Particle-Based Fluid Simulation on the GPU

Conference Paper · May 2006

DOI: 10.1007/11758549_35 · Source: DBLP

CITATIONS
38

READS
1,082

3 authors, including:

Nathan A. Carr
Adobe Systems Inc
78 PUBLICATIONS **1,630** CITATIONS

SEE PROFILE

Gavin S. P. Miller
Adobe Systems Inc
47 PUBLICATIONS **2,445** CITATIONS

SEE PROFILE

Particle-Based Fluid Simulation on the GPU

Kyle Hegeman[1], Nathan A. Carr[2], and Gavin S.P. Miller[2]

[1] Stony Brook University
kyhegem@cs.sunysb.edu
[2] Adobe Systems Inc.
{ncarr, gmiller}@adobe.com

Abstract. Large scale particle-based fluid simulation is important to both the scientific and computer graphics communities. In this paper, we explore the effectiveness of implementing smoothed particle hydrodynamics on the streaming architecture of a GPU. A dynamic quadtree structure is proposed to accelerate the computation of inter-particle forces. Our method readily extends to higher dimensions without undue increase in memory or computation costs. We show that a GPU implementation runs nearly an order of magnitude faster than our CPU version for large problem sizes.

1 Introduction

In computer graphics, particles are a popular primitive for the simulation and rendering of numerous effects including cloth, water, steam, smoke, and fire [1, 2, 3]. Producing high quality liquid animations can require hundreds of thousands of particles and take several minutes to compute a single frame [4]. This requirement to scale to large problem sizes has led us to investigate the use of graphics hardware to accelerate the computations. A challenging aspect of this problem is determining all the neighbors within a radius of a given particle, a common operation during the evaluation of inter-particle forces. For CPU based implementations, a simple and effective approach is to place particles in buckets on a uniform grid. Each cell in the grid is the size of the neighborhood radius. To find neighbors for a given particle, only particles within its bucket and buckets of neighboring grid cells need to be considered. Unfortunately, on current GPU architectures it is difficult to maintain a dynamic list of particles. In this paper, we propose an alternative method that uses a hierarchical tree structure to accelerate these queries. We describe algorithms for constructing, maintaining, and evaluating a quadtree data structure in graphics hardware. These quadtree data structures are efficient to traverse and recompute as the particle configuration evolves over time.

2 Previous Work

Recent advances in technology have enabled a host of new applications to profit from the floating point power and bandwidth available in graphics hardware [5]. Several papers [6, 7] have demonstrated that particle systems can be effectively mapped to GPUs. Chiara et al. [8] implement a flocking behavior model with much of the computation being performed on the GPU, the CPU is used to compute neighbor lists for particles. Rather than finding neighbors explicitly, Kolb and Cuntz [9] implement a particle based

V.N. Alexandrov et al. (Eds.): ICCS 2006, Part IV, LNCS 3994, pp. 228–235, 2006.

fluid model model by splatting kernels in image space. This method is less accurate than our proposed approach since it involves discretizing kernels onto a grid.

The use of tree data structures such as kd-trees, octrees, and hierarchical bounding volume (HBV) trees in graphics hardware is still relatively new and has been employed for GPU ray-tracing [10, 11], as well as 3D paint [12, 13]. Tree based schemes in graphics hardware are severely limited due to the lack of recursion and register indexing within fragment programs. For this reason image based algorithms have been used on the GPU to accelerate collision detection [14, 15, 16]. Tree based algorithms are a popular means to perform collision detection on the CPU [17, 18]. Existing work with HBV trees on GPUs has focused on efficient tree evaluation, but has relegated tree construction and maintenance to the CPU. We have chosen a fixed topology HBV tree for our implementation. This structure has the nice property that it can be traversed and maintained on the graphics card. We can extend this structure to handle particle-surface interaction by adapting the CPU geometry image collision detection approach of [19] to run entirely in the graphics hardware. Our work is most closely aligned to that of Simonsen et al. [11] who noted that HBV trees map very well onto graphics hardware.

3 Dynamic Quadtrees on the GPU

In this section we describe our algorithms for building and traversing an HBV quadtree on the GPU. A key contribution of this paper is the method for constructing and maintaining the tree in a dynamic simulation. Previous work on GPU acceleration structures rely on the CPU to perform this construction; only the traversal stage is implemented on the GPU.

3.1 Quadtree Storage

Particles are arranged in a 2D array with a resolution of $N \times N$. These particles form the leaves of a quadtree. One level higher in the tree there are $\frac{N}{2}\text{x}\frac{N}{2}$ nodes. Each node has 4 children. The next level is $\frac{N}{4}\text{x}\frac{N}{4}$, etc. until the root of the tree is reached. These levels of the tree form a pyramid. A key observation is that each level of the tree can be computed as a simple reduction operation using data from a lower level. This operation finds the smallest sphere that contains the 4 child spheres. Computation of such reduction pyramids is a common step in graphics algorithms, for example in texture mipmap construction.

Given a tree with fixed depth we can precompute the order of traversal. At each node, we store two links. One link descends deeper into the tree if there is a collision, this is called a “hit” link. The other link is followed if there is no collision, a “miss link”. The logical link structure is shown in Figure 1. In memory, the links are stored in 2D arrays that mirror the structure of the tree. There are special links to denote leaf nodes and the end of traversal.

3.2 Traversal

The precomputed traversal links simplify the algorithm for traversing the tree. Traversal of the tree is performed in a top down manner. Starting from the root of the tree, an

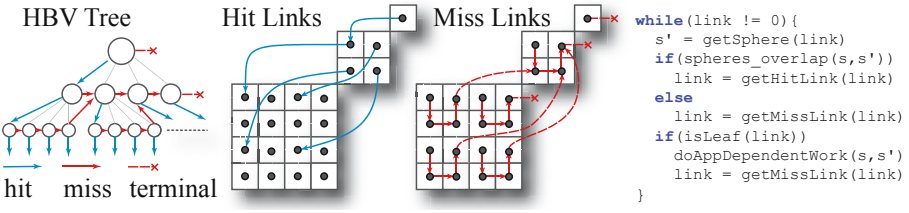


Fig. 1. Quadtree sphere bounding volume hierarchy with link traversal(*left*), hit links stored on the GPU (*middle*), and miss links stored on the GPU(*right*). The algorithm for GPU link hierarchy traversal is provided(*far right*).

overlap test is performed between the bounding volume being tested and the bounding volume of the node. Based on the result of the overlap test, we follow one of two precomputed links. If there is overlap the hit link is followed, otherwise the miss link. A link location can be used to determine whether the traversal has hit a leaf node. When this occurs, an application dependent processing step is performed. Traversal is performed until a null *terminal* link is reached, denoting that all overlapping leaf nodes have been visited. Pseudo code for the traversal loop is given in Figure 1(far right).

3.3 Quadtree Optimization

The HBV quadtree is rebuilt each time particle positions are updated in our algorithm. This rebuilding process does not change the tree's topology, however, to construct an efficient tree we require that the particles are stored in a spatially coherent manner. If this is not the case, the reduction algorithm computes large bounding volumes resulting in inefficient traversal. Thus, tree quality only affects performance and not the correctness of the computation. It is easy to initialize the system such that particles are spatially coherent, however, particles are constantly moving in a dynamic system. As shown in Section 5, the degradation in spatial coherency for a particle based liquid simulation is slow. We take advantage of this fact by lazily reoptimizing every α seconds, where α is a user tunable parameter. We can do this optimization asynchronously. For example, particle locations can be read back to the CPU at time t, meanwhile, the GPU continues the simulation process. Once the CPU has completed the tree optimization at some time $t + \Delta t$, the new updated information can be transmitted to the card. In this manner the tree optimized by the CPU is slightly behind the simulation. Nonetheless, this approach results in a tight bounding volume tree when Δt is small.

A top-down recursive bisection algorithm is used to compute an optimized particle configuration. We start by choosing a splitting axis along which to partition our particles into two equal sized sets. A partitioning plane orthogonal to this axis is found such that half the particles fall on either side of the plane. This partitioning process is applied to each of these two sets, resulting in four equal sized sets that form the second level of our quad-tree. By iteratively applying this bisection process, all levels of the tree are built. The choice of splitting axis plays an important role in the bisection process. Our CPU implementation uses principal component analysis (PCA) to determine an axis of greatest variation. Our GPU implementation uses a lower quality approach of choosing

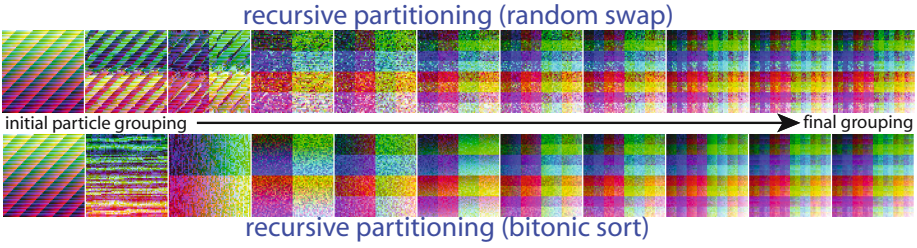


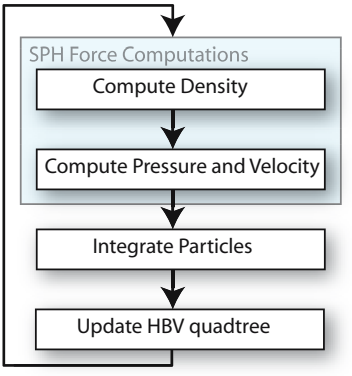
Fig. 2. GPU algorithm for spatial re-ordering of particles to improve tightness of hierarchical bounding tree. Image containing unsorted particles (left), is recursively partitioned and either sorting (bottom) or random swapping(top) is used to place particles into tighter spatial clusters.

axis aligned splitting planes. We alternate between the principal coordinate axes $x, y\,[, z]$. Efficiently re-ordering n particles across a chosen splitting plane requires computation of the ith order statistic which can be done in $O(n)$ time. A less effective choice is to use sorting. To accomplish this task on the GPU, however, we must resort to parallel algorithms such as bitonic sort which has been shown to run in $O(\lg^2 n)$ [20] on graphics hardware. In practice, however, we can use an approximate method and achieve similar results. For this we propose a random swapping algorithm.

Our random swapping algorithm also utilizes recursive bisection to form a complete tree. We start by partitioning our particle image vertically into two rectangular regions. These two regions are partitioned horizontally, forming four square subregions. The process of partitioning subregions vertically and then horizontally is recursively applied until every subregion covers a single particle. Each time we partition a sub-region in half we select a coordinate value (e.g. $x, y[, z]$) to compare against. A random offset is generated and sent to a fragment shader. This shader uses the offset and modula arithmetic to swap particles between partitions based on the magnitude of the selected coordinate value. For example, during the first partitioning we divide the particle image vertically. Particles in the lower partition are compared along the random offset to their corresponding particle in the upper partition. Particles with greater x values are swapped to the top partition(Figure 2). This swapping process between partitions can be iterated a number of times to increase the likelihood that particles are sorted into the correct partition. In practice we have found that applying $\lg(w \times h)$ passes of swapping, where w and h are the width and height of the subregion, produces satisfactory results. Figure 2 shows a comparison of random swapping versus recursively performing GPU bitonic sorting. Note that the random swap algorithm produces nearly the same final particle clustering. By increasing the number of random swaps performed at each level we approach the quality of the tree provided by the more expensive recursive bitonic sorting algorithm. In this way, we can trade off the cost of particle reordering against the quality of the resulting bounding volume tree.

4 Smoothed Particle Hydrodynamics

Smoothed Particle Hydrodynamics (SPH), was developed for use in astrophysics [21, 22]. Müeller et al. [2] use SPH to simulate liquids in interactive computer graphics




```
// compute density for particle_i
// with neighbors particle_j
density_i = 0;
while(link != 0){
  s' = getSphere(link)
  if(spheres_overlap(s,s'))
    link = getHitLink(link)
  else
    link = getMissLink(link)
  if(isLeaf(link))
    particle_j = getParticle(link)
    dist=|particle_i-particle_j|
    density_i += Wpoly6(dist,h)
    link = getMissLink(link)
}
density_i *= ParticleMass
```

Fig. 3. High level overview of the SPH computation(*left*) and pseudo code for computing density utilizing the quadtree(*right*)

applications. In this section, we describe how we use our quadtree implementation to accelerate the SPH model of Müeller et al. [2], see Figure 3(left) for an overview of the process. Scalar quantities and inter-particle forces in this model are computed using a spatial kernel surrounding each particle with radius h. For example, the equation for density of a particle at position r_i is:

$$\rho(r_i) = \sum_j m_j W(r_i - r_j, h), \quad W_{poly6}(r,h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - r^2)^3 & \text{if } 0 \le r \le h, \\ 0 & \text{otherwise} \end{cases}$$

Where m is mass, h is the smoothing radius and W is the symmetric kernel W_{poly6}. A naive implementation of this model has complexity $O(N^2)$, however the kernels used in the model have finite support over radius h. This means that only particles within a local neighborhood need to be considered. Figure 3(right) contains pseudo code for an algorithm that uses the quadtree to limit the computation to this region. For each particle, the entire tree is traversed using a sphere that represents the smoothing kernel W, e.g. the radius of the sphere is equal to the support of the kernel. This traversal finds all the neighboring particles and evaluates the kernel for each. Pressure and viscosity are computed in a similar fashion using the equations given in [2].

In addition to accelerating force computations, we also use the quadtree for particle to environment collisions. To handle dynamic surfaces using our method, we must store the mesh such that a quadtree can be easily updated. Geometry images [23] encode mesh data in an array where each 2×2 block represents two triangles. This layout has the spatial coherence property required for our reduction method to compute an efficient quadtree. The construction routine is modified slightly such that the first level is built by constructing a bounding volume for the two triangles. After this step, the construction continues in the manner described in Section 3.

To intersect a moving particle with a mesh, we construct a ray from the previous particle position to the new one. We use the quadtree to accelerate the intersection test by traversing using a sphere that encapsulates the ray. If the sphere overlaps a leaf, we perform a ray-triangle intersection with the two triangles bound by the leaf.

The output of the traversal is the nearest point of intersection, if one occurred, and the corresponding surface normal at this point. This information is used to update the position and velocity of particles that collide with the mesh.

5 Results

To evaluate our implementation, we have set up several experiments using the breaking dam configuration as shown in Figure 4(top)[1]. Our test machine is a 3.19 GHz Xeon with 3GB of RAM running a 256MB GeForce FX 7800 GPU. The graph in Figure 4 (bottom right) compares the performance differences as the problem size increases.

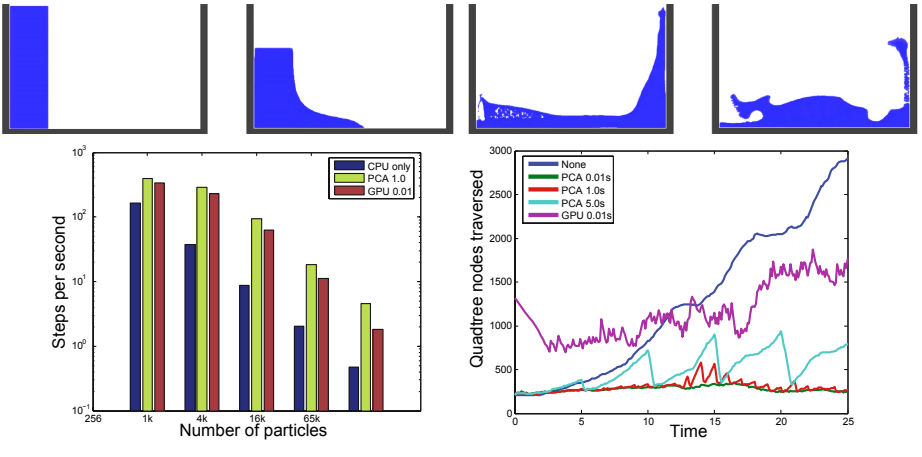


Fig. 4. Images of the breaking dam simulation(*top*). CPU and GPU solvers using both optimization methods for different problem sizes(*bottom left*). Maximum node traversals on the GPU required to resolve particle interactions(*bottom right*).

We compare three solvers: a GPU solver using CPU based PCA tree optimization, a GPU solver using the GPU random sorting method, and a CPU only solver that uses the grid based acceleration strategy. Our CPU implementation is optimized using efficient data structures. Further performance may be achievable by taking advantage of native CPU SIMD instruction sets such as SSE along with cache friendly memory layout [2]. The performance numbers are given as averages over the total simulation run visualization disabled. As the problem size increases, the GPU solvers outperform the purely CPU implementation in some cases by a factor of ten.

As mentioned in Section 3.3, the quality of the quadtree becomes less efficient as the particles move. Since performance is limited by the worst case, a simple measure of this quality is the number of nodes traversed to find the neighboring particles. A second experiment was run with 65k particles using each of the different optimization strategies described in the paper. For each time step, we record the maximum number

[1] A movie is available at `http://www.cs.sunysb.edu/~kyhegem/gpgpu06/sph.mov`

of nodes traversed for a particle(Figure 4 bottom left). As can be seen, the quality of the tree quickly degrades if no optimization is performed, thus the performance of our simulation steadily decreases over time. To achieve the best performance, we have experimented with different optimization frequencies. The quality of the tree produced by the GPU optimization is not as good as the CPU, but is significantly cheaper to run because there is no transfer of data. Because of the cost associated with transferring particle positions to the CPU, it is beneficial to overall performance to reduce the frequency. For this example, optimizing the tree on the CPU once a second produces the best performance.

6 Conclusion and Future Work

In this paper we have presented methods for using a dynamic quadtree structure for accelerating nearest neighbor queries in particle systems. Our method efficiently rebuilds a quadtree each step of the simulation. Since the quality of the quadtree degrades over time, we have proposed lazily reoptimizing the tree. As future work, we would like to extend our GPU random sorting routine to use PCA for choosing a more optimal splitting axis. Lastly further research is needed to explore our method's usefulness in dynamic 3D environments.

References

1. Miller, G., Pearce, A.: Globular dynamics: A connected particle system for animating viscous fluids. Computers and Graphics **13**(3) (1989) 305–309
2. Müller, M., Charypar, D., Gross, M.: Particle-based fluid simulation for interactive applications. In: Proc. of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation. (2003) 154–159
3. Baraff, D., Witkin, A.: Large steps in cloth simulation. In: Proc. of SIGGRAPH '98. (1998) 43–54
4. Premože, S., Tasdizen, T., Bigler, J., Lefohn, A., Whitaker, R.T.: Particle-based simulation of fluids. In: Proc. of Eurographics 2003. Volume 22. (2003)
5. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. In: Eurographics 2005, State of the Art Reports. (2005) 21–51
6. Kolb, A., Latta, L., Rezk-Salama, C.: Hardware-based simulation and collision detection for large particle systems. In: Proc. of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware. (2004) 123–131
7. Krüger, J., Kipfer, P., Kondratieva, P., Westermann, R.: A particle system for interactive visualization of 3D flows. IEEE Trans. on Visualization and Computer Graphics **11**(6) (2005)
8. Chiara, R.D., Erra, U., Scarano, V., Tatafiore, M.: Massive simulation using GPU of a distributed behavioral model of a flock with obstacle avoidance. In: Proc. of the Vision, Modeling, and Visualization Conference 2004. (2004) 233–240
9. Kolb, A., Cuntz, N.: Dynamic particle coupling for GPU-based fluid simulation. In: Proc. of 18th Symposium on Simulation Technique. (2005) 722–727
10. Foley, T., Sugerman, J.: KD-tree acceleration structures for a GPU raytracer. In: Proc. of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware. (2005) 15–22

11. Simonsen, L.O., Thrane, N., Ørbæk, P.: A comparison of acceleration structures for GPU assisted ray-tracing. Masters Thesis (2005)
12. Lefohn, A., Kniss, J.M., Strzodka, R., Sengupta, S., Owens, J.D.: Glift: Generic, efficient, random-access GPU data structures. ACM Transactions on Graphics **25**(1) (2006)
13. Lefebvre, S., Hornus, S., Neyret, F.: Octree textures on the GPU. In Pharr, M., ed.: GPU Gems 2. Addison Wesley (2005) 595–613
14. Govindaraju, N.K., Redon, S., Lin, M.C., Manocha, D.: CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. In: Proc. of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware. (2003) 25–32
15. Govindaraju, N.K., Lin, M.C., Manocha, D.: Fast and reliable collision culling using graphics hardware. In: Proc. of the ACM symposium on Virtual reality software and technology. (2004) 2–9
16. Govindaraju, N.K., Knott, D., Jain, N., Kabul, I., Tamstorf, R., Gayle, R., Lin, M.C., Manocha, D.: Interactive collision detection between deformable models using chromatic decomposition. ACM Transactions on Graphics **24**(3) (2005) 991–999
17. Gottschalk, S., Lin, M.C., Manocha, D.: OBBTree: a hierarchical structure for rapid interference detection. In: Proc. of SIGGRAPH '96. (1996) 171–180
18. Klosowski, J.T., Held, M., Mitchell, J.S.B., Sowizral, H., Zikan, K.: Efficient collision detection using bounding volume hierarchies of *k*-dops. IEEE Trans. on Visualization and Computer Graphics **4**(1) (1998) 21–36
19. Beneš;, B., Villanueva, N.G.: GI-COLLIDE: collision detection with geometry images. In: Proc. of the 21st spring conference on Computer graphics. (2005) 95–102
20. Purcell, T.J., Donner, C., Cammarano, M., Jensen, H.W., Hanrahan, P.: Photon mapping on programmable graphics hardware. In: Proc. of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware. (2003) 41–50
21. Lucy, L.B.: A Numerical Approach to Testing the Fission Hypothesis. The Astronomical Journal **82**(12) (1977) 1013–1924
22. Gingold, R., Monaghan, J.: Smoothed particle hydrodynamics: Theory and application to non-spherical stars. Monthly Notices of the Royal Astronomical Society **181** (1977) 375–389
23. Gu, X., Gortler, S.J., Hoppe, H.: Geometry images. In: Proc. of SIGGRAPH '02. (2002) 355–361