A GPU Accelerated Volumetric Ray Tracer for Incandescent Gas

Andrew Zonenberg, Sylvia Forrest
Rensselaer Polytechnic Institute
110 8th Street
Troy, New York U.S.A. 12180
zonena@cs.rpi.edu, forres3@rpi.edu

May 5, 2011

1 Introduction

The initial goal of this project was to create a physically accurate GPU-accelerated simulation of fire. Due to limited time available in the semester (combined with the inherent difficult of debugging CUDA code) we ended up reducing the scope somewhat and focusing on a realistic GPU-accelerated technique for rendering incandescent gas, as in flames, without doing a full fluid simulation.

The application was written in C (GPU code) and C++ (CPU code) and tested on a laptop running 64-bit Linux with a GTX 460m card.

2 Related work

[1] lays out a basic algorithm for volumetric ray tracing: it renders a given volume based on local data specified at each voxel within the volume without the use of geometric primitives. In his method, voxels are treated as samples on a continuous function for color/opacity, and the specific values at a point are derived with local operators. Front-to-back eye rays are modeled as vectors of color/opacity values sampled from a given number of evenly spaced points along the ray. These sample values, in turn, are created from a trilinear interpolation of the locally derived values from the 8 voxels surrounding the one which the ray is passing through. The final returned from a given eye ray is the composite of all the values in the vector, combined with a completely opaque background. Optimization is achieved by stopping eye rays once a high enough opacity value is encountered, and by hierarchical spatial enumeration.

Another early paper on the matter, [2] presents an alternative method which combines volumetric rendering with standard ray tracing in which geometric primitives are used. Local volumetric information is not specified everywhere. Ray-object intersections are handled in a variety of ways, depending on user specifications for the scene and the actual data/objects involved in the intersection. The intersections are either point or segment intersections, and optimization is achieved by sorting calculations by closeness of intersection points to the camera (in rays where there is point intersection, only the closest one to the camera and all segments in front of it need be evaluated).

3 Raytracing algorithm

While [2]'s model is more comprehensive, we were not planning to use any geometric primitives or multiple complex effects, so we used a method more like [1]. We used a slightly different method for antialiasing which did not require us to sample around each point the way that [1] does - taking a large number of samples at sub-voxel spacing (our current implementation typically uses 0.25 or 0.5 voxels) along the ray. This method appeared to provide better cache locality, which is critical given the massive memory bandwidth used by such an algorithm.

4 Blackbody rendering

[3] describes a method for accurately simulating the color and flow of fire, and its interaction with different kinds of fuels, backgrounds, and other objects. The fire is modeled with an implicit surface representing the blue core of the flame, blackbody radiation, and soot/smoke, using Navier-Stokes incompressible flow equations to mimic the movement of the flame. We were primarily concerned with the blackbody radiation aspect of this, which creates accurately colored fire for a full spectrum. Very simply, Planck's formula is used to produce a spectrum in XYZ colorspace, which is then converted to RGB. A Von Kries transformation can be applied to adjust the colors such that the highest temperature becomes the white point for the output render.

Our blackbody rendering algorithm is largely based on [4], with additional information and conversion values taken from [5]. For a given voxel, we compute a spectrum of blackbody emission values based on the voxel's temperature (Kelvin) and wavelengths between 380 nm and 650 nm, at 15 nm intervals, using Planck's formula:

$$L_e(T,\lambda) = \frac{c_1 \lambda^{-5}}{\left(e^{\frac{c_2}{\lambda T}} - 1\right)}$$

where $L_e(T,\lambda)$ is light emitted at wavelength λ by a blackbody at T degrees Kelvin, c1 = 3.74183E-16, and c1 = 1.4388E-2.

The emitted light is then multiplied by a set of 3 CIE conversion factors from a lookup table in order to create an x, y, and z value for that wavelength. Optionally, we account for sodium in the fuel (which would create the familiar oranges and yellows of fire) by adding an extra call to the spectrum evaluator at 590 nm (the closest value in our lookup table to sodium's D-lines, which are at 588.6 nm and 589 nm), and scaling that by a sodium percentage value. We then obtain a single (x,y,z) chromaticity value for the voxel like so:
$x = X/(X+Y+Z)$
$y = Y/(X+Y+Z)$
$z = Z/(X+Y+Z)$
where X, Y, and Z are the separate summations of x, y, and z values returned from each wavelength in the spectrum.

Given more time the same method could be used for visually accurate rendering of other heated gases, by inputting the spectrum of each gas into a table and having voxel properties contain the fraction of each gas.

The conversion from perceptual (x,y,z) colorspace to computer (r,g,b) colorspace is achieved by multiplication by a set of constant values which mimic a Von Kries transformation calibrated for the sRGB white point. Adjustments are then made to make sure the values are non-negative and the magnitude is normalized, at which point they are scaled by the blackbody intensity. (Actual blackbody radiation intensity is proportional to the fourth power of the absolute temperature, however rendering this proved impossible due to the limited dynamic range of computer monitors. After some experimenting we found that an exponent of 1.5 produced believable values with a more manageable dynamic range.) During the volume rendering, these values are summed along the ray (as floating-point numbers) and capped at 255.

With more time, we would have implemented the pressure-based heat flow as in [3]; however as that was not our priority and we were short on time, we used several (much simpler) time-varying models for our example renders.

5 GPU implementation

CUDA presents a SIMT (single instruction multiple thread) parallel programming model in which multiple threads execute the same kernel function. The threads are grouped into blocks of user-defined size (up to hardware limits such as number of registers); a grid of one or more blocks is then launched on the card. Thread and block IDs are 3-vectors (often, but not always, used to define points in some sort of 2- or 3-dimensional space).

Each thread block is mapped to one fairly wide (16- or 32-way) SIMD processor on the GPU; if more blocks than processors are requested then not all blocks will execute simultaneously. (The ordering of blocks in this situation is undefined; the programmer is expected to design code with no inter-block sequencing dependencies.)

Each thread in the block is mapped to one SIMD unit of the processor; as with blocks if more threads than SIMD units are requested they are executed in an undefined order. All threads are given the opportunity to execute one instruction before the next instruction is run; this permits local barrier synchronizations to be performed between all threads in the block.

Each SIMD processor in current CUDA GPUs is equipped with 16KB of shared memory to be used by the threads for communication and storage of frequently used data, 16KB of dedicated L1 cache, and 32KB of fast SRAM which may be configured as either L1 cache or shared memory (but not both simultaneously). Cards before the GTX 4xx / Tesla C2xxx series lack the L1 cache and extra memory, featuring only the 16KB of shared memory.

Our raytracer uses one screen pixel per thread, and currently runs with 256 threads per block and one scanline per grid. (This does not fully load GPus with more than 8 SIMD processors; for debugging the improved response time and ability to interrupt a render after a single scanline was considered more valuable than performance. Our design could easily be optimized to reduce kernel-call overhead by rendering ten or fifteen scanlines per grid.)

The kernel then computes a ray from the eye to the desired pixel position and traces it through the volume until the far clip plane is reached. Sub-voxel sampling (spacings of 0.25 and 0.5 were used for most of the example renders) is used to produce smoother results.

6 Results and conclusion

Progress on the project was slow as only one of us (Andrew) had any prior experience with CUDA development, or even a CUDA-capable machine to test on. This was made worse by the lack of memory protection or the other niceties of protected-mode operation on a GPU. On several occasions an infinite loop was created in the raytracer, causing the affected machine to become completely unresponsive and require a hard reset. (This is not too unusual for low-level GPU work; the price paid for improved runtime performance is massively increased development time!)

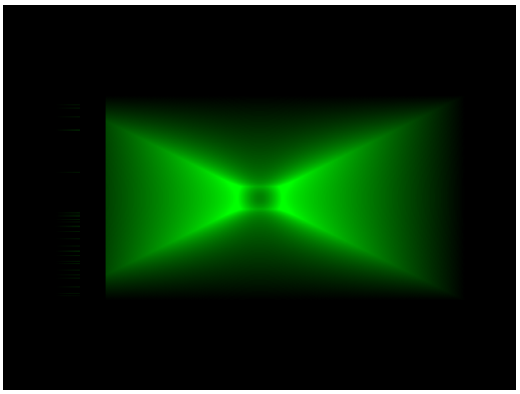

Figure 1: First test render (before blackbody implementation)

Our early test renders (see Fig. 1) consisted of a cubic volume with a hard-coded color as we did not have the blackbody coloring working yet. This image used an extremely wide field of view, hence the fish-eye effect on the corners. The cause of the artifacts at left is currently unknown.

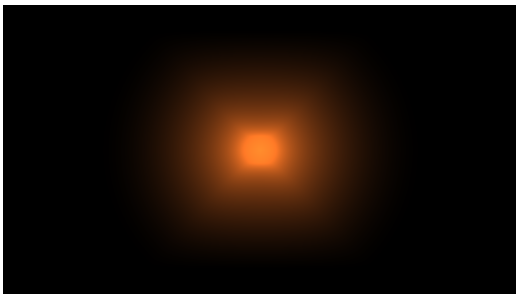

Figure 2: First blackbody test render

The next sample image (see Fig. 2) was a roughly spherical volume of a transparent blackbody material (such as soot particles in air) heated to around 3000K. Note saturation of intensity at center of image. (We had

considered HDR rendering techniques but did not have adequate time to implement them.)

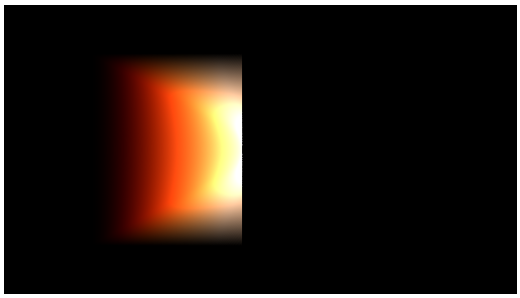

Figure 3: Frame 104 of video sequence

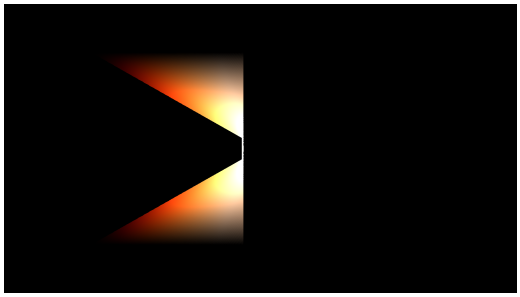

Figure 4: Frame 105 of video sequence

The next test was a 250-frame video sequence depicting an extremely hot planar object (around 10,000 K) moving at high speed through an air-filled volume. Gas touching the object becomes heated, and gradually cools off as the heat is removed. (The simulation here was not physically accurate as the focus was on rendering volumes of heated gas; the simulation was simply a means of producing data to render.)

This last render (completed the day before the paper was due) uncovered a bug in the raytracer whose cause remains unknown. Once the leftmost column of voxels has cooled to below some threshold a strange triangular artifact appears (starting at frame 105). The same issue can be seen on the right side of the volume.

References

[1] Mark Levoy, "Efficient Ray Tracing of Volume Data", 1995

[2] Lisa M. Sobierajski, Arie E. Kaufman, "Volumetric Ray Tracing", 1995

[3] Duc Quang Nguyen et al, "Physically Based Modeling and Animation of Fire" , 2002

[4] John Walker, "Colour Rendering of Spectra", 1996

[5] Bruce Lindbloom, "Useful Color Equations", 2009. Available HTTP: http://www.brucelindbloom.com/