CS3451:
Ray Tracing

Bo Zhu

School of Interactive Computing

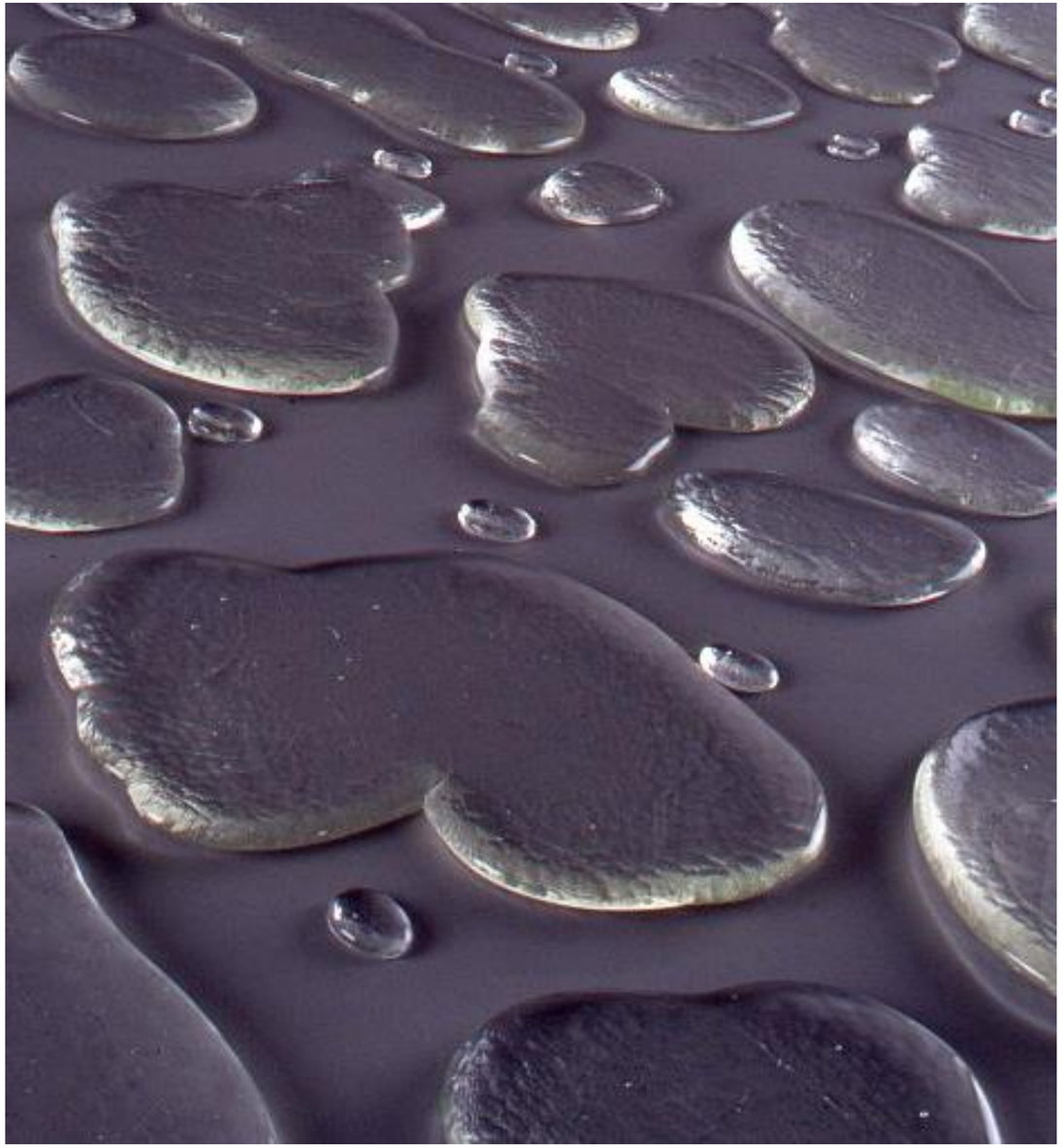
Georgia Institute of Technology

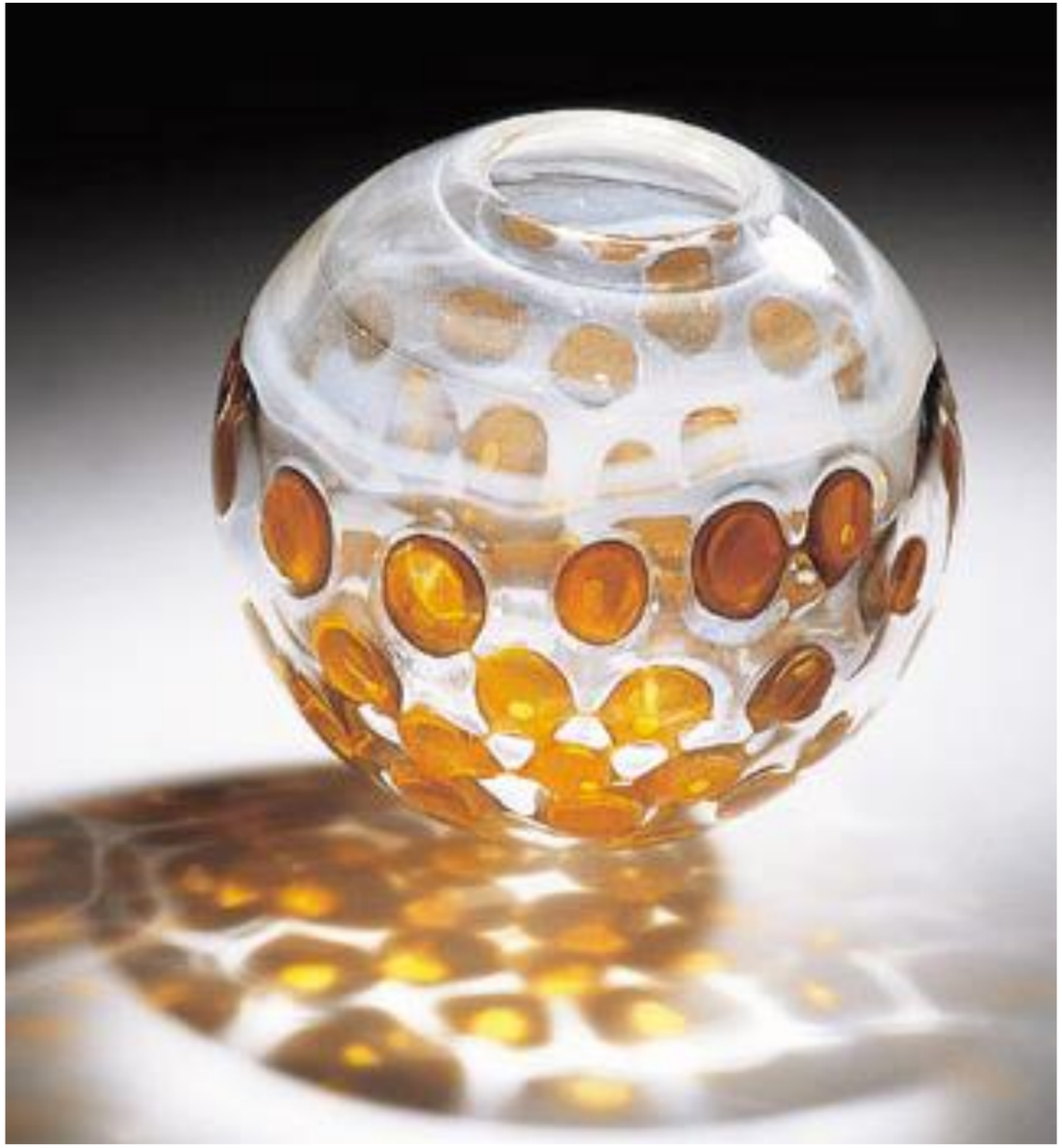
Motivational Video: Lumion 2023: The Glass House

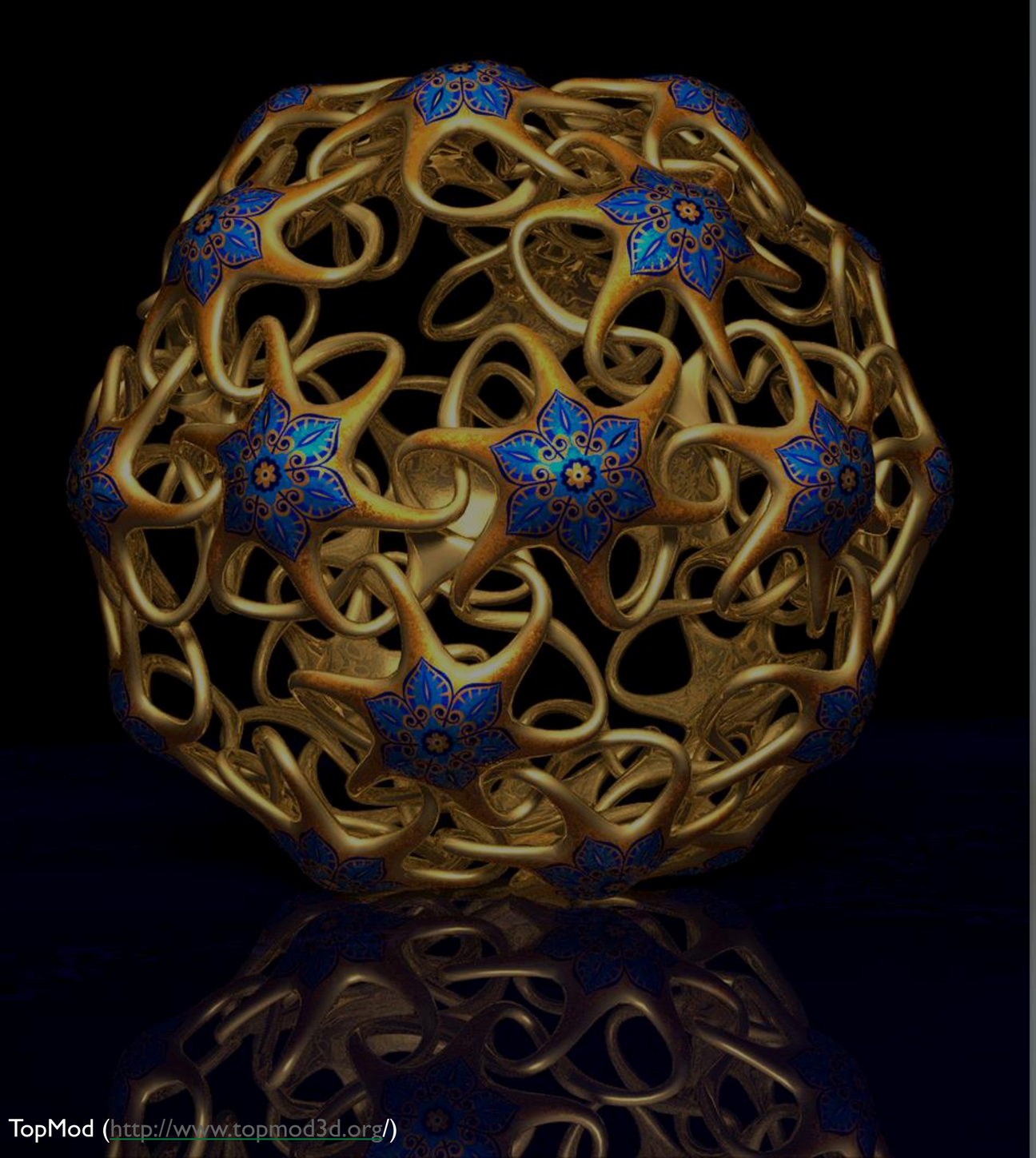
Motivational Video: UE4 Ray Tracing Engine

Motivational Video: NVIDIA Real-time Ray Tracing

https://www.youtube.com/watch?v=tiUiCzzVu8g







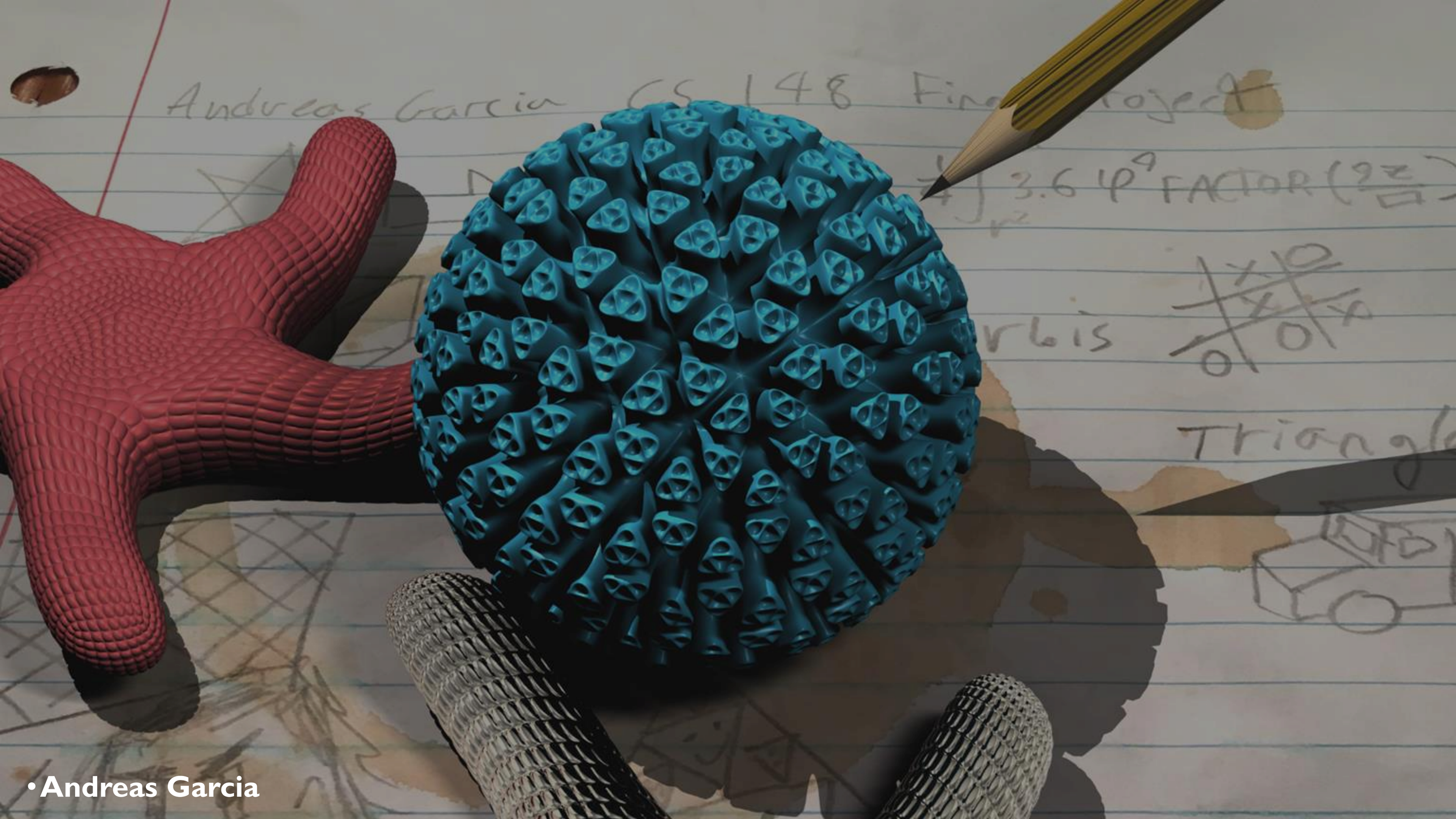
TopMod (http://www.topmod3d.org/)

- **Shyamal Buch & Wilbur Yang**

- Carol Hsin

- Michael M Pozulp & Nathan Lee

- Lucy Yiming Wang & Arit Paul

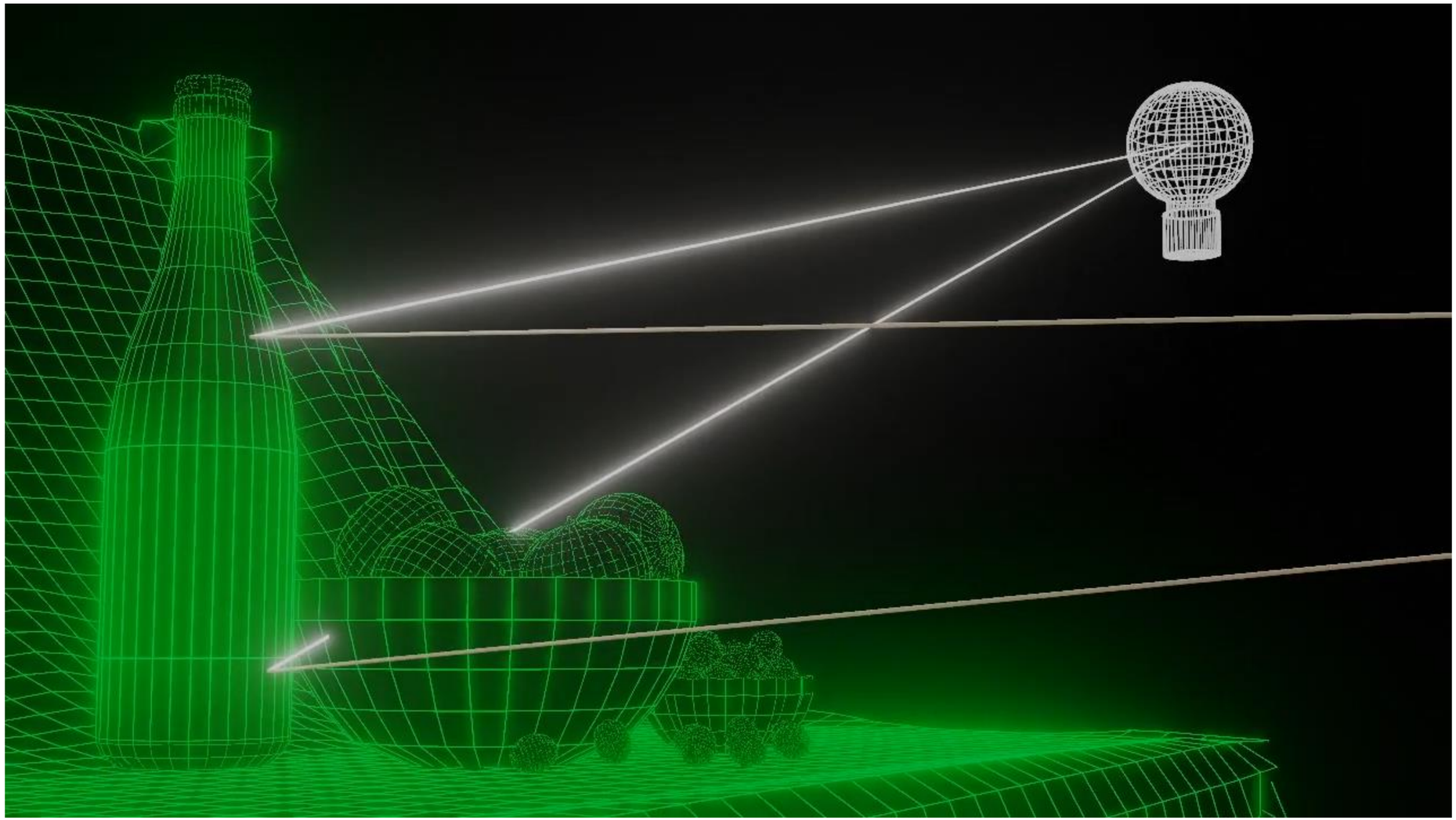


- Andreas Garcia

Study Plan

- Basic Ray Tracing
  - Ray
  - Ray-object intersection
  - Lighting
  - Shadow
- Recursive Ray Tracing
  - Reflection
  - Refraction

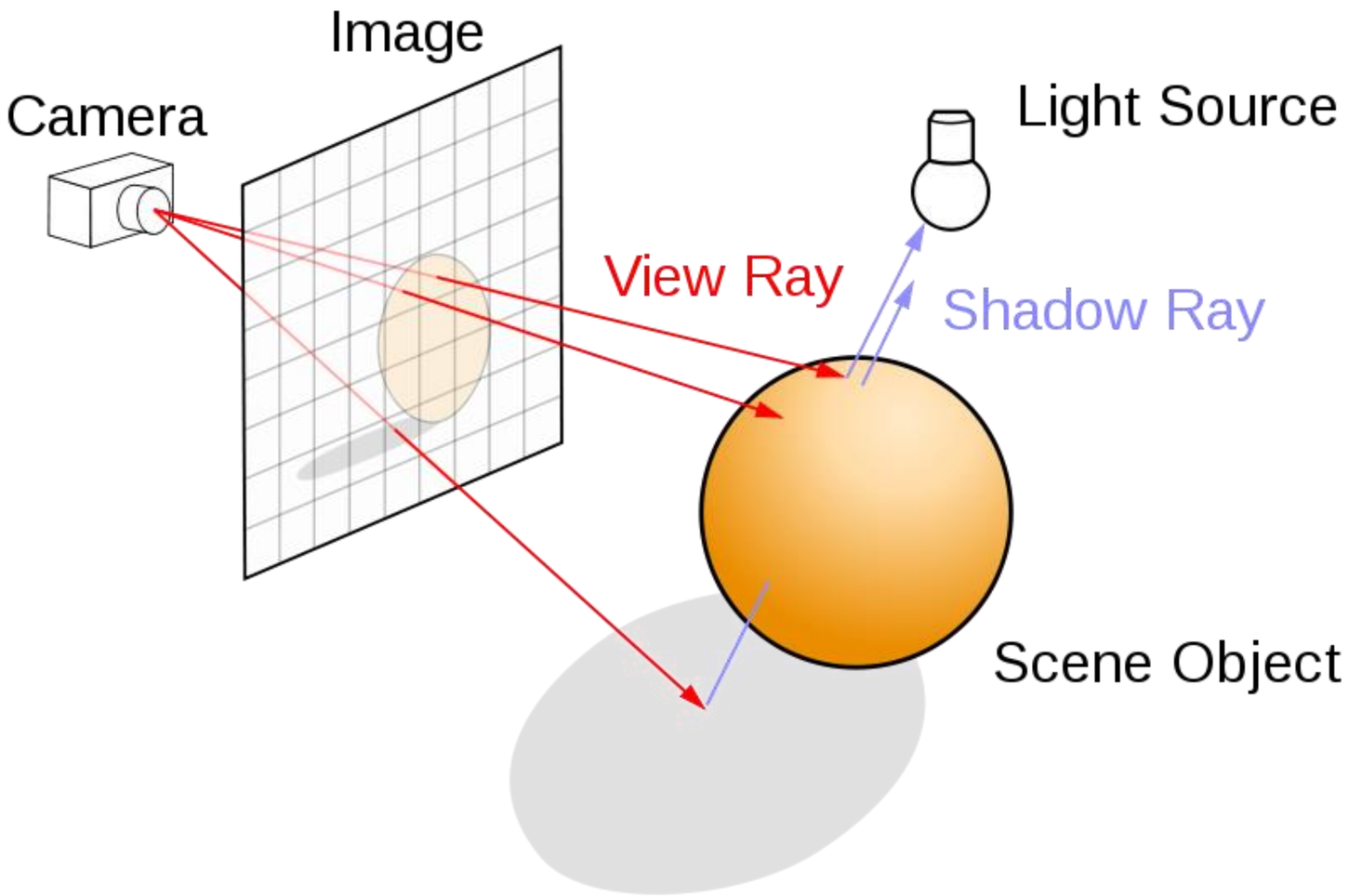
Two-Minute Explaination of Ray Tracing



Elbert With An E: https://www.youtube.com/watch?v=oCsgTrGLDiI

Ray Tracing – Key Idea

- Forward rays to calculate pixel colors in a 3D scene

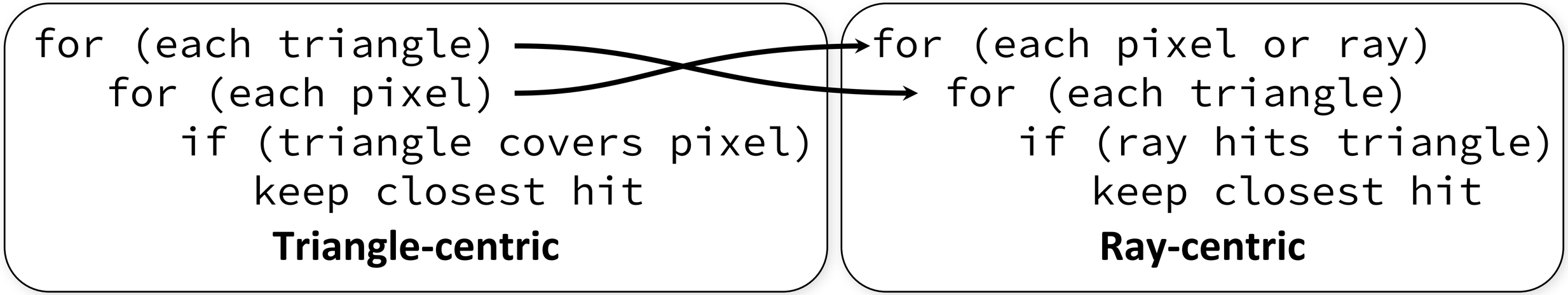


OpenGL Rendering v.s. Ray Tracing

- Rasterization: object point to image plane
  - start with a 3D object point
  - apply transforms
  - determine the 2D image plane point it projects to
- Ray tracing: image plane to object point
  - start with a 2D image point
  - generate a ray
  - determine the visible 3D object point

OpenGL rendering and ray tracing are inverse processes to each other

Rasterization vs. Ray casting

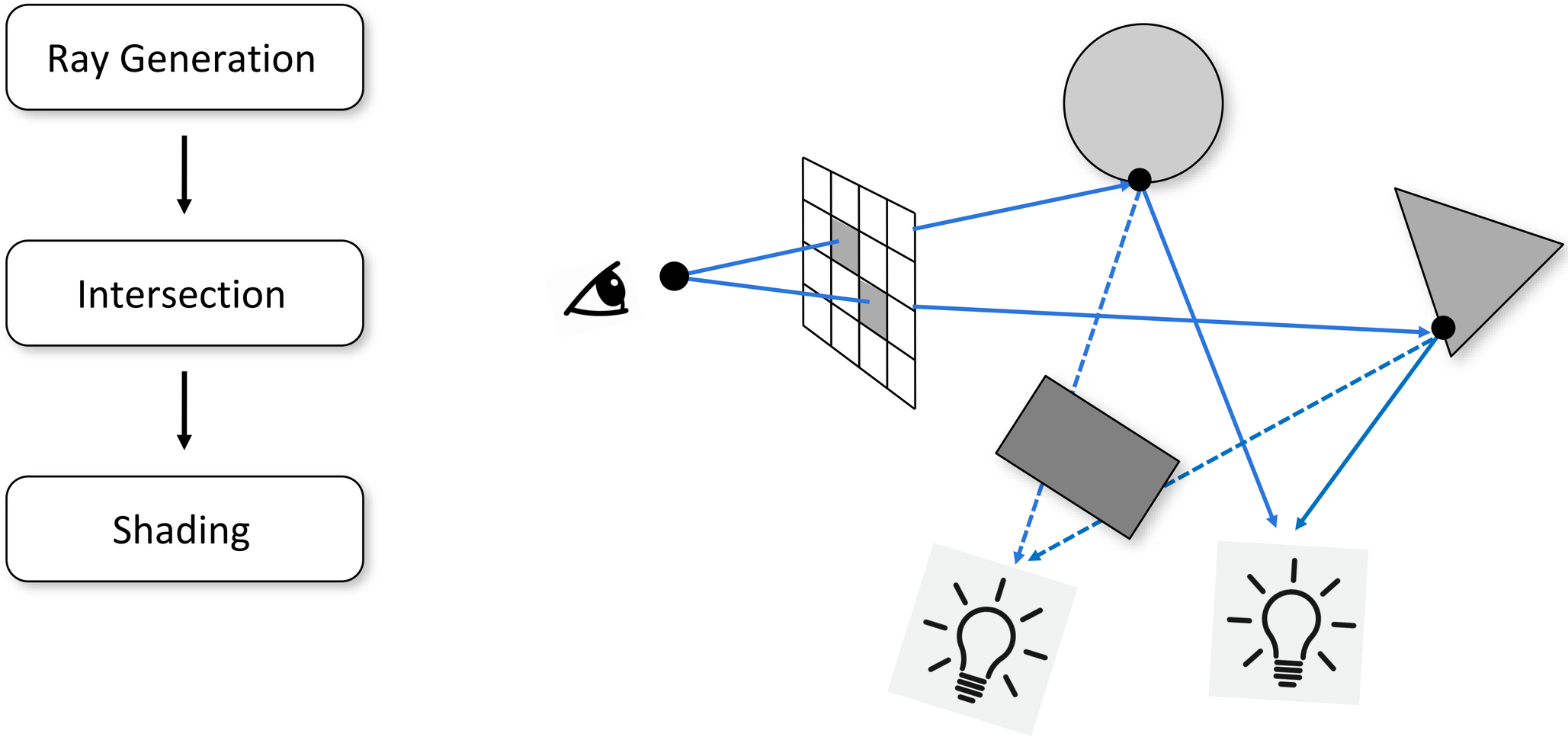


- What needs to be stored in memory in each case?

  - The rasterizer only needs one triangle at a time, *plus* the entire image and associated depth information for all pixels

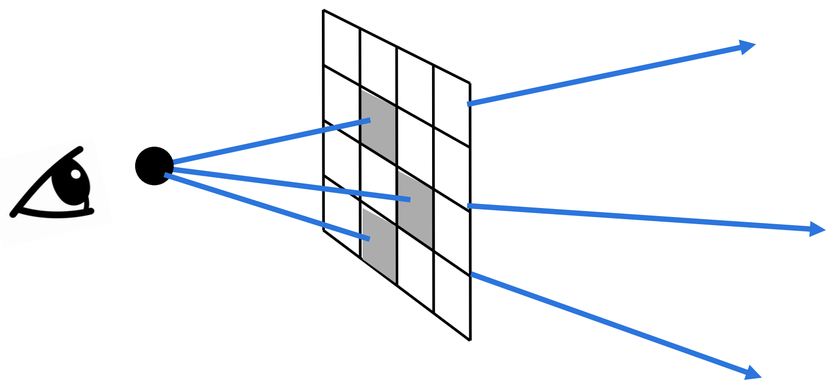
  - In this basic form, ray tracing needs the entire scene description in memory at once

Basic Ray Tracing Pipeline



Pseudocode: Basic Ray Tracing

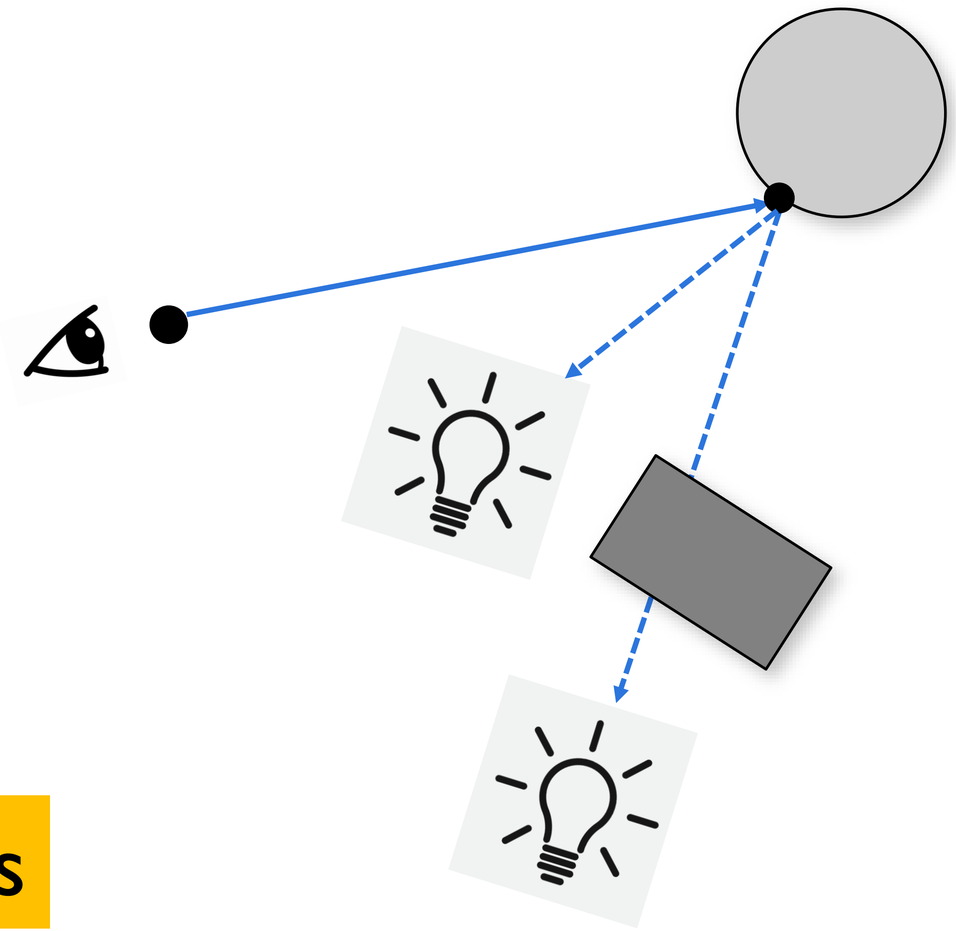
```
rayTraceImage()
{
    parse scene description

    for each pixel
        ray = generateCameraRay(pixel)
        pixelColor = trace(ray)
}
```



Pseudocode: Basic Ray Tracing

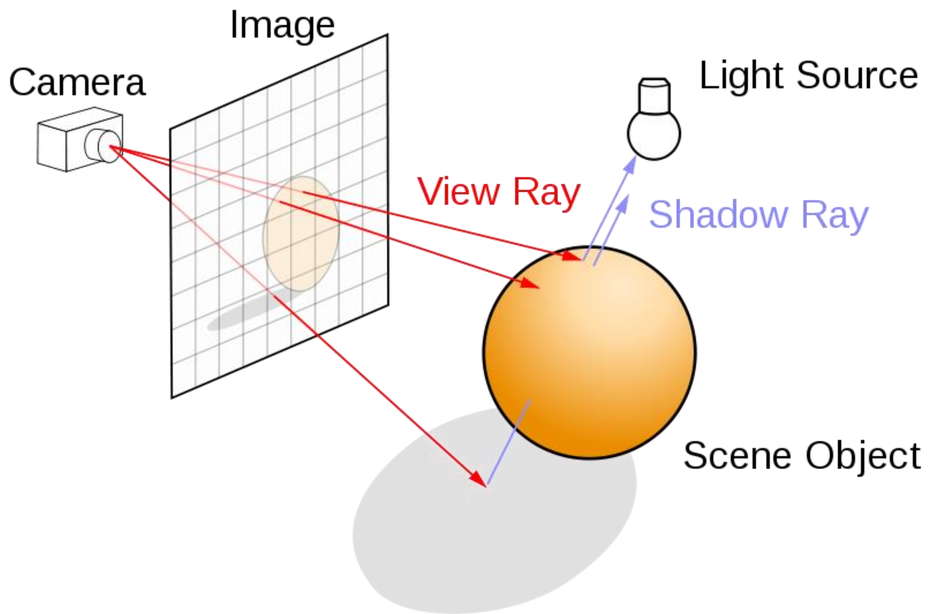
```
trace(ray)
{
    hit = find first intersection
            with scene objects

    color = shade(hit)
    return color
}
```

handle shadow by tracing more rays

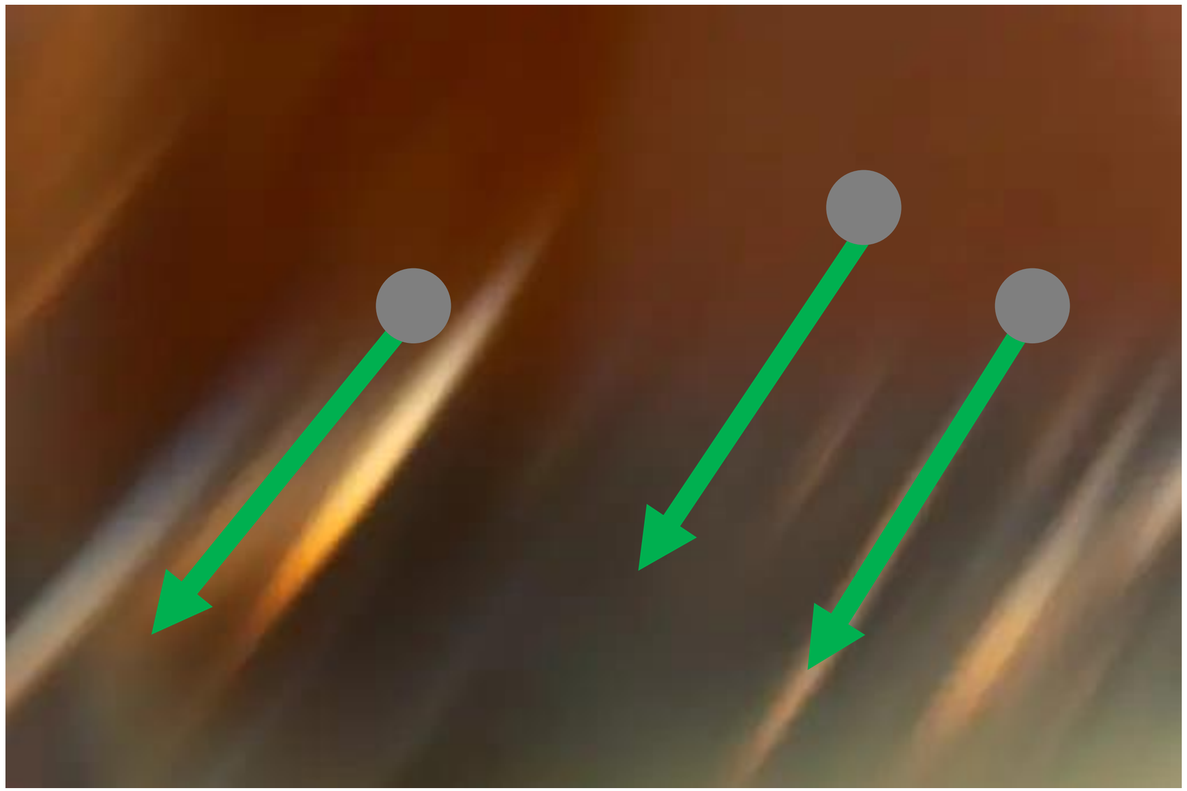


Questions for Basic Ray Tracing

- Q1: How do we generate a ray?
- Q2: How does a ray intersect with an object?
  - Sphere
  - Plane
  - Triangle
- Q3: How do we calculate lighting?
- Q3: How do we calculate shadow?



Ray

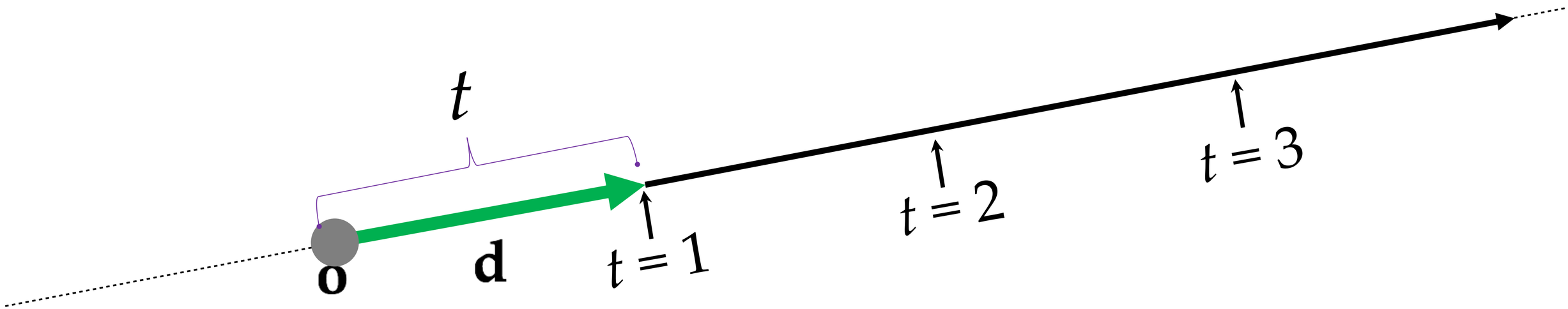


Ray: a Half Line

- Standard representation: origin (point) $\mathbf{o}$ and direction $\mathbf{d}$

$$\boldsymbol{r}(t) = \boldsymbol{o} + t\boldsymbol{d}, \text{ with } t > 0$$

  - Note replacing $\mathbf{d}$ with $a\mathbf{d}$ doesn't change ray (for $a > 0$)

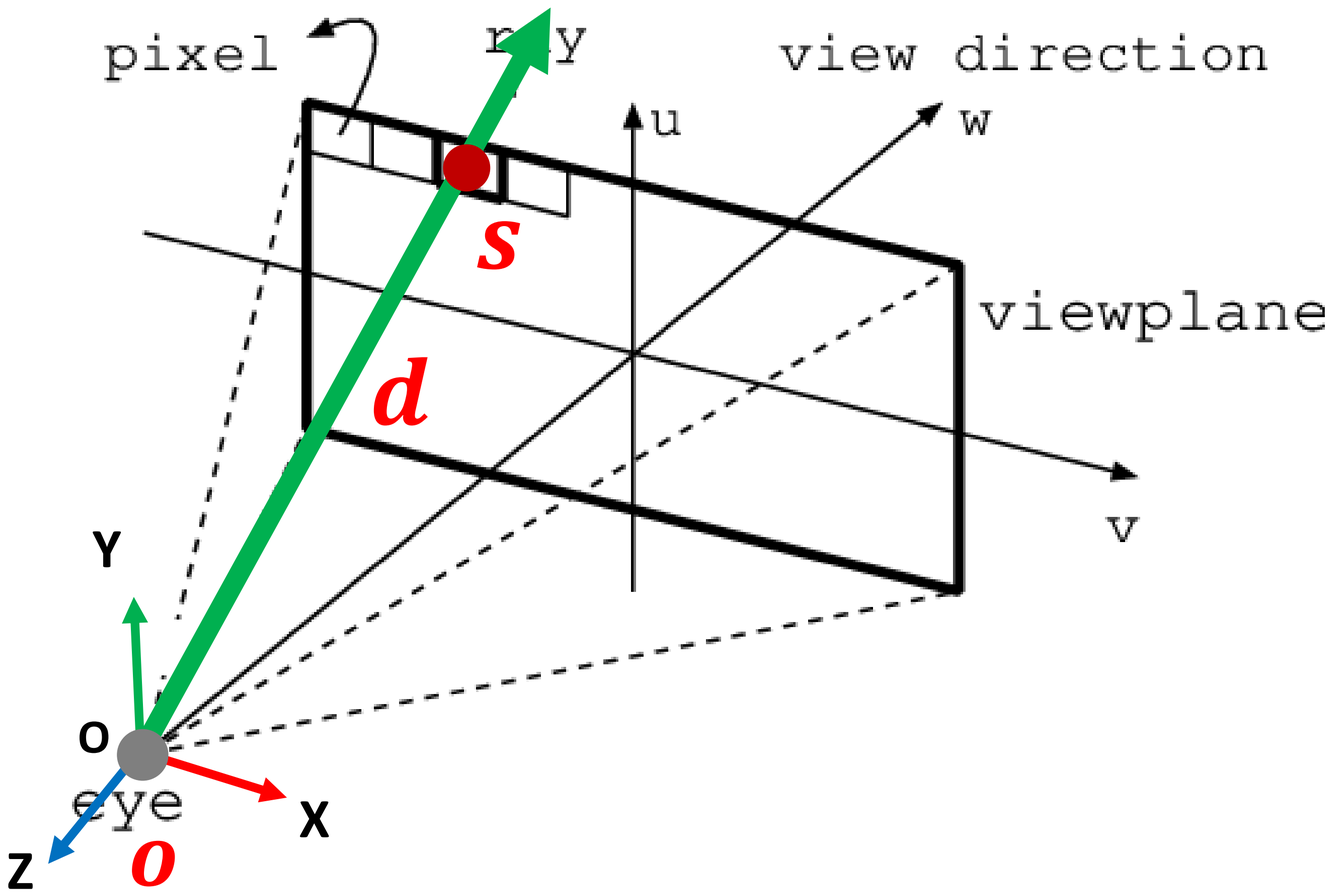


Generating Rays for Each Pixel

- Establish a view rectangle
- Create a ray for each pixel on the rectangle

$$\boldsymbol{d} = \boldsymbol{s} - \boldsymbol{o}$$

$$\boldsymbol{r}(t) = \boldsymbol{o} + t\boldsymbol{d}, \text{ with } t > 0$$



Ray-Object Intersection

Ray-Sphere Intersection

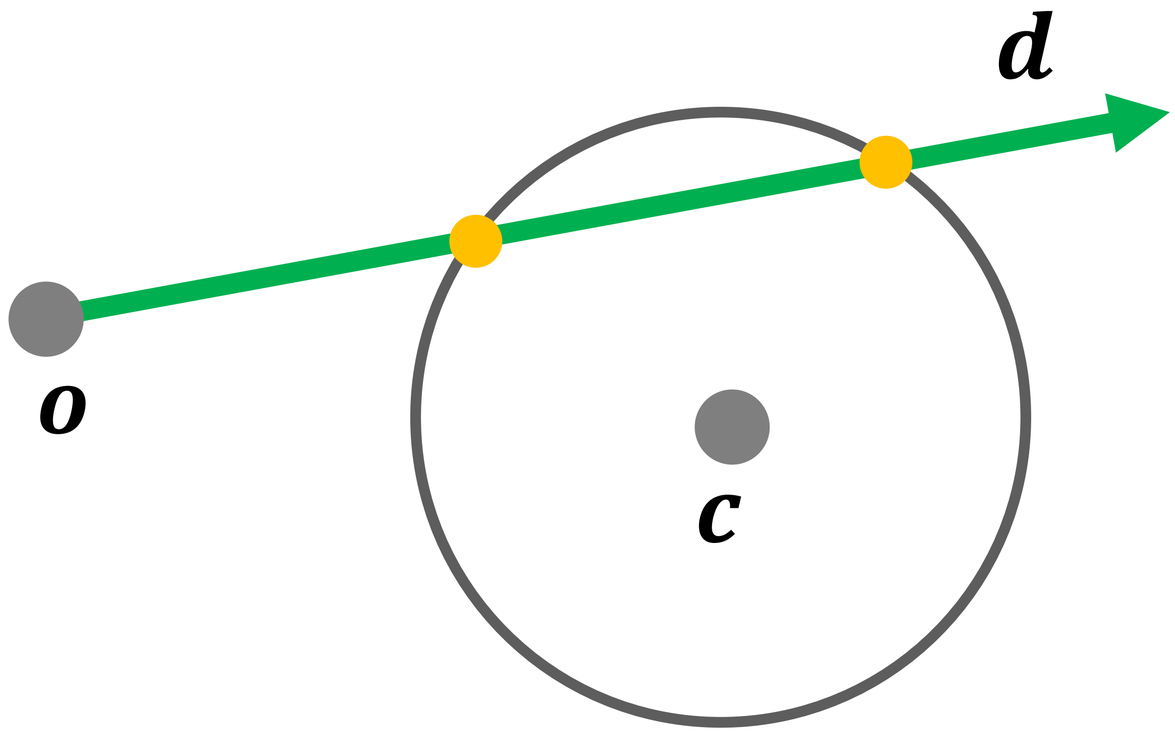
- Algebraic approach:
  - **Condition I:** point is on ray:

    $\boldsymbol{x}(t) = \boldsymbol{o} + t\boldsymbol{d}$, with $t > 0$

  - **Condition II**: point is on sphere:

    $|\boldsymbol{x}(t) - \boldsymbol{c}|^2 = \boldsymbol{r}^2$

  - Substitute I into II and solve for t



Solve a quadratic equation for t

$$|\boldsymbol{o} + t\boldsymbol{d} - \boldsymbol{c}|^2 = \boldsymbol{r}^2$$

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

$$\underbrace{\boldsymbol{d}^2 t^2}_{\boldsymbol{A}} + \underbrace{2(\boldsymbol{o} \cdot \boldsymbol{d} - \boldsymbol{c} \cdot \boldsymbol{d})t}_{\boldsymbol{B}} + \underbrace{(\boldsymbol{o}^2 + \boldsymbol{c}^2 - 2(\boldsymbol{o} \cdot \boldsymbol{c}) - \boldsymbol{r}^2)}_{\boldsymbol{C}} = \boldsymbol{0}$$

$$At^2 + Bt + C = 0$$

$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

$$\boldsymbol{A} = \boldsymbol{d}^2$$

$$\boldsymbol{B} = 2(\boldsymbol{o} \cdot \boldsymbol{d} - \boldsymbol{c} \cdot \boldsymbol{d})$$

$$\boldsymbol{C} = (\boldsymbol{o}^2 + \boldsymbol{c}^2 - 2(\boldsymbol{o} \cdot \boldsymbol{c}) - \boldsymbol{r}^2)$$

No intersection if $B^2 - 4AC < 0$

Geometric Intuition

$$PQ = PM - QM$$
$$QM^2 = QN^2 - MN^2$$
$$MN^2 = PN^2 - PM^2$$

- If we assume $\boldsymbol{d}$ has a unit length, we will get

$$\boldsymbol{A} = \boldsymbol{d}^2 = 1$$

$$\boldsymbol{B} = 2(\boldsymbol{o}\cdot\boldsymbol{d} - \boldsymbol{c}\cdot\boldsymbol{d}) = -2(\boldsymbol{c}-\boldsymbol{o})\cdot\boldsymbol{d} = -2\boldsymbol{l}\cdot\boldsymbol{d}$$

Here, we define $\boldsymbol{l} = (\boldsymbol{c}-\boldsymbol{o})$

$$\boldsymbol{C} = \left(\boldsymbol{o}^2 + \boldsymbol{c}^2 - 2(\boldsymbol{o}\cdot\boldsymbol{c}) - \boldsymbol{r}^2\right) = (\boldsymbol{c}-\boldsymbol{o})^2 - \boldsymbol{r}^2 = \boldsymbol{l}^2 - \boldsymbol{r}^2$$

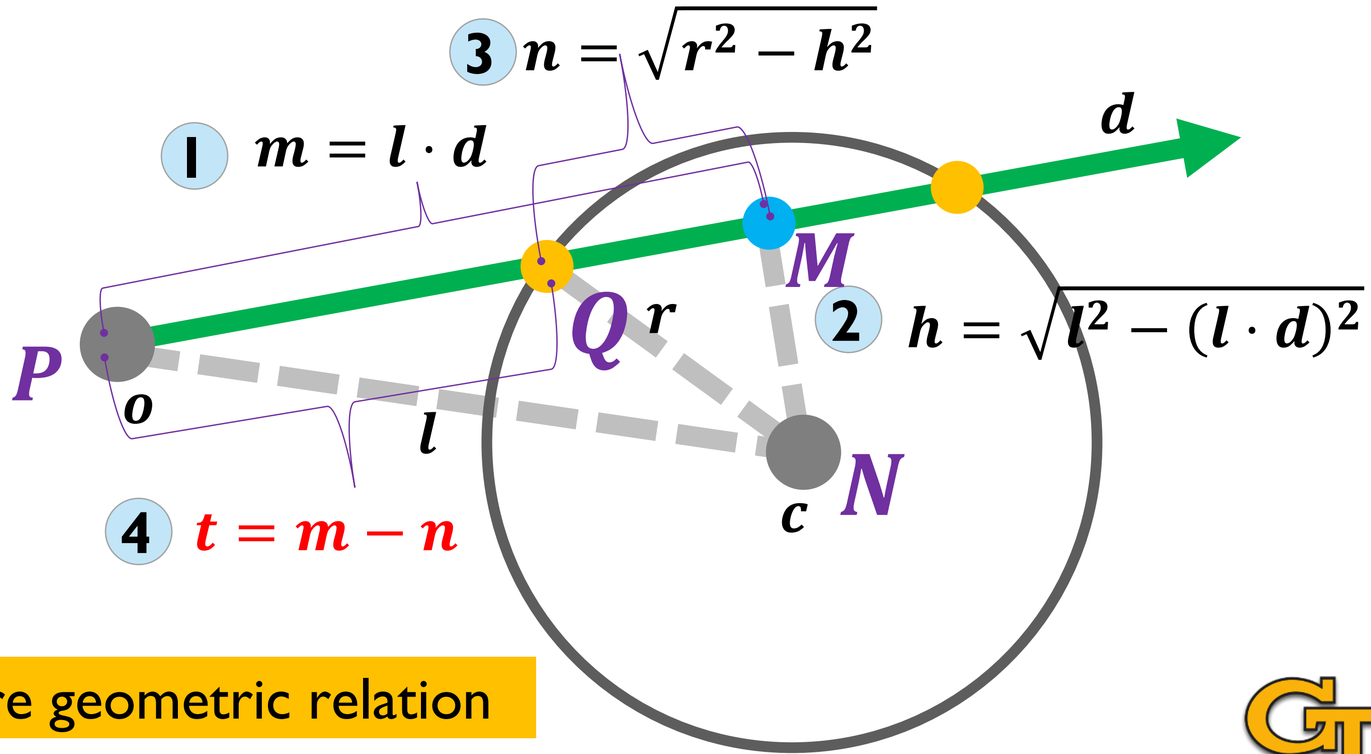
$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2}$$

$$= \frac{\left(2\boldsymbol{l}\cdot\boldsymbol{d} \pm \sqrt{4(\boldsymbol{l}\cdot\boldsymbol{d})^2 - 4(\boldsymbol{l}^2 - \boldsymbol{r}^2)}\right)}{2}$$

$$= \boldsymbol{l}\cdot\boldsymbol{d} \pm \sqrt{(\boldsymbol{l}\cdot\boldsymbol{d})^2 - (\boldsymbol{l}^2 - \boldsymbol{r}^2)}$$

$$= \boldsymbol{l}\cdot\boldsymbol{d} \pm \sqrt{\boldsymbol{r}^2 - (\boldsymbol{l}^2 - (\boldsymbol{l}\cdot\boldsymbol{d})^2)}$$

$$\boldsymbol{m} \pm \boldsymbol{n}$$



The intersection can be calculated using pure geometric relation

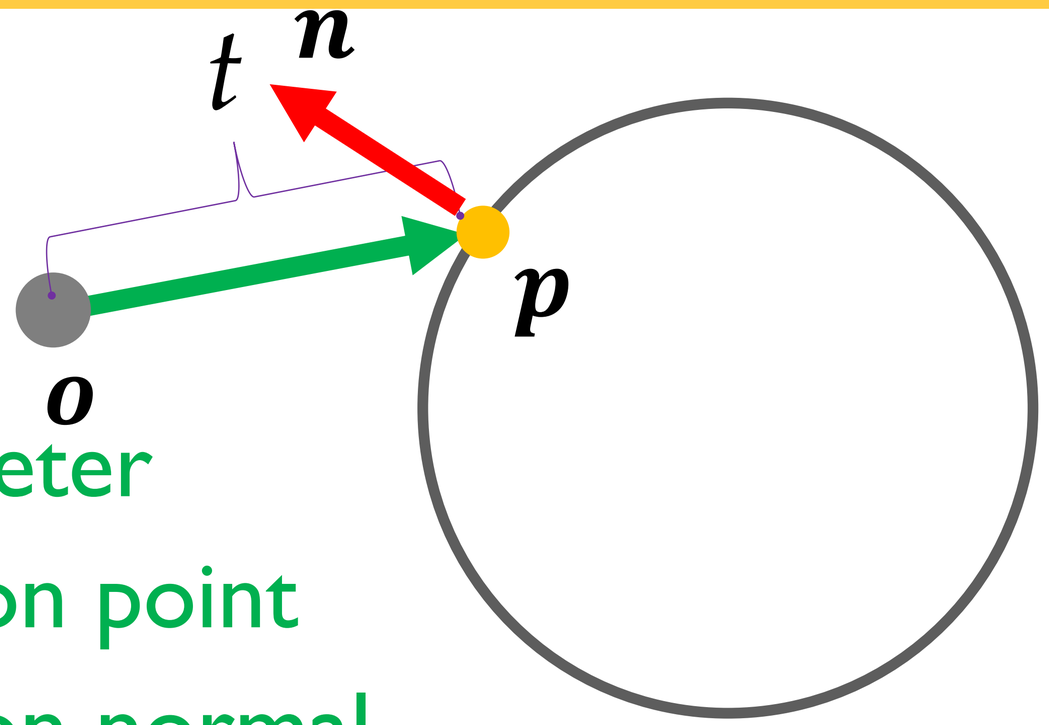
Ray Data Structure

```
struct Ray
{
    vec3 ori;     //// ray origin
    vec3 dir;     //// ray direction
};
```

Sphere Data Structure

```
struct Sphere
{
    vec3 ori;       //// sphere origin
    float r;        //// sphere radius
    int matId;      //// sphere material ID
};
```

Hit Data Structure




```
struct Hit {
    float t;            //// ray parameter
    vec3 p;             //// intersection point
    vec3 normal;        //// intersection normal
    int matId;          //// material ID
};
```

Example:

```
Hit hit = noHit;
/* Your implementation on calculating t, p, n, and matId */
hit = Hit(t, p, n, matId);
return hit;
```

Default Hit Value for No Intersection

```
const Hit noHit = Hit(
                    /* negative t */          -1.0,
                   /* hit position */       vec3(0),
                   /* hit normal */          vec3(0),
                   /* hit material id*/    -1);
```

We can declare a default Hit value with **a negative t** to represent no hit

Ray-Sphere Intersection Pseudocode

- **Input**: Ray r, Sphere s
- **Output**: hit
- **Algorithm**:

  hit = noHit;
  calculate A, B, C based on r and s
  calculate $\Delta = B^2 - 4AC$, if $\Delta < 0$, return noHit
  calculate $t = (-B - \sqrt{\Delta})/(2A)$, if t<0, return noHit
  calculate the hit point p and normal n using t
  hit = Hit(t, p, n, s.matId)
  return hit

Ray-Plane Intersection

- Algebraic approach:
  - **Condition I:** point is on ray:

$$\boldsymbol{x}(t) = \boldsymbol{o} + t\boldsymbol{d}, \text{ with } t > 0$$

  - **Condition II**: point is on plane:

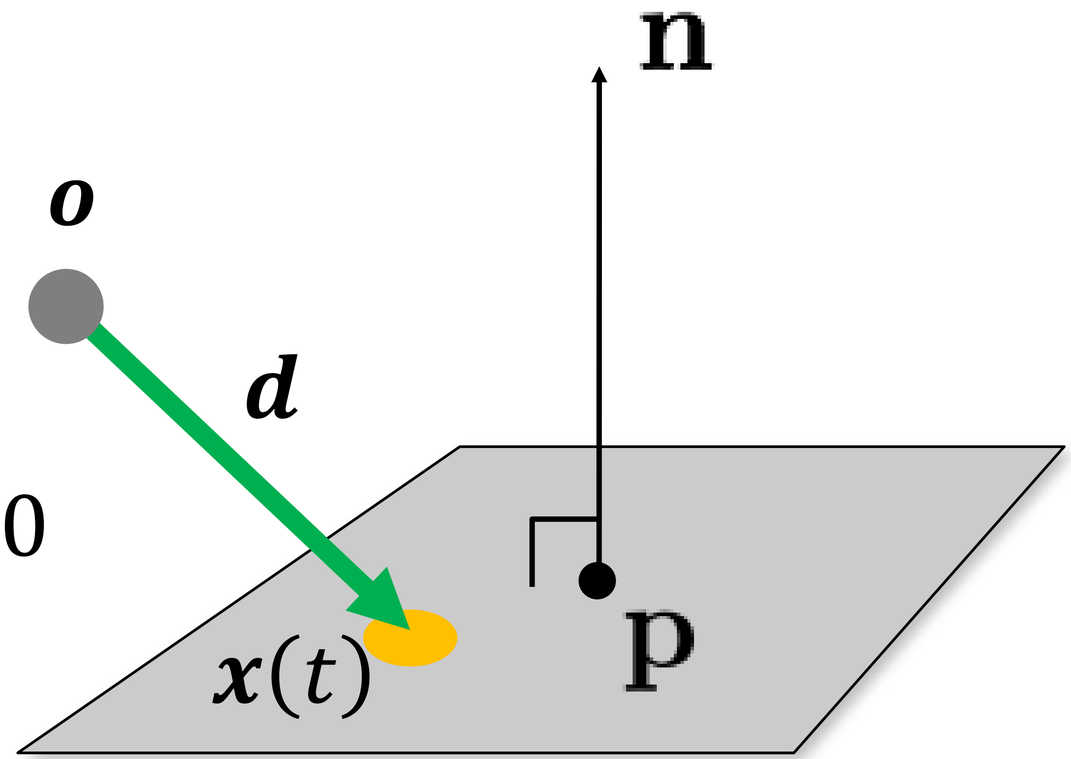
$$(\boldsymbol{x}(t) - \boldsymbol{p}) \cdot \boldsymbol{n} = 0, \text{ with } t > 0$$

point of interest — $\boldsymbol{x}(t)$

point on plane — $\boldsymbol{p}$

plane normal — $\boldsymbol{n}$

  - Substitute I into II and solve for t



Plane Data Structure

```
struct Plane
{
    vec3 n;          //// plane normal
    vec3 p;          //// plane point
    int matId;       //// plane material ID
};
```

Ray-Plane Intersection Pseudocode

- **Input**: Ray r, Plane pl
- **Output**: hit
- **Algorithm**:

```
hit = noHit;
solve t using ray-plane equations
calculate the hit point p and normal n using t
hit = Hit(t, p, n, pl.matId)
return hit
```

Ray-Triangle Intersection

- Algebraic approach:
  - **Condition I:** point is on ray:

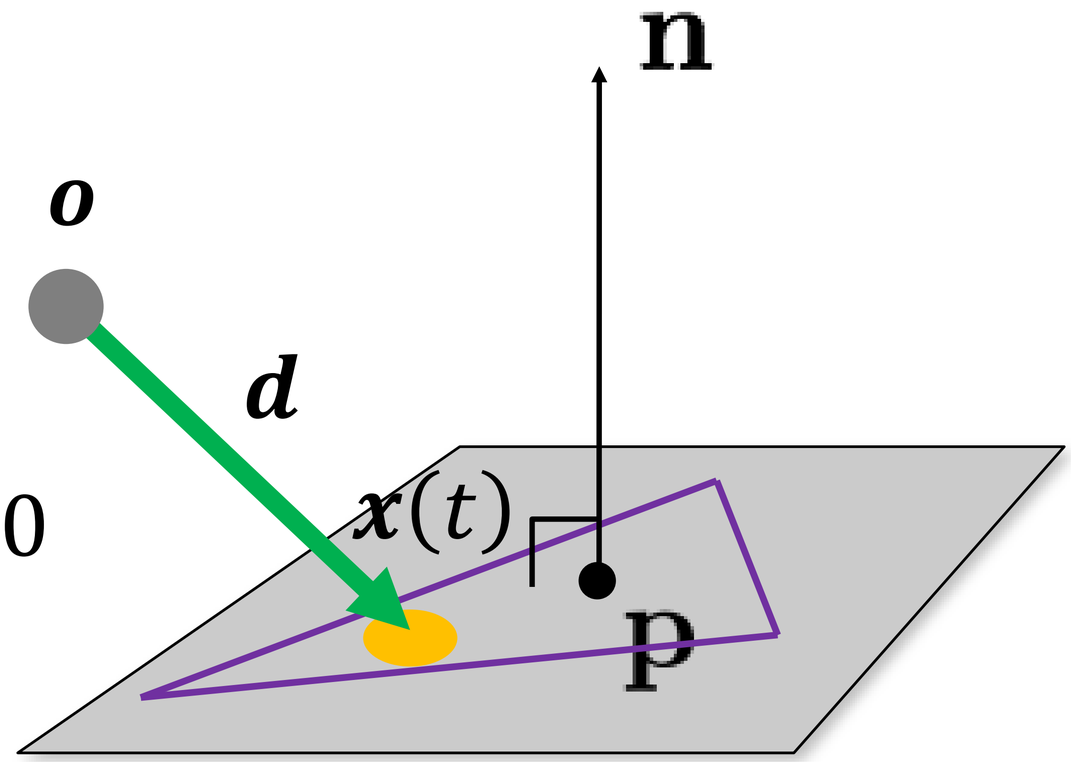
$$\boldsymbol{x}(t) = \boldsymbol{o} + t\boldsymbol{d}, \text{ with } t > 0$$

  - **Condition II**: point is on plane:

$$(\boldsymbol{x}(t) - \boldsymbol{p}) \cdot \boldsymbol{n} = 0, \text{ with } t > 0$$

point of interest — point on plane — plane normal

  - **Condition III:** point is inside triangle



Substitute **I** into **II** and solve for t, and then check if the point is inside the triangle

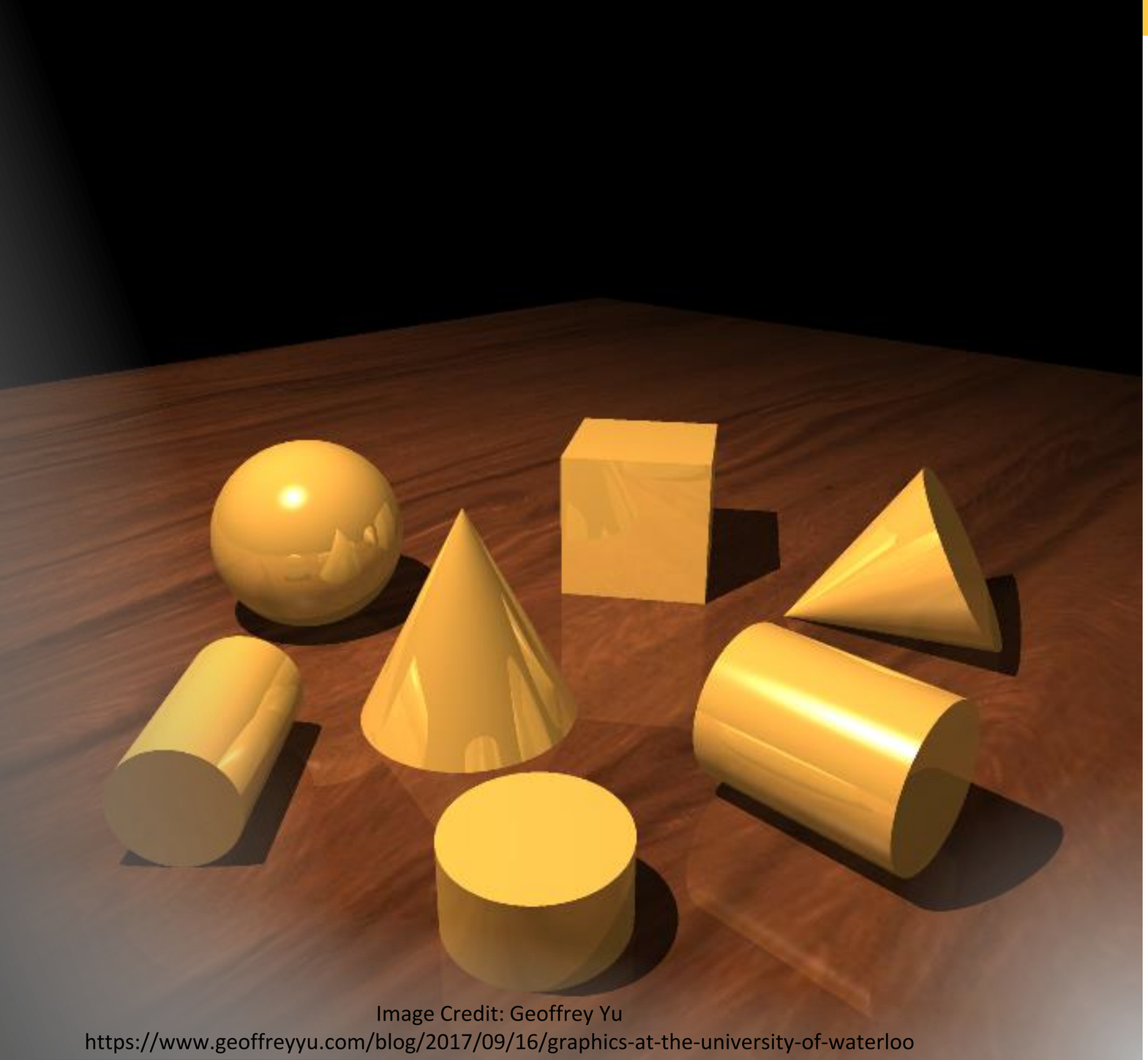


Image Credit: Geoffrey Yu
https://www.geoffreyyu.com/blog/2017/09/16/graphics-at-the-university-of-waterloo

Ray Intersections with other Primitives

Other primitives

- cylinder
- cone
- torus
- disk
- general polygons,
- meshes

Combination of Primitives for More Complicated Shapes

Constructive Solid Geometry (CSG)

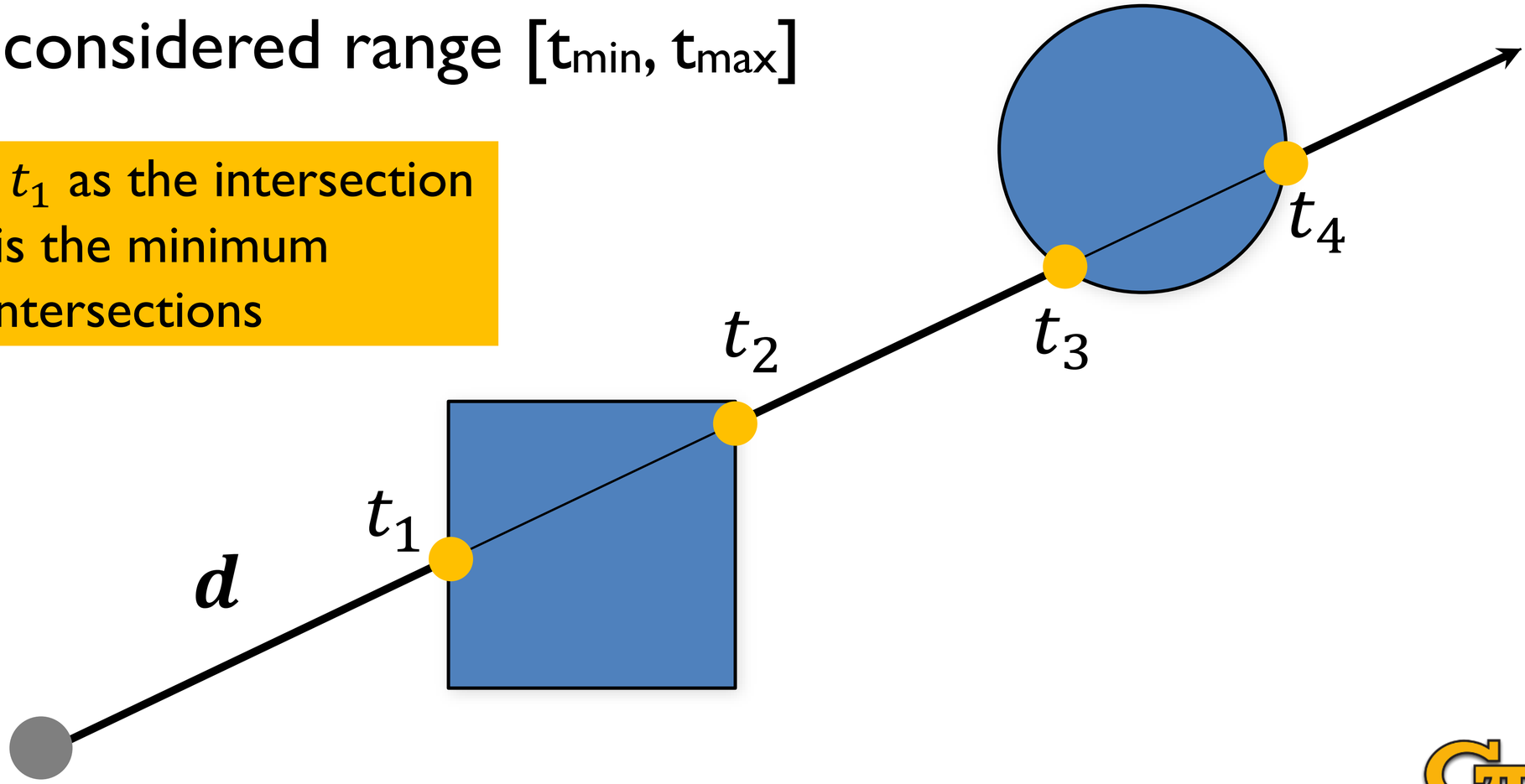
- Union
- Difference
- Intersection

Image Credit: Geoffrey Yu
https://www.geoffreyyu.com/blog/2017/09/16/graphics-at-the-university-of-waterloo

Intersecting Many Shapes

- **Idea**: test intersect with each primitive and then pick the closest intersection
  - Only within considered range [t_{min}, t_{max}]

Will return t_1 as the intersection because it is the minimum among all intersections

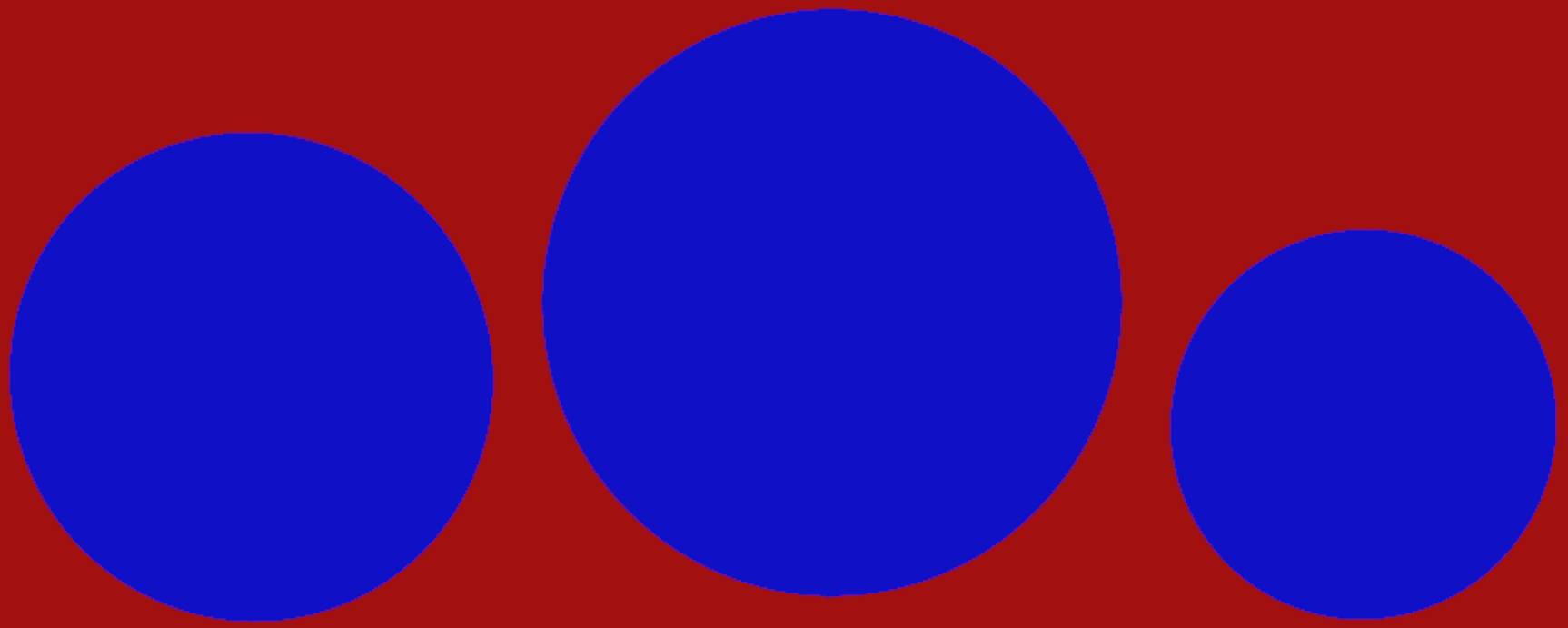


Ray-Many-Objects Intersection Pseudocode

- **Input**: Ray r, list of objects list
- **Output**: hit
- **Algorithm**:

```
hit = noHit;
t_min = MAX
for each object obj in the list
    calculate t using ray-object intersection
    if t < 0
        continue
    if t < t_min
        t_min = t
        update hit according to t_min and the current object
return hit
```

Find the smallest (positive) t and return its hit

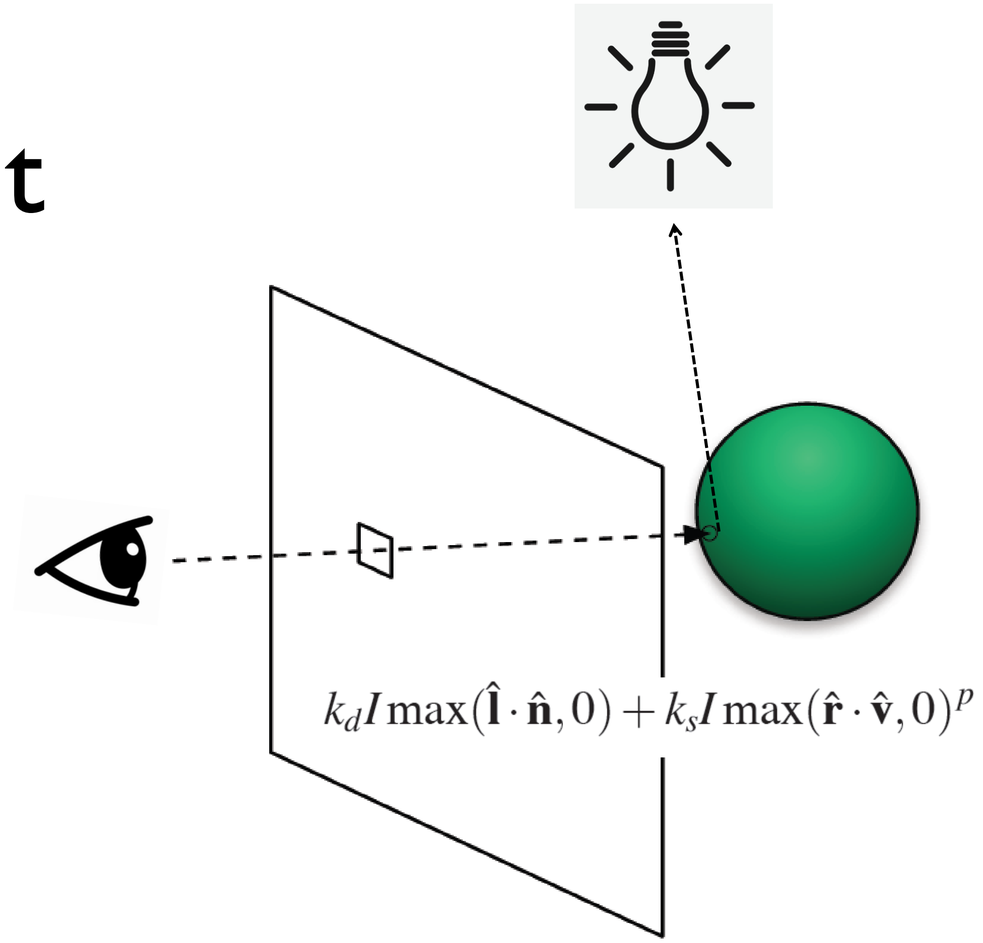


After implementing ray-sphere and ray-plane Interactions:
We will put lighting and shading effects next

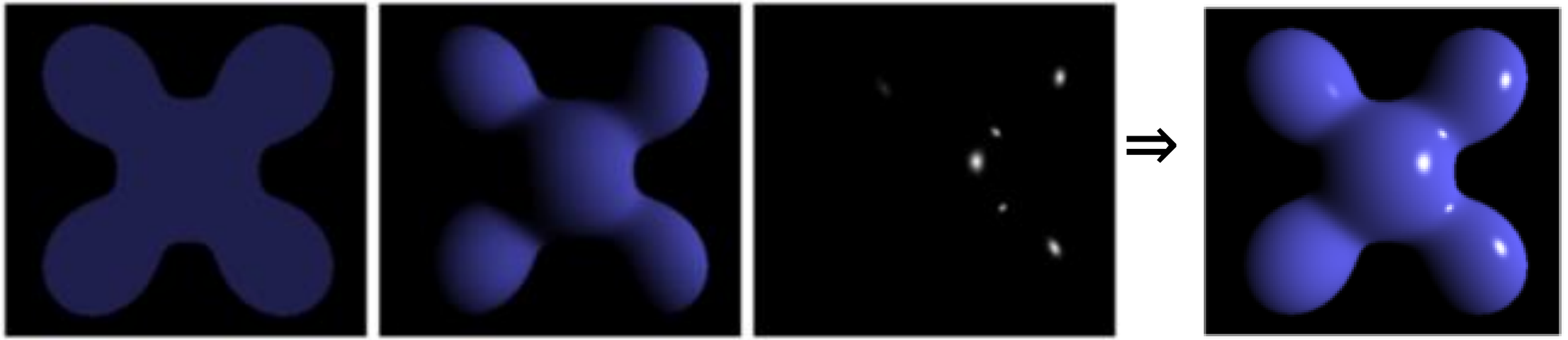
Lighting & Shading with Ray Tracing

Shading the Intersection Point

- Similar to the OpenGL shading model: we can use the Phong shading model directly in a ray tracing algorithm
- The shading on each intersection point is the sum of contributions from all light sources



Quick Recap: Phong Shading in GLSL

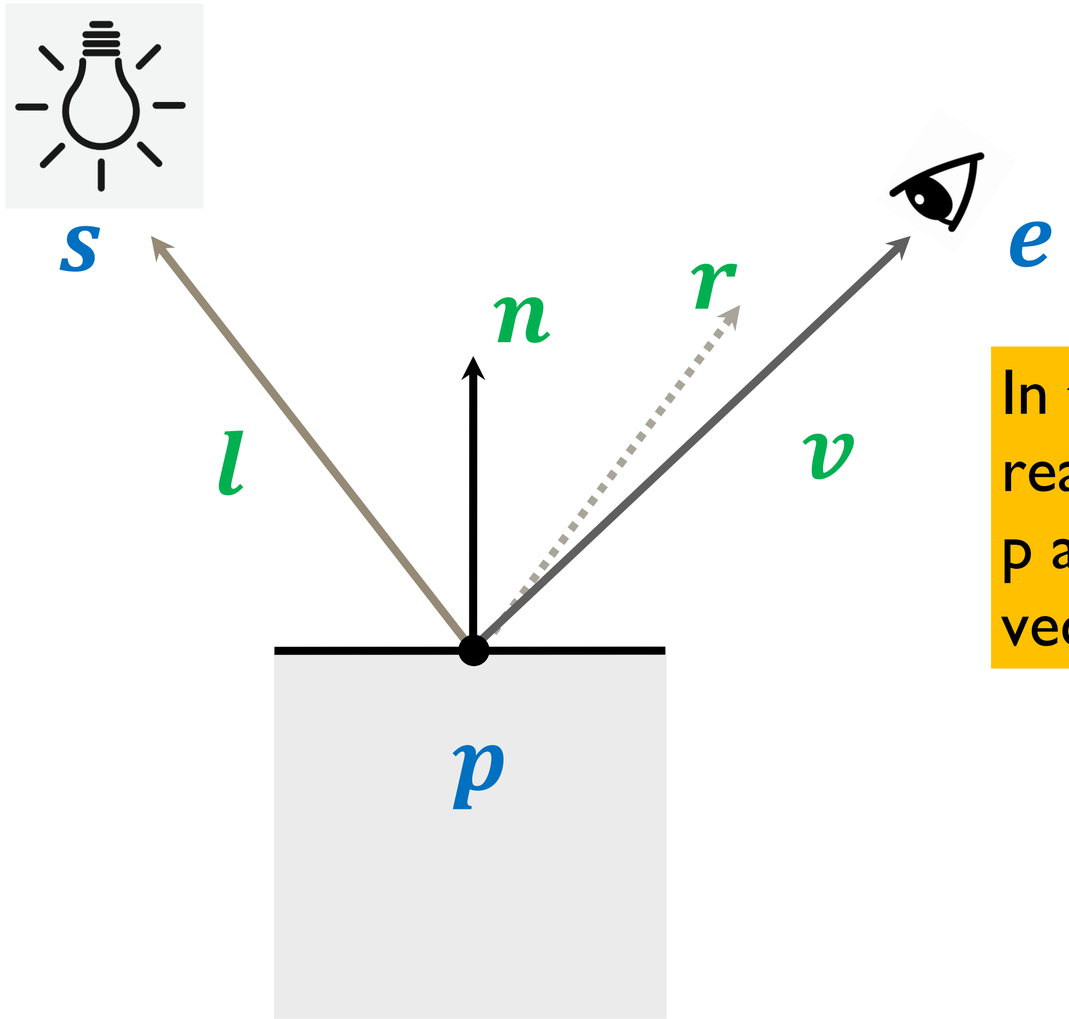


Ambient Diffuse Specular Final Image

$$\boldsymbol{L}_{Phong} = \sum_{j \in lights} (k_a \boldsymbol{I}_a^j + k_d \boldsymbol{I}_d^j \max(0, \boldsymbol{l}^j \cdot \boldsymbol{n}) + k_s \boldsymbol{I}_s^j \max(0, \boldsymbol{v} \cdot \boldsymbol{r}^j)^p)$$

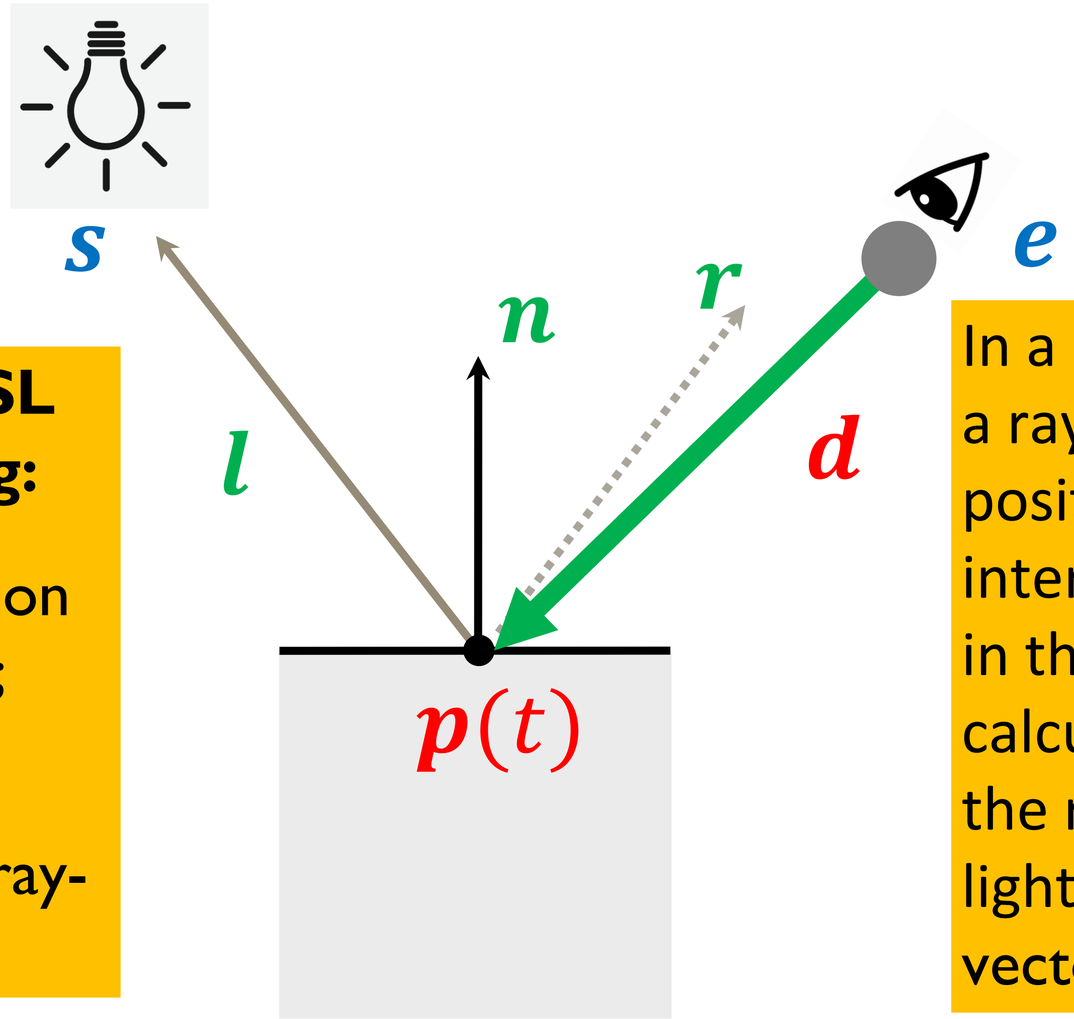
Ambient Diffuse Specular

Quick Recap: Phong Shading in GLSL Shader



In the fragment shader, we read the fragment position p and use it to calculate the vector l and v

Phong Shading in Ray Tracing

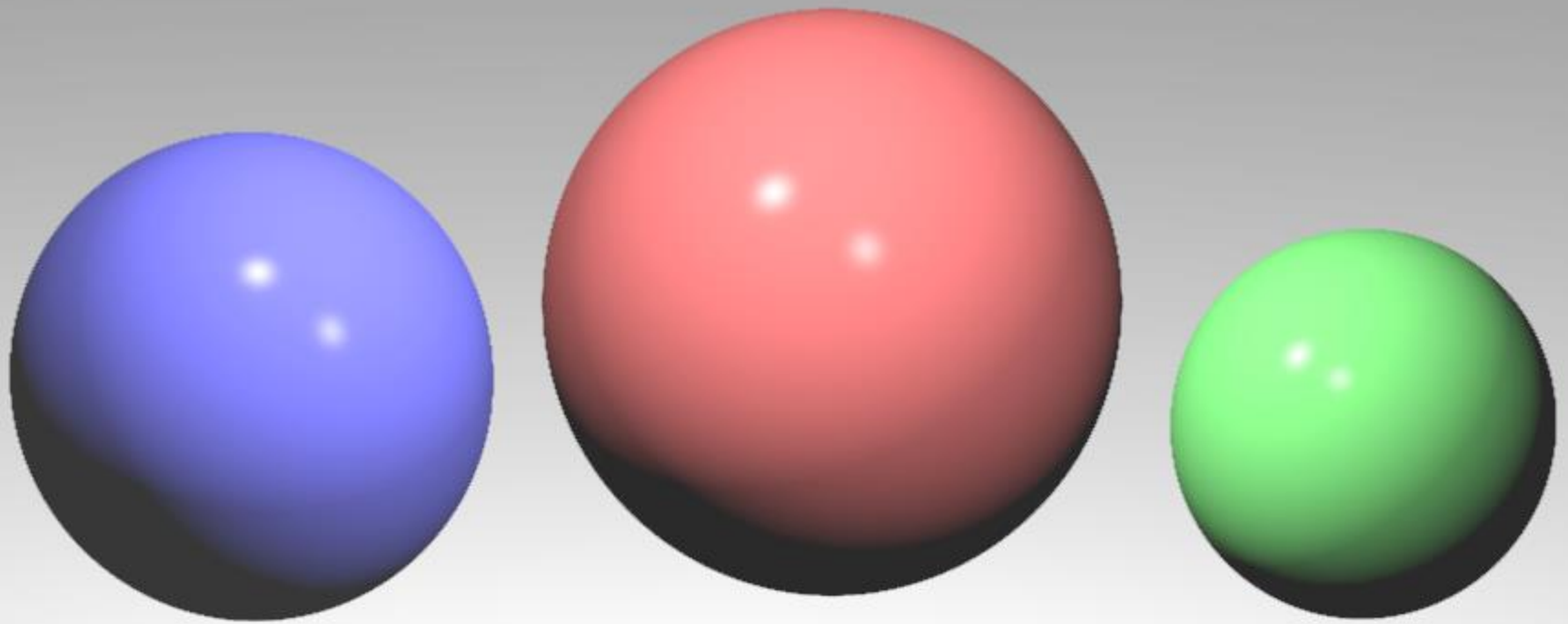


Difference between GLSL Shading and Ray Tracing:

- In GLSL, the surface position is known for each fragment;

- In ray tracing, the surface position is calculated using ray-object intersection.

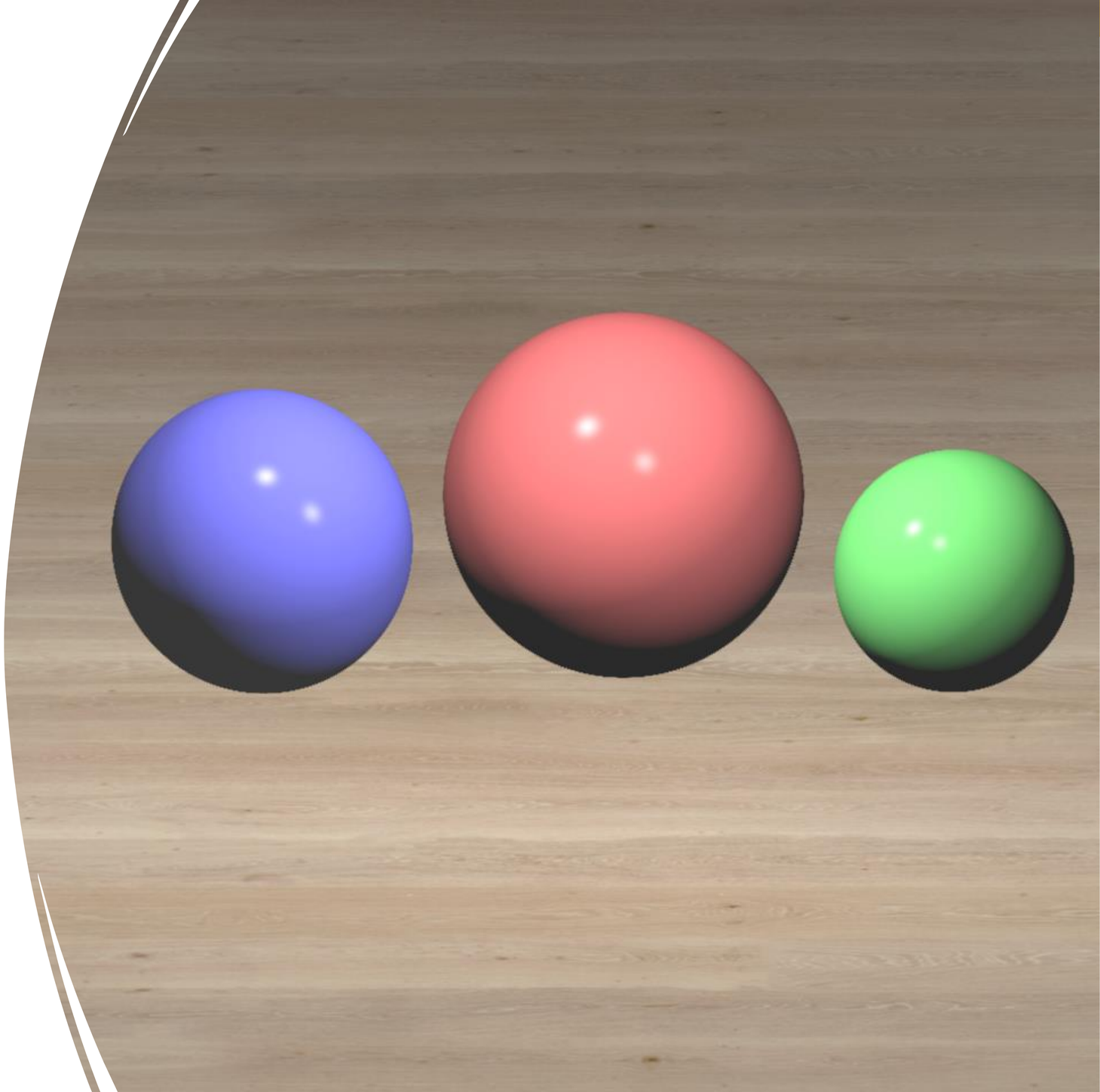
In a ray tracer, we shoot a ray from the eye position and calculate its intersection with objects in the scene; then we calculate the color of the ray based on the lights in the scene using vectors $\boldsymbol{p}$, $\boldsymbol{d}$, and $\boldsymbol{l}$.



Shading the sphere and ground with Phong shading model in Ray Tracing

Putting Texture onto the Object's Surface

- It is straightforward to read a texture and use it to synthesize surface color; the procedure is the same as what we have done previously
- Because the object is represented as an implicit function in ray tracing (we don't have a mesh, and therefore cannot store its uv coordinates), we need to calculate the uv given an input coordinate input



Shadow

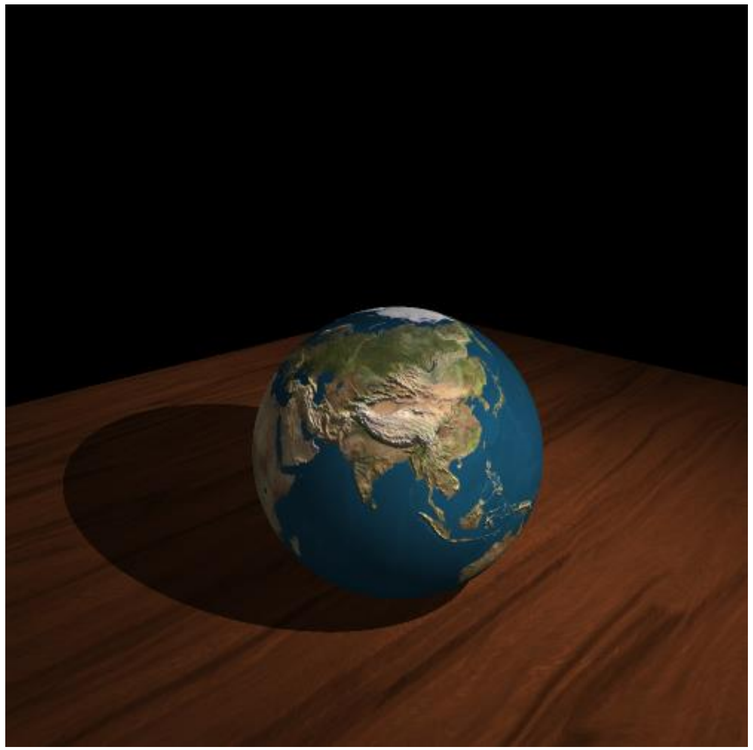
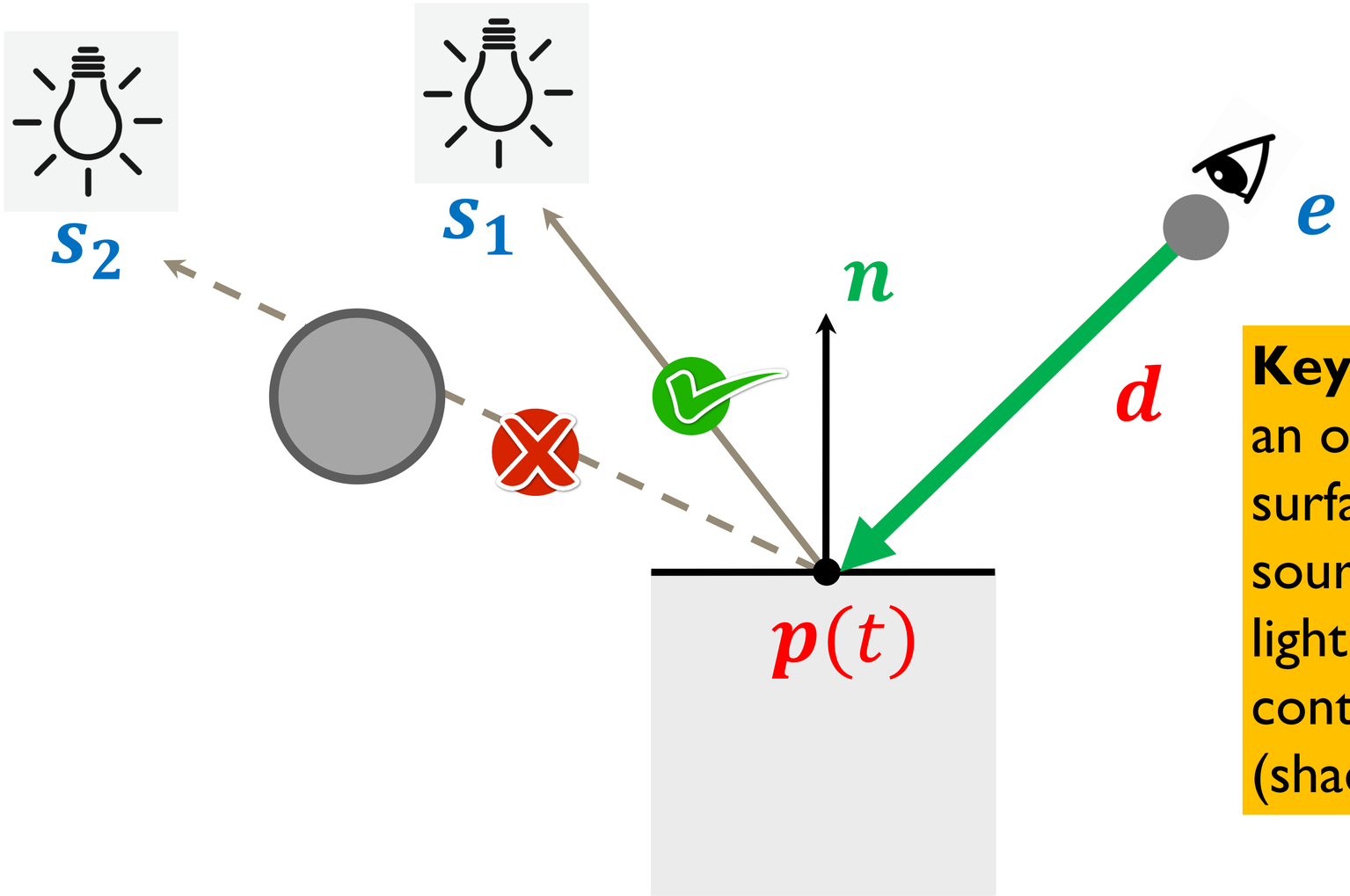


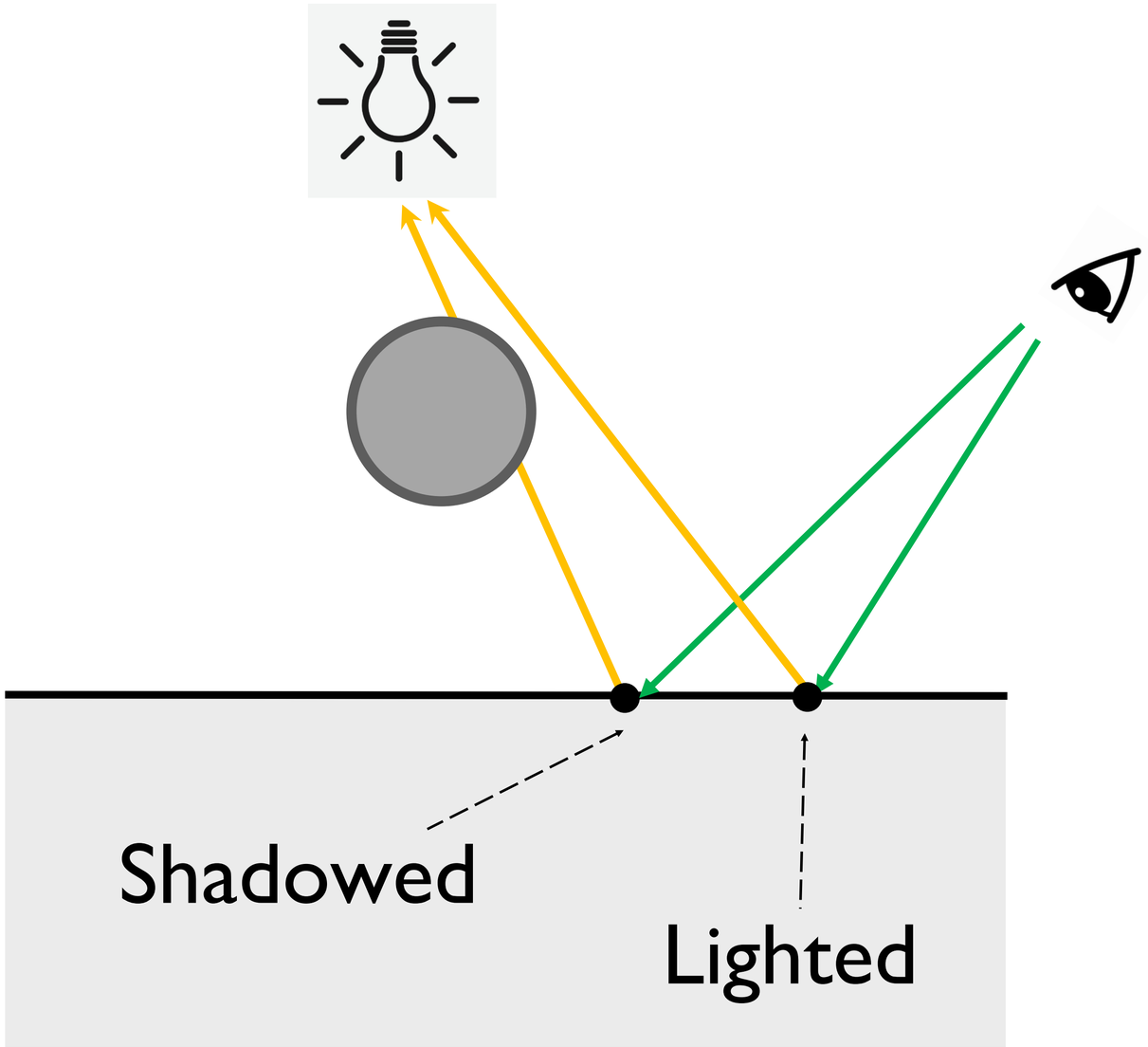
Image Credit: Geoffrey Yu
https://www.geoffreyyu.com/blog/2017/09/16/graphics-at-the-university-of-waterloo

Shadow in Ray Tracing



Key Idea: Check if there is an obstacle between the surface point and the light source: If there is, then the light source does not contribute to the ray color (shadow).

Shadow in Ray Tracing



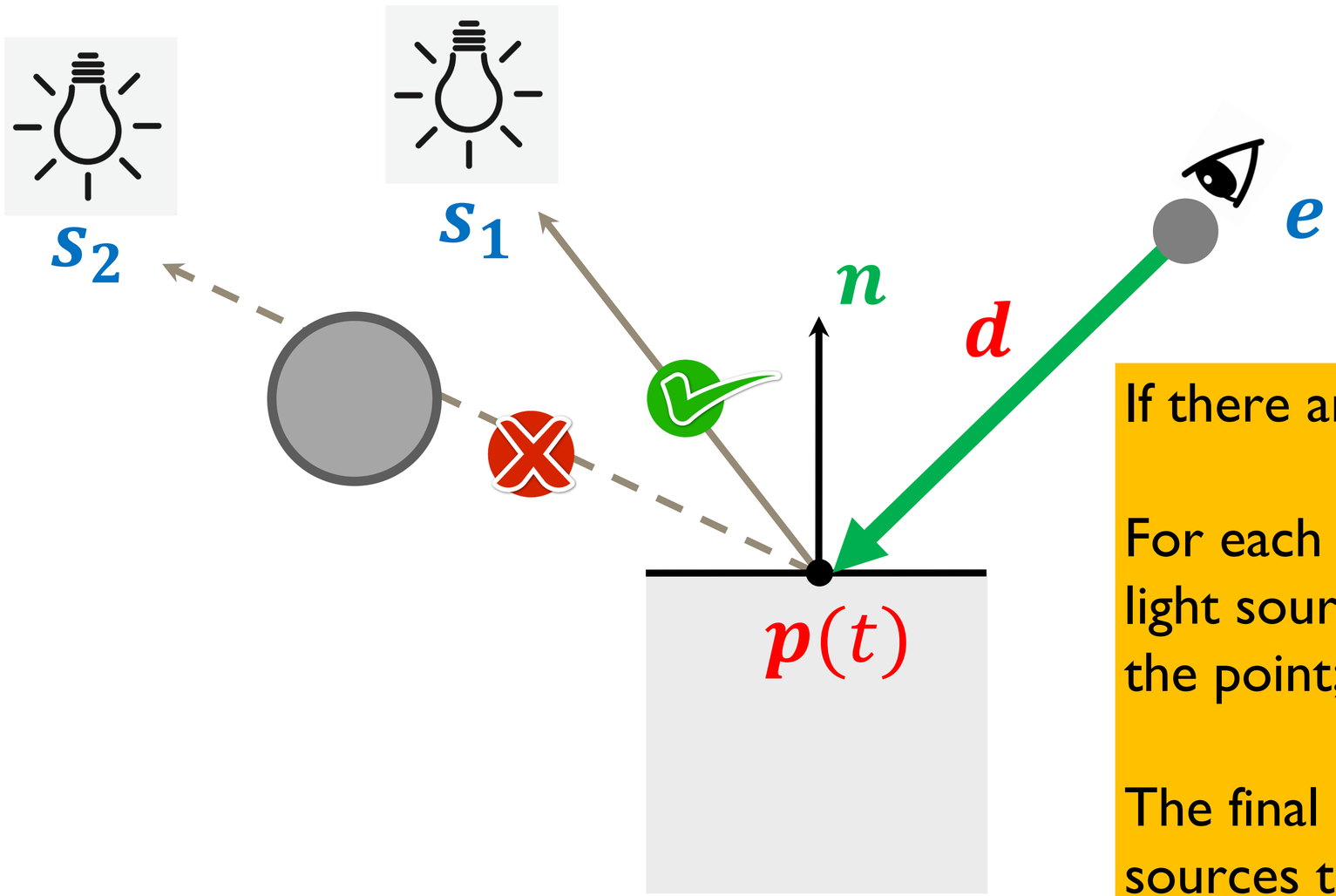
Key Idea: Check if there is an obstacle between the surface point and the light source: If there is, then the light source does not contribute to the ray color (shadow).

How do we check if there is an obstacle between a surface point and a light source?

We detect shadows by **casting a shadow ray** to each light source:

- If the shadow ray returns an intersection, there is a shadow, and we skip this light;
- If the shadow ray does not return an intersection, there is no shadow, we calculate the lighting color from this light source

Calculating Shadow with Multiple Light Sources

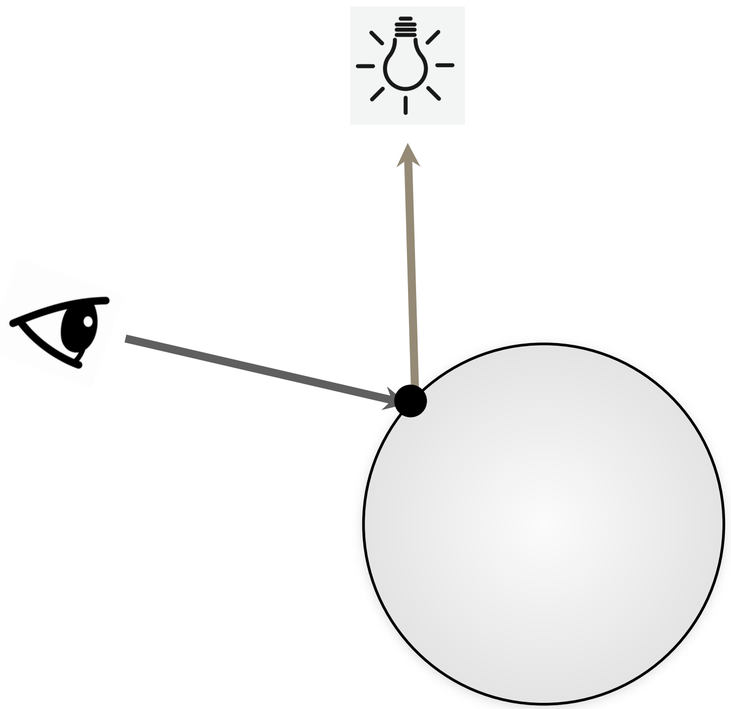


If there are multiple light sources in the scene:

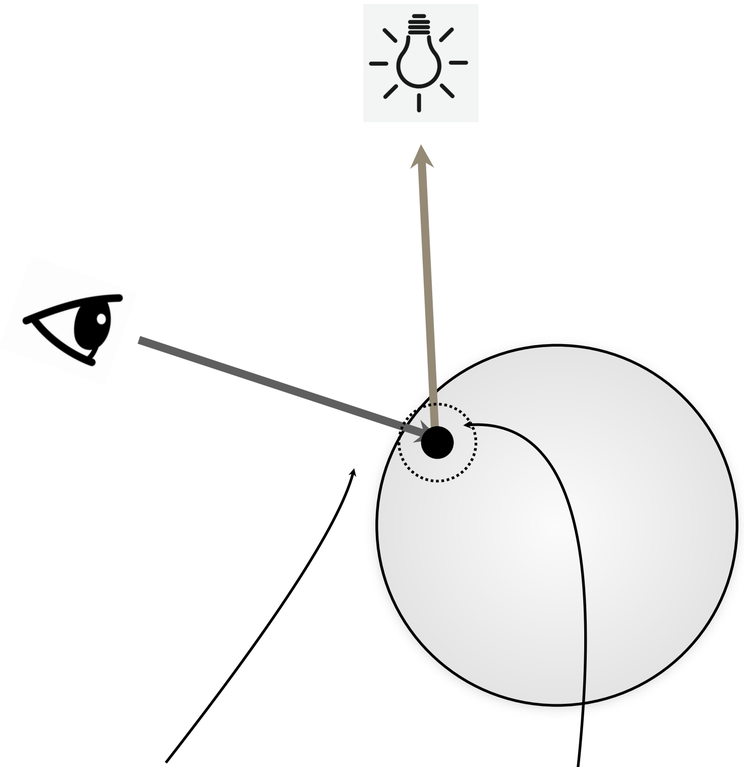
For each point, we shoot a shadow ray to each light source and check if it casts a shadow on the point;

The final color of the ray is the sum of all light sources that do not cast a shadow

Problem: Rounding Errors

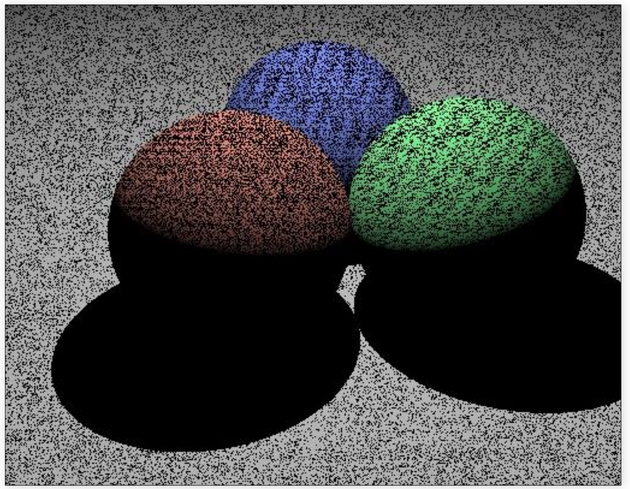


Ideal Situation: Ray hits the surface and bounce off



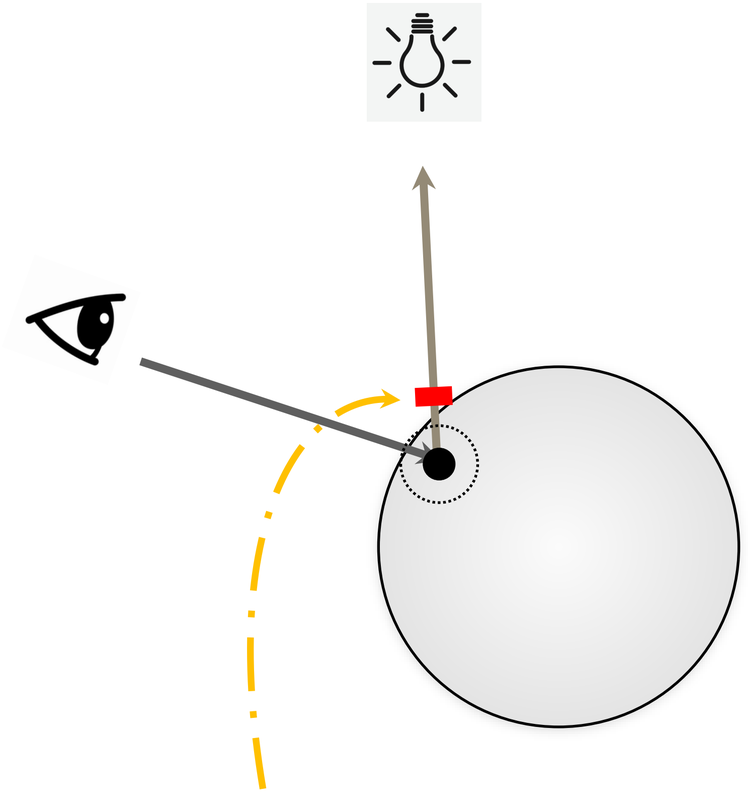
Actual Situation: intersection inside due to floating point error

Issue: ray blocked by self-intersection

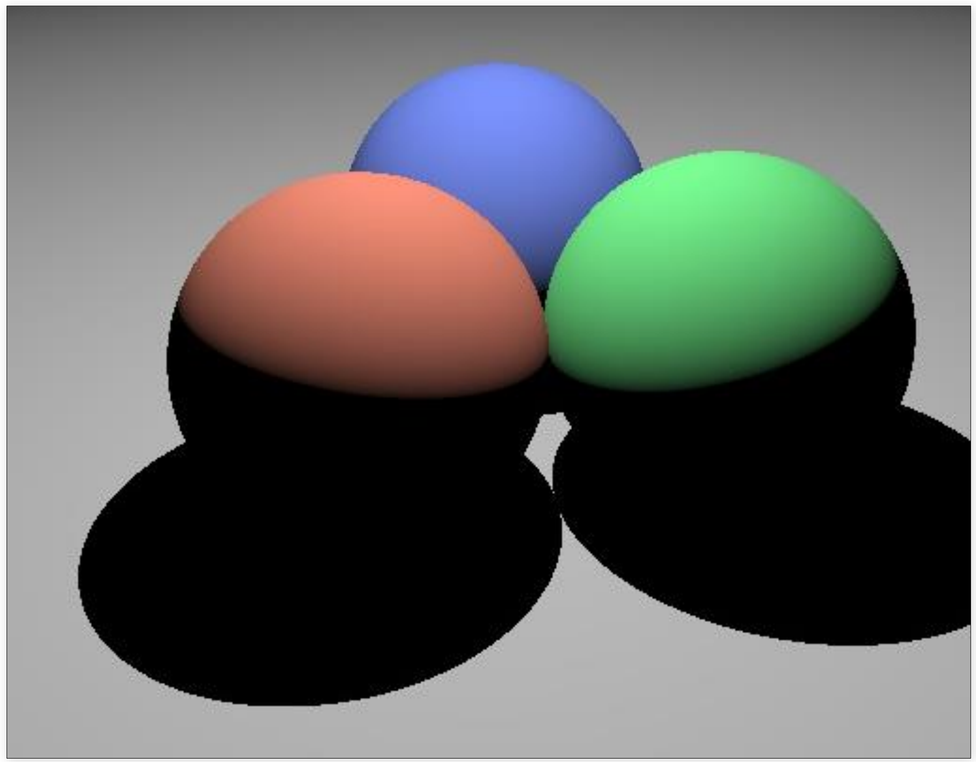


Shadow Rounding Errors

- **Solution**: shadow rays start a tiny distance from the surface



Do this by limiting
the *t* range



Shadow Ray Pseudocode

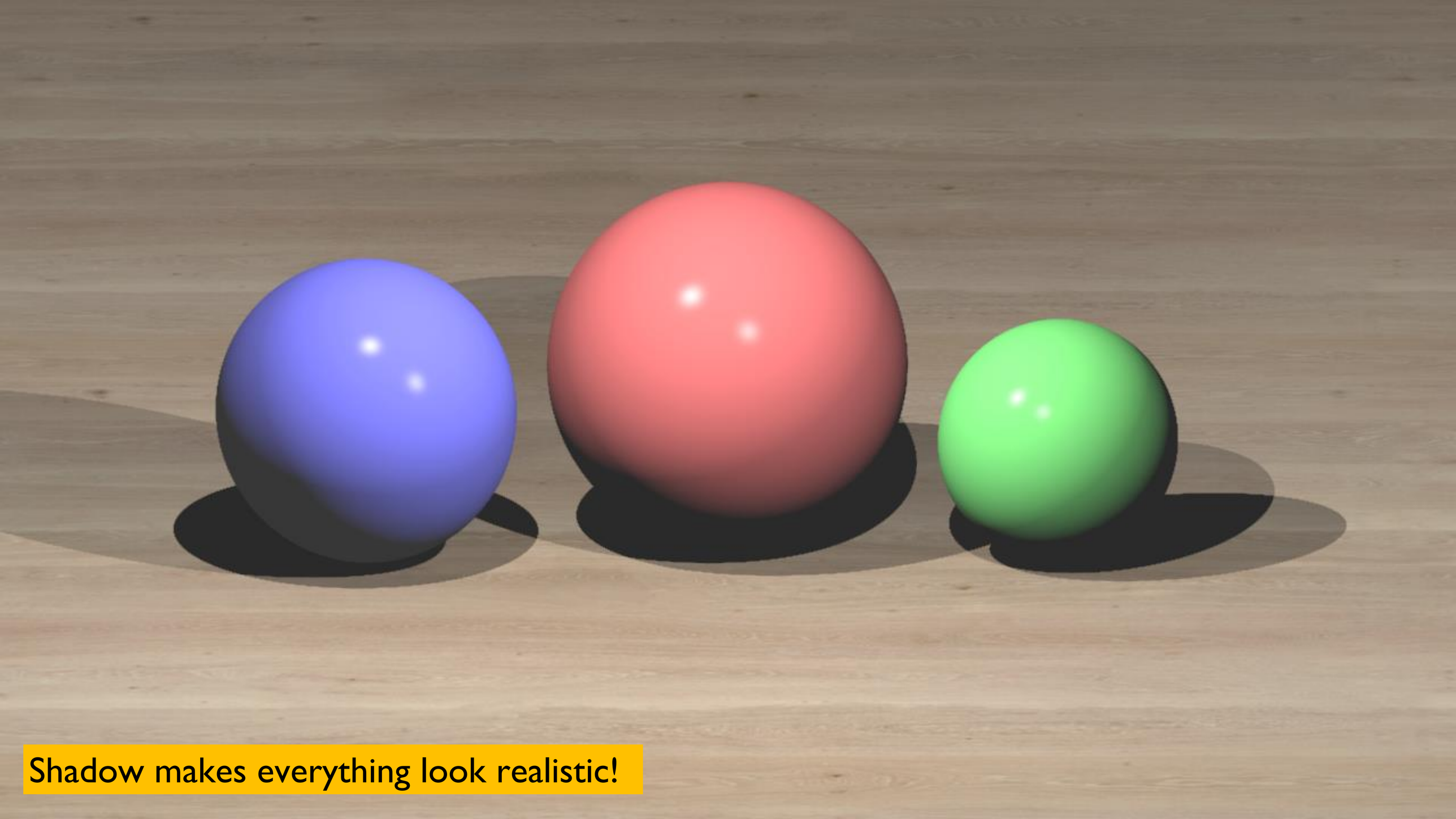
- **Input**: Light light, Hit h
- **Output**: a boolean indicating shadowed or not
- **Algorithm**:

Create a shadow ray starting from the hit point and pointing to the light source (the starting point needs to plus epsilon to avoid self-intersect)

Calculate t_max as the length between the ray origin and light position

Calculate t by calling the ray-object intersection function in the scene

```
if t>0 and t<t_max
        return true
else return false
```



Shadow makes everything look realistic!

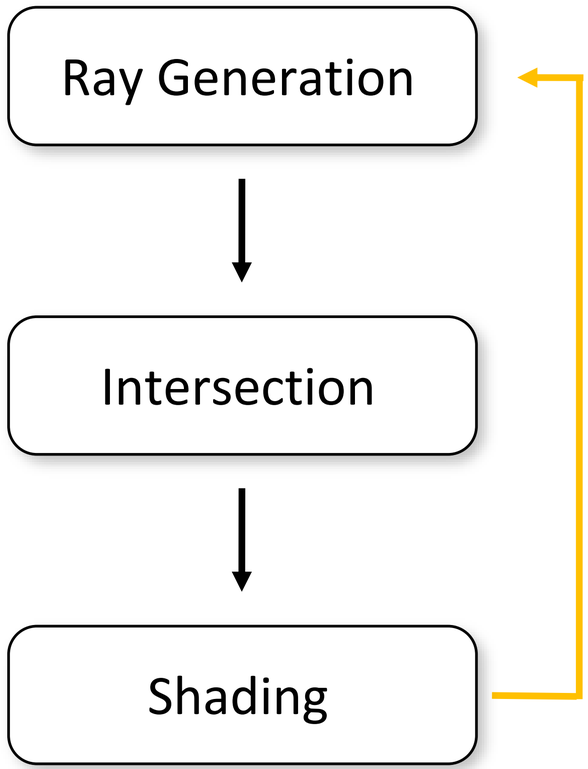
Key Takeaway for a Basic Ray Tracer

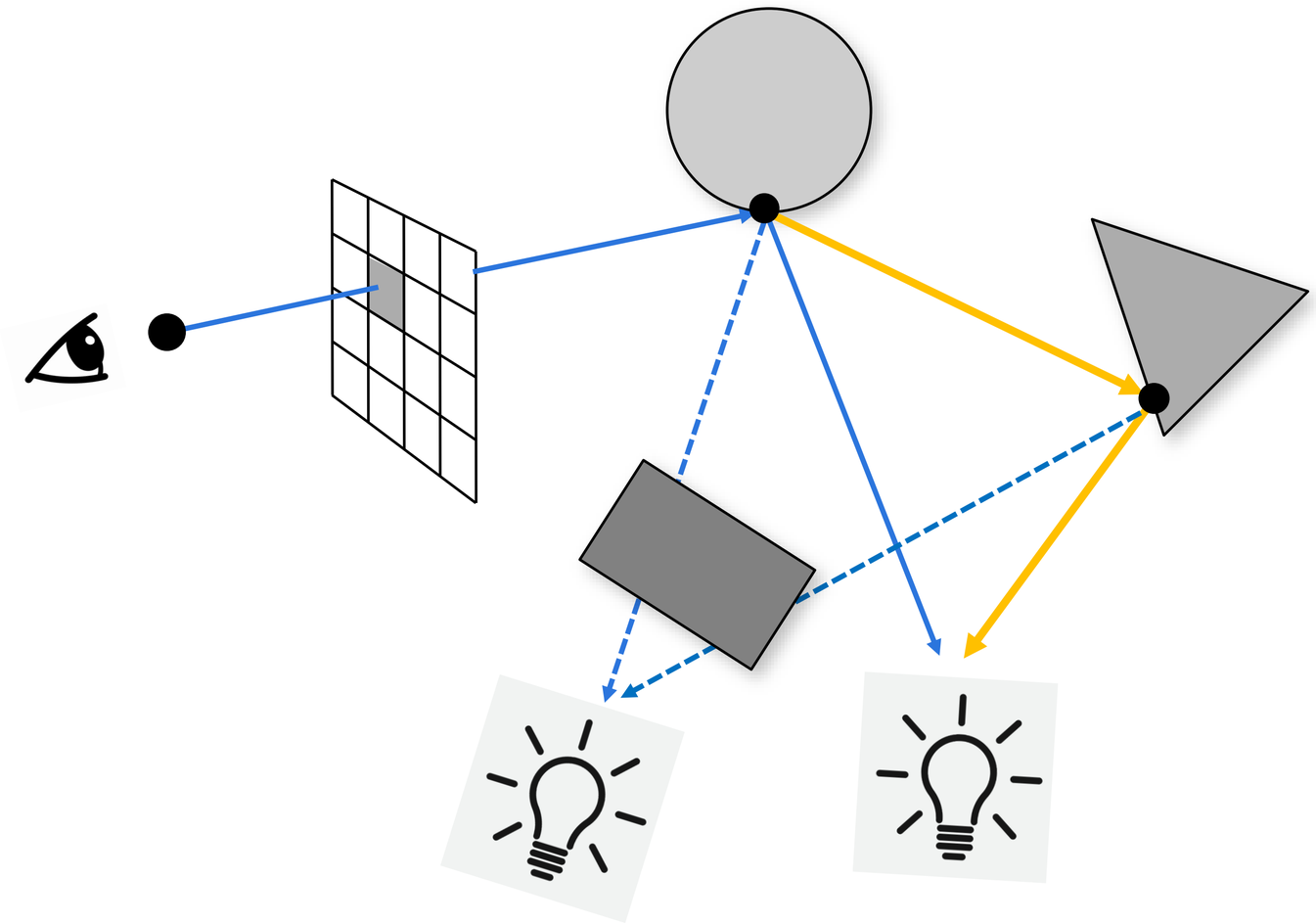
- **Trace a primary ray** from the eye through each pixel and detect the first intersection point with the objects in the scene
- **Trace secondary shadow** rays from the intersection point towards light sources
- Use **Phong shading model** to calculate color by summing the contributions from each light (if the light is visible)

Recursive Ray Tracing

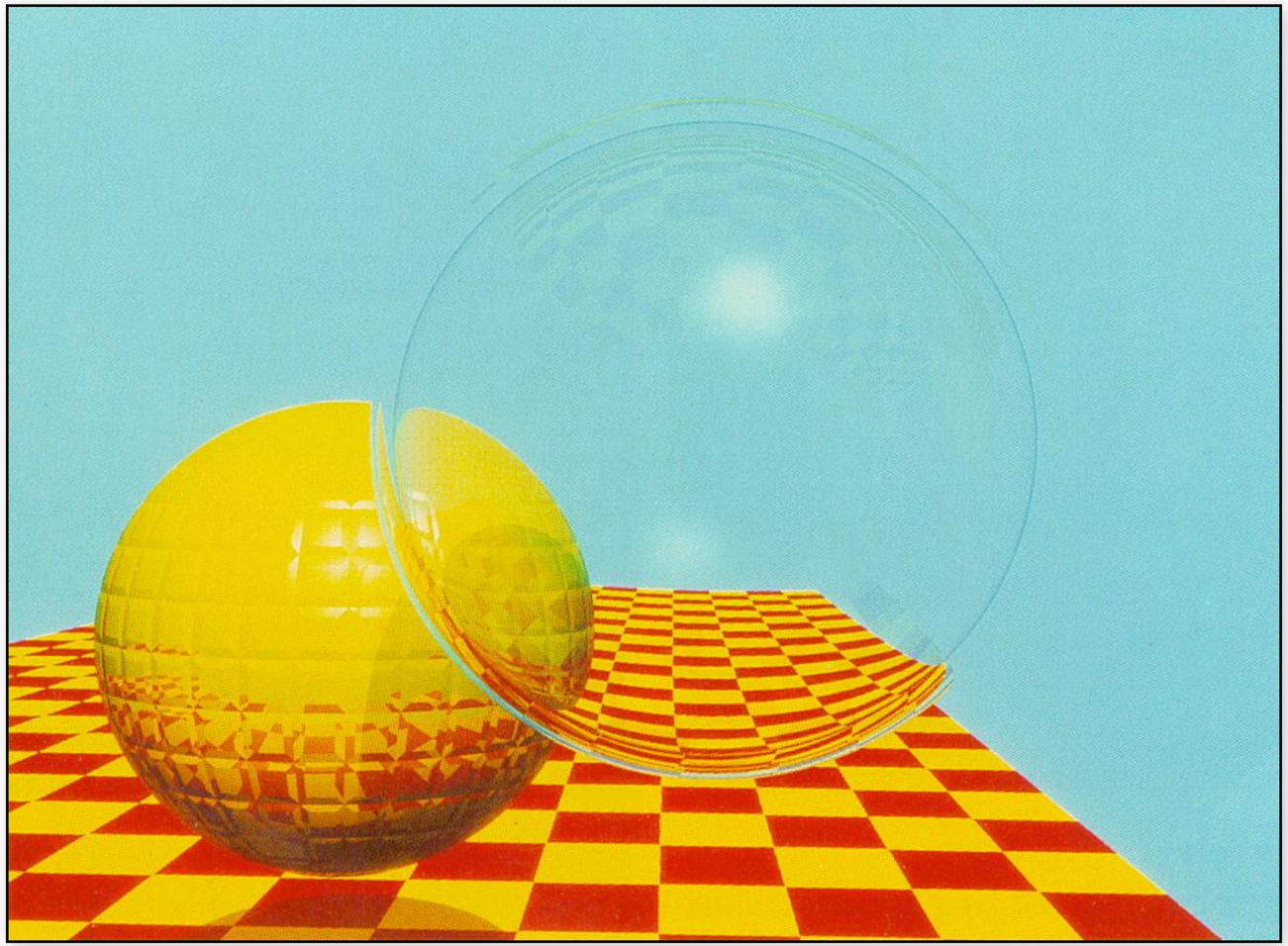

Soumrak

Recursive Ray Tracing Pipeline





First Recursive Ray Tracing Image



Whitted (1979)

Recursive Ray Tracing

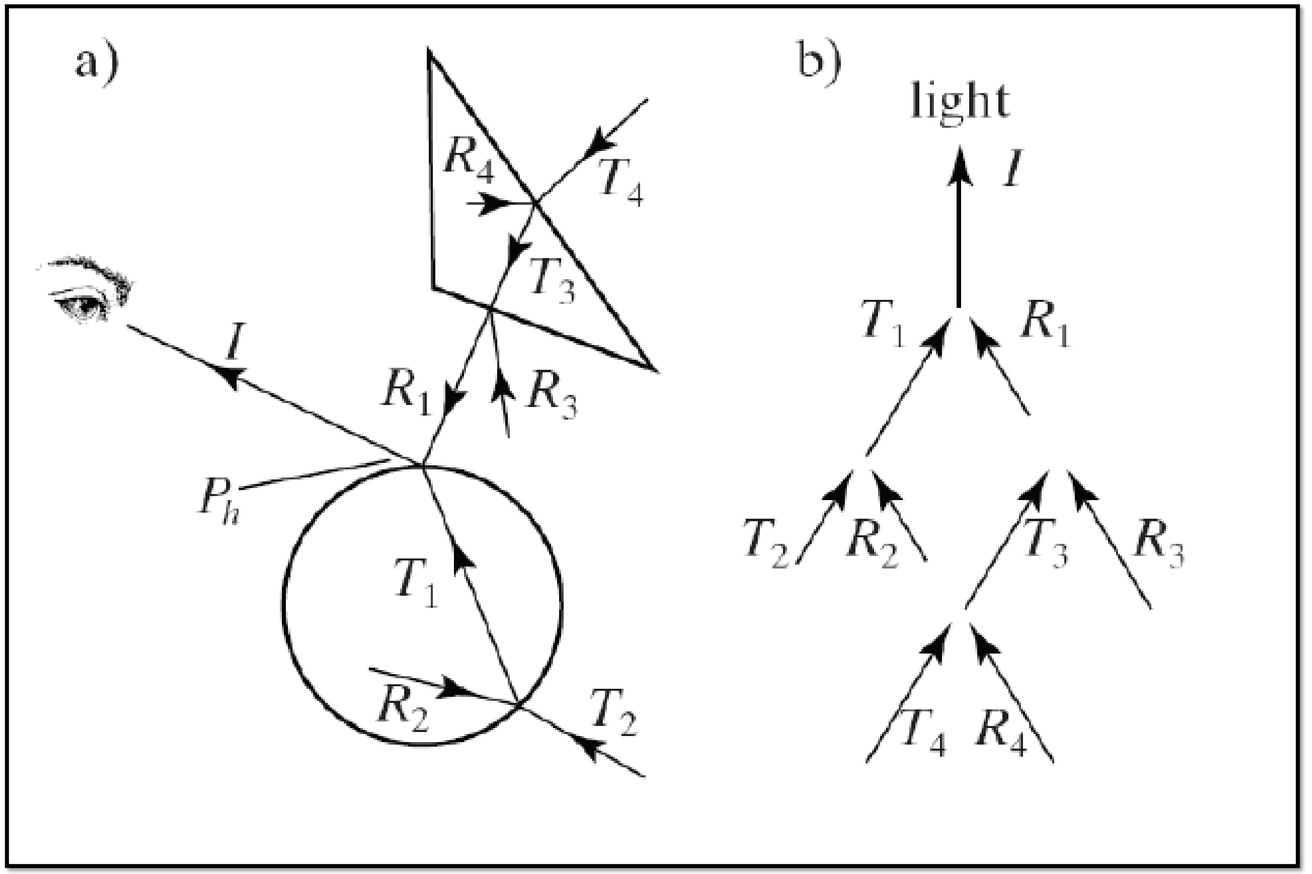
- **Key Idea:** Light at a point may not just come directly from light sources:
  - Light can bounce off other objects (reflection)
  - Light can pass through objects (transparency/refraction)
- Need to trace more rays to look for more lighting information
  - Send secondary rays from the intersection point
  - Recursively compute the color for these secondary rays and sum them onto the primary ray
- Light equation becomes:

$$\boldsymbol{L}_0 = \sum_{j \in lights} \left(k_a \boldsymbol{I}_a^j + k_d \boldsymbol{I}_d^j \max\left(0, \boldsymbol{l}^j \cdot \boldsymbol{n}\right) + k_s \boldsymbol{I}_s^j \max\left(0, \boldsymbol{v} \cdot \boldsymbol{r}^j\right)^p\right) + \boldsymbol{I}_{reflection} + \boldsymbol{I}_{refraction}$$

direct illumination **recursive reflection** **recursive refraction**

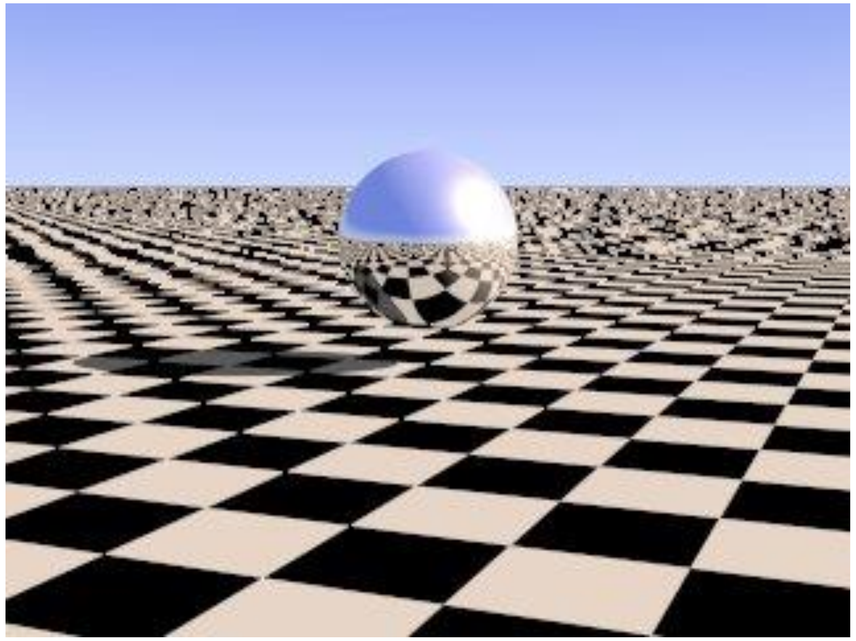
Ray Tree

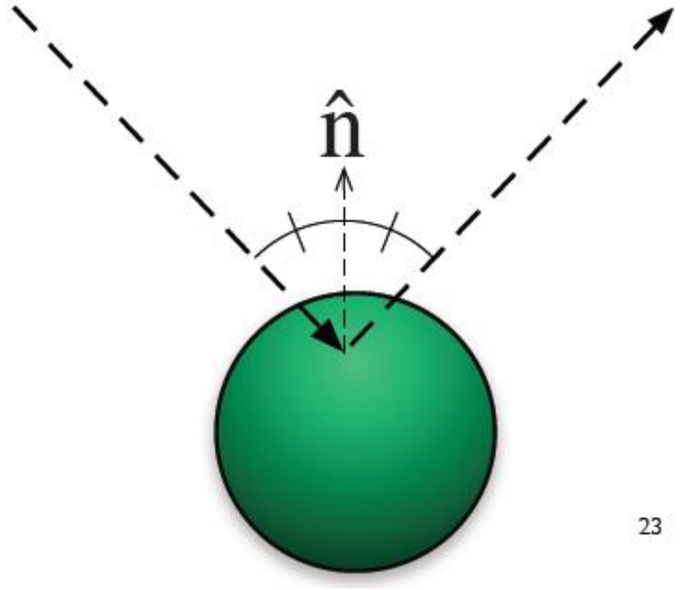
- Each reflected or refracted ray may spawn their own shadow, reflection, or refraction rays, which is a **recursive process**.
- All such rays together form a ray tree



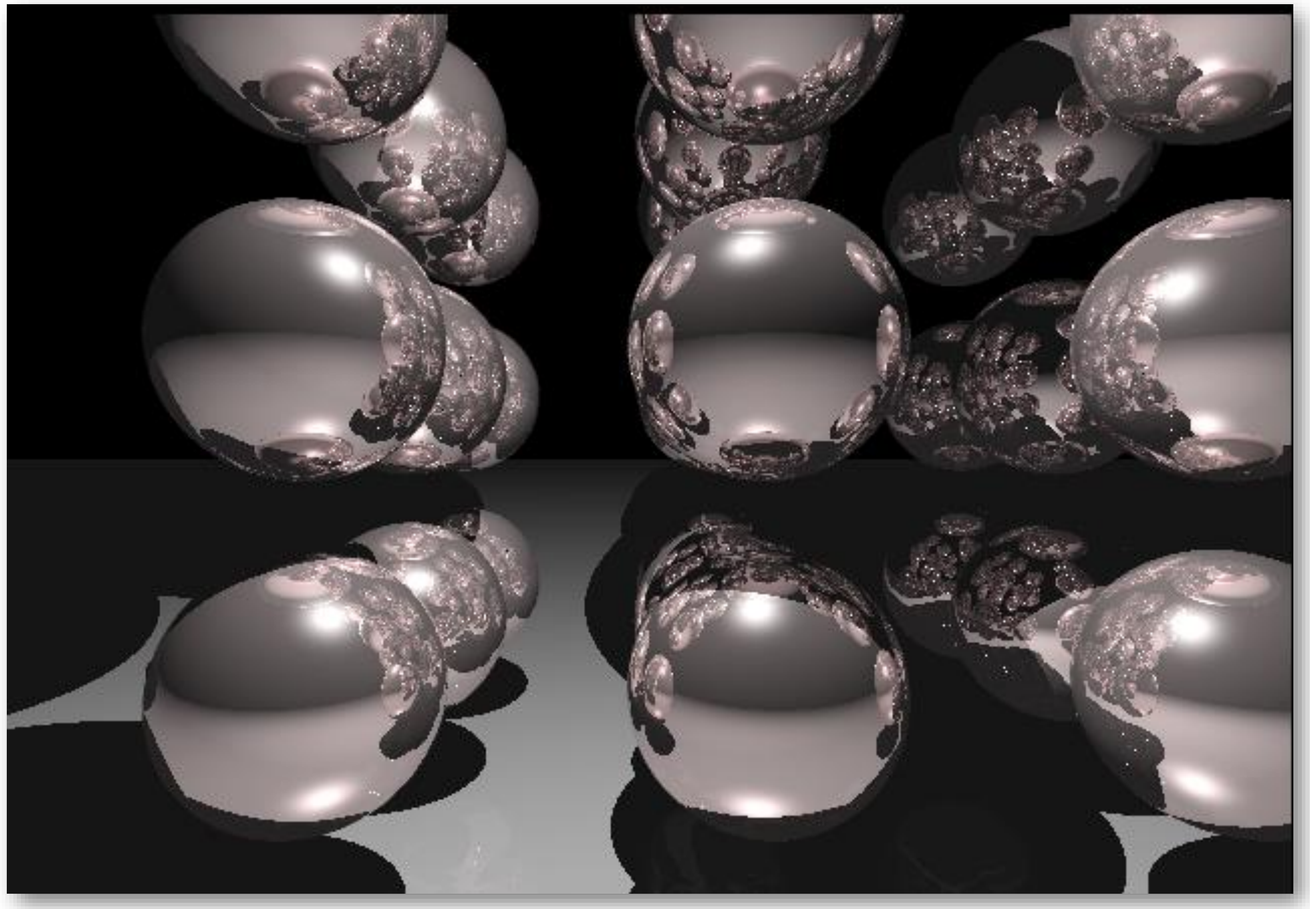
Reflection

- Recursive shading
  - Ray bounces off object
  - Treat bounce rays (mostly) like primary rays from the camera
  - Shade bounce ray and return color
    - Multiply by the reflection coefficient k_r
  - Add reflection ray color to shading at the original point

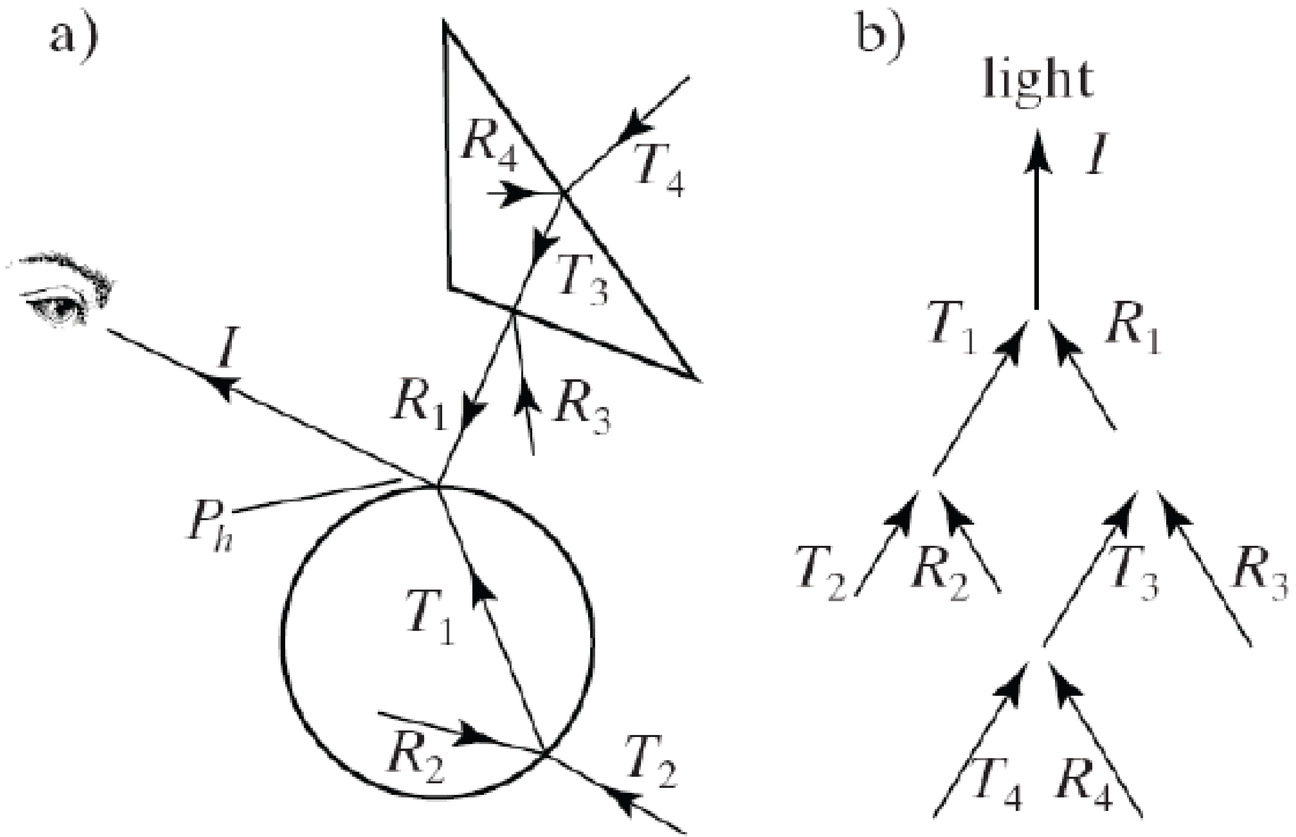




Reflection Example



The Maximum Depth of a Ray Tree

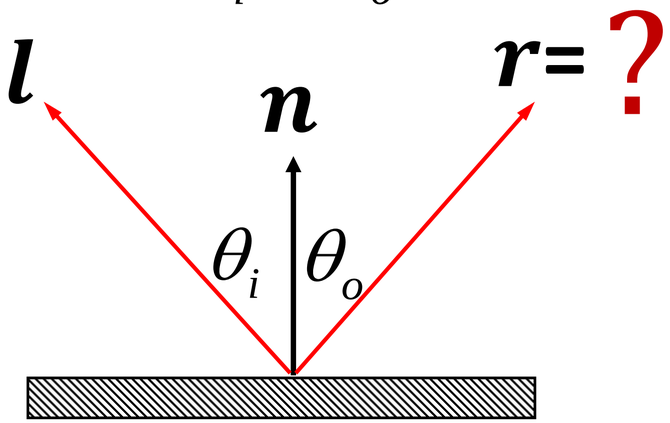


- The maximum depth of a ray tree needs to be carefully chosen:
- If a ray tree is too deep, it gets too bushy affecting the performance of the ray tracer
- Deeper nodes in the tree contribute less to the final image in most cases, can truncate when the contribution is below a threshold
- Be aware of the size of the recursion stack (which usually depends on the hardware) to avoid recursive stack overflow

Quick Recap: How to calculate the mirror reflection vector?

- Mirror - ideally smooth and specular surface
- Law of reflection

$$\theta_i = \theta_o \qquad \boldsymbol{r} = -\boldsymbol{l} + 2(\boldsymbol{l} \cdot \boldsymbol{n})\boldsymbol{n}$$

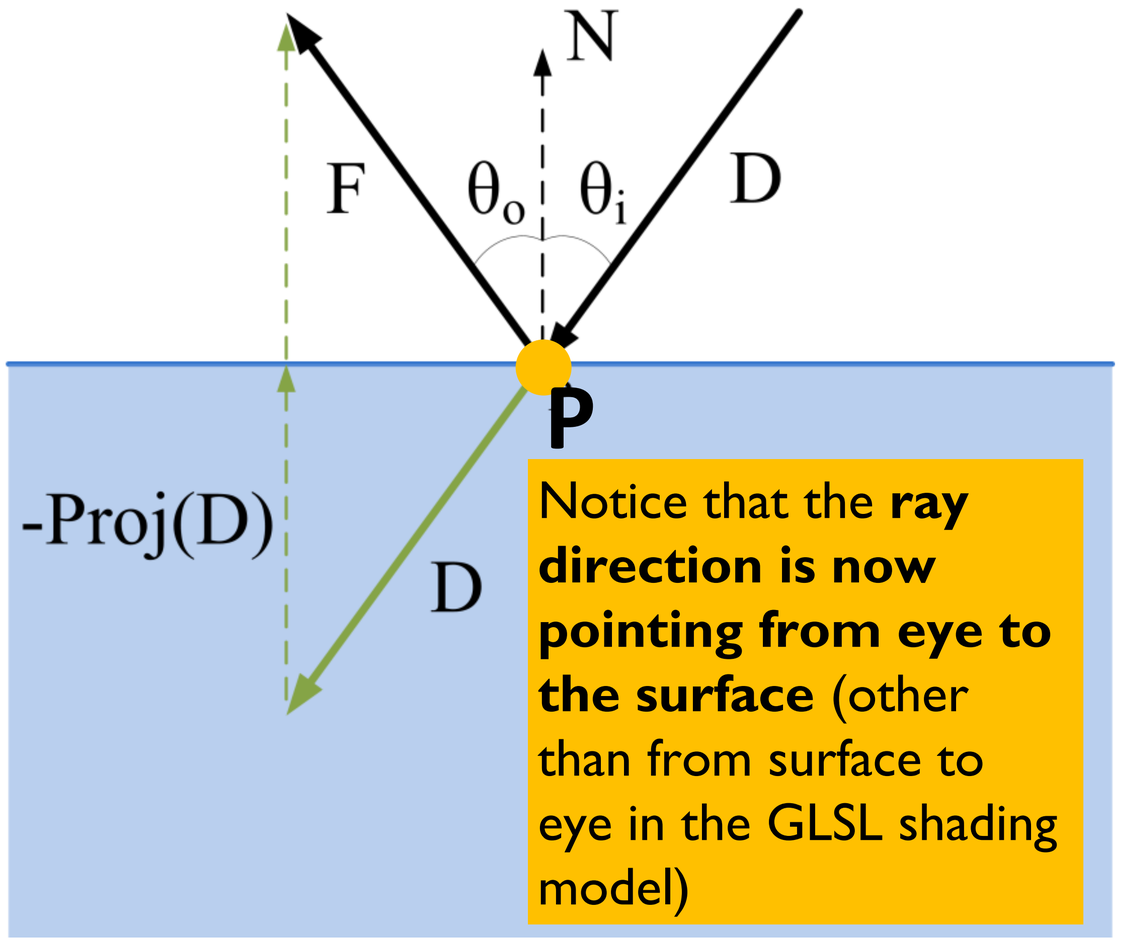


vec3 reflect(vec3 I, vec3 N)
For a given incident vector *I* and surface normal *N* **reflect** returns the reflection direction calculated as *I* - 2.0 * **dot**(*N*, *I*) * *N*.
N should be normalized in order to achieve the desired result.

- Incident light direction $\boldsymbol{l}$, surface normal vector $\boldsymbol{n}$, and reflected light direction $\boldsymbol{r}$ are all coplanar
- In GLSL we implement reflection vector with built-in **reflect()**

Extend the Same Idea to Calculate Reflective Ray

- Given an incident ray, the reflective ray can be computed as:
  - The start point **P** is the intersection point of the incident ray and the surface
  - The direction **F** is the reflected vector of **D** with respect to the surface normal **N**
  - The incident angle θ_i equals the reflection angle θ_o



$$\boldsymbol{F} = \boldsymbol{D} - 2\boldsymbol{Proj}(\boldsymbol{D}) = \boldsymbol{D} - 2(\boldsymbol{D} \bullet \boldsymbol{N})\boldsymbol{N}$$

Recursive Ray Tracing Pseudocode

- Input: Ray recursiveRay, int recursiveDepth
- Output: reflected color
- Algorithm:

set recursiveRay as the primary ray using screen coordinates

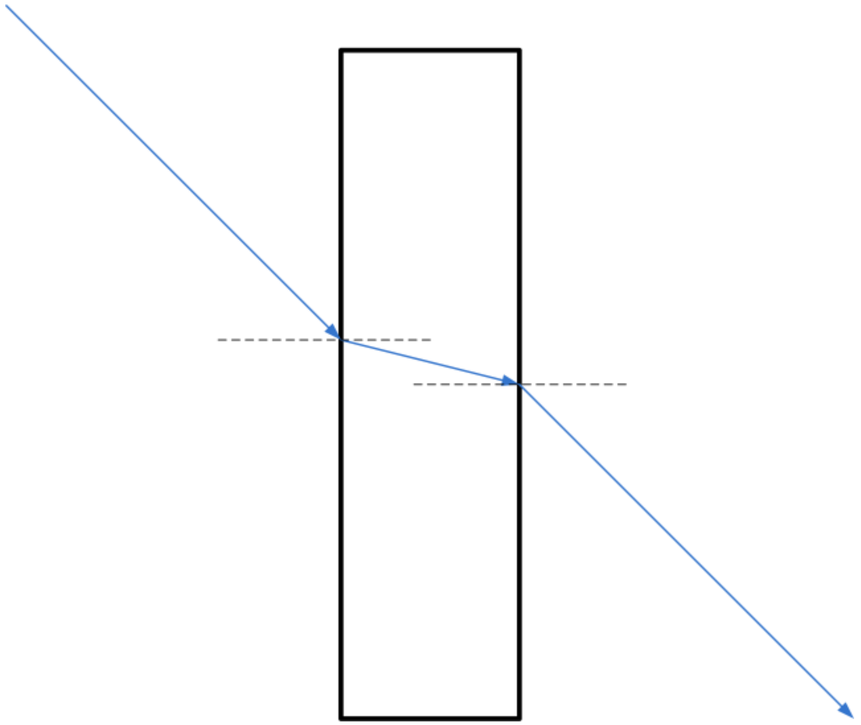
for i = 0 to recursiveDepth

    ray trace the current recursiveRay and accumulate its color

    update recursiveRay as the reflected ray on the current surface


This is the final effect of our A7 rendering!

Refraction



- Trace a refracted (transmitted) ray if the object is transparent
  - Ray passes through object
  - Shade transmitted ray and return color
  - Add color to shading at the original point
    - Multiply by the refraction coefficient k_t

Question: Spear Fishing

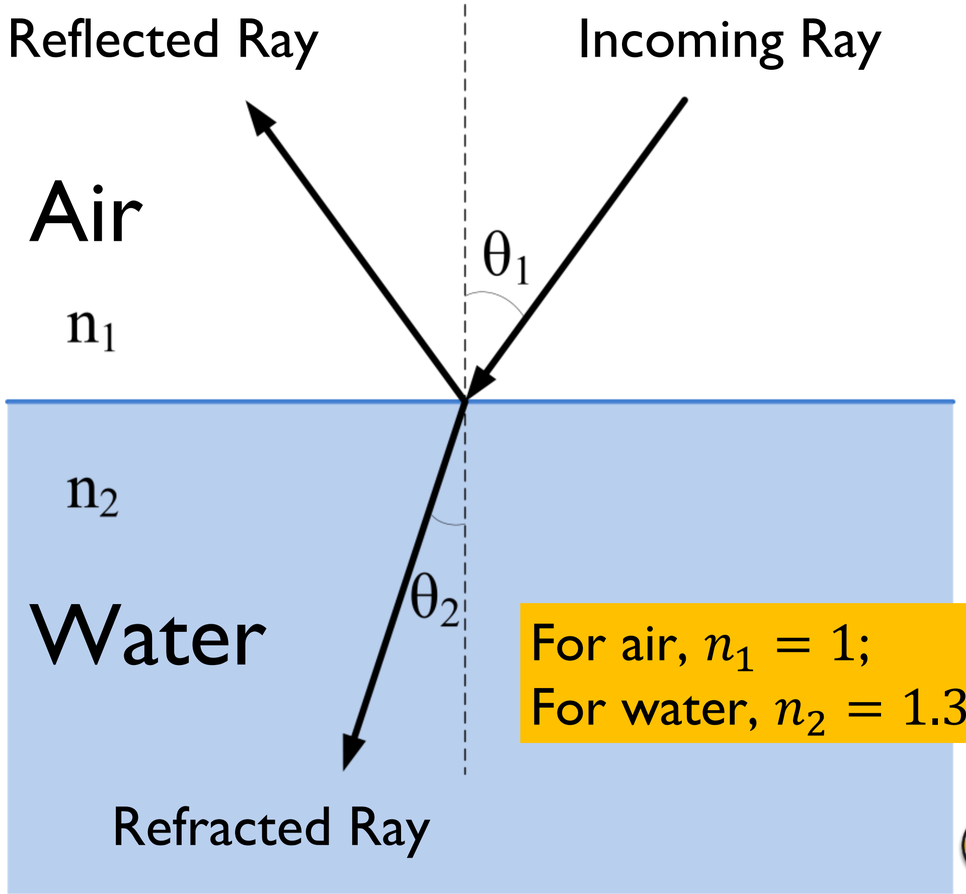
Aim at a lower or higher angle?

Snell's Law

- The relationship between the angles of incidence and refraction for light passing through a boundary between two different isotropic media is:

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{v_1}{v_2} = \frac{n_2}{n_1}$$

- θ_1 and θ_2 are angles of incidence.
- v_1 and v_2 are the phase velocities in two media.
- n_1 and n_2 are their indices of refraction.



Critical Angle

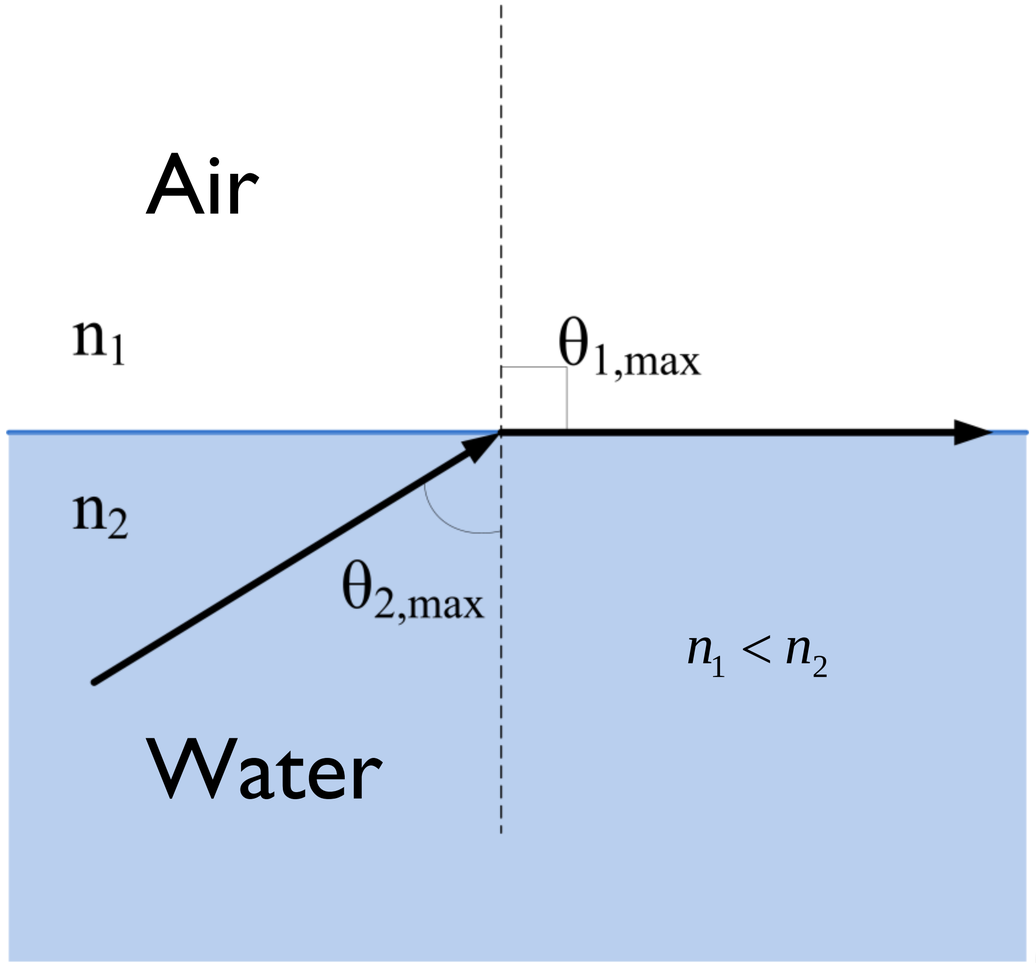
- We can compute the critical angle using Snell's Law:

Assuming $n_1 < n_2$:

$$\sin\theta_2 = \frac{n_1}{n_2}\sin\theta_1$$

$$\theta_{1,\max} = \frac{\pi}{2} \Rightarrow \theta_{2,\max} = \arcsin\left(\frac{n_1}{n_2}\right)$$

The critical angle is **48.7 degrees** for a transmission from water to air

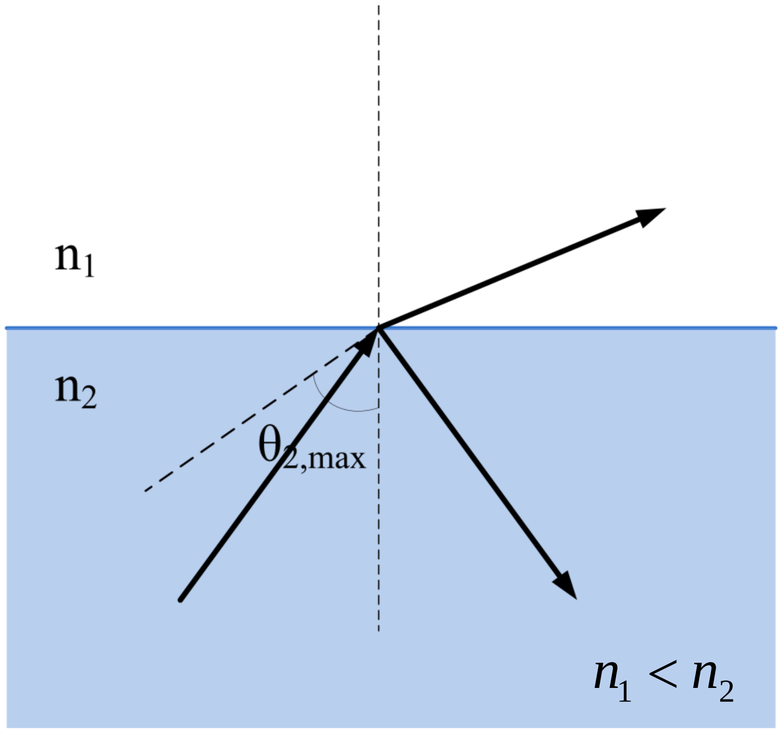


Total Internal Reflection

- When light shoots from a material with a higher refraction index (n_2) to a material with a lower refraction index (n_1), there is no refraction if the incident angle exceeds some critical angle (all the light reflects)

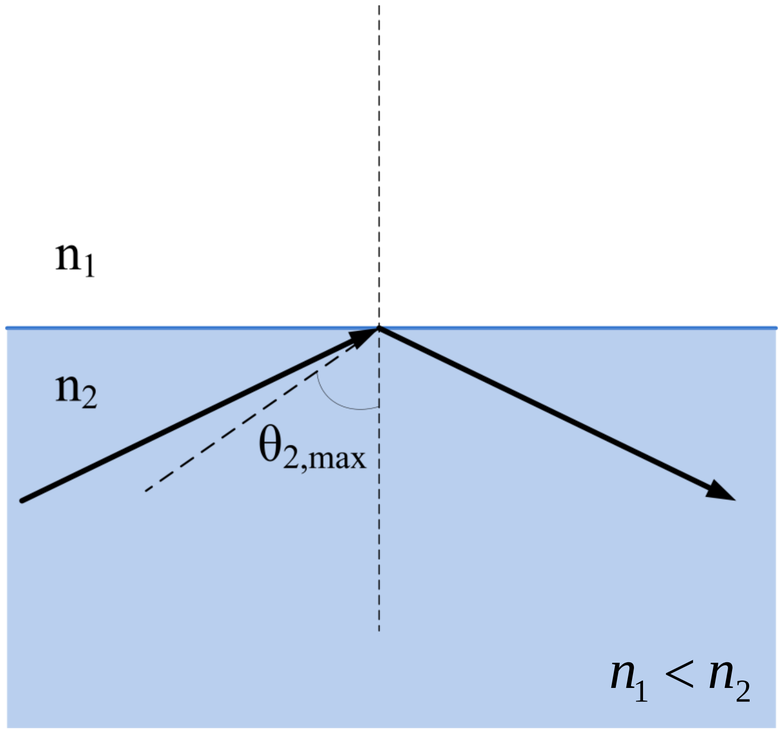
Air

Water



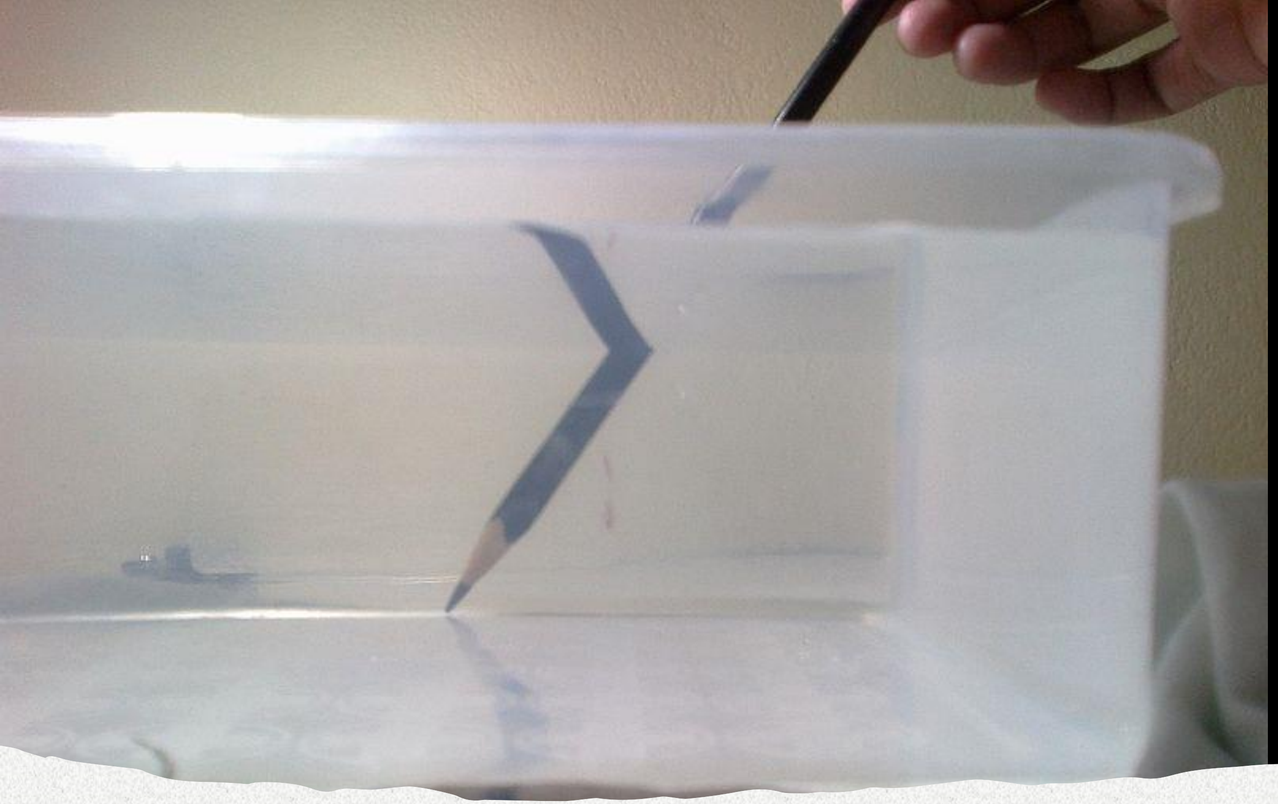
when $\theta_2 < \theta_{2,max}$,

both refraction and reflection



when $\theta_2 < \theta_{2,max}$,

only reflection



Total Internal Reflection

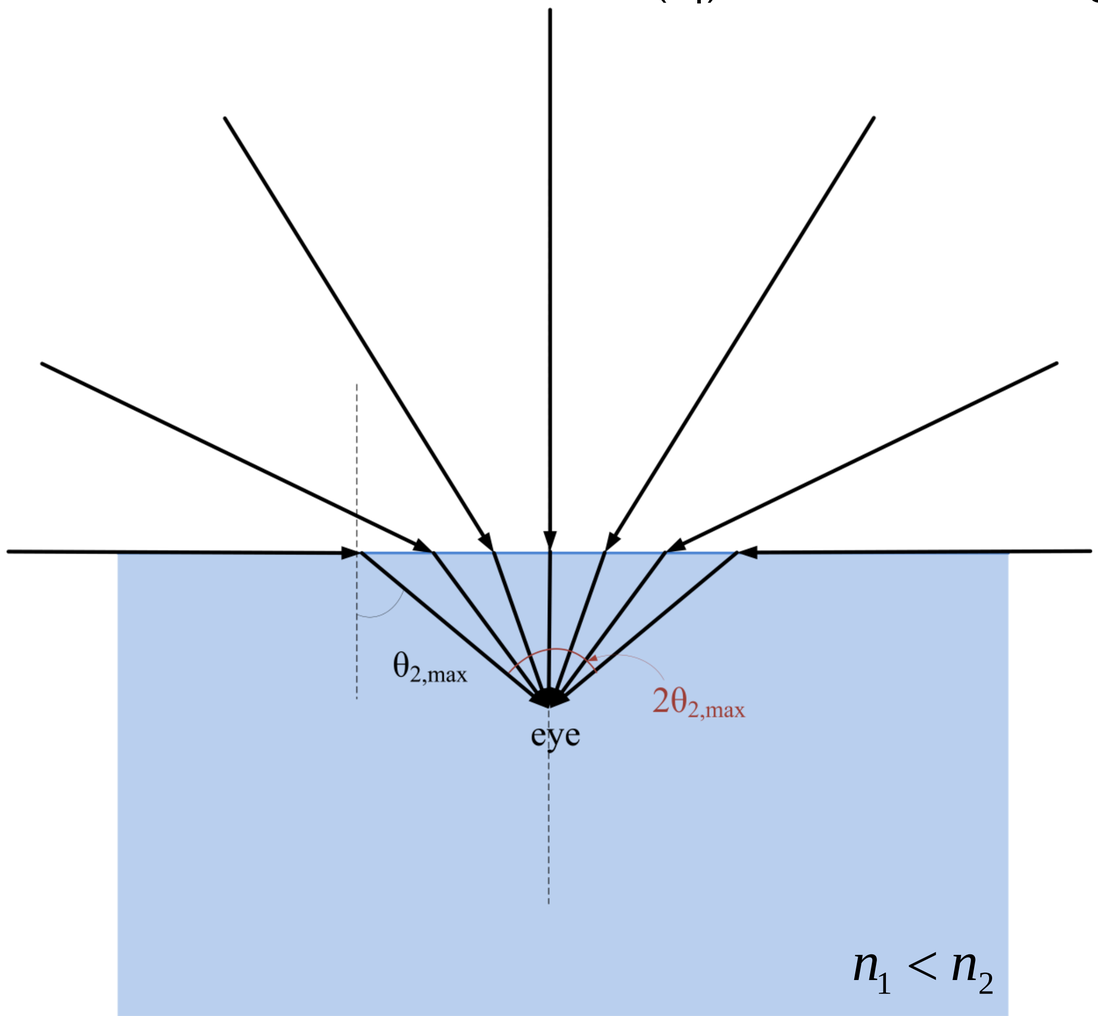
- Total Internal Reflection occurs when light hits a boundary at an angle larger than the critical angle, causing it to reflect entirely back into the original medium instead of refracting through the second medium.
- Total Internal Reflection is responsible for much of the rich appearance of glass and water

Snell's Window

- Light shoots from the media with lower refractive index (n_1) to the one with high refractive index (n_2)

Air

Water



Snell’s Window

This phenomenon allows an underwater viewer to see everything above the surface through a cone of light, caused by the refraction of light as it enters the water, governed by Snell's Law.

http://en.wikipedia.org/wiki/Snell’s_window

Ray Attenuation

- The intensity of a ray will be attenuated when a ray travels through a media.
- The attenuation effect from the media is stronger over longer distances
- Different colors attenuate with different rates
  - Shallow water is clear, deeper water attenuates all the red light and looks blue-green, even deeper water attenuates the green light and looks blue, and eventually one can only see blackish blue or nothing

Beer's Law

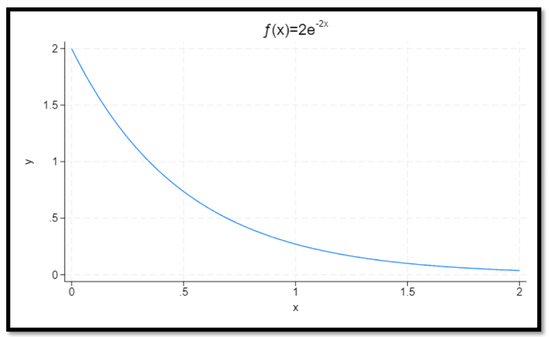
- If the media is homogeneous, the attenuation along the ray can be described using Beer's Law:

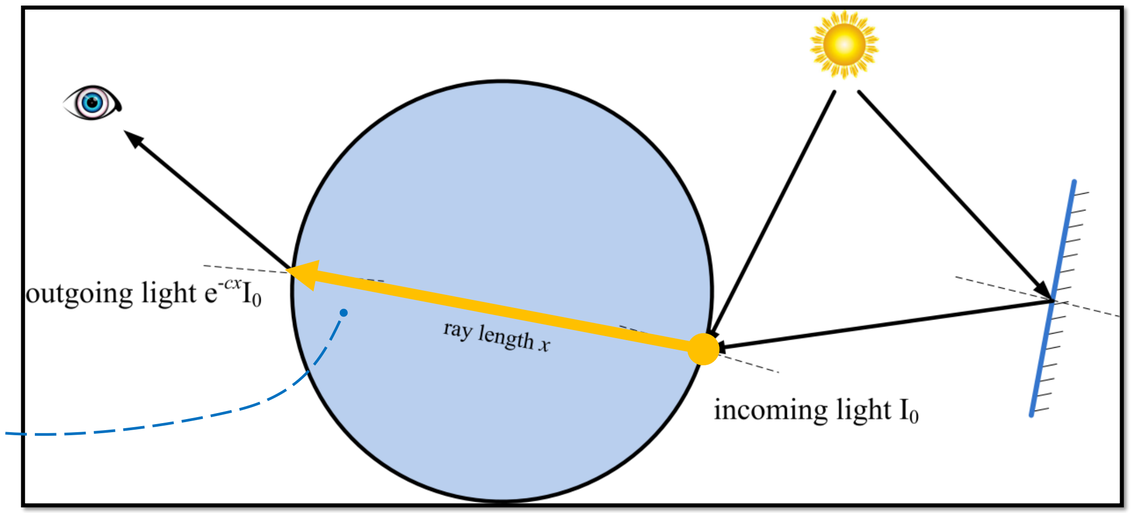
$$\frac{dI}{dx} = -cI$$

in which *I* is the light intensity, *x* is the distance along the ray, and *c* is the attenuation constant.

- Solve this ODE with the initial value $I(0) = I_0$, we have:

$$I(x) = I_0 e^{-cx}$$

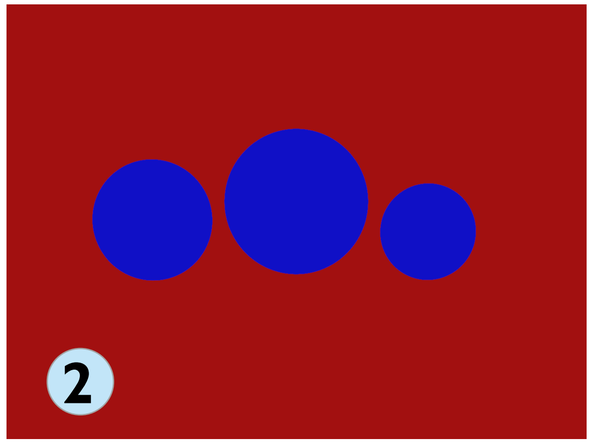


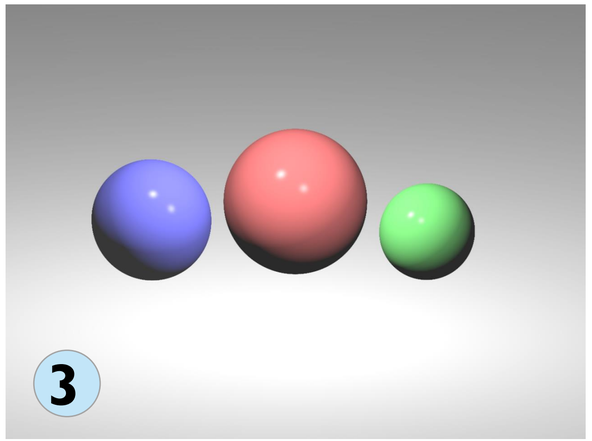


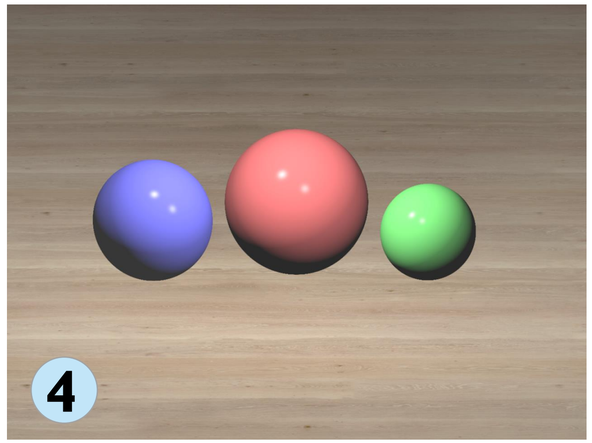
Key Takeaway for Recursive Ray Tracing

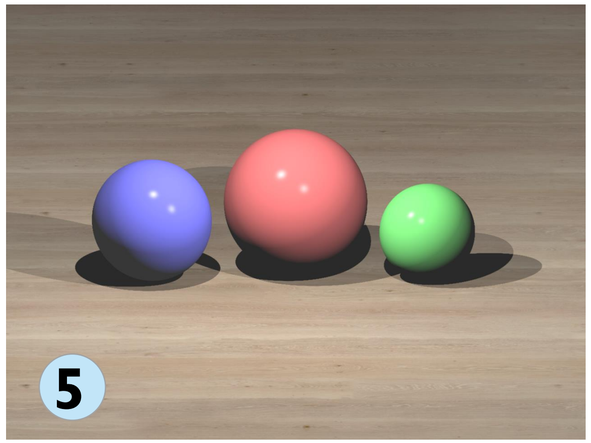
- **Define a recursive depth** to control the maximum number allowed for recursive tracing rays shot for each pixel
- For each pixel, **shoot rays recursively** by calculating a new ray on each hit (reflective, refractive, or both)
- The recursion **stops** at the maximum depth allowed for each pixel
- **Accumulate** the contribution from each recursive ray onto the pixel color

Live Code Demo: A7 Starter Code







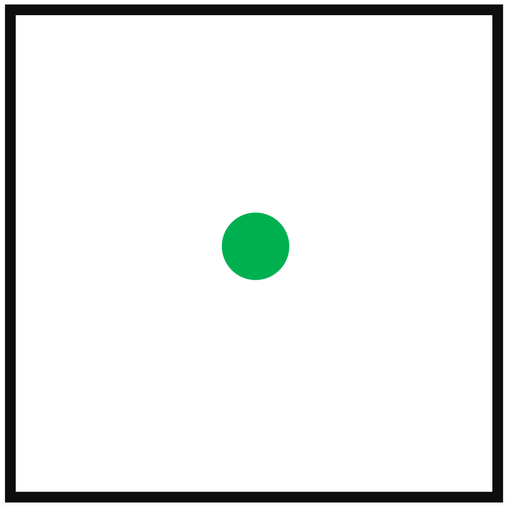


Advanced Ray Tracing

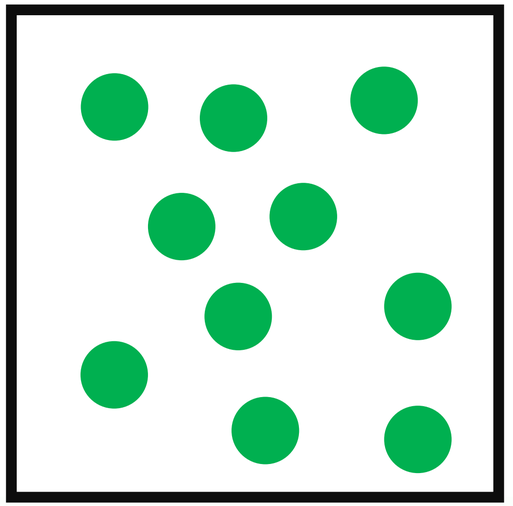
Antialiasing

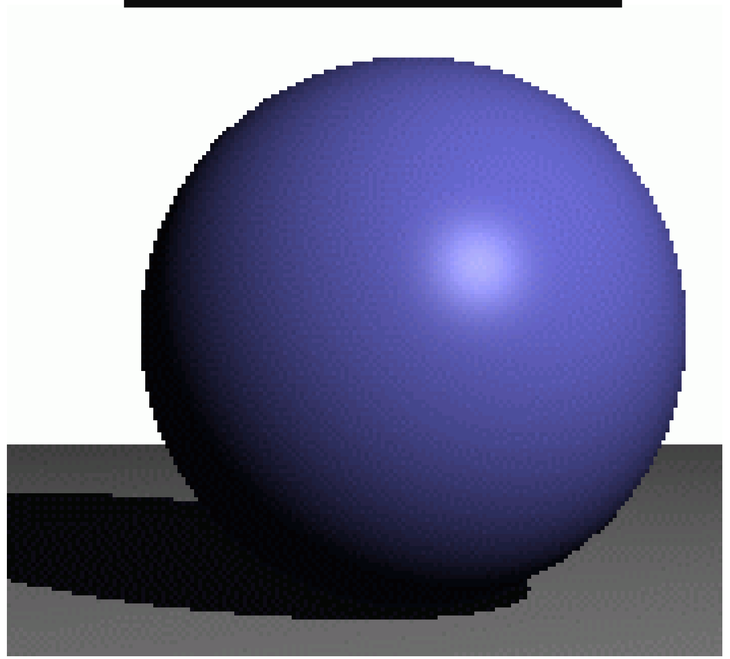
Key Idea: trace multiple rays per pixel and average their colors

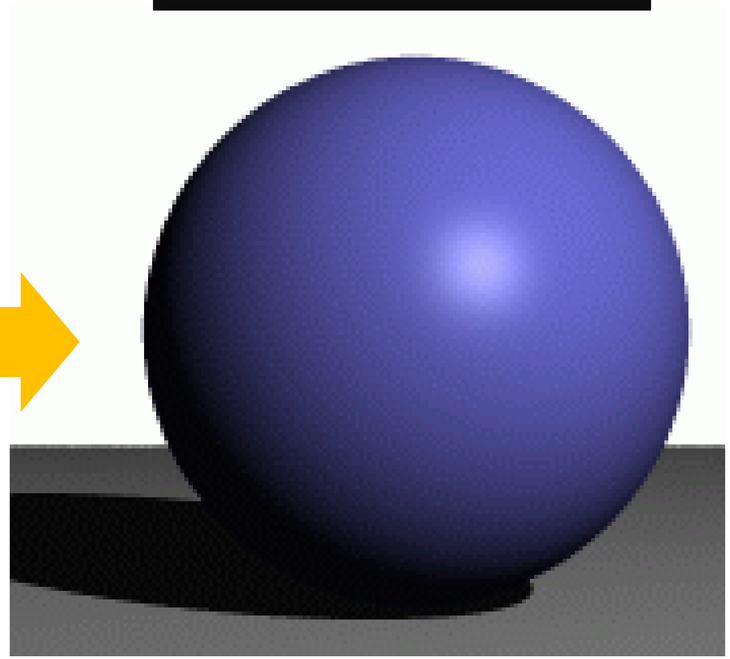
Sample one ray per pixel

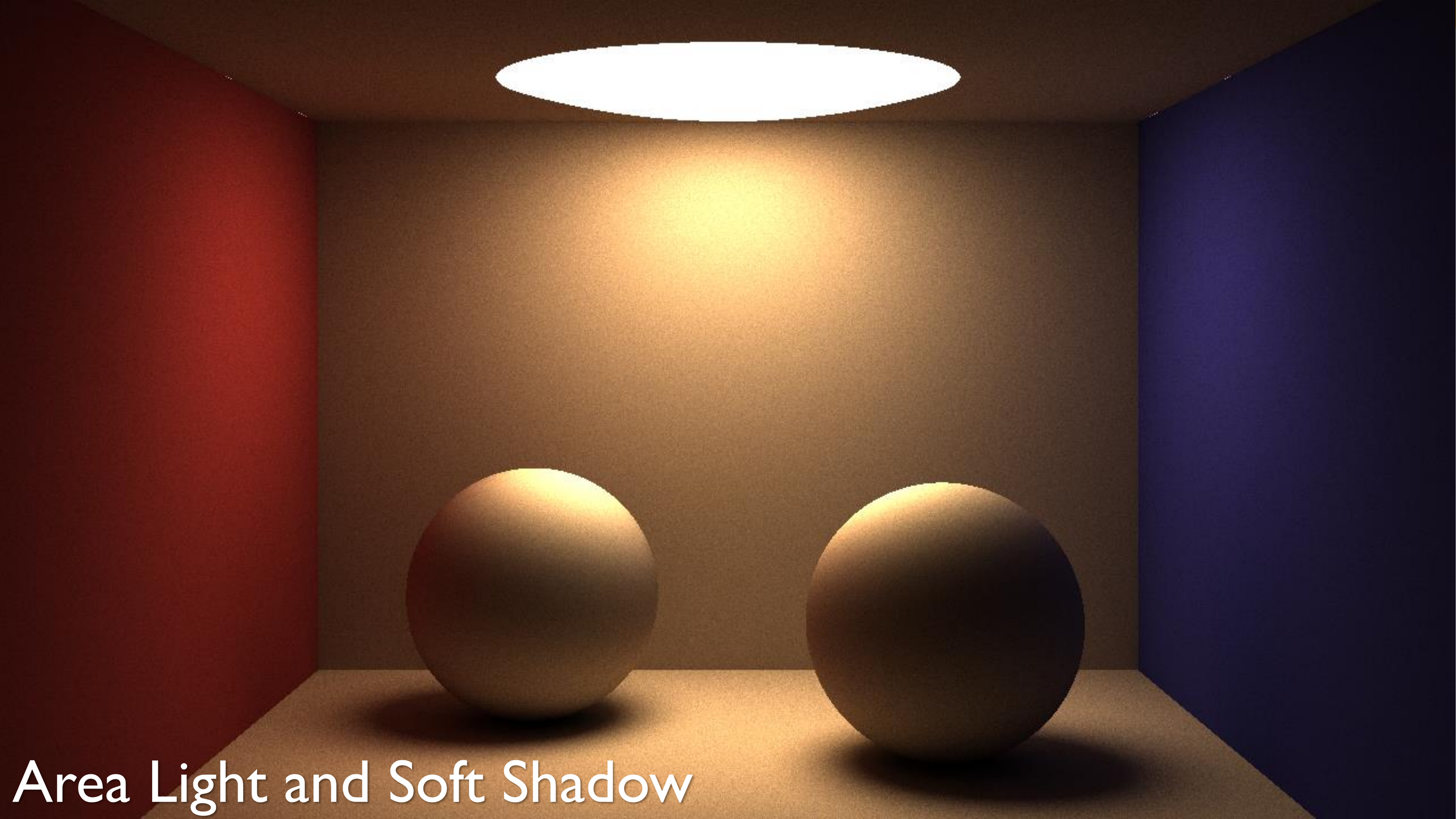


Sample many rays per pixel

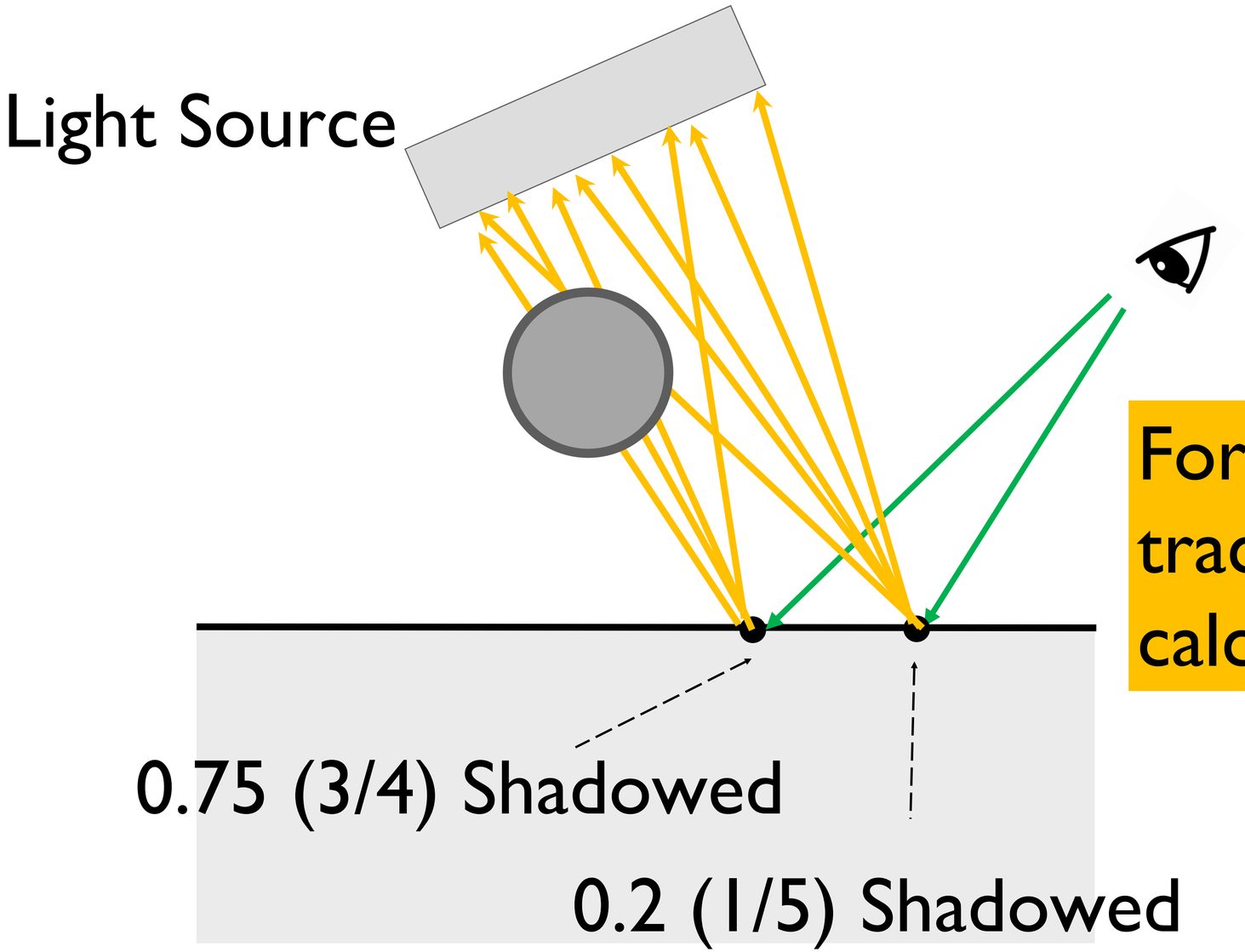







Area Light and Soft Shadow

Key Idea: Trace Multiple Shadow Rays



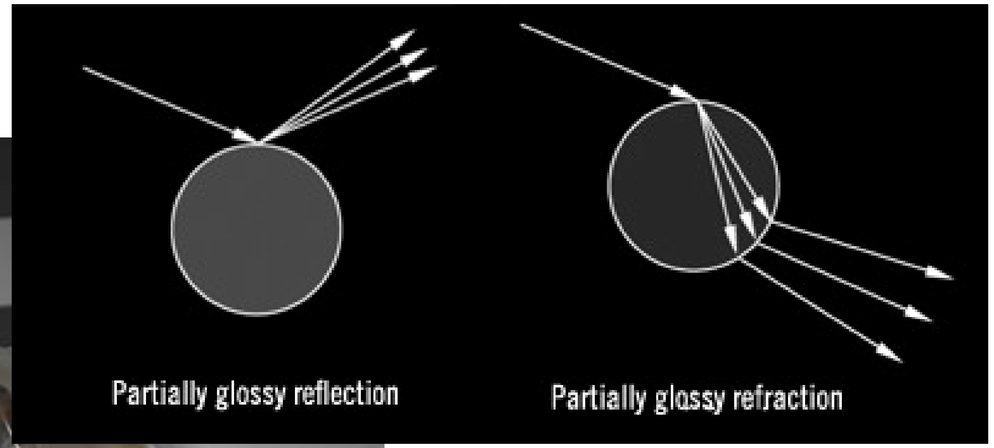
For each intersection, we trace multiple shadow rays to calculate its soft shadow value

Soft Shadow Pseoducode

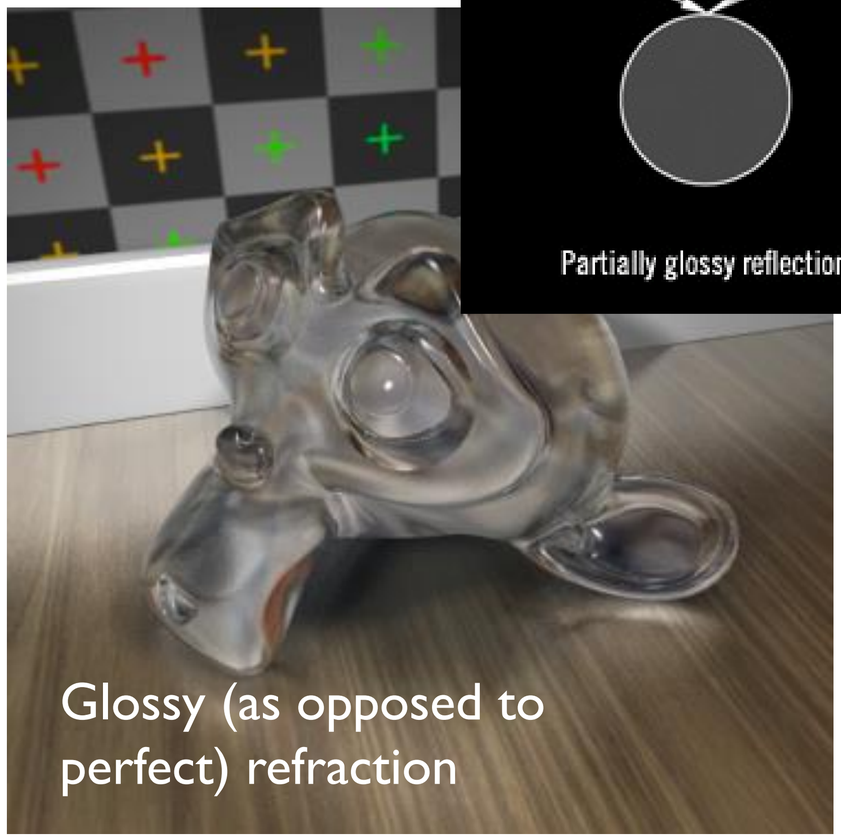
```
trace(Ray ray):
        hit = surfaces.intersect(ray)
        if hit
                emit = hit->mat->emit(ray)
                sRay = hit->mat->scatter(ray)
                return emit + trace(sRay)
        else
                return backgroundColor
```

Non-Ideal Reflection/Refraction

- **Key Idea:** Trace multiple rays for each reflection/refraction to consider the non-ideal surface



Glossy (as opposed to mirror) reflection

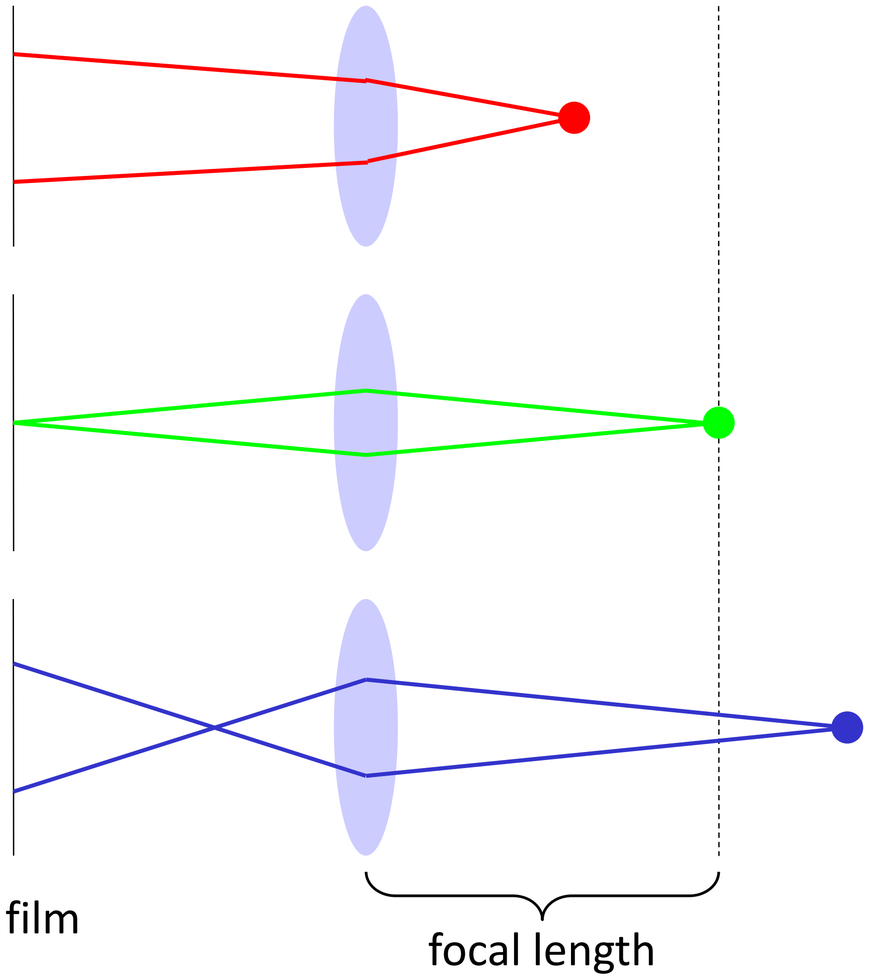


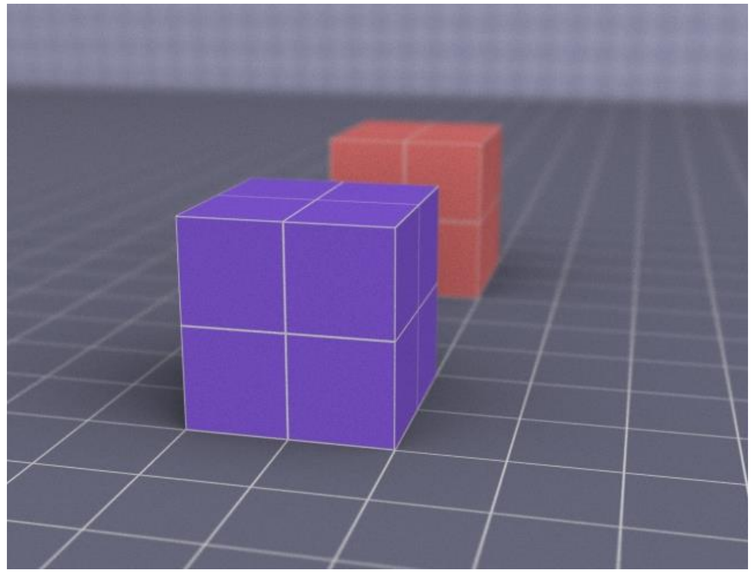
Glossy (as opposed to perfect) refraction

Depth of Field

Depth of Field

- **Key Idea:** trace multiple rays per pixel to sample the lens aperture





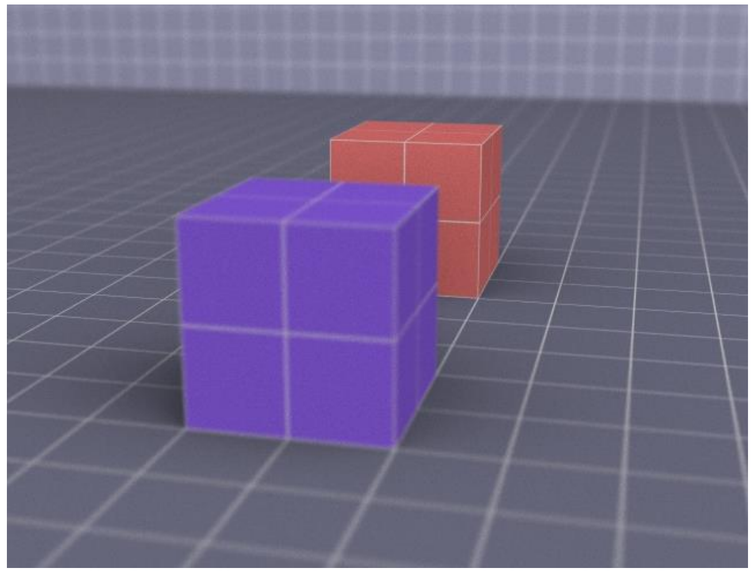
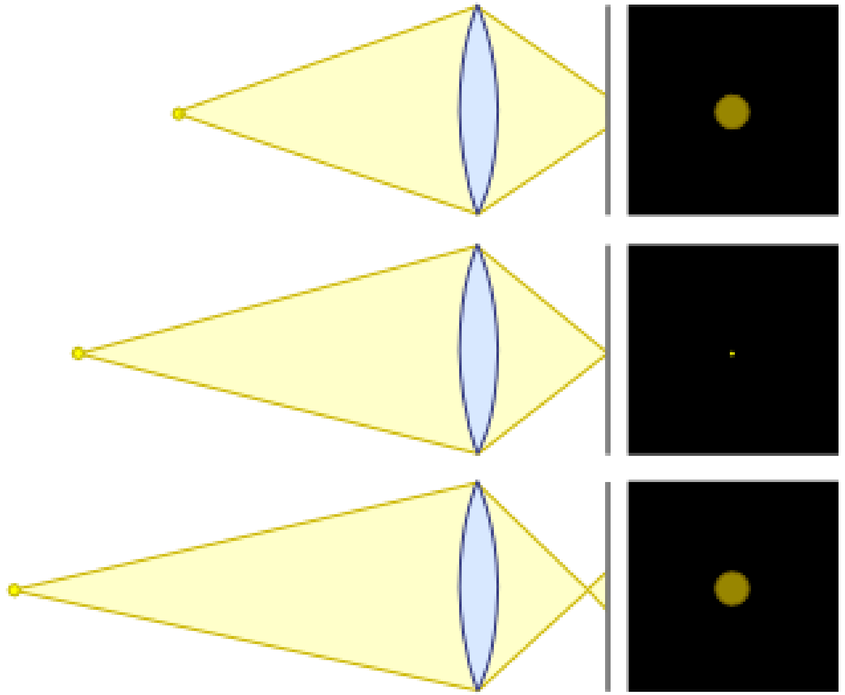


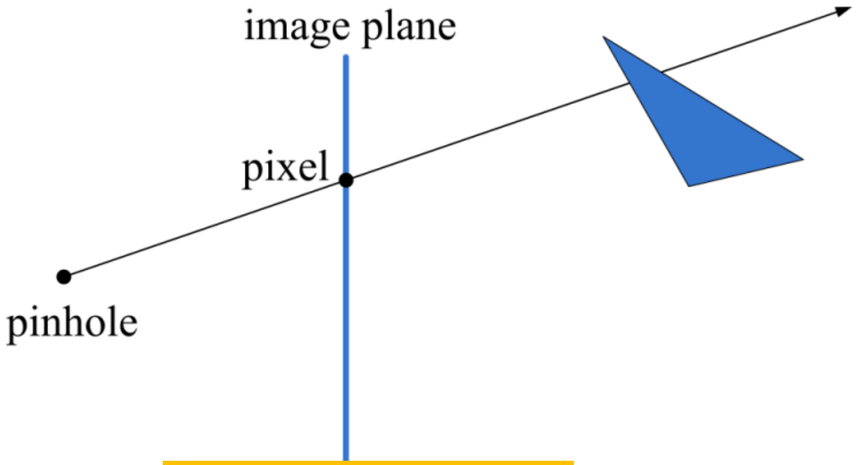
Image by Justin Legakis

Circle of confusion

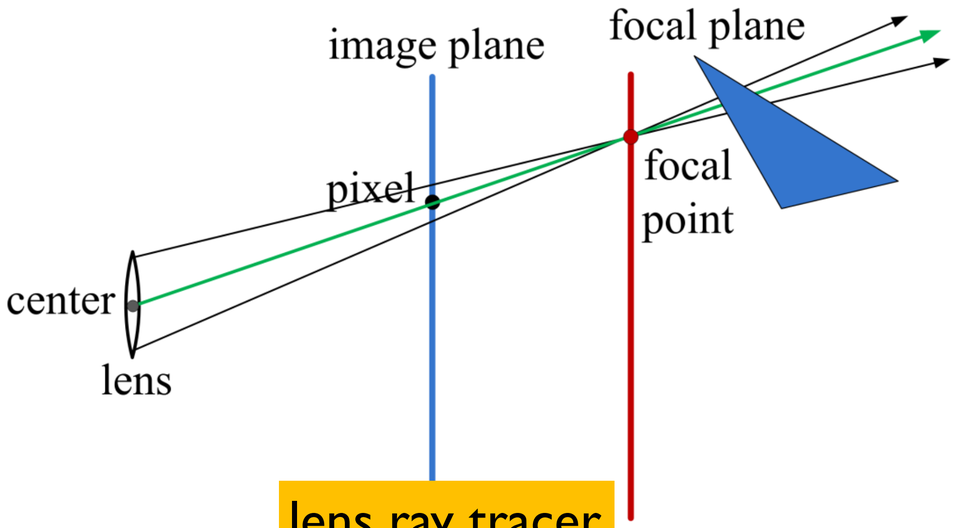


- An optical spot caused by a cone of light rays from a lens not coming into perfect focus when imaging a point source
- When the spot is approximately equal to the size of a pixel on the sensor, the object seems to be "in focus"
- Objects at varying distances from the camera require the sensor to be placed at different distances from the lens in order for the object to be "in focus"
- **Depth of Field** - the distance between the nearest and farthest objects in a scene that appear roughly "in focus" (the circle of confusion is not too big)

Depth of Field Implementation



pinhole ray tracer



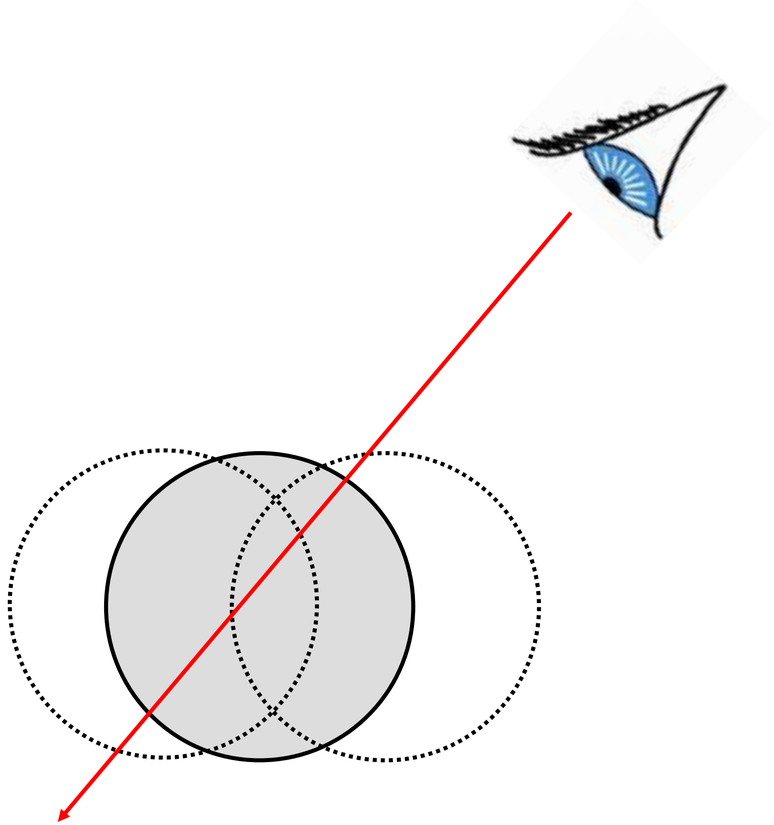
lens ray tracer

Algorithm

- Specify a focal plane (red) where objects will be in focus
- The "focal point" is calculated as the intersection of the ray (green) and the focal plane
- For each pixel, replace the pinhole "eye" with a circular region
- Shoot multiple rays from sampled points in the circular region through the focal point
- Average the color: objects further away from the focal plane will have more blurring

Motion Blur

- **Key Idea:** Sample objects temporally over time interval

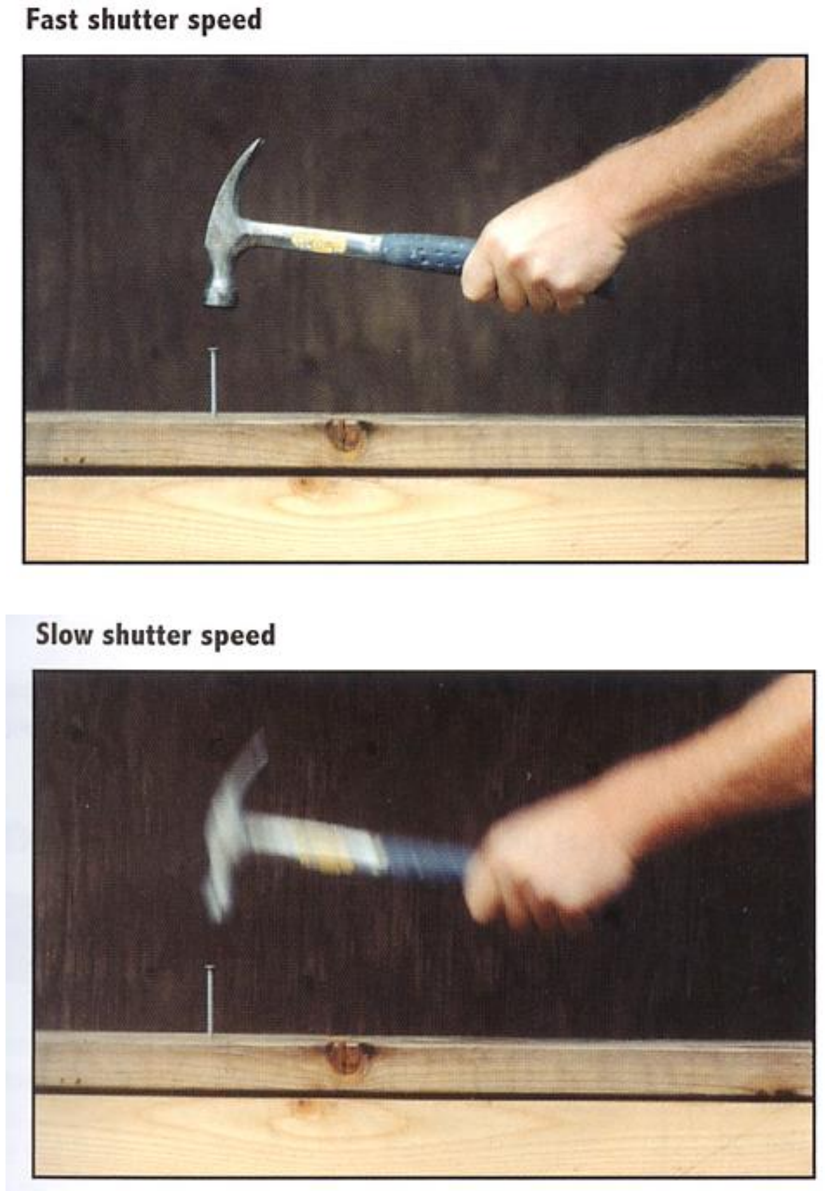


Shutter Speed

- The shutter allows light to hit the sensor for a finite duration of time
- Objects which move while the shutter is open create multiple images on the sensor, resulting in motion blur
- A faster shutter speed prevents motion blur, but can severely limit the amount of light available to the sensor making the resulting image too dark (especially when the aperture size is small)

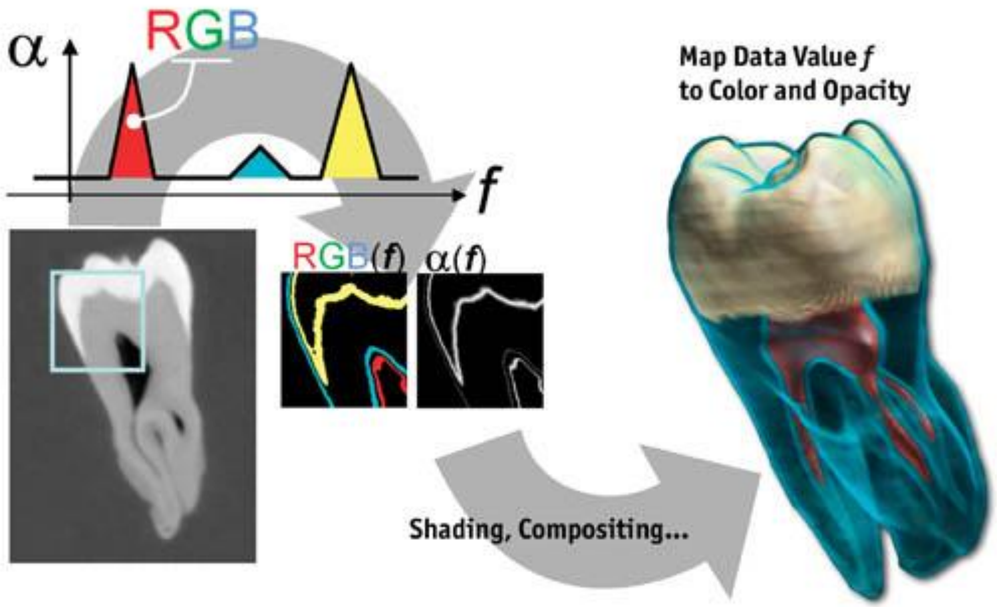
Motion Blur Ray Tracing

- Set up animations for moving objects during the time interval in which the shutter is open $[T_0, T_1]$
  - E.g. describe the transform of the object by a function F(t) for $t \in [T_0, T_1]$
- For each ray:
  - Assign a random time $t_{ray} = (1-\alpha)T_0 + \alpha T_1$
  - All objects in the scene are placed in their time t_{ray} locations
  - Trace the ray with the time t_{ray} scene and get a color for that ray



Volumetric Ray Tracing

- Generate images of a 3D volumetric data set without explicitly extracting geometric surfaces

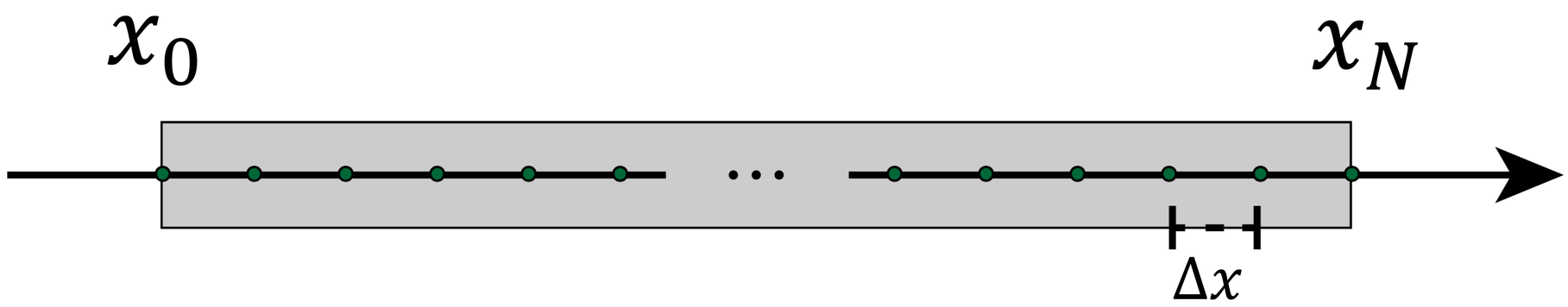


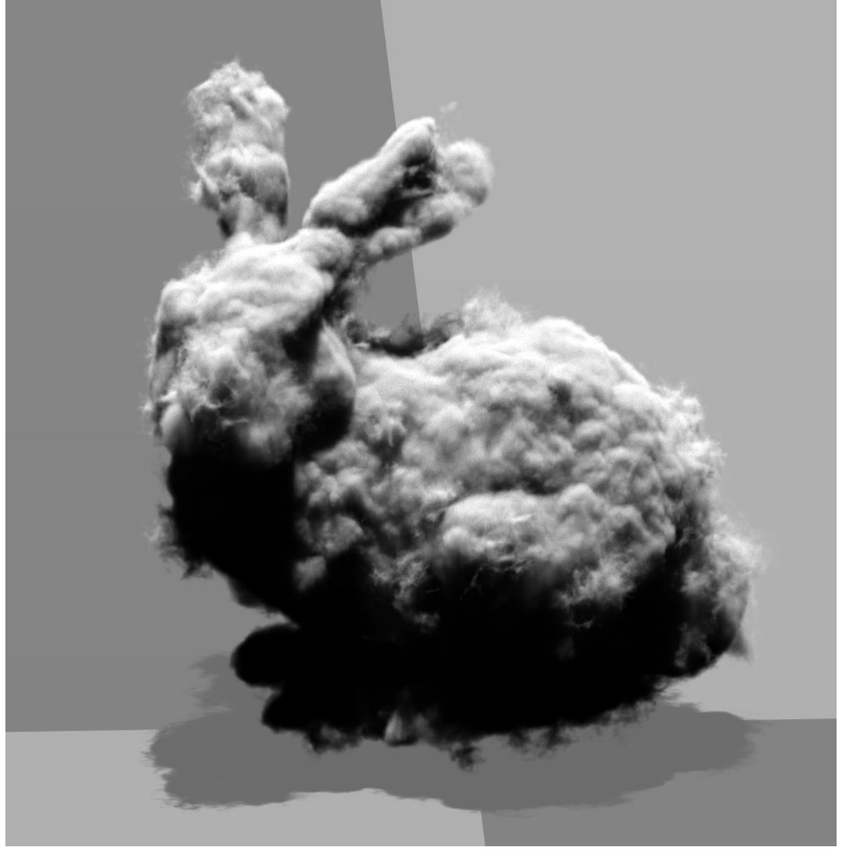
https://developer.download.nvidia.com/books/HTML/gpugems/gpugems_ch39.html

Key Idea: Discretize a ray into many small segments

- **Key Idea: Discretize the ray into N segments**, and accumulate the attenuated color from each segment along the ray
- The attenuation along the i-th segment is set to be $e^{-c(.5(x_{i-1}+x_i))\Delta x}$
  - $\Delta x = (x_N - x_0)/N$ is the segment length, $c(.5(x_{i-1}+x_i))$ is the attenuation constant of each segment, and $x_i = x_0 + \mathrm{i}\Delta x$
- The total attenuation along the ray is computed via multiplication:

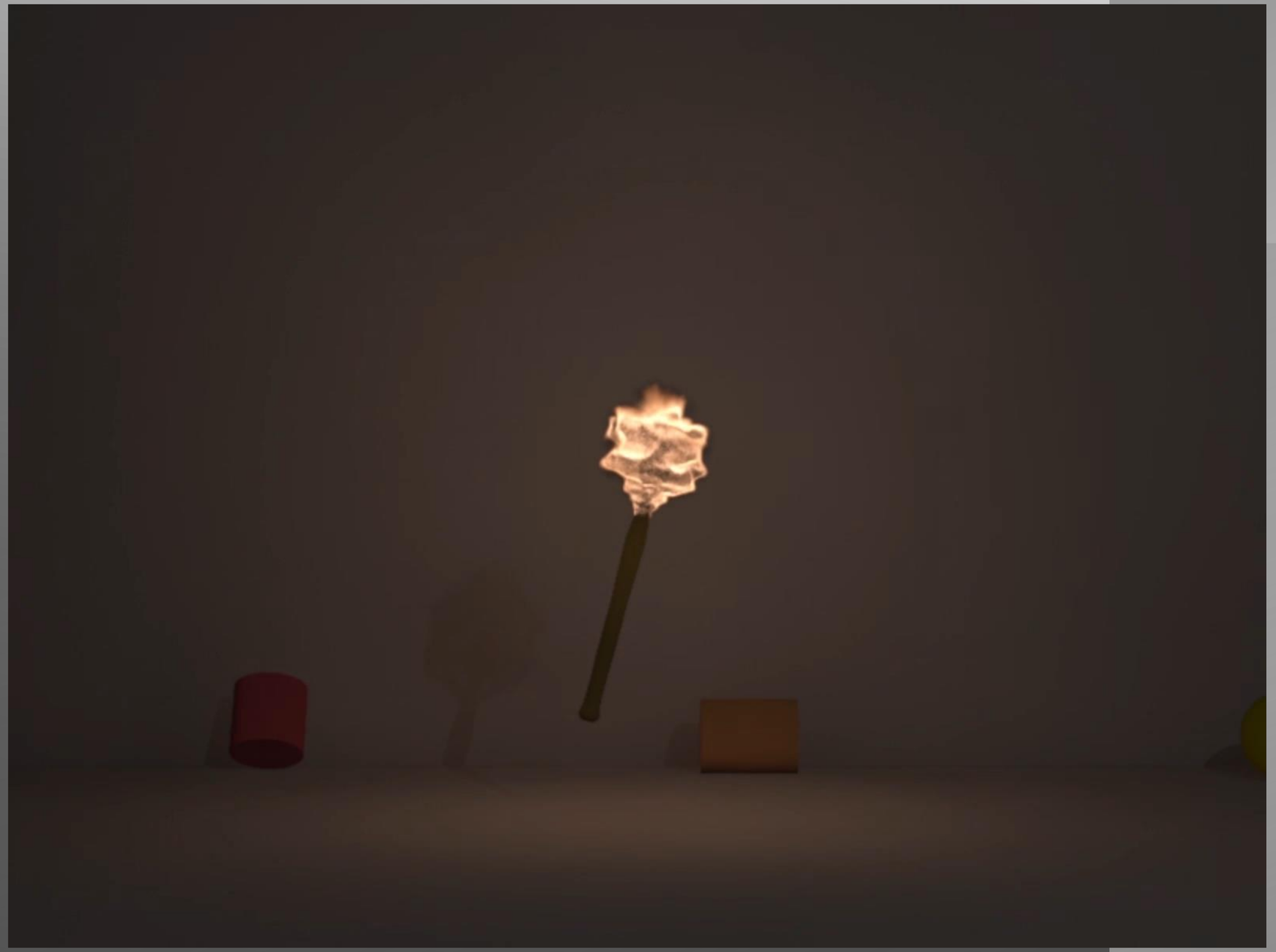
$$e^{-c(.5(x_0+x_1))\Delta x} e^{-c(.5(x_1+x_2))\Delta x} \dots e^{-c\left(.5(x_{N-1}+x_N)\right)\Delta x}$$





Ray Tracing of Participating Media

- Trace rays into a volume of cloud/smoke/fire
- Sample each ray with many small segments
- Calculate the attenuation and emission of each segment
- Calculate the final color by accumulating colors along each ray



Reading Materials

- Ray tracing in one weekend: https://raytracing.github.io/books/RayTracingInOneWeekend.html
- Ray Tracing: The Next Week: https://raytracing.github.io/books/RayTracingTheNextWeek.html
- https://raytracing.github.io/