TA 10

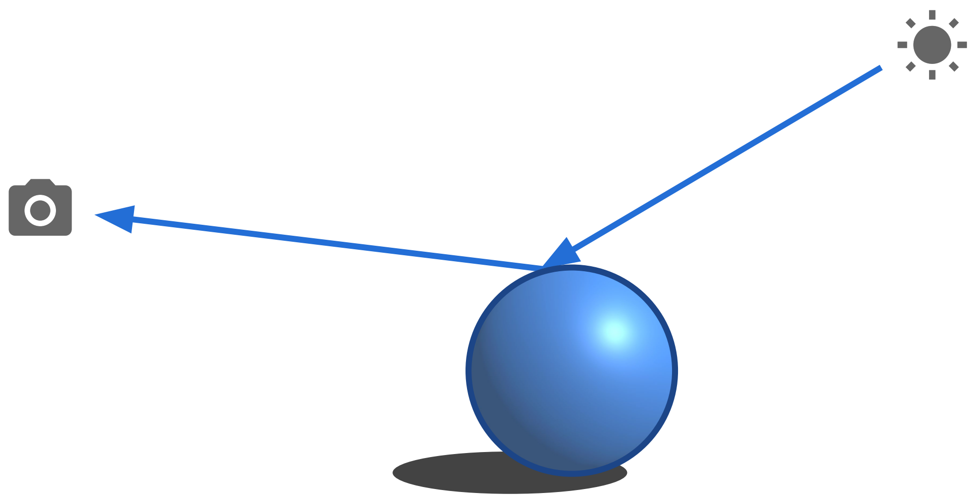
- The Ray Tracing Algorithm
- Aliasing
- Acceleration Structures

Ray Tracing

Computer Graphics 2020

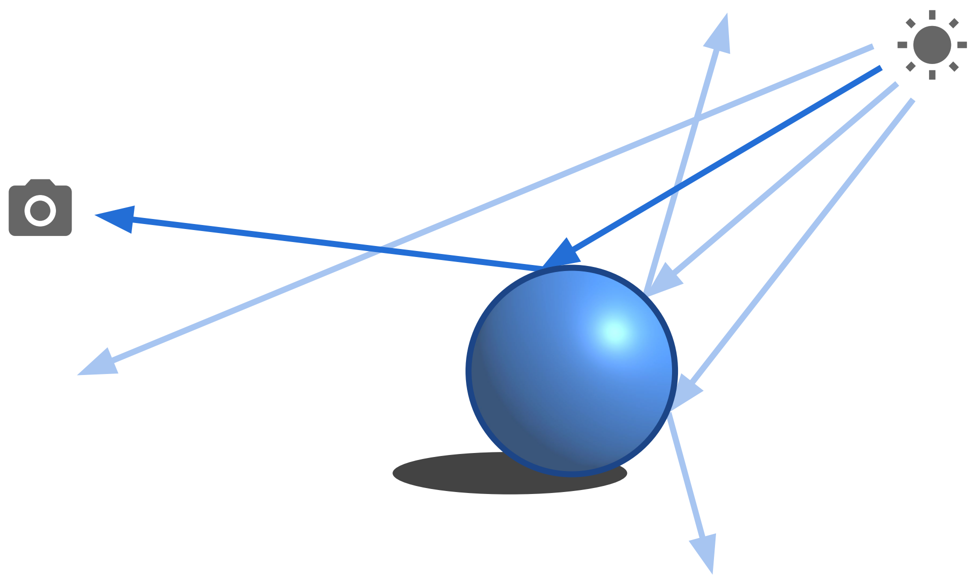
Ray Tracing

- In the real world, light travels in *rays* from a light source, hits objects and bounces around until it finally enters our eyes / a camera



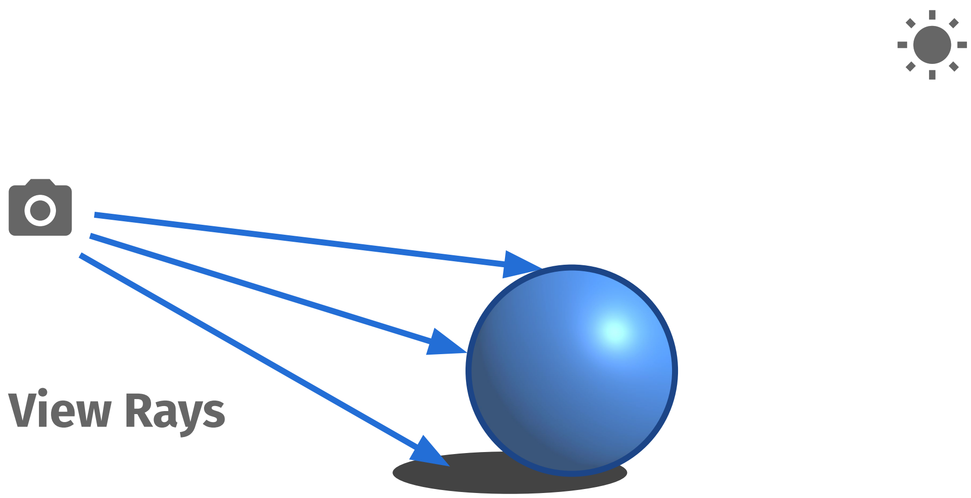
Ray Tracing

- If we *trace* each ray, bouncing around the scene until it hits the camera, we can create an image
- But only a tiny fraction of the rays actually reach our camera!



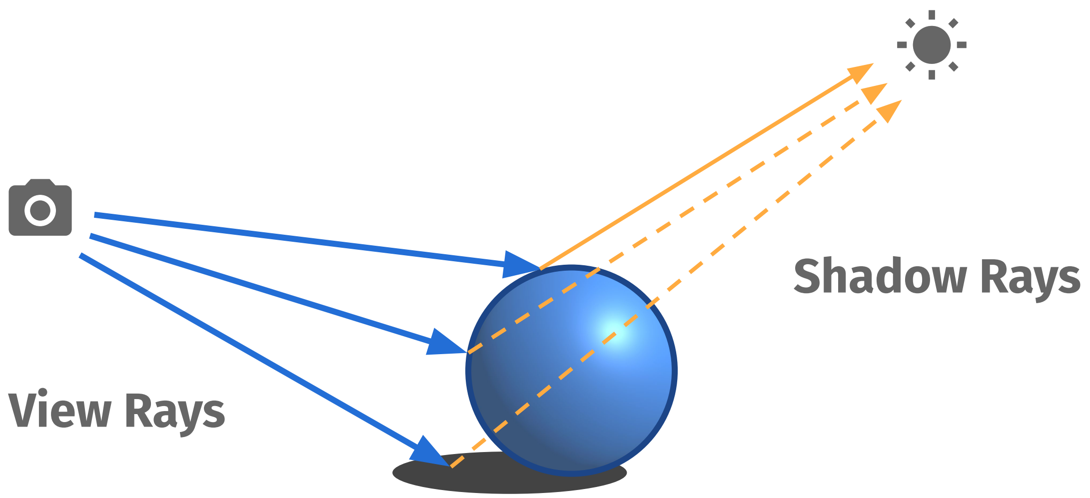
Ray Tracing

- Instead of shooting rays from light sources, we can *cast* rays from the camera into the scene, and trace them until they hit something



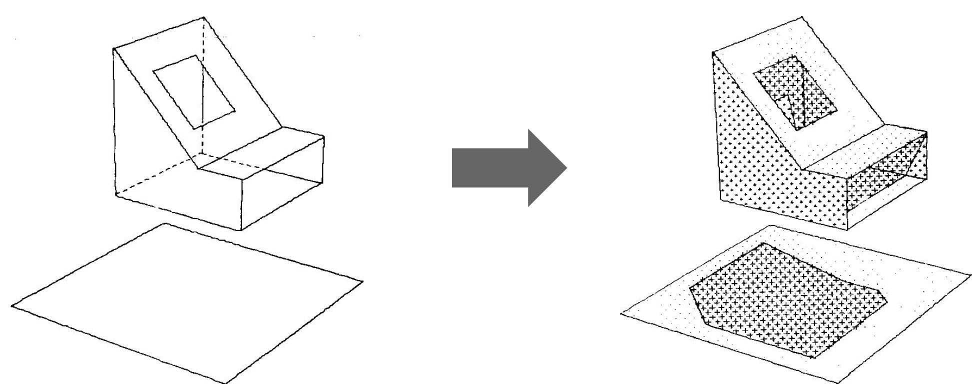
Ray Tracing

- We can then shoot another ray from the hit point towards the light source and check for collisions to figure out if the point is in light or in shadow



Ray Tracing

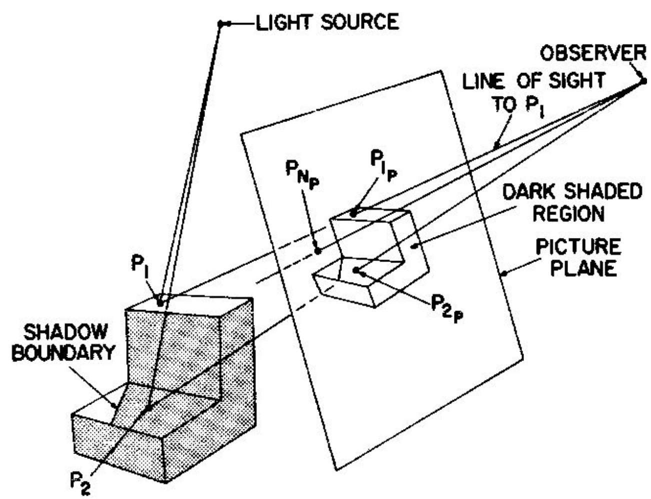
- This idea is called ***Ray Tracing***
- The first ray tracing algorithm used for rendering was presented by Arthur Appel in 1968



Figures from Appel's 1968 paper

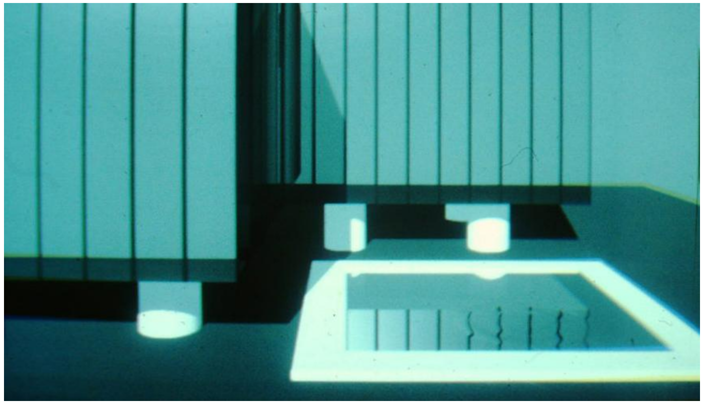
Ray Tracing

- Appel traced rays toward light sources to draw shadows
- At the time, the output device was not a screen but a pen plotter!



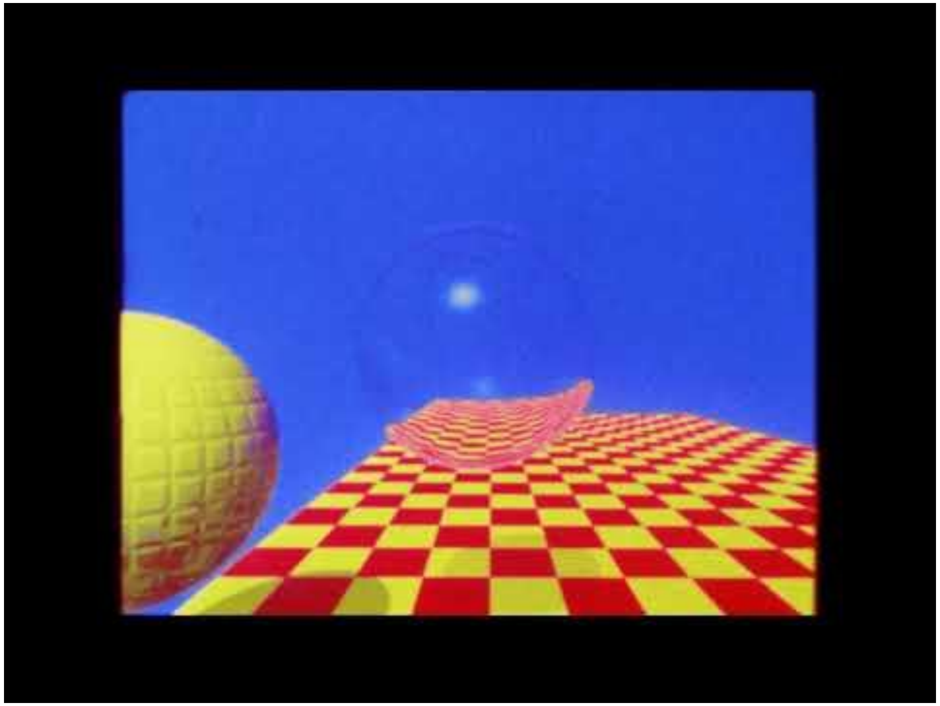
Whitted Ray Tracing

- In 1980, J. Turner Whitted published a seminal paper describing various algorithms and techniques to render 3D scenes using ray tracing
- Many of the methods and ideas from the paper are still in use today



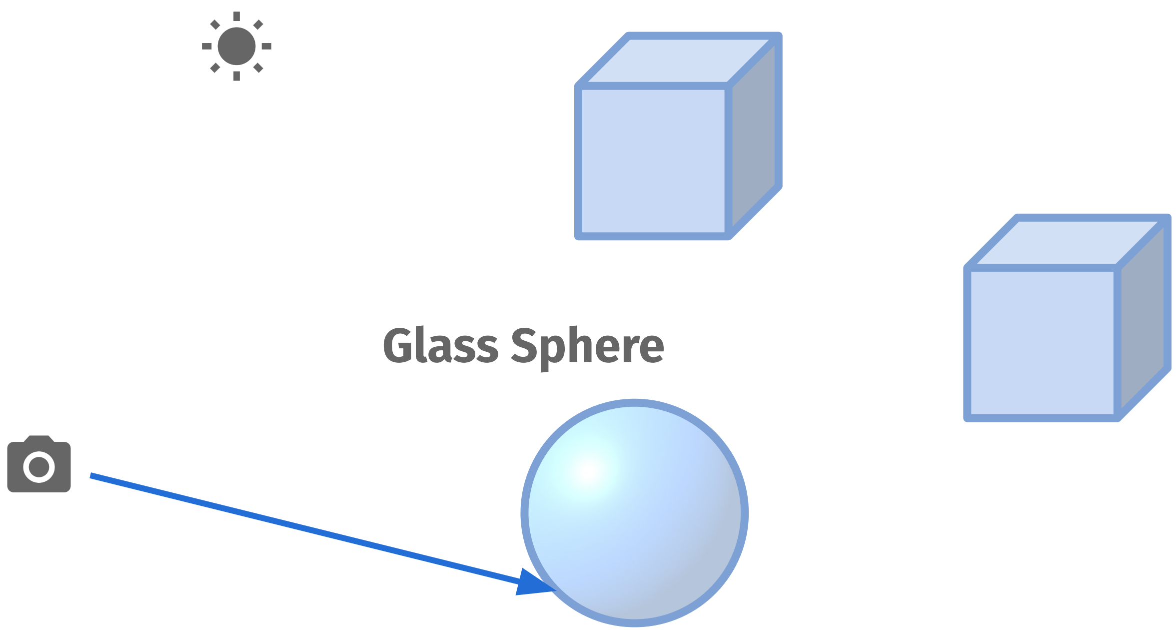
Whitted Ray Tracing

- Whitted's idea was to recursively cast rays from each hit point, creating effects like shadows, reflections and refractions:



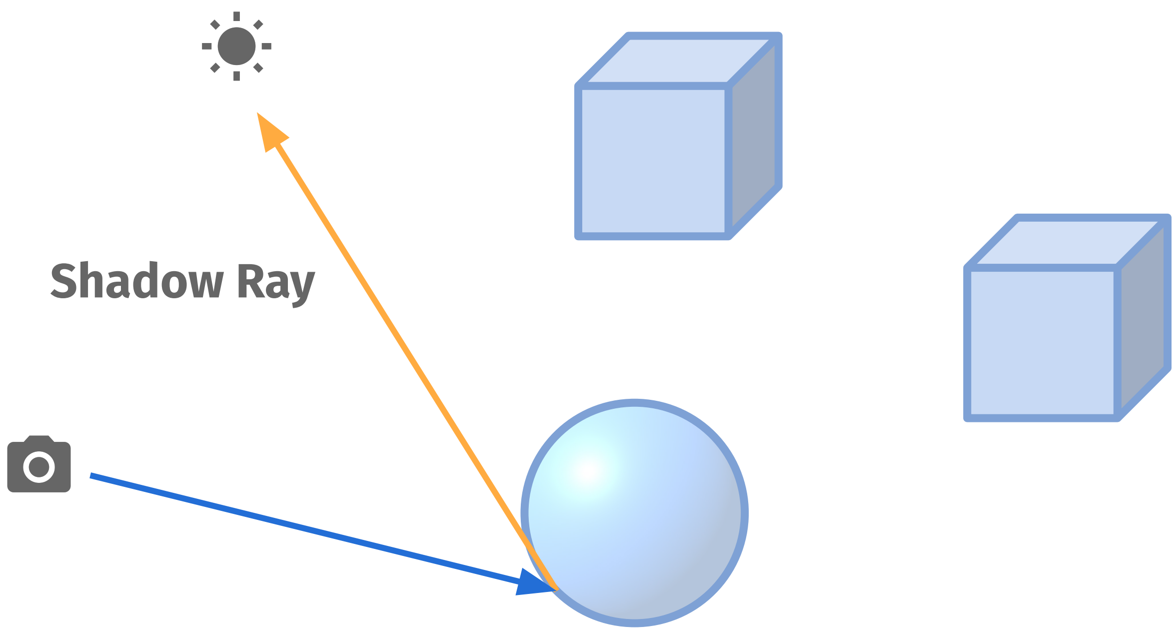
Whitted Ray Tracing

- Cast a ray from the camera and find the first hit point



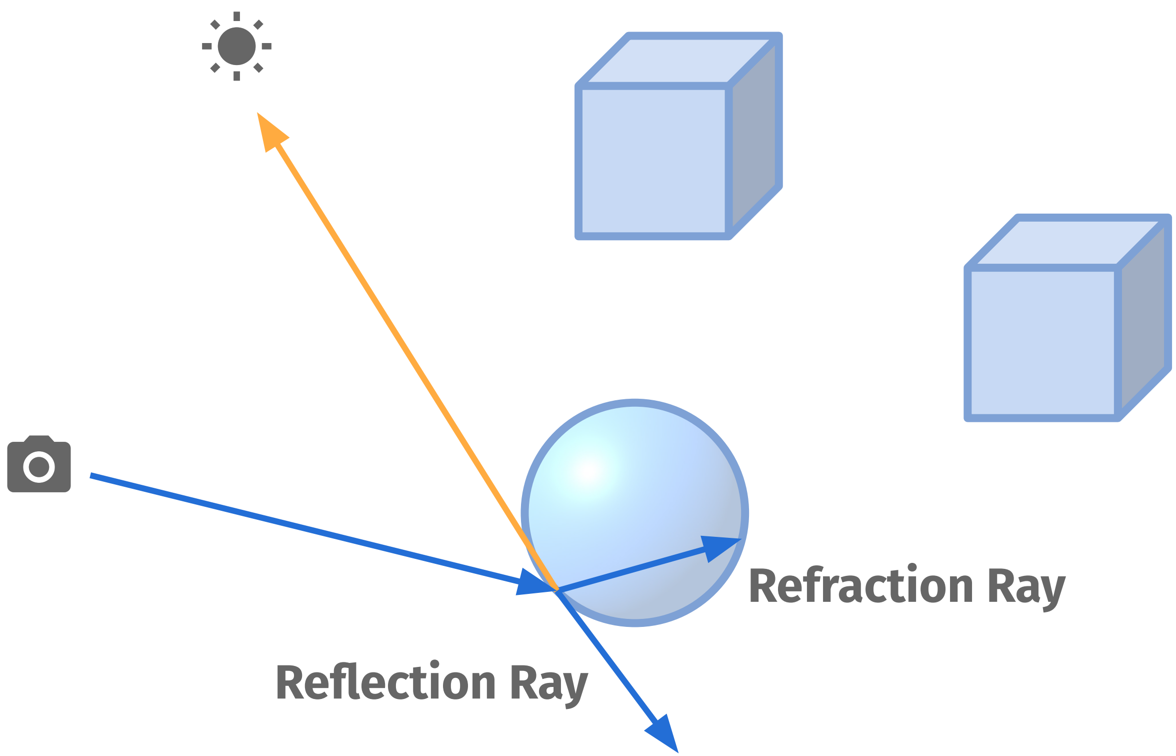
Whitted Ray Tracing

- Then cast a shadow ray from the hit point towards the light



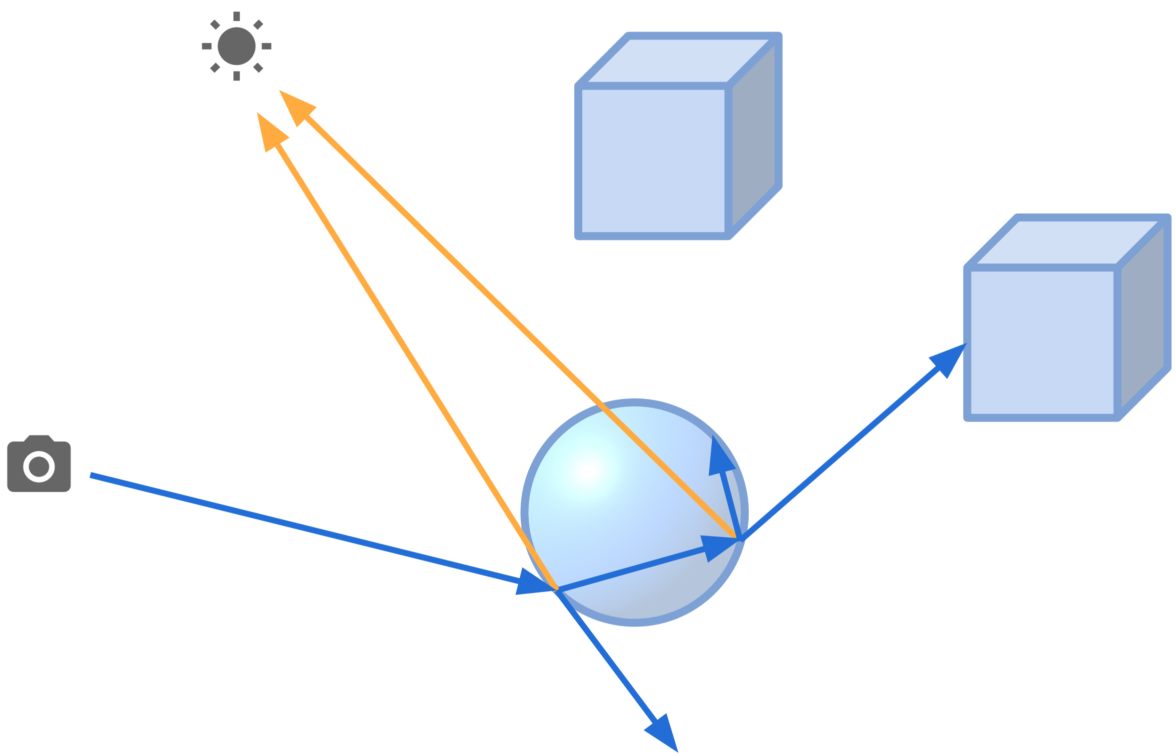
Whitted Ray Tracing

- Calculate the reflection and refraction angles, then cast rays in those directions



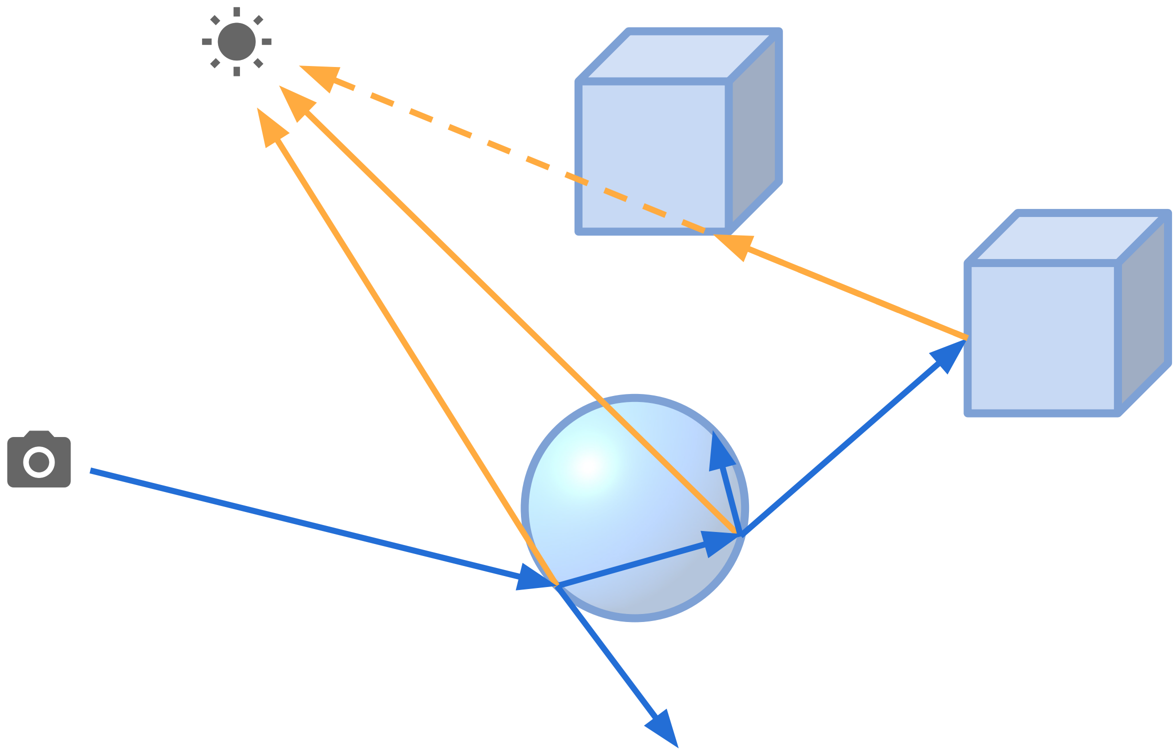
Whitted Ray Tracing

- Following the refraction ray, cast more shadow, refraction and reflection rays



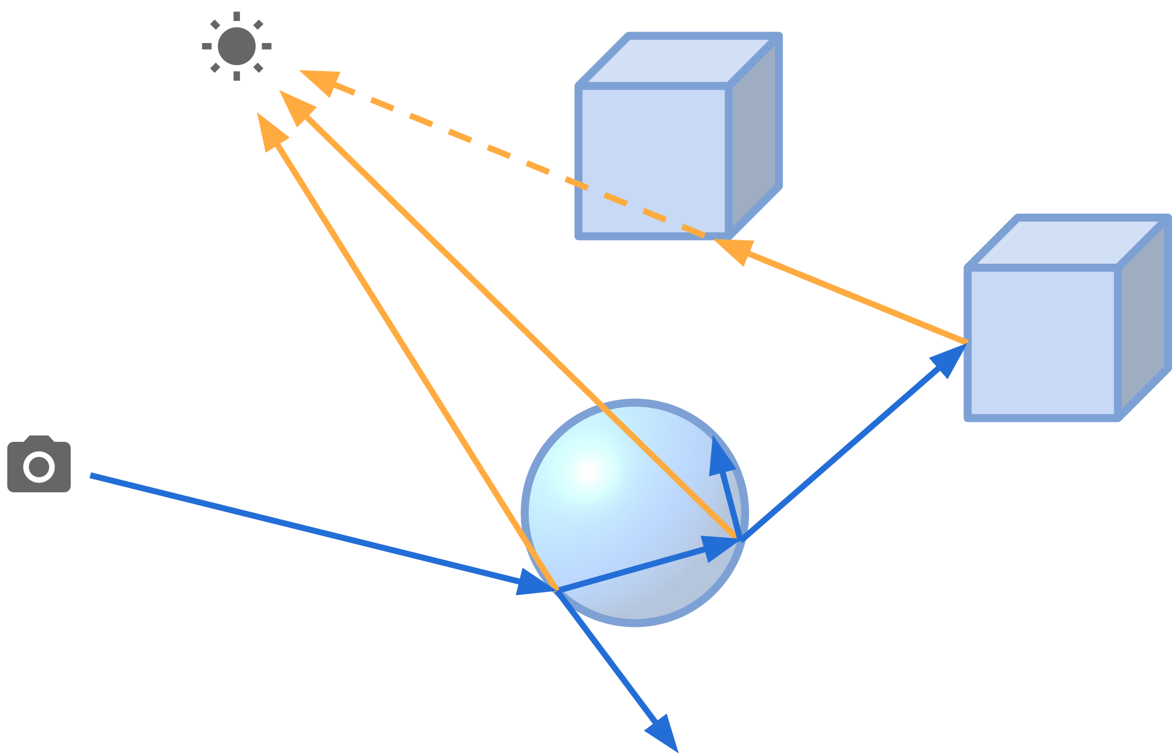
Whitted Ray Tracing

- We continue tracing rays recursively until we reach a specified bounce limit



Whitted Ray Tracing

- Combining the contributions of all these rays we can calculate the final pixel color



Ray Tracing Algorithm

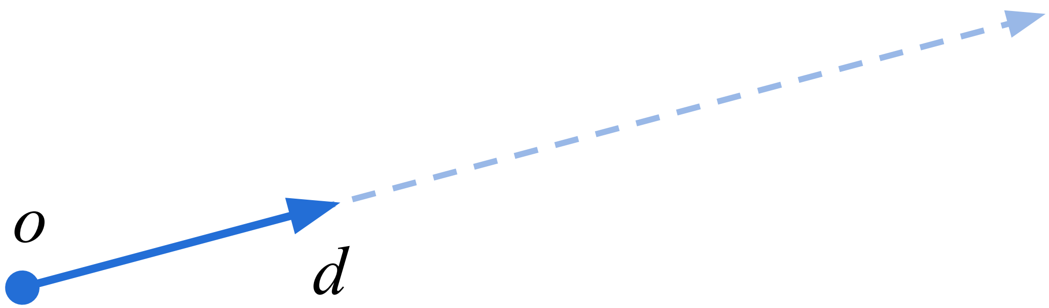
- For each pixel:
    1. Determine ray direction
    2. Intersect ray with the scene
    3. Compute lighting & shading
    4. Set pixel color

Ray Tracing Algorithm

- For each pixel:
    1. **Determine ray direction**
    2. Intersect ray with the scene
    3. Compute lighting & shading
    4. Set pixel color

Ray

- A ***Ray*** is defined by 2 parameters:
  - An origin point $o = (o_x, o_y, o_z)$
  - A direction vector $d = (d_x, d_y, d_z)$

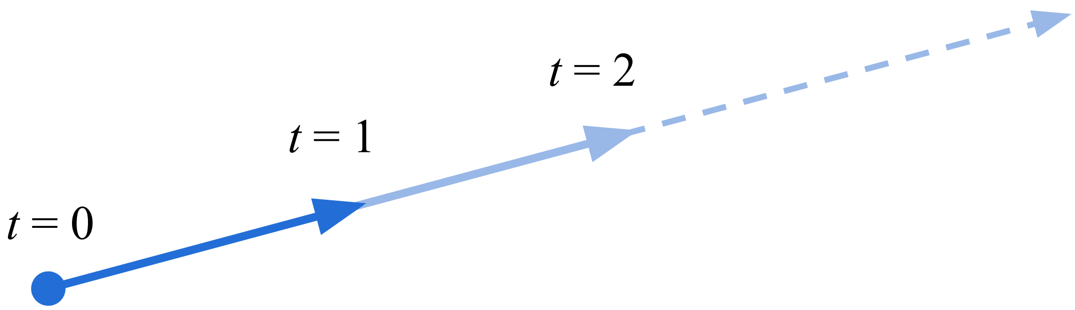


Ray

- Parametric representation:

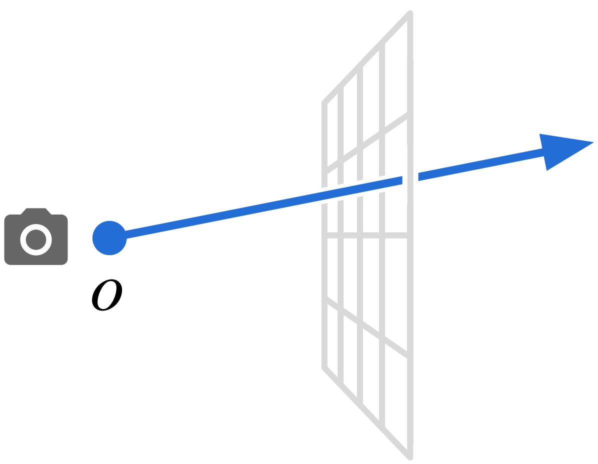
$$ray(t) = o + d\,t$$

$$t > 0$$



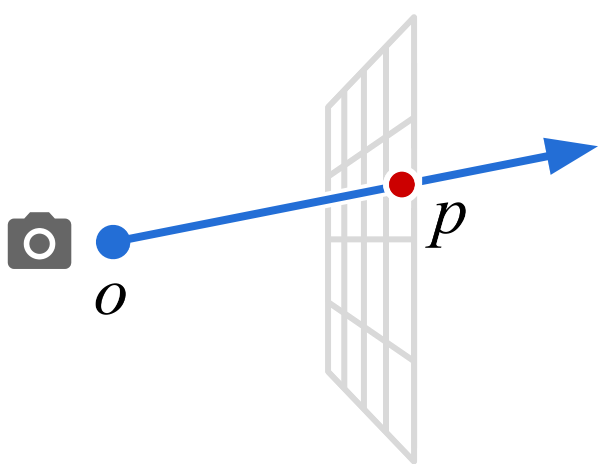
Generating View Rays

- We need to generate a ray for each pixel in the final image, i.e. find its origin & direction
- When using a perspective camera, the origin o will be the camera position



Generating View Rays

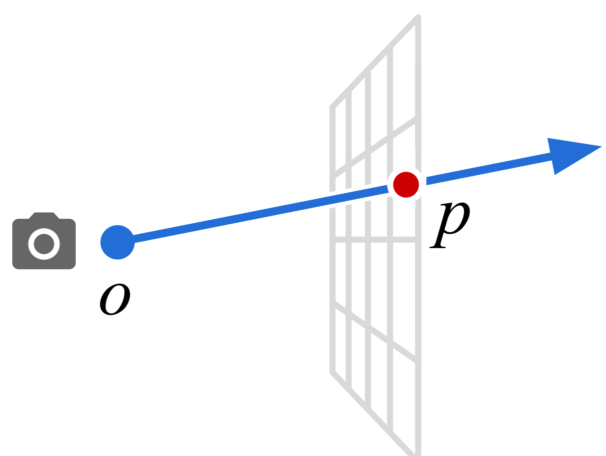
- Transform a pixel location (x, y) to coordinates on our image plane $p \in [-1, 1]^2$
- We get a direction vector $(p - o) \,/\, \|(p - o)\|$



Generating View Rays

- Remember we must transform both p and q to world-space
- The view ray for each pixel is

$$ray(t) = o + t\,(p - o) \,/\, \|(p - o)\|$$

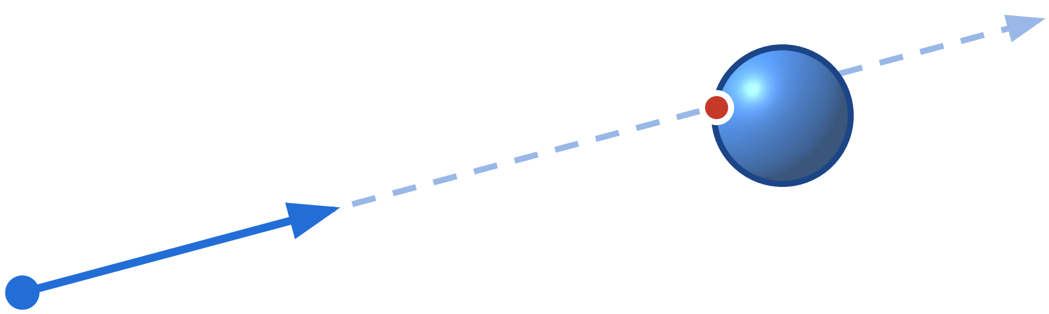


Ray Tracing Algorithm

- For each pixel:
    1. Determine ray direction
    2. **Intersect ray with the scene**
    3. Compute lighting & shading
    4. Set pixel color

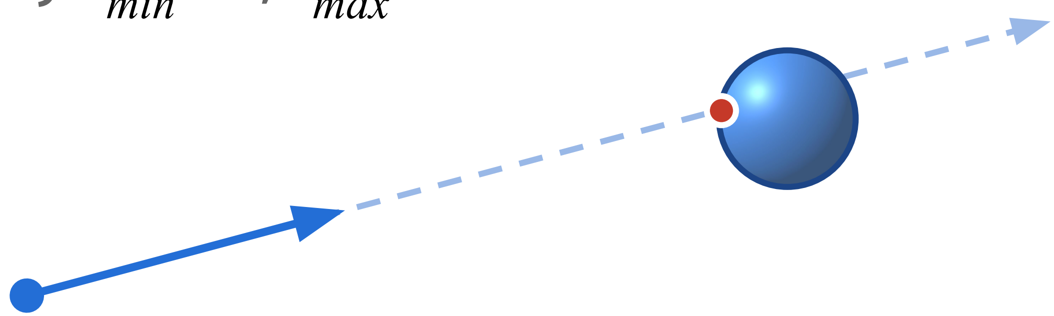
Ray Casting

- ***Ray Casting*** is the process of checking for intersections along a ray
- we “shoot” the ray and check what gets hit
- Useful in general, e.g. collision detection



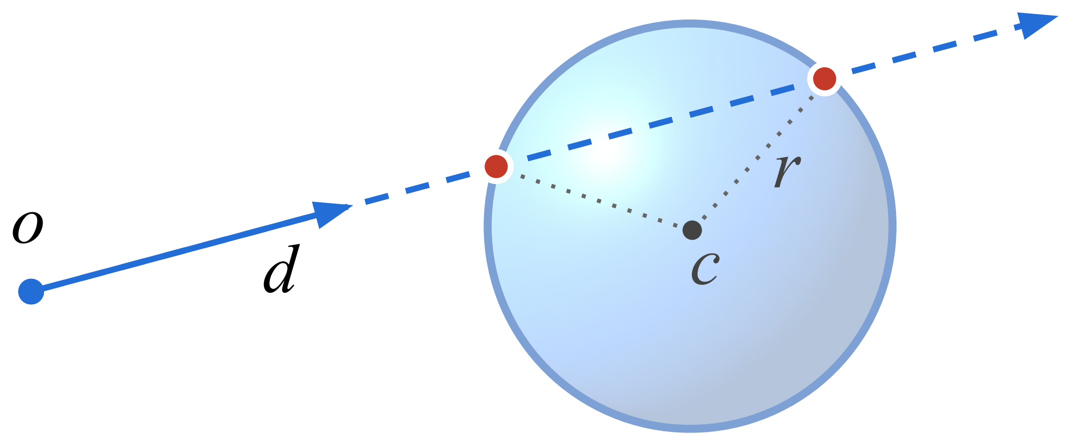
Ray-Object Intersection

- Given an implicit representation of some object $f(p) = 0$, we want to find $t \in (t_{min}, t_{max})$ s.t. $f(r(t)) = 0$
- Finding intersection points is essentially finding the roots of this equation
- Generally $t_{min} = \varepsilon$, $t_{max} = \infty$



Ray-Sphere Intersection

- We want to find the intersection of a ray with a sphere centered at point c with radius r:



Ray-Sphere Intersection

- To find the intersection we need to meet 2 conditions:

1. The point is on the ray:

$$ray(t) = o + d\,t$$

2. The point is on the sphere:

$$||p - c|| = r \Leftrightarrow (p - c)\cdot(p - c) - r^2 = 0$$

Ray-Sphere Intersection

- Under these 2 conditions we get:

$$0 = (o + d\,t - c)\cdot(o + d\,t - c) - r^2 =$$

$$d\cdot d\,t^2 + 2(o - c)\cdot d\,t + (o - c)\cdot(o - c) - r^2$$

- This is a quadratic equation in relation to t :

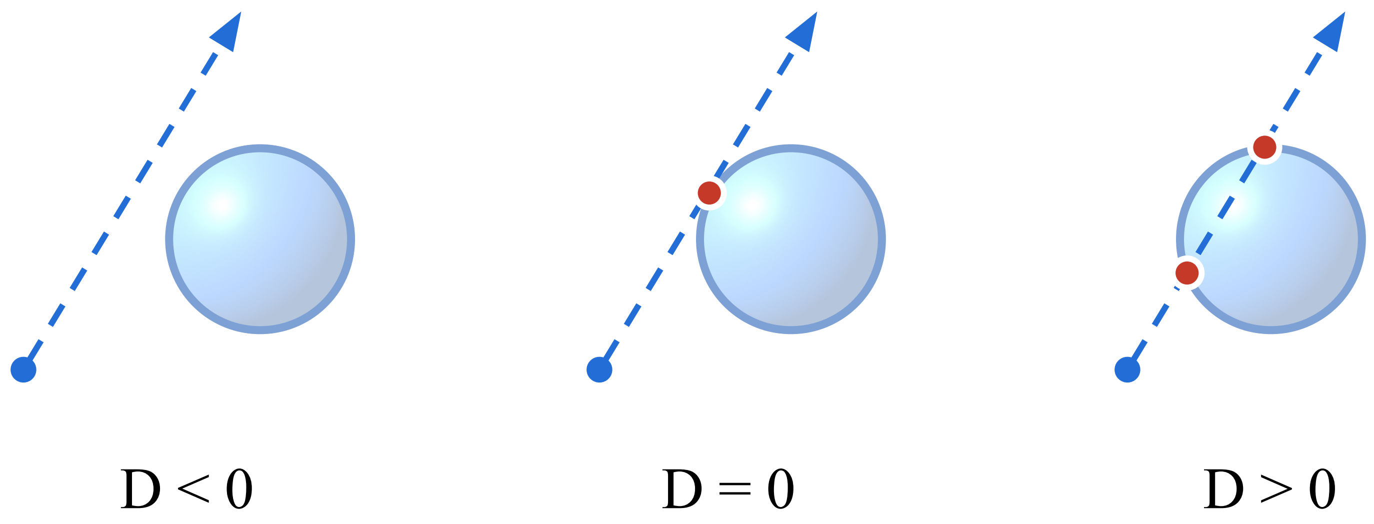
$$\mathrm{A} = d\cdot d = 1 \quad \mathrm{B} = 2(o - c)\cdot d \quad \mathrm{C} = (o - c)\cdot(o - c) - r^2$$

- We know how to find the roots:

$$t_{0,1} = (-\,\mathrm{B} \pm \sqrt{\mathrm{B}^2 - 4\mathrm{AC}}\,)\,/\,2\mathrm{A}$$

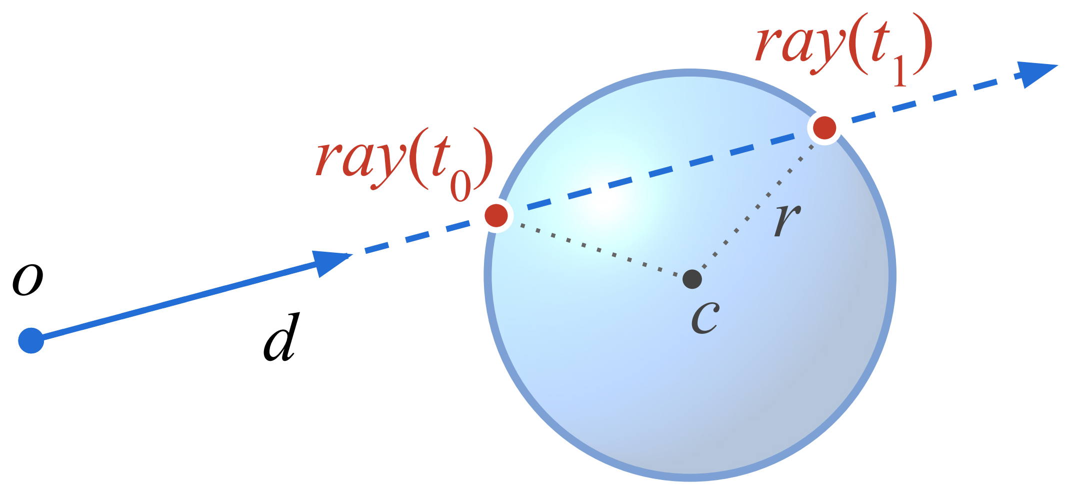
Ray-Sphere Intersection

- We get 0, 1 or 2 solutions, according to the discriminant $D = B^2 - 4AC$:



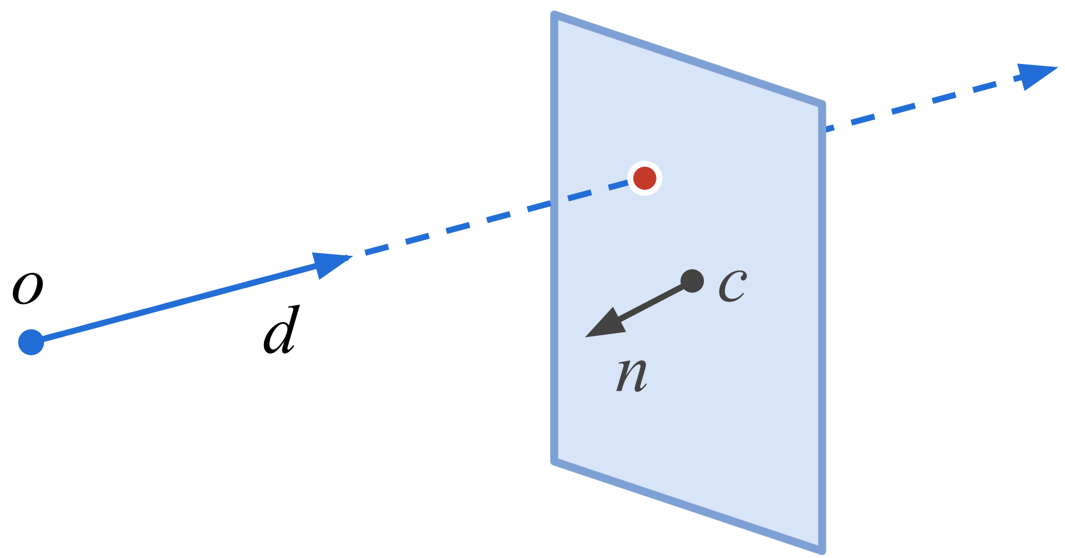
Ray-Sphere Intersection

- If we get 2 solutions - pick the smaller one to get the first hit point
- Remember to make sure $t > 0$



Ray-Plane Intersection

- We want to find the intersection of a ray with a plane going through point c with normal n :



Ray-Plane Intersection

- To find the intersection we need to meet 2 conditions:

1. The point is on the ray:

$$ray(t) = o + d\,t$$

2. The point is on the plane:

$$(p - c) \cdot n = 0$$

Ray-Plane Intersection

- Under these 2 conditions we get:

$$0 = (ray(t) - c) \cdot n = (o + d\,t - c) \cdot n$$

$$= (o - c) \cdot n + (d \cdot n)t$$

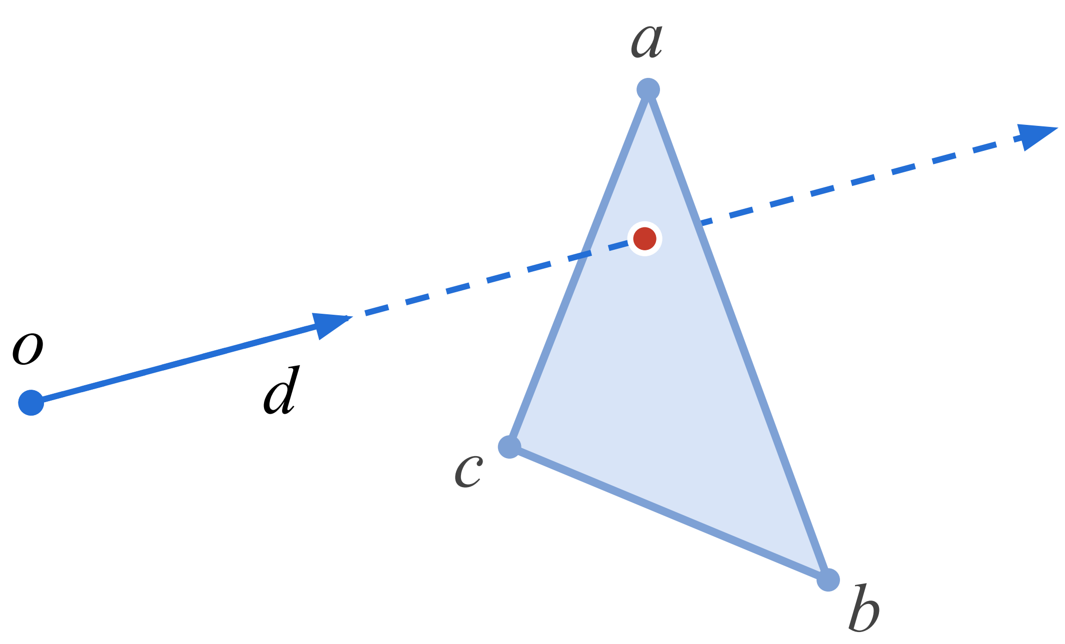
- Assuming $d \cdot n \neq 0$ we get one solution:

$$t = \frac{-(o - c) \cdot n}{d \cdot n}$$

- $ray(t)$ is our intersection point

Ray-Triangle Intersection

- We want to find the intersection of a ray with a triangle (a, b, c) :

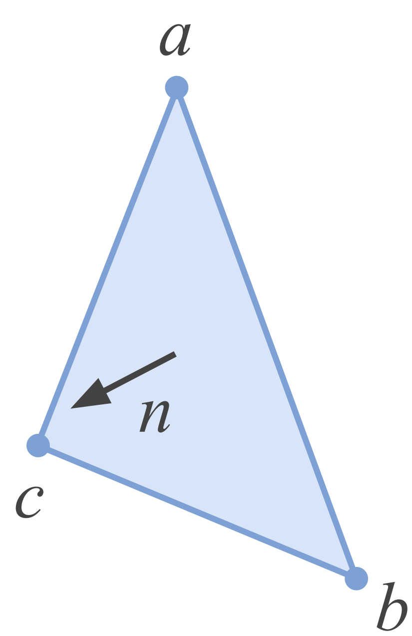


Ray-Plane Intersection

- We can divide the problem into 2 parts:
  1. Check if the ray intersects the plane on which the triangle (a, b, c) lies
  2. Assuming we found an intersection point p, check if p the falls inside or outside the triangle (a, b, c)

Ray-Triangle Intersection

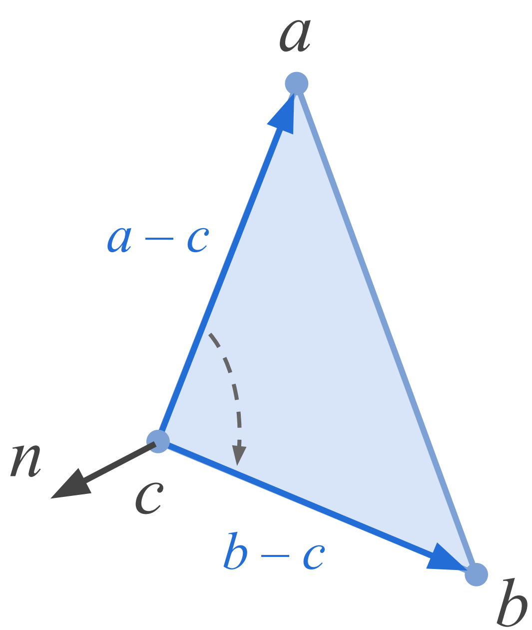
- To check the ray-triangle intersection, we must find the normal direction n



Ray-Triangle Intersection

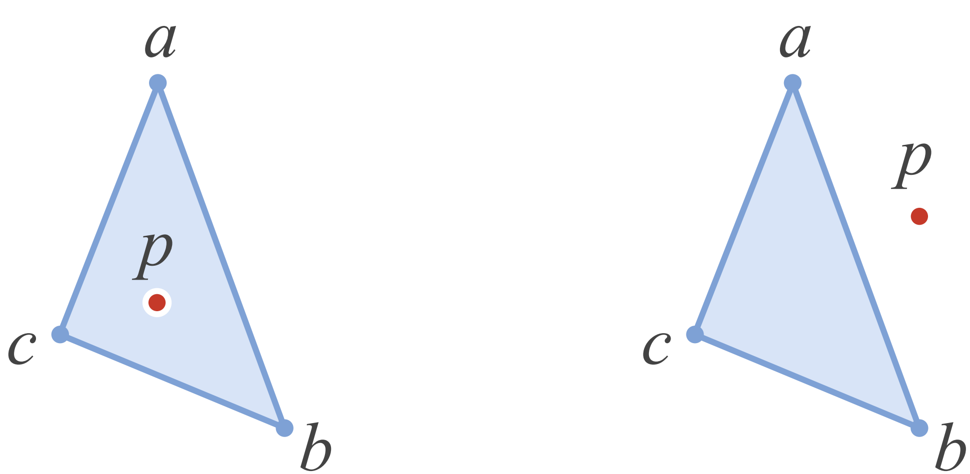
- We can calculate using the cross product:

$$n = (a - c)\times(b - c) \;/\; \|(a - c)\times(b - c)\|$$



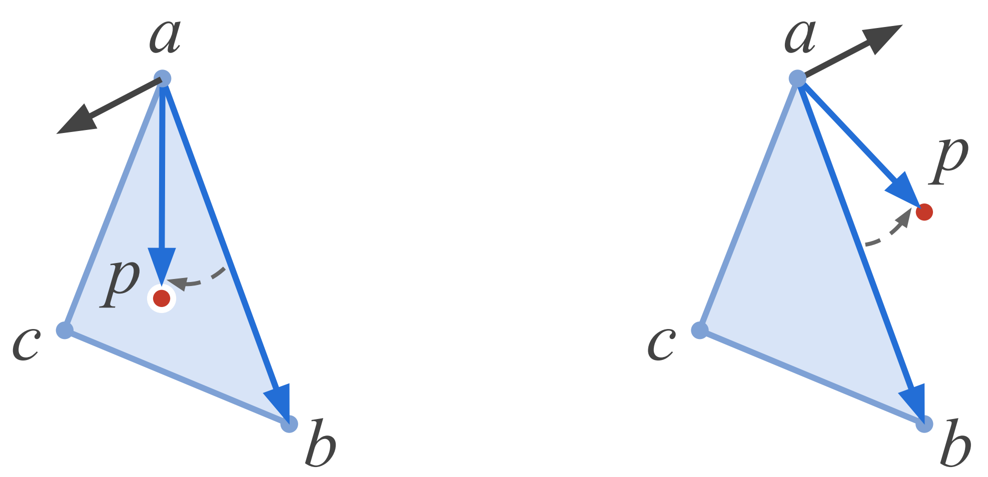
Ray-Triangle Intersection

- Assume we found an intersection point p, how do we check if p the falls inside the triangle?
- We can use cross products to determine which side of each edge p is on



Ray-Triangle Intersection

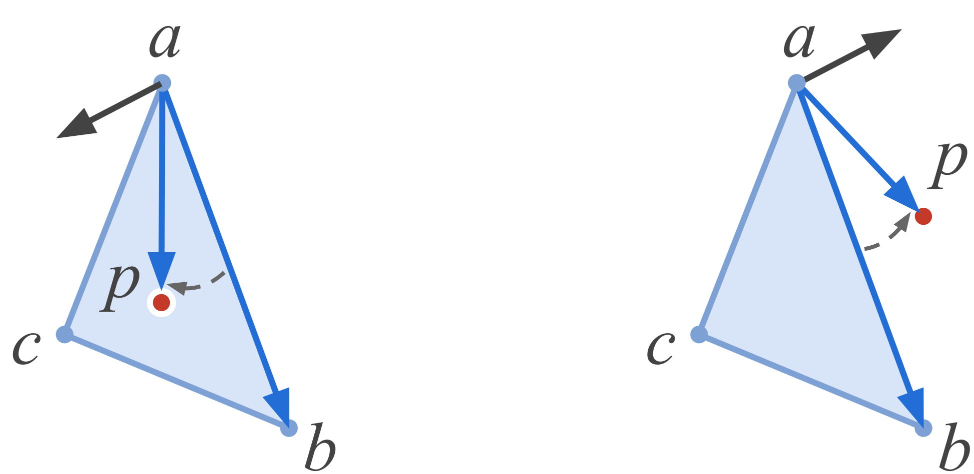
- For example, look at vectors $b - a$ and $p - a$
- Their cross product $(b-a)\times(p-a)$ in the same direction as the normal vector only if p is inside of the edge ab:



Ray-Triangle Intersection

- In other words, p is inside of the edge ab if the following holds:

$$(b-a)\times(p-a)\cdot n \geq 0$$



Ray-Triangle Intersection

- Meaning that p is inside of the triangle (a, b, c) if all these conditions are met:

$$(b-a)\times(p-a)\cdot n \geq 0$$

$$(c-b)\times(p-b)\cdot n \geq 0$$

$$(a-c)\times(p-c)\cdot n \geq 0$$

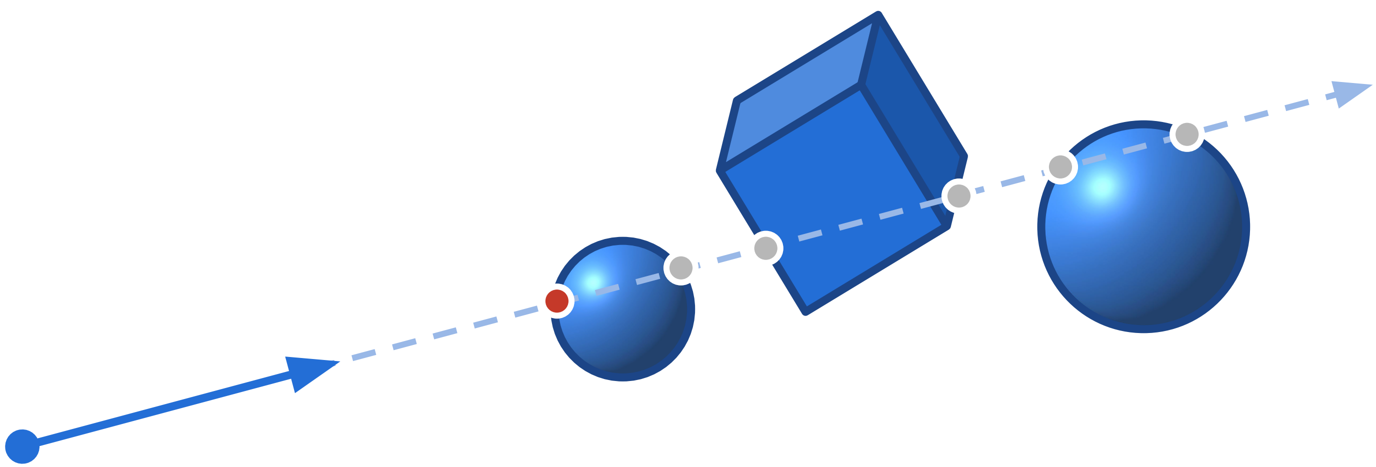
- If any one of these tests fails, then p is not inside the triangle and the result is no intersection

Ray-Triangle Intersection

- Using ray-triangle intersections we can render triangular 3D meshes with ray-tracing!
- Note that this is not the most efficient way to compute ray-triangle intersections
- A lot of research went into this problem in the early days of ray-tracing

Ray-Scene Intersections

- For each camera ray we must check intersections with each object in the scene
- Finally we save only the closest intersection - this is our “Best Hit”



Ray Tracing Algorithm

- For each pixel:
    1. Determine ray direction
    2. Intersect ray with the scene
    3. **Compute lighting & shading**
    4. Set pixel color

Lighting

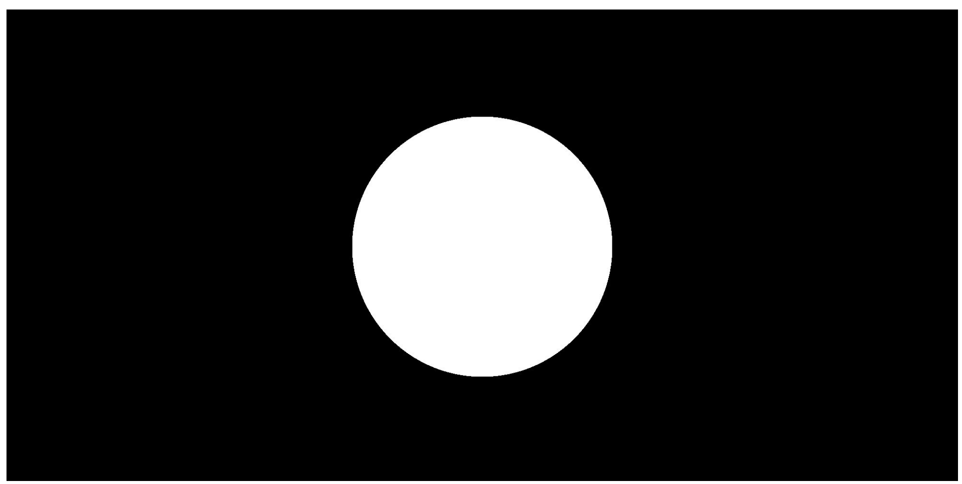
- Let $ray(t) = o + d\,t$ be a view ray, cast from the camera position o at direction d
- We'll also define e to be the *energy* of the ray, for now think of it as $e = 1$
- The function trace will return the energy or color c associated with the ray, shot into our scene:

$$c = \mathrm{trace}(o, d, e) = \begin{cases} ec_{\mathrm{hit}} & \text{ray hit point } p \\ ec_{\mathrm{miss}} & \text{ray missed all geometry} \end{cases}$$

Lighting

- The simplest shading - color the pixel white if its ray has hit something, black otherwise

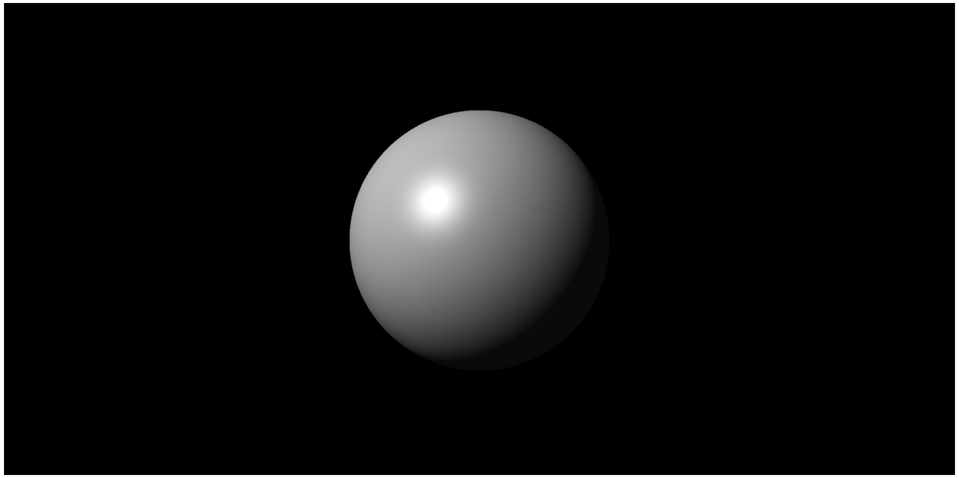
$$c_{\text{hit}} = 1 \qquad c_{\text{miss}} = 0$$



Lighting

- We can do better - use Blinn-Phong lighting to shade each hit point

$$c_{\text{hit}} = \text{BlinnPhong}(p, d) = \max(l \cdot n, 0) + \max(n \cdot h, 0)^{\sigma}$$



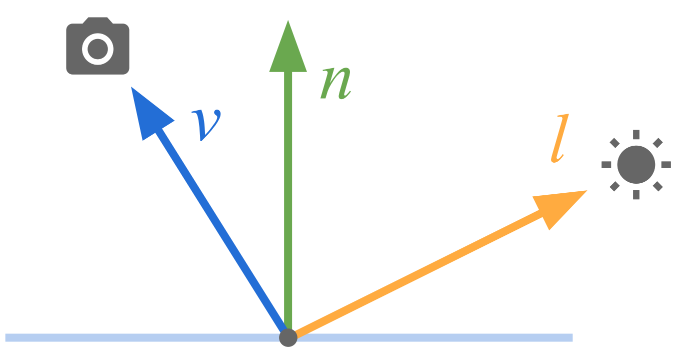
Lighting

- We need 3 direction vectors to use Blinn-Phong:

  l - Light direction - assume it is given

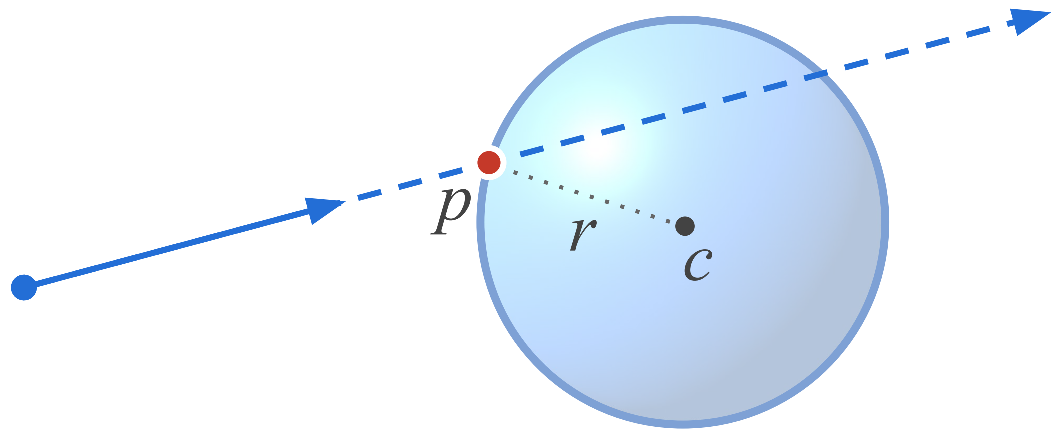
  v - View direction - given from the view ray $-d$

  n - Surface normal - we need to calculate the normal when we find the hit point p



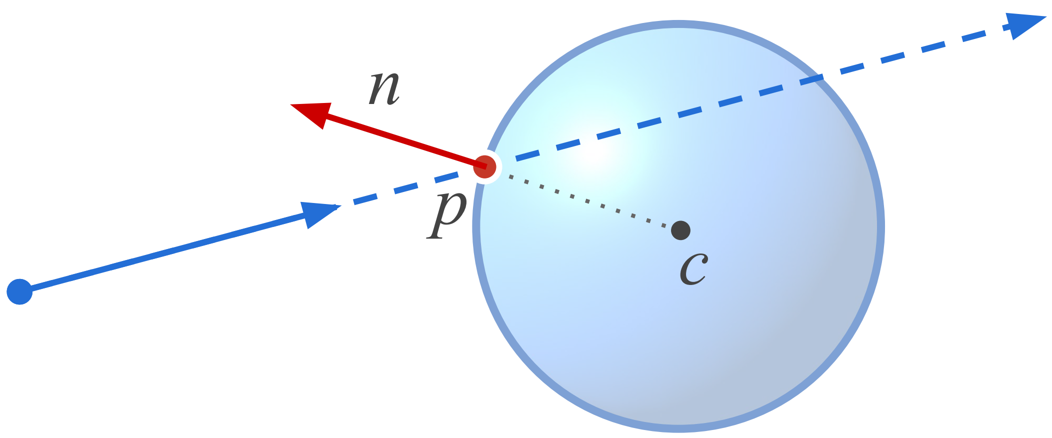
Lighting

- For example, assume we found the intersection point p with a sphere centered at c with radius r
- What is the surface normal n at this point?



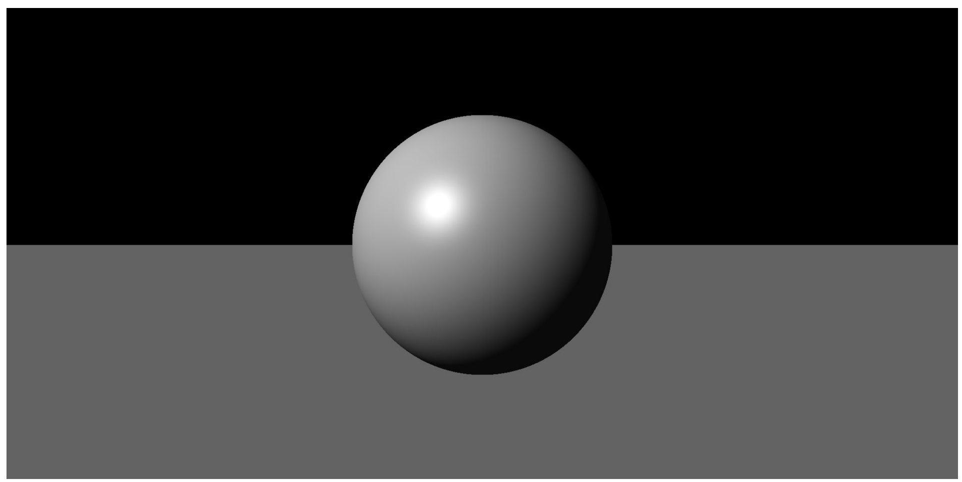
Lighting

- We can easily calculate it using the hit point and the center of the sphere $n = (p - c) \,/\, \|p - c\|$
- We save this surface normal as part of the hit



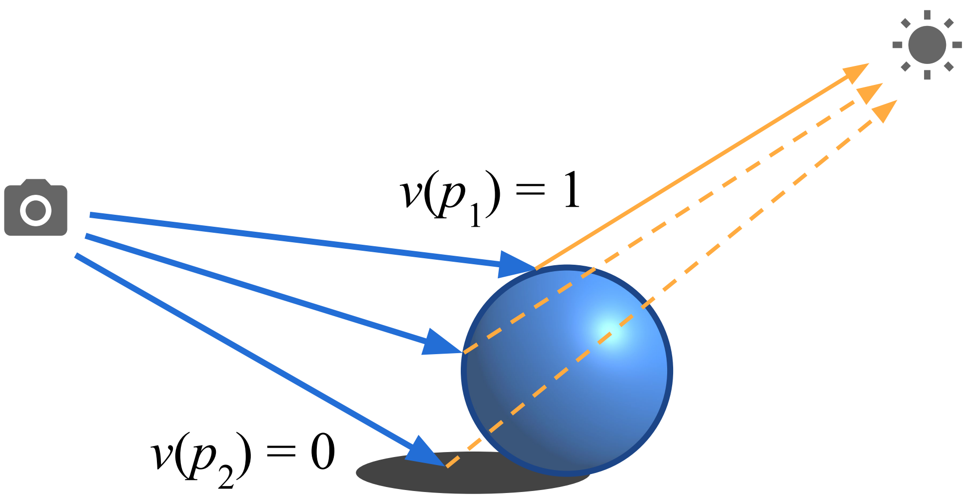
Ray Traced Shadows

- Let’s add a floor plane to our scene, so we can see cast shadows:



Ray Traced Shadows

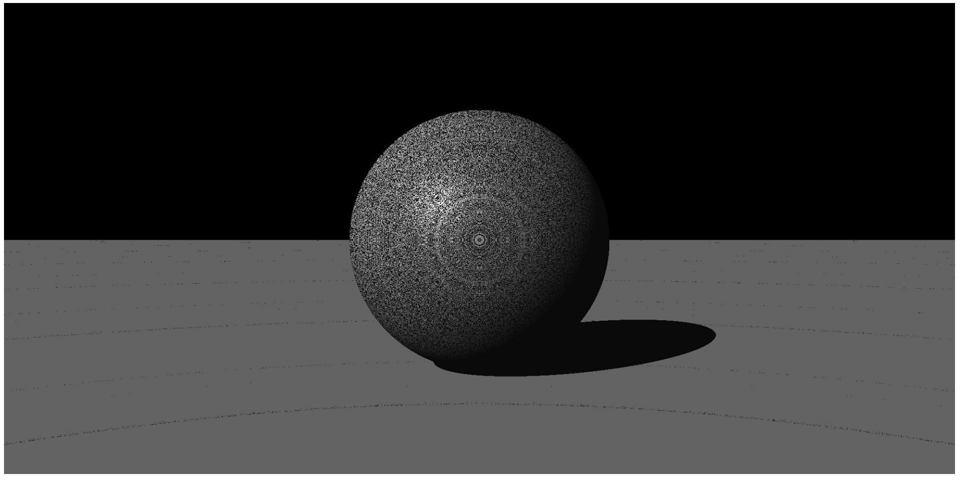
- From each hit point p we cast a *shadow ray* in the direction of the light to get a visibility term $v(p)$
- If we hit something, the point is in shadow and so $v(p) = 0$, otherwise $v(p) = 1$



Ray Traced Shadows

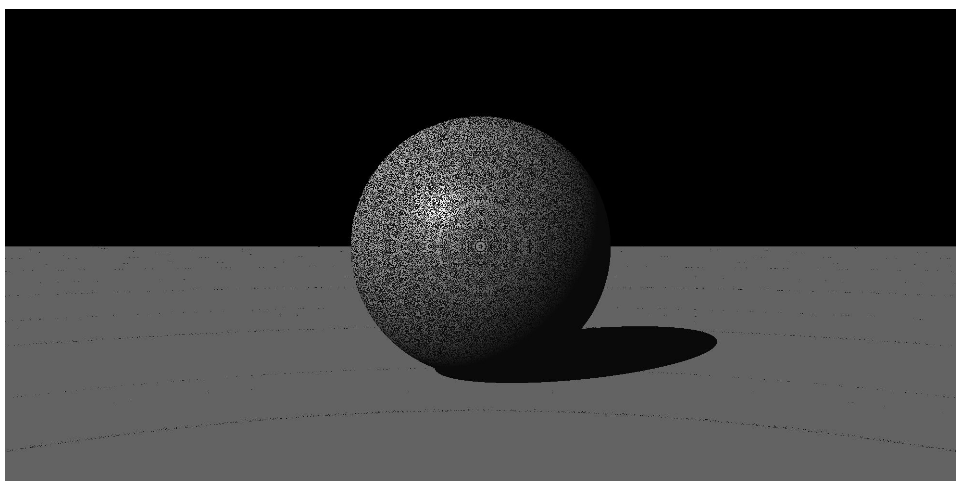
- When we try to use this, we get ugly artefacts known as “shadow acne”:

$$c_{\text{hit}} = v(p)\text{BlinnPhong}(p, d)$$



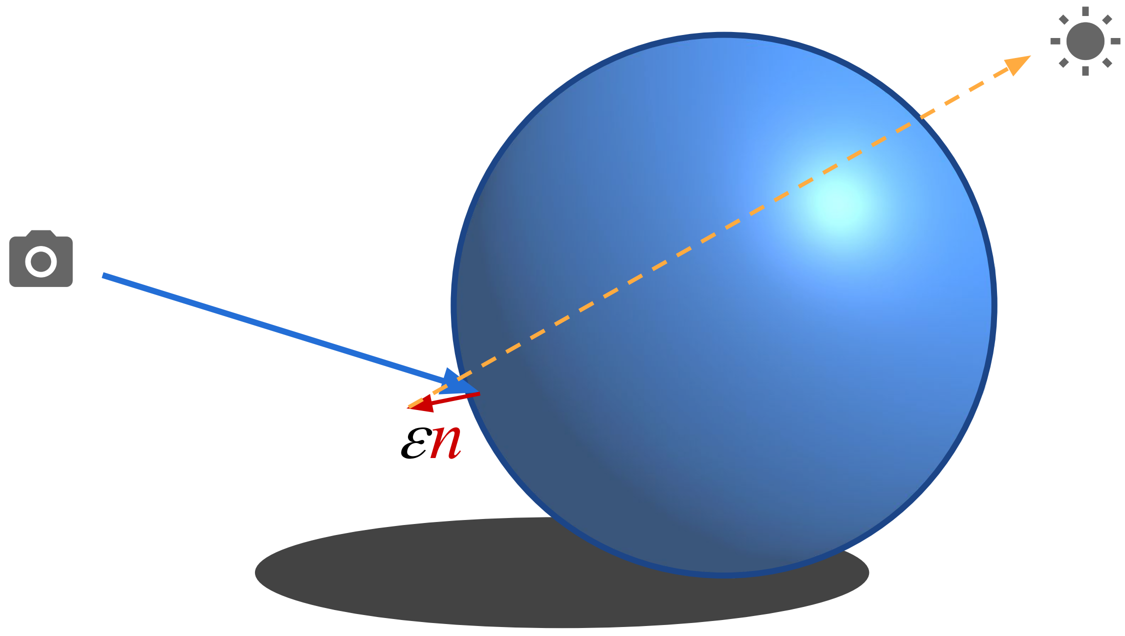
Ray Traced Shadows

- The shadow ray intersects with the surface it originated from!
- Numerical precision errors cause the “noisy” appearance



Ray Traced Shadows

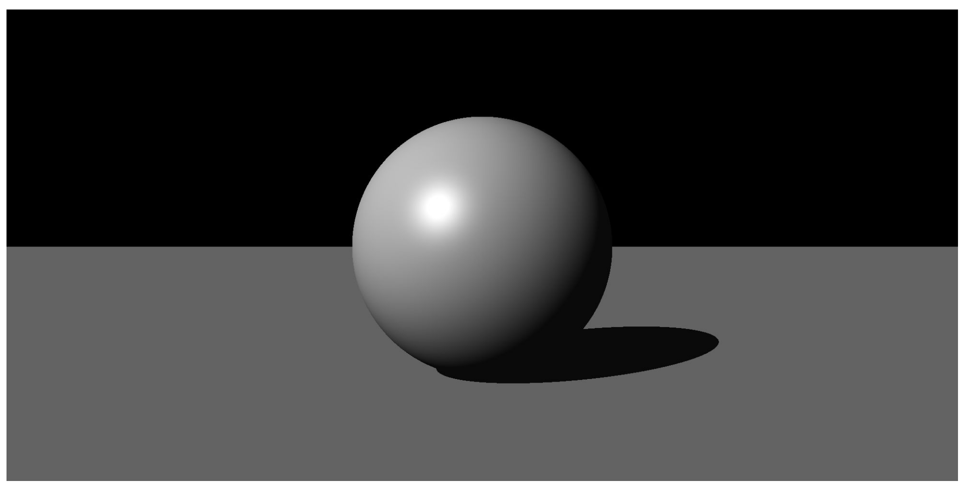
- The solution - offset by ε in the normal direction n before casting the shadow ray



Ray Traced Shadows

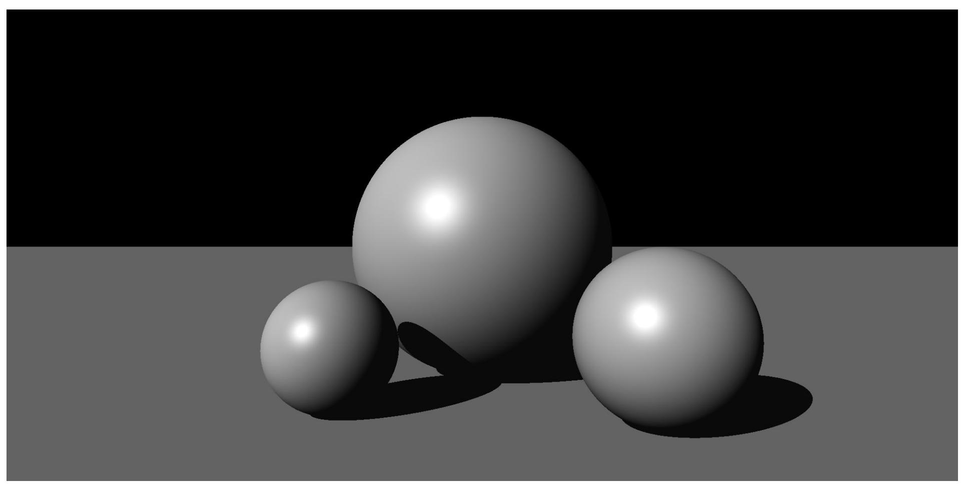
- No more shadow acne!
- Note that we get *binary shadows* with hard edges

$$c_{\text{hit}} = v(p+\varepsilon n)\text{BlinnPhong}(p, d)$$



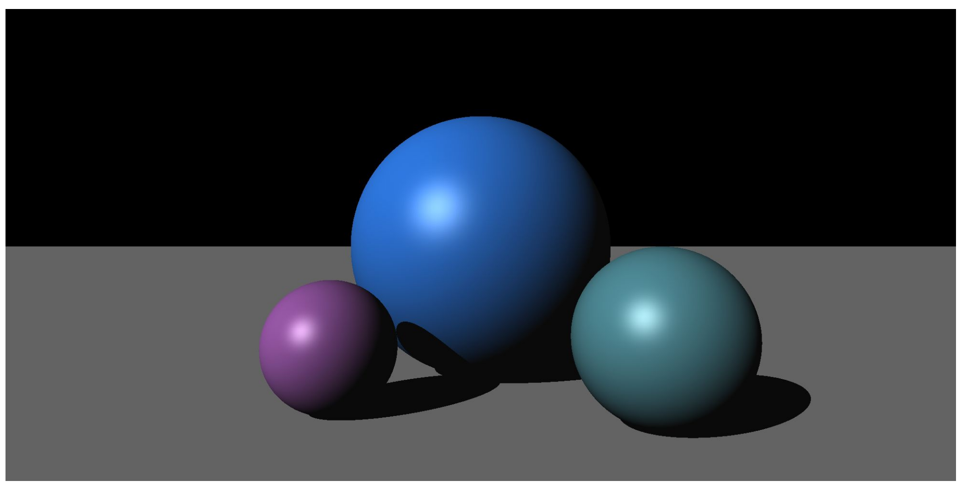
Materials

- Let’s add a few more spheres to our scene
- We can associate each sphere with a material that defines how it interacts with light rays



Materials

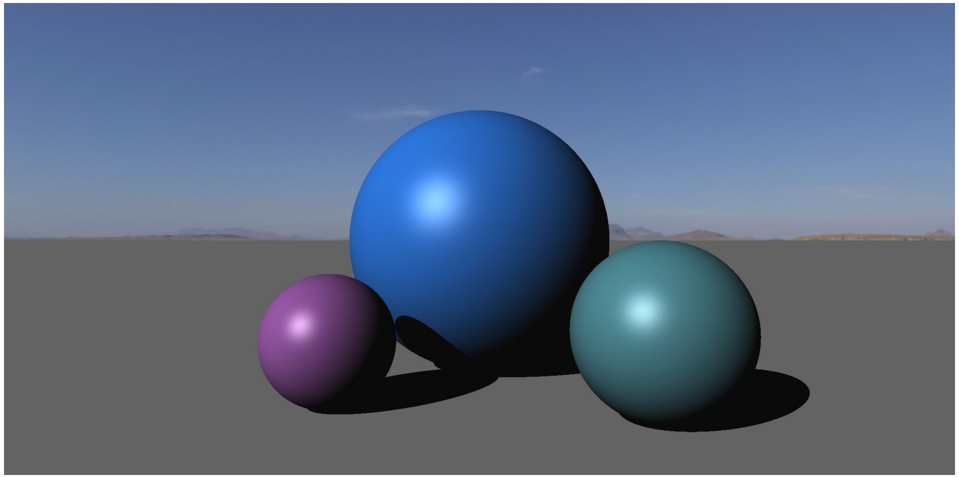
- At each ray hit we store the material properties and then use that for shading
- For example, we can give each sphere a different diffuse color k_d to use in Blinn-Phong



Environment Mapping

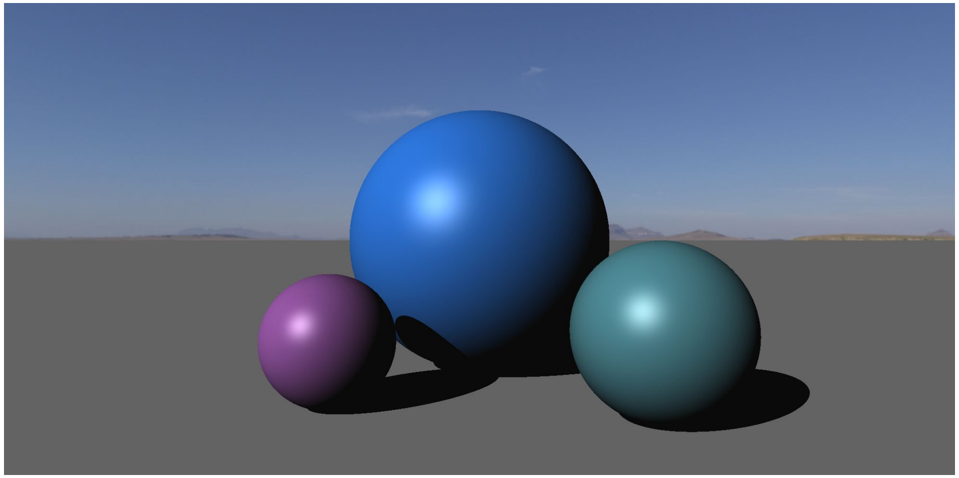
- Instead of returning black for all missed rays, we can use the ray direction d to sample an environment map, similar to reflection mapping

$$c_{\text{miss}} = \text{skybox}(d)$$



Ray Traced Reflections

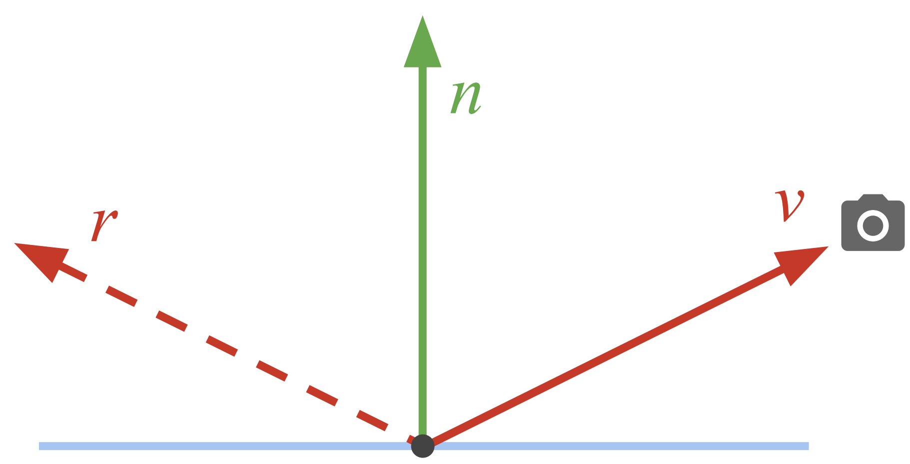
- To get reflections, we must cast a new ray in the reflection direction, calculate its shading and add it to the final color of the pixel



Ray Traced Reflections

- Reminder - to find the reflection direction:

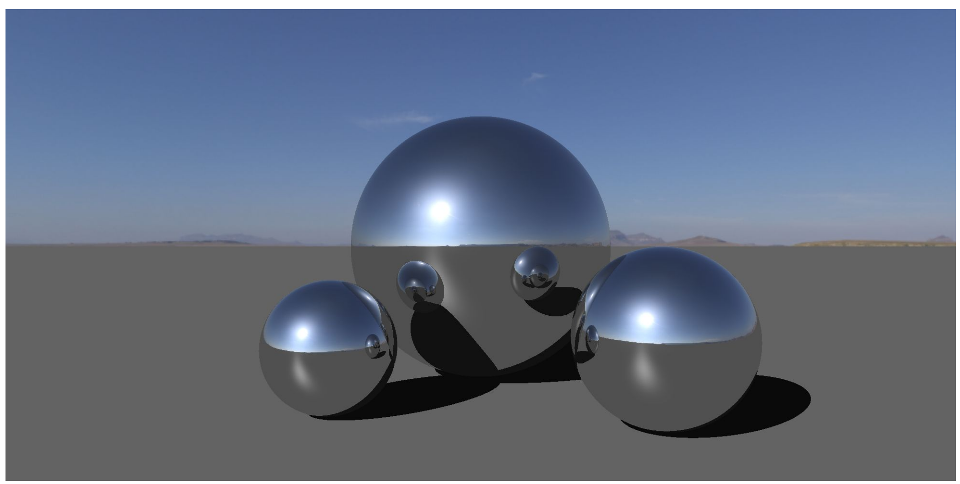
$$r = 2(v \cdot n)n - v$$



Ray Traced Reflections

- We recursively trace the new ray from the hit point p at direction r

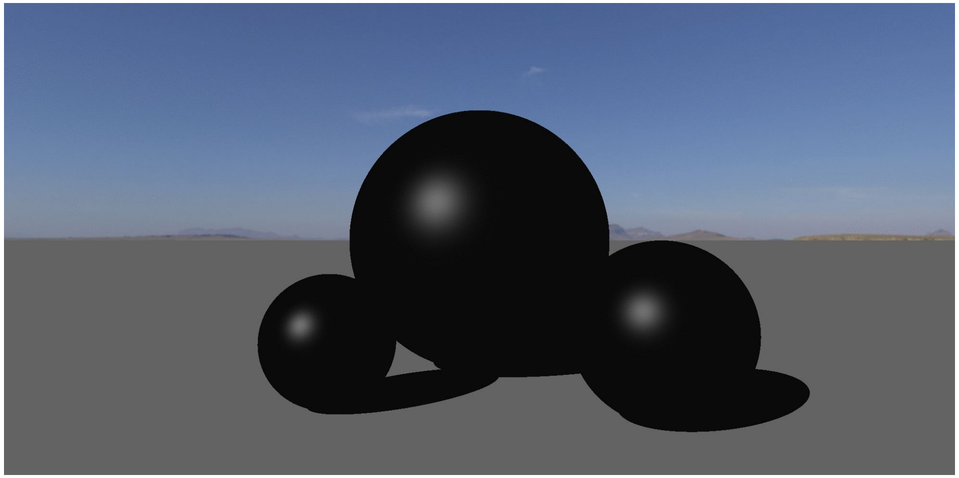
$$c_{\text{hit}} = v(p+\varepsilon n)\text{BlinnPhong}(p, d) + \text{trace}(p, r, e)$$



Ray Traced Reflections

- Note that we must set a bounce limit l for the light rays otherwise the recursion can go on forever!

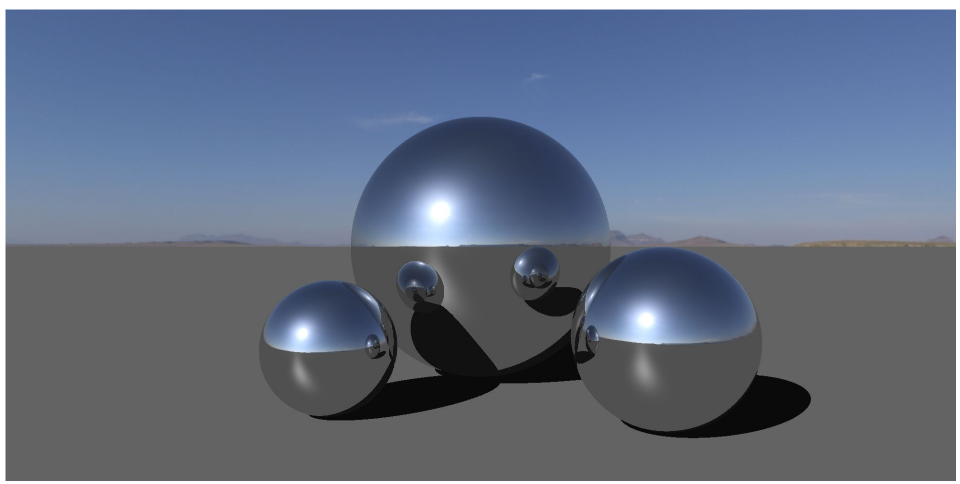
$$l = 1 \ldots 5$$



Ray Traced Reflections

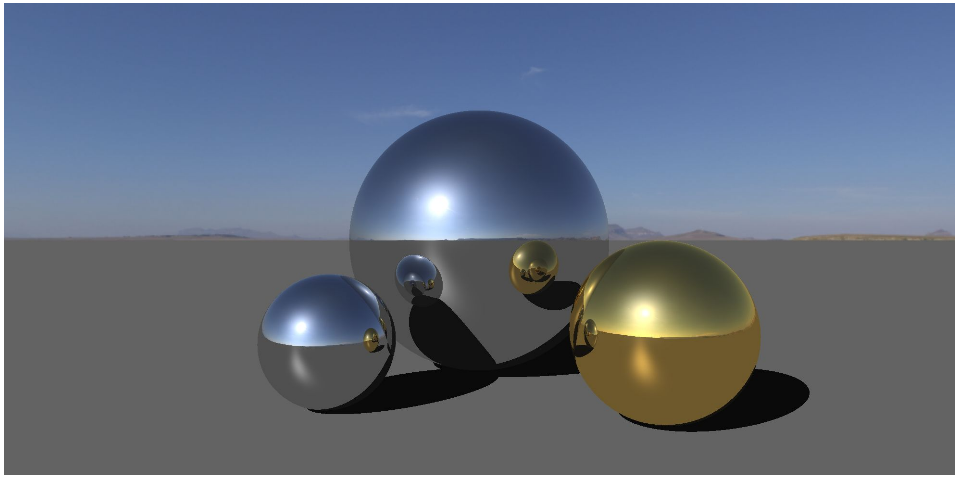
- We multiply the reflected ray's energy by the *specular coefficient* k_s to account for absorption

$$c_{\text{hit}} = v(p+\varepsilon n)\text{BlinnPhong}(p, r_d) + \text{trace}(p, r, k_s e)$$



Ray Traced Reflections

- We can now save this specular coefficient in the material to get all kinds of effects
- For example, we know that gold has a specular reflectivity of roughly $k_s = (1, 0.78, 0.34)$



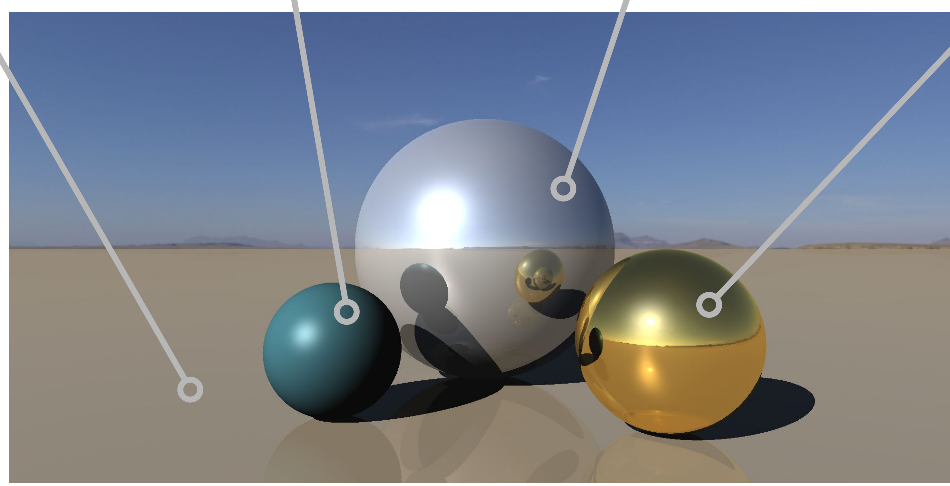
Ray Traced Reflections

$k_d = (0.2, 0.5, 0.5)$
$k_s = (0.0, 0.0, 0.0)$

$k_d = (0.5, 0.5, 0.5)$
$k_s = (0.5, 0.5, 0.5)$

$k_d = (0.9, 0.8, 0.7)$
$k_s = (0.1, 0.1, 0.1)$

$k_d = (0.0, 0.0, 0.0)$
$k_s = (1.0, 0.7, 0.3)$

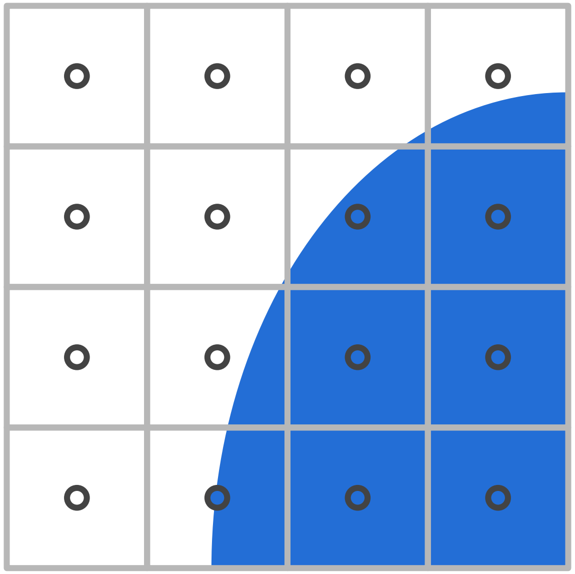


Aliasing

- If you look closely at the images we rendered, you can see aliasing artefacts - especially on the edges of objects and shadows

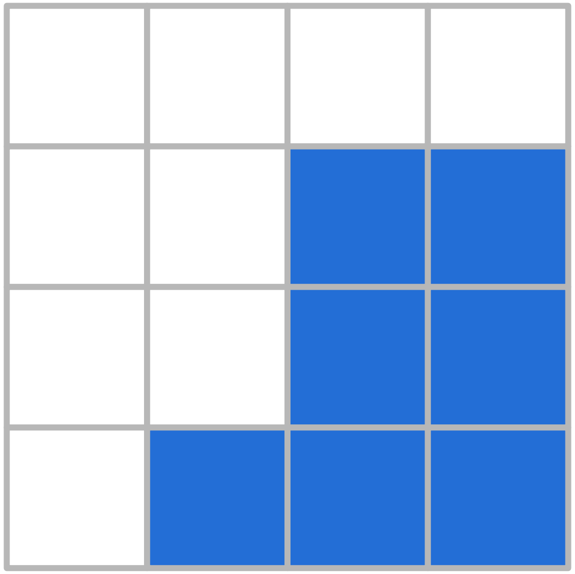
Aliasing

- This happens because we send just one ray from the center of each pixel:



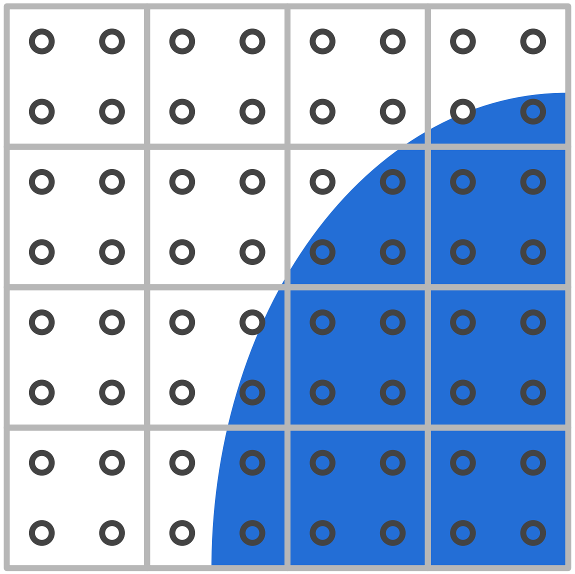
Aliasing

- This happens because we send just one ray from the center of each pixel:



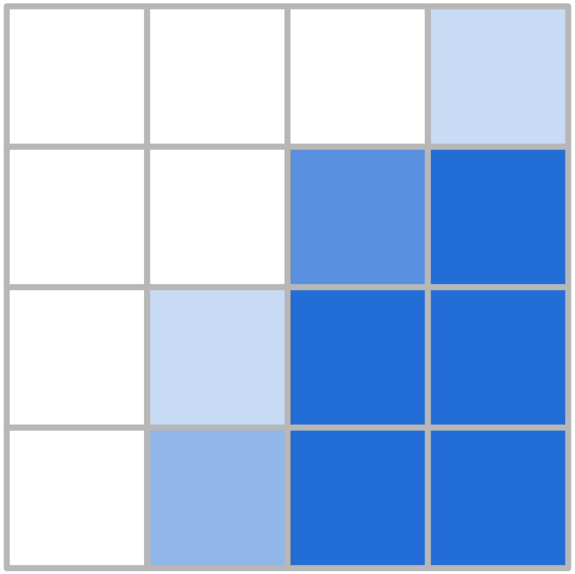
Supersampling

- The solution - send multiple rays from each pixel and average the result



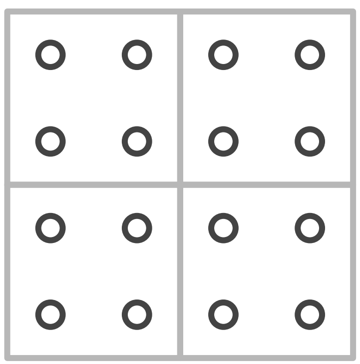
Supersampling

- This is called ***Supersampling*** anti-aliasing or *SSAA*

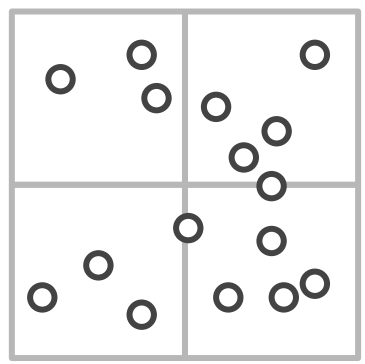


Supersampling

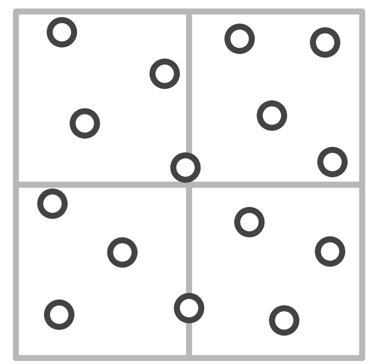
- There are all kinds of supersampling patterns we can use, each has strengths and weaknesses



Grid



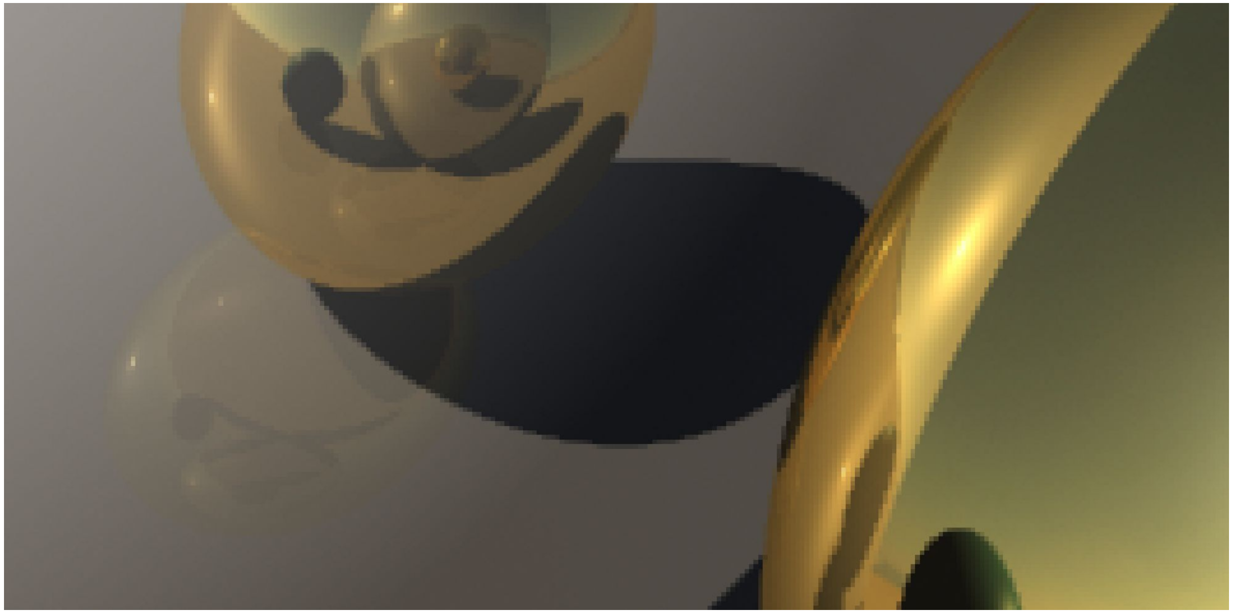
Random



Poisson disc

Aliasing

- We can see a big improvement in the zoomed-in rendering result:

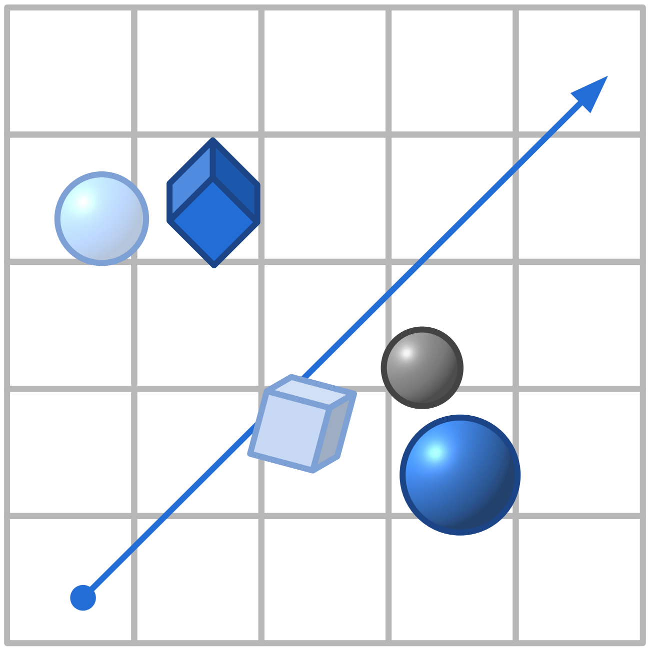


Acceleration Structures

- Let's say we want to render a 1280×1024 pixel image with 5 rays per pixel
- A recursion depth limit of 8
- 1000 objects to intersect in the scene (a common mesh can typically have much more triangles)
- 1280×1024×5×8×1000 = **52,428,800,000**
- This is a lot of work, even for a modern GPU!
- We need some sort of acceleration...

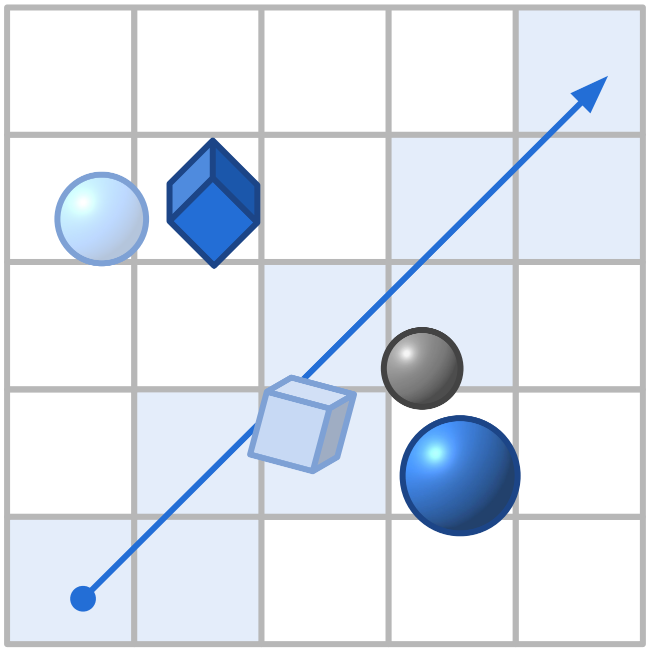
Uniform Grid Space Subdivision

- We can divide our scene into axis-aligned boxes, and associate each one with the objects within



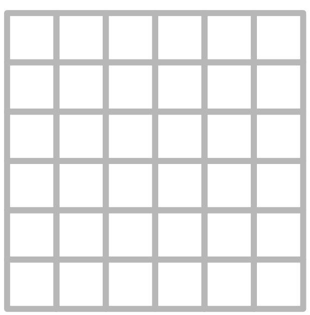
Uniform Grid Space Subdivision

- We efficiently check which grid cells are collided, and then check intersections with the relevant objects

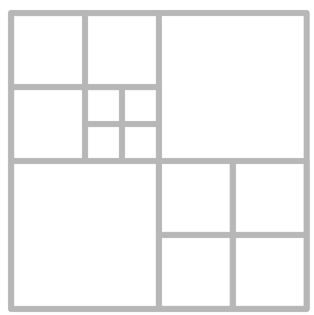


Space Subdivision

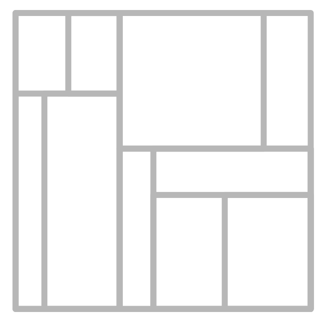
- There are various methods to subdivide space, each with advantages and drawbacks



Uniform Grid



Octree / Quadtree

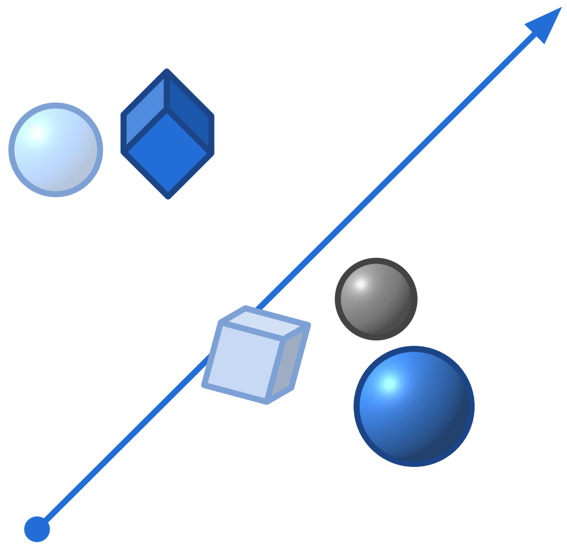


k-d tree

BSP tree

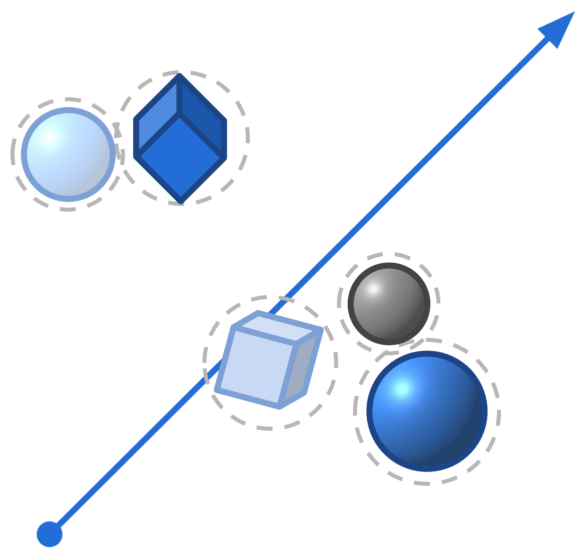
Bounding Volumes

- We can wrap objects in the scene with bounding volumes that will completely contain them



Bounding Volumes

- First check collisions with these volumes which is more efficient, then with the bounded objects

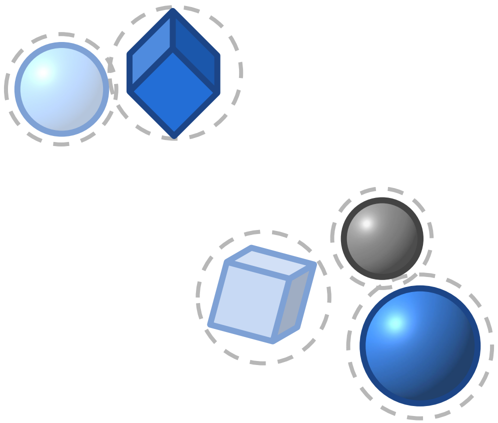


Bounding Volumes

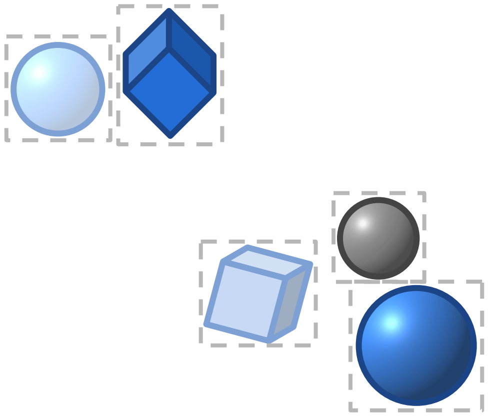
- We would like to use bounding volumes that have a very simple shape, such that intersection tests and distance computations are simple and fast
- On the other hand, we would like to have bounding volumes that fit the corresponding data objects very tightly

Bounding Volumes

- Two common choices:



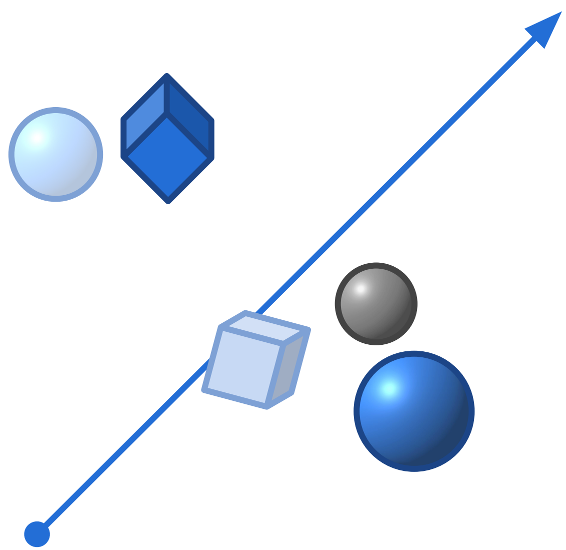
Spheres



Axis-aligned bounding boxes

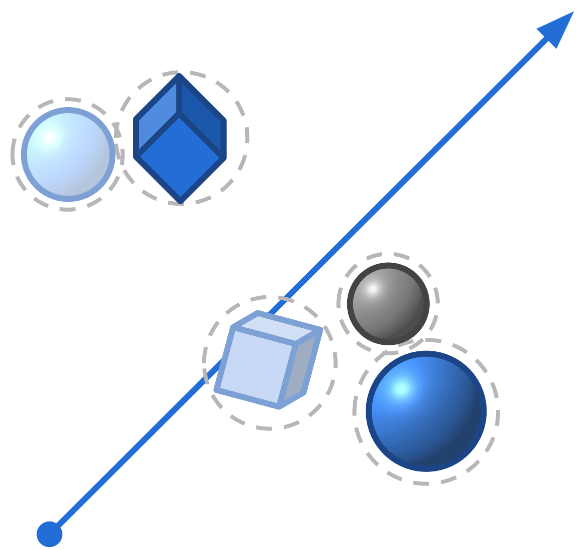
Bounding Volume Hierarchy

- A ***Bounding Volume Hierarchy*** or *BVH* is a tree structure on a set of geometric objects



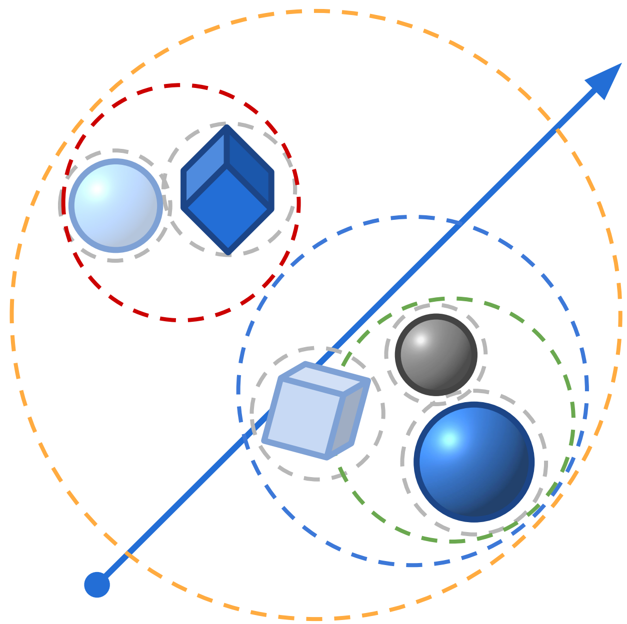
Bounding Volume Hierarchy

- First wrap all objects in bounding volumes that form the leaf nodes of the tree



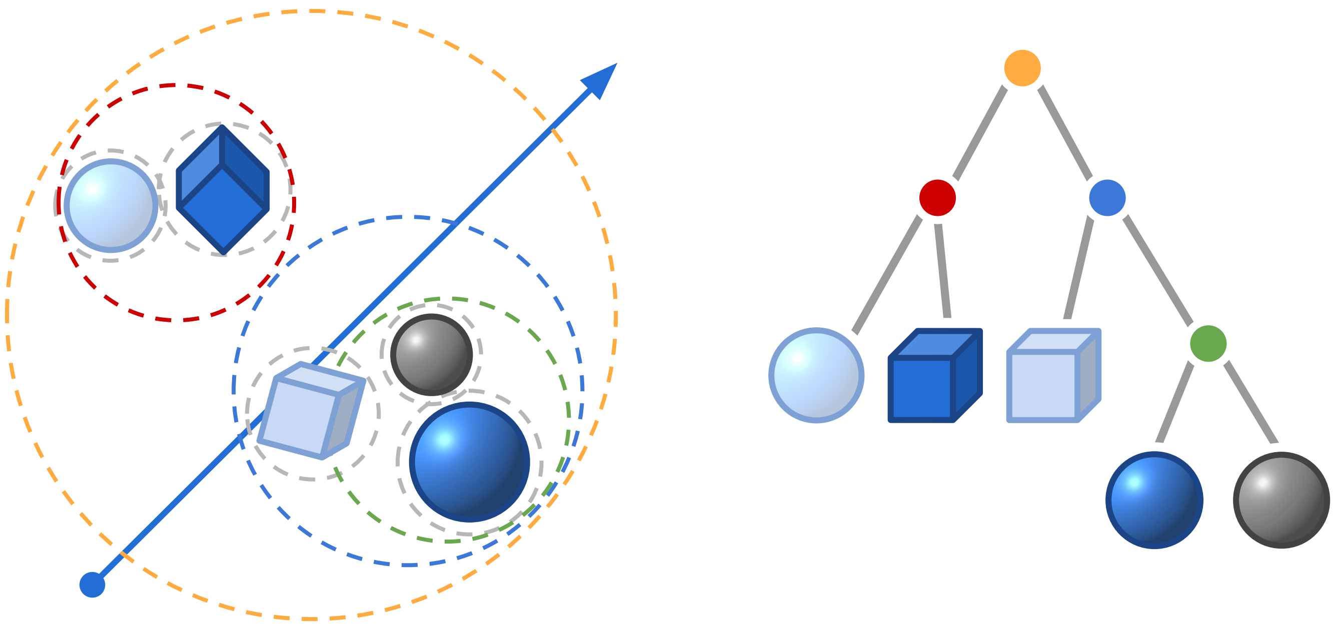
Bounding Volume Hierarchy

- These nodes are then grouped as small sets and enclosed within larger bounding volumes



Bounding Volume Hierarchy

- When checking for intersections, we first check the bounding volumes then travel down the tree



Bounding Volume Hierarchy

- Without BVH, checking collisions for the entire scene is $O(n)$
- With BVH, it is typically $O(\log n)$