Smoke simulation
TN1008, Linköping University

Viktor Axelsson , vikax333@student.liu.se
Anders Nord, andno922@student.liu.se
Alexander Svensson, alesv019@student.liu.se
Amaru Ubillus, amaub859@student.liu.se

September 1, 2013

Abstract

In this paper an efficient method for simulating smoke entirely on the GPU is implemented. The model is unconditionally stable and produces complex fluid-like flows. A brief explanation of fluid simulation and the associated mathematics are also discussed. After reading this paper, the reader should have got good knowledge of basic fluid dynamics and how to simulate fluids using the GPU.

Contents

Chapter 1

Introduction

Fluid simulation is nowadays daily used in the special effect industry and is a popular tool in computer graphics for generating realistic animations of water, smoke, explosions, and related phenomena.

This report explains a method to simulate smoke on the GPU[1]. Because of the large amount of parallelism in graphics hardware, the simulation runs much faster on the GPU than on the CPU[2].

1.1 Previous Work

The *Navier-Stokes equations* are a set of equations for describing fluids in motion. Dynamical models of fluid based on the Navier-Stokes equations were first implemented in two dimensions by Yaeger and Upson 1986 [4]. Foster and Metaxas 1996 [5] and 1997 [6] did some work in three-dimensional Navier Stokes equation to create fluidlike animations. Their model has the advantage of being simple to code, since it is based on a finite differencing of the Navier-Stokes equations and an explicit time solver. Explicit solvers calculate the state of the fluid at a later time from the state at the curent time, while implicit solvers find a solution by solving equation involving both the current state of the fluid and the later one. The main problem with explicit solvers is that the numerical scheme can become unstable for large time-steps.

Stam 1999 [8] proposed an unconditionally stable algorithm that solves the full Navier-Stokes equations. The method uses both Langrangian and implicit methods to solve the Navier-Stokes equations, instead of an explicit Eulerian scheme to obtain a stable solver. This allows the user to simulate fluids on larger grids in real-time since the time steps can be much larger.

[1]Graphics Processing Unit
[2]Central Processing Unit

1.2 Background

Previous methods such as Foster and Metaxas 1997 [6]; Stam 1999 [8] and Fedkiw et al.2001 [2], can produce near real time results on small grids. On larger grids these methods are ineffective and they require a lot of memory just to store the density and velocity fields.

The method described in this report was originally supposed to be a GPU implementation of the paper *"Smoke Simulation For Large Scale Phenomena"* by Rasmussen et al.2003 [7]. But the work ended up as a combination between that paper and two others.

The paper "stable fluid" by Stam 1999 was used to achive a stable solver. But while Stam's simulations used a CPU implementation, the simulation method described in the following sections is implemented on the GPU. The article *"Fast Fluid Dynamics Simulation on the GPU"* by Mark Harris, 2004 [3] describes how to implement the method of Stam on the GPU.

By implementing the Navier-Stokes equations on the GPU higher performance can be achieved through its greater parallel processing power.

1.3 Structure of report

In chapter 2 the simulation method is discussed, this will cover all the background information needed to understand the Navier-Stokes equations and the matematical background to fluid simulation.

In chapter 3 more specific information related to the implementation is discussed and in chapter 4 the performance of the implemented algorithm will be presented and analysed.

In chapter 5 the result will be discussed and finally in chapter 6 a conclusion can be drawn of the work presented.

Chapter 2

Simulation Method

2.1 Mathematical Background

A mathematical representation of the state of the fluid is needed in order to simulate the behaviour of a fluid. The most important quantity needed to represent a fluid is the velocity, which determines how a fluid moves along itself and the things inside it. A fluid can be represented by a *vector field* and varies in both time and space. For this report we assume a two-dimensional Cartesian grid where the velocity vector field of our fluid is defined.

The key to fluid simulation is to take steps in time and, at each time step, correctly determine the current velocity field. This can be done by solving the Navier-Stokes equations.

2.2 Navier-Stokes Equations

The famous Navier-Stokes equations describe how a fluid flow changes over time. The flow is described by **V**, a vector field. In the Navier-Stokes equations this vector field denotes the velocity of the fluid at every point in space and thus **V** is referred to as the velocity field of the fluid. The equations for incompressible flow can be written as:

$$\frac{\delta \mathbf{V}}{\delta t} = -(\mathbf{V} \cdot \nabla)\mathbf{V} - \frac{\nabla p}{p} + v\nabla^2\mathbf{V} + \mathbf{F} \tag{2.1}$$

$$\nabla \cdot \mathbf{V} = 0 \tag{2.2}$$

The Navier-Stokes equations may initially look complex, but to understand them better it is possible to break them into more simple pieces.

- $(\mathbf{V} \cdot \nabla)\mathbf{V}$ is called the *self-advection* term and is responsible for "moving the flow with itself". The velocity of a fluid causes the fluid to transport objects densities along with the flow. This non-linear term is very important for the behaviour of swirls and vortices.
- $\frac{\nabla p}{p}$ is the pressure field p. When a force is applied to a fluid, it does not instantly propagate through the entire volume. Instead, the particles close to the force push on those farther away, and pressure builds up. Pressure is force per unit area and any pressure in the fluid naturally leads to acceleration.
- $v\nabla^2\mathbf{V}$ is the diffusion term which correspond to the fluids internal friction. It controls the thickness of the fluid. Viscosity is a measure of how resistive a fluid is to flow. Tick fluids like honey have high viscosity and flow slowly. Thin fluids like alkohol, flow quickly and have low viscosity.
- $\mathbf{F}$, is the external force term. It applies forces at the fluid such as gravity wind and buoyancy. The forces can be either *local forces* or *body forces*. Local forces are applied to specific regions of the fluids while body forces affect the entire fluid.
- $\nabla \cdot \mathbf{V} = 0$ is an additional constraint that enforces a divergence free solution. This means that the volume of the fluid must remain constant over time.

The terms in the Navier-Stokes equation contain three different uses of the symbol ∇, which is known as the *nabla operator*. The three applications of nabla are the gradient ∇, the divergence $\nabla\cdot$, and the Laplacian operators ∇^2.

2.3 Solving the Navier-Stokes Equations

The Navier-Stokes equations cannot be solved analytically but it is possible to solve them with numerical integration techniques. A first step would be to transform the equations into a form that is more close to numerical solution and divide the solution of the Navier-Stokes into simple steps.

By using a technique known as *operator splitting* the Navier-Stokes equations can be solwed in parts. Given an initial vector field $\mathbf{V}_0$ at some point in time t_0 the new vector field $\mathbf{V}_{\Delta t}$ is calculated by solving a series of subproblems. The terms in equation 2.1 can be arranged as in figure 2.1 to solve the problem:

$$\boldsymbol{V}_0 \overset{(\boldsymbol{V}\cdot\nabla)\boldsymbol{V}}{\rightarrow} \boldsymbol{V}_1 \overset{\boldsymbol{F}}{\rightarrow} \boldsymbol{V}_2 \overset{v\nabla^2\boldsymbol{V}}{\rightarrow} \boldsymbol{V}_3 \overset{\frac{\nabla p}{\rho},\nabla\cdot\boldsymbol{V}}{\rightarrow} \boldsymbol{V}_{\Delta t}$$

Figure 2.1: Operator splitting.

1. The temporary field $\mathbf{V}_1$ is calculated from $\mathbf{V}_0$ by solving for the advection term, $(\mathbf{V} \cdot \nabla)\mathbf{V}$.

2. Then $\mathbf{V}_2$ can be calculated from $\mathbf{V}_1$ by adding an external force, $\mathbf{F}$.

3. The vector field $\mathbf{V}_3$ can be calculated from $\mathbf{V}_2$ by solving for the diffusion term, $v\nabla^2\mathbf{V}$.

4. Finally the new vector field $\mathbf{V}_{\Delta t}$ can be calculated from $\mathbf{V}_3$ by projecting the velocity field onto it divergence free parts, this includes the terms, $\frac{\nabla p}{p}$ and $\nabla \cdot \mathbf{V} = 0$.

2.4 Advection

Advection is the process by which a fluids velocity transports itself and other quantities in the fluid. To compute how a quantity moves along the velocity filed it is possible to imagine that each grid cell is represented by a particle. The grid may then be updated in the same way as a particle system by using Eulers method for explicit (forward) integration to move each particle forward along the velocity field.

The problem with this method is that using explicit methods for advection will unfortunately result in a solver that easily becomes unstable for large time steps.

The solution is to use the implicit method described by Stam 1999 [8]. In order to calculate the new temporary velocity field $\mathbf{V}_1$ from $\mathbf{V}_0$ we need to solve the following eqution:

$$\frac{\delta \mathbf{V}_1}{\delta t} = -(\mathbf{V}_0 \cdot \nabla)\mathbf{V_0} \tag{2.3}$$

These equations can be solved by the *method of characterestics*.

Instead of computing where a particle moves over the current time step, the particle is traced from each grid cell back in time to its previous position. Given a point x, the point x is backtraced through the velocity field $\mathbf{V}_0$ over a time Δt. The new velocity at the point x is then set to the velocity the particle, now at x, had at its previous location a time Δt ago:

$$\mathbf{V}_1(x) = \mathbf{V}_0(p(x, -\Delta t)) \tag{2.4}$$

As can be seen in figure 2.2 by tracing the velocity field back in time leads to the green $\mathbf{x}$ which are bilineary interpolated, and the result is copied to the starting grid cell.

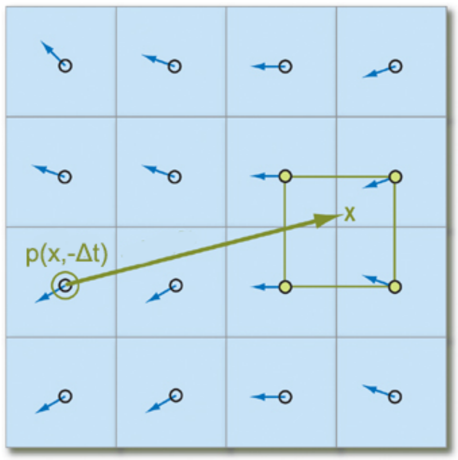


Figure 2.2: Computing fluid advection.

2.5 External Forces

To solve the external force term, the force-field $\mathbf{F}$ is simply defined and $\mathbf{V}_2$ can be created from $\mathbf{V}_1$ by using Euler integration:

$$\mathbf{V}_2 = \mathbf{V}_1 + \Delta t \cdot \mathbf{F} \tag{2.5}$$

2.6 Diffusion

Viscous fluids have a certain resistance to flow, which results in diffusion of velocity. A partial differential equation for viscous diffusion is:

$$\frac{\delta \mathbf{V}_2}{\delta t} = v \nabla^2 \mathbf{V}_2. \tag{2.6}$$

Again the stable method by Stam [8] is used, and an implicit formulation of equation 2.6 can be written as:

$$(\mathbf{I} - v \delta t \nabla^2)\mathbf{V}_3(x, t + \delta t) = \mathbf{V}_2(x, t) \tag{2.7}$$

Where $\mathbf{I}$ is the identity matrix. This equation is a *Poisson equation* for velocity and can be solved using an iterative technique, see section 2.8.

2.7 Projection

The final step in the Navier-Stokes solver algorithm is to enforce incompressibility. Solving the Navier-Stokes equations involves three computations to update the velocity at each time step: advection, force application and diffusion. The result is a new velocity field, $\mathbf{V}_3$, with nonzero divergence. But equation 2.2 requires that each time step end with a divergence free velocity.

The *Helmholtz-Hodge decomposition* states that it is always possible to split a vector field into a *divergence free* part and a *curl free* part:

$$\mathbf{V}_3 = \mathbf{V}_{df} + \mathbf{V}_{cf} \tag{2.8}$$

The final divergence free vector field $\mathbf{V}_{\Delta t}$ can be identified as $\mathbf{V}_{df}$. A gradient field is always curl free and can therfore be constructed as the gradient of some scalar field p:

$$\mathbf{V}_3 = \mathbf{V}_{\Delta t} + \nabla p \tag{2.9}$$

The divergence of the velocity can then be corrected by rearrange equation 2.9 and subtract the gradient of the resulting pressure filed.

$$\mathbf{V}_{\Delta t} = \mathbf{V}_3 - \nabla p \tag{2.10}$$

To compute and find the value of the pressure field p the divergence operator is applied to both sides of Equation 2.9

$$\nabla \cdot \mathbf{V}_3 = \nabla \cdot (\mathbf{V}_{\Delta t} + \nabla p) = \nabla \cdot \mathbf{V}_{\Delta t} + \nabla^2 p \tag{2.11}$$

And since equation 2.2 say that $\nabla \cdot V = 0$ the equation can be simplified to:

$$\nabla^2 p = \nabla \cdot \mathbf{V}_3 \tag{2.12}$$

Which is a Poisson equation for the pressure of the fluid. Since $\mathbf{V}_3$ is known this allows us to find the p that fulfills the Helmholtz-hodge decomposition, equation 2.9. The new divergence-free field $\mathbf{V}_{\Delta t}$ is now possible to compute from equation 2.10.

2.8 Solution of Poisson Equations

Equation 2.12 and equation 2.7 belong to a class of equations called Poisson equations. In this report an iterative solution technique is used to solve the Poisson equations.

The Poisson equation is a matrix equation in the form $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{x}$ is the vector of values for which we are solving ($\mathbf{V}$), $\mathbf{b}$ is a vector of constant, and $\mathbf{A}$ is a matrix. $\mathbf{A}$ is implicitly represented in the Laplacian operator ∇^2 and does not need to be explicitly stored as a matrix. The iterative solution technique used is one of the simplest and is called Jacobi iteration. More complex method such as conjugate gradient and multigrid methods are better and converge faster. But Jacobi iteration has the advantage of being simple and easy to implement.

The technique starts with an initial guess $\mathbf{x}^{(\mathbf{0})}$ and for each step k an improved solution is calculated. Equation 2.12 and equation 2.7 may look different but both can be discretized and rewritten in the form:

$$x_{i,j}^{k+1} = \frac{x_{i-1,j}^{(k)} + x_{i+1,j}^{(k)} + x_{i,j-1}^{(k)} + x_{i,j+1}^{(k)} + \alpha b_{i,j}}{\beta} \tag{2.13}$$

In this equation the values α and b are constants and the values of x, β, a and b are different depending to which equation they belong.

In the Poisson-pressure equation 2.12 x represents p, β represents $\nabla \cdot w$, a $= -(dx)^2$ and $b = 4$.

For the viscous diffusion equation 2.7, both x and β represent V, $a = (dx)^2/ndt$, and $b = 4 + a$.

To solve the equations a number of iterations are run and equation 2.13 is applied at every grid cell. The result from the previous iteration is used as input to the next and since Jacobi iteration converges slowly many iterations is needed to achieve a good result.

Chapter 3

Implementation

In order to decrease the amount of computations when rendering a volume, the method describes dividing the volume into slices. The slices are described as either stacked or rotated about a center axis (figure 3.5).

The idea is to obtain an implicit 3D-volume by interpolation of the slices.

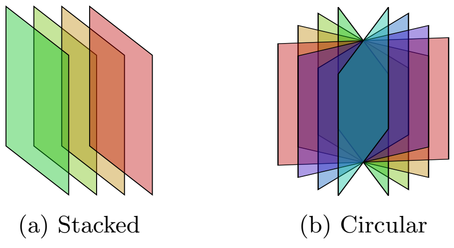
3.1 Adaptation for GPU

When a shader is run, it is unaware of the result of its last execution. The Navier-Stokes equations, however, require values from previous time steps.

To solve this problem, the 2D data fields for every slice are saved in textures. The shaders are set up to read from an array of *previous* textures and will render to a corresponding array of *current* textures.

It is the usage of framebuffer objects in OpenGL that enables this rendering to off-screen textures.

1. For each slice in every time step:
    (a) Render each field to a texture, using previous textures as input.
    (b) Copy the current textures to previous textures for the next time step.
2. Render smoke to the screen using the density texture.



(a) Stacked (b) Circular

Figure 3.1: Slice constellations

3.2 Implementation method

When implementing a Navier-stokes-fluid on the GPU there is no need for creating a grid. The pixels in the texture will be used as grid-cells. Each cell(pixel) in a texture can contain 4 values. For a picture this would be r, g, b and a, which can be compared to a vector containing 4 values.

The velocity-texture for example contains vectors in each cell that tells us the velocity in the x and y-axis. See figure 3.2.

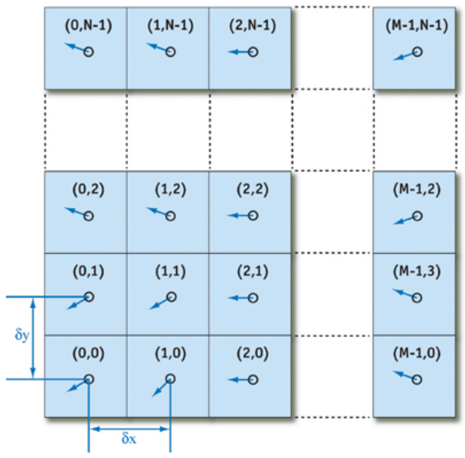


Figure 3.2: The Velocity Grid represented by vectors [1]

These are the shaders currently implemented, in the order they are executed:

1. Advection
2. Diffusion
3. Projection (3 shaders)
   Divergence
   Jacobi iteration (Calculates the pressure field)
   Gradient

These are the textures currently implemented:

- Velocity
- Density
- Divergence
- Pressure

3.2.1 Advection

The first shader in the pipeline is the advection shader. The values in the density-texture are advected according to section 2.4 by the velocity field.

The velocity field is also advecting itself after it has advected the density. This is called *self-advection*.

As written in section 2.4 the interpolation is bilinear. This can be set when creating the texture and is handled automatically by the GPU.

The implementation of the advection shader is shown in listing 3.1

```

    -------Advection Shader Program-------

        //Take a step back to previous coordinate

        vec2 prevCoordD = vec2(st - (coordVelocity.xy * dt));

        vec3 prevCoordDensity = texture2D(fPrev, prevCoordD.xy).xyz;

        fField = prevCoordDensity;

      //Take a step back to previous coordinate in the velocity field

        vec2 prevCoordV = vec2(st - (coordVelocity.xy * dt));

        //To make the velocity field move along itself
        //(self advection)

        vec3 prevCoordVelocity = texture2D(vPrev, prevCoordV.xy).xyz;

        xyz = vec3(prevCoordVelocity.x, prevCoordVelocity.y + buoyancy↩
            ,
        prevCoordVelocity.z);
```

Listing 3.1: A part of the advection program

In this code **v** represent the velocity field texture and **f** represent the density field texture. **xyz** is the field that is to be advected.

3.2.2 Diffusion

The second shader will determine how viscous the fluid is. This is done with Jacobi iterations. The accuracy is enhanced with every iteration of the shader. A good value would be 40 iterations. Less iterations was tested but the results was not as good. The Jacobi Iteration program is shown in listning 3.2

```
        -------Jacobi Iteration Program-------

    float viscosity = 0.000000000000000001; //Arbitrary number

    //left, right, bottom, and top samples
    vec3 coordVelL    = texture2D(vPrev, vec2((st.x - pixdist), st.y)).xyz;
    vec3 coordVelR    = texture2D(vPrev, vec2((st.x + pixdist), st.y)).xyz;
    vec3 coordVelB    = texture2D(vPrev, vec2(st.x, (st.y - pixdist))).xyz;
    vec3 coordVelT    = texture2D(vPrev, vec2(st.x, (st.y + pixdist))).xyz;

    //Constants
    float alpha = pow(pixdist, 2.0) / (viscosity * dt);
    float beta  = 4.0 + alpha;

    vec3 b = texture2D(vPrev, st.xy).xyz;

    //Estimate Jacobi iteration
    vec3 coordDiffuseVelocity = (coordVelL + coordVelR + coordVelB + coordVelT + (alpha * b)) / beta;
```
Listing 3.2: Jacobi iteration program

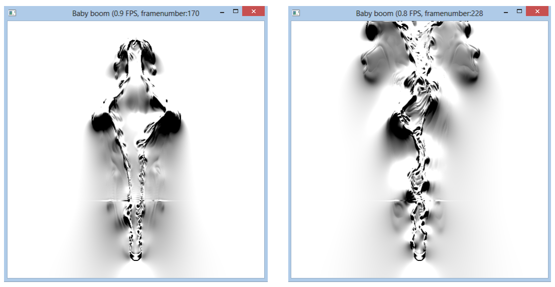
The value of the viscosity variable was arbitrarily chosen by testing different values to look like smoke from a candle. The constants alpha and beta are taken from section 2.8, used to solve the viscous diffusion equation 2.7.

3.2.3 Projection

The projection step is divided into three steps, hence three shaders.

3.2.3.1 Divergence shader

In the divergence-shader a divergence field is calculated from the velocity field, and saved to the divergence-texture.


Baby boom (0.9 FPS, framenumber:170)
Baby boom (0.8 FPS, framenumber:228)

(a) Timestep 1 (b) Timestep 2

Figure 3.3: Visualization of the divergence of the vector field

The black parts in figure 3.3 are sinks which will attract the smoke. The white positive divergence represents places the smoke is moving away from. The divergence shader program is shown in listning 3.3

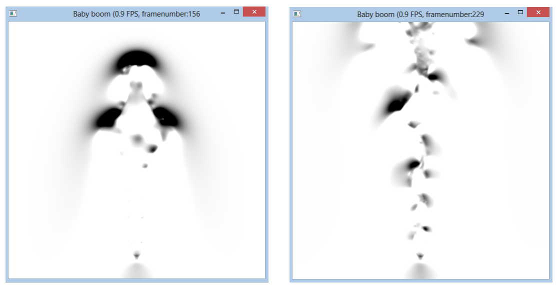
```
  ------The Divergence Fragment Program------

//left, right, bottom, and top samples
vec3 coordVelocityLeft = texture2D(vPrev, vec2((st.x - pixdist),
  st.y)).xyz;
vec3 coordVelocityRight = texture2D(vPrev, vec2((st.x + pixdist),
  st.y)).xyz;
vec3 coordVelocityBottom = texture2D(vPrev, vec2(st.x, (st.y -
  pixdist))).xyz;
vec3 coordVelocityTop = texture2D(vPrev, vec2(st.x, (st.y +
  pixdist))).xyz;

float temp = 0.5 / pixdist;

//calculate divergence
vec3 divergence = temp * vec3((coordVelocityRight.x - ↩
    coordVelocityLeft.x) +
(coordVelocityTop.y - coordVelocityBottom.y));
```

Listing 3.3: The divergence program

The divergence is written to the divergence texture and is later used as input to the **b** parameter of the Jacobi iteration program for poisson pressure equation 2.12.

3.2.3.2 Pressure shader

The pressure shader will create a new pressure field every loop of the shaders. This is possible by using the divergency field in a Jacobi iteration. At the same time the pressure shader itself is being iterated many times. After the first iteration the pressure field will not be especially good. Let it iterate about 80 times, and it will be a fairly good estimation.


Baby boom (0.9 FPS, framenumber:156)
Baby boom (0.9 FPS, framenumber:229)

(a) Timestep 1 (b) Timestep 2

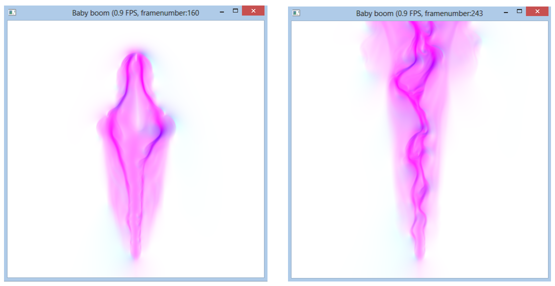
Figure 3.4: Visualization of the pressure in the vector field

The black parts in figure 3.4 symbolizes high pressure. The smoke is pushing upwards and creates a higher pressure in this area. Compare 3.4 with the divergence field in figure 3.3 and notice how they compliment eachother.

The jacobi iteration program for pressure is the same as for viscous diffusion apart from different constants and can be found in the apendix A.

3.2.3.3 Gradient shader

The shader calculates the gradient of the pressure field at a specific coordinate and then subtracts it from the velocity field. This means that the Navier-Stokes equations have been solved. The updated veolcity field will be used in the next shader-loop and advection.


Baby boom (0.9 FPS, framenumber:160)
Baby boom (0.9 FPS, framenumber:243)

(a) Timestep 1 (b) Timestep 2

Figure 3.5: Visualization of the velocity in the vector field

The pink parts in figure3.5 represent a velocity in the positive y-axis. The blue parts represent a positive force in the x-axis. And the purple parts are a mix of these two. The gradient subtraction program can be seen in listning 3.4.

```
  -------The Gradient Subtraction Program-------

//left, right, bottom, and top samples
vec3 coordPreassureLeft   = texture2D(pPrev, vec2((st.x - pixdist),
  st.y)).xyz;
vec3 coordPreassureRight  = texture2D(pPrev, vec2((st.x + pixdist),
  st.y)).xyz;
vec3 coordPreassureBottom = texture2D(pPrev, vec2(st.x, (st.y -
  pixdist))).xyz;
vec3 coordPreassureTop    = texture2D(pPrev, vec2(st.x, (st.y +
  pixdist))).xyz;

pixdist = 0.5 / pixdist ;

//Subtract gradient.
vFieldd -= vec3(pixdist * (coordPreassureRight.x - coordPreassureLeft↩
    .x), pixdist * (coordPreassureTop.y - coordPreassureBottom.y), ↩
    vField.z);
```

Listing 3.4: The gradient subtraction program

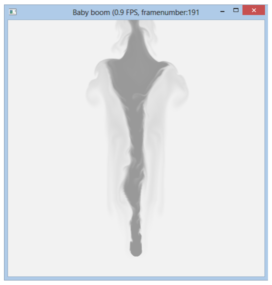
3.2.4 External forces

The only external forces implemented are buoyancy and a Gaussian splat force-source.

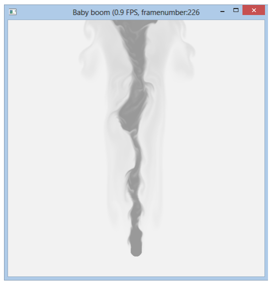
A Gaussian splat is basically a force-source, like the wind, which has a dircetion and a magnitude. This force is then added to the velocity-field. See [3] for formula.

These are both calculated in the advection shader, before the actual advection. So if it would have been a separate shader, it would have been placed before the advection-shader.

3.2.5 Final smoke



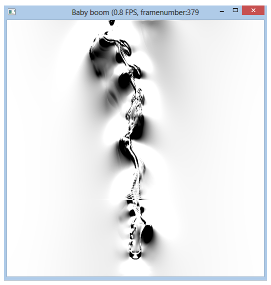
(a) Timestep 1



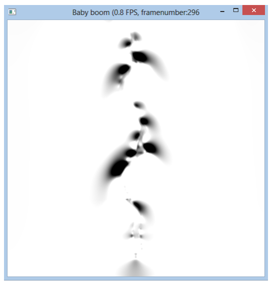
(b) Timestep 2

Figure 3.6: Final smoke

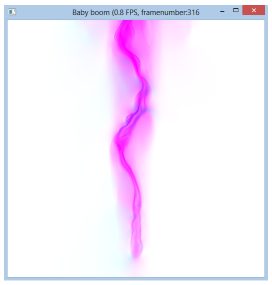
The final result can be seen in fig 3.6. A summary can be seen in fig 3.7.



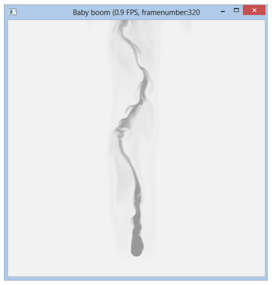
(a) Divergence field



(b) Pressure field



(c) Velocity field



(d) Final smoke

Figure 3.7: A summary of the different vector fields creating the final smoke

Chapter 4

Results

4.1 Speed

One of the benefits of utilizing the GPU is the speed gain compared to a similar execution on the CPU.

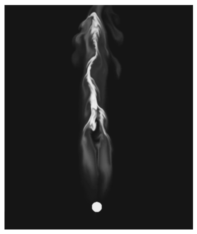
Table 4.1: Render speed

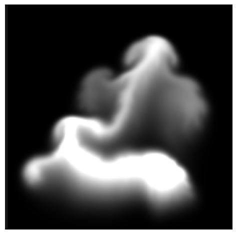
Resolution	Number of slices	FPS
128x128	3	5.0
256x256	3	1.2
512x512	3	0.3
1024x1024	3	0.1
512x512	1	0.9
512x512	2	0.5
512x512	3	0.3
512x512	4	0.2
512x512	5	0.2

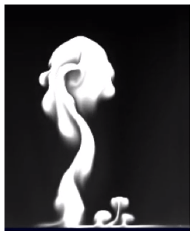
Table 4.1 shows performance with different resolutions and number of slices. The test was made with 40 diffuse and 80 pressure iterations per frame.

4.2 Visual appeal

A snapshot of the simulation is shown in figure 4.1. Our results are quite a long way off the renderings in *"Smoke Simulation For Large Scale Phenomena"*. This of course due to first and foremost being two-dimensional and not as much work having been put into colouring and fall-off gradients. Being three-dimensional also allows for ray-tracing of light which adds a lot of visual appeal. When compared to other two-dimensional simulations however our results measure up quite well, see figure 4.1.


(a) Our simulation


(b) Stable Fluids Jos Stam [8]. [10]


(c) GPU Gems I. Ch.38 [9]. [11]

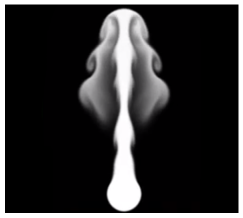

(d) Master thesis work [12]

Figure 4.1: A comparison between our results and similiar projects based on different methods.

4.3 Offline rendering

When using large textures and/or several slices, the FPS dropped significantly so a function to save the frames to PNG files was implemented, and these images were concatenated into a movie.

Chapter 5

Discussion

We made a working two-dimensional smoke simulation, three-dimensionality was unfortunately not fully achieved. The layer-based method used in *"Smoke Simulation For Large Scale Phenomena"* by Rasmussen et al.2003 [7] caused issues by not being very well documented. So while a fluid could be simulated simultaneously on several layers there is no interaction between them in the form of, for example, advection and diffusion. It might have been easier to use a three-dimensional texture and calculate the fluid as a volume but the described layer method had large performance benefits.

While the mathematical concepts are quite advanced and hard to grasp the actual implementation were often suprisingly simple. An example is the Jacobi iterator which solves the constant pressure equations. Its approximation which converges on an acceptable value replaces the theoretical solution which includes solving a matrix equation for a matrix the size of $\mathbf{n^2} \times \mathbf{n^2}$ where n is the size of the fluid grid. A grid of $\mathbf{512} \times \mathbf{512}$ would for example require the solving of $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ would be a matrix with $\mathbf{2^{36}} \approx \mathbf{6.8} * \mathbf{10^{10}}$ elements.

Chapter 6

Future work

6.1 Vorticity confinement

Vorticity confinement is a method for maintaining small swhirl-like motions in low-viscosity fluids that often dissapear due to grid cells being too large to accomodate these small rotational movements. Mathematically it is not a very advanced step in the process to implement and would most likely improve the quality of the smoke simulation.

6.2 3D implementation

To use the method with multiple slices instead of a continuous third dimension of the data as described in *"Smoke Simulation For Large Scale Phenomena"* by Rasmussen et al. 2003 [7] would take further research into how the implementation is done with respect to transferral of values between the layers. We have set up the structure for using layers and can run simulations with several layers, however they do not interact with each other.

6.3 Dynamic ray marching

In order to show the 3D-texture that contains the slices, a primitive ray marching was implemented. It marches from the front of the cube to the back and returns the largest value it encounters.

This is a special case of when looking straight at the front of the cube and thus the 3D is a bit redundant. The ray marcher could very well be extended to properly handle pan-tilt-zoom navigation.

6.4 Kolmogorov spectrum

The interpolation between the slices is currently linear, *"Smoke Simulation For Large Scale Phenomena"* by Rasmussen et al. 2003 [7] presents an added noise to this interpolation. The noise specified in the report uses the Kolmogorov spectrum as a means to reduce the visibility of artifacts caused by the linear interpolation.

The Kolmogorov spectrum is poorly documented and we therefore did not implement it and settled for linear interpolation. Future work could include researching the Kolmogorov spectrum further and implement if possible or finding other noise methods that could be applicable.

Bibliography

[1] Bridson, R. 2008.
Fluid Simulation for Computer Graphics
September 18, 2008 by A K Peters/CRC Press - 246 Pages

[2] Fedkiw, R., J. Stam, and H.W. Jensen. 2001.
Visual Simulation of Smoke.
In Proceedings of SIGGRAPH 2001.

[3] Harris, M. 2004
Fast Fluid Dynamics Simulation on the GPU.
`http.developer.nvidia.com/GPUGems/gpugems_ch38.html`

[4] L.Yaeger and C.Upson. 1986.
Combining Physical and Visual Simulation.
Creation of the Planet Jupiter for the Film 2010.
ACM Computer Graphics (SIGGRAPH 86), 20(4):85-93, August 1986.

[5] N. Foster and D. Metaxas. 1996.
Realistic Animation of Liquids.
Graphical Models and Image Processing, 58(5):471-483, 1996.

[6] N. Foster and D. Metaxas. 1997
Modeling the Motion of a Hot, Turbulent Gas.
In Computer Graphics Proceedings, Annual Conference Series, 1997, pages 181-188, August 1997.

[7] Rasmussen, N., Nguyen, D. Q., Geiger, W., and Fedkiw,R. 2003.
Smoke Simulation For Large Scale Phenomena.
In Proceedings of ACM SIGGRAPH 2003, vol. 22, 703707

[8] Stam, J. 1999.
Stable Fluids.
In Proceedings of SIGGRAPH 1999.

[9] Mark J. Harris. 2004.
GPU Gems
Pearson Education, Inc.

[10] http://users.csc.calpoly.edu/ zwood/teaching/csc572/final08/khaughey/ *CSC 572 Final Project - GPU Smoke Simulation*
Kyle Haughey

[11] http://www.youtube.com/watch?v=nK12L2TLO9g *2D Smoke using Fluid Simulation*

[12] http://www.youtube.com/watch?v=zfz9RbhXFyA *2D Smoke Simulation*
Vignesh Kumar Ramalingam

Appendix A

Shader code

```

uniform int frame;
uniform int winWidth;
uniform int winHeight;
uniform float depth;
uniform sampler2D vPrev;
uniform sampler2D vCurr; //is gl_Fragdata[0]
uniform sampler2D fPrev;
uniform sampler2D fCurr; //is gl_Fragdata[1]
varying vec2 st;
varying vec3 stp;

float dt = 1.0/30.0;
float pixdist = 1.0/ float(winWidth);


void main( void )
{
  vec3 fField = texture2D(fPrev, st).xyz; //Density
  vec3 vField = texture2D(vPrev, st).xyz; //Velocity
  float time = float(frame)*dt;


  vec3 xyz = vec3(0.0f, 0.0f, depth);
  vec3 coordVelocity = texture2D(vPrev, st).xyz;

  //Initial values
  if (frame < 2)
  {
    //Initiate density
    if ( pow(st.x - 0.5f, 2.0) + pow(st.y - 0.5f, 2.0) + pow(depth - ↩
        0.5, 2.0)  < 0.01)
    {
      fField = vec3(0.0f);
    }else
      {
        fField = vec3(0.05f);
      }

    // nitiate velocity field to 0
    xyz = vec3(0.0);

  }else
  {

    //Create the different forces ---------------
    //Gaussian splat (a force-source, like the wind, which has a dircetion and a magnitud)
    float radius_x = 0.3;
    float radius_y = 0.3;
    float r = 1.0 + depth;
    float F = abs(cos(time)) + 1.0;

    float c = (exp(-( (pow(st.x-radius_x, 2.0) + pow(st.y-radius_y, 2.0)) / pow(r,2.0))));
    c = abs(c * dt * F);

    //Gravity
    float gravity = 9.82 * dt;


    //buoyancy - could be a texture with temperature.
    vec2 temp = texture2D(fPrev, vec2((st.x), (st.y - pixdist * 5.0))).xy;
    vec2 temp2 = texture2D(fPrev, vec2((st.x), (st.y + pixdist * 5.0))).xy;
    float diffval = (abs((temp.x + temp.y) - (temp2.x + temp2.y)));

    float buoyancy = 0.7 * dt * diffval;

    //Add the forces ------------

    //Gravity
    //coordVelocity.y = coordVelocity.y - gravity;

    //Gaussian splat
    if(1 > 0) // on/off
    {
      //"force-pointsouce" shooting away from source
      if((st.x - radius_x) < 0.0)
      {
        //coordVelocity.x = coordVelocity.x - c;
      }else
      {
        //coordVelocity.x = coordVelocity.x + c;
      }
      if((st.y - radius_y) < 0.0)
      {
        coordVelocity.y = coordVelocity.y - c;
      }else
      {
        coordVelocity.y = coordVelocity.y + c;
      }
    }

    //Advection and buoyancy -------------
    //Take a step back to previous coordinate
    vec2 prevCoordD = vec2(st - (coordVelocity.xy * dt));
    vec3 prevCoordDensity = texture2D(fPrev, prevCoordD.xy).xyz;
    fField = prevCoordDensity;

    //Take a step back to previous coordinate in the velocity field
    vec2 prevCoordV = vec2(st - (coordVelocity.xy * dt));
    //To make the velocity field move along itself (self-advection)
    vec3 prevCoordVelocity = texture2D(vPrev, prevCoordV.xy).xyz;
    xyz = vec3(prevCoordVelocity.x, prevCoordVelocity.y + buoyancy, prevCoordVelocity.z);

    float moveY = 0.1;
    float moveX = 0.5;
    if(sin(time) * cos(time) > -2.0)
    {
        //The radius of the density circle
        if ( (st.x - moveX)*(st.x - moveX) + (st.y - moveY)*(st.y - ↩
            moveY) < 0.0005)
        {
          //Density value. Higher value creates increased buoyancy
          fField = vec3(0.4f);
        }
      }

    }
    //Save information to the textures
    gl_FragData[0] = vec4(xyz, 1.0);
    gl_FragData[1] = vec4(fField, 1.0);
}
```

Listing A.1: Advection shader

```

uniform int frame;
uniform int winWidth;
uniform int winHeight;
uniform sampler2D vPrev;
uniform sampler2D vCurr; //is gl_Fragdata[0]
uniform sampler2D fPrev;
uniform sampler2D fCurr; //is gl_Fragdata[1]
varying vec2 st;
varying vec3 stp;

float dt = 1.0/30.0;
float pixdist = 1.0f/(float(winWidth));

void main( void )
{
  vec3 fField = texture2D(fPrev, st).xyz;
  vec3 vField = texture2D(vPrev, st).xyz;

  vec3 coordVelocity = texture2D(vPrev, st).xyz;

  vec2 prevCoord = vec2(st - (coordVelocity.xy * dt));

    float viscosity = 0.000000000000000001; //Arbitrary number

    //jacobi iteration
    vec3 coordVelL    = texture2D(vPrev, vec2((st.x - pixdist), st.y))↩
        .xyz;
    vec3 coordVelR    = texture2D(vPrev, vec2((st.x + pixdist), st.y))↩
        .xyz;
    vec3 coordVelB    = texture2D(vPrev, vec2(st.x, (st.y - pixdist)))↩
        .xyz;
    vec3 coordVelT    = texture2D(vPrev, vec2(st.x, (st.y + pixdist)))↩
        .xyz;

    float alpha = pow(pixdist, 2.0) / (viscosity * dt);
    float beta  = 4.0 + alpha;

    vec3 b = texture2D(vPrev, st.xy).xyz;

    vec3 coordDiffuseVelocity = (coordVelL + coordVelR + coordVelB + ↩
        coordVelT + (alpha * b)) / beta;

  gl_FragData[0] = vec4(coordDiffuseVelocity, 1.0);
  gl_FragData[1] = vec4(fField, 1.0);
}
```

Listing A.2: Diffusion shader

```
uniform int frame;
uniform int winWidth;
uniform int winHeight;
uniform sampler2D vPrev;
uniform sampler2D vCurr; //is gl_Fragdata[0]
uniform sampler2D fPrev;
uniform sampler2D fCurr; //is gl_Fragdata[1]
uniform sampler2D dCurr;
uniform sampler2D dPrev; //is gl_Fragdata[2]
uniform sampler2D pPrev;
uniform sampler2D pCurr;
varying vec2 st;
varying vec3 stp;

float dt = 0.0166666666f;
float pixdist = 1.0f/float(winWidth);

void main( void )
{
  vec3 vField = texture2D(vPrev, st).xyz;
  vec3 fField = texture2D(fPrev, st).xyz;

  //Calculate divergence
  vec3 coordVelocityLeft    = texture2D(vPrev, vec2((st.x - pixdist), ↩
      st.y)).xyz;
  vec3 coordVelocityRight   = texture2D(vPrev, vec2((st.x + pixdist), ↩
      st.y)).xyz;
  vec3 coordVelocityBottom  = texture2D(vPrev, vec2(st.x, (st.y - ↩
      pixdist))).xyz;
  vec3 coordVelocityTop     = texture2D(vPrev, vec2(st.x, (st.y + ↩
      pixdist))).xyz;

  float temp = 0.5 / pixdist;
  vec3 divergence = temp * vec3((coordVelocityRight.x - ↩
      coordVelocityLeft.x) +
              (coordVelocityTop.y - coordVelocityBottom.y));


  gl_FragData[0] = vec4(vField , 1.0f);
  gl_FragData[1] = vec4(fField , 1.0f);
  gl_FragData[2] = vec4(divergence , 1.0);
  gl_FragData[3] = vec4( vec3(0.0) , 1.0); //Set the pressure-texture ↩
      to 0 before the iteration.
}
```

Listing A.3: Divergence shader

```
uniform int frame;
uniform int winWidth;
uniform int winHeight;
uniform sampler2D vPrev;
uniform sampler2D vCurr; //is gl_Fragdata[0]
uniform sampler2D fPrev;
uniform sampler2D fCurr; //is gl_Fragdata[1]
uniform sampler2D dCurr;
uniform sampler2D dPrev; //is gl_Fragdata[2]
uniform sampler2D pPrev;
uniform sampler2D pCurr; //is gl_Fragdata[3]
varying vec2 st;
varying vec3 stp;

float dt = 1.0/30.0;
float pixdist = 1.0/float(winWidth);

```

```

void main( void )
{
  vec3 vField = texture2D(vPrev, st).xyz;
  vec3 fField = texture2D(fPrev, st).xyz;
  vec3 dField = texture2D(dPrev, st).xyz;
  vec3 pField = texture2D(pPrev, st).xyz;

    //jacobi iteration.
    vec3 coordVelocityLeft    = texture2D(pPrev, vec2((st.x - pixdist)↩
        , st.y)).xyz;
    vec3 coordVelocityRight   = texture2D(pPrev, vec2((st.x + pixdist)↩
        , st.y)).xyz;
    vec3 coordVelocityBottom  = texture2D(pPrev, vec2(st.x, (st.y - ↩
        pixdist))).xyz;
    vec3 coordVelocityTop     = texture2D(pPrev, vec2(st.x, (st.y + ↩
        pixdist))).xyz;

    float alpha = -pow(pixdist, 2.0);
    float beta  = 4.0;
    float b = texture2D(dPrev, st.xy).x;

    //Pressure field
    pField = (coordVelocityLeft + coordVelocityRight + ↩
        coordVelocityBottom + coordVelocityTop + (alpha * b)) / beta;

  gl_FragData[0] = vec4(vField , 1.0);
  gl_FragData[1] = vec4(fField , 1.0);
  gl_FragData[2] = vec4(dField , 1.0);
  gl_FragData[3] = vec4(pField , 1.0);

}
```

Listing A.4: Jacobi iteration shader (pressure field)

```
uniform int frame;
uniform int winWidth;
uniform int winHeight;
uniform sampler2D vPrev;
uniform sampler2D vCurr; //is gl_Fragdata[0]
uniform sampler2D fPrev;
uniform sampler2D fCurr; //is gl_Fragdata[1]
uniform sampler2D dCurr;
uniform sampler2D dPrev; //is gl_Fragdata[2]
uniform sampler2D pPrev;
uniform sampler2D pCurr; //is gl_Fragdata[3]
varying vec2 st;
varying vec3 stp;

float dt = 1.0/30.0;
float pixdist = 1.0/float(winWidth);


void main( void )
{
  vec3 vField = texture2D(vPrev, st).xyz;
  vec3 fField = texture2D(fPrev, st).xyz;
  vec3 dField = texture2D(dPrev, st).xyz;
  vec3 pField = texture2D(pPrev, st).xyz;

    vec3 coordPreassureLeft   = texture2D(pPrev, vec2((st.x - pixdist)↩
        , st.y)).xyz;
  vec3 coordPreassureRight  = texture2D(pPrev, vec2((st.x + pixdist), ↩
      st.y)).xyz;
  vec3 coordPreassureBottom = texture2D(pPrev, vec2(st.x, (st.y - ↩
      pixdist))).xyz;
  vec3  coordPreassureTop     = texture2D(pPrev, vec2(st.x, (st.y + ↩
      pixdist))).xyz;

  pixdist = 0.5 / pixdist ;
  vField -= vec3(pixdist * (coordPreassureRight.x - coordPreassureLeft↩
      .x),
             pixdist * (coordPreassureTop.y - coordPreassureBottom.y), ↩
                 vField.z);

  gl_FragData[0] = vec4(vField , 1.0f);
  gl_FragData[1] = vec4(fField , 1.0f);
  gl_FragData[2] = vec4(dField , 1.0f);
  gl_FragData[3] = vec4(pField , 1.0f);

}
```
Listing A.5: Gradient shader