Ray Tracing

- OpenGL projects triangles onto the image plane and rasterizes them to determine which pixels they cover
- Scanline rendering is a per triangle operation

- Ray Tracing is a per pixel operation
- For each pixel, a ray is created to represent the light coming inwards from the world that hits that pixel
- That ray is traced backwards out of the camera into the world to find out where the light came from
  - and thus determine what color to shade that pixel
Parallelization

- Ray tracing is inherently parallel, since the rays for one pixel are independent of the rays for other pixels
- Can take advantage of modern parallel CPU/GPU/Clusters to significantly accelerate a ray tracer
  - Threading (e.g., Pthread, OpenMP) distributes rays across cores
  - Message Passing Interface (MPI) distributes rays across processors on different machines
  - OptiX/CUDA distributes rays on the GPU
- Memory coherency helps when distributing rays to various threads/processors
  - Assign spatially neighboring rays (on the image plane) to the same core/processor
  - These rays tend to intersect with the same objects in the scene, and thus access the same memory
Real Time Ray-Tracing

- NVIDIA Optix...
Real Time Ray-Tracing

- PS5?
Basic Ray Tracing
Ray Tracing

- Backwards trace the path of light through pixels on the image plane
- Simulate the interaction of light with objects
Aperture & Image Plane

Start with a pupil/aperture (focal point), along with an image plane with pixels
Constructing Rays

• Ray equation:
  \[ R(t) = E + t(P - E) \]
  \[ t \in [0, +\infty) \]

• Eye pupil located at \( t=0 \)
• Pixel center located at \( t=1 \)
Object Intersection

Shoot the ray from the eye through the pixel, and find the first object it hits.
 Shoot a shadow ray toward the light, and see if the intersection point is shadowed
If the light is visible from the intersection point, compute the shading using the light source

\[ k_d I \max (\hat{\mathbf{l}} \cdot \hat{\mathbf{n}}, 0) + k_s I \max (\hat{\mathbf{r}} \cdot \hat{\mathbf{v}}, 0)^p \]
Shading the Intersection Point

Similar to OpenGL:

• The shading of each intersection point is the sum of contributions from all light sources
  – Cast rays from the intersection point to all light sources

• Light types:
  – Ambient light, point light, directional light, spot light, area light, volume light, etc.

• Material properties:
  – Diffusion, specular, shininess, emission, etc.

• Shading models:
  – Diffusive shading, Phong shading, BRDFs, etc.
Pixel Color

Use the results of the shading to record a color for that pixel.
Pseudocode

Image Raytrace (Eye eye, Scene scene, int width, int height) {
    Image image = new Image (width, height) ;
    for (int i = 0 ; i < height ; i++)
        for (int j = 0 ; j < width ; j++)
            {
                Ray ray = RayThruPixel (eye, i, j) ;
                Intersection hit = Intersect (ray, scene) ;
                image[i][j] = FindColor (hit) ;
            }
    return image ;
}
Light Types

- Point Light
- Directional Light
- Spot Light
- Area Light
- Area Light from a light tube
- Volume light
Area Lights

– Treat as a bunch of point lights
– Shoot a number of rays from the intersection point to different points on the area light
– Take the average of the results
– Creates soft shadows
Question 1 (short/long)

• If you break an area light into 100 point lights, how should you set the strength of each point light?
Shadow Rays

• Detect shadows by casting rays to the light source

\[ R(t) = S + t(L - S) \]
\[ t \in [\epsilon, 1) \]

• Test for occluder intersection
  – No occluder intersection: shade normally (e.g. Phong, BRDF)
  – Yes, occluder intersection: skip light (don’t skip ambient)
Spurious Self-Occlusion

Incorrect self-shadowing

Correct
Spurious Self-Occlusion

• Add $\varepsilon$ to the starting point of shadow rays to avoid accidental re-intersection with the surface: $t \in [\varepsilon, 1)$
  – Can fail for grazing rays near the object’s silhouette

• Better to offset in the normal direction
  – The direction of the shadow ray from the perturbed point to the light may be slightly different from the original direction

• Need to avoid placing the new starting point too close to or inside other nearby objects!
Question 2 (short/long)

• Why not just ignore the object completely when casting a shadow ray?
Ray-Object Intersections
Ray-Object Intersections

• Given ray \( R(t) = A + tD \), find the first intersection with any object where \( t \geq t_{\text{min}} \) and \( t \leq t_{\text{max}} \)

• The object geometry can be a polygon mesh, an implicit surface, a parametric surface, etc.

• Doesn’t require decomposition into triangles (as OpenGL did)

• Task: write intersection code for all kinds of different geometries
Ray-Sphere Intersections
Ray-Sphere Intersection

- Ray equation: \( R(t) = A + tD \)
- Implicit equation for a sphere: \( (X - C)^2 = r^2 \)
- Combine: \( (A + tD - C)^2 = r^2 \)
- Quadratic equation:
  \[
  t^2 + 2(A - C) \cdot Dt + (A - C)^2 - r^2 = 0
  \]
- Discriminant:
  \[
  \Delta = 4[(A - C) \cdot D]^2 - 4(A - C)^2 + r^2
  \]
Ray-Sphere Intersection

Two solutions

\[ \Delta > 0 \]

One solution

\[ \Delta = 0 \]

Imaginary

\[ \Delta < 0 \]

For the case with two solutions, choose the first intersection (careful when inside sphere)
Ray-Sphere Intersection

• Intersection Point:

\[ P = A + t_{\text{int}} D \]

• Intersection Normal:

\[ N = \frac{P - C}{|P - C|} \]
Implicit Surfaces
Ray Implicit Surface Intersection

• Substitute ray equation into the implicit surface representation $f(x, y, z) = 0$
• Search for the first root

The Ray-Sphere intersection was an example of this!
Ray-Plane Intersections
Ray-Plane Intersection

- Ray equation: \( R(t) = A + tD \)
- Implicit equation for a plane: \( ax + by + cz + d = 0 \)
- Combine and solve for \( t \) to find the intersection point:

\[
a(x_A + tx_D) + b(y_A + ty_D) + c(z_A + tz_D) + d = 0
\]

Just another implicit surface intersection!
Ray-Triangle Intersections
Ray-Triangle Intersection

- First, find the ray-plane intersection for the plane containing the triangle
- Then, project the 3 triangle vertices and the ray-plane intersection point to a truly 2D plane
- Finally, run the point-inside-triangle test in 2D as we did for rasterization in Week 2
Or, avoiding projection...

- Compute normal direction $\mathbf{n}$ orthogonal to edge $\mathbf{e}$, and pointing towards the edge’s opposite vertex in the plane of the triangle:

$$
\mathbf{n} = \mathbf{e}_1 - \frac{\mathbf{e}_1 \cdot \mathbf{e}_2}{\|\mathbf{e}_2\|^2} \mathbf{e}_2
$$

- Given endpoint $\mathbf{P}_1$ of $\mathbf{e}$, the intersection point is “inside” the edge when $\langle \mathbf{P} - \mathbf{P}_1, \mathbf{n} \rangle \geq 0$
Recall Barycentric Coordinates...

\[ u = p_0 - p_1 \]

\[ \beta v \]

\[ \alpha u \]

\[ \alpha u + \beta v \]

\[ v = p_2 - p_1 \]

\[ \alpha + \beta \leq 1 \]

\[ \alpha, \beta > 0 \]
Ray-Triangle Intersection

- Ray equation: \( R(t) = A + tD \)
- Parametric equation for triangle:
  \[
  X = P_1 + \alpha(P_2 - P_1) + \beta(P_3 - P_1)
  \]
- Combine: \( A + tD = P_1 + \alpha(P_2 - P_1) + \beta(P_3 - P_1) \)
Ray-Triangle Intersection

\[ A + tD = P_1 + \alpha(P_2 - P_1) + \beta(P_3 - P_1) \]

\[ P_1 = (x_1, y_1, z_1), \ P_2 = (x_2, y_2, z_2), \ P_3 = (x_3, y_3, z_3) \]

\[
\begin{align*}
x_A + tx_D &= x_1 + \alpha(x_2 - x_1) + \beta(x_3 - x_1) \\
y_A + ty_D &= y_1 + \alpha(y_2 - y_1) + \beta(y_3 - y_1) \\
z_A + tz_D &= z_1 + \alpha(z_2 - z_1) + \beta(z_3 - z_1)
\end{align*}
\]

3 equations with 3 unknowns...
Ray-Triangle Intersection

Matrix form:

\[
\begin{bmatrix}
  x_2 - x_1 & x_3 - x_1 & -x_D \\
  y_2 - y_1 & y_3 - y_1 & -y_D \\
  z_2 - z_1 & z_3 - z_1 & -z_D \\
\end{bmatrix}
\begin{bmatrix}
  \alpha \\
  \beta \\
  t \\
\end{bmatrix} =
\begin{bmatrix}
  x_A - x_1 \\
  y_A - y_1 \\
  z_A - z_1 \\
\end{bmatrix}
\]

Satisfying

\[
\begin{cases}
  t_{\text{min}} \leq t \leq t_{\text{max}} \\
  0 \leq \alpha \leq 1 \\
  0 \leq \beta \leq 1 - \alpha
\end{cases}
\]
Ray-Triangle Intersection

• Cramer’s rule...

\[ \alpha = \frac{\begin{vmatrix} x_A - x_1 & x_3 - x_1 & -x_D \\ y_A - y_1 & y_3 - y_1 & -y_D \\ z_A - z_1 & z_3 - z_1 & -z_D \end{vmatrix}}{\begin{vmatrix} x_2 - x_1 & x_3 - x_1 & -x_D \\ y_2 - y_1 & y_3 - y_1 & -y_D \\ z_2 - z_1 & z_3 - z_1 & -z_D \end{vmatrix}} \]

\[ \beta = \frac{\begin{vmatrix} x_2 - x_1 & x_A - x_1 & -x_D \\ y_2 - y_1 & y_A - y_1 & -y_D \\ z_2 - z_1 & z_A - z_1 & -z_D \end{vmatrix}}{\begin{vmatrix} x_2 - x_1 & x_3 - x_1 & -x_D \\ y_2 - y_1 & y_3 - y_1 & -y_D \\ z_2 - z_1 & z_3 - z_1 & -z_D \end{vmatrix}} \]

\[ t = \frac{\begin{vmatrix} x_2 - x_1 & x_3 - x_1 & x_A - x_1 \\ y_2 - y_1 & y_3 - y_1 & y_A - y_1 \\ z_2 - z_1 & z_3 - z_1 & z_A - z_1 \end{vmatrix}}{\begin{vmatrix} x_2 - x_1 & x_3 - x_1 & -x_D \\ y_2 - y_1 & y_3 - y_1 & -y_D \\ z_2 - z_1 & z_3 - z_1 & -z_D \end{vmatrix}} \]

• Only 4 unique matrices
• have some common columns
• so the number of operations can be reduced by reusing numbers when computing determinants
bool RayTriangle (Ray R, Vec3 V1, Vec3 V2, Vec3 V3, Interval [t_{\text{min}}, t_{\text{max}}])
{
    compute t;
    if(t < t_{\text{min}} \text{ or } t > t_{\text{max}}) return false;
    compute \alpha;
    if(\alpha < 0 \text{ or } \alpha > 1) return false;
    compute \beta;
    if(\beta < 0 \text{ or } \beta > 1- \alpha) return false;
    return true;
}

// Notice conditions for early termination.
// Recall: for 2D point inside triangle test, one can return early as well when testing the 3 edges
Question 3 (short/long)

- Why might we need more than one way of computing ray-triangle intersections?
Normals

• Barycentric interpolation
  – As usual, interpolate from the 3 normals at the triangle’s vertices:

\[ N = (1 - \alpha - \beta)N_1 + \alpha N_2 + \beta N_3 \]

• Bump mapping, normal mapping, displacement mapping
  – all done similarly as in OpenGL (textures too...)
Transformed Objects
Ray Tracing Transformed Objects

• A triangle is still a triangle after transformation
• But a sphere can become an ellipsoid:

• Write another intersection routine?
• Or, reuse ray-sphere intersection code....
Ray Tracing Transformed Objects

• Intersect the untransformed object with the inverse-transformed ray (put into object space)

\[ M \cdot S \]

\[ M^{-1} \cdot R \]

• Transform the results back into world coordinates

\[ p = Mp \quad \text{and} \quad N' = M^{-T}N \]

Careful with the normal!
Final Project
Question 4 (short form)

• Brainstorm some on your final project...
Question 4 (long form)

• Brainstorm some on your final project...
  (at least 500 words)