Interactive Fluid-Particle Simulation using Translating Eulerian Grids

Jonathan M. Cohen* NVIDIA
Sarah Tariq NVIDIA
Simon Green NVIDIA

Abstract

We describe an interactive system featuring fluid-driven animation that responds to moving objects. Our system includes a GPU-accelerated Eulerian fluid solver that is suited for real-time use because it is unconditionally stable, takes constant calculation time per frame, and provides good visual fidelity. We dynamically translate the fluid simulation domain to track a user-controlled object. The fluid motion is visualized via its effects on particles which respond to the calculated fluid velocity field, but which are not constrained to stay within the bounds of the simulation domain. As particles leave the simulation domain, they seamlessly transition to purely particle-based motion, obscuring the point at which the fluid simulation ends. We additionally describe a hardware-accelerated volume rendering system that treats the particles as participating media and can render effects such as smoke, dust, or mist. Taken together, these components can be used to add fluid-driven effects to an interactive system without enforcing constraints on user motion, and without visual artifacts resulting from the finite extents of Eulerian fluid simulation methods.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Physically based modeling;

Keywords: GPU computing, fluid simulation, particle simulation

1 Introduction

Within the field of computer graphics, a variety of successful algorithms have been developed for high quality simulation of fluid phenomena. These methods have been used with great effect in visual effects and for other off-line applications. While many methods have been demonstrated to run interactively in certain situations, they have not been used in widely deployed interactive applications such as video games. However, two recent developments have the potential to alter the applicability of interactive fluid dynamics: the rise of highly programmable high-throughput consumer GPUs, and the development of high quality, efficient, and unconditionally stable methods for evolving 3D fluid systems in time.

This paper presents an approach that exploits these recent algorithmic and architectural advances in order to enhance interactive applications with fluid-driven effects. Our contribution is not simply a collection of algorithms from the existing literature, but a complete system-level view of the various issues involved in making fluid simulation practical and visually impressive on modern hardware. In particular, we develop an algorithmic framework that will allow for increasing visual fidelity over the next several hardware generations as the throughput of consumer parallel processors increases.

Many existing methods will not hold up under the extreme performance and robustness requirements of interactive systems like games. For example, while preconditioned conjugate gradients is a good choice for many applications, multigrid methods are inherently more parallel. Consequently, they perform very well on modern parallel platforms, allowing for medium size problems to be solved at interactive rates on current consumer hardware. As another example, many popular numerical methods are formally stable, but actually decay to stable white noise under certain conditions. Care must be taken to avoid all such instances of instability, while maintaining realistic response to user inputs. The system we present has been shown to be extremely robust, capable of running for several hours under constant use without any noticeable instabilities.

In addition to high visual quality and stability, simulation algorithms used in interactive applications must have predictable and constant computation time and robustness to all possible user inputs. For deployment in popular applications such as games or online virtual worlds, ease of integration with existing software architectures and scalability across a range of consumer hardware platforms are required. Even more important than these practical considerations is the aesthetic desire to get beyond the “fluid-in-a-box” look typically associated with performing complex fluid calculations inside a fixed rectilinear domain, where the fluid effects stop at the edge of the domain.

Particle-based methods like Smoothed Particle Hydrodynamics (SPH) are attractive because they do not suffer from the limitation to be inside a box. However, exactly enforcing incompressibility is important for accurate production of turbulence, and SPH methods have a hard time enforcing incompressibility efficiently. They also can have difficulty allocating computational elements throughout space efficiently. For these reasons, they have not been demonstrated to be effective for calculating single phase flows such as air around a car. Vortex-based Lagrangian methods enforce incompressibility easily and can produce turbulent effects. But they are difficult to implement in 3D, especially for the types of complex dynamic boundary conditions that we are interested in.

Eulerian methods can be made interactive for complex flows [Crane et al. 2007] and can efficiently generate turbulent effects. However, they typically are confined to a finite rectilinear domain and therefore are hard to use in large environments with unconstrained fluid motion. Grid reshaping and adaptive resolution techniques exist for Eulerian methods, but these techniques are slower than regular grid methods due to their use of sparse data structures, and to date have not been demonstrated to run at interactive rates.

We demonstrate a method that combines the high fidelity and high performance of modern Eulerian methods with the flexibility and controllability of particle systems. By moving all data-intensive computation onto the data-parallel processor, the GPU, we achieve results that will scale nearly linearly with the number of GPU processing cores. As demonstrated in this paper, current high-end GPU solutions are capable of producing high quality fluid motion at around 25 frames per second. Since GPU core counts are increasing roughly with Moore’s Law (doubling every hardware generation), and the performance of our system will scale linearly with floating point throughput, our approach will become increasingly practical due to hardware trends. While our system could be dropped into current games at low resolution, because we are riding an exponential performance curve, these methods will be practical on mid-range consumer hardware at high quality in a few years.

*{jocohen,stariq,sgreen}@nvidia.com

The major contributions of this paper are:

- a collection of techniques for decoupling grid-based fluid calculations from a grid-free particle simulation,
- a highly robust Eulerian fluid simulation algorithm designed to run efficiently on parallel GPU architectures, and
- an optimized particle engine suitable for parallel architectures.

We begin with a review of related work before explaining the algorithms used in our system and their implementations.

2 Related Work

Since Stam's Stable Fluids paper [Stam 1999], there have been several real-time applications for 3D fluid solvers, including follow-up work by Stam himself [Stam 2003]. Real-time simulation techniques based on a variety of other methods have been developed, including particle-based hydrodynamics [Muller et al. 2003], Lattice Boltzman models [Zhao et al. 2007], wave particles [Yuksel et al. 2007], model reduction [Wicke et al. 2009], and pseudo-compressible schemes [West 2007].

We make the simplifying assumption that the fluid velocity field can be split into near-field region which requires simulation, and a far-field region which can be handled via boundary conditions. Separations of this type are used in a number of scientific applications, for example to track local atmospheric features such as hurricanes or tornadoes [Klemp and Wilhelmson 1978].

A number of authors have described GPU implementations of incompressible inviscid Navier-Stokes or other fluid models At first, implementing a numerical algorithm on a GPU required mapping different stages of the algorithm to graphics operations, and using a graphics API as the programming model [Goodnight et al. 2003; Bolz et al. 2003; Crane et al. 2007]. The fire simulation method presented in [Horvath and Geiger 2009] achieves impressive performance and quality by using an OpenGL-based fluid solver. However, it is designed to work from only a single camera view and therefore would not be appropriate for most interactive 3D applications.

More recent APIs such as NVIDIA's CUDA allow for direct control over the GPU hardware. This had led to a large number of GPU-based fluid dynamics implementations, primarily in the engineering and scientific computing fields [Phillips et al. 2009]. This body of literature is growing rapidly, and we only cite one representative article.

We use particles to visualize our fluid simulations. Particle systems have been implemented on GPUs for interactive applications [Drone 2007; Kipfer et al. 2004] There are hundreds of implementations of particle systems for interactive applications and cinematic visual effects systems, but most implementation details are unpublished.

Volume rendering is often paired with fluid simulation as a means to visualize fluid motion or visualize scalar field quantities because it aids in understanding complex 3D motion. Stam implemented a simple scattering model for real-time visualization and used ray casting to calculate self-shadowing [Stam 1999]. This work was expanded into a more realistic model [Fedkiw et al. 2001]. Deep shadow maps [Lokovic and Veach 2000] are often used to precompute attenuation along light rays for volume rendering. While we traverse the volumetric elements in an order that allows for only a single slice of the shadow map to be stored at a time, several authors have demonstrated how to map similar algorithms to graphics hardware by calculating and storing the entire shadow map [Kim and Neumann 2001; Hadwiger et al. 2006].

3 System Overview

The user freely moves an object, in our case a vehicle, through a virtual world. Our goal is to create the visual illusion that the car is moving through an infinite fluid, and to simulate the effects arising from the car's displacement of air. Rather than fill the entire world with an enormous simulation grid, we only perform a simulation in a small region around the object of interest. Other dynamic objects that are not under direct user control, such as dust particles or debris, are influenced by the simulated fluid when they are near the object of interest, and are animated via simpler methods when farther away. Figure 1 shows screen captures from our interactive car simulator, with the extent of the fluid simulation domain visualized in panels (a) and (b). As can be seen in panel (c) and on the accompanying video, visible artifacts due to the finite extents of the simulation domain are minimal and typically not distracting.

Figure 2 shows a schematic overview of our system. The system consists of three components: a GPU-optimized Eulerian incompressible inviscid Navier-Stokes solver, a GPU-optimized particle engine, and a hardware volume rendering system. The Navier-Stokes solver, described in Section 5, uses numerical methods that have guaranteed stability and constant per-frame computation time, but are accurate enough to capture visually important flow features. By exploiting Galilean invariance, we allow the fluid domain to translate so as to follow the moving car. The particle system, described in Section 6, is driven by the calculated fluid velocity field for particles inside the fluid domain, and follows other rules such as Newtonian dynamics for particles outside the fluid domain. Because the transition from inside to outside is seamless, the actual extents of the fluid domain are visually obscured. When the particles are used to simulate participating media such as dust or smoke, we render the particles via a volume rendering system, described in Section 7.

4 GPU Architecture

Our solver is implemented using the CUDA 2.0 parallel programming platform from NVIDIA [NVIDIA Corporation 2008] and is designed to run on GT200-series NVIDIA GPUs [Lindholm et al. 2008]. The GT200 architecture consists of an array of 240 stream processors, divided into 30 multiprocessors with 8 cores each. GT200 is optimized for overall computational throughput rather than the latency experienced by any particular thread. Up to 30,720 threads may be in flight simultaneously, with the large amount of multithreading used to hide latency for accessing off-chip RAM. CUDA programs are called *kernels*, and are executed in Single-Program, Multiple-Data (SPMD) mode, with all threads running the same program simultaneously.

Most of the routines in our fluid solver are memory bandwidth-limited. Therefore our optimizations focus on organizing memory accesses to use available bandwidth as efficiently as possible. On GT200, threads are grouped into batches of 32 called *warps* that execute in lockstep SIMD fashion. A warp is made up of two half-warps, threads 0-15 and threads 16-31. If threads in the same half-warp read from the same 64-byte cache line in the same cycle, these reads are batched into a single vector load via a process known as *memory coalescing*. Consequently, uncoalesced loads and stores waste $15/16^{ths}$ of off-chip bandwidth.

GT200 also has a small read-only cache called the *texture cache*, and a small managed cache called *shared memory*. With thousand of simultaneous active threads, data will only live in the cache for

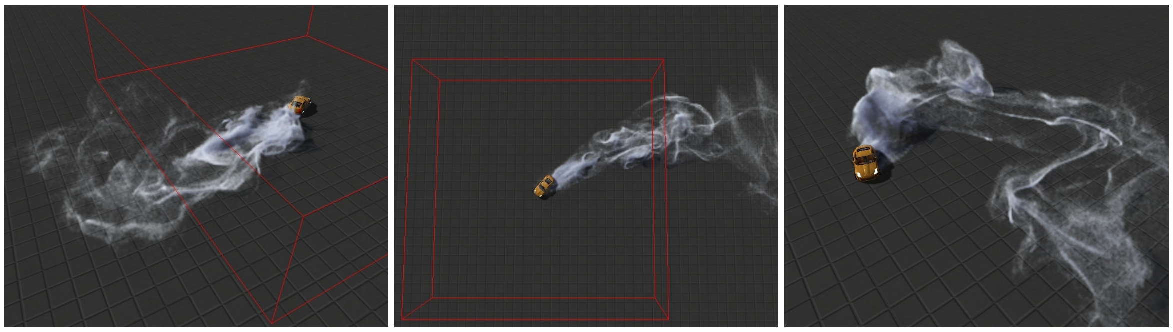


Figure 1: *Screen captures from our interactive car simulator. The red box indicates the extent of the Eulerian simulation grid (resolution is* $128 \times 32 \times 132$*).*

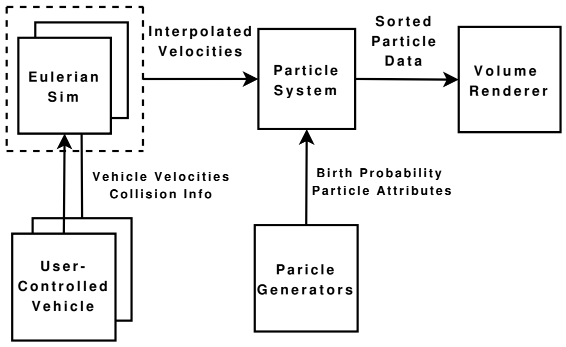


Figure 2: *Schematic overview.*

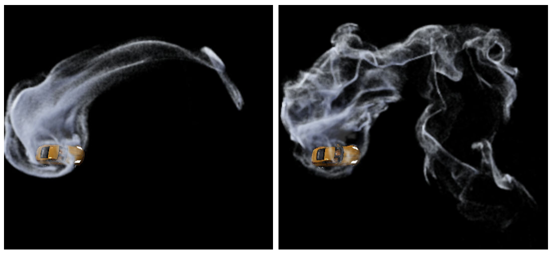


Figure 3: *The stable MacCormack method (right) produces visually superior results to first order semi-Lagrangian (left), even without restricting the time step based on a CFL condition.*

a few cycles before being ejected. On a cache miss, reads via the texture cache are higher latency than uncached reads. Therefore it is only worthwhile to enable the texture cache if it can reduce off-chip bandwidth by enough to offset this increased latency. Because on-chip caches are beneficial only if threads scheduled to the same processor access the same cache lines at roughly the same time, optimizing for coalescing and texture cache performance both require arranging work so that threads in the same warp will access sequential memory locations at the same time (coalescing alone) or at almost the same time (texture cache).

5 Incompressible Navier-Stokes Solver

5.1 Numerical Method

The single-phase incompressible inviscid Navier-Stokes equations, suitable for approximating air moving at low speeds, are

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \nabla p \qquad (1)$$

$$\nabla \cdot \mathbf{u} = 0 \qquad (2)$$

where $\mathbf{u} = (u, v, w)$ is the fluid velocity field and p is the fluid pressure field. An overview of our numerical method for solving these equations is given in Algorithm 1.

We solve Equations 1-2 over a regular staggered grid using the stable second-order MacCormack scheme proposed in [Selle et al. 2008] because it preserves detail well, is stable for any time step size, is trivially parallelizable, and is efficient on a GPU. That paper mentions two limiters for enforce unconditional stability. We found that the limiter of reverting to first-order in the case of overshoot to be quite stable in practice, while the limiter of clamping extrema often leads to bounded but unstable oscillations. We take a single time step per interactive frame, regardless of the CFL time-step restriction. As demonstrated in [Selle et al. 2008], the MacCormack method is only guaranteed to be second-order accurate when the time step is restricted based on the CFL condition. Because we drop this restriction, our solver will not necessarily be second-order accurate in space. However, in practice the MacCormack method yields visually superior results to first-order semi-Lagrangian. Figure 3 shows images from an identical simulation computed with both advection schemes. Please see the video for an animated comparison.

We use a multigrid pressure solver, treating internal solid boundary conditions via the Iterated Orthogonal Projection (IOP) framework [Molemaker et al. 2008]. We believe multigrid is a superior method for interactive GPU implementation because it converges extremely quickly, with a convergence rate independent of grid size. Furthermore, the individual steps of multigrid are all inherently data parallel, resulting in an efficient mapping onto a GPU [Cohen and Molemaker 2009]. FFT-based solvers, while fast and parallel, are

Algorithm 1 Fluid Simulation Algorithm

```
1: $\mathbf{u} \leftarrow$ SelfAdvection $(\mathbf{u})$ (using MacCormack scheme)
2: for $i = 1$ to iop do
3:     Enforce solid boundary conditions on $\mathbf{u}$
4:     Apply impulses directly to $\mathbf{u}$
5:     $p \leftarrow 0$
6:     Solve $\nabla^2 p = \nabla \cdot \mathbf{u}$ using multigrid method
7:     $\mathbf{u} \leftarrow \mathbf{u} - \nabla p$
8: end for
```

only suitable for periodic boundary conditions. Long and Reinhard have demonstrated a fast pressure solver based on sine and cosine transforms that can enforce full-slip wall boundaries [Long and Reinhard 2009]. Such a solver may be suitable for use in our system, although we are not aware of GPU implementations of general sine and cosine transforms.

In IOP, the boundary conditions are enforced, then divergence-free projection is applied, and the process repeats until the boundary conditions are met to the desired level of accuracy. As shown in [Molemaker et al. 2008], this is guaranteed to converge to the correct solution, although the rate of convergence may be poor. IOP is a particularly attractive method for handling boundary conditions because it allows for arbitrary impulses to be applied to $\mathbf{u}$ prior to the projection step. We use these impulses, for example, to apply jets behind each wheel in the moving car. Because we apply the divergence-free projection *after* applying impulses in each iteration, it does not matter if the impulses are incompatible with the condition that $\nabla \cdot \mathbf{u} = 0$. This also allows for very clean decoupling between fluid impulses, boundary conditions, and pressure solver in the source code. In our examples, we have found a single IOP iteration is enough – additional iterations do not noticeably improve visual quality.

Our multigrid solver uses the over-relaxation scheme described in [Yavneh 1996] and always performs a fixed number of multigrid cycles without checking for convergence: 1 full multigrid (FMG) cycle followed by 2 v-cycles with 2 pre and post-sweeps. This enforces a constant simulation time and still converges very well over a range of resolutions, as shown in Table 1.

A subtle issue can arise when using a fixed number of multigrid iterations rather than checking for convergence. To improve the error obtained with an iterative solver such as multigrid, it is common practice to use the pressure solution from the previous time step as a starting point for the iterative solver in the current time step. If we performed multigrid cycles until convergence, this would not be a problem. However, without a *guarantee* of convergence, which we do not have due to performing a fixed number of iterations, reusing the previous pressure can lead to a mode in the pressure error that is not damped sufficiently by our 1-FMG+2-vcycle scheme. We have observed that after running the simulation for several minutes, the error in pressure may eventually grow to the same magnitude as the velocity field itself, resulting in a velocity field dominated by random noise. Simply initializing p to zero before beginning multigrid iterations prevents this from happening.

5.2 Tracking a Region of Interest

Our system is designed to be used in an interactive setting, where the user controls a dynamic object such as a vehicle. For physical correctness, we would need to create simulation grid large enough to encompass all of the volume that the vehicle may travel through, which is impractical even for small environments. To make this problem tractable, we make the following simplifying assumption: there is some region of interest local to vehicle within which the fluid motion will be influenced by the vehicle. Outside this region-of-interest, we assume $\mathbf{u} = 0$. While more sophisticated models could be used, we have found this assumption to work remarkably well.

We dynamically translate the simulation grid to be centered on the vehicle. At the edge of the simulation domain, we enforce the far-field boundary condition. To apply translational motion to the simulation domain's coordinate frame, we can exploit Galilean invariance to calculate the simulation on a fixed simulation grid using a method similar to [Shah et al. 2004]. Shah *et al.* exploit the fact that expressing the incompressible Navier-Stokes equations in a translating coordinate frame induces an apparent uniform flow in the direction opposite to the motion of the domain. Figure 4 depicts this relationship graphically.

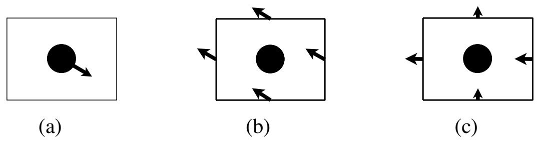


Figure 4: *In incompressible flows, Galilean invariance says that translating an object with velocity t (a) is equivalent to forcing a flow past an object with velocity $-t$ in a coordinate frame translating with velocity t (b). With full-slip boundary conditions, this is equivalent to specifying the flux across domain boundaries (c). Arrows represent either object motion or fluid flow relative to the coordinate frame.*

We will use a subscript *world* to denote vector quantities relative to a global fixed reference frame, and the subscript *local* to denote vector quantities relative to the moving reference frame that tracks a region of interest. In reference frame translating with velocity t, fluid velocities are related to world-space velocities via the expression

$$\mathbf{u}_{world} = \mathbf{u}_{local} + t. \tag{3}$$

We wish to enforce the boundary condition $\mathbf{u}_{world} = 0$ at the edge of the simulation domain. This is equivalent to setting $\mathbf{u}_{local} = -t$. For full-slip boundary conditions, this can be achieved by setting the Dirichlet boundary conditions for the fluxes over the external faces as $u_{local} = -t_x$ on the $\pm x$-faces of the domain, $v_{local} = -t_y$ on the $\pm y$-faces, $w_{local} = -t_z$ on the $\pm z$-faces. This is depicted in Figure 4(c). We use these boundary conditions to induce uniform flow, rather than the explicit method employed in [Shah et al. 2004], because it allows us to skip the explicit update step. Note that inducing uniform flow via boundary conditions will only work with an accurate pressure solver, since the changes to the domain boundaries must propagate instantaneously throughout the entire velocity field due to the Poisson equation's elliptic nature. If the pressure solver does not fully converge, the induced flow will be (incorrectly) non-uniform. The choice to handle domain translation this way therefore goes hand-in-hand with the choice of pressure solver.

5.3 Parallel Fluid Solver Optimizations

Overall, the incompressible Navier-Stokes solver spends about 35% of its time calculating advection (line 1 in Algorithm 1), and 65% enforcing incompressibility (lines 2-8). Many kernels in our system compute finite difference stencils with small support, for example computing the divergence of $\mathbf{u}$. For these kernels threads will access nearby memory locations within a few cycles. Therefore reading data via the texture cache is a net win, improving performance by a factor of $1.1\times$ to $2.0\times$. In our application, the extra logic required to utilize shared memory frequently makes it more expensive than using the texture cache, even though it may result in greater bandwidth savings.

The memory layout of a 3D computational grid is depicted in Figure 5. Grids are represented as k-major linear arrays, where the index calculations are performed explicitly by the kernels. For example, to access an element i, j, k from grid u, the kernel calculates

$$address = ubase + i * istride + j * jstride + k$$

where *ubase* is the pointer to the grid location $(0, 0, 0)$, *istride* is the offset between adjacent cells in the i direction, and *jstride* is

Grid Size	Mean Time	Std Dev Time	Time / N	Error Reduction
$64 \times 16 \times 64$	14.8 ms	0.2 ms	22.6×10^{-5} ms	$8,567\times$
$128 \times 32 \times 128$	26.4 ms	0.2 ms	5.0×10^{-5} ms	$11,079\times$
$256 \times 64 \times 256$	84.6 ms	0.5 ms	2.0×10^{-5} ms	$11,856\times$

Table 1: *Computation time and error reduction for the incompressible Navier-Stokes solver over several seconds of a simulation at different resolutions. Timings recorded on a dual NVIDIA GTX285 GPU system, although the simulation runs on only one GPU. The fourth column shows the simulation time per computational grid cell, which decreases as the amount of available parallelism increases. The error reduction in the right-most column is the ratio of the maximum divergence measured before and after the multigrid pressure solver runs, indicating a typical L_∞ error reduction of 5 orders of magnitude.*

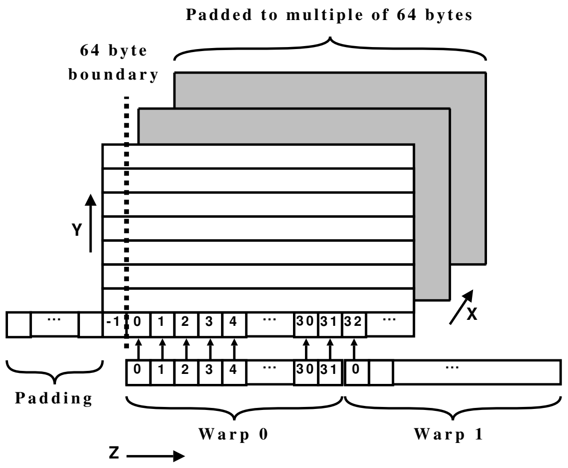


Figure 5: *Memory layout of a computational grid, showing how CUDA threads and warps are mapped to grid elements.*

the offset between adjacent cells in the j direction. We allocate a ring of ghost cells around the exterior of the grids to facilitate handling of boundary conditions. To perform a calculation over the entire grid, we assign one CUDA thread per grid cell, with threads in the same warp assigned to sequential grid cells in k. We pad the arrays so that for every i, j column, the cell at $k = 0$ will start at a 64-byte cache line boundary, reducing the number of coalesced loads required to service all threads in a warp for regular memory accesses. Proper alignment improves performance of most kernels by over $2\times$.

Because we use a staggered grid, the u, v, w, and p grids will have different dimensions. However, we pad them so that *istride* and *jstride* will be the same for all grids. This allows a thread to calculate the offsets from the base pointer once and reuse it for multiple grids. This optimization reduces instruction count by 10-15% for kernels which access multiple grids.

CUDA does not support writing directly to 3D texture objects because hardware 3D textures are laid out in memory in an opaque way. However, it is possible to write directly to 1D textures by because they are laid out as linear arrays (in CUDA you read from a 1D texture via a texture fetch call, but you can also directly modify the underlying bytes via a C pointer). Therefore, we do not use texture filtering hardware to implement the trilinear interpolation required in the point sampling routine for the advection scheme. Rather, we read the 8 values at the corners of the grid cell containing the sample location from a 1D texture, performing the index calculation and interpolation in software. Reading values via 1D textures rather than directly from memory improves performance by about $2\times$ because the texture cache can exploit coherency in these accesses. We tried copying our linear data structure into a hardware 3D texture, and then using the texture filtering hardware to perform trilinear interpolation, but the overhead of the copy made this slower overall.

Algorithm 2 Particle System Update

```
1: for all $i$ do
2:     $Vel_{inside} \leftarrow$ interpolated world-space velocity at $Pos[i]$
3:     $Vel_{outside} \leftarrow Vel[i] + \Delta t \cdot Force[i]$
4:     $w \leftarrow$ blend weight at $Pos[i]$
5:     $Vel[i] \leftarrow w \cdot Vel_{inside} + (1 - w) \cdot Vel_{outside}$
6:     $Pos[i] \leftarrow Pos[i] + \Delta t \cdot Vel[i]$
7: end for
```

6 Particle Engine

6.1 Particle Dynamics

Rather than render a density field that is driven by the fluid simulation via a grid-based technique such as ray marching, we render particles which have been advected through the fluid velocity field. Particles allow the rendering to be decoupled from the simulation, which enables high quality rendering even while using lower resolution fluid simulation grids. This decoupling means that particles are not constrained to lie inside the simulation grid, which is an important aesthetic criteria. It also enables a particle system to be driven by multiple fluid simulation grids, as described in Section 6.2.

Although we choose the far-field approximation $\mathbf{u}_{world} = 0$ for the fluid simulation, enforcing this rule on the particle system would freeze particles as soon as they left the simulation domain. However, there is no requirement that the particle system obey the same rules as the fluid simulation. We can choose any plausible behavior for the particles *outside* the simulated region, as long as we transition smoothly to following the fluid motion while *inside* the simulated region.

Using the scheme in Algorithm 2, particles move under the influence of the fluid velocity field as massless marker particles when inside the region of interest, and follow simple Newtonian dynamics with momentum, gravity, and drag when outside the region of interest. Both the interpolated velocity and blend weight depend on the position of each particle relative to the simulation domain. For a particle at position Pos, the blend weight in the x dimension is calculated as

$$w_x = \text{clamp}\left(0, 1, \min\left(\frac{Pos_x - Min_x}{\beta}, \frac{Max_x - Pos_x}{\beta}\right)\right). \tag{4}$$

where Min_x and Max_x are the extents of the (axis-aligned) simulation domain in world space, and the parameter β controls the width of the blend region. Because the simulations grids lie coincident

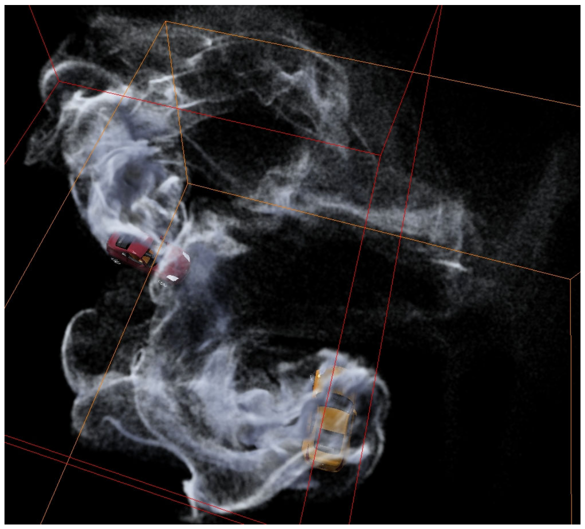


Figure 6: *Two cars interacting. The red and orange boxes indicates the extents of the simulation grids.*

with the ground plane, w_y does not fall off to zero in the negative y direction. The final weight is then $\min(w_x, w_y, w_z)$. We typically choose β to be the width of a few grid cells, but the simulation results are fairly insensitive to varying this value. Furthermore, as long as the transition from inside to outside velocities is smooth, it does not appear to matter much how $\mathbf{u}$ behaves towards the boundaries of the simulation domain. This allows us to use trivial far-field boundary conditions when computing $\mathbf{u}$.

6.2 Multiple Fluid Simulations

A particle can move under the influence of any number of Eulerian fluid simulations. Say we have two uncoupled simulations, $\mathbf{u}^A$ and $\mathbf{u}^B$, which are calculated in different reference frames that have different positions and velocities. We replace w in Algorithm 2, Line 4, with

$$w = \max(w^A, w^B)$$

where w^A and w^B are the weights of the particle position relative to the two simulation grids given by Equation 4. The interpolated velocity becomes a weighted blend

$$Vel_{inside} = \frac{w^A Vel^A_{inside} + w^B Vel^B_{inside}}{w^A + w^B}.$$

Note that when the particle overlaps only one of the simulations, this is equivalent to the single-simulation case. Larger numbers of simulation grids are handled similarly.

This simple blending rule has the benefit of being efficient and stable, but it is not physically motivated and in many cases it could produce incorrect results. Other options include choosing the velocity vector with the highest magnitude or some other non-linear blending rule. We have experimented with a variety of such rules, and overall it is hard to say which method is superior – they all break down in certain cases. However, a simple linear combination works remarkably well and serves to obscure the boundaries of the different simulation domains, so we have chosen it for our implementation.

A consequence of visualizing multiple simulations via their effects on particles is that we do not require the fluid simulations be coupled. Rather, as the data flow in Figure 2 shows, the simulations all respond to the same set of (world space) boundary conditions resulting from the moving car via the IOP method. As demonstrated in Figure 6 and on the accompanying video, multiple moving objects can all appear to interact with particle systems, even without any coupling of the fluid dynamics calculations.

6.3 Parallel Particle System Optimizations

Our particle system is optimized for GPU implementation so that all particles may be processed in parallel, with no global communication. Particle life cycles are controlled by 2 attributes: *Age*, elapsed time since a particle was born, and *Lifespan*, age at which a particle will die. To mark a particle as dead, we use lazy in-place deletion by setting *Lifespan* to zero. In the rendering stage described below, we sort dead particles to the end of the list by considering dead particles to be infinitely far away, and then do not render them at all. This allows us to avoid resizing any buffers when particles are born or die, which would be a serialization point.

We process each particle on a separate CUDA thread, so there is a correspondence between particle i and thread i. "New" particles can be created by reanimating dead particles, which involves setting their *Lifespan* to something non-zero, setting *Age* to zero, and initializing their dynamic state (such as position and velocity) based on some initialization rules. We could keep track of a total number of particles to reanimate at each time step and increment a global counter for every particle that is reanimated until the desired number is reached. However, this approach requires global information to be updated by all threads as they process their particles. A more parallel approach is to set a probability that dead particles will be reanimated in a given frame. When a dead particle is processed by a thread, we calculate pseudo-random number between zero and one, and if that number is less than the per-frame probability, the particle will be reanimated. Since particles are now independent, the particle simulation loop can be performed in parallel.

As explained in Section 4, threads in CUDA are grouped into batches of 32 called warps which all execute in lockstep SIMD fashion. If threads within the same warp take different paths through conditional code, the performance will drop by a factor equal to the number of divergence code paths taken by threads in that warp. To reduce intra-warp execution divergence, we ensure that *Lifespan* and *Age* are the same for all particles in processed by threads in the same warp. Furthermore, the decision as to whether to reanimate a particle is made at the per-warp granularity, rather than for each particle individually. This optimization decreases the particle simulation time by an average of 15% in our test cases. Enforcing that particles are born and die at warp-level granularity does not result in visible artifacts because we use $524,288$ particles. With a $128 \times 32 \times 128$ simulation grid, the particle simulation takes between $1.1\ ms$ and $4\ ms$, depending on how many particles are alive at any given time.

7 Volume Rendering

Data stored in CUDA arrays can be bound to OpenGL vertex buffer objects for rendering, which allows us to keep all particle data on the GPU without any transfers across the PCI-express bus. To support a dual GPU configuration, we can run the fluid and particle simulations on one GPU, and the rendering on a separate GPU. In this case, we must transfer the particle data from the first GPU to the CPU memory, then copy this data to the second GPU as a vertex buffer object. Because these transfers can run asynchronously with CUDA kernel execution or OpenGL rendering calls, we schedule the transfers to overlap computation and rendering via double buffering. We achieve roughly $1.6\times$ scaling across two GPUs.

	2 GPUs (1 GPU)		
	$64 \times 16 \times 64$	$128 \times 32 \times 128$	$256 \times 64 \times 256$
1 car	26 (20)	25 (16)	11 (8)
2 cars	26 (15)	16 (11)	5 (NA)
3 cars	19 (12)	11 (8)	NA (NA)

Table 2: *Frame/second achieved for a variety of configurations of our car simulator. All timings were recorded on a system with two NVIDIA GTX285 GPUs and an Intel Core2Duo running at 3.17GHz. Times in parenthesis are run with only one GPU enabled. Some of the highest resolution simulations did not run due to lack of memory. All simulations use* $524,288$ *particles.*

If no shadows are desired, particles can be sorted back-to-front using a fast GPU radix sort [Satish et al. 2009] then rendered in order as semi-transparent sprites. Rather than sort the particle data directly, we sort an array of indices, which we then treat as an index buffer in OpenGL. Motion blur is handled with a geometry shader that generates a small quadrilateral oriented along the screen-space projection of each particle's velocity vector.

The alpha transparency for a particle is defined in a pixel shader via the function

$$\alpha(x,y) = \text{clip}(noise(x,y) - |r| + \textit{offset}, \textit{cutoff}).$$

The function $\text{clip}(a, \textit{cutoff})$ is defined as a if $a > \textit{cutoff}$, otherwise 0. The *cutoff* value is typically set to something small but just above zero, such as 0.1, in order to give the particles a crisper edge. The texture map $noise(x,y)$ stores several octaves of Perlin noise that have been precomputed, and subtracting the normalized distance from the center of the sprite $|r|$ causes it to fall off radially. The value *offset* is animated from zero to -1 as the particle's *Age* approaches *Lifespan* via the function

$$\textit{offset} = -\left(\frac{\textit{Age}}{\textit{Lifespan}}\right)^3.$$

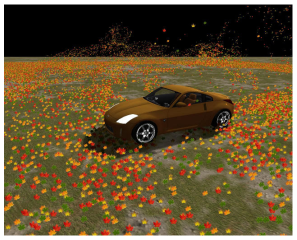
Consequently, α will erode away smoothly from the outside inward as the particle ages, but in an irregular way. Particle transparency functions such as this are used in cinematic visual effects applications, but we are unaware of previous publications that describe these approaches in detail.

To support shadows, we use the half-angle algorithm introduced by [Ikits et al. 2004] and described in detail by [Green 2009]. This algorithm only supports a single light source, but requires less temporary storage and achieves much higher performance than direct volume rendering techniques such as ray marching. Because of the diffuse nature of volumetric shadows, the shadow buffer can be much lower resolution than the display. In the demo we use a shadow buffer 512 or 1024 pixels square to cover the whole scene.

8 Application Details

We have implemented this system in the context of a car simulator, shown in Figures 1 and 6. The goal is to create the appearance of a turbulent wake around a moving car as it kicks up dust or debris. Table 2 shows the total system speeds for different configurations of the application.

Motion of a point on a rigid object can be decomposed as $\mathbf{v} = \bar{\mathbf{v}} + \omega \times \mathbf{r}$ where $\bar{\mathbf{v}}$ is the velocity of the center of mass, ω is the angular velocity, and $\mathbf{r}$ is the vector from the point to the center of mass. We position the center of a simulation grid at the center of mass of each car, with one grid per car. In order to translate a domain so that it stays centered on the car's center of mass, we set $t = \mathbf{v}$. By Equation 3, motion of a point on the car relative to the simulation's coordinate frame becomes $\mathbf{v}_{local} = \omega \times \mathbf{r}$. We therefore enforce interior boundary conditions $\mathbf{u}_{local} = \omega \times \mathbf{r}$ in the grid cells which overlap the car's geometry via IOP. We also apply impulses to create jets of fluid behind the four wheels along a vector opposite the car's velocity.


(a) Leaf simulation.

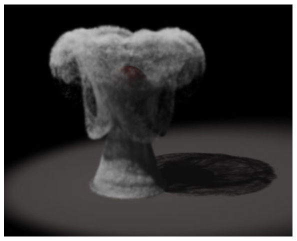

(b) The smoke responds interactively to the solid sphere.

Figure 7: *Example applications.*

Resolution	Present Work	Crane 2007	Long 2009
32^3	182	233	96.3
64^3	86	65	14.3
128^3	22	10	1.6

Table 3: *Frames per second of the current algorithm compared against previous work, ignoring rendering times. The results for the present work and Crane* et al. *were both obtained running on an NVIDIA GTX285. The results for Long and Reinhard are as reported in their paper, and were obtained on a quad-core CPU.*

To simulate tire smoke, we generate particles in regions under the 4 wheels and under the entire car body. The birth probability depends on acceleration and velocity. The *Lifespan* attributes are set to a few seconds, and are randomized slightly at per-warp granularity. Since they do not affect flow-of-control, gravity and drag forces are randomized per-particle to enhance the visual appearance.

We have implemented two additional applications to demonstrate the generality of our system. In the first application, shown in Figure 7(a), we have implemented a simple lift model for blowing around leaves. Leaves are attached to particles that are initially scattered over the entire ground plane and have infinite *Lifespan* values. Leaves are transported by the fluid when the fluid is moving fast enough, otherwise they fall under the influence of gravity. Outside the simulation domain, leaves always fall. In the second application, shown in Figure 7(b), we have added a simple Boussinesq thermodynamics model, which requires additionally advecting a temperature field and adding a thermal buoyancy force to the right-hand side of Equation 1.

Table 3 compares performance of the solver portion only against [Crane et al. 2007] and [Long and Reinhard 2009]. The results for Crane *et al.* were obtained using 60 Jacobi iterations, which is not enough to produce a divergence-free vector field, and therefore has poor visual quality at high resolutions. Our solver shows better performance scaling to high resolutions, while maintaining high visual quality across all grid sizes.

9 Conclusion and Future Work

We have presented a system for creating interactive fluid-driven effects that does not suffer from the "fluid-in-a-box" look common to other Eulerian methods. Rather than use an enormous simulation grid, we calculate an incompressible Navier-Stokes solution only in the vicinity of an object of interest. This exploits the idea that turbulent fluid effects only occur in the presence of moving objects – for a largely static environment, we assume the fluid is motionless away from moving objects. We have also described an efficient GPU-based particle simulation and rendering algorithms.

The technology that makes our system possible is programmable many-core GPU processors. Amdahl's Law says that the advantage of parallelism will be limited by serial bottlenecks. Since transfers across the PCI-express bus are a serial bottleneck, we therefore structure our code to perform as much computation on the GPU as possible. This leads us to choose algorithms which will perform well on a GPU, such as the IOP-based pressure solver, particle system dynamics that prevent intra-warp divergence, and an entirely GPU-based volume renderer.

We could improve the visual quality of the Eulerian simulation in a number of ways, for example by tracking high vorticity regions [Pfaff et al. 2009]. The difficulty of combining vorticity confinement and other similar forcing terms is that they are not guaranteed to be unconditionally stable, but may inject unbounded kinetic energy into the flow. One area for future work is to derive unconditionally stable vortex forcing terms suitable for interactive use.

10 Acknowledgements

We wish to thank Peter Shirley and the anonymous reviewers for many helpful comments.

References

BOLZ, J., FARMER, I., GRINSPUN, E., AND SCHRODER, P. 2003. Sparse matrix solver on the GPU: Conjugate gradients and multigrid. *ACM Trans. Graph. 22*, 917–924.

COHEN, J. M., AND MOLEMAKER, M. J. 2009. A fast double precision CFD code using CUDA. In *Proceedings of ParCFD '09*, in press.

CRANE, K., TARIQ, S., AND LLAMAS, I. 2007. Real time simulation and rendering of 3d fluids. *GPU Gems 3*.

DRONE, S. 2007. Real-time particle systems on the GPU in dynamic environments. In *ACM SIGGRAPH 2007 courses*, 80–96.

FEDKIW, R., STAM, J., AND JENSEN, H. W. 2001. Visual simulation of smoke. In *SIGGRAPH 2001*, 15–22.

GOODNIGHT, N., WOOLLEY, C., LEWIN, G., LUEBKE, D., AND HUMPHREYS, G. 2003. A multigrid solver for boundary value problems using programmable graphics hardware. *Graphics Hardware*, 102–135.

GREEN, S. 2009. Volumetric particle shadows. Tech. rep., NVIDIA.

HADWIGER, M., KRATZ, A., SIGG, C., AND BÜHLER, K. 2006. GPU-accelerated deep shadow maps for direct volume rendering. In *Graphics Hardware*, 49–52.

HORVATH, C., AND GEIGER, W. 2009. Directable, high-resolution simulation of fire on the GPU. *ACM Trans. Graph. 28*, 3, 1–8.

IKITS, M., KNISS, J., LEFOHN, A., AND HANSON, C. 2004. Volume rendering techniques. *GPU Gems: Programming techniques, tips, and tricks for real-time graphics*.

KIM, T.-Y., AND NEUMANN, U. 2001. Opacity shadow maps. In *Eurographics Workshop on Rendering Techniques*, 177–182.

KIPFER, P., SEGAL, M., AND WESTERMANN, R. 2004. UberFlow: a GPU-based particle engine. In *Graphics Hardware*, 115–122.

KLEMP, J. B., AND WILHELMSON, R. B. 1978. The simulation of three-dimensional convective storm dynamics. *J. Atmos. Sci. 35*, 1070–1096.

LINDHOLM, E., NICKOLLS, J., OLBERMAN, S., AND MONTRYM, J. 2008. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro 28(2)*, 39–55.

LOKOVIC, T., AND VEACH, E. 2000. Deep shadow maps. In *SIGGRAPH '00*, 385–392.

LONG, B., AND REINHARD, E. 2009. Real-time fluid simulation using discrete sine/cosine transforms. In *I3D '09*, ACM, New York, NY, USA, 99–106.

MOLEMAKER, J., COHEN, J. M., PATEL, S., AND NOH, J.-Y. 2008. Low viscosity flow simulations for animation. In *SCA '08*, Eurographics, 9–18.

MULLER, M., CHARYPAR, D., AND GROSS, M. 2003. Particle-based fluid simulation for interactive applications. In *SCA '03*, 154–159.

NVIDIA CORPORATION. 2008. *CUDA Programming guide*. Version 2.0.

PFAFF, T., THUEREY, N., SELLE, A., AND GROSS, M. 2009. Synthetic turbulence using artificial boundary layers. *ACM Trans. Graph. 28*, 5, 1–10.

PHILLIPS, E. H., ZHANG, Y., DAVIS, R. L., AND OWENS, J. D. 2009. Rapid aerodynamic performance prediction on a cluster of graphics processing units. In *Proceedings of the 47th AIAA Aerospace Sciences Meeting*, no. AIAA 2009-565.

SATISH, N., HARRIS, M., AND GARLAND, M. 2009. Designing efficient sorting algorithms for manycore GPUs. In *Proc. 23rd IEEE Intl Parallel & Distributed Processing Symposium*, To appear.

SELLE, A., FEDKIW, R., KIM, B.-M., LIU, Y., AND ROSSIGNAC, J. 2008. An unconditionally stable MacCormack method. *J. Sci. Computing 35*, 350–371.

SHAH, M., COHEN, J. M., PATEL, S., LEE, P., AND PIGHIN, F. 2004. Extended Galilean invariance for adaptive fluid simulation. In *SCA '04*, Eurographics, 213–221.

STAM, J. 1999. Stable fluids. In *SIGGRAPH 99*, 121–128.

STAM, J. 2003. Real-time fluid dynamics for games. In *Proceedings of the Game Developer Conference*.

WEST, M. 2007. Practical fluid dynamics: Parts I and II. *Game developer* (March-April).

WICKE, M., STANTON, M., AND TREUILLE, A. 2009. Modular bases for fluid dynamics. *ACM Trans. Graph. 28*, 3, 1–8.

YAVNEH, I. 1996. On red-black SOR smoothing in multigrid. *SIAM J. Sci. Comput. 17*, 1, 180–192.

YUKSEL, C., HOUSE, D. H., AND KEYSER, J. 2007. Wave particles. *ACM Trans. Graph. 26*, 3, 99.

ZHAO, Y., QIU, F., FAN, Z., AND KAUFMAN, A. 2007. Flow simulation with locally-refined LBM. In *I3D '07*, 181–188.