Real-Time Simulation of 3D Smoke on GPU

QING YANG
Computer Science and Information Technology College
Zhejiang Wanli University
No.8,South Qian Hu Road Ningbo, Zhejiang
P.R.CHINA
http://www.computer.zwu.edu.cn

Abstract: - In this paper, we investigate the fluid simulation on GPU, and implement real time smoke animation based on GPU by numerically solving Navier-Stokes equation. Based on Navier-Stokes equation, we discuss the details of the method, "Stable Fluids". This scheme makes implementation on the GPU simple, because there is a straightforward mapping between grid cells and voxels in a 3D texture. After optimizing, this method not only fit to GPU, but also is steady and realistic greatly.

Key-Words: - GPU, 3D, smoke, real-time, simulation

1 Introduction

Fluid simulation has been a focus of computer graphics research. Fluid in nature, such as smoke, water and fog, appears to be simple, but in reality to hide extremely complex rules. Early, fluid simulation is based on the parameters modeling, such as the simulation of wave from [1]. However, this method can only be side the movement in the vicinity of particles' initial position, and the lack of boundary condition judge enables the simulation of the effect is very natural. At the same time, it is difficult to control this type of model, and to simulate some of the more complex and the more detailed of fluid phenomenon. The method of parameters modeling is limited to the ability of the computer hardware at that time. With the rapid development of computer hardware, the simulation of fluid is often based on physical model complexly.

In this paper, a method is discussed that is used to realize real-time simulation of smoke, and based on "Stable Fluids" method of Stam 1999 [2]. However, while Stam's simulations used a CPU implementation, we choose to implement ours on GPU (Graphic Process Unit） because GPUs are well suited to the type of computations required by fluid simulation. The simulation we describe is performed on a grid of cells. Programmable GPUs are optimized for performing computations on pixels, which we can consider to be a grid of cells. GPUs achieve high performance through parallelism: they are capable of processing multiple vertices and pixels simultaneously.

2 Simulation

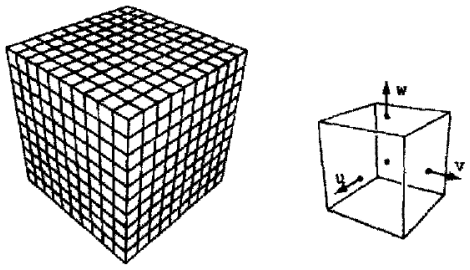
2.1 Mathematical Background

To simulate the behavior of smoke, we must have a mathematical representation of the state of the kind of fluid at any given time. The most important quantity to represent is the velocity of the fluid, because velocity determines how the fluid moves itself and the things that are in it. The fluid's velocity varies in both time and space, so we represent it as a vector field. We use consistent mathematical notation throughout the paper. In equations, *italics* are used for variables that represent scalar quantities, such as pressure, p. **Boldface** is used to represent vector quantities, such as velocity, $\mathbf{u}$.

A vector field is a mapping of a vector-valued function onto a parameterized space, such as a Cartesian grid. The velocity vector field of our fluid is defined such that for every position $\mathbf{x} = (x, y, z)$, there is an associated velocity at time t, $\mathbf{u}(\mathbf{x}, t) = (u(\mathbf{x}, t), v(\mathbf{x}, t), w(\mathbf{x}, t))$, as shown in Fig.1.

Fig.1: The Fluid Velocity Grid [3]

In physics, it's common to make simplifying



assumptions when modeling complex phenomena. Fluid simulation is no exception. We assume an incompressible, homogeneous fluid. A fluid is incompressible if the volume of any subregion of the fluid is constant over time. A fluid is homogeneous if its density, ρ, is constant in space. The combination of incompressibility and homogeneity means that

density is constant in both time and space. These assumptions are common in fluid dynamics, and they do not decrease the applicability of the resulting mathematics to the simulation of real fluids, such as water and air [4].

We simulate fluid dynamics on a regular Cartesian grid with spatial coordinates **x** = (*x*, *y*, *z*) and time variable *t*. The fluid is represented by its velocity field **u**(**x**, *t*), as described earlier, and a scalar pressure field *p*(**x**, *t*). These fields vary in both space and time. If the velocity and pressure are known for the initial time $t = 0$, then the state of the fluid over time can be described by the Navier-Stokes equations for incompressible flow:

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \mathbf{f} ,$$

subject to the incompressibility constraint:

$$\nabla \cdot \boldsymbol{u} = 0 ,$$

where p is the pressure, ρ is the mass density, **f** represents any external forces (such as gravity), and ∇ is the differential operator:

$$\left[\frac{\partial}{\partial x}\frac{\partial}{\partial y}\frac{\partial}{\partial z}\right]^T .$$

To define the equations of motion in a particular context, it is also necessary to specify boundary conditions (that is, how the fluid behaves near solid obstacles or other fluids).

The basic task of a fluid solver is to compute a numerical approximation of **u**. This velocity field can then be used to animate visual phenomena such as smoke particles.

2.2 Solving for Velocity

The popular "stable fluids" method for computing velocity was introduced in [2], and a GPU implementation of this method for 2D fluids was presented in [5].In this section we briefly describe how to solve for velocity but refer the reader to the cited works for details.

In order to numerically solve the momentum equation, we must discretize our domain (that is, the region of space through which the fluid flows) into computational elements. We choose an Eulerian discretization, meaning that computational elements are fixed in space throughout the simulation—only the values stored on these elements change. In particular, we subdivide a rectilinear volume into a regular grid of cubical cells. Each grid cell stores both scalar quantities (such as pressure, temperature, and so on) and vector quantities (such as velocity). This scheme makes implementation on the GPU simple, because there is a straightforward mapping between grid cells and voxels in a 3D texture. Lagrangian schemes (that is, schemes where the computational elements are not fixed in space) such as smoothed-particle hydrodynamics [6] are also popular for fluid animation, but their irregular structure makes them difficult to implement efficiently on the GPU.

Because we discretize space, we must also discretize derivatives in our equations: finite differences numerically approximate derivatives by taking linear combinations of values defined on the grid. As in [5], we store all quantities at cell centers for pedagogical simplicity, though a staggered MAC-style grid yields more-robust finite differences and can make it easier to define boundary conditions (See [7] for details) .

In a GPU implementation, cell attributes (velocity, pressure, and so on) are stored in several 3D textures. At each simulation step, we update these values by running computational kernels over the grid. A kernel is implemented as a pixel shader that executes on every cell in the grid and writes the results to an output texture. However, because GPUs are designed to render into 2D buffers, we must run kernels once for each slice of a 3D volume.

To execute a kernel on a particular grid slice, we rasterize a single quad whose dimensions equal the width and height of the volume. In Direct3D 10 we can directly render into a 3D texture by specifying one of its slices as a render target. Placing the slice index in a variable bound to the SV_RenderTargetArrayIndex semantic specifies the slice to which a primitive coming out of the geometry shader is rasterized [8]. By iterating over slice indices, we can execute a kernel over the entire grid.

Rather than solve the momentum equation all at once, we split it into a set of simpler operations that can be computed in succession: advection, application of external forces, and pressure projection. Implementation of the corresponding kernels is detailed in [5].

2.3 Improving Details

The semi-Lagrangian advection scheme used by Stam is useful for animation because it is unconditionally stable, meaning that large time steps will not cause the simulation to "blow up." However, it can introduce unwanted numerical smoothing, causing smoke to lose detail. To achieve higher-order accuracy, we use a MacCormack scheme that performs two intermediate semi-Lagrangian advection steps. Given a quantity Φ and an advection scheme *A*, higher-order accuracy is obtained using the following sequence of operations [9]:

$$\hat{\phi}^{n+1} = A(\phi^n)$$
$$\hat{\phi}^n = A^R(\hat{\phi}^{n+1}) \quad .$$
$$\phi^{n+1} = \hat{\phi}^{n+1} + \frac{1}{2}(\phi^n - \hat{\phi}^n)$$

Here, ϕ^n is the quantity to be advected, $\hat{\phi}^{n+1}$ and $\hat{\phi}^n$ are intermediate quantities, and ϕ^{n+1} is the final advected quantity. The superscript on A^R indicates that advection is reversed (that is, time is run backward) for that step.

Unlike the standard semi-Lagrangian scheme, this MacCormack scheme is not unconditionally stable. Therefore, a limiter is applied to the resulting value ϕ^{n+1}, ensuring that it falls within the range of values contributing to the initial semi-Lagrangian advection. In the GPU solver, this means we must locate the eight nodes closest to the sample point, access the corresponding texels exactly at their centers (to avoid getting interpolated values), and clamp the final value to fall within the minimum and maximum values found on these nodes. As shown in Fig.2, the result of the advection (black) is clamped to the range of values from nodes (white) used to get the interpolated value at the advected "particle" (the cross) in the initial semi-Lagrangian step.

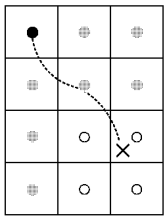


Fig.2: Limiter Applied to a MacCormack Advection Scheme

Once the intermediate semi-Lagrangian steps have been computed, the pixel shader completes advection using the MacCormack scheme. On the GPU, higher-order schemes are often a better way to get improved visual detail than simply increasing the grid resolution, because math is cheap compared to bandwidth [10].

Although the velocity field describes the fluid's motion, it does not look much like a smoke when visualized directly. To get interesting visual effects, we must keep track of additional quantities that are pushed around by the smoke. For instance, we can keep track of density and temperature to obtain the appearance of smoke [11]. For each additional quantity Φ, we must allocate an additional texture with the same dimensions as our grid. The evolution of values in this texture is governed by the same advection equation used for velocity:

$$\frac{\partial \phi}{\partial t} = -(\boldsymbol{u} \cdot \nabla)\phi \quad .$$

To get a more physically plausible appearance, we must make sure that hot smoke rises and cool smoke falls. To do so, we need to keep track of the fluid temperature T (which again is advected by **u**). Temperature values have an influence on the dynamics of the smoke. This influence is described by the buoyant force:

$$\mathrm{f}_{\mathrm{buoyancy}} = \frac{Pmg}{R}\left(\frac{1}{T_0} - \frac{1}{T}\right)\boldsymbol{z},$$

where P is pressure, m is the molar mass of the gas, g is the acceleration due to gravity, and R is the universal gas constant. In practice, all of these physical constants can be treated as a single value and can be tweaked to achieve the desired visual appearance. The value T_0 is the ambient or "room" temperature, and T represents the temperature values being advected through the flow. **z** is the normalized upward-direction vector. The buoyant force should be thought of as an "external" force and should be added to the velocity field immediately following velocity advection.

3 Implementation on GPU

Now that we know the problem and the basics of solving it, we can move forward with the implementation. A good place to start is to lay out some pseudocode for the algorithm. The algorithm is the same every time step, so this pseudocode represents a single time step. The variables d, u and p hold the divergence, velocity and pressure field data (See [10] for details of program).

```
// advect the current cell
u=PS_ADVECT_VEL(u);
// Get velocity values from neighboring cells,
// and compute the velocity's divergence
// using central differences.
d = PS_DIVERGENCE(u);
// Compute the new pressure value for the center cell
// after Getting the divergence at the current cell
// and the pressure values from neighboring cells.
p = PS_JACOBI(p, d);
// Compute the gradient of pressure at the current cell
// by taking central differences of
// neighboring pressure values,
// and project the velocity onto its divergence-free
// component by subtracting the gradient of pressure.
u = PS_PROJECT(p, u);
```

This pseudocode contains no implementation-specific details. We perform all the steps on the GPU.

Textures on current GPUs support all the basic operations necessary to implement a fluid simulation. Because textures usually have three or four color channels, they provide a natural data structure for vector data types with two to four components. Alternatively, multiple scalar fields can be stored in a single texture. We need at least three textures to represent the state of the fluid: one for velocity, one for pressure, and another for divergence.

GPUs do not have the capability to perform nested loop over each texel in a texture. However, the fragment pipeline is designed to perform identical computations at each fragment. To the programmer, it appears as if there is a processor for each fragment, and that all fragments are updated simultaneously. In the parlance of parallel programming, this model is known as single instruction, multiple data (SIMD) computation. Thus, the GPU analog of computation inside nested loops over an array is a fragment program applied in SIMD fashion to each fragment.

On the GPU, the output of fragment processors is always written to the frame buffer. Think of the frame buffer as a two-dimensional array that cannot be directly read. There are two ways to get the contents of the frame buffer into a texture that can be read:

1. Copy to texture (CTT) copies from the frame buffer to a texture.

2. Render to texture (RTT) uses a texture as the frame buffer so the GPU can write directly to it.

CTT and RTT function equally well, but have a performance trade-off. For the sake of generality we do not assume the use of either and refer to the process of writing to a texture as a texture update.

Earlier we mentioned that, in practice, each of the four steps in the algorithm updates a temporary grid and then performs a swap. RTT requires the use of two textures to implement feedback, because the results of rendering to a texture while it is bound for reading are undefined. The swap in this case is merely a swap of texture IDs. The performance cost of RTT is therefore constant. CTT, on the other hand, requires only one texture. The frame buffer acts as a temporary grid, and a swap is performed by copying the data from the frame buffer to the texture. The performance cost of this copy is proportional to the texture size[5].

4 Conclusion

This application was run on an Intel(R) Core(TM) 2 Duo CPU E4500 with 2.20GHz and tested on Windows Vista Ultimate 32 bit OS with NVIDIA GeForce 8500GT. The HLSL compiler and Microsoft Visual Studio 2005 was used to generate the 3D smoke, as show as Fig.3.

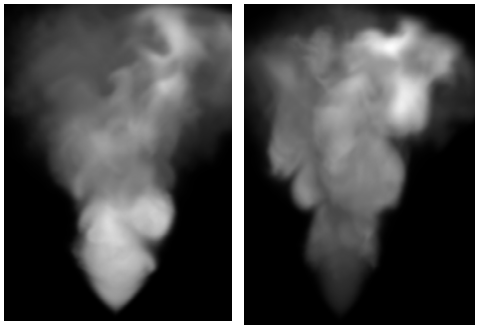


Fig.3: The experimental results

The power and programmability now available in GPUs enables fast simulation of a wide variety of phenomena. Underlying many of these phenomena is the dynamics of fluid motion. From these ideas you can experiment with your own simulation concepts and incorporate fluid simulation into graphics applications.

References:

[1] Alain Fournier, William T.Reeves, A Simple Model of Ocean Waves, *Compute Graphics*, Vol.20, No.4, 1986, pp.75-84.

[2] Stam, Stable Fluids, *Proceedings of SIGGRAPH*, Vol.99, 1999, pp.121-128.

[3] Fedkiw, R., J. Stam, and H. W. Jensen, Visual Simulation of Smoke, *Proceedings of SIGGRAPH 2001*, pp.15–22.

[4] LIU Daozhi, *Computational Fluid Dynamics Basic*, Beijing University of Aeronautics & Astronautics Press, 1989.

[5] Harris M. , Fast fluid dynamics simulation on the GPU.*GPU Gems*, Mar.2004, pp.637-665.

[6] Müller, Matthias, David Charypar, and Markus Gross, Particle-Based Fluid Simulation for Interactive Applications, *Proceedings of the 2003 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, 2003, pp. 154-159.

[7] Harlow, F., and J. Welch, Numerical Calculation of Time-Dependent Viscous Incompressible Flow of Fluid with Free Surface, *Physics of Fluids 8*, 1965, pp. 2182–2189.

[8] Blythe, David, The Direct3D 10 System, In ACM Transactions on Graphics (Proceedings of SIGGRAPH 2006) 25(3), pp. 724–734.

[9] Selle, A., R. Fedkiw, B. Kim, Y. Liu, and J. Rossignac, An Unconditionally Stable

MacCormack Method, Journal of Scientific Computing (in review), 2007, Available online at http://graphics.stanford.edu/~fedkiw/papers/stanford2006-09.pdf.

[10] Keenan Crane, Ignacio Llamas, and Sarah Tariq, Real-Time Simulation and Rendering of 3D Fluids, *GPU Gems 3*, 2007, edited by Hubert Nguyen, Manager of Developer Education at NVIDIA, Available online at http://developer.nvidia.com/object/gpu-gems-3.html.

[11] Fedkiw, R., J. Stam, and H. W. Jensen, Visual Simulation of Smoke, Proceedings of SIGGRAPH 2001, pp. 15–22.