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 instead works as 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.
Real Time Ray-Tracing

• Typically not used in games
  – Could be used in games, but typically considered too expensive
• However, GPUs are very good at tasks that are easily parallelizable
• For example, NVIDIA Optix real time ray tracer...
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 among cores
  – Message Passing Interface (MPI) distributes rays among processors across different machines
  – OptiX/CUDA distributes rays on the GPU

• Memory coherency helps when distributing rays to various threads/processors
  – Assign the 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 need access to the same memory
Ray Tracing

- Generate an image by backwards tracing the path of light through pixels on an image plane
- Simulate the interaction of light with objects
Basic Ray Tracing Algorithm

Start with an eye pupil or aperture (a 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
Shadow Ray

Shoot a shadow ray toward the light, and see if the intersection point is in the shadow.
Shading

If not in shadow, compute shading from the light source

\[ k_d I \max(\hat{\mathbf{l}} \cdot \hat{\mathbf{n}}, 0) + k_s I \max(\hat{\mathbf{r}} \cdot \hat{\mathbf{v}}, 0)^p \]
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 ;
}
Ray-Object Intersections

• Given a 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 have to be decomposed into triangles (as was required by OpenGL)
• Goal: write intersection code for all kinds of various geometries
Ray-Sphere Intersection

- Ray equation: \( R(t) = A + tD \)
- Implicit equation for sphere: \( (X - C)^2 = r^2 \)
- Combine them together: \( (A + tD - C)^2 = r^2 \)
- Quadratic equation in \( t \):
  \[
  t^2 + 2(A - C) \cdot Dt + (A - C)^2 - r^2 = 0
  \]
- With 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 (smaller \( t \))
Ray-Sphere Intersection

- Intersection Point:
  \[ P = A + t_{\text{int}} D \]

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

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

- Ray equation: $R(t) = A + tD$
- Implicit equation for a plane: $ax + by + cz + d = 0$
- Combine them together and solve for $t$ to find the point of intersection: $a(x_A + tx_D) + b(y_A + ty_D) + c(z_A + tz_D) + d = 0$
- For ray-triangle intersection, we can project the 3 vertices of the triangle and the intersection point with the plane onto a truly 2D plane, and run the point-inside-triangle test in 2D as we did for rasterization in Lecture 2
Or avoiding projection...

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

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

- For an endpoint $\mathbf{P}_1$ of $\mathbf{e}$, test whether $(\mathbf{P} - \mathbf{P}_1) \cdot \mathbf{n} \geq 0$ to see if the intersection point is inside that edge.
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...

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

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

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

- The 4 unique matrices have some common columns, so we can reduce the number of operations by reusing numbers when computing determinants
bool RayTriangle (Ray R, Vec3 V1, Vec3 V2, Vec3 V3, Interval [t_{min}, t_{max}])
{
    compute t;
    if(t < t_{min} or t > t_{max}) return false;
    compute α;
    if(α < 0 or α > 1) return false;
    compute β;
    if(β < 0 or β > 1- α) return false;
    return true;
}

// Notice the conditions for early termination.
// Recall: for the 2D point inside triangle test, one can return early as well when testing the 3 edges
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...)
Ray Tracing Transformed Objects

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

![Triangles](image1.png)

![Ellipsoid](image2.png)

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

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

\[ \mathbf{M} \cdot \mathbf{S} \]

• Transform the results back into world coordinates

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

Be careful with the normal!
Shading the Intersection Point

• Similar to the OpenGL shading model

• The shading on 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.
Light Types

Point Light

Directional Light

Spot Light

Area Light

Area Light from a light tube

Volume light
Area Lights

– Treat the area light like 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
Shadows Rays

• Detect shadows by casting rays to the light source

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

• 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 \( \epsilon \) to the starting point of shadow rays to avoid accidental re-intersection with the original surface: \( t \in [\epsilon, 1) \)
  - This can often fail for grazing shadow rays near the objects silhouette

- Better to offset the intersection point in the normal direction from the surface
  - The direction of the shadow ray shot from the perturbed point to the light may be slightly different from the direction of the original shadow ray

- May also need to avoid placing the new starting point too close to or inside other nearby objects!

The perturbed point may still be inside the object

offset in the ray direction

offset in the normal direction