An Improved Study of Real-Time Fluid Simulation on GPU

Enhua Wu[1, 2], Youquan Liu[1], Xuehui Liu[1]
[1]Laboratory of Computer Science, Institute of Software
Chinese Academy of Sciences, Beijing, China
[2]Department of Computer and Information Science,
Faculty of Science and Technology, University of Macau, Macao, China
email: ehwu@umac.mo, lyq@ios.ac.cn, lxh@ios.ac.cn
http://lcs.ios.ac.cn/~lyq/demos/gpgpu/gpgpu.htm

Abstract

Taking advantage of the parallelism and programmability of GPU, we solve the fluid dynamics problem completely on GPU. Different from previous methods, the whole computation is accelerated in our method by packing the scalar and vector variables into four channels of texels. In order to be adaptive to the arbitrary boundary conditions, we group the grid nodes into different types according to their positions relative to obstacles and search the node that determines the value of the current node. Then we compute the texture coordinates offsets according to the type of the boundary condition of each node to determine the corresponding variables and achieve the interaction of flows with obstacles set freely by users. The test results prove the efficiency of our method and exhibit the potential of GPU for general-purpose computations.

KEY WORDS: Graphics Hardware, GPU, Programmability, NSEs, Fluid Simulation, Real-Time

Introduction

Real-time fluid simulation is a hot topic not only in computer graphics research field for the special effects in movies and video games, but also in engineering applications field. However, the computation of Navier-Stokes equations (NSEs) used to describe the motion of fluids is really hard to achieve real-time. A few years ago, the semi-Lagrangian method was introduced to computer graphics community by Stam [20]. This method allows adopting large time step to solve the NSEs with solid stability. Though the method is not yet accurate enough for engineering computation, it can capture the fluid motion characteristics with good visual appearance. Therefore in recent years, it has been widely used to simulate fluid-like material motions in computer graphics.

Noticeably, with the development of hardware, today's commodity graphics hardware begins to provide programmability both at the fragment level and the vertex level. Especially after the fragment program supports IEEE 32 bits float precision, more and more people turn to graphics processing unit (GPU) to solve general-purpose problems [7]. As pointed out by Macedonia that the GPU had entered computing's mainstream [15], the limited parallelism has made GPU surpass CPU in many kinds of computations and people are able to promote computation on low-priced commodity graphics hardware.

This paper focuses on solving the NSEs on GPU. We incorporate the semi-Lagrangian method with the parallelism of GPU to simulate the real time fluid effects with arbitrary boundary conditions set freely by users. The main contributions of this paper include that we incorporate the scalar variables computation with vector variables to reduce the number of rendering pass, and by grouping the grid nodes into different types and generating the texture offsets for the velocity and pressure variables of those nodes which are used to modify those variables on boundaries according to the specified conditions, our method is able to process arbitrary boundary conditions which is more general than other methods [9] and not restricted to outer boundaries.

In the following sections we will firstly introduce previous work on fluid simulation in computer graphics, such as those for smoke, fire, water, etc.; and present some general-purpose applications on GPU as well. Then in Section 3, we will present our improved method in achieving real-time fluid simulation on GPU in detail, including our processing method for boundary conditions. Finally the experimental result and related discussion are given in Section 4 and 5.

Related Work

The simulation of fluid has received much attention in computer graphics for many years. And there is much work related to this field, such as modeling, rendering and dynamic simulation. Here we just refer to some recent work closely related.

To simulate the turbulence in smoke, Stam [19] decomposed the turbulent wind field into two components where the Kolmogorov spectrum was used to model the small-scale random vector field. Foster et al. [4] used an explicit integration scheme to solve the NSEs to simulate the complex behavior of fluids. But they had to set a small time step to keep the whole simulation from blowing. To alleviate this problem, Stam [20] introduced the semi-Lagrangian method to solve NSEs, which is unconditionally stable. But the numerical dissipation was severe so that the vorticity confinement was used to inject the lost energy back into the fluid [3], which added more scale details to the smoke simulation. Later when simulating the complex surface of water, Enright[2] and Foster[5] et al. used this method to update the velocity field as well as in [16] to simulate the flame. Similarly, Rasmussen et al. [17] used the semi-Lagrangian method to simulate the large-scale smoke in 2D, and at the same time the Kolmogorov spectrum was used to add 3D small-scale details.

With the development of GPU, especially the popularization of its programmability, many people turn to solve general-purpose computation problems by using GPU as a stream processor. In 2001, Rumpf et al.[18] used the multi-texture and image subsets of OpenGL to calculate the diffusion equation to smooth images. In 2002 Li et al. [14] mapped LBM (Lattice Boltzmann Method) to graphics hardware with register combiner to simulate the fluid effects. Harris et al. [8] used register combiner with texture shader to solve CML (Coupled Map Lattice) problem to achieve interactive fluid simulation.

With the programmability of fragments on GPU, Krüger et al. [12] computed the basic linear algebra problems, and further computed the 2D wave equations and NSEs on GPU. Bolz et al.[1] rearranged the sparse matrix into textures, and utilized the multigrid method to solve the fluid problem. Similarly, Goodnight et al.[6] used the multigrid method to solve the boundary value problems on GPU. Harris[9， 10] solved the PDEs of fluid motion to get cloud animation.

GPU is also used to solve other kinds of PDEs. For example, Kim et al. [11] solved the crystal formation equations on GPU. Lefohn et al. [13] packed the level-set isosurface data into a dynamic sparse texture format, which was used to solve the PDEs. Another creative usage was to pack the information of the next active tiles into a vector message, which was used to control the vertices and texture coordinates needed to send from CPU to GPU. To learn more applications about GPU for general-purpose computations, readers can refer to [7].

In the following sections, we will introduce our own work in detail about how to use GPU to accelerate the PDEs solving for simulating the fluid effect in real-time.

Incompressible NSEs Solver on GPU

Because the NSEs can describe the general fluid phenomena accurately, we use them in 2D to simulate the fluid effects. But we have to point out that, though we just compute in 2D domain, the method present here is also suitable for 3D problems. The original NSEs mainly consist of two parts, one is continuous equation (1) to ensure mass conservation, and the other is the momentum equation (2) to ensure the momentum conservation.

$$\nabla \cdot \mathbf{u} = 0 \tag{1}$$

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} + v\nabla^2\mathbf{u} - \nabla p + \mathbf{f} \tag{2}$$

here ***u*** is the velocity vector, v is the coefficient to control the diffusion, ***f*** is the external force and p is the pressure. The two scalar variables, density ρ and temperature T, are passively convected by the velocity field ***u***, which is similar to the momentum equation. Here we consider not only the advection but also the self-diffusion.

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + k_\rho\nabla^2\rho + S_\rho \tag{3}$$

$$\frac{\partial T}{\partial t} = -(\mathbf{u} \cdot \nabla)T + k_T\nabla^2 T + S_T \tag{4}$$

The fluid will fall downwards due to the gravity and will rise up due to the buoyancy. Similar to [3, 16, 17], we take the buoyancy into account for the effects of density and temperature on the velocity field:

$$\mathbf{f}_{buoy} = -\alpha\rho\mathbf{z} + \beta(T - T_{amb})\mathbf{z} \tag{5}$$

where ***z*** is the upward vertical direction, (0, 1), T_{amb} is the ambient temperature of the around air. α and β are used to control the density and temperature effects respectively.

To compensate the characteristics smoothing of the small-scale details due to numerical dissipation, similarly, we introduce the vorticity confinement technique as follows,

$$\omega = \nabla \times \mathrm{u} \tag{6}$$

$$\mathrm{N} = \nabla|\omega| / |\nabla|\omega|| \tag{7}$$

$$\mathbf{f}_{conf} = \varepsilon h(\mathbf{N} \times \boldsymbol{\omega}) \tag{8}$$

where ε is used to control the amount of small scale detail added back into the whole velocity field. The spatial step h guarantees that as the mesh is redefined the physically accurate solution can still be obtained.

Here we discretize the rectangle computation domain firstly. Different from the staggered grid of [4], we set the velocity, pressure and other scalar variables on the same nodes to reduce the number of instructions on GPU. Then we compute the whole field at time $t+1$ from the previous time t. After updating the whole field, we render the effects with density and velocity. Here the density distribution is used to simulate the flowing effects.

We use the semi-Lagrangian method to solve Eq. 2, 3, 4. And readers can refer to [20] for more details. Firstly we compute the intermediate velocity field $\boldsymbol{u}^*$ without the pressure term with the following equations (Eq. 9-11), then correct it with the pressure field (Eq. 12-13).

$$\mathbf{u}^* = \mathbf{u}^\mathbf{t} + (\mathbf{f}_{buoy} + \mathbf{f}_{conf} + \mathbf{f}_{user}) \cdot \Delta t \tag{9}$$

$$\partial\mathbf{u}^*/\partial t = -(\mathbf{u}^* \cdot \nabla)\mathbf{u}^* \tag{10}$$

$$\partial\mathbf{u}^*/\partial t = v\nabla^2\mathbf{u}^* \tag{11}$$

$$\nabla^2 p = \frac{1}{\Delta t}\nabla \cdot \mathbf{u}^* \tag{12}$$

$$\mathbf{u}^{\mathbf{t+1}} = \mathbf{u}^* - \Delta t \cdot \nabla p \tag{13}$$

Here we use the first order characteristic method (Eq. 14) to solve the convection equation (Eq. 10).

$$\mathbf{u}(\mathbf{x}) = \mathbf{u}(\mathbf{x} - \Delta t\mathbf{u}(\mathbf{x})) \tag{14}$$

Hardware Implementation

As a stream processor, GPU has many advantages over CPU in general-purpose computations. We use fragment programs to solve all the equations mentioned above that read from a set of input textures and write to an output texture. Though vertex and

fragment are both programmable with full IEEE float precision and vertex programs can support branching and loop, our solver is implemented with fragment programs for several reasons. Fragment programs can fetch texture data directly and possesses the ability to process fragments. Furthermore, the pipelines in fragment level are more than the corresponding one in vertex level, which implies a higher parallelism.

Even though we can assemble many basic operations to compute the complex equations like [12], we prefer to adopt more flexible techniques for optimized performance. On account of the similar form of the velocity, density and temperature equations, we pack these three variables into RGBA-4 channels (the velocity has two components). In this way a fragment program can calculate all the four variables together, and so the changes of states and the number of rendering pass are reduced greatly. We also adopt the same technique to the source term and the boundary condition processing. When the problem has too many scalar variables, this packing method is restricted and additional passes are inevitable. For such cases we can use the idea of flat 3D textures from [9] to map the whole 3D computation domain to the 2D one. Therefore, all the key variables will be solved in one pass.

However we have to pay much attention to some steps during the fragment computation. In some fragment programs the density ρ and temperature T are processed similarly to velocity, but in some other fragment programs, the processing is different slightly.

To attain high performance, we adopt Jacobi iteration method instead of Gauss-Seidel method for higher parallelism. One reason to do so is that the Jacobi method is more efficient than the conjugate gradient method for each iteration even though the latter may converge more quickly. The latter method needs an additional vector reduction operation[1，12] which is a multi-pass operation and requires to retrieve data from GPU to the main memory. And we don’t compute the residuals to determine whether the whole computation is converged like [6], but only simply specify the number of iterations to reduce one rendering pass.

After discretizing the equations, and allocating the texture memory for these variables on GPU, we perform the following steps to simulate the fluid effects:

1. Users interactively input the distributions of obstacles in the whole domain with mouse or outer images, The fragment color is set to zero if the node is occupied by the obstacle, otherwise set to 1. In this way we get an obstacle texture.
2. According to the obstacle texture we compute the node types and generate the texture coordinate offsets for each variable with one fragment program, which will be explained in Section 3.2.
3. Compute the velocity field and other scalar fields with the following steps in turn:
    a) Compute the source term for density and temperature, and compute the force from user input, the buoyancy force from Eq. 5 and the vorticity confinement force from Eq. 8, then add them to the current velocity field with Eq. 9;
    b) Compute the advection equation (Eq. 10), and modify the values on the boundary with the accordingly texture coordinate offsets;
    c) Compute the diffusion equation (Eq. 11) with the Jacobi iteration method with one fragment program. For each iteration, the boundaries need to be processed specially;
    d) Compute the divergence of the current velocity field;
    e) Compute the Poisson equation (Eq. 12) with the Jacobi iteration method, here it is required to modify the pressure values on the boundaries according to the texture coordinate offsets;
    f) Revise the velocity according to the pressure distribution, and modify the velocity on the boundaries;
    g) Compute other scalar variables such as density and temperature. Since we just focus on the 2D problems here, the density and temperature variables are packed with velocity into 4 channels of textures. So actually we have omitted this step.
4. Render the whole scene with density distribution to simulate the flowing effects.

Through experiments, we find that 4 iterations can satisfy the visual effects very well. So the number of rendering pass is 14 for each time step. If we take account of the diffusion procedure, the pass will increase to 22. And if we consider the buoyancy and the vorticity confinement effects, the whole number of rendering pass will be 25 at most. Further, if we want to describe the fluid motion more exactly, we need more iteration, which implies more passes, slower performance.

And if we do not pack the scalar variables such as the density and temperature together with velocity variables, the whole rendering pass number will be larger. For each time step, there would need 4 more passes if we do not consider the diffusion procedure, otherwise it would need 24 more passes, which results in much slower simulation.

In the following we will discuss the boundary condition processing and its implementation in detail.

Boundary Conditions

It is very important to process the arbitrary boundary conditions for real problems. The method in this paper can solve this problem effectively. As shown in Figure 1, because of the existence of boundary conditions, the surrounding nodes determine those ones on the boundaries. In [6] the stencil buffer was used to process the extension of the state-space of the simulation. Different from their work, we absorb the idea from [4] to group the nodes. Our method is similar to that by Harris et al. in [9]. However, rather than just processing the outside boundary like in Harris's method, an improvement has been made in our method to allow arbitrary boundary conditions.

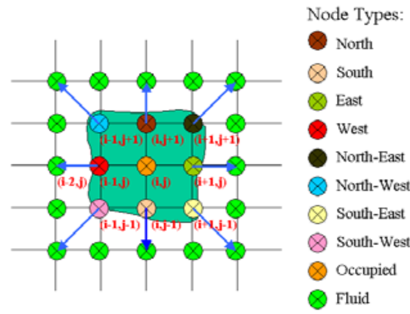


Figure 1: *Boundary condition processing*

(The inside object is the obstacle which is surrounded by fluid, and the arrows stand for offsets)

The boundary conditions include Dirichlet (prescribed value), Neumann (prescribed derivative), and mixed (coupled value and derivative), see Eq. 15:

$$a\phi + b\frac{\partial \phi}{\partial \mathbf{n}} = c \tag{15}$$

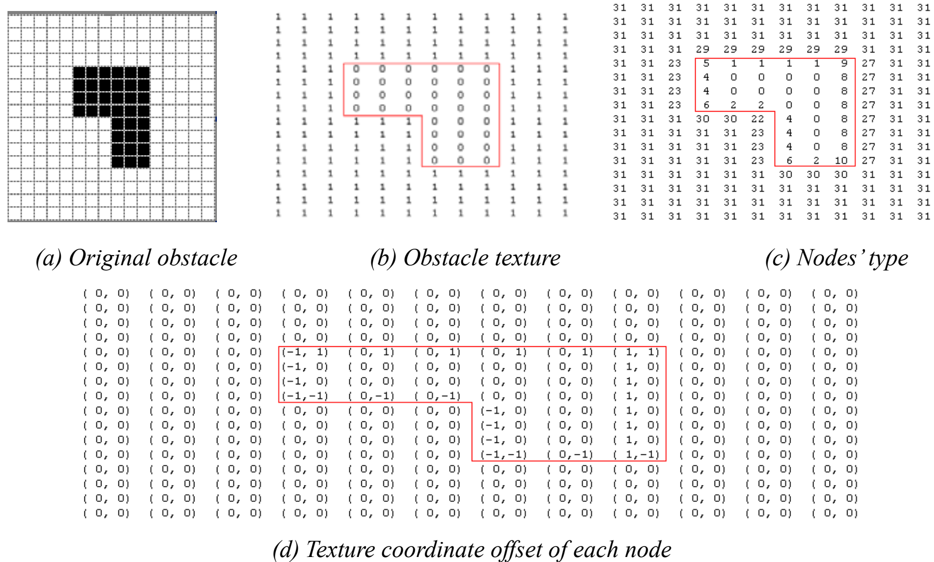
where ϕ stands for velocity, density, temperature or pressure etc., a, b and c are the coefficients. Firstly we divide all the nodes into two types according to the occupation of obstacles (see Figure 2-a), represented by 0 or 1 (see Figure 2-b). We discretize the equation (Eq. 15) with the first order precision, so the values of the node can be determined by a certain node among its surrounding ones. But according to Eq. 15, we have to compute the derivative in the normal direction. For the convenience of the processing on GPU, we simplify the normal direction into 8 directions (see Figure 1). So we only need to check the 8 nodes to get the relative orientation. To process arbitrary complex boundaries we use the following coding method to determine the node type from the obstacles information on GPU.

*Node Type(i,j) = Obstacle(i,j)*16 + Obstacle(i+1,j)*8 + Obstacle(i-1,j)*4 + Obstacle(i,j+1)*1+ Obstacle(i,j-1)*2*

Here *Obstacle(i,j)* can only be 1 or 0 which stands for fluid or obstacle respectively. If we do it in the binary form, it will be clearer to understand this coding scheme. In this way the nodes can be grouped into ten types that represent different texture coordinate offsets. That is $\phi_{boundary} = d\phi_{offset} + e$, where d and e are determined by the discretization of Eq. 15, which are packed

to one texture.

Because the boundary involves the velocity, pressure and density variables etc., in the following steps, we generate the texture of coefficients *d* and *e* for the corresponding variables. For pressure, the pure Neumann boundary condition is used, that's is $\partial p/\partial \mathbf{n}=0$, the values on the boundary are set equal to the values of its adjoining nodes. For example, in Figure 1, for the boundary node (i-1, j), we set its node type West, the texture offset of this node is (-1, 0), then according to $\partial p/\partial \mathbf{n}=0$, we set $p_{i-1,j} = p_{i-2,j,}$. The rest nodes may be deduced by analogy. For velocity on the static boundary, we set the velocity component normal to the obstacle surface to zero; for that on the dynamic boundary, we can set the normal component to the obstacle's velocity at this point. For the non-slip boundary which means that the obstacle drags the fluid at the surface, we set the tangent component of velocity on the boundary to the negative value of velocity of its neighbor fluid nodes; for the slip boundary, the tangent component on the boundary is just set equal to the adjacent corresponding component of fluid directly. To fluid nodes themselves (the tenth node type in Figure 1) and obstacle interior (the ninth node type in Figure 1) the texture coordinate offsets are all zero, meaning that fragment programs will not modify the values of these nodes. All these adjustments are embodied with the coefficients *d* and *e*.



(a) Original obstacle *(b) Obstacle texture* *(c) Nodes' type*

(d) Texture coordinate offset of each node

Figure 2: *The procedure of texture coordinate offsets generation (a→b→c→d)*

We process such general boundary conditions with only two special fragment programs for pressure and velocity respectively. Figure 2 provides a simple example to illustrate the whole procedure.

Obviously, this method can be used to process periodic boundaries by just modifying texture offset of those nodes on one boundary of the domain to the opposite boundary. To dynamic boundaries, we just need to update the texture timely, and adjust the values of the corresponding nodes occupied by obstacles.

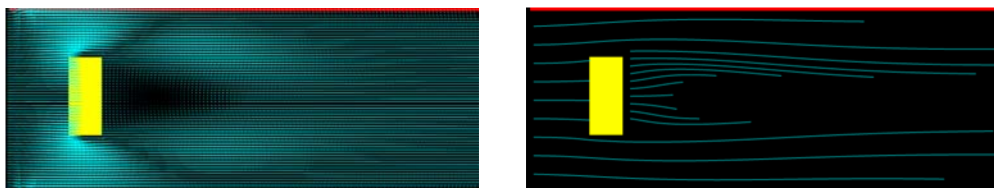


Figure 3: *Velocity vector and streamline graphs with a directly evaluated boundary (Frame577)*

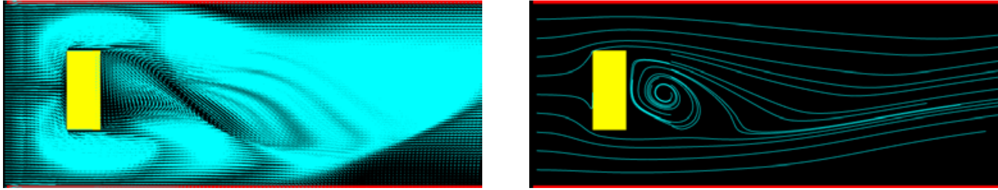


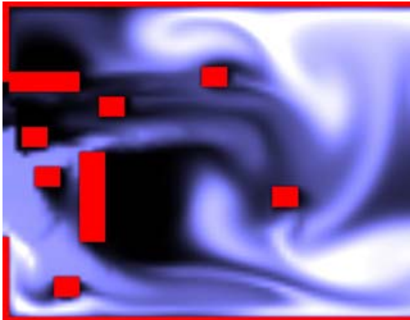
Figure 4: *Velocity vector and streamline graphs with a non-slip boundary (Frame577)*

Apparently the simplest way is to set the variables of those nodes occupied by obstacles equal to the variables from obstacles themselves. But this method is too coarse to describe the fluid effects as shown in Figure 3 and Figure 4 (rectangle blocks stand for obstacles, same to other figures). However, the method with the non-slip boundary condition processing can depict the boundary around the obstacles more exactly, which is more distinguishable when closer to the obstacles.

But as a result of the more general boundary conditions processing, we cannot use line primitives over the boundaries like in [9]. We have to use additional rendering passes to process the boundaries specially. One possible way to reduce the cost is to set up boundary boxes for the obstacles to reduce the fragments needed for processing.

Results and Analysis

Experiments were made by a few test examples, and the test result demonstrated the effectiveness of our method for fast, physically based fluid simulation implemented on programmable graphics hardware. All the testing experiments were implemented on a Dell machine with Intel Pentium 2.8GHz, 2G main memory. The graphics chip is GeForce FX5950 Ultra with 256M video memory and 375MHz core frequency, and the operating system is Windows 2000.



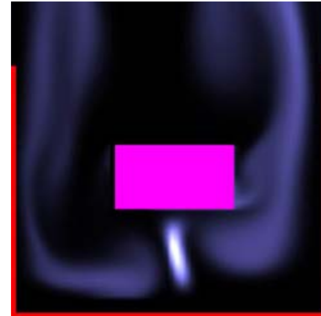
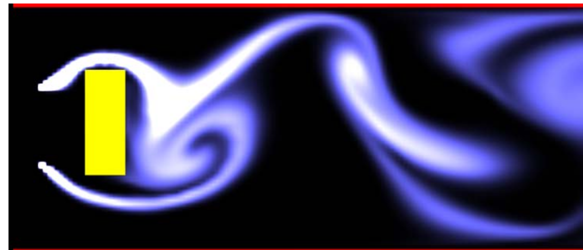


Figure 5: *Fluid flows around arbitrary obstacles input freely by users* ***Figure 6:*** *Natural convection effect (the bottom plate serves as the heat source)*

Some frames in the real time animation are shown here. In Figure 5, we can see the fluid effects running around the arbitrary obstacles interactively input by users. In Figure 6 we present a natural convection effect in an open box whose bottom is heated. In Figure 7, we can see the distinct Von Karman vortex street effect in a classic example in computational fluid dynamics (CFD) field. Further demos can be visited at http://lcs.ios.ac.cn/~lyq/demos/gpgpu/gpgpu.htm.

(a) Frame 1038



(b) Frame 1082

Figure 7: *Flow around an obstacle with the obvious effect of Von Kamann Vortex Street*

Table 1 gives the comparison of the performances of the same algorithm run on GPU and CPU on the same platform. We can find the performance on GPU exceed that on CPU about 14 times in Table 1. To make sure these tests are implemented under the same condition, we didn't consider the diffusion for both, and didn't take account of the buoyancy and the vorticity confinement either. In these tests, 4 iterations were executed for the Poisson equation.

Grid scale	Average GPU time (ms)	Average CPU time (ms)	Speedup
64*64	0.76	3.15	4.1X
128*128	1.15	13.07	11.4X
256*256	30.00	332.30	11.1X
512*512	49.00	719.19	14.7X
1024*1024	201.00	2819.48	14.0X

Table.1: *Comparison of the performance on GPU and CPU (rendering size 741*580)*

Conclusions and Future Work

We adopt the semi-Lagrangian method to solve NSEs on GPU and obtain real-time fluid effects. To further accelerate the whole computational processing, we pack the vector and scalar variables into 4 channels together to reduce the number of rendering pass. As for the boundary conditions, we provide a more general method that can handle arbitrary obstacles in the fluid domain. The experimental results demonstrate the efficiency of our method and exhibit the potential of GPU for general-purpose computations.

Since the whole computation is based on fragments, the bottleneck in our computation should be the fragment processing. We also think that the most efficient computation should rely on the balance among the CPU, AGP bus, vertices and fragments. The work in [13] provided a good example in this aspect with the GPU-to-CPU message passing scheme.

In this paper we just solve the 2D fluid dynamics problem and we would like to extend our work to 3D domain further, to simulate more complex fluid phenomena such as complex water surface. Another problem we have to concern is that the solution so far is still not sufficient enough to solving the general CFD problems due to the low precision of the semi-Lagrangian method adopted. We hope to introduce some other methods from CFD field, such as multigrid method [1, 6] to meet the engineering requirements. For computer graphics research, we would like to enhance the rendering effects of fluid with fragment programs on GPU in the near future.

Up to now, new graphics chips, such as NV40, begin to support multiple rendering targets in fragment program similar to that in vertex program, so that the number of rendering passes could be substantially reduced. Besides, improvement of the video memory allocation scheme could be another point in helping people to do general-purpose computations better on GPU.

Acknowledgements

We would like to thank Mark J. Harris and Aaron E. Lefohn for their assistance with programming on GPU, and Gang Yang for productive discussions. This work is financially supported by the NSFC (60223005, 60033010), the National Grand Fundamental Research Grant of Science and Technology (973 Project: 2002CB312102), and the Research Grant of University of Macau.

References

1. Jeff Bolz, Ian Farmer, Eitan Grinspun and Peter Schröoder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. ACM Transactions on Graphics (Proceedings of SIGGRAPH), pages917-924, July 2003.
2. Douglas Enright, Stephen Marschner, Ronald Fedkiw, Animation and Rendering of Complex Water Surfaces. ACM

Transactions on Graphics (Proceedings of SIGGRAPH), pages736-744, July 2002.

3. Ronald Fedkiw, Jos Stam and Henrik Wann Jensen. Visual Simulation of Smoke. In Proceedings of SIGGRAPH, pages15-22, August 2001.
4. Nick Foster and Dimitri Metaxas. Realistic Animation of Liquids. Graphical Models and Image Processing, pages471-483, Volume 58, Issue 5. September 1996.
5. Nick Foster and Ronald Fedkiw. Practical Animation of Liquids. In Proceedings of SIGGRAPH, pages23-30, August 2001.
6. Nolan Goodnight, Cliff Woolley, David Luebke and Greg Humphreys. A Multigrid Solver for Boundary Value Problems Using Programmable Graphics Hardware. In Proceeding of Graphics Hardware, pages102-111, July 2003.
7. GPGPU website. http://www.gpgpu.org
8. Mark J. Harris, Greg Coombe,Thorsten Scheuermann and Anselmo Lastra. Physically-Based Visual Simulation on Graphics Hardware. In Proceedings of Graphics Hardware, pages109-118, September 2002.
9. Mark J. Harris, William V. Baxter III, Thorsten Scheuermann and Anselmo Lastra. Simulation of Cloud Dynamics on Graphics Hardware. In Proceedings of Graphics Hardware, pages92-101, July 2003.
10. Mark J. Harris. Real-Time Cloud Simulation and Rendering. PhD thesis, 2003.
11. Theodore Kim and Ming C.Lin. Visual Simulation of Ice Crystal Growth. In Proceedings of SIGGRAPH/Eurographics Symposium on Computer Animation, pages86-97, July 2003.
12. Jens Krüger and Rüdiger Westermann. Linear Algebra Operators for GPU Implementation of Numerical Algorithms. ACM Transactions on Graphics (Proceedings of SIGGRAPH), pages908-916, July 2003.
13. Aaron E. Lefohn, Joe M. Kniss, Charles D. Hansen and Ross T. Whitaker. Interactive Deformation and Visualization of Level Set Surfaces Using Graphics Hardware. In IEEE Visualization, pages75~82. 2003.
14. Wei Li, Xiaoming Wei, and Arie Kaufman. Implementing Lattice Boltzmann Computation on Graphics Hardware. The Visual Computer, vol. 19, no.7-8, pages444~456. 2003.
15. Michael Macedonia. The GPU Enters Computing's Mainstream. IEEE Computer Society, pages106-108, October 2003.
16. Duc Quang Nguyen, Ronald Fedkiw and Henrik Wann Jensen. Physically Based Modeling and Animation of Fire. ACM Transactions on Graphics (Proceedings of SIGGRAPH), pages721-728, July 2002.
17. Nick Rasmussen, Duc Quang Nguyen，Willi Geiger and Ronald Fedkiw. Smoke Simulation for Large Scale Phenomena. ACM Transactions on Graphics (Proceedings of SIGGRAPH), pages703-707, July 2003.
18. Martin Rumpf and Robert Strzodka. Using Graphics Cards for Quantized FEM Computations. In Proceedings of VIIP, pages98–107, 2001.
19. Jos Stam and Eugene Fiume. Turbulent Wind Fields for Gaseous Phenomena. In Proceedings of SIGGRAPH, pages369-376, September 1993.
20. Jos Stam. Stable Fluids. In Proceedings of SIGGRAPH, pages121-128, July 1999.

The Journal of Computer Animation and Virtual World, *John Wiley & Sons* (CASA2004), July 2004.