Fire and Flame Simulation using Particle Systems and 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

February 2013

ABSTRACT

Simulating fire, flames or other natural phenomena can be difficult because of the inherently complex systems used to model them, while also requiring an adequate amount of realism visually. Simulating such a system in real-time can also be a problem if the system is too large, so a parallel computing techniques can be used to good effect. Particle systems have been shown to simulate flames and fires particularly well at relatively low computational cost. We describe how a simple particle system approach can be used to simulate a fire or flame in real-time in conjunction with using data parallelization, achieving a substantial performance speed up on graphical processing units (GPUs). Using NVidia's Compute Unified Device Architecture (CUDA) and OpenGL interoperability functionality allows for further performance increases when rendering the simulation with GPUs. Additionally, different rendering techniques are used to investigate trade-offs between performance speed and visual realism.

KEY WORDS
fire; flames; visualisation; rendering; simulation; turbulence; GPU

1 Introduction

Fire and flame simulation is an interesting and important area of research in computer graphics [3]. As with most natural phenomena, it can be a challenging to simulate, particularly due to its complex, turbulent nature [12]. To simulate such a complex system in real time is difficult even by today's computing power standards. Sufficient realism is an important aspect of fire simulation as well, and thus many different rendering techniques have emerged to make the simulated fire look and behave as convincingly as possible. Fire simulation has applications in many industries, such as movie making special effects, video games and scientific visualization, as well as areas such as fire control and military emulation [19] [6].

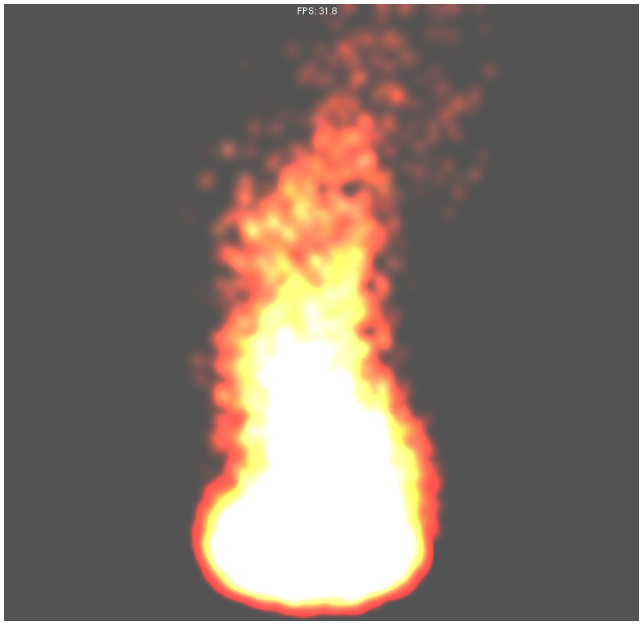

Figure 1: A fire particle system simulation

To date there have been many methods on simulating and rendering a fire or flame in real time. Some methods include using a spring-mass model to model flame kinematics [1] allowing external forces such as gravity and wind to be incorporated for added realism, or a method for rendering fire on the surface of a polygon mesh [2] by generating points on the surface of the polygon and using individual flame primitives to render the fire, or by simulating the fire as an "evolving front" of particles [9] moving across a polygonal mesh. Other methods combine simple particle systems and advanced rendering techniques [6] to generate highly detailed fire simulations at relatively low computational cost. Combustion models [17] [20] and hydrodynamics [18] have also been used to model fires.

There are many different rendering techniques to consider

when simulating a fire. One technique involves uses two dimensional sprites or "splats" [16] as textures to simulate the fire, giving the illusion of the fire being three-dimensional by using rendering techniques such as billboarding (rotating the image to always face the camera) and blending (colours in the background partially fade through to the foreground). Three dimensional techniques such as utilizing polygons or polygonal surfaces to model flames are far more complicated but can provide much more realistic results.

Particle systems are a simple but effective way of modeling a lot of complex systems, and is the core basis for many fire simulation methods [6] [9] [19]. A particle system consists of many particles sharing similar attributes such as position, velocity, and lifetime, all controlled by a specific set of rules or functions. Particles will be created and destroyed throughout the lifetime of the system. The size of the system can vary from just a few hundred particles to tens of thousands of particles, but realistic and real-time results will depend on the computing hardware used. Graphics Processing Units (GPUs) [10] can help with this limitation [13, 14], and using GPU implementations allow simulations to be only limited by the particle data transfer between the processor and GPU [7] [8].

GPU's are designed to handle problems which can be expressed in parallel [15] such as particle systems [4]. Particles can be divided up into blocks and updated in parallel as each particle's update function will typically be the same for all particles in the system. Compute Unified Device Architecture (CUDA) [11] is an extension of the C programming language developed by NVidia specifically designed for usage on NVidia GPUs. Using CUDA we can take advantage of not only the data parallelism of the GPU, but also CUDA's built-in functionality to allow it to interact with the OpenGL rendering library [5] or the purpose of rendering directly on the GPU, resulting in even better performance from the program.

This paper focuses on using a simple particle system to simulate a fire or flame, as well as using CUDA and OpenGL to simulate and render the system using various simple rendering techniques. An example of a flame rendered using particles is shown in Figure 1. In Section 2 we describe the methods and functionality used, describing the core CUDA and OpenGL concepts behind simulating the fire. Section 3 shows some screen caps of the program in action highlighting the differences in the rendering techniques. Section 4 presents a discussion of the performance of the program using different rendering setups, and finally Section 5 lays forth some concluding remarks on the project as well as some future work suggestions.

2 Implementation Method

This fire particle system uses the following properties - particles are born in a random position inside of a circular area (based on the radius and angle specified by the kernel), with an upwards velocity and a slightly deviated acceleration along the x and z axes. Wind effects increase or decrease this deviation using according to a sine wave function. The fire is coloured using a randomly generated colour when it is born as well as a randomly generated colour when it dies. Throughout its lifetime the colour will not only transition from its start colour to its end colour, but also the alpha component will also decrease. All particles are born with an alpha component of 1.0 and die with an alpha component of 0.0 - in other words, particles will fade as they grow older until they completely disappear when the particle dies.

Algorithm 1 gives a general idea of how a particle system works.

Algorithm 1 the general layout of a particle system kernel

```
for all particles in system do
  update life
  if particle dies then
    destroy particle
    create new particle
    randomize particle values
  end if
  update positions, velocities
  update collisions, etc...
end for
```

CUDA has built in OpenGL interoperability functionality which can improve performance levels [11]. To take advantage of this, the general method is to use vertex buffer objects (VBOs) to store rendering data such as points and colours so that once the system has been updated, the particles can be rendered directly using the GPU rather than passing the values back to the CPU after the kernel has been executed. This takes slightly longer preparation as CUDA needs to map resources using the functions cudaGraphicsMapResources() and cudaGraphicsResourceGetMappedPointer() before executing the kernel. Additionally, OpenGL will use glBindBuffer() to bind an array buffer to the VBO needed. For this simulation, two VBOs were used - a vertex array for particle positions, and a colour array for particle colours.

Colours can be applied to the fire in two ways depending on the rendering method - when rendering on the CPU, colours can be applied using glColor4f, using four percentage floats to determine the red, green, blue and alpha components of the colour. The fire will always have a red component of 1.0f (100 percent red) while the green and blue components will be randomized to give a more orange or yellow colour to the fire. If the simulation is rendered on the GPU using CUDA-OpenGL interoperability, colours can be applied using a colour vertex buffer object (VBO) and using the functions glColorPointer() and glEnableClientState(GL COLOR ARRAY) to bind the colour VBO to OpenGL's

```
//Initialize the curand pseudo-random number generator for the device
__global__ void setup_kernel(curandState *state){

    //Each thread gets the same seed, but a different sequence number
    unsigned int x = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int y = threadIdx.y + blockIdx.y * blockDim.y;
    unsigned int id = x + y * blockDim.x * gridDim.x;
    curand_init(1234, id, 0, &state[id]);
}
```

Figure 2: Setup kernel for initializing the CURAND pseudo-random number generator

colour array buffer.

Pseudo-random numbers are sufficient to provide randomized data values when a new particle is created. However, conventional C pseudo-random number generators will not work when used in CUDA kernels. For this reason, NVidia recently developed the CURAND random number library for use of random numbers in parallel. CURAND random numbers can be generated either on the host or device. When generating on the device, curand_init must first be run to initialize a RNG state for each particle in the system. As shown in Figure 2 when using CURAND in parallel it is recommended to use the same random number seed and a different sequence number for each particle in the system.

In this case, 1234 was used as the seed for all particles, while each particle's id number was used as the sequence number. What results is an array of "curand states" which can be passed to the kernel similar to float arrays for position or colour. The kernel can then use this state to generate as many random numbers as needed, such as in Figure 2 when randomizing a particle's starting colour and decay rate. In this example, the function curand_uniform is used, which generates a uniformly distributed number between 0 and 1, however CURAND supports many other distributions as well.

The fire simulation was run on a single NVidia Quadro 4000 graphics card.

3 Visualisation Results

The particle system was tested using a range of system sizes (64x64, 128x128, 256x256, and 512x512), or from 4096 in the smallest system tested up to 262144 particles in the largest system tested. The frame rates were also monitored and displayed on screen in real-time. Additionally, a variety of rendering techniques were used to investigate their impact both visually and on the computational performance of the program.

Figure 4 shows the initial visualization of the system with a size of 128x128. Smaller systems (64x64) did not produce visually strong simulations so those screenshots were not

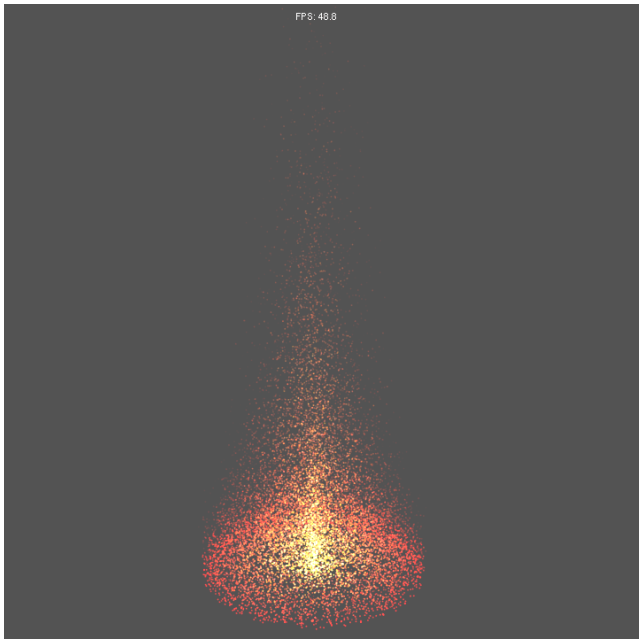

Figure 4: A simulation of the 128x128 fire particle system (16384 particles) rendered using simple GL POINTS

used. This particular simulation used simple GL POINTS to represent each particle. While not the most visually interesting rendering method, GL POINTS did give the best frame rate performance of all the rendering methods. Note that the redness of the particles increases the further away from the center they get. This was intended to better simulate a flame, however it is not always apparent in other rendering methods seen later.

Figure 5 shows the same system size rendered using the CUDA-OpenGL interoperability functionality. A wind component has also been added. The simulation is almost identical to the non-VBO rendered simulation, however one big difference is that the frame rate has increased substantially.

Figure 6 uses GL LINES to render the fire instead of nor-

```
col[id].x = 1.0f;
//Make the red component of the colour more prominent towards the outside of the flame
col[id].y = 1.0f * ((0.5 + (0.8 - 0.5) * curand_uniform(&localState))* (1-(r)));
col[id].z = 1.0f * ((0.0 + (0.5 - 0.0) * curand_uniform(&localState))* (1-(r)));
col[id].w = 1.0f;

decay[id].x = (1.0f - col[id].x) / life[id];
decay[id].y = (0.4f * curand_uniform(&localState) - col[id].y) / life[id];
decay[id].z = (0.1f * curand_uniform(&localState) - col[id].z) / life[id];
```

Figure 3: Example of using CURAND to randomize the colour and decay rate of a created particle

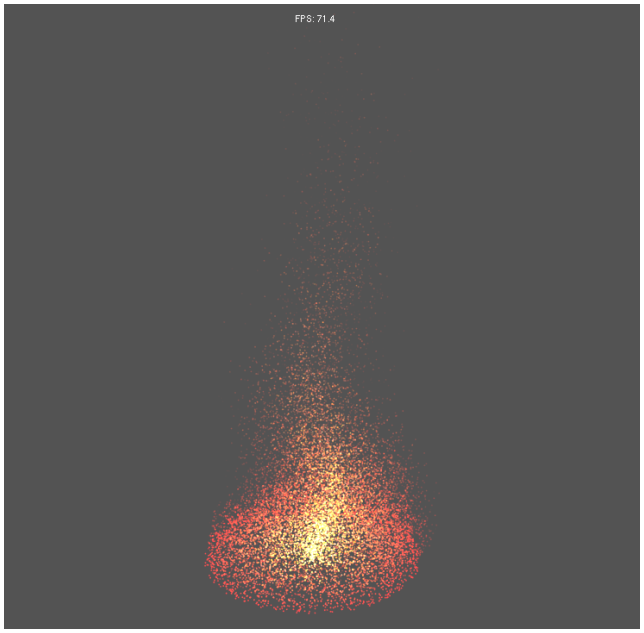

Figure 5: A simulation of the 128x128 fire particle system (16384 particles) rendered using CUDA-OpenGL interoperability (vertex and colour arrays)

mal points. This simulation was also rendered on the GPU using CUDA-OpenGL interoperability. In this simulation the flame appears to be slightly larger - this is due to the lines not taking into account the alpha component of the colour. The particles are showing up fully throughout their entire life when they should be fading away. The lines also accentuate the red colour a lot more as the outer red lines are drawn over the top of the inner yellow and white lines.

Figure 7 uses GL TRIANGLES instead of lines or points. While the colour progression looks much better than the lines or points, the shape of the flame is rather ”pointy” (similar to Figure 6). Because this simulation is rendered using polygons, GL BLEND has been enabled, however this means that the red colour on the outer particles is almost lost as there are far more lighter particles on the inside of the fire which are blended through. It is also worth noting that this simulation suffered a significant frame rate drop when compared to Figure 4 and Figure 6.

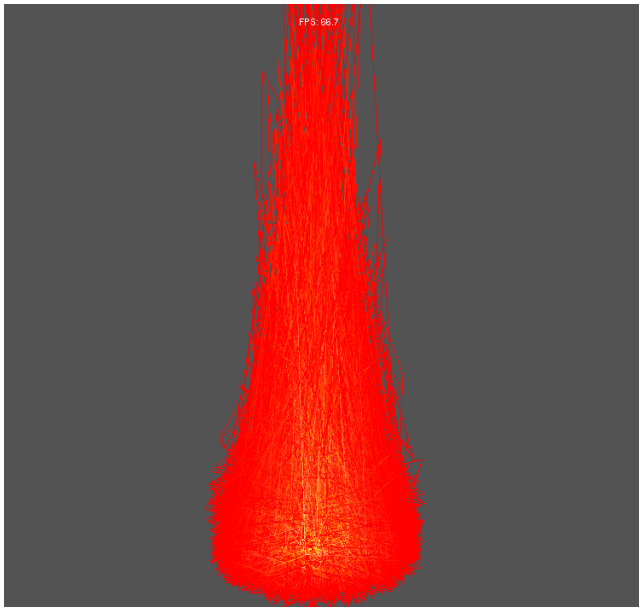

Figure 6: A simulation of the 128x128 fire particle system (16384 particles) rendered using VBOs and GL LINES

Figure 8 uses a texture mapped onto GL QUADS to represent each particle. This simulation produced the best visualization for a flame but at the cost of the lowest frame rate for all the 128x128 particle systems.

Figure 9 shows the simulation of the larger 256x256 particle system with added wind effect rendered using points. The system is much more dense which is good visually, but the system suffers from a dramatic frame rate drop when compared to the 128 x 128 system. Using more advanced rendering methods other than GL POINTS such as triangles or textures would reduce the frame rate to unmanageable levels.

Finally, Figure 10 shows the simulation of the largest system tested, 512x512 or 262144 particles. At this point, the frame rate is low enough to the point that it has started affecting the kernel executions. As a result, ”fluctuations” occur in the flame as many more particles die in between exe-

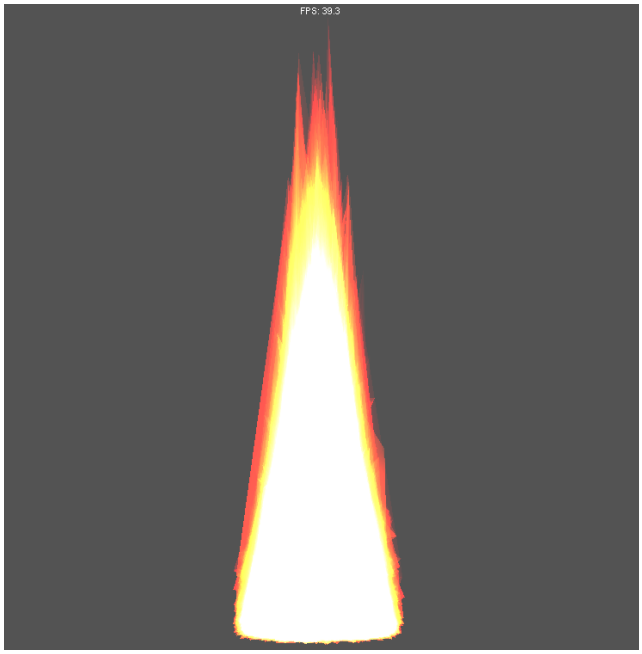

Figure 7: A simulation of the 128x128 fire particle system (16384 particles) rendered using VBOs and GL TRIANGLES

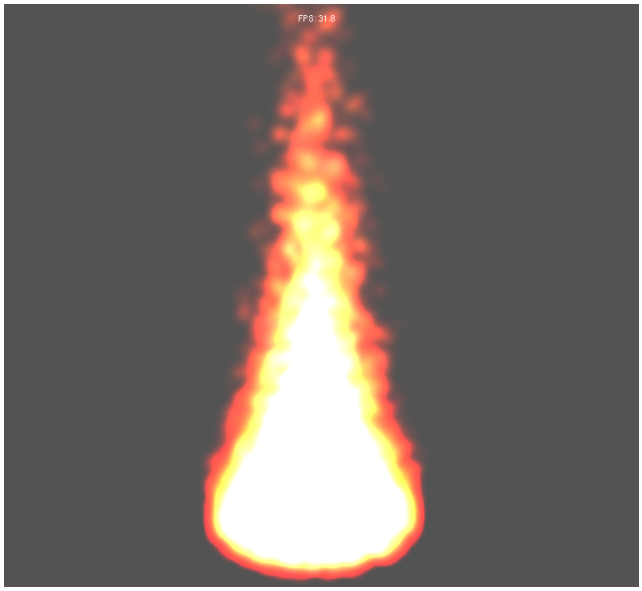

Figure 8: A simulation of the 128x128 fire particle system (16384 particles) rendered using texture maps

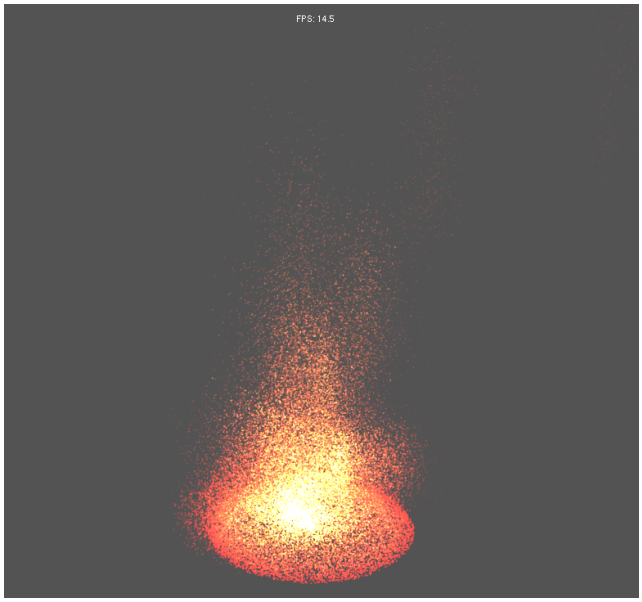

Figure 9: A simulation of the 256x256 fire particle system (65536 particles) rendered using GL POINTS

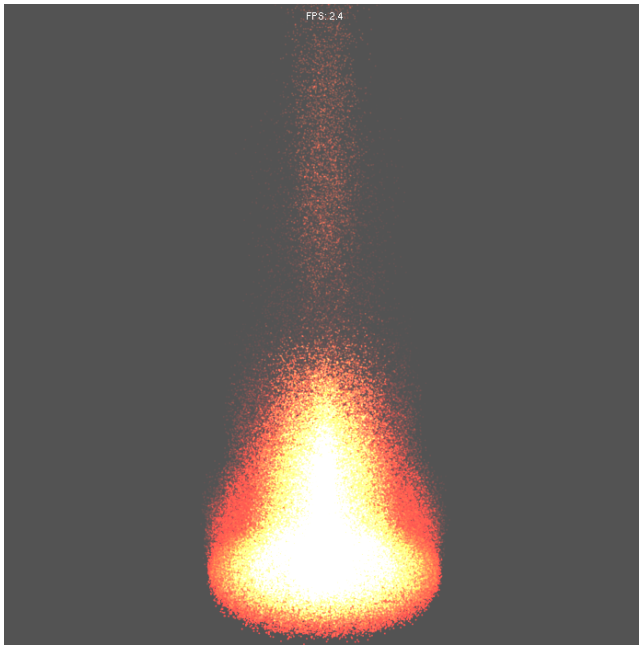

Figure 10: A simulation of the 512x512 fire particle system (262144 particles) rendered using GL POINTS

cutions of the kernel than would otherwise be expected.

4 Discussion

The performance of the system was monitored in various ways. Firstly, a comparison was made between the performance of sequential and parallel versions of the same update kernel. Performance was timed over 10,000 kernel executions and a mean kernel execution time was calculated. Secondly, performance was monitored for differences when using CUDA-OpenGL interoperability to render the simulation on the GPU. Additionally, the average frame rate of the simulation was also monitored and displayed in real-time above the fire rendering. As mentioned previously, the simulations were carried out a single NVidia Quadro 4000 graphics card.

No. of Particles	Seq. time seconds	Para. time seconds
64x64	0.0009	ca. 0.00001
128x128	0.0036	ca. 0.00001
256x256	0.0143	ca. 0.00001
512x512	0.0557	ca. 0.00002

Table 1: Performance results of a particle system of various sizes simulated sequentially and in parallel

Table 1 shows the performance results of a sequential kernel vs a parallel kernel execution. It is important to first note that the parallel execution speeds of the 64x64, 128x128 and 256x256 systems are not identical. The execution speeds of all three kernels were too fast for the precision used by the timer. This is mainly due to the simplicity of the particle code, which can be easily scaled and made far more complicated in future work. Nevertheless the parallel versions are clearly performing far better than their sequential counterparts.

An interesting point that was found was that if the parallel system synchronized its threads before the next kernel execution (using cudaThreadSynchronize) the parallel execution time was actually slower than the sequential time. This was not an issue with this simulation as thread synchronization was not needed, but in more complex systems which require it this might become a problem. Using VBOs and CUDA-OpenGL interoperability did not affect the execution time of the kernel whatsoever, simply because both methods used exactly the same kernel. However, there was a slight increase (0.0001 secs) in all update times when using the VBOs due to the graphics resource and pointer mapping and unmapping needed before and after the kernel execution. This extra time was negligible, however.

Table 2 compares the frame rates of both rendering methods on various particle system sizes. All renderings were done using OpenGL points. In all instances, rendering using VBOs and CUDA-OpenGL interoperability resulted in an improved frame rate of around ten percent. Considering there was no difference visually between the two methods, rendering using VBOs is clearly a better choice. It is important to note that NVidia Quadro cards can allow for even more performance improvements when used in a multi-GPU setup. A Quadro card performs OpenGL interoperability better than NVidia GeForce or Tesla cards. Therefore, in a multi-GPU set up it is preferential to use the Quadro card purely for rendering the simulation while using the other card or cards to perform the parallel computation parts of the program. Although this paper did not use a multi-GPU setup, it would be interesting to try this setup in the future.

No. of Particles	Avg. Norm FR frames / sec	Avg. VBO FR frames / sec
64x64	220.3	247.8
128x128	63.7	69.7
256x256	13.9	16.7
512x512	2.4	2.6

Table 2: Average frame rates of a particle system rendered normally vs using VBOs.

Render Method	Avg. Frame Rate seconds
Points	69.7
Lines	64.9
Triangles	39.3
Textures	31.0

Table 3: Average frame rates of a 128x128 particle system using different rendering methods.

Table 3 shows the average frame rates of each different rendering method using a 128x128 size particle system. In general, a more realistic looking rendering method resulted in a lower frame rate, which was to be expected. All methods resulted in an acceptable frame rate (anything below 20-25 frames per second becomes undesirable very quickly). A 128x128 sized particle system seems to be the optimal size at this point, because even using simple OpenGL points the frame rate drops to at most 16.7 frames per second and using a more complicated render method would lower this even further.

5 Conclusions

A simple particle system model was used to simulate a dynamical fire or flame, using OpenGL to visualize the simulation in real-time. Significant performance increases were observed when a CUDA parallel approach to the system was used, as opposed to a traditional sequential model. Several rendering techniques for the fire were investigated, each

with certain trade-offs in performance and visual aspects. We have presented screenshots showing the major effects and tradeoffs that we found and explored.

Using simple points to represent particles resulted in fast frame rates, while using textures simulated a more visually appealing flame at the cost of a slower frame rate. Using CUDA-OpenGL interoperability increased the potential rendering performance across all rendering techniques by around 10 percent (measured in average frames per second). It was also found the ideal particle system size for the rendering techniques and kernels used was 128x128 or 16384 particles, as it provided the best trade-off between speed and visual appeal.

Future work in this area would be very interesting, having a look at other more complicated rendering techniques, as well as more complex kernels in order to simulate a more accurate and realistic fire or flame. As mentioned in Section 4, running this simulation on a multi-GPU set up to take advantage of the superior Quadro CUDA-OpenGL interoperability should also be possible in the near future. Other complex natural phenomena such as fluids have also simulated using particle systems, and this is also an area that could benefit from the techniques described here. Fountains or other structured particle model systems could be implemented using these techniques. More generally, visualising and modelling other plasma based systems is potentially of interest for both the gaming and movie industries where realistic physically based models of these complex systems can aid in attaining enhanced realism.

References

[1] Balci, M., Faroosh, H.: Real-time 3d fire simulation using a spring-mass model. In: Proc. 12th Int. Multi-Media Modelling Conference. pp. 108–115. Beijing, China (2006)

[2] Beaudoin, P., Paquet, S., Poulin, P.: Realistic and controllable fire simulation. In: Proc. Graphics Interface (GRIN'01). Toronto, Ontario, Canada (2001)

[3] Foley, J.D., van Dam, A., Feiner, S.K., Hughes, J.F.: Computer graphics: principles and practice (2nd ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1990)

[4] Hawick, K.A., Playne, D.P., Johnson, M.G.B.: Numerical precision and benchmarking very-high-order integration of particle dynamics on gpu accelerators. In: Proc. International Conference on Computer Design (CDES'11). pp. 83–89. No. CDE4469, CSREA, Las Vegas, USA (18-21 July 2011)

[5] Hearn, D., Baker, M.P.: Computer Graphics with OpenGL. No. ISBN 0-13-015390-7, Pearson Prentice Hall, third edition edn. (2004)

[6] Horvath, C., Geiger, W.: Directable, high-resolution simulation of fire on the gpu. ACM Trans. on Graphics 28(3), 41–1–8 (August 2009)

[7] Kolb, A., Latta, L., Rezk-Salama, C.: Hardware-based simulation and collision detection for large particle systems. In: Proc. Graphics Hardware (2004)

[8] Latta, L.: Building a million particle system. In: Game Developers Conference (2007)

[9] Lee, H., Kim, L., Meyer, M., Desbrun, M.: Meshes on fire. In: EG Workshop on Computer Animation and Simulation. pp. 75–84 (2001)

[10] Leist, A., Playne, D.P., Hawick, K.A.: Exploiting Graphical Processing Units for Data-Parallel Scientific Applications. Concurrency and Computation: Practice and Experience 21(18), 2400–2437 (25 December 2009), CSTN-065

[11] NVIDIA® Corporation: CUDA™ 3.1 Programming Guide (2010), `http://www.nvidia.com/`, last accessed September 2010

[12] Peyret, R., Taylor, T.D.: Computational Methods for Fluid Flow. Springer Series in Computational Physics, Springer-Verlag (1983)

[13] Playne, D.P., Hawick, K.A.: Classical mechanical hard-core particles simulated in a rigid enclosure using multi-gpu systems. In: Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'12). pp. 76–82. CSREA, Las Vegas, USA (16-19 July 2012)

[14] Playne, D.P., Johnson, M.G.B., Hawick, K.A.: Benchmarking GPU Devices with N-Body Simulations. In: Proc. 2009 International Conference on Computer Design (CDES 09) July, Las Vegas, USA. pp. 150–156. WorldComp, Las Vegas, USA (13-16 July 2009)

[15] Wei, W., Huang, Y.: Real-time flame rendering with gpu and cuda. Int. J. Info. Tech and Computer Science 1, 40–46 (2011)

[16] Wei, X., Li, W., Mueller, K., Kaufman, A.: Simulating fire with texture splats. In: Proc. IEEE Conf. on Visualization (VIS'02). Bostan, MA., USA (27 October - 1 November 2002)

[17] Xue, H., Ho, J.C., Cheng, Y.M.: Comparison of different combustion models in enclosure fire simulation. Fire Safety Journal 36, 37–54 (2001)

[18] Zhang, F., Hu, L., Wu, J., Shen, X.: A sph-based method for interactive fluids simulation on the multi-gpu. In: Proc. ACM SIGGRAPH Int. Conf on Virtual Teality Continuum and its applications in Industry (VRCAI'11). Hong Kong, China (11-12 December 2011)

[19] Zhaohui, W., Zhong, Z., Wei, W.: Realistic fire simulation: A survey. In: Proc. 12th Int. Conf. On Computer-Aided Design and Computer Graphics (CAD/Graphics 11). pp. 333–340. Jinan, China (September 2011)

[20] Zhou, J., Chang, Y., Wu, E.: Realistic, fast, and controllable simulation of solid combustion. Computer Animation and Virtual Worlds 22, 125–132 (2011)