Real-Time Fluid Simulation on the GPU

Niklas Strandberg[*] Aron Söderling[†]

Lund University
Sweden

Abstract

The simulation of fluids in real-time graphics can have a great impact on the visual appearance, but to simulate this on a GPU using a traditional rasterization-pipeline is not straightforward.

In this paper, we describe one way to implement this on the GPU, using an existing rasterization-based pipeline.This is done by doing multiple render-passes per frame and saving the computed values for several physical phenomena into textures, similar to deferred shading where the rendering is split into serveral render-passes. To be able to achieve this in real-time, the fluids are simulated using simplified physical models that are efficiently calculated on the GPU, but that reasonably well reflect the real world. Due to the limitations of the project we had to focus on making one effect and this effect ended up being smoke.

1 Introduction

Simulating fluid physics is a very demanding task for a processor to run, there are many computations that has to run for every element in the fluid. However this is also a very parallel task, since every single element has to run the exact same operation every loop cycle, which fits the GPU very good. Our implementation utilises the computational power of the GPU by running shader programs for every pixel and storing intermediate results into textures. The simulation can be used both as a computer graphics tool to produce good looking fluids such as smoke, water, fire etc. or as a research tool which could be used for example to simulate how smoke spreads buildings or the flow of blood in medical studies. This project also works as an introduction to general-purpose GPU(GPGPU) and how powerful the GPU can be when running highly parallel computations.

The algorithm for running fluid simulation is based on the Navier-Stokes equations for incompressible flow. The equations describes how the velocity of a flow varies over time incident to the diffusion of the flow, the pressure in the fluid and the influence of external forces. A description of how to solve these equations and how to implement it on a GPU is presented in this report. We implemented this algorithm using the Renderchimp framework and started with the deferred shading assignment code from the course.

Our initial plan was to render three dimensional fluids such as water, fire and smoke but we struggled with the implementation and only had time to pleasingly render two dimensional smoke. It should however be possible to extend our solution to cover those things as well.

2 Algorithms

2.1 Introducing the Navier-Stokes equations

The flow of a fluid can be represented by a velocity and pressure in every molecule of the fluid. There are two methods of implementing this, Lagrangian and Eulerian. The Lagrangian method is based on particles that store velocity and pressure. The Eulerian method is based on using grids for storage. We chose to implement the Eulerian method.

[*]adi10nst@student.lu.se
[†]adi10aso@student.lu.se

Using this method the flow is represented by a vector field U for velocity and a scalar field P for pressure. These fields vary both in time and space. When initial state of the fields are known then the fluid can be represented by the Navier-Stokes equations (1, 2):

$$\frac{\delta u}{\delta t} = -(u \cdot \nabla)u - \frac{1}{\rho}\nabla p + v\nabla^2 u + F \tag{1}$$

$$u \cdot \nabla = 0 \tag{2}$$

Eq (1) consists of four terms, each describing a factor that influence the flow of the fluid. These factors are called advection, pressure, diffusion and external forces. All of them are accelerations and they represent different physical phenomena.

Advection: Advection (3) describes how a flow transports other objects and itself. This term represents how the flow transports itself.

$$-(u \cdot \nabla)u \tag{3}$$

Pressure: Pressure builds up when molecules collide, this pressure then pushes other molecules away and the movement propagates though the fluid. This is represented by the pressure term (4).

$$-\frac{1}{\rho}\nabla p \tag{4}$$

Diffusion: Fluids can have a resistance to movement. This resistance is described by the viscosity of a fluid. High viscosity leads to slower movement e.g. honey, low viscosity leads to more movement e.g. smoke. Viscosity leads to diffusion of the flow and is represented by this term in the equation (5).

$$v\nabla^2 u \tag{5}$$

External forces: This term represents external forces such as gravity, pushing objects, wind etc. 6.

$$F \tag{6}$$

2.2 Solving the Navier-Stokes equations

To solve the different terms of the Navier-Stokes equations we need to use some vector mathematics. The following formulas describe the different uses of the nabla operator ∇ that are used to solve the different terms:

Gradient

$$\nabla p = (\frac{\delta p}{\delta x}, \frac{\delta p}{\delta y}) \Rightarrow \frac{p_{i+1,j} - p_{i-1,j}}{2\delta x}, \frac{p_{i,j+1} - p_{i,j-1}}{2\delta y} \tag{7}$$

Divergence

$$\nabla \cdot u = \frac{\delta u}{\delta x} + \frac{\delta u}{\delta y} \Rightarrow \frac{u_{i+1,j} - u_{i-1,j}}{2\delta x}, \frac{v_{i,j+1} - v_{i,j-1}}{2\delta y} \tag{8}$$

Laplacian

$$\nabla^2 p = (\frac{\delta^2 p}{\delta x^2}, \frac{\delta^2 p}{\delta y^2}) \Rightarrow \frac{p_{i+1,j} - 2p_{i,j} + p_{i-1,j}}{(\delta x)^2}, \frac{p_{i,j+1} - 2p_{i,j} + p_{i,j-1}}{(\delta y)^2}$$
$$\overset{\delta x = \delta y}{=} \frac{p_{i+1,j} + p_{i-1,j} + p_{i,j+1} + p_{i,j-1} - 4p_{i,j}}{(\delta x)^2} \tag{9}$$

Worth noting is that the right side of the arrow in (7), (8) and (9) is the finite difference form of the left side.

It is only possible to solve the Navier-Stokes equations analytically in a few simple cases so we use an incremental numerical approach instead.

To compute the new velocity field at a time step we can simply take the previous velocity field, apply advection, calculate diffusion and add influence of external forces. This result will however violate the second part of the Navier-Stokes equations, the divergence will no longer be zero. This can be solved by using the Helmholtz-Hodge decomposition theorem:

$$w = u + \nabla p \tag{10}$$

Where *w* is the non-divergence free velocity field, *u* is the divergence free velocity field and *p* is the pressure field. From this theorem we get:

$$u = w - \nabla p \tag{11}$$

Basically we can get the divergence free field by subtracting the gradient of the pressure field. Now we just need a way to compute the pressure field which is given by applying the divergence to the same theorem (10):

$$\nabla \cdot w = \nabla \cdot u + \nabla \cdot \nabla p = \nabla \cdot u + \nabla^2 p \tag{12}$$

Since (2) demands that divergence of *u* is zero this leads to:

$$\nabla^2 p \cdot w = \nabla \cdot w \tag{13}$$

This is a poisson-equation which can be solved with Jacobi iterations using *w* as input, which will be presented in 2.3. Since we now know how we can get pressure we just need an exact way of calculating w. From the definition of the dot product we know that we can compute the projection of a vector on a unit vector by applying dot product between the two vectors. This operation can also be applied to vector field. We can take advantage of this and define a projection operation $\mathbb{P}$ that takes the field *w* and projects it onto the divergence-free field *u*. Now we can apply this projection operation to (10) and since $\mathbb{P}(w) = \mathbb{P}(u) = u$ we get:

$$\mathbb{P}(\nabla p) = 0 \tag{14}$$

Now we can apply the same projection operation to the first Navier-Stokes equation (1):

$$\mathbb{P}(\frac{\delta u}{\delta t}) = \mathbb{P}[-(u \cdot \nabla)u - \frac{1}{\rho}\nabla p + \upsilon \nabla^2 u + F] \tag{15}$$

Since *u* is divergence free so is it's derivative so the projection disappears from the left hand side of the equation. Combining this with what we know from (14) we arrive at the following equation which is the final equation that our implementation will solve.

$$\frac{\delta u}{\delta t} = \mathbb{P}[-(u \cdot \nabla)u - \frac{1}{\rho} + \upsilon \nabla^2 u + F] \tag{16}$$

This is an equation we can compute at each time step. We have access to the velocity field *u* and can simply apply the three inner operations, add them together to produce *w* and apply the projection operation to arrive at the final result which is the new velocity field.

Typically, the different components are not computed as in (16) but instead calculated using state transformations. Each component takes a vector field as input, performs calculations and produces a new vector field that is used as input to the next component. We therefore define $\mathbb{S}$ as the solution to (16) for a single time step. $\mathbb{S}$ can be split into operations (applied right to left):

$$\mathbb{S} = \mathbb{P} \circ \mathbb{F} \circ \mathbb{D} \circ \mathbb{A}(u) \tag{17}$$

$\mathbb{A}$ = Advection, $\mathbb{D}$ = Diffusion, $\mathbb{F}$ = External Forces, $\mathbb{P}$ = Projection

Advection To compute the advection at the current time we need to access the velocity at the previous time step. Since we intend to implement this in fragment shaders and we cannot change the position of the current pixel, we instead back trace the velocity for each grid cell and copy the contents from this position into the current pixel. This can be applied on velocity, density, temperature etcetera.

Diffusion

$$(I - \upsilon \delta t \nabla^2) u(x, t + \delta t) = u(x, t) \tag{18}$$

where I is the identity matrix.

This is a Poisson equation that we solve using Jacobi Iterations (described in 2.3).

2.3 Solving the Poisson equations

To solve the diffusion equation (18) as well as the Poisson pressure equation (4) we use several Jacobi iterations. Both equations can be expressed on the form:

$$x_{i,j}^{(k+1)} = \frac{x_{i-1,j}^{(k)} + x_{i+1,j}^{(k)} + x_{i,j-1}^{(k)} + x_{i,j+1}^{(k)} + \alpha b_{i,j}}{\beta} \tag{19}$$

This equation is evaluated once per iteration and the result is given as input for the next iteration. The initial input has to be guessed and in our implementation the initial guess is 0. After an amount of iterations (20-40) the result starts to converge towards the desired value. There are other solutions than Jacobi that requires less iterations, but are far more complicated to implement on the GPU hence we chose to use Jacobi.

2.4 Initial and boundary conditions

In order to have a fully functioning model, we need to know what the initial state is for our velocity and pressure. We set the initial velocity and pressure to zero throughout the grid. We also need to know and what happens at the boundaries; we need to specify our boundaries and boundary conditions. We imagine our simulation takes place in a box that is positioned so that its entire contents is visible on the screen. To get the effect of our fluid being contained in this box, we wanted the fluid to "bounce" on contact with the walls. Therefore our velocity boundary condition was to take the negative value of the nearest cell. For pressure we set the boundary to have the same pressure as the nearest cell to get a realistic simulation.

3 Implementation

From mathematical background we get (17) which we can implement. Every time step is the same, this code represents a single time step:

```
u = advect(u);
u = diffuse(u);
u = addForces(u);
u = project(u);
```

Where the project operation consists of two operations:

```
p = computePressure(u);
u = subtractPressureGradient(u, p);
```

This code could be implemented both on the CPU and the GPU, however as stated in the introduction, the parallel nature of the GPU makes it a much better match. To implement this kind of algorithm on the GPU we need to learn how to do this kind of computation on the GPU. In Renderchimp the algorithm looks somthing like this:

RCInit():

- Create quad that span the entire grid

- Setup textures to store all the different values at the grid cells(velocity, temperature, pressure, etc)
- Initiate the fragment shaders that will perform the neccessary calculations

RCUpdate():

- Render the quad into the different buffers using the different shader programs that perform the operations(advect, diffuse, divergence etc.)
- Iterate Jacobi renders to compute the pressure
- Subtract the pressure gradient to arrive at the final velocity field

To run the simulation we need at least two buffers that we render to, velocity and pressure. Running only this simulation is however not very interesting. To get something interesting our of it you need to put somthing into the fluid that is carried through the flow and can be visualized. Since we focused on making smoke we chose to visualize this by adding a buffer for the density and a buffer for temperature. In reality the temperature of smoke influences the velocity of the flow, hot smoke rises faster and cool smoke slows down. This also creates a swirliness. Smoke without this swirliness doesn't look like smoke at all so we had to implement another operation in our algorithm, bouyancy. This operation takes the temperature field, compares the temperature to the ambient temperature and adds a value to the velocity. Since changes in temperature now causes the velocity to change we no longer need to add forces directly to the velocity, but can instead add an impulse in temperature to start the flow.

Another property of smoke is that it has a very low diffusion. So low that it really has no effect on visual appearance and therefore the diffusion operation could be skipped. We also added some simple interaction where left click moves the temperature and density impulse to the clicked location and right click lowers the temperature at the clicked area.

4 Results

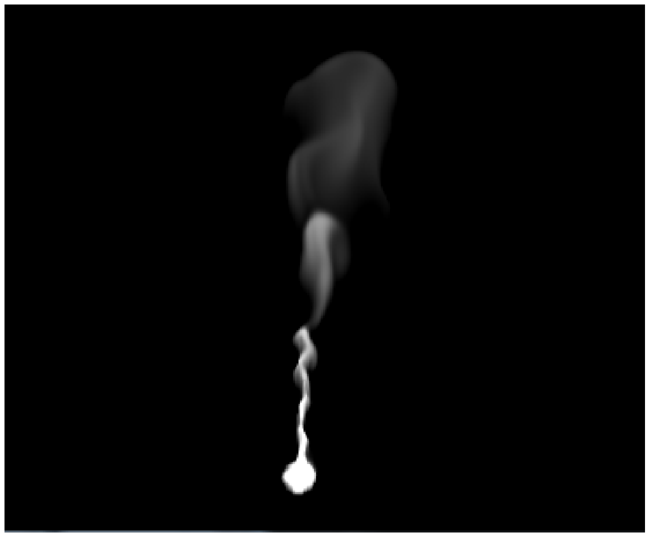
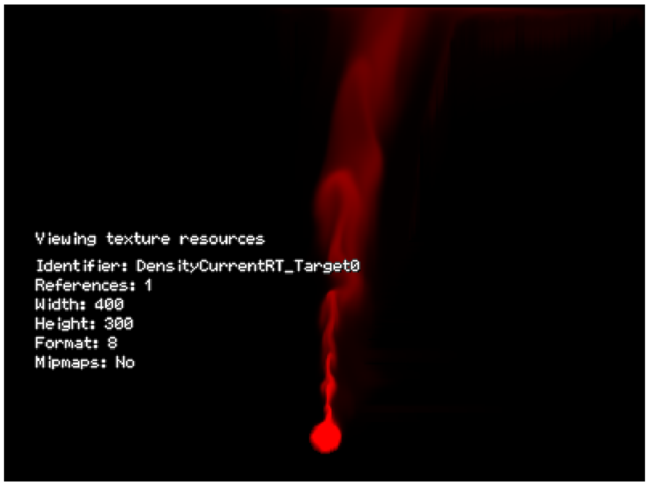


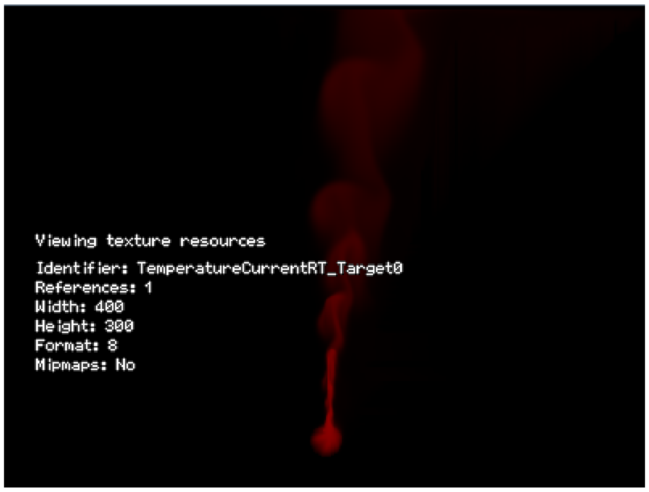
Figure 1: Smoke rendered using a combination of density and temperature

Unfortunately, we spent a lot of time struggling with various bugs in the 2D simulation which meant we did not manage to get a working 3D implementation. This should (at least theoretically) not be too difficult by using a multiple vector fields instead of just one and extending the calculations to three dimensions.



```
Viewing texture resources
Identifier: DensityCurrentRT_Target0
References: 1
Width: 400
Height: 300
Format: 8
Mipmaps: No
```

Figure 2: The density texture during simulation



```
Viewing texture resources
Identifier: TemperatureCurrentRT_Target0
References: 1
Width: 400
Height: 300
Format: 8
Mipmaps: No
```

Figure 3: The temperature texture during simulation

Due to the difficulties we had we were not able to experiment with different kinds of fluids but focused mainly on smoke. We did however achieve visually pleasing smoke rendering as seen in figure 1. Figure 2 and 3 are rendered with the same parameters but shows the density and temperature textures during the simulation.

The performance of the simulation could be greatly improved by optimizing the implementation. We have not focused on this but instead focused on making an as simple and straightforward solution as possible. Possible optimizations are discussed further in section 5.

5 Discussion

We struggled a lot with small bugs in the 2D simulation resulting in weird visual artifacts. One particularly persistent bug we believe was caused by small rounding errors when converting from a RGB-value in the range 0 to 255 to a float in the range -1 to 1.

We currently use the same grid size for the physics simulation as the rendered image to simplify the implementation. This is not optimal and reducing the grid size for the simulation would dramatically increase performance without having a great negative impact on the final result. With this optimization we could increase the number of Jacobi iterations to get an even more realistic result.

Normally in a game you don't want smoke covering an entire scene, but only around certain objects such as a chimney or a fire. Since the smoke would dissipate it wouldn't reach the boundaries of the grid and thus should not produce any weird looking artifacts.

Another solution that we considered was to use OpenCL for the computational parts of our program to make it even more effective. After some research however we agreed that it would be to much work to learn OpenCL as well as implement our program to be able to finish in time.

References

CRANE, K. 2008. *GPU Gems 3*. In *GPU Gems* [Nguyuen and Corporation 2008], ch. 30. Real-Time Simulation and Rendering of 3D Fluids.

FERNANDO, R. 2006. *GPU gems: programming techniques, tips, and tricks for real-time graphics*. Addision-Wesley.

HARRIS, M. J. 2006. *GPU gems: programming techniques, tips, and tricks for real-time graphics*. In [Fernando 2006], ch. 38. Fast Fluid Dynamics Simulation on the GPU.

NGUYUEN, H., AND CORPORATION, N. 2008. *GPU Gems 3*. GPU Gems. Addison Wesley Professional.