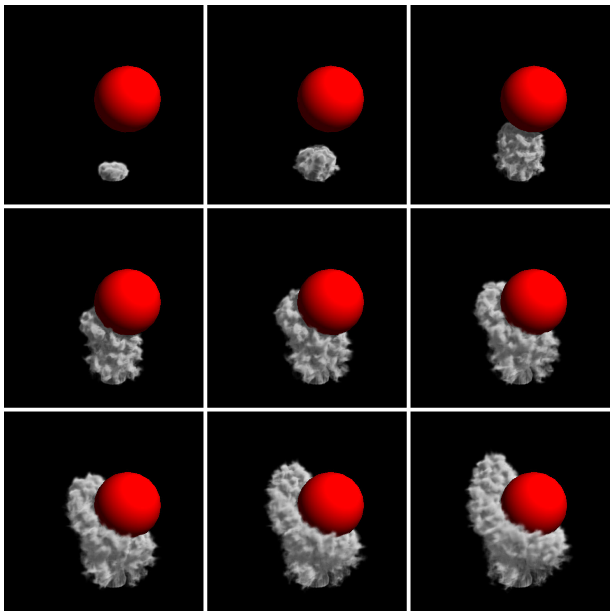
Real-time fluid simulations with wavelet turbulence

Real-time Fluid Simulations with Wavelet Turbulence



Basil Fierz

Master's Thesis
September 2008

Prof. Dr. Markus Gross
Dr. Nils Thürey

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Computer Graphics Laboratory ETH Zurich

Abstract

This thesis discusses the realisation of real-time interactive fluid simulations with Wavelet Turbulence. The Wavelet Turbulence approach uses Kolmogorov's power distribution to synthesise turbulence in fluid simulations, and separates the simulation of large and small scale details.

The large scale flow is simulated using a conventional fluid solver. The relevant fluid behaviour is captured on a coarse grid. The calculations for this coarse simulation can be done with consumer level CPUs. To synthesise interesting small scale details, we scale the coarse grid to a higher resolution grid by adding Wavelet Turbulence. For this post-processing step we use NVIDIA's CUDA system to achieve real-time performance.

We first give an overview of the theory our implementation is based on. Next, we develop different implementation approaches for Wavelet Turbulence and evaluate their performance. An outlook to extensions of our implementation concludes the thesis.

Zusammenfassung

Diese Diplomarbeit beschreibt die Realisierung einer interaktiven Fluidsimulation mit Wavelet Turbulenzen. Die Methode der Wavelet Turbulenzen verwendet Kolmogorovs Energieverteilung um Turbulenzen in Fluidsimulationen zu synthetisieren. Sie trennt die Simulation der gross- und kleinskaliger Details.

Die grossskaligen Details werden mit Hilfe üblicher Fluidsolver simuliert. Um das relevante Fluidverhalten zu beschreiben reicht ein grobes Gitter, welches auf handelsüblichen Prozessoren simuliert werden kann. Um die interessanten kleinskaligen Details zu synthetisieren skalieren wir das grobe Gitter auf ein höheraufgelöstes Gitter durch hinzufügen von Wavelet Turbulenzen hoch. Für diesen Nachbearbeitungsschritt verwenden wir NVIDIAs CUDA System, um ein Echtzeitverhalten zu erreichen.

Wir geben zuerst einen Überblick über die Theorie, auf welcher unsere Umsetzung basiert. Als nächstes verfolgen wir verschiedene Implementierungsansätze für Wavelet Turbulenzen und beurteilen deren Laufzeit verhalten. Wir schliessen diese Diplomarbeit mit einem Ausblick auf mögliche Erweiterungen ab.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Computer Graphics Laboratory ETH Zurich

Master Thesis

Real-time Fluid Simulations with Wavelet Turbulence

Topic:

As physical simulations are a crucial part of many modern games, it is important to handle all necessary effects within the engines. Fluids are necessary for a variety of effects, such as smoke, fire and explosions, but are highly costly to compute with commonly used methods. Typically, the expensive computations for the iterative system to be solved for the pressure correction step, and the high memory requirements prevent interesting real-time applications.

The goal of this thesis is to make use of the wavelet turbulence approach to compute detailed fluid motions in real-time. To run the underlying coarse fluid simulation, the CPU is sufficient. The wavelet turbulence will be evaluated using a GPU with CUDA. The nature of the wavelet turbulence approach make it very suitable for a parallel computation using the GPU, and a good performance is expected. The first task of this thesis is to port the wavelet turbulence to the GPU and adapt it to the GPU architecture. As a second step, a renderer for smoke and possibly fire effects, that computes lighted volumetric effects for a realistic visual appearance, will be developed. While both steps can first be performed on fixed grids, they can also be performed using particles for the detailed advection, which will reduce the required memory. The overall goal is to develop an interactive program that can simulate realistic smoke and fire effects, and allows the user to place and move objects in the flow.

Remarks:

- A written report and an oral presentation conclude the work.
- The thesis is overseen by Prof. Markus Gross and is supervised by Nils Thuerey.

Contents

List of Figures

Listings

1 Introduction

Ever since the advent of graphics co-processors (GPU) for consumer level hardware, people tried to use these computing capacities for other things than rendering triangles. GPUs are parallel architectures, mostly because of the nature of the problem they try to solve – rendering huge amounts of independent elements (triangles, fragments). GPUs had and still have the advantage that their price is surprisingly low compared to the amount of computing power they provide.

The first generations of GPUs did not offer many possibilities. The programmer had to use the fixed-function graphics pipeline to do calculations and the available amount of device memory limited the problem set, which could be solved. The first systems, which could be called programmable in a modern sense, appeared in late 2000 with the introduction of Microsoft DirectX 8.0. The hardwired vertex and pixel processors where replaced with programmable shaders. This allowed computer game developers to port vertex operations to the GPU, which had to be done on the host processor in the past. It also allowed for custom shading models, which were not possible with the fixed function pipeline.

Although the computing capabilities were still quite primitive and the floating point number precision was very low, game developers adopted the new programmable pipeline fast. Over the next two generations of GPUs the capabilities were expanded. The computing power increased and the precision improved. GPUs were used more and more for non graphical applications.

With the latest GPU generation NVIDIA provided, in addition to the graphics API (OpenGL, DirectX), an alternative access path to the resources of their GPUs. Their CUDA system provides direct access to the GPU without the need to adhere to the structure of the 3D rendering pipeline. For a more complete overview of the history of GPU Computing, we recommend [OHL$^+$08].

Simulation of fluids is a long time topic in visual computing. After Joe Stam ([Sta99]) introduced grid based fluid simulations to computer graphics, many advances have been made. The first simulations were far away from being real-time and details dissipated fast due to numerical errors. Mostly these errors come from the Semi-Lagrange advection method, which was used because it is cheap and reasonably correct. Different algorithms were developed to counter this dissipation effect: Vorticity confinement [FSJ01], Vortex Particles [SRF05] and MacCormack advection [SFK^{+}07].

The best real-time implementations to date to simulate 3D fluids in real-time are [CLT08] using standard graphic API and [MCPN08] using a CUDA based multigrid Poisson solver. The main problems are the high memory requirements and the need to solve large systems of equations for high resolution simulations, which make real-time simulations using high resolution grids very hard. Adding more details adds memory and computing effort with cubic complexity.

In this thesis, we take the approach to separate detail generation from the physically based simulation. Often we do not need an overly realistic simulation but are content with the behaviour of a simple simulation. On the other hand the simpler the simulation is, the less appealing it looks. Our goal is to show that the separation method of [KTJG08] is usable under real-time constraints. The main tool to achieve this goal is NVIDIAs CUDA technology.

2

Wavelet Noise

The fluid simulation method presented in [KTJG08] separates the simulation of small and large scale details. The large scale details are obtained using a standard simulation systems like the one introduced by [Sta99]. The small scale details are added in a post processing step. This additional step modifies the velocity field for a more detailed visualisation, but leaves the fluid simulation itself untouched. The modifications to the velocity field depend only on the fluid simulation and are smooth over time.

2.1. Fluid Simulation

Fluid Simulations in real-time computer visualisation are mostly based on the Navier-Stokes equations for incompressible fluids. These are a set of equations that are valid on the whole simulation domain. They describe the local velocity $\vec{u}$ as a function of position and time. The first equation, Equation 2.1, is also called “momentum equation” and describes how the fluid reacts to the forces acting on the it:

$$\frac{\partial \vec{u}}{\partial t} = -\vec{u} \cdot \nabla \vec{u} - \frac{1}{\rho} \nabla p + \vec{f}, \tag{2.1}$$

where p is the pressure, ρ is the mass density, and $\vec{f}$ represents external forces. The second equation, Equation 2.2, states that the fluid we consider has to be incompressible:

$$\nabla \cdot \vec{u} = 0 \tag{2.2}$$

The complete Navier-Stokes equations contain an additional term to describe the viscosity of the fluid. It is typically ignored for low viscous fluid behaviour, which we will use here. The reason for this is very simple. Numerical simulations using floating point arithmetic typically introduce rounding errors, which are similar to small viscosity values. For a complete derivation of the equations in this section see [BMF07] as very good introduction.

Simulating fluids means nothing else than keeping the equations valid over time. Solving these partial differential equations requires us to discretise the simulation domain. We choose a pure Eulerian approach, meaning that all simulation quantities are fixed to a distinct position in space. The simulation domain is discretised as a regular volumetric staggered grid. Figure 2.1 shows one cell of the such a grid. Each of these cells store scalar and vector quantities (such as pressure, temperature, density values and velocities).

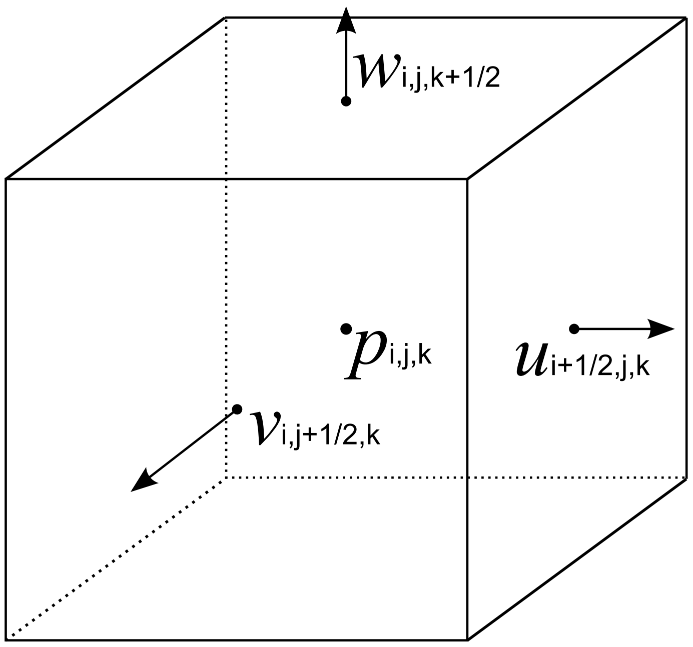


Figure 2.1.: *One cell from a three dimensional staggered grid.*

The staggered grid stores the scalar quantities (pressure, temperature and density) in the centre of the cell, while the velocity is stored at the centre of the faces of a grid cell. The velocity $\vec{u} = [u_{i,j,k}, v_{i,j,k}, w_{i,j,k}]$ is decomposed and each component is stored in its own face. The top cell stores $w_{i,j,k+1/2}$, the front cell stores $v_{i,j+1/2,k}$ and the right face $u_{i+1/2,j,k}$. The components of the velocity at a grid node itself are the averages of the values of the six opposite faces. This makes it easier to calculate the spatial derivatives we need.

To solve Equation 2.1 we apply a common trick. We split the equation into parts and solve them sequentially.

1. Add the forces $\vec{f}$ to the velocities $\vec{u}$.
2. Make the velocity field incompressible, i.e. solve $\frac{\partial \vec{u}}{\partial t} + \frac{1}{\rho}\nabla p$ s.t. $\nabla \cdot \vec{u} = 0$.

3. Apply the advection operator A (see Section 2.3) to the velocity field $\vec{u}$.

2.1.1. Boundary Conditions

Correctly handling boundary conditions is tricky if we do not restrict the type of boundary. The simplest case of obstacles are "solid walls", where the fluid is not supposed to pass through. Thus the velocity of the fluid has to be zero with respect to the normal of the boundary:

$$\vec{u} \cdot \hat{n} = 0 \tag{2.3}$$

If the solid is not static but moving, the fluid has to move in the same direction as the solid:

$$\vec{u} \cdot \hat{n} = \vec{u}_{solid} \cdot \hat{n} \tag{2.4}$$

The simplest boundary condition we can think of is called a "no slip" boundary, where the fluids velocity at the boundary is $\vec{u} = 0$ for static boundaries, and $\vec{u} = \vec{u}_{solid}$ for moving obstacles. These boundary conditions have to be treated as additional constraints when making the velocity field incompressible.

2.1.2. Making The Fluid Incompressible

The most expensive step in the simulation is keeping the fluid incompressible. Here is also a convenient place to resolve the boundary conditions. Discretising the pressure equation with respect to time and solving for the next time step's velocity field $\vec{u}$, we get:

$$\vec{u}^{n+1} = \vec{u}^{n} - \Delta t \frac{1}{\rho} \nabla p \tag{2.5}$$

We try to solve Equation 2.5 in a way that it satisfies the incompressibility:

$$\nabla \cdot \vec{u}^{n+1} = 0 \tag{2.6}$$

and the solid wall boundary condition:

$$\vec{u}^{n+1} \cdot \hat{n} = \vec{u}_{solid} \cdot \hat{n} \tag{2.7}$$

The resulting system of equations leads to a sparse matrix, with a bandwidth of 7, and can be solved with any appropriate solver e.g., a Conjugate Gradients solver.

2.1.3. Smoke

To test the algorithm we implemented a smoke simulation. Smoke like behaviour is achieved by simulating smoke quantities and the surrounding air. Rising smoke is lighter than the surrounding air, which we will capture through a buoyanesc external force. Buoyancy is a force that acts on mass that is less dense than its environment. The lighter material rises upwards, which holds for smoke particles. The buoyancy is controlled by a diffusion algorithm, which simulates ambient temperature.

To counter the phenomena of vortices dissipating to fast [FSJ01] introduced the vorticity confinement force. Dissipation of details is a side effect of applying the advection operator to the simulation quantities. Vorticity confinement strengthens existing vortices by creating a force field, which tries to counter this effect.

Vorticity is defined by

$$\vec{\omega} = \nabla \times \vec{u}. \tag{2.8}$$

To strengthen a vortex we need the unit vector pointing to the center of the vortex by normalising the gradient of $|\vec{\omega}|$:

$$\vec{N} = \frac{\nabla|\vec{\omega}|}{||\nabla|\vec{\omega}|||}. \tag{2.9}$$

The vorticity confinement force is defined as:

$$\vec{f}_{conf} = \epsilon \Delta x (\vec{N} \times \vec{\omega}), \tag{2.10}$$

where ϵ is a parameter to control the strength of the confinement and Δx is the grid spacing. The vorticty confinement force will not generate new eddies. Adding details is discussed later in Section 2.2.4.

2.2. Wavelet Turbulence

The Wavelet Turbulence algorithm is based on procedural noise to generate interesting details. We use Wavelet Noise [CD05] instead of Perlin Noise [Per85] because its properties fit our needs better.

2.2.1. Wavelet Noise

The main properties, we are interested in, are incompressibility, easy generation of derivatives and band-limitation, which are the main advantages over Perlin Noise. The goal is to create a noise band N from a random image R:

1. Create an image R filled with random noise.
2. Downsample R to create a half size image $R\downarrow$.
3. Upsample $R\uparrow$ to a full size image $R\downarrow\uparrow$.
4. Subtract $R\downarrow\uparrow$ from the original R to create N.

N contains only the band-limited parts from R. It is used in the same way as Perlin Noise.

Noise Evaluation

To evaluate the noise we use a weighted sum. We use the uniform quadratic B-spline basis function B to reconstruct the noise value:

$$w(x_i) = \sum_{q=0}^{2} B(x_{i-1+q})N(x_{i-1+q}) \tag{2.11}$$

To keep the runtime costs of the evaluation low [CD05] use a small tile of pre-generated noise instead of evaluating N each time.

2.2.2. Vector Wavelet Noise

For the next step of the algorithm the scalar Wavelet Noise function w has to be extended to a vector-valued function. [BHN07] show how to derive a divergence-free vector field from a scalar field by taking curl of a vector-valued potential field (w_1, w_2, w_3):

$$\vec{\phi} = \left(\frac{\partial w_1}{\partial y} - \frac{\partial w_2}{\partial z}, \frac{\partial w_3}{\partial z} - \frac{\partial w_1}{\partial x}, \frac{\partial w_2}{\partial x} - \frac{\partial w_3}{\partial y} \right) \tag{2.12}$$

This requires us to use either three different noise tiles. To evaluate the derivatives we directly take the partial derivatives of the noise function.

$$\frac{\partial w(x)}{\partial x} = \sum_{i=0}^{2} \frac{\partial w_i(x)}{\partial x} \cdot f_{x-\lfloor x \rfloor + i} \tag{2.13}$$

The corresponding weights are $\left[\frac{t^2}{2}, \frac{1}{2} + t - t^2, \frac{1}{2} - t + \frac{t^2}{2}\right]$ and $[t, 1-2t, t-1]$ for the derivatives.

2.2.3. Turbulence

Based on Kolmogorov's energy distribution [KTJG08] developed a turbulence function:

$$\vec{w}_t(\vec{x}) = \sum_{i=i_{min}}^{i_{max}} \vec{w}(2^i \vec{x}) 2^{-\frac{5}{6}(i-i_{min})} \tag{2.14}$$

Through i_{min} and i_{max} the spectral bands which apply to $\vec{w}_t(\vec{x})$ can be controlled. With the exception of the noise function $\vec{w}(\vec{x})$ the turbulence function is the same as the one defined in [Per85].

2.2.4. Velocity Noise Generation

In fluid dynamics two things can happen to eddies. They can break up to eddies half of their original size (*forward scatter*) or they can merge together to larger eddies (*backward scatter*). As eddies break up they move into a higher frequency domain. The fluid simulation we looked at in Section 2.1 produces a velocity field $\vec{u}$, with the low resolution n^3. The eddies in $\vec{u}$ can only be scattered to the theoretical Nyquist limit of $\frac{n}{2}$. With vorticity confinement we have a method to prevent eddies to dissipate before reaching the limit. This section presents the algorithm by [KTJG08], which allows eddies to be scattered to higher frequencies than the limit frequency of the simulation grid.

From the velocity field $\vec{u}$ we generate a high resolution velocity field $\vec{U}$ of size N^3. $\vec{U}$ is used to advect the density field, which will be visualised later on. To generate $\vec{U}$ we use the function $I(\vec{u}, \vec{X})$, which interpolates the velocity in $\vec{U}$ at the position $\vec{X}$ from the low resolution field $\vec{u}$:

$$\vec{U}(\vec{X}) = I(\vec{u}, \vec{X}) \tag{2.15}$$

$\vec{U}$ is a smoothed out version of $\vec{u}$, but it does not contain new eddies. What we are interested in is to produce new eddies in the spectral bands $\left[n, \frac{N}{2}\right]$. We use a simplified algorithm, which omits the texture coordinate advection.

To generate eddies we add turbulence to the velocity field. We are only interested in details in the newly created bands $\left[n, \frac{N}{2}\right]$. Thus we set $i_{min} = \log_2 n$ and $i_{max} = \log_2 \frac{N}{2}$:

$$\vec{U}(\vec{X}) = I(\vec{u}, \vec{X}) + \vec{w}_t(\vec{X}) \tag{2.16}$$

The problem using this method is that eddies are created where we do not expect them to occur, even if we use the total energy e_t of $\vec{u}$ at the band '$\frac{n}{2}$' as a weight. We only want to create new eddies where an eddy in the spectral band $\frac{n}{2}$ forward scatters to higher frequencies.

To find the correct places to add the eddies, we calculate the kinematic energy of the velocity field:

$$e(\vec{x}) = \frac{1}{2} \left|\vec{u}(\vec{x})\right|^2 \tag{2.17}$$

We use the spectral component $\hat{e}\left(\vec{x}, \frac{n}{2}\right)$ of the energy field e as a weight. This spatial information is obtained by a Wavelet decomposition of the energy of $\vec{u}$ and ensures that no unwanted eddies are created. At obstacle boundaries special care has to be taken. The velocities in $\vec{u}$ are set zero if the cell is marked as obstacle. This discontinuity causes jumps in the Wavelet decomposition of the energy. To smooth the effect out the energy is extrapolated into the obstacles. The final function to create the high resolution velocity field is:

$$\vec{U}(\vec{X}) = I(\vec{u}, \vec{X}) + 2^{-\frac{5}{6}} I\left(\hat{e}\left(\vec{x}, \frac{n}{2}\right), \vec{X}\right) \vec{w}_t(\vec{X}). \tag{2.18}$$

Compared to Equation 9 in [KTJG08], Equation 2.18 omits advected texture coordinates, which would add another level of animation to the simulation. Observations showed that their level of animation is not important enough to consider them for this thesis. Additionally this method showed to be very costly. Implementing texture coordinate advection is a topic for future work.

2.3. Advection

An overview of the state of the art of advection algorithms can be found in [SFK$^+$07]. Further they introduce a modified MacCormack advection, which is a modified Back and Forth Error Compensation and Correction (BFECC) advection algorithm that is unconditionally stable. Algorithm 2.1 summarises the unconditionally stable MacCormack advection. The idea behind it is that after one step forward and one backward you should arrive at the starting point. The difference between the starting point and the estimated location is equal to double the error. This error is then used to correct the forward step. To keep the simulation stable the result is clamped to the minimum and maximum values of the origin. So its main parts are two Semi-Lagrange advection steps and an error correcting.

Algorithm 2.1 MacCormack Advection

```
1: parameter: $\phi^n$ { Input field }
2: Apply the forward advection operator $A$ to get $\hat{\phi}^{n+1} = A(\phi^n)$
3: Apply the backward advection operator $A^R$ to get$\hat{\phi}^n = A^R(\hat{\phi}^{n+1})$
4: Adjust the advected property using the error estimate $e = (\hat{\phi}^n - \phi^n)/2$ to get $\phi^{n+1} =$
   $\hat{\phi}^{n+1} + (\hat{\phi}^n - \phi^n)/2$
5: Clamp the result $\phi^{n+1}$ between $\min(\phi^n, \phi^{n+1})$ and $\max(\phi^n, \phi^{n+1})$
6: return $\phi^{n+1}$
```

The Semi-Lagrange advection step applies only one forward advection operator A to get $\phi^{n+1} = A(\phi^n)$.

3 CUDA - Compute Unified Device Architecture

CUDA ([NVI08]) is a parallel programming model and a development environment to expose the general purpose computing capabilities of the recent NVIDIA graphics hardware to a broader audience. It uses an extended version of the C programming language to hide many of the hardware details from the user. At its core CUDA uses a multi level approach to distribute the work load between CUDA devices and with each device. The user therefore has to decompose his computations into suitable sub problems.

The user is not bound anymore to use graphics APIs (OpenGL, DirectX) to access the computing capabilities of the graphics hardware. CUDA applications can run side by side with graphics applications. CUDA and graphics APIs can be used and interoperate within a single program.

3.1. Programming Model

CUDA exposes the GPU as a parallel coprocessor with a very high number of concurrent threads to the host. It is ideal to offload data-parallel, compute-intensive algorithms from the host CPU to the CUDA device. To program a device CUDA offers an extended version of the popular programming language C. Code written for the CUDA device is organised in kernels. Each kernel is run N times on N threads in parallel. To allow each kernel instance to do different work it is provided with an ID.

3.1.1. Thread Organisation

Threads on a CUDA device are organised in two levels. Threads are grouped in blocks and blocks are organised in a grid (Figure 3.1). CUDA allows threads within one block to communicate with each other through *shared memory*. It is possible to define barriers to synchronise threads within a block. To allow an arbitrary amount of threads to be executed, blocks can be grouped in a grid. Any two blocks in the grid are independent from each other.

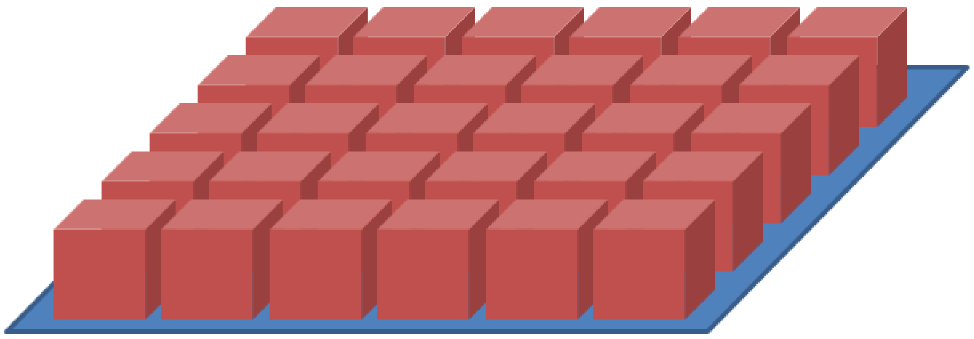


Figure 3.1.: *Thread blocks organised in a grid*

CUDA executes a number of threads in parallel per block. This group is called a warp and consists of two half-warps. The warp is the smallest granularity of parallelism in CUDA.

3.1.2. Memory Model

CUDA provides multiple memory spaces to store data. Figure 3.2 illustrates the memory hierarchy. Each thread has its own, fast memory, which can only be accessed by the thread itself. All threads in a block have access to a partition of shared memory, which is only available to one block during its life time. Finally all threads can access global memory. The life time of data in global memory space is controlled by the host.

The global memory space is shared between read/write uncached linear memory, a small partition of cached constant memory (read only) and cached texture memory (read only).

The CUDA memory model is optimised for coalesced memory access. Threads must access memory in a linear sequence with respect to their ID. The k^{th} thread in the half-warp must access the k^{th} word in memory. If threads of a half-warp do not meet this requirement separate memory transactions have to be issued, which reduces the data throughput significantly. In graphics random access is a frequent operation. It occurs when shaders read values from textures or from the shader constant memory. These graphics specific features are open to use in CUDA. Constant and texture memory enable high random access throughput for CUDA applications through the caches they have.

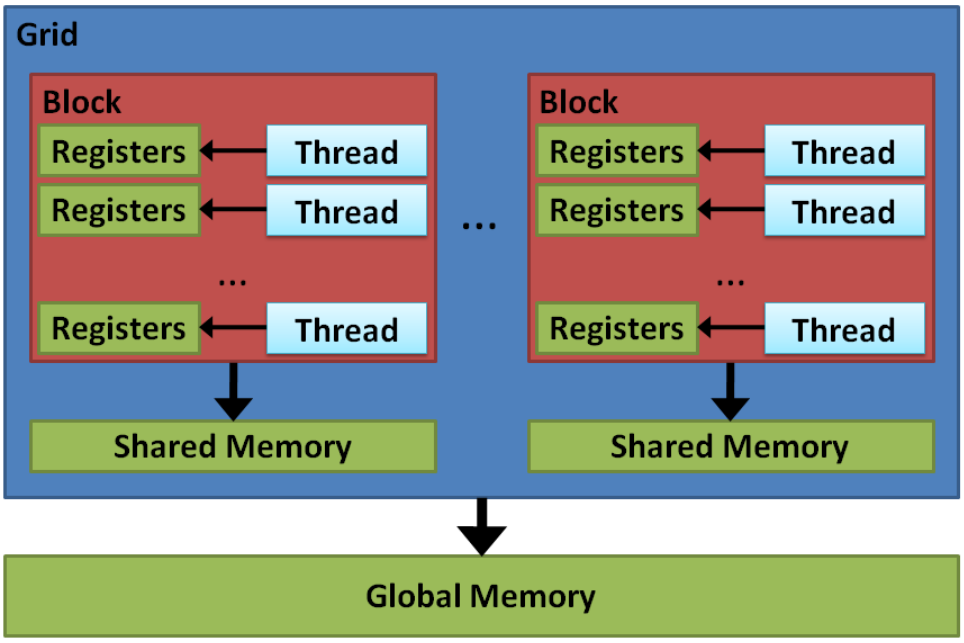


Figure 3.2.: *Memory hierarchy*

3.2. Hardware Implementation

NVIDA introduced CUDA along with the G80 chip in November 2006. Figure 3.3 shows a schematic figure of the G80. CUDA is an abstraction of this hardware design, which hasn't changed significantly up to the current GT200 chip. These chips are built around a scalable array of multithreaded Streaming Multiprocessors (SMs). A SM contains eight scalar processor cores (SP), two special function units (SFU), an instruction decoder and dispatcher, four texture units (TU) and on-chip shared memory. The G80 contains 16 of these SMs, while the GT200 contains 30 SMs.

Each thread block of a computation grid is dispatched to a single SM. This way threads in a block can communicate with each other through shared memory. Each SM manages hundreds of threads. To allow fast switching between threads, all threads in a SM allocate their registers at initialisation. The maximum number of threads and blocks a SM can manage is limited by the amount of resources each thread needs to successfully complete its calculations.

An important thing to note is that the instruction decoder and dispatcher work only with a forth of the speed of the SPs. Every 4 cycles a new instruction is issued. In that time the SPs can execute the same instruction on four sets of threads. These 4 sets build a warp. From the dispatcher's point of view these 4 thread sets are executed in parallel.

Each SM contains a block of shared memory, caches for texture and constant memory and a multi-banked register file. The registers are shared between all the threads hosted on a SM, which allows to maximise the utilisation of the SM. Sharing resources and fast switching of thread contexts is the key to help hiding memory latencies. Switching the set of executing threads needs only one clock cycle. This way the SPs do not need to wait for data to arrive and can work on other threads. Waiting for data from the global memory space takes hundreds of

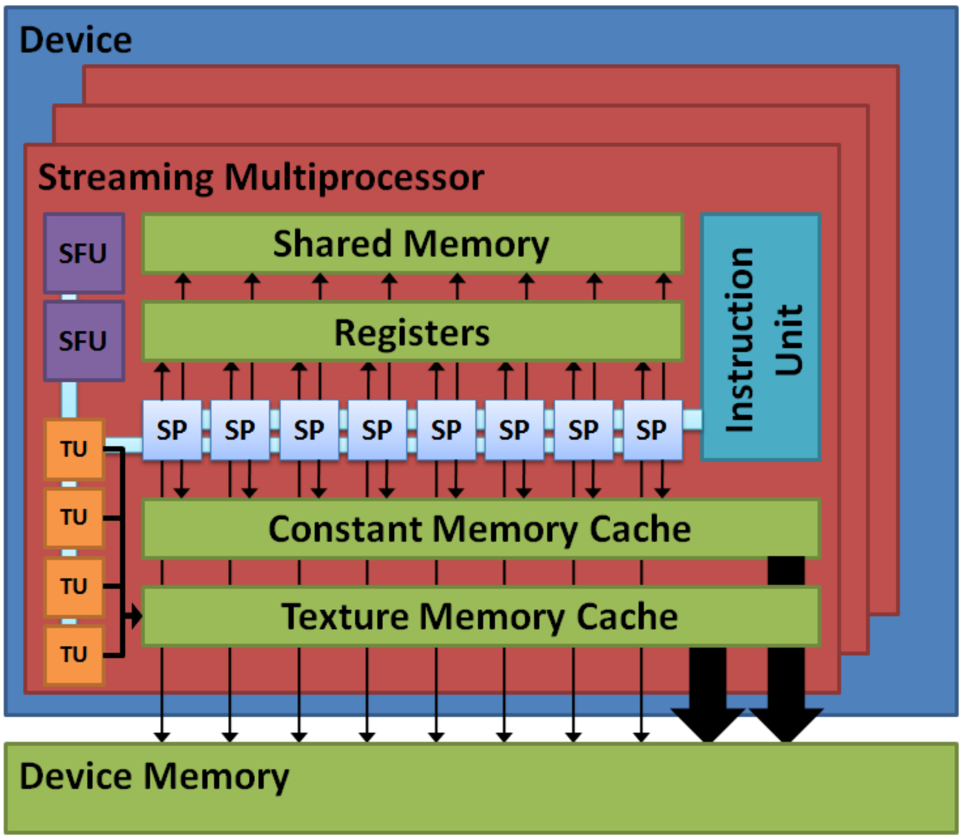


Figure 3.3.: *Hardware Model*

cycles. Switching between running threads thus is important to keeping the chip busy.

3.3. Host-side Application Programming Interface

CUDA supports two different set of APIs – a high level and a low level API (also called runtime and driver API). In this section we only focus on the runtime API. The feature set of both varieties is almost the same. The low level API additionally supports more device management functionalities and allows a more fine grain management of the CUDA kernel binaries. The runtime API is built on top of the low level API as shown in Figure 3.4.

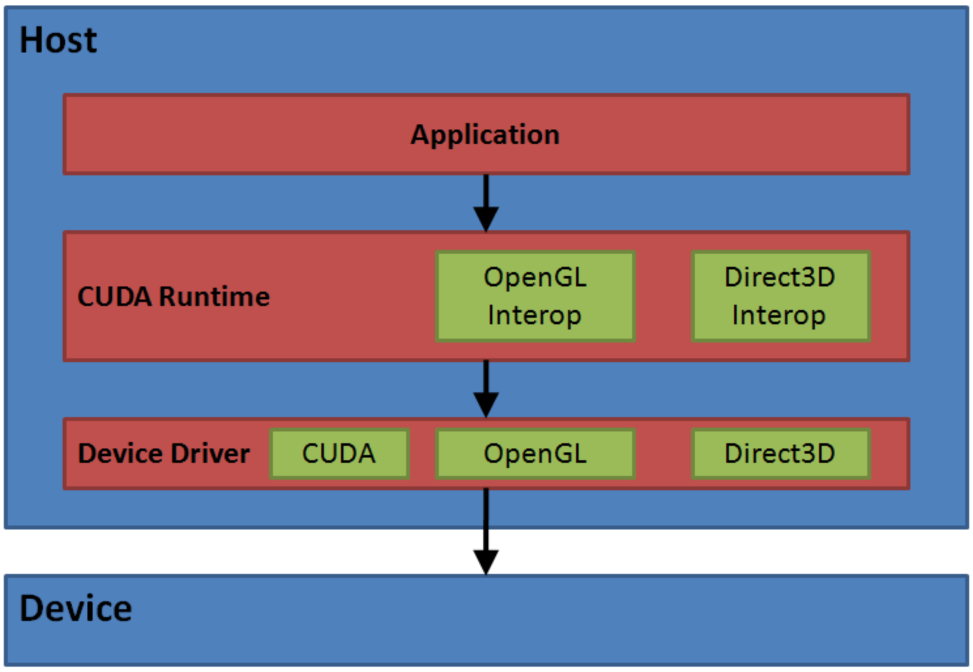


Figure 3.4.: *Software stack*

3.3.1. Device Management

Using the runtime API an explicit initialisation of a CUDA device is not necessary. Each CUDA device has to be associated to its own host thread. The first device found by the runtime is automatically assigned to the current host thread. Before any CUDA kernel is called the user can call **cudaSetDevice()** to choose the device associated to the host thread. If more than one CUDA device is present, the programmer may assign each device to its own thread and thus use all the devices available.

3.3.2. Memory Management

Since CUDA 2.0 it is sufficient to know only three allocation functions. **cudaMallocHost()** can be used to allocate page-locked memory in the host memory. To allocate memory on the CUDA device there are **cudaMalloc3D()** for linear memory and **cudaMalloc3DArray()** for CUDA arrays. There are other functions for memory allocation, but their functionality is perfectly captured by the above three functions.

To move data from the host to the device, from the device to the host, and on the device itself, the function **cudaMemcpy3D()** is sufficient. For memory transfers, there is also an asynchronous version (**cudaMemcpy3DAsync()**), which does not block the host thread.

3.3.3. Launch Control

To launch a device kernel the environment has to be initialised correctly. Through **cudaConfigureCall()** grid and block sizes are set and **cudaSetupArgument()** sets the kernel parameters. Finally **cudaLaunch()** calls a kernel. The last function returns directly, even if the kernel is still running. To wait the kernel to finish **cudaThreadSynchronize()** can be used.

3.3.4. OpenGL, DirectX Interoperability

In its current version CUDA allows to use 1D and 2D OpenGL and DirectX resources as CUDA memory. This allows for a direct cooperation of CUDA and the graphics APIs. OpenGL allows binding Vertex Buffer Objects (VBO) and Pixel Buffer Objects (PBO), while DirectX allows Vertex Buffers, Index Buffers, Surfaces and 1D and 2D textures.

3.4. Device-side Application Programming Interface

CUDA offers the developer an extended version of the C programming language. The set of extensions includes constructs to allow the programmer to access the different parts of the device that would otherwise not be accessible. The extensions include built-in variables to identify the thread and obtain information about the device configuration, additional keywords to specify the different memory locations, new primitive data types and built-in functions to access the texture units.

3.4.1. Primitive Data Types

Besides the primitive data types available in C there are corresponding vector types. They contain one, two, three or four elements. In our implementation we mostly use **float3 and float4**. There is an additional type **dim3**, which is the same as **uint3**. It is special because variables of this type are initialised to $(1, 1, 1)$.

3.4.2. Built-in Variables

Upon calling a kernel, the following variables are defined by CUDA:

gridDim This variable of type **dim3** contains the size of the computation grid.

blockIdx This variable of type **uint3** contains the block index within the grid.

blockDim This variable of type **dim3** contains the dimensions of the block.

threadIdx This variable of type **uint3** contains the thread index within the block.

warpSize This variable is of type **int** and contains the warp size in number of threads.

All these variables are read-only and it is not allowed to use any address operators on them.

3.4.3. Memory Locations

When declaring variables there are three places they can be stored: In global, uncached device memory (keyword: **__device__**), in constant memory (keyword: **__constant__**) or in shared memory (keyword: **__shared__**). While the first two are accessible to all the threads running on the device, variables in shared memory are only accessible to the threads of one block. Thus shared memory variables have to be initialised in the kernel itself. Constant memory (since it is read-only) has to be initialised by the host.

3.4.4. Texturing Unit Extensions

To access the content of a texture CUDA provides built-in functions **tex1D, tex2D, tex3D**. These functions take a reference to globally specified textures. Such a texture is defined using the keyword **texture**. Its content has to be allocated and defined by the host.

4

Real-time Algorithm

The application developed in this thesis implements a fully configurable smoke simulator. It also contains an interactive viewer using our simulator. This section focuses on the small scale detail generation (as discussed in Section 2.2) and the rendering of the simulated smoke.

4.1. Simulation Pipeline

The implemented simulation is a single threaded algorithm. It uses different multi-core architectures to parallelise parts of the simulation. To reduce the latencies produced by memory transfers between the used devices, we use asynchronous memory transfers. The coarse fluid simulation runs entirely on the host CPU. We use OpenMP to distribute the calculations to all the processors available. To speed up the generation of the small scale eddies we use NVIDIA's CUDA architecture. Algorithm 4.1 gives an overview over the whole pipeline.

4.2. CUDA Configuration

We implemented the simulator using a pure Eulerian solution, thus all our algorithms have to operate on three-dimensional arrays. The first and most important decision, which has to be taken when implementing an algorithm using CUDA, is how to partition the work. The natural decision in our case is to use a one-to-one mapping between CUDA threads and the grid nodes. As explained in Section 3.1 threads are organised in blocks and sequential memory access patterns are best.

Best performance is achieved if continuous blocks of 16 floats can be read or written at the

Algorithm 4.1 Overview of the implementation

```
1: Add a smoke inlet to the density field $D$
2: Start streaming $D$ to the CUDA device
3: Fluid simulation step (velocity field $\vec{u}$)
4: Generate high resolution velocity field $\vec{U}$ from $\vec{u}$
5: Wait for $D$ to arrive t the CUDA device
6: Advect $D$
7: Start streaming $D$ back to the host
8: Damp the coarse grid velocities $\vec{u}$
9: Wait for $D$ to arrive at the host memory
10: Use $D$ to generate an OpenGL 3D texture $T_D$
11: Render $T_D$
```

time. Because it is also easiest to implement, we decided to use blocks, where the elements have sequential indices matching the position in memory. Each block is a sequential partition of the work aligned to 16 elements. Because of resource limitations we can only support grid sizes, which don't exceed 256 elements in the first dimension and are a multiple of 16.

4.3. Wavelet Turbulence Generation

To determine where we need to generate turbulence we calculate the energy as explained in Section 2.2.4 and do a wavelet decomposition of the field. Algorithm 4.2 outlines the implementation to compute the spectra components of the energy.

Implementing the down and up sampling methods with CUDA was straight forward using the thread block and grid layout described in Section 4.2. In the down sampling step we reduce the size of each dimension by a factor of 2, which means that we only need thread blocks of half the size of each dimension. We had to take special care for the down- and up sampling kernels for the x-direction. To optimise the memory access behaviour we used shared memory to cache the work set of the active thread block. The work set is double the size of the thread block. The CUDA kernels for down- and up sampling can be found in Appendix A.1.

The velocity field and the obstacle field of the coarse grid and the energy spectra computed with Algorithm 4.2 serve as input to Algorithm 4.3, which generates the velocity field with the small scale details.

Algorithm 4.2 Compute the spectral components of the velocity fields energy

```
1: { Compute the energy $e$ of $\vec{u}$ with resolution $n^3$ }
2: for $i = 1$ to $n^3$ do
3:     $e[i] = v_x^2[i] + v_y^2[i] + v_z^2[i]$
4: end for
5: { Pseudo-march the energy into the obstacle. 4 iterations are enough because the Wavelet
   up-sampler has a neighbourhood of 3x3x3. }
6: for $i = 1$ to 4 do
7:     March the values from $e$ by one field into the obstacles.
8: end for
9: { Downsample the field in all three dimensions. }
10: $e_x \downarrow = Downsample_x(e)$
11: $e_{xy} \downarrow = Downsample_y(e_x \downarrow)$
12: $e_{xyz} \downarrow = Downsample_z(e_{xy} \downarrow)$
13: { Upsample the field in all three dimensions. }
14: $e_{xy} \downarrow\uparrow = Downsample_z(e_{xyz} \downarrow)$
15: $e_x \downarrow\uparrow = Downsample_y(e_{xy} \downarrow)$
16: $e \downarrow\uparrow = Downsample_x(e_x \downarrow)$
17: { Calculate the spectral components $\hat{e}$ of the energy field $e$ in the spectral band $\frac{n}{2}$ }
18: $\hat{e} = e - e \downarrow\uparrow$
```

Algorithm 4.3 Compute the high resolution velocity field

```
1: parameter: $\vec{u}$ { Coarse scale velocity field }
2: parameter: $o$ { Coarse scale obstacle field }
3: parameter: $\hat{e}$ { Spectral component field of the engery of $\vec{u}$ }
4: parameter: $\vec{s}$ { Resolution of the result $\vec{u}$ }
5: parameter: $\vec{S}$ { Resolution of the result $\vec{U}$, range of $\vec{X}$ }
6: parameter: $threshold$ { Threshold, where to cull noise bands }
7: parameter: $scale$ { Noise scale }
8: parameter: $strength$ { Noise strengh }
9: for all $\vec{X}$ do
10:    { Use the interpolation function $I$ to create high resolution versions of the low resolution
       fields }
11:    $\vec{U}(\vec{X}) = I(\vec{u}, \vec{X})$ { Velocity }
12:    $O(\vec{X}) = I(o, \vec{X})$ { Obstacle }
13:    $\hat{E}(\vec{X}) = I(\hat{e}, \vec{X})$ { Energy }
14:    $a = scale \cdot 2^{-\frac{5}{6}} \cdot \sqrt{2 \cdot \left|\hat{E}(\vec{X})\right|}$
15:    if $a > threshold$ and $Inside(O(\vec{X}))$ then
16:       $amplify = \vec{S}/\vec{s}$
17:       $\vec{t} = 2 \cdot \vec{X}/(amplify * scale)$
18:       for $i = i_{min}$ to $i_{max}$ do
19:          $\vec{U} = \vec{U} + a \cdot N(\vec{t})$ { Evaluate the noise function $N$ }
20:          $a = a \cdot 2^{-\frac{5}{6}}$
21:          $\vec{t} = \vec{t} \cdot 2$
22:       end for
23:    end if
24: end for
25: return $\vec{U}$
```

```
template<int N>
float wavelet_noise(float noise_tile[N][N][N], float3 p)
{
  int3 mid = (int3) ceil(p - float3(0.5f, 0.5f, 0.5f));
  float3 t = (float3) mid - (p - float3(0.5f, 0.5f, 0.5f));
  mid -= int3(1, 1, 1);

  float w[3][3];
  w[0][0] = t.x * t.x * 0.5f;
  w[0][1] = 0.5f + t.x - t.x * t.x;
  w[0][2] = 0.5f - t.x + 0.5f * t.x * t.x;

  w[1][0] = t.y * t.y * 0.5f;
  w[1][1] = 0.5f + t.y - t.y * t.y;
  w[1][2] = 0.5f - t.y + 0.5f * t.y * t.y;

  w[2][0] = t.z * t.z * 0.5f;
  w[2][1] = 0.5f + t.z - t.z * t.z;
  w[2][2] = 0.5f - t.z + 0.5f * t.z * t.z;

  float result = 0.0f;
  for (int z = 0; z < 3; z++)
  for (int y = 0; y < 3; y++)
  for (int x = 0; x < 3; x++)
  {
    float n = noise_tile[mod(mid.x + x, N)]
                        [mod(mid.y + y, N)]
                        [mod(mid.z + z, N)];

    float weight = w[0][x] * w[1][y] * w[2][z];

    result += n * weight;
  }

  return result;
}
```

Listing 4.1: *Noise Evaluation*

4.3.1. Wavelet Noise Evaluation

Basic Algorithm

The basic version of the noise evaluation algorithm is an exact copy from [CD05]. It is illustrated in Listing 4.1. This sketch represents the starting point for the CUDA implementation and optimisation.

Modified Algorithm

In the algorithm in Listing 4.1 there are 27 random accesses to the noise tile. For an implementation on the GPU this is not optimal. Each random access to memory is expensive. As explained in Section 3.1.2 random access is best implemented using the texture unit of the GPU. The next logical step is to exploit the interpolation unit included in the texture unit. It computes the convex combination

$$N(x) = (1 - \alpha) \cdot N(i) + \alpha \cdot N(i + 1), \tag{4.1}$$

where $i = \lfloor x \rfloor$ is the integral part and $\alpha = x - i$ is the fractional part of x. As derived by [SH05] cubic splines can be replaced, under certain circumstances, by a recursive application of linear interpolations. The algorithm uses a general linear combination $N(x) = a \cdot N(i) + b \cdot N(i+1)$ where $a, b \in \mathbb{R}$. As long as $0 \leq b/(a+b) \leq 1$ holds the equation can be rewritten as

$$N(x) = (a+b) \cdot N\left(\frac{i+b}{a+b}\right) \tag{4.2}$$

Splitting the middle weight of the evaluation sum 2.11 we get a function to which we can apply the reduction pattern:

$$\begin{aligned} N(x) &= w_0(x) \cdot N(i-1) + w_1(x) \cdot N(i) + w_2(x) \cdot N(i+1) \\ &= w_0(x) \cdot N(i-1) + \frac{w_1(x)}{2} \cdot N(i) + \frac{w_1(x)}{2} \cdot N(i) + w_2(x) \cdot N(i+1) \\ &= \left(w_0(x) + \frac{w_1(x)}{2}\right) \cdot N\left(i - 1 + \frac{w_1(x)}{2w_0(x) + w_1(x)}\right) \\ &+ \left(\frac{w_1(x)}{2} + w_2(x)\right) \cdot N\left(i + \frac{w_2(x)}{\frac{w_1(x)}{2} + w_2(x)}\right) \end{aligned} \tag{4.3}$$

Using this evaluation scheme, the number of texture reads can be reduced from 3 to 2 reads. For the whole noise evaluation, where 3 reads in each dimension are required, the new improved algorithm needs only 8 instead of 27 texture reads. This improvement directly translates to an improvement in the execution time by a factor of 3.

4.3.2. Vector Wavelet Noise Evaluation

How to evaluate the wavelet noise function is described in Listing 4.1. We have seen that it is possible to modify the algorithm to better fit the GPU architectures. For the Wavelet Turbulence algorithm we need the derivatives of the wavelet noise. As shown in Section 2.2.2 we can use the same sum just with modified weights. Unfortunately these new weights do not fulfil the properties needed for the convex combination. Thus we cannot use our modified evaluation algorithm and have to use to the full evaluation with 27 accesses to the noise tile.

During the course of the project we evaluated different implementations. All variations of the algorithm share two common characteristics. To get a high speedup it is necessary to use the texture unit since noise evaluation relies on random memory access. Second, the implementations need many registers, which significantly reduces performance. None of the implementations allows for more than one thread block to reside on a single Streaming Multiprocessor (SM), which is bad regarding the device occupation. The following list describes the different implementations in detail. The paragraphs **a** to **c** present the different approaches to optimise the our solution. Paragraph **d** describes, which of the presented optimisation attempts we chose for our implementation. The final CUDA source code ist listed in Appendix A.2.

a – Loop Unrolling The evaluation algorithm is a three level nested loop over a 3x3x3 cube in the noise tile. In theory, the more work can be done between access to the noise tile and using the accessed values, the better the algorithm should perform. In practice, this optimisation fails completely because it uses more registers than affordable. Reducing register usage (and thus maximising the CUDA device occupancy) is the main tool to achieve more performance in our case.

b – Derivative evaluation The x, y, z-derivatives can either be evaluated separately in sequence or all at the same time. Both variations need about the same amount of registers. The later approach has the advantage that it needs to access the memory less often, which results in a higher performance. This means the algorithm has to access the texture unit 27, 54, or 81 times. Because the ratio between the number of texture units and processors is less than one, it is beneficial to access the texture units as few as possible.

c – Noise Tile Organisation To create a divergence free vector noise field we need three different noise tiles; or as stated in [KTJG08] we can also use different offsets to the same noise tile. We implemented 4 different variations to test how the different memory access patterns perform.

Three tiles, three textures Three different noise tiles match the theory exactly. It forms the baseline where me measure all other approaches against.

One tile with offsets Using only one tile instead of three does not reduce the register pressure nor does it reduce the number of accesses to the texture units. It needs less memory and thus leaves more spare device memory for other parts of the application.

Three tiles, one texture To reduce the number of texture accesses we can pack all three noise tiles in one four channel texture. CUDA only allows 1, 2 or 4 channel textures, so we waste one channel using a 4 channel texture.

Three tiles, one texture, compression The key idea and the difference to the method above is to pack and compress the noise tile. All three noise tiles used are packed in one texture. CUDA devices allow three types of textures: 32 bit floating point textures, 8 bit or 16 bit integer textures. The last two can only represent values between 0 and 1, but are represented as integers. The values of the noise tiles thus need to be packed to a integer range of $[0, 2^{16} - 1]$. The compression of the noise tile $n(x, y, z)$ packs the values of the floating point range from $[-n, n]$ to the integer range $[0, 2^{16} - 1]$, where $n = \max |n(x, y, z)|$. This compression scheme is lossy since no attempt to make it lossless was undertaken. For the 16-bit version the errors are small and in the range of 10^{-4} per tile entry. For the 8-bit version the error per element is 10^{-2}, which is significantly higher. The 16-bit version is a good balance between saving bandwidth and paying attention to compression errors.

d – Implemented combination: Three Nested Loops, Parallel Derivatives, 16-bit Compressed Noise The implementation presented here (see Listing A.5) takes care of some CUDA issues we ran into while adapting the algorithm. Depending on the problem, algorithms can be optimised in different ways. As explained in Section 3.2, performance is better the more threads can be hosted on a Streaming Multiprocessor to hide memory latencies.

This fact drove the need to reduce the maximum number of registers used in the evaluation algorithm to a minimum. Most registers are allocated within that single subroutine. For this reason we use three nested loops (one per dimension) and recalculate the cubic B-spline weights each time they are needed. We tried using shared memory to store pre-calculated weights, but it did not result in higher performance. To save memory bandwidth and to reduce the number of accesses to the texture units we use the 16-bit compressed noise and evaluate all three partial derivatives in the same loop body.

4.3.3. Pre-Evaluated Noise

A common scheme in the field in real-time graphics is the use pre-evaluated functions in the form of lookup tables. Although lookup tables need more memory, reading a value from a table is faster than evaluating complex functions. We applied this technique to our problem and created a lookup table per noise octave. The method is essentially the same as in Listing 4.1. To create the lookup table the noise function is evaluated on each node of a 64^3 grid, scaled by a factor 2 per octave. This way we get a velocity noise function which can be used directly. If the noise tiles for evaluating the noise function have the same size as the tiles for the pre-evaluated noise, eddies created this way will look the same. This method is most probably not advisable if animated texture coordinates for the lookup are used. The coordinates are less structured organised. The necessary interpolation adds errors to the result.

4.4. Advection

Following Section 2.3 the implementation of both advection algorithms (Semi-Lagrange and MacCormack) is straightforward. Appendix A.3 contains a full source code of the advection operation. The most important design choice we had to make to implement the advection is to decide if we favour speed or memory consumption. Section 3.1.2 explains the need for using texture memory if we need to deal with random access.

Copying the density values from linear memory to texture memory and doing the advection afterwards is about twice as fast as doing the advection with lookups from linear memory. To get the highest speedup we need two additional memory buffers (textures). Algorithm 4.4 sketches the MacCormack advection implementation.

Algorithm 4.4 Implementation of the MacCormack advection

```
1:  parameter: $dt$ { Time step }
2:  parameter: $D^n$ { Density field at current time step }
3:  parameter: $\vec{U}$ { Velocity field }
4:  parameter: $\vec{X}$ { Resolution of $D^n$ and $\vec{U}$ }
5:  { Copy the density field $D^n$ to a CUDA array (texture memory) $T_D^n$. }
6:  $T_D^n \leftarrow D^n$
7:  for all $\vec{X}$ do { Forward advect the density. }
8:      $\vec{t} = \left\lfloor \vec{X} - \Delta t \cdot \vec{U} \right\rfloor$
9:      $\hat{D}^{n+1}(\vec{X}) = tex3D(T_D^n, \vec{t})$
10: end for
11: { Copy the density field $D^n$ to a CUDA array (texture memory) $T_D^n$. }
12: $\hat{T}_D^{n+1} \leftarrow \hat{D}^{n+1}$
13: for all $\vec{X}$ do { Backward advect the density. }
14:     $\vec{t} = \left\lfloor \vec{X} + \Delta t \cdot \vec{U} \right\rfloor$
15:     $\hat{D}^n(\vec{X}) = tex3D(T_D^{n+1}, \vec{t})$
16: end for
17: { Correct the $\hat{D}^{n+1}$ with the error estimate. }
18: $D^{n+1} = \hat{D}^{n+1} + (\hat{D}^n - D^n)/2$
19: for all $\vec{X}$ do { Clamp $D^{n+1}$ to value ranges of $D^n$. }
20:     $\vec{t} = \left\lfloor \vec{X} - \Delta t \cdot \vec{U} \right\rfloor$
21:     $min = \min_{i,j,k \in \{0,1\}} tex3D(T_D^n, \vec{t} + \{i,j,k\})$
22:     $max = \max_{i,j,k \in \{0,1\}} tex3D(T_D^n, \vec{t} + \{i,j,k\})$
23:     $D^{n+1} = clamp(D^{n+1}, min, max)$
24: end for
25: return $D^{n+1}$
```

4.5. Rendering

To display the results of our simulation we need a volume rendering algorithm, which is useable under real-time constraints. [IKLH04] give a compact introduction to volume rendering on modern consumer graphics hardware. Texture based volume rendering techniques are a practical approach for real-time rendering, because texturing is well supported on commodity graphics hardware. It is easy to implement and fits well into the graphics pipeline. The rendering algorithm we employ is based on [KPHE02]. They introduce a hardware accelerated volume rendering algorithm and shading model for translucent and semi-transparent materials.

The renderer uses the smoke density values of the simulation to do the visualisation. For more realism the obstacles in the simulation grid and a dynamic point light are included. The output produced by the fluid simulation is a three dimensional scalar density field. Since the rendering algorithm works with view dependent slices we use the density field directly as a 3D texture.

1. Bind the density volume texture and a frame buffer object – with to render targets– to the rendering pipeline.
2. Render the scene (depth values only) to buffer 0.
3. Disable updating the z-buffer and enable blending.
4. Create slices of the simulation volume along the half vector of viewer and light.
5. Loop through all slices from front to back.
    a) Bind buffer 1 as shadow texture to the rendering pipeline.
    b) Render and blend the slice to buffer 0 using the smoke density volume texture. To render the slice, the pixel shader in Listing 4.2 is used. It interpolates the density value at a given position in the volume and shades it using the shadow texture. An arbitrary transfer function is used to assign a colour value to the density.
    c) Render and blend the slice to buffer 1 to determine the amount of light that is blocked at a given point. Obstacles block the light completely. To render the slices the pixel shader in Listing 4.3 is used. The shader looks up the density value at a given position. If that position is within an obstacle, the opacity is set to the maximum value to block the light completely.
6. Blend buffer 0 to the screen.

Ignoring shadowing simplifies the algorithm considerably. Switching buffers is not needed anymore because buffer 1, which is needed to accumulate the shadow value, is not updated anymore. However, shadows help to create a richer visual experience and are worth the additional effort. Figure 4.1 shows the difference between rendering with and without shadows taken into account.

Extending the algorithm to more than one light should be possible. An attempt worth to try would be to render each light separately and then to blend the different images. If the lights are close to each other it should be possible to render all of them to a single image.

```
void fs_pass_one
(
  in  float3 density_pos  : TEXCOORD0,
  in  float2 shadow_pos   : TEXCOORD1,
  out float4 colour       : COLOR
)
{
  float density = tex3D(volume_data, density_pos);
  float shadow = tex2D(shadow_texture, shadow_pos);

  // Transfer function
  float c = saturate((1.0f - density) * 0.8f + 0.2f);

  // Pre-weighted colour
  c *= density;

  // Final colour
  colour.xyz = float3(c, c, c) * saturate((1.0f - shadow * 0.75f) * 0.8f + 0.2f);
  colour.w = density;
}
```

Listing 4.2: *Shader to render the eye buffer*

```
void fs_pass_two
(
  in  float3 texcoord  : TEXCOORD0,
  out float4 colour    : COLOR
)
{
  float density = tex3D(volume_data, texcoord);
  float obstacle = tex3D(obstacles_texture, texcoord);

  if (obstacle < 0.95f)
    colour = float4(density, density, density, density);
  else
    colour = float4(1.0f, 1.0f, 1.0f, 1.0f);
}
```

Listing 4.3: *Shader to render the shader buffer*

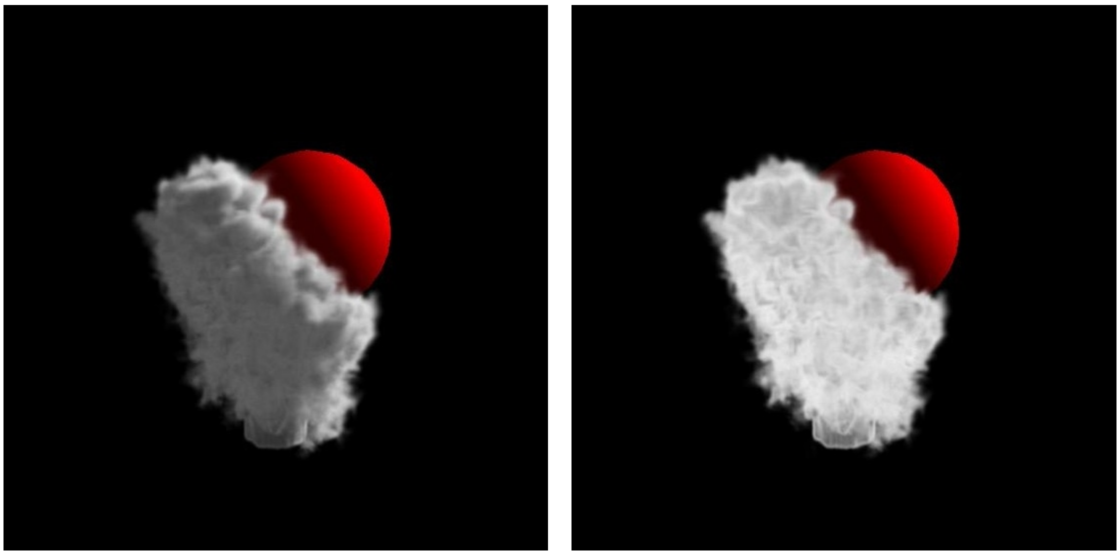
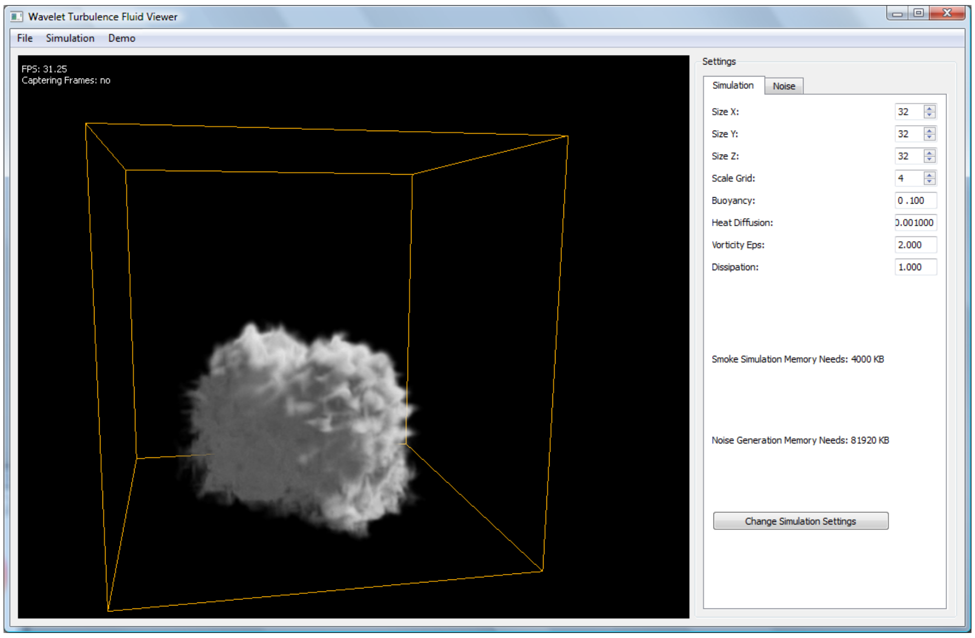

(a) With self shadowing *(b) Without self shadowing*

Figure 4.1.: *Difference between taking self shadowing into account or not*

4.6. Viewer



FPS: 31.25
Captering Frames: no

Size X:	32
Size Y:	32
Size Z:	32
Scale Grid:	4
Buoyancy:	0 .100
Heat Diffusion:	).001000
Vorticity Eps:	2.000
Dissipation:	1.000

Smoke Simulation Memory Needs: 4000 KB

Noise Generation Memory Needs: 81920 KB

Figure 4.2.: *Viewer*

To test and demonstrate the interactive simulation pipeline we developed a simple viewer (Figure 4.2). It is platform independent and runs on Windows, Linux and MacOS X using Trolltechs Qt framework. The viewer

- allows changing the simulation parameters.
- supports human interaction by letting the user place smoke inlets in the simulation volume.
- supports interaction of static and dynamic obstacles with the simulation.
- supports capturing image sequences of a running simulation.

4.6.1. Examples Of User interaction

In this section we demonstrate two ways of user interaction with the simulation – dynamic obstacles and drawing. Figure 4.3 shows a spherical obstacle, which is thrown in the simulation. The smoke particles are pushed away from the obstacle as soon it is near enough.

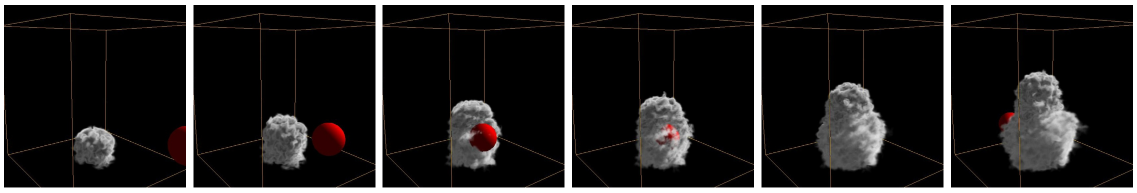


Figure 4.3.: *Obstacles interacting with the fluid*

Figure 4.4 shows how the user can add smoke inlets into the simulation.

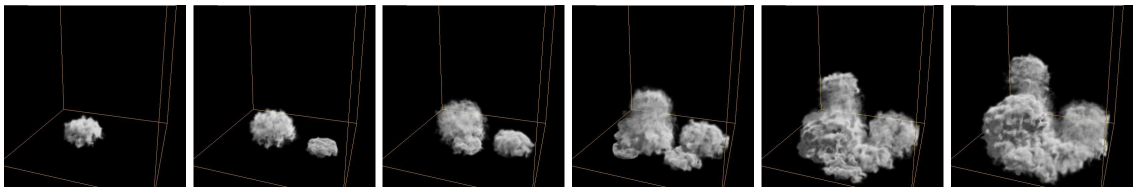


Figure 4.4.: *Adding custom smoke inlets*

5

Results

Figure 5.1 shows the difference between a standard fluid simulation (Figure 5.1(a)) using the method of [FSJ01] and the Wavelet Turbulence (Figure 5.1(b)) method by [KTJG08]. In later one much smaller eddies are visible. The grid in the rigth Figure is scaled by a factor of 4, which allows two octaves of noise to be added.

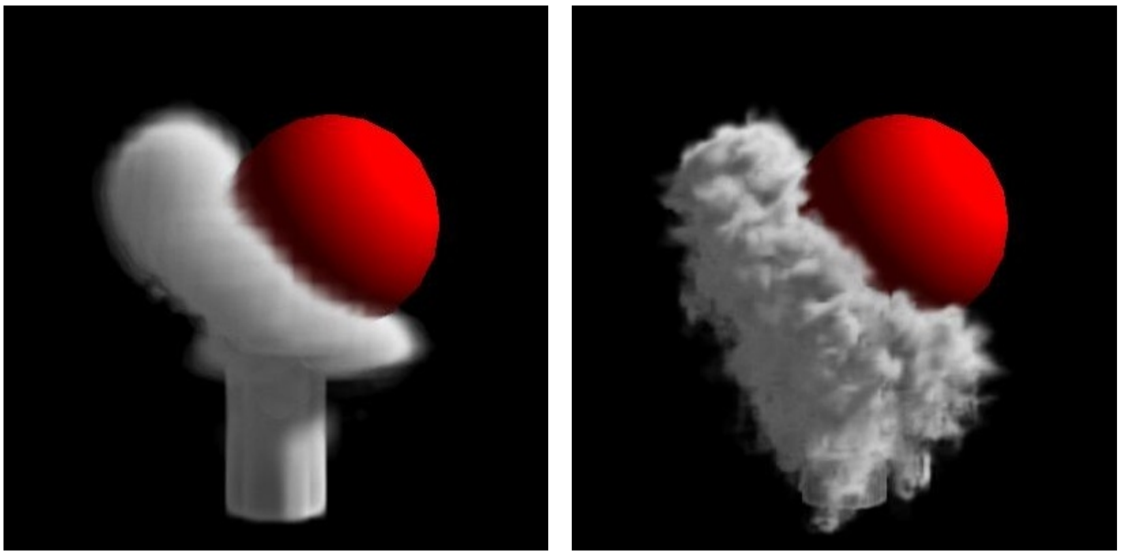

(a) Standard Fluid Simulation (b) Wavelet Turbulence

Figure 5.1.: *Comparison of the fluid simulation with the Wavelet turbulence method.*

5.1. Benchmark

The scene used for the benchmarks is shown in Figure 5.2. It shows a common scenario with a smoke producer on the ground and an obstacle in the middle. No forces apart from the buoyancy force act on the densities. If nothing else is stated the benchmark scene has a sphere in the middle. The grid for the underlying simulation has 32^3 nodes and 128^3 nodes for the small scale details. Two octaves of noise are added to create the small scale details.

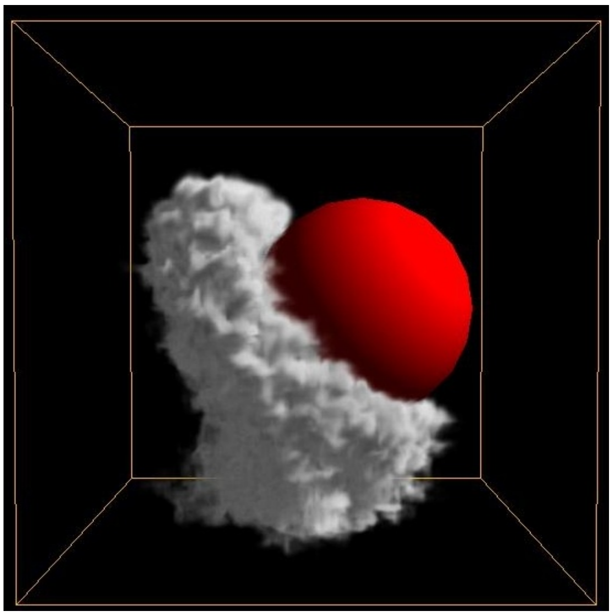


Figure 5.2.: *Benchmark scene (32x32x32 grid, 4x amplify, 2 noise octaves)*

The system for the benchmark runs was equipped with an Intel Core2 Quad Q6600 running at 2.4 GHz and a NVIDIA 8800 GTX as graphics card. The computer was setup with Microsoft Windows XP SP2 running the beta version of CUDA 2.0 with the experimental driver version 177.35. For comparison and demonstration purposes a second system was used; a HP 8510w laptop equipped with a NVIDIA Quadro FX 570M graphics card and the Intel Core2 Duo T9300 running at 2.5 GHz. The system was running Microsoft Windows Vista using the experimental driver version 177.35. Any further reference to these systems is done using either the graphics chip name or the processor identifier. Table 5.1 lists the difference between the graphics chips.

	Quadro FX 570M	8800 GTX
Streaming Multiprocessors	4	16
Shader Core Speed	950 MHz	1350 MHz
Memory Interface	128 bit	384 bit
Memory Bandwidth	22.4 GB/s	86.4 GB/s

Table 5.1.: *Comparison of CUDA devices*

5.2. Performance

This section covers the performance characteristics of the different parts of the simulator and the viewer. We show that the noise evaluation algorithm – our main target to optimisation – uses only a small fraction of the whole simulation time. Later in this section we show how the rest of the simulation pipeline performs.

5.2.1. Noise Evaluation

The whole application is more like a test suite and prototype for the noise evaluation, which can be decoupled from the actual simulation and used easily in other applications. This section contains the performance numbers of this core method. We mesured the performance of the noise evalution algorithm optimised as described in Section 4.3.2. As [KTJG08] showed that the algorithm scales well with the number of CPUs that are used for the simulation. Figure 5.3 shows the relative performance of the noise evaluation algorithm running on different hardware. The base line is the CPU implementation running on the Intel Core2 Quad Q6600. It is interesting to note that the performance scales well with the number of Streaming Multiprocessors.

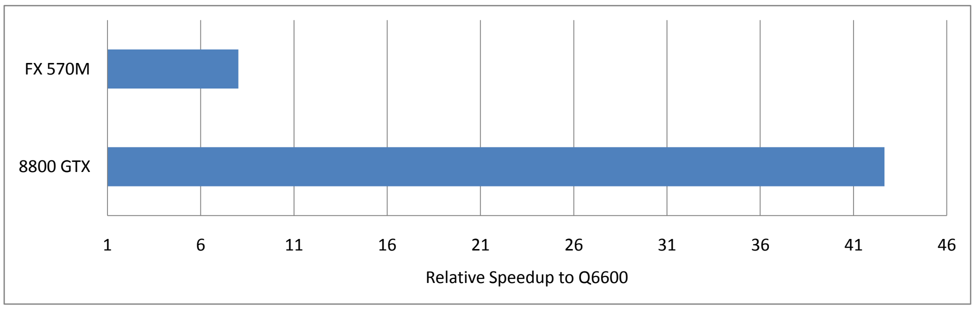


Figure 5.3.: *Noise evaluation speedup on the GPU compared to the CPU*

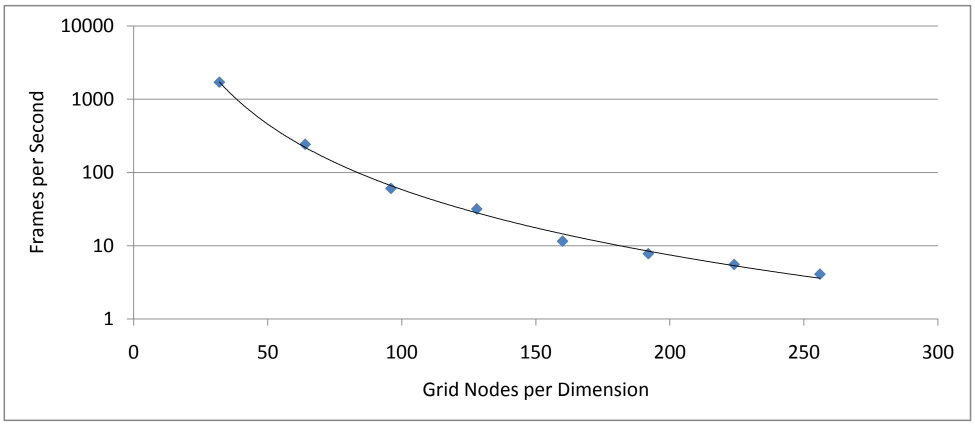


Figure 5.4.: *Noise evaluation performance: Amplification*

In Figure 5.4 the cubic trendline shows that the noise evaluation performs as expected with respect to resolution. The setting is the same as in Figure 5.2 with the difference that the resolution generated velocity field varies. A 32^3 velocity field is scaled up and two noise octaves are added. The vertical axis shows the number of generated fields per second. The scale is logarithmic to keep the results of the higher resolutions visible.

From a practical point of view any resolution beyond 128^3 is not recommended for real-time settings. Still the speed is adequate enough to enable high resolution fluid simulations in offline contexts. The picture looks a lot different when we compare different number of noise octaves. Depending on the amplification a the number of octaves that are meaningful is $\log_2 a$ (see Section 2.2.4). To reduce the impact on the runtime we can choose not to use the maximum number of octaves but a lower one. Figure 5.5 shows that the method scales linearly with the number of octaves. This result is expected because the noise evaluation method has constant running time.

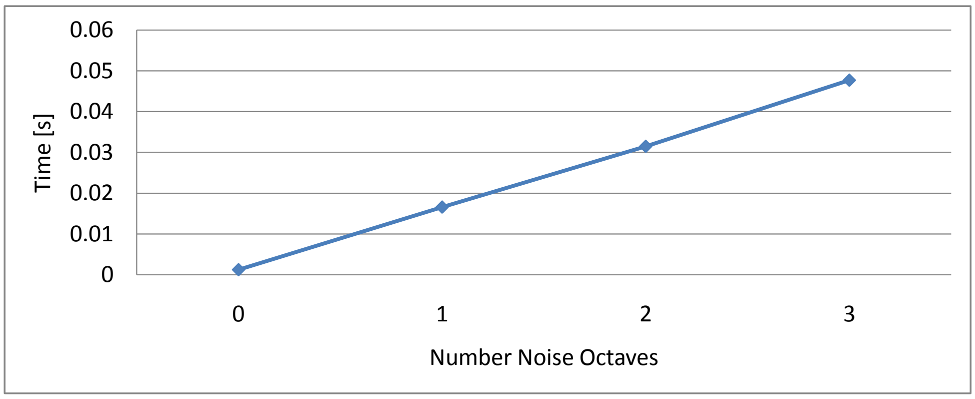


Figure 5.5.: *Noise evaluation performance: Octaves*

Pre-Evaluated Noise Using the pre-evaluation technique mentioned in Section 4.3.3 the performance penalty for generating noise is almost zero. The algorithm reduces to scaling up the velocity field and adding velocity noise from a lookup table. Pre-evaluation reduces the runtime costs of the noise evaluation method by another factor of 12. For most real-time environments this method is accurate and fast enough to generate the small scale details.

5.2.2. Application

Our initial goal was to create an interactive application using the proposed method. Speedup in the noise evaluation allowed us to try to implement the whole pipeline in real-time. The benchmark scene is running with 9 frames per second. Throughout this section we show how much time is spent on the different parts of the simulation.

As can be seen from Figure 5.6 the simulation uses about 70% of the whole runtime. As will be shown below the rendering costs are high because of technical limitations.

Simulation Following Figure 5.7 the simulation step can be decomposed into 4 steps. First we need to add the new inlet and transfer the data to the GPU. Second we do a step in the fluid

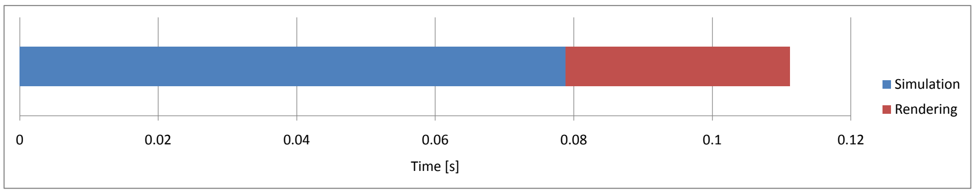


Figure 5.6.: *Application performance using the renderer with shadows enabled*

simulation. In the third step we scale up the velocity field and create the Wavelet Turbulence. In the fourth and last step we need to transfer the density values to the CPU memory to create the volume texture for rendering.

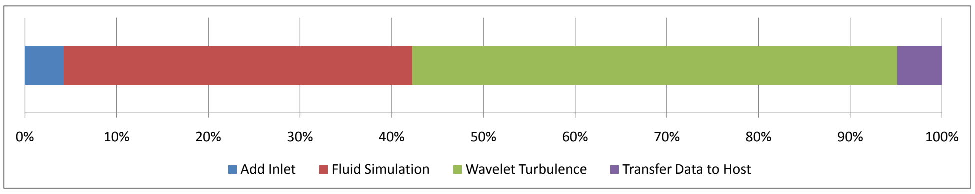


Figure 5.7.: *Simulation Performance*

With a better rendering pipeline steps 1 and 4 could be almost eliminated, because it would be possible to keep the density values on the GPU only. This is a topic for future work.

Rendering The complexity of the rendering algorithm has two sources. After the simulation step the density values are in host memory. The renderer first has to create a 3D texture from the smoke density values. For it is not possbile to write to an OpenGL volume texture from CUDA we have to take this detour through the host memory. For a visualisation grid of size 128^3 creating the volume texture takes a significant amount of time.

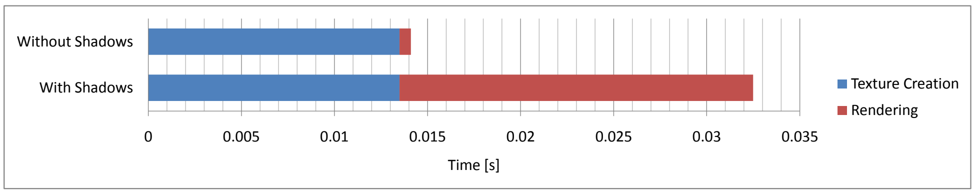


Figure 5.8.: *Rendering Performance*

Figure 5.8 show a result that is very difficult to explain. While the workload roughly doubles when shadows are enabled the slowdown is much higher. The rendering is implemented using off-screen buffers. With the recent release of OpenGL 2.0 [SWND05] it is possible to perform fast off-screen rendering. So called frame buffer objects (FBO) do not suffer from the performance penalties of older approaches (like Copy-To-Texture, or Render-To-Pixel-Buffer-Object).

Rendering to different buffers of one FBO is still very expensive, which makes the algorithm in [IKLH04] not practical for real-time rendering setting other than this prototype. Removing the rendering code completely and only leaving the buffer switches active, does not change the performance characteristics significantly. This leads to the conclusion that the buffer switches themselves are the expensive part. Currently we know of no solution to this problem other than hope for a better driver release by NVIDIA.

5.3. Memory requirements

The memory requirements can be split in three parts. The first set of memory fields is used to store the noise tiles. Depending on the implementations we employ (the different versions were explained in Section 4.3.2) the required amount of memory varies from 16 MB to 32 MB for a 128^3 noise tile.

The second part consists of the fields required for generating the velocity field. The input to the velocity generation algorithm and the energy calculation are 10 low resolution fields ($6 \cdot 4$ bytes per element, $2 \cdot 16$ bytes per element, $2 \cdot 1$ byte per element). This sums to a total of 58 bytes per node in the low resolution grid. Our benchmark in Figure 5.2 has a 32^3 simulation grid, which consumes 1.8 MB memory.

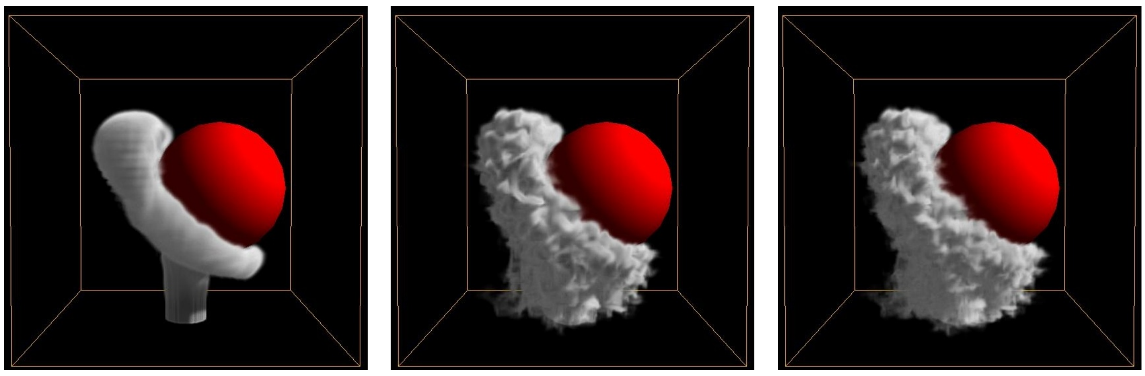
The third part contains the fields necessary for the density advection. They do not belong to the noise evaluation itself, but are part of the rendering. The fields include the density field and the high resolution velocity field. The velocity field needs $3 \cdot 4$ bytes per element. The MacCormack advection needs 3 buffers, and 2 textures to speed up random memory access ($5 \cdot 4$ bytes per element). For a scene with 128^3 nodes this results in a memory consumption of 64 MB. If we would only use a normal Semi-Lagrange advection we would only need 1 buffer and 1 texture, thus cutting the memory requirements to one third for the advection.

5.4. Visual Quality

This section looks at the results of our implementation from a pure optical perspective.

5.4.1. Number Of Noise Octaves

Section 2.2.4 describes how to determine the maximal number of noise octaves, which have an effect on the simulation. Within the range $[0, i_{max}] \subset \mathbb{N}_0$ the user is free to apply any number of octaves. The more noise octaves are applied to the velocity field the richer the visual experience is. The most difference and thus the “cheapest” noise band with respect to quality improvement is the first one. With every further octave eddies shrink in their size by a factor of two, thus the costs per detail grow.



(a) 0 Octaves *(b) 1 Octaves* *(c) 2 Octaves*

Figure 5.9.: *Benchmark scene with different number of noise octaves.*

5.4.2. Noise Tile Size

To avoid repetitions in the generated noise a sufficiently large noise tile has to be used. We have used a 128^3 noise tile, which is oversized for most real-time applications. Considering the memory and computation needs of the rest of the algorithms, the size of the fine grid is limited to 128 or 256 elements per dimension. For this size it is sufficient to use a 64^3 noise tile, which reduces the memory space needed by a factor of 8. If the scene is turbulent enough (through obstacles) even smaller noise tiles are useable, because repetitions in the noise evaluation are less visible.

5.4.3. Pre - Evaluated Noise

The simulation using pre-evaluated noise does not look the same as using noise evaluation (even using the same noise tile). But it looks plausible enough to be a useful method. Figure 5.10 shows a side by side comparision of the benchmark scene using pre-evaluated noise. The top row shows the simulation using runtime noise evalution while the bottom row uses pre-evaluated

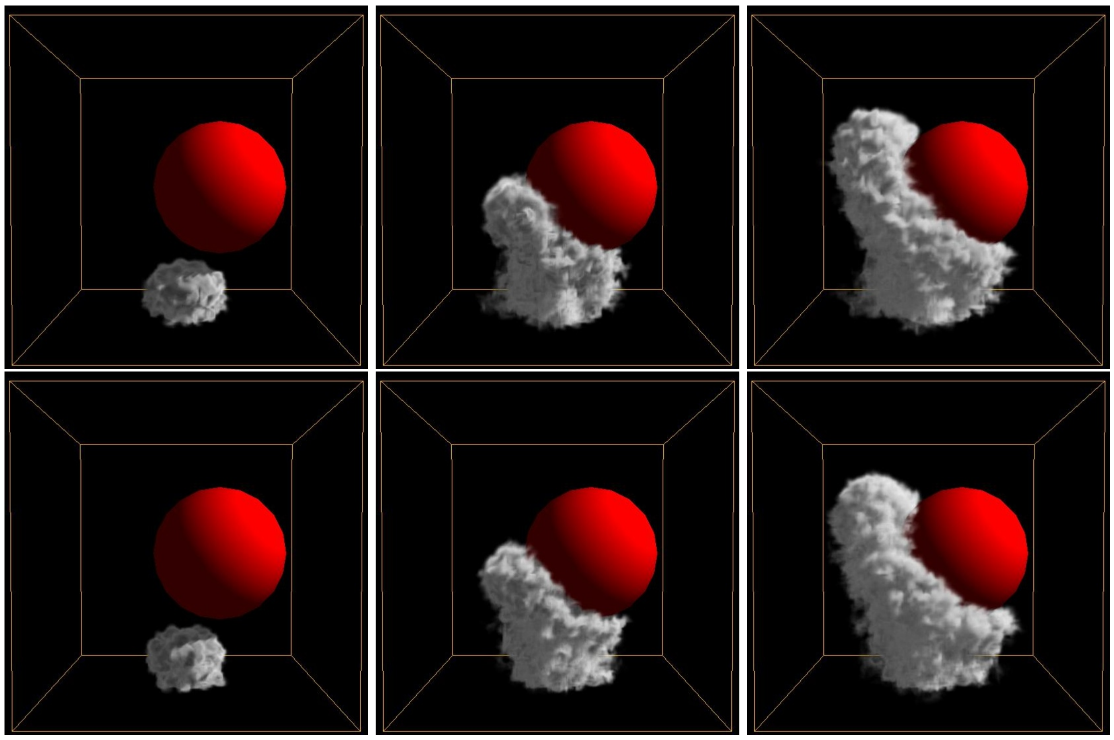
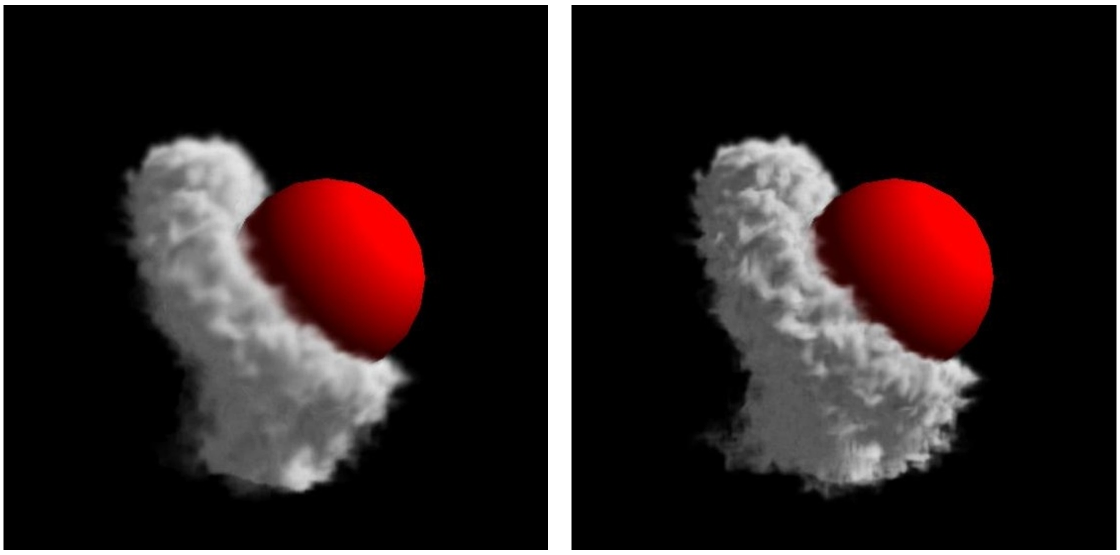


Figure 5.10.: *Benchmark scene using pre-evaluated noise*

noise. Comparing the runtime evaluation method and the pre-evalated noise, the average error per element is in the order of 10^{-3}, which is acceptable for visual plausibility.

5.4.4. Advection



(a) Semi Lagrange Advection (b) Mac Cormack Advection

Figure 5.11.: *Advection Methods*

Figure 5.11 shows the difference between using a simple Semi-Lagrange advection step (5.11(a)) and more expensive Mac Cormack advection step (5.11(b)). Semi-Lagrange advection blurs out many of the interesting high frequency details. Thus if the resources are available, Mac Cormack advection is preferable. As explained in 4.4 the Semi-Lagrange advection needs a memory copy from linear memory to texture memory, which make the advection step expensive. As can be seen in Figure 5.12 the advection performance is proportional to the number of elements. The Mac Cormack advection is roughly about three times more expensive than the Semi-Lagrange advection.

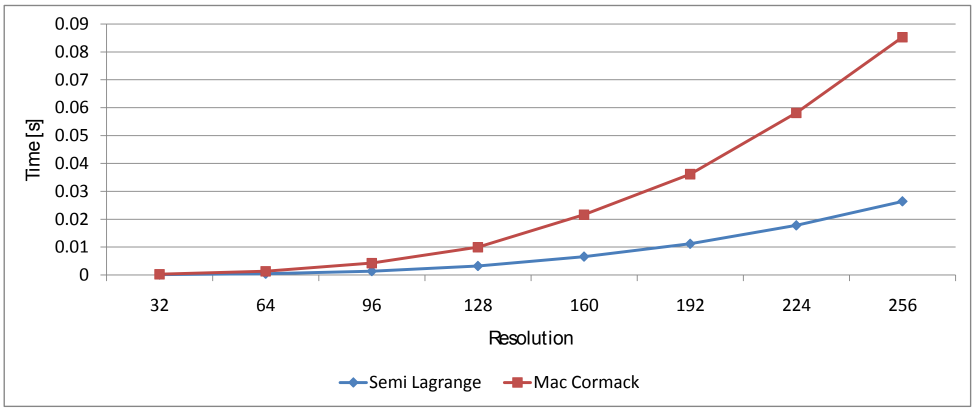


Figure 5.12.: *Advection Performance*

6

Conclusion and Outlook

We have demonstrated that it is possible to perform highly detailed interactive smoke simulations using a modified version of the approach described in [KTJG08] implemented with CUDA. Although the simulation itself with high frame rates, the memory needs are too high for most real-time environments at the moment. On the other hand the method is well suited to enhance fluid simulations in movies, where the rendering time per frame is also a crucial part of the whole production time.

Fluid simulations on a grid only basis have to update values on the whole simulation domain, even if no density particles are present. Separating large and small scale detail enables us to do a full grid only simulation on the large details. To track the density values and the small scale details, it is also possible to use particles. Although a large quantity of particles is needed, they could be beneficial with the current API. Dynamically creating 3D textures each frame is very expensive because the interaction between CUDA and graphics APIs is limited to 2D resources. For the particles, the interaction could use a 1D vertex buffer. Because the velocities are recomputed each frame, we only need to store the position of each particle. Taking a 128^3 grid as an example we can pack over 5 million particles into the same memory space.

In addition the implementation has some space for improvements. Most importantly, one part of the algorithm (the animated texture coordinates) was not implemented. This additional animation would further enhance the visual appeal. The implementation involves two steps:

- Advect each dimension of the texture coordinates.
- Take the Jaccobian at each node (3x3 matrix) and calculate its Eigenvalues.

The advection is easy to do. Calculating the Eigenvalues should also be well suited for the GPU. A 3x3 system can be solved directly and allows the GPU to well utilise the floating point math performance.

As easy as the simulation scales to multiple CPUs it should scale to multiple CUDA devices. Using particles the method scales almost trivially, where each device updates a partition of the particle buffer. With our method based on advected density fields it is more complex to use multiple CUDA devices. Each device would contribute to a separate part of the simulation volume. For the advection the density field has to be present on all of the devices. Limiting the magnitude of the velocities though would allow us to synchronise only small portions of the whole field and save expensive memory bandwidth.

A

Source Code

A.1. Wavelet Decompostion

This section contains the full source code listings of the CUDA code used for the Wavelet decomposition.

```
__constant__ float aCoeffs[32] = {
        0.000334f,-0.001528f, 0.000410f, 0.003545f,-0.000938f,-0.008233f, 0.002172f, 0.019120f,
        -0.005040f,-0.044412f, 0.011655f, 0.103311f,-0.025936f,-0.243780f, 0.033979f, 0.655340f,
        0.655340f, 0.033979f,-0.243780f,-0.025936f, 0.103311f, 0.011655f,-0.044412f,-0.005040f,
        0.019120f, 0.002172f,-0.008233f,-0.000938f, 0.003546f, 0.000410f,-0.001528f, 0.000334f
};

__device__ inline void cuda_energy_downsample(float* to, const float* from, int i, int n, int stride)
{
        const float* const aCoCenter= &aCoeffs[16];

        float result = 0.0f;
        for (int k = 2 * i - 16; k < 2 * i + 16; k++)
        {
                // handle boundary
                float fromval;
                if(k<0) {
                        fromval = from[0];
                } else if(k>n-1) {
                        fromval = from[(n-1) * stride];
                } else {
                        fromval = from[k * stride];
                }

                result += aCoCenter[k - 2*i] * fromval;
        }
        to[i * stride] = result;
}
```

Listing A.1: *Generic Downsampling Kernel*

```
extern "C" __global__ void cuda_energy_downsample_x
(
        float* to,
        const float* from,
        const dim3 res
)
{
        __shared__ float cache[256];

        int x = threadIdx.x;
        int y = blockIdx.x*blockDim.y+threadIdx.y;
        int z = blockIdx.y*blockDim.z+threadIdx.z;

        const int index = y * res.x + z * res.x * res.y;

        cache[threadIdx.y * res.x + threadIdx.x] = from[index + x];
        cache[threadIdx.y * res.x + res.x / 2 + threadIdx.x] = from[index + res.x / 2 + x];
        __syncthreads();

        cuda_energy_downsample(&to[index], &cache[threadIdx.y * res.x], x, res.x, 1);
}

extern "C" __global__ void cuda_energy_downsample_y
(
        float* to,
        const float* from,
        const dim3 res
)
{
        int x = threadIdx.x;
        int y = blockIdx.x*blockDim.y+threadIdx.y;
        int z = blockIdx.y*blockDim.z+threadIdx.z;

        const int index = x + z * res.x * res.y;
        cuda_energy_downsample(&to[index], &from[index], y, res.y, res.x);
}

extern "C" __global__ void cuda_energy_downsample_z
(
        float* to,
        const float* from,
        const dim3 res
)
{
        int x = threadIdx.x;
        int y = blockIdx.x*blockDim.y+threadIdx.y;
        int z = blockIdx.y*blockDim.z+threadIdx.z;

        const int index = x + y * res.x;
        cuda_energy_downsample(&to[index], &from[index], z, res.z, res.x * res.y);
}
```

Listing A.2: *Direction-specific Downsampling Kernels*

```
__constant__ float pCoeffs[4] = { 0.25f, 0.75f, 0.75f, 0.25f };

__device__ void cuda_energy_upsample(float *to, const float *from, int i, int n, int stride)
{
        const float *const pCoCenter = &pCoeffs[2];

        float result = 0.0f;
        for (int k = i / 2; k <= i / 2 + 1; k++)
        {
                // handle boundary
                float fromval;
                if(k>n/2) {
                        fromval = from[(n/2) * stride];
                } else {
                        fromval = from[k * stride];
                }

                result += pCoCenter[i - 2 * k] * fromval;
        }
        to[i * stride] = result;
}
```

Listing A.3: *Generic Upsampling Kernel*

```
extern "C" __global__ void cuda_energy_upsample_x
(
        float* to,
        const float* from,
        const dim3 res
)
{
        __shared__ float cache[128];

        int x = threadIdx.x;
        int y = blockIdx.x*blockDim.y+threadIdx.y;
        int z = blockIdx.y*blockDim.z+threadIdx.z;

        const int index = y * res.x + z * res.x * res.y;

        cache[threadIdx.y * blockDim.x + threadIdx.x] = from[index + x];
        __syncthreads();

        cuda_energy_upsample(&to[index], &cache[threadIdx.y * blockDim.x], x, res.x, 1);
}
```

```
extern "C" __global__ void cuda_energy_upsample_y
(
        float* to,
        const float* from,
        const dim3 res
)
{
        int x = threadIdx.x;
        int y = blockIdx.x*blockDim.y+threadIdx.y;
        int z = blockIdx.y*blockDim.z+threadIdx.z;

        const int index = x + z * res.x * res.y;
        cuda_energy_upsample(&to[index], &from[index], y, res.y, res.x);
}

extern "C" __global__ void cuda_energy_upsample_z
(
        float* to,
        const float* from,
        const dim3 res
)
{
        int x = threadIdx.x;
        int y = blockIdx.x*blockDim.y+threadIdx.y;
        int z = blockIdx.y*blockDim.z+threadIdx.z;

        const int index = x + y * res.x;
        cuda_energy_upsample(&to[index], &from[index], z, res.z, res.x * res.y);
}
```

Listing A.4: *Direction-specific Upsampling Kernels*

A.2. Velocity Noise Generation

This section contains the full source code listings of the CUDA code used for the vector noise evaluation and the Wavelet Turbulence generation.

```
template<char X, int N, int M>
__device__ inline float vulcanus_noise_derivative_weight(const float t)
{
        if ((X == 'x' && N == 0) || (X == 'y' && N == 1) || (X == 'z' && N == 2))
        {
                if (M == 0)
                        return t;
                else if (M == 1)
                        return 1.0f - 2.0f * t;
                else
                        return t - 1.0f;
        }
        else
        {
                if (M == 0)
                        return t * t * 0.5f;
                else if (M == 1)
                        return 0.5f + t - t*t;
                else
                        return 0.5f - t + 0.5f * t * t;
        }
}

template<typename T>
__device__ float3 vulcanus_vectornoise_evaluate
(
        texture<T, 3, cudaReadModeNormalizedFloat> noise,
        const float p[3]
)
{
        float ext = 1.0f / (float) NOISE_TILE_SIZE;

        float midX = ceil(p[0] - 0.5f);
        float t0   = midX - (p[0] - 0.5f);
        midX -= 1.0f;
        midX *= ext;

        float midY = ceil(p[1] - 0.5f);
        float t1   = midY - (p[1] - 0.5f);
        midY -= 1.0f;
        midY *= ext;

        float midZ = ceil(p[2] - 0.5f);
        float t2   = midZ - (p[2] - 0.5f);
        midZ -= 1.0f;
        midZ *= ext;
        float3 v;
        v.x = 0.0f;
        v.y = 0.0f;
        v.z = 0.0f;

        for (int k = 0; k < 3; k++, midZ += ext)
        {
                float w2_x = (k == 0) ? vulcanus_noise_derivative_weight<'x', 2, 0>(t2) :
                                (k == 1) ? vulcanus_noise_derivative_weight<'x', 2, 1>(t2) :
                                           vulcanus_noise_derivative_weight<'x', 2, 2>(t2);
                float w2_y = w2_x;

                float w2_z = (k == 0) ? vulcanus_noise_derivative_weight<'z', 2, 0>(t2) :
                                (k == 1) ? vulcanus_noise_derivative_weight<'z', 2, 1>(t2) :
                                           vulcanus_noise_derivative_weight<'z', 2, 2>(t2);

                for (int j = 0; j < 3; j++, midY += ext)
                {
                        float w1_x = (j == 0) ? vulcanus_noise_derivative_weight<'x', 1, 0>(t1) :
                                        (j == 1) ? vulcanus_noise_derivative_weight<'x', 1, 1>(t1) :
                                                   vulcanus_noise_derivative_weight<'x', 1, 2>(t1);

                        float w1_y = (j == 0) ? vulcanus_noise_derivative_weight<'y', 1, 0>(t1) :
                                        (j == 1) ? vulcanus_noise_derivative_weight<'y', 1, 1>(t1) :
                                                   vulcanus_noise_derivative_weight<'y', 1, 2>(t1);
                        float w1_z = w1_x;

                        for (int i = 0; i < 3; i++, midX += ext)
                        {
                                // Read the noise texture
                                float4 n = tex3D(noise, midX, midY, midZ);

                                float w0_x = (i == 0) ? vulcanus_noise_derivative_weight<'x', 0, 0>(t0) :
                                                (i == 1) ? vulcanus_noise_derivative_weight<'x', 0, 1>(t0) :
                                                           vulcanus_noise_derivative_weight<'x', 0, 2>(t0);

                                float w0_y = (i == 0) ? vulcanus_noise_derivative_weight<'y', 0, 0>(t0) :
                                                (i == 1) ? vulcanus_noise_derivative_weight<'y', 0, 1>(t0) :
                                                           vulcanus_noise_derivative_weight<'y', 0, 2>(t0);
                                float w0_z = w0_y;

                                // Decompress noise
                                n.w = 1.0f / n.w;
                                n.x = (n.x - 0.5f) * n.w;
                                n.y = (n.y - 0.5f) * n.w;
                                n.z = (n.z - 0.5f) * n.w;

                                // Calculate the final weights
                                float w_x = w0_x * w1_x * w2_x;
                                float w_y = w0_y * w1_y * w2_y;
                                float w_z = w0_z * w1_z * w2_z;

                                // Add the weighted noise
                                v.z += n.y * w_x;
                                v.y -= n.z * w_x;

                                v.z -= n.x * w_y;
                                v.x += n.z * w_y;

                                v.y += n.x * w_z;
                                v.x -= n.y * w_z;
                        }
                        midX -= 3.0f * ext;
                }
                midY -= 3.0f * ext;
        }

        return v;
}
```

Listing A.5: Vector noise evaluation

```
// Vector noise input
texture<ushort4, 3, cudaReadModeNormalizedFloat> tex_noise;

// Coarse input data
texture<float2, 3, cudaReadModeElementType> tex_small_eigen;
texture<float4, 3, cudaReadModeElementType> tex_small_velocity_energy;
texture<float4, 3, cudaReadModeElementType> tex_small_tex_coord;
texture<unsigned char, 3, cudaReadModeNormalizedFloat> tex_small_obstacles;

extern "C" __global__ void cuda_noise_inject
(
        float   inv_amplify,              // Inverse amplification (small to large grid)
        float   inv_noise_scale,          // Noise scale factor (inverted)
        int     nr_octaves,               // Number noise octaves to use
        float   noise_culling_threshold,  // Threshold where to cull noise bands
        float   noise_strength,           // Noise strengh
        float   persistence,              // Magic constant
        float3  noise_animation_offset,   // Noise animation offset
        float*  velocity_x,               // Velocity field (x component)
        float*  velocity_y,               // Velocity field (y component)
        float*  velocity_z,               // Velocity field (z component)
        dim3    high_res,               // Resolution of the fine grid
        dim3    low_res                 // Resulution of the coarse grid
)
{
        const int i = threadIdx.x;
        const int j = blockIdx.x*blockDim.y+threadIdx.y;
        const int k = blockIdx.y*blockDim.z+threadIdx.z;

        if (0 < i && i < high_res.x - 1 && 0 < j && j < high_res.y - 1 && 0 < k && k < high_res.z - 1)
        {
                // Fetch the obstacle, velocity and energy from the coarse grid (linear interpolation)
                const float obstacle = tex3D(tex_small_obstacles,
                                             i * inv_amplify + 0.5f,
                                             j * inv_amplify + 0.5f,
                                             k * inv_amplify + 0.5f);

                float4 velocity_energy = tex3D(tex_small_velocity_energy,
                                               i * inv_amplify + 0.5f,
                                               j * inv_amplify + 0.5f,
                                               k * inv_amplify + 0.5f);
                float3 velocity;
                velocity.x = velocity_energy.x;
                velocity.y = velocity_energy.y;
                velocity.z = velocity_energy.z;

                // Create the texture coordinates.
                // Multiply the texture coordinate by 2 * low_res so that it creates
                // vortices half the size of a coarse grid cell
                float3 tex_coords = make_float3(
                        (float) i * inv_amplify * 2.0f * inv_noise_scale,
                        (float) j * inv_amplify * 2.0f * inv_noise_scale,
                        (float) k * inv_amplify * 2.0f * inv_noise_scale
                );

                // Compute the Wavelet energy on the highest level
                float coefficient = sqrtf(2.0f * fabs(velocity_energy.w));

                // Base amplitude for octave 0
                float amplitude = fabs(noise_strength * coefficient) * persistence;

                if (amplitude > noise_culling_threshold && obstacle <= 0.95f)
                {
                        for (int octave = 0; octave < nr_octaves; octave++)
                        {
                                // Animated by shifting in space over time
                                const float pos[3] = { tex_coords.x + noise_animation_offset.x,
                                                       tex_coords.y + noise_animation_offset.y,
                                                       tex_coords.z + noise_animation_offset.z };

                                float3 noise_velocity = vulcanus_vectornoise_evaluate<ushort4>(tex_noise, pos);

                                // Multiply the vector noise times the maximum allowed
                                // noise amplitude at this octave and add it to the total
                                velocity.x += noise_velocity.x * amplitude;
                                velocity.y += noise_velocity.y * amplitude;
                                velocity.z += noise_velocity.z * amplitude;

                                // scale coefficient for next octave
                                amplitude *= persistence;

                                // Double the texture coordinate scaling in preparation
                                // for the next octave, effectively halving the size of the
                                // vortices that will be produced
                                tex_coords.x *= 2.0f;
                                tex_coords.y *= 2.0f;
                                tex_coords.z *= 2.0f;
                        }
                }

                const unsigned int index = i + j*high_res.x + k*high_res.x*high_res.y;
                velocity_x[index] = velocity.x;
                velocity_y[index] = velocity.y;
                velocity_z[index] = velocity.z;
        }
}
```

Listing A.6: *Wavelet turbulence generation*

A.3. Advection

This section contains the full source code listings of the CUDA code used for the advection step.

```
// Global data
texture<float, 3, cudaReadModeElementType> tex_old_field;

extern "C" __global__ void cuda_advect_semi_lagrange
(
        const float dt,
        const float* velx, const float* vely, const float* velz,
        float* new_field,
        const dim3 res
)
{
        int i = threadIdx.x;
        int j = blockIdx.x*blockDim.y+threadIdx.y;
        int k = blockIdx.y*blockDim.z+threadIdx.z;
        int index = i + j*res.x + k*res.x*res.y;

        // Backtrace
        float xTrace = i - dt * velx[index];
        float yTrace = j - dt * vely[index];
        float zTrace = k - dt * velz[index];

        // Clamp backtrace to grid boundaries
        xTrace = clamp(xTrace, 0.5f, res.x - 1.5f);
        yTrace = clamp(yTrace, 0.5f, res.y - 1.5f);
        zTrace = clamp(zTrace, 0.5f, res.z - 1.5f);

        new_field[index] = tex3D(tex_old_field, xTrace + 0.5f, yTrace + 0.5f, zTrace + 0.5f);
}
```

Listing A.7: *Semi-Lagrange Advection*

```
// Global data
texture<float, 3, cudaReadModeElementType> tex_old_field;

extern "C" __global__ void cuda_advect_mac_cormack_merge_clamp
(
        const float dt,
        const float* velx, const float* vely, const float* velz,
        const float* phiN, const float* phiHatN, const float* phiHatN1, float* phiN1,
        const dim3 res
)
{
        __shared__ float phi_hat_n1[MAX_CUDA_THREADS];
        __shared__ float phi_n1[MAX_CUDA_THREADS];

        int x = threadIdx.x;
        int y = blockIdx.x*blockDim.y+threadIdx.y;
        int z = blockIdx.y*blockDim.z+threadIdx.z;

        // Index
        const int index = x + y * res.x+ z * res.x*res.y;
        const int tid = threadIdx.x + blockDim.x*threadIdx.y;

        // phiN1 = phiHatN1 + (phiN - phiHatN) / 2
        phi_hat_n1[tid] = phiHatN1[index];
        phi_n1[tid] = phi_hat_n1[tid] + (phiN[index] - phiHatN[index]) * 0.5f;

        if (0 < x && x < res.x - 1 && 0 < y && y < res.y - 1 && 0 < z && z < res.z - 1)
        {
                // Velocities
                const float vel_x = velx[index];
                const float vel_y = vely[index];
                const float vel_z = velz[index];

                // Backtrace
                float xTrace = x - dt * vel_x;
                float yTrace = y - dt * vel_y;
                float zTrace = z - dt * vel_z;

                // See if it goes outside the boundaries
                bool hasObstacle =
                        (zTrace < 1.0f) || (zTrace > res.z - 2.0f) ||
                        (yTrace < 1.0f) || (yTrace > res.y - 2.0f) ||
                        (xTrace < 1.0f) || (xTrace > res.x - 2.0f);

                float xBackward = x + dt * vel_x;
                float yBackward = y + dt * vel_y;
                float zBackward = z + dt * vel_z;

                hasObstacle = hasObstacle ||
                        (zBackward < 1.0f) || (zBackward > res.z - 2.0f) ||
                        (yBackward < 1.0f) || (yBackward > res.y - 2.0f) ||
                        (xBackward < 1.0f) || (xBackward > res.x - 2.0f);
                // Reuse old advection instead of doing another one...
                if(hasObstacle) { phi_n1[tid] = phi_hat_n1[tid]; }
                else
                {
                        // Clamp backtrace to grid boundaries and locate neighbors to interpolate
                        const int x0 = (int) clamp(xTrace, 0.5f, res.x - 1.5f);
                        const int y0 = (int) clamp(yTrace, 0.5f, res.y - 1.5f);
                        const int z0 = (int) clamp(zTrace, 0.5f, res.z - 1.5f);

                        float min_field = 3.402823466e+38F;
                        float max_field = -3.402823466e+38F;

                        for (int k = 0; k < 2; k++)
                        for (int j = 0; j < 2; j++)
                        for (int i = 0; i < 2; i++)
                        {
                                const float value = tex3D(tex_old_field, x0 + i, y0 + j, z0 + k);
                                min_field = min(value, min_field);
                                max_field = max(value, max_field);
                        }

                        phi_n1[tid] = clamp(phi_n1[tid], min_field, max_field);
                }

                phiN1[index] = phi_n1[tid];
        }
        else { phiN1[index] = 0.0f; }
}
```

Listing A.8: *MacCormack correction and clamping*

A.4. Volume Renderer

This section contains the source code for the volume renderer. The shaders used in the function are listed in Listing 4.2 and 4.3.

```
void ViewAlignedSlices3D::RenderWithShadows(const Scene* scene)
{
        using namespace Unicorn::Mathematics;

        // Disable writing to the z-buffer
        glDepthMask(GL_FALSE);

        // Set the blend mode
        glEnable(GL_BLEND);

        // Bind the frame buffer
        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, mFbo);
        glPushAttrib(GL_VIEWPORT_BIT | GL_COLOR_BUFFER_BIT);
        glViewport(0, 0, 512, 512);

        // Draw to the eye buffer
        glDrawBuffer(GL_COLOR_ATTACHMENT0_EXT);

        // Clear the buffer
        glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
        glClear(GL_COLOR_BUFFER_BIT);

        // Draw to the light buffer
        glDrawBuffer(GL_COLOR_ATTACHMENT1_EXT);

        // Clear the buffer
        glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
        glClear(GL_COLOR_BUFFER_BIT);

        // Create the matrix of for the light
        const Camera& shadow_frustum = scene->GetShadowFrustum();
        Matrix4x4f light_view_proj = shadow_frustum.Projection() * shadow_frustum.View();

        // Enable shader profiles
        cgGLEnableProfile(CgProfileVertexShader());
        cgGLEnableProfile(CgProfileFragmentShader());

        // Create the sampling ray
        AABB<float> aabb(mLower, mUpper);
        bool front_to_back = true;
        float cos_half_angle = 1.0f;
        Ray<float> sampling_ray
          = CreateSamplingRay(aabb, scene->GetCamera().Position(), scene->GetLight().Position, cos_half_angle, front_to_back);
        Vector3f begin = sampling_ray.GetPointOnRay(sampling_ray.Min());
        Vector3f end = sampling_ray.GetPointOnRay(sampling_ray.Max());

        Vector3f plane_normal = (begin - end).Normalised();
        Vector3f view_ray = (end - begin).Normalised();
        float length = (end - begin).Length();

        // Render the slices
        int nr_slices = length / (cos_half_angle * 0.01);
        float slab = length / (float) (nr_slices - 1);

        for (int i = 0; i < nr_slices; i++)
        {
                // Front to back compositing from the lights point of view
                Vector3f point_on_plane = begin + view_ray * (float) i * slab;

                Plane<float> plane(point_on_plane, plane_normal);

                Vector3f corners[6];
                int nr_corners;
                if (aabb.Intersect(plane, corners, nr_corners))
                {
                        Convex2D<float> conv(plane, corners, nr_corners);

                        // Pass one: Render the eye buffer

                        // Draw to the eye buffer
                        glDrawBuffer(GL_COLOR_ATTACHMENT0_EXT);

                        if (front_to_back)
                        {
                                // Front to back compositing
                                glBlendFunc(GL_ONE_MINUS_DST_ALPHA, GL_ONE);
                        }
                        else
                        {
                                // Back to front compositing
                                glBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
                        }

                        // Enable depth buffer tests
                        glEnable(GL_DEPTH_TEST);

                        // Setup the shader parameters
                        CGparameter param = cgGetNamedParameter(mPassOneFS, "volume_data");
                        cgGLSetTextureParameter(param, mVolumeTexture);
                        cgGLSetupSampler(param, mVolumeTexture);

                        param = cgGetNamedParameter(mPassOneFS, "shadow_texture");
                        cgGLSetTextureParameter(param, mShadowBuffer);
                        cgGLSetupSampler(param, mShadowBuffer);

                        param = cgGetNamedParameter(mPassOneVS, "eye_view_proj");
                        cgGLSetStateMatrixParameter(param, CG_GL_MODELVIEW_PROJECTION_MATRIX, CG_GL_MATRIX_IDENTITY);

                        param = cgGetNamedParameter(mPassOneVS, "light_view_proj");
                        cgGLSetMatrixParameterfr(param, &light_view_proj.m11);

                        param = cgGetNamedParameter(mPassOneVS, "aabb_min");
                        cgGLSetParameter3f(param, mLower.x, mLower.y, mLower.z);

                        param = cgGetNamedParameter(mPassOneVS, "aabb_max");
                        cgGLSetParameter3f(param, mUpper.x, mUpper.y, mUpper.z);

                        param = cgGetNamedParameter(mPassOneVS, "tex_ext");
                        cgGLSetParameter3f(param, mVolumeTextureExtent.x, mVolumeTextureExtent.y, mVolumeTextureExtent.z);

                        // Load the vertex shader
                        cgGLBindProgram(mPassOneVS);

                        // Load the fragment shader
                        cgGLBindProgram(mPassOneFS);

                        glBegin(GL_TRIANGLE_FAN);
                        {
                                glVertex3f(conv.mCenter.x, conv.mCenter.y, conv.mCenter.z);

                                for (int j = 0; j < nr_corners; j++)
                                {
                                        glVertex3f(conv[j].x, conv[j].y, conv[j].z);
                                }
                                glVertex3f(conv[0].x, conv[0].y, conv[0].z);
                        }
                        glEnd();

                        // Pass two: Update the light buffer

                        // Draw to the light buffer
                        glDrawBuffer(GL_COLOR_ATTACHMENT1_EXT);

                        // Front to back compositing
                        glBlendFunc(GL_ONE_MINUS_DST_ALPHA, GL_ONE);

                        // Enable depth buffer tests
                        glDisable(GL_DEPTH_TEST);

                        // Setup the shader parameters
                        param = cgGetNamedParameter(mPassTwoFS, "volume_data");
                        cgGLSetTextureParameter(param, mVolumeTexture);
                        cgGLSetupSampler(param, mVolumeTexture);
                        param = cgGetNamedParameter(mPassTwoFS, "obstacles_texture");
                        cgGLSetTextureParameter(param, mObstaclesTexture);
                        cgGLSetupSampler(param, mObstaclesTexture);

                        param = cgGetNamedParameter(mPassTwoVS, "light_view_proj");
                        cgGLSetMatrixParameterfr(param, &light_view_proj.m11);

                        param = cgGetNamedParameter(mPassTwoVS, "aabb_min");
                        cgGLSetParameter3f(param, mLower.x, mLower.y, mLower.z);

                        param = cgGetNamedParameter(mPassTwoVS, "aabb_max");
                        cgGLSetParameter3f(param, mUpper.x, mUpper.y, mUpper.z);

                        param = cgGetNamedParameter(mPassTwoVS, "tex_ext");
                        cgGLSetParameter3f(param, mVolumeTextureExtent.x, mVolumeTextureExtent.y, mVolumeTextureExtent.z);

                        // Load the vertex shader
                        cgGLBindProgram(mPassTwoVS);

                        // Load the fragment shader
                        cgGLBindProgram(mPassTwoFS);

                        // Render the slice
                        glBegin(GL_TRIANGLE_FAN);
                        {
                                glVertex3f(conv.mCenter.x, conv.mCenter.y, conv.mCenter.z);

                                for (int j = 0; j < nr_corners; j++)
                                {
                                        glVertex3f(conv[j].x, conv[j].y, conv[j].z);
                                }
                                glVertex3f(conv[0].x, conv[0].y, conv[0].z);
                        }
                        glEnd();
                }
        }

        // Disable the fragment shader
        cgGLDisableProfile(CgProfileFragmentShader());

        // Disable the vertex shader
        cgGLDisableProfile(CgProfileVertexShader());

        // Release framebuffer
        glPopAttrib();
        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);

        // Draw the framebuffer to the screen
        glBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA);

        // Blend the volume with the screen
        RenderTextureToScreen(mColourBuffer, 512, 512);

        // Disable blending
        glDisable(GL_BLEND);

        // Enable writing to the z-buffer
        glDepthMask(GL_TRUE);
}
```

Listing A.9: Volume Rendering

Bibliography

[BHN07] Robert Bridson, Jim Houriham, and Marcus Nordenstam. Curl-noise for procedural fluid flow. *ACM Trans. Graph.*, 26(3):46, 2007.

[BMF07] Robert Bridson and Matthias Muller-Fischer. Fluid simulation: Siggraph 2007 course notes. In *SIGGRAPH '07: ACM SIGGRAPH 2007 Courses*, pages 1–81, New York, NY, USA, 2007. ACM.

[CD05] Robert L. Cook and Tony DeRose. Wavelet noise. *ACM Trans. Graph.*, 24(3):803–811, 2005.

[CLT08] Keenan Crane, Ignocio Llamas, and Sarah Tariq. *Real-Time Simulation and Rendering of 3D Fluids*, volume GPU Gems 3, pages 633 – 675. Addison-Wesley Professional, 2008.

[FSJ01] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. Visual simulation of smoke. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 15–22, New York, NY, USA, 2001. ACM.

[IKLH04] Milan Ikits, Joe Kniss, Aaron Lefohn, and Charles Hansen. *Volume Rendering Techniques*, volume GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics, pages 667 – 692. Addison-Wesley Professional, 2004.

[KPHE02] Joe Kniss, Simon Premoze, Charles Hansen, and David Ebert. Interactive translucent volume rendering and procedural modeling. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 109–116, Washington, DC, USA, 2002. IEEE Computer Society.

[KTJG08] Theodore Kim, Nils Thürey, Doug James, and Markus Gross. Wavelet turbulence for fluid simulation. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages

1–6, New York, NY, USA, 2008. ACM.

[MCPN08] Jeroen Molemaker, Jonathan M. Cohen, Sanjit Patel, and Jonyong Noh. Low viscosity flow simulations for animation, 2008.

[NVI08] NVIDIA. Nvidia cuda compute unified device architecture programming guide. version 2.0. http://developer.download.nvidia.com/compute/cuda/2.0-Beta2/docs/Programming_Guide_2.0beta2.pdf, June 7 2008.

[OHL$^+$08] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[Per85] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, 1985.

[SFK$^+$07] Andrew Selle, Ronald Fedkiw, ByungMoon Kim, Yingjie Liu, and Jarek Rossignac. An unconditionally stable maccormack method. *Journal of Scientific Computing (in press)*, 2007.

[SH05] Christian Sigg and Markus Hadwiger. *Fast Third-Order Texture Filtering*, volume GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems), pages 313 – 317. Addison-Wesley Professional, 2005.

[SRF05] Andrew Selle, Nick Rasmussen, and Ronald Fedkiw. A vortex particle method for smoke, water and explosions. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 910–914, New York, NY, USA, 2005. ACM.

[Sta99] Jos Stam. Stable fluids. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 121–128, 1999.

[SWND05] Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2 (5th Edition) (OpenGL)*. Addison-Wesley Professional, 2005.