articles, the simulation and visualization are too coupled. In a scientific setting that consistency might be an advantage, but for visual effects, authors want to tune motion and appearance independently. Fortunately, that coupling is merely an artifact of how this demo code is written and not an intrinsic problem with using vortex particles. Using programmable shaders would reduce problem. It's a straightforward problem to solve, and absolutely crucial for making a useful tool, but it goes beyond the scope of these articles.

This model could be enhanced to (very roughly) approximate radiation by introducing a "radiative loss" term which would depend on T^4. You could also approximate radiation incorporating another diffusion term which would depend on temperature, or simply by changing the thermal diffusivity to depend on temperature: Diffusivity could increase with temperature to the 4th power.

You could also explicitly model the presence and consumption of oxygen, for example to model a flame extinguishing itself in a closed environment. A future article could describe how to handle enclosed environments. In principle the inclusion of oxygen would entail adding another mass fraction parameter and consuming oxygen at a rate proportional to the fuel-to-flame reaction rate, and also making the fuel-to-flame reaction rate depend on the fraction of oxygen.

You could try this cheap trick to visualize a blue core: Tag particles that just turned from fuel to flame for the first time. For that one frame, render those particles as luminous blue. For subsequent frames, render them as a regular (orange) flame. Another idea: Use a color ramp where particles near the activation temperature are blue, and slightly cooler (but still glowing hot) flames are orange. Again, using a programmable shader would make this kind of experimentation easy.

You could compute per-particle shading to simulate self-illumination and self-shadowing. Images in the third article demonstrated that approach for the mushroom cloud rendering. Now that the simulation includes actual combustion, the visual effect of internal glowing could look pretty nice.

Future Articles

The next article will revisit the vorticity-from-velocity algorithm. Liquids take the shape of their containers on all but one surface, so modeling liquids also implies modeling containers. Future articles will include extending boundary conditions to include planes, convex hulls and interiors, which will allow for creating containers. That will pave the way for a discussion of free surface tracking and surface tension -- properties of liquids.

Further Study

Richard Feynman (1983): Fun To Imagine: Fire. *BBC TV series*. (http://www.youtube.com/watch?v=ITpDrdtGAmo)

Jos Stam & Eugene Fiume (1995): Depicting fire and other gaseous phenomena using diffusion processes. *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pp. 129-136.

Sten Odenwald (1995): The Astronomy Cafe. http://www.astronomycafe.net/qadir/q490.html.

Lawrence Anderson-Huang: The Physics of Art and Visual Perception. http://astro1.panet.utoledo.edu/~lsa/_color/06a_flames.htm

Nguyen, *et al*.: Physically based modeling and animation of fire.

Thierry Poinsot & Denis Veynante (2005): **Theoretical and Numerical Combustion**. Philadelphia: R.T. Edwards, Inc. (also http://elearning.cerfacs.fr/combustion/index.php).

Thierry Poinsot (2011): **Introduction to Combustion**. Video series on YouTube, University of Toulouse (http://www.youtube.com/watch?v=JK-K-QTSOqY).