Simulating and Rendering Explosions on the GPU

Julian Fuchs

MIT 6.837 (Computer Graphics), Fall 2016

1 Motivation

It is apparently customary in the explosion simulation literature to begin by pointing out that real explosions are rarely encountered by most people. They are, however, ubiquitous in films and video games, and pose interesting simulation and rendering problems. Explosions combine fire and smoke, which are closely related but behave and appear very differently. They also combine small- and large-scale physical phenomena: the explosion itself happens very quickly in a relatively small region, while the resulting fireball and plume of smoke move much more slowly through a much larger space.

The reasons for the GPU-centric approach are twofold. First, modern GPUs can be literally thousands of times faster than CPUs for highly parallelizable workloads. Both the simulation and rendering steps are computationally intensive and "embarrassingly parallel," so *not* implementing them on the GPU seems like it would be a wasted opportunity. Second and more personally, having worked at NVIDIA last summer on a project very far from the graphics layer, I was curious about working with GPUs from the user perspective.

2 Previous work

Smoke and fire are both frequently written about in the computer graphics world, and fluid simulation in general is a very heavily-studied field. And the computer-aided modeling of explosions dates to the very origin of computers. Despite this, there are relatively few papers published specifically about explosions from a computer graphics perspective.

I read every paper I could find about explosions in graphics, and developed my implementation by combining ideas from several of them. I also drew from a few more general papers about fluids, smoke, and flames. Papers will be referenced where their ideas are used.

3 Simulation

3.1 Fluid model

This project takes the Eulerian approach to fluid simulation, storing the system's state on a regular three-dimensional grid. Each cell, or voxel, in the grid contains values for the various fluid properties (velocity, temperature, etc.). Since the fluid in question is a gas, there are no free surfaces, only fluid-solid boundaries. Solid objects are voxelized and represented directly on the grid.

To simplify simulation, air is modeled as an inviscid and incompressible fluid. While air does indeed have very low viscosity, it is not actually incompressible, and its compressibility gives rise to effects like shockwaves that play an important part in the physics of real explosions. However, shockwaves are nearly invisible and for this project I chose to focus on more easily-visualized secondary effects like fire and smoke.

The behavior of an inviscid, incompressible fluid is governed by the Navier-Stokes equations:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} + \nabla p = \mathbf{F}$$

$$\nabla \mathbf{u} = 0$$

where $\mathbf{u}$ is the velocity field, p is the pressure field, and $\mathbf{F}$ represents all external forces such as gravity. The second equation states that the velocity field is divergence-free, which ensures incompressibility and mass conservation; both of those constraints will need to be loosened to simulate explosions.

To solve these equations, I used the advection-projection approach described by Stam in [7, 8]. Some useful implementation tips also came from [2], particularly regarding discretization. The first core idea of this method is the semi-Lagrangian advection scheme, which updates a cell's values by tracing $\mathbf{u}$ backwards to a "start point" and then advecting forwards the interpolated values at that point. This is how temperature, smoke, fuel, and the velocity field itself are transported by the fluid's motion. The second idea is projection, which maintains the divergence-free property of $\mathbf{u}$; it relies on the fact that $\mathbf{u}$ can always be expressed as the sum of a divergence-free field and a gradient field (the Hodge decomposition). The advected velocity field $\mathbf{w}$ produced by the semi-Lagrangian method above will likely not be divergence-free; to fix that, solve the equation $\nabla^2 p = \nabla \cdot \mathbf{w}$ for pressure and compute a new divergence-free velocity $\mathbf{u}' = \mathbf{w} - \nabla p$. For a full derivation see [7].

The pressure-solving step is by far the most computationally-intensive, taking around 80% of the time spent simulating (i.e., not rendering). I use the Jacobi iterative method, which is neither the fastest nor the most accurate, but is quite simple and usually good enough. Given more time I would have liked to switch to a preconditioned conjugate gradient method as in [1] and [2].

The advantages of this advection-projection method are that it is relatively straightforward to implement, even on a GPU, and that it is unconditionally stable, avoiding any unwanted "explosions". One disadvantage is that it leads to numerical dissipation, a gradual damping and fading of the fluid's fields. This can actually be useful in some cases – it diffuses smoke and heat without having to explicitly model that process. However, it also results in much smoother smoke plumes than one would expect from an explosion. To add some turbulence back in, I implemented vorticity confinement, as first described in [3]: first find the vorticity field $\omega = \nabla \times \mathbf{u}$ and compute $\eta = \nabla|w|$ and $\mathbf{N} = \eta/|\eta|$. Then the vorticity confinement force (added to the $\mathbf{F}$ component of the Navier-Stokes equation) is:

$$\mathbf{f}_{vc} = \epsilon h(\mathbf{N} \times \omega)$$

where $\epsilon > 0$ is an adjustable parameter and h is the cell side length. Values of ϵ from roughly 5 to 10 result in good-looking swirly structures in the smoke; higher values result in noisier but probably more realistic plumes.

3.2 Combustion model

As explained above, the fluid model does not support shockwaves, so the simulated explosions are deflagrations rather than detonations (that is, combustion is propagated by heat transfer rather than by a shock front). This kind of explosion includes both liquid/vapor explosions (e.g., gasoline, natural gas) and "suspended particle" explosions (e.g., coal, sawdust) as described in [4].

The heat and combustion model is similar to the one in [5]. Each cell contains a mixture of air, fuel, and smoke. Air is the default and its amount is not stored explicitly. Fuel, when not burning, has no effect on the dynamics of the fluid. Smoke is heavier than air and slowly sinks, though that force is usually countered by buoyancy (a force applied to each cell proportional to how much hotter it is than the ambient temperature). Lastly, cooling is simulated using a physically derived equation found in [6]:

$$T' = T - c_t \left(\frac{T - T_{amb}}{T_{max} - T_{amb}} \right)^4$$

where c_t is the cooling constant, T_{amb} is the ambient temperature, and T_{max} is an arbitrarily-chosen maximum temperature (6000K in the simulation).

Combustion occurs if a cell contains fuel and has a temperature above the ignition point (set to 500K, just below the autoignition point of gasoline). Fuel burns at a fixed rate and produces a proportional amount of smoke and heat. To mimic the rapid gas expansion caused by real combustion, I use the technique introduced in [4]: a combusting cell has its divergence artificially augmented by an amount proportional to the fuel consumed; that is, instead of solving $\nabla \mathbf{u} = 0$, the projection step will solve for $\nabla \mathbf{u} = \phi$, where ϕ is the extra divergence. The rates of fuel consumption and heat, smoke, and divergence production are configurable parameters whose values significantly affect the behavior of the explosion, with the divergence rate acting as a sort of "explosiveness" factor.

The results of this method, despite the essentially non-physical explosion mechanism, are quite impressive. An explosion can be created by adding a region of concentrated fuel with a high-temperature region in the center. The explosion expands rapidly, consuming all the fuel in a fraction of a second and producing a large amount of hot smoke. As the smoke cools and rises, it forms a realistic-looking plume, which through careful choice of parameters can even be made to resemble a mushroom cloud.

4 Rendering

The renderer uses a volumetric ray casting method derived mainly from [3]. A more complex photon-mapping renderer that accounts for internal scattering is also presented in [3], but is unnecessary for the low-albedo smoke dealt with here. For emission-related details and general implementation techniques, I also referred to [9].

First, the renderer casts a ray from the eye through each pixel of the image plane. Then, starting at the intersection with the bounding cube, it marches along the ray, sampling the voxel grid at equidistant points. The step size is $\sqrt{3}/n_{steps}$, with n_{steps} typically set to double the grid side length and $\sqrt{3}$ being the length of the cube's diagonal.

Each ray keeps track of a transmittance value t as it progresses through the grid. At each cell, t is reduced slightly in order to account for the absorption and scattering of light. The new value is $t' = (1 - \rho\sigma)t$, where ρ is the density of smoke in the cell and σ is a constant describing the absorbance of the smoke. When t falls below a threshold, the ray terminates early – anything farther along would be completely occluded by the smoke in front of it.

To determine the amount of light entering each cell, the renderer also traces rays from each cell towards a point light source. The incident-light value L_i of the cell is set to the light source's intensity scaled by the transmittance of this secondary ray. This allows the smoke to both shadow itself and cast shadows on any solid objects in the scene. Since the transmittance is a value in the range $[0, 1]$, the shadows have dark and light regions depending on the thickness of the smoke, as one would expect. (For efficiency, these secondary rays are only sent from cells that actually contain smoke.)

Each cell can also emit light with color and intensity determined by blackbody radiation. For a blackbody at temperature T, the emitted spectral radiance of wavelength λ is given by Planck's law:

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

where $C_1 \approx 3.7418 \cdot 10^{-16}\, Wm^2$ and $C_2 \approx 1.4388 \cdot 10^{-2}\, mK$. A lookup table mapping temperature to color is precomputed during initialization by sampling temperature values at regular intervals from 0 to t_{max}, then for each temperature, integrating the equation above against the CIE color-matching functions over the visible wavelengths (380 to 780 nm). The resulting XYZ tristimulus values are converted to RGB by normalizing them so that the brightest color has the value 1, then multiplying by the relevant 3x3 matrix. The light intensity is computed and stored as a fourth color component. During rendering, the light L_e emitted by a cell is calculated by linearly interpolating the two colors nearest to the cell's temperature and scaling by the intensity.

The overall light sent back to the eye from a cell is $L_o = (L_i + L_e)t$, and the total light accumulated along a ray is the sum of that formula for all intersected cells. This captures how more obscured cells contribute less to the final output color.

Lastly, solid objects are rendered as perfectly diffuse surfaces using Lambertian reflectance, which is not particularly interesting, but solids are not the point of this renderer.

5 Implementation

The simulation and rendering code are written in OpenCL and run entirely on the GPU. The choice of OpenCL instead of CUDA was made for me by the fact that I had no NVIDIA GPUs on which to develop. And moreover, this way the code should run on any GPU brand. The CPU-side code, written in C++, is mainly responsible for some setup procedures – reading the config file, setting up the OpenCL context, allocating but not filling blocks of video memory – and periodically enqueuing commands to the GPU. It also saves the rendered images to disk as PNG files. Because the state buffers are initialized by OpenCL code and never copied back to the CPU side, there are essentially no significant memory transfers apart from the output images.

All the code, along with a number of config files describing various scenes, are available on GitHub: https://github.com/jgfuchs/explode.

6 Results

Unfortunately, the only GPU that I had access to during this project was the Intel integrated chip in my three-year-old laptop.[1] Though still providing several orders of magnitude more computational power than the CPU, it was not able to run the code any faster than a few frames per second (on a 128x128x128 grid with 256 samples per ray). Due to the highly parallelizable nature of every step, I believe that performance would scale up very well on more powerful GPUs. Back-of-the-envelope calculations suggest it could reach real-time performance on an NVIDIA GTX 1070 or similar device.

A compilation of videos produced by running the simulation on a number of scenes with varying objects and parameters can be found here: https://www.youtube.com/watch?v=Jx933hGdaI4.

[1]Currently waiting on an Amazon AWS support ticket to let me launch GPU instances on EC2.

References

[1] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröoder. Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, July 2003.

[2] Robert Bridson and Matthias Müller-Fischer. Fluid simulation: SIGGRAPH 2007 course notes. In *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, pages 1–81, 2007.

[3] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. Visual simulation of smoke. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 15–22. ACM, 2001.

[4] Bryan E. Feldman, James F. O'Brien, and Okan Arikan. Animating suspended particle explosions. *ACM Trans. Graph.*, 22(3):708–715, July 2003.

[5] Nipun Kwatra, Jón T. Grétarsson, and Ronald Fedkiw. Practical animation of compressible flow for shock waves and related phenomena. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 207–215, 2010.

[6] Duc Quang Nguyen, Ronald Fedkiw, and Henrik Wann Jensen. Physically based modeling and animation of fire. *ACM Trans. Graph.*, 21(3):721–728, July 2002.

[7] Jos Stam. Stable fluids. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 121–128, 1999.

[8] Jos Stam. Real-time fluid dynamics for games. In *Proceedings of the Game Developer Conference*, 2003.

[9] Magnus Wrenninge and Nafees Bin Zafar. Volumetric methods in visual effects: SIGGRAPH 2010 course notes. In *ACM SIGGRAPH 2010 Courses*, SIGGRAPH '10. ACM, 2010.