Fire and Flame Simulation using Particle Systems and CUDA

T.S. Lyes and K.A. Hawick

Computer Science, Massey University, Albany, North Shore 102-904, Auckland, New Zealand

http://complexity.massey.ac.nz

Introduction

Fire and flame simulation is an interesting and important area of research in computer graphics [1]. As with most natural phenomena, it can be a challenging to simulate, particularly due to its complex, turbulent nature [2].To simulate such a complex system in real time is difficult even by today's computing power standards, especially if the system is particularly large, so parallel computing techniques can be used to good effect. 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 [3] [4].

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 [5] 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 [6] 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 [7] moving across a polygonal mesh. Other methods combine simple particle systems and advanced rendering techniques [4] to generate highly detailed fire simulations at relatively low computational cost. Combustion models [8] [9] and hydrodynamics [10] 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" [11] 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 [4]. 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, and 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)[12] can help with this limitation, and using GPU implementations allow simulations to be only limited by the particle data transfer between the processor and GPU [13]. It has been stated that using GPUs or multi-GPU set ups may enable system sizes of over one million particles to be simulated in real-time [14].

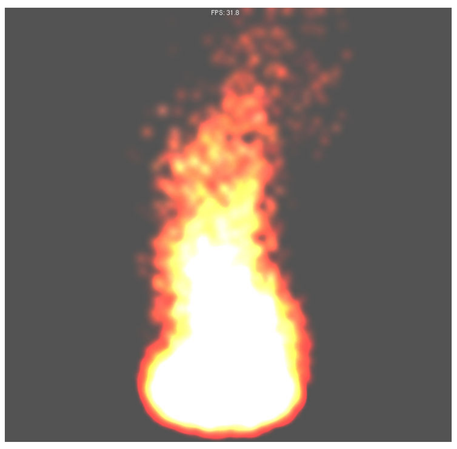

Figure 1: A particle system fire simulation

GPU's are designed to handle problems which can be expressed in parallel [15] such as particle systems [16]. 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) [17] 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 [18] or the purpose of rendering directly on the GPU, resulting in even better performance from the program.

When particle systems are used with GPUs, each particle can be created and updated using its own thread on the GPU. Because GPU memory is most efficient in powers of two, it is also recommended to make the particle system size a power of two. Most complex particle systems can have sizes of more than 10,000 particles, so a system size of 128 x 128 (16384 particles) would be an ideal size to work with on the GPU.

The chaotic nature of a flame or fire is one of the more difficult aspects to simulate. Enough randomization of the fire variables is enough to produce a decent looking simulation, however it will not be a highly accurate representation of a fire or flame system and will rely more heavily on rendering techniques to make the fire seem more realistic. Recently the CURAND random number generator library was created for use with CUDA kernels, as previously random numbers were difficult to generate on a device in parallel. Using CUDA, CURAND, OpenGL rendering and a simple particle system code, it is easy to simulate a reasonably large fire system which is not extremely computationally expensive while also maintaining a decent degree of realism.

Method

When using a particle system to simulate a fire, it can be set up in the following way; 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.

The general layout of a particle system might look something like this...

```
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
```

Unlike most particle systems, a fire system will not require the particles to implement collision with one-another. It is basically not needed. Simple rendering techniques such as blending can be used to make the system appear as a single moving entity regardless of the positions of the individual particles, so long as the particles themselves still behave the way they are supposed to. Rendering techniques are the key to making the fire appear more lifelike and realistic. Moreover, since there are so many particles in the system already, collisions would be detrimental to the performance of the program. Having to check the position of every particle relative to every other particle would slow the simulation down a lot, even when run in parallel, and is an issue especially when the simulation is to be run in real time.

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 four percentage floats to determine the red, green, blue and alpha components of the colour. The fire will always have a 100 percent red component, 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 OpenGL colour pointers and arrays to bind the colour VBO to OpenGL's colour array buffer.

CUDA has built in OpenGL interoperability functionality which can improve performance levels [17]. 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 before executing the kernel. Additionally, OpenGL will have 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.

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 must first run an initialization to create an RNG state for each particle in the system. 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. Initializing CURAND when setting up the kernel might look something like this...

```
//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: Code fragment for initializing CURAND when setting up the CUDA kernel

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 when randomizing a particle's starting colour and decay rate. CURAND can support many random number distributions, however for this simulation a uniform distribution was used to generate random floats between 0 and 1.

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

Visualization

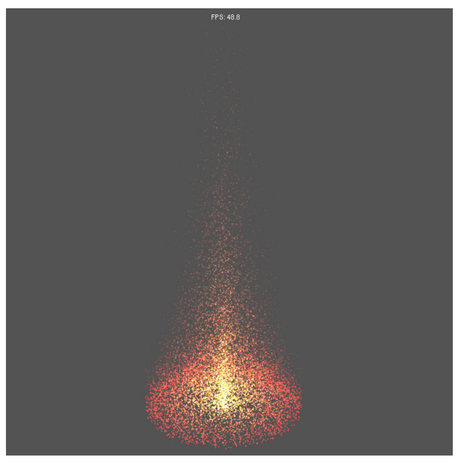

Figure 3: A fire rendered using OpenGL points

This is the initial visualization of the system with a size of 128x128. Smaller systems (64x64) did not produce visually strong simulations. This particular simulation used simple OpenGL points to represent each particle. While not the most visually interesting rendering method, OpenGL 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; the majority of the particles in the system are coloured far lighter, and when used with blending functions, these white and yellow colours come through far stronger then the outside red particles, even though the red particles should be more visible to the viewer.

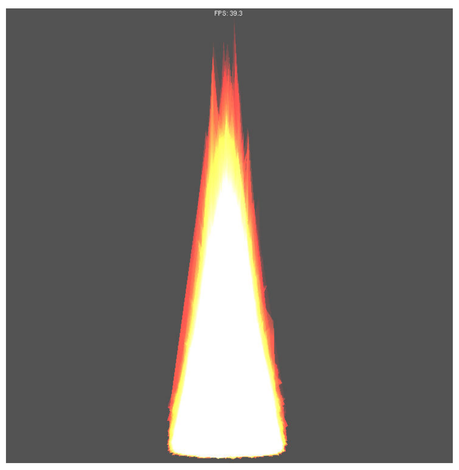

Figure 4: The same system rendered using triangular polygons

This image uses OpenGL triangles instead of lines or points. Three particles in the system make the vertices for each triangle, and multiple particles will be used as vertices for multiple triangles (there are as many triangles in the system as there are particles). While the colour progression looks much better than the OpenGL points, the shape of the flame is rather jagged. Because this simulation is rendered using polygons, OpenGL blending has been enabled, however this means that the red colour on the outer particles is almost lost. It is also worth noting that this simulation suffered a significant frame rate drop when compared to other rendering methods.

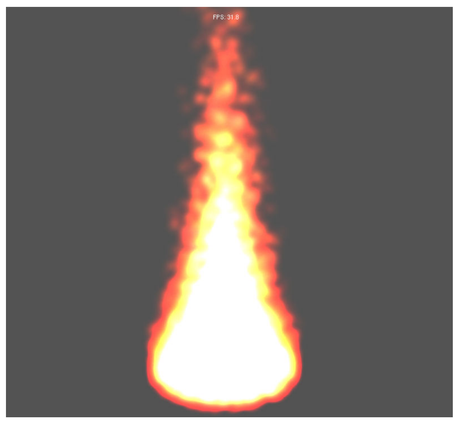

Figure 5: A fire rendered using texturing

Finally, this image shows a fire system using texturing to render the particles. This resulted in the most realistic looking fire, however this also suffered the highest framerate drop of all the methods used. Texturing is a very flexible rendering method. Each particle rendered using an OpenGL quad with an alpha texture to represent the alpha component of the colour. This particular simulation used a spherical sprite texture, but this texture could easily be changed to suit the type and behaviour of the desired fire.

Wind effects could also be used for added realism. This was fairly simple to implement; a wind element was generated using a simple sine wave function, and this was added to the x- and z-axis acceleration component of each particle in the system. Figure 1 shows the visual effect of wind on the fire simulation rendered using textures.

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.

It was found that the parallel versions of the update kernels provided substantial speed ups in performance; in all particle system sizes, speed-ups of more than 1000 times faster were observed. It is not known exactly how fast each parallel kernel was, as the execution time was faster than the timer's precision could detect. This would mainly be due to the simplicity of the particle code, which can easily be 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 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 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.

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

Figure 6: Comparison of frame rates in different system sizes when using the VBO rendering method

This table 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.

Frame rates were also monitored for each different rendering method using a 128x128 size particle system. Using OpenGL points gave the best frame rate, while using texturing resulted in the lowest frame rate. 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), which suggests that a 128x128 sized particle system seems to be the optimal size at this point. This is apparent when rendering larger system sizes, since when using simple OpenGL points the frame rate drops to at most 16 frames per second and using a more complicated render method would lower this even further.

See: `http://www.massey.ac.nz/~kahawick/cstn/168/cstn-168.html` for more information.

References

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

[2] R. Peyret and T. D. Taylor, *Computational Methods for Fluid Flow*. Springer Series in Computational Physics, Springer-Verlag, 1983.

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

[4] C. Horvath and W. Geiger, "Directable, high-resolution simulation of fire on the gpu," *ACM Trans. on Graphics*, vol. 28, pp. 41–1–8, August 2009.

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

[6] P. Beaudoin, S. Paquet, and P. Poulin, "Realistic and controllable fire simulation," in *Proc. Graphics Interface (GRIN'01)*, (Toronto, Ontario, Canada), 2001.

[7] H. Lee, L. Kim, M. Meyer, and M. Desbrun, "Meshes on fire," in *EG Workshop on Computer Animation and Simulation*, pp. 75–84, 2001.

[8] H. Xue, J. C. Ho, and Y. M. Cheng, "Comparison of different combustion models in enclosure fire simulation," *Fire Safety Journal*, vol. 36, pp. 37–54, 2001.

[9] J. Zhou, Y. Chang, and E. Wu, "Realistic, fast, and controllable simulation of solid combustion," *Computer Animation and Virtual Worlds*, vol. 22, pp. 125–132, 2011.

[10] F. Zhang, L. Hu, J. Wu, and X. Shen, "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.

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

[12] A. Leist, D. Playne, and K. Hawick, "Exploiting Graphical Processing Units for Data-Parallel Scientific Applications," *Concurrency and Computation: Practice and Experience*, vol. 21, pp. 2400–2437, December 2009. CSTN-065.

[13] A. Kolb, L. Latta, and C. Rezk-Salama, "Hardware-based simulation and collision detection for large particle systems," in *Proc. Graphics Hardware*, 2004.

[14] L. Latta, "Building a million particle system," in *Game Developers Conference*, 2007.

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

[16] K. Hawick, D. Playne, and M. Johnson, "Numerical precision and benchmarking very-high-order integration of particle dynamics on gpu accelerators," in *Proc. International Conference on Computer Design (CDES'11)*, no. CDE4469, (Las Vegas, USA), July 2011.

[17] NVIDIA® Corporation, *CUDA™ 3.1 Programming Guide*, 2010. Last accessed September 2010.

[18] D. Hearn and M. P. Baker, *Computer Graphics with OpenGL*. No. ISBN 0-13-015390-7, Pearson Prentice Hall, third edition ed., 2004.