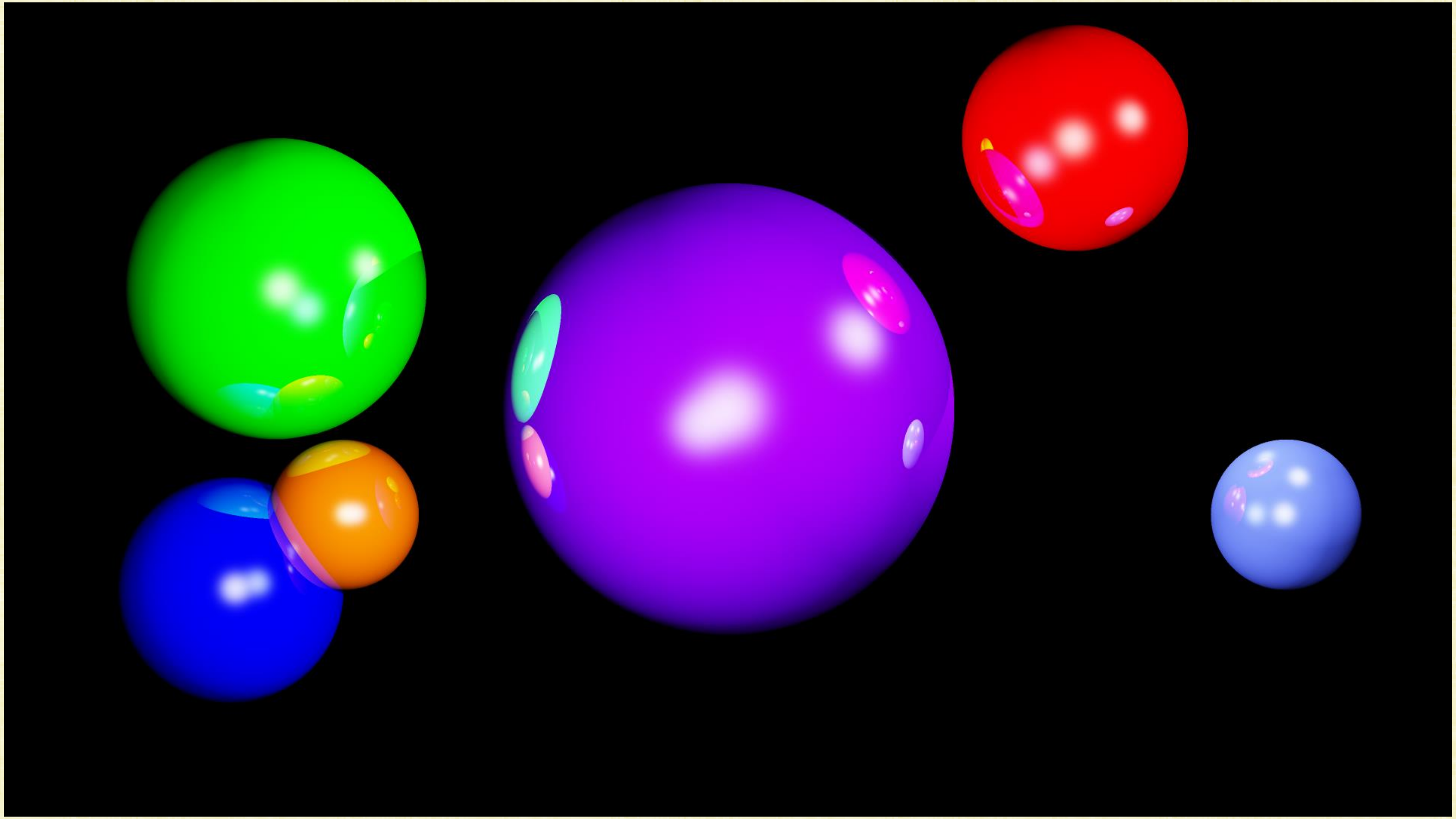
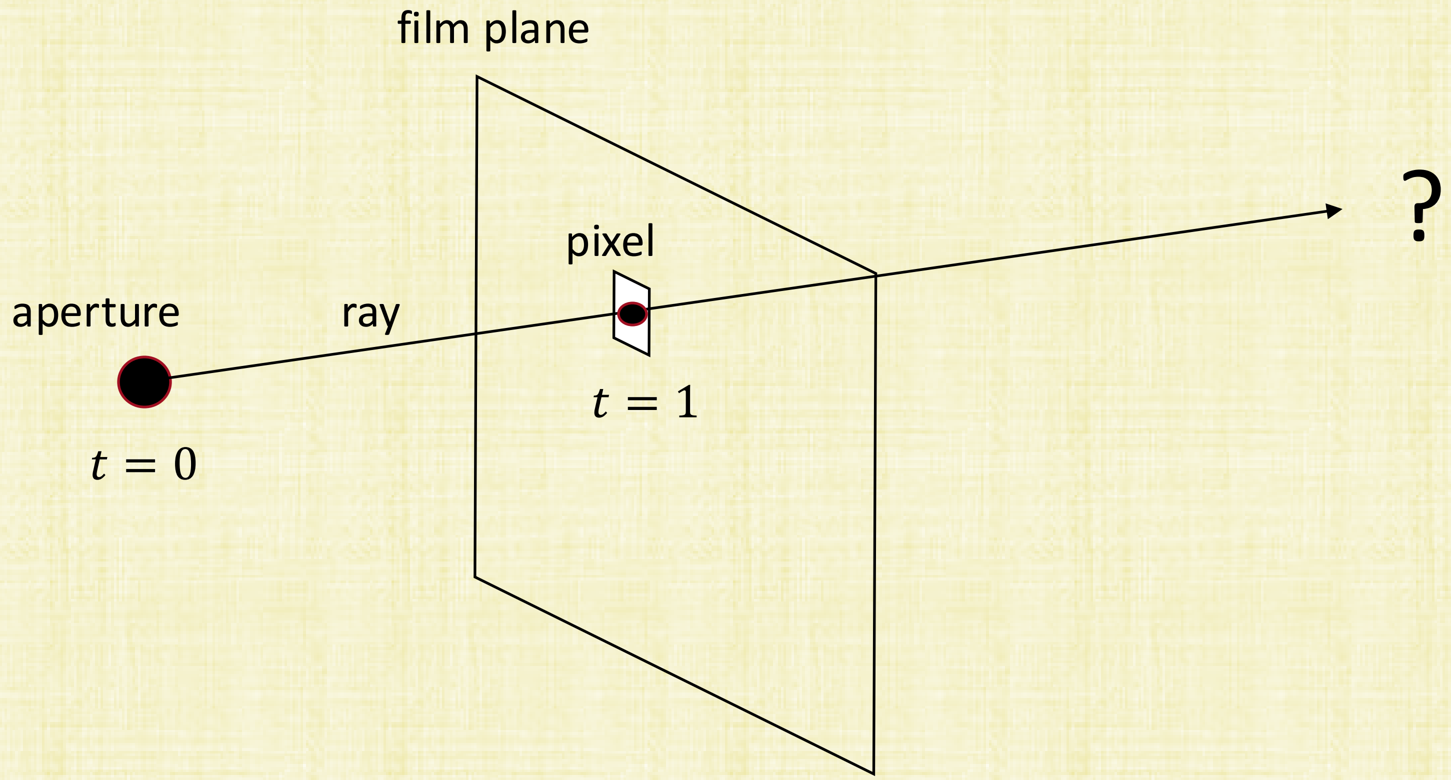
Ray Tracing



Constructing Rays

- For each pixel: create a ray and intersect it with objects in the scene
  - The first intersection is used to determine a color for the pixel
- The ray is $R(t) = A + (P - A)t$ where A is the aperture and P is the pixel location
- The ray is defined by $t \in [0, \infty)$, although only $t \in [1, t_{far}]$ is inside the viewing frustum
  - We only care about the first intersection with $t \geq 1$



Parallelization

- Ray tracing is a per pixel operation (scanline rendering is per triangle)
- Ray tracing is inherently parallel (the ray for each pixel is independent of the rays for other pixels)

- Can utilize parallel CPUs/Clusters/GPUs for code acceleration
  - Threading - distributes rays across CPU cores
  - Message Passing Interface (MPI) - distributes rays across CPUs on different machines (unshared memory)
  - OptiX/CUDA - distributes rays on the GPU
- Memory coherency is important, when distributing rays to various threads/processors
  - Assign spatially neighboring rays (passing through neighboring pixels) to the same core/processor
  - These rays tend to intersect with the same objects in the scene, and thus tend to access the same memory

- Note: Scanline rendering is parallelized to handle one triangle at a time (usually on a GPU)

Ray-Triangle Intersection

- Given the enormous number of triangles, many approaches have been implemented and tested in various software/hardware settings:

- Option 1: Triangles are contained in planes, so consider a Ray-Plane intersection first
  - A Ray-Plane intersection yields a point, and a subsequent test determines if that point lies inside the triangle
  - Option 1A: Project both the triangle and the intersection point into 2D; then, use the 2D triangle rasterization test (to the left of all 3 rays)
    - Can project into the xy, xz, yz plane by just dropping the z, y, x coordinate (respectively) from the triangle vertices and the intersection point
    - Most robust to drop the coordinate with the largest component in the triangle's normal (so that the projected triangle has maximal area)
  - Option 1B: There is a fully 3D version of the 2D rasterization

- Option 2: Skip the Ray-Plane intersection and consider Ray-Triangle intersection directly
  - This is similar to how ray tracing works for non-triangle geometry (ray tracers handle non-triangle geometry far better than scanline rendering does)

Ray-Plane Intersection

- A plane is defined by a point p_o (that lies on it) and a normal direction N
- A point p is on the plane if $(p - p_o) \cdot N = 0$
- A ray $R(t) = A + (P - A)t$ intersects the plane when:

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

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

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

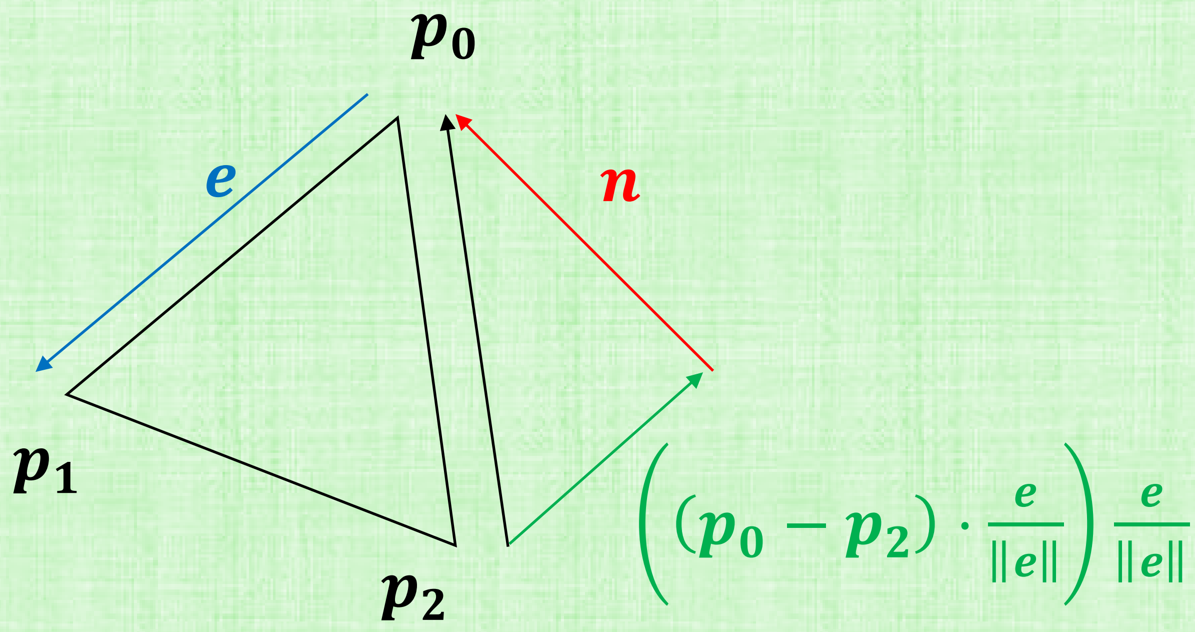
- As always, if $t \notin [1, t_{far}]$ or another intersection has a smaller t value, then this intersection is ignored

Notes:

- The length of N cancels, so it need not be unit length
- Compute N by taking the cross product of any two edges, as long as the triangle has nonzero area
- Any triangle vertex can be used as a point on the plane

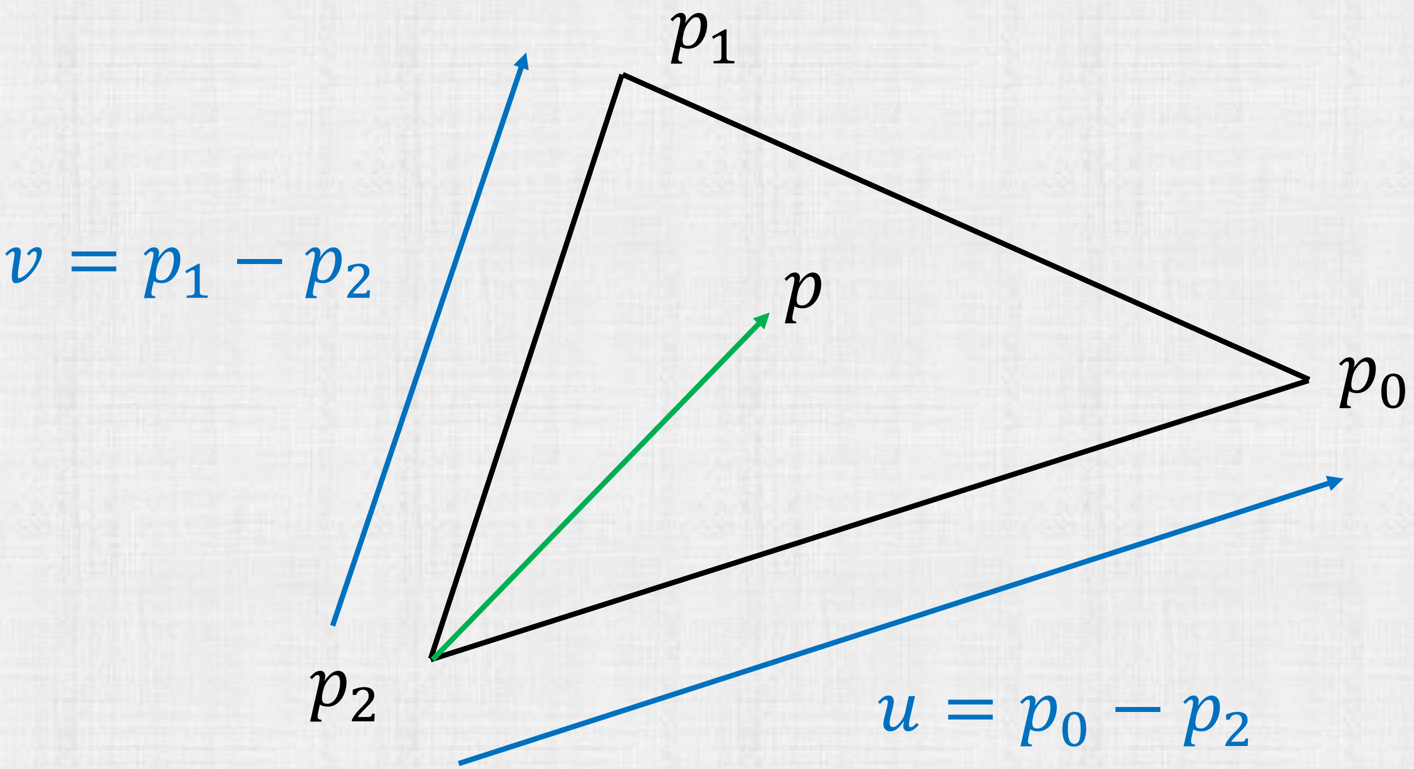
3D Point Inside a 3D Triangle (Option 1B)

- Given $t_{int} = \frac{(p_o - A)\cdot N}{(P - A)\cdot N}$, compute $R(t_{int}) = R_o$ as the intersection point
- Given edge $e = p_1 - p_0$, compute its normal $n = (p_0 - p_2) - \left((p_0 - p_2) \cdot \frac{e}{\|e\|}\right)\frac{e}{\|e\|}$
- R_o is to the left of to e when $(R_o - p_0) \cdot n < 0$
- If R_o is to the left of all three edges, it is interior to the triangle



Recall: Triangle Basis Vectors

- Compute edge vectors $u = p_0 - p_2$ and $v = p_1 - p_2$
- Points in the triangle have the form $p = p_2 + \beta_0 u + \beta_1 v$ with $\beta_0, \beta_1 \in [0,1]$ and $\beta_0 + \beta_1 \leq 1$
- Substitutions give $p = \beta_0 p_0 + \beta_1 p_1 + (1 - \beta_0 - \beta_1) p_2$ implying that: $\alpha_0 = \beta_0,\ \alpha_1 = \beta_1,$ $\alpha_2 = 1 - \beta_0 - \beta_1 = 1 - \alpha_0 - \alpha_1$



Direct Ray-Triangle Intersection (Option 2)

- Triangle Basis Vectors: $p = p_2 + \beta_0 u + \beta_1 v$ with $\beta_0, \beta_1 \in [0,1]$ and $\beta_0 + \beta_1 \leq 1$
- Ray: $R(t) = A + (P - A)t$
- An intersection point has:

$$A + (P - A)t = p_2 + \beta_0 u + \beta_1 v$$

$$\begin{pmatrix} u & v & A - P \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \\ t \end{pmatrix} = A - p_2$$

3x3 matrix 3x1 vector

- 3 equations with 3 unknowns:
  - The 3x3 coefficient is degenerate when its columns are not full rank
  - That happens when the triangle has zero area or the ray direction, $P - A$, is perpendicular to the plane's normal
  - Otherwise, there is a unique solution

- $R(t_{int})$ is inside the triangle, when: $\beta_0, \beta_1 \in [0,1]$ and $\beta_0 + \beta_1 \leq 1$
- As always, if $t \notin [1, t_{far}]$ or another intersection has a smaller t value, then this intersection is ignored

Solving via Cramer's Rule

- Solving the 3x3 linear system via Cramer's Rule allows for code optimization:

- Compute the determinant of the 3x3 coefficient matrix $\Delta = |(u \quad v \quad A - P)|$, which is nonzero when a solution exists
- Compute $t = \frac{\Delta_t}{\Delta}$ where the numerator is the determinant $\Delta_t = |(u \quad v \quad A - p_0)|$
- When $t \notin [1, t_{far}]$ or there is an earlier intersection, quit early (ignoring this intersection)

- Compute $\beta_0 = \frac{\Delta_{\beta_0}}{\Delta}$ where $\Delta_{\beta_0} = |(A - p_0 \quad v \quad A - P)|$
- When $\beta_0 \notin [0,1]$, quit early

- Compute $\beta_1 = \frac{\Delta_{\beta_1}}{\Delta}$ where $\Delta_{\beta_1} = |(u \quad A - p_0 \quad A - P)|$
- When $\beta_1 \in [0,1 - \beta_0]$, the intersection is marked as true

Ray-Object Intersections

- As long as a Ray-Geometry intersection routine can be written, ray tracing can be applied to any representation of object geometry
  - In contrast to scanline rendering, where objects need to be turned into triangles
- Besides triangles, ray tracers readily use: analytic descriptions of geometry, implicitly defined surfaces, parametric surfaces, etc.

Implicitly defined geometry:

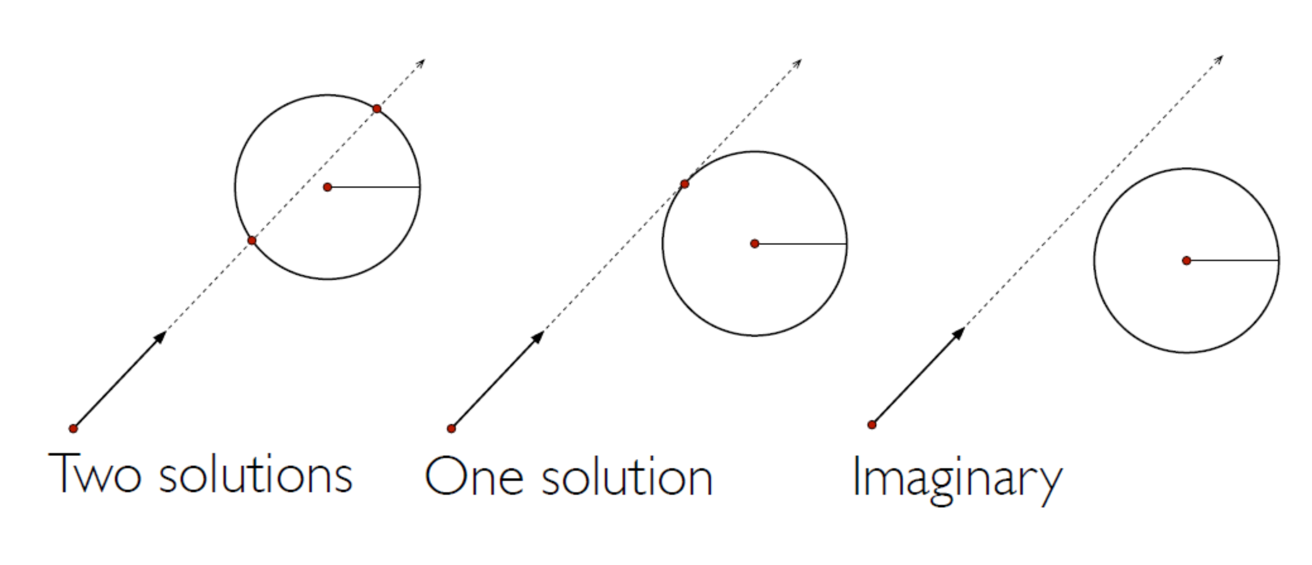
- Define an implicit function f where $f(p) = 0$ defines a surface (e.g. the equation for a plane)
- Sometimes there are additional constraints (such as on the barycentric weights for triangles)
- Ray-Object intersection routines then proceed as follows:
  - substitute the ray equation in for the point: $f(R(t)) = 0$
  - solve for t
  - check the solution against any additional constraints

Ray-Sphere Intersections

- A sphere with center C and radius r can be defined via $\|p - C\|_2 = r$
  - Square both sides: $(p - C) \cdot (p - C) = r^2$
- Substitute $R(t) = A + (P - A)t$ in for p to get a quadratic equation in t:

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

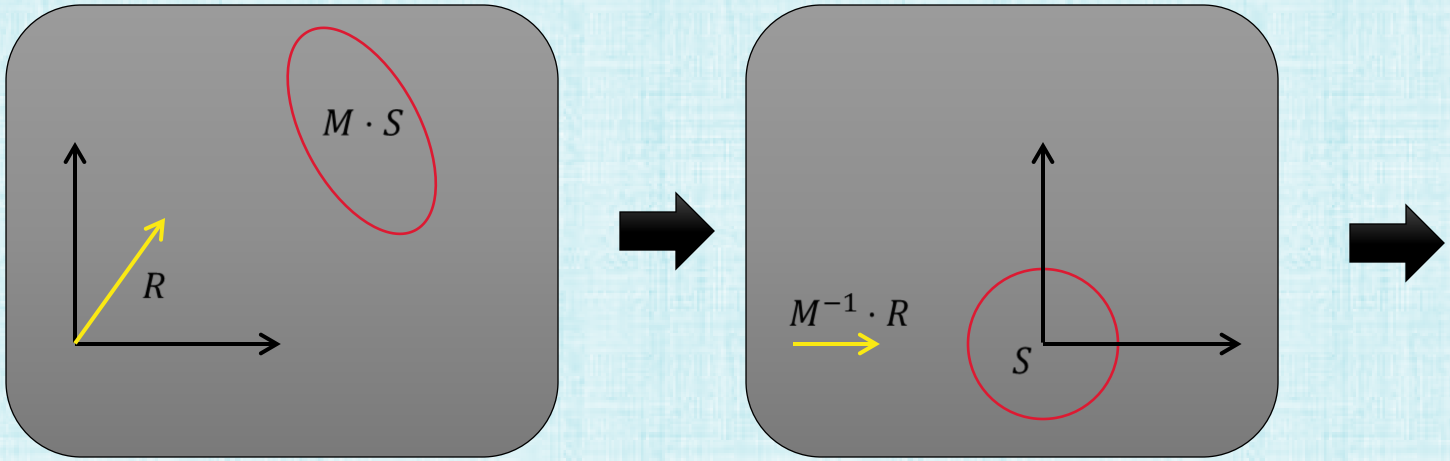
- When the discriminant is positive, there are two solutions (choose the one the ray hits first)
- When the discriminant is zero, there is one solution (the ray tangentially grazes the sphere)
- When the discriminant is negative, there are no solutions



Ray Tracing Transformed Objects

- It is typically preferable to ray trace in object space, rather than world space
  - Geometry is typically kept in object space
  - The object space representation is typically <u>simpler</u> to deal with
  - E.g., spheres can be centered at the origin, objects are not sheared, coordinates may be non-dimensionalized for numerical robustness, there may be (auxiliary) geometric acceleration structures, more convenient color and texture information, etc.

- <u>Transform the ray into object space</u> and find the ray-object intersection
- Then, transform the relevant information back to world space



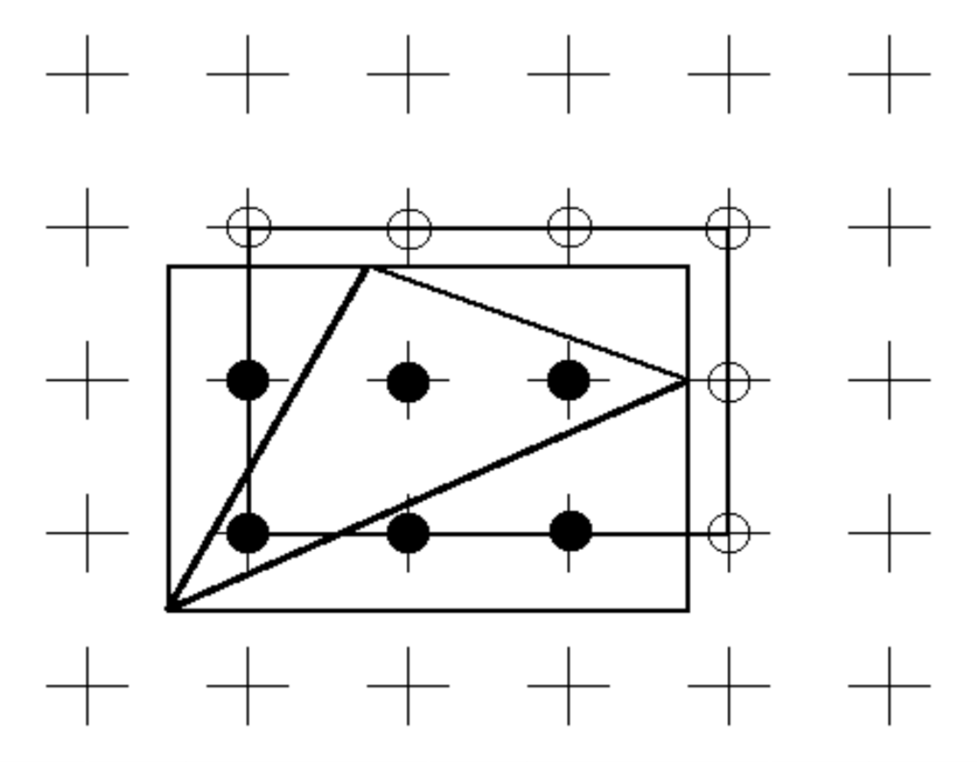
$$\boldsymbol{M} \cdot \boldsymbol{R}_{\boldsymbol{object}}(\boldsymbol{t}_{\boldsymbol{int}})$$

Recall: Bounding Box Acceleration

- Checking every pixel against every triangle is computationally expensive
- Calculate a bounding box around the triangle, with diagonal corners:

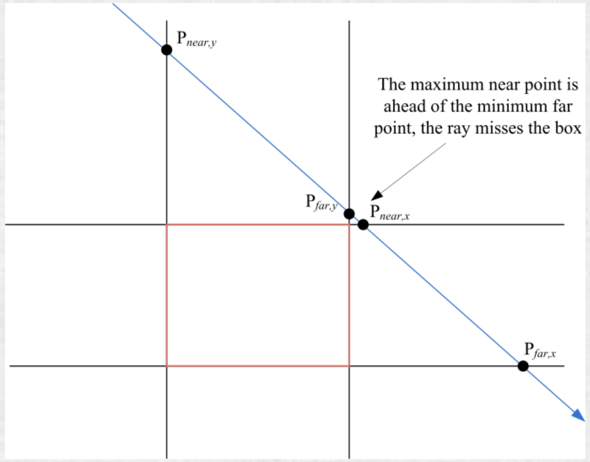
$$(\min(x_o, x_1, x_2), \min(y_0, y_1, y_2)) \text{ and } (\max(x_o, x_1, x_2), \max(y_0, y_1, y_2))$$

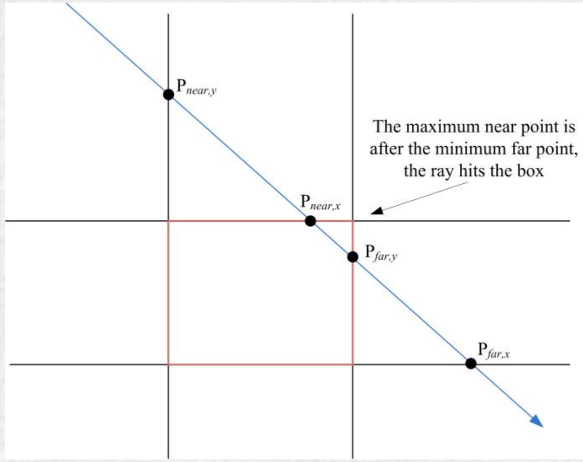
- Then, round coordinates upward to the nearest integer to find all relative pixels

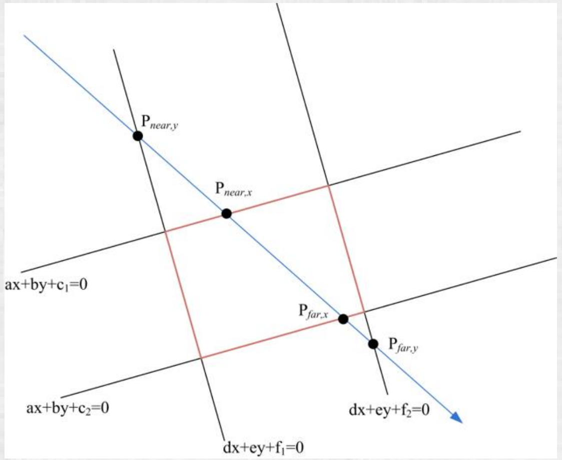


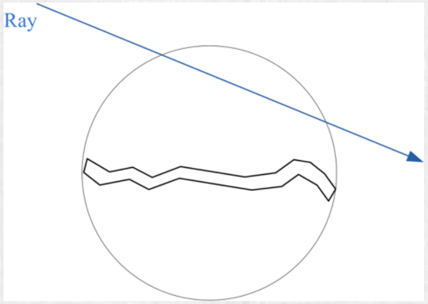
Aside: Code Acceleration (Bounding Volumes)

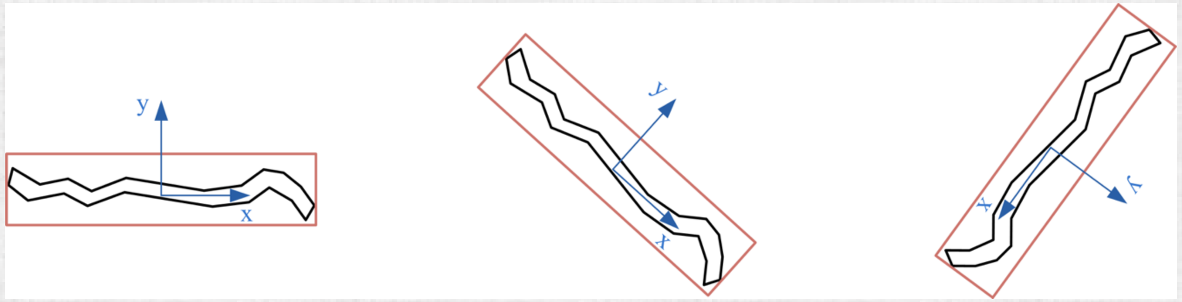
- Ray-Object intersections can be expensive
- So, put complex objects inside simpler objects, and first test for intersections against the simpler object (potentially skipping tests against the complex object)
- Simple <u>bounding volumes</u>: spheres, axis-aligned bounding boxes (AABB), or oriented bounding boxes (OBB)





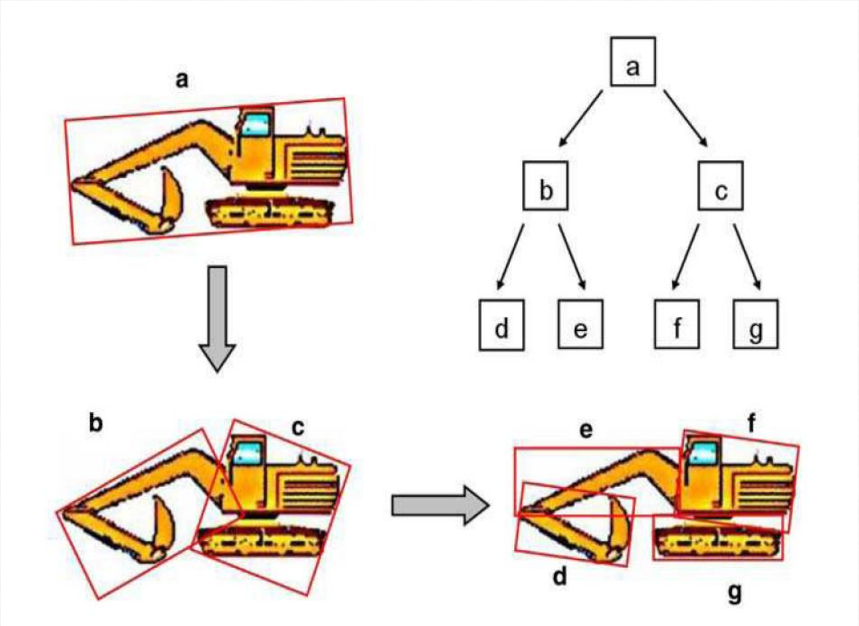


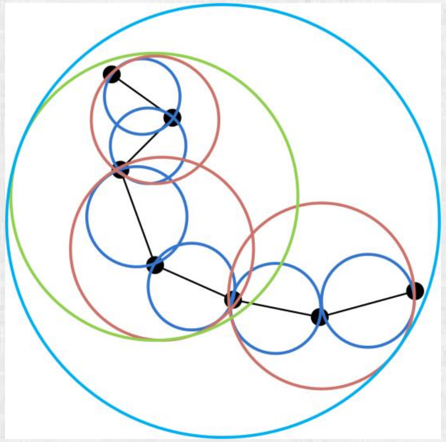


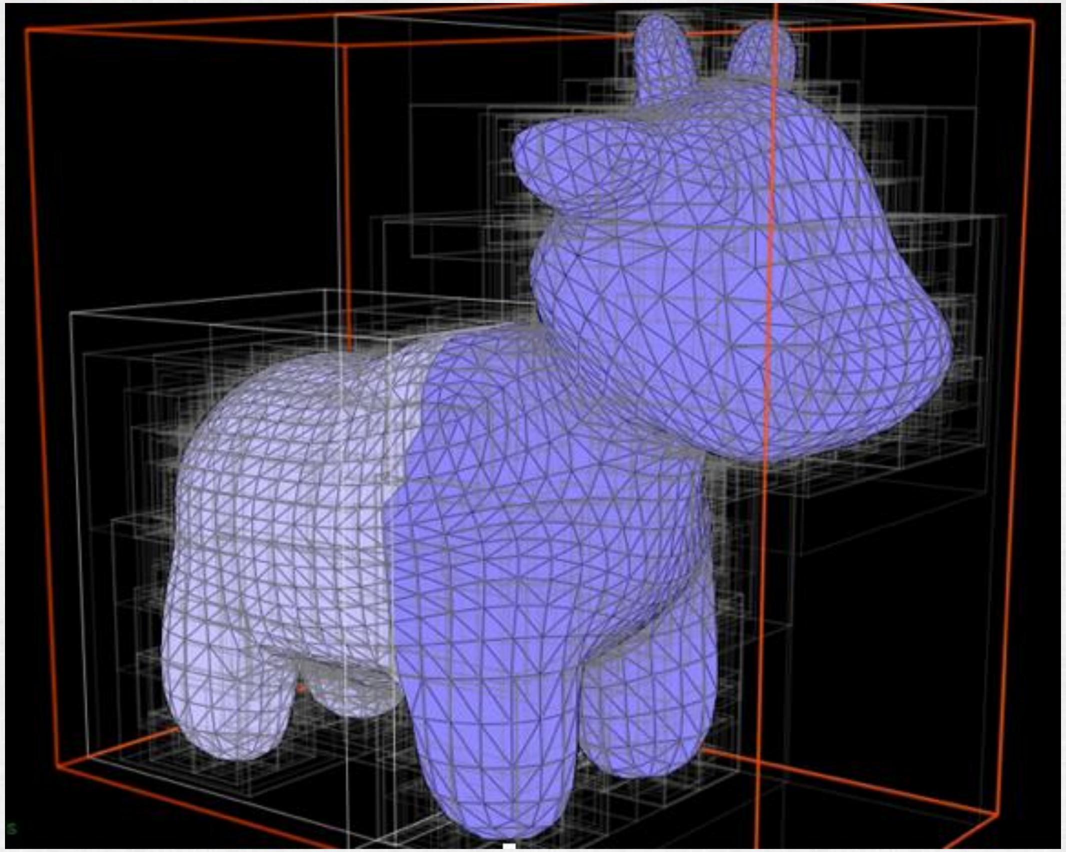


Aside: Code Acceleration (Hierarchical Bounding Volumes)

- For complex objects, build a <u>hierarchical tree structure</u> in object space
- The lowest levels of the tree contain the primitives used for intersections (and have simple geometry bounding them); then, these are combined hierarchically into a $\log n$ height tree
- Starting at the top of a Bounding Volume Hierarchy (BVH), one can prune out many nonessential (missed) ray-object collision checks

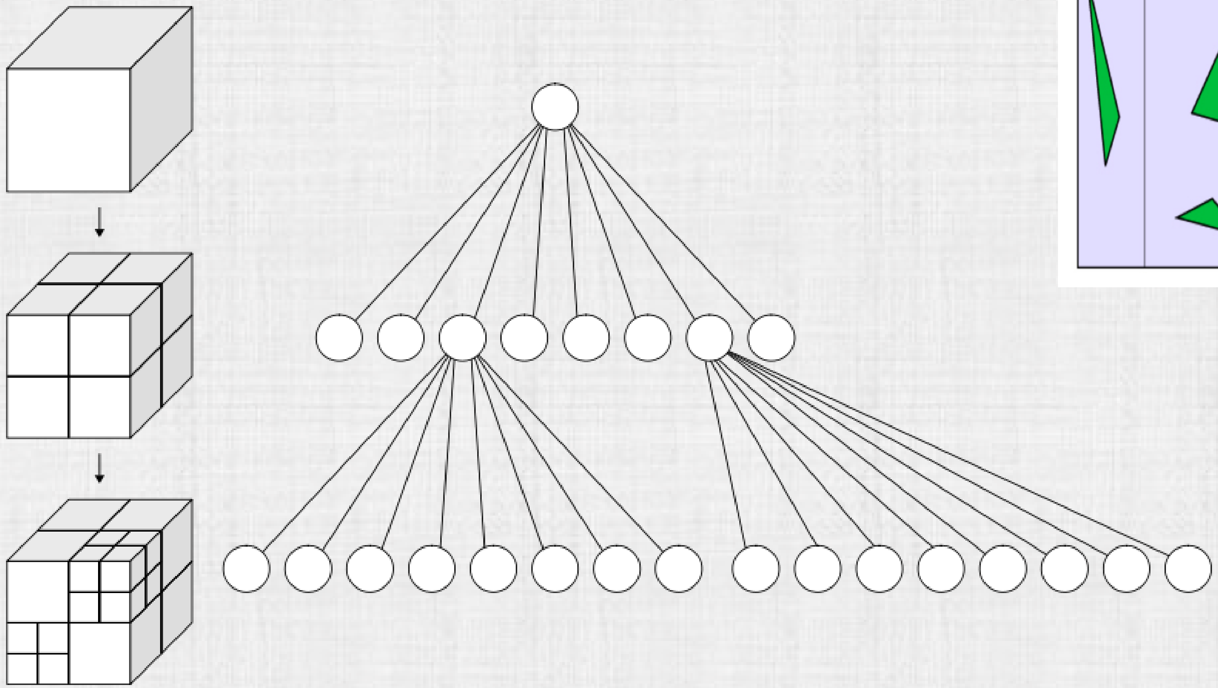


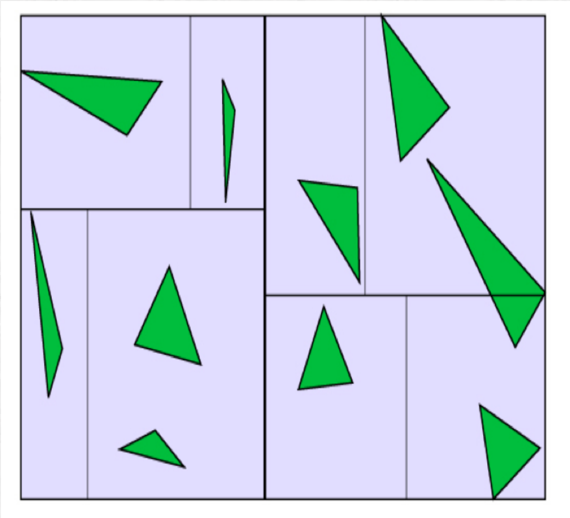


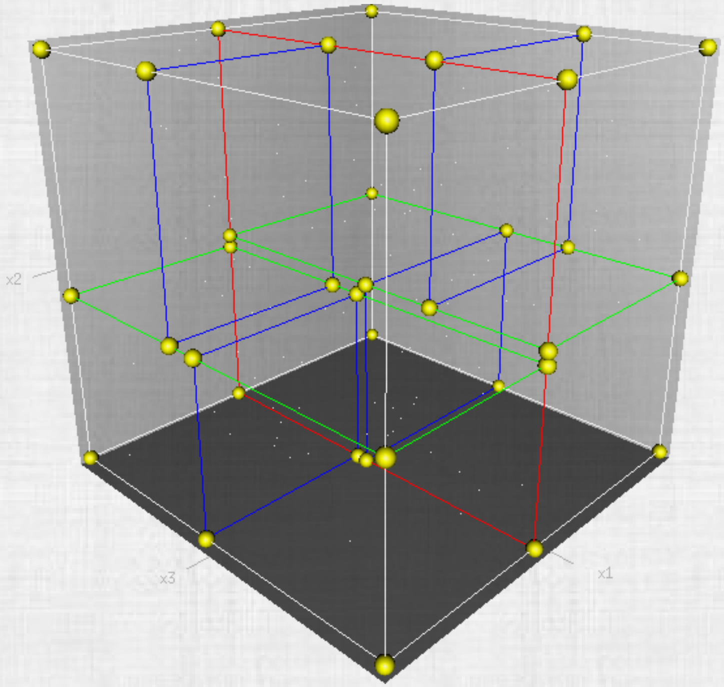


Aside: Code Acceleration (Hierarchical Bounding Volumes)

- Instead of a bottom-up bounding volume hierarchy approach, <u>octrees</u> and <u>K-D trees</u> take a top-down approach to hierarchically partitioning objects (and space)







Normals

- The surface normal at the intersection point $R(t_{int})$ can be used to approximate a plane (locally) tangent to the surface

- Objects tilted towards the light are bombarded with more photons than those tilted away from the light
- Compare the (unit) incoming light direction $\hat{L}$ with the (unit) normal $\hat{N}$ to approximate the tilt:

$$-\hat{L} \cdot \hat{N} = \cos\theta$$

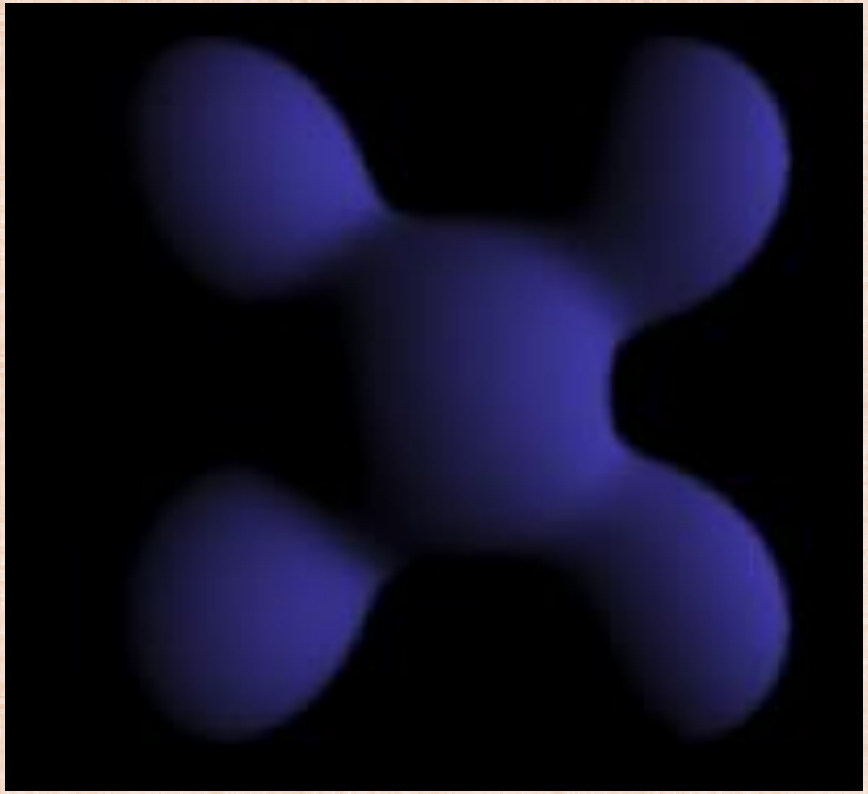
- Incoming light with intensity I is scaled down to $I \max(0, \cos\theta)$
  - the max with 0 prunes surfaces facing away from the light

- If (k_R, k_G, k_B) is the RGB color of a triangle, where $k_R, k_G, k_B \in [0,1]$ are surface reflection coefficients, then the pixel color would be $(k_R, k_G, k_B)\, I \max(0, \cos\theta)$

Ambient vs. Diffuse Shading

- Ambient shading colors a pixel when its ray intersects the object
- Diffuse shading attenuates object color based on how far the unit normal is tilted away from the incoming light (note how your eyes/brain imagine a 3D shape)

Ambient



Diffuse

Computing Unit Normals

- The unit normal to a plane is used in the plane's definition, and is thus readily accessible
  - though, it might need to be normalized to unit length

- The unit normal to a triangle is computed by normalizing the cross product of two edges
- Be careful with the edge ordering to ensure that the normal points outwards from the object (as opposed to inwards)

- The (outward) unit normal of a sphere is computed via $\hat{N} = \frac{R(t_{int}) - C}{\|R(t_{int}) - C\|_2}$

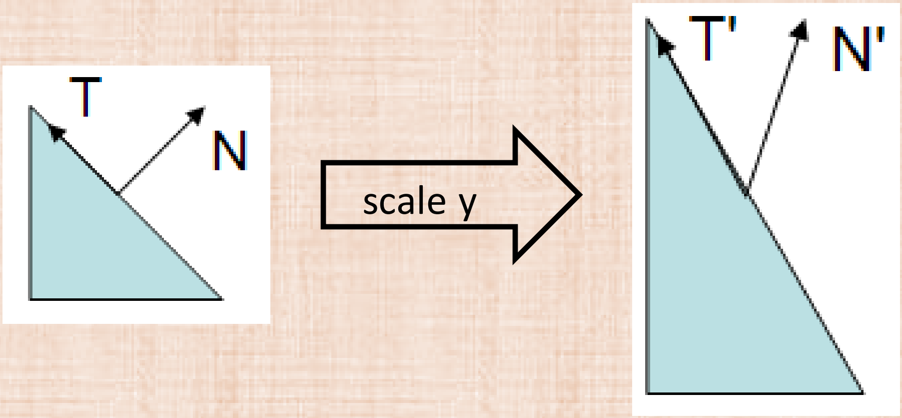
- For other objects, need to provide a function that returns an (outward) unit normal for any intersection point

Ray Tracing Transformed Objects

- When ray tracing in object space, the object space normal needs to be transformed back into world space (along with the intersection point)

- Let u and v be edge vectors of an object space triangle
- Let Mu and Mv be the corresponding world space edges
- The object space normal $\hat{N}$ is transformed to world space via $M^{-T}\hat{N}$
  - $Mu \cdot M^{-T}\hat{N} = u^T M^T M^{-T}\hat{N} = u^T\hat{N} = u \cdot \hat{N} = 0$, and $Mv \cdot M^{-T}\hat{N} = 0$
  - $M^{-T}\hat{N}$ needs to be normalized to make it unit length

- Careful, DO NOT USE $M\hat{N}$ as the world space normal:



N’ is not the normal

Shadows

- Shadow rays are used to determine whether photons from a light source are blocked by other objects or by parts of the same object

- A shadow ray is cast from the intersection point $R(t_{int}) = R_o$ in the direction of the light $-\hat{L}$

$$S(t) = R_o - \hat{L}t \quad \text{where} \quad t \in (0, t_{light})$$

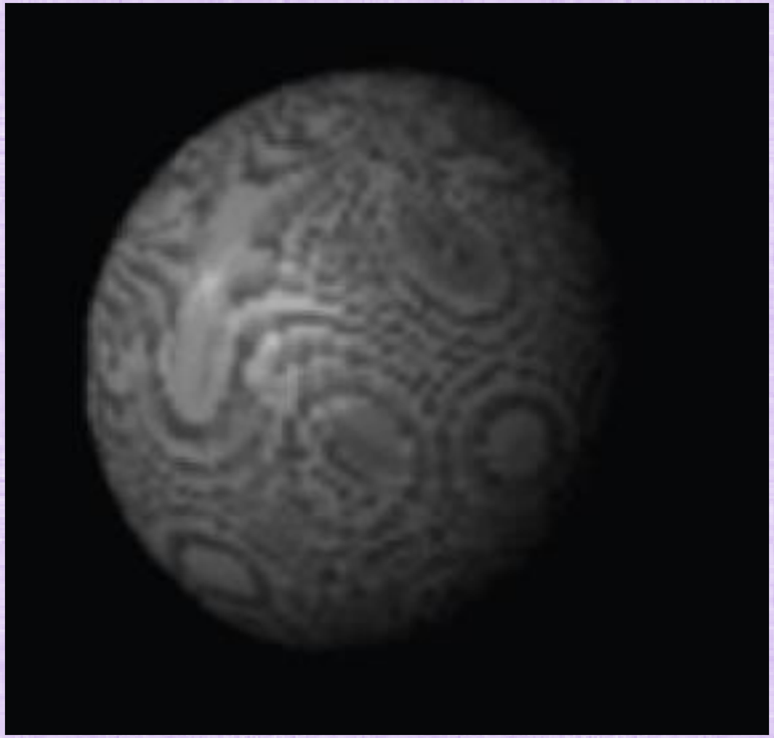
- If no intersections are found in $(0, t_{light})$, the light source is unobscured
- Otherwise, the point is shadowed, and the light source is ignored

Notes:

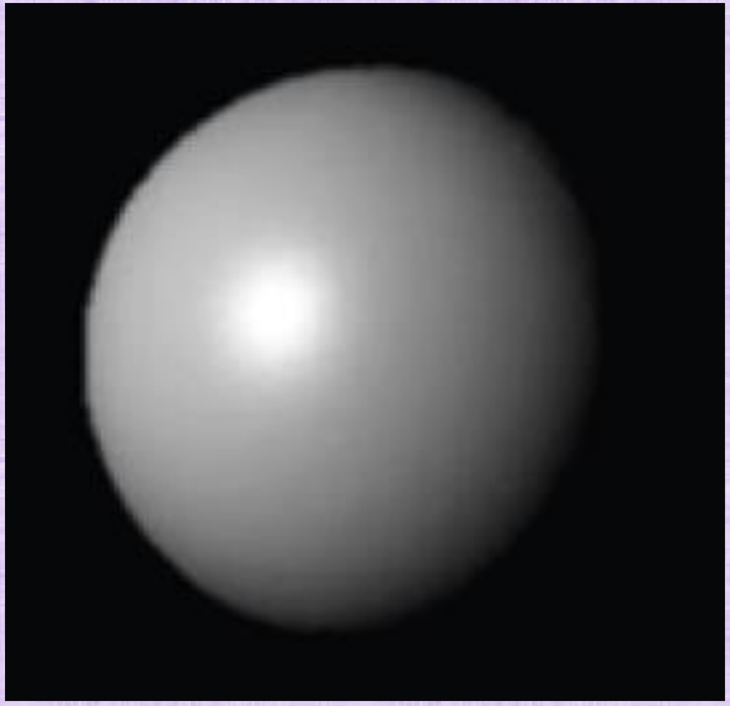
- Every light source is checked with a separate shadow ray
- Ambient shading is often used for points shadowed from all lights, so that they are not completely black

Spurious Self-Occlusion

- $t = 0$ is not included in $t \in (0, t_{light})$, to avoid incorrect self-intersections near R_o
  - This can still happen because of issues with numerical precision
- Note: Some shadow rays should self-intersect, such as those on the back-side of an object



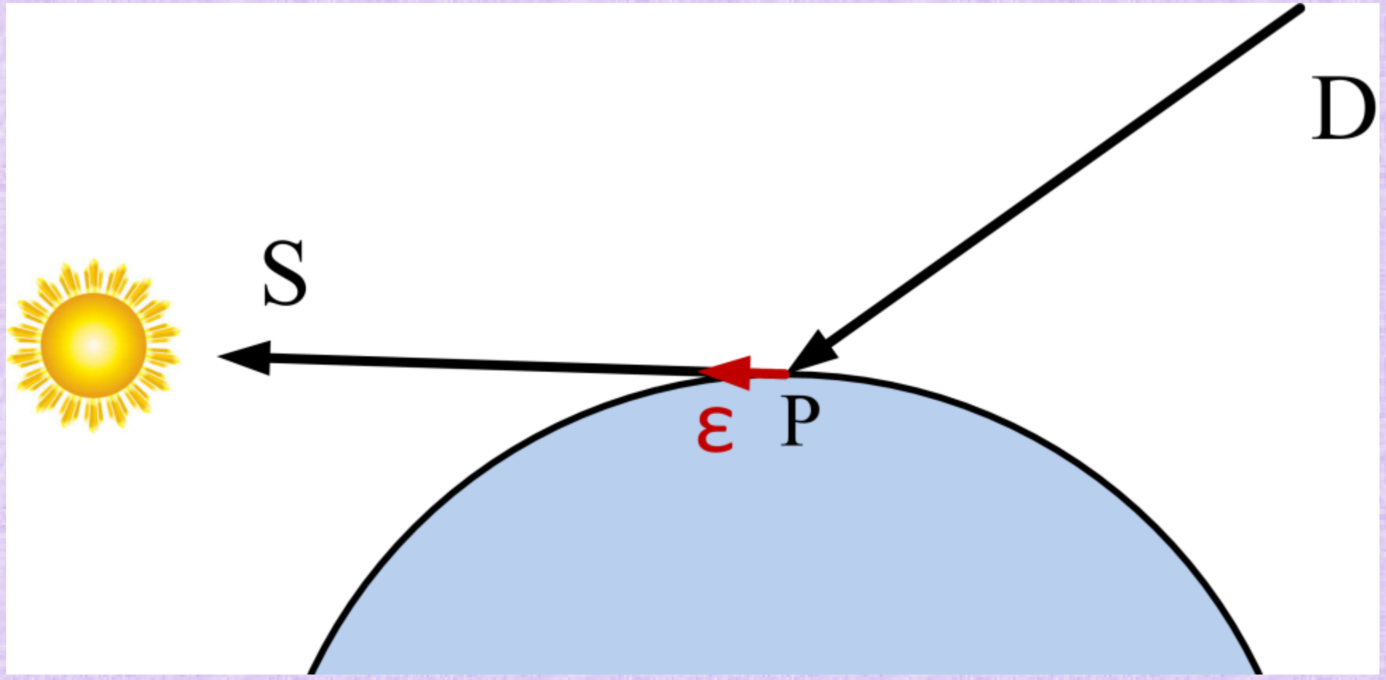
incorrect self-shadowing



correct self-shadowing

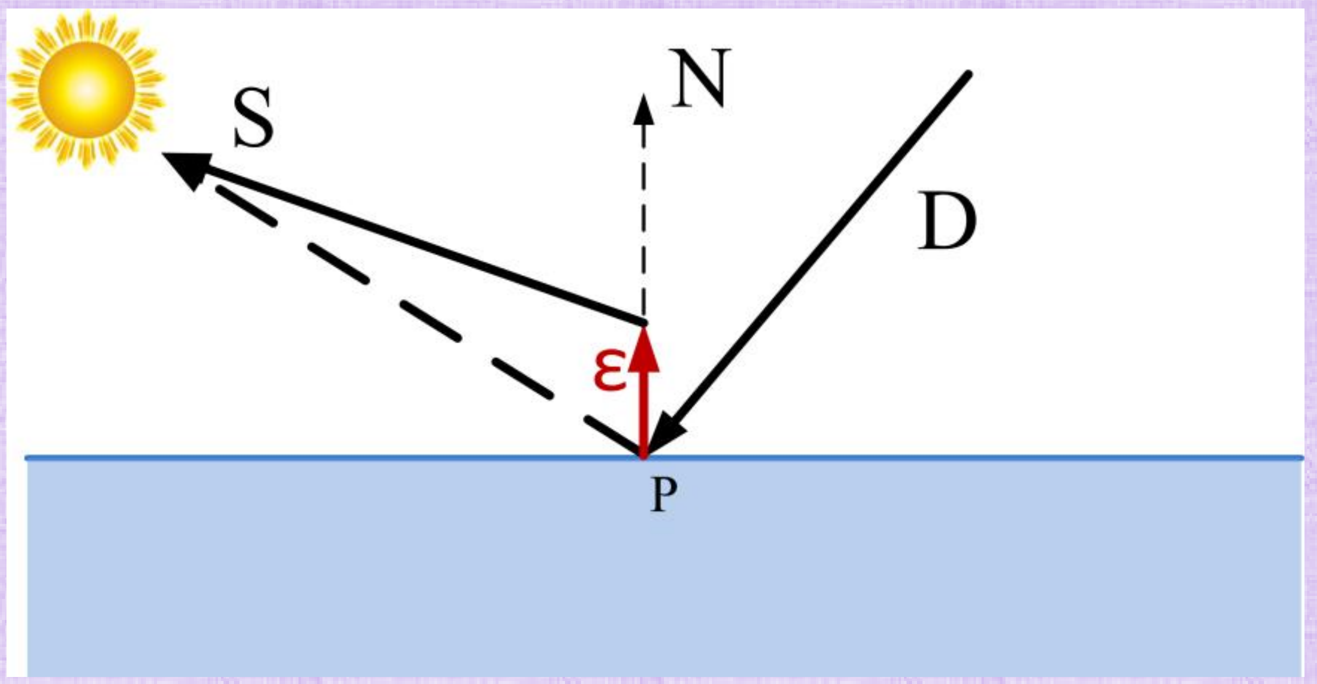
Fixing Spurious Self-Occlusion

- Use $t \in (\epsilon, t_{light})$ with an $\epsilon > 0$ large enough to avoid numerical precision issues
- This works well for many cases
- However, grazing shadow rays may still incorrectly re-intersect the object



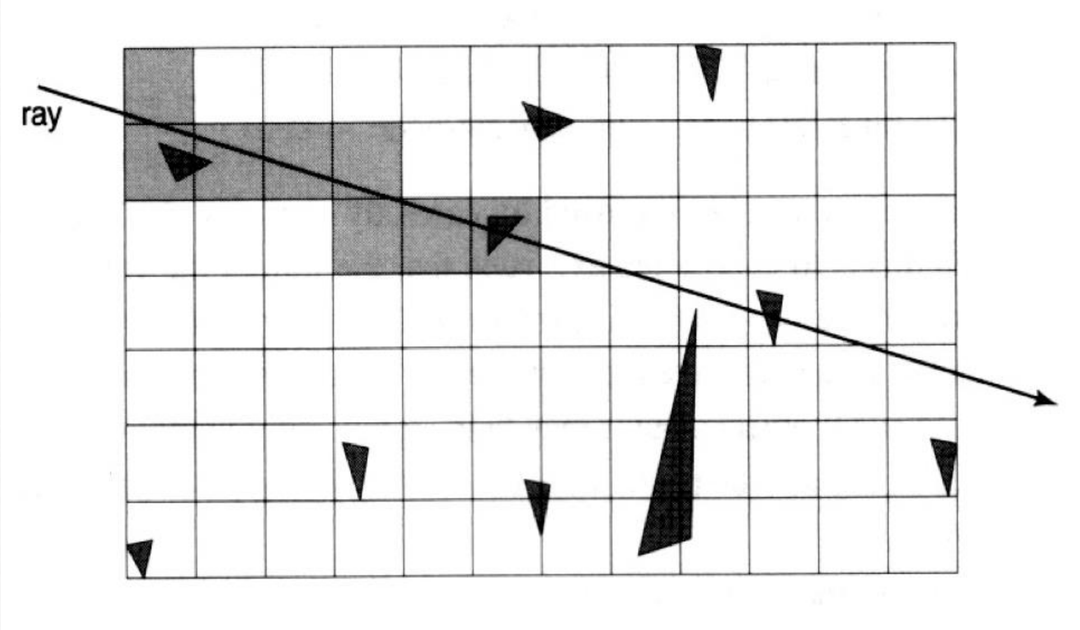
Fixing Spurious Self-Occlusion

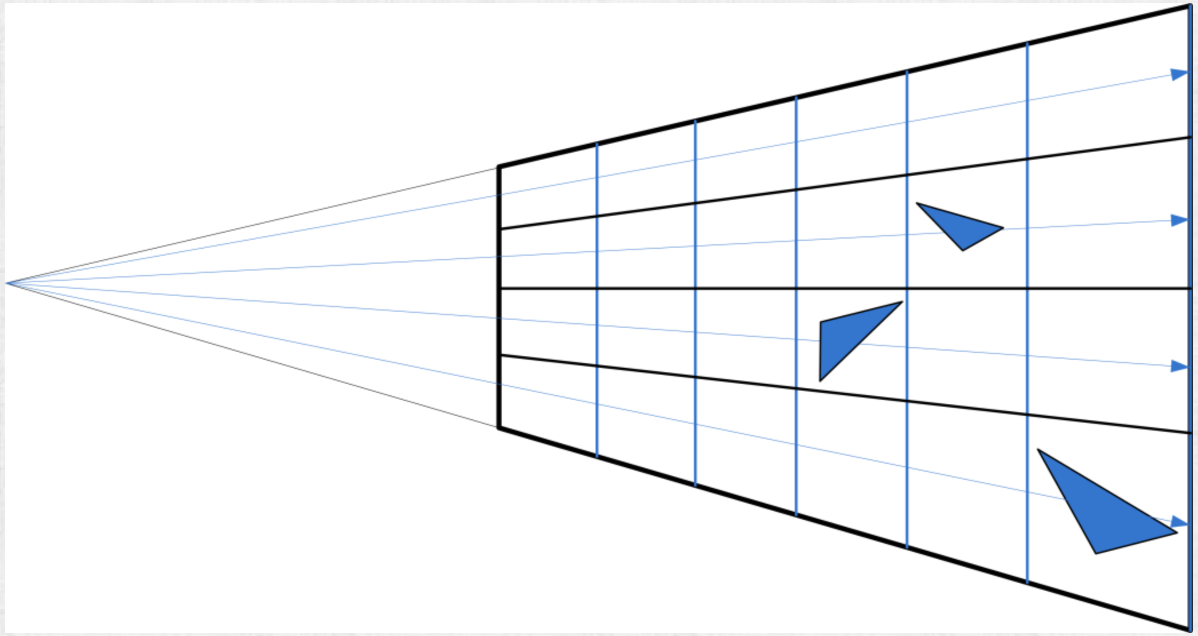
- Perturb the starting point of the shadow ray (typically in the normal direction), i.e. from R_o to $R_o + \epsilon\widehat{N}$
- The ray direction needs to be modified too, to go from $R_o + \epsilon\widehat{N}$ to the light
- The new shadow ray is $S(t) = R_o + \epsilon\widehat{N} - \hat{L}_{mod}t$
- Need to be careful that the new starting point isn't inside (or too close to) any other geometry



Aside: Code Acceleration (Scene Partitioning)

- When there are many objects in the scene, checking rays against all of their top level simple bounding volumes can become expensive
- Thus, bounding volume hierarchies (octrees, K-D trees, etc..) are used to partition the world space scene
- Also useful (but flat instead of hierarchical) are uniform spatial partitions (uniform grids) and viewing frustum partitions





Aside: Code Acceleration (Scene Partitioning)

- There are many variants: rectilinear grids with movable lines, hierarchies of uniform grids, and a structure proposed by [Losasso et al. 2006] that allows octrees to be allocated inside the cells of a uniform spatial partition