Real-Time 3D Fluid Simulation on GPU with Complex Obstacles

Youquan Liu[1,3], Xuehui Liu[1], Enhua Wu[1,2]
[1]*Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China*
[2]*Department of Computer and Information Science,Faculty of Science and Technology, University of Macau, Macao, China*
[3]*Graduate School of the Chinese Academy of Sciences, China*
{lyq,lxh}@ios.ac.cn; ehwu@umac.mo

Abstract

In this paper, we solve the 3D fluid dynamics problem in a complex environment by taking advantage of the parallelism and programmability of GPU. In difference from other methods, innovation is made in two aspects. Firstly, more general boundary conditions could be processed on GPU in our method. By the method, we generate the boundary from a 3D scene with solid clipping, making the computation run on GPU despite of the complexity of the whole geometry scene. Then by grouping the voxels into different types according to their positions relative to the obstacles and locating the voxel that determines the value of the current voxel, we modify the values on the boundaries according to the boundary conditions. Secondly, more compact structure in data packing with flat 3D textures is designed at the fragment processing level to enhance parallelism and reduce execution passes. The scalar variables including density and temperature are packed into four channels of texels to accelerate the computation of 3D Navier-Stokes Equations (NSEs). The test results prove the efficiency of our method, and as a result, it is feasible to run middle-scale problems of 3D fluid dynamics in an interactive speed for more general environment with complex geometry on PC platform.

1. Introduction

With the rapid development of hardware today, it becomes a hot topic to take advantage of the programmability and parallelism to realize complex computation on the commodity graphics hardware. Especially after the fragment program supports full IEEE single precision and the high level shading language becomes popular, the graphics processing unit (GPU) had rapidly entered computing's mainstream [22], and more and more people turn to GPU to solve general-purpose problems [9]. The limited parallelism has made GPU over CPU in various kinds of computations.

Real-time fluid simulation is of importance not only in the engineering applications but also in computer graphics research field for the special effects in movies and video games. In order to describe the flow phenomena precisely in fluid simulation, we have to solve complex equations, such as the Navier-Stokes equations (NSEs). However, the solution to the NSEs had been suffering from the limitations that it is not only time-consuming but also sensitive to the time step size for the reason of stability, until several years ago when Stam [28] introduced the semi-Lagrangian method for the fluid simulation in computer graphics. The method by Stam allows large time-steps to be applied in solving the NSEs with solid stability. Though the method is not accurate enough for engineering computation, it does capture the characteristics of fluid motion with nice visual appearance. Therefore in recently years, the method has been widely used to simulate the motions of fluids in computer graphics [4,5,7,23,25,29]. In taking advantage of the parallelism and programmability of GPU to accelerate the fluid dynamics problem computation, much work has been proposed to migrate the solution of partial differential equations (PDEs) from CPU to GPU to accelerate the whole simulation of fluid flow [2,8,12,16]. But until now most of them focus on the 2D domain with simple processing of boundary condition due to the reason in lack of flexible control operations on GPU. However, for real-world applications, the fluid would permeate where it can pass, and the ability to specify arbitrary complex boundary conditions is of essence and significance.

This paper focuses on solving the NSEs with the semi-Lagrangian method on GPU, for a purpose of running on GPU with the ability to process arbitrary boundary conditions generated from a complex geometry scene, while keeping the high performance of parallelism of GPU. By the solution, we are able to simulate complex flow effects in real-time.

To cope with the complex boundaries produced from comprehensive obstacles, we clip the whole 3D scene with stencil buffer to get the actual boundaries despite of the complexity of the whole geometry scene. To promote the physical calculation on GPU at the fragment level we pack those scalar variables with similar features into RGBA 4 channels of a single texel to reduce the number of rendering passes. And by grouping the voxels into different types according to the position of obstacles, generating the offsets and the modification factors for the velocity and pressure variables of those voxels, arbitrary internal boundary conditions, more general than other methods [12] could be processed without restriction to outer boundaries. A method of ray marching is used on GPU to obtain the shading effects of semi-transparent material, extendable on new graphics hardware. We integrate the fluid computation and rendering into a seamless system, even without any intervention of CPU apart from the user's interactive operations. As a result, it is feasible to run middle-scale problems of 3D fluid dynamics in an interactive speed for more general environment with complex geometry on PC platform.

In the following sections we will firstly introduce previous work on fluid simulation in computer graphics, such as those for smoke, fire, water, etc. Then we will discuss some general-purpose applications on GPU as well. In Section 3, we will present our method in achieving real-time 3D fluid simulation on GPU in detail including our processing method for the arbitrary boundary conditions. In Section 4, the rendering method based on GPU is described. Finally the experimental results and related discussion are given in Section 5 and 6.

2. Related Work

The simulation of fluid has received much attention in computer graphics community for many years. And there is much work related to this field. Here we only introduce some recent work closely related.

To simulate the turbulence of smoke, Stam [27] 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. [6] 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 [28] introduced the semi-Lagrangian method to solve NSEs that 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 [5], which added the small-scale perturbation to the smoke simulation. Later when simulating the complex surface of water, Enright [4] and Foster [7] et al. used this method to update the velocity field as well as in [23] to simulate the two-phase flow. Similarly, Rasmussen et al. [25] 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. [26] used the multi-texture and image subsets of OpenGL to calculate the diffusion equation to smooth images. In 2002, Li et al. [20] mapped LBM (Lattice Boltzmann Method) to graphics hardware with register combiners to simulate the fluid effects. Harris et al. [11] used register combiners with texture shader to solve CML (Coupled Map Lattice) problem to achieve interactive fluid simulation.

From the year of 2003, with the programmability of fragments on GPU, Krüger et al. [16] computed the basic linear algebra problems, and further computed the 2D wave equations and NSEs on GPU. Bolz et al. [2] rearranged the sparse matrix into textures, and utilized the multigrid method to solve the fluid problem. Similarly, Goodnight et al. [8] used the multigrid method to solve the boundary value problems on GPU. More similar to our work, Harris et al. [12,13] solved the NSEs of fluid motion to get cloud animation. Reference [14] provides a simple example in 2D. Batty et al. [1] used the preconditioned conjugate gradient approach on GPU in their fluid simulation system.

GPU is also used to solve other kinds of PDEs. For example, Kim et al. [15] solved the crystal formation equations on GPU. Lefohn et al. [18] packed the level-set isosurface data into a dynamic sparse texture format that 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 [9].

To achieve the realistic flowing effects, attentions have been paid to the rendering of these volume materials. For the reason of semi-transparent property of the volume, researchers trace rays directly through a set of particles. Reference [27] rendered the gas with a standard ray-tracer combined with a tree data structure. Reference [25] used ray marching to compute the light attenuated from light source, treated each particle as a small blackbody radiator to simulate incandesent effects, and the light scattering was also included. Reference [10,12] took two passes with hardware blending to render the cloud with shading. Based on recent GPU, Krüger et al. [17] implemented ray marching directly on GPU with multi-pass, where early ray termination and empty-space skipping were also addressed. This method is object-order independent, much suitable for those large volumetric data sets including opaque structures exhibiting occlusions and empty regions.

In the following sections, we will introduce our own work in detail on how to use GPU to accelerate the PDEs solution for simulating the flow around complex obstacles in 3D domain in real-time. And at the same time GPU is also used to cast shadow on the fluid volume itself.

3. Three-dimensional Flow Solver on GPU

To simulate the fluid phenomena accurately, researchers in computational fluid dynamics (CFD) field have done a lot of work. In computer graphics domain, for those low-speed flowing, such as smoke, fire, or water, they are issued to be incompressible without viscosity. Therefore, impressible NSEs could be applied to express the flowing effects. In this paper, part of our work is the same as those previously proposed [5,12,23,25,28]. To make the whole material complete, however, we still address them in the following section.

3.1. The Equations of Fluid Flow

The original NSEs mainly include two parts, the continuous equation (1) to ensure mass conservation, and the momentum equation (2) to ensure the momentum conservation.

$$\nabla \cdot \mathbf{u} = 0 \qquad (1)$$

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

Here u is the velocity vector, v is the coefficient for control of the diffusion, f is the external force and p is the pressure.

To illustrate the flowing effects, the two scalar variables, density ρ and temperature T are introduced which are passively convected by the velocity field $\mathbf{u}$, similar to the momentum equation.

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

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

The semi-Lagrangian method is used to solve Eq. 2, 3, 4, and readers can refer to [28] for more details. Firstly we compute the intermediate velocity field without the pressure term, and then correct it with the pressure field. And the implementation in detail is the same to that of the 2D problem, which can also be found in [31], illustrated in the Figure 1. And it happens totally on GPU.

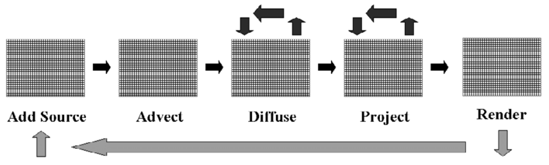


Figure 1: the Algorithm Flow

Both the density and temperature field affect the velocity distribution. The fluid will fall downwards due to the gravity and will rise up due to the buoyancy. Similar to [5,23,25,31], a linear relation is used to take the buoyant force into account:

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

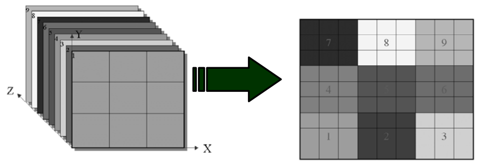
Here $\mathbf{y}$ is the upward vertical direction, (0, 1, 0), and T_{amb} is the ambient temperature of the around air. α and β are used to control the density and temperature effects respectively. These parameters can be adjusted to adapt to the specific problems.

To compensate the characteristics smoothing of the small-scale details due to numerical dissipation of Semi-Lagrangian method, similarly we introduce the vorticity confinement force $\mathbf{f}_{conf} = \varepsilon h(\mathbf{N} \times \boldsymbol{\omega})$, where $\boldsymbol{\omega} = \nabla \times \mathbf{u}$, $\mathbf{N} = \nabla|\boldsymbol{\omega}| / |\nabla|\boldsymbol{\omega}||$. And 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.

3.2. Texture Setup for Computation

Similar to a SIMD machine, GPU has many advantages over CPU in general-purpose computations, in particular when the programmability at vertex and fragment level is supported and full IEEE single-precision floating point throughout the whole pipeline is provided. Similar to [12,16], we map the whole computation domain directly to texture memory and use fragment programs to solve the equations described above, different from the method in [1], where the matrix representing Laplacian operator are rearranged into 7 seperate textures.

We firstly discretize the computation domain, that's, the bounding box of the whole scene that we are interested in. As shown in Figure 2 (a), we dice up the domain into identical voxels along one proper coordinate axis. Different from the staggered grid in [5,6], the velocity, pressure and other scalar variables are defined at the same center of each voxel to reduce the number of instructions needed on GPU.



(a) 3D texture (b) Flat 3D texture

Figure 2. Flat 3D Textures

Our solver uses two voxel grids for all the main variables, including velocity, density and temperature. And we update one at time t+1 from the other one of time t, and then swap them for the next step. During the numerical simulation, iterative technique is required to solve the Poisson equation for pressure and the diffusion equation. Due to the fact that current graphics hardware does not allow both read and write the same memory in the same pass with fragment programs, the iterative methods currently used on GPU are just limited to Jacobi solver [12,13], conjugate gradient solver[1,2,16], and Red-Black Gauss-Seidel solver[8, 12,13]. However, in this paper, we just take fixed iterations of Jacobi solver to meet the visually acceptable requirements, not like [8] to iterate until converged, which can be implemented with the occlusion query feature of recent GPU. The Jacobi method is more efficient than the conjugate gradient method for each iteration even though the latter converges more quickly. The conjugate gradient method needs an additional vector reduction operation [2,16] which is a multi-pass operation and requires to retrieve data from GPU to the main memory. And the Red-Black Gauss-Seidel method also requires two passes to complete one iteration.

Though we can pack one scalar variable and the velocity variable (three components) together into a single RGBA texel like that proposed in [31] for 2D fluid dynamics to reduce the whole rendering pass, just one color output from fragment program is supported in a single pass on our graphics hardware. However, there are multiple scalar variables (density and temperature etc.), therefore it's inevitable to employ additional passes to compute those scalar variables. Even so, we can also pack the temperature and the density together into a single RGBA texel to reduce the rendering pass. In this paper just two channels are occupied, but we can still incorporate other scalar variables into it.

With our packing method, we compute the velocity field first, and then compute the density and temperature fields simultaneously. After updating the whole field, we render the flowing effect. Here the density and temperature distribution can be used to simulate the smoke, the fire or other effects [5,23].

Another important approach to reduce the rendering pass is to use flat 3D textures instead of 3D textures, or a stack of 2D textures, put forward by Harris et al. [12, 13]. With flat 3D textures, only one texture update is required for the whole 3D computation domain as shown in Figure 2(b). Different slightly from [12], we precompute a 2D texture instead of 1D lookup texture that contains the position of front and back slices in neighbor for each slice. In this way we prevent from computing the position in fragment programs in each pass.

3.2.1. Boundary Generation. It is very important to process the arbitrary boundary conditions for real problems. The boundaries make the flowing distinguishable from each other. And the fluid flowing around the obstacles generates many interesting effects. In the 3D computation domain, we can process the boundary conditions easily on CPU [33] since CPUs are designed for powerful controls. But it's difficult to implement on GPU. And if we process the boundary conditions on CPU, and compute the main equations on GPU, we have to read back the result data from GPU to CPU, which is time consuming. What's more, we have to determine each voxel's relation with the polygon scene by some algorithm to decide whether it is occupied by the obstacles or not. So it's better to process the boundary conditions directly on GPU with a novel method.

However the current GPU we used does not support branch operations within fragment programs natively. We generate the boundaries in image space. By using this boundaries image to generate the offsets of each voxel and the modification factors of main variables (velocity, pressure) according to the boundary conditions, we can use two fragment programs to perform the variables modification on GPU directly. Figure 3 gives the whole procedure of the boundary processing. More details on the boundary conditions processing will be explained in Section 3.3.

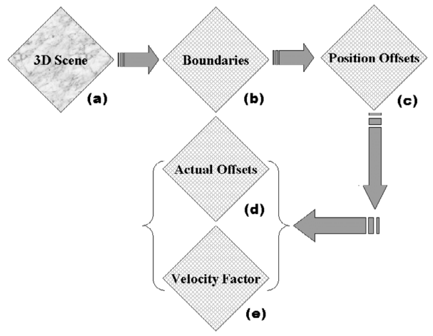


Figure 3. The whole initialization procedure

More recently, the paper [21] used depth peeling to voxelize the scene to generate dynamic boundaries for LBM. But different from their method, our method is based on Semi-Lagrangian. A straightforward approach to generate the boundaries in image space is to use the clipping operation. But this method does not work correctly because the scene is composed of surface geometry objects (See Figure 4 (b)). Here we introduce capping clipped solids with the stencil buffer to get the correct boundaries. By this method we can maintain the cross section when slicing the 3D scene along one axis as shown in Figure 4(c).

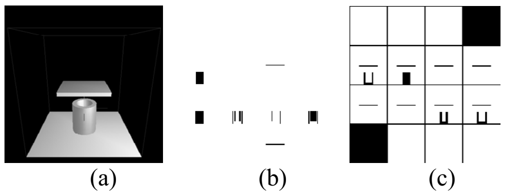


Figure 4. Boundary generation with 16 slices along z-axis

((a) is the solid model, (b) is the texture chart with just clipping and (c) is the capping clipped texture chart)

With the stencil buffer operations, we clip the whole bounding box along the chosen axis slice by slice. In this way we get precise cross sections that represent the boundaries. Due to the flat 3D textures used, the cross sections are reorganized into one texture chart. Readers can refer to [30] to learn how to do so in detail. Note that in order to process arbitrary 3D surface models, firstly object surface polygons must have their vertices ordered so that they face away from the interior to facilitate the face culling process.

But actually, during the advection equation computation with one fragment program, we have to make sure that the particle tracing will not penetrate the boundaries of each slice, which is tiled on the flat 3D textures. As shown in Figure 4 (b), we add an actual geometry boundary box to the 3D scene as the boundaries to avoid the penetration. The cross lines, the left-bottom and the right-top block stand for the boundaries besides those colored regions, somewhat like the line primitives used in [12].

3.3 Process of Boundary Conditions

After we obtain the boundaries, we have to modify those values calculated according to the specified boundary conditions. For the existence of boundary conditions, as shown in Figure 5, the values on the boundaries are determined by the surrounding ones. In [8] 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 [6] to group the nodes. However, rather than just processing the outside boundary in Harris's method [12], an improvement has been made in our method to allow arbitrary boundary conditions.

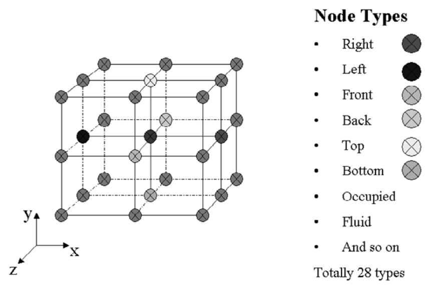


Figure 5. Relative orientation of voxels

The boundary conditions include Dirichlet (prescribed value), Neumann (prescribed derivative), and mixed (coupled value and derivative), as expressed by the equation:

$$a\phi + b\,\partial\phi/\partial\mathbf{n} = c \tag{6}$$

Where ϕ stands for velocity, density, temperature or pressure etc, *a, b* and *c* are the coefficients. Firstly we divide all the voxels into two types according to the occupation of obstacles (see Figure 4(c)), represented by 0 or 1. The equation (Eq. 6) is discretized with the first order precision, so the values of one voxel can be determined by a certain voxel among its surrounding ones. But according to Eq. 6, we have to compute the derivative in the normal direction. For the convenience of the processing on the GPU, we simplify the normal direction into 28 directions as shown in Figure 5, with each node as a voxel. Thus we only need to check the 7 closest nodes to get the relative orientation with coding method as shown below to determine the node type from the obstacles information on GPU. That is

$\phi_{boundary} = d\phi_{offset} + e$, where *d* and *e* are determined by the discretization of Eq. 6, serving as the modification factors.

*Node's Type = Obstacle(i,j,k)*64 +*
*Obstacle(i+1,j,k)*1 + Obstacle(i-1,j,k)*2+*
*Obstacle(i,j+1,k)*8 + Obstacle(i,j-1,k)*4+*
*Obstacle(i,j,k+1)*16 + Obstacle(i,j,k-1)*32*

Here *Obstacle(i,j)* can only be 1 or 0 which stands for fluid or obstacle respectively. With this coding scheme, we could just use 7 nodes to determine the 28 directions regardless of the complexity of the 3D scene. Table 1 gives the result of this coding scheme. We arrange 64 results of the coding scheme into 28 directions, stored in the position offset texture (Figure 3 (c)). Then we use the offsets to generate the actual texture offsets on the flat 3D textures (Figure 3 (d)). At the same time, the velocity factor is calculated according to the boundary conditions (Figure 3 (e)), which will be explained in the following paragraphs. This can be implemented easily with only two fragment programs.

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. Currently in our system, we use the Neumann boundary condition for pressure, that is $\partial p / \partial \mathbf{n} = 0$. The values on the boundary are set equal to the values of its adjoining nodes, and so the factor for pressure is always 1. For the static boundary, we set the velocity component normal to the obstacle surface to zero; for the dynamic boundary, we can set it to the obstacle's velocity at this point. For the non-slip boundary where the obstacle drags the fluid at the surface, we set the tangent component of the velocity on the boundary to the negative value of velocity of its neighbor fluid nodes; for the free slip boundary, the tangent component on the boundary is just set equal to the adjacent corresponding component of the fluid directly. In this paper only non-slip boundaries are used which is embodied with the velocity factor stored in one texture (Figure 3 (e)). It is easier to understand in a 2D domain, which can be found in the previous work from the authors [31].

We process such general boundary conditions with only two special fragment programs for pressure and velocity respectively. To fluid nodes themselves and obstacle interior the texture coordinate offsets are all zeros, meaning that fragment programs will not modify the values of these nodes.

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

To the scalar variables, it is much simpler to implement the boundary conditions. The temperature and density variables of those occupied voxels are just set to the corresponding ones of the obstacles.

However, as a result of the more general boundary conditions processing, we cannot use line primitives over the boundaries like in [12]. 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.

Besides that we can clip the scene along one coordinate axis, we can choose one proper axis to clip the scene to optimize the computation. For example, to the 3D pipe flowing, the parameters vary more greatly along radial direction than those along circular direction. So we can generate the boundaries along the circle direction with fewer slices, as seen from Figure 6. In this way we can simulate those true 3D problems at a little lower cost, unlike [21,25] and more general than [21]. But in such kind of situations, we should pay attention to the interpolation between different

Table 1. The result of this coding scheme (T_{offset} is the flat 3D texture to index front and back slices)

Relative Orientation	Coding (Binary Form)	3D offset (Figure 3 (c))	Actual texture offset (Figure 3 (d))	Velocity Factor (Figure 3 (e))
Right	000001	(1,0,0)	(1,0)	(0,-1,-1)
Left	000010	(-1,0,0)	(-1,0)	(0,-1,-1)
Bottom	000100	(0,-1,0)	(0,-1)	(-1,0,-1)
Top	001000	(0,1,0)	(0,1)	(-1,0,-1)
Front	010000	(0,0,1)	($T_{offset-x}$, $T_{offset-y}$)	(-1,-1,0)
Back	100000	(0,0,-1)	($T_{offset-z}$, $T_{offset-w}$)	(-1,-1,0)
Bottom-Right	000101	(1,-1,0)	(1,-1)	(0,0,-1)
Top-Right	001001	(1,1,0)	(1,1)	(0,0,-1)
…	…	…	…	…

slices, which is different from that between parallel slices [25].

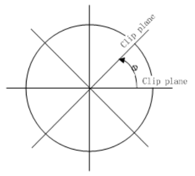


Figure 6. The cross section of the 3D computation domain

With the idea of flat 3D textures, the passes needed is nearly same to the 2D problem. In our current system, we just apply it to the static scenes. But this method here is still adaptable to those dynamic scenes.

4. Rendering

For each time step after the whole field is calculated and stored in the flat 3D textures, we use the density distribution to simulate the flowing effects. In order to take account of the light attenuation along the light path, the rendering process consists of two steps. In the first step the intensity of light reaching the center of each voxel is calculated. In order to do so, we can integrate the radiance along the light ray for each voxel directly within one fragment program. The opacity of each voxel is calculated as $\alpha = 1 - \boldsymbol{exp}(-\tau \cdot \rho \cdot dz)$, where dz is the depth of the voxel in the direction of ray. However the current pixel shader does not support loop operation, so we have to process all slices along the light ray for each voxel. If the point of intersection is not between the current voxel and light source, we ignore it with adding a zero value to the current voxel's radiance. The result including attenuation and opacity of each voxel is written to a flat 3D texture. After calculating the light attenuation on every voxel we render the slices in an order from back to front with hardware blending similar to [3,10,32]. In this way we get the shading effects of the volume fluid. Although our approach is simple to implement and fast, it still has many limitations. Due to the constraint of limited instruction number, the multiple light scattering is not considered, and interpolation of the radiance along the light ray has not been done either, with just the value from the nearest voxel along the light ray sampled. We expect that all these limitations could be removed on the new GPU [24] to get a better shading effect.

In this paper, we generate the fragment program for shading computation according to the scale along the clipping plane. But on our current GPU, the number of instructions and registers will overflow the resource limits if we use Cg to compute the shading, so we rewrite the fragment program for shading computation with assemble language with limited number of the slices. This will not be a problem again with fixed iteration-count loops supported in fragment programs without length limitation [24]. Our approach is easily extendable on these new GPUs.

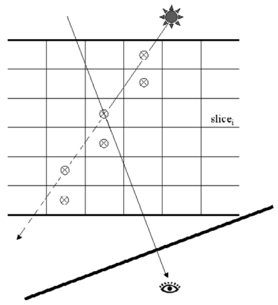


Figure 7. Two passes rendering on GPU

In our current system, if we want to draw the velocity vector graph, which consists of line primitives, we have to fetch the result texture of velocity from GPU to main memory, and then regenerate the vertex arrays sent to GPU for drawing again. With Render-To-Vertex Array we can use particles to visualize the flowing effects on GPU without communication between these two processors, see Figure 8.

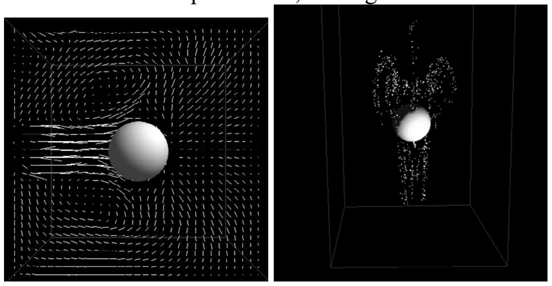


Figure 8. Velocity vector and particles tracing graphs of one slice around a sphere

5. Results and Discussion

Experiments were made by a few test examples, and the test result demonstrated the effectiveness of our method for fast, physical-based 3D 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. The OS is Windows 2000. And all the computation in this paper is based on 32-bit float precision to make it suitable to real-world problems.

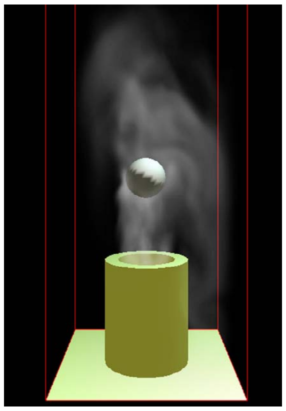


Figure 9. Flowing out of a barrel with shading

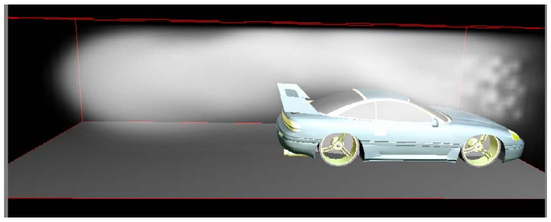


Figure 10. Flowing around a car without shading

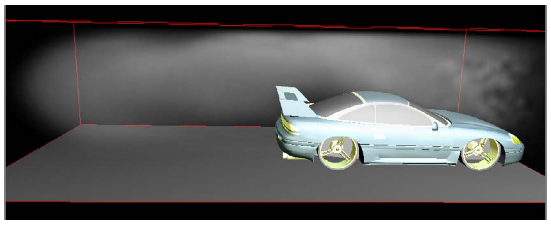


Figure 11. Flowing around a car with shading

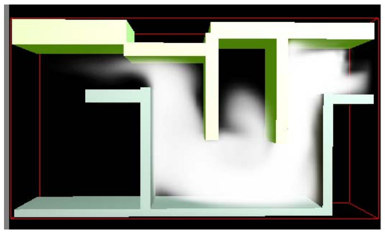


Figure 12. Flowing in a maze without shading

Some frames in the real time animation are as shown here. In Figure 9, the smoke rises from the inside of a barrel. In Figure 10 and Figure 11 the smoke flows around a car. If we consider the shading effects, the whole animation looks much better. In Figure 12 and 13, we demonstrate the flowing in a maze and a city respectively. Further demos can be visited at http://lcs.ios.ac.cn/~lyq/demos/gpgpu/gpgpu.htm.

Currently, we just use copy to textures during the whole process. So still, there is room for improvement. With Render-to-Texture we can get better performance [8,18]. The whole time consists of four parts. In the static scene, the boundaries generation and the preparation of boundary conditions just need to be processed once. So during the animation, we only have to compute the whole field and render the 3D scene, which make up the main part of time consuming.

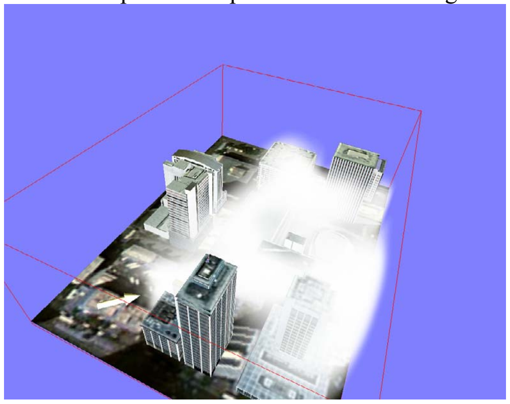


Figure 13. Flowing in a city

The car scene is made up of 14978 triangles (Figure 10 & Figure 11), the boundaries generation takes about 75 ms, and the rendering of geometry scene takes 11.5ms. If we do not consider the diffusion procedure, the simulation of the car scene is about 42.8 fps with $64\times17\times16$ voxels. And the simulation of the maze scene (Figure 12) is about 21.4fps with $64\times34\times16$ voxels. Here the cost of time includes rendering one and just 6 iterations are used to meet the visual effects requirement. We can find from Figure 14 that more iterations means less speed, and the computation time makes up the most part of the total one. Large grid size will lead to drastic performance decrease due to the texture memory limitation.

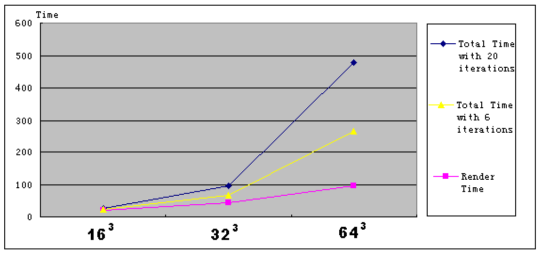


Figure 14. The cost time of simulation (ms)

6. Conclusions and Future Work

We solve the NSEs totally on GPU and obtain real-time fluid effects. To further accelerate the whole computational processing, we pack the scalar variables into 4 channels together to reduce the number of rendering passes. With regard to the complex boundary conditions, we provide a more general method that could handle arbitrary obstacles in the fluid domain, regardless of the complexity of the whole 3D scene. The experimental results demonstrate the efficiency of our method.

Due to the limitation of texture memory on graphics card, the method presented is not yet suitable for those large-scale problems [25]. In such kind of situation, texture compression and multi-pass scheme with blocks may be a solution to alleviate this problem.

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, vertex processing and fragment processing. References [18,19] provided us a good example to do so with the GPU-to-CPU message-passing scheme.

Our current implementation is based on the assemble language. The popularity of high level shading language will help us to get much more optimized performance with some tools provided from chip manufacturers. So we would like to transfer our algorithm to CG in the near future.

We would like to further extend our work to simulate more complex fluid phenomena such as complex water surface, fire etc. with realistic rendering. And we hope to introduce some other high-precision methods from CFD field, such as multigrid method [2,8] to meet engineering requirements instead of the low precision of the semi-Lagrangian method.

With the latest graphics chips, such as NV40 [nvidia], we expect to reduce the rendering pass number further with the ability to support multiple render targets in fragment programs. Besides, improvement of the video memory allocation scheme [19] could be another issue crucial in assisting people to do general-purpose computations better on GPU.

Acknowledgements

The authors would like to thank Mark J. Harris, Jens Krüger and Aaron E. Lefohn for their assistance in programming on GPU, to thank the reviewers for their suggestions. And the car model is downloaded from www.3dcafe.com, the city model from www.Planet9.com. The work is supported by the NSFC (60223005, 60033010), the National Grant Fundamental Research of Science and Technology (973 Project: 2002CB312102), and the Research Grant of University of Macau.

References

[1] C.Batty, M.Wiebe, and B.Houston. High Performance Production-Quality Fluid Simulation via NVIDIA's QuadroFX. http://film.nvidia.com/docs/CP/4449/frantic_GPUAccelerationofFluids.pdf. 2003

[2] J.Bolz, I.Farmer, E.Grinspun and P. Schröoder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, pp.917-924, July 2003.

[3] Y.Dobashi, K.Kaneda, H.Yamashita, T.Okita, and T. Nishita. A Simple, Efficient Method for Realistic Animation of Clouds. In *Proceedings of SIGGRAPH*, pp.19-20, July 2000.

[4] D.Enright, S.Marschner, and R.Fedkiw, Animation and Rendering of Complex Water Surfaces. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, pp.736-744, July 2002.

[5] R.Fedkiw, J.Stam and H.W.Jensen. Visual Simulation of Smoke. In *Proceedings of SIGGRAPH*, pp.15-22, August 2001.

[6] N.Foster and D.Metaxas. Realistic Animation of Liquids. In *Proceedings of SIGGRAPH*, pp.471-483, August 1996.

[7] N.Foster and R.Fedkiw. Practical Animation of Liquids. In *Proceedings of SIGGRAPH*, pp.23-30, August 2001.

[8] N.Goodnight, C.Woolley, D.Luebke and G.Humphreys. A Multigrid Solver for Boundary Value Problems Using Programmable Graphics Hardware. In *Proceedings of Graphics Hardware*, pp.102-111, July 2003.

[9] GPGPU. http://www.gpgpu.org

[10] M.J.Harris, and A.Lastra. Real-Time Cloud Rendering. *Computer Graphics Forum (Proceedings of Eurographics 2001)*, Blackwell Publishers, pp.76-84, 2001.

[11] M.J.Harris, G.Coombe, T.Scheuermann and A.Lastra. Physically-Based Visual Simulation on Graphics Hardware. In *Proceedings of Graphics Hardware*, pp.109-118, September 2002.

[12] M.J.Harris, W.V.Baxter III, T.Scheuermann and A.Lastra. Simulation of Cloud Dynamics on Graphics Hardware. In *Proceedings of Graphics Hardware*, pp.92-101, July 2003.

[13] M.J.Harris. Real-Time Cloud Simulation and Rendering. PhD thesis, 2003.

[14] M.J.Harris. Flo: A real time fluid float simulator written in Cg, 2003. http://www.markmark.net/gdc2003/

[15] T.Kim and M.C.Lin. Visual Simulation of Ice Crystal Growth. In *Proceedings of SIGGRAPH/Eurographics*

Symposium on Computer Animation, pp.86-97, July 2003.

[16] J.Krüger and R.Westermann. Linear Algebra Operators for GPU Implementation of Numerical Algorithms. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, pp.908-916, July 2003.

[17] J.Krüger and R.Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *IEEE Visualization*, pp.287-292, October 2003.

[18] A.E.Lefohn, J.M.Kniss, C.D.Hansen and R.T.Whitaker. Interactive Deformation and Visualization of Level Set Surfaces Using Graphics Hardware. In *IEEE Visualization*, pp.75-82, October 2003.

[19] A.E.Lefohn, J.M.Kniss, C.D.Hansen, and R.T.Whitaker. A Streaming Narrow-Band Algorithm: Interactive Computation and Visualization of Level Sets. *IEEE Transactions on Visualization and Computer Graphics*. Volume 10, Issue 4, pp.422-433, July 2004.

[20] W.Li, X.Wei, and A.Kaufman. Implementing Lattice Boltzmann Computation on Graphics Hardware. *The Visual Computer*, Volume 19, pp.444-456. 2003.

[21] W.Li, Z.Fan, X.Wei, and A.Kaufman, GPU-Based Flow Simulation with Complex Boundaries. *Technical Report 031105*, Computer Science Department, SUNY at Stony Brook, Nov 2003.

[22] M.Macedonia. The GPU Enters Computing's Mainstream. *IEEE Computer Society*, Volume 36, Issue 10, pp.106-108, October 2003.

[23] D.Q.Nguyen, R.Fedkiw and H.W.Jensen. Physically Based Modeling and Animation of Fire. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, pp.721-728, July 2002.

[24] nVidia. http://www.nvidia.com/

[25] N.Rasmussen, D.Q.Nguyen，W.Geiger and R.Fedkiw. Smoke Simulation For Large Scale Phenomena. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, pp.703-707, July 2003.

[26] M.Rumpf and R.Strzodka. Using graphics cards for quantized FEM computations. In *Proceedings of VIIP*, pp.98–107,2001.

[27] J.Stam and E.Fiume. Turbulent Wind Fields for Gaseous Phenomena. In *Proceedings of SIGGRAPH*, pp.369-376, September 1993.

[28] J.Stam. Stable Fluids. In *Proceedings of SIGGRAPH*, pp.121-128, July 1999.

[29] J.Stam. Flows on Surfaces of Arbitrary Topology. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, pp.703-707, July 2003.

[30] R.Westermann and T.Ertl. Efficiently Using Graphics Hardware in Volume Rendering Applications. In *Proceedings of SIGGRAPH*, pages169-177, September 1998.

[31] E.Wu, Y.Liu, and X.Liu. An Improved Study of Real-Time Fluid Simulation on GPU. *Computer Animation and Virtual World (Computer Animation and Social Agents)*. Volume 15. July 2004.

[32] Y.Xu, Y.Chen, S.Liu, H.Zhong, E.Wu, B.Guo, and H.Shum. Photorealistic Rendering of Knitwear Using the Lumislice. In *Proceedings of SIGGRAPH*, pp.391-398, September 2001.

[33] S.Yoshida and T.Nishita. Modeling of Smoke Flow Taking Obstacles into Account. In *Proceedings of Pacific Graphics*, pp.135-145,October 2000.