CSE 274: Ray Tracing

Lecturer: Ravi Ramamoorthi

Scribed by Owen Jow on October 04, 2018

1 History

- *Appel, '68, ray casting*: send ray through pixel, then from intersected object to light
- *Whitted, '80, recursive ray tracing*: send ray through pixel, then from intersected object to light and also in reflection/refraction direction (recursively)

2 Complexity

Let p be the number of pixels. Let n be the number of objects.

The naive ray tracing paradigm is "for all p pixels, for all n objects." Since we have to intersect each ray with each object, it has complexity $O(pn)$. Using acceleration, we can reduce this to $O(p \log n)$.

By contrast, the rasterization paradigm is "for all n objects, for all c pixels covered by the object." It has complexity $O(nc)$; presumably $c \ll p$.

So rasterization is much faster than naive ray tracing. However, accelerated ray tracing can have an advantage when applied to the complex, many-polygon scenes of today for which $n \gg p$ (basically the $\log n$ term comes in clutch). It also gives realistic effects such as inter-reflections for free.

3 Algorithm

For each pixel, trace a ray through the eye. Find the first intersection (if there is one), then trace secondary rays to all lights; recursively trace a reflection (/refraction) ray as well. Set pixel color to accumulated intensity from light illumination models along the way.*

* If the scene has a lot of objects or is in some kind of box, the recursion might be never-ending. Therefore we should set a max recursion depth.

Intersection tests are big. For example, ray-triangle intersection is typically optimized to a far greater degree than the "intersect ray with plane, check if inside triangle acc. to barycentric coordinates" method discussed in class.

To intersect transformed objects, we can apply the inverse transform M^{-1} to the ray, intersect with the normal version of the object, and finally apply the transform M to the intersected point.

For an area light source, we can grid up the light and trace a ray to each cell, replacing color banding with noise (good!) by randomizing ("jittering") the direction of the ray within each grid box.

4 Acceleration

One approach to acceleration is to make fewer intersections. We can do this using hierarchical bounding boxes; if a ray doesn't intersect with a bounding box it doesn't intersect with anything inside the bounding box.

After spatially subdividing the scene into grid cells, we can store overlapping triangles in the cells and march a ray along the grid. Along this march, we can intersect the ray *only* against triangles associated with passed-through cells. In doing so, we avoid intersecting against irrelevant objects.

5 Developments

Whitted's paper came out in ∼1980 and everyone was very excited, but production studios (e.g.) didn't exactly adopt it until 2010-11 when the new algorithms and better hardware finally got them all to switch. Things have only gotten better, and these days ray tracing can be done in real-time.

Back in 2002, people started to map elements of ray tracing (generation of primary rays, intersection routines, ...) into vertex and fragment shaders. A few years later, NVIDIA introduced their OptiX ray tracing API (like OpenGL for ray tracing), which provided the ability to do ray tracing on NVIDIA cards and get excellent performance.

Since then, there has been an even greater convergence between hardware and ray tracing. Only a month or so ago, NVIDIA announced and released its RTX graphics cards, integrated with the DirectX API (i.e. allowing for DirectX ray tracing) and able to trace billions of rays per second. (If rendering a million pixels, we effectively get a thousand rays per pixel per second.) One can do quite a bit with this, especially using the sampling and reconstruction methods of this class to reduce the number of rays traced.