Vol. 14, No. 2: 45–59

A Hierarchical Hashing Scheme for Nearest Neighbor Search and Broad-Phase Collision Detection

Mickael Pouchol, Alexandre Ahmad, Benoit Crespin and Olivier Terraz
XLIM Research Institute, University of Limoges

Abstract. Increasing computational power allows computer graphics researchers to model spectacular phenomena such as fluids and their interactions with deformable objects and structures. Particle-based (or Lagrangian) fluid and solid simulations are commonly managed separately and mixed together for the collision-detection phase. We present a unified dynamic acceleration model to be used for particle neighborhood queries and broad-phase collision detection, based on a hierarchical hash table data structure. Our method is able to significantly reduce computations in large, empty areas, and thus gives better results than existing acceleration techniques, such as multilevel hashing schemes or KD-trees, in most situations.

1. Introduction

Fluid simulation has been a major focus in the computer graphics community for the last two decades. The difficulty of capturing the complex motion of a fluid encouraged researchers to focus on physically-based simulations. Particle-based (or Lagrangian) simulations are widely used, mainly because particle systems are easy to implement. This type of simulation has proved to give excellent results, such as the well-known smoothed particle hydrodynamics (SPH) method described in [Müller et al. 03] for water simulation.

© A K Peters, Ltd.

1086-7651/09 $0.50 per page

When considering interactions between a set of objects and a fluid modeled by a set of particles, we rely on neighborhood computations which can be divided into in two main cases:

1. For each particle, we must determine a set of neighboring particles when computing the fluid dynamics (a fluid particle acts upon the surrounding fluid particles). This problem is known as *nearest neighbor search* (NNS).
2. We must also check if a particle collides with objects in the scene in order to compute correct fluid–object coupling. This collision detection is a two-phase process: the *broad phase* consists of finding a set of candidate objects; these candidate objects are checked in the narrow phase for collision using exact arithmetic computations. It is therefore critical that the broad phase is implemented efficiently to get the smallest possible set of candidates, and that all the situations where the particle has no chance of colliding with an object can be quickly discarded.

Since particles and objects are in motion, neighborhood information must be updated at every integration step, which is the most computationally expensive part of the whole simulation process. Therefore, we must provide an acceleration method in order to efficiently determine the neighborhood. Particle-based fluid simulations usually rely either on space-partitioning data structures such as KD-trees [Adabala and Manohar 00, Adams et al. 07] or hash tables [Teschner et al. 03, Mirtich 96, Eitz and Lixu 07] that store particles and objects and are dynamically updated at each time step. As stated in [Keiser 06], KD-trees supposedly show better computational performance when dealing with a large number of particles, however hash tables have other advantages— they represent an unbounded, implicit grid which does not limit the spatial extent of the simulation, and they are also very easy to implement.

Our method extends the multilevel hashing scheme described in [Eitz and Lixu 07] in order to accelerate broad-phase collision detections and NNS queries by storing explicit relationships between hash cells and subcells using a *hierarchical* hashing. Since it is capable of efficiently pruning large, empty areas in the scene, our approach combines both the benefits of hash tables and hierarchical structures such as KD-trees. Test results show significant improvements over existing methods, especially for broad-phase collision detection.

2. Background

2.1. Spatially Uniform Hashing for NNS

In particle-based simulations, spatially uniform hash tables are well-adapted for the NNS problem. Since fluid particles usually interact with all particles

lying in a sphere of radius r [Müller et al. 03], this value can be used to efficiently store the particles in a hash table. One of the most efficient hash functions is based on the XOR operation, denoted as $\oplus$:

$$h(\mathbf{p}) = h(p_x, p_y, p_z) = \left(\left(\left(\alpha \left\lfloor \frac{p_x}{r} \right\rfloor\right) \oplus \beta \left\lfloor \frac{p_y}{r} \right\rfloor\right) \oplus \gamma \left\lfloor \frac{p_z}{r} \right\rfloor\right) \mod n, \quad (1)$$

where α, β, and γ are big prime numbers, n is the hash table's length, and p_x, p_y, and p_z are the x-, $y-$, and z-axis components of point $\mathbf{p}$ [Eitz and Lixu 07].

If a particle $\mathbf{p}$ is hashed to a given cell $h(\mathbf{p})$, then neighboring particles lying within radius r are found in the same cell and its 26 neighbors (or eight neighbors in 2D); the hashkeys of these neighboring cells are obtained by $h(p_x + r, p_y + r, p_z + r)$, $h(p_x - r, p_y + r, p_z + r)$, $\ldots$.

2.2. Multilevel Hashing for Broad-Phase Collision Detection

Unfortunately, the use of a unique hash size r is computationally expensive if used simultaneously for NNS and broad-phase collision detection, since fluid particles may collide with objects of size $b \gg r$. Consequently, the computational complexity of a uniform hashing scheme is proportional to $\left(\frac{b}{r}\right)^3$, and very large objects may cover thousands of cells as shown in Figure 1 (left). Depending on the configuration of the 3D scene, this is the bottleneck for simulations that involve fluid–solid interactions.

An efficient and elegant solution presented in [Eitz and Lixu 07] consists of defining multiple hash sizes, so that no object can cover more than eight cells. Each object is hashed using the most appropriate size, as shown in Figure 1 (right); thus the number of hash cells is significantly reduced. Hash sizes are defined as powers of 2, and a cell may contain eight subcells like in a recursive octree structure. Only cells of defined sizes need to be stored,

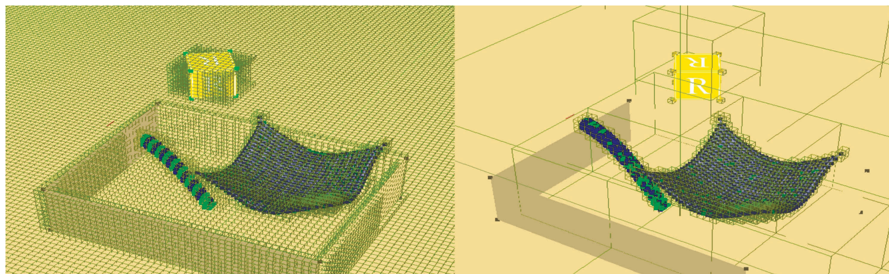


Figure 1. A scene "rasterized" with a unique cell size (left) and using a multilevel model (right).

which reduces memory consumption, especially when there are big differences in object size (e.g., $2^{-2}, 2^4, 2^{15}$).

The hash size corresponding to an object must be at least the size of the longest edge l of its axis-aligned bounding box (AABB). This size s is computed by converting l into the nearest but greater power of 2:

$$s = 2^{\lceil \log_2(l) \rceil}. \quad (2)$$

This computation is performed for each object during a preprocessing step at the beginning of the simulation, yielding a set of different *classes* associated with different sizes. We consider in the following that a given class can be identified either by its number c or its size $s = 2^c$.

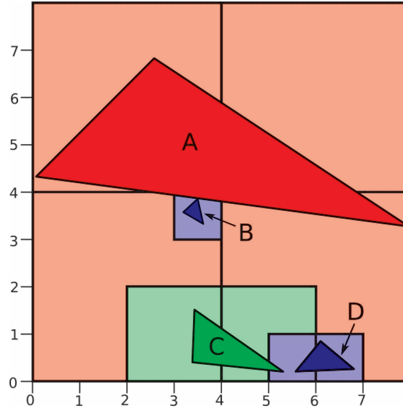
This hashing scheme can handle deformable objects, i.e., those whose size can change through time. At each integration step, if the class c corresponding to an object was not computed in the preprocessing step, then c is given by the nearest greater class. The object is thus hashed in the smallest possible class by considering the eight endpoints of its AABB: for each point $\mathbf{p}$, the object's ID is added to the hash table at the position given by $h^c(\mathbf{p})$, defined by extending Equation (1):

$$h^c(\mathbf{p}) = h^c(p_x, p_y, p_z) = \left(\left(\left(\alpha \left\lfloor \frac{p_x}{s} \right\rfloor \right) \oplus \beta \left\lfloor \frac{p_y}{s} \right\rfloor \right) \oplus \gamma \left\lfloor \frac{p_z}{s} \right\rfloor \right) \mod n. \quad (3)$$

An example of the hash structure obtained with this method is presented in Figure 2 (left). For the sake of clarity, we use left-bottom coordinates to identify cells, and hashkeys are computed without modulo operations (i.e., $n = +\infty$). A unique hashtable, or alternatively one hashtable per class c, is sufficient to hash objects with minimal *hash collisions* (which should not be confused with collision detection), i.e., distant objects will unlikely have the same hashkey, although this also depends on the hashtable's length n.

Using this multilevel hashing scheme, the broad-phase collision detection between a given object and the objects stored in the hash structure consists of considering each class and checking whether one of its cells overlaps the object's AABB; if positive, IDs stored in this cell are retrieved for the narrow phase. Since a reduced number of cells has to be checked, this process is far more efficient than if a unique cell size is used.

As noted in [Eitz and Lixu 07], another advantage to this method is the ease of implementation, since the multilevel hashing scheme adapts itself to any scene setup, including moving and deforming objects over time as shown in Figure 3. However, if there are no constraints on the deformations of the objects, an AABB can be larger than the largest class: in that case, it must be subdivided and hashed to several cells.



Multilevel Hashing

Hashkey	Objects
$h^2(0,0)$	A
$h^1(2,0)$	C
$h^2(4,0)$	A
$h^2(0,4)$	A
$h^1(4,0)$	C
$h^2(4,4)$	A
$h^0(3,3)$	B
$h^0(5,0)$	D
$h^0(6,0)$	D

Hierarchical Hashing

Hashkey	Objects	hasChildren
$\tilde{h}^2(0,0)$	A	true
$\tilde{h}^1(2,0)$	C	false
$\tilde{h}^2(4,0)$	A	true
$\tilde{h}^2(0,4)$	A	false
$\tilde{h}^1(2,2)$		true
$\tilde{h}^2(4,4)$	A	false
$\tilde{h}^1(4,0)$	C	true
$\tilde{h}^1(6,0)$		true
$\tilde{h}^0(3,3)$	B	
$\tilde{h}^0(5,0)$	D	
$\tilde{h}^0(6,0)$	D	

Figure 2. Top: A 2D example where objects with different sizes are hashed to corresponding classes 0, 1, or 2 (with sizes 1, 2, and 4, respectively). Bounding boxes are not represented. Bottom left: Non-empty hash cells obtained with multilevel hashing [Eitz and Lixu 07], using Equation (3); Bottom right: Non-empty hash cells obtained with our method using Equation (4).

3. Hierarchical Hashing

3.1. Top-Down Hashing

We extend the hash function defined in Equation (3) with a hierarchical scheme: an element (object or particle) calls this hash function from the

largest class k to the class c corresponding to its size (top-down), i.e., from $[k \ldots c]$, where $c \in [0, k]$:

$$\tilde{h}^c(\mathbf{p}) = \left(h^k(\mathbf{p}) + h^{k-1}(\mathbf{p}) + \cdots + h^c(\mathbf{p})\right) \mod n. \tag{4}$$

Then, the element may be inserted in the cell given by the key $\tilde{h}^c$. A Boolean flag is also stored in the hash table to indicate whether the cell has at least one child, i.e., if a cell exists in a smaller class that overlaps $\tilde{h}^c$ and contains at least one element. Figure 2 (right) shows the hash structure obtained with our method compared to the multilevel hashing scheme of [Eitz and Lixu 07].

Our hash structure is built upon two main data structures:

- a hashmap `hm`: for each hash key, it stores overlapping objects' IDs and a Boolean `hasChildren`

- a set `classSizes`, initialized in a preprocessing step as in Section 2.2 and sorted in decreasing order. The minimal size stored in `classSizes` should be the value r of the interaction radius for fluid particles. Since fluid simulations usually require that this value is the smallest possible, it is safe to assume that no object in the scene will be smaller than r.

Although multiple calls to the hash function h are necessary, the additive hash function described by Equation (4) helps to prevent useless future and expensive computations by reducing hash collisions. Consider two 3D points $\mathbf{p}_1, \mathbf{p}_2$ that are not in the same $(c+1)$-class spatial cell, i.e., $h^{c+1}(\mathbf{p}_1) \neq h^{c+1}(\mathbf{p}_2)$. Then suppose that a hash collision occurs when hashing these points into class c, i.e., $h^c(\mathbf{p}_1) = h^c(\mathbf{p}_2)$. This hash collision is avoided using Equation (4), since $h^c(\mathbf{p}_1)+h^{c+1}(\mathbf{p}_1) \neq h^c(\mathbf{p}_2)+h^{c+1}(\mathbf{p}_2)$. The offset must be the same for all the points coming from the same parent class, which is true in our case because we make use of hierarchical coherence. Hash collisions may still happen because nothing guarantees that the new cell is free; nonetheless, this hashing scheme will obviously give better results than a unique hashing. We point out that increasing the hash table size also reduces the probability of hash collisions.

3.2. Particles and Objects Insertions

We consider in Listing 1 the insertion of a single point `p` in the structure, which belongs to an element referenced by its `id`, into a given class `size`. The hash function `h` is implemented as in Equation (3).

```
void insert(point p, int id, int size) {
 int hashKey = 0;
 for each (int currSize in classSizes) {
  hashKey += h(p, currSize);
  if (size == currSize) {
   hm[hashKey].insert(id); break;
  }
  else if (! hm[hashKey].hasChildren)
   hm[hashKey].hasChildren = true;
 }
}
```

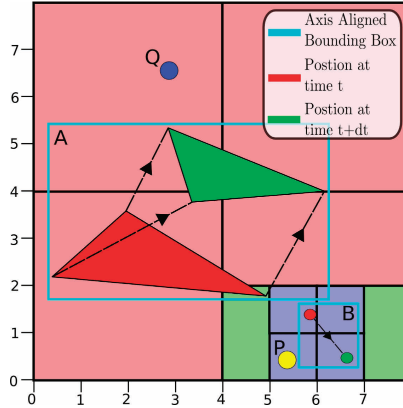
Listing 1. Top-down point hashing.

The insertion of a particle is straightforward: we simply use its position and ID as input values and the interaction radius r as the hash size (see Listing 1). Since r is also the minimal size in `classSizes`, each of its elements will be considered. The insertion of a larger object is also rather simple. First, we compute its AABB and determine the hash size s in which it should be hashed as in Section 2.2. Then, top-down hashing is achieved for each of the eight endpoints of the AABB, with the object's `id` and `size` s as input values.

This approach can be used to insert either static or moving elements. Assuming linear motion during one integration step, we simply consider the AABB of its entire path—e.g., the path of a moving triangle is a prism, as shown in Figure 3—and insert the object as shown in Listing 1. In this case, the top-down hashing stops as soon as the size corresponding to the AABB is met; since we assume in Listing 1 that `size` belongs to the precomputed `classSizes` set, it may be necessary to choose the nearest greater size as in Section 2.2. Unlike some other approaches [Kondoh et al. 04], this method is capable of inserting moving elements independently of the distance covered during a time step.

3.3. Hash Structure Queries

Any query in the hash structure will take advantage of the top-down hashing of static or moving objects as previously described. As shown in Figure 3 (right), finding possible candidates that may collide or interact with a single point consists of iteratively hashing the point for each class, starting from the largest class, and retrieving the IDs stored in the corresponding cell as candidates. The process ends if the cell has no child or if the smallest class is reached. Unfortunately, the problem is more challenging for both broad-phase collision detection and NNS. This simple method is thus extended below to handle more complex queries.



Hashkey	Objects	`hasChildren`
$\tilde{h}^2(0,0)$	A	false
$\tilde{h}^2(4,0)$	A	true
$\tilde{h}^2(0,4)$	A	false
$\tilde{h}^2(4,4)$	A	false
$\tilde{h}^1(4,0)$		true
$\tilde{h}^1(6,0)$		true
$\tilde{h}^0(5,1)$	B	
$\tilde{h}^0(5,0)$	B	
$\tilde{h}^0(6,1)$	B	
$\tilde{h}^0(6,0)$	B	

Point P
$\downarrow$
$\tilde{h}^2(4,0)$: retrieve object A
$\downarrow$
$\tilde{h}^1(4,0)$
$\downarrow$
$\tilde{h}^0(5,O)$: retrieve object B

Point Q
$\downarrow$
$\tilde{h}^2(0,4)$: retrieve object A

Figure 3. Top: Hashing moving objects. Bottom left: Corresponding, non-empty hash cells obtained with our method; Bottom right: Top-down broad-phase collision detection for points P and Q (marked as colored circles)

3.3.1. Broad-Phase Collision Detection

We are interested in checking if a given AABB overlaps any cell in the hash structure that may contain possible candidates for the narrow-phase collision detection. The broad-phase algorithm is described in Listing 2. In the context of fluid–solid interactions, the AABB encloses the path of a particle. In a more general context, it corresponds to a static or moving object that may collide with other objects in the scene as in Figure 3.

The main advantage of our method is the use of the Boolean `hasChildren`, which quickly ends the process if negative by emptying the `points` set. Since we deal with an AABB `a` of arbitrary size, we may have to call `a.subdivide(t)`

```
set<int> broadPhase(AABB a) {
 set<int> candidateObjects;
 // Build a set with the 8 endpoints
 // (the initial hash key for these points is set to 0)
 vector<point> points = a.getEndPoints();

 for each (int currSize in classSizes) {
  int delta = a.size() - currSize;
  // Subdivide AABB if necessary
  if (delta > 0) points.insert(a.subdivide(delta));
  // Find possible candidates in corresponding cells
  for each (point p in points) {
   p.hashKey += h(p, currSize);
   candidateObjects.insert(hm[p.hashKey].retrieveIds());
   // Remove this point if the cell has no child
   if (! hm[p.hashKey].hasChildren) points.remove(p);
  }
 }
 return candidateObjects;
}
```

Listing 2. Get all the objects that may collide with a given AABB.

to subdivide the box t times as in an octree decomposition, in order to ensure that we do not miss hash cells smaller than `a.size()`. A more complex implementation (not presented here) could also optimize the process by ensuring that a value `p.hashKey` is not processed multiple times, which happens if `a.size()` is very small since several endpoints can be hashed in the same cell.

3.3.2. Nearest Neighbor Search

As stated in Section 2.1, if a particle $\mathbf{p}$ is hashed to a given cell, then neighboring particles lying within radius r are simply found in the same cell and its 26 neighbors. The hash keys of these neighboring cells are simply given by $\tilde{h}^r(p_x + r, p_y + r, p_z + r)$, $\tilde{h}^r(p_x - r, p_y + r, p_z + r)$, ...

Unfortunately, since r is the smallest class, this simple approach does not perform well with our method because computing neighboring hash keys requires us to consider all classes stored in `classSizes`. We thus propose to determine neighboring particles directly when a particle is inserted. We add a new attribute to the hash structure called `neighbors` that stores, for a given particle, the references of its neighbors. Then each particle is processed using the modified insertion procedure, which makes use of the interaction radius r

```
void insert(particle P) {
 set<int> candidateParticles;
 // Build a set with P and 26 neighboring points
 // (the initial hash key for these points is set to 0)
 vector<point> points = get27Points(P.position);

 for each (int currSize in classSizes)
  // Retrieve possible candidates if size r is reached
  for each (point p in points) {
   p.hashKey += h(p, currSize);
   if (currSize == r)
    candidateParticles.insert(hm[p.hashKey].retrieveIds());
   if (! hm[p.hashKey].hasChildren) points.remove(p);
  }
 // Update the 'neighbors' attribute if the distance
 // between particles is lower than r
 for each (int id in candidateParticles)
  if (distance(P.id, id) <= r) {
   neighbors[P.id].insert(id);
   neighbors[id].insert(P.id);
  }
 // Insert the particle (finally !)
 insert(P.position, P.id, r);
 }
}
```

Listing 3. Particle insertion and NNS computation.

and of a method `get27Points(p)` that produces the set p, (p_x+r, p_y+r, p_z+r), $(p_x - r, p_y + r, p_z + r)$, $\ldots$ (see Listing 3).

This on-the-fly NNS computation gives better results than the uniform hash scheme described in Section 2.1, again thanks to the Boolean flag `hasChildren` that discards empty areas. When the whole set of particles has been processed, `neighbors[id]` stores the neighbors of a particle `id` which can be retrieved for fluid dynamics computations.

4. Results and Comparisons

The results presented below are performance comparisons of our method with several other acceleration structures on random distributions, conducted on a single i686 processor running at 2.66 GHz, in order to validate our model on typical particle-based simulations.

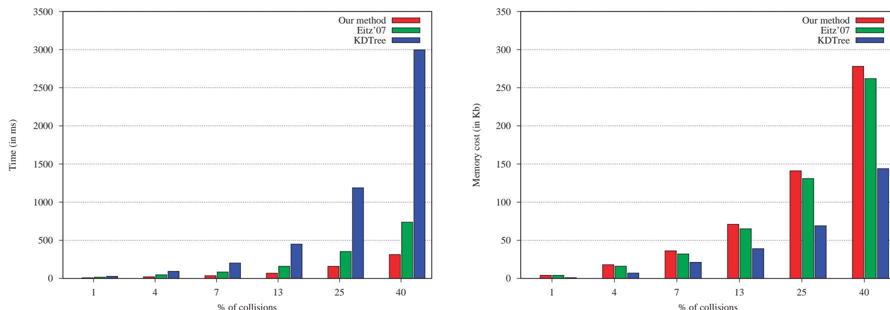


Figure 4. Left: Timing comparisons for broad-phase collision detection. Right: Memory consumption comparisons.

4.1. Comparisons on 2D Random Distributions

Figure 4 shows the result of broad-phase collision detection tests, using a KD-tree, the multilevel hashing scheme of [Eitz and Lixu 07], and our method. A distribution of N objects is generated randomly in a 2D square of size 2^{13} and inserted in the acceleration structure using each object's AABB. Then, a distribution of 1M segments is computed, which accounts for moving particles in the scene; for each segment we query the structure to determine which objects it may intersect. The objects' size distribution is implemented in a way that approximately 10% of the objects have an average size of 2^{10}, 25% have an average size of 2^6, and the remaining objects have an average size of 2^3, whereas the length of each particle's motion vector does not exceed 2^4. The computation times are depicted as a function of the ratio of particles that collide with at least one of N objects; this ratio increases according to N. Our method improves the timings obtained by Eitz and Lixu's method by at least a factor of two in all situations, whereas the memory overhead induced by our hierarchical structure does not exceed 10%. Moreover, our tests indicate that hash-based methods, in general, seem a better choice for broad-phase collision detection over KD-trees, although more memory-consuming.

We also evaluate the computational cost of our model for NNS, compared to a KD-tree, and to a uniform hashing scheme.[1] A distribution of 1M points is generated randomly in a 2D square of size 2^{20} and inserted in the acceleration structure; then, for each point we query its neighbors located in a circle of radius r. The computation times are depicted in Figure 5 (left) as a function of the maximal number of neighbors lying within an increasing radius r. We note that our model is competitive over uniform hashing when particles are far from each other, i.e., when the maximal neighbor count is below five. However, the

[1] In that case, a uniform hashing scheme [Teschner et al. 03] can be considered as a restriction of Eitz and Lixu's hashing scheme with only one level.

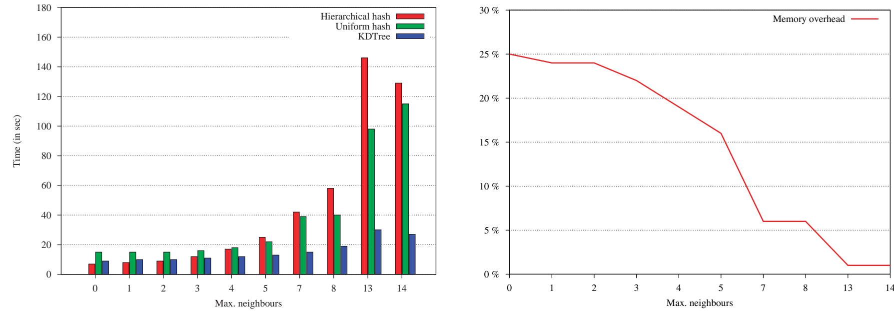


Figure 5. Left: Timing comparisons for nearest neighbor search. Right: Memory overhead induced by our method.

memory overhead in that case ranges from 10% to 25%. For more dense points distributions, the KD-tree exhibits the best performances, partly because the objects to be inserted all have the same size in the NNS case. Implementations of particle-based simulations that involve high neighbor count should thus use KD-tree acceleration instead of hash tables for NNS.

4.2. Comparisons on a 3D Fluid Simulation

Our 3D test scene is presented in Figure 6 (left). A particle-based fluid (in blue) falls and bounces over a set of 2000 static, solid spheres (in red), and

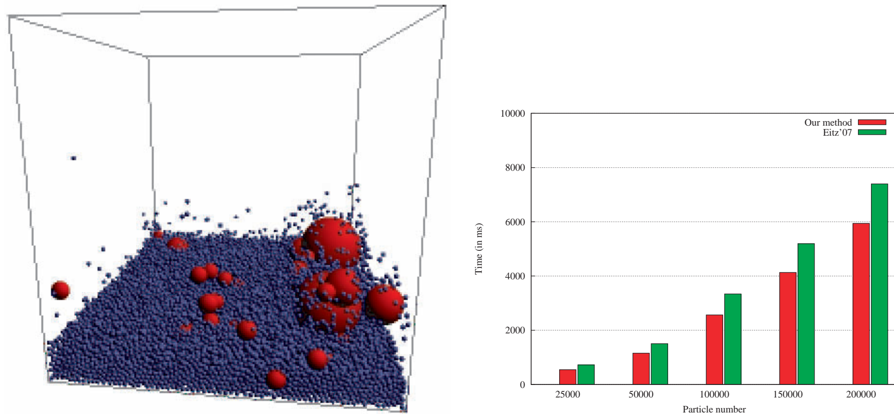


Figure 6. Left: Our 3D test scene. Right: Performance comparisons for collision detection (in millisecond per simulation frame)

eventually stabilizes around its equilibrium position at the bottom of the simulation box. We have chosen a particle-based model with simple collision response for the fluid simulation, which was conducted over 1000 successive iterations, but other Lagrangian models such as SPH could be used [Müller et al. 03]. Dynamic particles and static spheres are initially located randomly in the box; static spheres' sizes range from 2^{-4} to 2^{-2}, whereas particles are 2^{-6} in size.

Timings are given as the time required to compute the whole collision detection process per frame: hashing objects (although spheres and walls are static in this example), hashing moving particles, retrieving possible candidates for each particle, and finally computing exact collisions. A uniform hashing scheme was used for nearest neighbor search. As shown in Figure 6 (right), our method outperforms Eitz and Lixu's by approximately 25%; memory measurements (not presented here) showed that memory overhead needed to store hierarchical relationships never exceeds 20%, as in the 2D case. One can note that, due to the progressive aggregation of particles at the bottom of the box, the number of isolated particles tends to decrease over time. This explains why the acceleration achieved by our method is not as spectacular as in the 2D case, but remains significant nonetheless.

4.3. Discussion and Improvements

The 2D tests presented in this paper were implemented using the STL's `hash_multimap` container [Musser et al. 01]. This simple implementation avoids using a fixed length for our hash structure, which means we can discard the modulo operation from Equations (1) and (4) and consider $n = +\infty$. Although this comes at the price of logarithmic time complexity to find objects associated with a given key, it also demonstrates that, without optimizations, our data structure already provides nice results even for 1M particles.

A more optimized implementation was used for our 3D experiments, based on a fixed-length vector. Again, satisfying results were obtained compared to Eitz and Lixu's approach, but in that case a specific value for n must be chosen. This problem has a significant impact on memory consumption if n is too high; on the other hand, a lower value may imply more hash collisions, which in turn increase the number of candidate objects obtained after the broad phase. Our experiments showed that the overhead computations induced by these "false" candidates is negligible if n is set to the approximate number of polygons in the scene multiplied by 100.

In the specific case of dynamic simulations, we could make use of a timestamp, i.e., a parameter that distinguishes two time steps [Teschner et al. 03]. If a new object is to be inserted in a hash cell and the timestamps of the cell and the object differ, then the object does not necessarily need a new

memory allocation to be stored, since it can reuse the memory allocated for another object at the previous time step. The timestamp is thus used to reduce overall memory allocation when hash-list insertions are achieved. By using this scheme, the hash table is never entirely swept between successive time steps. In STL-based implementations this approach is readily obtained since the `clear` operation does not necessarily free its memory when called at each time step. Performances also depend on simulation parameters such as the time step: the larger the time step is, the bigger the distance an element will cover and consequently a longer distance will have to be hashed.

One can point out that the complexity of our hash function $\tilde{h}^c$ is proportional to the number of different classes, whereas the multilevel hashing scheme requires a single call to the hash function h^c. Nevertheless, it is not necessary to check all different classes to find candidate objects, thanks to the Boolean flag `hasChildren` that indicates whether smaller objects could be found in a given area; this explains the good performances of our method, especially for collision detection.

Acknowledgments. The authors thank C. Rousselle for her corrections and the reviewers who suggested numerous improvements.

References

[Adabala and Manohar 00] Neeharika Adabala and Swami Manohar. "Modeling and Rendering of Gaseous Phenomena using Particle Maps." *Journal of Visualization and Computer Animation* 11:5 (2000), 279–293.

[Adams et al. 07] Bart Adams, Mark Pauly, Richard Keiser, and Leonidas J. Guibas. "Adaptively Sampled Particle Fluids." In *Transactions on Graphics: Proc. SIGGRAPH 2007* 26:3 (2007), 26:3, (2007), Article 48.

[Eitz and Lixu 07] Mathias Eitz and Gu Lixu. "Hierarchical Spatial Hashing for Real-Time Collision Detection." In *SMI '07: Proceedings of the IEEE International Conference on Shape Modeling and Applications 2007*, pp. 61–70. Washington, DC: IEEE Computer Society, 2007.

[Keiser 06] Richard Keiser. "Meshless Lagrangian Methods for Physics-Based Animations of Solids and Fluids." Ph.D. thesis, ETH Zurich, 2006.

[Kondoh et al. 04] Nobuhiro Kondoh, Shuuji Sasagawa, and Atsushi Kunimatsu. "Creating Animations of Fluids and Cloth with Moving Characters." In *SIGGRAPH '04: ACM SIGGRAPH 2004 Sketches*, p. 136. New York: ACM Press, 2004.

[Mirtich 96] Brian Vincent Mirtich. "Impulse-Based Dynamic Simulation of Rigid Body Systems." Ph.D. thesis, University of California, Berkeley, 1996.

[Müller et al. 03] Matthias Müller, David Charypar, and Markus Gross. "Particle-Based Fluid Simulation for Interactive Applications." In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pp. 154–159. Aire-la-Ville, Switzerland: Eurographics Association, 2003.

[Musser et al. 01] David R. Musser, Gilmer J. Derge, and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*, second edition. Reading, MA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[Teschner et al. 03] Matthias Teschner, Bruno Heidelberger, Matthias Müller, Danat Pomerantes, and Markus H. Gross. "Optimized Spatial Hashing for Collision Detection of Deformable Objects." In *Proceedings of the Vision, Modeling, and Visualization Conference 2003*, pp. 47–54. Heidelberg, Germany: Akademische VerlagsgesellschaftAKA GmbH, 2003.

Web Information:

http://jgt.akpeters.com/papers/Poucholetal09/

Mickael Pouchol, Laboratoire XLIM (UMR CNRS 6172), University of Limoges,
83 rue d'Isle, 87000 Limoges, France (mickael.pouchol@xlim.fr)

Alexandre Ahmad, Laboratoire XLIM (UMR CNRS 6172), University of Limoges,
83 rue d'Isle, 87000 Limoges, France (alexandreahmad@free.fr)

Benoit Crespin, Laboratoire XLIM (UMR CNRS 6172), University of Limoges,
83 rue d'Isle, 87000 Limoges, France (benoit.crespin@xlim.fr)

Olivier Terraz, Laboratoire XLIM (UMR CNRS 6172), University of Limoges,
83 rue d'Isle, 87000 Limoges, France (olivier.terraz@xlim.fr)

Received May 25, 2008; accepted in revised form October 2, 2009.