Sliced Data Structure for Particle-Based Simulations on GPUs

Takahiro Harada* Seiichi Koshizuka† Yoichiro Kawaguchi‡

The University of Tokyo

Abstract

We present a sliced data structure that is effective for use in neighboring particle search for particle-based simulations. In this method, a grid is dynamically constructed to fit to a particle distribution. Rather than computing the grid to fit perfectly to the particle distribution, it compute a grid with some margin to the distribution. This lowers the computation cost of constructing the data structure. Before storing particle indices on a grid, key values which are used to compute the index of a voxel are calculated. The proposed data structure can be introduced into particle-based simulations that run entirely on the Graphics Processing Unit (GPU) because the construction of this data structure and access to storing values can also be also performed entirely on the GPU. The proposed data structure removes the restriction of a computation region with a fixed grid and makes it possible to simulate particle motions over a larger area. Moreover, the cost of the proposed method is low enough for use in real-time applications. In this paper, we first introduce the sliced data structure and then describe its implementation on the GPU. Finally, we apply the propsed method to particle-based simulations and present its quantitative evaluations.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Physically based Modeling; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation

1 Introduction

Fluid simulation plays an important role in computer graphics, because the motion of fluid is complex and it is difficult to manually create an animation. Several kinds of studies have been conducted on this topic. Simulation methods for fluids are classified into two categories: Lagrangian and Eulerian methods. Particle methods are Lagrangian methods and they do not have to track the surface of a fluid unlike the Eulerian methods, because particles themselves represent a fluid. Furthermore, Eulerian simulations cannot compute a phenometa smaller than the size of the grid, but Lagrangian methods have some ability in this respect. For example, they can easily compute fine splashes. However, the drawback of particle methods is their high computational cost. In each iteration, neighboring particles have to be searched for for each particle because the particles do not have connectivity and their distribution can change dynamically. For To improve the efficiency, a uniform grid can be introduced for storing the index of a particle in the voxel to which the particle belongs. A uniform grid is defined to cover the computational region. Using a uniform grid, the neighboring particles of a particle are restricted to those whose indices are stored in the neighboring voxels. However, this approach is not efficient from the point of view of the memory consumption when the fluid distribution changes because there are many voxels in which no particles are located. These voxels are nothing but a waste of memory. In general, limited memory is available. Therefore, we can use much memory for the physical values of the fluids and more particles can be used in a simulation as less memory is required by the grid. The other choice would be a hash grid, which computes a hash value for each voxel and it maps the all the voxels onto a fixed-size memory. Thus, we can compute using a smaller amount of memory containing less free space[Teschner et al. 2003]. However, hash collision may occur because the hash function does not guarantee one by one mapping of the voxels to the memory in general. It means that several voxels are mapped to the same location that is not efficient when GPUs are used because the load balance is not equivalent. We need to check many indices if several voxels are mapped to the same location while checking of only a small number of indices is needed for a voxel where no other voxel is mapped to this location. A perfect hash can be calculated as presented by [Sylvain and Huges 2006], but it has a high construction cost. Therefore, it is not a good choice in a particle-based simulation in which the hash table have to be reconstructed for every iteration.

This paper presents a memory-efficient data structure of a grid for neighboring particle search in particle-based simulations. In this method, the computational region is sliced into several two-dimensional planes and one index and other several other values on each slice are calculated before storing data on the grid. The index of a voxel is calculated using these values and we can access the memory allocated for the voxel. Although the grid constructed by this method is not perfect and leaves several empty voxels, it improves the memory efficiency in comparison to a uniform grid. The construction of a perfect grid is computationally expensive whereas a uniform grid is inexpensive but requires much memory. The proposed data structure achieves a balance between the ease of construction and memory requirements. The present method can be implemented entirely on GPUs; therefore it can be integrated into a particle-based simulation running on GPUs. In the sliced data structure, the memory used to store values of voxels is more densely allocated than on a fixed grid, thus improving the cashe hit and therefore resulting in an improvement in performance. This method is useful not only for an offline simulation but also for an real-time application. If we compute the same simulation area with the proposed data structure and the fixed uniform grid, the amount of memory required for the former is less than that for the latter. Therefore, using the proposed method, we can increase the resolution of the simulation or we can extend the area of the computational region. These are the benefits of the present method.

After discussing the related works and requirements for a grid used in particle-based simulations, we are going to introduce the data structure and then describe the method for constructing the data structure on the GPU. Finally, we integrate the proposed data structure into particle-based simulations that are performed entirely on the GPU and we show the capability of the proposed method through several quantitative evaluations.

*e-mail: takahiroharada@iii.u-tokyo.ac.jp

†e-mail:koshizuka@q.t.u-tokyo.ac.jp

‡e-mail:yoichiro@iii.u-tokyo.ac.jp

GRAPHITE 2007, Perth, Western Australia, December 1–4, 2007.
© 2007 ACM 978-1-59593-912-8/07/0012 $5.00

2 Related Work

The goal of this study is to construct an efficient data structure for particle-based simulations. We first describe several studies on the topic. Moving Particle Semi-Implicit (MPS) method is a particle-based method for computing fluid motion[Koshizuka and Oka 1996]. In MPS, a Poisson equation deduced from the mass conservation equation is solved for an incompressible fluid. However, the solution of this large equation is computationally expensive and so Smoothed Particle Hydrodynamics (SPH), which has been developed in the field of astronomy[Monaghan 1992a], is often used in the computer graphics community because it can be solved fully explicit. Premoze *et al.* introduced MPS to the computer graphics community[Premoze et al. 2003] and Müller *et al.* studied SPH for real-time applications[Müller et al. 2003]. Adams *et al.* studied adaptive sampling in SPH simulation to compute a large area efficiently[Adams et al. 2007]. In particle simulations, particles themselves do not have connectivities among them and can move freely. This is a benefit of particle simulations but the drawback is that neighboring particles with which a particle interacts have to be searched for in every time step. If we use the brute-force method, the computation cost is $O(n^2)$ and so the cost grows as the number of particles increases. Therefore, a fixed uniform grid which stores indices of particles are used to make it efficient[Clifford 1992; Monaghan 1992b]. A uniform grid which has identical-sized voxels is effective when all the particles have the same effective radius. However, a hierarchical grid such as octree is used when the effective radius differs for each particle[Hernquist and Katz 1989]. The reason why a uniform grid is prefered for a simulation with identical effective radius is that the cost to access is low; $O(1)$ for a uniform grid while $O(\log n)$ for a hierarchical grid to access a leaf node.

Another background of this study is about GPUs. The architecture of GPUs is designed as parallel processors to specialize in processing graphics task. Therefore, it can compute graphics tasks such as vertex transformation much faster than a CPU. The current GPUs are programmable, i.e., we can write shader programs to control the GPU. Therefore, it can be used to accelerate not only graphics applications but also other applications[Owens et al. 2005] such as cellular automata[Harris et al. 2002], Eulerian fluid simulation[Harris et al. 2003], and cloth simulation[Cyril 2005]. There are several researches on accelerating particle simulation using the GPU. Kipfer *et al.* accelerated particle simulation without perfect collisions using the GPU[Kipfer et al. 2004]. Amada *et al.* used the GPU to accelerate SPH but the neighboring list of each particle was calculated on the CPU[Amada et al. 2004]. Therefore data transfer between these processors was necessary in each time step, which lowered the computation efficiency. Kolb *et al.* first calculated physical values on the grid and then computed the values for the particles using the data on the grid[Kolb and Cuntz 2005]. Although particle-based simulation is free from the numerical dissipation of advection computation, its first step introduces undesirable numerical dissipation. Harada *et al.* developed a method that overcomes these problems[Harada et al. 2007], wherein they used a fixed uniform grid to search for neighboring particles and store particle indices on the grid by multi-pass rendering. Although they could exploit the computational power of the GPU, the use of a fixed uniform grid is not efficient from the point of view of memory consumption as discussed above. This is fatal when we simulate a large computational area.

Other than the uniform grid or hash grids as discussed above, there are several alternatives when constructing a grid. Purcell *et al.* studied a method to use a sorting[Purcell et al. 2003] wherein particle indices are first sorted by the voxel indices and the value in a grid is searched for using a binary search. This method is efficient from the memory usage viewpoint because nothing is prepared to the empty voxel. However, it is more expensive to identify the memory address of the voxel to which a point is belongs than a uniform grid in which the address of the data storing the voxel values can be calculated in a few arithmetic calculations. Greβ *et al.* studied bounding volume hierarchy using the GPU[Greβ et al. 2006]. In each traversal step, the result of overlap grew by a factor of 16, therefore they developed a method to remove null references and to pack a sparse data into a compacted stream by using mipmapping. If we use this method in a particle-based simulation to construct a dense grid, we still have to prepare a large uniform grid. Therefore, it does not solve the problem of high memory consumption.

3 Requirements

The data structures that can be introduced to make neighboring particle search efficient are classified into three categories; uniform grids, hash grids and hierarchical grids. There are three requirements for a grid used in particle-based simulations. The first is that the construction cost should be low because it has to be reconstructed for every iteration. The second is that it should be easy to access the memory for the voxel to which a point belongs because the data stored in the memory is frequently referred to during simulations. The last is that the size of memory that is used for the grid should be small. In the following, we discuss these points in the three grids.

Let the total number of particles used in a simulation be n, the construction cost of a uniform grid is $O(n)$ which is small enough. Accessing the memory of a voxel to which a point belongs can be performed by coordinate transformation from the simulation space to the grid space which is small number of arithmetic operations. Therefore, the cost does not depend on the total number of particles n. Although a uniform grid satisfies the two requirements, it needs a large memory for all the voxels in the computation region. There are a large number of empty voxels storing no particles which is nothing but waste of memory.

A hash grid would be a good choice if there were no hash collision because it can access to the memory easily and it does not require a large memory. However, there are generally many hash collisions and this leads to nonuniform computation cost to access the data. Therefore, the costs of access to a voxel and construction of a grid increase as much hash collision occurs.

Because the construction cost of a hierarchical grid is $O(n \log n)$, it is much expensive than a uniform grid. However, it achieves high memory efficiency because of the hierarchical structure. The drawback of the hierarchy is the cost to access a leaf node, which requires $O(\log n)$. This cost is not small when a large number of particles are used or the computation space is large in comparison to the particle size.

When the computation is performed on the GPU, all the computation should be parallelized. If some operations need to use the CPU, data have to be transferred between them. The speed of data transfer is slower than computations. Another requirement is that the computation burden on each unit should be uniform. Harada *et al.* showed that construction of a uniform grid can be parallelized entirely on the GPU[Harada et al. 2007] and the loads to access voxels are equivalent on all the computation units. A hash grid has difficulty to satisfy these requirements because of hash collision which result in unequivalent accessing cost. To summarize the discussion, a uniform grid is memory inefficient and a hash grid is not suited to implement on the GPU. Construction of a grid and accessing to a voxel is computationally expensive in a hierarchical grid. The sliced data structure proposed in this paper satisfies all of these requirements. The cost of construction of a grid is $O(n)$, the cost to

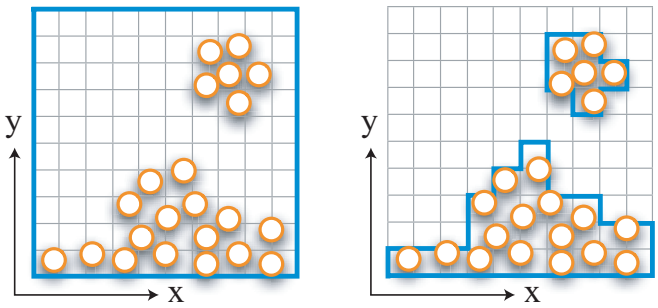


Figure 1: A fixed grid (left) and a dynamic grid (right) constructed using the proposed method. Memory is allocated in the region inside of the blue lines.

access to a voxel does not depend on n and it is almost the same to a uniform grid. The sliced data structure is memory efficient because it culls most of the empty voxels. Furthermore, all the computations are performed on the GPU and the load balance is equivalent among the units.

4 Sliced Data Structure

We first prepare a grid in the computational space. The size of the grid is infinite and the grid does not change the configuration in the space. Note that the grid is not allocated in the memory at this time. The first step is to determine the number of voxels needed and to allocate memory for them. When we used a fixed grid, a bounding box was defined to enclose the computational region and memory for the voxels inside of the bounding box was allocated.

We define an axis and divide the grid in the space into slices perpendicular to this axis. Each slice has one voxel thickness in the direction of the axis. Thus, the slices have one dimension less than the spatial dimension of the computation region. When the computation region is three dimensions and Y axis is chosen as the axis, the slices are two-dimensional grids spread out in the directions over XZ plane. The following explanation assumes this situation. After dividing the computational space into slices, a two-dimensional bounding box is computed on each slice, i.e., the largest and smallest voxel coordinate in XZ directions $bx_{i,max}, bx_{i,min}, bz_{i,max}, bz_{i,min}$ are computed where i is the index of the slice. With these values, the number of voxels in the X and Z directions nx_i, nz_i are computed as

$$nx_i = \frac{bx_{i,max} - bx_{i,min}}{d} + 1 \quad (1)$$

$$nz_i = \frac{bz_{i,max} - bz_{i,min}}{d} + 1 \quad (2)$$

where d is the side length of voxels. Then, the number of voxels in slice i is calculated as $n_i = nx_i \times nz_i$. The proposed method allocates memory to voxels inside of the bounding box of a slice.

After a bounding box is defined, we can calculate the index of a voxel located within the bounding box. However, what we need is the global index in the all the computational region, i.e., the index in all the bounding boxes. The next step is to calculate the global indices. We compute the first indices in bounding boxes because all of the indices of voxels in a bounding box can be calculated with the index of the first voxel and several values defining the bounding box. Assume that the computational region is divided into n slices $\{S_0, S_1, S_2, \cdots, S_{n-1}\}$. Then the index of the first voxel in slice S_i is defined as the sum of the numbers of voxels from slice S_0 to slice S_{i-1};

$$p_i = \sum_{j<i} n_j. \quad (3)$$

The index of the voxel to which a point (x, y, z) belongs is calculated in two steps. In the first step, the number of the slice in which the point is located is calculated using the minimum coordinate of voxels by_{min} in the Y direction as follows.

$$i = \left[\frac{y - by_{min}}{d} \right] \quad (4)$$

Then the index of the voxel $v(x, y, z)$ is calculated using the values that determine the bounding box in the slice.

$$v(x, y, z) = p_i + \left(\left[\frac{x - bx_{i,min}}{d} \right] + \left[\frac{z - bz_{i,min}}{d} \right] nx_i \right) \quad (5)$$

From Equation 5, we can see that the values needed to identify the index of the voxel wherein a point is located are the values that define the bounding box in the slice $bx_{i,min}, bz_{i,min}, nx_i$, the value to determine the index of the slice by_{min}, the index of the first voxel in the slice p_i and the side length of a voxel d. An example of how the voxels are allocated in memory is illustrated in Figure 1.

To construct the sliced data structure, the bounding boxes of each slice is determined and the index of the first voxel in each slice is calculated. After the values that are needed to identify the index of voxel are thus computed, the values are stored in the voxels.

5 Implementation on GPUs

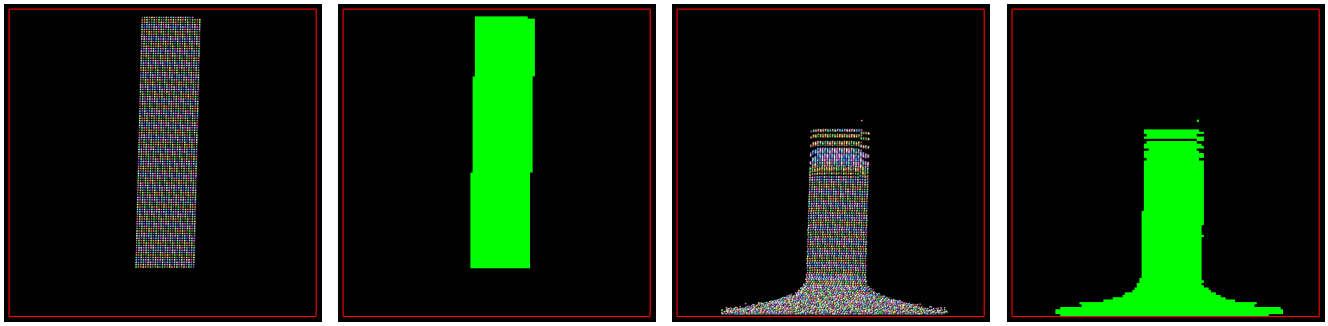
In this section, we first describe the way to store the data used in the proposed method and then describe the operations performed to construct the data structure. This section explains an example, how the particle indices are stored in a grid, but it is also applicable for storing other values.

5.1 Data Structure

To compute on the GPU, data have to be stored in the video memory in the form of textures. A one-dimensional texture is prepared to store values that identify the index of the voxel in each slice. When the number of slices exceeds the maximum size of a one-dimensional texture, we can store them in a two-dimensional texture. The values needed for each slice are four; $p_i, bx_{i,min}, bz_{i,min}, nx_i$. Therefore, one texture is sufficient to store them by packing the values in the RGBA channels. A two-dimensional texture is prepared to store indices in voxels. This texture is called *index pool texture*. In this study, one texel is assigned to a voxel and the number of texels in an index pool texture is the limit of maximum number of the voxels that can be allocated in the memory. The address of a pixel to which a voxel is allocated is calculated using the unique one-dimensional index and the side length of the texture. Figure 3 shows how the voxel in the slices are stored in a texture memory.

5.2 Computation of Bounding Boxes in Slices

To determine the bounding box in a slice, the largest and smallest voxel coordinates in XZ axes have to be calculated. The slice index of a particle is computed in Equation 4 and the values are computed by selecting the largest and smallest XZ coordinates of the particles in each slice. When the GPU is used, a vertex is assigned to each particle and written onto a one-dimensional texture as point primitives of size 1. A texel of the one-dimensional texture will be used



(a) (b) (c) (d)

Figure 2: Three-dimensional particle-based simulation using a dynamic grid which is constructed by the propsed method. The area in the red lines are allocated in the memory when a uniform grid is used. The green areas in (b) and (d) indicate the grids allocated in the memory.

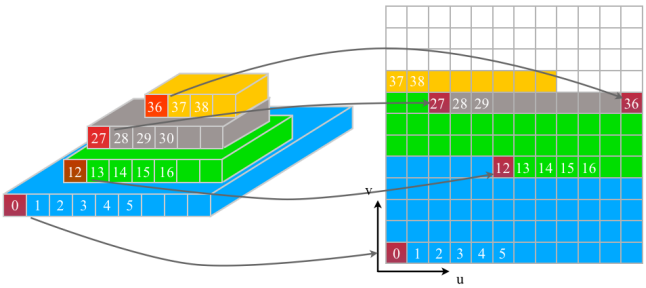


Figure 3: This figure shows that how the voxels are allocated in the memory. The voxel is allocated in the memory from the bottom of the slices.

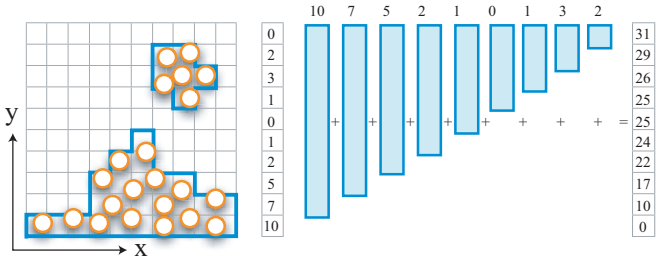


Figure 4: Computation of the index of the first voxel in each slice.

to store the key values of a slice. In the vertex shader, the index of the slice, which is the position to write, is calculated by Equation 4 and set as the output position of the vertex. The fragment shader outputs the XZ coordinates of the particle as color. In this way, particles in a slice is written to the same pixel. Among these values, we have to select the maximum and minimum coordinates. These values can be selected by using a function of GPUs such as the alphablending to select the maximum values.

5.3 Computation of The Indices of The First Voxels

The number of voxels in the slice can be computed using the largest and smallest voxel coordinates in XZ axes. The next step is the evaluation of Equation 3 to obtain the index of the first voxel in each slice. If we evaluate the equation by reading values from n_0 to n_{i-1} for each slice i, $m(m+1)/2$ memory accesses operations are necessary where m is the total number of slices. So the total computation time is limited by the latency of the memory accesses. When the GPU is used, this step is a gathering operation by a fragment shader. Furthermore, the workload differs among pixels. Therefore, another strategy is employed to calculate the indices of the first voxels.

We prepare a one-dimensional array of slices and initialize the elements of the array to zero. For slice i, the number of voxels n_i in the slice is read and then added to the elements from $i+1$ to m of the array. In this way, the indices of the first voxels in the slices are evaluated by m times of memory accesses. This operation is a scattering operation and so it cannot be performed in a fragment shader. In this study, a vertex shader is used to evaluate the equation. The array to be written is a one-dimensional texture with m texels. For each slice i, values have to be written to texels from $i+1$ to m. This operation is performed by rendering two points at $i+1$th and m th texels, respectively and connecting them with a line. The value n_i which is the total number of voxels in slice i is written as pixel colors. The summation is done by using additive operation of the alpha blending. In this way, the computation of Equation 3 is accomplished by rendering m lines. Figure 4 illustrates this process in two spatial dimensions.

5.4 Storing and Reading Values

After computing the index of the first voxel in each slice and the values defining the bounding boxes, the particle indices are written in the index pool texture. The slice index is computed with the Y coordinate of the particle and read the index of the first voxel p_i, the smallest coordinate of the voxel in the slice $bx_{i,min}, bz_{i,min}$ and the number of voxels in the X axis nx_i. Using these values, Equation 5 is evaluated and the index of the voxel to which the particle belongs is calculated. For writting a value to the corresponding texel, a vertex is assigned to each particle and rendered it as a point primitive of size 1 and the value is output in the color. A value stored in a voxel to which a point belongs is read by calculating the index of the voxel as described above.

6 Application to Particle-based Simulations

Because the computation of the construction of the data structure is computed entirely on the GPU, it can be introduced to particle-based simulations that are also executed entirely on the GPU. The benefit of using the GPU for a simulation is its speed as discussed in [Harada et al. 2007]. We applied the proposed method to Distinct Element Method (DEM) and Smoothed Particle Hydrodynamics (SPH). We only describe DEM in this section because the detail

Table 1: Ratio of maximum number of voxels used in the proposed method and total number of voxels in the computational region.

Number of particles	Ratio of voxels
16,386	0.0394
65,536	0.0619
262,144	0.269
589.824	0.488

Table 2: Computation times of DEM simulations in 128^3 grid (in milliseconds). Computation time for construction of a fixed grid is shown in (a) and that for construction of a grid using the proposed method is shown in (b). Total computation time using a fixed grid is shown in (c) and that using the proposed method is shown in (d). Maximum number of voxels used in the proposed method is shown in (e).

Number of particles	(a)	(b)	(c)	(d)	(e)
16,386	1.48	0.688	2.53	2.19	82,675
65,536	3.91	2.56	9.68	8.58	129,781
262,144	13.9	10.3	42.0	36.9	564,354
589.824	31.7	23.0	96.4	90.9	1,022,551

and implementation of SPH are included in [Harada et al. 2007]. To store up to four particles in a voxel, the method uses the depth and stencil tests and the color masks is used[Harada et al. 2007].

6.1 Distinct Element Method

Granular materials are represented by particles in DEM. The motion of particle i is calculated with contact force exerted by the colliding particles and gravity as shown below.

$$\frac{d\mathbf{v}_i}{dt} = \frac{1}{m}\sum_{j\in contact}\mathbf{f}_{ij}^c + \mathbf{g} \tag{6}$$

$$\frac{d\mathbf{x}_i}{dt} = \mathbf{v}_i \tag{7}$$

$$\tag{8}$$

where $\mathbf{x}_i, \mathbf{v}_i, m$ and $\mathbf{g}$ are the position vector of the particle center, velocity of the particle, mass and gravity, respectively. $\mathbf{f}_{ij}^c$ is the force of collision response.

There are many models in literature for computing repulsion force[Mishra 2003]. We used linear springs and dash pots. Let $\mathbf{x}_i, \mathbf{x}_j, \mathbf{v}_i, \mathbf{v}_j$ and $d/2$ be the position vectors of particles i and j, the velocity vectors and the radius of these particles, respectively. Two particles collide when the distance between the particles $|\mathbf{r}_{ij}|$ becomes smaller than their diameter d. When the two particles collide, the repulsion force $\mathbf{f}_{ij}^{spring}$ which is proportional to penetration works in the direction from the center of particle i to that of particle j.

$$\mathbf{f}_{ij}^{spring} = -k_s(d - |\mathbf{r}_{ij}|)\frac{\mathbf{r}_{ij}}{|\mathbf{r}_{ij}|} \tag{9}$$

where k_s is the spring coefficient. Damping force is modeled proportional to the relative velocity.

$$\mathbf{f}_{ij}^{damp} = \eta(\mathbf{v}_j - \mathbf{v}_i) \tag{10}$$

where η is the damping coefficient. The force from the particle j to particle i is the sum of $\mathbf{f}_{ij}^{spring}$ and $\mathbf{f}_{ij}^{damp}$.

7 Results

The proposed method was implemented on a PC equipped with a Core2 X6800 CPU, a GeForce 8800GTX GPU and 2.0GB RAM. The programs were written in C++, Op enGL and C for Graphics[Fernando and Kilgard 2003].

A three-dimensional simulation result of DEM using 65,536 particles is shown in Figure2. The particle motions are simulated in a cube illustrated by red lines and the initial distribution of particles is shown in the left of the figure. The particles are dropped and the results are rendered by orthogonal projection in the Z axis direction. The point sprites are used for rendering. When we use a fixed grid, we have to allocate memory for 128^3 voxels in the region inside of the red lines. However, with the proposed method, we have to allocate memory only for the green region. When a fixed grid is used, 32 MB of memory are needed to store particle indices. On the other hand, the sliced data structure requires a maximum of only 2 MB of memory. We can see that the configuration of the voxels is tighter to the particle distribution and it improves the efficiency of memory consumption. Table 1 shows the ratio of the size of the memory needed for the proposed method to that needed for a fixed grid whose size was 128^3 voxels in several simulations where the total number of particles were changed. We can see that from four to fifty percent of voxels are needed for the sliced data structure. Table 2 shows the comparison of the computation times of DEM simulations. The computation time for construction of a grid is the times to calculate the key values and to store particle indices to the voxels. The times for construction of a grid for the sliced data structure is less in all the cases although additional computations are required to calculate the key values. We can also see that the difference decreases as the total number of particles increases. In other words, the number of voxels prepared for the present method gets closer to that for the fixed grid. Therefore, this is because the size of the texture to store the particle indices is smaller and so the overhead of preparation of the texture must be the cause of this difference.

The next examples are results of three-dimensional SPH simulations. Table 3 shows the comparison between the computation times of simulations using a fixed grid and the sliced data structure. To enclose the entire simulation region, we had to prepare $256\times256\times256$ voxels and hence 256MB of memory were needed for the grid. In contrast, the proposed method required a maximum of only 15 MB when 1,048,576 particles were used. We can also see from Table 3 that simulations using the proposed method are faster than simulations using fixed grids. The procedure of grid construction of is the same as that for the DEM simulation. Therefore, the grid construction phase in the SPH simulation is 1.35 ms faster with the proposed method when 65,536 particles are used. However, Table 3 also shows that the computation time for one iteration with the same number of particles is 7.8 ms less with the proposed method. This indicates that the computation time excluding the grid construction is faster with the sliced data structure. The difference in the programs is the part that read particle indices from a texture storing them and the proposed method requires additional floating-point operations to identify the memory address of the voxel in the texture. Thus the proposed method is computationally more expensive. We conclude that the difference in the speeds came from the usage of the cashe. For the sliced data structure, more values which must be read from the texture memory remain in the cashe memory because the particle indices are stored more densely in the index pool texture.

Screenshots of two simulations and the index pool textures at the corresponding time step are shown in Figure 7. Point sprites are used for rendering and the color indicates the density of the fluid. A particle with high density is colored with blue and one with low

density is colored with white. The color in the index pool textures shows whether there are particles; a black texel represents an empty voxel and red, yellow and white texels represents voxels in which there are one, two and three indices, respectively. The scene shown at the top of Figure 7 gives an instance where the proposed method is especially effective because there are few empty voxels. For the scene shown at the bottom, the indices of particles are packed densely in the index pool texture but it has more empty voxels in comparison to the scene shown at the top. However it is definitely more efficient than the simulation with the fixed grid. The reason of empty voxels at the top is because of the disturbance of the fluid surface caused by a droplet.

Harada *et al.* showed that SPH simulation can be accelerated on the GPU[Harada et al. 2007]. Although they did not compare the time for grid construction, it must be faster on the GPU by considering the acceleration of the computation for a time step. We measured the computation times for construction of the sliced data structure and storing particle indices to it on the GPU and the CPU. The comparison is shown in Figure 5. We can see that the computation on the GPU is much faster than the computation on the CPU.

As we described in Section 3, a hierarchical grid is efficient from a perspective of memory consumption although the cost to access to a voxel is high. Figure 6 shows the comparison of the memory consumption between the sliced data structure and octee. This figure shows the memory consumption in the simulation shown in Figure 2. The sliced data structure does not waste any memory at the begining of the simulation where particles are arranged in a rectangular solid and the amount of memory is the same to the memory for the leaf nodes in octree. This is the reason why the sliced data structure requires less memory to the octree. However, as the simulation proceeds, the sliced data structure requires much memory because it allocates memory for several empty voxels. At the last of the simulation, the memory allocated in the sliced data structure and octree is 6.25% and 3.24% of the memory required for the uniform grid, respectively. This indicates that the sliced data structure can lower the allocated memory although it is not as efficient as the hierarchical grid.

The proposed method also removes the restriction on the size of the computational region because it requires no fixed grid. Therefore, we can easily simulate a scene in a large area. Figure 8 is a simulation result for a large area using about four million particles and the size of the computational region is $700 \times 700 \times 256$ voxels. When we use a fixed grid, about 2 GB of memory are needed just for the grid storing the particle indices. We could not simulate the scene with a fixed grid because the size was larger than the size of video memory which is available at the time of development. However, the size of memory required by the proposed method was about 150 MB and we could run the simulation with the current hardware.

We can also determine the total number of voxels required for the index pool texture by reading just one texel in the one-dimensional texture. This is another benefit of the proposed method because a small amount of data transfer is needed to determine the size of the index pool texture. Therefore, we can easily adjust the size of the index pool texture to fit the memory requirement of the voxels. However, we did not dynamically change the size of the index pool texture because it runs slower than programs with a fixed-sized index pool texture. This is because of the time required for allocation and deletion of an index pool texture.

The sliced data structure cannot improve the memory efficiency when there are two separated simulation region. It allocates memory for the empty voxels between the two regions. However, it can be improved by dividing the computation region into two and constructing the sliced data structure for each region.

Table 3: Computation times of SPH simulations in 256^3 grid (in milliseconds). Total computation times using a fixed grids are shown in (a) and those using the proposed method are shown in (b).

Number of particles	(a)	(b)
65,536	60.9	53.1
262,144	280.5	231.3
589.824	685.9	567.9
1,048,576	1160.9	1070.3

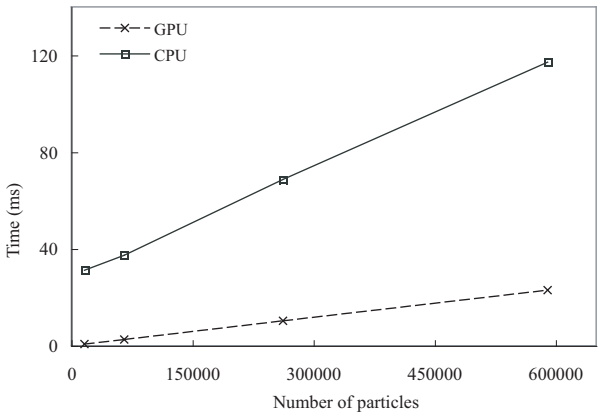


Figure 5: Comparison of construction times of the sliced data structure on the GPU and on the CPU.

8 Conclusions and Future Work

We have presented the sliced data structure wherein the memory for a grid is dynamically allocated. The computational region is divided into slices and bounding boxes are defined for each slice. The index of a voxel is calculated using several key values that are the indices of the first voxels and the values defining the bounding boxes. The proposed sliced data structure achieved a balance between the ease of construction and memory requirements. We also presented how to construct the data structure on the GPU and integrate the present data structure in particle-based simulations that run entirely on the GPU. We showed that the proposed method can achieve dense packing of the values of voxels and that this improves the cashe efficiency. As a result, a simulation with the proposed method runs faster than with the uniform grid although it needs more arithmetic computation to compute the memory address of a voxel.

In this paper, we used the sliced data structure to store particle indices in a grid and applied to DEM and SPH simulations. It is also applicable to a rigid body simulation using particles[Harada 2007]. The proposed data structure is also useful other than storing particle indices in a grid. An example is a grid to storing implicit function of particles, which is often used to construct the surface from a result of a particle simulation. With the proposed method, we can dynamically make the grid which fit to the particle distribution and improve the efficiency of the memory usage. Integrating the proposed method to a visualization system using the GPU is a future work.

References

ADAMS, B., PAULY, M., KEISER, R., AND GUIBAS, L. 2007. Adaptively

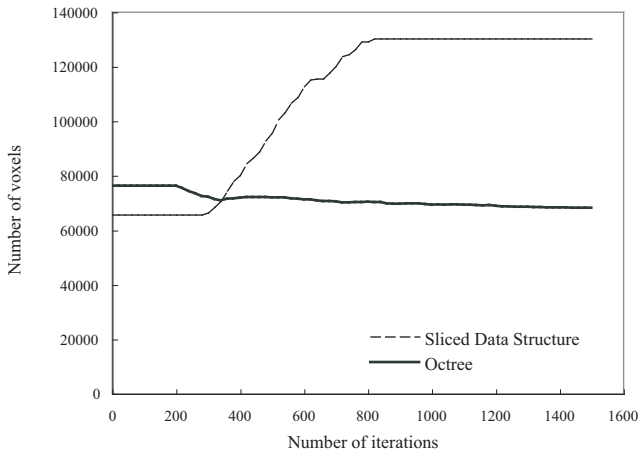


Figure 6: Comparison of the number of voxels.

sampled particle fluids. *ACM Transactions on Graphics 26*, 3.

AMADA, T., IMURA, M., YASUMOTO, Y., YAMABE, Y., AND CHIHARA, K. 2004. Particle-based fluid simulation on gpu. In *2004 ACM Workshop on General-Purpose Computing on Graphics Processors*.

CLIFFORD, E. R. 1992. A fast algorithm for calculating particle interactions in smooth particle hydrodynamics simulation. *Computer Physics Communications 70*, 478–482.

CYRIL, Z. 2005. Cloth simulation on the GPU. In *ACM SIGGRAPH Sketches, No.39*.

FERNANDO, R., AND KILGARD, M. 2003. *The Cg Tutorial*. Addison-Wesley Pearson Education.

GREβ, A., GUTHE, M., AND KLEIN, R. 2006. Gpu-based collision detection for deformable parameterized surfaces. *Computer Graphics Forum 25*, 4, 497–696.

HARADA, T., KOSHIZUKA, S., AND KAWAGUCHI, Y. 2007. Smoothed particle hydrodynamics on GPUs. In *Proc. of Computer Graphics International*, 63–70.

HARADA, T. 2007. *GPU Gems3*. Addison-Wesley Pearson Education, ch. Real-time Rigid Body Simulation on GPUs.

HARRIS, M., COOMBE, G., SCHEUERMANN, T., AND LASTRA, A. 2002. Physically-based visual simulation on graphics hardware. *Proc. of the SIGGRAPH / Eurographics Workshop on Graphics Hardware*, 109–118.

HARRIS, M., BAXTER, W., SCHEUERMANN, T., AND LASTRA, A. 2003. Simulation of cloud dynamics on graphics hardware. In *Proc. of the SIGGRAPH / Eurographics Workshop on Graphics Hardware*, 92–101.

HERNQUIST, L., AND KATZ, N. 1989. Treesph: A unification of sph with the hierarchical tree method. *The Astrophysical Journal Supplement Series 70*, 419–446.

KIPFER, P., SEGAL, M., AND WESTERMANN, R. 2004. Uberflow: A GPU-based particle engine. *Proc. of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 115–122.

KOLB, A., AND CUNTZ, N. 2005. Dynamic particle coupling for GPU-based fluid simulation. In *Proc. of 18th Symposium on Simulation Technique*, 722–727.

KOSHIZUKA, S., AND OKA, Y. 1996. Moving-particle semi-implicit method for fragmentation of incompressible fluid. *Nucl.Sci.Eng. 123*, 421–434.

MISHRA, B. 2003. A review of computer simulation of tumbling mills by the discrete element method: Parti-contact mechanics. *International Journal of Mineral Processing 71*, 1, 73–93.

MONAGHAN, J. 1992. Smoothed particle hydrodynamics. *Annu.Rev.Astrophys. 30*, 543–574.

MONAGHAN, J. 1992. Smoothed particle hydrodynamics. *Annual Review of Astronomy and Astrophysics 30*, 543–574.

MÜLLER, M., CHARYPAR, D., AND GROSS, M. 2003. Particle-based fluid simulation for interactive applications. In *Proc. of SIGGRAPH Symposium on Computer Animation*, 154–159.

OWENS, J. D., LUEBKE, D., GOVINDARAJU, N., HARRIS, M., KRÜGER, J., LEFOHN, A. E., AND PURCELL, T. J. 2005. A survey of general-purpose computation on graphics hardware. *Eurographics 2005, State of the Art Reports*, 21–51.

PREMOZE, S., TASDIZEN, T., BIGLER, J., LEFOHN, A., AND WHITAKER, R. 2003. Particle-based simulation of fluids. *Computer Graphics Forum 22*, 3, 401–410.

PURCELL, T., CAMMARANO, M., JENSEN, H., AND HANRAHAN, P. 2003. Photon mapping on programmable graphics hardware. *Proc. of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 41–50.

SYLVAIN, L., AND HUGES, H. 2006. Perfect spatial hashing. *ACM Transactions on Graphics 25*, 3.

TESCHNER, M., HEIDELBERGER, B., MÜLLER, M., POMERANETS, D., AND GROSS, M. 2003. Optimized spatial hashing for collision detection of deformable objects. In *Proc. of Vision, Modeling, Visualization*, 47–54.

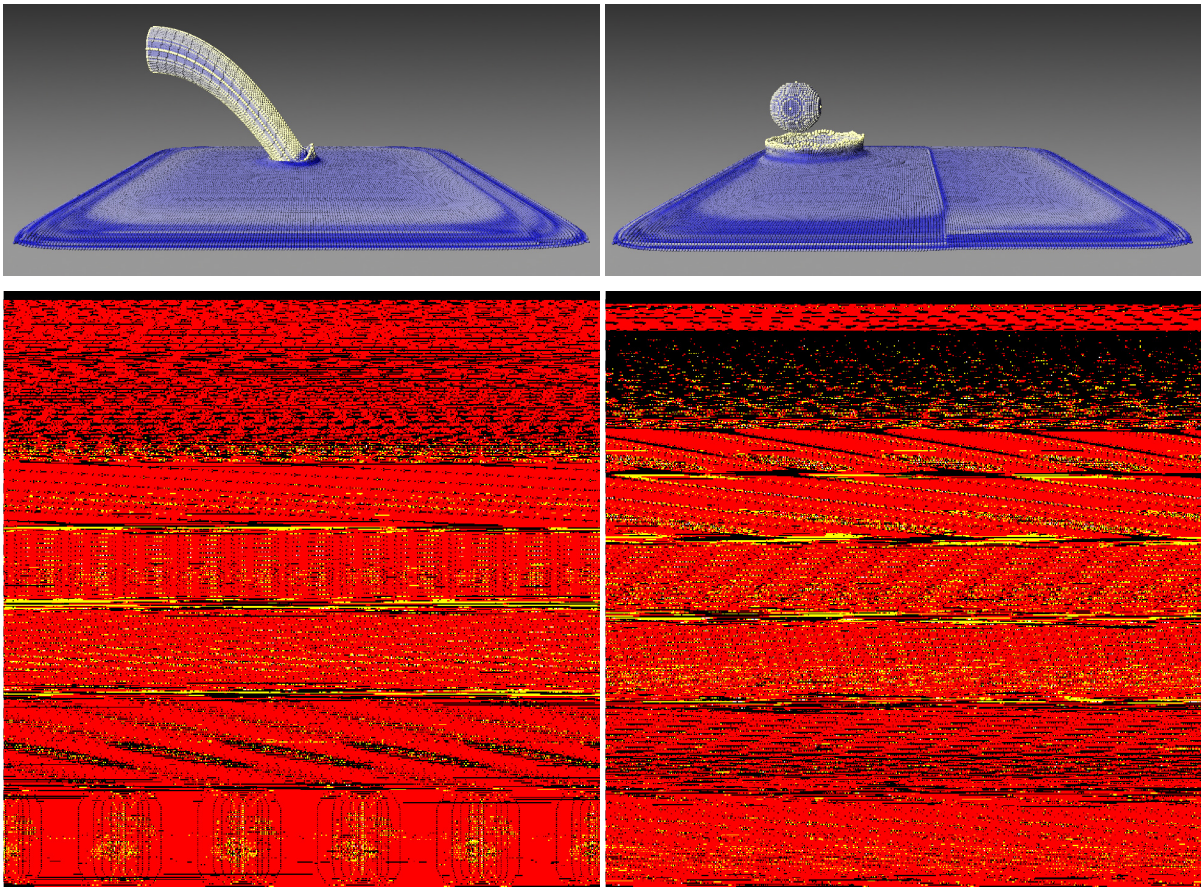


Figure 7: Simulation results and index pool textures. Black texels in the textures indicate empty voxels. Note that 1, 2 and 3 indices are stored in red, yellow and white texels, respectively.

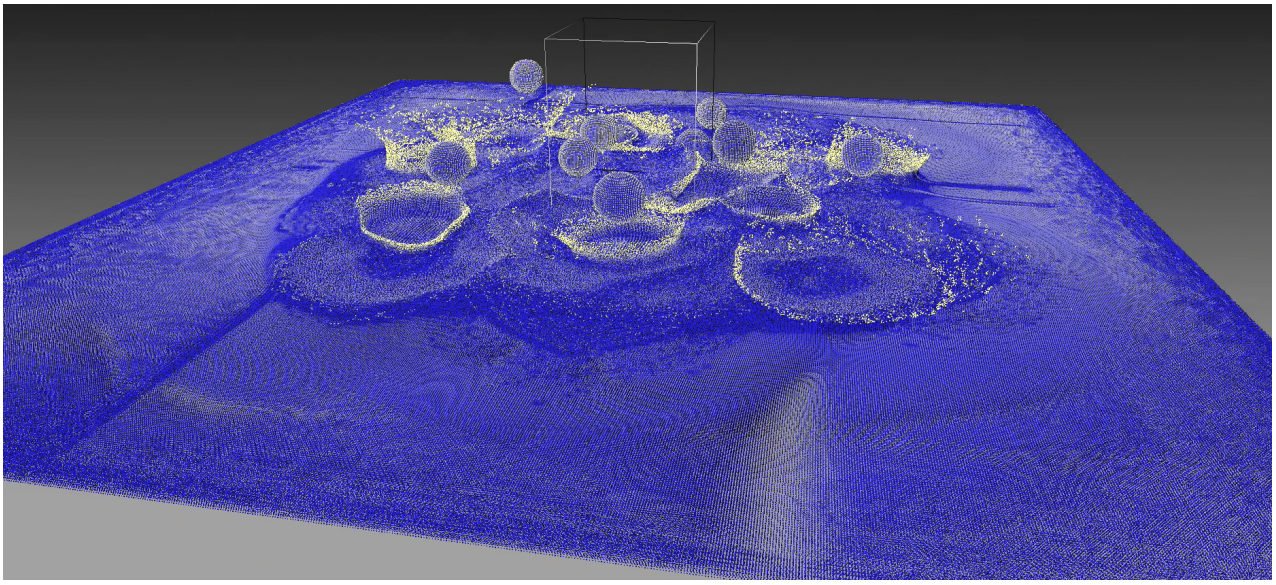


Figure 8: A simulation of a large area.