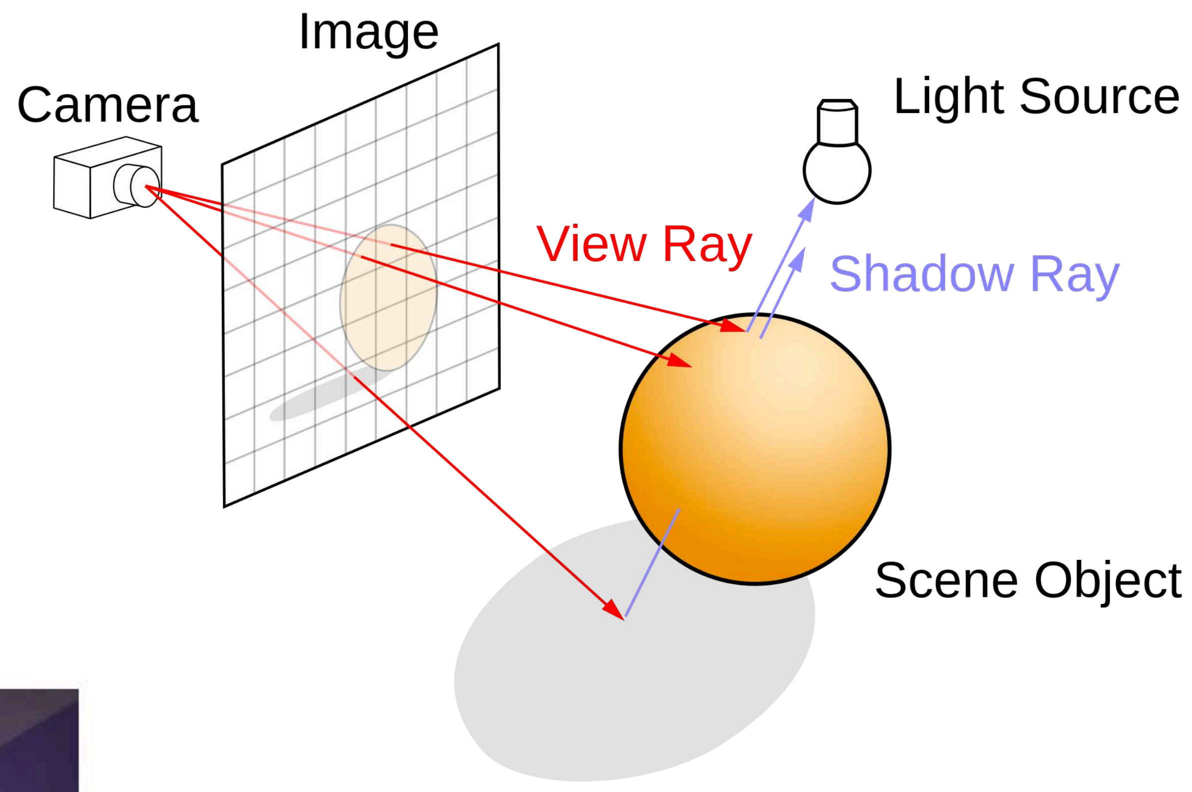
CSE 167 (FA22)
Computer Graphics:
Ray Tracing

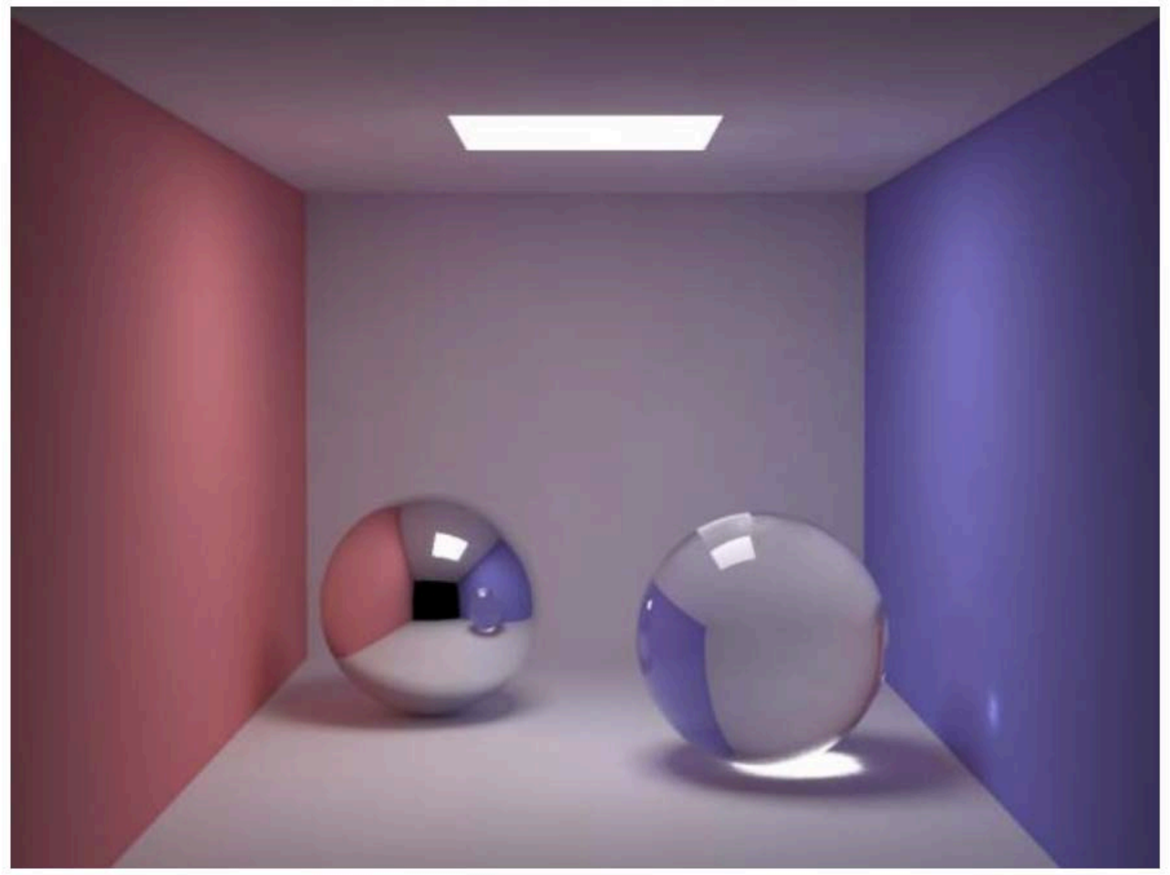
Albert Chern

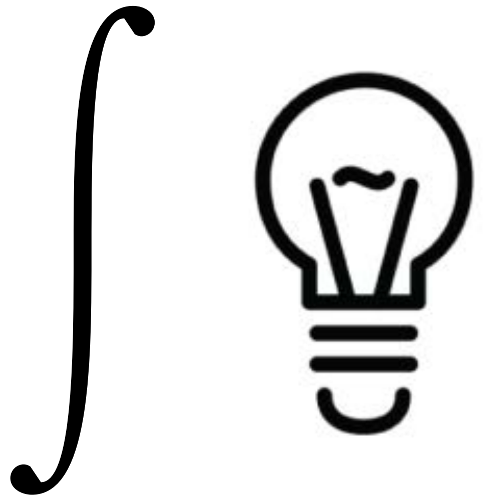
UC San Diego
JACOBS SCHOOL OF ENGINEERING

Overview

- Goal: photorealism
- Ray tracing framework
- Global illumination
- Rendering equation (next Monday 11/14 pre-recorded lecture)



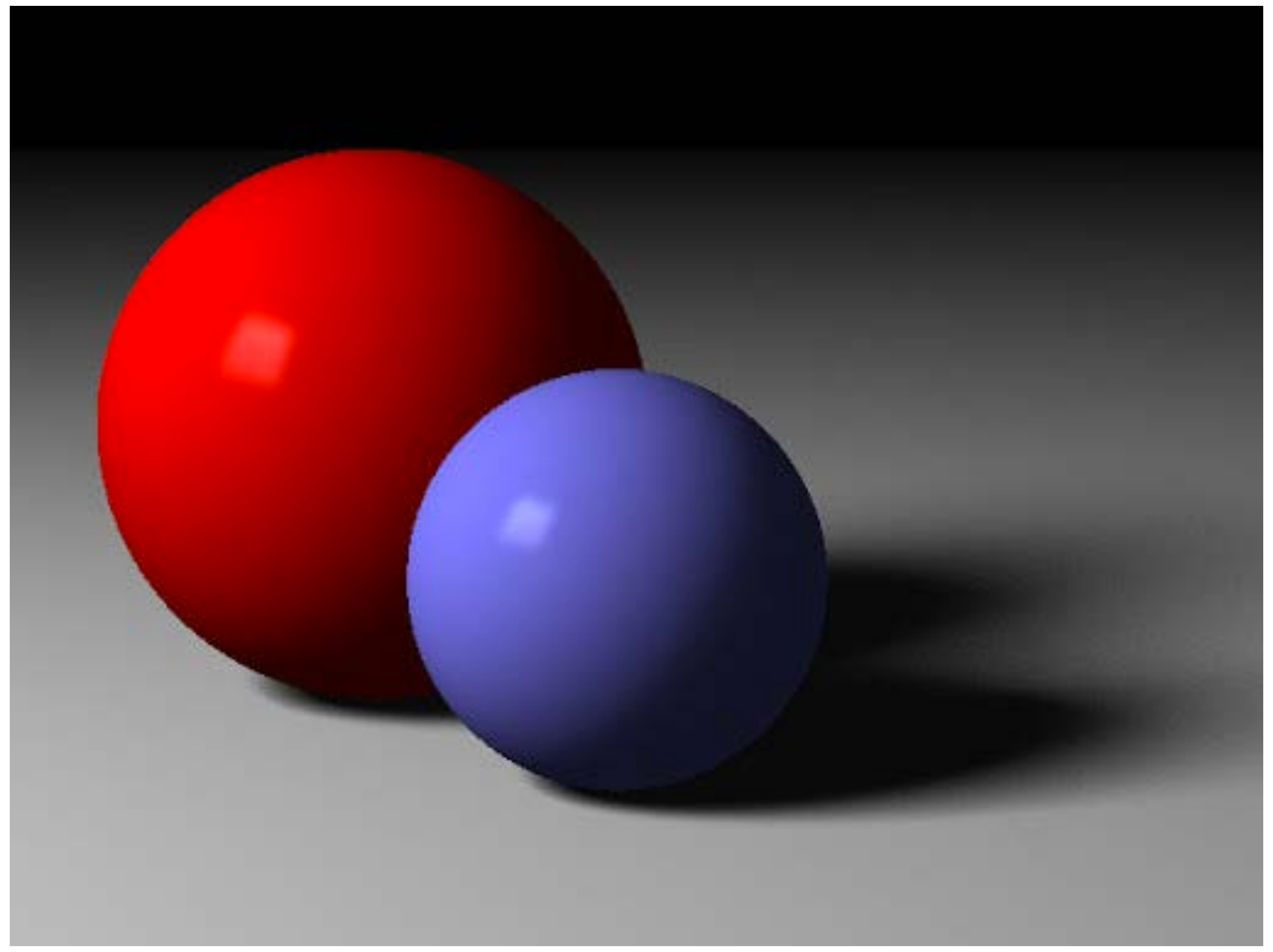
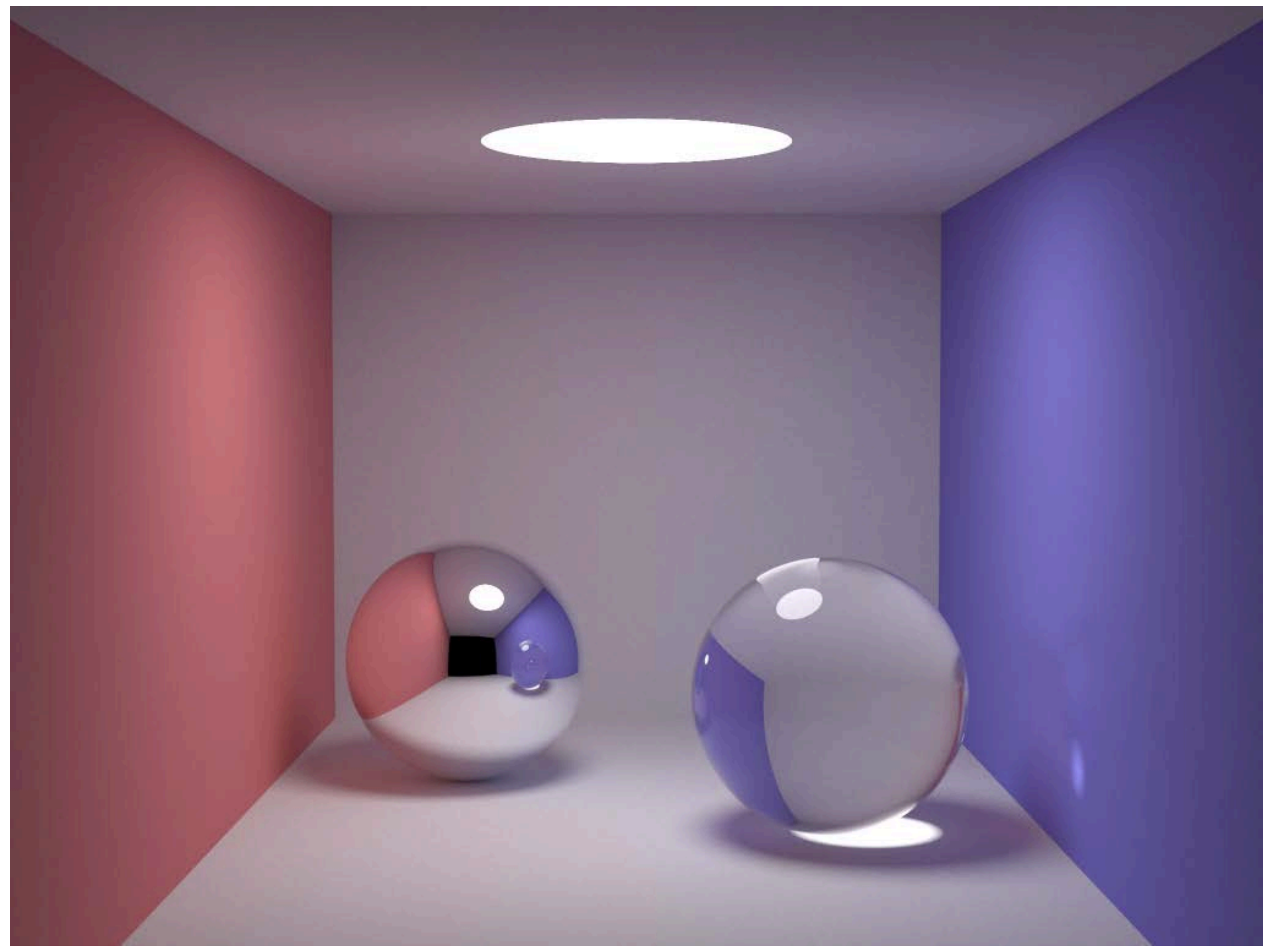
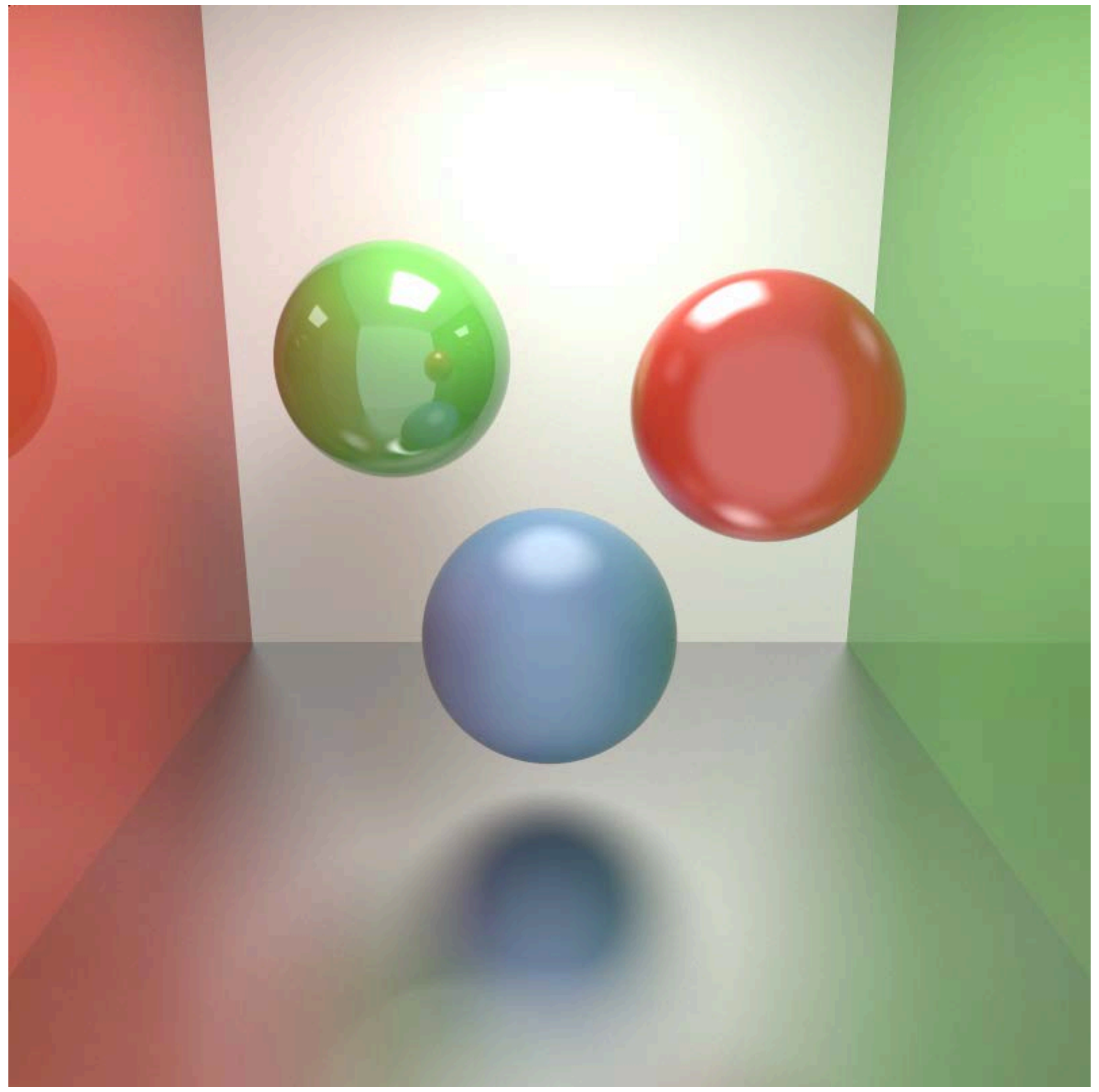
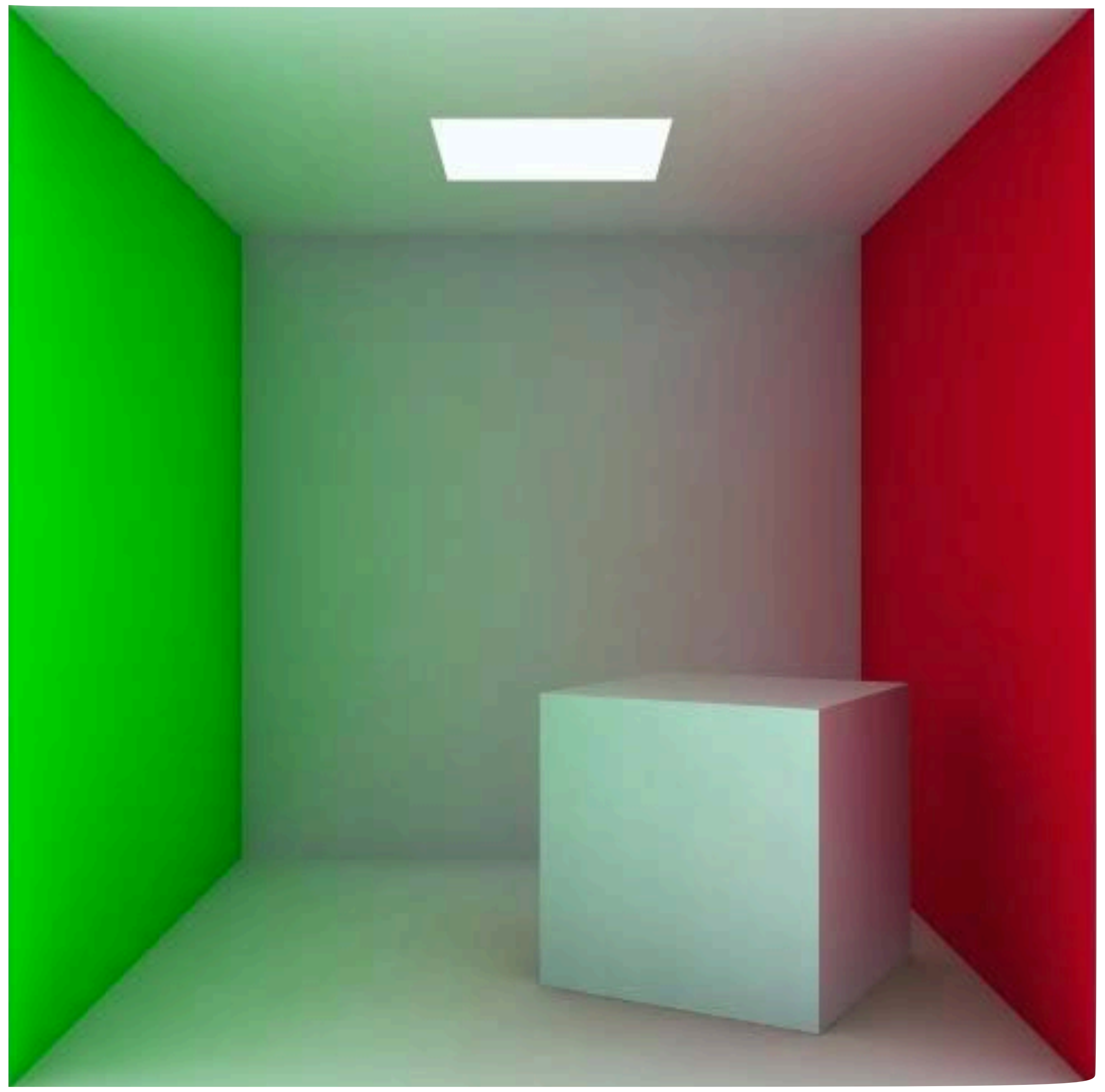
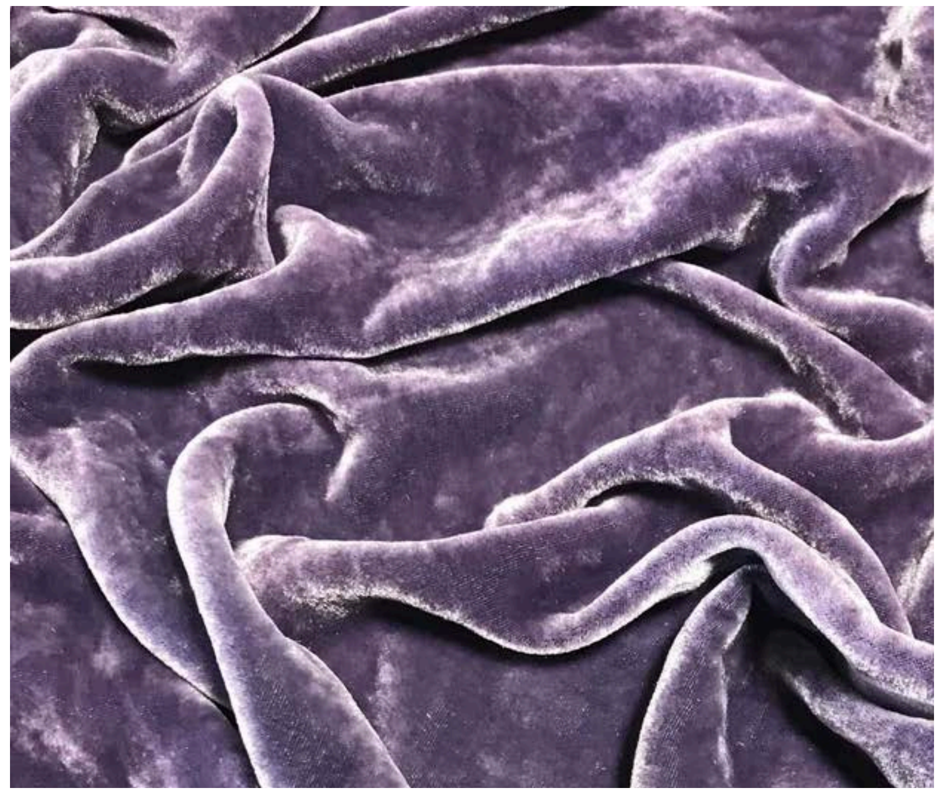




Rendering photorealistic images

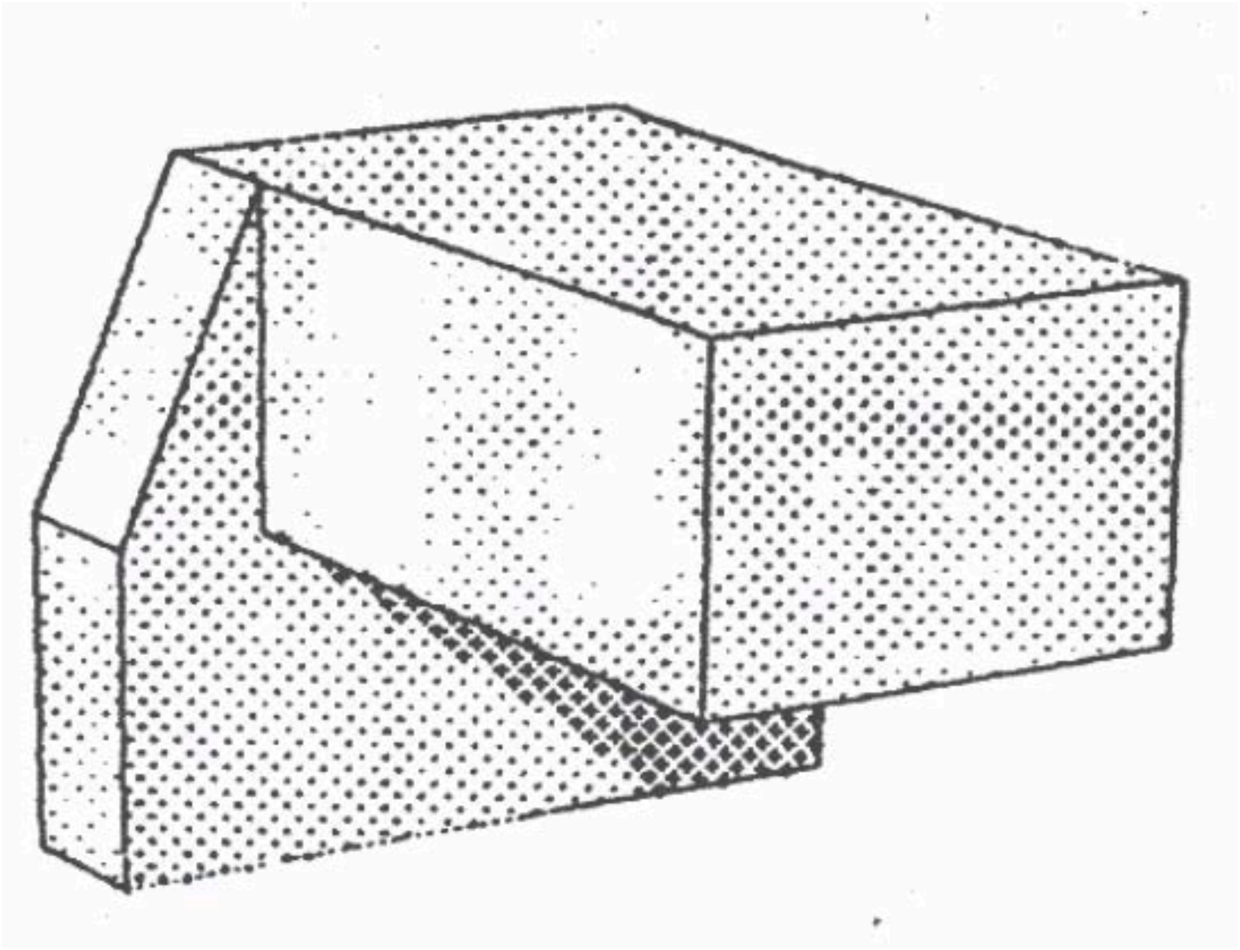
- Effects for realistic images
  - (Soft) shadows
  - Reflections (mirror and glossy)
  - Transparent (water, glass)
  - Inter-reflections (color bleeding)
  - Realistic materials

- Difficult in OpenGL pipeline
- Easy in raytracing framework

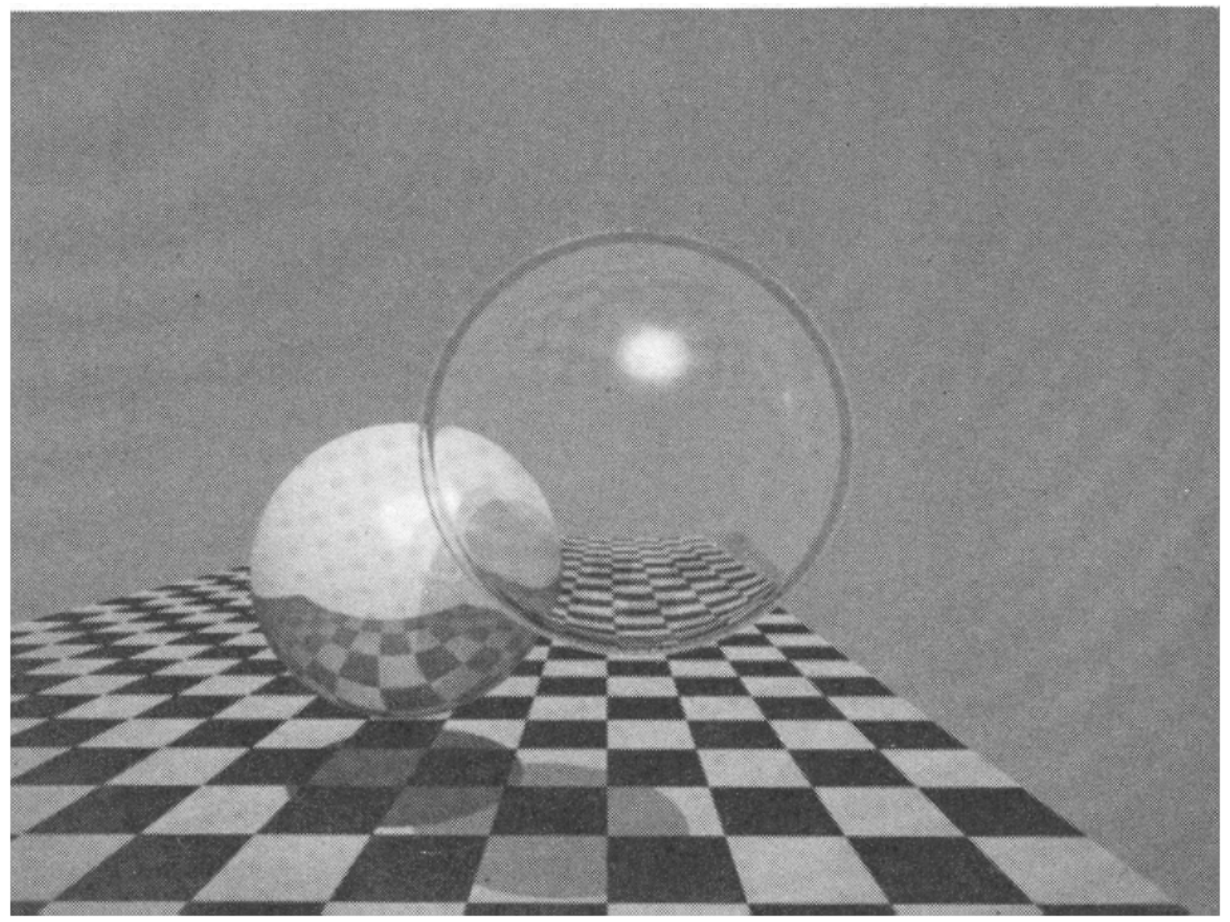







History of ray casting/tracing

- Appel 1968



- Whitted 1980 recursive ray tracing



- Lots of work on photorealism, accelerations.

- Real time ray tracing
  - 2009 Nvidia OptiX API
  - 2020 PlayStation5, Xbox series X and S

Ray Tracing Framework

- Ray tracing framework
- Ray through pixel
- Ray-geometry intersection
- Organizing image and scene
- Global illumination

Rasterization v.s. Ray tracing

Rasterization

for each geometry **in** scene
  for each pixel **in** screen
    output the fragment **if** the triangle occupies that pixel.
  end for
end for

Ray tracing (a.k.a. ray casting)

for each pixel **in** screen
  for each geometry **in** scene
    output the intersection **if** the triangle occupies that pixel.
  end for
end for

has the information of
the geometry (position, normal)
and the in-coming ray (ray direction or pixel)

Rasterization v.s. Ray tracing

Rasterization

for each geometry **in** scene
  for each pixel **in** screen
    output the fragment **if** the triangle occupies that pixel.
  end for
end for

Ray tracing (a.k.a. ray casting)

for each pixel **in** screen
  for each geometry **in** scene
    output the intersection **if** the triangle occupies that pixel.
  end for
end for

- These two approaches give the same images as long as the shading models are the same
- But it is much easier for ray tracer to include realistic shading model

Ray tracing framework

- Does not rely on OpenGL (OpenGL is a rasterizer).
- We will prepare our own "buffers" as C++ arrays/containers.
- We will run our own loop in C++ to search for ray-geo intersection.
- Most of the HW3 framework (scene building, camera control) is re-usable.
  (replace setting OpenGL buffers, skip shaders,…)
- OpenGL could be used to visualize final result by setting our computed pixel color in a texture and show it on a square.
  GLUT is still useful for keyboard controls.

Ray tracing framework

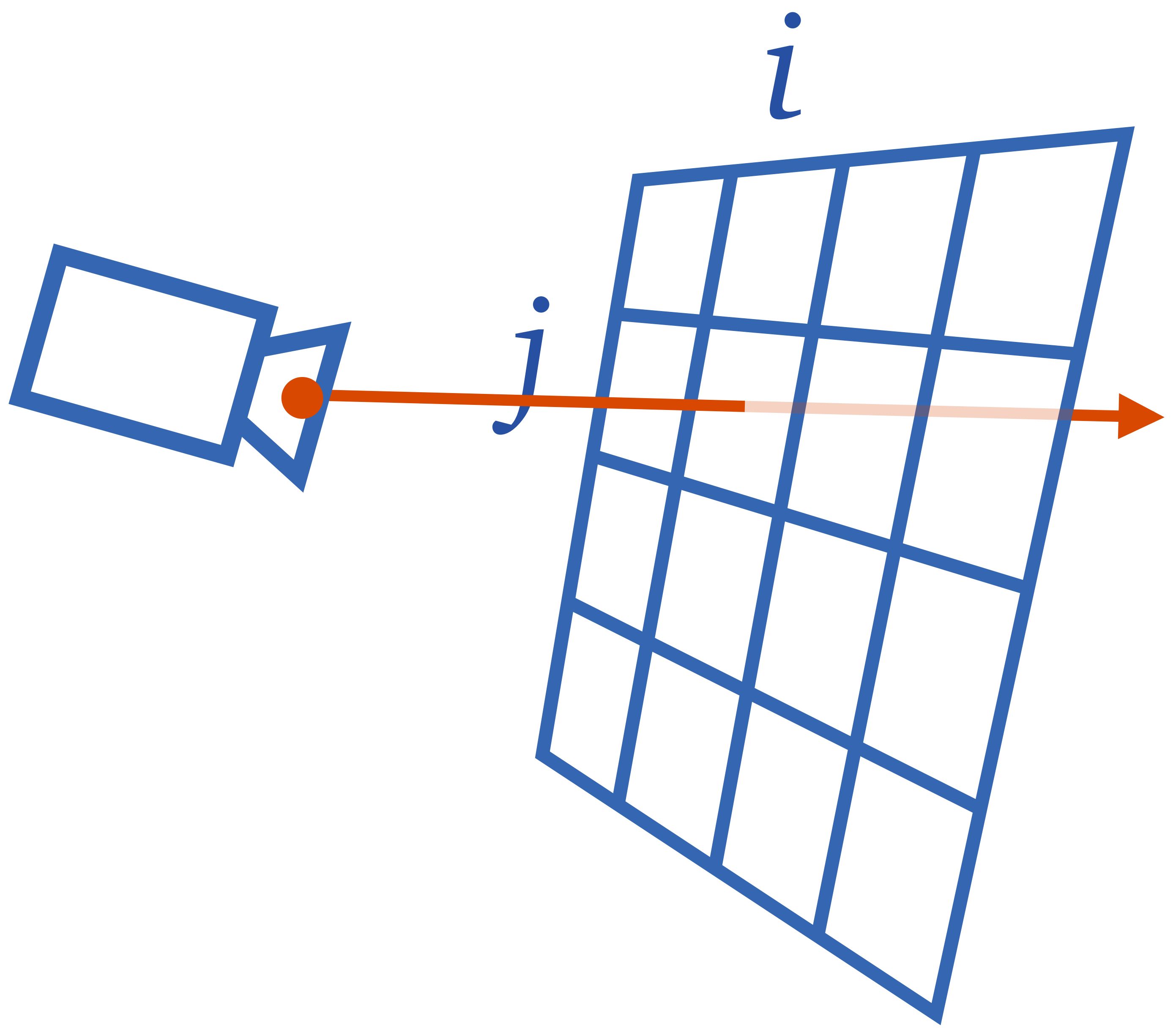
- Essential objects
  - Scene (container for geometries, lights, etc)
  - Image (container for pixel colors, info of width and height)
  - Camera (position, orientation, field of view angle, etc)
  - Ray (position and direction)
  - Intersection (geometry info and ray info)

```
void Raytrace(Camera cam, Scene scene, Image &image){
  int w = image.width; int h = image.height;
  for (int j=0; j<h; j++){
    for (int i=0; i<w; i++){
      Ray ray = RayThruPixel( cam, i, j, w, h );
      Intersection hit = Intersect( ray, scene );
      image.pixel[i][j] = FindColor( hit );
    }
  }
}
```

Ray tracing framework

```
void Raytrace(Camera cam, Scene scene, Image &image){
  int w = image.width; int h = image.height;
  for (int j=0; j<h; j++){
    for (int i=0; i<w; i++){
      Ray ray = RayThruPixel( cam, i, j, w, h );
      Intersection hit = Intersect( ray, scene );
      image.pixel[i][j] = FindColor( hit );
    }
  }
}
```

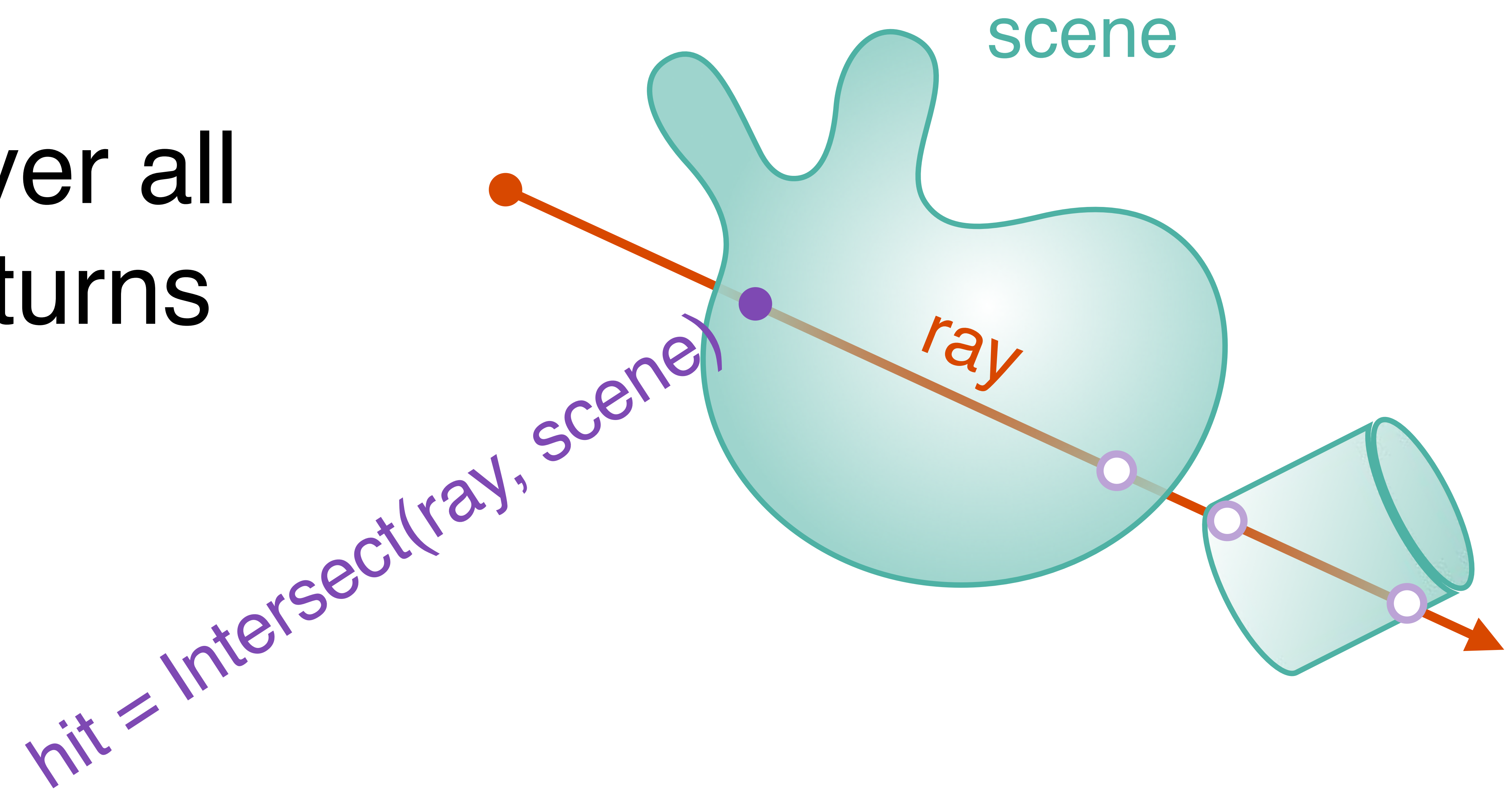
- **RayThruPixel(cam, i, j, w, h)** generates a ray originated from the camera position, through the center of the (i,j) pixel into the world



Ray tracing framework

```
void Raytrace(Camera cam, Scene scene, Image &image){
  int w = image.width; int h = image.height;
  for (int j=0; j<h; j++){
    for (int i=0; i<w; i++){
      Ray ray = RayThruPixel( cam, i, j, w, h );
      Intersection hit = Intersect( ray, scene );
      image.pixel[i][j] = FindColor( hit );
    }
  }
}
```

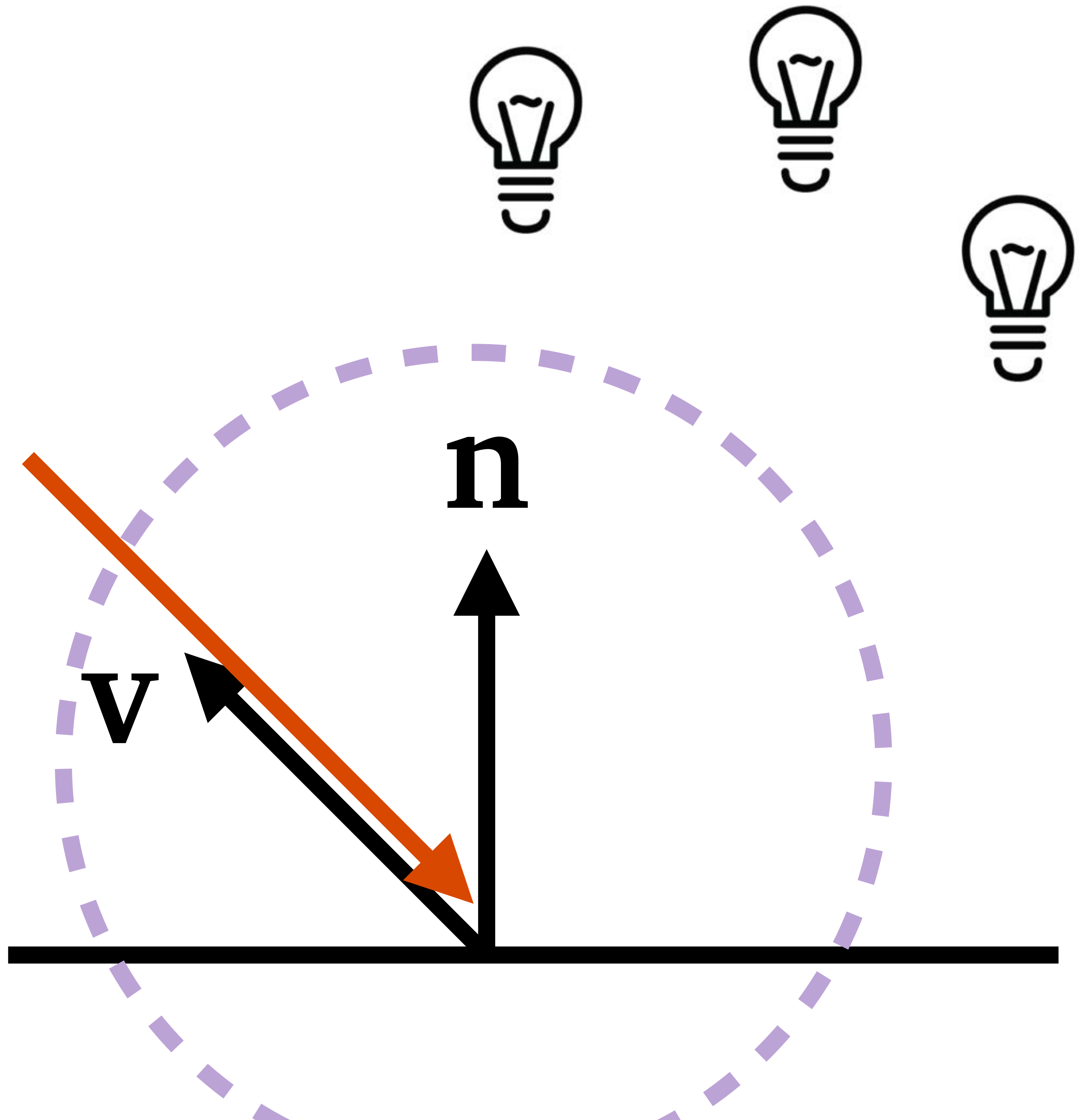
- **Intersect(ray, scene)** searches over all all geometries in the scene and returns the closest hit



Ray tracing framework

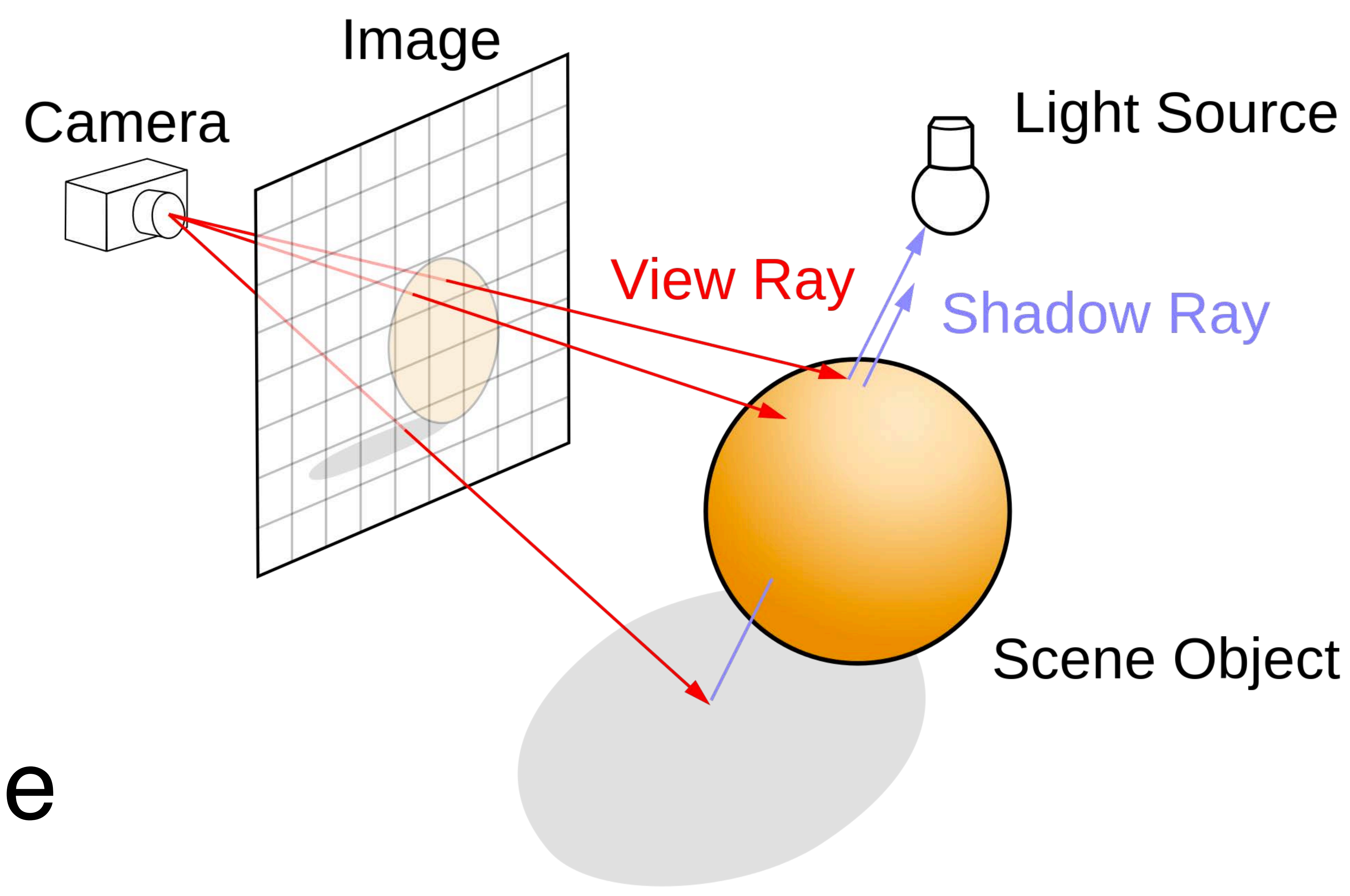
```
void Raytrace(Camera cam, Scene scene, Image &image){
  int w = image.width; int h = image.height;
  for (int j=0; j<h; j++){
    for (int i=0; i<w; i++){
      Ray ray = RayThruPixel( cam, i, j, w, h );
      Intersection hit = Intersect( ray, scene );
      image.pixel[i][j] = FindColor( hit );
    }
  }
}
```

- **FindColor(hit)** shade the light color seen by the in-coming ray
  - For example,
    Ambient + Lambertian-diffuse
    + Blinn–Phong formula



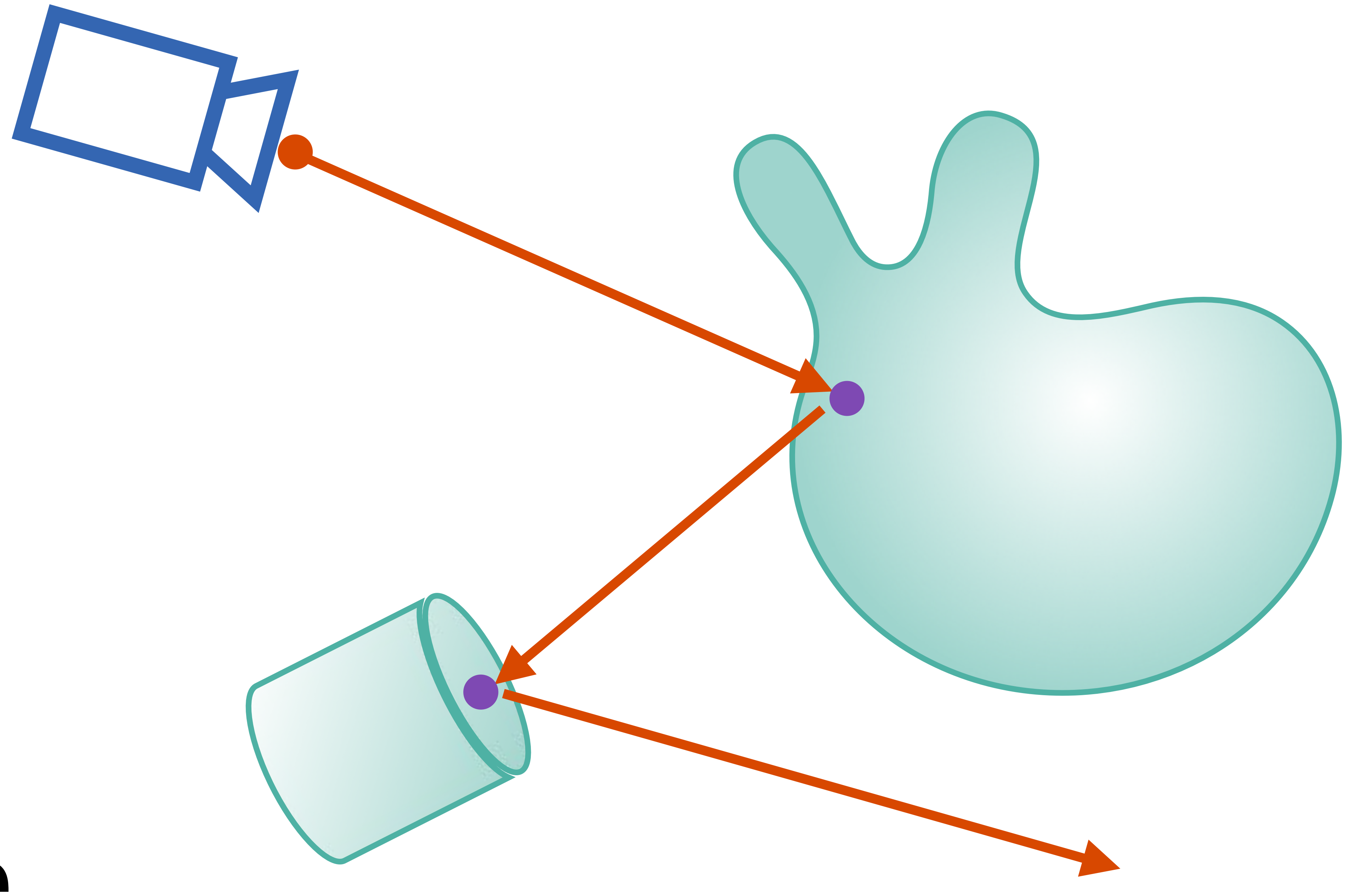
Ray tracing framework

- **FindColor(hit)** shade the light color seen by the in-coming ray
  - For example,
    Ambient + Lambertian-diffuse
    + Blinn–Phong formula
  - Add the contribution of light only when the ray connecting the hit and the light source does not have any intersection with the scene. (Shadows!)
  - To avoid self-shadowing, the secondary ray is shot off slightly above the hitting point.



Ray tracing framework

- **FindColor(hit)** shade the light color seen by the in-coming ray
  - For example,
    Ambient + Lambertian-diffuse
    + Blinn–Phong formula
  - Add the contribution of light only when the ray connecting the hit and the light source does not have any intersection with the scene. (Shadows!)
  - Instead of ambient+diffuse+specular, do recursive ray tracing.



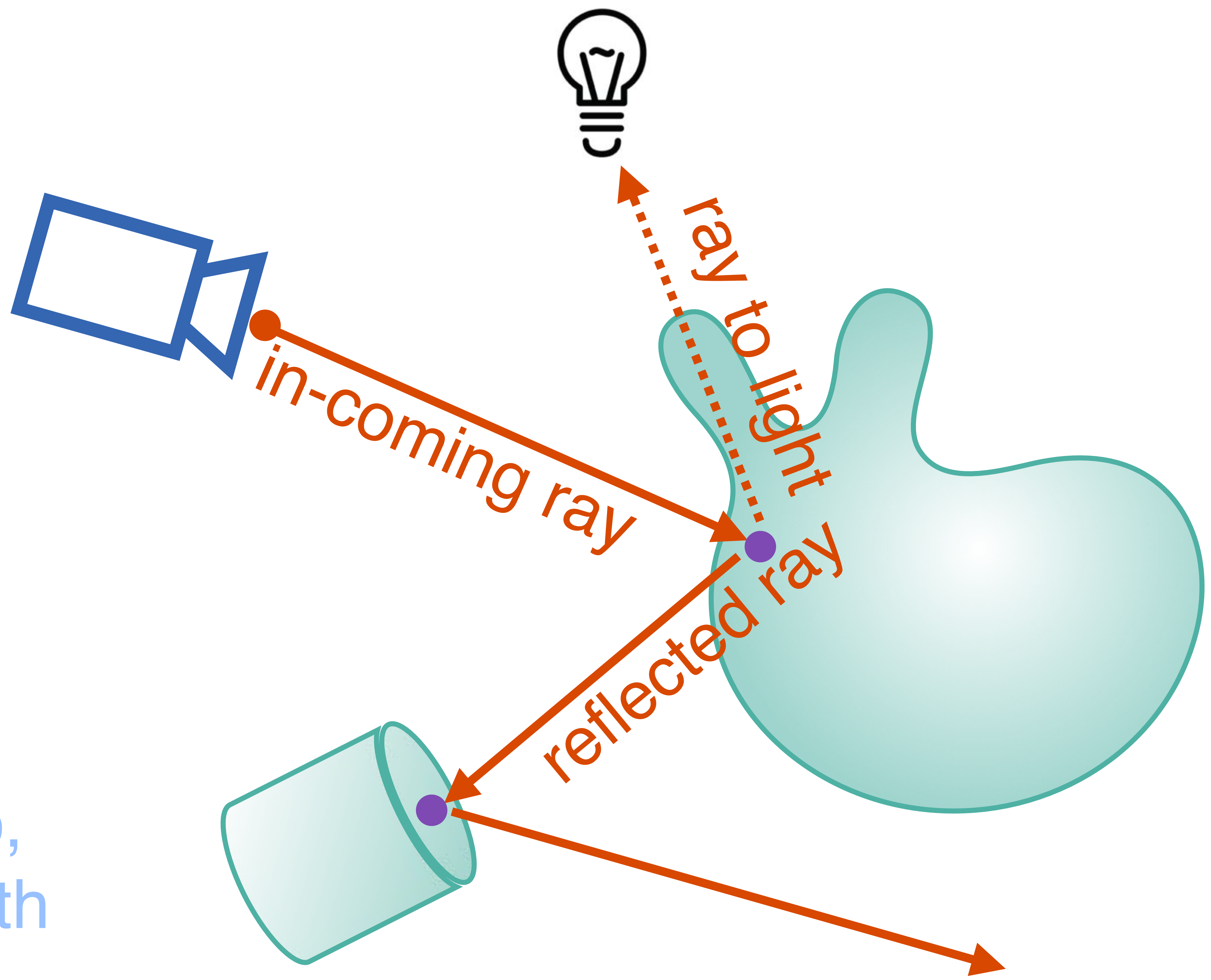
Example: Adding mirror reflection

Color FindColor(hit){

- Generate secondary rays to all lights
  - color = Visible? ShadingModel : 0;
- ray2 = Generate mirror-reflected ray
  - hit2 = Intersect(ray2, scene);
  - color += specular * FindColor(hit2);
- return color;

}

- Recursion might never stop, so set a max recursion depth

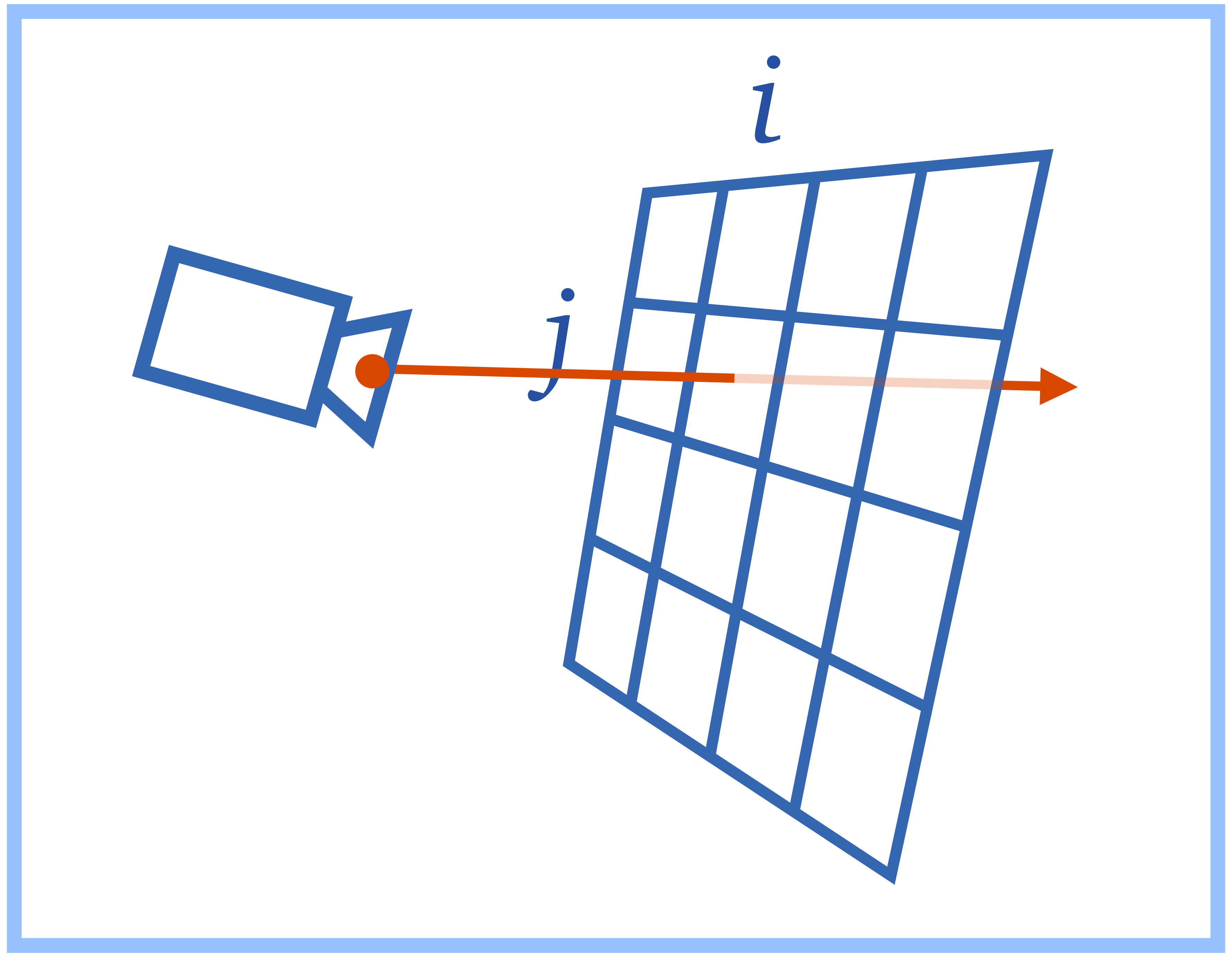


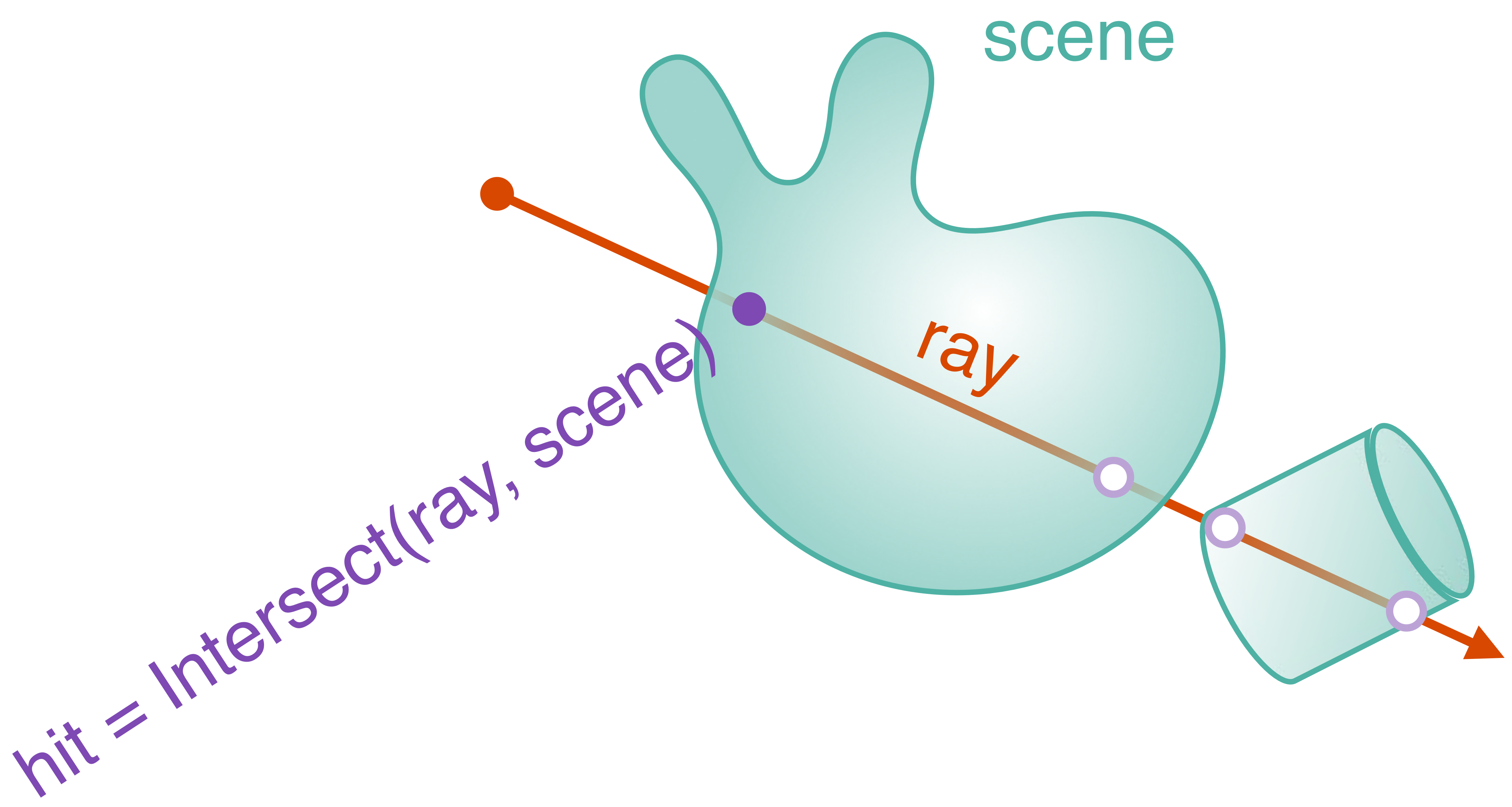
Implementation Details

- Ray tracing framework
- Ray through pixel
- Ray-geometry intersection
- Organizing image and scene
- Global illumination

The essential functions

- Ray ray = RayThruPixel(cam, i, j, width, height)
- Intersection hit = Intersect(ray, scene)





Ray through pixel

- Ray tracing framework
- Ray through pixel
- Ray-geometry intersection
- Organizing image and scene
- Global illumination

Ray

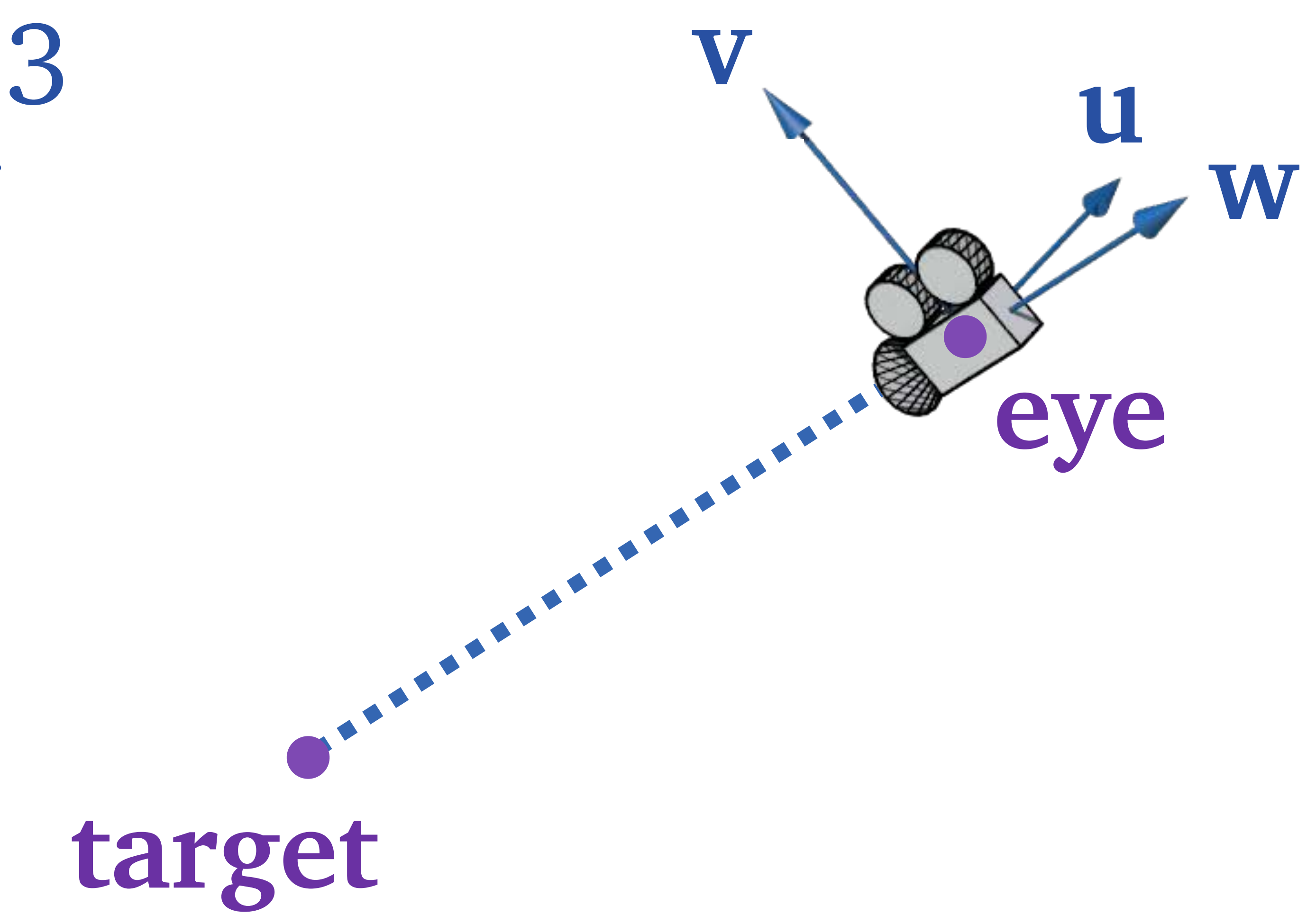
- A **ray** is a described by a point $\mathbf{p}_0 \in \mathbb{R}^3$ and a direction $\mathbf{d} \in \mathbb{R}^3$.
- Mathematically, the ray is a continuous set of points parametrized as

$$\mathbf{p}(t) = \mathbf{p}_0 + t\mathbf{d} \qquad t > 0$$

Camera

- A **camera** has position and orientation described by

$$\mathbf{eye} \in \mathbb{R}^3 \quad \mathbf{u} \in \mathbb{R}^3 \quad \mathbf{v} \in \mathbb{R}^3 \quad \mathbf{w} \in \mathbb{R}^3$$



- Recall that the camera matrix is

$$\mathbf{C} = \begin{bmatrix} | & | & | & | \\ \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{eye} \\ | & | & | & | \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Given $\mathbf{eye} \in \mathbb{R}^3 \quad \mathbf{target} \in \mathbb{R}^3 \quad \mathbf{up} \in \mathbb{R}^3$

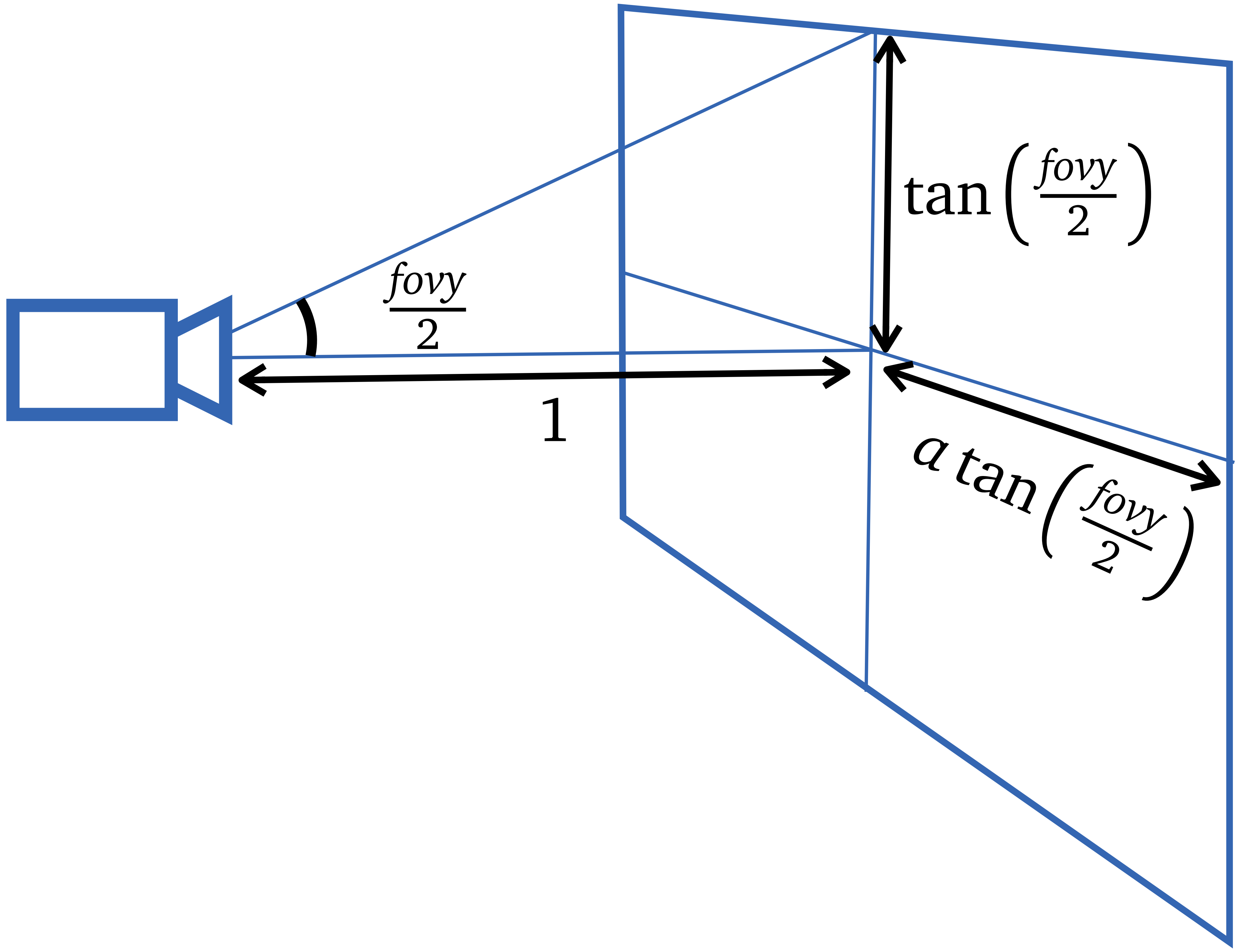
$$\mathbf{w} = \frac{\mathbf{eye} - \mathbf{target}}{|\mathbf{eye} - \mathbf{target}|} \qquad \mathbf{u} = \frac{\mathbf{up} \times \mathbf{w}}{|\mathbf{up} \times \mathbf{w}|} \qquad \mathbf{v} = \mathbf{w} \times \mathbf{u}$$

Camera

- Other relevant parameters:

aspect ratio $a = \frac{\text{width}}{\text{height}}$

field of view (angle) $fovy$



Ray through pixel

- Given camera $\mathbf{eye} \in \mathbb{R}^3$ $\mathbf{u} \in \mathbb{R}^3$ $\mathbf{v} \in \mathbb{R}^3$ $\mathbf{w} \in \mathbb{R}^3$

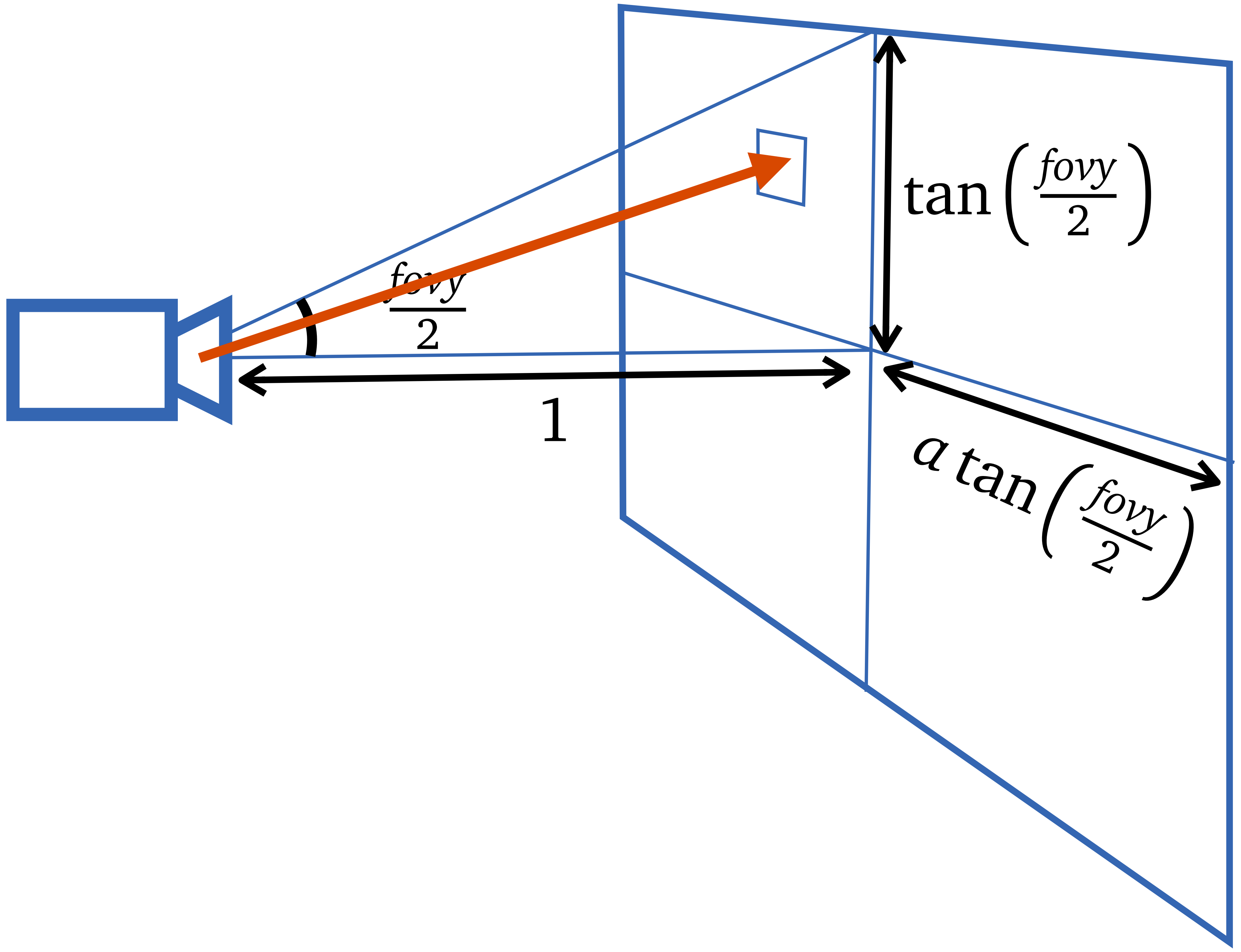
  $a = \frac{\text{width}}{\text{height}}$ $\quad fovy$

- Given pixel (i, j)

  $i \in \{0, \ldots, \text{width} - 1\}$

  $j \in \{0, \ldots, \text{height} - 1\}$

  (index space)



- Our goal is to work out the ray through the center of the pixel

Ray through pixel

- If screen ranges from (0,0) to (1,1) from top-left to bottom right

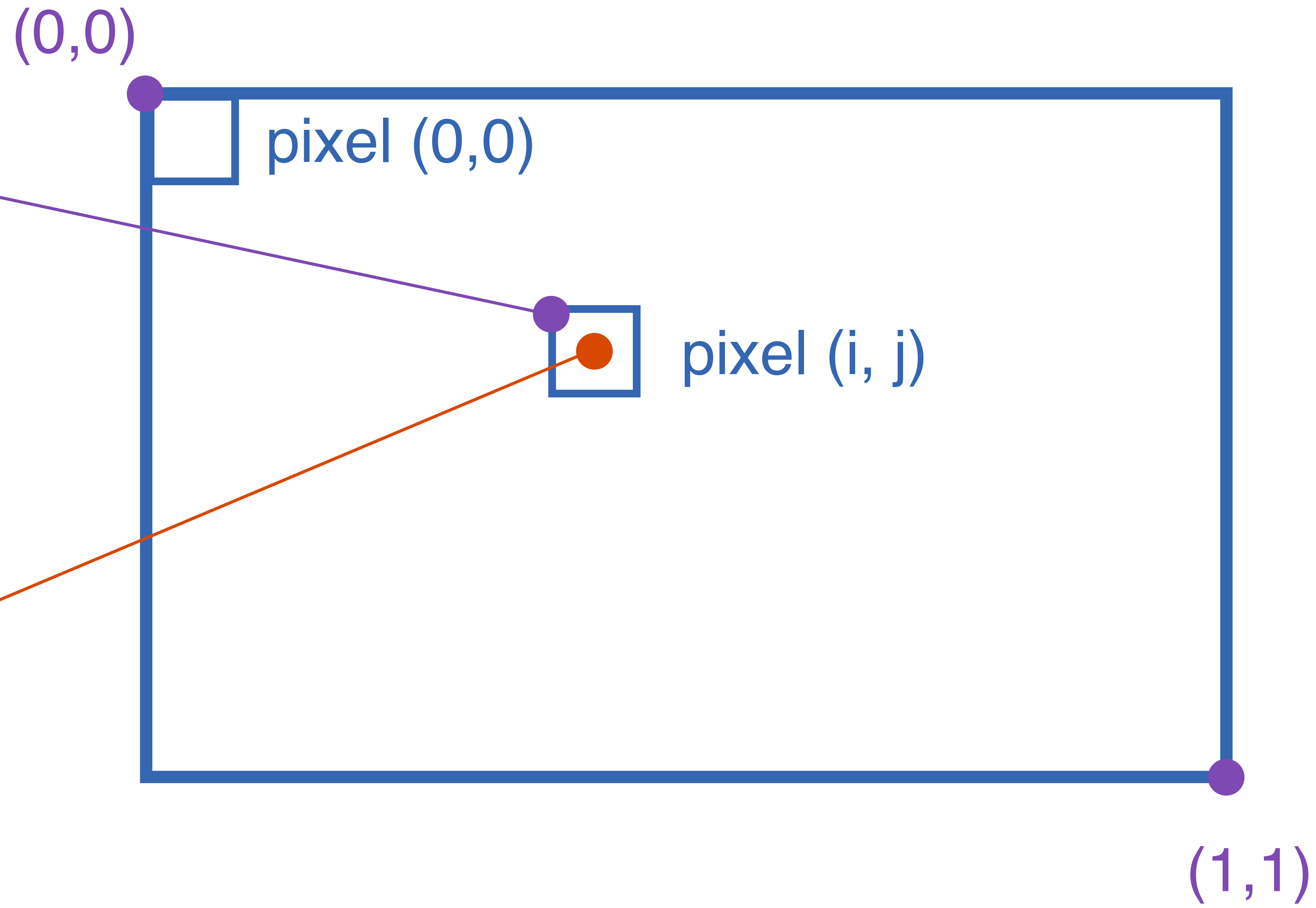
(screen space coordinate)

- The corner of pixel (i, j)

$$\left(\frac{i}{\text{width}}, \frac{j}{\text{height}}\right)_{\text{(screen)}}$$

- The center of pixel (i, j)

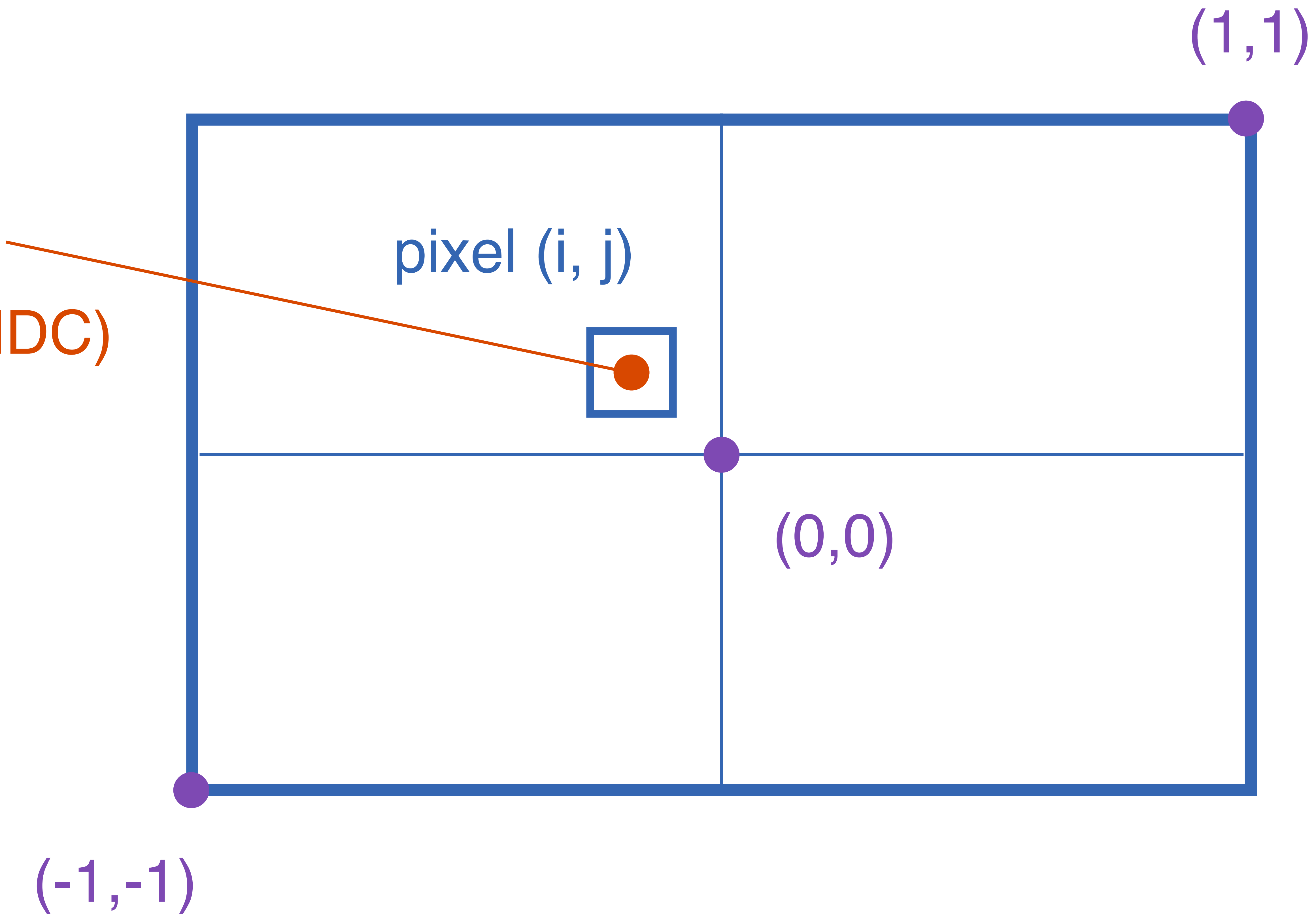
$$\left(\frac{i+\frac{1}{2}}{\text{width}}, \frac{j+\frac{1}{2}}{\text{height}}\right)_{\text{(screen)}}$$



Ray through pixel

- If screen ranges from (-1,-1) to (1,1) from bottom-left to top right (normalized device coordinate NDC)
- The center of pixel (i, j)

$$\left(2 \cdot \frac{i+\frac{1}{2}}{\text{width}} - 1, 1 - 2 \cdot \frac{j+\frac{1}{2}}{\text{height}}\right)$$ (NDC)



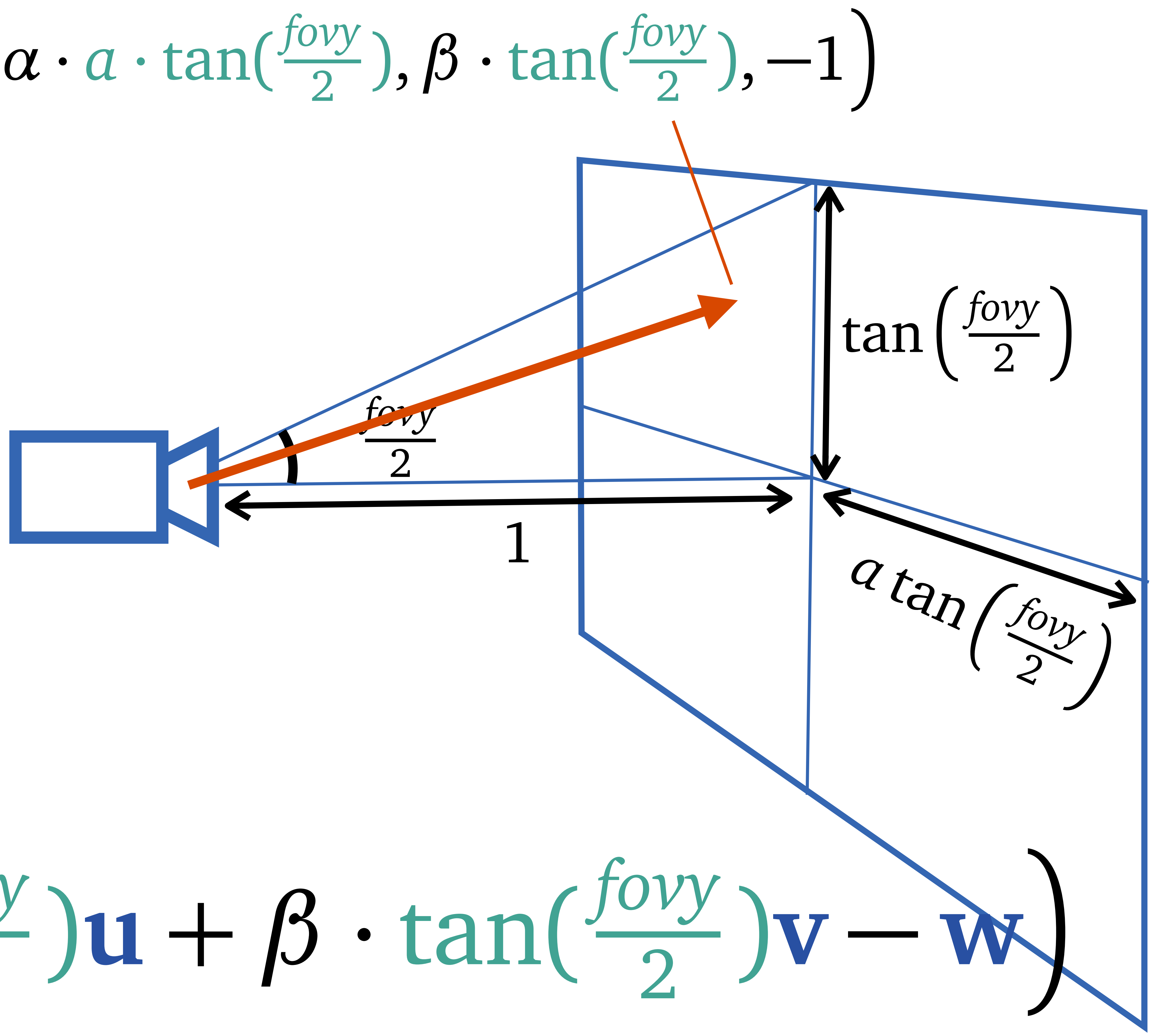
Define

$$\alpha = 2 \cdot \frac{i+\frac{1}{2}}{\text{width}} - 1$$

$$\beta = 1 - 2 \cdot \frac{j+\frac{1}{2}}{\text{height}}$$

Ray through pixel

- Given camera $\mathbf{eye} \in \mathbb{R}^3$ $\mathbf{u} \in \mathbb{R}^3$ $\mathbf{v} \in \mathbb{R}^3$ $\mathbf{w} \in \mathbb{R}^3$ $a = \frac{\text{width}}{\text{height}}$ $fovy$
- Given pixel (i, j)
- In camera coordinate,
  - Source of ray = (0,0,0)
  - Ray passes through $\left(\alpha \cdot a \cdot \tan(\frac{fovy}{2}), \beta \cdot \tan(\frac{fovy}{2}), -1\right)$
- In world, the ray is given by



$$\mathbf{p}_0 = \mathbf{eye}$$

$$\mathbf{d} = \text{NORMALIZE}\left(\alpha \cdot a \cdot \tan(\frac{fovy}{2})\mathbf{u} + \beta \cdot \tan(\frac{fovy}{2})\mathbf{v} - \mathbf{w}\right)$$

Ray through pixel

- Given camera $\mathbf{eye} \in \mathbb{R}^3$ $\mathbf{u} \in \mathbb{R}^3$ $\mathbf{v} \in \mathbb{R}^3$ $\mathbf{w} \in \mathbb{R}^3$ $a = \frac{\text{width}}{\text{height}}$ $fovy$
- Given pixel (i, j)
- In world coordinate,

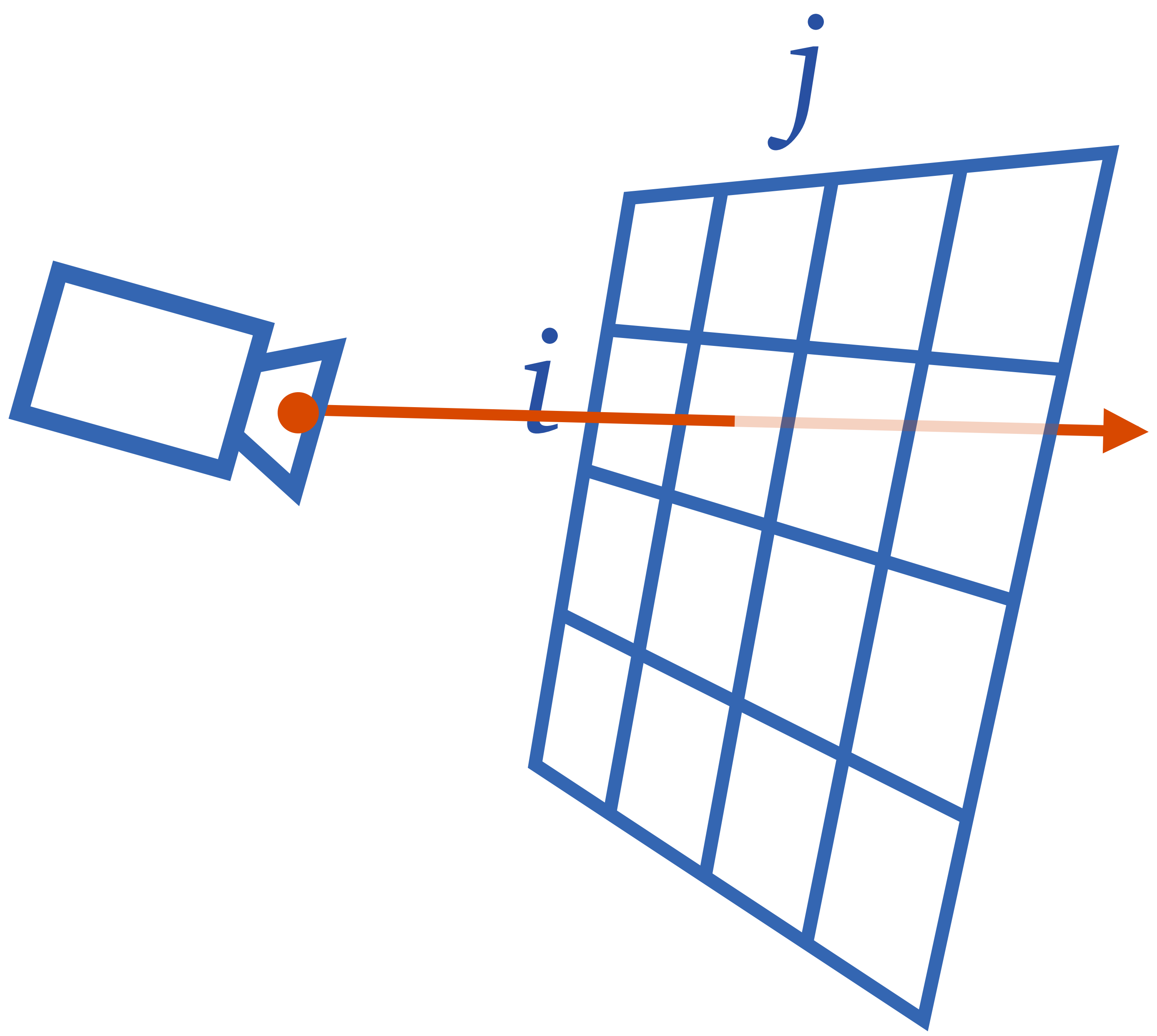
$$\mathbf{p}_0 = \mathbf{eye}$$

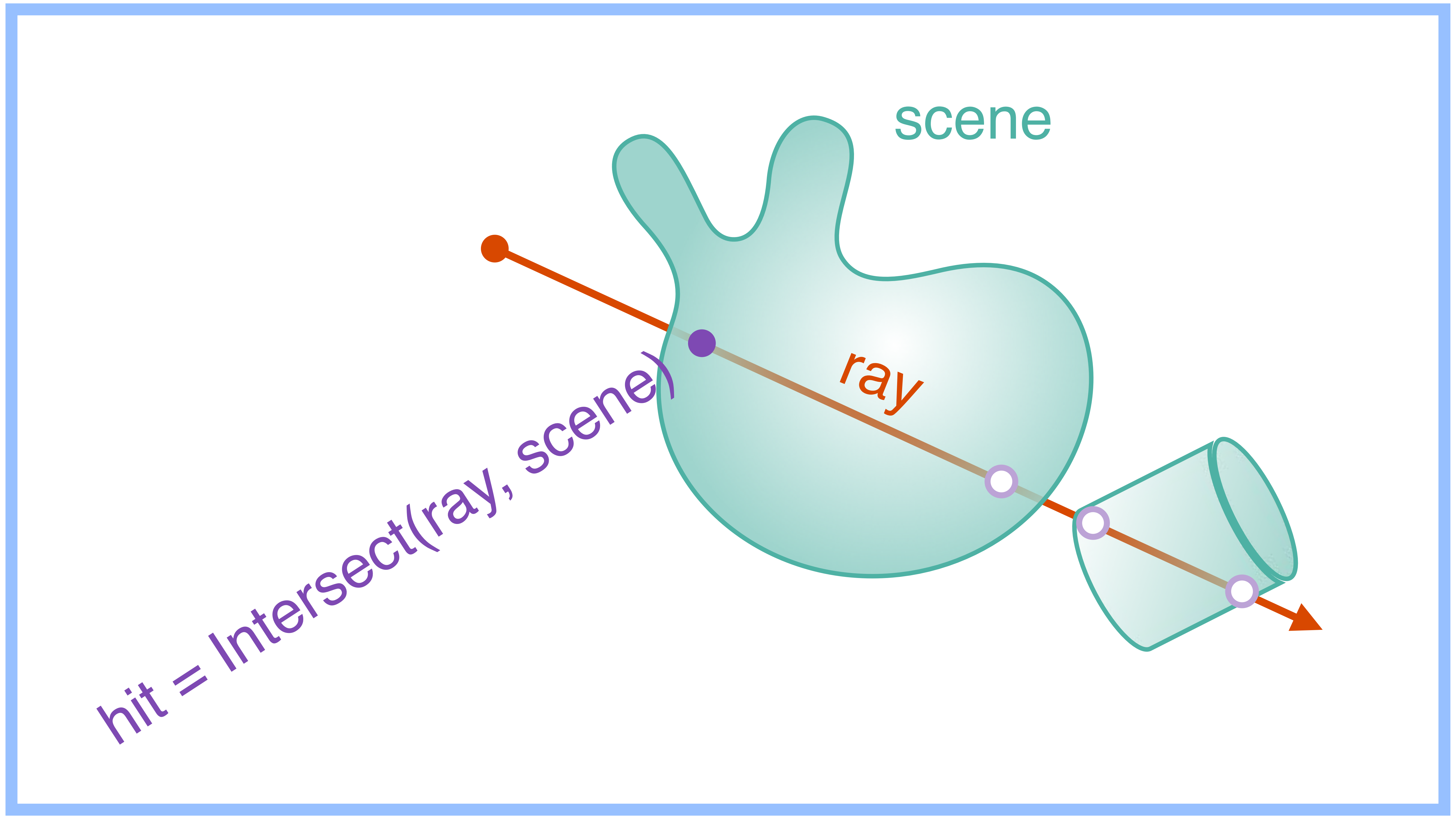
$$\mathbf{d} = \textsc{Normalize}\left(\alpha \cdot a \cdot \tan(\tfrac{fovy}{2})\mathbf{u} + \beta \cdot \tan(\tfrac{fovy}{2})\mathbf{v} - \mathbf{w}\right)$$

$$\alpha = 2 \cdot \frac{i + \frac{1}{2}}{\text{width}} - 1 \qquad \beta = 1 - 2 \cdot \frac{j + \frac{1}{2}}{\text{height}}$$

The essential functions

- Ray ray = RayThruPixel(cam, i, j, width, height)
- Intersection hit = Intersect(ray, scene)



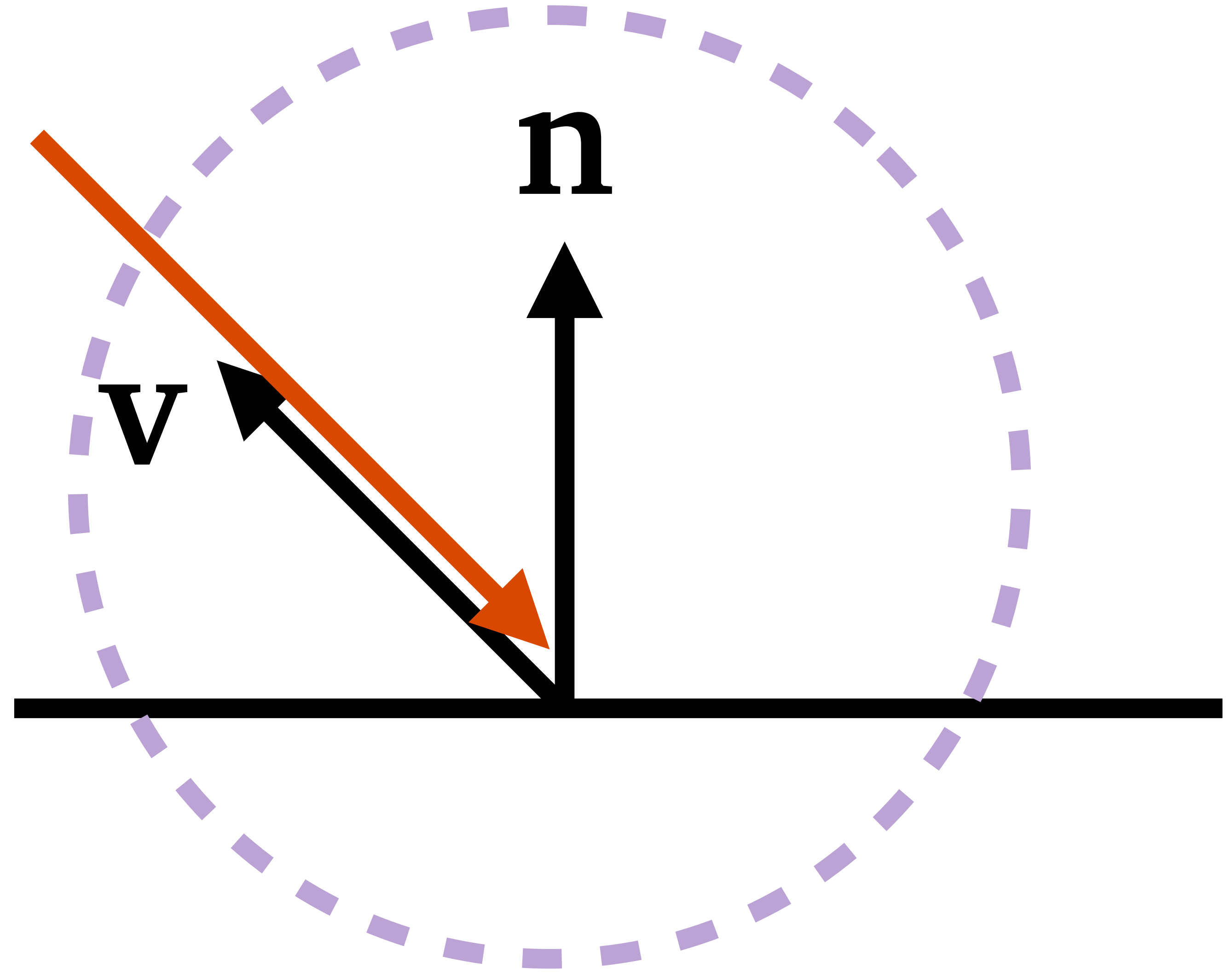


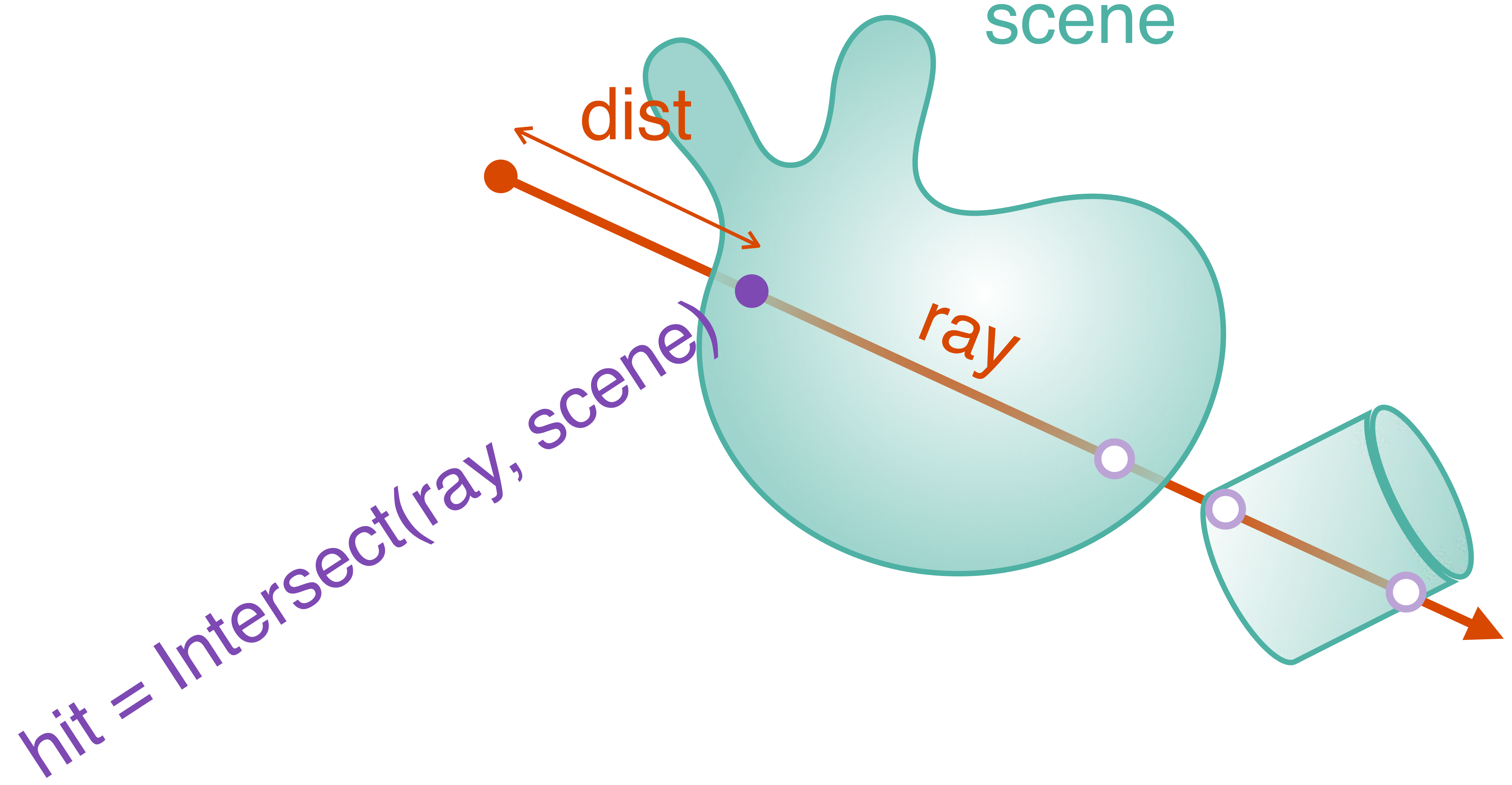
Intersection

- Ray tracing framework
- Ray through pixel
- Ray-geometry intersection
- Organizing image and scene
- Global illumination

Information in intersection

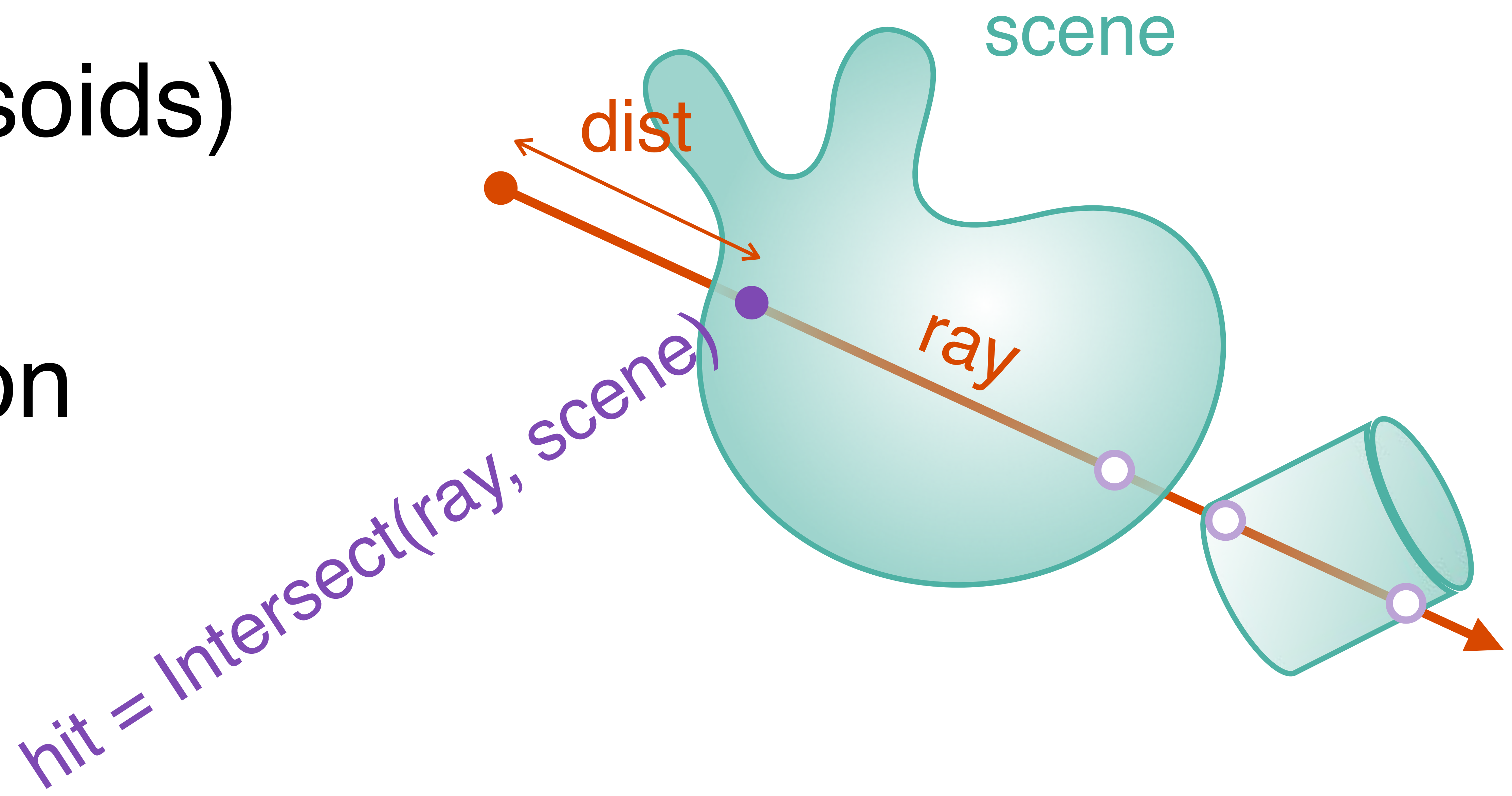
- A ray-scene intersection contains the following information
  - Position of the intersection
  - Surface normal **n**
  - Direction to the in-coming ray **v**
  - Pointers to material, or object etc
  - *Distance to the source of ray*





Ray-object intersection

- The core helper function is
  - Intersection Intersect(Ray ray, Object element);
- Elements can be
  - Triangle
  - Sphere
  - Transformations of sphere (ellipsoids)
  - … any other element which you know how to compute intersection



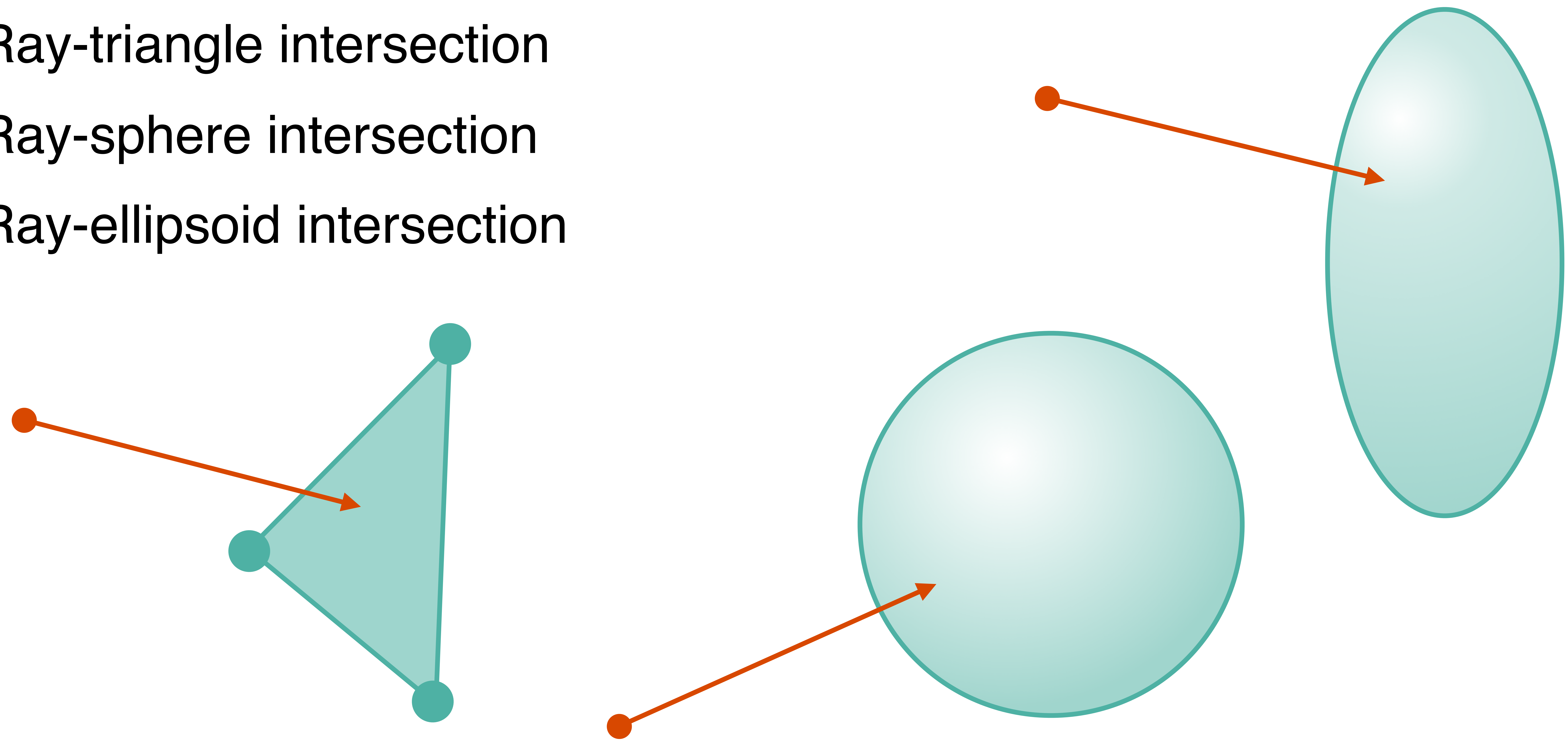
Ray-scene intersection

- Once we have ray-object intersection
- Ray-scene intersection follows the pseudocode

```
Intersection Intersect(Ray ray, Scene scene){
  Distance mindist = INFINITY;
  Intersection hit;
  foreach (object in scene){  // Find closest intersection; test all objects
    Intersection hit_temp = Intersect(ray, object);
    if (hit_temp.dist< mindist){  // closer than previous hit
      mindist = hit_temp.dist;
      hit = hit_temp;
    }
  }
  return hit;
}
```

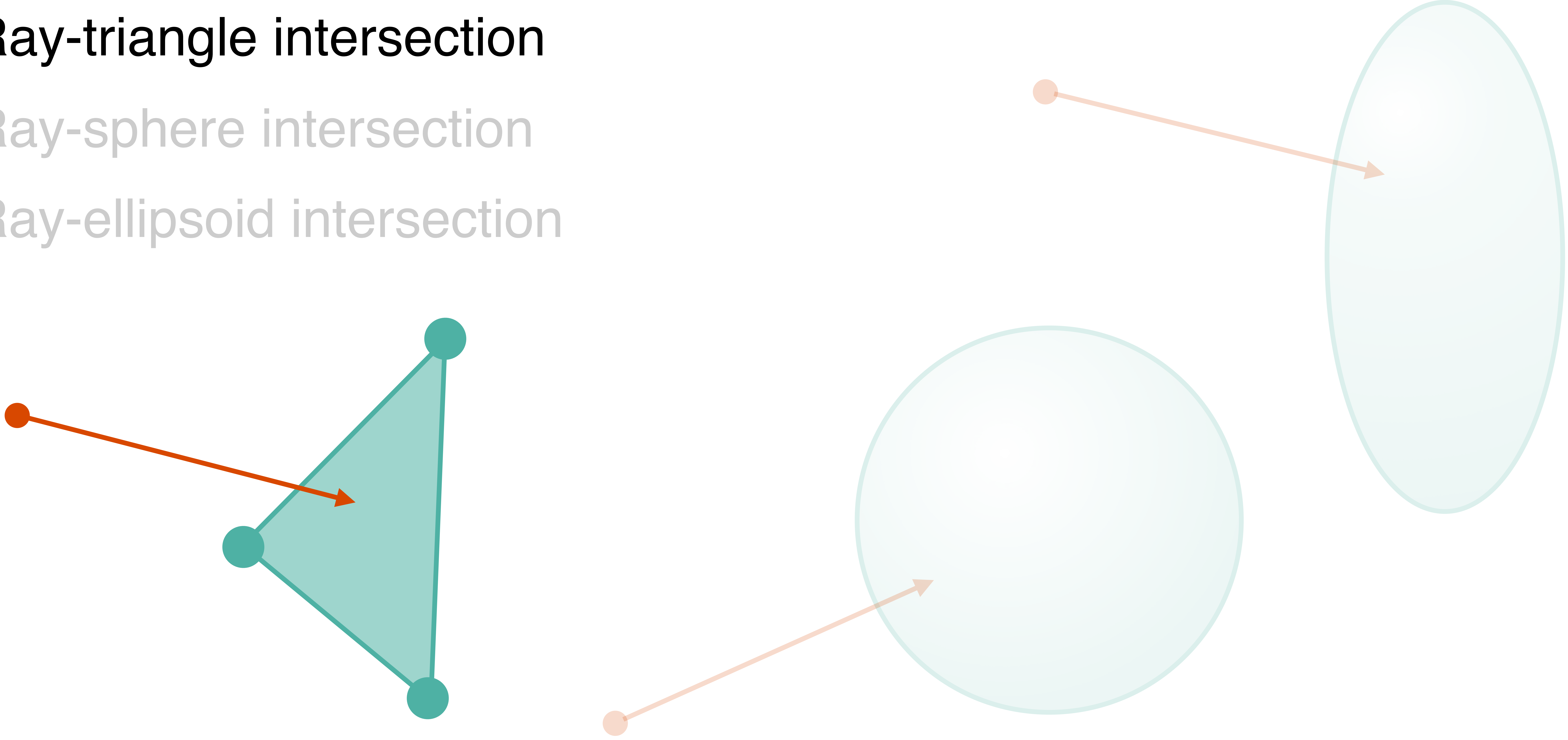
Ray-object intersection

- We will focus on
  - Ray-triangle intersection
  - Ray-sphere intersection
  - Ray-ellipsoid intersection



Ray-object intersection

- We will focus on
  - Ray-triangle intersection
  - Ray-sphere intersection
  - Ray-ellipsoid intersection



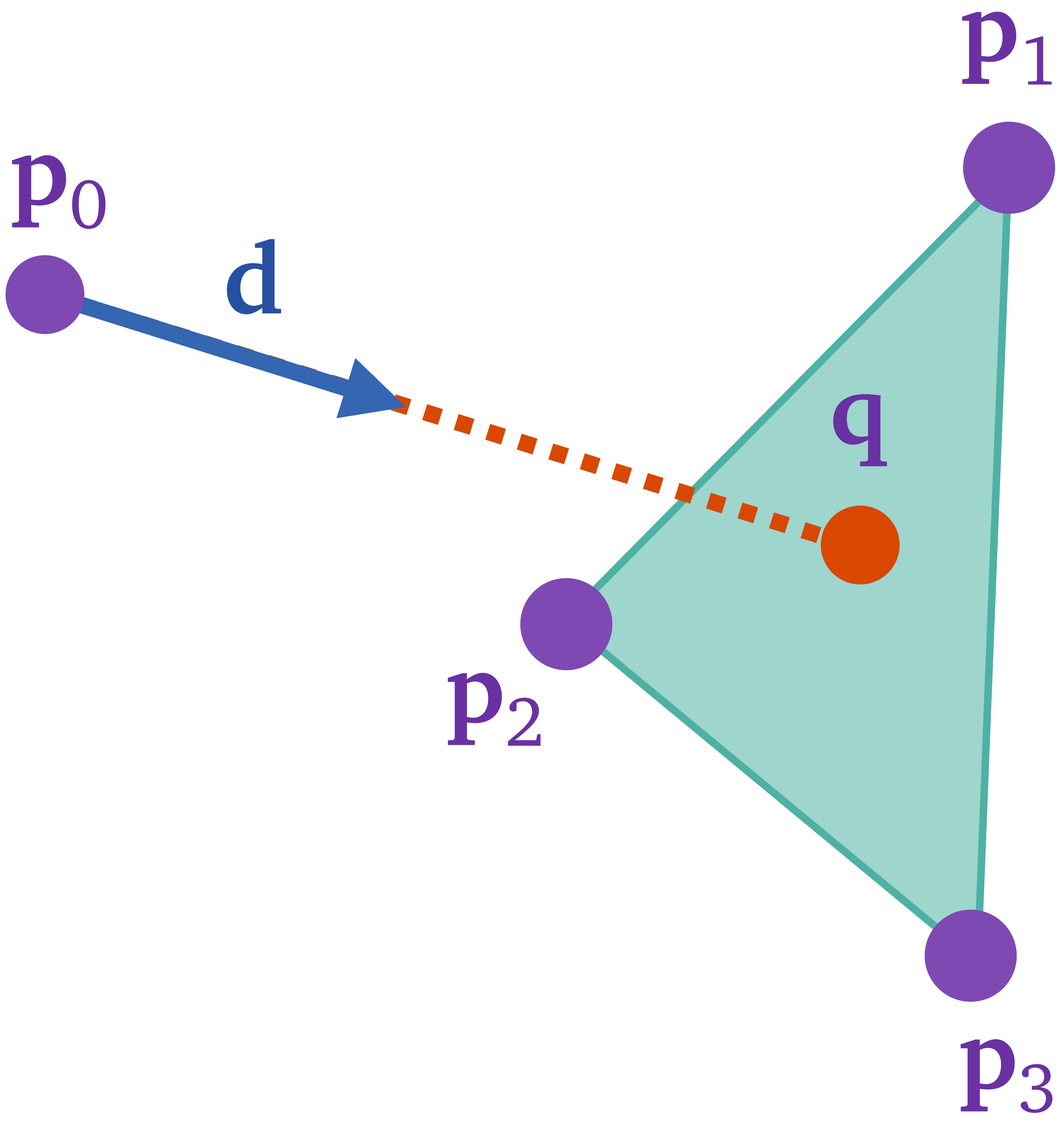
Ray-triangle intersection

- Given ray $(\mathbf{p}_0, \mathbf{d})$
- Given triangle $\mathbf{p}_1\ \mathbf{p}_2\ \mathbf{p}_3$
- Any point along the ray takes the form

$$\mathbf{q} = \mathbf{p}_0 + t\mathbf{d}$$

- Any point on the plane spanned by the triangle takes the form

$$\mathbf{q} = \lambda_1\mathbf{p}_1 + \lambda_2\mathbf{p}_2 + \lambda_3\mathbf{p}_3$$
$$\lambda_1 + \lambda_2 + \lambda_3 = 1$$



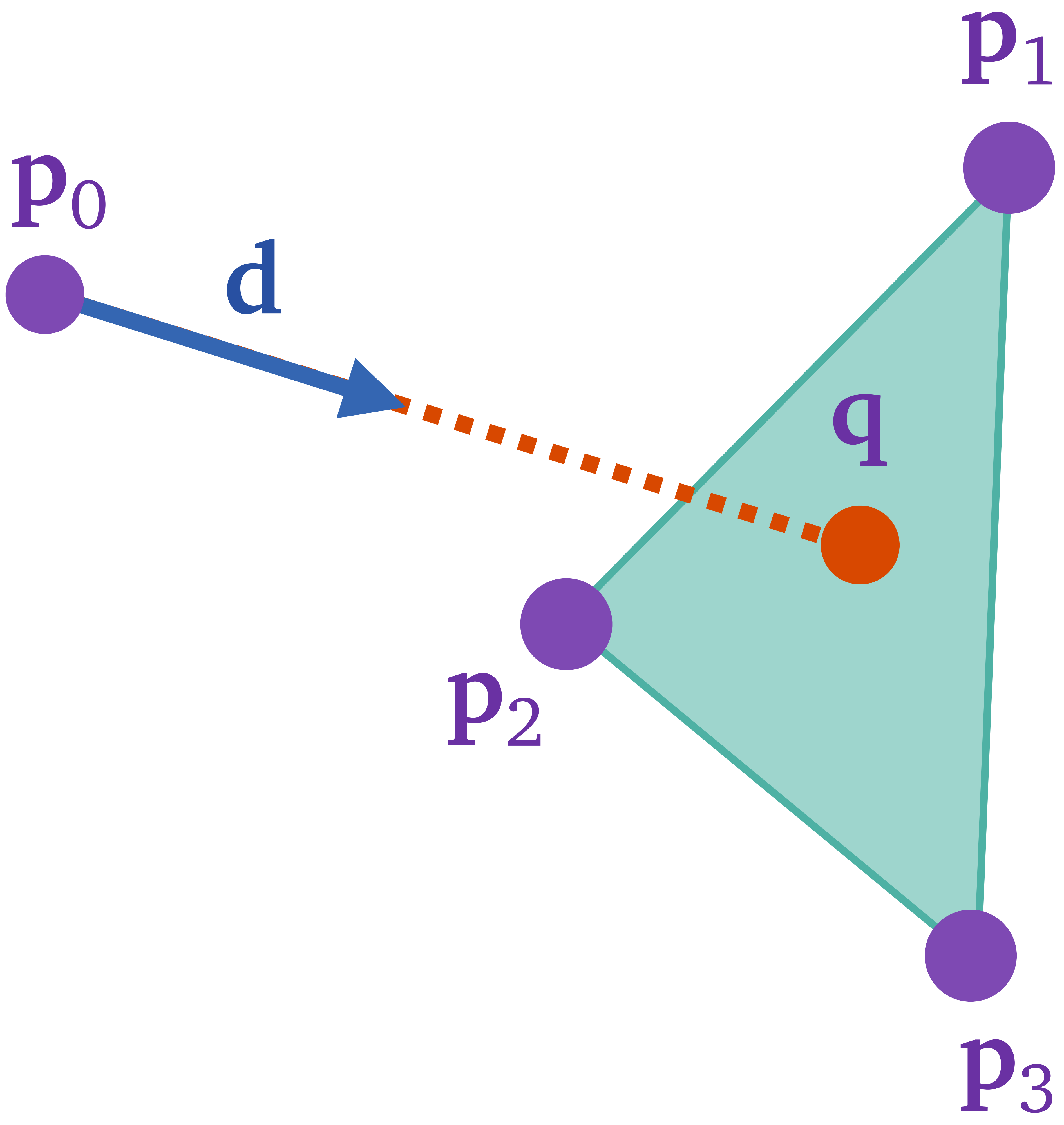
Ray-triangle intersection

$$\mathbf{q} = \mathbf{p}_0 + t\mathbf{d}$$

$$\mathbf{q} = \lambda_1\mathbf{p}_1 + \lambda_2\mathbf{p}_2 + \lambda_3\mathbf{p}_3$$

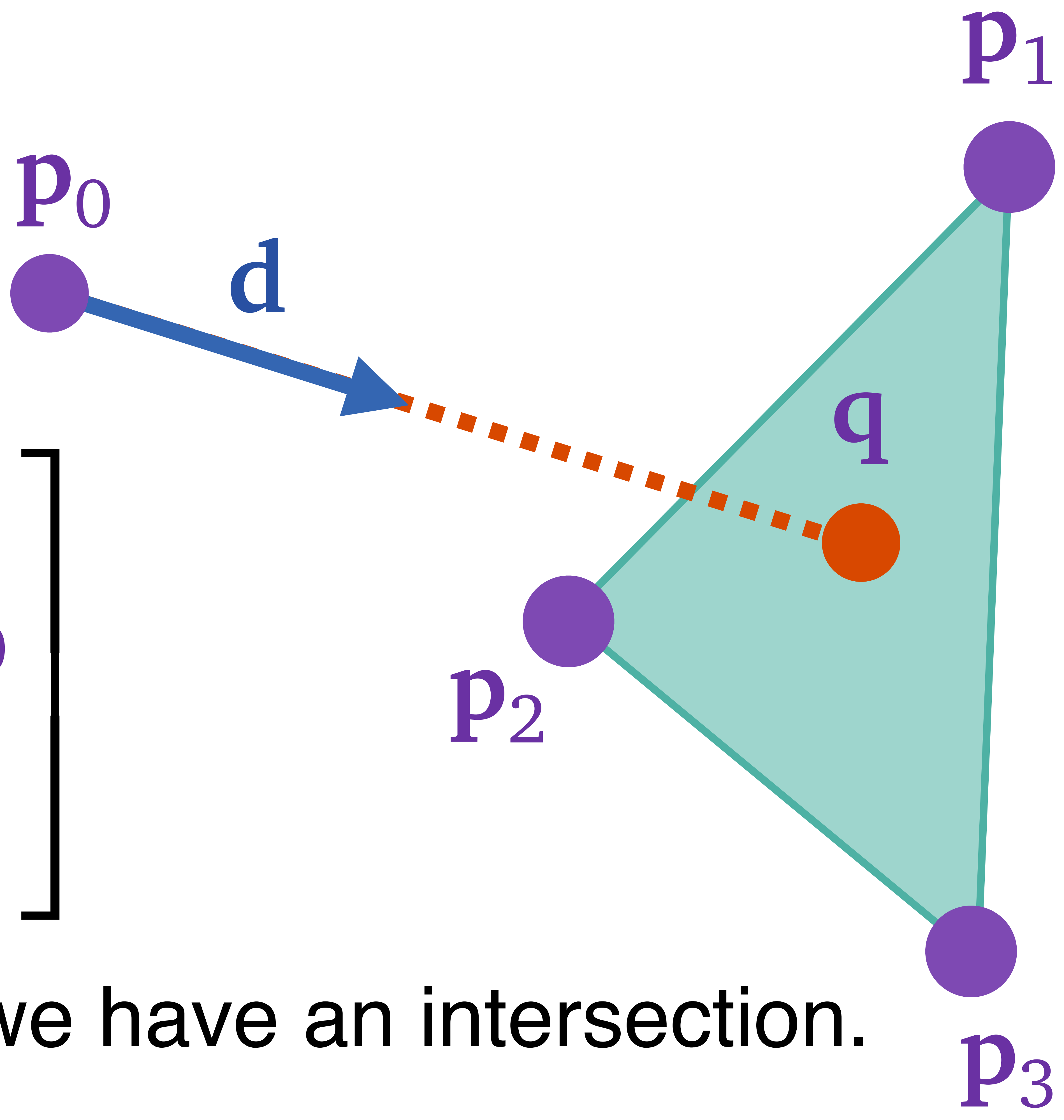
$$\lambda_1 + \lambda_2 + \lambda_3 = 1$$

$$\begin{cases} \lambda_1\mathbf{p}_1 + \lambda_2\mathbf{p}_2 + \lambda_3\mathbf{p}_3 - t\mathbf{d} = \mathbf{p}_0 \\ \lambda_1 + \lambda_2 + \lambda_3 = 1 \end{cases}$$



Ray-triangle intersection

$$\begin{cases} \lambda_1 \mathbf{p}_1 + \lambda_2 \mathbf{p}_2 + \lambda_3 \mathbf{p}_3 - t\mathbf{d} = \mathbf{p}_0 \\ \lambda_1 + \lambda_2 + \lambda_3 = 1 \end{cases}$$



Solve

$$\begin{bmatrix} | & | & | & | \\ \mathbf{p}_1 & \mathbf{p}_2 & \mathbf{p}_3 & -\mathbf{d} \\ | & | & | & | \\ 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ t \end{bmatrix} = \begin{bmatrix} | \\ \mathbf{p}_0 \\ | \\ 1 \end{bmatrix}$$

If all $\lambda_1, \lambda_2, \lambda_3$ and t are ≥ 0 then we have an intersection.

Ray-triangle intersection

- If we have an intersection, use the barycentric coordinate (what we just solved)

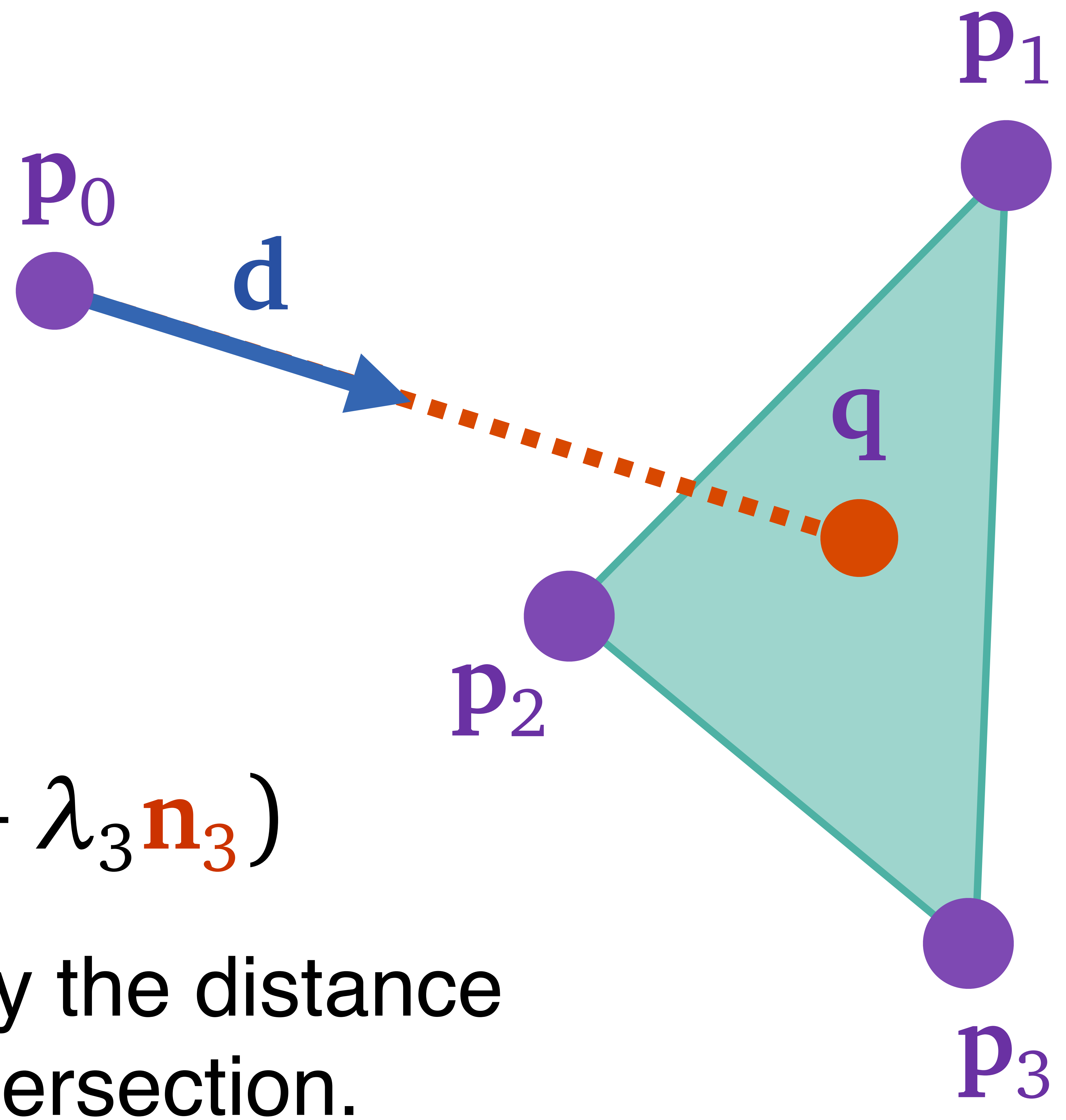
$$\lambda_1, \lambda_2, \lambda_3$$

to interpolate position and vertex attributes, such as normals

$$\mathbf{q} = \lambda_1 \mathbf{p}_1 + \lambda_2 \mathbf{p}_2 + \lambda_3 \mathbf{p}_3$$

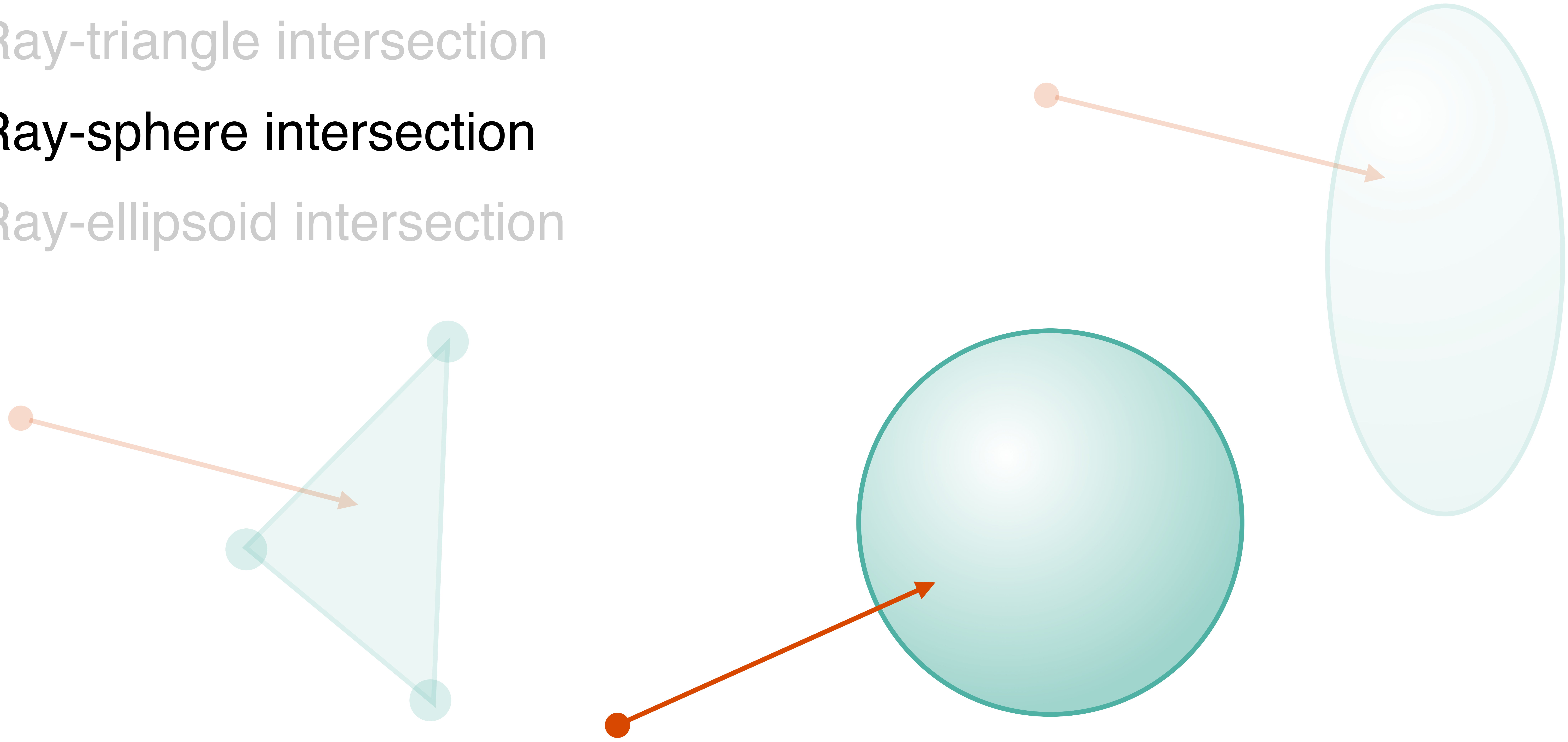
$$\mathbf{n} = \text{normalize}(\lambda_1 \mathbf{n}_1 + \lambda_2 \mathbf{n}_2 + \lambda_3 \mathbf{n}_3)$$

- The variable t we solved is exactly the distance between the ray source and the intersection.



Ray-object intersection

- We will focus on
  - Ray-triangle intersection
  - Ray-sphere intersection
  - Ray-ellipsoid intersection

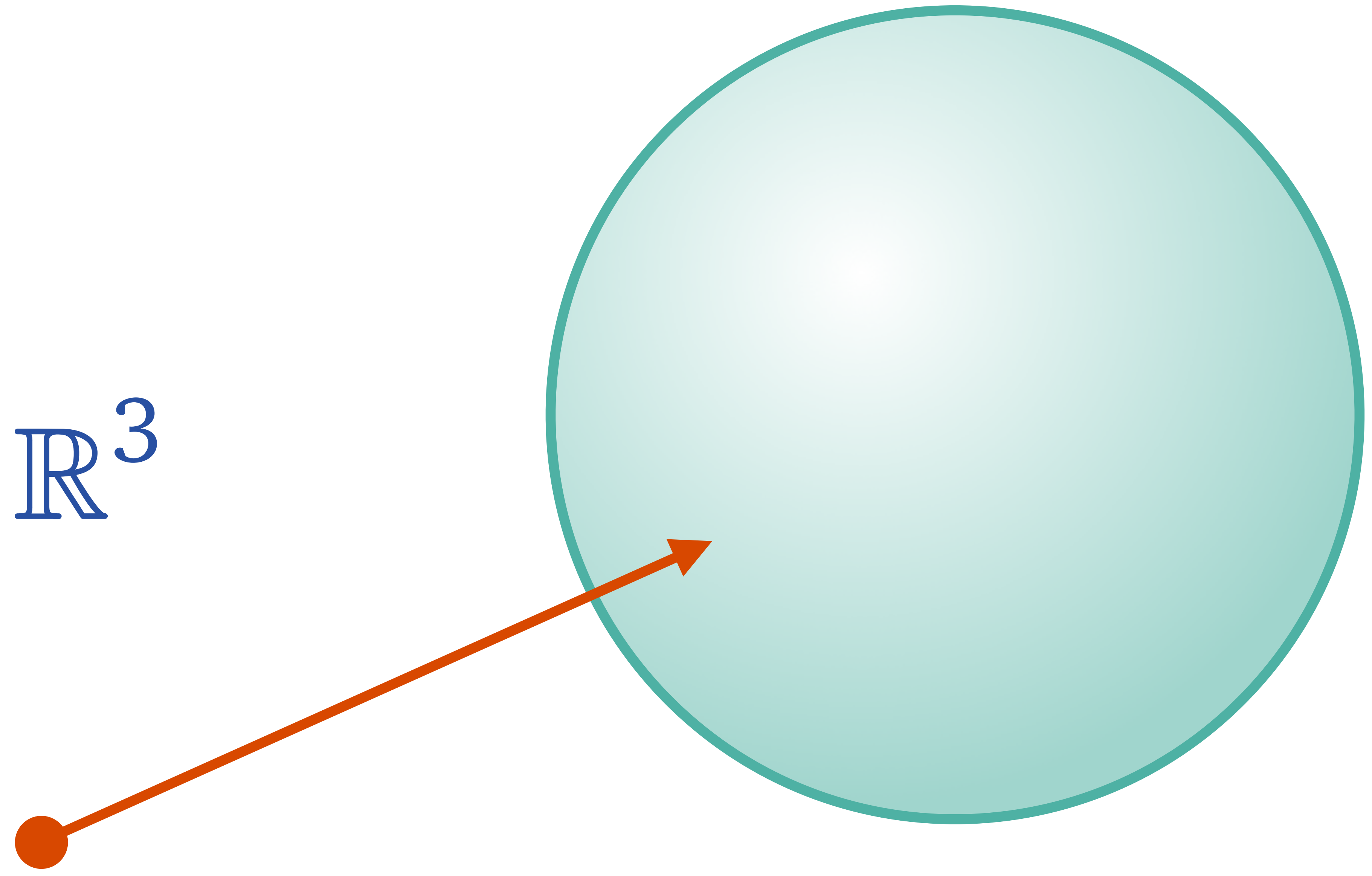


Ray-sphere intersection

- Sphere representation
  - Center $\mathbf{c} \in \mathbb{R}^3$
  - Radius $r > 0$
- A point $\mathbf{q} \in \mathbb{R}^3$ lies on the sphere if and only if

$$(\mathbf{q} - \mathbf{c}) \cdot (\mathbf{q} - \mathbf{c}) = r^2$$

- Ray representation
  - Source $\mathbf{p}_0 \in \mathbb{R}^3$ and direction $\mathbf{d} \in \mathbb{R}^3$
- Any point along the ray takes the form $\mathbf{q} = \mathbf{p}_0 + t\mathbf{d}$



Ray-sphere intersection

$$(\mathbf{q}-\mathbf{c})\cdot(\mathbf{q}-\mathbf{c})=r^2$$

$$\mathbf{q}=\mathbf{p}_0+t\mathbf{d}$$

- Substitution

$$(\mathbf{p}_0+t\mathbf{d}-\mathbf{c})\cdot(\mathbf{p}_0+t\mathbf{d}-\mathbf{c})=r^2$$

- Expand

$$|\mathbf{d}|^2t^2+2\mathbf{d}\cdot(\mathbf{p}_0-\mathbf{c})t+|\mathbf{p}_0-\mathbf{c}|^2-r^2=0$$

- The ray direction is always normalized $|\mathbf{d}|=1$

$$t^2+2\mathbf{d}\cdot(\mathbf{p}_0-\mathbf{c})t+|\mathbf{p}_0-\mathbf{c}|^2-r^2=0$$

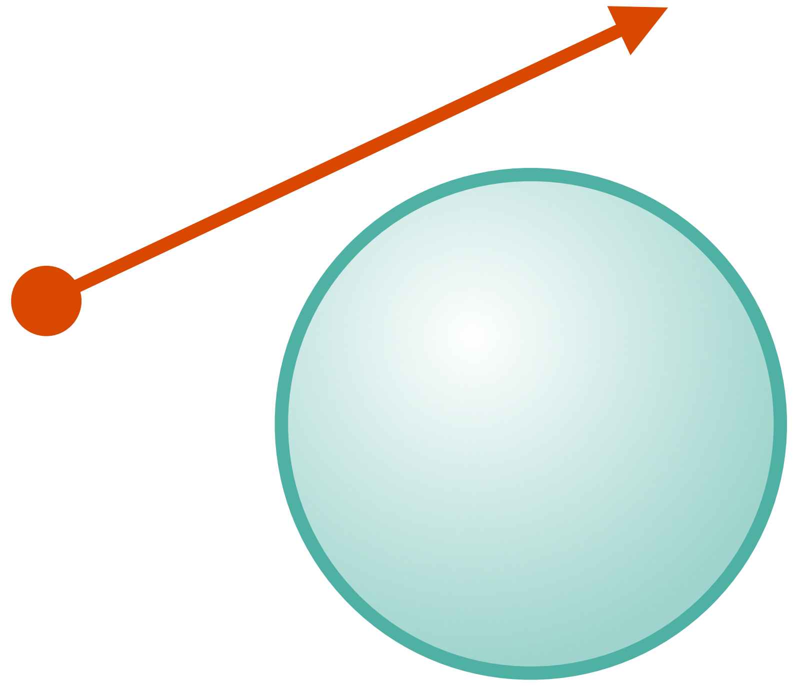
Ray-sphere intersection

$$t^2 + 2\mathbf{d}\cdot(\mathbf{p}_0 - \mathbf{c})t + |\mathbf{p}_0 - \mathbf{c}|^2 - r^2 = 0$$

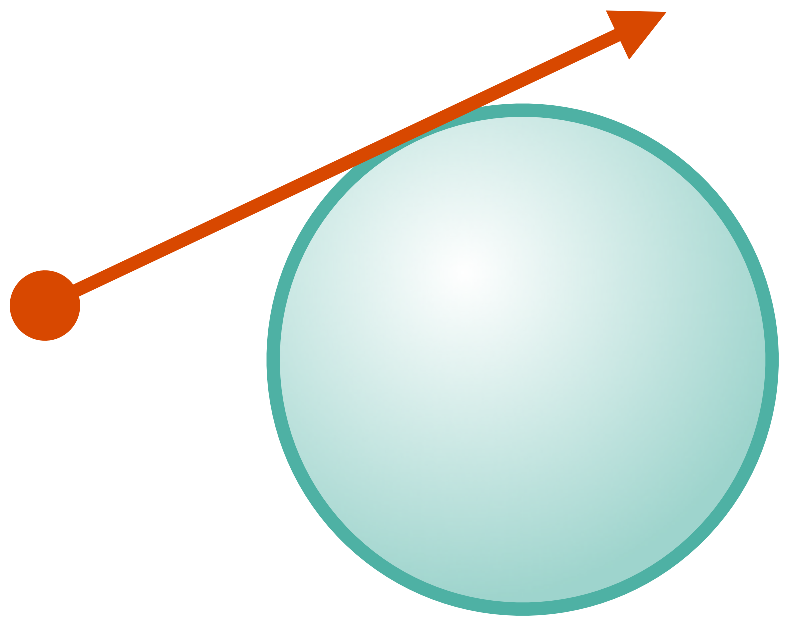
- Quadratic formula

$$t = -\mathbf{d}\cdot(\mathbf{p}_0 - \mathbf{c}) \pm \sqrt{(\mathbf{d}\cdot(\mathbf{p}_0 - \mathbf{c}))^2 - |\mathbf{p}_0 - \mathbf{c}|^2 + r^2}$$

- If the expression in $\sqrt{\cdot}$ is negative
  - no intersection



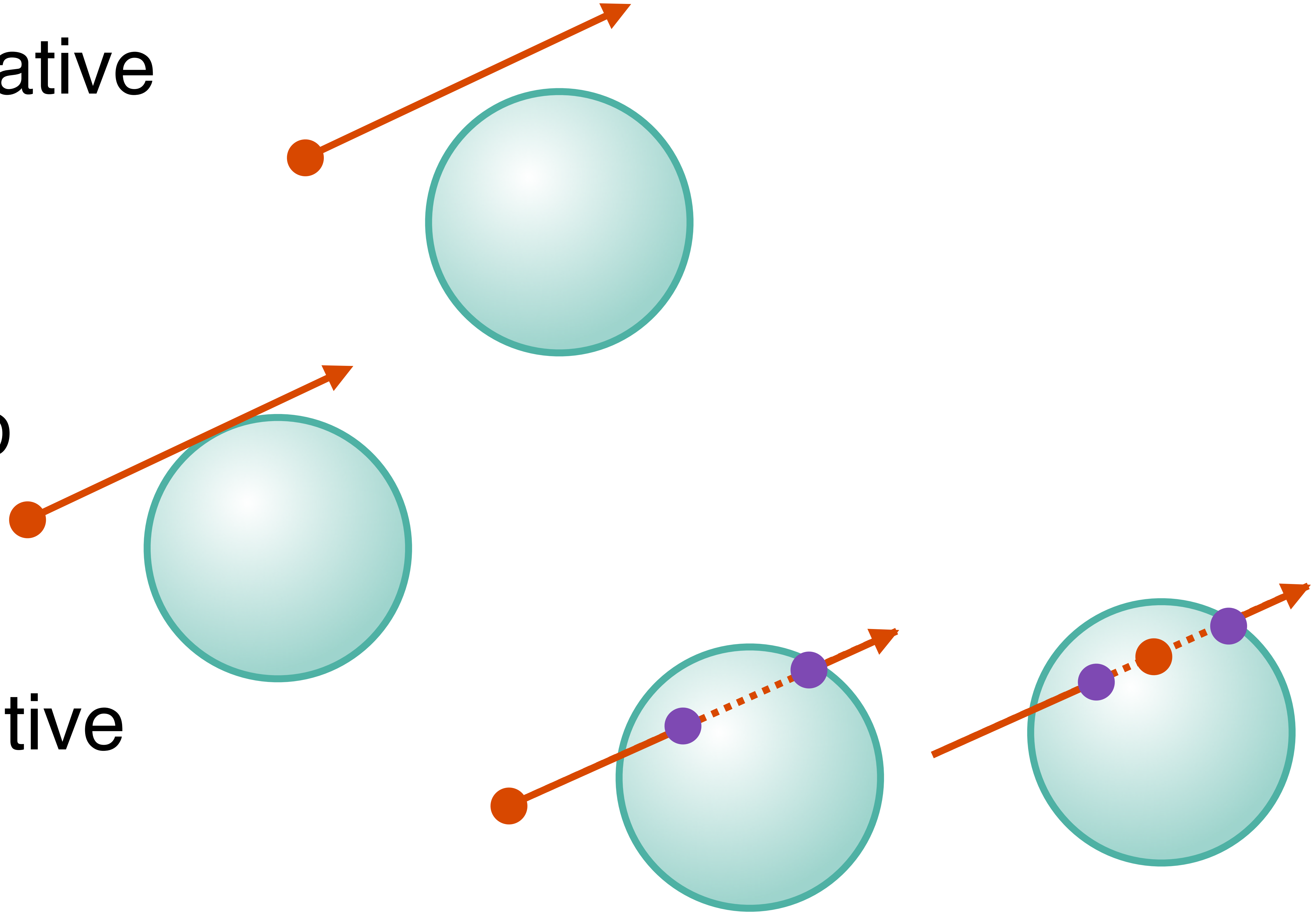
- If the expression in $\sqrt{\cdot}$ is zero
  - tangent



Ray-sphere intersection

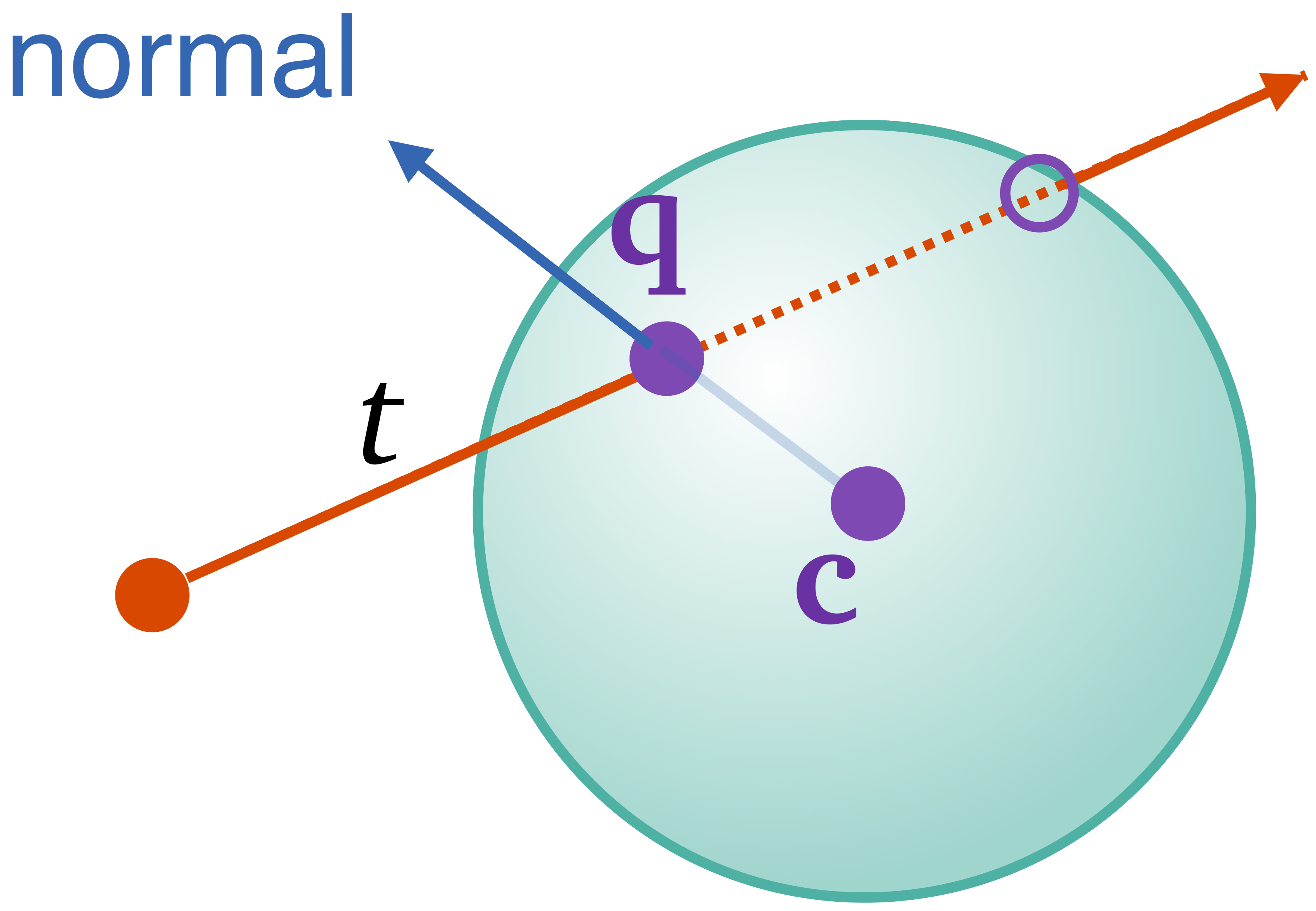
$$t = -\mathbf{d} \cdot (\mathbf{p}_0 - \mathbf{c}) \pm \sqrt{(\mathbf{d} \cdot (\mathbf{p}_0 - \mathbf{c}))^2 - |\mathbf{p}_0 - \mathbf{c}|^2 + r^2}$$

- If the expression in $\sqrt{\cdot}$ is negative
  - no intersection
- If the expression in $\sqrt{\cdot}$ is zero
  - tangent
- If the expression in $\sqrt{\cdot}$ is positive
  - two intersections
  - Need to take the smallest positive t



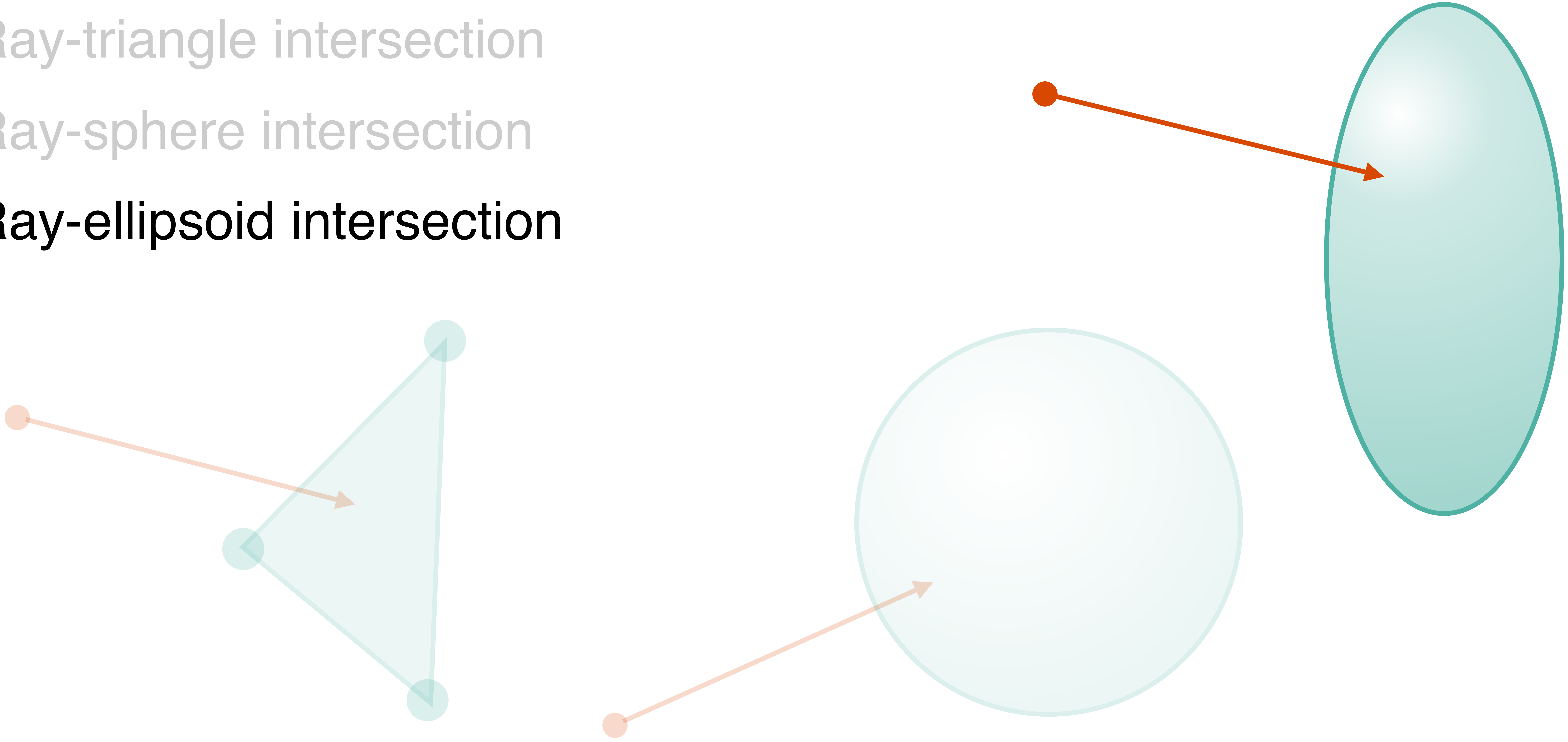
Ray-sphere intersection

- Once we find t (which is distance to the source)
- Position is given by $\mathbf{q} = \mathbf{p}_0 + t\mathbf{d}$
- Normal is given by $\text{normalize}(\mathbf{q} - \mathbf{c})$



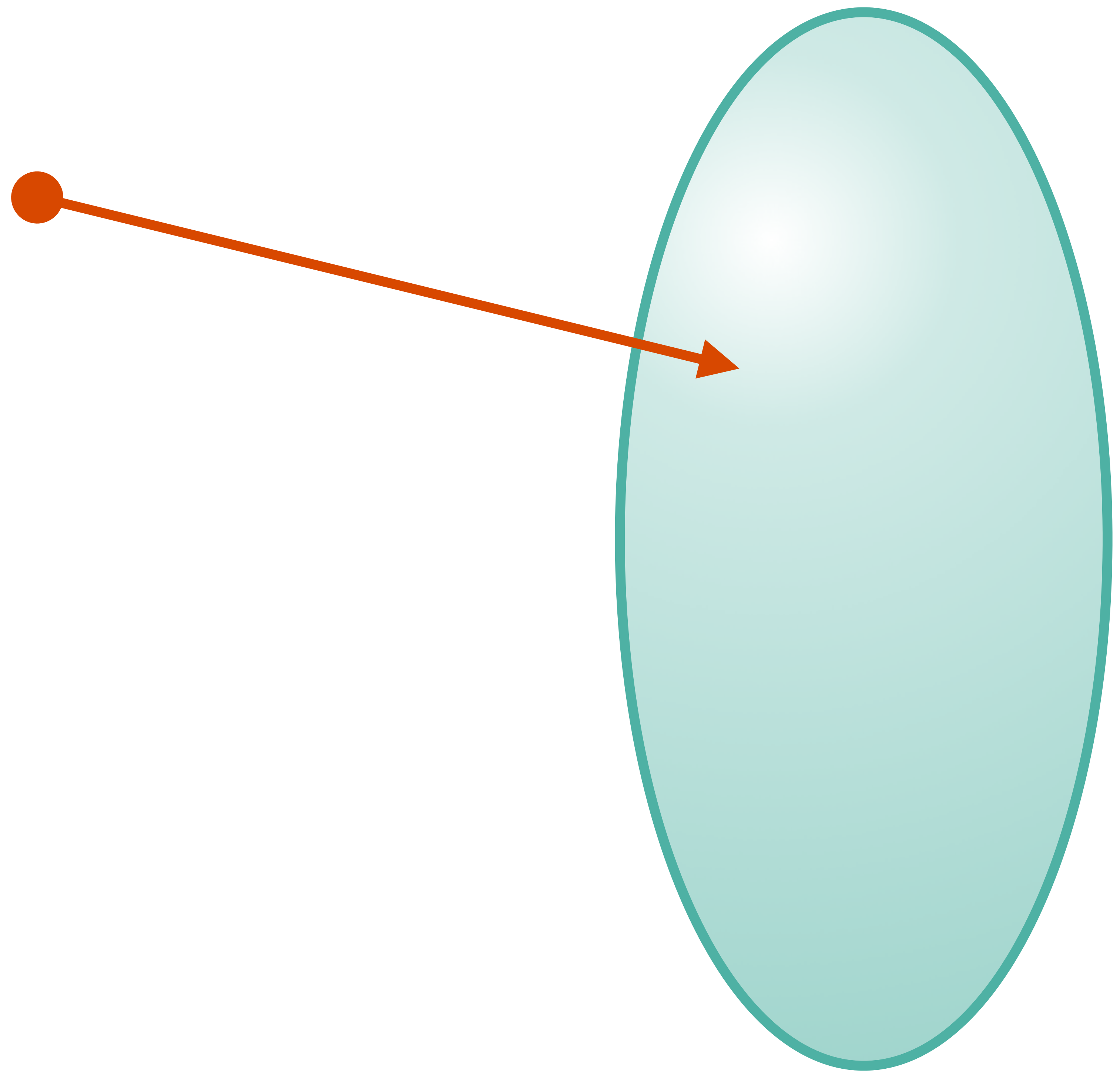
Ray-object intersection

- We will focus on
  - Ray-triangle intersection
  - Ray-sphere intersection
  - Ray-ellipsoid intersection



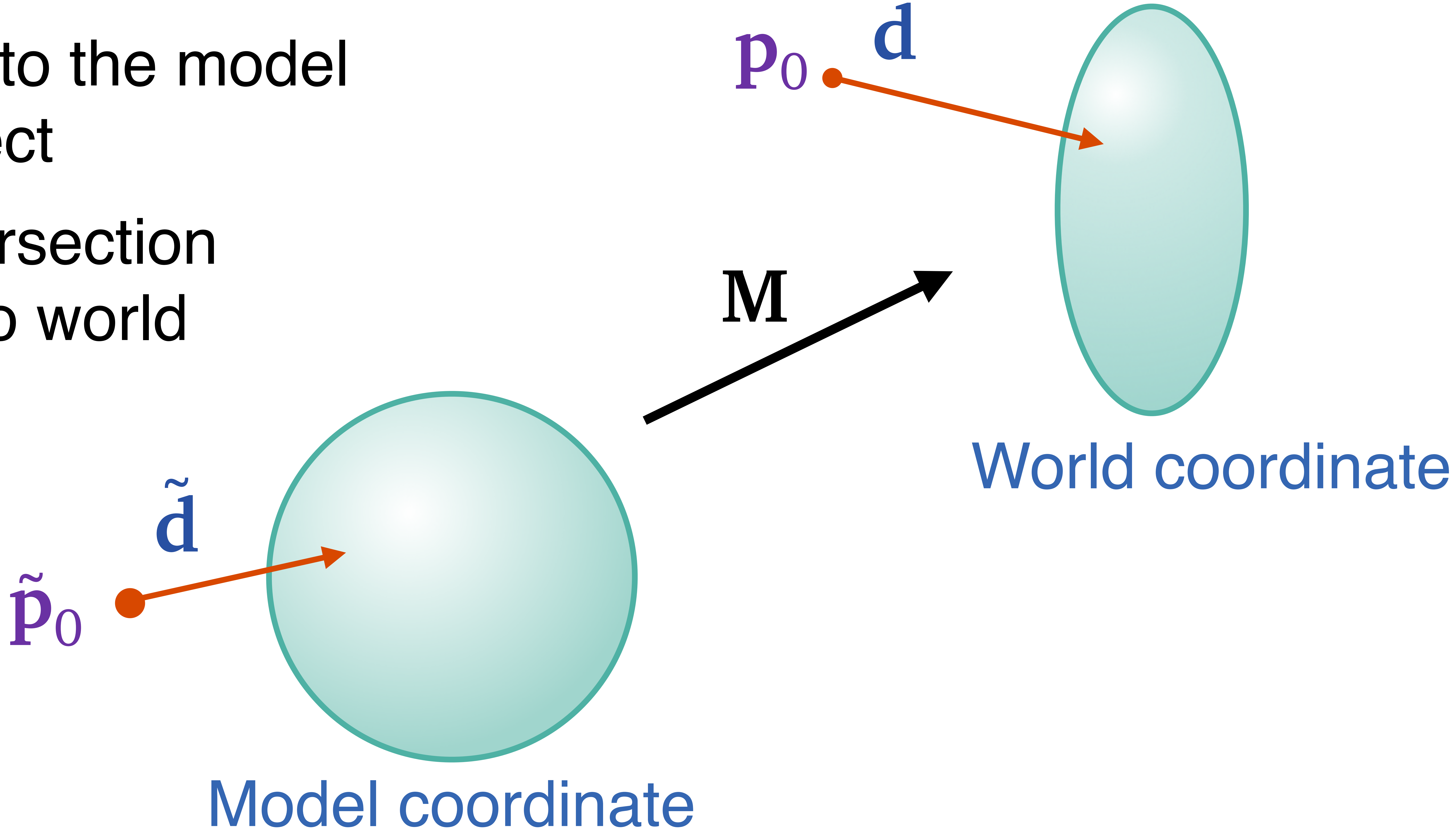
Transformed object

- Ellipsoid is a transformed sphere
  - We only need to talk about the transformation rule for ray-object intersection under change of coordinate



Transformed object

- Suppose the model coordinate and the world coordinate are related by a 4x4 model matrix $\mathbf{M}$
- Transform the ray to the model coordinate, intersect
- Transform the intersection information back to world



Transformed object

- Given a ray $(\mathbf{p}_0, \mathbf{d})$ in the world coordinate
- The ray in the model coordinate is computed by

$$\begin{bmatrix} | \\ \tilde{\mathbf{p}}_0 \\ | \\ 1 \end{bmatrix} = \mathbf{M}^{-1} \begin{bmatrix} | \\ \mathbf{p}_0 \\ | \\ 1 \end{bmatrix} \qquad \begin{bmatrix} | \\ \tilde{\mathbf{d}} \\ | \end{bmatrix} = \mathbf{A}^{-1} \begin{bmatrix} | \\ \mathbf{d} \\ | \end{bmatrix}$$

$$\tilde{\mathbf{d}} \leftarrow \text{normalize}(\tilde{\mathbf{d}})$$

where **A** = mat3(**M**), that is $\mathbf{M} = \begin{bmatrix} & & & * \\ & \mathbf{A} & & * \\ & & & * \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Transformed object

- Perform intersect(ray, obj) in the model coordinate
  - Obtain intersection position $\tilde{\mathbf{q}}$ and normal $\tilde{\mathbf{n}}$
- Transform the intersection position and normal back to the world

$$\begin{bmatrix} | \\ \mathbf{q} \\ | \\ 1 \end{bmatrix} = \mathbf{M} \begin{bmatrix} | \\ \tilde{\mathbf{q}} \\ | \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} | \\ \mathbf{n} \\ | \end{bmatrix} = \mathbf{A}^{-\intercal} \begin{bmatrix} | \\ \tilde{\mathbf{n}} \\ | \end{bmatrix}$$

$$\mathbf{n} \leftarrow \text{normalize}(\mathbf{n})$$

- Compute the rest of the intersection info in the world coordinate

$$t = |\mathbf{q} - \mathbf{p}_0|$$

Tips for handling Image and Scene

- Ray tracing framework
- Ray through pixel
- Ray-geometry intersection
- Organizing image and scene
- Global illumination

Image

```
void Raytrace(Camera cam, Scene scene, Image &image){
  int w = image.width; int h = image.height;
  for (int j=0; j<h; j++){
    for (int i=0; i<w; i++){
      Ray ray = RayThruPixel( cam, i, j, w, h );
      Intersection hit = Intersect( ray, scene );
      image.pixel[i][j] = FindColor( hit );
    }
  }
}
```

`[j*w + i]` if using linear array instead of multi-array

- **Image** is a list of pixels

```
class Image{
  public:
    int width, height;
    std::vector<glm::vec3> pixel;
    void initialize();
}
```

- Calling Raytrace will assign pixel values to the image

Image

- To show an image on screen, you can store it as a texture and transfer it to the frame buffer.

Image

- ***Global variables*** *(or encapsulated in your* `Image` *class)*

```
unsigned int fbo; // frame buffer object
unsigned int texture; // texture buffer object
```

- ***Initialize buffers*** *(e.g. in initialization of* `Image` *class)*

```
glGenFrameBuffers(1,&fbo);
glGenTextures(1,&texture);
```

- ***Display*** *(e.g. in a "draw" method of* `Image` *class)*

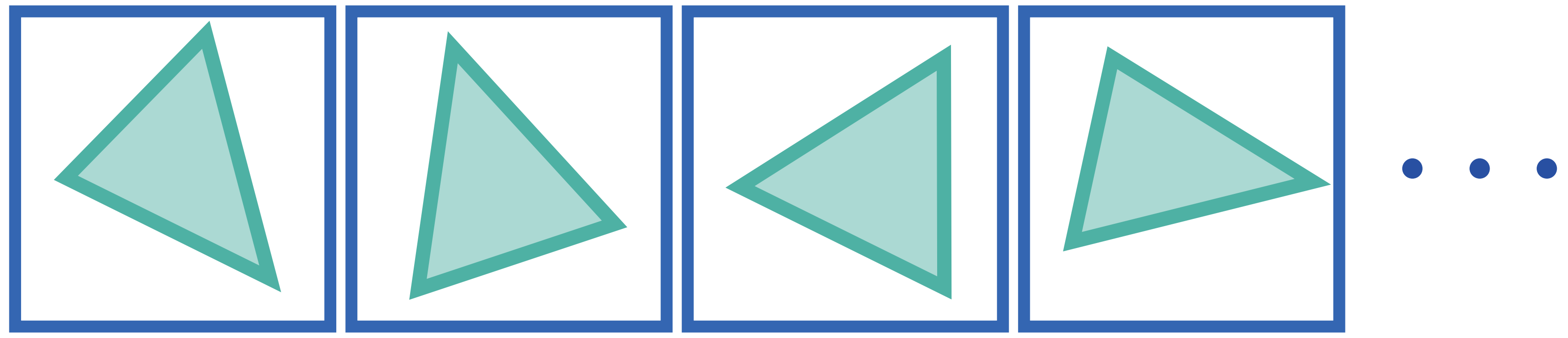
```
glBindTexture(GL_TEXTURE_2D, texture);
glTexImage2D(GL_TEXTURE_2D,0,GL_RGB,widht,height, // load texture
             0,GL_RGB,GL_FLOAT, &pixel[0][0]);    // with image data

glBindFramebuffer(GL_READ_FRAMEBUFFER, fbo);
glFramebufferTexture2D(GL_READ_FRAMEBUFFER,        // attach texture
  GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, texture, 0);// and frame

glBlitFramebuffer(0,0,width,height, 0,0,width,height, // copy data from
  GL_COLOR_BUFFER_BIT, GL_NEAREST);    // the read-buffer to draw-buffer
```

Scene

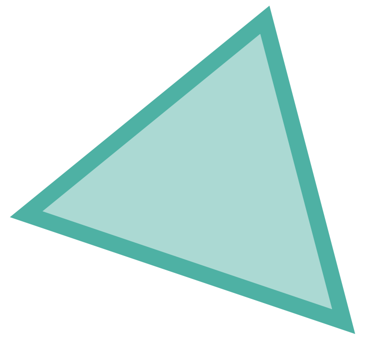
```
void Raytrace(Camera cam, Scene scene, Image &image){
  int w = image.width; int h = image.height;
  for (int j=0; j<h; j++){
    for (int i=0; i<w; i++){
      Ray ray = RayThruPixel( cam, i, j, w, h );
      Intersection hit = Intersect( ray, scene );
      image.pixel[i][j] = FindColor( hit );
    }
  }
}
```

- **Scene** contains a list (or some data structure) of triangles (or other geometric primitives)

```
std::vector<Triangle> triangle_soup;
```



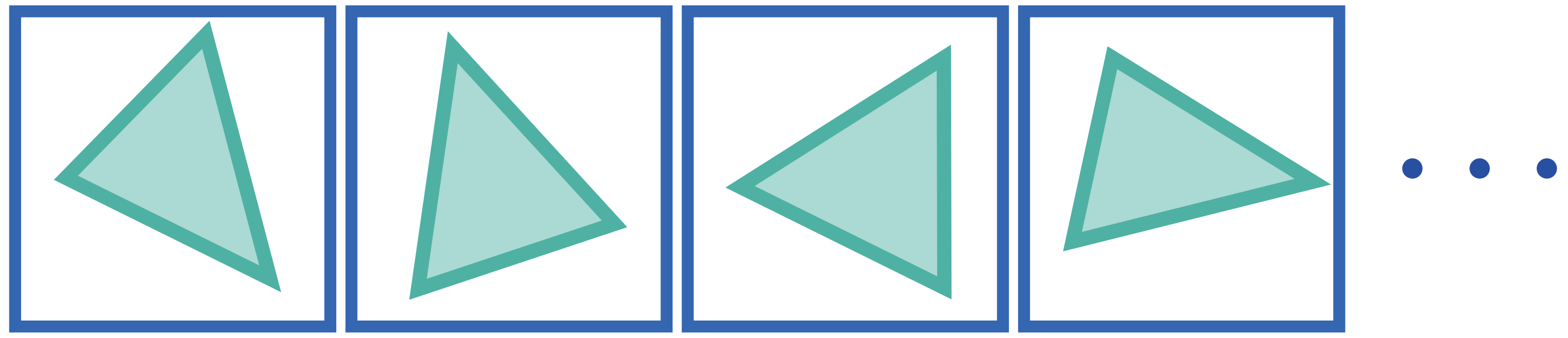
class Triangle{
3 vertex positions, 3 vertex normals, pointer to material }



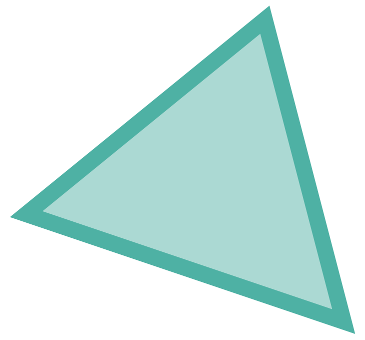
Scene

- **Scene** contains a list (or some data structure) of triangles (or other geometric primitives)

std::vector<Triangle> triangle_soup;



class Triangle{
3 vertex positions, 3 vertex normals, pointer to material }



- When searching for intersection in "`Intersect(ray, scene)`" we can iterate `triangle` over `scene.triangle_soup`

- We can still build complex scene like in HW3

Scene

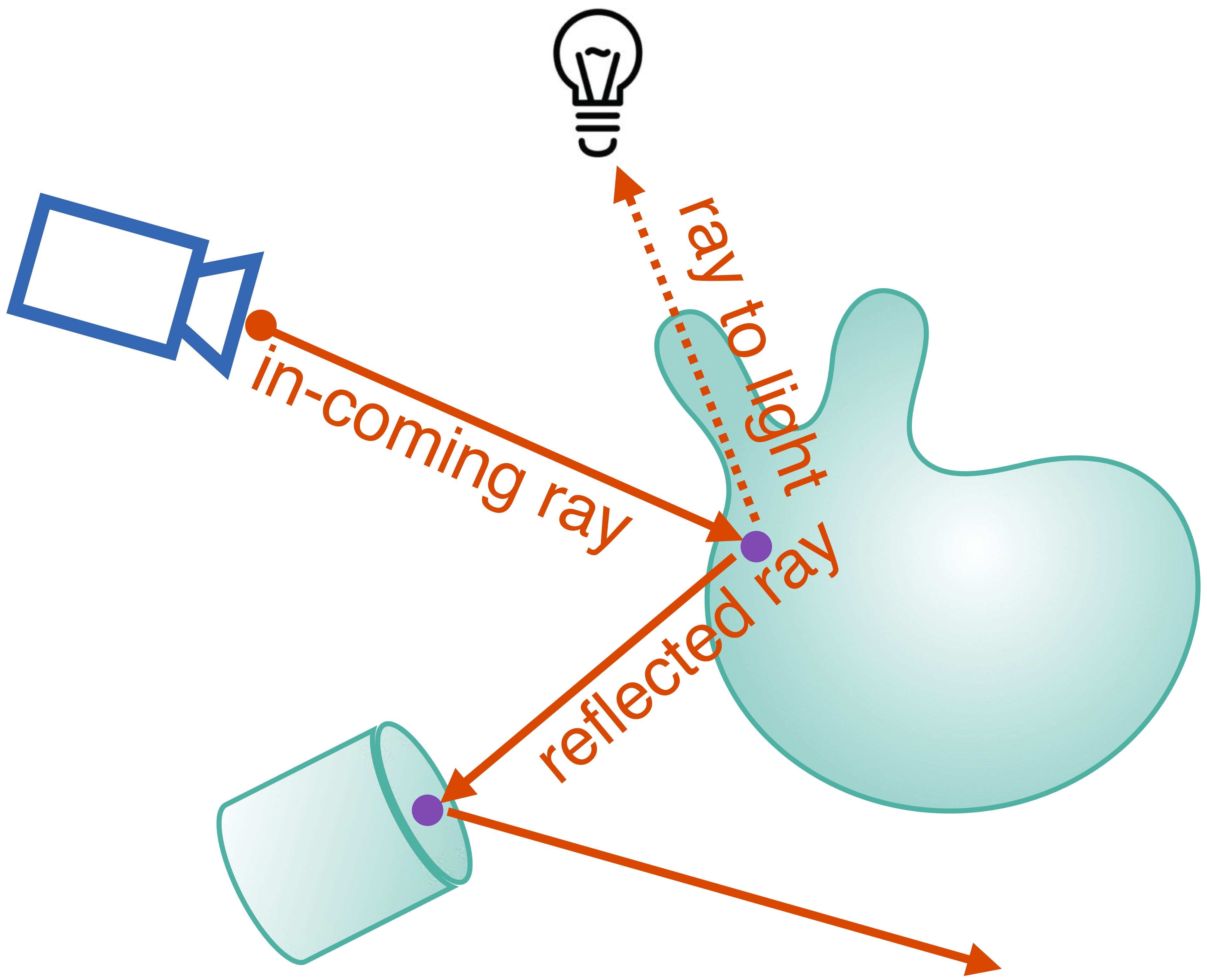
- **Scene** contains a list (or some data structure) of triangles (or other geometric primitives)
- Re-use HW3 scene graph description. During depth first search, instead of calling “draw” model, just dump all triangles into a list (with position/normal transformed to the world coordinate)

Global Illumination

- Ray tracing framework
- Ray through pixel
- Ray-geometry intersection
- Organizing image and scene
- Global illumination

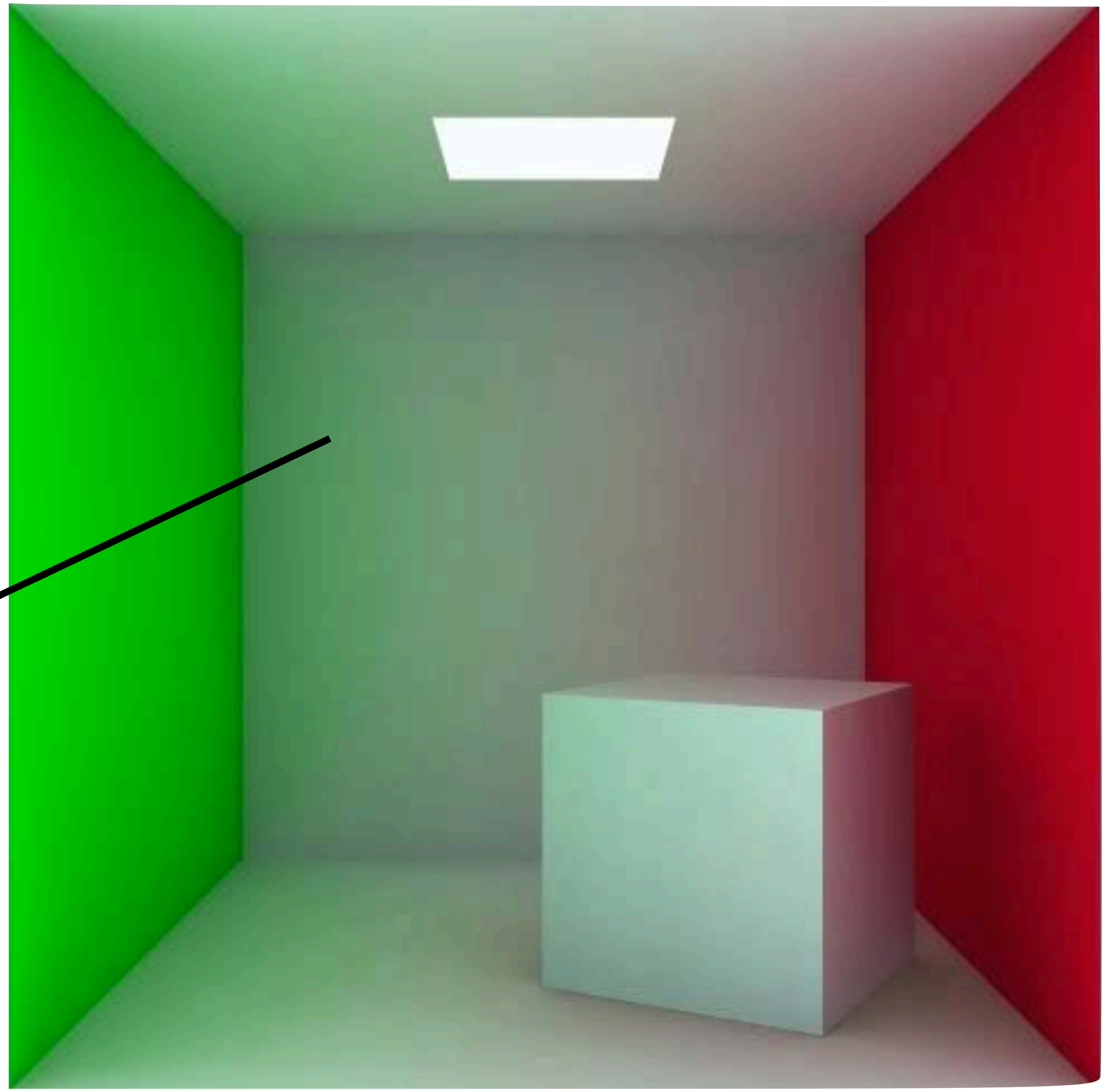
Global illumination

- Local illumination evaluates color directly using light source.
- We have seen a glimpse of global illumination.
    - Visibility test from light source
    - Recursive mirror reflection
- In a more realistic global illumination, the diffuse color is also recursive!



Global illumination

- In a more realistic global illumination, the diffuse color is also recursive!



indirect lighting effect

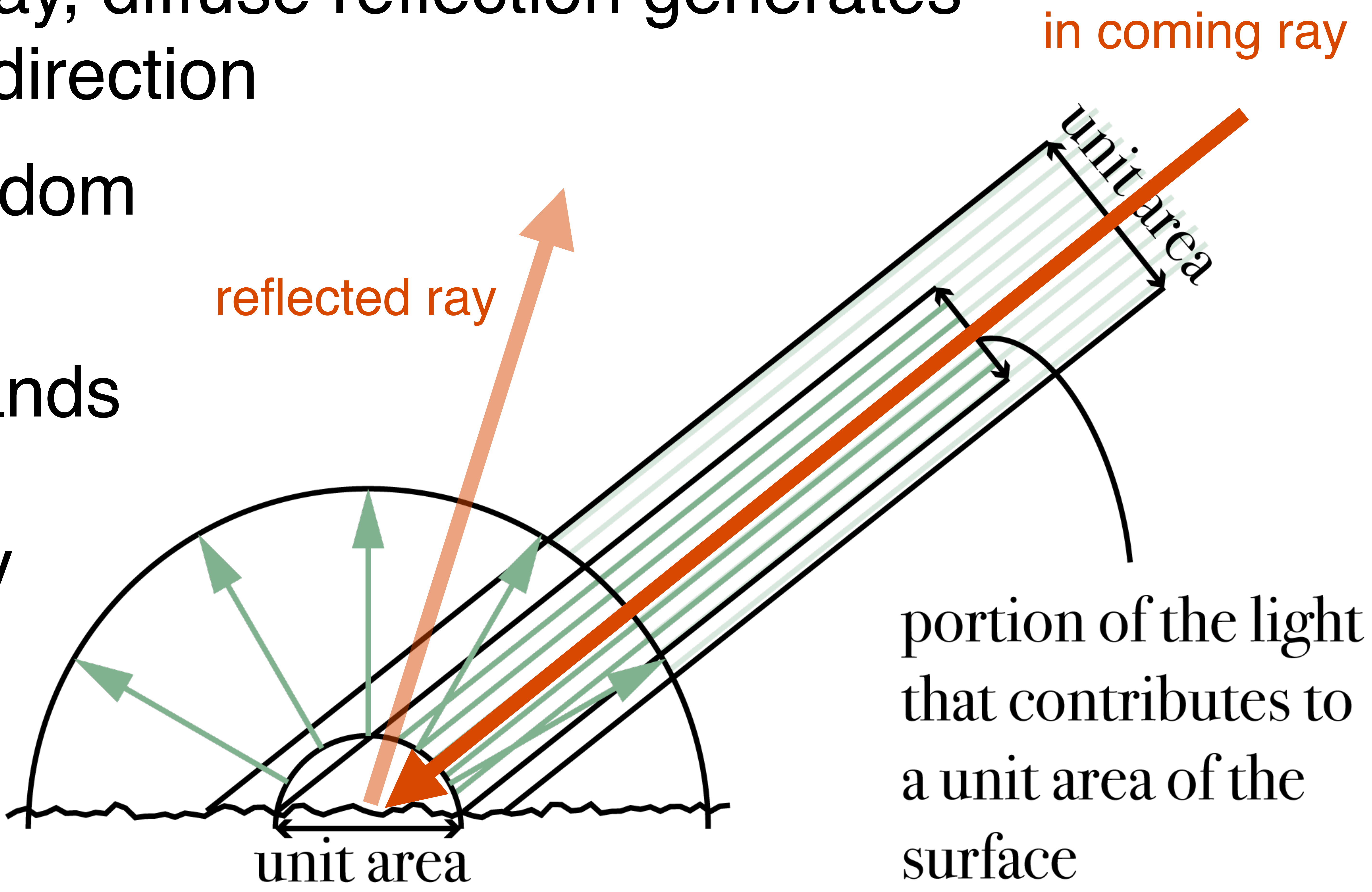
Global illumination

- Local illumination is also called direct lighting.
- We add indirect lighting, which are paths that have more bounces.

Direct lighting + Indirect lighting =

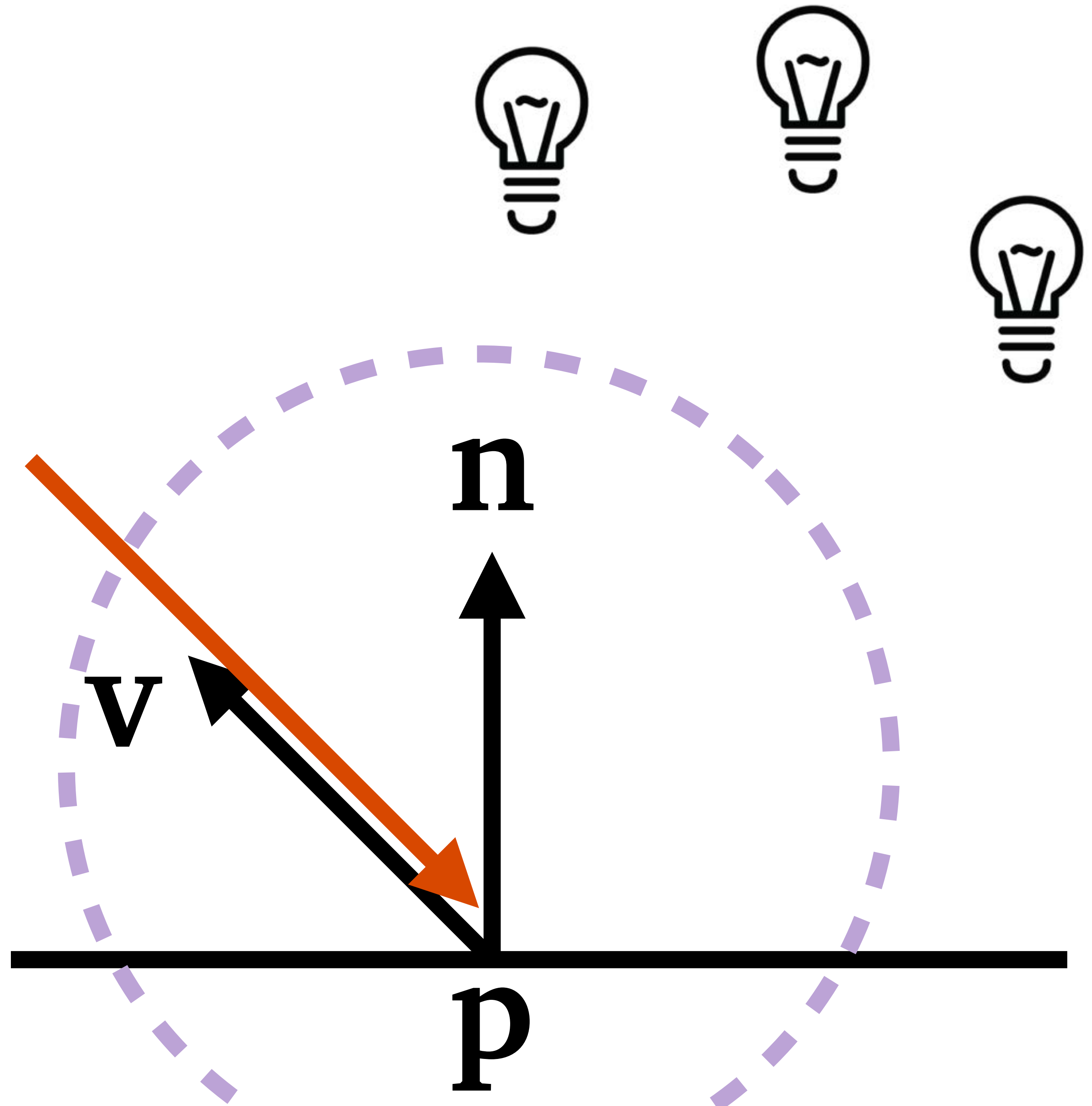
Diffuse light

- To make both diffuse and specular reflection recursive, evaluate color by the color of the reflected ray
- Instead of mirror reflecting ray, diffuse reflection generates a reflected ray in a random direction
  - The color shaded by a random reflection won't look right
  - But after averaging thousands of random samples, the resulting color is physically accurate.



Shading model (from direct to recursive)

- Let us recall OpenGL shading model
- Given a ray-object intersection “hit” (or fragment)
  - **v**: direction to the source of in coming ray
  - **n**: surface normal
  - **p**: position of this hit
  - Material color $\mathbf{C}_{\text{diffuse}}$ $\mathbf{C}_{\text{specular}}$
- Output light color $\mathbf{L}_{\text{seen}}$ seen by in-coming ray

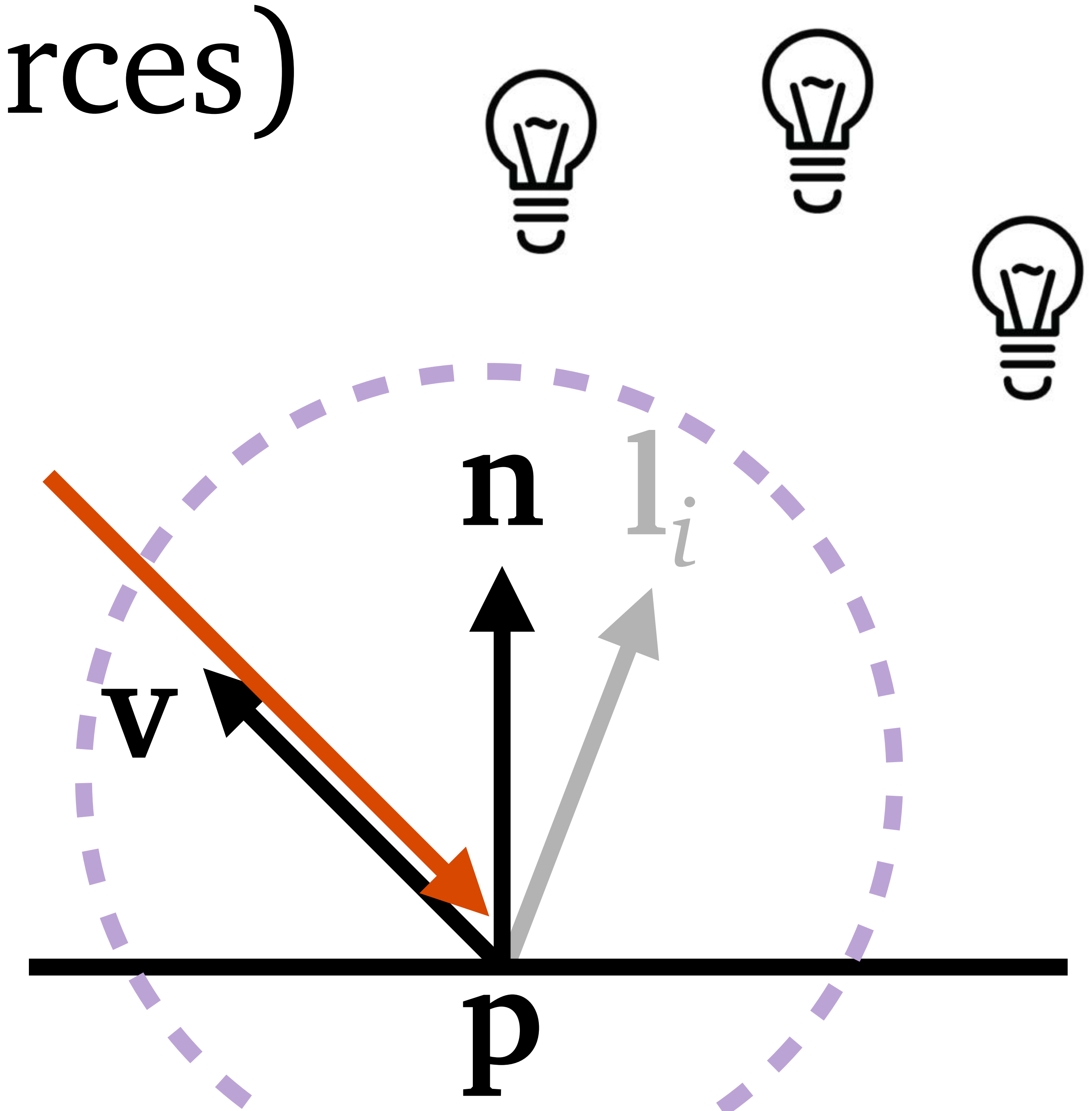


Shading model (from direct to recursive)

- Direct shading model we did in OpenGL

$$\mathbf{L}_{\text{seen}} = \sum_{i \in \text{lights}} \mathbf{C}_{\text{diffuse}} \mathbf{L}_{\substack{\text{light} \\ \text{source}_i}} \max(\mathbf{n} \cdot \mathbf{l}_i, 0)$$

$$+ \mathbf{C}_{\text{specular}} \text{BlinnPhong}(\mathbf{v}, \mathbf{n}, \text{LightSources})$$



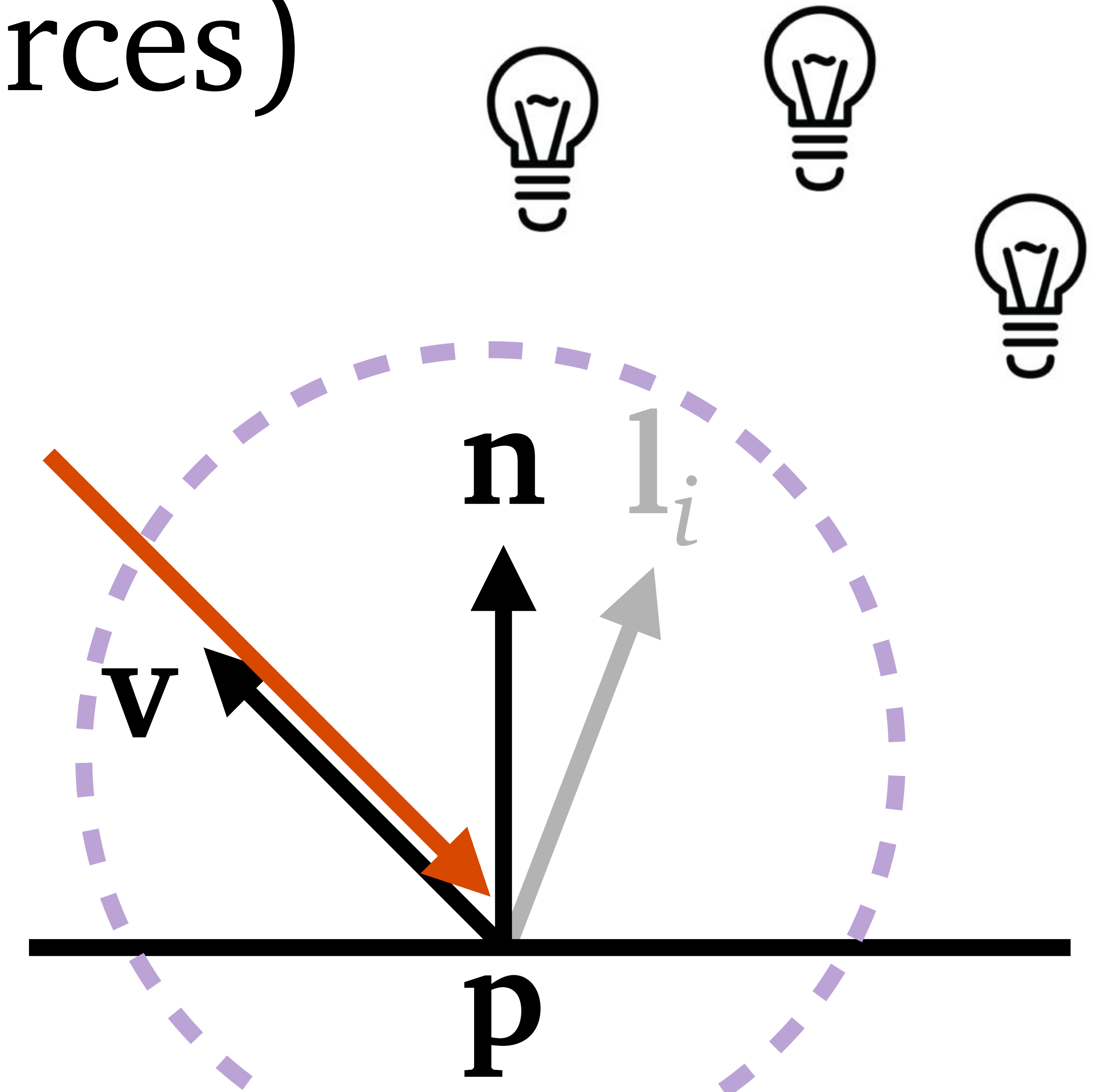
Shading model (from direct to recursive)

- Add shadow in ray tracing

0 or 1 depending whether ray to i-th light is blocked

$$\mathbf{L}_{\text{seen}} = \sum_{i \in \text{lights}} \mathbf{C}_{\text{diffuse}} \mathbf{L}_{\text{light source}_i} \max(\mathbf{n} \cdot \mathbf{l}_i, 0) \boxed{\text{visibility}_i}$$

$$+ \mathbf{C}_{\text{specular}} \text{BlinnPhong}(\mathbf{v}, \mathbf{n}, \text{LightSources})$$

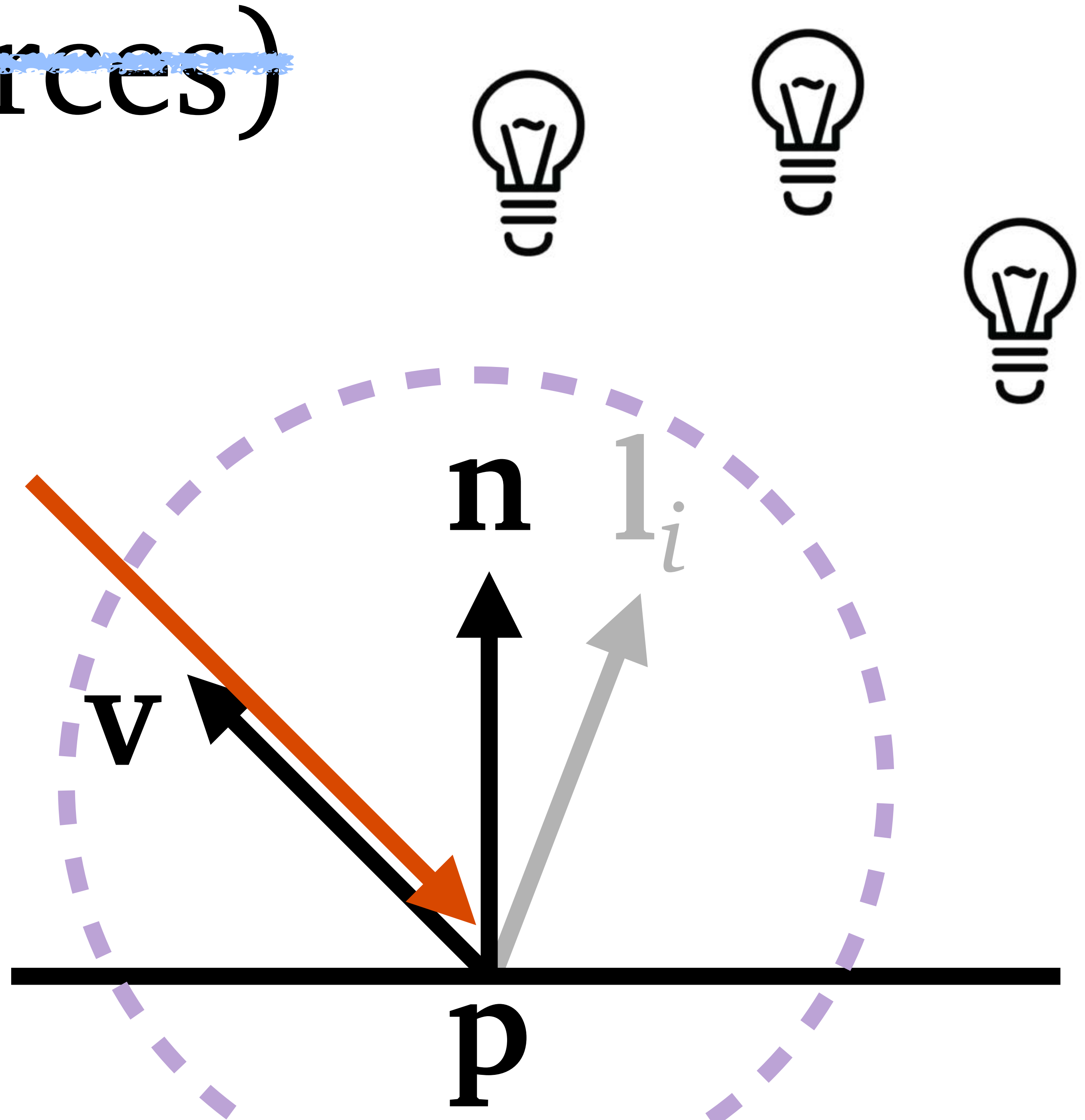


Shading model (from direct to recursive)

- Add recursive specular reflection

$$\mathbf{L}_{\text{seen}} = \sum_{i \in \text{lights}} \mathbf{C}_{\text{diffuse}} \mathbf{L}_{\substack{\text{light} \\ \text{source}_i}} \max(\mathbf{n} \cdot \mathbf{l}_i, 0) \ \text{visibility}_i$$

$$+ \mathbf{C}_{\text{specular}} \text{BlinnPhong}(\mathbf{v}, \mathbf{n}, \text{LightSources})$$



Shading model (from direct to recursive)

- Add recursive specular reflection

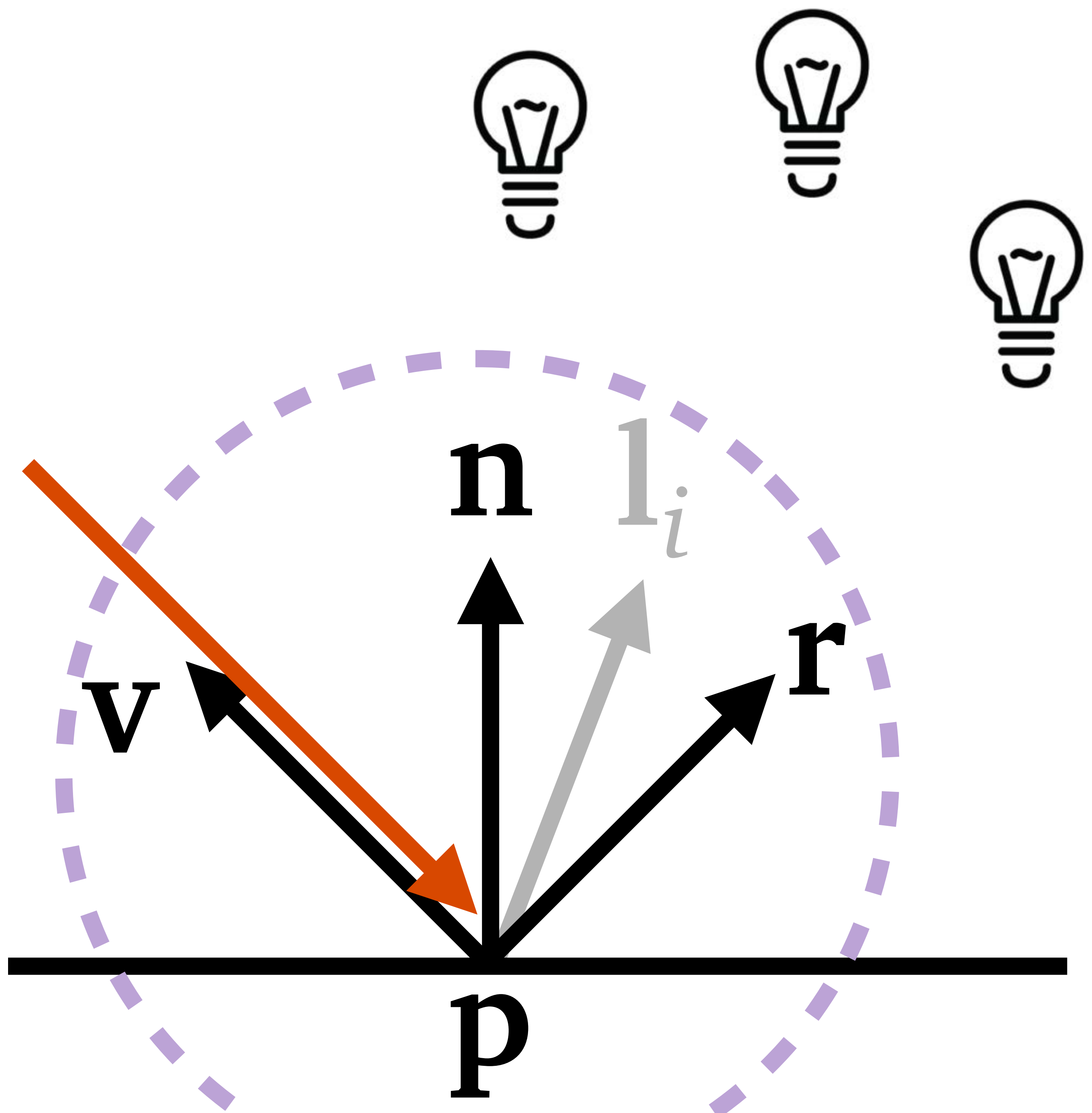
$$\mathbf{L}_{\text{seen}} = \sum_{i \in \text{lights}} \mathbf{C}_{\text{diffuse}} \mathbf{L}_{\substack{\text{light}\\ \text{source}_i}} \max(\mathbf{n} \cdot \mathbf{l}_i, 0) \ \text{visibility}_i$$

$$+ \mathbf{C}_{\text{specular}} \mathbf{L}_{(\mathbf{p},\mathbf{r})}$$

color seen by ray ($\mathbf{p}$, $\mathbf{r}$)

- mirror reflection direction

$$\mathbf{r} = 2(\mathbf{n} \cdot \mathbf{v})\mathbf{n} - \mathbf{v}$$



Shading model (from direct to recursive)

- As for adding recursive diffuse shading

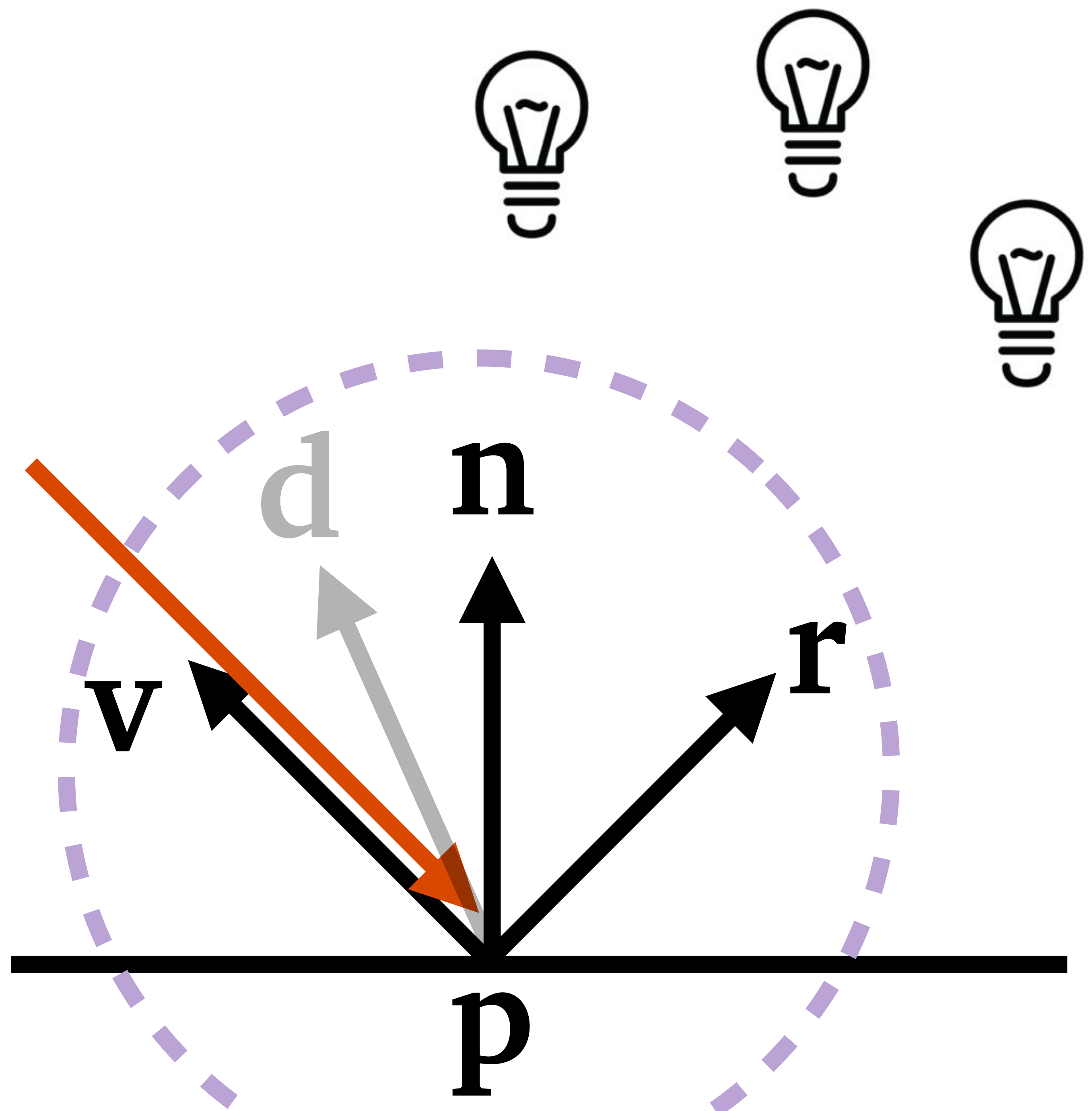
$$\mathbf{L}_{\text{seen}} = \sum_{i \in \text{lights}} \mathbf{C}_{\text{diffuse}} \mathbf{L}_{\text{light source}_i} \max(\mathbf{n} \cdot \mathbf{l}_i, 0) \ \text{visibility}_i$$

$$+ \mathbf{C}_{\text{specular}} \mathbf{L}_{(\mathbf{p},\mathbf{r})}$$

replace by

$$\mathbf{C}_{\text{diffuse}} \mathbf{L}_{(\mathbf{p},\mathbf{d})} (\mathbf{n} \cdot \mathbf{d})$$

where $\mathbf{d}$ is a random direction uniformly distributed on the hemisphere



Shading model (from direct to recursive)

- As for adding recursive diffuse shading

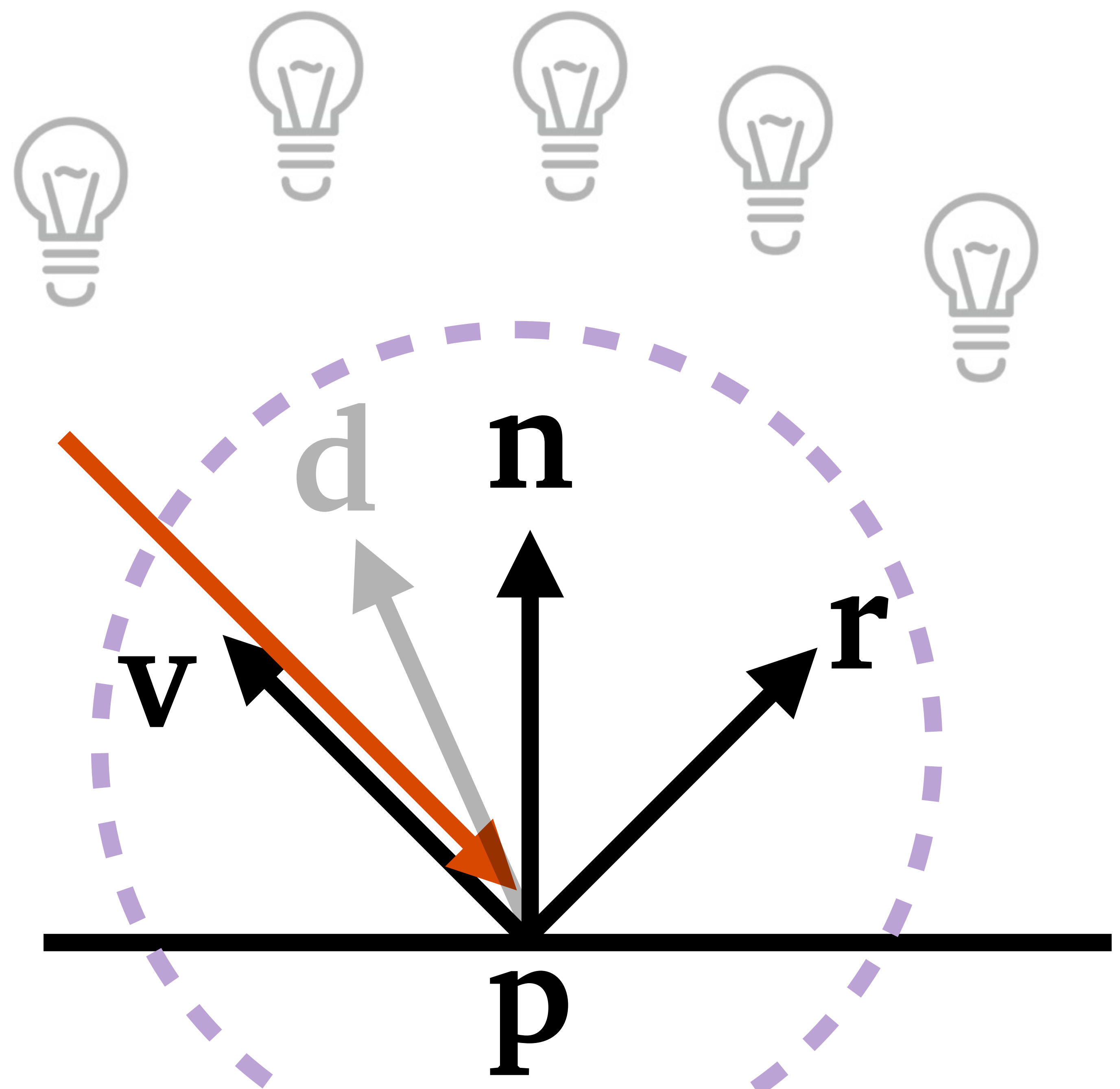
$$\mathbf{L}_{\text{seen}} = \boxed{\sum_{i \in \text{lights}} \mathbf{C}_{\text{diffuse}} \mathbf{L}_{\text{light source}_i} \max(\mathbf{n} \cdot \mathbf{l}_i, 0)\ \text{visibility}_i}$$

$$+ \mathbf{C}_{\text{specular}} \mathbf{L}_{(\mathbf{p},\mathbf{r})}$$

replace by

$$\mathbf{C}_{\text{diffuse}} \mathbf{L}_{(\mathbf{p},\mathbf{d})} (\mathbf{n} \cdot \mathbf{d})$$

- This is like thinking of every direction is a light source



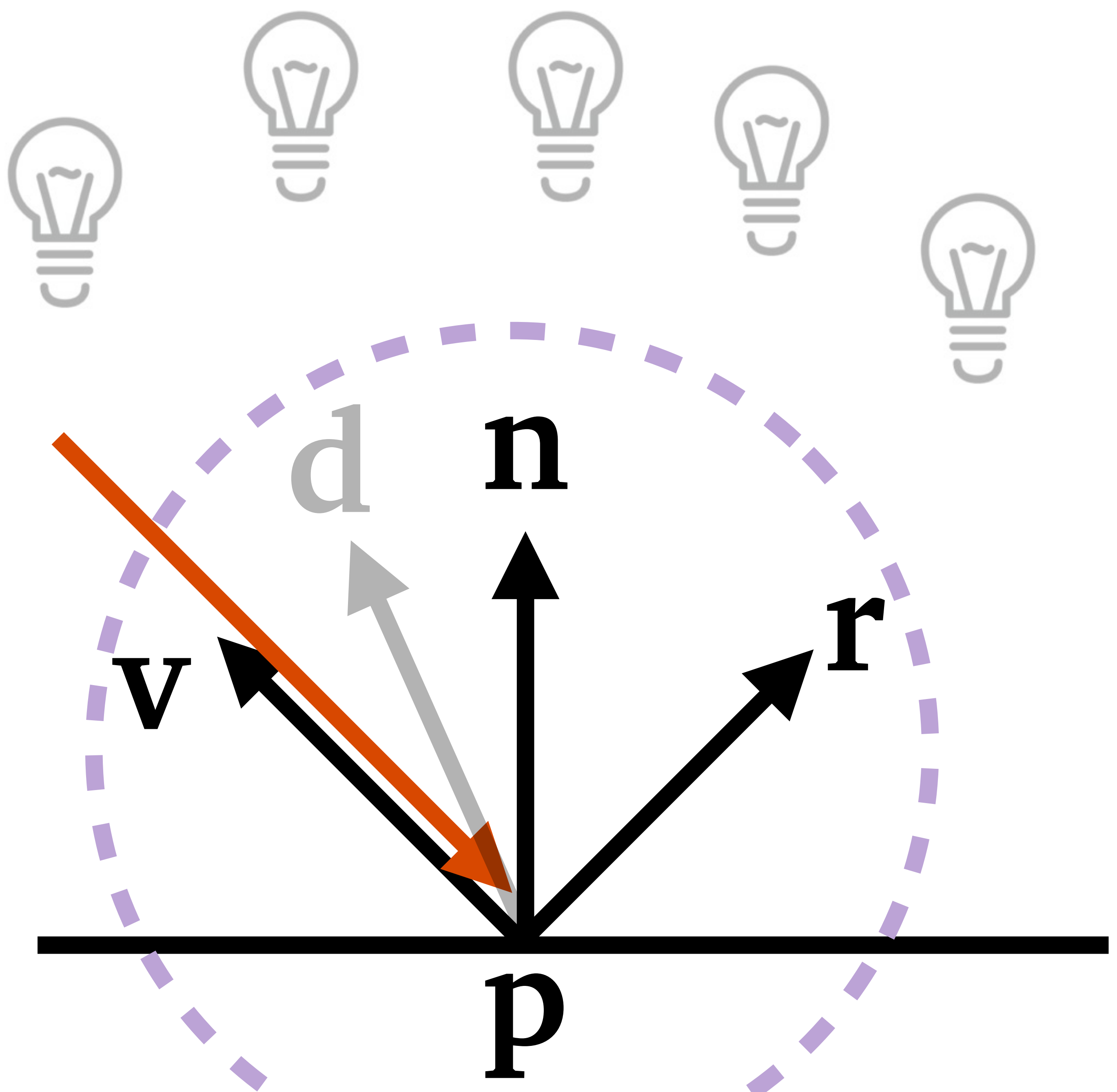
Shading model (from direct to recursive)

- As for adding recursive diffuse shading

$$\mathbf{L}_{\text{seen}} = \mathbf{C}_{\text{diffuse}} \mathbf{L}_{(\mathbf{p},\mathbf{d})} (\mathbf{n} \cdot \mathbf{d})$$

$$+ \mathbf{C}_{\text{specular}} \mathbf{L}_{(\mathbf{p},\mathbf{r})}$$

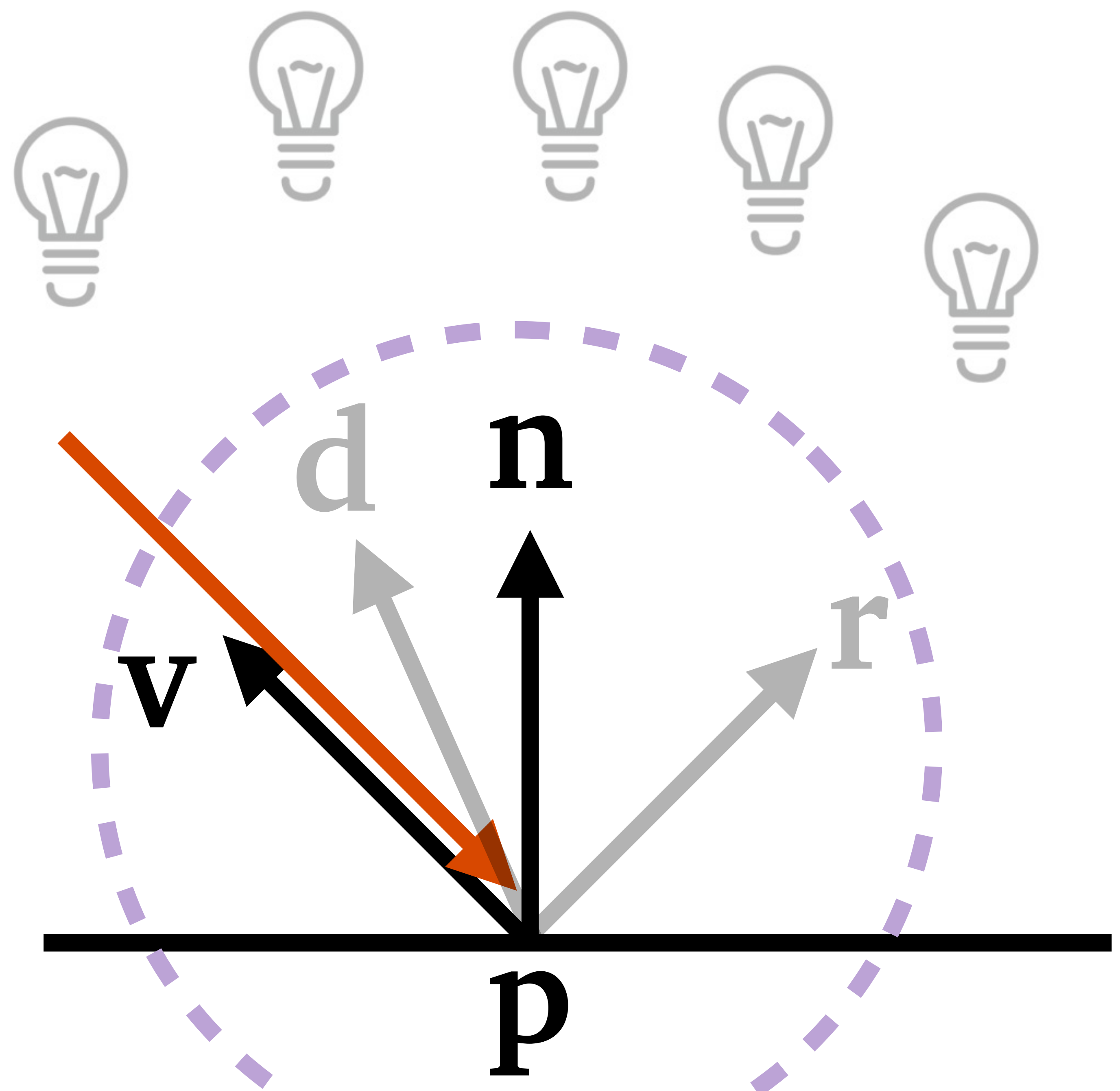
- Problem: evaluating this color require generating two additional rays.
  - The number of rays in the recursion will grow exponentially $O(2^n)$
  - Solution: just combine the two terms



Shading model (from direct to recursive)

- Final shading model

$$\mathbf{L}_{\text{seen}} = \begin{cases} \mathbf{C}_{\text{diffuse}} \mathbf{L}_{(\mathbf{p},\mathbf{d})} (\mathbf{n} \cdot \mathbf{d}) & \text{with probability } 0.5 \\ \quad \text{or} & \\ \mathbf{C}_{\text{specular}} \mathbf{L}_{(\mathbf{p},\mathbf{r})} & \text{with probability } 0.5 \end{cases}$$



Shading model (from direct to recursive)

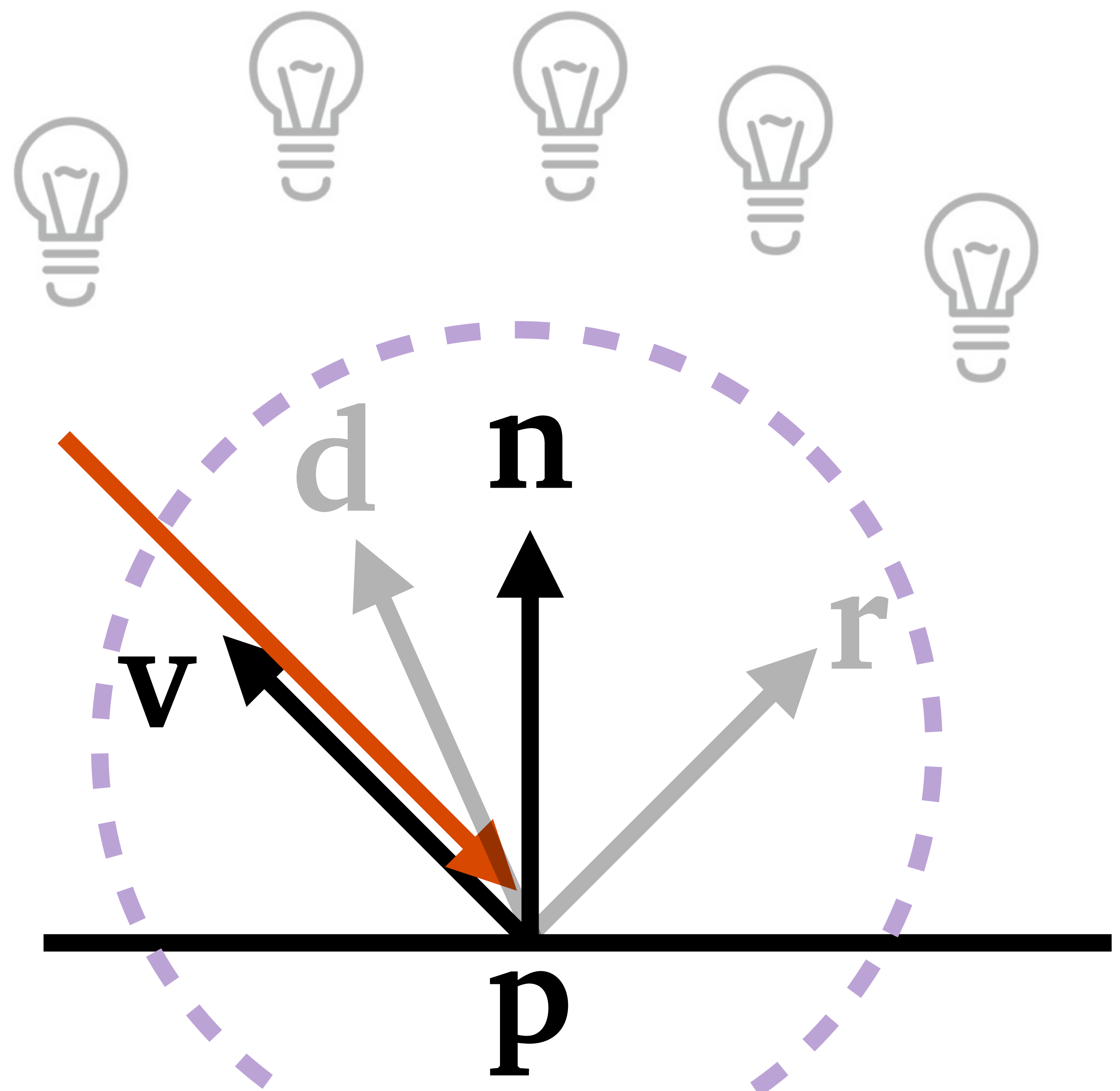
- Final shading model

$$\mathbf{L}_{\text{seen}} = \begin{cases} \mathbf{C}_{\text{diffuse}}\mathbf{L}_{(\mathbf{p},\mathbf{d})}(\mathbf{n}\cdot\mathbf{d}) & \text{with probability 0.5} \\ \quad \text{or} & \\ \mathbf{C}_{\text{specular}}\mathbf{L}_{(\mathbf{p},\mathbf{r})} & \text{with probability 0.5} \end{cases}$$

- Most general shading model

$$\mathbf{L}_{\text{seen}} = \mathbf{C}_{\text{BRDF}}(\mathbf{v},\mathbf{d})\mathbf{L}_{(\mathbf{p},\mathbf{d})}$$

  - BRDF: Bidirectional reflectance distribution function
  - What one would do in CSE168 (advanced rendering)



Averaging the randomized color

For k = 1,…,N (number of samples)

- Shoot a ray through a *random* point in the pixel
- Hit some surface and evaluate the color of the hit:

$$\mathbf{L}_{\text{seen}} = \begin{cases} \mathbf{C}_{\text{diffuse}} \mathbf{L}_{(\mathbf{p},\mathbf{d})} (\mathbf{n} \cdot \mathbf{d}) & \text{with probability 0.5} \\ \text{or} & \\ \mathbf{C}_{\text{specular}} \mathbf{L}_{(\mathbf{p},\mathbf{r})} & \text{with probability 0.5} \end{cases}$$

- Let the recursion unfold with a max recursion depth.
- If the max depth is reached, set the color as old-school diffuse

$$\sum_{i \in \text{lights}} \mathbf{C}_{\text{diffuse}} \mathbf{L}_{\substack{\text{light} \\ \text{source}_i}} \max(\mathbf{n} \cdot \mathbf{l}_i, 0) \ \text{visibility}_i$$

- Accumulate $\mathbf{L}_{\text{cum}} += \mathbf{L}_{\text{seen}}$

EndFor

$$\mathbf{L}_{\text{pixel}} = \frac{1}{N} \mathbf{L}_{\text{cum}}$$

Averaging the randomized color

1 sample path per pixel | 10 sample paths per pixel | 1000 sample paths per pixel

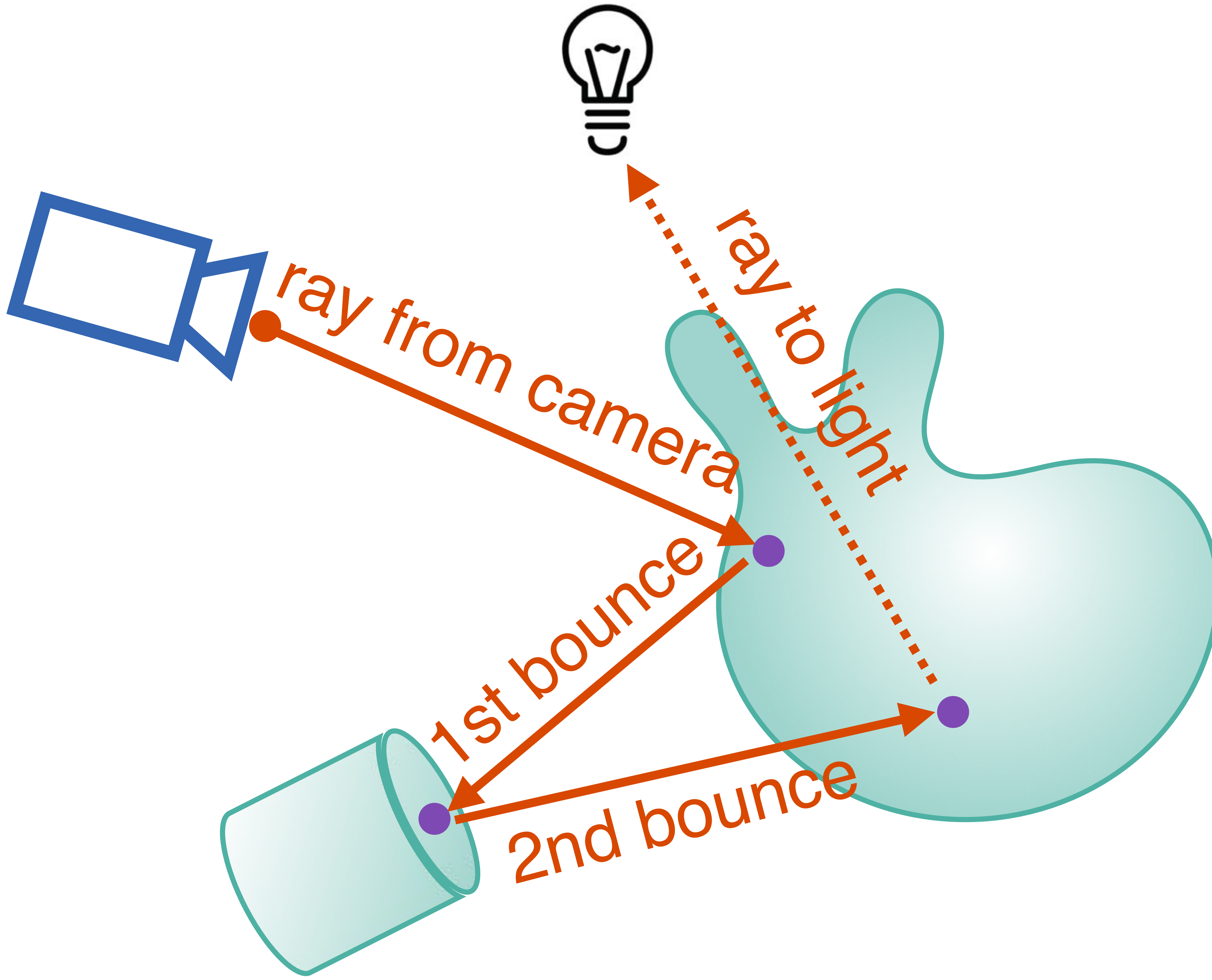
Path lengths (recursion depth)

- The recursion depth is also the number of bounces of ray
- If we just set a fixed recursion depth, the result will be too dark

Disney Big Hero 6 (2014)

1 bounce + 2 bounce

1 bounces + 2 bounces
+ … + 9 bounces



Path lengths (recursion depth)

- The recursion depth is also the number of bounces of ray
- If we just set a fixed recursion depth, the result will be too dark

Disney Big Hero 6 (2014)

1 bounce + 2 bounce

1 bounces + 2 bounces + … + 9 bounces

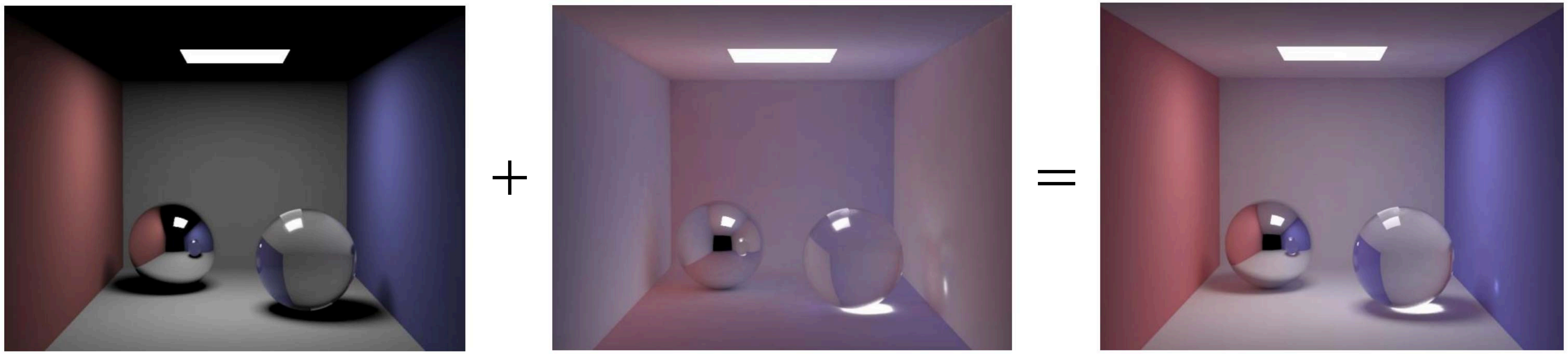
A lot of bounces

Infinite sum

- The color of pixel should be

  [Color of 1-bounce paths] + [Color of 2-bounce paths] + . . . + [Color of L-bounce paths] + . . .

  (how do you compute infinite sum?)



Direct lighting

Indirect lighting

A method for infinite sum

- The color of pixel should be

  [Color of 1-bounce paths] + [Color of 2-bounce paths] + . . . + [Color of L-bounce paths] + . . .

- The method of Russian Roulette:
  - Let the ray bounce indefinitely until randomly terminated
  - Every bounce has a termination probability *p*
  - The probability of getting a k-bounce paths is $(1-p)^k p$
  - If we get a k-bounce path, weight the result by $\frac{1}{(1-p)^k p}$
  - Expectation: $\sum_{k=1}^{\infty} \frac{\text{result with } k \text{ bounces}}{\cancel{(1-p)^k p}} \cdot \cancel{(1-p)^k p}$

probability of getting k bounces

Next

- Radiosity
- Rendering equation