Volume 0 (1981), Number 0 pp. 1–000

Practical CFD Simulations on Programmable Graphics Hardware using SMAC

Carlos E. Scheidegger[1], João L. D. Comba[2] and Rudnei D. da Cunha[3]

[1] Scientific Computing and Imaging Institute, University of Utah
[2] Instituto de Informática, UFRGS
[3] Instituto de Matemática, UFRGS

Abstract

The explosive growth in integration technology and the parallel nature of rasterization-based graphics APIs changed the panorama of consumer-level graphics: today, GPUs are cheap, fast and ubiquitous. We show how to harness the computational power of GPUs and solve the incompressible Navier-Stokes fluid equations significantly faster (more than one order of magnitude in average) than on CPU solvers of comparable cost. While past approaches typically used Stam's implicit solver, we use a variation of SMAC (Simplified Marker and Cell). SMAC is widely used in engineering applications, where experimental reproducibility is essential. Thus, we show that the GPU is a viable and affordable processor for scientific applications. Our solver works with general rectangular domains (possibly with obstacles), implements a variety of boundary conditions and incorporates energy transport through the traditional Boussinesq approximation. Finally, we discuss the implications of our solver in light of future GPU features, and possible extensions such as three-dimensional domains and free-boundary problems.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Line and Curve Generation

1. Introduction

Using the modern programmable graphics hardware processing power for general computation is a very active area of research [**?**] [**?**] [**?**]. Although this is not a new idea [**?**] [**?**], only recently has the graphics hardware used in consumer-level personal computers become powerful enough for scientific applications, in terms of data representation, raw performance and programmability.

Nowadays, modern GPUs have IEEE 754 numbers throughout the pipeline, with highly programmable vertex and fragment units. The GPUs have been described as *stream processors* [**?**] [**?**], where streams are defined as sets of independent uniform data. This is mostly why GPUs are so fast: since computations on pieces of the stream are independent from each other, it is possible to use multiple functional units to process the data efficiently, in parallel.

Obviously, some problems are not easily decomposable in independent pieces. A GPU algorithm is, in most cases, a carefully constructed sequence of graphics API calls, with textures serving as storage for data structures, and vertex and fragment programs serving as computational engines. Often,

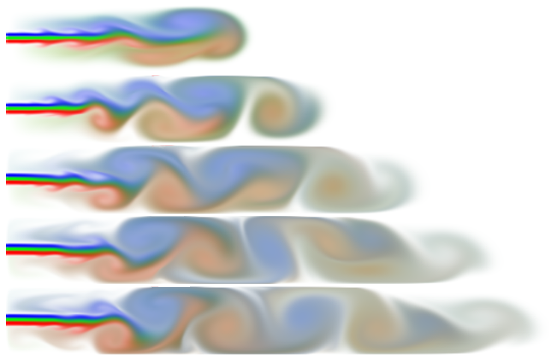
Figure 1: *A* 1024×128 *Navier-Stokes simulation running at interactive rates,* $Re = 10000$.

the algorithm must be significantly changed to be amenable to GPU implementation [?]. For our solver, we use the NV35 and NV40 architectures from NVIDIA. An implementation of this kind requires a thorough understanding of the interplay between the different parts of the graphics system, as, for example, the different pipeline stages and respective capabilities, CPU/GPU communication issues and driver and API quirks.

We show that SMAC [?], a CFD algorithm used in engineering applications, can be implemented as one appropriately constructed sequence of graphics API calls. We will see that in some cases, this GPU version outperforms a single-CPU reference implementation by as much as 21 times; on average, it runs about sixteen times faster. We use OpenGL and Cg for our implementation, and include most vertex and fragment programs in the appendix.

2. Related Work

Stam's stable fluids [?] are a standard computer graphics technique for the simulation of fluid dynamics. Stam's solver relies on the Hodge decomposition principle and a projection operator based on a Poisson equation. Being an unconditionally stable solver, it is able to use much larger timesteps than explicit solvers, that typically are stable only under certain conditions. Although Stam's solution to the Navier-Stokes equations produce visually pleasing fluids, the implicit solver creates too much numerical dissipation. This deteriorates the solution to the point where it has no more relation to fluids in real life. We want to show that GPUs are suitable for numerical processing in engineering situations, we must not allow experimental discrepancies in the simulations.

Stable fluids running on graphics hardware are abundant in the literature [?] [?]. Also related is Goodnight et al.'s multigrid solver [?], which is used to solve the stream portion of a stream-vorticity formulation of the Navier-Stokes equations. Harris et al. [?] show a variety of natural phenomena can be visually reproduced on graphics hardware. Stam's stable fluids were also used as the dynamics engine for a cloud simulation system [?].

Recently, Buck et al. [?] developed a data-parallel programming language that uses the GPU as an execution backend. The Brook compiler converts high-level code to specific data-parallel backends, from CPU SIMD instructions to APIs such as DirectX as OpenGL. This is a notable exception in GPU programming, and a major step towards its perception as a viable computing platform by the general developer.

3. The SMAC Method

In the following, we'll briefly explain how the SMAC algorithm solves the Navier-Stokes equations numerically, without going into our GPU implementation. The following pseudocode shows the basic operation:

```
SMAC-UV()
  $t \leftarrow 0, n \leftarrow 0$
  while $t < t_{end}$
    do Set boundary conditions for $u$ and $v$
       Compute $F^{(n)}$ and $G^{(n)}$ according to Eqs. (6) and (7)
       Solve discrete Poisson equation (Eq.(9))
       Compute $u^{(n+1)}$ and $v^{(n+1)}$ according to (Eq.8)
       $t \leftarrow t+\delta t, n \leftarrow n+1$
       Select $\delta t$ based on stability conditions (Eqs (10) and (11))
```

3.1. The Navier-Stokes Equations

The Navier-Stokes equations are a standard tool for dealing with fluid dynamics, and the SMAC method relies on a discretization of these equations. The incompressible Navier-Stokes equations, in their vector form, are:

$$\frac{\partial u}{\partial t} + u \cdot \nabla u = -\frac{1}{\rho}\nabla p + \nu \nabla^2 u + g, \quad (1)$$

$$\nabla \cdot u = 0 \quad (2)$$

where u is the velocity vector field and p is the pressure scalar field. ν and ρ are the viscosity and the density of the fluid, and g represents external forces acting on all of the fluid (gravity, for example). Our implementation uses the adimensional, two-component cartesian version of the equations:

$$\frac{\partial u}{\partial t} + \frac{\partial p}{\partial x} = \frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) - \frac{\partial (u^2)}{\partial x} - \frac{\partial (uv)}{\partial y} + g_x, \quad (3)$$

$$\frac{\partial v}{\partial t} + \frac{\partial p}{\partial y} = \frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) - \frac{\partial (uv)}{\partial x} - \frac{\partial (v^2)}{\partial y} + g_y, \quad (4)$$

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \quad (5)$$

where Re is the Reynolds number, relating viscous and dynamic forces.

3.2. Boundary Conditions and Domain Discretization

We assume a rectangular domain $[0,w] \times [0,h] \subset \mathbb{R}^2$ in which we restrict the simulation. This means we have to deal with the appropriate boundary conditions along the borders of the domain. We implemented boundary conditions to model walls, fluid entry and exit — these allow a variety of real-life problems to be modeled. The walls and fluid entry are Dirichlet boundary conditions: the velocity field has a certain fixed value at the boundary. The outflow condition is different. The exit of fluid from a flow is modeled to mean that we essentially do not care about what happens to fluid

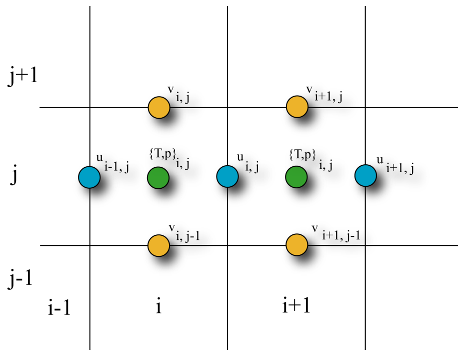

Figure 2: *In a staggered grid discretization, different variables are sampled at different places.*

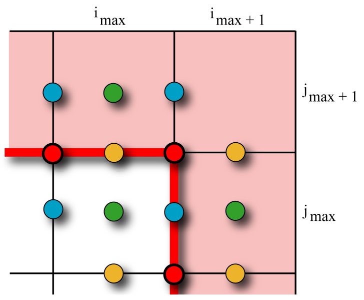

Figure 3: *The thick red line represents the boundary, and the light red cells are the boundary strip. The red circles show the boundary points that aren't sampled directly (for which we need interpolation assumptions), and the remaining circles show the field sampling positions for different fields near the boundary. The points that are sampled inside the boundary strip are manipulated to enforce the boundary conditions.*

parcels that leave the domain through this boundary. This is obviously not physically realizable, but it's still usable in practice to implement situations like a part of a riverbank. We only need to assure the boundary is placed in "uninteresting" parts of the domain, because we essentially lose information. We approximate such a boundary condition by assuming that the fluid that leaves the domain is uninteresting and behaves exactly as the neighborhood of the boundary that is inside the domain. This gives us Neumann conditions: the derivative of the velocity field is fixed across the boundary (in our case, at zero).

To solve the equations numerically, we approximate the rectangular subset of $\mathbb{R}^2$ with a regular grid, ie. the velocity and pressure scalar fields are sampled at regular intervals. We discretized the domain using a *staggered grid*, which means that different variables are sampled in different positions. This representation is used because of its better numerical properties [?]. The grid layout for our simulation is shown in Figure 2.

The boundary conditions in the grid are simulated by adding a *boundary strip*. The boundary strip is a line surrounding the grid cells that will be used to ensure that the desired boundary condition holds. In Figure 3, we show one corner of the boundary strip. We discretize the boundary conditions by making appropriate use of the boundary strip. Consider, for example, the wall boundary condition, where the velocity components must all become zero. Some of the values on our grid are sampled directly on the boundary — these can be simply set to zero. For the values that aren't, we assume that the underlying continuous fields are simply a linear interpolation of the sampled values, and we then set the boundary strip variables so that the interpolated value in the boundary is zero. This idea can be applied to all boundary conditions, as will be described later.

3.3. Discretization of the Equations

The Navier-Stokes equations will be numerically solved by time-stepping: from known velocities at time t, we compute new values at time $t+\Delta t$. The values in the varying timesteps will be called $u^{(0)}, u^{(1)}, \ldots$. To discretize the Navier-Stokes equations, we first introduce the following equations:

$$F = u^{(n)} + \delta t \left[\frac{1}{Re} \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) - \frac{\partial (u^2)}{\partial x} - \frac{\partial (uv)}{\partial y} + g_x \right] \tag{6}$$

$$G = v^{(n)} + \delta t \left[\frac{1}{Re} \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) - \frac{\partial (uv)}{\partial x} - \frac{\partial (v^2)}{\partial y} + g_y \right] \tag{7}$$

Rearranging (3) and (4) and discretizing the time variable using forward differences, we have

$$u^{(n+1)} = F - \delta t \frac{\partial p}{\partial x}, v^{(n+1)} = G - \delta t \frac{\partial p}{\partial y} \tag{8}$$

This gives us a way to find the values for the velocity field in the next step. F and G, when discretized, will depend only on known values of u and v and can be computed directly. We use central differences and a hybrid donor cell scheme for the discretization of the quadratic terms, following the reference CPU solution [?]. We are left to determine the pressure values. To this end, we substitute the continuous version of Equations (8) into Equation (5) to obtain a Poisson equation:

$$\frac{\partial^2 p^{(n+1)}}{\partial x^2} + \frac{\partial^2 p^{(n+1)}}{\partial y^2} = \frac{1}{\partial t} \left(\frac{\partial F^{(n)}}{\partial x} + \frac{\partial G^{(n)}}{\partial y} \right) \tag{9}$$

When discretizing the pressure values, we notice that we cannot compute them directly: each pressure value $p^{(n+1)}$ depends linearly on other pressure values $p^{(n+1)}$ from the same timestep. In other words, the discretization of the Poisson equation results in a linear system of equations, with as many unknowns as there are pressure samples in the grid. This system can be solved with many different methods, such as Jacobi relaxation, SOR, conjugate gradients, multigrids, etc. With the pressure values, we can determine the velocity values for the next timestep, using Equations 8. We then repeat the process for the next timestep.

3.4. Stability Conditions

SMAC is an explicit method, and, as most such methods, is not unconditionally stable. To guarantee stability, we have to make sure that these inequalities hold:

$$\frac{2\delta t}{Re} < \left(\frac{1}{\delta x^2} + \frac{1}{\delta y^2}\right)^{-1} \tag{10}$$

$$|u_{max}|\delta t < \delta x \text{ , } |v_{max}|\delta t < \delta y \tag{11}$$

Here, δt, δx and δy refer to the timestep sizes, and distance between horizontal and vertical grid lines, and u_{max} and v_{max} are the highest velocity components in the domain. During the course of the simulation, δx and δy are fixed, so we must change δt accordingly. In practice, one wants to use a safety multiplier $0 < s < 1$ to scale down δt.

4. Energy Transport

We have augmented our original SMAC solver with energy transport to give an example of the flexibility of the approach. We introduce an additional scalar field, accompanied by suitable differential equations that govern its evolution, discretize both the field and the equations, and incorporate them in our original solver with very small changes in data structures.

4.1. The Energy Equation

In CFD simulations it is often necessary to take into account the effects of temperature on the flow. Effects of temperature on a fluid include density and volume changes, which may lead to additional buoyancy forces. In order to augment the SMAC base model proposed in section 3 so that it accounts for such effects, we need to include the fluid temperature T in our model. From the principle of conservation of energy we obtain the energy equation for a constant thermal diffusivity α, with negligible viscous dissipation and a heat source q''':

$$\frac{\partial T}{\partial t} + \vec{u} \cdot \nabla T = \alpha \nabla^2 T + q''' \tag{12}$$

We make additional assumptions, known collectively as the *Boussinesq approximation*. Basically, we assume that the density difference due to temperature is negligible except in the buoyancy terms (so that we can still solve the incompressible version of the Navier-Stokes equations), and that most other fluid properties are temperature-invariant. We incorporate these assumptions and make the energy equation adimensional by adding a new dimensionless quantity Pr (the Prandtl number) which relates the relative strensth of the diffusion of momentum to that of heat. The energy equation then becomes:

$$\frac{\partial T}{\partial t} + \vec{u} \cdot \nabla T = \frac{1}{Pr}\frac{1}{Re}\Delta T + q''' \tag{13}$$

Both Dirichlet and Neumann boundary conditions are supported, and we use the same linear interpolation assumption to compute the boundary strip values for the flow. When otherwiser unspecified, walls implement adiabatic boundary conditions — no transfer of energy.

4.2. Discretization of the Energy Equation

In order to compute the temperatures numerically, we need to discretize the Equation (13) (given here in component form):

$$\frac{\partial T}{\partial t} + \frac{\partial (uT)}{\partial x} + \frac{\partial (vT)}{\partial y} = \frac{1}{Re}\frac{1}{Pr}\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) + q''' \tag{14}$$

We put the temperature samples in the center of the grid cells, just like the pressure values. For the actual discretization, we use the same donor cell scheme as we did before for the momentum equations. The time variable is discretized using forward differences, and so we compute the sequence of temperature values by explicit Euler integration.

We need to rewrite the quantities F and G to take into account the bouyancy forces described by the Boussinesq term:

$$\tilde{F}^{(n)}_{i,j} = F^{(n)}_{i,j} - \beta\frac{\partial t}{2}\left(T^{(n+1)}_{i,j} + T^{(n+1)}_{i+1,j}\right) g_x \tag{15}$$

$$\tilde{G}^{(n)}_{i,j} = G^{(n)}_{i,j} - \beta\frac{\partial t}{2}\left(T^{(n+1)}_{i,j} + T^{(n+1)}_{i,j+1}\right) g_x \tag{16}$$

The discretized momentum equations are rewritten as:

$$u^{(n+1)} = \tilde{F} - \delta t\frac{\partial p}{\partial x}, v^{(n+1)} = \tilde{G} - \delta t\frac{\partial p}{\partial y} \tag{17}$$

4.3. Stability Conditions

The transport equation discretization is also only conditionally stable, and because of that we need an additional stability condition to hold:

$$\frac{2\delta t}{RePr} < \left(\frac{1}{\delta x^2} + \frac{1}{\delta y^2}\right)^{-1} \tag{18}$$

This inequality is added to the previously described ones (Equations (10) and (11)). The stepsize δt is then chosen appropriately.

4.4. Algorithm

We list below the SMAC algorithm incorporating energy transport:

```
SMAC-UVT()
    $t \leftarrow 0, n \leftarrow 0$
    while $t < t_{end}$
        do Set boundary conditions for $u$, $v$ and $T$
            Compute $T^{(n+1)}$ (14)
            Compute $\tilde{F}^{(n)}$ and $\tilde{G}^{(n)}$ according to Eqs. 6 and 7
            Solve Poisson Equation using numerical solver (Eq. 9)
            Compute $u^{(n+1)}$ and $v^{(n+1)}$ according to
                Eq. 17 using $\tilde{F}^{(n)}$ and $\tilde{G}^{(n)}$
            $t \leftarrow t+\delta t, n \leftarrow n+1$
            Select $\delta t$ based on stability conditions (Eqs (10), (11), (18))
```

5. The Implementation in a GPU

In this section we show the GPU implementation of the SMAC method using NVIDIA's NV35 and NV40 architectures. First we show how the data structures are stored into texture memory, followed by the presentation of all programs used to implement the algorithm.

5.1. Representation

We use a set of floating-point p to store the values of the velocity fields and intermediate variables. All tests were performed using 32-bit precision floating-point representations. The textures used to store data are more precisely called *pbuffers*, or *pixel buffers*, because they can be also the target of a rendering primitive, similar to writing to the frame buffer. Each *pbuffer* can have multiple *surfaces*. Each surface is basically a separate physical copy of the data. The important thing to notice is that switching the write target between surfaces from the same *pbuffer* is *much* faster than switching between different *pbuffers*. In our current version, we have five *pbuffers*:

- **uvt**: This will store the velocity field, together with the temperature. Each of the three channels will respectively store u, v, and T. This is a double-surface *pbuffer*, so that we can write the boundary conditions in one surface while reading from the other.
- **FG**: This *pbuffer* will be used to store the intermediate F and G values, each on one channel.
- **p**: This *pbuffer* will store the pressure values. This is also a double-surface *pbuffer*, so that we can use ping-pong rendering (which will be described shortly)
- **ink**: This *pbuffer* will store ink values, not used in the simulation but used for the visualization of the velocity field. This is also double-surfaced so that ink boundary conditions can be applied.
- **r**: This auxiliary buffer will be used in *reduction* operations described later. This, too, has two surfaces, and for the same reason as the pressure *pbuffer*, which will be described shortly.

We have a couple of read-only auxilliary data structures to handle complex boundary conditions and domains, both stored in textures. The first of these simply signals whether the cells is an obstacle cell or a fluid one. The second and more interesting one stores *texture access offsets* instead of color intensities. These will be used in the computation of boundary conditions, as will be described shortly.

It is important to mention that the NV35 and the NV40 do not allow simultaneous reads and writes to the same same surface [**?**], which are needed by many iterative algorithms. To circumvent this problem, we use a standard GPU technique called *ping-pong rendering*. The idea of ping-pong rendering is to successively alternate the roles of two surfaces of a *pbuffer*. We first write some data on surface 1 while reading values from surface 2, and then we switch their roles, writing values on surface 2 while reading the just-written values from surface 1. By carefully arranging the order of computation, we can implement many of the algorithms that simultaneously read and write the same memory portion. Because of this, the **r**, **uvt** and **p** *pbuffers* have two surfaces, and take twice the amount of memory as would be otherwise necessary.

Notice that there is a sharp distinction in GPU algorithms between writable memory and purely constant data to be accessed. This is a legacy from previous graphics APIs, where an application typically only wrote to the frame buffer. As things become more complex and developers start using offscreen buffers more commonly, we will see this difference disappearing. For now, writable memory (specially when it comes to floating-point buffers) is accessible through a limited interface (if compared to "constant" texture memory).

5.2. Setting the Boundary Conditions

The first step in the algorithm is to enforce the boundary conditions. A fragment program reads the velocity values and the status texture, gets the necessary texture offsets and determines the correct velocity components for the boundaries. We use textures to store precomputed texture access patterns for the boundary treatment because this decreases immensely the complexity in the fragment program. A fragment program without branching executes much faster than one with branching. We need to use the right offsets because boundaries in different directions are determined from different neighbors. All of our boundary conditions can be calculated with one fragment program when we notice that they share a common structure: for each component (in the 2D case, only u, v and T), we only need to sample one direct neighbor. Then, the boundary conditions are of the following form:

$$u_{ij} = \alpha_u u_{ij} + \beta_u u_{neighbor} + \gamma_u$$

$$v_{ij} = \alpha_v v_{ij} + \beta_v v_{neighbor} + \gamma_v$$

$$T_{ij} = \alpha_T T_{ij} + \beta_T T_{neighbor} + \gamma_T$$

We store the appropriate α, β and γ values, along with the offsets to determine the neighbor, in the status texture. Packing operations allow us to put more than 4 values on the RGBA channels, and some of the values can be determined from the other ones. If the cell happens not to be a boundary cell, we simply set $\alpha = 1, \beta = 0, \gamma = 0$, so that we have an identity operator for these cells. This fragment program is used to render a domain-sized quadrilateral, and the end effect is that the boundary conditions for temperature and velocity will have been set for the entire domain.

5.3. Computing FG

The velocity field with enforced boundary conditions is used to compute the **FG** buffer. The **FG** *pbuffer* is computed simply by rendering another domain-sized quad, using the **uv** *pbuffer* as input, and a fragment program that represents the discretization of Equations (6) and (7).

5.4. Determining Pressure Values

With the **FG** values, we can now determine the pressure value. As mentioned above, we must solve the equation system generated by the Poisson equation discretization. In CPUs, SOR is the classical method used to solve these systems, because of the low memory requirements and the good convergence properties. The main idea of SOR is to use, in iteration it, not only the values of the pressure in the iteration $it - 1$, but the values in it that have just been calculated. In a GPU, unfortunately, we cannot do that efficiently: it would require reading and writing the same texture simultaneously.

The solution we adopted is to implement Jacobi relaxation as a fragment program. To check for convergence, we must see if the norm of the residual has gone below a user-specified threshold. The norm is a computation that combines all of the values in a texture, differently from every other fragment program described so far. We must find a special way of doing the calculation, since data-parallel architectures don't usually provide such a means of combination.

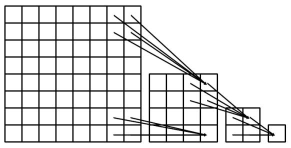

Figure 4: *Combining all elements in a SIMD architecture through reductions.*

We implement what is called a *reduction*. In each reduction pass, we combine values of a local neighborhood into a single cell, and recursively do this until we have but one cell. This cell will hold the result of the combination of all original cells. Figure 4 illustrates the process. Not only this computation is significantly more expensive than the relaxation step, there is a measurable overhead in switching between fragment programs and *pbuffers*. We use a more clever scheme to reduce the number of switches: instead of computing the residual at each relaxation step, we adaptively determine whether a residual calculation is necessary, based on previous results using an exponential backoff algorithm. That is, we calculate the residual for the i^{th} time only after 2^i relaxation steps. After the first pressure solution is determined, we use the number of relaxation steps that were necessary in the previous timestep as an estimate for the current one. This results in significantly better performance.

5.5. Computing the $t^{(n+1)}$ Velocity Field

After computing the pressure values, we can determine the velocity field for the next timestep using Equation (8). This is done by another fragment program that takes the appropriate textures and renders, again, a domain-sized quad. The final step is ensuring that the stability conditions (10) and (11) hold.

The first condition is easy to determine, since it is constant for all timesteps and can be pre-calculated. The other ones, though, require the computation of the maximum velocity components. This is an operation that requires a combination of all the grid values, and again a reduction is needed. This time, though, we use the maximum of the neighbors instead of the sum as the reduction operation.

5.6. Obstacles

To implement the obstacles, we simply extend the idea used in the wall boundary condition to work inside the domain. Our status and obstacle textures will hold special value denoting a wall for visualization purposes, but the original fragment program that deals with boundary conditions works without changes.

One must take into account, however, that not all obstacle configurations are valid. Remember from Section 3.2 and Figure 3 that a boundary condition is specified by relating

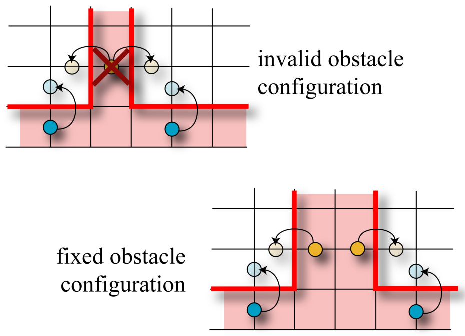

Figure 5: *While the boundary strip always specifies a valid boundary condition, obstacles can be ambiguous: Should the vertical obstacle cell enforce the boundary conditions of the left or the right fluid portion?*

a cell of the boundary strip to a given cell in the fluid in a special way, so that we can say things about the values of the fields at the domain boundary. In the simple case of actual domain boundaries, each cell on the boundary strip will only ever "see" one specific cell inside the domain. The same role is played by the "crust" of the domain obstacles (their one-cell border). There are some obstacles, however, whose "crust" is too thin, and it can see more than one fluid cell. In these cases, the boundary conditions are underspecified, as can be seen in Figure 5. There is an ambiguity (compare to Figure 3), as we would have to use the obstacle crust to specify boundary conditions for two different fluid cells. Fortunately, this can be easily fixed with a finer subdivision or with a thicker boundary, so it is not a critical issue. In our system, we detect such invalid domains and pad them with obstacle cells to ensure simulation validity.

5.7. Visualization

Usually, the simulation of Navier-Stokes is not fast enough to allow interactivity, and so the results are simply stored in a file to be interpreted later. We instead take advantage of the fact that the simulation runs at interactive rates, and that the data is already in the graphics memory to implement interactive visualization tools.

For our original system, we developed a visualization tool inspired on the use of colored smoke in real-life airflow visualization. We store, in addition to the velocity fields, an *ink field*, which is a passive field that does not affect the velocity in any way. The ink field is advected by the velocity field, and the motion of the ink is used to visualize features such as vortices. *Ink emitters* of different colors can be arbitrarily placed and moved around in the domain, allowing to investigate areas of flow mixture or separation.

The advection step occurs right after the boundary conditions are enforced. A first shot in an algorithm for the advection would be to get the current velocity at the center of the cell, and, using the timestep value, determine the position for this parcel of fluid. This approach has two problems: first, we would have to write to different cells, because the timestep never takes an ink particle more than a grid width or height (consider the stability conditions for the discretization). Second, and more seriously, we don't know, prior to running the fragment program, what are the cells in which to write our results. This is known as a *scatter* operation [?], and is one that is missing from GPUs: the rasterization stage issues a fixed output place for each fragment. We need to replace the scatter operation with a *gather* one: an operation in which we don't know, prior to running the program, what are the cells we will *read*. This is implemented in GPUs through the use of *dependent texturing* [?]. We illustrate our solution in Figure 6. Instead of determining the position that the ink in the present position will be, we will determine what portion of ink was in a past position. To do this, we assume that the velocity field is sufficiently smooth, and we use a step backward in time using the present velocity. We have to sample the velocity at center of the grid cell, because that's where the ink is stored. Since the velocities are stored in a staggered grid, this requires careful coding in the interpolation routine.

This basic visualization technique works fine for inspecting local portions of the domain. But we would like to have a more global visualization, that enables us to quickly spot all major features of the flow, to maybe later use ink splats to analyze specific features. For this end, we adapted the image-based flow visualization of van Wijk [**?**] to run entirely on GPUs. Originally, the visualization technique relies on an eulerian advection of regular patches through the velocity field. This implies a scatter operation, or at least a texture lookup in a vertex program. Since texture lookups inside vertex programs are limited to NV40's, and we wanted portability across different GPU models, we adapted the original model to use instead a lagrangian backward step, similar to the ink advection solution. In fact, implementing IBFV required only five additional lines of code in the fragment program, to sample the modulated noise texture and to blend it appropriately. We then create an array of uncorrelated noise textures, noting that a good PRNG (ie, not the one in the standard library) is necessary to avoid spatially correlated noise. We show some visualizations using our new implementation next.

Flow visualization is a huge research area in and of itself, so there are many other techniques we could have used. We chose IBFV because not only it is extremely simple to implement, it offers objective advantages. For example, LIC [**?**] doesn't show the relative velocities of the fluid, and more advanced techniques, such as UFAC, [**?**] requires storage of the velocity field at different times. IBFV offered the ideal compromise between visualization quality while meeting our restrictions on performance and storage requirements.

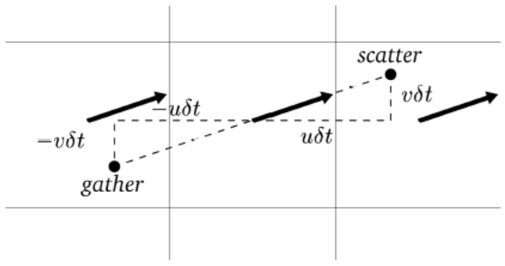

Figure 6: *Stepping backward in time to avoid a scatter operation.*

CPU	32 × 32	64 × 64	128 × 128
Re = 100	1.73s	35.71s	428.05s
Re = 1000	5.52s	122.47s	903.63s

NV35	32 × 32	64 × 64	128 × 128
Re = 100	3.36s	13.34s	60.29s
Re = 1000	6.14s	28.60s	110.36s

NV40	32 × 32	64 × 64	128 × 128
Re = 100	1.54s	5.29s	30.79s
Re = 1000	2.11s	9.15s	42.89s

Table 1: *Timings for CPU, NV35 and NV40*

6. Results

To judge the performance of the GPU implementation, we compared our solution to a CPU reference code provided by Griebel et al [?]. We used a classical CFD verification problem, the *lid-driven cavity*. The problem begins with the fluid in a stationary state, and the fluid is moved by the drag of a rotating lid. This is a *steady* problem, no matter what are the conditions such as Reynolds number and lid velocity: when *t* increases, the velocity field tends to stabilize. Knowing this, we run the simulations until the changes in the velocity field are negligible.

We conducted our tests using two different Reynolds numbers and three different grid sizes. The results can be seen in Table 1. Figure 7 shows the ratio of improvement of the GPU solution. The CPU is a Pentium IV running at 2 GHz, and the GPUs are a GeForce FX 5900(NV35) and GeForce 6800 Ultra(NV40). Both programs were compiled with all optimization options enabled, using Microsoft Visual Studio .NET 2003. Our GPU implementation, with all source code for the vertex and fragment programs and movie fragments is available at `http://www.sci.utah.edu/~cscheid/smac`.

As we can see, the only case where the GPU was out-

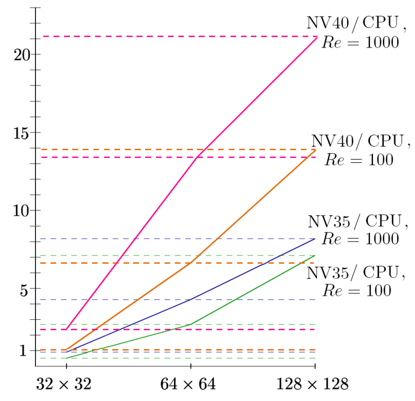

Figure 7: *GPU-CPU ratio timings*

performed by the CPU was in very small grids with the NV35. This is a situation where convergence is very quick, and the overhead due to *pbuffer* switches [?] probably overshadowed the parallel work of the GPU. Also, the ratio between GPU computation and CPU-GPU communication was smallest in this case. In all the other situations, the GPU implementation was significantly faster, with the NV40 achieving a speedup factor of 21 in large grids with large Reynolds numbers.

6.1. Quality

To judge the quality of the GPU implementation when compared to the reference CPU implementation, we ran both programs with exactly the same problem specifications, and compared the velocity fields at each timestep. In our experiments, the difference between velocity components computed in the two programs was always less than 10^{-2}, and most of the time less than 10^{-3}. The problems had velocity ranges between 0 and 1. The largest differences were found in high pressure areas, probably due to the difference between the Jacobi and the SOR algorithms.

The reference CPU implementation didn't allow for general domains, so for that part of the implementation, we had to rely on qualitative measurements. For example, we expect vortices around corners with high speed fluid, and we can see this in Figure **??**. Some well-known phenomena, such as the *Kárman vortex street* [?], were also experienced in our software, in accordance to experimental results. See Figure **??**.

6.2. CFD Simulation Cases

We tested our solution with several other CFD classic problems discussed in [?]. The simulations and their main parameters are listed below, as well as pointers to corresponding figures:

- Lid-Driven Cavity: Square container filled with fluid is affected by the movement of the lid along a given constant velocity. We used a 256×256 grid, with $Re = 10000$. The expected counter-eddies in the corner for large Re can be seen on the left image of Figure **??**. We show a similar simulation ($Re = 1000$) with our IBFV-based visualization tool (right image of Figure **??**). Notice how we can clearly spot the relative flow velocity and vorticity from the noise patterns.
- Domain with Obstacles: Square container filled with obstacles, with a single fluid entry and two exits. We used a 128×128 grid, with $Re = 1000$, with inflow in the lower west, outflow everywhere else. (left image of Figure **??**). We also show a similar simulation (64×64, $Re = 100$) using our IBFV-based visualization tool (right image of Figure **??**). Notice how the global behavior of the fluid is captured, even where there's no ink splats.
- Smoke Simulation: 128×1024. In this experiment we observe the effects on using large Reynold numbers, generating smoke-like vector trails. We used a 128×1024 grid, with $Re = 10000$, inflow in the south, outflow in the north (Figure **??**).
- Wind tunnel mock-up: This experiment simulates wind tunnel mockup conditions with inflow fluid coming from the east, around an object resembling a vehicle, and outflow in the east (Figure **??**). We used a 256×64 grid, $Re = 100$.
- Flow Past an Obstacle: In this experiment we consider flow past a simple obstacle. For Reynold numbers above 40, a Kárman vortex street effect can be seen on the trail of vertices. We used a 256×64 grid, $Re = 1000$, with inflow in the west, outflow in the east (Figure **??**).
- Natural Convection with Heated Lateral Wall. In this experiment we have a square container with one heated lateral wall, and no-slip boundary conditions everywhere. The temperature difference drives fluid movement. We used a 64×64 grid, with $Pr = 7$, $Re = 985.7$, $\beta = 2.1e-4$ and gravity components $g_x = 0$ and g_y=-9.706e-2. Note the central vertex and expected circular fluid movement (Figure **??**).

7. Analysis

The GPU achieves top performance when doing simple calculations on massive amounts of data, and this is the case in our algorithm. Most of the computation is done on the GPU. The CPU only orchestrates the different GPU programs and buffers, adjusts the timestep and determines the convergence of the Poisson equation.

Measuring the amount of time taken in each part of our algorithm, we noticed that more than 95% percent of the time was spent solving the Poisson equation. This was the main motivation for the exponential backoff residual calculation step. This change doubled the overall performance.

We could have implemented a conjugated gradient solver, but since this method requires two dot products at each timestep, additional reductions would need to be performed (which is a slower operation). Alternatively, we could have used a multigrid solver for the Poisson equation, such as the one developed by Goodnight et al. [?]. We chose not to do so because we did not have a suitable CPU multigrid code to compare to, and we did not want to skew the results in either way. In addition, this solver requires a more involved treatment of boundary conditions, specially in complex environments with several obstacles.

In the simulation depicted in Figure 1, we have a 1024×128 grid, and the simulation runs at approximately 20 frames per second, allowing real-time visualization and interaction.

8. Future Work and Conclusion

The Navier-Stokes GPU solver shown here can be easily extended to three dimensions. The **uv** and **FG** *pbuffers* would have to hold an additional channel. Additionally, we can't use 3D textures as *pbuffers*, so the texture layout would probably follow [?]. The fragment programs would not fundamentally change, and the overall algorithm structure would stay the same.

A more ambitious change is to incorporate free boundary value problems to our solver. In this class of problems, we have to determine both the velocity field of the fluid and the interface between the fluid and the exterior (for sloshing fluid simulations, for example). The approach that is proposed in the SMAC algorithm is to, starting with a known fluid domain, place particles throughout the domain and then displace them according to the velocity field. At the next timestep, the algorithm checks whether any particles arrived in cells that had no fluid. These cells are then appropriately marked, and the simulation continues. We can't do that directly on the GPU, because that would require a scatter operation. A possible solution is to use the *volume-of-fluid* method [?]. The volume-of-fluid method keeps track of the fraction of the fluid that leave the cells through the edges. This way, all cells that are partially filled are marked as border cells, the ones completely filled are marked as fluid cells, and the ones without any fluid are marked as empty cells. Such a scheme could be implemented using GPUs, since the calculation of fluid transfer between cells can be done for each cell individually, without having to write to arbitrary locations. However, this remains to be implemented.

Nevertheless, we have shown that the GPU is a viable computing engine for the complete solution of the Navier-Stokes via a explicit solver, suitable for engineering contexts. Our solution takes advantage of the streaming nature of the GPU and minimizes the CPU/GPU interaction, resulting in the high performances reported. We hope that the fact that GPU performance growth is largely outpacing the CPU will serve as an additional motivation for the implementation of other similar applications.

9. Acknowledgments

The authors thank NVIDIA for providing the graphics hardware used in this paper, specially the NV40 reference board and drivers. The work of Carlos Eduardo Scheidegger has been partially supported by National Science Foundation under grants CCF-0401498, EIA-0323604, and OISE-0405402.

References

J. Bolz, I. Farmer, E. Grinspun, P. Schröder. *Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid*. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2003), 2003.

B. Cabral, L. Leedom. *Imaging vector fields using line integral convolution*. Proceedings of SIGGRAPH 1993.

J. Comba, C. Dietrich, C. Pagot, C. Scheidegger. *Computation on GPUs: From A Programmable Pipeline to an Efficient Stream Processor*. Revista de Informática Teórica e Aplicada, Volume X, Número 2, 2003.

N. Goodnight, C. Woolley, G. Lewin, D. Luebke, G. Humphreys. *A Multigrid Solver for Boundary-Value Problems Using Programmable Graphics Hardware*. Proceedings of the Eurographics/SIGGRAPH Graphics Hardware Workshop, 2003.

M. Griebel, T. Dornseifer, T. Neunhoffer, *Numerical Simulation in Fluid Dynamics*. SIAM, 1998.

N. Govindaraju, S. Redon, M. Lin, D. Manocha. *CULLIDE: Interactive Collision Detection Between Complex Models in Large Environments using Graphics Hardware*. Proceedings of the Eurographics/SIGGRAPH Graphics Hardware Workshop, 2003.

M. Harris, W. Baxter III, T. Scheuermann, A. Lastra. *Simulation of Cloud Dynamics on Graphics Hardware*. proceedings of the Eurographics/SIGGRAPH Graphics Hardware Workshop, 2003.

M. Harris, G. Coombe, T. Scheuermann, A. Lastra. *Physically-Based Visual Simulation on Graphics Hardware*. Proceedings of the Eurographics/SIGGRAPH Graphics Hardware Workshop, 2002.

K. Hillesland, S. Molinov, R. Grzesczuk. *Nonlinear Optimization Framework for Image-Based Modeling on Programmable Graphics Hardware*. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2003), 2003.

G. Kedem, Y. Ishihara. *Brute Force Attack on UNIX passwords with SIMD Computer*. Proceedings of the 8th USENIX Security Symposium, 1999.

J. Krüger, R. Westermann. *Linear Algebra Operators for GPU Implementation of Numerical Algorithms*. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2003), 2003.

NVIDIA Corporation. *OpenGL Extension Specifications*. Web site last visited on May 17th, 2004. `http://developer.nvidia.com/ object/nvidia_opengl_specs.html`

E. Larsen, D. McCallister. *Fast Matrix Multiplies using Graphics Hardware*. Supercomputing 2001.

T. Purcell, I. Buck, W. Mark, P. Hanrahan. *Ray Tracing on Programmable Graphics Hardware*. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2002), 2002.

I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, P. Hanrahan. *Brook for GPUs: Stream Computing on Graphics Hardware*. To appear in ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004), 2004.

J. Stam. *Stable Fluids*. ACM Transactions on Graphics (Proceedings of SIGGRAPH 1999), 1999.

D. Weiskopf, G. Erlebacher, T. Ertl. *A Texture-Based Framework for Spacetime-Coherent Visualization of Time-Dependent Vector Fields*. IEEE Visualization 2003. Seattle, Washington, 2003.

J. van Wijk. *Image-based Flow Visualization*. ACM Transaction on Graphics (Proceedings of SIGGRAPH 2002), 2002.

Appendix: Vertex and Fragment Programs in Cg

Vertex programs are used in PGHFlow to precompute texture access patterns, called *stencils* in the CFD literature. Fragment programs are the main computational kernels of PGHFlow, used for every computation on the velocity, pressure , ink and temperature fields. A sample of the fragment programs is listed to give a flavor for possible optimizations. The reader is advised to read them together with the appropriate equations the programs are computing.

- Poisson equation stencil

```
// fivestar_stencil precomputes the tex-
ture coordinates for the
// five-star stencil access pat-
tern of the Poisson solver.
// There are 5 positions we want to ac-
cess: the center of the stencil,
// and directly left, right, up and down. In-
stead of storing all 5
// pairs, we take advantage of the shar-
ing of coordinates between
// them and store all of them in only 2 registers:
//   R1 = (x_left, x_center, x_right, y_center)
//   R2 = (y_up,   y_center, y_down,  x_center)
// This way, we can use swiz-
zling in the fragment pro-
gram to reconstruct
// the texture coordinates without any per-
formance overhead.

void fivestar_stencil(
    uniform float4x4 mvp,
    float4 ipos: POSITION,
    float2 coords: TEXCOORD0,
    out float4 opos: HPOS,
    out half4 xcoords: TEXCOORD0,
    out half4 ycoords: TEXCOORD1,
    out half2 ocoords: TEXCOORD2)
{
    opos = mul(mvp, ipos);
    xcoords = half4(coords.x - 1, co-
ords.x, coords.x + 1, coords.y);
    ycoords = half4(coords.y - 1, co-
ords.y, coords.y + 1, coords.x);
    ocoords = coords;
}
```

- Reduction stencil

```
// reduce_stencil precomputes the sten-
cil texture coordinates for the
// reduce stencil (square-shaped) ac-
cess pattern of the different
// fragment programs for reduction.
// Instead of storing all 4 pairs in 8 co-
ordinates, we take
// advantage of repeated coordi-
nates and compress them into a single
// four-element vector. We recon-
struct the texture coordinates without
// overhead by using appropriate swiz-
zling operations.

void reduce_stencil(
    uniform float4x4 mvp,
    float4 ipos: POSI-
TION, out half2 opos: HPOS,
    half2 coords: TEXCOORD0,
    out half4 ocoords: TEXCOORD0)
{
    opos = mul(mvp, ipos);
    ocoords = half4(coords.x - 0.5, co-
ords.x + 0.5,
                    coords.y - 0.5, co-
ords.y + 0.5);
}
```

- FG field seven-star stencil

```
void sevenstar_stencil(
    uniform float4x4 mvp,
    float4 ipos: POSI-
TION, out float4 opos: HPOS,
    half2 coords: TEXCOORD0,
    out half4 xcoords: TEXCOORD1,
    out half4 ycoords: TEXCOORD2,
    out half4 dcoords: TEXCOORD3,
    out half2 ocoords: TEXCOORD0)
{
    opos = mul(mvp, ipos);
    xcoords = half4(coords.x - 1, co-
ords.x, coords.x + 1, coords.y);
    ycoords = half4(coords.y - 1, co-
ords.y, coords.y + 1, coords.x);
    dcoords = half4(coords.x + 1, co-
ords.y - 1, coords.x - 1, coords.y + 1);
    ocoords = coords;
}
```

- Fragment program to compute the discrete FG field with the Boussinesq buoyancy terms (equations (15) and (16)):

```
void compute_fg_temperature
(half4 xcoords: TEXCO-
ORD1, // these three are
 half4 ycoords: TEXCOORD2, // precom-
puted on
 half4 dcoords: TEXCOORD3, // ver-
tex programs
 half2 coords:  TEXCOORD0,
 out float2 value: COLOR,
 uniform texobjRECT uv,
 uniform texobjRECT flag,
 uniform float4 dxdy,
 uniform float Re,
 uniform float dt,
 uniform float gamma,
 uniform float beta,
 uniform float2 g)
{
  // we place as many val-
ues in float4s and float2s as possible
  // to take advantage of the 4-
element parallelism
```

```
  float4 val5 = f2texRECT(uv, xcoords.yw).xyxy,
    val42 = float4(f2texRECT(uv, xcoords.xw),
                   f2texRECT(uv, ycoords.wx)),
    val68 = float4(f2texRECT(uv, xcoords.zw),
                   f2texRECT(uv, ycoords.wz));
  float2 val73 = {texRECT(uv, dcoords.zw).x,
                  texRECT(uv, dcoords.xy).y};
  float4 ddd2 = (val68 -
2 * val5 + val42) * dxdy.yyww;
  float2 lap = Re * (ddd2.xy + ddd2.zw);
  float4 m5658 = (val5     + val68) / 2.0;
  float4 m4525 = (val5     + val42) / 2.0;
  float4 dc23  = (val5     - val68) / 2.0;
  float4 dc14  = (val42    - val5 ) / 2.0;
  float2 dc65  = (val42.xw + val73) / 2.0;
  float4 t = float4(m4525.x, dc65, m4525.w);
  float4 d = dxdy.xxzz * ((m5658.xzyw * m5658 -
t * m4525) +
         gamma * (abs(m5658.xzyw) * dc23 -
abs(t) * dc14));

  // Boussinesq term for tempera-
ture influence
  float this_t = f3texRECT(uv, coords).z;
  float2 b_term = { this_t + f3texRECT(uv, xcoords.zw).z,
                    this_t + f3texRECT(uv, ycoords.wz).z};
  b_term = -b_term * g * (dt/2) * beta;
  value = val5.xy + dt * (lap + g -
float2(d.x + d.z, d.y + d.w))
                  + b_term;

  // this sets the FG boundary  condi-
tions appropriately
  float4 flags = f4texRECT(flag, coords);
  if (flags.y == 0) value.x = 0;
  if (flags.w == 0) value.y = 0;
}
```

- Jacobi relaxation: Most used fragment program, because the time to solve the Poisson equation dominates the total running time. Notice that there are a lot of obscure techniques to take advantage of the 4-way parallelism of the fragment processor.

```
void jacobi_step(
    half4 xcoords: TEXCOORD0,
    half4 ycoords: TEXCOORD1,
    half2 coords:  TEXCOORD2,
    out float value: COLOR,
    uniform texobjRECT fg,
    uniform texobjRECT p_old,
    uniform texobjRECT neighbors,
    // z=0, so we can throw .z of dot-
ted vector away
    uniform float3 one_over_dxdt_dydt,
    uniform float2 dxdy) // { 1/(dx*dx), 1/(dy*dy) }
{
    float4 nsew = 1 -
f4texRECT(neighbors, xcoords.yw);
    // 4 multiplies with one instruction
    nsew *= dxdy.xxyy;

    float4 val4628;
    val4628.x = f1texRECT(p_old, xcoords.xw);
    val4628.y = f1texRECT(p_old, xcoords.zw);
    val4628.z = f1texRECT(p_old, ycoords.wx);
    val4628.w = f1texRECT(p_old, ycoords.wz);

    // 4 multiplies with one instruction
    val4628 *= nsew.wzyx;

    // fg only stores usely r and g, but we dot it
    // with (...,...,0) so b is thrown away
    // We do this be-
cause there is no dot2 in Cg, only dot3.
    float3 fg5 = f3texRECT(fg, xcoords.yw);
    fg5.x -= f2texRECT(fg, xcoords.xw).x;
    fg5.y -= f2texRECT(fg, ycoords.wx).y;

    // 4 multiplies and 3 adds in one instruction
    float factor = dot(float4(1,1,1,1), nsew);

    // 2 multiplies and 1 add in one instruction
    float  rhs = dot(fg5, one_over_dxdt_dydt);
    if (factor == 0.0)
        value = 0;
    else {
        // 4 multi-
plies and 3 adds in one instruction
        value = 1/fac-
tor * (dot(float4(1,1,1,1), val4628) -
rhs);
    }
}
```

- Ink advection: main part of ink advection fragment programs. We omit auxiliary functions such as the linear interpolating texture lookups.

```
void ink_advection
(float2 coords: TEXCOORD0,
 out float4 value: COLOR,
 uniform float       dt,
 uniform float       supersample,
 uniform float       velscale,
 uniform float2      dxdy,
 uniform samplerRECT uv,
 uniform samplerRECT ink,
 uniform samplerRECT noise1,
 uniform samplerRECT noise2,
 uniform float       noise_alpha,
 uniform float       noise_modulate)
{
  float2 offset = fmod(coords, 1);
  float2 correctionx = float2(offset.x < 0.5 ? 0 : -
1, 0);
  float2 correctiony = float2(0, off-
set.y < 0.5 ? 0 : -1);

  float2 offsetv = offset + correc-
tionx + float2(1, 0);
  float2 offsetu = offset + correc-
tiony + float2(0, 1);
```

```
  // staggeredbilerp is a bilinear interpo-
lating texture lookup that takes
  // into account the stag-
gered setup for u and v
  float u = staggered_bilerp_lookup(uv, co-
ords + correctionx, offsetu).x;
  float v = staggered_bilerp_lookup(uv, co-
ords + correctiony, offsetv).y;

  // This is a lagrangian back-
ward step in time, so that ink advection
  // becomes a gather op instead of a scat-
ter op.
  // velscale is used to in-
crease speeds for interactive visualization
  float2 direction = float2(u,v) * velscale;

  coords -= dt * direction / dxdy;
  coords *= supersample;
  value = bilerp_lookup(ink, coords);

  // modulate noise with a saw-
tooth profile.
  float4 noise1_value = texRECT(noise1, coords);
  float4 noise2_value = texRECT(noise2, coords);
  float t1 = fmod(noise_modulate,1), t2 = 1 -
t1;
  float4 noise_value = t1 * noise1_value + t2 * noise2_value;

  // interpolate noise with previ-
ous ink values
  value = (1-noise_alpha) * value + noise_alpha * noise_value;
}
```

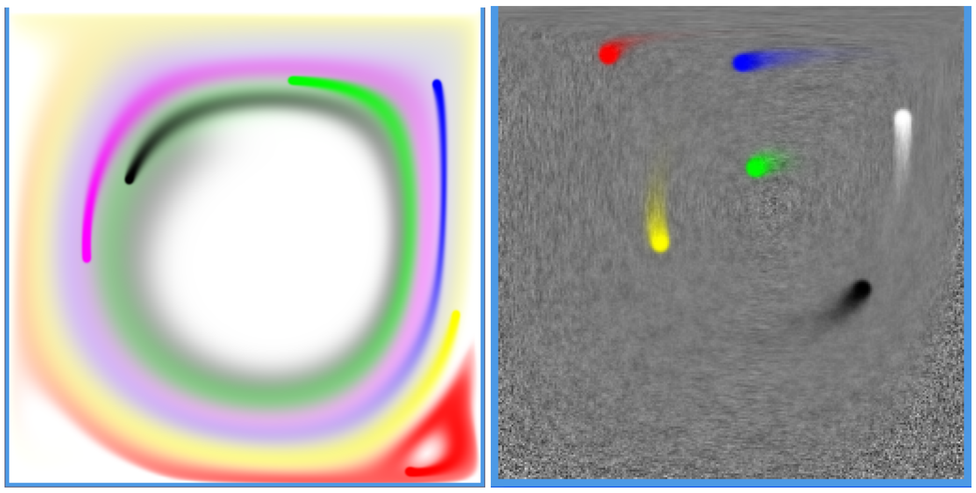

Figure 8: *Lid-driven cavity, visualized with ink field and IBFV.*

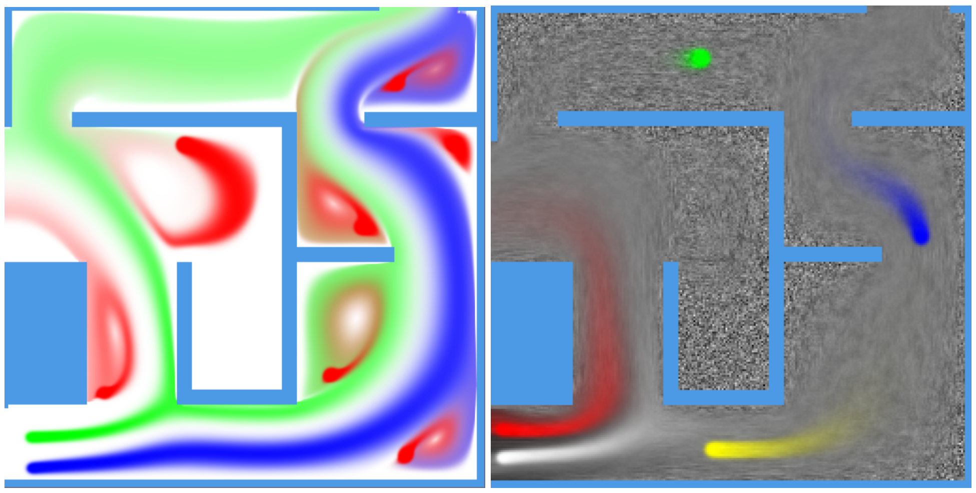

Figure 9: *Domain with obstacles, visualized with ink field and IBFV.*

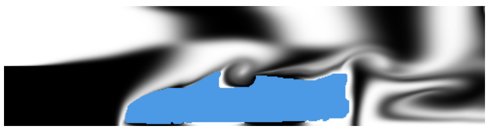

Figure 10: *Wind tunnel mock-up.*

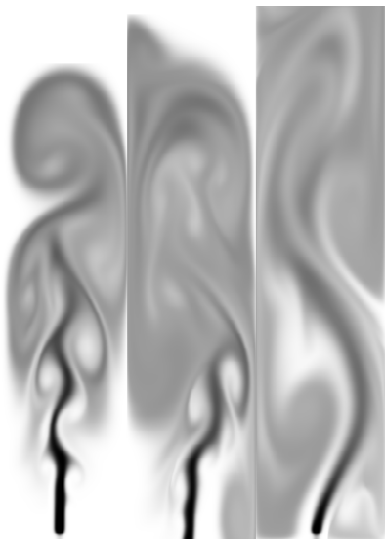

Figure 11: *Smoke simulation with large Re.*

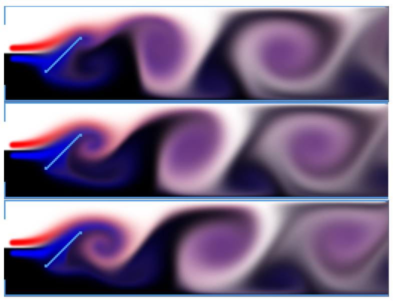

Figure 12: *The* Kárman vortex street.

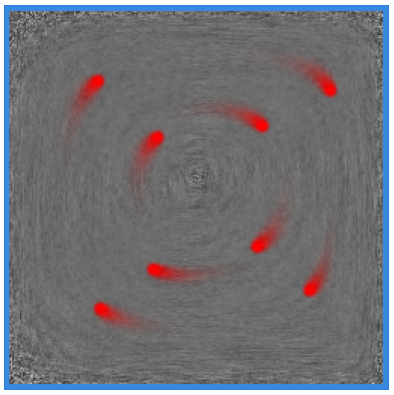

Figure 13: *Natural convection with a heated lateral wall and image-based flow visualization.*