Interactive Volume Rendering Aurora on the GPU

Orion Sky Lawlor* Jon Genetti†
Department of Computer Science, University of Alaska Fairbanks

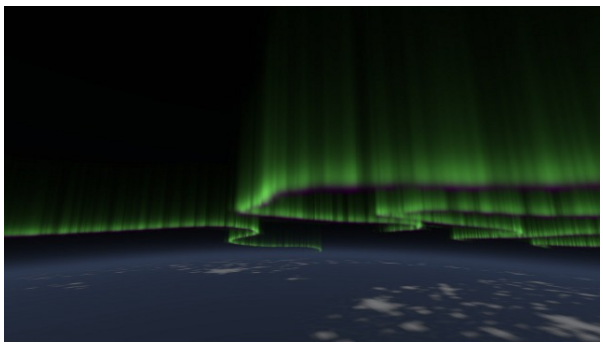

Figure 1: Our rendered aurora, 60km above Finland.

ABSTRACT

We present a combination of techniques to render the aurora borealis in real time on a modern graphics processing unit (GPU). Unlike the general 3D volume rendering problem, an auroral display is emissive and can be factored into a height-dependent energy deposition function, and a 2D electron flux map. We also present a GPU-friendly atmosphere model, which includes an integrable analytic approximation of the atmosphere's density along a ray. Together, these techniques enable a modern consumer graphics card to realistically render the aurora at 20–80fps, from any point of view either inside or outside the atmosphere.

Keywords: Volume rendering, aurora borealis, atmospheric scattering.

1 THE AURORA

The aurora borealis and aurora australis are beautiful phenomena that have fascinated viewers in Earth's polar regions for centuries. Aurora are generated when charged particles trapped by a planet's magnetic field collide with and excite gas in the upper atmosphere.

*e-mail:lawlor@alaska.edu
†e-mail:jdgenetti@alaska.edu

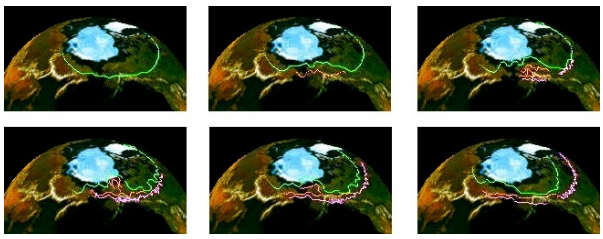

Figure 2: Global progress of a typical auroral substorm.

On Earth, these charged particles rarely penetrate below 50 kilometers altitude, and the aurora become difficult to discern above 500 kilometers due to the thin atmosphere.

The charged particle fluxes visible as auroral displays are driven by magnetohydrodynamics that are complex and the details are poorly understood, but the effects can be qualitatively described. As is typical in magnetohydrodynamics, magnetic effects expel currents from the body of a conductive plasma, compressing the charged particle currents flowing through the magnetosphere into thin sheets around one kilometer thick. As these current sheets are bent along magnetic field lines and intersect the atmosphere, they become visible as auroral "curtains," long linear stripe-like features. Depending on the activity level of the aurora, curtains can be nearly featureless greenish blur, or an extremely complex and jagged path.

A typical "auroral substorm" [Aka64] begins with simple, smooth curtains. These then grow and begin to fold over during substorm onset, resulting in many overlapping and interacting curtains, which become more and more complex and fragmentary as the substorm breaks up, and finally substorm recovery gives dim pulsating aurora. Recent work by Nishimura et al. [Nis10] has linked ground observations of pulsating aurora to space-based observations of electromagnetic waves deep in Earth's magnetotail, using the THEMIS satellites.

Because the detailed interactions of the charged particles and magnetic fields that drive auroral substorms are poorly understood, for rendering purposes we approximate their effect. We represent an auroral curtain's path using a time-dependant 2D spline curve "footprint," which are animated by hand to match the broad global outlines of an auroral substorm as it moves over the surface of the planet as shown in Figure 2.

1.1 Algorithm Overview

In this paper, we present a combination of techniques to interactively render the aurora on modern graphics hardware. To summarize our interactive GPU rendering algorithm:

1. We begin with aurora curtain footprints, described in Section 2, stored as 2D splines curving along the planet's surface.
2. We add 2D complexity to those curtain footprints by wrapping a long thin fluid dynamics simulation along them as described in Section 2.
3. We preprocess the curtain footprints into a 2D distance field described in Section 3.2, and stored in another GPU 2D texture and used to accelerate rendering.
4. We stretch the curtains into 3D using an atmospheric electron deposition function, as described in Section 2.1. The deposition function is expensive and constant, so it is stored as a GPU texture lookup table.
5. For each frame, we shoot rays from the camera through each pixel onscreen. Any camera model may be used.
6. For each ray, we determine the portion of the ray that intersects the aurora layer and atmosphere, and determine the layer compositing order as described in Section 3.1.
7. To intersect a ray with an aurora layer, we step along the ray at conservative distances read from the distance field, as described in Section 3.2. At each 3D sample point, we sum up the auroral emission as the product of the 2D curtain footprint and the vertical deposition function.
8. To intersect a ray with the lower atmosphere, we evaluate a closed-form airmass approximation as described in Appendix A.
9. Final displayed pixels are produced by compositing together the resulting aurora, atmosphere, and planet colors followed by an sRGB gamma correction, as described in Section 3.3.

Section 4 describes the performance of our algorithm on various graphics hardware.

2 MODELING THE AURORA IN 3D

Because curtains become fragmented and complex during the highly excited periods of an auroral substorm, splines alone do not convey the complexity of real curtains, as illustrated in Figures 3 and 4. Several approaches have been used to simulate this complexity,

Figure 3: Photograph of auroral curtains during a moderate substorm. The shutter was open for four seconds.

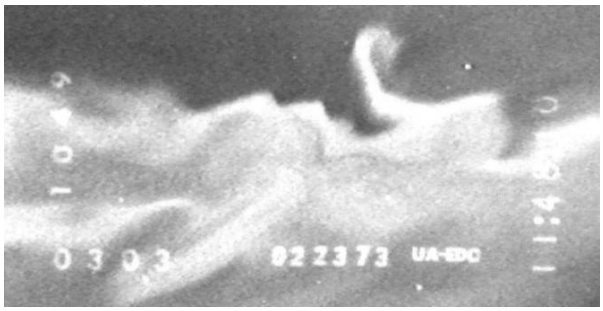

Figure 4: High-speed video of a portion of a very active curtain. Field of view is 4km wide.

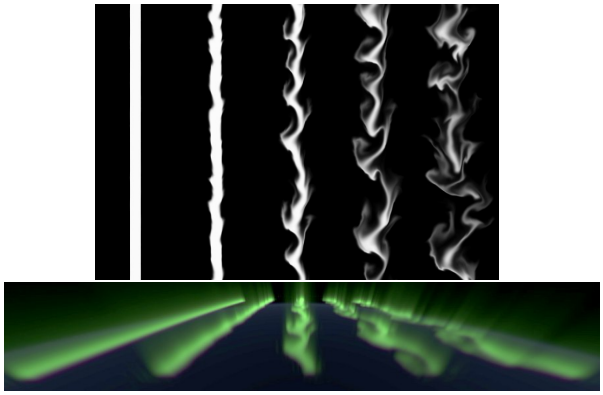

Figure 5: Portions of the 2D fluid dynamics simulations we use to model small-scale curtain complexity, and the resulting 3D auroral curtains.

such as raycasting caustics, but we find the phenomena are better matched by a fluid dynamics simulation.

To simulate aurora curtain footprint complexity, we use an simple 2D Stam-type [Sta99] fluid advection simulator. We use a multigrid divergence correction approach for the Poisson step, which is both asymptotically faster than an FFT or conjugate gradient approach, and makes the simulator amenable to a graphics hardware implementation. The simulator is solving a Kelvin-Helmholtz instability problem, with the fluid shear zone lying along the flux center of the auroral curtain, as illustrated in Figure 5. We perform the simula-

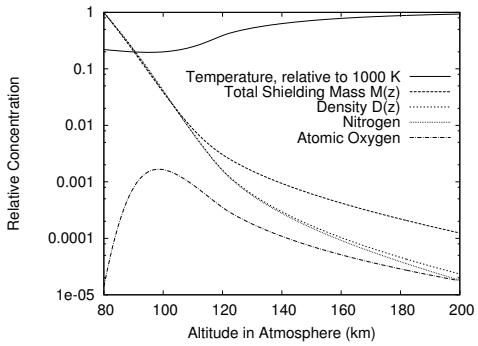

Figure 6: MSIS atmosphere as a function of altitude.

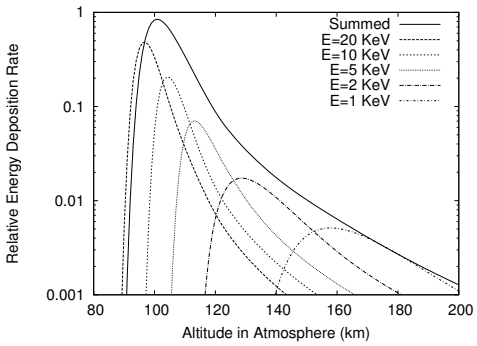

Figure 7: Auroral energy as a function of altitude.

tion in a long vertical domain with periodic boundary conditions, so we could replicate the same simulation along an arbitrarily long spline. The resulting simulated auroral curtain is stretched along the spline that defines the center line of the curtain. For quiet early periods during the substorm, we use the initial steps of the simulation, before substantial turbulence has distorted the smooth initial conditions; later more chaotic curtains are represented using later steps in the simulation, when the simulation's fluid turbulence results in a very complex electron flux pattern. The magnetosphere's actual plasma dynamics are of course very different from simple Navier-Stokes fluids, but this simulation seems to approximate the final turbulent appearance of the aurora reasonably well.

2.1 Aurora Vertical Deposition

We use splines to impose the global location of the auroral curtains, and fluid dynamics to approximate the small-scale variations in brightness, but both of these give only an electron flux footprint on the surface of the planet, in 2D. To create a full 3D volume model of the aurora, we must specify how the electrons are deposited through the atmosphere, via an electron deposition function.

The depth that charged particles penetrate the atmosphere depends on both the velocity of the charged particles and the atmosphere's state. However, the state of the upper atmosphere is not constant, due to variable energy input from solar radiation, ground-based upward travelling radiation, and even variable auroral energy deposition itself. Since the auroral energy deposition profile depends on a variety of factors, including feedback due to auroral heating, an exact deposition model would require us to simulate the spatial and temporal variations in the upper atmosphere's density, temperature, and chemistry. Software exists to do this, such as NCAR's thermospheric general circulation model, but it is not amenable to either the GPU or to realtime interactive simulation. Instead, we begin with the standard MSIS-E-90 atmosphere [Hed91], as shown in Figure 6.

We then apply the Lazarev charged particle energy deposition model [Lum92], which is still the definitive model for low-energy auroral electrons [Fan08]. The inputs to the Lazarev model are the particle energy E and the atmosphere's mass M_z and density D_z at the desired height z, and the output is the auroral energy deposition rate A_z:

E Initial energy of incoming particles, in thousands of electron-volts [keV]. For aurora, this is 1–30keV [Fan08]. These equations work well below 32keV.

z Altitude above surface to evaluate deposition.

D_z Atmosphere's density at altitude z [g/cm^3]. This is listed directly in the MSIS data.

$M_z = \int_z^\infty D_{z'} dz'$ Atmosphere's total shielding mass above altitude z [g/cm^2].

$M_E = 4.6 \times 10^{-6}\, E^{1.65}$ Characteristic shielding mass for particles of energy E [g/cm^2]

$r = M_z / M_E$ Relative penetration depth [unitless]

$L = 4.2\, r\, e^{-r^2 - r} + 0.48\, e^{-17.4\, r^{1.37}}$ Lazarev's unscaled interaction rate [unitless]

$A_z = L\, E\, (D_z / M_E)$ Aurora energy deposition rate at altitude z.

The result of the Lazarev deposition model is shown in Figure 7 for several discrete input energies, as well as a sum over energies from 1keV to 20keV. Various parameterizations of this deposition function exist, such as the popular Thermospheric General Circulation Model [Rob87], which assumes a Maxwellian distribution of electron energies. TGCM is actually simple enough to evaluate per pixel at runtime, as we explore in Section 4. However, both the Lazarev or TGCM models need an atmosphere model as input, and the thermosphere's density profile D_z is complex, as shown in Figure 6. Since we will need a lookup table to store the atmosphere's shielding mass and density, we simply pre-evaluate the deposition function for various energies and store the result in a table.

2.2 Prior Work in Aurora Modeling

We extend the excellent and rigorous aurora rendering work of Baranoski et al. [Bar00] in several ways. First, this prior work forward maps aurora curtain points onscreen followed by a guassian blur, while our renderer walks backward along camera rays accumulating visible energy. Our raytracing approach allows us to render to arbitrary resolutions and produce sharp rendered images. Second, we provide an interactive GPU implementation which includes the effect of the lower atmosphere on the aurora and allows us to render the aurora from any point inside or outside the atmosphere. In the prior work, electron-atmosphere impacts are simulated explicitly, while we simply look up their well known altitude dependent statistical energy deposition function. Finally, the prior work's curtains are constructed from a combination of sine wave with phase shift oscillations and a caustic-type electron beam deflection model; while our curtains begin as splines, with smaller turbulent deflections applied via a fluid dynamics simulation.

The later work of Baranoski et al. [Bar05] presents a detailed physically plausible model of the magnetohydrodynamics of a charge sheet's path through the magnetosphere prior to becoming visible as an auroral curtain. There appears to be an almost exact analogy between this work and our fluid dynamics simulation of curtain dynamics: electric charges with inertia interact via an electrostatic field, while fluid parcels with inertia interact via a pressure field. Both electrodynamic and fluid dynamic simulations use a multigrid Poisson solver to control field divergence, and the results appear roughly similar as well. One difference is we have not yet attempted to specialize our initial conditions to generate the spiral structures visible as auroral surges.

3 GPU RAYTRACING THE AURORA

Raytracing is a rendering technique that finds a scene's color along a ray by intersecting the ray with the scene geometry. Raytracing is computationally demanding, and the first interactive raytracers used a combination of carefully constructed scenes (such as a set of spheres) and massive parallel computing horsepower. University of Utah researchers [Par98] used a large shared-memory machine for this, while John Stone's Tachyon [Sto98] used a network of distributed-memory workstations. GPU raytracing is such a natural fit that initial work in this area [Pur02] actually preceded fully programmable GPU hardware, and an abundance of modern work exists. Similarly, volume rendering via raytracing is a venerable and well known technique [Kaj84].

3.1 Aurora rendering geometry

The aurora are almost perfectly emissive phenomena, since the degree of absorption and scattering by the atmosphere is vanishingly small around 100km altitude. Even at sea level air's optical properties are reasonably close to that of vacuum, and at 100km altitude the air's density is a millionfold smaller. The isotropic emissions, and lack of absorption and scattering, simplifies Kajiya's rendering equation [Kaj84] for the aurora layer into a single integral along the path of the ray.

Since aurora only appear in the upper layers of the atmosphere, we can treat them as a separate purely emissive "aurora layer." Below 80km is the bulk of the lower atmosphere, which both absorbs and scatters light as discussed in Appendix A. Underneath all of this is the planet's surface. Because the lower atmosphere includes scattering, implemented using alpha blending, we must composite the layers in the correct order.

A general-purpose raytracer typically uses recursion to resolve the depth order of multiple layers of translucent geometry that intersect a ray, but this general solution is not appropriate in our case. First, GPU hardware that directly supports recursion was only introduced in 2010 with the NVIDIA Fermi line, and most current cards do not directly support recursion. Second, even where it is supported this recursive search for geometry is expensive, typically requiring $O(n^2)$ intersection tests to determine the depth order of n translucent layers, and we find the many branches required can become a limiting factor in a high performance GPU raytracer.

Thus instead of a recursive search, for each ray we programmatically determine the correct compositing order of the intersected geometry, as summarized in Figure 8. The easy case is 8(a), where the ray misses all geometry and heads out into deep space. Case 8(b) is a ray that enters the aurora layer, accumulates some emitted energy, and exits. The most complex case is 8(c), where the aurora layer is entered twice: once before the atmosphere, then some aurora light is scattered out by the atmosphere, and finally a disjoint stretch of aurora layer emits more light into the ray. Finally, case 8(d) begins on the planet's surface, whose light is attenuated by the atmosphere, and then some aurora light is picked up before reaching the viewer. The same cases apply for a viewer inside the aurora layer. For a viewer inside the lower atmosphere, the only two compositing possibilities are atmosphere then planet, or atmosphere then aurora.

One limitation of our explicit ray compositing order is we do not support atmospheric refraction. However, Earth's atmosphere only very gently refracts rays, resulting in a maximum curvature near the horizon which is less than 1/6 of the planet's curvature, so we feel it is acceptable to ignore atmospheric refraction.

Given a portion of a ray that intersects the aurora layer, in principle we step through the layer accumulating aurora energy, at each step sampling the aurora curtain footprint in 2D and multiplying it by the height-dependent energy deposition function. The step size is

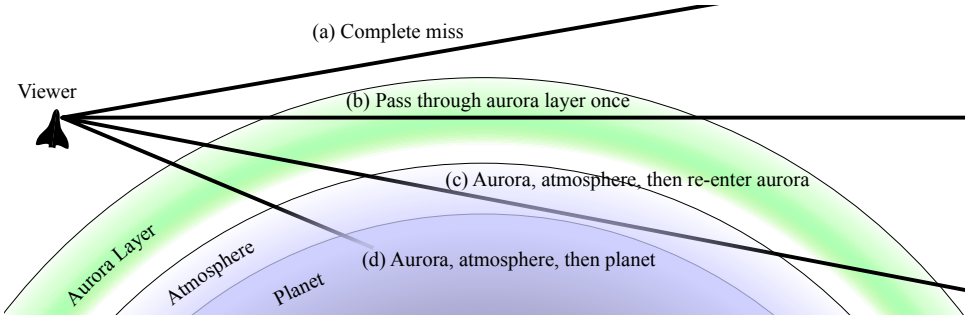

Figure 8: Possible ray/geometry intersection paths for camera rays originating outside the atmosphere.

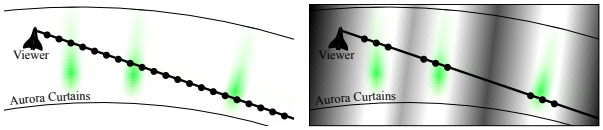

Figure 9: Naive ray stepping, left, is inefficient when curtains are sparse. Using a distance field, as shown on the right, allows the raytracer to take much larger steps in the empty spaces between curtains.

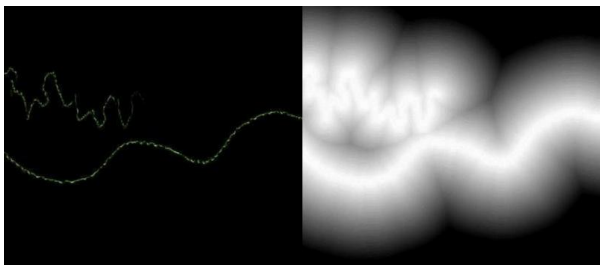

Figure 10: On the left, aurora curtain footprints. On the right, the distance field to accelerate raytracing those curtains.

a tunable parameter, with finer steps giving more aurora detail but as we show in Section 4, taking more time to compute. The step size is limited by the resolution of the aurora footprint texture: an 8192x8192 aurora footprint stretched across a 12742km diameter planet gives pixels that are 1.55km along the coordinate axes, and 2.2km diagonally. We find a 2km step size gives a reasonable quality image, but with naive sampling is quite slow. In the next section, we show how to accelerate the aurora sampling process.

3.2 Acceleration via a Distance Field

The auroral layer is hundreds of kilometers high, and wraps around a planet thousands of kilometers in diameter. Yet auroral curtains are only a few kilometers thick, so as we step along a ray we must sample the aurora layer at least every few kilometers to avoid missing curtains. Even modern GPU hardware cannot support thousands of such 3D samples per pixel in real time, since there are millions of onscreen pixels.

However, most of the auroral layer does not actually contain curtains, so if we could skip over the empty space between curtains, we could dramatically improve our overall performance. Figure 9 illustrates the problem, and the solution we use: a distance field [Coh94]. This field stores the distance to the nearest geometry, which allows the raytracer to take much larger steps through empty space.

The distance field is stored as a 2D texture, with a slightly lower resolution than the aurora curtain image. As we step along a ray, we read the step size from the distance field, so we step at a fine 2km/step rate while inside curtains; yet can take much longer steps far from curtains, up to 1000km/step, without ever skipping over a curtain. In pseudocode, our sampling loop through the aurora is as follows.

```
float t = ray.start;
while (t < ray.end) {
    vec3 P = ray.origin + ray.dir*t;
    t += distance_field(P);
    aurora += sample_aurora(P);
}
```

One surprising aspect of the GPU branch hardware is that it is actually a performance loss to skip the aurora sampling when distant from a curtain. We found it to be at least 18% slower to do the following "optimized" sampling; our other attempts at similar optimizations have been up to sevenfold slower!

```
float t = ray.start;
while (t < ray.end) {
    vec3 P = ray.origin + ray.dir*t;
    float d = distance_field(P);
    t += d;
    if (d<ε) /* inside curtain */
        aurora += sample_aurora(P);
}
```

The performance problem in this sort of loop is branch divergence, when some GPU threads take the distance-dependent branch and sample the curtain while others do not. The large GPU branch divergence penalty exceeds the savings from avoided samples, which makes

it faster to simply sample everywhere than to carefully decide whether to sample or not.

We generate the distance field from the curtain image on the GPU, but as a preprocess before rendering. We use a clever constant-time algorithm known as "jump flooding" [Ron06], which takes distance propagation steps at power of two distances to fill the distance field across the 2D image.

3.3 Coloring the Aurora

On short timescales, the upper layers of the aurora are green, while the lower layers have a purple tinge. We use the Baranoski et al. [Bar00] approach to convert the auroral emissions' isolated spectral color peaks to CIE XYZ and then a linear sRGB colorspace.

More difficult are numerical problems encountered while summing thousands of dim samples. In Baranoski et al., aurora samples are forward mapped and summed in a framebuffer, while we step along camera rays in a loop on the GPU. Because the GPU registers are floating-point, and floating-point framebuffers are expensive, a raytracer can more efficiently sum aurora samples in a high precision and high dynamic range linear colorspace. We then convert to the standard sRGB gamma of 2.2 using the following function, which outputs a color with vector magnitude equal to the old magnitude raised to the 1/2.2 power.

```
float brightness=length(color);
return color*pow(brightness,1/2.2-1);
```

4 PERFORMANCE ANALYSIS

We use the standard OpenGL Shading Language, GLSL, to implement our GPU aurora raytracer. Unlike the general-purpose GPU languages CUDA and OpenCL, the older GLSL is specialized for rendering tasks, so it directly supports graphics hardware features such as anisotropic mipmapping. Recent work on VOREEN [Men10] showed CUDA only improves performance when volume samples overlap, such as in gradient calculations. Table 1 compares the performance of our GPU aurora rendering algorithm across various GPU families, and a C++ OpenMP multicore CPU version of the algorithm. Even using four cores and nearest-neighbor texture sampling, the CPU runs about a hundred times slower than the GPU versions.

Table 2 lists the performance impact of various algorithm and parameter modifications. This is a list of alternatives not chosen for the current implementation, although many of these could still be useful.

Our raytracer acceleration distance field results in rather dramatic per-pixel performance variations, as shown in Figure 11. The corresponding frame is shown in Figure 1. Where multiple curtains cross camera rays the rendering cost can be hundreds of nanoseconds per pixel, while empty regions of space require less than

GPU	FPS
NVIDIA GeForce GTX 280	60fps
NVIDIA GeForce 8800M GTS	38fps
ATI Radeon HD 4830	23fps
Intel Q6600 2.4GHz Quad-Core CPU	0.4fps

Table 1: Comparing renderer performance across hardware. Resolution is 720p: 1280x720.

Modified Rendering Method	Cost
No distance field, use naive stepping	+350%
Make aurora layer 100km thicker	+32%
Take 1km steps through aurora, not 2km	+60%
Take 4km steps through aurora, not 2km	-33%
No table, use TGCM deposition function	+55%
No decibel map, linear deposition table	-10%
No deposition function, constant value	-14%
No curtain footprint image lookup	-14%
No exponential atmosphere	-15%
No planet texture	-0.6%
No sRGB gamma correction	-0.5%

Table 2: Performance impact of various alternatives. Positive time cost lowers framerate.

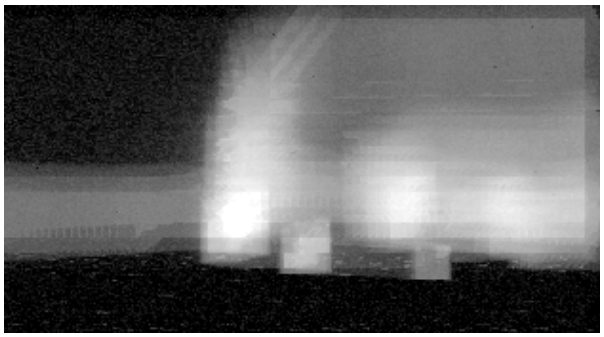

Figure 11: Measured rendering time per pixel: black represents 10ns/pixel, white represents 200ns/pixel.

ten nanoseconds per pixel. This experiment was run on the NVIDIA GeForce 8800M GTS; timings on different cards vary, but the ratios are similar. This figure is somewhat blurred due to the nature of GPU performance analysis: GPU hardware provides no means to time individual pixels, and in fact extensive GPU pipelining makes per-pixel timing difficult to even define, so instead we time overlapping blocks of 64x64 pixels. After several repetitions, the median per-block times are converted to per-pixel times by subtracting off the per-block overhead and dividing by the number of pixels. The remaining sampling jitter due to OS and driver overhead is approximately $\sigma = 2$ns/pixel.

4.1 GPU Aurora on a Powerwall

We used the parallelizing library MPIglut [Law08] to port our sequential OpenGL/GLUT aurora rendering application to a twenty-screen powerwall, as shown in

Figure 12: Interactive aurora rendering on a powerwall cluster with ten GPUs and twenty screens at 29fps.

# GPUs	Resolution	FPS	Speedup
1	1680x2100	35	1
2	3360x2100	30	1.6
4	6720x2100	27	3.0
8	6720x4200	29	6.5
10	8400x4200	29	8.2

Table 3: Parallel aurora rendering via MPIglut.

Figure 12. This was a surprisingly straightforward process, involving recompiling the rendering application with MPIglut instead of glut, and running the resulting binary. Scalability as shown in Table 3 is reasonably good, although view-dependent load imbalance becomes large when some screens must draw complex curtains and other screens only empty space; for the benchmark this impacts the two and four GPU values somewhat. The aggregate rendering rate on ten NVIDIA GeForce GTX 280 cards is a little over 29 frames per second at 8400x4200 resolution, or just over a billion finished pixels per second.

5 CONCLUSIONS

With only moderate programming effort, modern graphics hardware is capable of truly incredible amounts of computation. We have harnessed that power to render the aurora at interactive rates, but much work remains.

At the moment, our raytracer implementation stands alone, and includes no polygonal geometry. It would be relatively straightforward to extend this to a hybrid raytracer, where ordinary polygon-based geometry is first rasterized to a typical depth buffer, and these depth values are then used to limit the extent of each ray [Sch05]. This extension would allow the techniques described in this paper to add atmospheric and aurora effects to a scene that includes terrain, vegetation, spacecraft, or other geometry.

We currently render a single instantaneous snapshot of the aurora; the viewer is free to move, but the curtains are stationary. It should be straightforward to extend this to animating curtains, and we have done so offline, but image I/O and texture upload rate becomes an issue when rendering in realtime. Similarly, we currently do not integrate the curtains across the minutes-long timescale that gives high red aurora. This should be a simple change to our input curtain footprint images. Both changes should allow a detailed comparison with the widespread seconds-long-exposure photographic images of the aurora.

Since aurora are purely emissive phenomena, our atmospheric airmass model currently ignores clouds and the interesting multiple scattering effects of sunlight on the air. Incorporating these effects would allow us to simulate aurora at sunrise, or aurora rising over a thunderhead. More ambitiously, implementing a global illumination algorithm such as photon mapping or path tracing could allow aurora to cast light onto complex geometry, such as a mountainside or spacecraft.

Aurora are visible on many planets, and often display curtains and dynamics similar to those on Earth. However, the dynamics of aurora on planets without a single dominant magnetic field, such as Venus or Mars, can be quite different, and simulations would be beneficial for studying these fascinating phenomena.

ACKNOWLEDGEMENTS

The authors sincerely thank Dr. Syun-Ichi Akasofu for providing the schematics of a typical auroral substorm and the video capture in Figure 4, as well as Dr. Bill Brody for digitizing and animating that substorm as a series of continuous splines as shown in Figure 2. Our night earth texture is from NASA's Visible Earth project. Previous support for this project has been provided by the American Museum of Natural History.

REFERENCES

[Aka64] Syun-Ichi Akasofu. The development of the auroral substorm. *Planetary and Space Science*, 12(4):273 – 282, 1964.

[Bar00] G.V.G. Baranoski, J.G. Rokne, P. Shirley, T. Trondsen, and R. Bastos. Simulating the aurora borealis. In *Computer Graphics and Applications*, pages 422–432, october 2000.

[Bar05] Gladimir V. G. Baranoski, Justin Wan, Jon G. Rokne, and Ian Bell. Simulating the dynamics of auroral phenomena. *ACM Trans. Graph.*, 24(1):37–59, 2005.

[Coh94] D Cohen and Z Sheffer. Proximity clouds–an acceleration technique for 3D grid traversal. *The Visual Computer*, 11(1):27–38, 1994.

[Fan08] X. Fang, C. E. Randall, D. Lummerzheim, S. C. Solomon, M. J. Mills, D. R. Marsh, C. H. Jackman, W. Wang, and G. Lu. Electron impact

ionization: A new parameterization for 100 eV to 1 MeV electrons. *J. Geophys. Res.*, 113(A09311), 2008.

[Hed91] A. E. Hedin. Extension of the MSIS Thermosphere Model into the Middle and Lower Atmosphere. *J. Geophys. Res.*, 96(A2):1159–1172, 1991.

[Kaj84] James T. Kajiya and Brian P Von Herzen. Ray tracing volume densities. *SIGGRAPH Comput. Graph.*, 18(3):165–174, 1984.

[Law08] Orion Sky Lawlor, Matthew Page, and Jon Genetti. MPIglut: Powerwall programming made easier. *Journal of WSCG*, pages 130–137, February 2008.

[Lum92] D. Lummerzheim. Comparison of energy dissipation functions for high energy auroral electrons and ion precipitation. Technical Report UAG-R-318, Geophys. Inst., Univ. of Alaska-Fairbanks, April 1992.

[Men10] Jörg Mensmann, Timo Ropinski, and Klaus H. Hinrichs. An advanced volume raycasting technique using GPU stream processing. In *GRAPP Proceedings*, pages 190–198, 2010.

[Nis10] Y. Nishimura, J. Bortnik, W. Li, R. M. Thorne, L. R. Lyons, V. Angelopoulos, S. B. Mende, J. W. Bonnell, O. Le Contel, C. Cully, R. Ergun, and U. Auster. Identifying the driver of pulsating aurora. *Science*, pages 81–84, October 2010.

[Par98] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter-Pike Sloan. Interactive ray tracing for isosurface rendering. *Visualization Conference, IEEE*, 0:233, 1998.

[Pur02] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002.

[Rob87] R. G. Roble and E. C. Ridley. An auroral model for the NCAR thermospheric general circulation model (TGCM). *Annales Geophysicae, Series A - Upper Atmosphere and Space Sciences*, pages 369–382, december 1987.

[Ron06] Guodong Rong and Tiow-Seng Tan. Jump flooding in GPU with applications to Voronoi diagram and distance transform. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 109–116, New York, NY, USA, 2006. ACM.

[Sch05] Henning Scharsach. Advanced GPU raycasting. In *Proceedings of CESCG*, 2005.

[Sta99] Jos Stam. Stable fluids. In *SIGGRAPH '99 Conference Proceedings*, pages 121–128, 1999.

[Sto98] John Stone. An efficient library for parallel ray tracing and animation. Master's thesis, Dept. of Computer Science, University of Missouri Rolla, 1998. http://jedi.ks.uiuc.edu/~johns/.

[You69] A.T. Young. High-resolution photometry of a thin planetary atmosphere. *Icarus*, 11(1):1–23, March 1969.

A CALCULATING AIRMASS

The integral of atmospheric density along a ray, known as "airmass," is widely used in astronomy, and we use it to approximate both the aurora light lost to the atmosphere, and night sky light added. A gravitationally bound atmosphere of uniform temperature and composition falls off in density at an exponential rate with height: $D(z) = e^{-z/H}$, with the exponential constant H known as the atmosphere's "scale height." The airmass integral along a ray parameterized by t is then:

$$A = \int_{t_s}^{t_e} D(z(t))dt = \int_{t_s}^{t_e} e^{-z(t)/H}dt$$

Even assuming a spherical planet, height varies nonlinearly along the ray path: $z(t) = \text{length}(\vec{S}+t\vec{D}) - r = \sqrt{a+bt+ct^2} - r$, so:

$$A = \int_{t_s}^{t_e} e^{-\frac{\sqrt{a+bt+ct^2}-r}{H}}dt$$

This integral cannot be solved in closed form. A trigonometric substitution [You69] allows high-order terms to be discarded, giving an integral that is easy to evaluate at the surface of the planet or at infinity, but a general raytracer requires arbitrary start and end points. We do this by approximating $z(t)/H$ with a quadratic $m + lt + kt^2$. We can eliminate the linear term l by translating the ray parameter t to t', leaving m as the height of closest approach of the ray to the planet, and k as the quadratic slope of that approach, both measured in scale height units.

$$A \approx \int_{t'_s}^{t'_e} e^{-m-kt^2}dt$$

This integral can be evaluated exactly using the error function "erf":

$$A \approx e^{-m}\sqrt{\frac{\pi}{4k}}\left(\text{erf}(\sqrt{k}t'_e) - \text{erf}(\sqrt{k}t'_s)\right)$$

Some GPU languages like GLSL do not have a built-in erf, so we use the Winitzki approximation:

$$\text{erf}(x) \approx \sqrt{1 - e^{-x^2\frac{\frac{4}{\pi}+0.147x^2}{1+0.147x^2}}}$$

Despite the plentiful transcendentals, this performs quite well on the graphics card at runtime. Despite the stacked approximations, accuracy appears quite good as well, except where numerical roundoff causes the erf difference to approach zero. This case can be handled by either falling back to a linear approximation of $z(t)$, or by interpreting the finite difference of erf values as a scaled derivative of erf: e^{-kt^2}.