Introduction to DirectX Raytracing:

Overview of Ray Tracing

Peter Shirley, NVIDIA

More information: http://intro-to-dxr.cwyman.org
Turner Whitted 1980  

included a bounding volume hierarchy  
reflection and refraction  
adaptive antialiasing

The role of the illumination model is to determine how much light is reflected to the viewer from a visible point on a surface as a function of light source direction and strength, viewer position, surface orientation, and surface properties. The shading calculations can be performed on three scales: microscopic, local, and global. Although the exact nature of reflection from surfaces is best explained in terms of microscopic interactions between light rays and the surface, most shaders produce excellent results using aggregate local surface data. Unfortunately, these models are usually limited in scope, i.e., they look only at light source and surface orientations, while ignoring the overall setting in which the surface is placed. The reason that shaders tend to operate on local data is that traditional visible surface algorithms cannot provide the necessary global data.

A shading model is presented here that uses global information to calculate intensities. Then, to support this shader, extensions to a ray tracing visible surface algorithm are presented.

1. Conventional Models

The simplest visible surface algorithms use shaders based on Lambert’s cosine law. The intensity of the reflected light is proportional to the dot product of the surface normal and the light source direction, simulating a perfect diffuser and yielding a reasonable looking approximation to a dull, matte surface. A more sophisticated model is the one devised by Bui-Tuong Phong [8]. Intensity from Phong’s model is given by

\[ I = I_a + k_d \sum \frac{r}{\pi} (\hat{n} \cdot \hat{L}) + k_s \sum \frac{r}{\pi} (\hat{n} \cdot \hat{L})^2, \]

where
Ray Tracing flavors

**Whitted** 1980: shiny
- reflection rays off perfect surfaces
- shadow rays to point sources

**Cook** 1984 (distribution ray tracing): fuzzy
- reflection rays are randomly perturbed off pure specular direction
- soft shadows by sending rays to random points on area light source

**Kajiya** 1986 (path tracing): diffuse interreflection
- Now go fully random for diffuse surfaces
A typical batch 1980s ray tracing program (Whitted style ray tracing)

For each pixel

Send a visibility ray (or many for antialiasing)

If a diffuse surface, send a shadow ray to each light source and shade

If a mirror surface, send a reflection ray
A typical batch 1990s ray tracing program (Cook style ray tracing)

Allow shadow rays to go to a random point on area light

Allow specular rays to be perturbed specularly around the ideal reflection
Kajiya style diffuse interreflection

Note bounce light on bottom of squashed sphere on the upper left

Figure 6. A sample image. All objects are neutral grey. Color on the objects is due to caustics from the green glass balls and color bleeding from the base polygon.
Denoising
Ray Tracing Versus Rasterization for primary visibility

Rasterization: stream triangles to pixel buffer to see that pixels they cover

Ray Tracing: stream pixels to triangle buffer to see what triangles cover them
# Rasterization vs Ray Tracing

<table>
<thead>
<tr>
<th>Key Concept</th>
<th>Rasterization</th>
<th>Ray Tracing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fundamental question</td>
<td>What pixels does geometry cover?</td>
<td>What is visible along this ray?</td>
</tr>
<tr>
<td>Key operation</td>
<td>Test if pixel is inside triangle</td>
<td>Ray-triangle intersection</td>
</tr>
<tr>
<td>How streaming works</td>
<td>Stream triangles (each tests pixels)</td>
<td>Stream rays (each tests intersections)</td>
</tr>
<tr>
<td>Inefficiencies</td>
<td>Shade many tris per pixel (overdraw)</td>
<td>Test many intersections per ray</td>
</tr>
<tr>
<td>Acceleration structure</td>
<td>(Hierarchical) Z-buffering</td>
<td>Bounding volume hierarchies</td>
</tr>
<tr>
<td>Drawbacks</td>
<td>Incoherent queries difficult to make</td>
<td>Traverses memory incoherently</td>
</tr>
</tbody>
</table>
What is a Ray?

USUALLY, at least conceptually, it is a 3D origin **point** and a **direction** of propagation, like a laser beam.

How do I represent such a thing in code? As a line somehow?

In 2D a line might be \( y = mx + b \) (implicit equation)

In 3D can we do that? No, there is no implicit equation for a 1D object embedded in 3D

How about geometrically? The intersection of two planes? Is rarely done.
Blending points

A linear combination of two points $A$ and $B$

$$\text{vec3 } A$$

$$\text{vec3 } B$$

$$\text{vec3 } C = 0.9f^*A - 0.2f^*B$$

Arbitrary linear combinations of $A$ and $B$ like that are a point in the plane through the origin $\text{vec3}(0,0,0)$ and $A$ and $B$. 
Weighted average of points

vec3 A
vec3 B
vec3 C = (1.0f - 0.37f) * A + 0.37f * B
C will be on the 3D line through A and B
A “parametric” 3D line

vec3 A
vec3 B
float t
vec3 C = (1-t)*A + t*B

t is the **parameter** that says where on the line C is
A ray as a half line

\[ t \geq 0 \]
Variations on half-lines

Ray as half-line through $A$ and $B$ is all points $P$:

$P(t) = (1-t)A + tB$

$P(t) = A + t(B-A)$

$P(t) = A + tV$

$P(s) = A + s\text{unit\_vector}(V)$
Non-infinite rays

0.05 < t < MAXFLOAT
0.0 < t < 1.0
-0.2 < t < 0.5
What do we do with rays?

Ask various questions:

1. Does a ray hit anything for $t > \epsilon$ (shadow query)
2. How far away is the nearest thing in the direction of the ray (proximity query)
3. What if anything does a ray first hit for $t > \epsilon$ (viewing query)
4. What if anything does a ray first hit for $13.2 < t < 22.5$ (viewing query)
5. What are all the things hit by a ray (useful for things like CSG)
How to make rays faster

A brute force ray tracer

hit(ray)

Test ray against every triangle

A faster ray tracer (if many rays are sent):

Organize triangles into a tree structure

hit(ray)

Traverse tree and test ray against triangles not culled in traversal
The Bounding Volume Hierarchy (BVH)

There are many data structures used to help faster ray tracing but the BVH is the most common used.

Building a BVH is typically $O(N \log N)$ for $N$ triangles.

Updating a BVH is typically $O(N)$ for $N$ triangles.

Traversing a BVH (i.e., tracing a ray) is typically $O(\log N)$.

Usual program: build BVH as preprocess, then per frame then loop over

- If needed update BVH
- Trace rays
Closing remarks

There is no single ray tracing ``algorithm”

There are lots of algorithms that use ray tracing

The ray tracing query has lots of unexplored possibilities in the real time world-- it has never been available before

When programmable shading was made available to practitioners, they did many creative things few would have predicted. There isn’t a “right” way to use ray tracing; add it as a arrow in your programming quiver and go create!