Peter Kutz

Demo Reel Breakdown

Photorealizer: Physically Based Renderer

photorealizer.com

I started writing this renderer in C++ in 2010. I knew almost nothing about rendering at the time, but my goal was to produce photorealistic pictures. I wrote the program entirely from scratch except for a few things: the C ++ standard library, a couple libraries for loading and saving bitmap images, GUI libraries, and a few miscellaneous functions. I designed the architecture from scratch as well, and it currently contains around 150 classes.

I rendered all of the images in this section of my reel myself using Photorealizer. I composed, lit, and shaded each of the renders as well. I found most of the 3D models and bitmap texture maps on the web. I have not post-processed the images outside of Photorealizer, except to resize them and, in one case, do some minor color-correction.

Here's a list of the rendered images in my reel, along with a few of the notable features that I used to create each one:

Muscle Car on Bridge: translational and rotational motion blur, Cook-Torrance and Oren-Nayar BRDFs, simulated car paint, lit by an HDR environment map
Teacup and Spoon: depth of field, custom tablecloth texture, lit by an HDR environment map
Snow Globe: a variety of features, lit by an HDR environment map and a spherical light source, snow globe modeled by me using Houdini
Blue Translucent Lucy Statue: subsurface scattering using Monte Carlo path tracing, Henyey–Greenstein phase function
Translucent Orangish Bunny: approximate multiple scattering using a dipole diffusion approximation (using the "Better Dipole" model (d'Eon, 2012)) and a precomputed hierarchical point cloud of irradiance samples (primarily based on "A Rapid Hierarchical Rendering Technique for Translucent Materials" (Jensen & Buhler, 2002)), path-traced single scattering
Diamonds with Dispersion: dispersion created using the Sellmeier equation and spectral rendering
Rough Glass Lucy Statue: transmission through rough surfaces (based on the paper **"**Microfacet Models for Refraction through Rough Surfaces" (Walter et al., 2007))
Photon Mapping Cornell Box: radiometrically-accurate progressive photon mapping
Gold Bunny Sun Caustics: photons emitted from an HDR environment map of a sun and sky that I rendered in my sky renderer, direct illumination disabled to better highlight the caustics
Rasterizer: z-buffer rasterizer with perspective-correct texture mapping and Phong shading
Lots of Airplanes: 1272 copies of the same airplane using instancing, transformations, and BVHs, with a total of 194 million polygons
HDR Environment Map Importance Sampling: importance sampling using the inverse of a discrete CDF proportional to the radiance distribution of the environment map
HDR Environment Map Importance Sampling and Translucent Bunny: my approximate multiple scattering system uses both direct and indirect illumination when precomputing the irradiance point cloud
Anaglyph Box of Spheres: stereo 3D accomplished using asymmetric frustum perspective projection

In addition to the specific features covered above, many or all of the renders in my demo reel feature various forms of importance sampling, anti-aliasing using a Gaussian reconstruction filter, OBJ model loading, texture mapping, bounding volume hierarchy (BVH) acceleration structure created using the surface area heuristic,

shading normals, Fresnel reflection and refraction, a quasi-random number generator, sRGB encoding, S-shaped transfer curves, progressive rendering, multithreading, a custom templatized linear algebra library, and more.

Sky Renderer

skyrenderer.blogspot.com

For my senior project at Penn, I wrote this atmospheric light transport simulator to render physically accurate images of the sky lit by the sun. I used some parts of Photorealizer as a starting framework, and then wrote everything on top of that myself. I did a lot of this research for this project, and used a variety of resources (papers from the atmospheric sciences literature, NASA data and models, etc.). Here are some of the things that I did to create my sky renderer:

- Wrote a spectral rendering system that works for all wavelengths of light
- Used the U.S. Standard Atmosphere (1976) for the composition of Earth's atmosphere
- Simulated Rayleigh scattering (for air molecules) and Mie scattering (for aerosols)
- Simulated full multiple scattering, not just single scattering
- Worked in SI units for all parts of the system and compared some intermediate results to measured data
- Used unbiased distance sampling (in place of ray marching) to choose scattering and absorption distances
- Implemented direct sun sampling to dramatically decrease render times (the sun only covers 0.00047% of the total celestial sphere)
- Implemented various other forms of importance sampling, and various optimizations
- Included preferential absorption by ozone, which is typically not included in sky models (ozone has a very large effect on the color of the sky, especially when the sun is low or below the horizon; at twilight ozone is the main cause of the deep blue color of the zenith sky)
- Employed colorimetry to convert spectral data to images that can be displayed on computer displays
- Generated and saved panoramic HDR images that can be used as light sources in Photorealizer (and other renderers)

In the fisheye renders in my reel, the camera is pointed straight up towards the zenith, the sun is at the top of the circle, and the Earth's shadow is along the bottom of the circle (when applicable).

GPU Path Tracer

gpupathtracer.blogspot.com

Karl Li and I built an interactive GPU path tracer in CUDA for our GPU programming course. It's a physically based, brute force, unbiased path tracer that can render pretty pictures fast. This class was the first time that either Karl or I had done any GPU programming.

I personally programmed much of the core path tracing algorithm, many of the key GPU-specific optimizations, the interactive camera control, all of the BSDFs (including specular reflection and refraction), importance sampling for BSDFs, participating media / subsurface scattering, depth of field, image sample accumulation, and gamma correction. I also refined other parts of the program.

Smoke Simulation

Semi-Lagrangian fluid simulation written in C++. I implemented advection, pressure projection, solid wall boundaries, vorticity confinement, buoyancy, modified Euler / Heun's RK2 time integration, a sparse matrix

data structure, and a preconditioned conjugate gradient method for pressure projection. Most of the work was based on the “Fluid Simulation” SIGGRAPH course notes by Bridson & Müller-Fischer (2007).

The videos show 3D smoke that I rendered on a 90x90x36 grid. The first video was rendered in an OpenGL renderer that was provided for the school assignment (I added colors to represent temperature and density). I rendered the second video in Photorealizer (my own renderer, see above) using its volumetric ray rendering system. Photorealizer’s volumetric rendering system (which is separate from the participating media / subsurface scattering features mentioned earlier) uses ray marching, cubic interpolation of density values, and light visibility caching.

SPH Liquid Simulation

peterkutz.com/sph

This is a particle-based liquid simulator that I wrote from scratch in C++. It features weakly-incompressible SPH, pressure, viscosity, surface tension, gravity, boundaries using ghost particles, anisotropic kernels for smooth and detailed surface reconstruction (based on the paper “Reconstructing Surfaces of Particle-Based Fluids Using Anisotropic Kernels” (Yu & Turk, 2010)), surface reconstruction using marching cubes, OBJ export, and OpenGL display, and much more.

The videos in the section illustrate a variety of features, as noted in the reel. I rendered the still images at the end in Photorealizer (my renderer) after surface construction and OBJ export of a frame from a dam break simulation containing 40,000 particles.

Jell-O Cube Simulation

This is a mass–spring system written in C++ for a school assignment. I implemented spring forces, collision detection and response, and RK1 and RK2 time integration, and I added springs and set spring constants. In addition to structural and shear springs I used a random network of interior bend springs to give the Jell-O a more natural feel.

Stephen Colbert’s Green Screen Challenge

peterkutz.com/videos

I made these two entries for Stephen Colbert’s Green Screen Challenge while in high school, but I’m still proud of them and I think they’re fun to watch, so I put some clips in the reel. I did the green screen work, light saber glow, rock smashing, lightsaber trail, and other effects and compositing in After Effects. I edited the video in Sony Vegas, and did the lightsaber handle animation in Flash. Clips from both were shown on *The Colbert Report*.

Strange Attractors

This is one of the procedural images that I’ve created using C++. This isn’t my most advanced work, but I think it’s quite pretty so I put it in the reel.