Particle System Simulations

CS6068

Dr Fred Annexstein

Particles System

Collection of particles – In a particle system each particle has attributes that directly or indirectly effect the behavior of the particle or ultimately how and where the particle is rendered.

Often, particles are graphical primitives such as points or lines, but they are not limited to this. Particle systems have also been used to represent complex group dynamics such as flocking birds.

Stochastically defined attributes - The other common characteristic of all particle systems is the introduction of some type of random element. This random element can be used to control the particle attributes such as position, velocity and color. Usually the random element is controlled by some type of predefined stochastic limits, such as bounds, variance, or type of distribution.

Attributes of a Particle

Each object in (Reeves) particle system have the following attributes:

- **Position**
- **Velocity (speed and direction)**
- **Color**
- **Lifetime**
- **Age**
- **Shape**
- **Size**
- **Transparency**

- **Additional Attributes**
- **Force Rules**
- **Division Rule**

Particle Life Cycle

Each particle goes through three distinct phases in the particle system: generation, dynamics, and death.

Generation - Particles in the system are generated randomly within a predetermined location of the fuzzy object. This space is termed the generation shape of the fuzzy object, and this generation shape may change over time. Each of the above mentioned attribute is given an initial value. These initial values may be fixed or may be determined by a stochastic process.

Particle Dynamics

- The attributes of each of the particles vary over time.
- For example, the color of a particle in an explosion may get darker as it gets further from the center of the explosion, indicating that it is cooling off.
- In general, each of the particle attributes can be specified by a parametric equation with time as the parameter. Particle attributes can be functions of both time and other particle attributes. For example, particle position is going to be dependent on previous particle position and velocity as well as time.

Particle Life Cycle-Extinction

Each particle has two attributes dealing with length of existence: age and lifetime. Age is the time that the particle has been alive (measured in frames), this value is always initialized to 0 when the particle is created. Lifetime is the maximum amount of time that the particle can live (measured in frames). When the particle age matches it's lifetime it is destroyed. In addition there may be other criteria for terminating a particle prematurely:

Running out of bounds - If a particle moves out of the viewing area and will not reenter it, then there is no reason to keep the particle active.

Hitting the ground - It may be assumed that particles that run into the ground burn out and can no longer be seen.

Some attribute reaches a threshold - For example, if the particle color is so close to black that it will not contribute any color to the final image, then it can be safely destroyed.

Rendering

- When rendering system of thousands of particles, some assumptions have to be made to simplify the process.
- First, each particle is often rendered to a small graphical primitive (blob).
- Particles that map to the same pixels in the image are often additive - the color of a pixel is simply the sum of the color values of all the particles that map to it. Because of this assumption, no hidden surface algorithms are needed to render the image, the particles are simply rendered in order.
- Effects like temporal anti-aliasing (motion blur) are made simple by the particle system process. The position and velocity are known for each particle. By rendering a particle as a streak, motion blur can be achieved.

Modeling Flocking Birds

Reynolds used particles to model the behavior of birds moving in a flock [REY87]. Particles are used to represent "boids" (short for bird-object). This use of a particle system has a few differences from what was used by Reeves:

- Each boid is an entire polygonal object rather than a graphical primitive
- Each boid has a local coordinate system.
- There are a fixed number of boids - they are not created or destroyed.
- Traditional rendering methods can be used because there are a small number of boids.
- Boids behavior is dependent on external as well as internal state. In other words, a boid reacts to what other boids are doing around it.

Boid Behavior Model

Reynolds used observation of real flocks and research of flock behavior to come up with three primary needs of a boid.

- Collision avoidance - The boid does not wish to collide with other boids or obstacles.
- Velocity matching - Each boid attempts to go the same speed and direction as neighboring boids.
- Flock centering - Each boid attempts to stay close to nearby flockmates.
- The boid's movement is made by combining the impulses that are generated from these three needs. If each need produces a vector that represents the direction and speed it thinks the boid should move in, it may not be adequate to simply average these vectors. In the worst case, these three vectors may be pointing in completely divergent directions, and the net movement would be zero.

Fluid Flow Simulations

“The hardest shot in movie “Shrek”….. It’s the pouring of milk into a glass.” Jeffrey Katzenberg

Types of Particle Simulation

- Eulerian (grid-based) methods, which calculate the properties of the simulation at a set of fixed points in space
- Lagrangian (individual particle-based) methods, which calculate the properties of a set of particles as they move through space.

Lagrangian methods:

· They only perform computation where necessary.

· Generally require less storage and bandwidth since the model properties are only stored at the particle positions, rather than at every point in space.

· They are not necessarily constrained to a finite box.

· Conservation of mass is simple (since each particle represents a fixed amount of mass).

- Require a very large number of particles to obtain realistic results.
- Relatively easy to parallelize particle systems and the massive parallel computation capabilities of modern GPUs makes it possible to simulate large systems at interactive rates.
- Difficult to capture complex topological features and free surface flows.
- Fixed grid representation for the flow variables (pressure, densities, velocities, etc.) can alleviate some of the difficulties BUT to simulate complex surfaces are modeled with marker particles or level-sets are introduced.

Demonstration of a simple particle system in CUDA, including particle collisions using a uniform grid data structure.

Integration

The integration step is the simplest step. It integrates the particle attributes (position and velocity) to move the particles through space. We use Euler integration for simplicity - the velocity is updated based on applied forces and gravity, and then the position is updated based on the velocity. Damping and interactions with the bounding cube are also applied in this stage.

The particle positions and velocities are both stored in float4 arrays. The positions are actually allocated in an OpenGL vertex array object (VBO) so that they can be rendered from directly.

This VBO memory is mapped for use by CUDA using cudaGLMapBufferObject. The arrays are double-buffered so that updating the new values will not affect particles not yet processed

Particle-Particle Interactions

- **It is relatively simple to implement a particle system where particles do not interact with each other. Most particle systems used in games today fall into this category. In this case each particle is independent and they can be simulated trivially in parallel.**

- **The nbody sample included in the CUDA SDK includes interactions in the form of gravitational attraction between bodies. It demonstrates that it is possible to get excellent performance for n-body gravitational simulation using CUDA when performing the interaction calculations in a brute-force manner – computing all n^2 interactions for n bodies.**

- **We can improve performance by using spatial subdivision, since, for many types of interaction, the interaction force drops off with distance. This means that we can compute the force for a given particle by only comparing it with all its neighbors within a certain radius.Spatial subdivision techniques divide the simulation space so that it is easier to find the neighbors of a given particle.**

Uniform Grids

- A uniform grid is the simplest possible spatial subdivision. The techniques could be extended to more sophisticated structures such as hierarchical grids.
- A uniform grid subdivides the simulation space into a grid of uniformly sized cells. For simplicity, we use a grid where the cell size is the same as the size of the particle (double its radius).
  - each particle can cover only a limited number of grid cells (8 in 3D)
  - there is a upper bound on the number of particles per grid cell (4 in 3D)
- Each particle is assigned to only one grid cell based on its center point. Since each particle can potentially overlap several grid cells, when processing collisions we must also examine the particles in the neighboring (3 x 3 x 3 = 27) cells
- This method allows us to bin the particles into the grid cells simply by sorting them by their grid index.
- The alternative approach, where particles are stored in every cell that they touch, requires less work when processing collisions, but more work when building the grid.
- The grid data structure is generated from scratch each time step. It is possible to perform incremental updates to the grid structure on the GPU, but this approach is simple and the performance is constant regardless of the movement of the particles.
- We examine two different methods for generating the grid structure

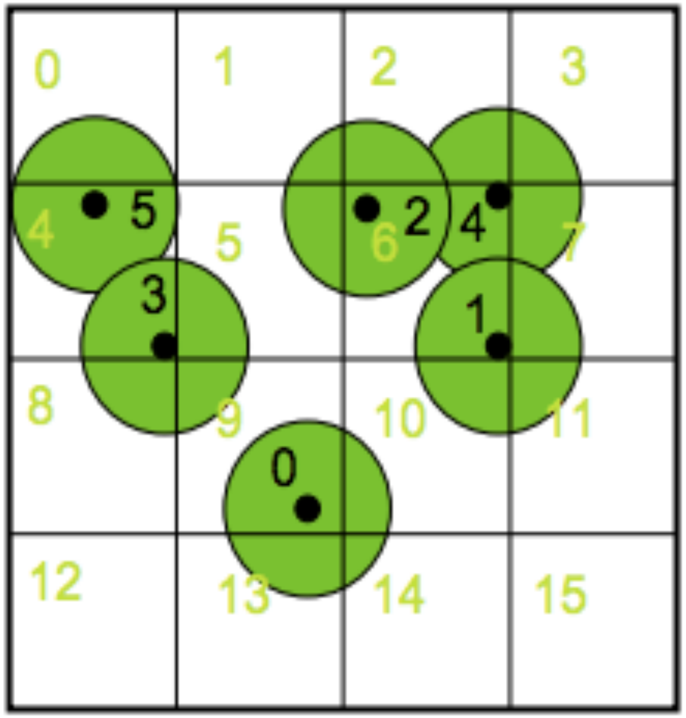
Building the Grid using Atomic Operations

Atomics is a relatively simple algorithm for building the grid, allow multiple threads to update the same value in global memory simultaneously without conflicts.

We use 2 arrays in global memory:
gridCounters[] – an array that stores the number of particles in each cell so far. It is initialized to zero at the start of each frame.

gridCells[] – an array that stores the particle indices for each cell, and has room for a fixed maximum number of particles per cell.

updateGrid() is kernel function updates the grid structure. Assigns one thread per particle tha calculates which grid cell it is in. It uses the atomicAdd function to atomically increment the cell counter corresponding to this location. It then writes its index into the gridCells[] array at the correct position (using a scattered global write).



Cell id	Count	Particle id
0	0	
1	0	
2	0	
3	0	
4	2	3, 5
5	0	
6	3	1, 2, 4
7	0	
8	0	
9	1	0
10	0	
11	0	
12	0	
13	0	
14	0	
15	0	

Figure 2 – Uniform Grid using Atomics

Building the Grid using Sorting

An alternative approach which does not require atomic operations is to use sorting. The algorithm consists of several kernels.
The first kernel **—calcHash()** calculates a hash value for each particle based on its cell id. We use the linear cell id as the hash, but alternatives include Z-order curve [8] to improve the coherence of memory accesses. The kernel stores the results to the **—particleHash**[] array in global memory as a uint2 pair (cell hash, particle id). We then sort the particles based on their hash values. The sorting is performed using the fast radix sort.
Sorting creates a list of particle ids in cell order, however, we need to be able to find the start of any given cell in the sorted list. This is achieved by running another kernel **---findCellStart()**.

Building the Grid using Sorting

—**findCellStart()** uses a thread per particle and compares the cell index of the current particle with the cell index of the previous particle in the sorted list. If the index is different, this indicates the start of a new cell, and the start address is written to another array using a scattered write. The current code also finds the index of the end of each cell in a similar way.

Uniform Grid Using Sorting

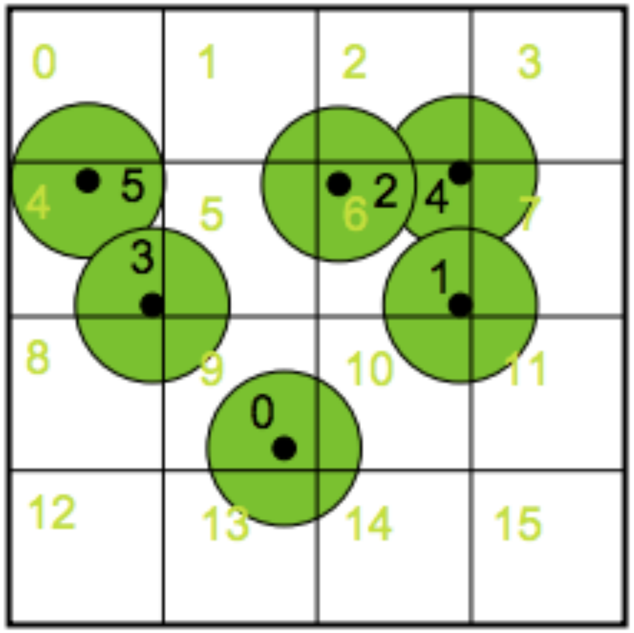


Figure 3 – Uniform Grid using Sorting

Index	Unsorted list (cell id, particle id)	List sorted by cell id	Cell start
0	(9, 0)	(4, 3)	
1	(6, 1)	(4, 5)	
2	(6, 2)	(6, 1)	
3	(4, 3)	(6, 2)	
4	(6, 4)	(6, 4)	0
5	(4, 5)	(9, 0)	
6			2
7			
8			
9			5
10			
11			
12			
13			
14			
15			

Particle Collisions

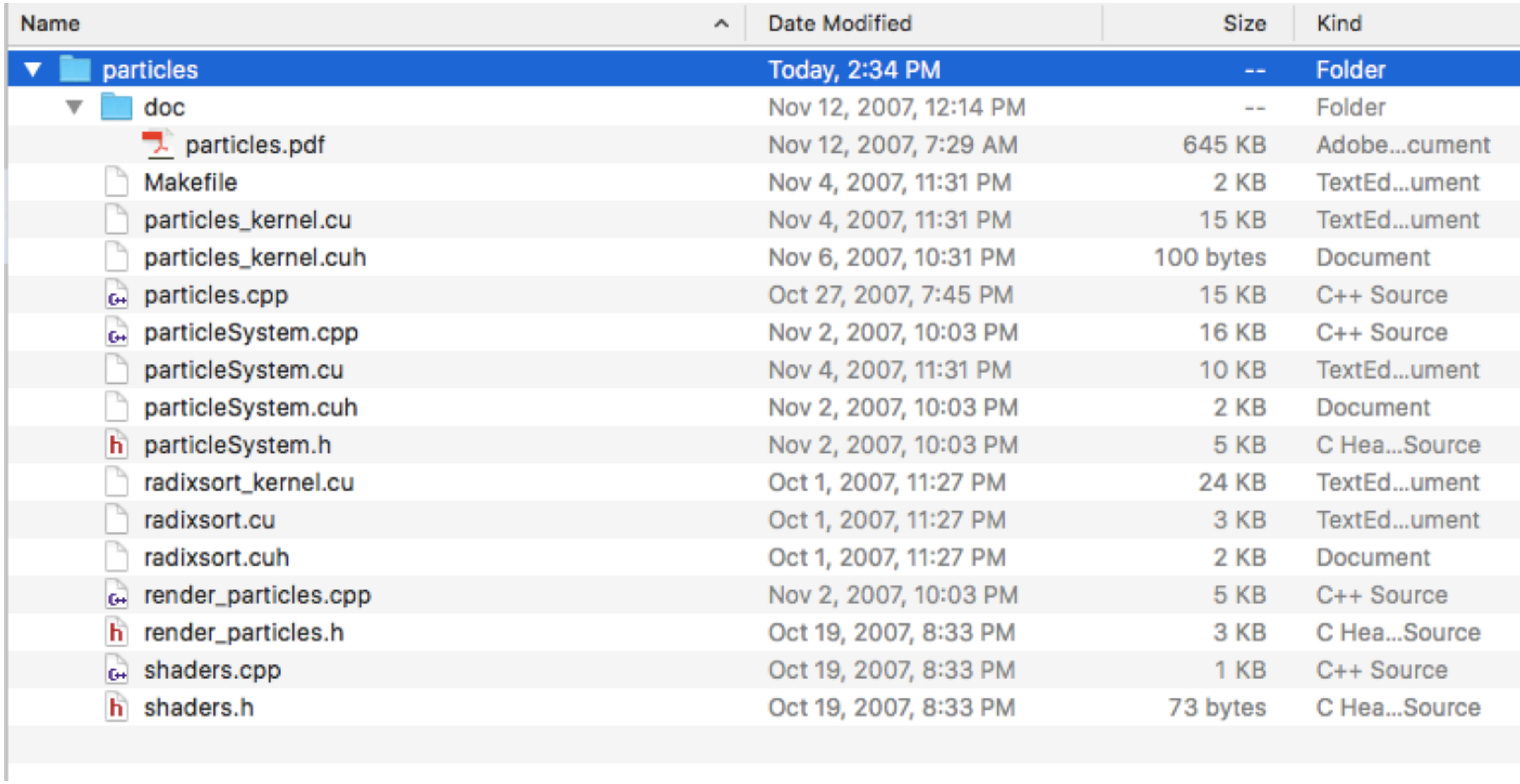
Once we have built the grid structure, we can use it to accelerate particle-particle interactions. In the sample code we perform simple collisions between particles using several forces, including a spring force which forces the particles apart, and a dashpot force which causes damping.

Each particle calculates which grid cell it is in. It then loops over the neighboring 27 grid cells (3x3x3 cells) and checks for collisions with each of the particles in these cells. If there is a collision the particle‘s velocity is modified.

Bibliography

1. Reeves, W. T. 1983. Particle Systems—a Technique for Modeling a Class of Fuzzy Objects. ACM Trans. Graph. 2, 2 (Apr. 1983), 91-108.
2. Monaghan J.: Smoothed particle hydrodynamics. Annu. Rev. Astron. Physics 30 (1992), 543.
3. Müller M., Charypar D., Gross M.: Particle-based fluid simulation for interactive applications. Proceedings of 2003 ACM SIGGRAPH Symposium on Computer Animation (2003), 154–159.
4. Müller, M., Heidelberger, B., Hennix, M., and Ratcliff, J. 2007. Position based dynamics. J. Vis. Comun. Image Represent. 18, 2 (Apr. 2007), 109-118.
5. Harada, T.: Real-Time Rigid Body Simulation on GPUs. GPU Gems 3. Addison Wesley.
6. Le Grand, S.: Broad-Phase Collision Detection with CUDA. GPU Gems 3, Addison Wesley,
7. Nyland, L., Harris, M., Prins, J.: Fast N-Body Simulation with CUDA. GPU Gems 3. Addison Wesley.
8. Z-order (curve), Wikipedia http://en.wikipedia.org/wiki/Z-order_(curve)
9. Ian Buck and Tim Purcell, A Toolkit for Computation on GPUs, GPU Gems, AddisonWesley.
10. Qiming Hou, Kun Zhou, Baining Guo, BSGP: Bulk-Synchronous GPU Programming, ACM TOG (SIGGRAPH 2008)
11. Ericson, C., Real-Time Collision Detection, Morgan Kaufmann 2005
12. Satish, N., Harris, M., Garland, M., Designing Efficient Sorting Algorithms for Manycore GPUs, 2009.
13. Joshua A. Anderson, Chris D. Lorenz, and Alex Travesset General purpose molecular dynamics simulations fully implemented on graphics processing units, Journal of Computational Physics 227 (2008)

CUDA ToolKit Sample Code



Name	Date Modified	Size	Kind
▼ particles	Today, 2:34 PM	--	Folder
▼ doc	Nov 12, 2007, 12:14 PM	--	Folder
particles.pdf	Nov 12, 2007, 7:29 AM	645 KB	Adobe...cument
Makefile	Nov 4, 2007, 11:31 PM	2 KB	TextEd...ument
particles_kernel.cu	Nov 4, 2007, 11:31 PM	15 KB	TextEd...ument
particles_kernel.cuh	Nov 6, 2007, 10:31 PM	100 bytes	Document
particles.cpp	Oct 27, 2007, 7:45 PM	15 KB	C++ Source
particleSystem.cpp	Nov 2, 2007, 10:03 PM	16 KB	C++ Source
particleSystem.cu	Nov 4, 2007, 11:31 PM	10 KB	TextEd...ument
particleSystem.cuh	Nov 2, 2007, 10:03 PM	2 KB	Document
particleSystem.h	Nov 2, 2007, 10:03 PM	5 KB	C Hea...Source
radixsort_kernel.cu	Oct 1, 2007, 11:27 PM	24 KB	TextEd...ument
radixsort.cu	Oct 1, 2007, 11:27 PM	3 KB	TextEd...ument
radixsort.cuh	Oct 1, 2007, 11:27 PM	2 KB	Document
render_particles.cpp	Nov 2, 2007, 10:03 PM	5 KB	C++ Source
render_particles.h	Oct 19, 2007, 8:33 PM	3 KB	C Hea...Source
shaders.cpp	Oct 19, 2007, 8:33 PM	1 KB	C++ Source
shaders.h	Oct 19, 2007, 8:33 PM	73 bytes	C Hea...Source

Particles.cpp

```
#define GRID_SIZE 64
#define NUM_PARTICLES 16384

Void key()
Case ‘’: pause
Case 13 : update
Case ‘q’: exit
Case ‘v’: view mode
Case ‘m’ : move mode
Case ‘p’:
Case ‘d’: dumpgrid
```

```
// particles_kernel.cu      // integrate particle attributes
__global__ void
integrate(float4* newPos, float4* newVel,
          float4* oldPos, float4* oldVel,
          float deltaTime,
          float damping,
          float particleRadius,
          float gravity) {
int index = __mul24(blockIdx.x,blockDim.x) + threadIdx.x;
float4 pos4 = oldPos[index];
float4 vel4 = oldVel[index];
float3 pos = make_float3(pos4);
float3 vel = make_float3(vel4);
float3 force = { 0.0f, gravity, 0.0f };
vel += force * deltaTime;
vel *= damping;
 // new position = old position + velocity * deltaTime
 pos += vel * deltaTime;
  // bounce off cube sides
  float bounceDamping = -0.5f;
```

```
void  integrateSystem(uint vboOldPos, uint vboNewPos,
              float* oldVel, float* newVel,
              float deltaTime,     float damping,
              float particleRadius,  float gravity,
              int numBodies) {
 int numThreads = min(256, numBodies);
 int numBlocks = (int) ceil(numBodies / (float) numThreads);
 float *oldPos, *newPos;
CUDA_SAFE_CALL(cudaGLMapBufferObject((void**)&ol
dPos, vboOldPos));
CUDA_SAFE_CALL(cudaGLMapBufferObject((void**)&n
ewPos, vboNewPos));
   // execute the kernel
   integrate<<< numBlocks, numThreads>>>
     ((float4*)newPos, (float4*)newVel, (float4*)oldPos,
(float4*)oldVel, deltaTime,  damping, particleRadius,  gravity
);
```

Cellular Design – Dr. Christoph Klemmt DAAP-Architecture

Research Project: Exploration of Neighbor Forces to achieve emergent Structural Features

Particle Interaction Rules:
If particles are closer than intended then apply repelling force
If particles are further but within threshold then attractive

Planarity Rule – calculate plane through 3 closest neighbor and apply normal force

Division Rule -- if only 2or 3 neighbors then assume on boundary and apply a particle division.

Ref: Cellular Design CK08.pdf