Cinematic Particle Systems with OpenCL

Tim Horton[*]
Rensselaer Polytechnic Institute

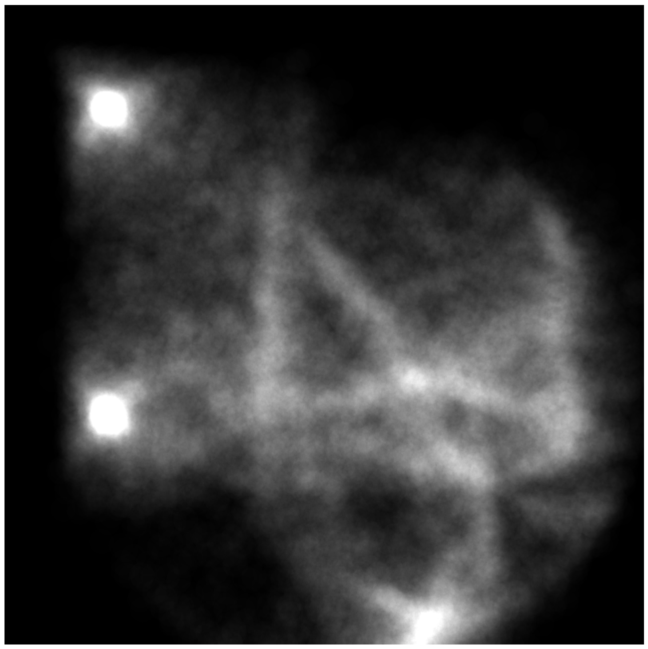

Figure 1: A simple gravity simulation with two emitters and three supermassive particles (approx. 45,000 particles)

Abstract

High-particle-count simulations are becoming increasingly crucial in many different aspects of our world today: both in entertainment — within video games, movies, and the like — and in scientific fields, where particle systems are capable of simulating and visualizing many interesting phenomena.

This paper will explore the possibility of parallelizing the simulation of these large particle systems and offloading them to very-parallel[1] hardware which is usually only used for *rendering*: the video card.

We will also touch briefly on ways to design a system for describing particle systems in a generalized way, though the majority of the work that is currently in a functional state centers around simulation and rendering.

1 Introduction

Simulation of large particle systems requires a large amount of computation — each particle must be inspected and updated. For certain types of particle systems — n-body gravity, for example — each particle is affected by its neighbors, increasing the required calculation time drastically.

This is what we aim to address with this paper. While it's unlikely that we can contribute any improved simulation algorithms, we can instead work to parallelize these algorithms and implement them with OpenCL, which will allow them to run on the GPU contained within modern graphics cards. This paper will demonstrate both the parallelization of the application of forces on large numbers of particles and the significant performance gained by doing so.

We will also explore a simple parallelized particle renderer, as well as the potential for the integration of our simulator into the popular open-source 3D package, Blender.

It is important to understand that the system we've developed is purely for offline work — it is not suitable for realtime simulation and rendering. Thus, *cinematic*. This is primarily because of the relatively poor performance of the renderer. However, as you'll notice in section 4, the speedup provided by our work *is* actually sufficient to make some simulations fast enough to run in real time, if one were to find a faster renderer.

2 Prior Art

[Drone 2007] discusses a system quite similar to ours — they use the GPU to simulate large-particle-count systems, including computationally-intensive forces like n-body gravity — however, they make one simplification which drastically improves performance: they assume that the particle data never needs to return to the CPU. This means that the extremely expensive copying to and from video memory which is done continually within our system is completely unnecessary, but also removes the ability to use the particle data in any flexible way (i.e. copying it to a file to be rendered later, etc.).

3 Implementation

The code for this paper is available under the two-clause BSD license at:

`http://github.com/hortont424/particles`

It consists of about 3000 lines[2] of C and Objective C, which encompass the curve editor and other design tools, an abstraction layer on top of OpenCL[3], and all of the code to drive simulation, preview, and rendering. In addition, there are well over 300 lines of OpenCL Kernel Language, which includes the code to apply each of the forces as well as the renderer.

3.1 Particle Systems

A particle system is defined by a simple JSON[4] file (figure 2 is a very simple example of such a file), which lays out all of the system's parameters: the number of particles to

[*]e-mail: hortot2@rpi.edu

[1]As opposed to *massively*-parallel, like the CCNI's Blue Gene/L, or the *slightly*-parallel CPU in most modern computers

[2]According to *sloccount*

[3]the Open Computing Language, a framework for creating programs using a language similar to GLSL and run on the GPU

[4]JavaScript Object Notation, a format often used today for information exchange in place of things like XML, due to its lightweight nature

```
{
    "initialParticles": {
        "count": 1
    },
    "forces": [
        {
            "kernel": "gravity",
            "strength": 1.0,
            "noise": 0.0,
            "mass": 100000000000.0,
            "particle": {
                "fixed": true,
                "x": 1.2,
                "y": 1.2,
                "z": 0.5
            }
        }
    ],
    "emitters": [
    {
        "birthRate": 20.0,
        "initialVelocity": 0.0,
        "particle": {
            "fixed": true,
            "x": -1.2,
            "y": 1.2,
            "z": 1.5
        }
    }],
    "integration": {
        "kernel": "verlet",
        "timestep": 0.005
    }
}
```

Figure 2: A simple sample curve file, with one emitter and one force

start out with, the location and properties of emitters, the location and properties of forces, and many other things. If *Interpolator* had been finished, this file would also contain the mappings between these properties and various curve files.

3.2 Design

The particle system design tools implemented during the course of this project are not entirely functional, as they took a backseat to the simulation, preview, and rendering components, mostly due to the fact that it's possible to design a system by hand.

The primary design tool, which can be seen in figure 3, is called *Interpolator*, as it is quite literally a tool to design interpolation curves. It's a Cocoa application which provides an interface to edit arbitrary Bézier splines and map them to properties of an element in the particle system. For example, one curve might dictate the x position of an emitter, while another might be mapped to the lifetime that emitter's children.

Since this tool was not finished in time for this paper, it is not integrated into the simulation and rendering phases introduced here.

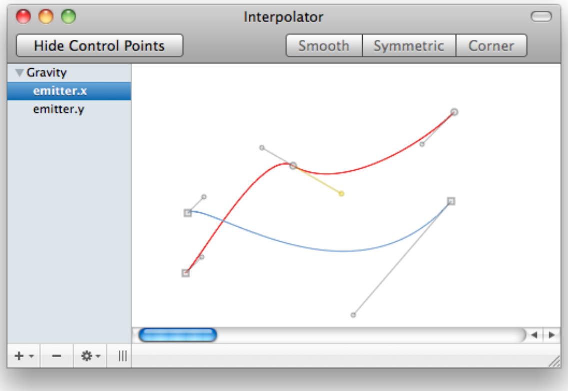

Figure 3: A screenshot of the unfinished *Interpolator* particle system design tool

3.3 Simulation

The simulation phase is the heart of our project. Starting from a system description file (as shown in figure 2), it eventually provides the position of each particle to either the previewing or rendering subsystems in a frame-by-frame manner. It involves various steps, which we'll outline below.

3.3.1 Parsing the .psys

The particle system description file, in JSON format, is parsed with Michael Clark's MIT-licensed **json-c** library. All of the properties are read into internal data structures and validated, with reasonable defaults being filled in for missing values.

3.3.2 Load necessary kernels

There are quite a few OpenCL kernels included with this project; since the program now knows which ones it needs, it can now instantiate one kernel for each force, as well as one copy of the interpolator kernel.

This is done by making a copy of the appropriate master kernel, all of which are loaded into a kernel library and compiled at the beginning of the process.

3.3.3 Randomly generate default particles

Some particle systems have an initial set of particles — some, because they don't include any emitters, others, because they simply want to start with some particles displayed. At the moment, the only parameter of the initial particles that can be controlled is the number; they are placed randomly in the unit cube with zero initial velocity. Eventually, it would be ideal to support manipulation of the range and location of random positions, as well as of the mass and initial velocity.

3.3.4 Evaluate emitters

Evaluation of emitters primarily involves adding new particles at the location of the particle and giving them an initial velocity (depending on the emitter's settings).

Emitters are evaluated on the CPU (instead of the GPU). This is done primarily because of a limitation in OpenCL where one cannot resize a buffer from inside a kernel — so, there would be no way to allocate space from inside an emitter kernel.

One key performance consideration while evaluating emitters is how many new particles to allocate space for. If we're given a constant emitter, we don't want to allocate *just* enough space during each frame to fit the new particles, because then we'd have to allocate more space on each new frame. Instead, we allocate enough space for 10 frames of said emitter plus 1024^5 extra particles, which seems to stave off the performance-crushing problem of constant buffer reallocation, at least for the systems we've tested.

3.3.5 Evaluate forces

At this point, all particle data is copied to the GPU. Each force's kernel is instantiated once for each particle, and many such kernels can operate on their respective particles in parallel.

Each instance of the force kernel computes its own contribution to the acceleration of the particle it affects, adding the result to that particle's global acceleration accumulator.

A sample force — pushing outward on all particles from one direction, with falloff — is included in figure 4.

3.3.6 Integrate positions

Our simulation uses a form of Verlet integration from [Dummer 2009], which is theoretically much more stable than basic Euler integration (which was used during the project's inception) in order to apply the acceleration values to the particles positions. This works by keeping around the previous position of the particle, instead of a velocity value:

$$p_{next} = (p - p_{last}) + (a_p * t^2), \tag{1}$$

where t is the timestep and a_p is the acceleration due to all of the forces on the given particle.

3.4 Preview

The preview subsystem takes the output from the simulator and uses an OpenGL view to draw all of the particles as uniformly sized red dots, as you can see in figure 5. This rendering is very rough but also very quick — fast enough to be real-time, in many cases.

Preview rendering should be used when designing particle systems, as it provides nearly instant — though relatively unattractive — feedback.

3.5 Rendering

Our renderer is currently very simple: it takes the particle data from the simulation phase and turns it into a greyscale image. It does this by iterating over each pixel of the output image, mapping the pixel's location into 'particle space', then searching through all of the particles, finding those which are within ϵ units of it, and updating the output image. Particles are orthographically projected onto the plane

[5]This is completely arbitrary, chosen by random dice roll!

```
__kernel void force(
    __global PAPhysicsParticle * input,
    __global PAPhysicsParticle * output,
    __global PAPhysicsNewtonian * newtonIn,
    __global PAPhysicsNewtonian * newtonOut,
    __global PAPhysicsForce * force,
    const unsigned int count)
{
    float4 fpoint, loc, accel, uv;
    float dist;

    int id = get_global_id(0);

    if(id > count || input[id].lifetime == 0)
        return;

    // Load location of force and current point
    loc = (float4)(input[id].x, input[id].y,
                   input[id].z, 0.0f);
    fpoint = (float4)(force->particle.x,
                      force->particle.y,
                      force->particle.z, 0.0f);

    // Compute acceleration on particle
    uv = normalize(loc - fpoint);
    accel = force->data.normal.strength * uv;
    dist = distance(loc, fpoint);
    accel = accel * (1.0f / powr(dist + 1.0f,
        force->data.normal.falloff.strength));

    // Accumulate acceleration on particle
    newtonOut[id].ax += accel.x;
    newtonOut[id].ay += accel.y;
    newtonOut[id].az += accel.z;
}
```

Figure 4: A simplified version of our 'normal' force kernel, which pushes outward on particles from a point, with falloff

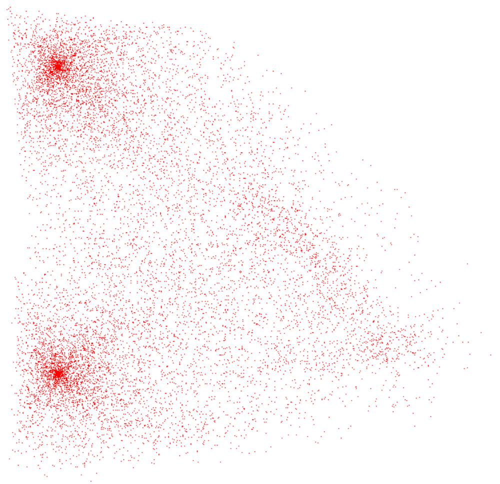

Figure 5: Example frame of preview output (simple gravity system); approximately 15,000 particles

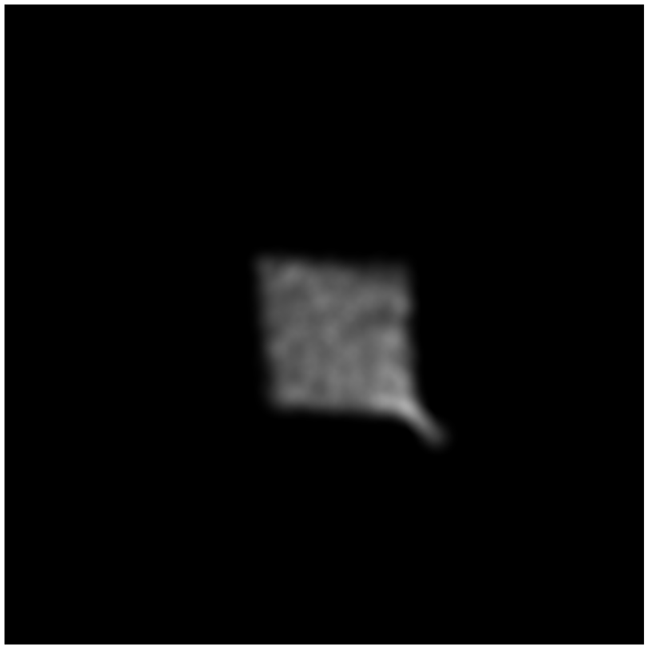

Figure 6: **Smooth** rendering mode

of the image, and — at the moment — depth information is completely ignored.

For the purposes of all of the systems contained in this paper, $\epsilon = 0.1$.

There are two main rendering modes at the moment: **smooth**, and **cloudy**.

Cloudy simply increments the luminance of the pixel for each particle it finds that's within ϵ units, leading to 'circles of influence' around each particle, as seen in figure 1.

Smooth weights the increase in luminance by the distance to the particle, according to $(0.1 - dist) \cdot k$, as seen in figure 6. k is an 'exposure' constant, and is determined by the number of particles likely to be near each other at any given time.

We've asked numerous people which of the two rendering modes they prefer, and have come up almost completely split, so they're both included for the time being.

4 Performance & Results

4.1 Hardware

All benchmarks and performance tests were taken on a machine running Mac OS X with a Core 2 Duo E7200 at 2×2.53GHz with 8GB of RAM and an ATI Radeon 4890 at 800×850MHz with 1GB of VRAM.

4.2 Benchmarks

Figure 7 shows the relative frame-rates of three different particle systems which we've been using as sample systems on the CPU versus the GPU.

The **n-body** system is a simulation of 16,384 particles all interacting with each other, simulating gravity. There are no emitters in this system, and only one force.

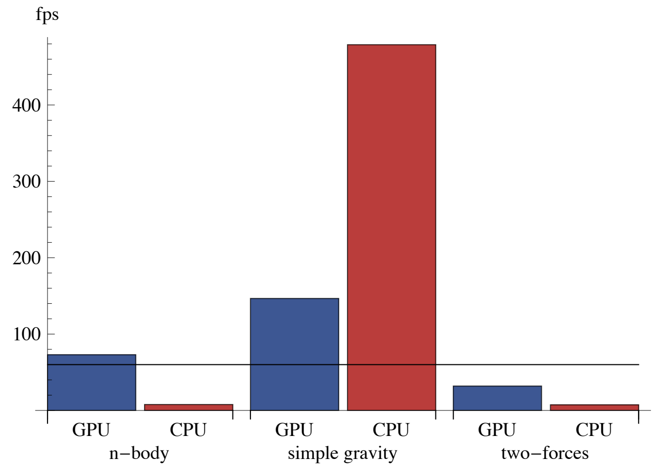

Figure 7: Simulation frame-rate of various different systems on CPU vs. GPU; higher values are better; the horizontal line rests at 60 fps

The **simple gravity** system (also pictured in figure 1) is a simulation with two emitters — which emit 20 particles per frame each — and three equal-strength gravity wells (in the top right, top left, and center). The gravity simulation in this system is much simpler than in **n-body**, as the particles don't interact with each other, just with the three wells.

The **two-forces** system is a simulation with 1,048,576 particles and two outward-acting normal forces placed equidistant from the center on a diagonal through the random cloud of particles.

You'll note that **n-body** shows a drastic speedup when moved to the GPU — almost 10x! This is primarily because the n-body force is perfect for parallelization, as it has a long per-particle runtime (each particle has to iterate over all of the other particles, since we lack a spatial hashing data structure).

Something similar is true for **two-forces**: while it's not actually nearly as computationally intensive per-particle, there are such a sheer number of particles that the parallelism comes into play in a big way, providing a less-impressive-but-still-significant 4× speedup.

The **simple gravity** system shows the weakness of our approach: since emitters are evaluated on the CPU no matter what (because of the infeasbility of resizing buffers on the GPU discussed in section 3.3.4), and incur an additional pair of copies to/from the GPU, the CPU is able to significantly outpace the GPU. This makes it clear that emitter-heavy systems with few expensive forces are *not* well suited for our approach.

4.3 Video

Both the preview and rendering subsystems have the ability to output a PNG file for each simulated frame. These can be recombined and encoded with most video editing tools into a usable format.

5 Applications

The most likely application for GPU-side particle systems is games; however, our approach isn't targeted at games — primarily because we copy the particle data back to the CPU, something unnecessarily expensive for a live game. Instead, our approach would more likely be useful to augment the current particle system tools within Blender, Apple's Motion, or Adobe's After Effects, all of which are meant for semi-offline work where the particle data is at some point needed for more generic computation on the CPU.

All three of these tools currently have advanced particle systems, but it seems reasonable, given the speedup demonstrated within this paper, that they might benefit significantly by offloading at least some of their physical simulation to the GPU.

6 Future Work

6.1 Spatial Hashing

One improvement which would massively improve the performance of both the renderer and some of the forces would be the inclusion of a spatial hashing mechanism.

For example, the renderer currently iterates over all of the particles to find the few which are very close to the pixel currently being rendered — this operation is currently implemented in the most naïve way possible, which is $O(n^2)$. The implementation of a kd-tree would make this $O(n^{2/3})$ instead — quite a significant improvement, though at the expense of a measurable increase in complexity.

Alternatively, [Drone 2007]'s "force splatting" approach might make even better use of the optimized hardware available, though it's unclear to me that the summing operation (which their approach improves) is actually the primary bottleneck.

6.2 Design Tools

The design tools need a significant amount of work before they're usable. Most importantly, a method for exporting curves to files which the simulator can import and use to manipulate properties must be implemented.

Additionally, there's no rulers on either axis in *Interpolator* — these would be necessary before one could even imagine using it as a design tool.

Really, the time constraints on this project made the implementation of both *Interpolator* **and** the remainder of the project very unlikely, and, as such, there's a lot left to do. We're actually planning on continuing work in our free time over the summer, as this has turned out to be an eye-opening project.

6.3 Rendering Improvements

In the future, the renderer could be improved visually through a few means. Firstly, it would be a significant advantage to be able to support color image buffers — this way, various sorts of information could be represented. For example, depth data could be encoded in one of the color channels, providing an easier way to understand the data. In addition, the ability to move the camera (and script the camera's movements with *Interpolator* curves) would make for much-improved visualizations, but would also require significant additional complication. However, we were planning on implementing raytracing through the particles in order to provide self-shadowing and other useful features, which would also imply the ability to move the camera.

7 Conclusion

After consuming quite a bit of processing power doing experiments, and after writing a few thousand lines of code, I think it's safe to say that we've introduced a feasible implementation of very-parallel particle simulation on the GPU. It's also quite clear that it's very possible to use this technology in order to speed up highly-physically-accurate simulations, especially ones with high algorithmic complexity, like the n-body problem.

It's also clear that our implementation is not the be-all-end-all of particle simulators. There are many missing cases, and a few conditions in which our simulation actually negatively impacts performance, but it seems that an intelligent algorithm could certainly switch between target devices depending on the nature of the system, massively improving performance for many cases.

We would have liked to have an extra month or two to work on this; it's likely that such time would have made for a much more interesting final result... alas, a semester is short, so we only had time for minor successes. There's always the summer...

References

Drone, S. 2007. Real-time particle systems on the GPU in dynamic environments. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, ACM, New York, NY, USA, 80–96.

Dummer, J., 2009. A simple time-corrected verlet integration method. `http://www.gamedev.net/reference/programming/features/verlet/`.

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.

Reeves, W. T. 1983. Particle systems—a technique for modeling a class of fuzzy objects. *ACM Trans. Graph. 2*, 2, 91–108.