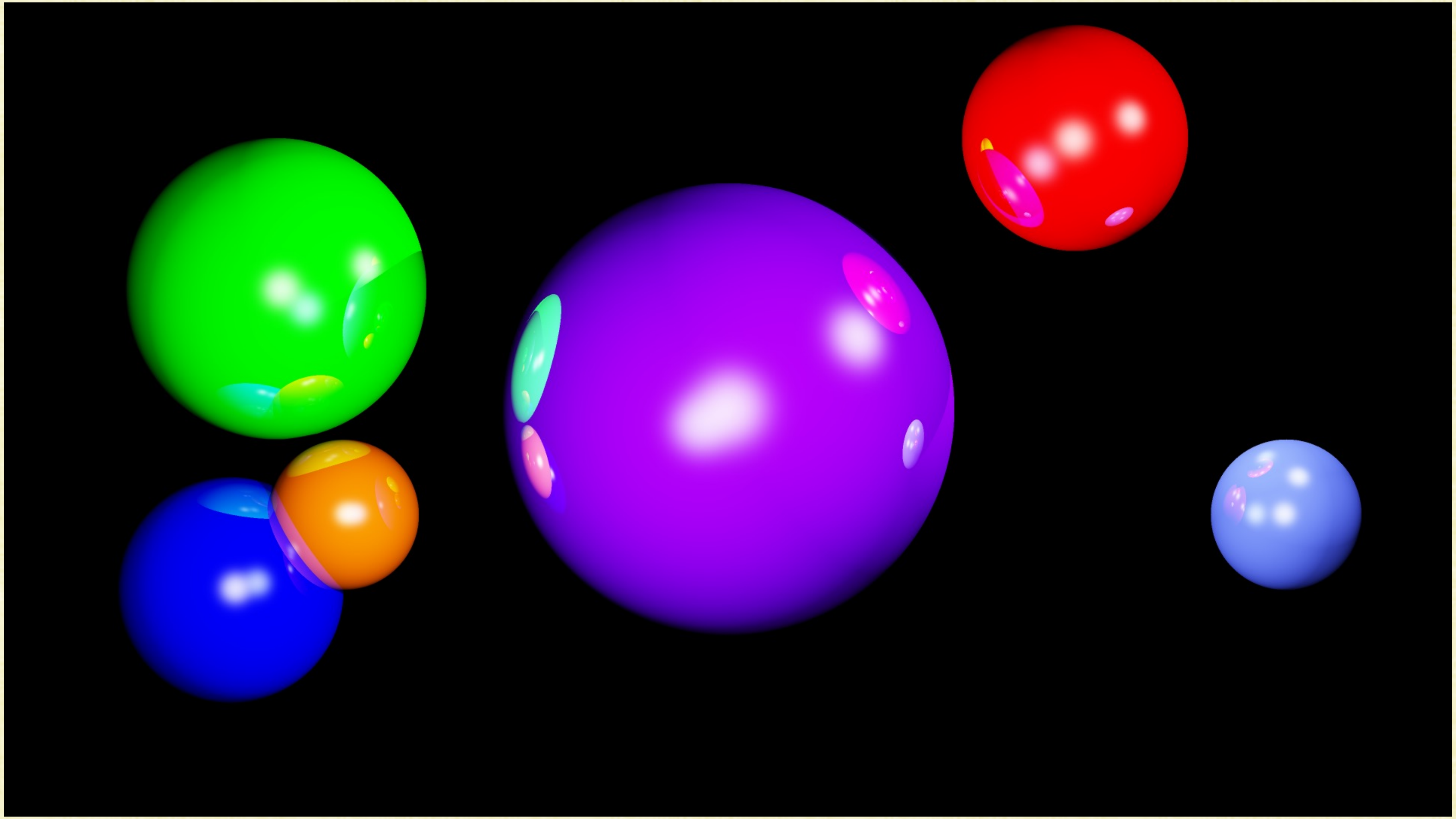
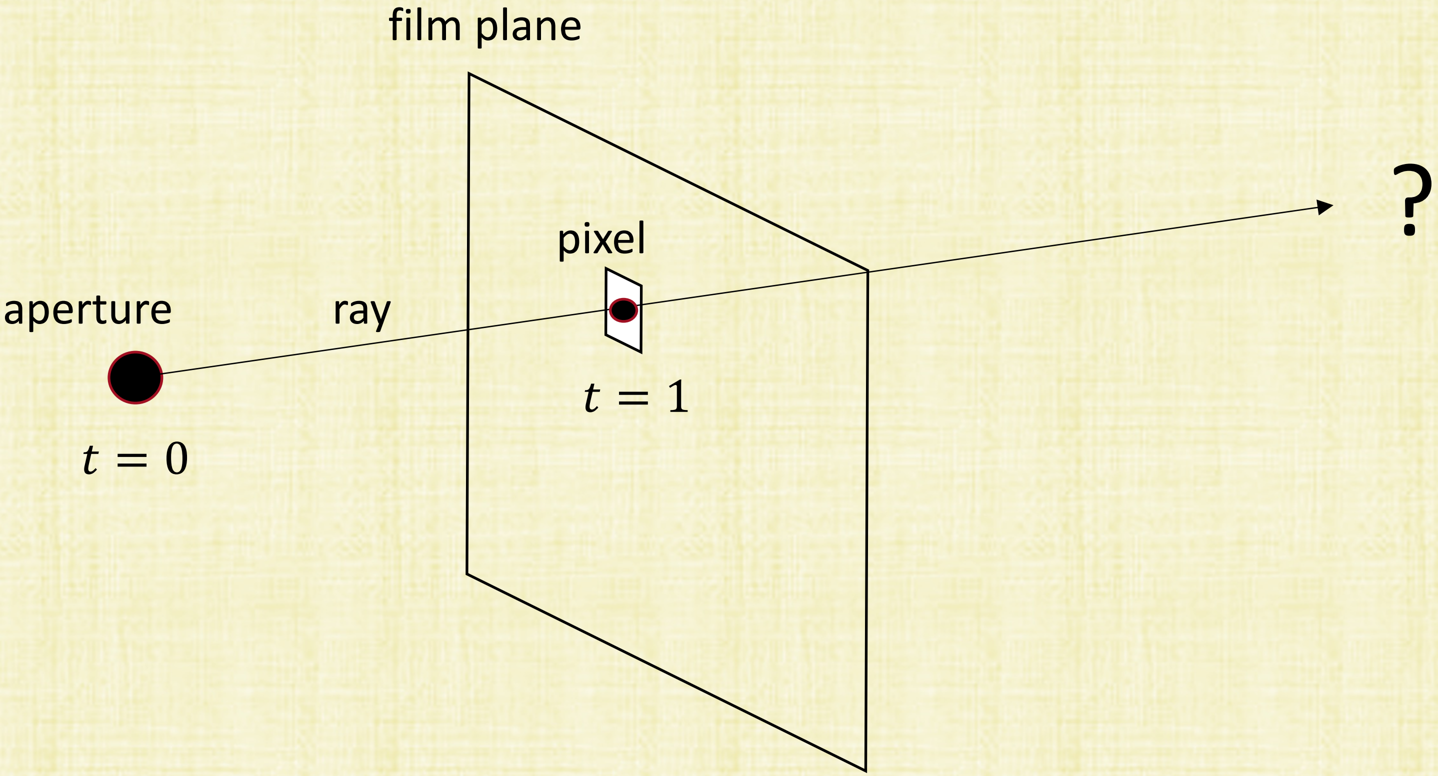
Ray Tracing



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 $t \in [1, t_{far}]$ will be 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 a per triangle operation)
- Ray tracing is inherently parallel (the ray for each pixel is independent of the rays for other pixels)

- Can utilize modern parallel CPUs/Clusters/GPUs to significantly accelerate ray tracing
  - Threading (e.g., Pthread, OpenMP) 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

- For the sake of comparison: Scanline rendering is a per triangle operation, and 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:

- Triangles are contained in planes, so it can be useful to look at Ray-Plane intersections first
- A Ray-Plane intersection yields a point, and a subsequent test determines whether that point is inside (or outside) the triangle
  - Both the triangle and the point can be projected into 2D, and the 2D triangle rasterization test (to the left of all 3 rays, discussed last week) can be used to determine “inside”
    - Can project can into the xy, xz, yz plane by merely dropping the z, y, x coordinate (respectively) from the triangle vertices and the point
    - Most robust to drop the coordinate with the largest component in the triangle’s normal (so that the projected triangle has maximal area)
  - Alternatively, there is a fully 3D version of the 2D rasterization

- One can skip the Ray-Plane intersection and consider the Ray-Triangle intersection directly
  - This is similar to how ray tracing works for non-triangle geometry (ray tracers handle non-triangle geometry better than scanline rendering does)

Ray-Plane Intersection

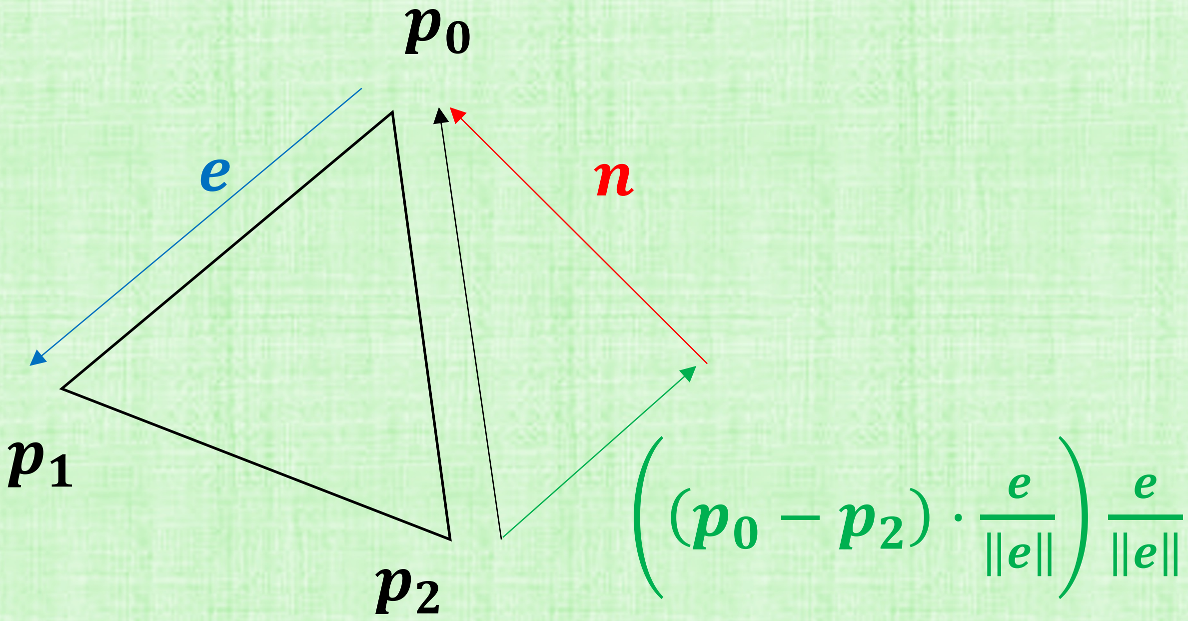
- A plane is defined by a point p_o (on it) and a normal direction N
- A point p is on the plane if $(p - p_o) \cdot N = 0$
- A ray $R(t) = A + (P - A)t$ intersects the plane when $(R(t) - p_o) \cdot N = 0$ for some $t \geq 0$
- That is, $(A + (P - A)t - p_o) \cdot N = 0$ or $(A - p_o) \cdot N + (P - A) \cdot Nt = 0$
- So, $t = \frac{(p_o - A) \cdot N}{(P - A) \cdot N}$
- Note: The length of N cancels (so it need not be unit length)

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

- Note: a (non-unit length) triangle normal can be computed by taking the cross product of any two edges (as long as the triangle does not have zero area)
- Note: Any triangle vertex can be used as a point on the plane

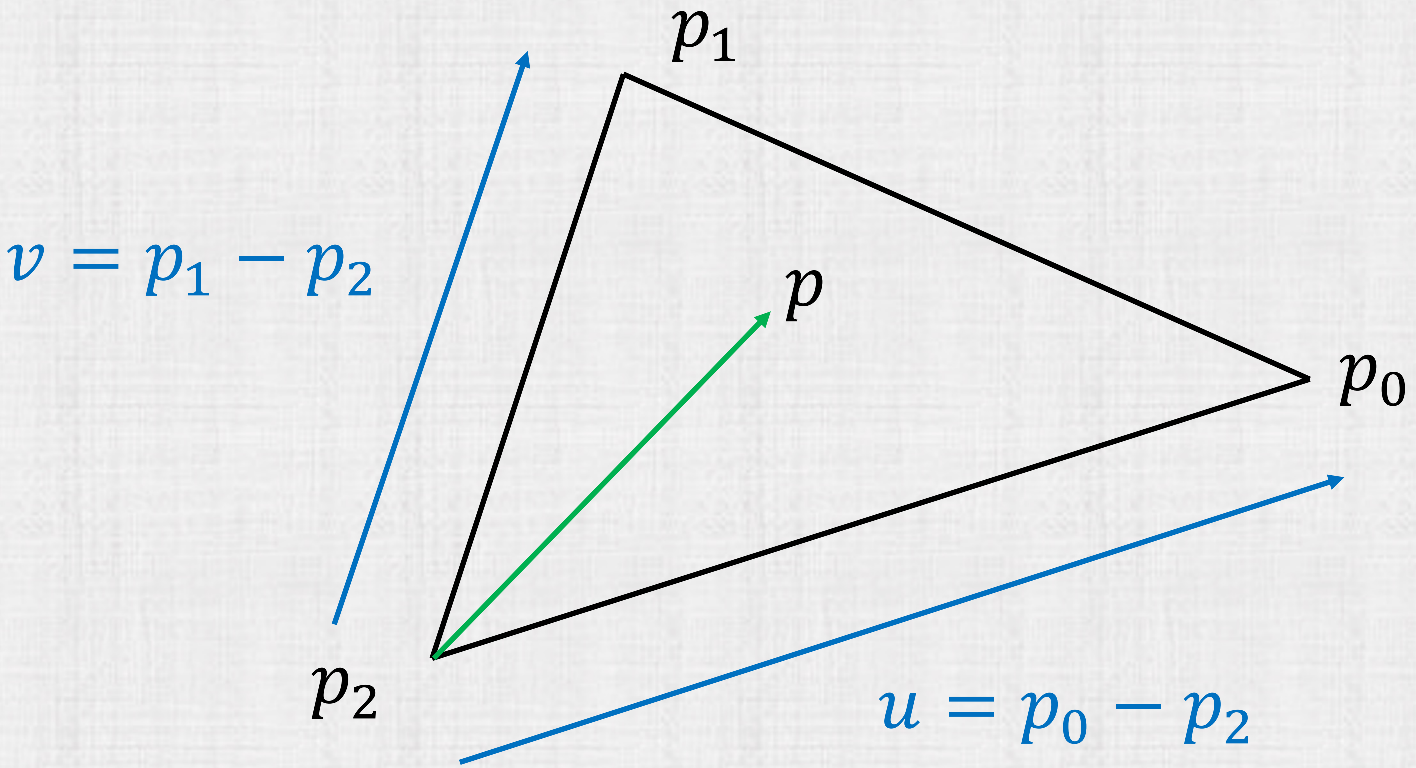
3D Point Inside a 3D Triangle

- Given $t_{int} = \frac{(p_o - A) \cdot N}{(P - A) \cdot N}$, evaluate $R(t_{int}) = R_o$ to find 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 interior to e when $(R_o - p_0) \cdot n < 0$
- If R_o is interior to all three edges, it is interior to the triangle



Recall: Triangle Basis Vectors

- Compute edge vectors $u = p_0 - p_2$ and $v = p_1 - p_2$
- Any point p interior to the triangle can be written as $p = p_2 + \beta_1 u + \beta_2 v$ with $\beta_1, \beta_2 \in [0,1]$ and $\beta_1 + \beta_2 \le 1$
- Substitutions and collecting terms gives $p = \beta_1 p_0 + \beta_2 p_1 + (1 - \beta_1 - \beta_2) p_2$ implying the equivalence: $\alpha_0 = \beta_1,\ \alpha_1 = \beta_2\,, \alpha_2 = 1 - \beta_1 - \beta_2$



Direct Ray-Triangle Intersection

- Triangle Basis Vectors: $p = p_2 + \beta_1 u + \beta_2 v$ with $\beta_1, \beta_2 \in [0,1]$ and $\beta_1 + \beta_2 \leq 1$
- Points on the ray have $R(t) = A + (P - A)t$
- An intersection point has $A + (P - A)t = p_2 + \beta_1 u + \beta_2 v$
- Or $\begin{pmatrix} u & v & A - P \end{pmatrix} \begin{pmatrix} \beta_1 \\ \beta_2 \\ t \end{pmatrix} = A - p_2$ where $\begin{pmatrix} u & v & A - P \end{pmatrix}$ is a 3x3 matrix and $A - p_2$ is a 3x1 vector (3 equations with 3 unknowns)
- This 3x3 system is degenerate when the columns of the 3x3 matrix 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 that unique solution has: $\beta_1, \beta_2 \in [0,1]$ and $\beta_1 + \beta_2 \leq 1$

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

Solving with Cramer’s Rule

- Solving the 3x3 system with Cramer’s Rule allows for code optimization:

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

- Compute $\beta_1 = \frac{\Delta_{\beta_1}}{\Delta}$ where $\Delta_{\beta_1} = |(A - p_0 \quad v \quad A - P)|$
- When $\beta_1 \notin [0,1]$, can quit early

- Compute $\beta_2 = \frac{\Delta_{\beta_2}}{\Delta}$ where $\Delta_{\beta_2} = |(u \quad A - p_0 \quad A - P)|$
- When $\beta_2 \in [0,1 - \beta_1]$, 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 geometry
- This is in contrast to scanline rendering where objects need to be turned into triangles
- In addition to triangle meshes, ray tracers often use: analytic descriptions of geometry, implicitly defined surfaces, parametric surfaces, etc.

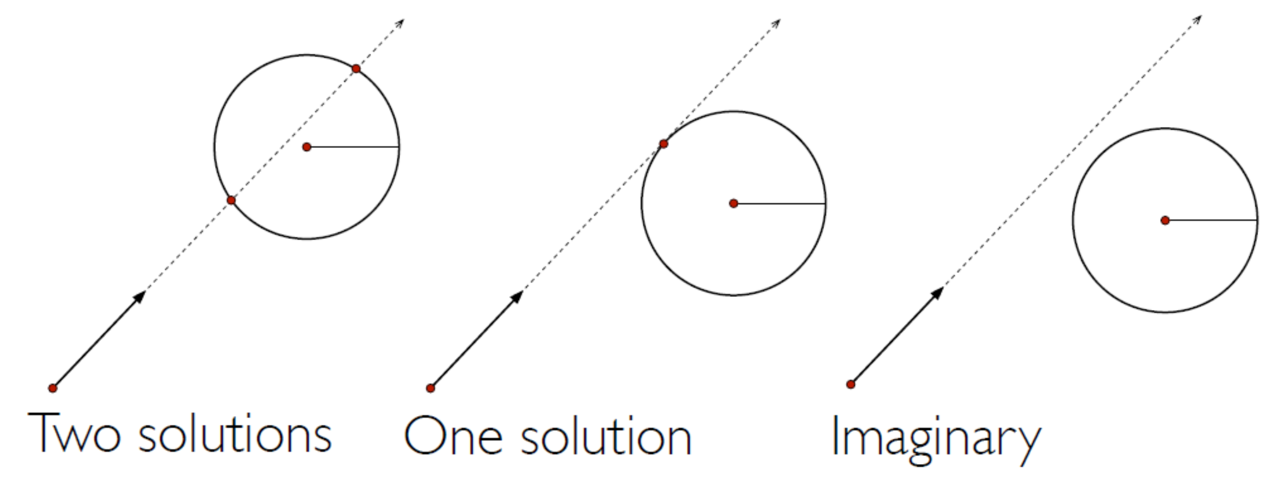
- The surfaces of many objects can be written as functions
- E.g., $f(p) = 0$ if and only if p is on the surface (e.g. the equation for a plane)
- Sometimes there are additional constraints (such as on the barycentric weights for triangles)
- One quite useful class of such objects are implicit surfaces (covered later in the class)
- Ray-object intersection routines often proceed down a similar path:
  - substitute the ray equation in for the point, i.e. $f(R(t)) = 0$
  - solve for t
  - check the solution against any additional constraints

Ray-Sphere Intersections

- A point p is on a sphere with center C and radius r when $\|p - C\|_2 = r$
  - Or (squaring both sides), when $(p - C) \cdot (p - C) = r^2$
- Substitute $R(t) = A + (P - A)t$ in for p to get a quadratic equation in t:

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

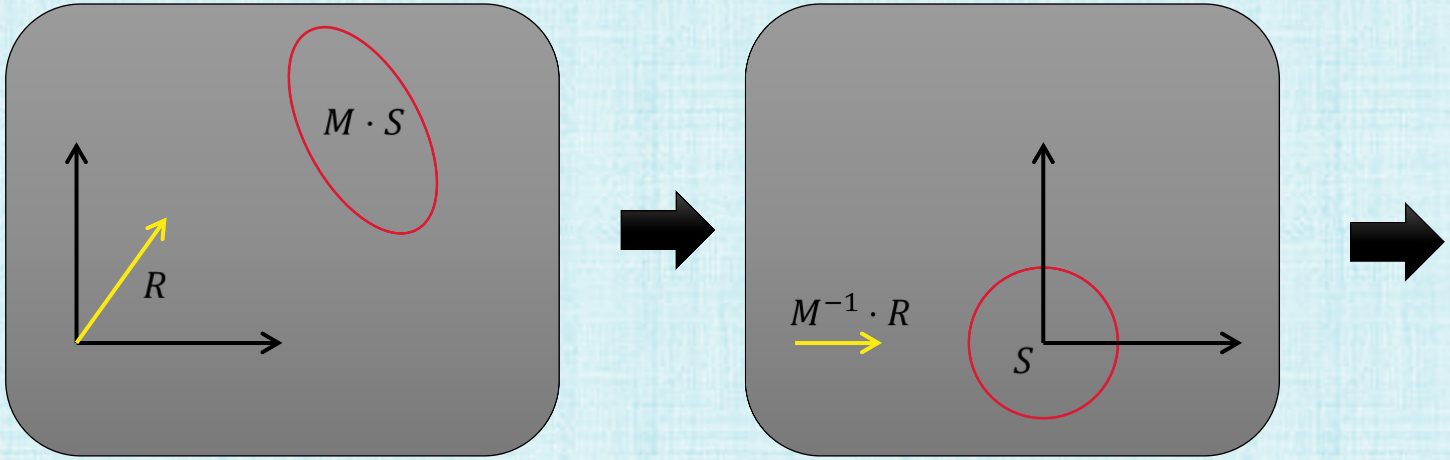
- When the discriminant of this quadratic equation is positive, there are two solutions (choose the one the ray hits first)
- When the discriminant is zero, there is one solution (the ray tangentially grazes the sphere)
- When the discriminant is negative, there are no solutions



Transformed Objects

- Geometry is often stored/represented in a convenient object space
- The object space can make the geometry simpler 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.

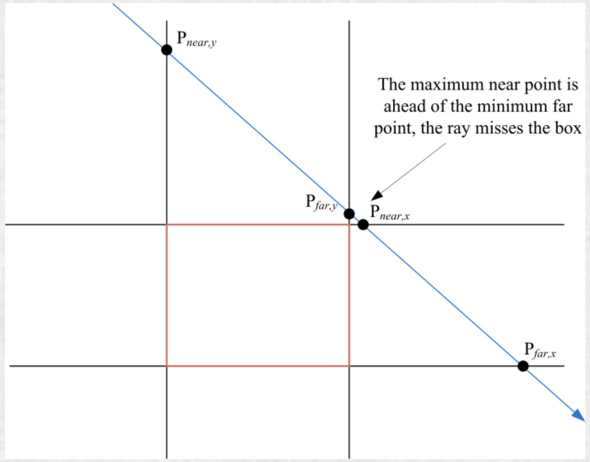
- We often prefer to ray trace in this convenient object space, rather than world space
- Transform the ray into object space and find the ray-object intersection, then transform the relevant information back to world space



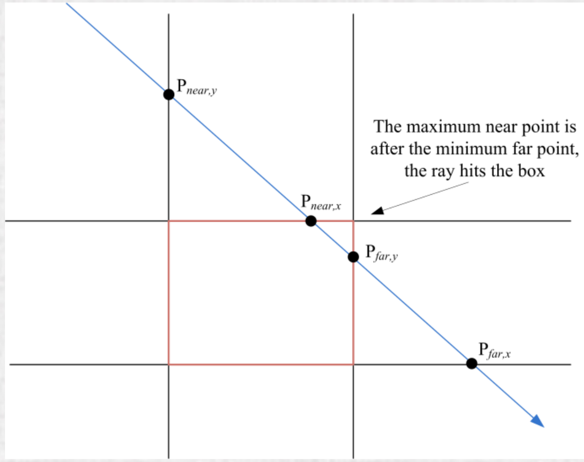
$$\boldsymbol{M} \cdot \boldsymbol{R}_{\boldsymbol{object}}(\boldsymbol{t}_{\boldsymbol{int}})$$

Aside: Code Acceleration

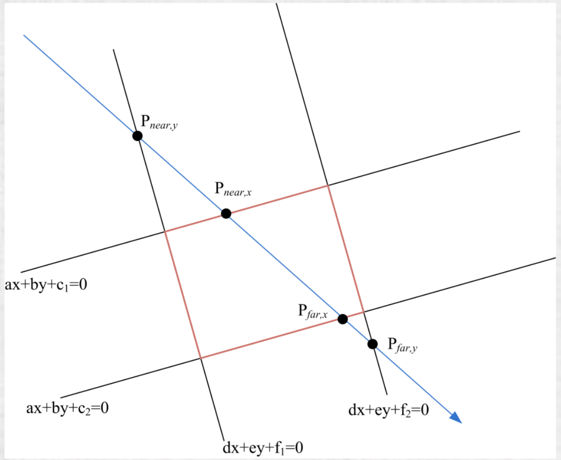
- Ray-Object intersections can be expensive
- So, put complex objects inside simpler objects, and first test for intersections against the simpler object (potentially skipping tests against the complex object)
- Simple <u>bounding volumes</u>: spheres, axis-aligned bounding boxes (AABB), or oriented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

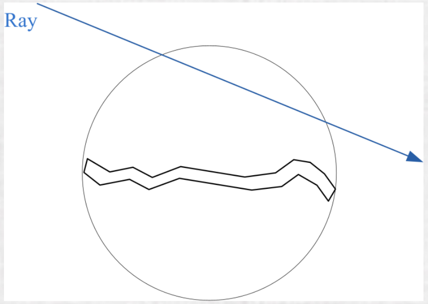


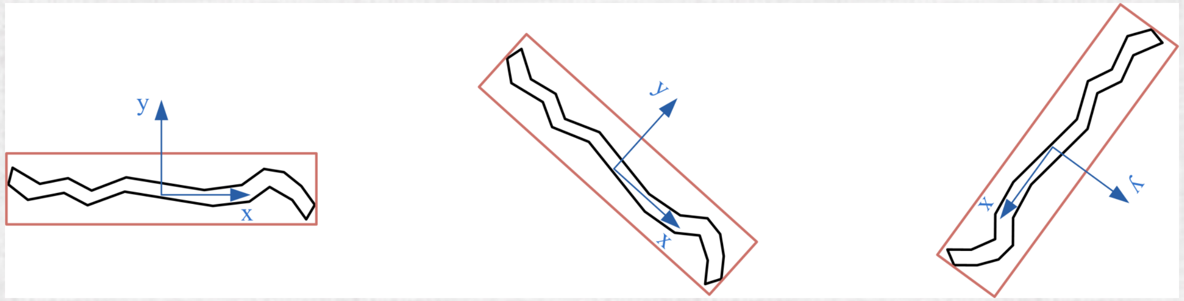
The maximum near point is ahead of the minimum far point, the ray misses the box



The maximum near point is after the minimum far point, the ray hits the box

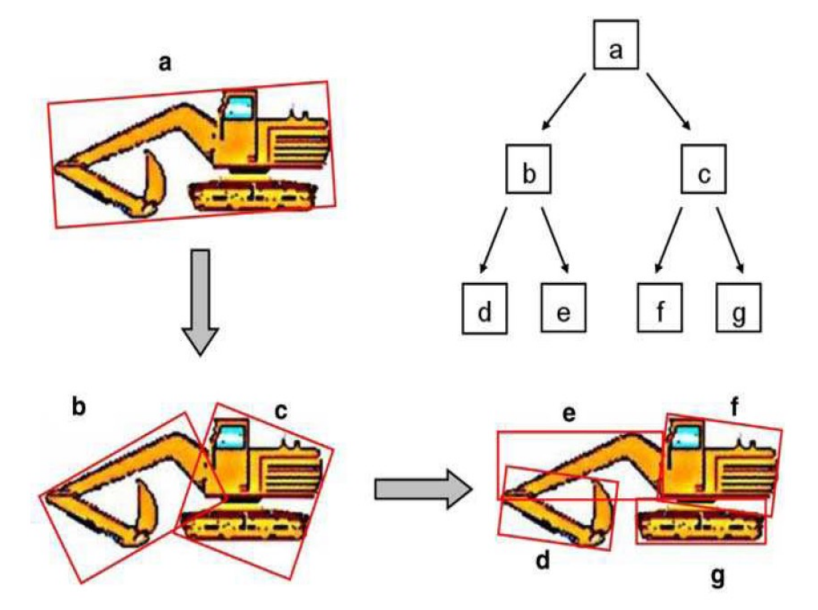


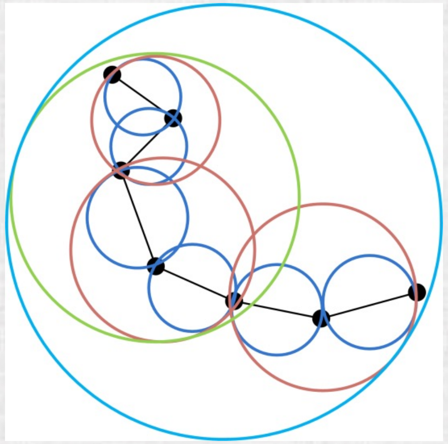


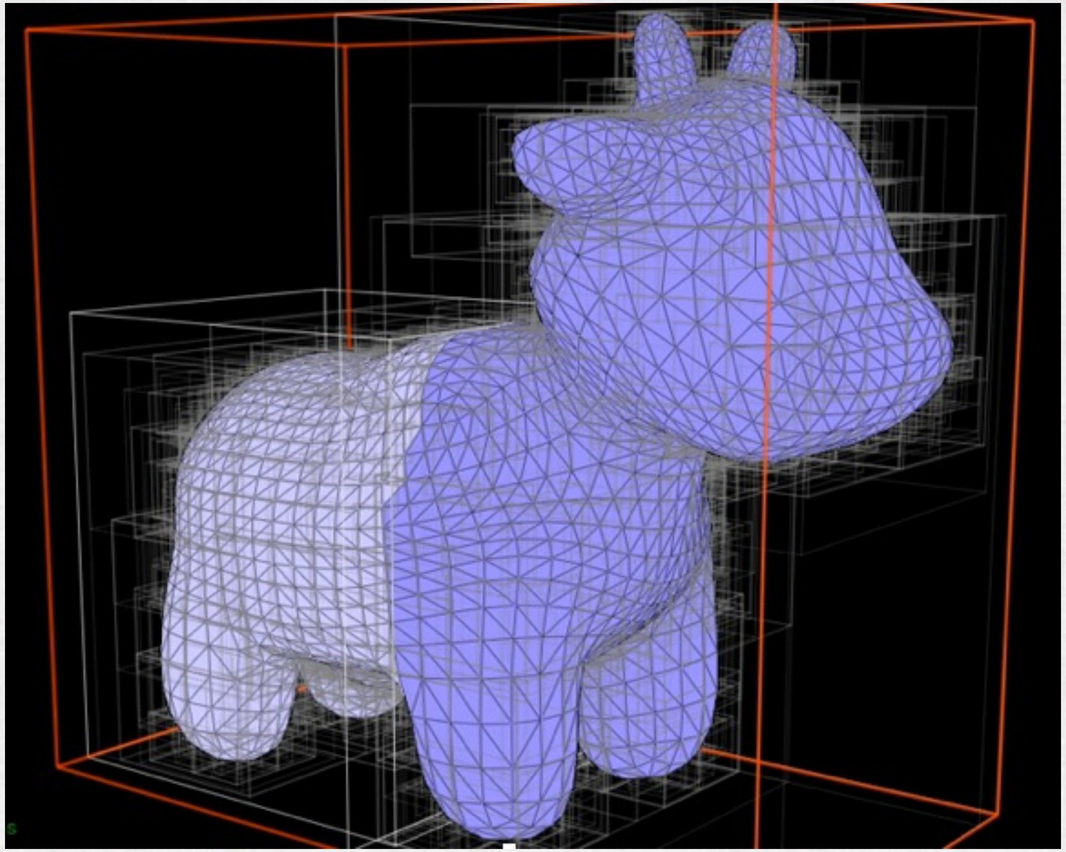


Aside: Code Acceleration

- For complex objects, build a hierarchical tree structure 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

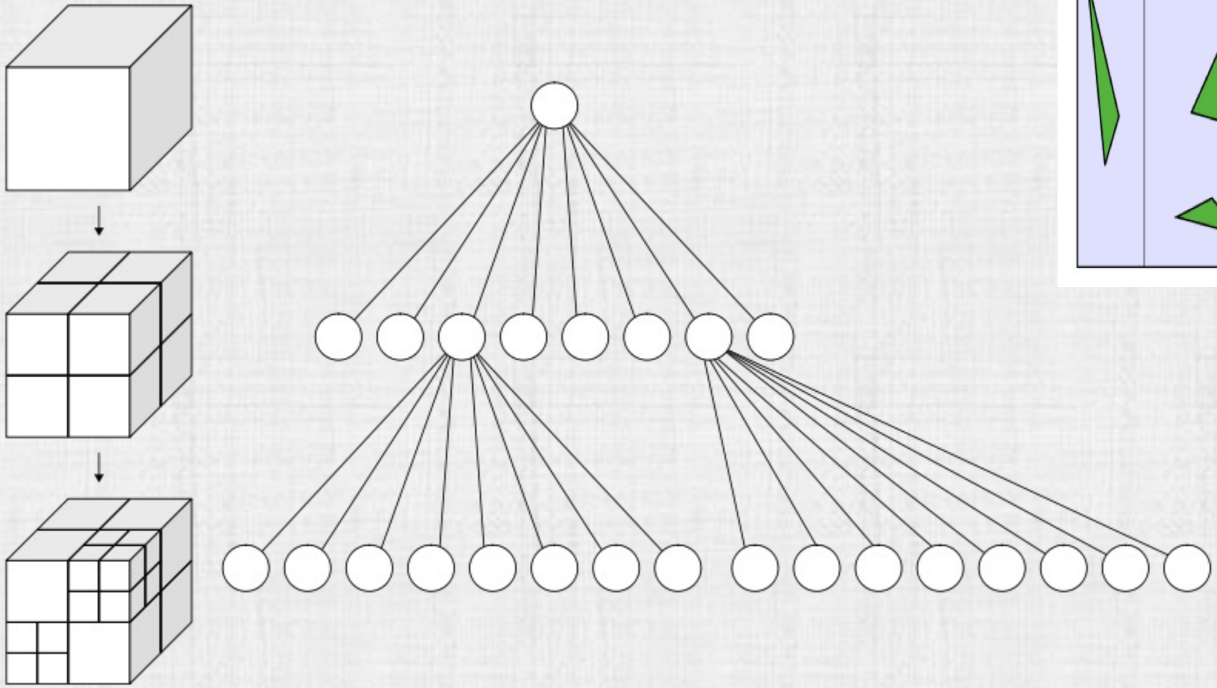


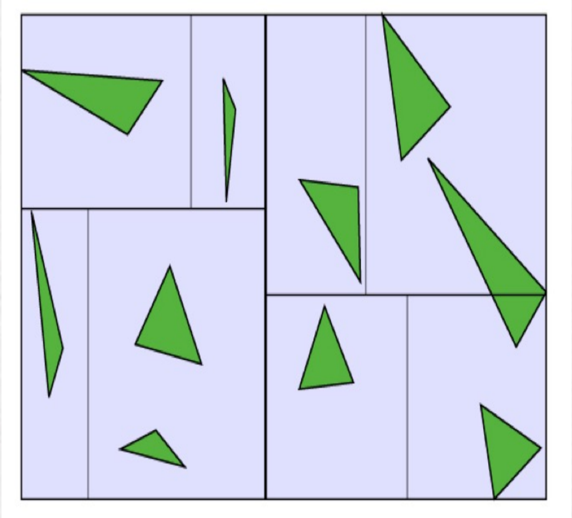


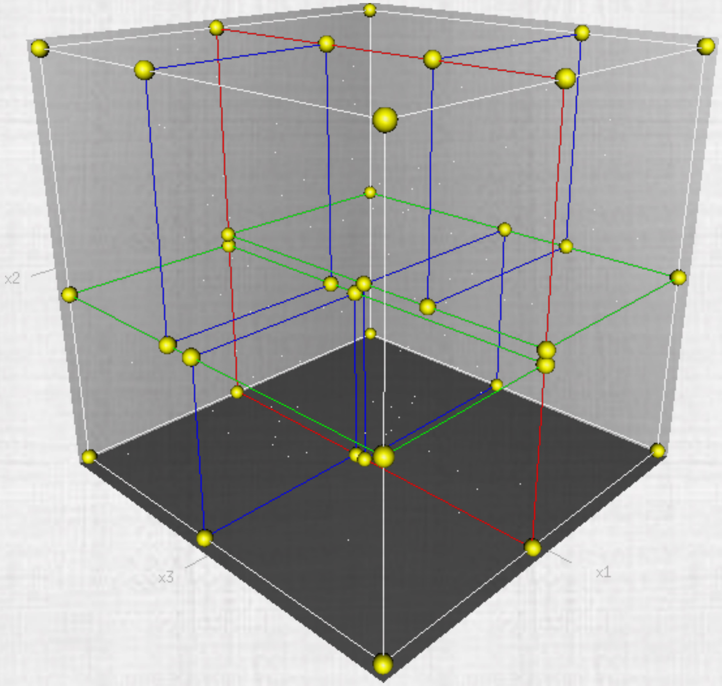


Aside: Code Acceleration

- 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

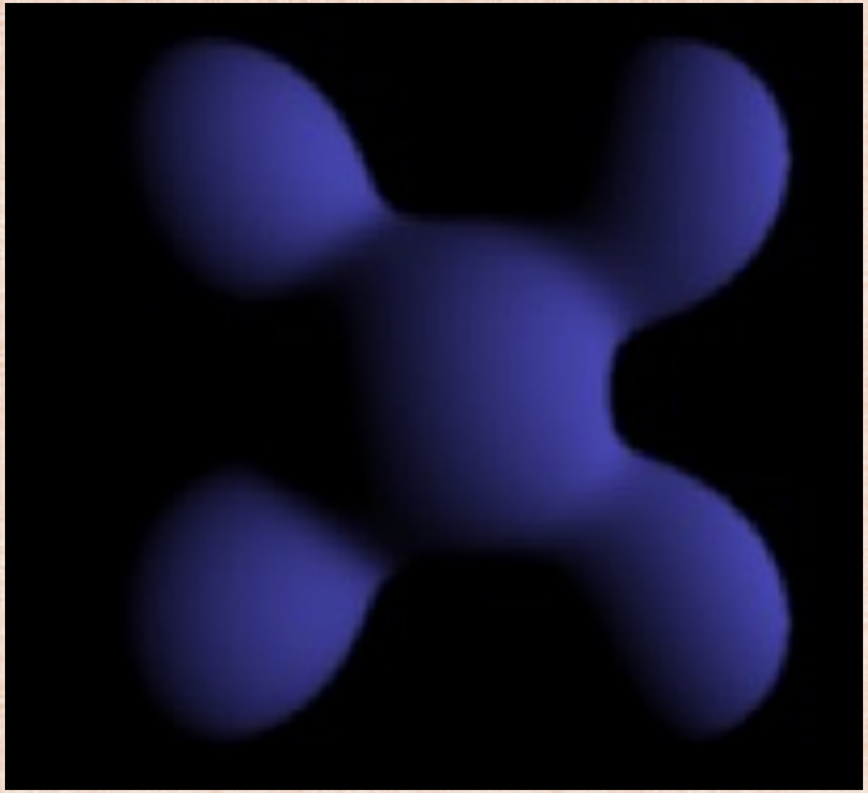
- Objects tilted towards the light are bombarded with more photons than those tilted away from the light

- The surface normal at the point $R(t_{int})$ can be used to approximate a plane (locally) tangent to the surface
- Compare the (unit) incoming light direction $\hat{L}$ with the (unit) normal $\hat{N}$ to approximate the titling angle via: $-\hat{L} \cdot \hat{N} = \cos\theta$
- 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 ($k_R, k_G, k_B \in [0,1]$ are reflection coefficients), then the pixel color is $(k_R, k_G, k_B)\, I \max(0, \cos\theta)$

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
  - although it might need to be normalized to unit length

- The unit normal to a triangle can be 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)

- For other objects: Need to provide a function that returns an (outward) unit normal for any point of intersection
- E.g., a sphere with intersection point $R(t_{int})$, has an (outward) unit normal of:

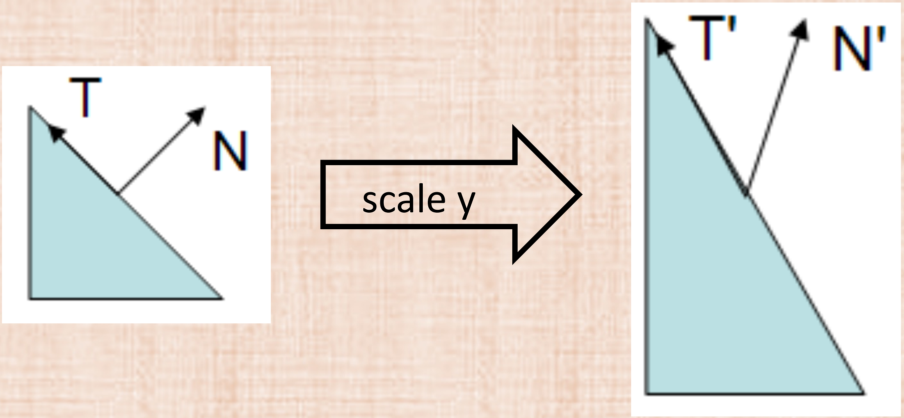
$$\hat{N} = \frac{R(t_{int}) - C}{\| R(t_{int}) - C \|_2}$$

Transformed Objects

- When ray tracing geometry 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 a triangle in object space
- Let Mu and Mv be their corresponding world space versions
- The object space normal $\widehat{N}$ is transformed to world space via $M^{-T}\widehat{N}$
  - Note: $Mu \cdot M^{-T}\widehat{N} = u^T M^T M^{-T}\widehat{N} = u^T\widehat{N} = u \cdot \widehat{N} = 0$, and $Mv \cdot M^{-T}\widehat{N} = 0$
  - Note: $M^{-T}\widehat{N}$ needs to be normalized to make it unit length

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



N' is not the normal

Shadows

- The incoming light intensity I needs to be reduced, when photons are blocked by other objects or parts of the same object
- Shadow rays determine whether photons from a light source are able to hit a point
- A shadow ray is cast from the intersection point $R(t_{int})$ in the direction of the light $-\hat{L}$,

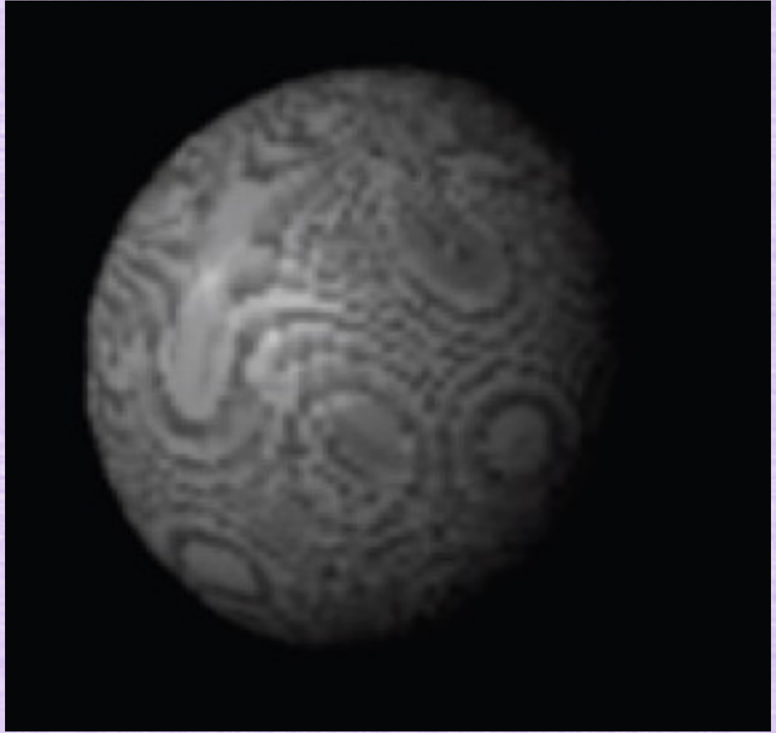
$$S(t) = R(t_{int}) - \hat{L}t \quad \text{with} \quad t \in (0, t_{light})$$

- If no intersections are found in $(0, t_{light})$, then the light source is unobscured
- Otherwise, the point is shadowed, and the light source is not used to color the pixel

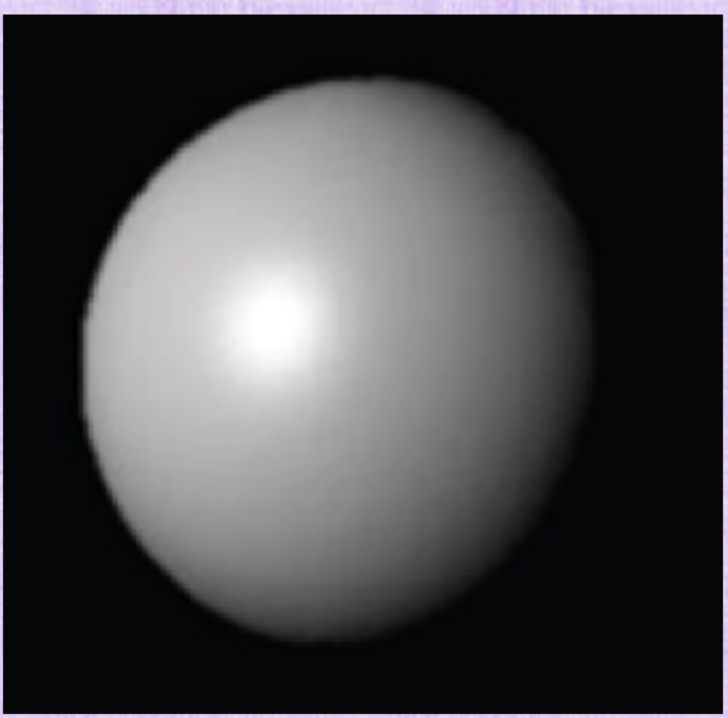
- Note: every light source is checked with a separate shadow ray
- Note: low intensity ambient shading is often used for points completely shadowed (so that they are not completely black)

Spurious Self-Occlusion

- $t = 0$ is not included in $t \in (0, t_{light})$, to avoid incorrect self-intersections near $R(t_{int})$
  - 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)



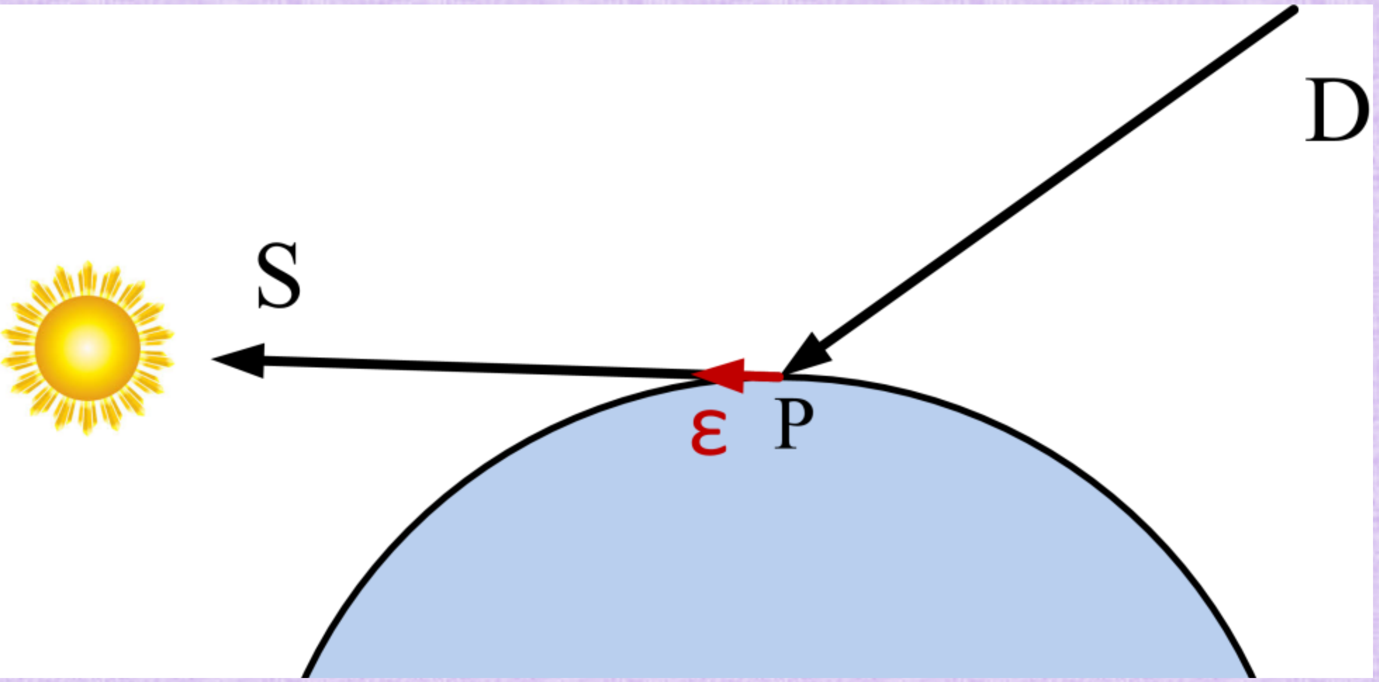
incorrect self-shadowing



correct shadowing

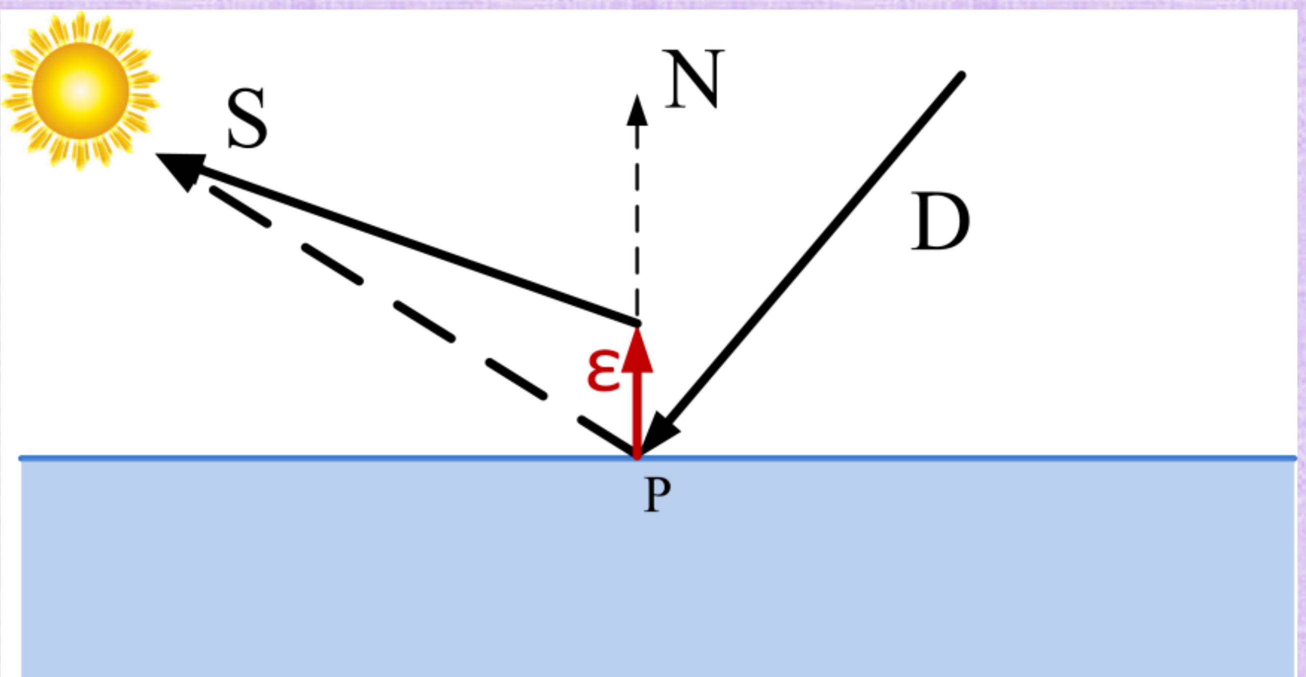
Spurious Self-Occlusion

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



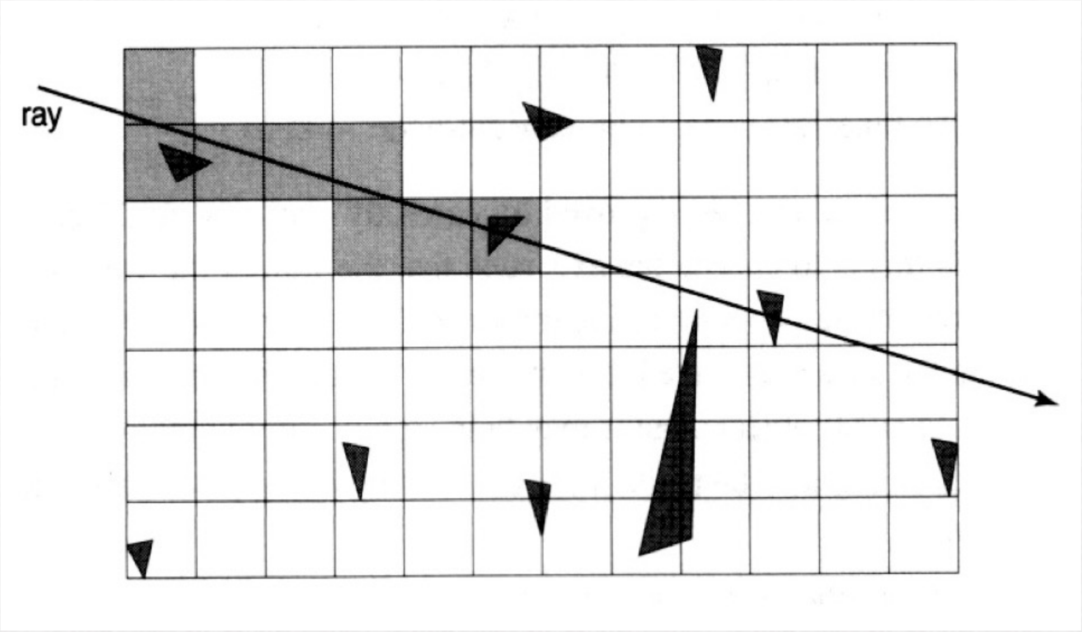
Spurious Self-Occlusion

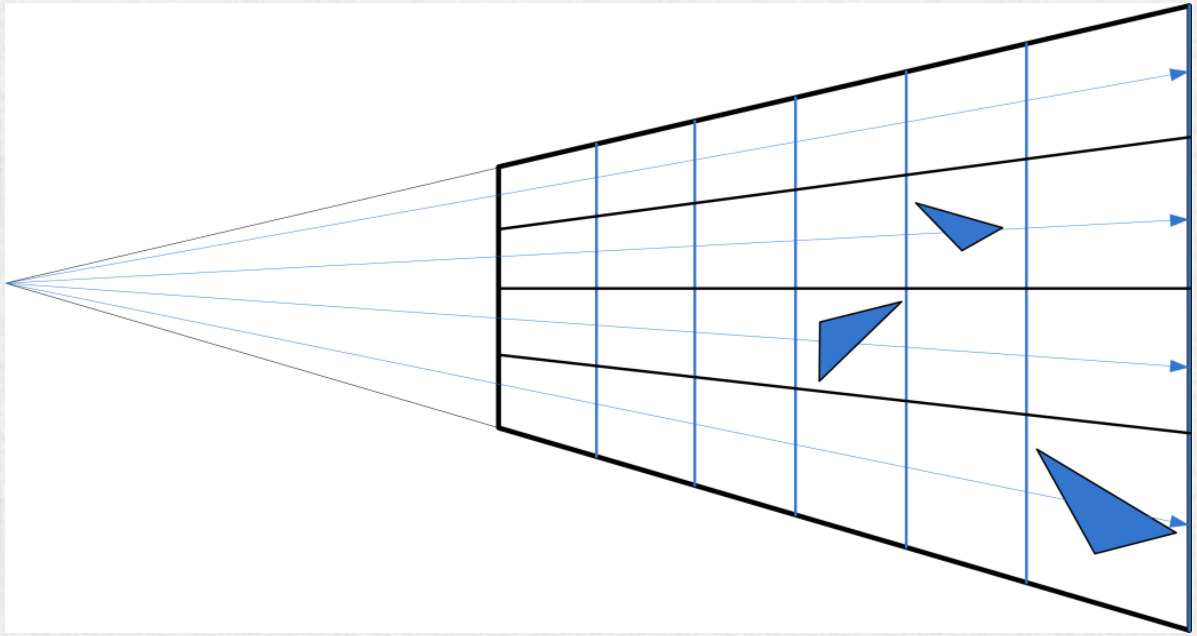
- Another option is to perturb the starting point of the shadow ray (typically in the normal direction), e.g. from $R(t_{int})$ to $R(t_{int}) + \epsilon\widehat{N}$
- The light direction needs to be modified, to go from the light to $R(t_{int}) + \epsilon\widehat{N}$
- The new shadow ray is $S(t) = \left(R(t_{int}) + \epsilon\widehat{N}\right) - \hat{L}_{mod}t$ where $t \in [0, t_{light})$
- Need to be careful that the new starting point isn't inside (or too close to) any other geometry



Aside: Code Acceleration

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





Aside: Code Acceleration

- 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