CS148 Summer 2025 - Lecture 6

07/10/2025

Raytracing I

cs148.stanford.edu

Looking Towards Ray Tracing

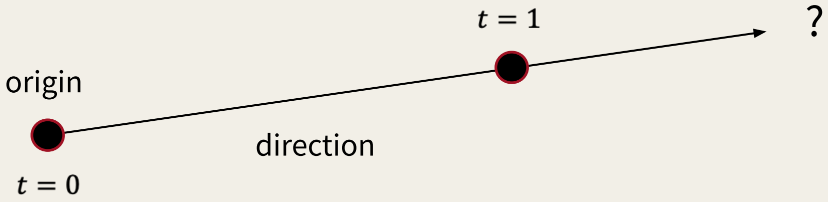
Rasterization (Scanline Rendering)

Ray Tracing

- 2-Part Lecture: ray-object intersections & shadows today
- reflections, transmissions, & other recursive concepts next class

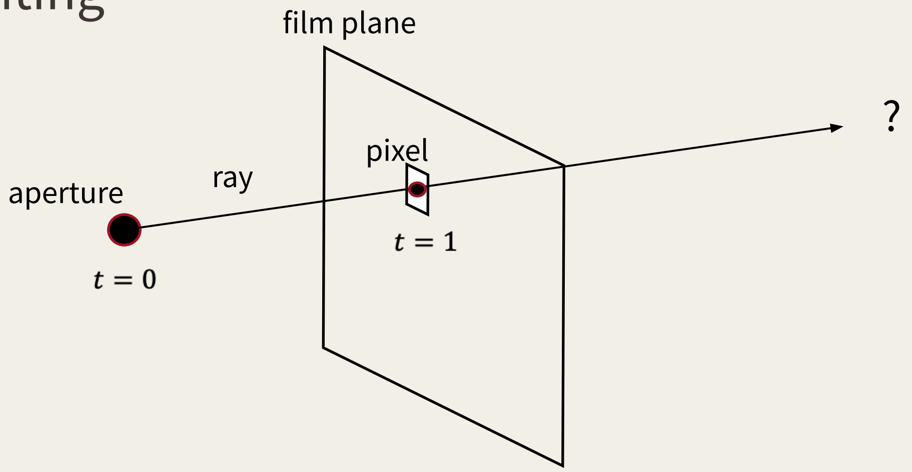
Constructing Rays

- Throughout these slides, we'll represent a "ray" as an equation for <u>derivation purposes</u>
- In practice, you will be handling rays in their simplest form when coding, in which case, you'll represent a ray as:
    - an origin point
    - a vector direction
    - a parameter t that tells us how far along the direction we are



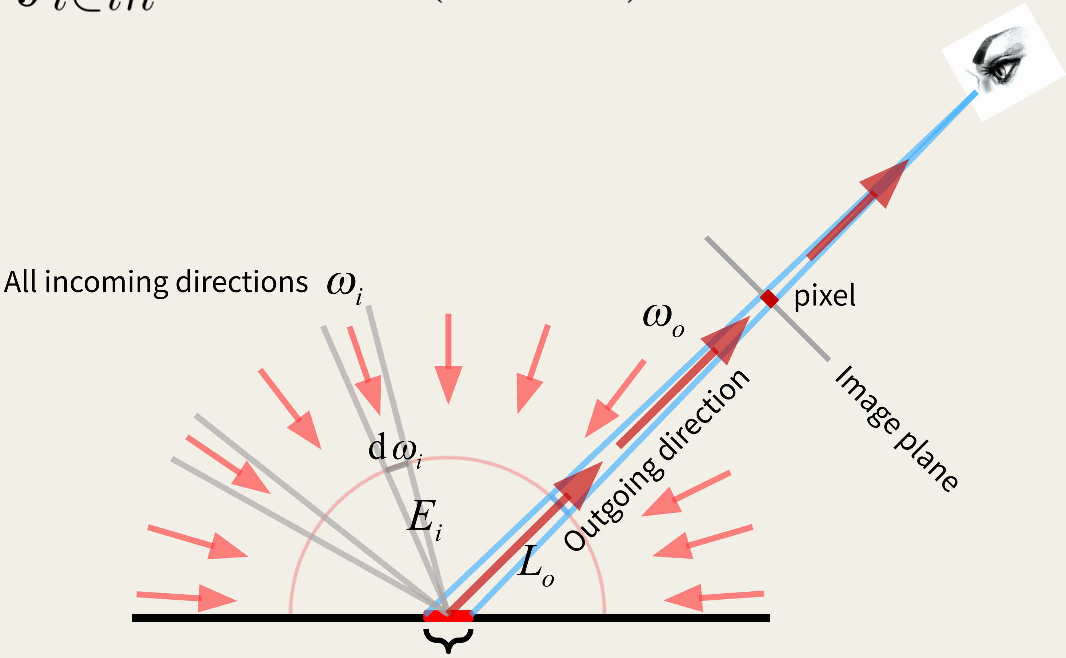
Constructing Rays

- For each pixel, shoot a ray $R(t) = A + (P - A)t$ where:
  - A is the the aperture (camera position), P is the pixel center
  - t is defined $t \in [0, \infty)$, technically $t \in [1, t_{far}]$ (inside frustum)
- Find the intersection with the smallest $t \in [1, t_{far}]$
- Then do lighting



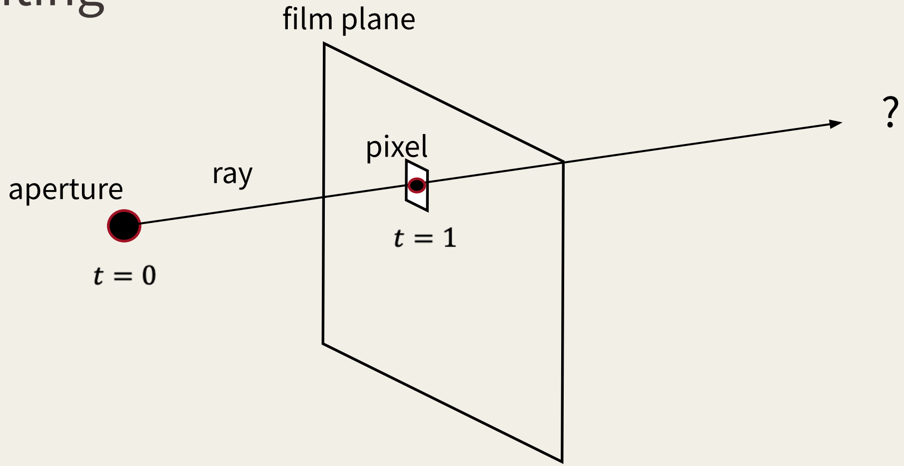
Recall: Lighting Equation

$$L_o(\omega_o) = \sum_{i \in in} L_o(\omega_i, \omega_o)$$
$$L_o(\omega_o) = \int_{i \in in} BRDF(\omega_i, \omega_o) L_i \cos\theta_i d\omega_i$$



Constructing Rays

- For each pixel, shoot a ray $R(t) = A + (P - A)t$ where:
  - A is the the aperture (camera position), P is the pixel center
  - t is defined $t \in [0, \infty)$, technically $t \in [1, t_{far}]$ (inside frustum)
- **Find the intersection** with the smallest $t \in [1, t_{far}]$
- Then do lighting



Ray-Triangle Intersection

- Recall: most of our objects will be triangle meshes
- So the question: how to we intersect our ray with a triangle?

- Observe: triangles are planar, i.e. 3 points are guaranteed in a plane
- One technique (2-step problem):
  - 1) Consider ray-plane intersection first for an intersection point
  - 2) Then, check if intersection point is inside the triangle
- Various ways to do 2)

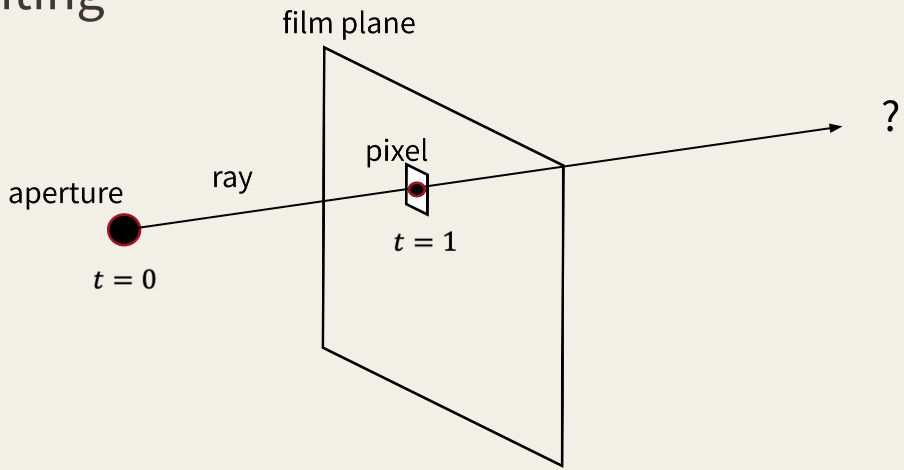
- Another approach: consider 3D ray-object intersection directly

Step 1: Ray-Plane Intersection

- From geometry, a plane is defined by:
  - p_o: a point on the plane (can use any triangle vertex)
  - N : a normal vector to the plane (can use triangle normal)
- A point p is on the plane if $(p - p_o) \cdot N = 0$

Recall "t"

- For each pixel, shoot a ray $R(t) = A + (P - A)t$ where:
  - A is the the aperture (camera position), P is the pixel center
  - t is defined $t \in [0, \infty)$, technically $t \in [1, t_{far}]$ (inside frustum)
- **Find the intersection** with the smallest $t \in [1, t_{far}]$.
- Then do lighting



Step 1: Ray-Plane Intersection

- From geometry, a plane is defined by:
  - p_o: a point on the plane (can use any triangle vertex)
  - N : a normal vector to the plane (can use triangle normal)
- A point p is on the plane if $(p - p_o) \cdot N = 0$

- Take our ray $R(t) = A + (P - A)t$ and solve for t:

$$(R(t) - p_o) \cdot N = 0$$

$$(A - p_o) \cdot N + (P - A) \cdot Nt = 0$$

$$t = \frac{(p_o - A) \cdot N}{(P - A) \cdot N}$$

Step 1: Ray-Plane Intersection

- Our ray $R(t) = A + (P - A)t$ intersects the plane $(p - p_o) \cdot N = 0$ when:

$$t = \frac{(p_o - A) \cdot N}{(P - A) \cdot N}$$

- Remember to restrict: $t \in [1, t_{far}]$

- Note that N is a vector; it does not cancel!
  - The lengths cancel though, so the normal doesn't have to be a unit vector
  - Useful if you're computing normals on the fly via e.g. cross products
- Once we have a $t \in [1, t_{far}]$, we plug it into $R(t)$ for our intersection

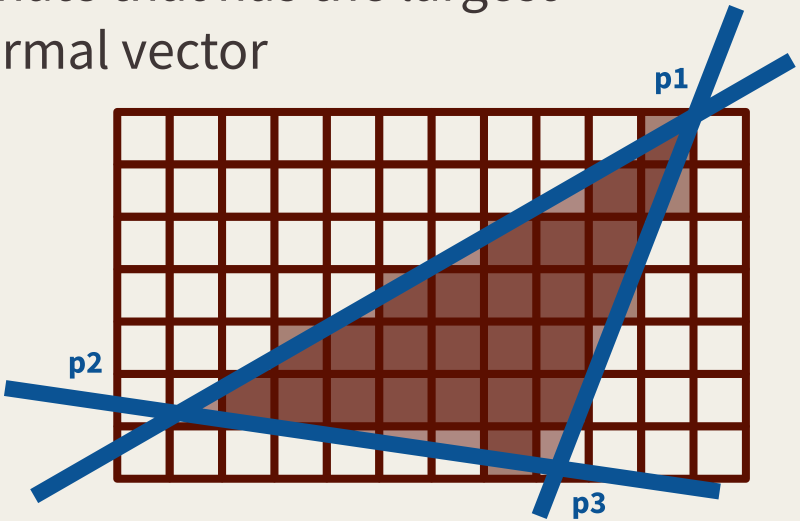
Step 2: Project Triangle & Intersection to 2D

- One technique (2-step problem):
  - 1) Consider ray-plane intersection first for an intersection point
  - 2) Then, check if intersection point is inside the triangle

- One approach to Step 2:
  - Once we have the ray-plane intersection, project both the intersection point and triangle into 2D
  - Example: project onto the xy-plane by dropping the z-coordinates of both the intersection point & triangle vertices

Step 2: Project Triangle & Intersection to 2D

- One approach to Step 2:
  - Once we have the ray-plane intersection, project both the intersection point and triangle into 2D
  - More robustly: drop the coordinate that has the largest component in the triangle's normal vector
  - Then use techniques from 2D rasterization to determine if the point is inside the triangle:



Alt Step 2: 3D Point Inside 3D Triangle

- One technique (2-step problem):
  - 1) Consider ray-plane intersection first for an intersection point
  - 2) <u>Then, check if intersection point is inside the triangle</u>

- Alternative approach to Step 2:
  - Don’t do the 2D projection
  - Instead: check if the intersection point is in the triangle using 3D geometry

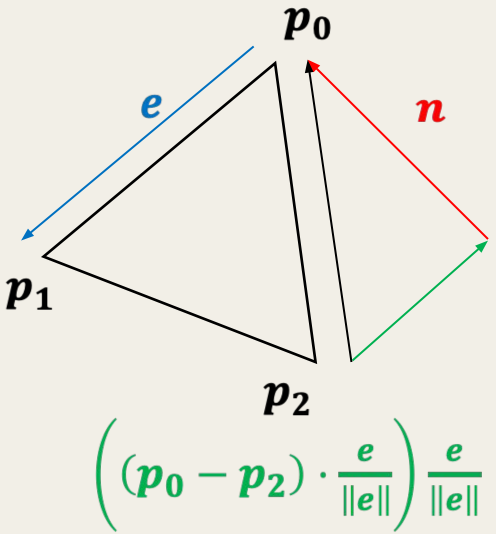
Alt Step 2: 3D Point Inside 3D Triangle

- Alternative approach to Step 2:
  - Don't do the 2D projection
  - Instead: check the intersection using 3D geometry
- Let R_o be our intersection point
- Take a directed edge on our triangle:

$$e = p_1 - p_o$$

compute a normal to edge as:

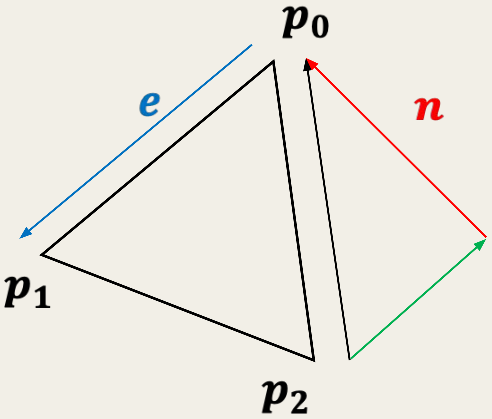
$$n = (p_o - p_2) - \left((p_o - p_2) \cdot \frac{e}{||e||}\right) \frac{e}{||e||}$$



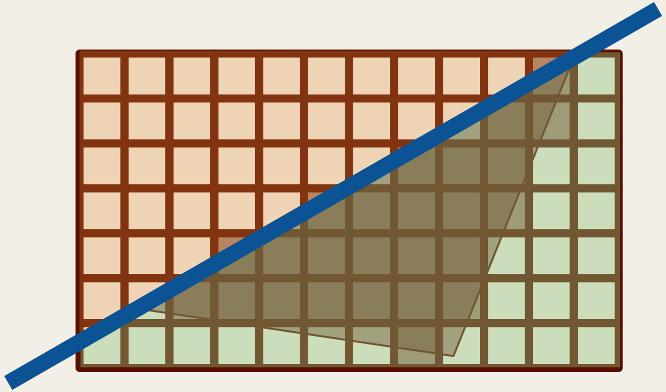
- R_o is interior to edge if $(R_o - p_o) \cdot n < 0$

Alt Step 2: 3D Point Inside 3D Triangle

- Recall for ray-plane intersection: $(R(t) - p_o) \cdot N = 0$
- R_o is interior to ray if $(R_o - p_o) \cdot n < 0$
- If interior to all 3 rays, then interior to the triangle



$$n = (p_o - p_2) - \left((p_o - p_2) \cdot \frac{e}{||e||}\right) \frac{e}{||e||}$$



Each edge of the triangle splits space into halves. Take the intersection of 3 half-planes!

Questions?

Ray-Triangle Intersection

- Recall: most of our objects will be triangle meshes
- So the question: how to we intersect our ray with a triangle?

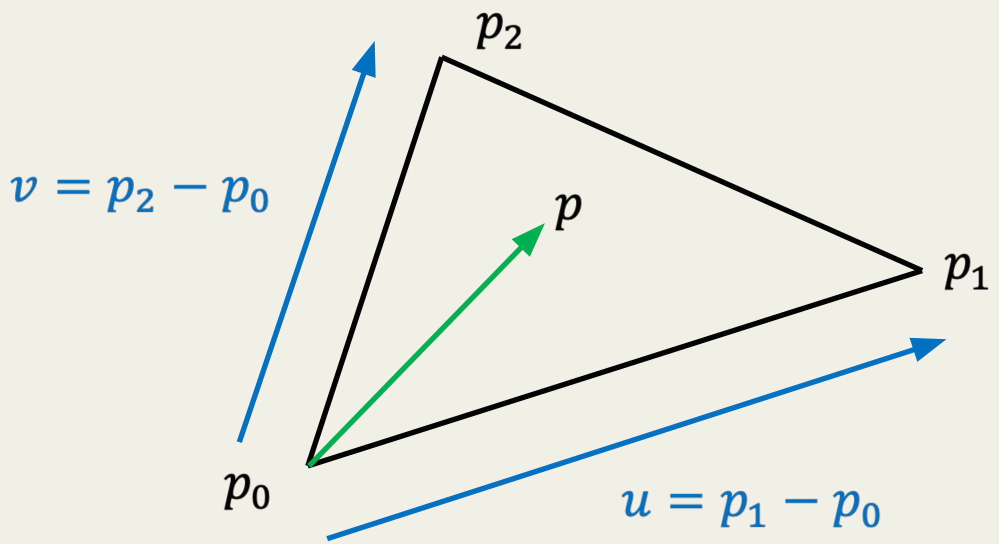
- ~~Observe: triangles are planar, i.e. they are contained in planes~~
- ~~One technique (2-step problem):~~
  - ~~1) Consider ray-plane intersection first for an intersection point~~
  - ~~2) Then, check if intersection point is inside the triangle~~
- ~~Various ways to do 2)~~

- Another approach: <u>consider 3D ray-object intersection directly</u>

Triangle Basis Vectors

- Any point inside the triangle can be written as a sum of one of the vertices plus scalings of the edge vectors:

$$p = p_o + \beta_1 u + \beta_2 v \text{ with: } \beta_1, \beta_2 \in [0, 1], \beta_1 + \beta_2 \leq 1$$



Direct Ray-Triangle Intersection

- A point inside a triangle is given by: $p = p_o + \beta_1 u + \beta_2 v$
- Substitute our ray: $R(t) = A + (P - A)t$

$$A + (P - A)t = p_o + \beta_1 u + \beta_2 v$$

$$(u, v, A - P) \begin{pmatrix} \beta_1 \\ \beta_2 \\ t \end{pmatrix} = A - p_o$$

- Solve matrix equation for: $\beta_1, \beta_2 \in [0, 1], \beta_1 + \beta_2 \leq 1$
- And: $t \in [1, t_{far}]$

Ray-Object Intersections

- Ray tracing generalizes well for non-triangular objects as long as we can have a good geometric representation for our objects
- In contrast to scanline rendering, which needs triangles to rasterize

- Can represent some geometry analytically, i.e. implicitly
- Implicit surfaces can be represented as functions:

  $$f(p) = 0$$

  for a point p on the surface
- Simplest example: a sphere

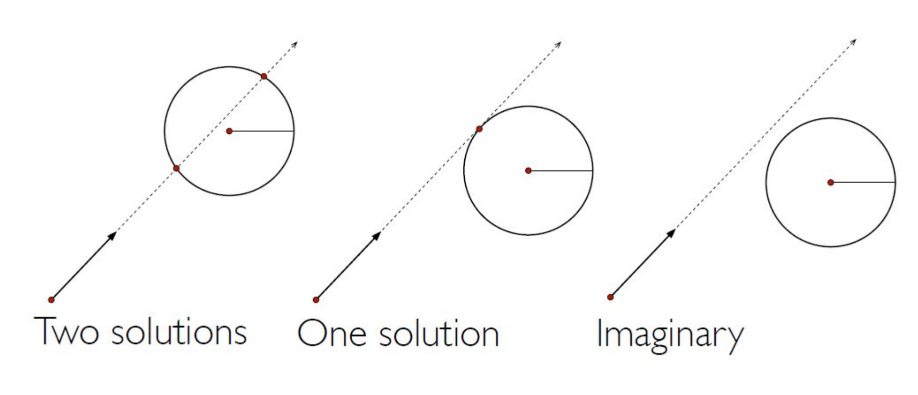
Ray-Sphere Intersections

- A point p on a sphere with center C and radius r satisfies:

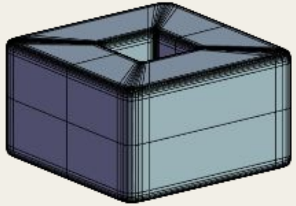
$$|p - C| = r \quad \rightarrow \quad (p - C) \cdot (p - C) = r^2$$

- Substitute our ray: $R(t) = A + (P - A)t$ for a quadratic equation:

$$(P - A) \cdot (P - A)t^2 + 2(P - A) \cdot (A - C)t + (A - C) \cdot (A - C) - r^2 = 0$$



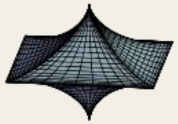
Ray-Superquadric Intersections

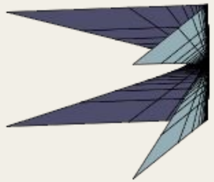


- Some more examples of implicit surfaces
- A superquadric centered at the origin is:

$$|x|^r + |y|^s + |z|^t = 1$$

- When r, s, t all equal 2, we have a sphere!
  - less than 1: pointy octahedron with concave faces
  - exactly 1: a regular octahedron
  - between 1-2: blunt octahedron with convex faces
  - greater than 2: a rounded cube
  - infinity: cube
  - And more, e.g. vary exponents for ellipsoid

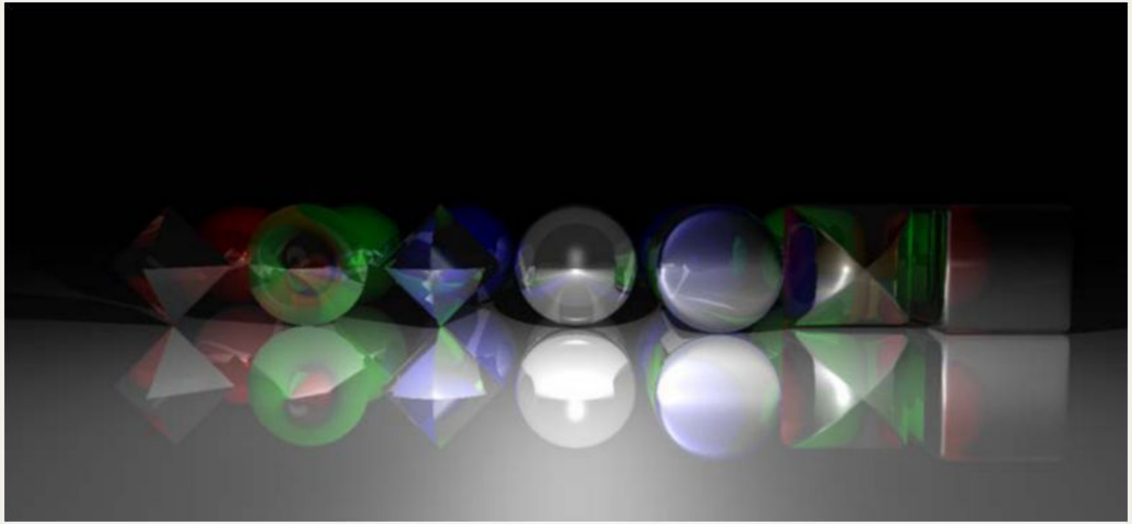




Ray-Superquadric Intersections

- Some more examples of implicit surfaces
- A superquadric centered at the origin is:

$$|x|^r + |y|^s + |z|^t = 1$$



Questions?

Parallelization

- Historically, ray tracing was too slow for real time rendering, hence optimization was spent on making scanline rendering real time

- Nowadays, we have parallel CPUs / clusters / GPUs to speed it up
- Threading (OpenMP), CUDA for GPU programming, etc
- Ray tracing is a per pixel operation, so inherently parallel

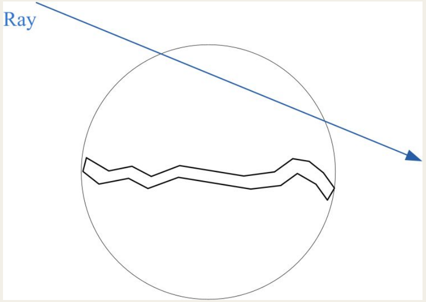
- Each ray is independent of any other ray
- Assign neighboring rays (nearby pixels) to the same core / processor
- Put object data in shared memory for each ray to access

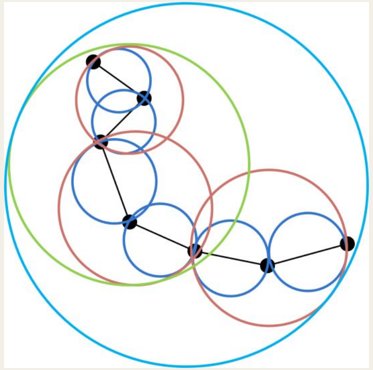
- Still relatively slow, but next gen consoles making progress

Code Acceleration in Software

- Ray tracing: for each pixel, shoot a ray to see if it intersects a triangle
- Basically requires a loop through every triangle for each pixel!

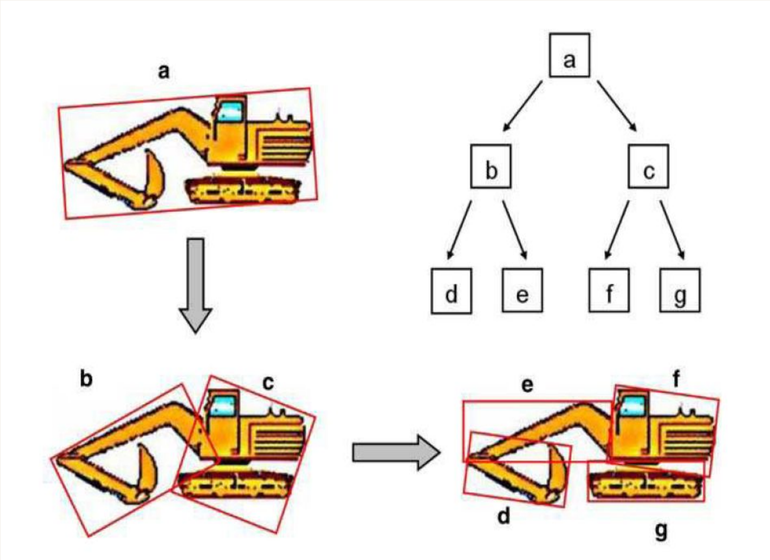
- Surround objects in bounding volumes, e.g. spheres
  - First, see if ray intersects the simpler bounding volumes
  - Then, worry about the triangles in your object

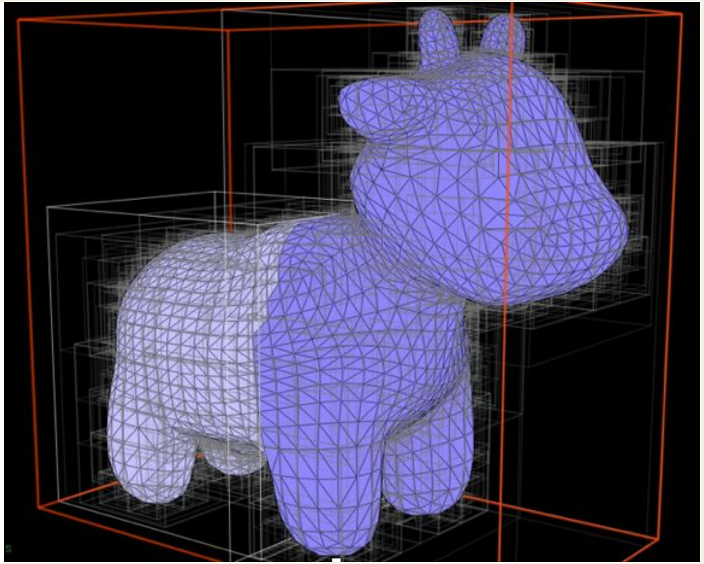




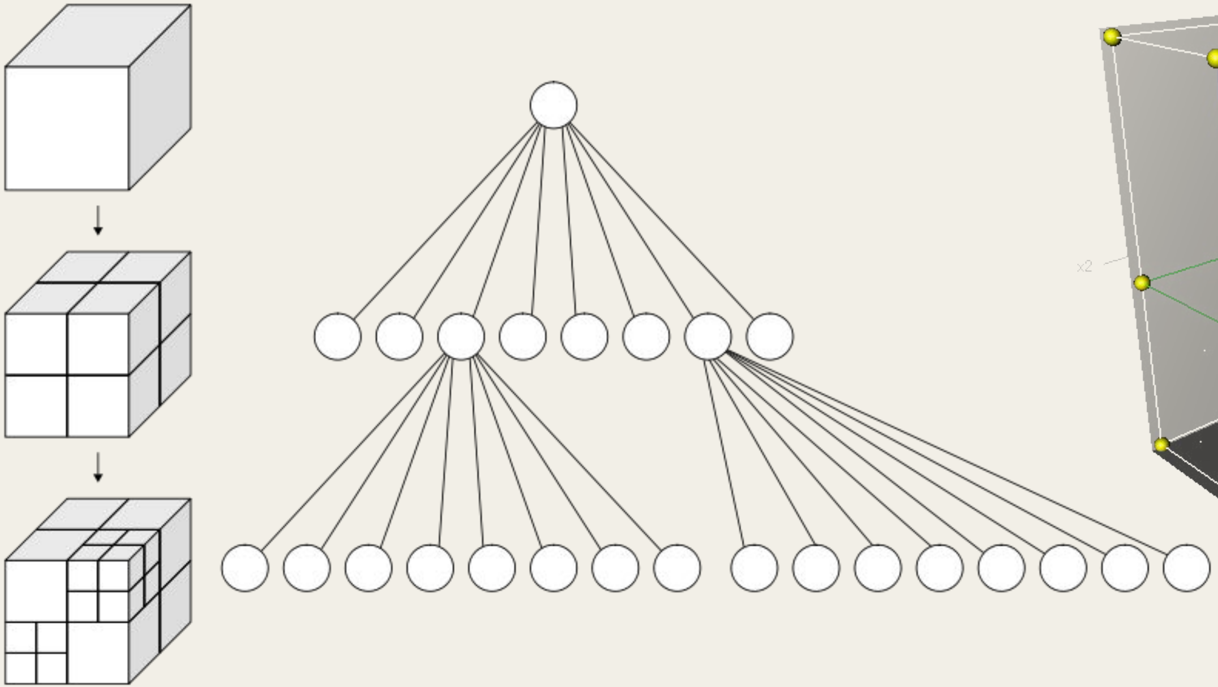
Bounding Volume Hierarchy (BVH)

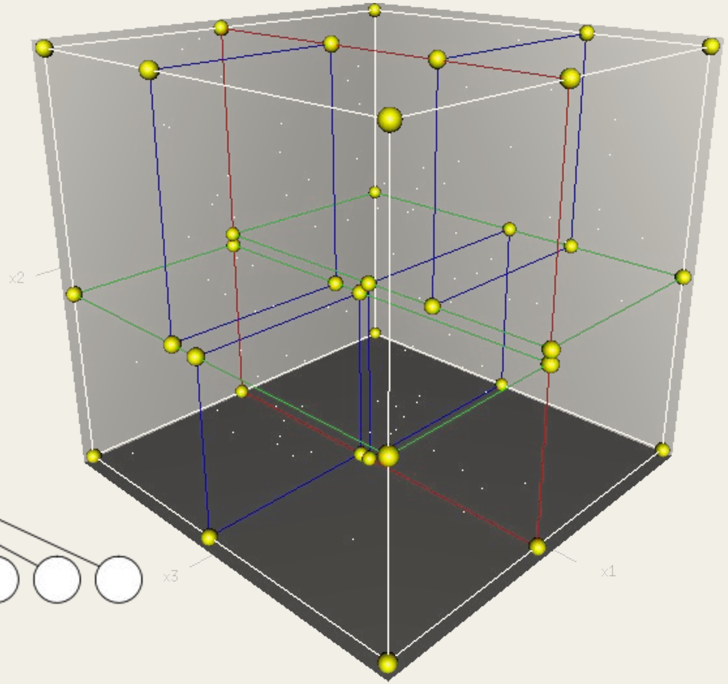
- Usually split bounding volumes into smaller bounding volumes, building a bounding volume hierarchy in object space
- O(n) triangle intersection operations sped up to O(log(n))





Bounding Volume Hierarchy (BVH)



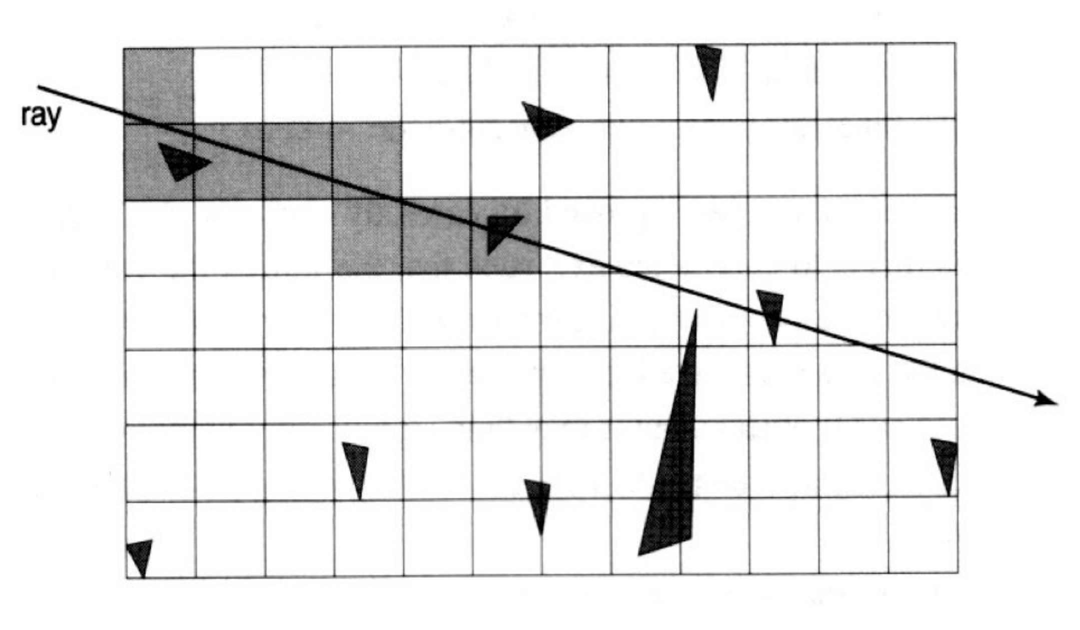


Octree: each volume split into 8 smaller volumes

K-D tree: each volume split into k smaller volumes

Uniform Partitions

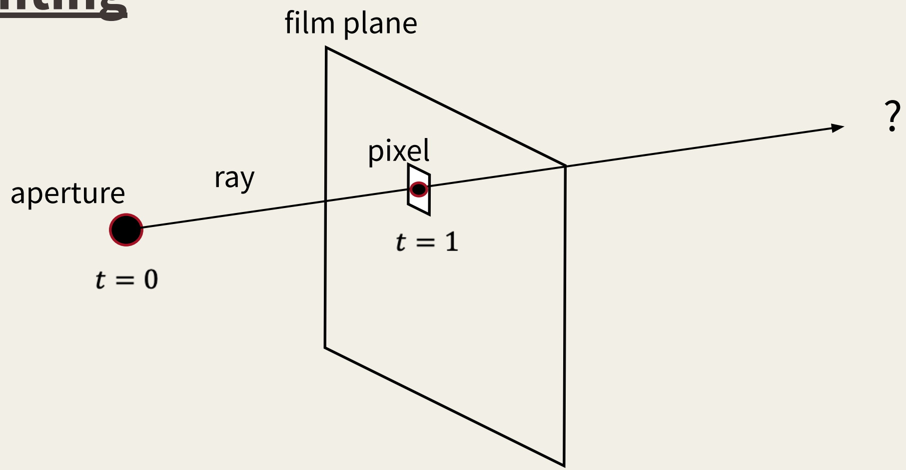
- BVH can still be expensive when there are many objects in the scene
- Can also use uniform special partitions (e.g. uniform grids):



Questions?

Constructing Rays

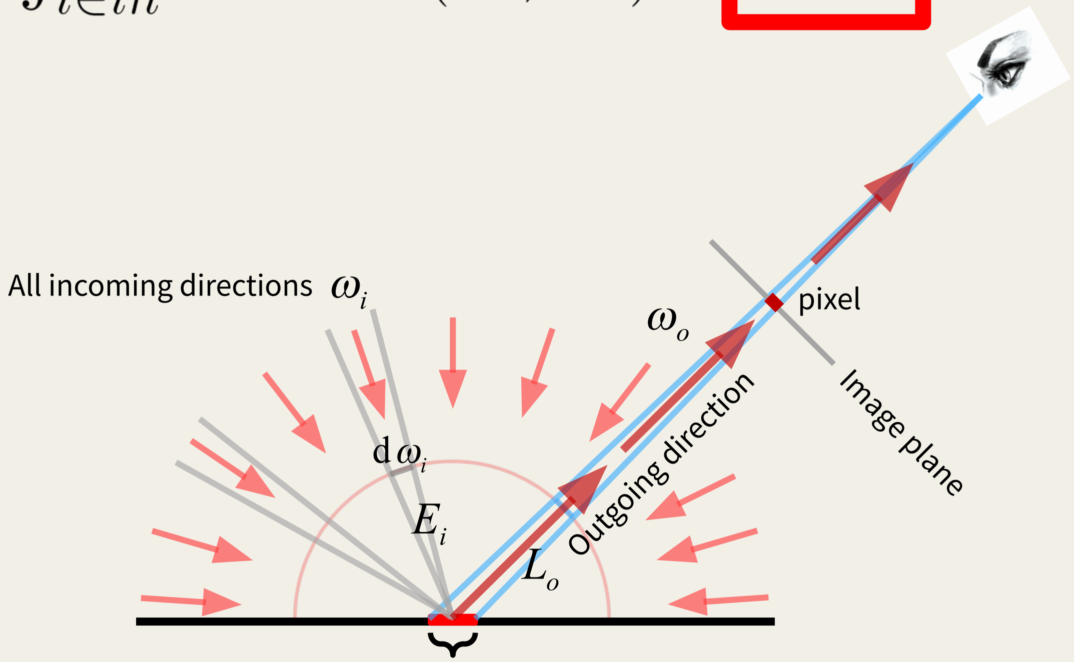
- For each pixel, shoot a ray $R(t) = A + (P - A)t$ where:
  - A is the the aperture (camera position), P is the pixel center
  - t is defined $t \in [0, \infty)$, technically $t \in [1, t_{far}]$ (inside frustum)
- ~~Find the intersection with the smallest $t \in [1, t_{far}]$~~
- **Then do lighting**



Recall: The Importance of the Normal

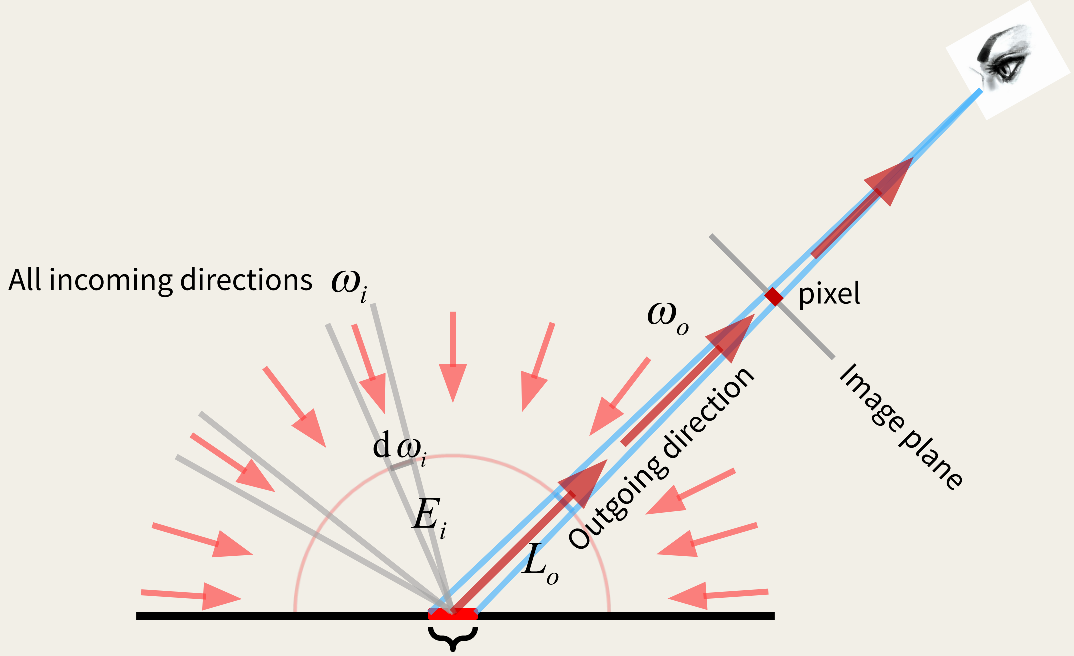
$$L_o(\omega_o) = \sum_{i \in in} L_o(\omega_i, \omega_o)$$

$$L_o(\omega_o) = \int_{i \in in} BRDF(\omega_i, \omega_o) L_i \cos\theta_i d\omega_i$$

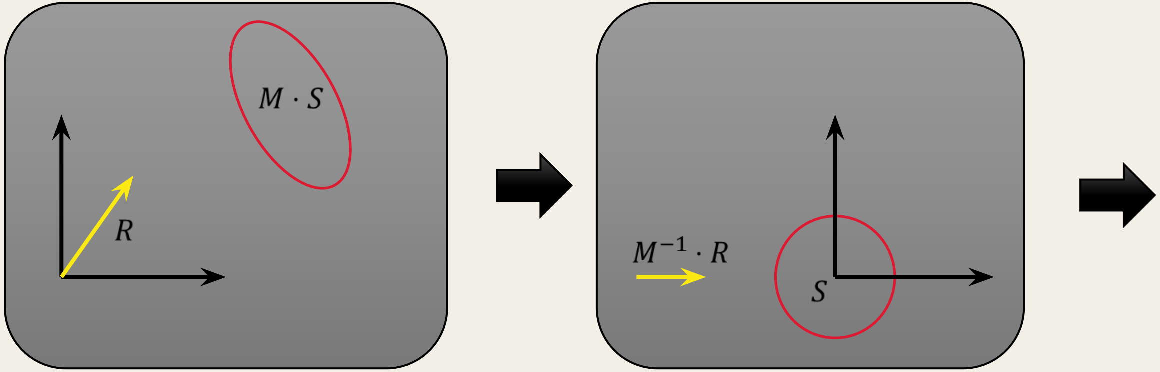


Normals in Object Space

- We typically define normals in object space
- But lighting computations are done in world space

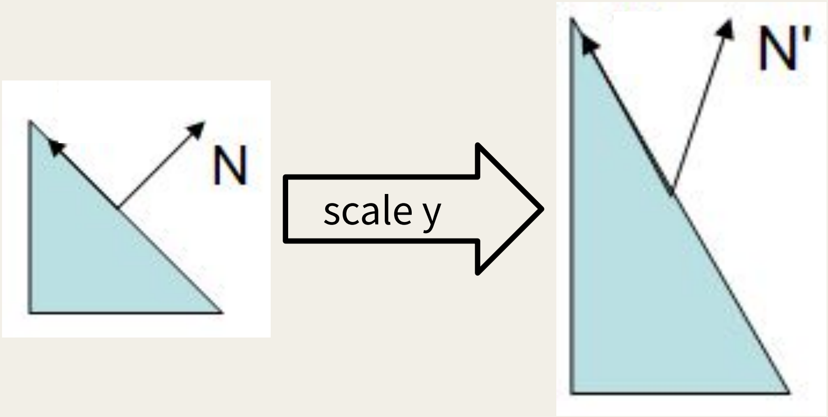


Transforming Normals

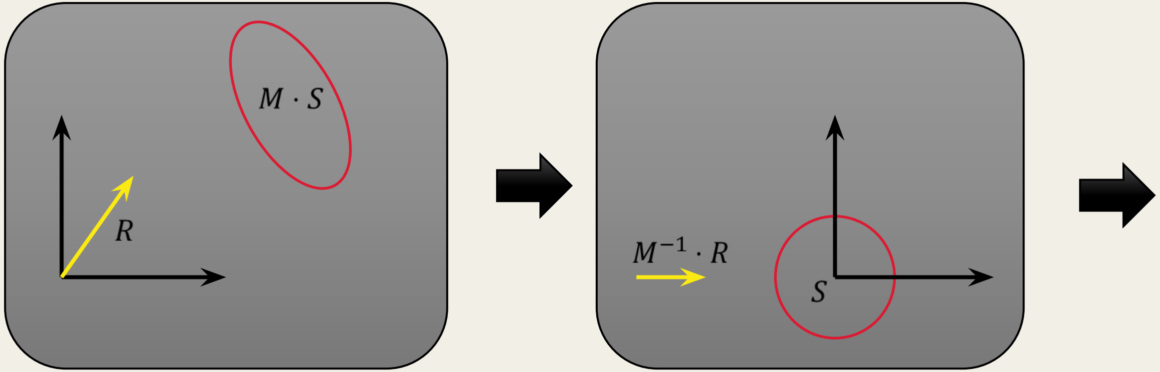


$$\boldsymbol{M \cdot R_{object}(t_{int})}$$

- NOT as simple as $M\hat{N}$!



Transforming Normals

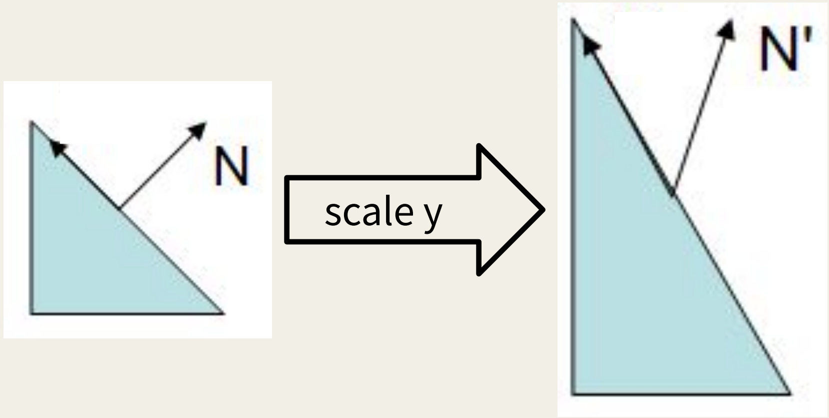


$$\boldsymbol{M} \cdot \boldsymbol{R}_{object}(\boldsymbol{t}_{int})$$

- NOT as simple as $M\hat{N}$!
- Let u be a triangle edge vector:

$$Mu \cdot M^{-T}\hat{N} = (Mu)^T M^{-T}\hat{N}$$
$$= u^T M^T M^{-T}\hat{N} = u^T\hat{N} = u \cdot \hat{N} = 0$$

- So actual transform is: $M^{-T}\hat{N}$

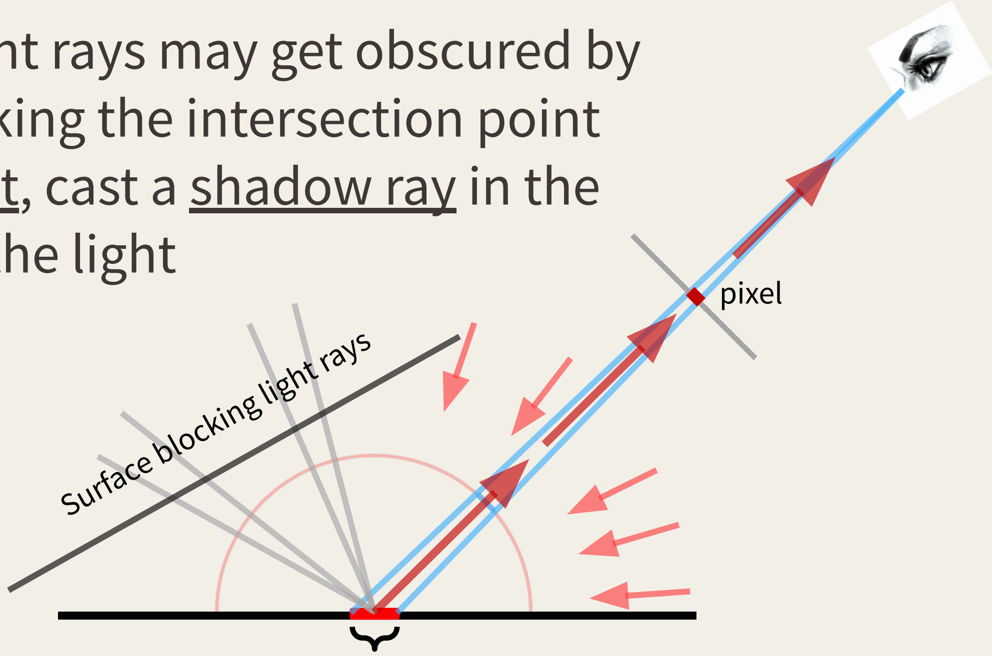


Questions?

Shadow Rays

$$L_o(\omega_o) = \int_{i \in in} BRDF(\omega_i, \omega_o) L_i \cos\theta_i d\omega_i$$

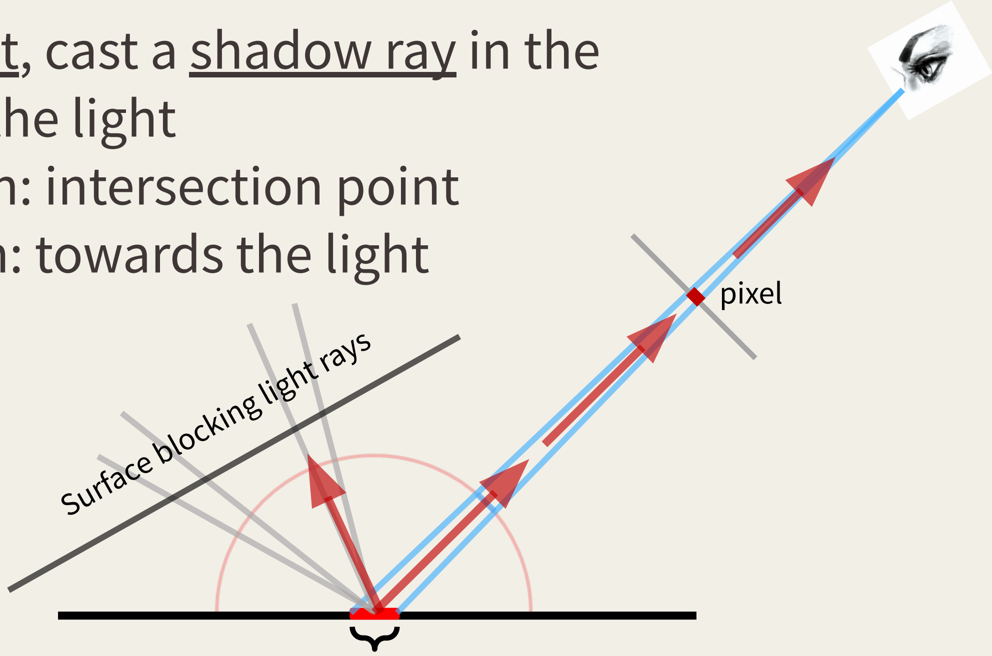
- Incoming light rays may get obscured by objects blocking the intersection point
- <u>For each light</u>, cast a <u>shadow ray</u> in the direction of the light



Shadow Rays

$$L_o(\omega_o) = \int_{i \in in} BRDF(\omega_i, \omega_o) L_i \cos\theta_i d\omega_i$$

- For each light, cast a shadow ray in the direction of the light
  - ray origin: intersection point
  - direction: towards the light

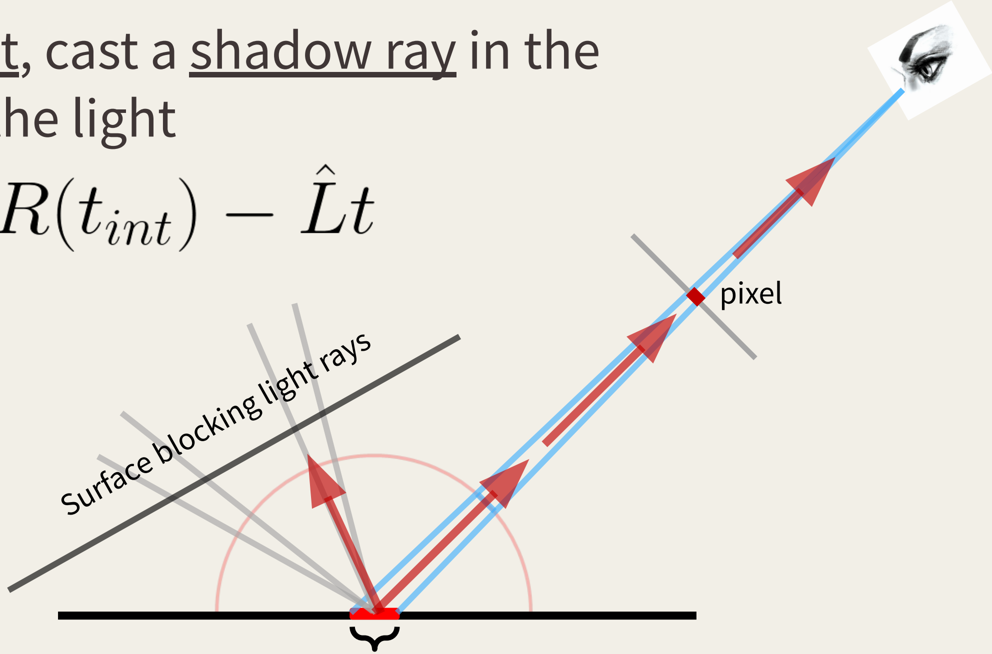


Shadow Rays

$$L_o(\omega_o) = \int_{i \in in} BRDF(\omega_i, \omega_o) L_i \cos\theta_i d\omega_i$$

- For each light, cast a shadow ray in the direction of the light

$$S(t) = R(t_{int}) - \hat{L}t$$



Shadow Rays

$$L_o(\omega_o) = \int_{i \in in} BRDF(\omega_i, \omega_o) L_i \cos\theta_i d\omega_i$$

- For each light, cast a shadow ray in the direction of the light:

$$S(t) = R(t_{int}) - \hat{L}t$$
$$t \in (0, t_{light})$$

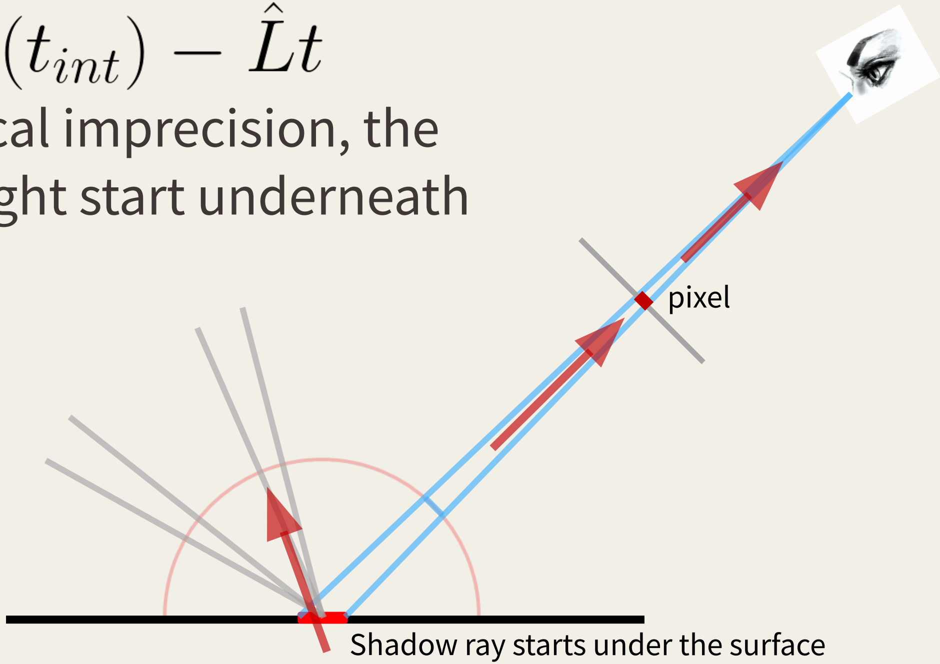
- Exact same ray tracing process as we've been discussing
- If no ray-object intersection for $0 < t < t_{light}$, then do usual lighting
- Else, an object is blocking the light to the intersection point, so there's 0 radiance coming from that blocked light

Caution: Spurious Self-Occlusion

- For each light, cast a shadow ray in the direction of the light

$$S(t) = R(t_{int}) - \hat{L}t$$

- Due to numerical imprecision, the shadow ray might start underneath the surface!

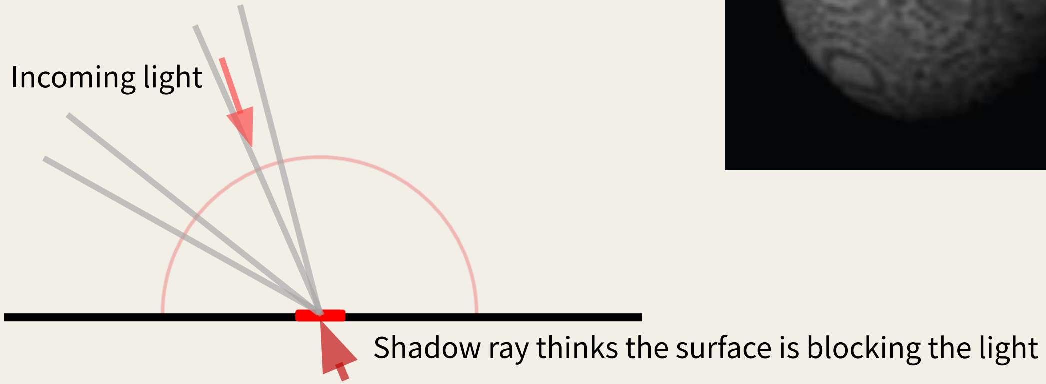


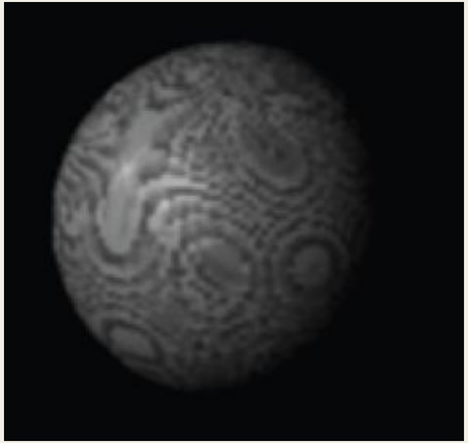
Caution: Spurious Self-Occlusion

- For each light, cast a shadow ray in the direction of the light

$$S(t) = R(t_{int}) - \hat{L}t$$

- Due to numerical imprecision, the shadow ray might start underneath the surface!



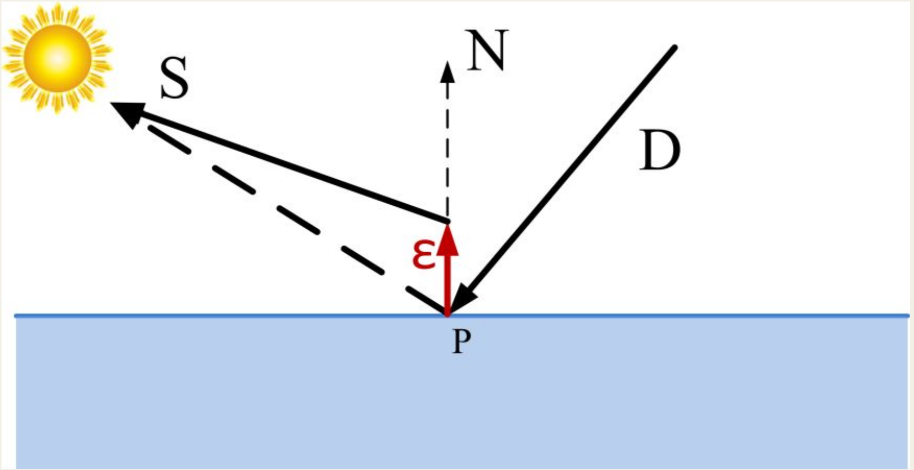


Caution: Spurious Self-Occlusion

- Solution: perturb by small epsilon in the normal direction

$$S(t) = (R(t_{int}) + \epsilon\hat{N}) - \hat{L}_{mod}t$$

- Light direction needs to modified slightly to start at $(R(t_{int}) + \epsilon\hat{N})$



Questions?