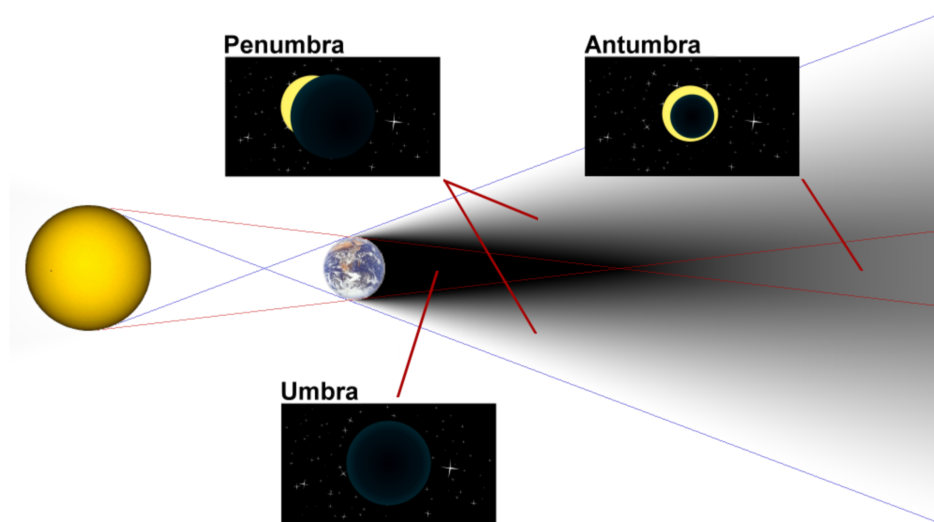
CSE 167x — Computer Graphics
Ramamoorthi — UC San DiegoX — Unit 3

1 Examples of Effects for Realism

1.1 Shadows

We need soft shadows in addition to hard shadows. Soft shadows arise when lights have areas (the realistic case, as no light is actually an infinitesimally small point), in spots where some but not all of an area light is blocked by an object. Then, instead of hard shadow edges we have a gradual shift from dark to light.



source: Wikipedia

There are three different characterizations within the shadow region:

- **Umbra:** the darkest part of a shadow. A region where you can't see any part of the light (all direct rays are blocked). With a point light, the entire shadow is essentially umbra.
- **Penumbra:** a "soft" part of the shadow. A region where part of the light is visible, and part of the light is blocked.
- **Antumbra:** another "soft" part of the shadow. A spot far enough behind the blocking object that you can see the light again. The intersection of penumbras.

1.2 Reflections

We want sufficient detail for the reflections of (e.g.) glossy or mirror surfaces.

1.3 Transparency

We should handle refraction, where light passes through a medium (water, glass) at a possibly distorted angle.

1.4 Interreflections

This is where one surface reflects colored light onto another (color bleeding).

2 Ray Casting

- With ray tracing, it's easy to compute shadows, transparency, etc.
- Recursive ray tracing was introduced in 1980 by Turner Whitted.
- Recently, ray tracing has become a real-time technique (e.g. NVIDIA's OptiX raytracer).

The algorithm:

Algorithm 1 Basic ray tracing

```
1: procedure RAYTRACE(camera, scene, w, h)
2:     image ← new image with width w and height h
3:     for y = 1 ... h do
4:         for x = 1 ... w do
5:             ray          ← RayThroughPixel(cam, y, x)
6:             intersection ← Intersect(ray, scene)
7:             image[y][x]  ← FindColor(intersection)
8:     return image
```

This is basic, *non-recursive* ray tracing, and gives the same images as those of HW2 rasterization.

We cast rays to determine visibility of objects in the scene on a *per-pixel* basis. For each pixel, we can shoot a ray through that pixel into the scene, see what object it hits (by testing intersection with all objects), and then shade as before.

Thus, in ray tracing, we traditionally need to consider all objects for all pixels, whereas in rasterization (where we loop over objects and then only deal with the pixels corresponding to each object), we only need to deal with the *covered* pixels for each object.

- Of course, the former case might be beneficial for a scene with many more objects than pixels.

3 Shadows and Reflections

Ray tracing facilitates the addition of many complex lighting and shading effects. For example, to add shadows, after a primary ray hits an object, we can then shoot a ray toward each light source and see whether it is blocked. If it is, the object is in shadow (no lighting contribution). If it isn't, the object is visible to the light source (lighting contribution).

Note: due to numerical inaccuracy, a shadow ray might intersect with the surface it was cast from, and thus shadow itself. To get around this, we should move a little toward a light before shooting a shadow ray.

For reflections, we can shoot a ray in the mirror direction after hitting an object. For refractions, we can shoot a ray in a direction according to Snell's law after hitting an object.

Algorithm 2 Recursive ray tracing

```
1: procedure RAYTRACE(camera, scene, w, h)
2:     image ← new image with width w and height h
3:     for y = 1 ... h do
4:         for x = 1 ... w do
5:             Trace primary ray from eye, compute nearest intersection
6:             Trace secondary shadow ray(s) to all lights(s), add color from illumination model if light visible
7:             Trace reflected rays, add color according to reflectivity ∗ color of reflected ray
8:             Trace refracted rays...
9:     return image
```

To determine the color of a reflected or refracted ray, we need to trace the ray again (recursion). To avoid rays being traced forever, we can enforce a maximum recursion depth.

- Note: capping the number of light bounces creates bias, because we're always throwing away the energy from later bounces. To get around this, we can use Russian roulette for unbiased termination.

4 Ray-Surface Intersection

4.1 Ray-Sphere Intersection

Ray:

$$\mathbf{p} = \mathbf{o} + \mathbf{d}t$$

for $\mathbf{p}$ a point on the ray, $\mathbf{o}$ an origin, $\mathbf{d}$ a direction, and $\mathbf{t}$ a distance parameter.

Sphere:

$$\|\mathbf{p} - \mathbf{c}\|_2^2 - r^2 = 0$$

for $\mathbf{p}$ a point on the sphere, $\mathbf{c}$ the center of the sphere, and r the radius of the sphere.

Ray-sphere intersection:

To determine the points of intersection, we substitute the ray equation into the sphere equation.

$$\begin{aligned} \|\mathbf{o} + \mathbf{d}t - \mathbf{c}\|_2^2 - r^2 &= 0 \\ (\mathbf{o} + \mathbf{d}t - \mathbf{c}) \cdot (\mathbf{o} + \mathbf{d}t - \mathbf{c}) - r^2 &= 0 \\ (\mathbf{d} \cdot \mathbf{d})t^2 + 2\mathbf{d} \cdot (\mathbf{o} - \mathbf{c})t + [(\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - r^2] &= 0 \end{aligned}$$

Solving this quadratic equation for t, we have that

- if we have two real positive solutions, we can just pick the intersection at the smaller t
- if we have one real positive solution, the ray is tangent to the sphere and the t can be used
- if we have one positive and one negative (real) solution, the ray originates inside the sphere and the intersection is at the positive solution for t
- if there is no real solution, there is no intersection

Before solving the equation, we can check the discriminant and make sure it's positive.

The surface normal at the point of intersection is $\mathbf{p} - \mathbf{c}$.

4.2 Ray-Triangle Intersection

We can intersect the ray with the triangle's plane, then check whether the intersection is in the triangle.

Triangle normal:

$$\mathbf{n} = \frac{(\mathbf{C} - \mathbf{A}) \times (\mathbf{B} - \mathbf{A})}{\|(\mathbf{C} - \mathbf{A}) \times (\mathbf{B} - \mathbf{A})\|}$$

for triangle **ABC**.

Plane:

$$(\mathbf{p} - \mathbf{A}) \cdot \mathbf{n} = 0$$

for $\mathbf{p}$ a point on the plane.

Ray-plane intersection:

$$\begin{aligned}
(\mathbf{o} + \mathbf{d}t - \mathbf{A}) \cdot \mathbf{n} &= 0 \\
(\mathbf{o} + \mathbf{d}t) \cdot \mathbf{n} &= \mathbf{A} \cdot \mathbf{n} \\
t &= \frac{\mathbf{A} \cdot \mathbf{n} - \mathbf{o} \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}
\end{aligned}$$

If $\mathbf{d}$ is orthogonal to $\mathbf{n}$, this quantity is not defined and we say there is no intersection.

Intersection inside polygon:

To identify whether the intersection point is inside a general polygon in the plane, we can draw a ray from the point in any direction within the plane, and count the number of times it intersects a polygon edge.

- If it intersects an even number of times, it's outside the polygon.
- If it intersects an odd number of times, it's inside the polygon.

Intersection inside triangle:

For triangles specifically, we can compute the barycentric coordinates of the point (weights of a point with respect to the triangle vertices) and determine whether all are in the range $[0, 1]$. If so, the point is in the triangle. If not, the point is not inside the triangle.

Barycentric coordinates α, β, γ:

$$\begin{aligned}
\mathbf{p} &= \alpha\mathbf{A} + \beta\mathbf{B} + \gamma\mathbf{C} \\
\text{s.t. } 0 \leq \alpha \leq 1,\ 0 \leq \beta \leq 1,\ 0 \leq \gamma \leq 1,\ \alpha + \beta + \gamma &= 1
\end{aligned}$$

Then

$$\begin{aligned}
\mathbf{p} - \mathbf{A} &= (\alpha - 1)\mathbf{A} + \beta\mathbf{B} + \gamma\mathbf{C} \\
\mathbf{p} - \mathbf{A} &= [\alpha - (\alpha + \beta + \gamma)]\mathbf{A} + \beta\mathbf{B} + \gamma\mathbf{C} \\
\mathbf{p} - \mathbf{A} &= -\beta\mathbf{A} - \gamma\mathbf{A} + \beta\mathbf{B} + \gamma\mathbf{C} \\
\mathbf{p} - \mathbf{A} &= \beta(\mathbf{B} - \mathbf{A}) + \gamma(\mathbf{C} - \mathbf{A})
\end{aligned}$$

which equates to three simultaneous equations we can use to solve for α, β, and γ.

- **Three** because $\mathbf{p}$, $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are 3D vectors.

4.3 Transformed Objects

Intersection with transformed objects: if an object has been transformed (e.g. sphere $\rightarrow$ ellipsoid), we can apply the inverse of the object transformation to the ray *and* the transformed object (or just use the untransformed object, if that is an option), then run the ray-object intersection routine as before. To get the actual intersection, we can then apply the object transformation on the intersection result (if there is one).

5 Acceleration Structures

It's expensive to test intersections with every object for every ray. In order to run fewer intersection tests, we can have **(hierarchical) bounding boxes** around groups of objects. If a ray doesn't intersect the bounding box, it doesn't intersect any of the objects within. We can organize the bounding boxes according to spatial hierarchies, and with spatial data structures such as octrees, k-d trees, and BSP trees.

- **Octree:** take 3D space, partition cuboids into eight octants each.

The other basic acceleration structures: **3D grids**.

- March ray through grid, test intersection against triangles in each cell, first we hit is first intersection.

6 Camera Ray Casting

How do we shoot a ray through a particular pixel on the screen – which direction should the ray have?

- We can transform objects into the camera frame and then shoot rays the same way every time, *or*
- we can simply determine the origin and direction of each ray in world coordinates:
  - *Ray origin*: camera center
  - *Ray direction*: a function of the extrinsic camera parameters and the shoot-through pixel (x, y)
    * first construct camera frame (in world coordinates) using up/viewing vectors
    * then camera looks down negative $\mathbf{w}$-axis ($\mathbf{w}$ being the viewing axis of the camera frame)
    * then the image plane is one unit away from the origin along the $\mathbf{w}$-axis (centered around it)
    * on the image plane, the ray will pass through a point some distance along the $\mathbf{u}$ and $\mathbf{v}$ axes
    * in normalized $[-1, 1]$ space, the amount we move along the $\mathbf{u}$-axis (x-direction) is $\frac{x - \texttt{width}/2}{\texttt{width}/2}$
    * $\tan\left(\frac{\texttt{fovx}}{2}\right)$ is the distance in world coordinates from $\texttt{width}/2$ to $\texttt{width}$ on the image plane
    * thus, in world coordinates, the amount we move along the $\mathbf{u}$-axis is $\tan\left(\frac{\texttt{fovx}}{2}\right)\frac{x - \texttt{width}/2}{\texttt{width}/2}$
    * likewise, the amount we move along the $\mathbf{v}$-axis is $\tan\left(\frac{\texttt{fovy}}{2}\right)\frac{\texttt{height}/2 - y}{\texttt{height}/2}$ (image y goes down)

$$\mathbf{d} = \texttt{normalize}\left\{\left[\tan\left(\frac{\texttt{fovx}}{2}\right)\frac{x - \texttt{width}/2}{\texttt{width}/2}\right]\mathbf{u} + \left[\tan\left(\frac{\texttt{fovy}}{2}\right)\frac{\texttt{height}/2 - y}{\texttt{height}/2}\right]\mathbf{v} - \mathbf{w}\right\}$$

7 Basic Add-Ons

7.1 Area Lights / Soft Shadows

We can break an area light into a grid of point lights (then shoot a ray to each point light in the grid). However, this can create a quantized, step-pattern look where points along a region have three lights visible, then two lights visible, then six lights visible...

- We should use jittering to fix this (randomize direction of shadow ray in direction of light source within small box). This is formally known as **stratified sampling**. It can also be used to remove shadow antialiasing, by shooting rays through a random location in each pixel region.

7.2 Path Tracing, etc.

Basic Whitted ray tracing can only handle mirror reflection and refraction, not full global illumination.