Exploring Parallelism for Real-Time Smoke Visualisation

Philippe C.D. Robert, Daniel Schweri

IAM-05-003

August 2005

Abstract

In this report we present a new system for visualising smoke an similar effects based on the Navier-Stokes equations. The system is optimised mainly for high rendering performances, targeting interactive real-time applications such as computer games or visual simulations. As such the system supports both static and moving obstacles of arbitrary shape. We demonstrate the effect of using SIMD operations when optimising the fluid solver and we introduce a parallel execution model for balancing the workload between multiple processor threads and the graphics hardware (GPU). Finally, we present four methods to visualise smoke and discuss their efficiency and the achieved visual realism.

CR Categories and Subject Descriptors: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

Keywords: Fluid Flow Rendering, Navier-Stokes, GPGPU

Contents

1 Introduction

In recent years smoke visualisation has become a topic of interest in many fields of computer graphics – e.g., visual simulations or special effects in movies and computer games. Consequently many efforts have been made in the field of visual fluid flow computation. Most of these recent efforts concentrate either on aspects of the physical simulation – e.g. the fluid solver – or how to implement a specific algorithm on programmable graphics hardware (GPU).

Our intention is to present a fluid flow rendering system based on this background, esp. J. Stam's contributions [20, 21, 22], which is suitable for integration into interactive, real-time applications. This implies that we cannot use the GPU or CPU exclusively to run our simulation, instead we have to balance the workload among all available resources. Thus, we focus on scalability and parallel rendering techniques. The contribution of our work is fourfold. First, we propose a parallel system for visualising smoke and other fluid flow based effects aimed at interactive applications. Our system is capable of balancing the computational tasks between the GPU and multiple threads running on the host processor(s). This facilitates the integration of our fluid flow rendering system into computational intensive applications. Second, we outline potential drawbacks and advantages of implementing a fluid solver using SIMD instructions. For this purpose we use a fluid solver based on the semi-Lagrangian method, which was initially introduced by Courant et al. [5]. Third, we describe an efficient approach to handle both static and moving obstacles in our smoke simulation. Finally we analyse various methods used to visualise the smoke, all of them geared towards high rendering performance.

This report is organised as follows: Section 2 covers important previous work in this field. In Section 3 we introduce our SIMD optimised fluid solver, then we describe the parallel execution mode mode in Section 4. In Section 5 we present multiple methods for visualising smoke, followed by the obtained results in Section 6. Finally some conclusions are presented in Section 7.

2 Background

In 1997 Foster and Metaxas [10] demonstrated the advantages of using the three-dimensional *Navier-Stokes equations* for generating motions of fluids. Jos Stam [20] then proposed an approach based on the *semi-Lagrangian method* for the simulation of unconditionally stable fluids in computer graphics. This method has been used by many others to simulate fluid flows of various kinds [8, 9, 6, 18]. Yoshida and Nishita [24] introduced a method of displaying swirling smoke, including the consideration of its passage round obstacles, using metaballs for representing the three-dimensional density distribution of smoke.

With the advent of programmable GPUs a few years ago, it became feasible to perform physically based simulations on GPUs using stream-based programming models. Bolz et al. [4] showed that numerical computations can be performed efficiently on the GPU by implementing a sparse matrix conjugate gradient solver and a regular-grid multigrid solver on a GeForceFX. Harris et al. [13] [12] employed graphics hardware to perform physically based simulation of fluids, clouds, and chemical reaction diffusion on GPUs. Batty et al. [3] presented a GPU-based preconditioned conjugate gradient solver used in a production quality fluid simulator, while Goodnight et al. [11] implemented a multigrid method for solving boundary value problems, such as systems of partial differential equations that arise in physical simulation problems like e.g. fluid flow. A framework for the implementation of linear algebra operators on GPU has been proposed by Krüger and Westermann [15]. They demonstrated their approach by implementing direct solvers for sparse

matrices with application to multi-dimensional finite difference equations, i.e. the incompressible Navier-Stokes equations. Li et al. [23] described the acceleration of the computation of Lattice-Boltzmann methods on graphics hardware by grouping particle packets into 2D textures and mapping the Boltzmann equations completely to the rasterization and frame buffer operations. The Lattice-Boltzmann model was then used to simulate smoke. Liu et al. [16] presented a way to process complex boundary conditions when simulating fluid flow using the Navier-Stokes equations on the GPU.

Other related work can be found on the website dedicated to *General-Purpose Computation using Graphics Hardware* (GPGPU)[1].

3 Fluid Flow

In physics fluid flow is commonly modelled using vector fields. The *Navier-Stokes equations* are the fundamental partial differentials equations that describe the flow of fluids.

$$\frac{\partial \vec{u}}{\partial t} = -(\vec{u} \cdot \nabla)\vec{u} + \nu \nabla^2 \vec{u} + f \tag{1}$$

The Navier-Stokes equations (1) describe the flow of incompressible fluids, where $\vec{u}$ is the velocity and ν the viscosity of the fluid.

Unfortunately most numerical solutions to solve these equations are very time consuming and thus not applicable for real-time usage. Stam addressed this problem by introducing a number of algorithms to solve these equations at high speed with less emphasis on physical accuracy [20]. He proposed a fluid solver based on the equation shown in (2) which is stable, never blows-up and can thus be used to apply large time-steps to the simulation. This is a crucial prerequisite for creating visual effects in real-time.

$$\frac{\partial \rho}{\partial t} = -(\vec{u} \cdot \nabla)\rho + \kappa \nabla^2 \rho + S \tag{2}$$

In equation (2) ρ represents the density of the fluid. Note that this is now a linear equation.

Consequently our implementation is based on Stam's work on real-time fluid dynamics [21, 22]. For the details concerning Stam's fluid solver we refer to the original articles. In the following subsections we present our modifications to Stam's original fluid solver, such as SIMD optimisations and the integration of obstacles.

3.1 SIMD Optimisations

To implement the fluid solver as proposed by Stam we rely on a spatial discretisation which we create by dividing the computation domain into identical voxels. Of course, the resolution of this discretisation affects not only the visual quality but also the computational complexity of the simulation. For real-time usage it is therefore crucial that the calculation is performed with maximum efficiency, hence, we optimise our solver using SIMD operations. Unlike some others we have decided not to implement the fluid solver on the programmable GPU but on the host CPU by using *Intel's Streaming SIMD Extensions* (SSE). We made this choice for the following reasons: First, a GPU based implementation would have forced us to implement the entire system on the GPU, hence, a parallel system as described in this report would have become almost impossible. Second, a CPU based implementation gives us the flexibility to use the data type `double` which is not possible on

[1] http://www.gpgpu.org/

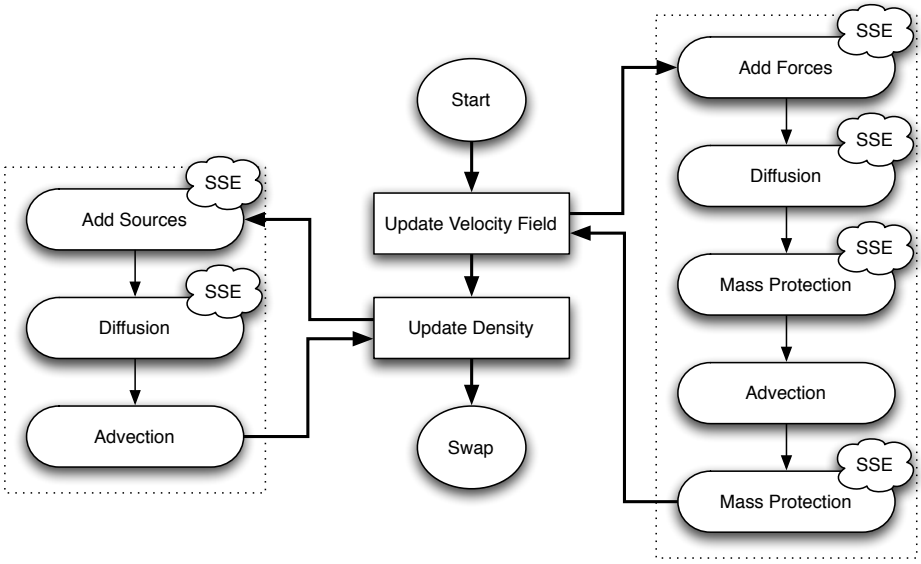

Figure 1: The execution model of the fluid solver

current generation GPUs, and third, the integration of moving obstacles becomes much easier.

The fluid solver is based on an iteration method, the solver starts with an initial set of values and thereafter updates the velocity for every time step by solving Equation 2. Changes to the velocity field are therefore caused by external forces, viscous diffusion and self-advection. Additional forces and density sources can be added to the system at any time during the simulation. We store the fluid's density and velocity values as well as additional forces in an aligned 3D grid which is appropriate to the SIMD programming model. Figure 1 depicts the execution model of our fluid solver and denotes those steps which are accelerated using SSE instructions. The operations which are most influential are:

- *Diffusion* controls the exchange of density and velocity values from a grid cell to its 6 neighbours. This function is implemented using the *Gauss-Seidel relaxation* and can only partially be optimised using SIMD operations: we can use SSE to compute the sum of the three neighbour voxels from the last iteration step. The sum of the other three elements has to be computed normally.

- *Mass protection* is based on the *Hodge decomposition of vector fields*, which says that each velocity field is the sum of a mass protecting field and a gradient field. This function is fully accelerated using SSE operations.

- *Advection* controls the influence of the velocity field on the density distribution and the velocity field itself. This function cannot be mapped easily to the SIMD programming model.

Moreover, the addition of external sources and forces to the system can be implemented using SSE operations as well. However this has only little impact on the achieved performance improvements.

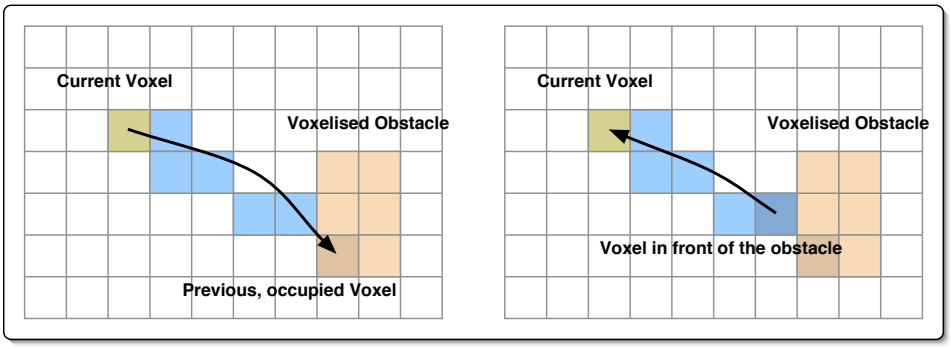

Figure 2: Tracing backward through a velocity field and updating the density value using the correct voxel

3.2 Obstacles

Our system provides support for static and moving obstacles of arbitrary shape by using efficient boundary conditions, extending the ideas from Stam's work. This allows for realistic interaction with the fluid flow in real-time. During simulation the system thus adheres to the following two constraints at any time:

1. Smoke must flow freely and without interference tangential to the obstacle
2. Smoke must never flow into an obstacle

The second constraint simply implies that density $\delta_i = 0$ where $Voxel_i$ is occupied by an obstacle. In the following we outline how obstacles are implemented in our system.

3.2.1 Static Obstacles

At the beginnig of every simulation the system marks the voxels of the grid which are taken by a static obstacle. This preprocessing step is implemented using common object-voxel intersection algorithms. The density value δ of these voxels is then set to be always 0. Obviously this affects the computation of the diffusion and advection. When computing the diffusion of the smoke densities we have to skip the occupied voxels. Hence, when calculating the density mean values of the neighbours of a voxel, those with $\delta = 0$ have to be left out in order to get correct results. Note that this does not apply to the computation of the diffusion of the velocity.

Furthermore, special care has to be taken when performing the advection of the density values. Stam explained how to solve this problem by tracing virtual particles backward in time through the velocity field [22]. Unfortunately, by enabling support for obstacles this algorithm may point to occupied voxels and thus lead to incorrect results - e.g., resulting in *shadow images* of the rendered obstacles. To avoid this problem we introduce a slightly modified version of this algorithm.

First, we detect the occupied voxels by rasterising a line between the current and the previous voxel in our grid. Then we test every voxel which is hit by that line against obstacle occupation. If we detect an obstacle, we use the density value of the voxel in front of the occupied voxel along the rasterised line to perform the advection operation. This procedure is depicted in Figure 2. Line rasterisation is implemented using a three-dimensional version of the Bresenham algorithm. Moreover, this procedure makes sure that no obstacle is accidentally missed when tracing backward through a velocity field using large time steps.

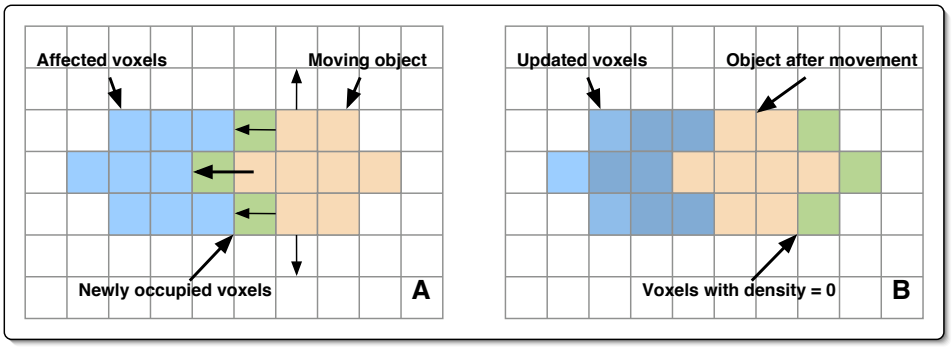

Figure 3: Updating voxel states when moving an obstacle

3.2.2 Moving Obstacles

Naturally, object motion affects the fluid particles located nearby the object by moving them around. In addition, when an object stops moving the fluid particles continue their motion, which decreases in time due to inertia and friction. We propose an algorithm to model this effect in two steps. The approach is depicted in Figure 3.

First, we compute the effect on the smoke particles around the moving object. This is achieved by adding the density values of the newly occupied voxels to the voxels affected by the motion.

Second, we model the effect of object motion on the velocity field. In case of an object translation we calculate the velocity vector of the moving object, and based on that determine the surrounding voxels which will be affected by this motion. We do this using the moving direction and the normal vectors of all object faces which are pointing in the moving direction. The bigger the angle between moving direction and normal vector, the smaller the impact on the nearby voxels and vice versa. In the case of an object rotation, we have to do the very same procedure using velocity vectors for every voxel occupied by the obstacle. Once we have determined the affected voxels we add the velocity of the moving object to these voxels.

4 Parallel Rendering

In order to balance the work load between the host processor(s) we use a multi-threaded execution model based on POSIX threads. With the advent of multi-core CPUs and simultaneous CPU multi-threading technologies, e.g., Intel's Hyper-Threading [17], this approach promises noticeable performance improvements and better scalability.

Our parallel execution model uses the main thread to perform all OpenGL rendering and event handling tasks, a second worker thread is used to perform the fluid solver. A third thread may be spawned to perform the aforementioned grid updates when dealing with obstacles in motion. Alternatively this task can also be executed by the main thread. Note that parallel rendering results in 1 or 2 frames of latency, respectively. In most cases this is tolerable when the achieved frame rate is high enough. In addition we can utilise a GPU shader program to perform further tasks depending on the visualisation mode – e.g., the computation of interpolation values when rendering smoke using particles (see Section 5). Depending on the application and the available GPU model we can use either a vertex shader or a fragment shader for this purpose. Figure 4 shows this particular scenario.

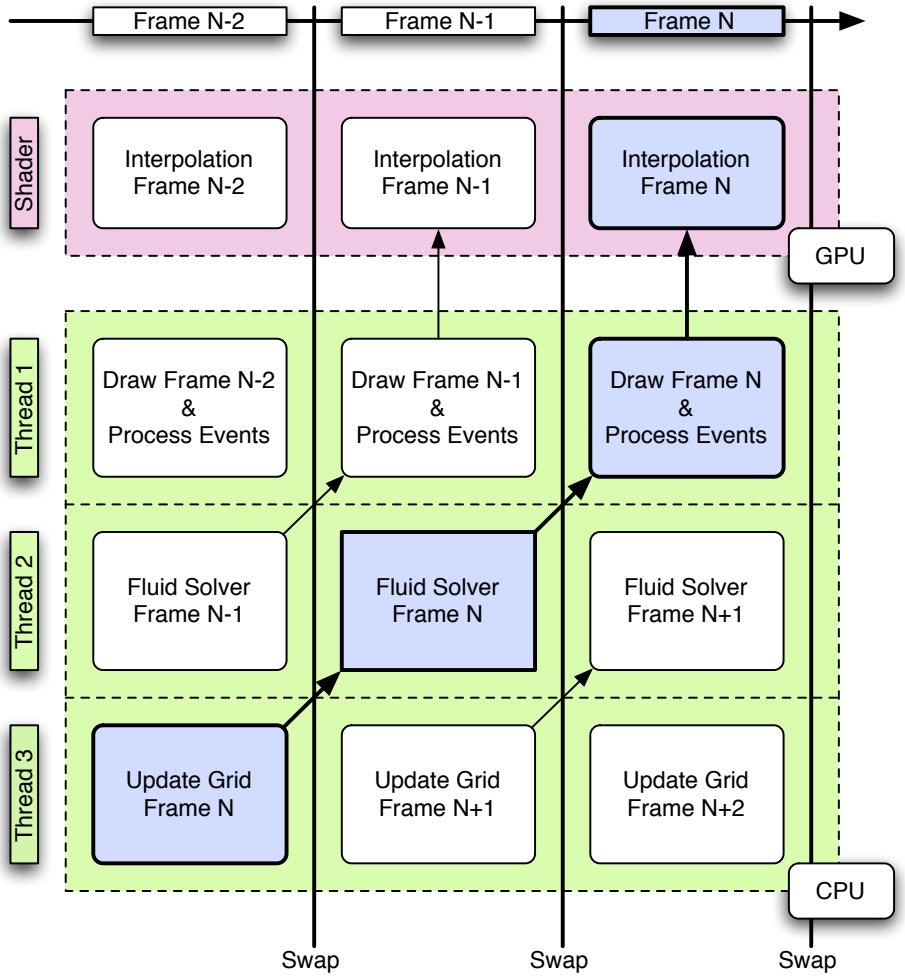

Figure 4: The parallel execution model using 3 CPU threads and 1 GPU shader program. The blue shaded rectangles depict the control flow when rendering frame N.

Please note that our parallel execution model does not require data readback operations from the GPU to the host system. It is therefore perfectly applicable on AGP bus based systems without severe performance degradations.

5 Visualisation

We have implemented and analysed several hardware accelerated methods to visualise fluid flows such as smoke and fire, for example. All of these approaches are tailored for high rendering performance and a high level of visual realism. Although it is evident that the visual quality depends on the resolution of the voxel grid; e.g., vortices are barely visible at low resolutions typically used in our testing scenarios.

5.1 Points

Our first visualisation mode is based on OpenGL point rendering. Instead of rendering the $n \times m \times p$ grid voxels as cubes we subdivide every grid voxel in q subvoxels and then draw one point at the centre of each such subvoxel. Note that this results in an equal distribution of points. Other, more sophisticated distributions might lead to even better visual results.

The colour of each point is determined by interpolating the density values of the 6 surrounding grid subvoxels using a weighted sum operation. This interpolation step is required to avoid disturbing aliasing effects and can be performed either on the CPU or on the GPU. We have implemented the GPU based colour interpolator both as fragment and as vertex shader program. While fragment shaders are usually more powerful they can slow down the rendering if many fragments have to be processed, e.g., when zooming close to the smoke. This does not happen when using a vertex shader based implementation, because the number of points remains constant.

```
float4 main( in float4 d1 : TEXCOORD0, // density 1
             in float3 d2 : TEXCOORD1, // density 2
             in float4 f1 : TEXCOORD2,
             in float3 f2 : TEXCOORD3 ) : COLOR
{
    const float grey = F * (dot(f1,d1) + dot(f2,d2));
    return float4(grey, grey, grey, alpha);
}
```

Figure 5: Colour interpolation implemented as fragment shader using Cg

The fragment shader implementation is straightforward, it calculates the weighted sum of the passed density values and returns the computed colour. Performing a lot of texture lookup operations slows down the shader execution performance, as demonstrated by Fatahalian et al. [7], we therefore pass all required input parameters as `float4` values using multi-texture coordinates. Listing 5 shows the Cg version of the colour interpolator implemented as a fragment shader program. Note that the vertex shader implementation is very similar to the fragment shader version, the only difference being the additional computation of the Model-View-Matrix transformation. The complete listing is shown in Appendix A.

Finally, the points are all rendered in no particular order. To avoid visual artefacts we use additive blending. However, since additive blending requires that all points are drawn no matter whether they are occluded by other points, depth buffering cannot be applied as usual. Hence, special care has to be taken when dealing with obstacles. We propose two methods to handle obstacles correctly:

1. Asynchronous OpenGL *occlusion queries* are used to detect the visibility of our grid subvoxels.

2. While rendering the points the depth buffer is disabled for writing. Thus all points in front of an obstacle are rendered as intended.

Please note that we also investigated variations of this visualisation method, i.e. using textured and non-textured disks/splats or point sprites using the OpenGL `GL_POINT_SPRITE_ARB` extension. Unfortunately these approaches did not result in convincing visual results because of coarse aliasing effects caused by overlapping rendering primitives.

5.2 Volume Rendering

Our second visualisation mode is based on traditional volume rendering techniques using 3D-texture mapping and does not depend on programmable shaders. This method therefore also runs with decent performance on elder graphics hardware.

The implementation is fairly simple, we store the calculated density values in a 3D texture and render n slices back-to-front through the grid with texturing enabled. The intersection points of the slices with the grid hull can further be used as 3D texture coordinates.

One limitation of this method is that the size of the 3D texture image must be $2^n + 2$ in each dimension, the resolution of the grid has to be defined accordingly. Moreover, for optimal visual results we apply bilinear filtering while rendering.

5.3 Section Planes

Our third method is similar to volume rendering as described above, but implemented as a fragment shader program. The algorithm goes as follows:

1. We store the density values for one frame in a 32bit floating-point 2D texture.
2. Then we intersect n planes orthogonal to the viewing direction with the grid hull. The resulting polygons are rendered with texturing and fragment shading enabled.
3. Finally, the fragment shader program assigns every fragment of these polygons the appropriate density value. This is done using the texture coordinates of the polygon which are automatically interpolated by the GPU for every fragment.

The visual quality of this approach directly depends on the number of rendered section planes. Unfortunately a high plane count slows down the rendering because of the vast number of fragments which have to be processed. Also note that we do not apply any interpolation or filtering because this would result in many more texture look-up operations and thus decrease the performance even more.

5.4 Impostors

To reduce the total number of fragment shader passes and the geometry data which has to be sent to the GPU we propose a method which is based on *impostors* [19, 14]. An impostor replaces the rendered grid with a billboard, textured with an image of the smoke from a certain point of view.

As opposed to the method described in 5.3 we only render one oriented plane which is close to the viewer, as shown in Figure 6. For every fragment of this plane the fragment shader program determines the grid voxels which are intersected by the ray going from the viewer through the respective fragment. Based on the resulting density values it calculates the final colour.

Please note that the implementation of this shader program results in a high instruction count caused by the unrolling of the loop to sum all density values. This prohibits the shader to run on elder graphics hardware with low shader length limits. This gets even worse when adding density interpolation to the computation.

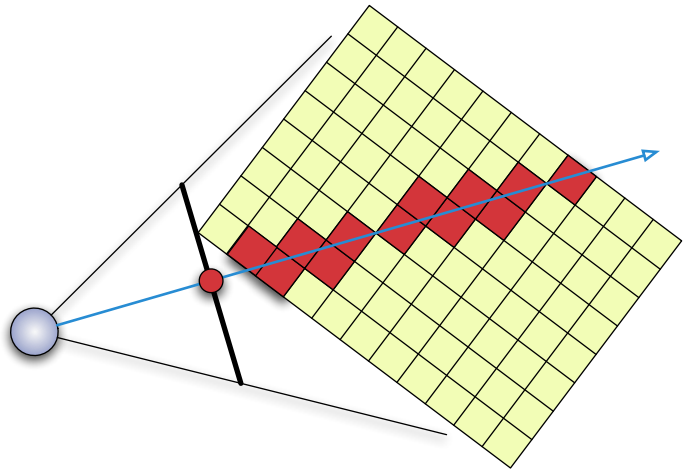

Figure 6: The projection of the grid densities onto a plane

6 Results

In this section, we analyse the efficiency of our parallel, SEE optimised smoke visualisation system. Furthermore, we compare the visual quality and efficiency of the visualisation methods described in Section 5 using two different test scenes.

The results presented in this report are collected on a PC workstation running Windows ™ XP with the following configuration:

- AMD Sempron™ 1.83 GHz
- 1.25 GB main memory
- ATI Radeon™ 9600 XT GPU (AGP), 500 MHz, 256 MByte VRAM

Additionally, parallel rendering is tested and benchmarked on a dual Xeon 2.8GHz PC with a Nvidia GeforceFX GPU running Fedora Core 2 Linux.

6.1 Solver Performances

The solver performance has the biggest impact on the overall rendering performance. Therefore, it is crucial to optimise the fluid flow computation as much as possible, especially when performing three dimensional simulations.

Ideally, it is possible to speed-up the computations by a factor 4 using the SIMD programming model. In reality this is almost never the case – e.g., because it is not possible to use SSE instructions exclusively or due to a less optimal memory layout. We were able to accelerate our fluid solver by roughly 40 percentage using SSE intrinsics compared to the traditional implementation. This is not as much as we have hoped to achieve, but it is nevertheless a good speed-up factor. Our results are depicted in Figure 7.

We believe that it is possible to increase this factor even more by using a memory layout which is more optimised for SIMD usage – e.g., by aligning all solver related data to 16 byte[2] and by using SSE assembler instructions to implement the performance critical parts.

[2]Currently we still use some read operations on 4 byte-aligned memory which forces us to use the `_mm_loadu_ps` intrinsic.

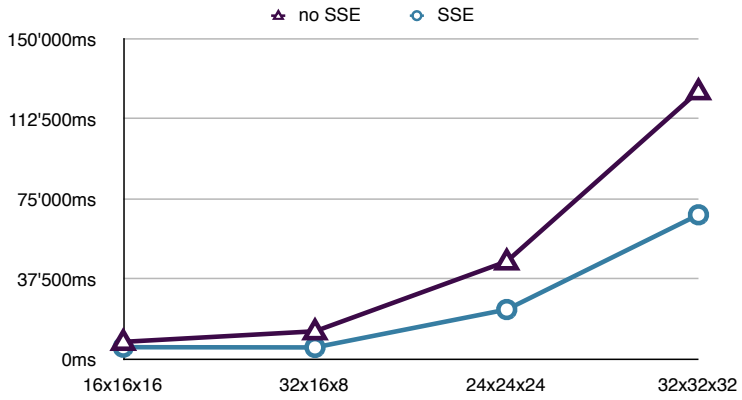

Figure 7: Time to compute 1000 frames depending on various grid resolutions

6.2 Visualisation

In general, we found that the volume rendering and point rendering modes lead to the best visual results while providing the highest frame rates – an example is shown in Figure 8. It even seems that volume rendering is more efficient than point rendering, most likely due to the high number of fragments which have to be processed when interpolating the density values on the GPU.

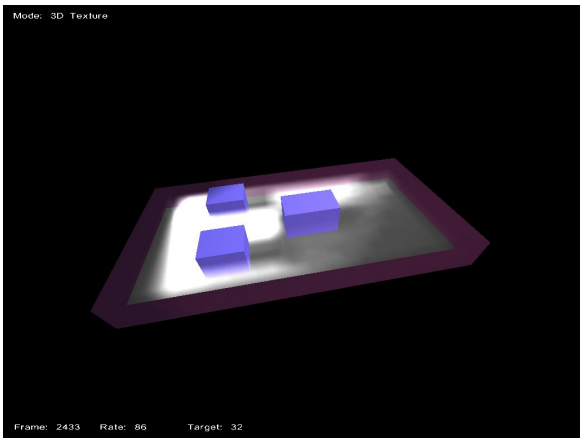

Figure 8: Smoke flowing around obstacles, rendered using a 3D texture

Our 3rd visualisation mode (5.3) suffers from a general lack of filtering. Unfortunately, this deficiency can only be compensated using a high number of section planes, which slows down the rendering performance noticeably. Finally, impostors provide good visual quality when applying filtering, but rendering performances are not competitive due to the loops and texture access operations of the fragment shader program.

Visualising Obstacles

As explained in Section 5.1 we have implemented two methods to visualise obstacles properly. It turns out that the method based on occlusion queries is much slower than the method based on depth testing operations. This outcome can be

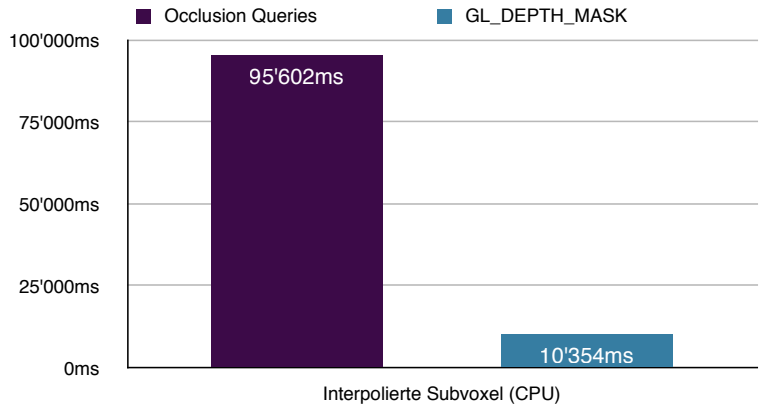

Figure 9: Time to render 1000 frames using occlusion queries and depth testing operations

explained by the fact that occlusion query results require an extra roundtrip to the graphics card - in our case via the AGP graphics bus - which is a major bottleneck, whereas otherwise only depth buffer write operations have to be disabled using `GL_DEPTH_MASK`. Figure 9 shows our results.

Parallel Rendering

As explained in Section 4, our POSIX thread based rendering mode allows us to perform the fluid flow computations and the OpenGL rendering in parallel. Using a dual processor system or a dual core CPU this can lead to a performance increase of factor 2, as shown in Table 1.

	$16x16x16$	$32x32x32$	$48x48x48$
3D texture, 2 threads	104	72	71
3D texture, 1 thread	63	14	5
GPU based interpolation, 2 threads	42	13	13
GPU based interpolation, 1 thread	47	7	4
CPU based interpolation, 2 threads	27	14	14
CPU based interpolation, 1 thread	25	9	4

Table 1: Frame rates (fps) for various grid sizes when using multiple POSIX threads on a dual processor PC

We also take advantage of the GPU as maths coprocessor by off-loading certain tasks to the vertex or fragment shader units. In the examples shown in Table 1 the GPU based interpolation has been benchmarked using a vertex shader program. We noticed that on lower end GPUs the vertex shader based implementation is usually faster than the fragment shader based implementation[3], most probably due to a reduced number of pixel pipelines. Performance numbers based on various point sizes showing this effect are depicted in the following Figure 10.

[3]Benchmarked using the same camera settings.

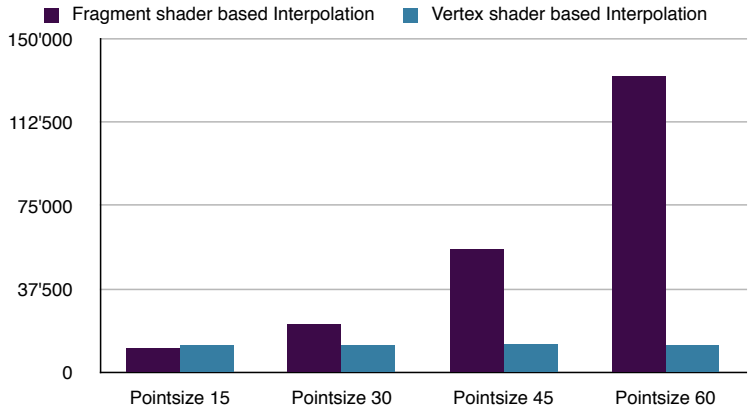

Figure 10: Performance results for fragment and vertex shader based interpolation

Fluid Flow Effects

As noted before, our fluid flow renderer can easily be configured to render not only smoke but also fire, plasma and other effects which can be simulated using fluid flow computations. One such example is shown in Figure 11.

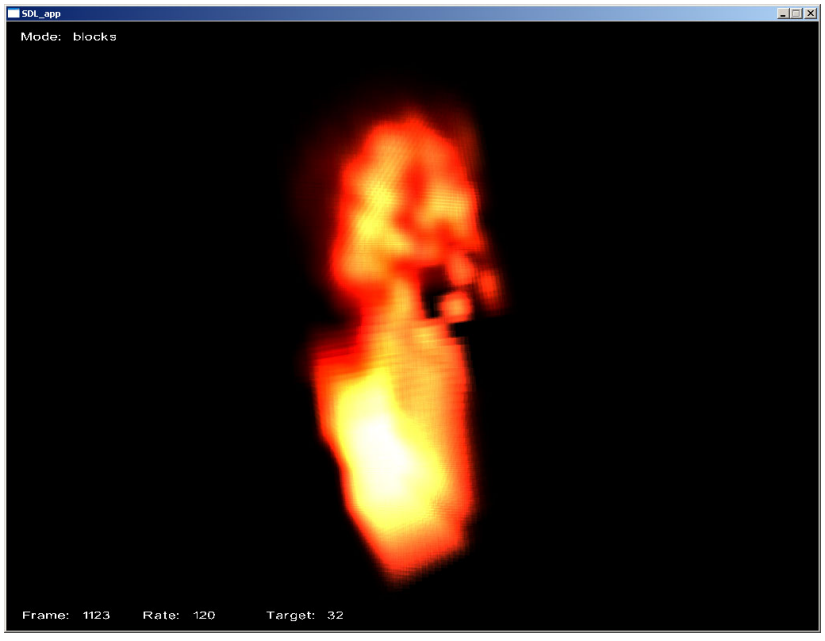

Figure 11: Rendering lava

7 Conclusions and Outlook

In this report we have presented a parallel system for visualising smoke and other effects based on fluid flow computations. We demonstrated the advantage of using a SIMD optimised fluid solver and presented several methods to visualise the simulated smoke. We showed that a well balanced system can lead to high frame rates, which is a prerequisite for games and interactive visual simulations. We have also shown that shader programs – even simple ones – have to be written carefully in order to achieve high frame rates; e.g., loops still impose a problem when writing efficient shader programs.

This research effort can be extended in many directions; e.g., we currently use points that are positioned according to an equal distribution. More sophisticated distributions could lead to better visual results. Also, to optimise the achieved visual realism we are investigating methods to apply self shadowing and light scattering effects.

Just recently, AGEIA Technologies, Inc. announced *PhysX*, the first commercially available Physics Processing Unit (PPU) [2] . It would be interesting to explore its usage for implementing hardware-accelerated fluid solvers.

References

[1] General-Purpose Computation using Graphics Hardware.
`http://www.gpgpu.org`

[2] AGEIA. Physics, gameplay and the physics processing unit, March 2005.

[3] C. Batty, M. Wiebe, , and B. Houston. High Performance Production-Quality Fluid Simulation via NVIDIA's QuadroFX. Technical report, Frantic Films, 2003.

[4] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.

[5] R. Courant, E. Isaacson, and M. Rees. On the solution of nonlinear hyperbolic differential equations by finite differences. *Comm. Pure. Appl. Math.*, 5:243–255, 1952.

[6] D. Enright, S. Marschner, and R. Fedkiw. Animation and Rendering of Complex Water Surfaces. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 736–744, 2002.

[7] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication. In *Proceedings of Graphics Hardware 2004*, 2004.

[8] R. Fedkiw, J. Stam, and H. W. Jensen. Visual Simulation of Smoke. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer Graphics and Interactive Techniques*, pages 15–22, 2001.

[9] N. Foster and R. Fedkiw. Practical Animation of Liquids. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 23–30, 2001.

[10] N. Foster and D. Metaxas. Modeling the Motion of a Hot, Turbulent Gas. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 181–188, 1997.

[11] N. Goodnight, C. Woolley, G. Lewin, D. Luebke, and G. Humphreys. A Multigrid Solver for Boundary Value Problems using Programmable Graphics Hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 102–111, 2003.

[12] M. J. Harris, W. V. Baxter, T. Scheuermann, and A. Lastra. Simulation of Cloud Dynamics on Graphics Hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 92–101, 2003.

[13] M. J. Harris, G. Coombe, T. Scheuermann, and A. Lastra. Physically-Based Visual Simulation on Graphics Hardware. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 109–118, 2002.

[14] S. Jeschke. *Accelerating the Rendering Process Using Impostors.* PhD thesis, Vienna University of Technology, March 2005.

[15] J. Krüger and R. Westermann. Linear Algebra Operators for GPU Implementation of Numerical Algorithms. *ACM Trans. Graph.*, 22(3):908–916, 2003.

[16] Y. Liu, X. Liu, and E. Wu. Real-Time 3D Fluid Simulation on GPU with Complex Obstacles. *Proc. PG'04*, pages 247–256, October 2004.

[17] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-Threading Technology Architecture and Microarchitecture. Technical Report 3, Intel, 2002.
`http://www.intel.com/technology/hyperthread/`

[18] N. Rasmussen, D. Q. Nguyen, W. Geiger, and R. Fedkiw. Smoke Simulation for Large Scale Phenomena. *ACM Trans. Graph.*, 22(3):703–707, 2003.

[19] F. Sillion, G. Drettakis, and B. Bodelet. Efficient Impostor Manipulation for Real-Time Visualization of Urban Scenery. *Computer Graphics Forum*, 16(3):C207–C218, 1997.

[20] J. Stam. Stable Fluids. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer Graphics and Interactive Techniques*, pages 121–128, 1999.

[21] J. Stam. Interacting with Smoke and Fire in Real Time. *Commun. ACM*, 43(7):76–83, 2000.

[22] J. Stam. Real-Time Fluid Dynamics for Games. *Proceedings of the Game Developer Conference*, 2003.

[23] X. Wei, W. Li, and A. Kaufman. Implementing Lattice Boltzmann Computation on Graphics Hardware. *The Visual Computer*, 2003.

[24] S. Yoshida and T. Nishita. Modelling of Smoke Flow Taking Obstacles into Account. *Pacific Graphics*, pages 135–144, 2000.

A Vertex Shader based Interpolation

```
struct a2v {
    float4 densI      : TEXCOORD0;
    float3 densII     : TEXCOORD1;
    float4 f1         : TEXCOORD3;
    float3 f2         : TEXCOORD2;
    float4 Position   : POSITION; //in object space
};

struct v2f {
     float4 Position  : POSITION; //in projection space
     float4 Color     : COLOR0;
};

v2f main( in a2v IN, uniform float4x4 ModelViewMatrix )
{
    v2f OUT;
    OUT.Position = mul(ModelViewMatrix, IN.Position);
    const float grey = 1.6* ( dot(IN.f1,IN.densI ) +
                              dot(IN.f2,IN.densII) );
    OUT.Color = float4(grey, grey, grey, 0.6);
    return OUT;
}
```

Listing 1: Vertex shader for interpolated subvoxels