Real-Time Particle System Control

Brian D. Strege

Abstract

This project aims to accelerate various particle system effects so that they may be used in a real-time environment. The particle system computations will be executed on the GPU so as to maintain high performance, and various approximation algorithms will be used to try and improve the efficiencies of the implementations of reasonably complex particle system effects. Such effects include the N-Body problem from physics, which is essentially a gravity model, as well as attempting to force particle system motion through pre-defined keyframes.

Keywords: particle system, GPU, approximation

1 Introduction

The purpose of this project is to be able to see more interesting effects produced with particle systems in a real-time environment. Particle systems are commonly used to render phenomena such as fire, smoke and water in real-time situations to great effect, whereas more interesting methods of controlling particle systems such as forcing them through specified keyframes or displaying massive particle interaction can be far too computationally intensive to use in a real-time environment. Certainly such effects as simple particle collision have been accomplished before in an efficient matter, but there many particle effects that have not been adapted in an efficient manner. These include the repulsion and/or attraction of all particles in a system to model things such as gravity or magnetism, as well as setting the desired result and/or intermediate steps of a particle system by way of keyframes.

Initially the goal will be to mimic an established method for performing particle system computations on the GPU, as the fastest implementations of these systems will take advantage of graphics hardware so as to both exploit its massive hardware capacity as well as to avoid any potential data bottleneck from the CPU. Once this has been completed, a series of traditionally computationally intensive particle system effects such as particle attraction and keyframe control will be implemented, and then various approximation methods with which to improve their performance will be applied. Specifically in terms of the particle attraction problem, the Barnes-Hut algorithm [Barnes and Hut 1986] has been explored as a method of approximation for this effect.

2 Related Work

In the literature for particle systems, there is a divide between papers that are trying to accomplish some effect in a particularly realistic way, and papers that are trying to accelerate existing effects in order to make them palatable to real-time systems. Many of the most interesting effects – as could be expected – come from the former section of papers, where the concern lies primarily on the visual appeal of the results rather than the performance with which these results were achieved. That is not to say they did not care at all about performance, but it was certainly secondary to quality. The papers that were more concerned with performance either attempted to figure out ways to perform approximate calculations for existing particle system effects, or to perform these computations on different hardware, i.e. by moving the computations from the CPU to the GPU.

There are a few very interesting papers that can be logically laid out in a chronological series [Treuille et al. 2003; McNamara et al. 2004; Wojtan et al. 2006] which discuss very rigid control of particle systems in order to achieve high-quality effects. They use a series of keyframes to allow an animator to control various intermediate stages of display for various particle systems, while retaining those systems' inherent properties be they that of smoke, water or cloth. They obtain the best sequence of motion mathematically by generating an objective function which is intuitively the measure of how incorrect a given effect looks, and minimizing that objective function. Given all of the variables involved, this can be very computationally intensive; far too slow to use in a real-time system. While later papers in this series discuss ways of improving the performance using what is known as the adjoint method, they remain firmly in the realm of non-real-time systems. Still, these papers provide good insight as to how one might go about enforcing control over particle systems, and if raytracing has taught us anything it is that looking at the brute-force solution to a problem can be a good place to start.

Essential work has also been done in the realm of moving particle system computations off of the CPU, and onto the GPU to take advantage of its natural parallelism for the inherently parallel nature of many particle system effects. Also, executing particle system computations entirely on the GPU eliminates any potential bottleneck from the bus to the CPU, which can also improve performance. There have been various methods [Kipfer et al. 2004; Kolb et al. 2004] developed for using the GPU to handle particle system computation, and they certainly will be of great use in this project.

Finally, there are a few very recent papers available describing how to perform a specific particle system effect both efficiently and realistically. These effects include splashing water [Kim et al. 2006] and stiffness in mass-spring systems [Volino and Magnenat-Thalmann 2006].

3 Implementation

3.1 N-Body Problem

The model effect that this project will be exploring is a simple gravity system of particles, where each particle contains its own mass and acts on all other particles in the system by way of the well-established Newtonian laws of gravity. In the realm of physics, this is known as the N-body problem. To simulate the N-Body problem, the following equations must be executed at every time step, where the duration of the time step can be arbitrarily chosen.

$$F_x = \frac{Gm_a m_b}{r^2}\left(\frac{x_b - x_a}{r}\right) \tag{1}$$

$$F_y = \frac{Gm_a m_b}{r^2}\left(\frac{y_b - y_a}{r}\right) \tag{2}$$

$$F_z = \frac{Gm_a m_b}{r^2}\left(\frac{z_b - z_a}{r}\right) \tag{3}$$

where r is the distance between particles. After all three dimensions of the force between two particles a and b are computed as shown

in equations 1, 2 and 3, the change to a particular particle's velocity, dv, must be computed. We will use Newton's second law for this:

$$F = ma \tag{4}$$

Substituting dv/dt for a, and solving for dv, we arrive at the following equation:

$$dv = \frac{Fdt}{m} \tag{5}$$

where dt is the time step. While these computations seem simple, they can become exceedingly difficult if the amount of particles increases greatly. The reason for this is that each and every **pair** of particles must execute all of these equations at each time step. For n particles, that will be n^2 times these equations must be computed per time step.

3.2 GPU Particle System

As stated earlier, there is a lot of highly relevant work already in existence to put the computations required in particle systems onto the GPU. This project will base its GPU solution on the work done by Kipfer et al. in regards to their GPU-based particle engine named UberFlow [Kipfer et al. 2004]. In their paper, they lay out a series of events that must happen at every time step in order to achieve their desired particle system effects, which are simply motion and collisions. The events they describe are as follows:

- Emission
- Collisionless motion of particles
- Sorting of particles
- Pairing of collision partners
- Collision response
- Enforcement of boundary conditions
- Particle rendering

The UberFlow system handles collisions in a way that is completely satisfactory for the goals of this project. Therefore, the part of this event pipeline that we will be modifying will come just before the sorting of particles, since UberFlow only sorts the particles for purposes of collision. A stage must be added to perform the N-Body problem, which will update the velocities of all the particles. Since we will want to then use this updated velocity to move the particles around, the most logical place for this new stage will be just before the collisionless motion of particles

Also, we will slightly modify the way in which UberFlow stores the particles. Currently, a 2D RGBA texture is used to store particle positions, with the RGB components corresponding to the XYZ values of these positions, and the A component corresponding to the time in which the given particle will be released. We are going to simplify this by treating all particles as if they had always existed, thereby removing the need for the emission stage of their pipeline. This will make the system now require a one-time initialization. Finally, we will use the A component of the texture to store the associated mass of the particle. Our system will also use a separate texture to store the velocities of the particles. The updated UberFlow events are as follows:

- **N-Body velocity computation**
- Collisionless motion of particles

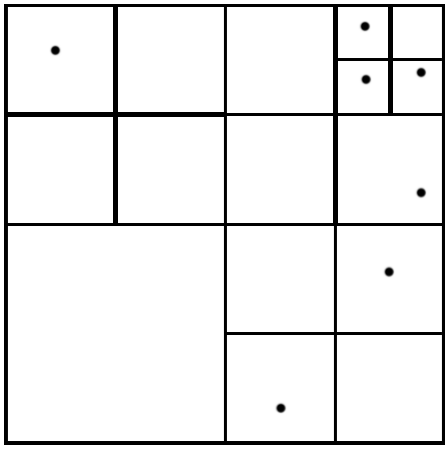

Figure 1: *Partitioning of a 2D particle scene using the Barnes-Hut algorithm.*

- Sorting of particles
- Pairing of collision partners
- Collision response
- Enforcement of boundary conditions
- Particle rendering

3.3 Inapplicable Parallel Solution

Now that the location in the event pipeline has been determined for the execution of the N-Body velocity computation, we must determine how exactly this problem would be solved. Since GPUs are massively parallel pieces of hardware, it seemed to make sense to use a common technique for performing N-Body computations known as the Barnes-Hut algorithm [Barnes and Hut 1986]. This algorithm takes a 3D world full of bodies, and continually partitions the world down into octants until every individual partition of the world contains either one or zero bodies. It creates an octtree while performing this partitioning, and any time a partition contains zero bodies it is thrown away. This is easier to visualize in 2D, where a quadtree would be created. As can be seen in Figure 1, the scene is cut further and further down into quadrants until all six bodies shown reside in their own partition.

Once this octree (or in the case of 2D, quadtree) has been created, the granularity at which the Barnes-Hut algorithm can choose to compute the N-Body forces can be varied throughout the world. Large numbers of bodies that reside in large areas of space can have their positions and masses averaged to simplify force computation for a far off body by treating that group as one large entity.

Unfortunately, this algorithm does not lend itself well for the purposes of this project. First, this algorithm is primarily used for real-world computation of planetary systems, which are much different than the gravity systems that are likely to be employed in a real time application such as a video game. In actualy planetary systems, there may be many distinct groups of bodies – such as galaxies – that would fit well into the Barnes-Hut algorithm's method of averaging together large groups of particles that are physically close to each other. In video games on the other hand, the large amount of particles are likely to all be very close together and quite chaotic, leaving no clear groups of bodies to be averaged together.

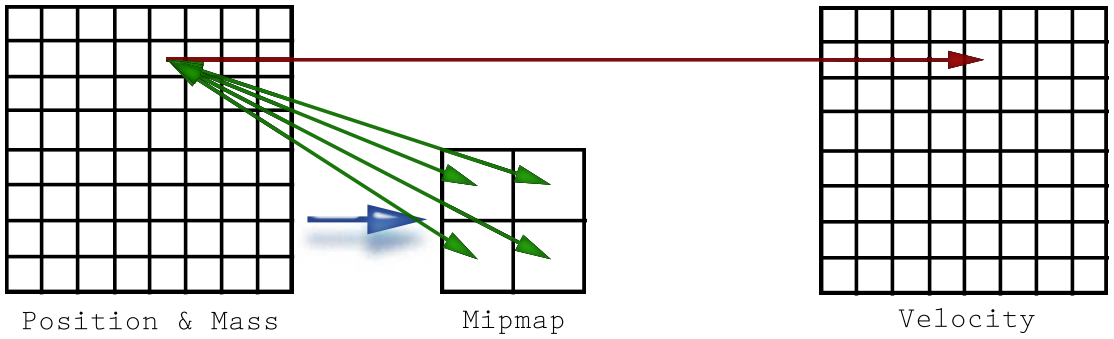

Figure 2: *N-Body computation with generated mipmap. The blue arrow indicates the mipmap generation from the position and mass texture. The green arrows indicate all of the force computation required for one particular particle. The red arrow indicates the update to the velocity texture for the same particle.*

Also, since this algorithm would be running on the GPU, it is important to note that there is no easy way of creating a new octree for the particle system at each time step, which is most likely each frame. Seeing as how there are many drawbacks to the Barnes-Hut algorithm, it has been abandoned as the algorithm to approximate the N-Body problem on the GPU.

3.4 Mipmap Solution

Keeping in mind the strengths and weaknesses of the Barnes-Hut algorithm, we have come up with an algorithm that should run quite well on the GPU, as well as significantly reduce the amount of computations required for the N-Body problem. Note that when we compute the forces in the N-Body problem, the texture that stores the positions and masses of the particles has not been sorted. We will generate a mipmap of this texture, and use it to approximate the forces of the N-Body problem. In Figure 2, a $16x16$ texture is using a $2x2$ texture from its mipmap set to compute the change in velocity for one particular particle. In this example, if the mipmap were not used it would require 256 force computations to determine the change in velocity for a single particle – now it takes just 4.

This method does require the overhead of computing the mipmap set for position and mass texture at each time step, however modern GPUs are designed to do this very efficiently. After the mipmap is created, it greatly reduces the amount of computation required for the problem. If there are n particles in a scene, and the mipmap used contains m elements, the amount of computation now required is nm, instead of n^2.

4 Results

At this time a GPU implementation of the system described has not yet been created. However, the N-Body portion of the system described has been "emulated" on the CPU, performing the same calculations as the GPU version would, except it directly calculates the mipmap size desired. Running the brute-force standard N-Body problem on the CPU with 65536 particles – a $256x256$ texture worth – the test system is running at around 8 frames per second. When we employ the mipmap method using a size of 2x2 for the mipmap, the test system runs at around 28 frames per second. This increase makes the mipmap method look quite promising, especially considering that graphics hardware will be able to compute mipmaps in a fraction of the time that a CPU based implementation can. The test system is a 3.2GHz Pentium 4 with 2GB of RAM.

5 Future Work

The first thing that is necessary before considering any other future work is to get a GPU based version of the described solution implemented, so performance can be tested. After that, a consideration would be to model magnetism in addition to gravity, so some particles attract each other and repel others.

References

BARNES, J. E., AND HUT, P., 1986. A hierarchical o(nlogn) force calculation algorithm.

KIM, J., CHA, D., CHANG, B., KOO, B., AND IHM, I. 2006. Practical animation of turbulent splashing water. In *SCA '06: Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 335–344.

KIPFER, P., SEGAL, M., AND WESTERMANN, R. 2004. Uberflow: a gpu-based particle engine. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ACM Press, New York, NY, USA, 115–122.

KOLB, A., LATTA, L., AND REZK-SALAMA, C. 2004. Hardware-based simulation and collision detection for large particle systems. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ACM Press, New York, NY, USA, 123–131.

MCNAMARA, A., TREUILLE, A., POPOVIĆ, Z., AND STAM, J. 2004. Fluid control using the adjoint method. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, ACM Press, New York, NY, USA, 449–456.

TREUILLE, A., MCNAMARA, A., POPOVIĆ, Z., AND STAM, J. 2003. Keyframe control of smoke simulations. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, ACM Press, New York, NY, USA, 716–723.

VOLINO, P., AND MAGNENAT-THALMANN, N. 2006. Simple linear bending stiffness in particle systems. In *SCA '06: Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 101–105.

WOJTAN, C., MUCHA, P. J., AND TURK, G. 2006. Keyframe control of complex particle systems using the adjoint method. In *SCA '06: Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 15–23.