Plasma Visualization in Parallel using Particle Systems on Graphical Processing Units

T.S. Lyes and K.A. Hawick
Computer Science, Massey University, North Shore 102-904, Auckland, New Zealand
email: \{t.s.lyes, k.a.hawick\}@massey.ac.nz
Tel: +64 9 414 0800 Fax: +64 9 441 8181

April 2013

ABSTRACT

Visualising and simulating charged plasma systems present additional challenges to conventional particle methods. Plasmas exhibit multi scale phenomena that often prevent the use of standard localisation approximations. Plasmas as particle systems that emit light are important in many interesting components of games, computer animated movies such as weapons fire, explosions, astronomical effects. They also have intrinsic value for simulating physical phenomena. We describe the use of various shader and texture methods to render a simulated plasma system based on explicitly charged particles. We report on attainable renderings and on coding approaches and performance using graphical processing units.

KEY WORDS
plasma simulation; special effects; particles; electrostatic charge; rendering.

1 Introduction

Plasmas present additional challenges for their simulation and visualization over and above those normally associated with particle systems. The key physics characteristic of a plasma is that it is an ionized gas [1]. Ionization can occur by applying extreme heat, pressure or electric discharge (such as a strong magnetic field). Plasma contains charged particles called ions; these can be either positively or negatively charged. This makes plasma strongly responsive to electromagnetic fields and electrically conductive. Such properties make plasma different to those of solid, liquid and gaseous states of matter. Our sun is an example of a large plasma system, but closer to earth the Aurora Borealis, ionosphere and neon signs are all examples of plasmas. Plasma physics is also an important part of fusion energy research [2].

Plasma simulation is often described as multi-scale or multi-level. This refers to the fact that plasma systems behave on a wide range of different lengths and time scales [3] [4]. This makes plasma systems difficult to simulate when trying to include all the relevant physics, thus, approximate models are used with trade-offs between accuracy and computational efficiency. A number of localisation approximations are common place in molecular dynamical simulations [5], but the multiple scales in plasma systems generally require a different simulation approach.

There are two main approaches to computational modeling of plasma systems - a fluid approach and a kinetic (particle) approach. Fluid approaches such as hydrodynamics or magnetohydrodynamics (MHD) are the most popular, but are inaccurate when more detailed kinetic processes (for example, particle interactions) affect the behaviour of the plasma. Kinetic simulation approaches can model plasma over larger ranges of density and temperature [6] and are more accurate, but are computationally expensive [7]. One way of dealing with this issue is the use of hybrid modeling techniques featuring elements of both fluid and particle systems which compromise between computational effectiveness and result precision [7].

The Particle-in-Cell (PIC) approach is an example of a kinetic approach to plasma simulation [8, 9]. Other well known kinetic methods include the Vlasov and Fokker-Planck methods [4]. Object-oriented methods of the Particle-in-Cell approach have proven successful in modeling plasma systems to a good degree of accuracy while also minimizing performance issues. VORPAL [10] is a plasma simulation code designed using an object-oriented style of PIC (OOPIC), while [2] investigates methods of parallelizing OOPIC systems in a variety of different programming languages. Other models treat plasma as a 6-dimensional phase-fluid [6], which are more complex than typical MHD models. Kinetic modeling has also been used in the simulation of interactions between plasma and pulse laser systems [11], and particle-based methods have been successful in simulating plasma systems relating to the motion of blood cells [12].

In graphics, plasma simulations can be an eye-catching edition to any video game or movie. In these situations, accuracy of physics can be approximated further as the plasma
systems need only to appear to behave correctly. However, the problem still remains of simulating a system which is accurate (visually) while also being computationally expensive. Parallelizing the behaviour of the plasma system can significantly improve the computational performance of the system. Graphics Processing Units (GPUs) [13] are excellent devices for simulating highly-parallel systems such as particles [14]. This paper uses a particle-based plasma system to simulate a ball of plasma. It uses NVidia’s Compute Unified Device Architecture (CUDA) [15] to parallelize the plasma code and inter-operate with OpenGL [16] to render the system in real time using a variety of rendering techniques. In Section 2 the core functionality of the plasma system is described and the main equations explained. Section 3 gives the results of the simulation, showing the different rendering techniques used. Section 4 presents a discussion of the performance of the program on two different GPU graphics cards as well as using different rendering setups, and performance statistics are presented in a variety of tables. Finally Section 5 concludes the project and offers some ideas on future work in the area.

2 Simulation Method

Our primary objective in this present article is to present the visual rendering approaches for charged plasma systems. We therefore do not dwell on time-integration algorithms and other simulation details such as finite differencing algorithms [17], but we do however summarise the main features that differentiate plasma simulations from other ad hoc potential particle simulations.

A plasma system can be simulated by using a particle system. Particles in the system represent the positive ions and negative electrons making up the plasma. A positive charge will attract a negative charge, while both positive or negative charges will repel each other. In this particular system, the plasma is contained in a bounding sphere as shown in Figure 1. Without some sort of container, a plasma system does not have a particular shape but just spreads as the ionized gas that it is.

There were two main equations used for simulating the plasma particles, besides the standard position and velocity time step functions present in almost all particle systems. Firstly, the equation for calculating the electrostatic charge, or Coulomb’s Law (see Equation 1) is used to calculate the attractive force between any two particles at one time. For simulation purposes, constants such as $\epsilon$ had to be altered slightly such that the system was able to be observed in real time. In any case, the equation implies the electrostatic force is determined primarily on the distance $r$ between the two particles in question (a smaller distance means a stronger force), while the charges of the two particles determine whether the force will attract or repel the particles.

$$F_{i,j} = \frac{q_i q_j}{4\pi\epsilon_0 r_{i,j}^2}$$

Secondly, the energy of each particle is given by the formula in Equation 2, which is the equation for kinetic energy of a particle. For the purpose of the simulation, it can be assumed that all particles in the system are the same mass. Additionally, no external forces (including gravity) are applied to the system during the simulation. The total energy of the system

$$E_{i} = \frac{1}{2} m_i v_{i}^2$$

Figure 1: A spherical shell is used to contain the particles used to simulate the plasma body.

Figure 2: A plasma system simulation of 32768 particles showing the energies that are the result of collisions.
must remain constant, so the potential energy of each particle is converted into kinetic energy during collisions and attractions, and this change in kinetic energy can be used to colour the simulation - a faster moving particle will be brighter than a slower moving one.

\[ E_k = \frac{1}{2} m v^2 \]  \hspace{1cm} (2)

The particular particle system used to model the plasma in this paper had several important properties. Firstly, unlike some other particle systems, particles in the plasma system cannot be created or destroyed beyond the initialization of the system. This is essential for maintaining some degree of equilibrium in the system and makes sure the particles do not break free from the confining boundaries. Secondly, particles positions and velocities are known and updated every time step of the simulation, and an attraction force is applied to each particle depending on its own electric charge in relation to other particles around it. The simulation is not treated as being run in a vacuum - therefore, some damping force is applied every time step. This is a trivial aspect to the simulation, as if we were to simulate a plasma system such as the Sun, such a system does occur in a vacuum (space). This particular system is contained with a spherical boundary. Particles colliding with the boundary are bounced back into the system with some damping force applied as well. Finally, particles are coloured depending on the rendering method used in the simulation.

Typically this would involve each particle belonging to a different non-neighbouring cell in the system, thus when computing collisions between neighboring cells, the interaction would never be established. This occurs for both attracting and repelling particles.

As with most particle systems, the system requires a large amount of particles for it to believably resemble a real plasma system. This can become problematic when requiring calculations of energy and charge between each and every particle in the system, resulting in often hundreds of millions of computations (order N-squared) every time step of the simulation. This issue can be solved by parallelizing parts of the system using CUDA. The update functions for each particle can be parallelized and run simultaneously, but the largest speedup in performance may come from using CUDA to sort the particle arrays in such a way that they need not communicate with every particle in the system when checking for collisions - only the particles within some area of the current particle need to be checked. While this sorting functionality does require additional time to sort the arrays, it makes up for this by dramatically decreasing the execution time of the collision checking functions. Specifically for this system, three main CUDA kernels were used; one for updating position and velocity arrays, one for sorting the arrays, and one for handling collisions between the particles. A generic overview of the algorithm used to update the plasma system is shown in Algorithm 1.

### Algorithm 1: A general plasma system update algorithm

```plaintext
for all timestep do
    calculate position hash table
    sort particles
    calculate cells
    for all particles in cell do
        calculate collisions
    end for
    for all particles in system do
        update positions, velocities
    end for
end for
```

Further performance improvements can be made using the interoperability functionality with CUDA and OpenGL [15]. This involves using vertex buffer objects (VBOs) to store data and render it directly on the graphics card, thus avoiding the overhead of copying the data to and from the device every time step. This particular simulation keeps both the positions and colours of the particles in VBOs, using a vertex array and colour array respectively. The velocity of the particle does not need to be kept in a VBO as it is not a visual aspect of the simulation, rather, a behavioural aspect. However, the velocity is still taken into account when rendering the particles based on the energy output - the colour of the particles is determined primarily on their velocity, so this will be used to populate the colour VBO.
The simulation was run using a series of different rendering methods to observe the performance changes in each. Firstly, the system was rendered using texturing, with the billboarding technique used similar to [18]. This method did not make use of the OpenGL-CUDA interoperability. The particles were coloured according to the charge of the particle - white if positive, and blue if negatively charged. A second rendering method used the particles calculated energy to colour the particles. Additionally, two shaders [19, 20] were used to render the system, one a spherical shader with depth perception, and one a sprite shader with blending. All simulations were run on a NVidia Quadro 4000 GPU and also on a newer GeForce GTX 680.

3 Performance Results

A range of system sizes were tested for the simulation. Frame rates were monitored and displayed on screen in real time and were also averaged over several thousand iterations for each rendering technique used. Comparisons were made between the different techniques regarding both their visual and computational performance.

Figure 4 shows the plasma system rendered using non-VBO texturing and the billboarding technique. This system size was 8192 particles - this system size was found to produce an acceptable level of frames per second for the texture rendering. The particles are coloured according to the charge of the particle - white if the charge is positive, and blue if the charge is negative. It is mainly used to demonstrate the distribution of charge throughout the entire system. During the initialization phase of the simulation particles are assigned a random charge value, irrespective of the particles position in the system.

Using a similar method, Figure 5 renders the plasma system as textures but this time they are coloured according to the energy of the particles from the collisions. The brighter the particle, the more energy it contains. Particles nearer the center of the system are far brighter than those around the edges, as they are constantly colliding (attracting) to multiple other particles. This becomes more apparent in Figure 2, where a much larger system (32768 particles) is rendered in the same way. Such a large system impacts a lot on the performance of the program, both in computational time and rendering time.

Figure 6 is a histogram showing the speeds of the particles of a 8192 particle system over 100 sample tests. This exhibits the expected Maxwell-Boltzmann thermal distribution.

Figure 7 shows the plasma system rendered using a spherical shader and VBOs to take advantage of CUDA-OpenGL interoperability. Using spheres it is easier to gauge the exact positions of individual particles due to the depth perception that is enabled as well as the particles not needing to be blended, however it does not look particularly realistic compared to other simulations. It is worth noting that spherical shaders produced the fastest frames per second results of all the rendering techniques tested.

A different shader was used in the simulation shown in Figure 8. The results are similar to those shown in Figures 4 and 4, despite being rendered using VBOs. While the average frame rate of the simulation while using this rendering technique is similar to other techniques, when the system is moved further away from the camera the FPS greatly increases (in
Figure 6: Graph showing the velocity distribution of particles over some frames.

Figure 7: A plasma system simulation using a spherical shader to render the particles.

other methods, this does not happen). This suggests this rendering technique would be best suited for rendering objects which are far away (for example, a sun in the sky) to get the best results.

4 Discussion

Performance was monitored by timing the kernel execution and averaging it over 10,000 executions. The mean execution times were taken for both cards and used to show the comparison between the Quadro card and the newer GeForce graphics card. The three kernels were not timed individually. Additionally, performance regarding the graphical frame rate was also monitored to observe the performance differences between the different rendering methods. Frame rates were displayed in real time as the simulation was run, as well calculating a mean frame rate over 10,000 executions. Statistics were collected for four system sizes; 4096, 8192, 16384 and 65536 particles.

<table>
<thead>
<tr>
<th>No. of Particles</th>
<th>time(Quadro) seconds</th>
<th>time(GTX 680) seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>0.0064</td>
<td>0.00348</td>
</tr>
<tr>
<td>8192</td>
<td>0.0153</td>
<td>0.00581</td>
</tr>
<tr>
<td>16384</td>
<td>0.0491</td>
<td>0.01406</td>
</tr>
<tr>
<td>65536</td>
<td>0.6541</td>
<td>0.14542</td>
</tr>
</tbody>
</table>

Table 1: Average kernel execution times for a plasma system of various sizes

Firstly, it is important to note that the system speeds up the
further the particles drift apart - this is because the particles will not need to perform the attraction collision functions as often if they are spread out, as there are less particles in the immediate vicinity and surrounding cells. When the simulation is first started, the particles are not distributed evenly within the bounding sphere - rather, they are distributed closer to the center, and gradually expand outwards as the simulation progresses. The times observed will therefore be slower than expected from a system in an almost equilibrium state, however they are more closely representative of an active plasma system which is constantly moving, which is also a more realistic system.

Table 1 shows the average execution times for all kernels for each particle system size, tested for both the Quadro 4000 card and the GeForce GTX 680 card. It can be shown that the execution times of the kernels increase exponentially as the size of the system increases. The render method used was the spherical shader method.

<table>
<thead>
<tr>
<th>No. of Particles</th>
<th>Avg FR Quadro (frames/ sec)</th>
<th>Avg. FR GTX 680 (frames/ sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>188.4</td>
<td>332.5</td>
</tr>
<tr>
<td>8192</td>
<td>73.6</td>
<td>205.7</td>
</tr>
<tr>
<td>16384</td>
<td>21.2</td>
<td>87.4</td>
</tr>
<tr>
<td>65536</td>
<td>1.4</td>
<td>6.9</td>
</tr>
</tbody>
</table>

Table 2: Average frame rates of a particle system of different sizes

Table 2 compares the frame rates observed for various plasma system sizes using both the Quadro and GTX 680 graphics cards. As mentioned earlier, the system speeds up as particles spread out, when there are less collisions per cell. This effects the average frame rate as well. Because they are averaged over this time frame, the observed FPS in tables 2 and 3 will be faster than when the system is first initialized, and slower than when the system has reached an almost equilibrium state. It was found that the optimum system size for the Quadro card was the 8192 particle system size, however the newer GTX 680 could easily render the system 16834 particle system.

Table 3 shows the average frame rates observed when using each rendering method. The simulation was restricted to run only using a system size of 8192 particles, as from previous tests it was decided that this system size gave the best result for the Quadro card, considering the simulation needed to be run in real-time at a realistic speed (larger systems run too slow), and maintain a realistic looking simulation as well (too few particles would not achieve this). The method using VBOs and a spherical shader produced the fastest FPS time. Surprisingly, although the sprite texture rendering method used VBOs as well, it did not initially perform as well as its sequential counterparts. However, it was found that, while rendering using this method, moving the camera away from the system increased the frames per second by a large amount.

Moving the camera back by a factor of 5 increased the FPS of the system from 39.8 to 76.5 on the Quadro card, and from 107.3 to 194.8 on the GTX 680. In the case of the Quadro card, this new averaged FPS was actually faster than that of the spherical shader. This was probably due to the fact that the sprite shader took into account the position of the camera when scaling the sprites it used to represent the particles; the closer the camera, the larger the sprites would need to appear, and thus the more pixels were needed to render. Interestingly enough, none of the other rendering methods (including the spherical shader) got FPS increases when moving the camera back. As mentioned previously, this observation suggests that the sprite pixel shader would be an ideal method to use when rendering objects that would need to appear far away (such as a sun on the horizon or in space).

What was also noticeable was the substantial increase in speed when using shader rendering methods on the GTX 680 compared to the other methods. Specifically, using the GTX 680 with textures resulted in an increase in FPS of 60.4 percent, while VBO spheres increased by 179.5 percent and VBO sprites by 169.6 percent. This suggests that these rendering methods are better designed for parallelization and up-scaling of the system in general.

There is scope to incorporate stereo rendering of the particles and plasma cloud more generally. The framerates are adequate and therefore with additional GPU hardware it is feasible to attain the necessary framerate doubling to render the system in stereo [21, 22]. This has the potential to aid the visualisation considerably.

5 Conclusion

A plasma system of charged particles was simulated using a particle system. OpenGL was used to visualize the simulation in real time, and CUDA was used to parallelize the system. The performance of the system was analyzed for various sizes or numbers of particles, and results were also compared between two different cards, the Quadro 4000 professional graphics card and a newer GeForce 580 graphics card. Different methods were used to render the plasma and were evaluated to determine trade-offs between the computational and
visual performance of the system. These methods included using VBOs and shaders to render the plasma.

A texture shader or sprite shader gave the best results visually, while the spherical shader resulted in the faster frame rates. The sprite shader, while initially having the slowest rendering speed, performed better when the camera was further away from the system, and this was not the case for the other rendering techniques. When rendering this particular plasma system on the Quadro card, it was found that a system size of 8192 particles was best when considering both the computational performance and visual aspects of the system. The GeForce GTX 680 card could handle a system size of at least 16384 particles in real time, however a system size of 65536 particles could not be rendered in real time by either card.

There is scope to extend this work. Writing new shaders specifically designed for representing a plasma system could result in great improvements to the simulation both computationally and visually. With regards to the behavioural mechanics of the system, the current simulation does not deal whatsoever with electromagnetic fields or fluxes. Plasma systems are known to respond strongly to electromagnetic fields and simulating this would be quite interesting. For example, having points on the bounding sphere acting as electric field points or introducing an electric field in some other way might lead to some interesting behaviour from the plasma system, such as the creation of beams or other complex behavior such as a sun’s corona.

References