Real-time GPU-based fluid simulation and rendering

Emil Manelius* Teodor Åberg†

Lund University
Sweden

Abstract

In this paper we discuss our implementation of a 3D fluid simulator based on the ideas of [Stam 2001] and [Fedkiw et al. 2001] with as much of the computational load on the gpu as possible in order to be able to run the simulation in real time. To render the smoke we use a ray-marching setup and let opacity be defined by the accumulated density along each ray as described in [Nguyen and Corporation 2008].

1 Introduction

Behaviours of natural phenomena are often dictated by complicated relations that can be expensive to simulate. When dealing with real-time graphics it is not only necessary that everything looks good but that it is also responsive. Based on [Stam 2001] and [Fedkiw et al. 2001] we try to implement these ideas on the GPU to be able to get realistic simulation while still retaining acceptable framerates.

2 Theory

2.1 Physics & Numerics

The basis of the simulations for the smoke is directly based on a combination of the simulations discussed in [Stam 2001] and [Fedkiw et al. 2001]. The Fundamental equation used here is the incompressible Euler equation

$$\begin{cases} \frac{\partial \bar{U}}{\partial t} = (\bar{U} \cdot \nabla)\bar{U} + \nabla p + \bar{F} \\ \nabla \cdot \mathrm{U} = 0 \end{cases}$$

as in [Fedkiw et al. 2001].

To implement the scheme a few different differential equations must be handled. The first of these are the most simple

$$\frac{\mathrm{d}y(x,t)}{\mathrm{d}t} = f(t, x, y(x,t)) \quad y(x_0, 0) = y_0$$

which can be approximated using a explicit Euler step with a time step Δt

$$y(x_0, \Delta t) \approx y_0 + f(0, x_0, y_0) \cdot \Delta t$$

The second type of equation needed to be solved are advection equations on the form

$$\frac{\partial V(x,t)}{\partial t} = -W \cdot \nabla V$$

which can be solved using the methods of characterisitcs described in [Fedkiw et al. 2001], where the approximate solution is

$$V(x_0, \Delta t) = V(P(x_0, -\Delta t), 0)$$

where $P(x_0, -\Delta t)$ is calculated by preforming an Euler step as described above with $f = W$ and a time step of $-\Delta t$.

The final differential equation that needs to be handled is the Poisson equation

$$\Delta Q = B$$

The main idea here is that the operator Δ can be discretized based on the imposed boundary conditions and represented as a matrix. As in [Fedkiw et al. 2001] this linear system was solved using the conjugate gradient method.

With these numerical methods layed out the solver can be constructed. Starting from some velocity field $\bar{U}^n = W_0$, density distribution ρ^n and Temperature distribution T^n at time n. First the driving force is calculated as

$$\bar{F}_{bouy} = -\alpha\rho^n\hat{z} + \beta(T^n - T_{amb})\hat{z}$$

$$\bar{F}_{conf} = \varepsilon h\big(N \times \omega\big) \text{ with } \begin{cases} \omega & = \nabla \times \bar{U}^n \\ N & = \frac{\nabla|\omega|}{|\nabla|\omega||} \end{cases}$$

$$\bar{F} = \bar{F}_{bouy} + \bar{F}_{conf}$$

Where α, β and ε are constants chosen to give desired behaviour for the simulation, T_{amb} is the ambient temperature, $\hat{z}$ is the chosen upward direction in the simulation and h is the distance between grid points in the discretization. $\hat{F}_{bouy}$ is to give the smoke bouyancy based on its temperature and density and $\hat{F}_{conf}$ is to introduce rotation into the field that otherwise dissipated due to numerical inaccuracies.

$$W_1 = W_0 + \Delta t \bar{F}_{tot}$$

Once the forces have been applied the field W_1 is advected with its own velocity

$$\frac{\partial W_1}{\partial t} = -W_1 \cdot \nabla W_1$$

which being solved with the method described earlier gives

$$W_2(x) = W_1(P(x, -\Delta t))$$

After this the solution can be projected such that it is divergence free. This is done by solving

$$\Delta p = \nabla \cdot W_2$$

for p and then

$$w_3 = w_2 - \nabla p$$

is the final updated velocity. The density and temperature are then update by advecting the previous fields along the updated velocity.

$$\rho^{n+1}(x) = \rho^n(P(x, -\Delta t))$$
$$T^{n+1}(x) = T^n(P(x, -\Delta t))$$

For the velocity field homogenous Dirichlet conditions apply for directions perpencular to the boundary and homogenous Von Neumann conditions apply to the pressure when solving the Poission Equation. The discretization of Δ is given by

*e-mail: emil.manelius@gmail.com
†teodor.aberg@gmail.com

$$\Delta U_{i,j,k} = \frac{1}{h^2}\Big(U_{i+1,j,k} + U_{i-1,j,k} + U_{i,j+1,k} + U_{i,j-1,k} + U_{i,j,k+1} + U_{i,j,k-1} - 6U_{i,j,k}\Big)$$

with some modifications at boundaries to ensure that conditions are met, where h is the grid distance. All quantities are calculated at the centers of voxels.

2.2 Rendering Smoke

The smoke is rendered using a ray marching approach similar to the one outlined in [Nguyen and Corporation 2008]. We model smoke as having some base color and a transparency that is proportional to the density of the smoke at each voxel.

3 Implementation

3.1 Rendering

The 3D-texture is attached to a model cube that is placed somewhere in the scene and rays are sent from the eye to the cube. To simplify our render step slightly, we align the model coordinates of the cube with the normalized texture coordinates of the 3D textures we store smoke data in. We march the rays through the cube using some fixed step size and accumulate color along the ray, setting the alpha value at each sample proportional to the smoke density in the 3D texture at that point. In order to do this, we need to compute the first intersection point with the cube for each ray that is sent from the eye. This is done by rendering the cube to a texture with dimensions identical to that of the current viewport with the backfaces culled. Each pixel in the texture is colored with the texture coordinates of the cube fragment being rendered. In a second pass, we cover the near plane of the view furstum with a screen-sized quad and render this quad using our raymarching shader. The shader will look up the color of each fragment from the texture generated in the previous step. If the color is empty, the ray from the eye through the fragment being rendered did not intersect the cube, and we do nothing. If the color value is nonempty, that means a ray from the eye through the fragment intersected the cube at the coordinate encoded in the texture, and we should start ray marching from that point. The coordinate we read from the cube is in the cube's texture space, and we want to do our ray march in this coordinate space since that simplifies the sampling of the 3D-texture. The camera position is transformed to the cube's model coordinate space (which in our case is the same as it's 3D-texture space) and we obtain a direction vector for our ray dir as

```
dir = normalize(intersection_point -
                camera_texture_space);
```

Since we now have a ray in the texture space of the 3D-texture we want to sample, we can perform ray marching in the following fashion

```
vec4 march_ray(vec3 start, vec3 dir,
               float step_size)
{
    vec4 accumulated_color = vec4(0, 0, 0, 0);
    vec3 position = start;
    while (inside_cube(position)) {
        float density = sampleDensity(position);
        accumulated_color += omega *
                             vec4(base_color,
                             k * density);
        position = position + dir * step_size;
    }
    return accumulated_color;
}
```

where ω is some factor to ensure that the final accumulated color is in a reasonable range and k a scale factor for the transparency of the smoke. To sample density, we sample the texture containing the smoke density at the point position since our ray is already in the correct texture space. To determine if we are in the cube, we perform a simple bounds check on the current position (coordinates in the range [0, 1] lie inside the cube). With alpha blending enabled the smoke can be placed in a background scene by performing the raymarching step last in the render order. Our current implementation does not support placing the smoke behind opaque objects in the scene, but this could be addressed by using depth maps in the render pass calculating the ray-cube intersections: if the distance from the eye to the cube front face is larger than the distance looked up from a depth map we store a zero instead of the texture coordinates.

3.2 Simulation

In order to try to make the simulation run in real time a majority of the simulation was implemented using compute-shaders to run on the GPU. Implementing the euler-step and advection is rather straight forward as these calculations are local. To calculate derivatives a symmetric approximation was used where applicable and forward/backward derivatives were used at boundaries as necessary. Trilinear interpolation is used to evaluate all quantities outside of grid points.

For the implementation of the conjugate gradient solver the steps needed for each iteration was implemented as separate compute shaders so that while the iteration is done on the CPU a majority of matrix and vector operations are done on the GPU. The necessary operations implemented were a scalar product, multiplying a vector U with the discretization of the Laplacian and taking linear combinations.

The scalar product was implemented by letting a compute shader calculate the partial sum of the scalar product in a $4 \times 4 \times 4$ subset of the grid and writing all these values to a texture getting the final scalar product by taking the sum of the resulting outputs on the GPU. The number 4 can be increased if it turns out that the summation of the resulting texture is a bottleneck for the simulation.

The structure of the discrete approximation of the Laplace operator means that multiplication with it can simply be implemented without having to do an explicit matrix multiplication. For an arbitrary matrix it could be necessary to for each output element go through each of the input elements, but this is not necessary in this case. From the expression presented in the theory only 7 elements have to be summed for each output element when calculating this matrix multiplication meaning each element of the output vector ΔU can be computed in constant time.

Linear combinations were computed by taking two textures as inputs and scalars and writing the linear combination of elements at some index to this index in the output texture.

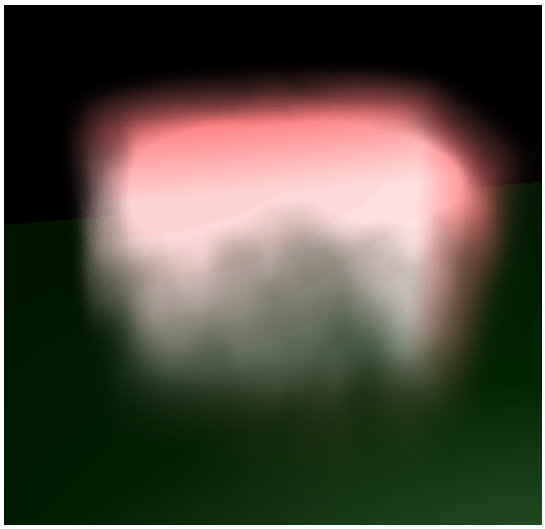
In order to get the conjugate gradient solver to converge it was necessary to impose a Dirichlet condition at a single boundary point as the Laplacian is not full-rank if this is not the case.

4 Results

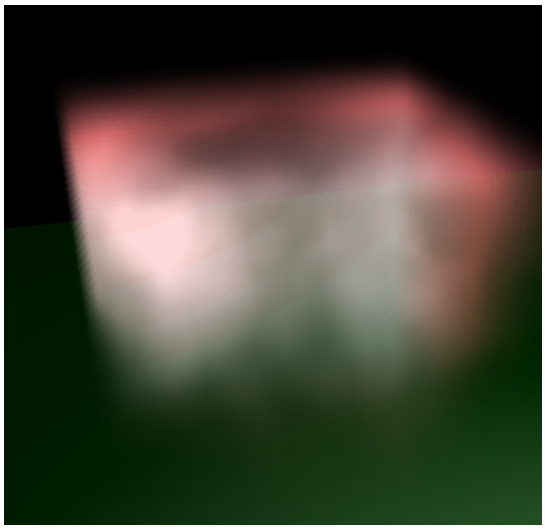
Generally we were satisfied with the results we were able to generate. The behaviour when colliding with the boundary seems to be correct with the simulation behaving as if the smoke was colliding with a solid plane. The effects effect of incresing ε is also clearly visible as more small-scale vortices can be seen, which is as expected.

For $N = 32$ we were able to run and render the simulation at 15 FPS. Increasing N decreased the frame rate in line with the expected increase in computational complexity: The size of the density texture scales like N^3. A summary of the performance for different values of N is listed in table 1 below.

(a) Initial state

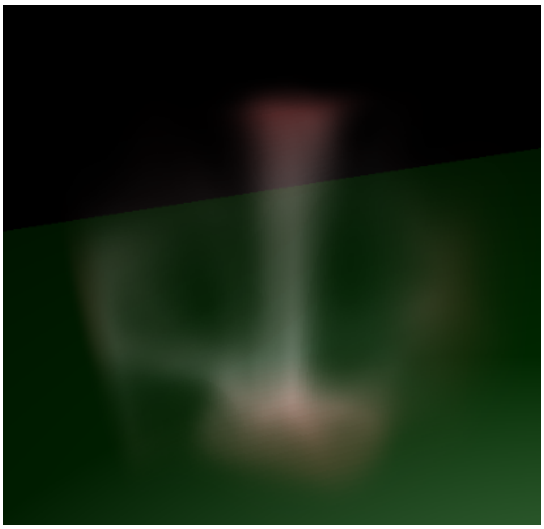


(b) Interaction with boundary



(c) State after majority of boundary interaction

Figure 1: Screenshots of smoke simulation in different states. Simulates a rising ball of smoke in a closed cube.



(a) Simulated mushroom cloud

Figure 2: Screenshot of a mushroom-cloud simulation.

N	Average framerate
32	16.5
44	6.5
52	3.8

Table 1: Frame rates depending on granularity of the grid. Ran on laptop with a NVIDIA GeForce GTX 1650

5 Discussion

We found that a simulation size of $N = 32$ provided a reasonable trade-off between visual quality and performance. 15 FPS seems reasonable when comparing the frame rates and simulation sizes obtained in [Stam 2001], considering the hardware difference. While slighlty higher performance would be required for the simulation to qualify as fully real time with modern standards, we believe this would be achievable with currently available stronger hardware.

We see a clear banding pattern when rendering our smoke. This is an artifact of the ray marching approach used for rendering and similar effects are described in [Nguyen and Corporation 2008] which we based our implementation on. The suggested solution in [Nguyen and Corporation 2008] is increase the sampling frequency or use a higher order filter when sampling the smoke texture.

We also attempted to implement the rendering algorithm described in [Fedkiw et al. 2001] in which a light source effects the luminescence of the smoke based on the viewing angle but were not able to make if fully work with the rest of the pipeline that we had set up for general density distributions, though we were able to make it look convincing for a uniform density distribution

Some improvements that could be plugged into the current pipeline could be to use some non-linear interpolation in order to determine that values of positions that are not voxel-centers more smoothly. Another is to implement support for smoke-solid interaction inside of the grid by tracking elements inside the simulation-volume and handling the boundary conditions and how deep the ray-marching. These ideas are discussed in [Nguyen and Corporation 2008] and [Fedkiw et al. 2001]

References

BOLZ, J., FARMER, I., GRINSPUN, E., AND SCHRÖDER, P. 2003.

Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. *ACM Trans. Graph. 22* (07), 917–924.

FEDKIW, R., STAM, J., AND JENSEN, H. 2001. Visual simulation of smoke. *ACM SIGGRAPH2001, 2001.* (06).

NGUYEN, H., AND CORPORATION, N. 2008. *GPU Gems 3*. No. v. 3 in Lab Companion Series. Addison-Wesley.

STAM, J. 2001. Stable fluids. *ACM SIGGRAPH 99 1999* (11).