Lecture 6:

Ray Tracing Basics

Image Synthesis
Stanford CS348b, Spring 2014
Overview

- Today:
  - Basic algorithms
  - Ray-surface intersection
  - Overview of pbrt

- Next lecture:
  - Efficiently rendering scenes with large numbers of primitives

- Next next lecture:
  - Parallelism for high-performance ray tracing
Light Rays

- Three key ideas about light rays
  - Light travels in straight lines (mostly)
  - Light rays do not interfere with each other if they cross
  - Light rays travel from light sources to the eye, but the physics is invariant under path reversal–reciprocity
Ray Tracing in Computer Graphics

- Appel 1968—Ray casting
  - Generate an image by sending one ray per pixel
  - Check for shadows by sending a ray to the light
Primary Visibility: Ray Tracing vs. Rasterization

- Essentially just a loop interchange...
- Spatial data structures / culling for both so that loops aren’t exhaustive
Ray Tracing in Computer Graphics

“An Improved Illumination Model for Shaded Display”, T. Whitted, CACM 1980

Time:
VAX 11/780 (1979): 74 min
PC (2006): 6 sec
GPU (2009): 30 fps
Spheres-over-plane.pbrt (Mirror depth 1)
Spheres-over-plane.pbrt (Mirror depth 2)
Spheres-over-plane.pbrt (Mirror depth 3)
Spheres-over-plane.pb (Mirror depth 10)
Spheres-over-plane.pbtt (Glass depth 1)
Spheres-over-plane.pbrt (Glass depth 2)
Spheres-over-plane.pbtrt (Glass depth 3)
Spheres-over-plane.pbrt (Glass+Mirror depth 10)
Ray-Surface Intersection
Ray-Plane Intersection

Ray representation: \( r(t) = o + t \vec{d} \)

Want to find \( t \) where the ray intersects the plane.

Plane representation: \( (p - p') \cdot \vec{n} = 0 \)
\[ ax + by + cz + d = 0 \]
Ray-Plane Intersection

Ray representation: \( r(t) = o + t \vec{d} \)

Plane representation: \((p - p') \cdot \vec{n} = 0\)

Substitute ray equation into plane equation:

\[
(p - r(t)) \cdot \vec{n} = (p - (o + t \vec{d})) \cdot \vec{n} = 0
\]

\[
t = -\frac{(o - p) \cdot \vec{n}}{\vec{d} \cdot \vec{n}}
\]
Finding The Closest Intersection

\[ r(0) \]

\[ r(t) \]

\[ \vec{d} \]

\[ r(t_0) \]

\[ r(t_1) \]
Optimizing Ray-Box

\[ t = -\frac{(\mathbf{o} - \mathbf{p}) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}} \quad \rightarrow \quad t = -\frac{\mathbf{o}_x - \mathbf{p}_x}{\mathbf{d}_x} \]
What About Rays Parallel to a Plane?

$t = -\frac{o_x - p_x}{d_x}$

Math says: $t = -\frac{o_x - p_x}{0} = \pm \infty$

IEEE Floating Point says: $\infty$

$+\infty$ returns $> \text{all other floating-point numbers}$

$-\infty$ returns $< \text{all other floating-point numbers}$

Whew!
[Not Matt]: But be careful

- Multiplying by inverse is a good, but dangerous optimization:
  - $t = (p_x - o_x) \times invD_x$

- When $d_x = +0$, $invD_x = +infty$ [Okay]

- What happens when $p_x == o_x$?
  - $t = (0) \times +infty \Rightarrow NaN$

- Luckily, comparisons with NaN return false
  - $NaN < notNan \Rightarrow False$
  - $Nan > notNan \Rightarrow False$
  - So order your comparisons as: maybeNaN < curValue

- Note: STL “swaps” this -- std::min(curValue, maybeNaN)
Review: Geometric Building-Blocks

- The signed area of the parallelogram given by the vectors $v_1 = (x_1, y_1)$ and $v_2 = (x_2, y_2)$ is given by
  \[
  \begin{vmatrix}
  x_1 & x_2 \\
  y_1 & y_2
  \end{vmatrix}
  = (x_1 y_2) - (x_2 y_1)
  \]

- Half of this area is the area of the triangle they specify

- The area of a triangle with vertices $p_0 = (x_0, y_0)$, $p_1 = (x_1, y_1)$, and $p_2 = (x_2, y_2)$ is
  \[
  \begin{vmatrix}
  x_1-x_0 & x_2-x_0 \\
  y_1-y_0 & y_2-y_0
  \end{vmatrix}
  = 0.5 \left( (x_1-x_0)(y_2-y_0) - (x_2-x_0)(y_1-y_0) \right)
  \]
Barycentric Coordinates

- $a_0(p) = \text{Area}(p_1, p_2, p)$
- $a_1(p) = \text{Area}(p_2, p_0, p)$
- $a_2(p) = \text{Area}(p_0, p_1, p)$

- Define barycentric coordinates:
  - $b_0 = a_0 / \text{Area}(p_0, p_1, p_2)$
  - $b_1 = a_1 / \text{Area}(p_0, p_1, p_2)$
  - $b_2 = a_2 / \text{Area}(p_0, p_1, p_2)$

- $p$ is inside the triangle if:
  - $0 \leq b_0 \leq 1$
  - $0 \leq b_1 \leq 1$
  - $0 \leq b_2 \leq 1$
Ray-Triangle Intersection

- 3 points define a plane: \( p = b_0p_0 + b_1p_1 + b_2p_2 \)
- Find ray-plane intersection point
- Test whether that point is inside the triangle

\[
\begin{bmatrix}
p_0 & p_1 & p_2
\end{bmatrix}
\begin{bmatrix}
b_0 \\
b_1 \\
b_2
\end{bmatrix}
= \begin{bmatrix}
p
\end{bmatrix}
\]

\[
\begin{bmatrix}
b_0 \\
b_1 \\
b_2
\end{bmatrix}
= \begin{bmatrix}
p_0 & p_1 & p_2
\end{bmatrix}^{-1}
\begin{bmatrix}
p
\end{bmatrix}
\]
Ray-Triangle Intersection

\[ \mathbf{o} + t \mathbf{d} = (1 - b_1 - b_2) \mathbf{p}_0 + b_1 \mathbf{p}_1 + b_2 \mathbf{p}_2 \]

\[
\begin{bmatrix}
  t \\
  b_1 \\
  b_2 
\end{bmatrix} = \frac{1}{\mathbf{s}_1 \cdot \mathbf{e}_1} \begin{bmatrix}
  \mathbf{s}_2 \cdot \mathbf{e}_2 \\
  \mathbf{s}_1 \cdot \mathbf{s} \\
  \mathbf{s}_2 \cdot \mathbf{d} 
\end{bmatrix}
\]

Where

\[
\begin{align*}
\mathbf{e}_1 &= \mathbf{p}_1 - \mathbf{p}_0 \\
\mathbf{e}_2 &= \mathbf{p}_2 - \mathbf{p}_0 \\
\mathbf{s} &= \mathbf{o} - \mathbf{p}_0 \\
\mathbf{s}_1 &= \mathbf{d} \times \mathbf{e}_2 \\
\mathbf{s}_2 &= \mathbf{s} \times \mathbf{e}_1 
\end{align*}
\]

[Möller and Trumbore 1997]
Ray-Sphere Intersection

Ray representation: \( r(t) = o + t\, d \)

Sphere representation: \( ||p - c||^2 - r^2 = 0 \)
\[
(o + t\, d - c)^2 - r^2 = 0
\]

\[at^2 + bt + c = 0\]

\[a = d \cdot d\]
\[b = 2(o - c) \cdot d\]
\[c = ((o - c) \cdot (o - c)) - r^2\]
Quadrics
Ray-Implicit Surface Intersection

Implicit surface definition:

\[ f(x, y, z) = 0 \]

Substitute ray equation:

\[
\begin{align*}
x &= o_x + td_x \\
y &= o_y + td_y \\
z &= o_z + td_z
\end{align*}
\]

Univariate root finding:

\[ f^*(t) = 0 \]
Floating-Point Precision and Ray Intersection...

radius = 1

(0,0) to (1930.420, 1973.505)
Numerical Error in Computed Intersection Point

Computed Intersection
(1930.4196..., 1973.5054...)

Most Accurate Intersection
(1929.7203..., 1972.7897...)
Numerical Error in Computed Intersection Point

Computed Intersection
(1930.4196..., 1973.5054...)

Most Accurate Intersection
(1929.7203..., 1972.7897...)

Abs. error: 0.69923
Rel. error: 0.0363%
5728 ulps

Abs. error: 0.71486
Rel. error: 0.0363%
5868 ulps
### What Happened?

\[ at^2 + bt + c = 0 \]

<table>
<thead>
<tr>
<th>Computed (32-bit float)</th>
<th>Actual Value</th>
<th>ulp Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ a = \vec{d} \cdot \vec{d} ]</td>
<td>1.0000000000</td>
<td>1.0000000000</td>
</tr>
<tr>
<td>[ b = 2(o - c) \cdot \vec{d} ]</td>
<td>-5521.318359375</td>
<td>-5521.318643019</td>
</tr>
<tr>
<td>[ c = ((o - c) \cdot (o - c)) - r^2 ]</td>
<td>7621239.0000</td>
<td>7621238.8894</td>
</tr>
<tr>
<td>[ b^2 ]</td>
<td>30484956.00</td>
<td>30484959.5577</td>
</tr>
<tr>
<td>[ 4ac ]</td>
<td>30484956.00</td>
<td>30484955.5577</td>
</tr>
<tr>
<td>[ b^2 - 4ac ]</td>
<td>0.0000000000</td>
<td>3.9999999999</td>
</tr>
</tbody>
</table>
**Error In Less Pathological Cases**

Center: (28.998543, 21.635460), radius 1

Most Accurate: (28.012575..., 21.468523...)

Computed: (28.012598..., 21.468538...)

8 ulps

12 ulps
Effect of Floating-Point
Effect of Floating-Point Precision Issues
Floating-Point Precision Remedies

- Use double (fp64) precision rather than float (fp32)
  - Can help significantly
  - May increase memory needed for scene representation, lower performance on some platforms
- Ignore reintersection with the last object hit
  - Only works for flat objects (e.g. triangles)
  - No help if model has coincident triangles
- Have a $t_{\text{min}}$ along ray to ignore close intersections
  - Hard to choose a good $t_{\text{min}}$ that isn’t too small and isn’t too big
  - Too big: miss valid nearby intersections
Remedy: Refine the Intersection Point

- Find more accurate intersection point using first one
  - Evaluate parametric surface at \((u,v)\)
  - Project onto surface

This point has error both in \(t\) and also from \(r(t) = \mathbf{o} + t \mathbf{d}\)
Remedy: Find Points on Each Side

- Offset refined point a few 10s of ulps in each direction along the surface normal
- Choose appropriate offset point for spawned rays

Reminder: at x=1.0f, 10 ulps = $1.19 \times 10^{-6}$
PBRT Overview
Table 1.1: Main Interface Types. Most of pbrt is implemented in terms of 13 key abstract base classes, listed here. Implementations of each of these can easily be added to the system to extend its functionality.

<table>
<thead>
<tr>
<th>Base class</th>
<th>Directory</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shape</td>
<td>shapes/</td>
<td>3.1</td>
</tr>
<tr>
<td>Aggregate</td>
<td>accelerators/</td>
<td>4.2</td>
</tr>
<tr>
<td>Camera</td>
<td>cameras/</td>
<td>6.1</td>
</tr>
<tr>
<td>Sampler</td>
<td>samplers/</td>
<td>7.2</td>
</tr>
<tr>
<td>Filter</td>
<td>filters/</td>
<td>7.7</td>
</tr>
<tr>
<td>Film</td>
<td>film/</td>
<td>7.8</td>
</tr>
<tr>
<td>Material</td>
<td>materials/</td>
<td>9.2</td>
</tr>
<tr>
<td>Texture</td>
<td>textures/</td>
<td>10.3</td>
</tr>
<tr>
<td>VolumeRegion</td>
<td>volumes/</td>
<td>11.3</td>
</tr>
<tr>
<td>Light</td>
<td>lights/</td>
<td>12.1</td>
</tr>
<tr>
<td>Renderer</td>
<td>renderers/</td>
<td>1.3.3</td>
</tr>
<tr>
<td>SurfaceIntegrator</td>
<td>integrators/</td>
<td>Ch. 15 intro</td>
</tr>
<tr>
<td>VolumeIntegrator</td>
<td>integrators/</td>
<td>16.2</td>
</tr>
</tbody>
</table>
Figure 1.17: Class Relationships for the Main Rendering Loop in the SamplerRenderer::Render() Method in renderers/sample.cpp. The Sampler provides a sequence of sample values, one for each image sample to be taken. The Camera turns a sample into a corresponding ray from the film plane, and the Integrators compute the radiance along that ray arriving at the film. The sample and its radiance are given to the Film, which stores their contribution in an image. This process repeats until the Sampler has provided as many samples as are necessary to generate the final image.
class Shape {
    public:
        // Shape interface
        virtual BBox ObjectBound() const = 0;
        virtual BBox WorldBound() const;
        virtual bool CanIntersect() const;
        virtual void Refine(vector<Reference<Shape>> &refined) const;
        virtual bool Intersect(const Ray &ray, float *tHit, 
                                float *rayEpsilon, DifferentialGeometry *dg) const;
        virtual bool IntersectP(const Ray &ray) const;
        virtual void GetShadingGeometry(const Transform &obj2world, 
                                         const DifferentialGeometry &dg, 
                                         DifferentialGeometry *dgShading) const {
            *dgShading = dg;
        }
        virtual float Area() const;
    
    // Shape Public Data
    const Transform *ObjectToWorld, *WorldToObject;
};
Differential Geometry Representation

```c
struct DifferentialGeometry {
    // Filled in by Shape::Intersect()
    Point p;
    Normal nn;
    float u, v;
    const Shape *shape;
    Vector dpdu, dpdv;
    Normal dndu, dndv;
};
```