The Visual Computer

Implementing lattice Boltzmann computation on graphics hardware

Wei Li, Xiaoming Wei, Arie Kaufman

Center for Visual Computing (CVC) and Department of Computer Science Stony Brook University Stony Brook, NY 11794-4400, USA
E-mail: {liwei,wxiaomin,ari}@cs.sunysb.edu

Published online: 15 July 2003
© Springer-Verlag 2003

The Lattice Boltzmann Model (LBM) is a physically-based approach that simulates the microscopic movement of fluid particles by simple, identical, and local rules. We accelerate the computation of the LBM on general-purpose graphics hardware, by grouping particle packets into 2D textures and mapping the Boltzmann equations completely to the rasterization and frame buffer operations. We apply stitching and packing to further improve the performance. In addition, we propose techniques, namely range scaling and range separation, that systematically transform variables into the range required by the graphics hardware and thus prevent overflow. Our approach can be extended to acceleration of the computation of any cellular automata model.

Key words: Graphics hardware – Lattice Boltzmann method – Flow simulation

1 Introduction

Simulations of fluid behavior are in great demand in film making, as well as in visual simulations such as animation design, texture synthesis, flight simulations, and scientific visualization. In Computational Fluid Dynamics (CFD), fluid properties, such as density and velocity, are typically described by the Navier–Stokes (NS) equations, which have nonlinear terms that make them too expensive to solve numerically in real time. Instead of calculating the macroscopic equations, we can simulate the linear microscopic LBM [2, 11, 13] to satisfy the NS equations. The fluid flow consists of many tiny flow particles and the collective behavior of these microscopic particles results in the macroscopic dynamics of a fluid. Since the macroscopic dynamics of a fluid is insensitive to the underlying details in microscopic physics [1], this gives us the possibility of using a simplified microscopic kinetic-type model to simulate its movements.
The LBM can be understood as a Cellular Automata representing discrete packets moving on a discrete lattice at discrete time steps. The calculation is performed on a regular grid. At each grid cell, there are variables indicating the status of that grid point. All of the cells modify their status at each time step based on linear and local rules. Although faster than other solutions to the NS equations, the computation of the LBM is still slow. The calculations at each point are simple, but there are usually a large number of cells. Therefore, a practical use of the LBM typically demands parallel supercomputers [11, 13].
Commodity graphics hardware can perform pixel-oriented operations very efficiently. Not only are the operations pipelined in dedicated hardware, but there are usually up to four color channels and multiple pixel pipelines that essentially provide parallel processing. The speed of graphics hardware doubles approximately every six months, which is much faster than the rate of CPU improvements. Inspired by the performance of graphics hardware and the resemblance of the computation patterns in the LBM and the rasterization stage, we propose to accelerate the LBM on commodity graphics hardware by storing packets of the LBM as textures and translating the Boltzmann equations into rendering operations of the texture units and the frame buffer. We further apply stitching of sub-textures to reduce the overhead of texture switching and packing into color channels to reduce the memory requirement as well as to exploit parallelism of the four color channels. In addition, we present range scaling that transforms

the range of arbitrary variables and any intermediate results to fulfill the requirements of the graphics hardware. These techniques guarantee no overflow no matter how the variables are evaluated, while the scaling factors are chosen to take the full precision of the hardware. We use LBM equations as an example to show how the scaling is applied and observe errors of up to only 1%. In addition, the range separation proposed in this paper overcomes the non-negative limits required in certain stages of the graphics pipeline.

Although we focus on the LBM and its applications to visual simulations of fluids and smoke, our approach can be extended to any cellular-automata-type calculation. We expect that the techniques presented in the paper, such as range scaling and range separation, will be developed into a compiler that automatically generates rendering instructions equivalent to various general computations, with the range of all variables and intermediate and final results properly transformed.

The rest of the paper is organized as follows. First, we review related work. In Sect. 3, we present range scaling and range separation, which are critical components of our approach in accelerating general computations on graphics hardware. In Sects. 4 and 5, after a brief introduction of the theory of the LBM, we present our methods of mapping the LBM equations as multi-pass rasterization operations and how the range scaling is applied to the LBM. In Sect. 6, we give the experimental results.

2 Related work

Graphics hardware has been extended to various applications beyond its originally expected usage. Examples include matrix multiplication [12], 3D convolution [8], morphological operations such as dilation and erosion [9], computation of Voronoi diagrams and proximity queries [6, 7], voxelization [3], algebraic reconstruction [14], and volumetric deformation [18].

A graphics architecture, such as OpenGL, can be treated as a general SIMD computer [15]. Various computations are implemented as the operations of the texture mapping unit and the frame buffer. Final results are obtained in one or more rendering passes. Both Peercy et al. [15] and Proudfoot et al. [16] have developed languages for programmable procedural shading systems, as well as compilers that automatically generate instructions corresponding to rendering operations on graphics hardware. However, to apply these ideas to other applications, the limited value range and accuracy of graphics hardware have to be considered. Trendall et al. [19] have given several formulas for scaled and biased functions whose value ranges are within the limits, and have applied the method to the computation of interactive caustics.

Some of the above applications scale and shift the variables in their computations so that they fit into the value range of the graphics hardware. Their scaling and shifting parameters are chosen either trivially or empirically. The range scaling proposed in this paper provides a systematic way for mapping general computations onto graphics hardware which guarantees that all the inputs and outputs as well as the intermediate results are not clamped by the hardware, in addition to exploiting the hardware precision as much as possible.

There are a few papers on accelerating flow visualization on graphics hardware. Heidrich et al. [5] exploit pixel texture to compute line integral convolution, which is a technique for visualizing vector data. Jobard et al. [10] translate texture advection computations into frame buffer operations to accelerate the visualization of the motion of 2D flows. Weiskopf et al. [21] extend Jobard et al.'s work to 3D flows. They also take advantage of the newly available OpenGL extensions, namely offset texture and dependent texture. All of these techniques are for the visualization of fluid with given velocity fields. In contrast, this paper focuses on the simulation, specifically the generation of the fields such as velocity that are required for the visualization. By employing similar techniques, our approach can map the whole simulation and visualization onto the graphics hardware without the need of transferring data back and force between the host memory and the graphics memory. Harris et al. [4] implement coupled map lattice (CML), a variation of cellular automata, on graphics hardware, which has similar motivation and applications to this paper. We expect our work on LBM to eventually result in a compiler for general computations on graphics hardware. Therefore one of the focuses of the paper is how to systematically map general equations to rendering operations, considering the range and precision limitations of the graphics hardware. We also propose techniques, such as texture packing and stitching, to further accelerate the execution on the graphics hardware.

3 General computation on graphics hardware

We use the rasterization units (e.g., Nvidia's register combiners) and the frame buffer in graphics hardware to implement addition, subtraction, and multiplication. Division and other more complicated calculations are replaced by lookup tables storing precomputed values. Input values are stored in textures and output values are either copied from the frame buffer to textures or written directly to the textures with the render-to-texture extension.

Values in graphics hardware are clamped to either $[0, 1]$ or $[-1, 1]$, depending on the stage of the graphics pipeline. Therefore, we need to transform the value ranges of all of the inputs and outputs. Trendall et al. [19] also mapped the lower bound of the range to 0 by biasing for better accuracy. In contrast, we generally avoid introducing any bias during the mapping. (The reason is explained in Sect. 3.2.) That is, we only apply scaling to change the numerical ranges.

Range scaling can be considered as a simulation of floating point on fixed-point hardware, whereas floating point texture and frame buffer have been suggested for future hardware [17]. However, even if the support for floating point were available in the rasterization stage of the graphics hardware, it would be significantly slower and could take more texture memory than its fixed-point counterpart. The range scaling proposed in this paper makes no assumption of the precision of the hardware.

We always prefer the rasterization units because they are more flexible than the frame buffer. However, distributing certain operations to the frame buffer reduces the number of rendering passes and the need of copying the frame buffer contents into textures.

3.1 Range scaling

The scaling factors should be carefully selected so that no clamping error occurs and the computation exploits the full precision of the hardware. For any input or output function $f(x)$, we divide it by its maximal absolute value and obtain a scaled function $\widetilde{f(x)}$, such that

$$f^{max}\widetilde{f(x)} = f(x)\,, \tag{1}$$

where $f^{max} = \max_x(|f(x)|)$, which we refer to as the left-hand scalar. Obviously, $\widetilde{f(x)} \in [-1, 1]$. We then use $\widetilde{f(x)}$ throughout the hardware pipeline.

We should also make sure that during the computation of $\widetilde{f(x)}$, no intermediate result is clamped. We denote by $U(f)$ the maximal absolute value of all intermediate results during the evaluation of $f(x)$, no matter what computation order is taken when the computation contains multiple operations. It is easy to see that $U(f) \geq f^{max}$. If we multiply by $1/U(f)$ on the right-hand side before computation, we guarantee that no overflow occurs. We refer to $U(f)$ as the right-hand scalar.

If a function $s(x)$ is a weighted sum of several other functions, $s(x) = \sum_i k_i f_i(x)$, we compute the scaled function $\widetilde{s(x)}$ as follows:

$$\widetilde{s(x)} = \frac{U(s)}{s^{max}} \sum_i \frac{k_i f_i^{max}}{U(s)} \widetilde{f_i(x)}\,. \tag{2}$$

The left-hand scalar and the upper bound of the intermediate values are computed as

$$s^{max} = \max\Big(\sum_i k_i f_i^{max}(\exists x,\, f_i(x) > 0),\\ \sum_i k_i f_i^{max}(\exists x,\, f_i(x) < 0)\Big)\,, \tag{3}$$

$$U(s) = \max\Big(\sum_i k_i U(f_i)(\exists x,\, f_i(x) > 0),\\ \sum_i k_i U(f_i)(\exists x,\, f_i(x) < 0)\Big)\,. \tag{4}$$

That is, each $\widetilde{f_i(x)}$ is scaled by $k_i f_i^{max}/U(s)$ before summation. We know that $|k_i f_i^{max}/U(s)| \leq 1$. The sum is then multiplied by a constant $U(s)/s^{max}$. Note that $U(s)/s^{max} \geq 1$. To multiply by a factor larger than 1, we have two choices: (1) utilizing the output scale mapping of the register combiners; (2) applying multiplication and addition or using the dot product. s^{max} is obtained by summing the maximal values of positive and negative entries separately and selecting the sum with the larger absolute value. $U(s)$ is computed similarly. Since $s(x)$ may not reach the computed maximal value s^{max}, it is better to replace s^{max} by the actual largest absolute value $\widetilde{s^{max}}$ such that $\widetilde{s^{max}} < s^{max}$. To measure $\widetilde{s^{max}}$, we implement a software-only version of the same calculations on the CPU and feed in various input values.

If $p(x)$ is the product of several functions, $p(x) = k\sum_i f_i(x)$, we have

$$\widetilde{p(x)} = \frac{U(p)}{p^{max}} k \prod_i \frac{f_i^{max}}{U(f_i)} \widetilde{f_i(x)}\,, \tag{5}$$

where

$$p^{max} = k \prod_i f_i^{max}\,, \tag{6}$$

$$U(p) = k \prod_i U(f_i)\,. \tag{7}$$

If the function $g(x)$ is an input for a rendering pass, we speculate that $U(g) = g^{max}$. Apparently, if all $f_i(x)$ are inputs, $p^{max} = U(p) = k\prod_i f_i^{max}$.
In each rendering pass, the hardware performs a mix of additions (subtractions) and multiplications[1]. We first obtain the absolute maximal values of inputs either by prior knowledge or measurement of software simulation on the CPU. Then, before evaluating a function $f(x)$, we compute its left-hand and right-hand scalars by grouping its right-hand side into sums and products and by recursively applying (3), (4), (6), and (7). Next, we divide the right-hand side by $U(f)$ and distribute $U(f)$ to each input function according to (2) and (5). All the scale coefficients of the inputs are computed in the software.

3.2 Range separation

Conventional graphics hardware only supports a numerical range of [0,1]. Recent OpenGL extensions, such as Nvidia's register combiners, expand the range to $[-1, 1]$ in the rasterization stage, whereas the final combiner and the frame buffer are still limited to [0, 1]. Hence, negative values have to be transformed before they are sent to the final combiner or the frame buffer. A solution is to apply bias and scaling to transform the range to [0, 1] before entering the final combiner stage and the frame buffer. However, it is then very difficult to utilize the final combiner or the frame buffer for multiplication or addition on the biased variables. Consequently, the number of rendering passes and the times of backing-up the contents of the frame buffer are likely to increase.

[1] Other computations are implemented with lookup tables, and hence are treated as inputs.

We propose to separate the positive and negative ranges and avoid biasing as an alternative solution. Similar to other approaches, we scale all the variables to fit into the range $[-1, 1]$ as the first step. If all the values of a variable are constantly non-negative or non-positive, it is trivial to map them to [0, 1].
To handle a variable containing both positive and negative values, we divide the range $[-1, 1]$ into two parts, $[-1, 0]$ and $[0, 1]$. For an arbitrary function f, we have

$$f = [f]^+ + [f]^- = [f]^+ - [-f]^+\,, \tag{8}$$

where $[\ ]^+$ and $[\ ]^-$ denote the clamping to [0, 1] and $[-1, 0]$, respectively. Obviously, both $[f]^+$ and $[-f]^+$ contain only 0 or positive values.
Let f and g be two functions. Addition (subtraction), multiplication (scalar and dot product) and division are then performed as follows:

$$f + g = ([f]^+ + [g]^+) - ([-f]^+ + [-g]^+)\,, \tag{9}$$

$$fg = [f]^+[g]^+ + [-f]^+[-g]^+ - ([f]^+[-g]^+ + [-f]^+[g]^+)\,, \tag{10}$$

$$\frac{f}{g} = \frac{fg}{g^2}\,. \tag{11}$$

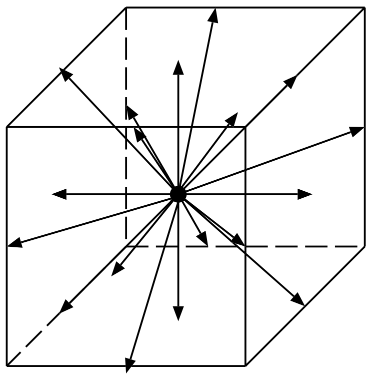
Note that in (11), g^2 can be computed as

$$g^2 = ([g]^+)^2 + ([-g]^+)^2\,. \tag{12}$$

Range separation introduces more calculations than the unseparated version, but recall that scaling and biasing require additional operations as well. In practice, we choose either biasing and scaling or range separation depending on which can be executed in fewer rendering passes and involves fewer texture units. We make a choice considering also the fact that range separation provides one additional bit of precision.

4 Lattice Boltzmann method

We review below the principles of the lattice Boltzmann method (LBM) [2, 11, 13] and see what computations are needed for the simulation of the LBM. The LBM consists of a regular grid and a set of packet distribution values. Each packet distribution f_{qi} corresponds to a velocity direction vector $\boldsymbol{e_{qi}}$ shooting from a node to its neighbor. The index qi



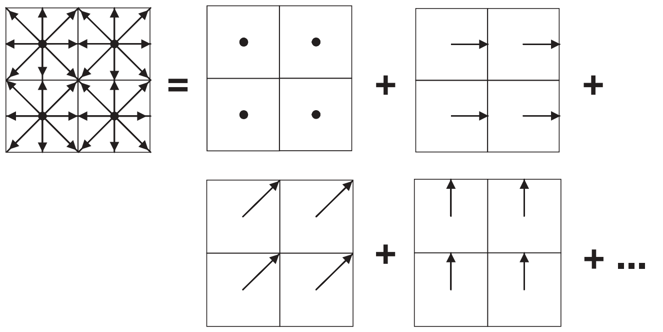


Fig. 1. The D3Q19 lattice geometry. The velocity directions of the 18 moving packet distribution are shown as arrows

Fig. 2. Division of the D2Q9 model. Only 4 grids out of 9 are shown. Packets are grouped according to their velocity directions

describes the D-dimensional sub-lattice, where q is the sub-lattice level and i enumerates the sub-lattice vectors. Figure 1 depicts a single node of the D3Q19 model (19 packets in 3D space), while the left part of Fig. 2 shows four grid nodes of the D2Q9 (9 packets in 2D space). The arrows in the figures represent the $\boldsymbol{e_{qi}}$ vectors.

The LBM updates the packet distribution values at each node based on two rules: collision and propagation. Collision describes the redistribution of packets at each local node. Propagation means the packets move to the nearest neighbor along the velocity directions. These two rules can be described by the following equations:

$$\text{collision}: f_{qi}^{new}(\boldsymbol{x}, t) - f_{qi}(\boldsymbol{x}, t) = \Omega_{qi}\,, \tag{13}$$

$$\text{propagation}: f_{qi}(\boldsymbol{x} + \boldsymbol{e_{qi}}, t+1) = f_{qi}^{new}(\boldsymbol{x}, t)\,, \tag{14}$$

where Ω is a general collision operator. Since components of $\boldsymbol{e_{qi}}$ can only be chosen from {-1, 0, 1}, the propagation is local.

The density and velocity are calculated from the packet distributions as follows:

$$\rho = \sum_{qi} f_{qi}\,, \tag{15}$$

$$\boldsymbol{v} = \frac{1}{\rho} \sum_{qi} f_{qi} \boldsymbol{e_{qi}}\,. \tag{16}$$

The collision operator is selected in such a way that mass and momentum are conserved locally. Suppose that there is always a local equilibrium particle distribution f_{qi}^{eq} dependent only on the conserved quantities ρ and $\boldsymbol{v}$. Then the collision step is changed to

$$f_{qi}^{new}(\boldsymbol{x}, t) - f_{qi}(\boldsymbol{x}, t) = -\frac{1}{\tau}(f_{qi}(\boldsymbol{x}, t) - f_{qi}^{eq}(\rho, \boldsymbol{v}))\,, \tag{17}$$

where τ is the relaxation time scale. f_{qi}^{eq} is decided by the following equation:

$$\begin{aligned} f_{qi}^{eq}(\rho, \boldsymbol{v}) = \rho(A_q &+ B_q\langle \boldsymbol{e_{qi}}, \boldsymbol{v}\rangle \\ &+ C_q\langle \boldsymbol{e_{qi}}, \boldsymbol{v}\rangle^2 + D_q\langle \boldsymbol{v}, \boldsymbol{v}\rangle)\,, \end{aligned} \tag{18}$$

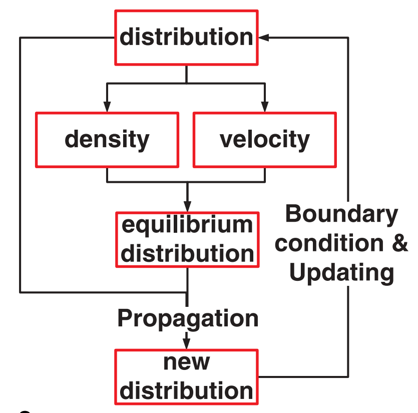
where $\langle \boldsymbol{x}, \boldsymbol{y}\rangle$ denotes the dot product of two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$. The constants A_q to D_q depend on the employed lattice geometry.

The simulation of the LBM then proceeds as follows: (1) compute density by (15); (2) compute velocity by (16); (3) compute equilibrium distribution by (18); (4) update distributions by (17) and go back to step (1). More details on the LBM model can be found in [20].

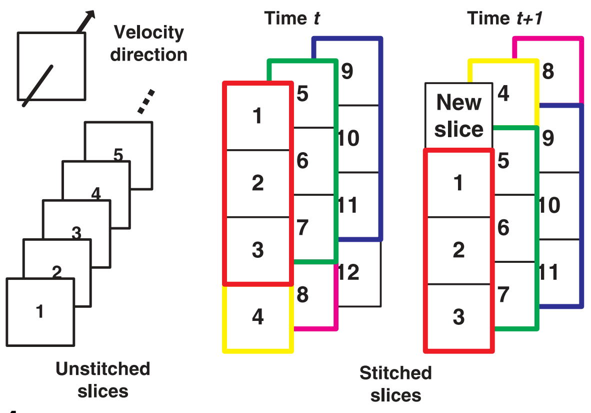
5 Mapping LBM to graphics hardware

5.1 Algorithm overview

To compute the LBM equations on graphics hardware, we divide the LBM grid and group the packet



3



4

Fig. 3. The data flow of the hardware-accelerated LBM computations

Fig. 4. Propagation of the packet distributions along the direction of the velocity component orthogonal to the slices

distributions f_{qi} into arrays according to their velocity directions. All the packet distributions with the same velocity direction are grouped into the same array, while keeping the neighboring relationship of the original model. Figure 2 shows the division of a 2D model. We then store the arrays as 2D textures. For a 2D model, all such arrays are naturally 2D, while for a 3D model, each array forms a volume and is stored as a stack of 2D textures. The idea of the stack of 2D textures is from 2D texture-based volume rendering, but note that we do not need three replicated copies of the dataset.

All the other variables, the density ρ, the velocity $\boldsymbol{v}$, and the equilibrium distributions f_{qi}^{eq} are stored similarly in 2D textures. We project multiple textured rectangles with the color-encoded densities, velocities, and distributions. For convenience, the rectangles are parallel to the viewing plane and are rendered orthogonally. Therefore, the texture space has the same resolution as the image space and the interpolation mode is set to nearest-neighbor.

As shown in Fig. 3, the textures of the packet distributions are the inputs. Density and velocity are then computed from the distribution textures. Next, the equilibrium distribution textures are obtained from the densities and the velocities. According to the propagation equation, new distributions are computed from the distributions and the equilibrium distributions. Finally, we apply the boundary conditions and update the distribution textures. The updated distribution textures are then used as inputs for the next simulation step.

To reduce the overhead of switching between textures, we stitch multiple textures representing packet distributions with the same velocity direction into one larger texture. Figure 4 shows an example, in which every four slices are stitched into a larger texture. The pipeline depicted in Fig. 3 is then operated on the stitched textures.

5.2 Propagation

According to (14), each packet distribution having non-zero velocity propagates to the neighboring grid every time step. Since we group packets based on their velocity directions, the propagation is accomplished by shifting distribution textures in the direction of the associated velocity. We decompose the velocity into two parts, the velocity component within the slice (*in-slice velocity*) and the velocity component orthogonal to the slice (*orthogonal velocity*). The propagation is done for the two velocity components independently. To propagate in the direction of the in-slice velocity, we simply translate the texture of distributions appropriately, as shown in Fig. 5.

If we do not stitch multiple slices into one texture, the propagation in the direction of the orthogonal velocity is done simply by renaming the distribution textures. Because of the stitching, we need to apply translation inside the stitched textures as well as copy sub-textures to other stitched textures. Figure 4 shows the out-of-slice propagation for stitched slices. The indexed blocks denote the slices storing packet distributions. The rectangles in thicker lines mark the sub-textures that are propagated. For example, the sub-texture composed of slices 1 to 3 is shifted down by the size of one slice in the Y dimension. Slices 4 and 8 are moved to the next textures. Note that in time step $t+1$, a new slice is added, owing to the inlet or the boundary condition, while a block (12) has moved out of the framework and is discarded.

5.3 Boundary condition

Packet distributions on the boundary should be handled differently from the internal ones. A general approach is to compute the new distributions for the boundary distributions and then set the new values into the distributions' textures. The computation can be done with either the CPU or the graphics hardware.
Bounce-back boundary conditions can be easily handled by the graphics hardware. Because the particles are grouped according to their velocity directions, we simply copy the boundary packet distributions to the texture of the opposite velocity direction. Similar to propagation, the bounce-back is treated for the in-slice velocity and the orthogonal velocity separately. For a boundary face that is not parallel to the slices, the intersection of the face with a slice is a line segment (or a curve, if we allow a non-planar boundary face). We set the distributions next to the intersection by drawing texture strips which are just one texel wide. Figure 5 shows the bounce-back from the left and the top walls. Note that the distributions leaving one slice become the new distributions of the slice with the opposite velocity direction (see red arrow in Fig. 5. For a boundary face parallel to the slices, usually a 2D texture needs to be updated. In Fig. 4, the block marked "new slice" is obtained from the slice at the same position but with an opposite direction, or it is set to the inlet distributions if the slice is adjacent to an inlet face.

5.4 Packing

In our preliminary work [20] of the hardware-accelerated LBM, we stored only the distributions of the same direction in a single texture. Due to

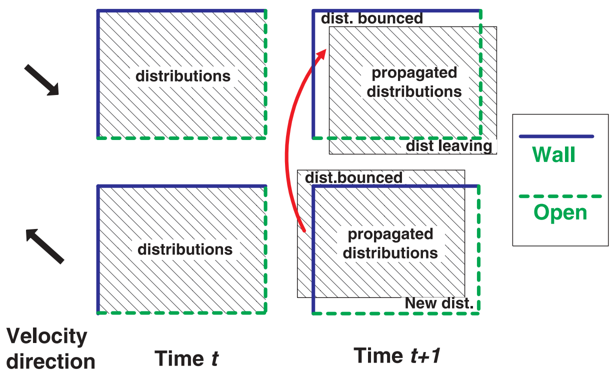
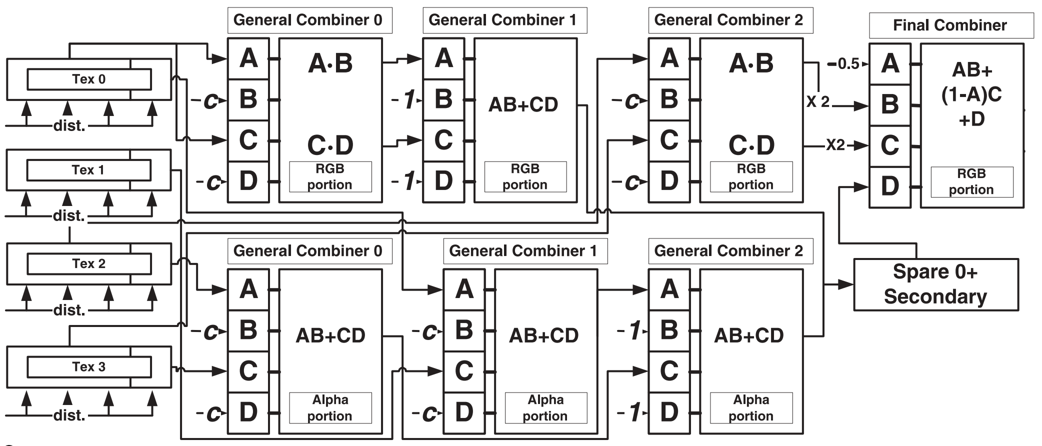
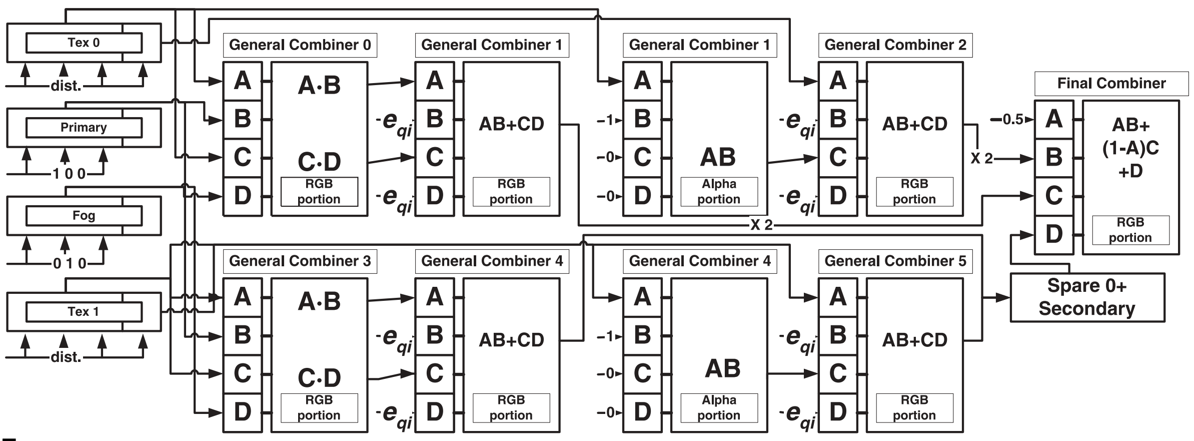


Fig. 5. Propagation of the packet distributions along the direction of the in-slice velocity and application of boundary condition. The rectangular edges in thick blue lines are the boundaries

the restriction of the current graphics hardware and considerations of efficiency, every distribution texture is in the format of RGBA. Hence, each f_{qi} is replicated 4 times into the RGBA channels, and the operations over the distributions are duplicated as well.
In this paper, we pack four f_{qi}s from different directions as an RGBA texel. That is, a single texture is composed of four distribution arrays with different velocity directions. This packing scheme reduces the memory requirement of distributions to nearly 1/4 of the design without packing.
To compute density ρ (refer to (15)), we use a dot-product to add the distributions stored in different color channels, as shown in Fig. 6. The packing essentially reduces the number of operations to one quarter for the computation of density. Multiple-distribution textures are added together with the OpenGL extensions of multi-textures and the register combiners. In addition, we also utilize the additive blending of the frame buffer to make it unnecessary to backup the intermediate contents by copying the frame buffer to a texture or switching to different frame (pixel) buffer.
The calculation of velocities is a little more complicated owing to the packing, since each distribution needs to be multiplied by its own direction vector $\boldsymbol{e}_{qi}$ and the RGB components cannot be read completely individually in the current implementation of the register combiners. Therefore, we need to dot-product the distributions in the RGB channels with (1, 0, 0) or (0, 1, 0) to separate the distributions. The value in the blue channel is extracted with an alpha



6



7

Fig. 6. The configuration of the register combiners for computing density from packed distributions. c denotes constants dependent on qi and the scaling factors

Fig. 7. The configuration of the register combiners for computing velocity from packed distributions

combiner. As shown in Fig. 7, we add eight distribution slices (stored in two textures) weighted by the corresponding $\boldsymbol{e}_{qi}$ with six combiner stages. Here we apply a trick on the final combiner stage so that it adds four inputs. Note that the inputs B and C to the final combiner stage are scaled by 2 and A is set to 0.5, while the other two inputs of the final combiner stage are sent to Spare0 and Secondary. Again, we use additive blending of the frame buffer to add consecutive outputs from the final combiner to avoid copying the frame buffer. Range separation is applied to $\boldsymbol{e}_{qi}$ so that negative values are not clamped. That is, we compute $\boldsymbol{v}^+$ and $\boldsymbol{v}^-$ separately and later add them together.

5.5 Scaling of the LBM equations

In this section, we show how to apply the range transformation described in Sect. 3 to the LBM equations. Assume f_q^{max} is the left-hand scalar of the packet distributions and the equilibrium packet distributions of sub-lattice q. We define the scaled distributions $\widetilde{f_{qi}}$ and the scaled density $\widetilde{\rho}$ as

$$\widetilde{f_{qi}} = \frac{1}{f_q^{max}} f_{qi} \,, \tag{19}$$

$$\widetilde{\rho} = \frac{\rho}{\rho^{max}} = \sum_{qi} \frac{f_q^{max}}{\rho^{max}} \widetilde{f_{qi}} \,. \tag{20}$$

Since all the f_{qi} are positive inputs, $\rho^{max} = U(\rho) = \sum_{qi} f_q^{max}$. We also define

$$\frac{1}{\widetilde{\rho}'} = \frac{\rho^{min}}{\widetilde{\rho}\rho^{max}}, \tag{21}$$

where ρ^{min} is the lower bound of the density and $\frac{1}{\widetilde{\rho}'} \in [0, 1]$.
According to (2) and the symmetry of the LBM, the right-hand factor of the scaled velocity $U(\boldsymbol{v})$ is

$$U(\boldsymbol{v}) = \frac{1}{\rho^{min}} \max_b \sum_{qi} f_q^{max}\{\boldsymbol{e_{qi}}[b] > 0\}, \tag{22}$$

where b is the dimension index of vector $\boldsymbol{e_{qi}}$. Note that $U(\boldsymbol{v})$ and v^{max} are scalars instead of vectors. Then, the scaled velocity is computed as

$$\widetilde{\boldsymbol{v}} = \frac{\boldsymbol{v}}{v^{max}} = \frac{U(\boldsymbol{v})}{v^{max}} \frac{1}{\widetilde{\rho}'} \sum_{qi} \left(\frac{f_q^{max}}{U(\boldsymbol{v})\rho^{min}} \boldsymbol{e_{qi}} \right) \widetilde{f_{qi}}. \tag{23}$$

With such range scaling, (17) and (18) become

$$\widetilde{f_{qi}}(\boldsymbol{x}+\boldsymbol{e_{qi}}, t+1) = \widetilde{f_{qi}}(\boldsymbol{x}, t) - \frac{1}{\tau}(\widetilde{f_{qi}}(\boldsymbol{x}, t) - \widetilde{f_{qi}^{eq}}), \tag{24}$$

$$\begin{aligned}\widetilde{f_{qi}^{eq}} = \frac{U(f_q^{eq})}{f_q^{max}} \widetilde{\rho} \Bigg(& \frac{A_q \rho^{max}}{U(f_q^{eq})} + \frac{2B_q v^{max}\rho^{max}}{U(f_q^{eq})} \left\langle \frac{\boldsymbol{e_{qi}}}{2}, \widetilde{\boldsymbol{v}} \right\rangle \\ & + \frac{4C_q (v^{max})^2 \rho^{max}}{U(f_q^{eq})} \left\langle \frac{\boldsymbol{e_{qi}}}{2}, \widetilde{\boldsymbol{v}} \right\rangle^2 \\ & + \frac{4D_q (v^{max})^2 \rho^{max}}{U(f_q^{eq})} \left\langle \frac{\widetilde{\boldsymbol{v}}}{2}, \frac{\widetilde{\boldsymbol{v}}}{2} \right\rangle \Bigg).\end{aligned} \tag{25}$$

For the D3Q19 model, $A_q \geq 0$, $B_q \geq 0$, $C_q \geq 0$, and $D_q \leq 0$, and hence

$$\begin{aligned} U(f_{qi}^{eq}) = \max(&(\rho^{max} A_q + 2B_q v^{max}\rho^{max} \\ &+ 4C_q (v^{max})^2 \rho^{max}), \\ &(2B_q v^{max}\rho^{max} - 4D_q (v^{max})^2 \rho^{max})). \end{aligned} \tag{26}$$

Note that in (25), we scaled the vectors before the dot products. The scaling factor is chosen to be a power of two so that it is easy to implement it in hardware.

6 Experimental results

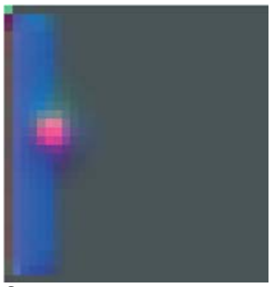
We have implemented our techniques on an Nvidia GeForce4 Ti 4600 card that has 128 MB of memory. For comparison, we also implemented the LBM in software on a PC with a 1.6G-Hz P4 processor and 512-MB DDR memory. We use the D3Q19 model throughout the experiments.

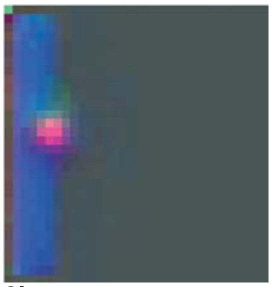
6.1 Accuracy

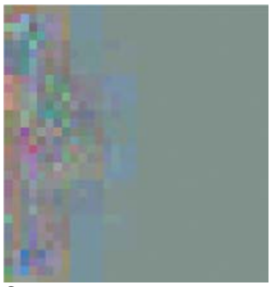
A major concern about using graphics hardware for general computation is the accuracy. Most graphics hardware supports only 8 bits per color channel. There have been a few limited supports of 16-bit textures but these are too restricted for a relatively complicated application such as the LBM simulation. Fortunately, the variables of the LBM fall into a small numerical range, which makes the range scaling effective. Besides, the property of the LBM that the macroscopic dynamics is insensitive to the underlying details of the microscopic physics [1] relaxes the requirement on the accuracy of the computation.
Figure 8a and 8b show two color-encoded velocity slices extracted from a 3D LBM simulation. Figure 8a was computed by the CPU with floating-point accuracy and Fig. 8b was obtained with our hardware approach. Figure 8c shows the exaggerated error image with all pixel values scaled up by 10. All the values were transformed to the range of [0, 255] for display. Visually, it is difficult to see any difference between Fig. 8a and 8b. Actually, the maximal pixel-wise difference is less than 1% after one step of simulation.

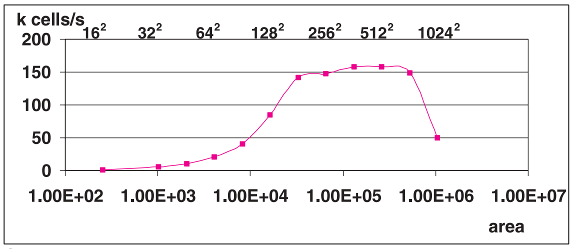
6.2 Performance

Stitching smaller textures into larger ones significantly reduces the overhead of texture switching. Before testing the performance of our hardware implementation of LBM, we first determine how large the stitched textures should be. Figure 9 shows the relationship between the area of the stitched textures and the number of cells that the hardware can handle per second. For our hardware configuration, a texture with a size of 512×512 performs the best. In fact, for textures smaller than 256×256, the computation time is nearly independent of the size of the textures. Note that we restrict the dimensions of the textures

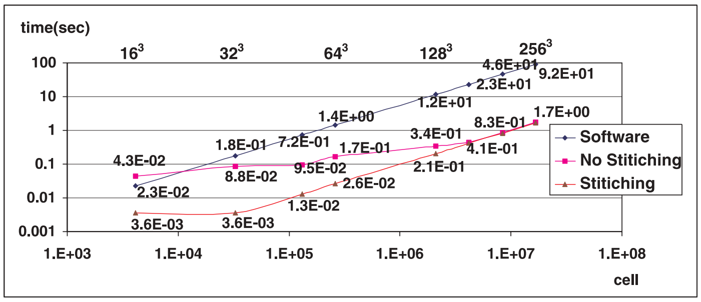

8a


8b


8c



9



10

Fig. 8. Velocity slice computed by **a** software and **b** graphics hardware. **c** The difference image between **a** and **b** scaled up by a factor of ten

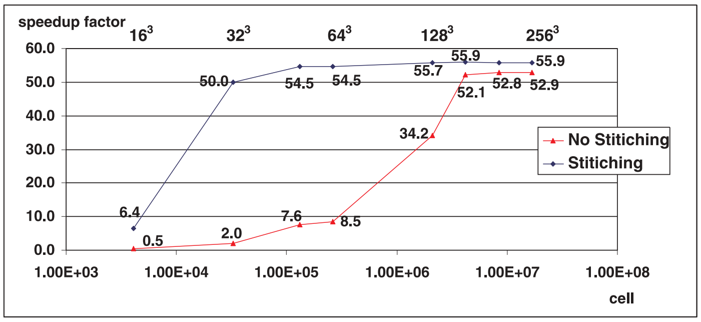
Fig. 9. Number of cells processed per second as a function of the area of the stitched textures. Note that the *X*-axis is on a logarithmic scale

Fig. 10. Time per step of the LBM computation in software and hardware with graphics hardware (with or without stitching)

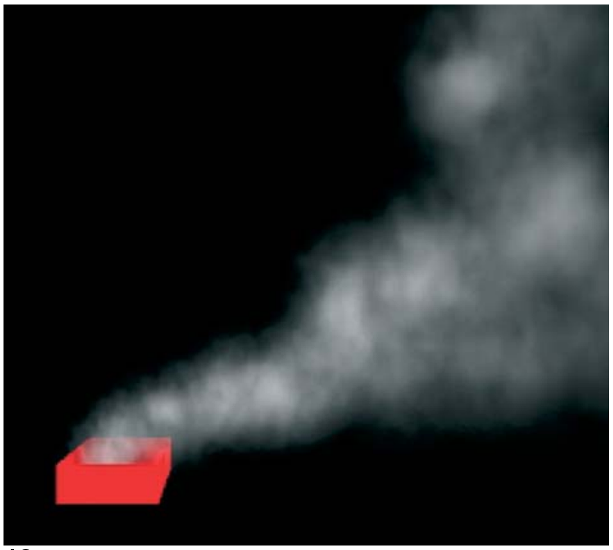
to be powers of two, although we could exploit the non-power-of-two-texture extension to achieve even better results.

Figure 10 compares the time (in seconds) per step of the hardware LBM with a software implementation. The statistics do not include the time for rendering. The "Stitching" curve refers to the performance after stitching small ones into 512×512 textures, while "No Stitching" does not. Note that the hardware-accelerated technique wins in speed for any size of the model, except that for the 16^3 grid, the "No Stitching" method is slower than software. However, simply by stitching the sixteen 16×16 textures into one 64×64 texture gains a speedup factor of 12. Both axes in Fig. 10 are on logarithmic scales. Note that stitching is very effective for grids equal to or smaller than 128^3.

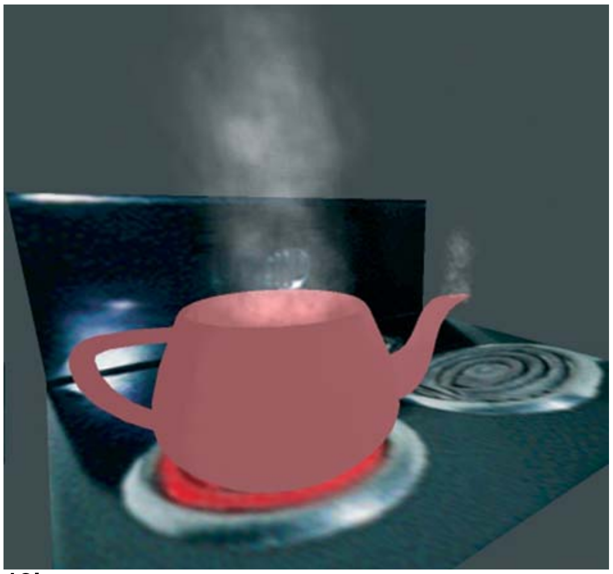
The speedup factor of the hardware-accelerated method against the software approach is more clearly shown in Fig. 11. The proposed method is at least 50 times faster than its software counterpart except for the 16^3 model. Note that the memory requirement of a 256^3-sized model is much larger than the memory on the graphics board. Actually, only the distribu-



11



12a



12b

Fig. 11. Speedup factor of our hardware-accelerated method compared with the software method
Fig. 12. Applications of the hardware-accelerated LBM: **a** Smoke emanating from a chimney and then blown up by the wind; **b** Hot steam rising up from a teapot and its spout

tions for a D3Q19 model need $256^3 \times 19 = 319$M. Our implementation still achieves 1.7 second per step for a 256^3 model, which is acceptable for an interactive application.

6.3 Application

We visualize the simulation results by either directly showing the color-encoded velocity field (as in Fig. 8) or by injecting particles into the velocity field from an inlet. These particles are considered massless. That is, they do not affect the flow calculation. The particles are advected according to the velocity of the grid cell. The number of particles depends on the density of the fluid. We then render the scene with texture splats [20]. Figure 12a shows an image of smoke emanating from a chimney that is then blown by the wind. The left side of the grid is assigned a speed along the X-axis to model the effect of wind. We also incorporate an upward force due to the difference in the temperature field. Figure 12b shows the result of hot steam rising up from a teapot and its spout according to a velocity field simulated with our LBM approach and hardware acceleration. Both the smoke and the steam are simulated on a 32^3 grid.

7 Discussion

In this paper, we presented algorithms for implementing LBM on commodity graphics hardware. Experimental results show that the LBM can be simulated on current low-cost computers in real time for a grid size of up to 64^3 and interactively for a grid size of 128^3.
Although we focused on the LBM, our techniques can be extended to other computations. It is also possible to generalize our methods into a framework of accelerating a large variety of applications on conventional graphics hardware and its future enhanced versions. One of our planned directions is a development environment including a language describing general parallel computations and a compiler that automatically translates code written in the language into available operations on graphics hardware. We would also like to develop a debugger for conveniently inspecting the intermediate results of the graphics pipeline.

Acknowledgements. This work has been supported by ONR Grant No. N000140110034. We thank Klaus Mueller and Suzi Yoakum-Stover for helpful discussions.

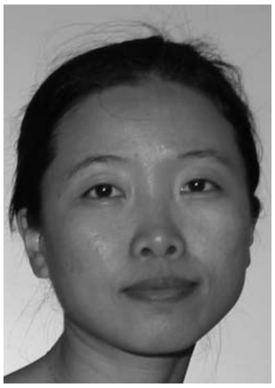
References

1. C. Cercignani (1990) Mathematical methods in kinetic theory. Kluwer Academic/Plenum Publishers
2. S. Chen, G.D. Doolean (1998) Lattice Boltzmann method for fluid flows. Annu Rev Fluid Mech 30:329–364
3. Fang S-F, Chen H-S (2000) Hardware accelerated voxelization. Comput Graphics 24:433–442
4. Harris M, Coombe G, Scheuermann T, Lastro A (2002) Physically-based visual simulation on graphics hardware. Hardware Workshop 02, Saarbrücken, Germany. In: SIGGRAPH/Eurographics Workshop on Graphics Hardware. ACM Press/ACM SIGGRAPH, pp 109–118
5. Heidrich W, Westermann R, Seidel H-P, Ertl T (1999) Applications of pixel textures in visualization and realistic image synthesis. I3D 99, Atlanta, GA, USA. In: ACM Symp Interactive 3D Graphics. ACM SIGGRAPH, pp 127–134
6. Hoff K, Culver T, Keyser J, Lin M, Manocha D (1999) Fast computation of generalized voronoi diagrams using graphics hardware. SIGGRAPH 99, Los Angeles, CA, USA. In: Proc SIGGRAPH. ACM SIGGRAPH/Addison-Wesley Longman, pp 277–286
7. Hoff K, Zaferakis A, Lin MC, Manocha D (2001) Fast and simple 2D geometric proximity queries using graphics hardware. In: ACM Symp Interactive 3D Graphics. ACM SIGGRAPH, Los Angeles, CA, USA, pp 145–148
8. Hopf M, Ertl T (1999) Accelerating 3D convolution using graphics hardware. IEEE Visualization '99, San Francisco, CA, USA. IEEE, pp 471–474
9. Hopf M, Ertl T (2000) Accelerating morphological analysis with graphics hardware. pp 337–345
10. Jobard B, Erlebacher G, Hussaini MY (2000) Hardware-accelerated texture advection for unsteady flow visualization. IEEE Visualization. pp 155–162
11. Kandhai BD (1999) Large scale Lattice-Boltzmann simulations. PhD thesis, University of Amsterdam
12. Larsen ES, McAllister D (2001) Fast matrix multiplies using graphics hardware. In: Int Conf High Performance Computing and Communications.
13. Muders D (1995) Three-dimensional parallel lattice Boltzmann hydrodynamics simulations of turbulent flows in interstellar dark clouds. PhD thesis, University at Bonn
14. Mueller K, Yagel R (1999) On the use of graphics hardware to accelerate algebraic reconstruction methods. In: SPIE Medical Imaging Conf
15. Peercy MS, Olano M, Airey J, Ungar PJ (2000) Interactive multi-pass programmable shading. In: SIGGRAPH 2000, New Orleans, LA, USA. Proc ACM SIGGRAPH. ACM Press/ACM SIGGRAPH/Addison-Wesley Longman, pp 425–432
16. Proudfoot K, Mark WR, Tzvetkov S, Hanrahan P (2001) A real-time procedural shading system for programmable graphics hardware. SIGGRAPH 2001, Los Angeles, CA, USA. In: Proc ACM SIGGRAPH. ACM Press/ACM SIGGRAPH, pp 159–170
17. Purcell T, Buck I, Mark W, Hanrahan P (2002) Ray tracing on programmable hardware. SIGGRAPH 2002, San Antonio, TX, USA. In: Transactions on Graphics 21(3):703–712
18. Rezk-Salama C, Scheuering M, Soza G, Greiner G (2001) Fast volumetric deformation on general purpose hardware.

In: Proc SIGGRAPH/Eurographics Workshop on Graphics Hardware.
http://citeseer.nj.nec.com/449275.html

19. Trendall C, Stewart AJ (2000) General calculations using graphics hardware with applications to interactive caustics. Rendering Techniques: 11th Eurographics Workshop on Rendering, Brno, Czech Republic. Eurographics, pp 287–298
20. Wei X-M, Li W, Mueller K, Kaufman A (2002) Simulating fire with texture splats. Proc IEEE Visualization. pp 227–234
21. Weiskopf D, Hopf M, Ertl T (2001) Hardware-accelerated visualization of time-varying 2D and 3D vector fields by texture advection via programmable per-pixel operations. Workshop on Vision, Modeling, and Visualization VMV. pp 439–446

WEI LI is a Ph.D. candidate in the Computer Science Department of Stony Brook University, USA. He received his B.E. and M.E. degrees in 1992 and 1995 respectively, both in Electrical Engineering, from Xi'an Jiaotong University, China, and a master degree in Computer Science from Stony Brook University in 2000.. His current research interests are focused on Volume Visualization and Graphics Hardware. For more information, see http://www.cs.sunysb.edu/∼liwei.



XIAOMING WEI is a Ph.D. candidate in the Computer Science Department at the Stony Brook University, where she is also a Research Assistant in the SUNYSB Center for Visual Computing (CVC). Xiaoming received her B.Sc. from Beijing University of Aeronautics and Astronautics of China in 1995 and a M.Sc. in Computer Science from Tsinghua University of China in 1998. Her research interests include physically-based modeling, natural phenomena modeling, and computer graphics. For more information, see http://www.cs.sunysb.edu/∼wxiaomin.

ARIE E. KAUFMAN is the Director of the Center for Visual Computing (CVC), a Leading Professor and Chair of the Computer Science Department, and Leading Professor of Radiology at the State University of New York at Stony Brook. He was the founding Editor-in-Chief of the IEEE Transaction on Visualization and Computer Graphics (TVCG), 1995–1998. Kaufman has been the co-Chair for multiple Eurographics/Siggraph Graphics Hardware Workshops and Volume Graphics Workshops, the Papers/Program co-Chair for the ACM Volume Visualization Symposium and the IEEE Visualization Conferences, and the co-founder and a member of the steering committee of the IEEE Visualization Conference series. He has previously chaired and is currently a director of the IEEE Computer Society Technical Committee on Visualization and Computer Graphics. He is an IEEE Fellow and the recipient of a 1995 IEEE Outstanding Contribution Award, the 1996 IEEE Computer Society's Golden Core Member, 1998 ACM Service Award, 1999 IEEE Computer Society's Meritorious Service Award, and 2002 State of New York Entrepreneur Award. Kaufman has conducted research and consulted for over 30 years, specializing in: volume visualization; graphics architectures, algorithms, and languages; virtual reality; user interfaces; and multimedia. He received a BS (1969) in Mathematics and Physics from the Hebrew University of Jerusalem, an MS (1973) in Computer Science from the Weizmann Institute of Science, Rehovot, and a PhD (1977) in Computer Science from the Ben-Gurion University, Israel. For more information, see http://www.cs.sunysb.edu/∼ari.