Computer Graphics

Raycasting. Raytracing. Raymarching.

Konstantin Tretyakov
kt@ut.ee

Nov 13, 2013

Next

- Raycasting
- Raytracing
- Raymarching / Sphere tracing

- Rendering equation solvers
  - Radiosity, Photon mapping, Path tracing

The “Emission theory”

- Ancient greeks (up to Aristotle and even later) believed that you see by using beams emanating from your eyes.

- “Winer et al. (2002) have found recent evidence that as many as 50% of American college students believe in emission theory.[3]”

http://en.wikipedia.org/wiki/Emission_theory_(vision)

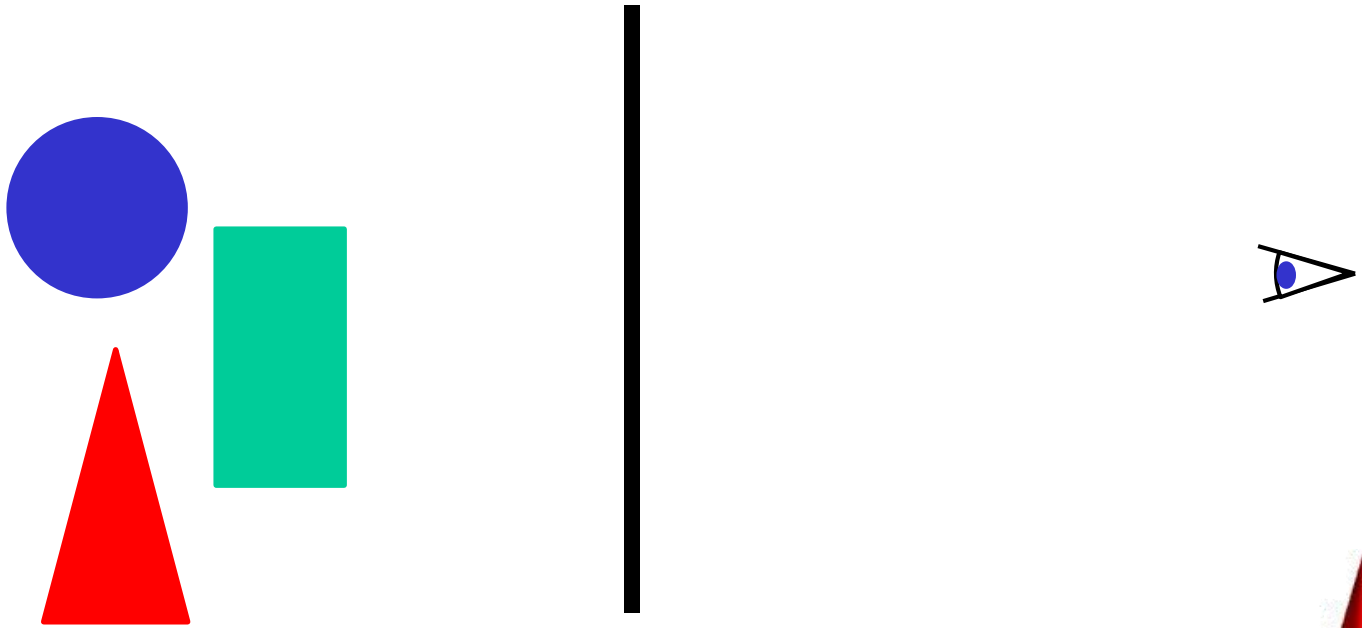
The “Emission theory”

- Mathematically, the two theories are equivalent in many respects, because light propagation rules are mostly direction-independent.

- Consequently, we can use the “Emission” approach in rendering. It is called raycasting.

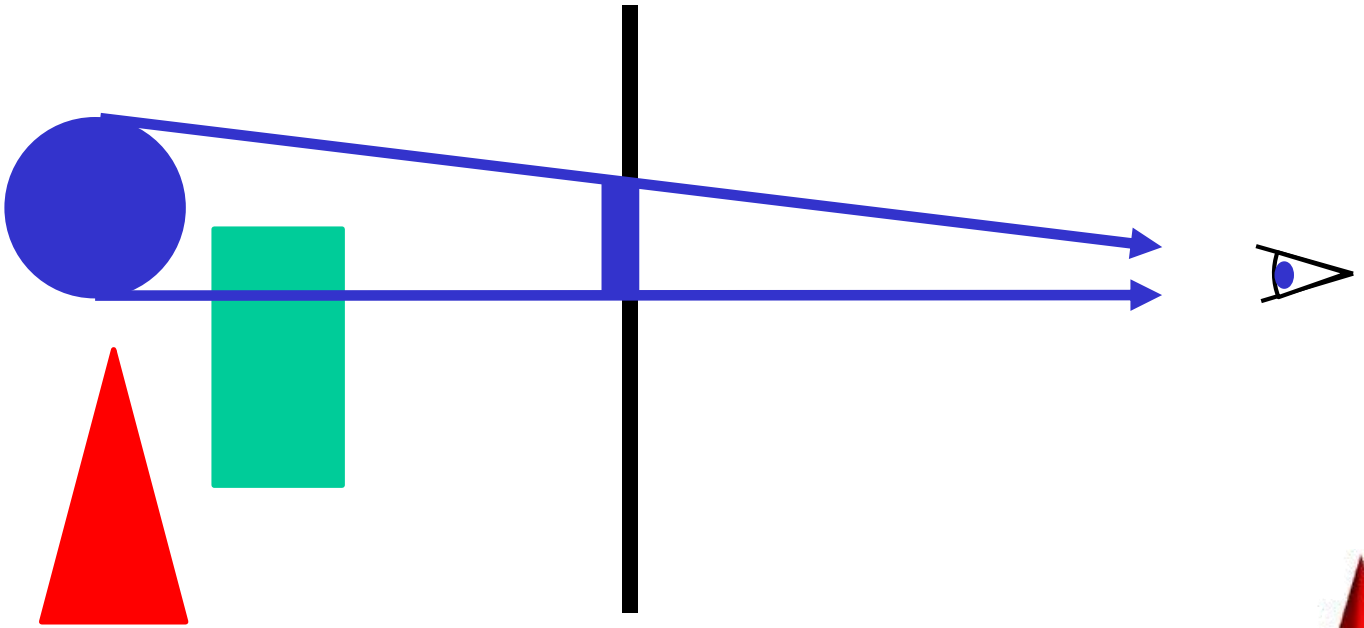
Graphics Pipeline vs Raycasting

- **Graphics pipeline:**
  - For each object, find its location on screen (and then color each pixel appropriately)



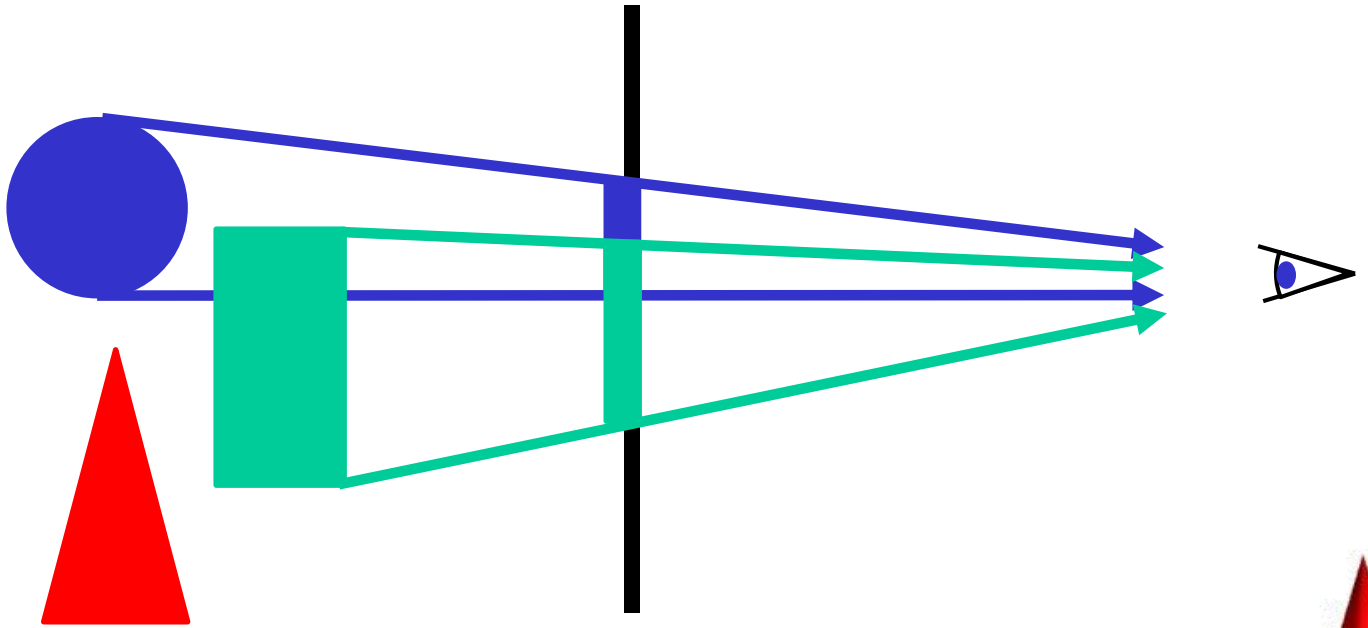
Graphics Pipeline vs Raycasting

- **Graphics pipeline:**
  - For each object, find its location on screen (and then color each pixel appropriately)



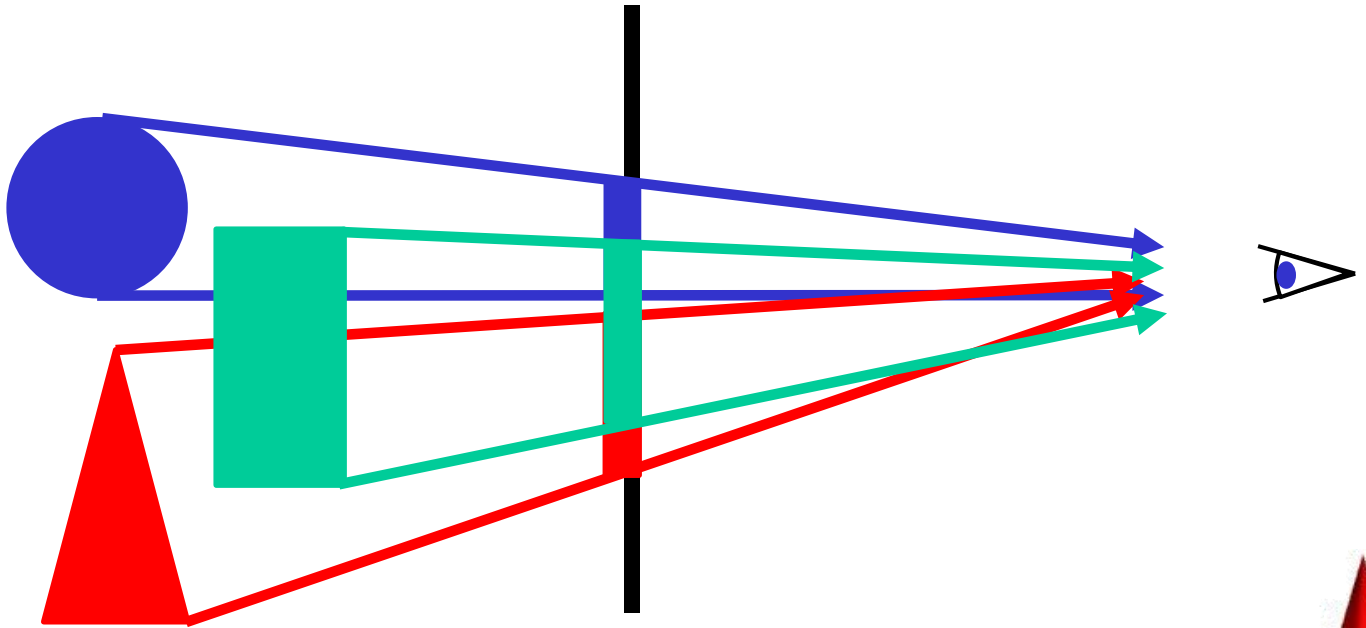
Graphics Pipeline vs Raycasting

- **Graphics pipeline:**
  - For each object, find its location on screen (and then color each pixel appropriately)



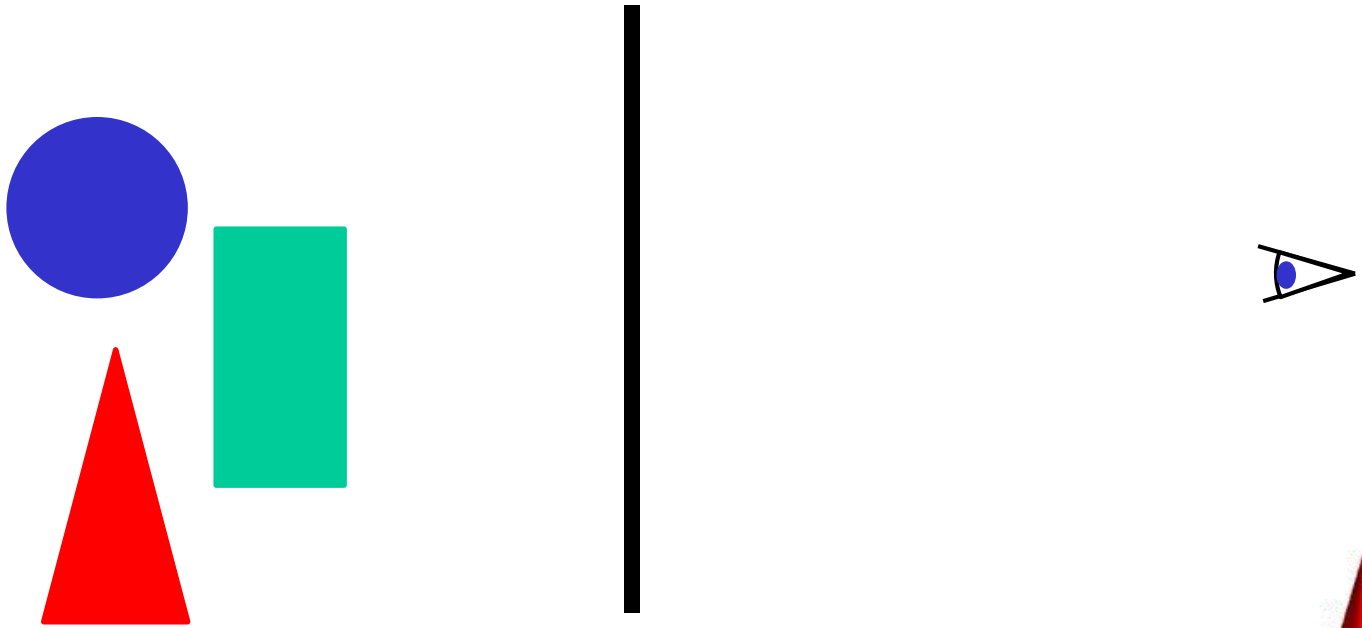
Graphics Pipeline vs Raycasting

- **Graphics pipeline:**
  - For each object, find its location on screen (and then color each pixel appropriately)



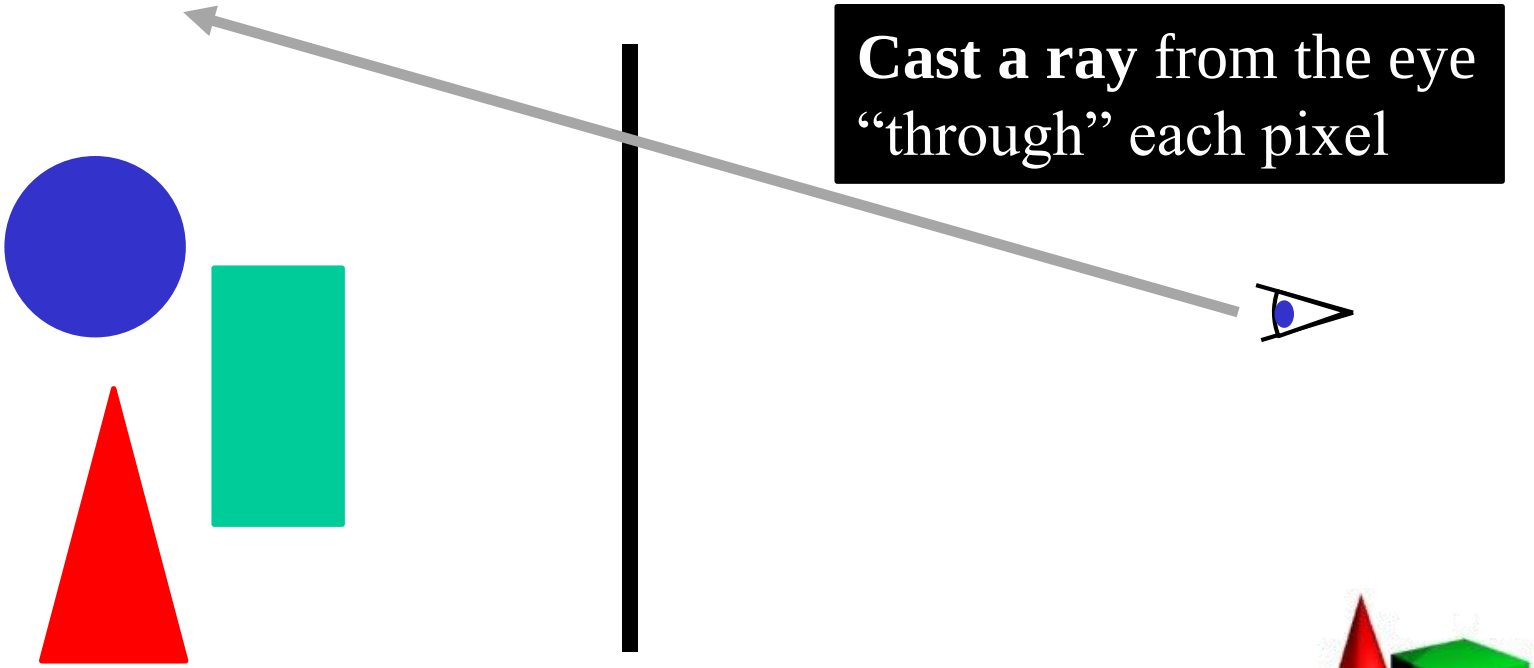
Graphics Pipeline vs Raycasting

- **Raycasting:**
  - For each **pixel**, find the corresponding **object** (and then color each pixel appropriately)



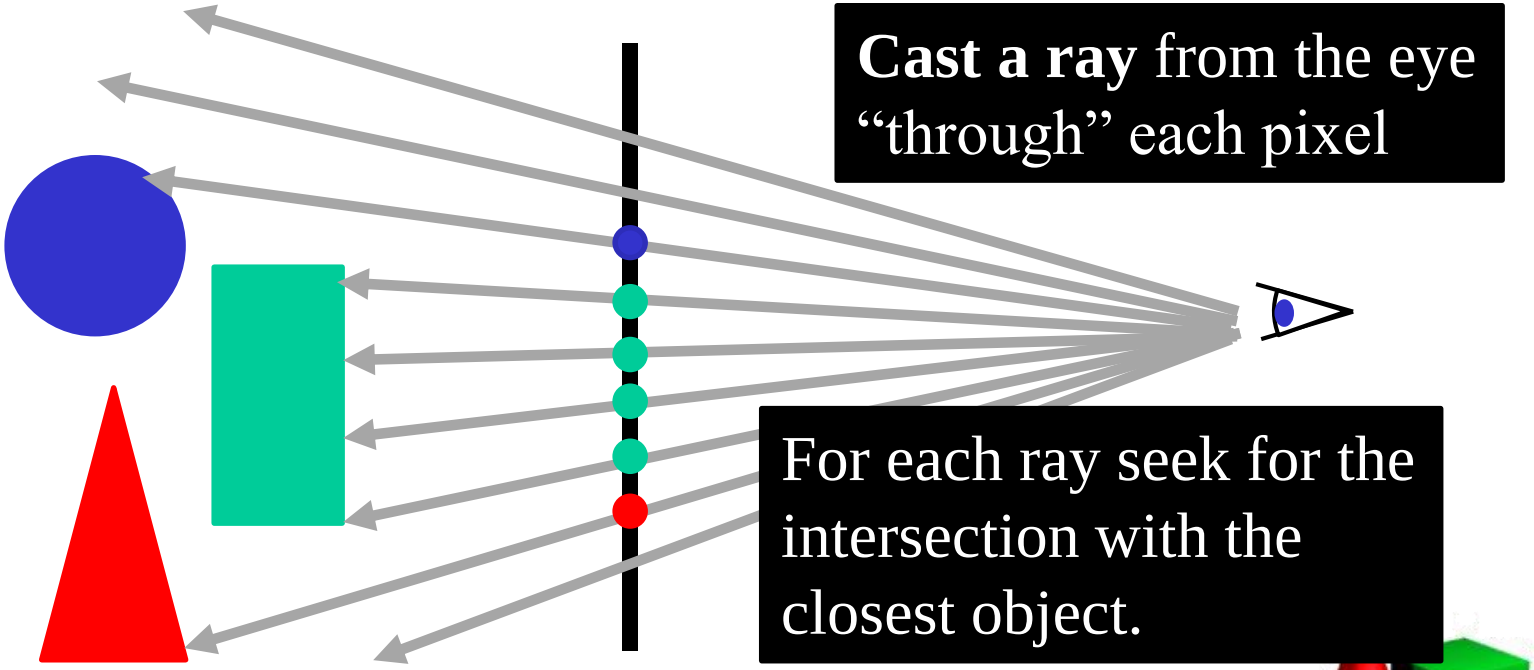
Graphics Pipeline vs Raycasting

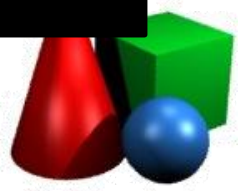
- **Raycasting:**
  - For each **pixel**, find the corresponding **object** (and then color each pixel appropriately)



Graphics Pipeline vs Raycasting

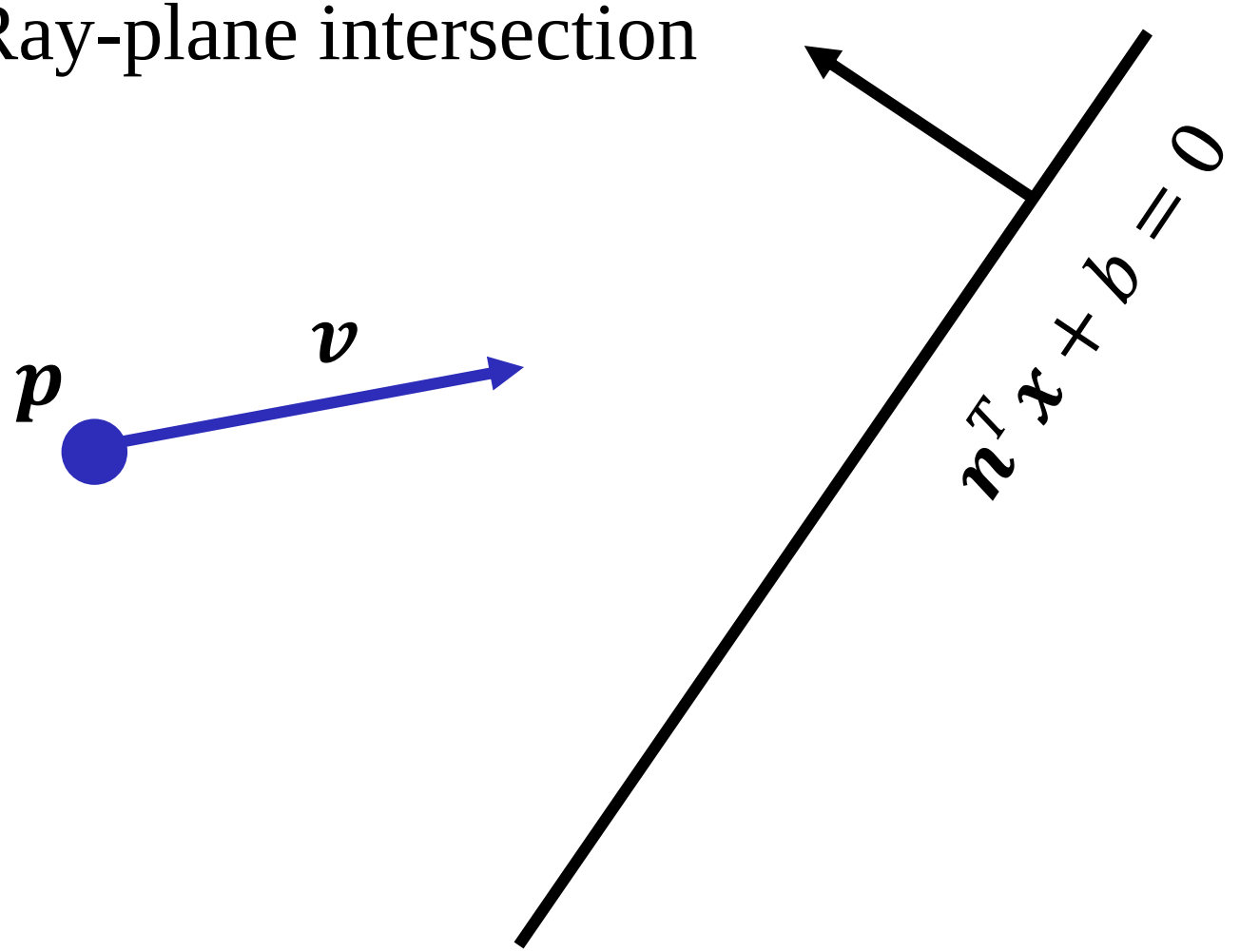
- **Raycasting:**
  - For each **pixel**, find the corresponding **object** (and then color each pixel appropriately)





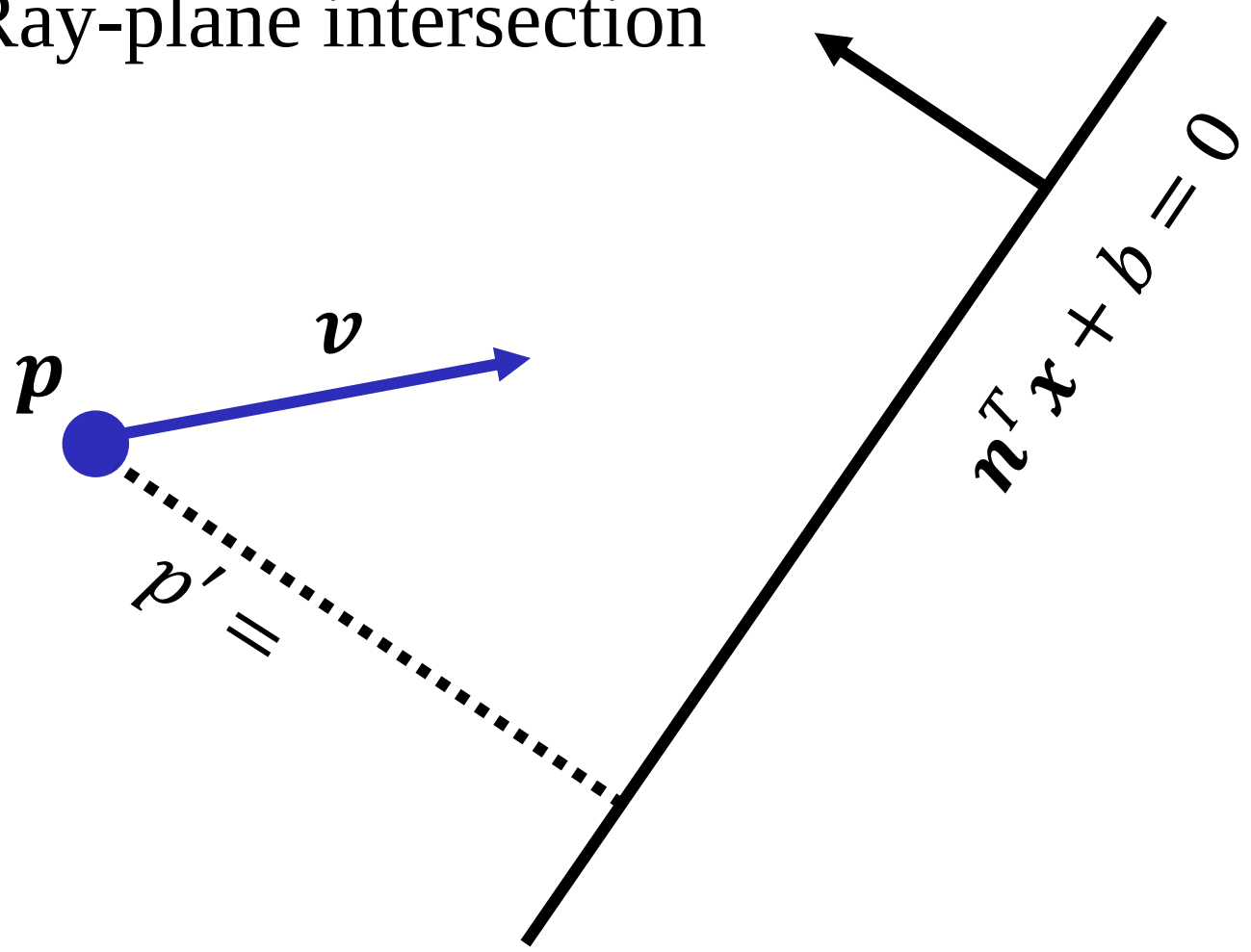
Example: Plane

Ray-plane intersection



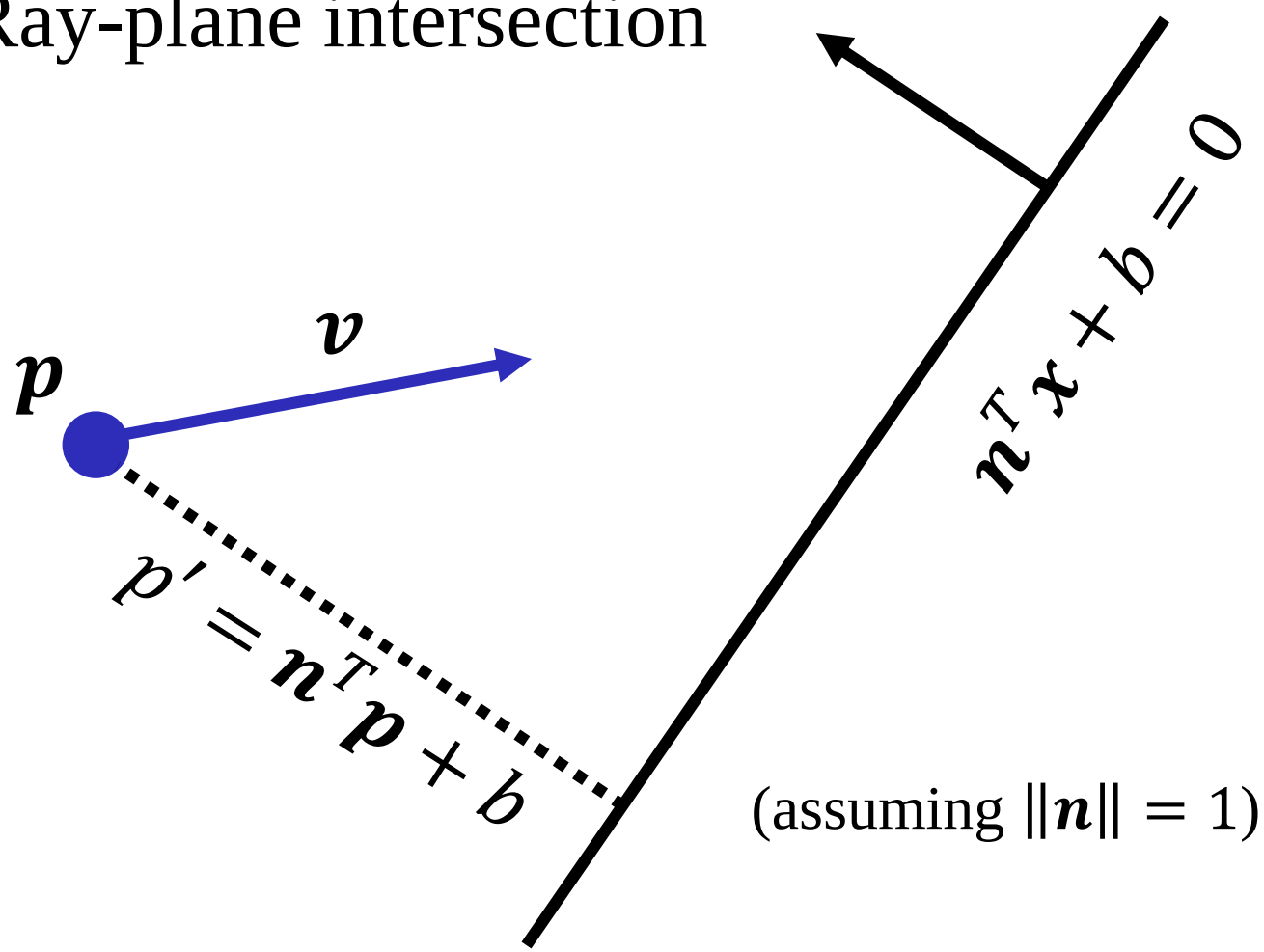
Example: Plane

Ray-plane intersection



Example: Plane

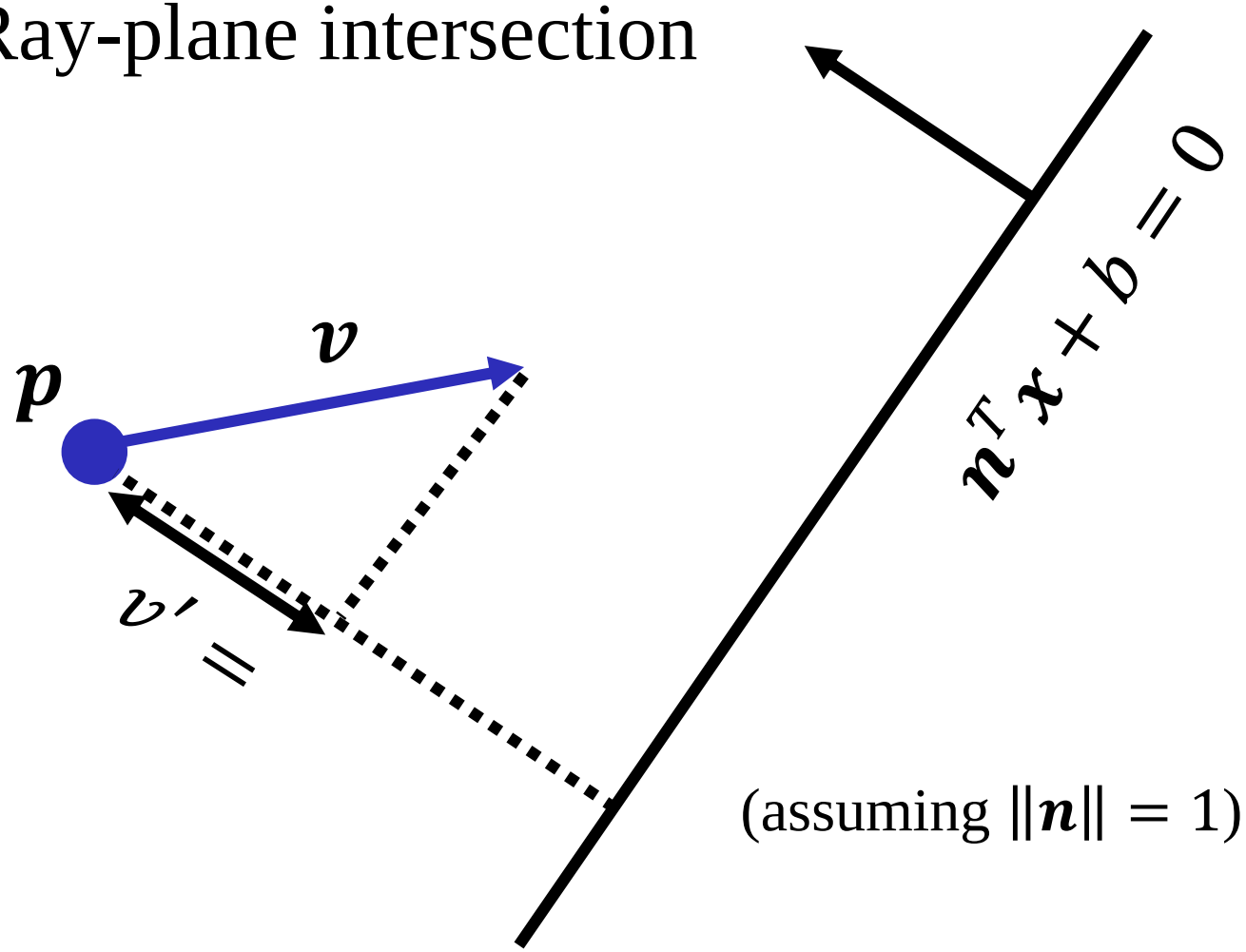
Ray-plane intersection



(assuming $\|\boldsymbol{n}\| = 1$)

Example: Plane

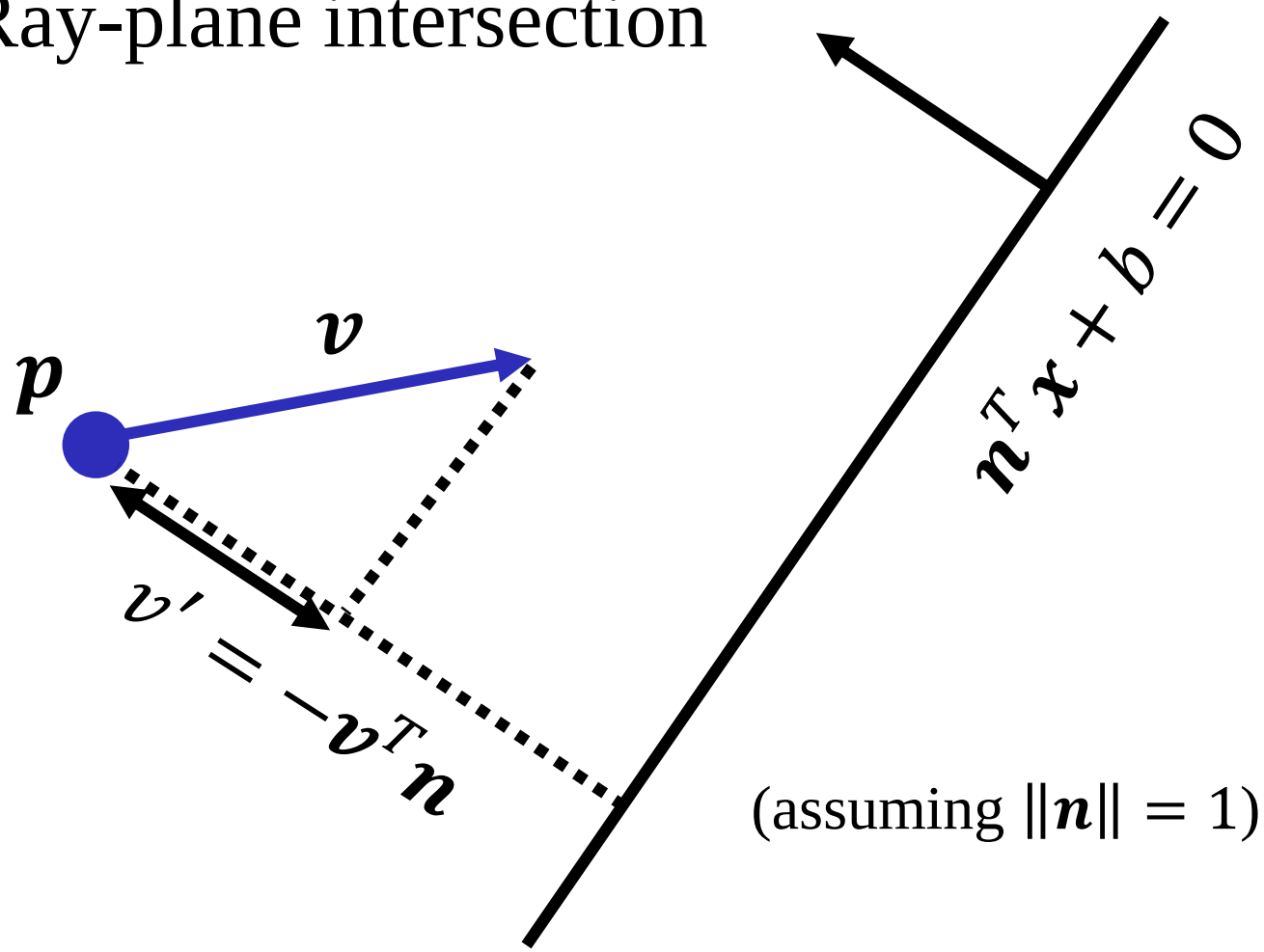
Ray-plane intersection



(assuming $\|\boldsymbol{n}\| = 1$)

Example: Plane

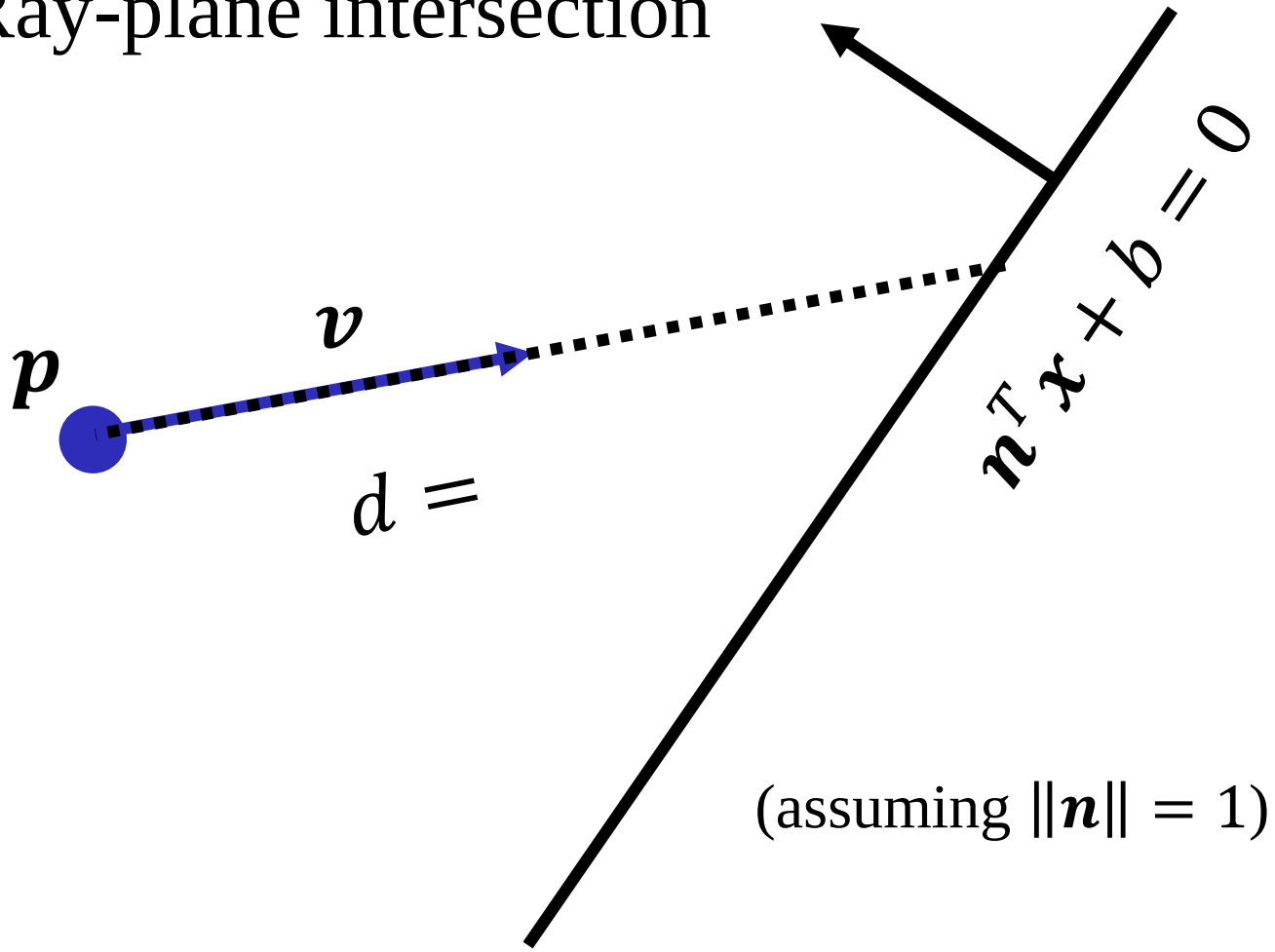
Ray-plane intersection



(assuming $\|\boldsymbol{n}\| = 1$)

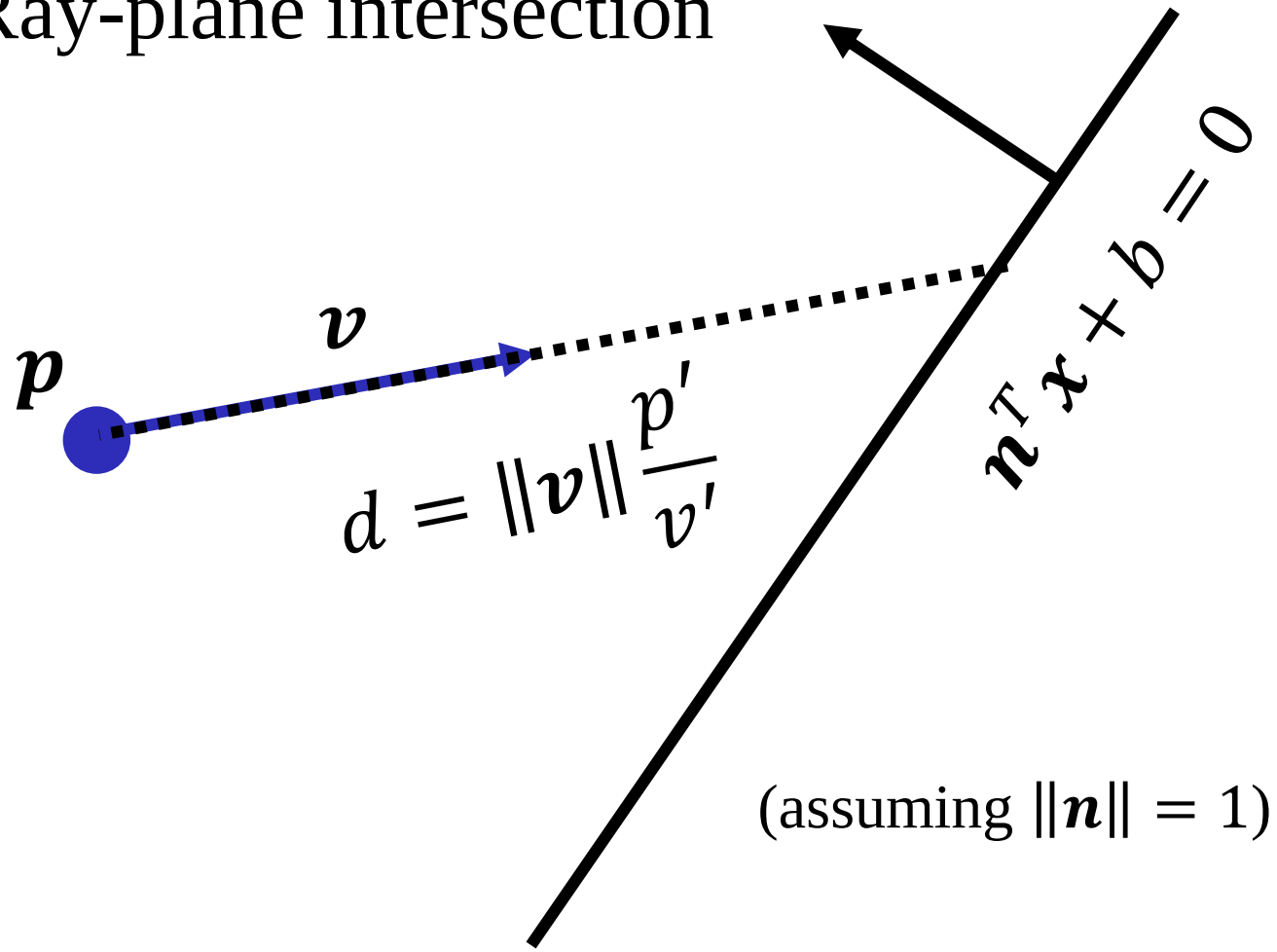
Example: Plane

Ray-plane intersection



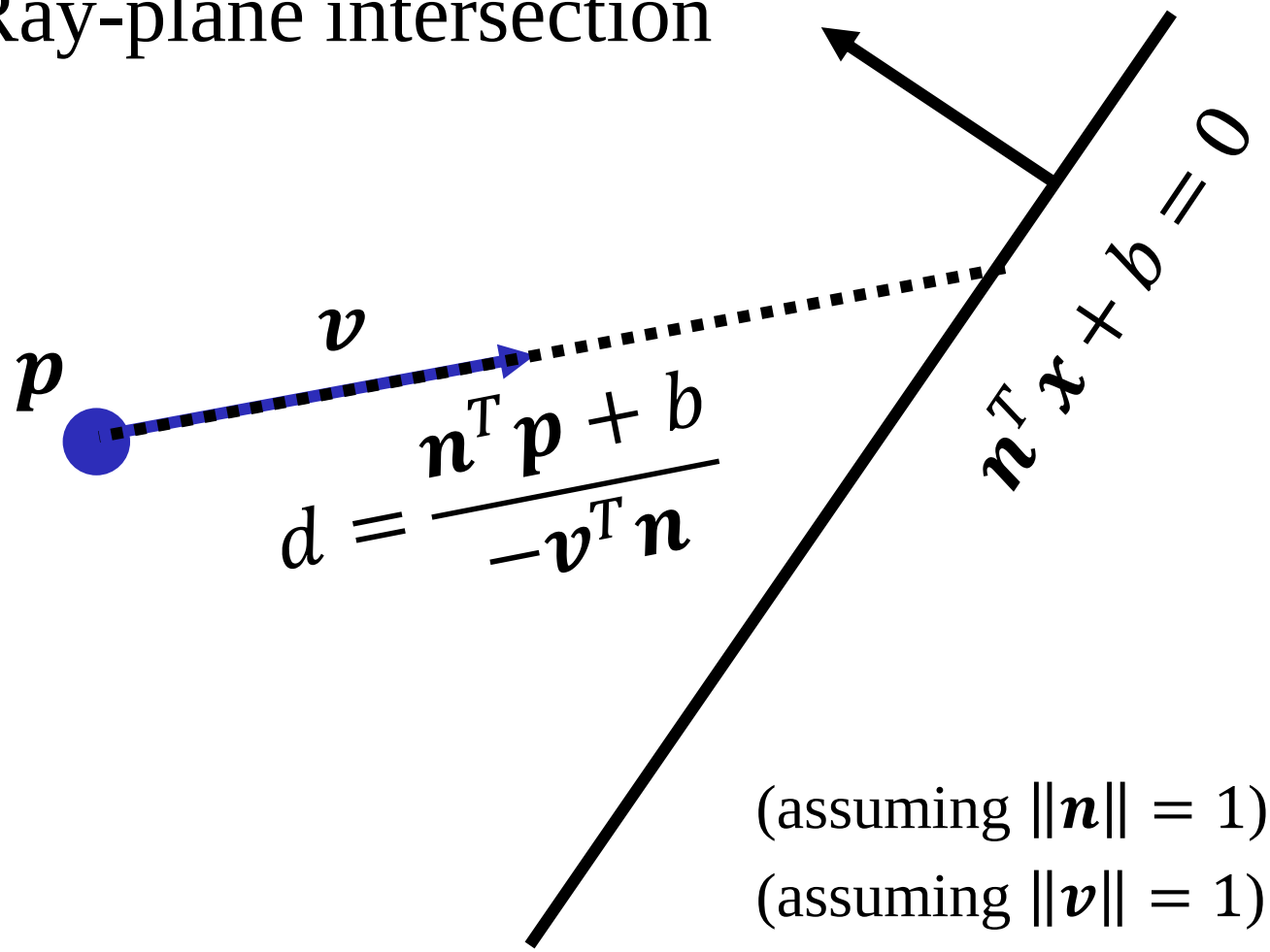
Example: Plane

Ray-plane intersection



Example: Plane

Ray-plane intersection



(assuming $\|\boldsymbol{n}\| = 1$)

(assuming $\|\boldsymbol{v}\| = 1$)

Example: Plane

```
float raycast_plane(vec3 from, vec3 dir,
                    vec3 n,    float b) {
   if (dot(n, dir) == 0)
      return -1;
   else
      return -(dot(n, from) + b)/dot(n, dir);
}
```

Example: Plane

```
float raycast_plane(vec3 from, vec3 dir,
                    vec3 n,    float b) {
   if (dot(n, dir) == 0)
      return -1;
   else
      return -(dot(n, from) + b)/dot(n, dir);
}
```

Separate ray-surface intersection routine can be derived for every type of surface.

Quiz

- Suppose we have n objects, each with screen area ~1000 pixels. Let the total screen area be 800x600 = 480 000 pixels.

- Complexity of (naïve) graphics pipeline:
  - ?
- Complexity of (naïve) raycasting:
  - ?

Quiz

- Suppose we have n objects, each with screen area ~1000 pixels. Let the total screen area be 800x600 = 480 000 pixels.

- Complexity of (naïve) graphics pipeline:
  - $1000\,n$
- Complexity of (naïve) raycasting:
  - ?

Quiz

- Suppose we have n objects, each with screen area ~1000 pixels. Let the total screen area be 800x600 = 480 000 pixels.

- Complexity of (naïve) graphics pipeline:
  - $1000\,n$
- Complexity of (naïve) raycasting:
  - $480\,000\,n$

Quiz

- Suppose we have n objects, each with screen area ~1000 pixels. Let the total
  - $1000\,n$
- Complexity of (naïve) raycasting:
  - $480\,000\,n$

In the worst case (each object occupies the whole screen) the complexities are equal.

Graphics Pipeline vs Raycasting

- Theoretically, the complexity of naïve raycasting is not worse than that of the graphics pipeline.

Graphics Pipeline vs Raycasting

- Theoretically, the complexity of naïve raycasting is not worse than that of the graphics pipeline.
- In fact, in the old days, properly-done (1D) raycasting was more efficient.

Graphics Pipeline vs Raycasting

- However,
  - Raycasting parallelizes over **outputs** (pixels) and uses **full scene data for each pixel**.
  - GP parallelizes over **inputs** (splits input into triangles) which makes memory access of each subtask *local*.
- For large scenes the latter approach fits the architecture of modern hardware much better (shared memory access is slow).

Benefits of Raycasting

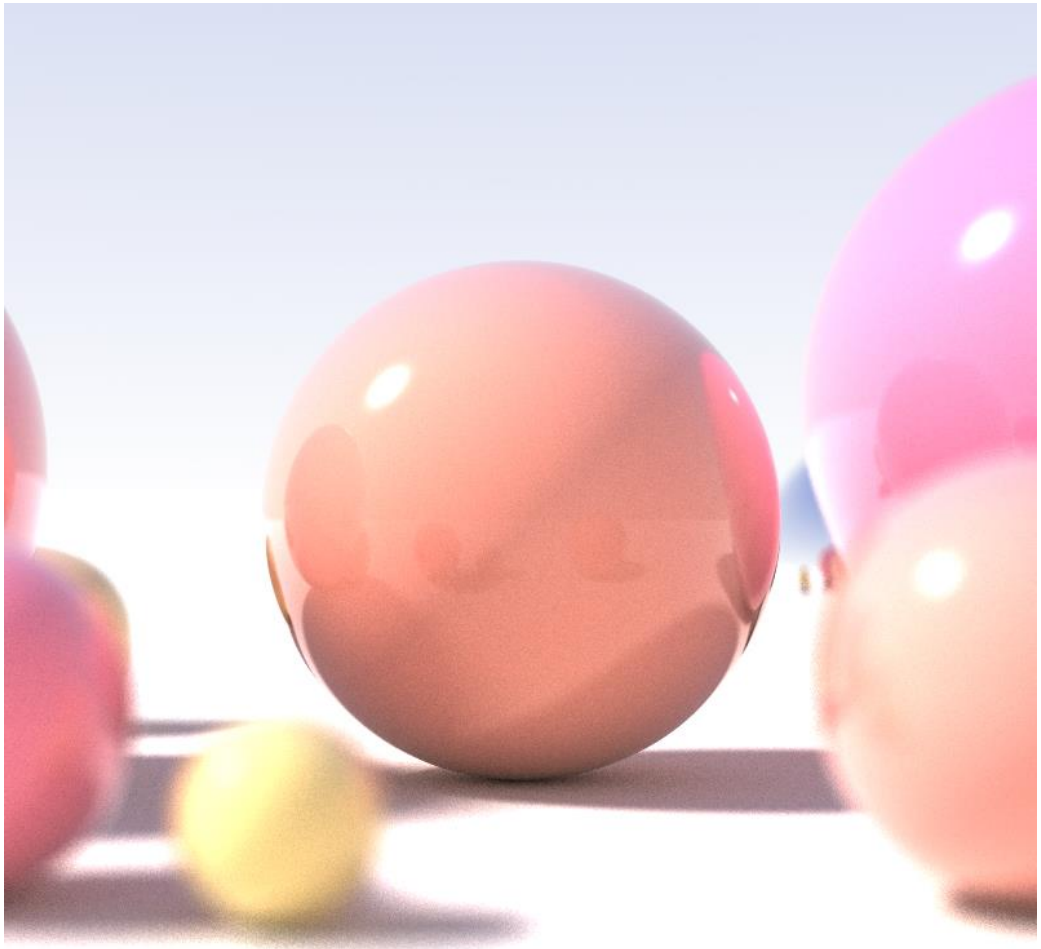
- On the other hand, having full access to the scene while rendering lets you implement **global illumination** effects much more naturally than in the standard graphics pipeline.

Global vs Local illumination

- **Local illumination model**:
  - Use only the information about light sources in lighting computations. Other objects in the scene do not matter.

- **Global illumination model**:
  - Take into account light scattering and occlusion due to **all** objects in the scene:
    - Shadows, reflection, refraction, ambient scattering, ambient occlusion

Global vs Local illumination



http://en.wikipedia.org/wiki/File:Recursive_raytrace_of_a_sphere.png

Raytracing

- **Raytracing** :=
  - Basic raycasting
  - + Shadows
  - + Reflections
  - + Refractions
  - + Recursion
  - + Backwards raytracing
  - + Stochastic raytracing

Shadows via Raycasting

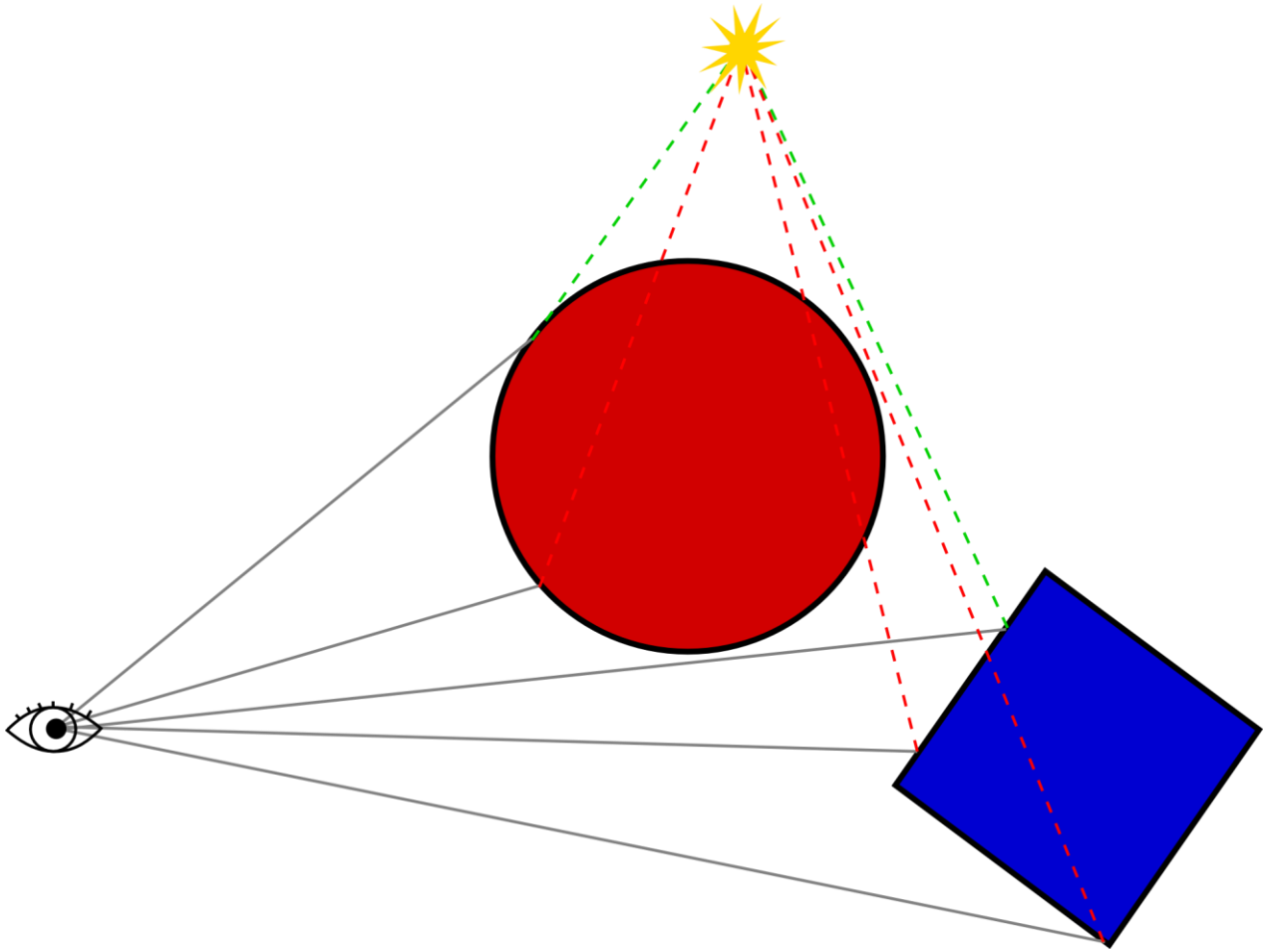
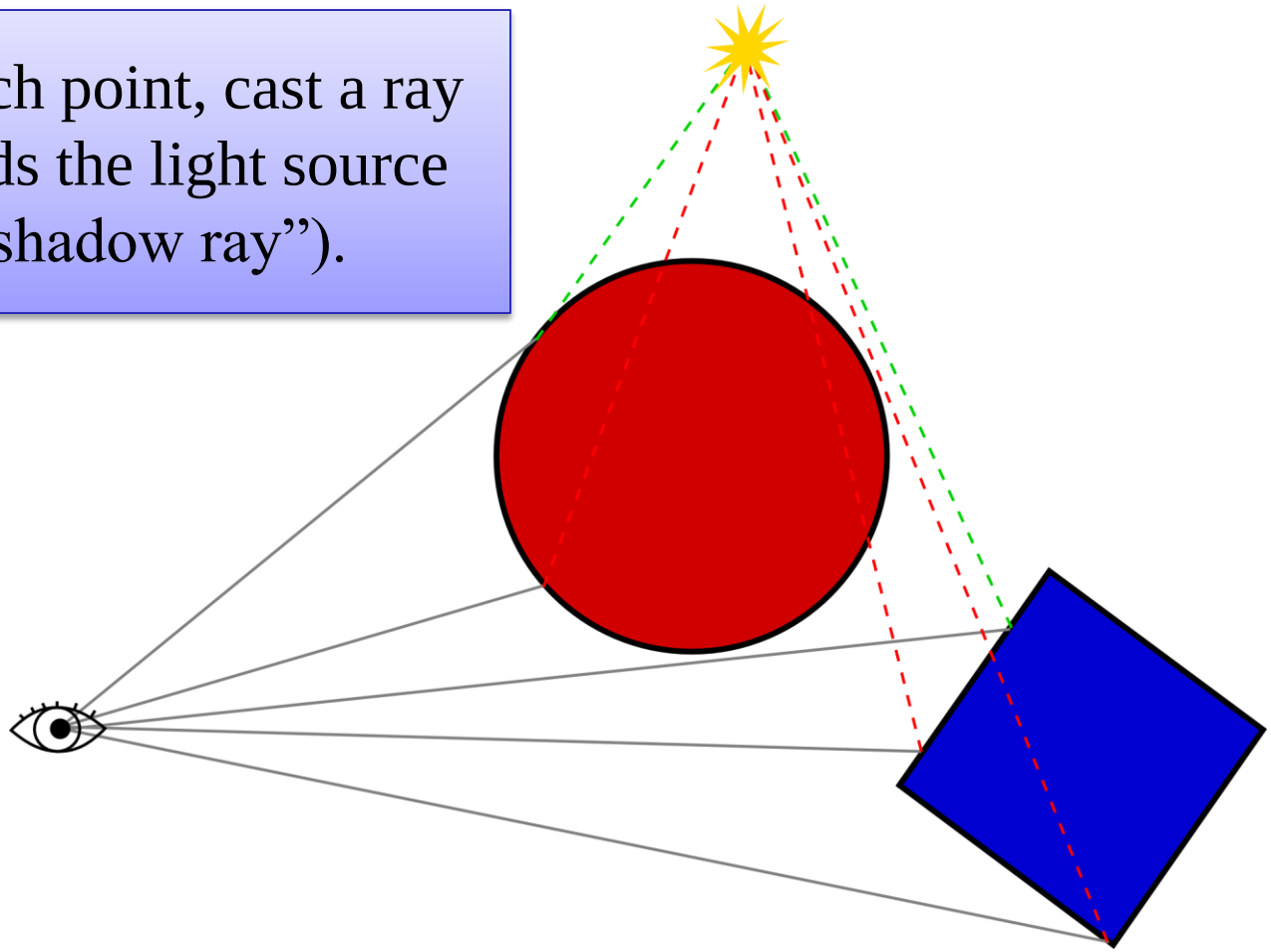


Illustration © Mark Fishel

Shadows via Raycasting

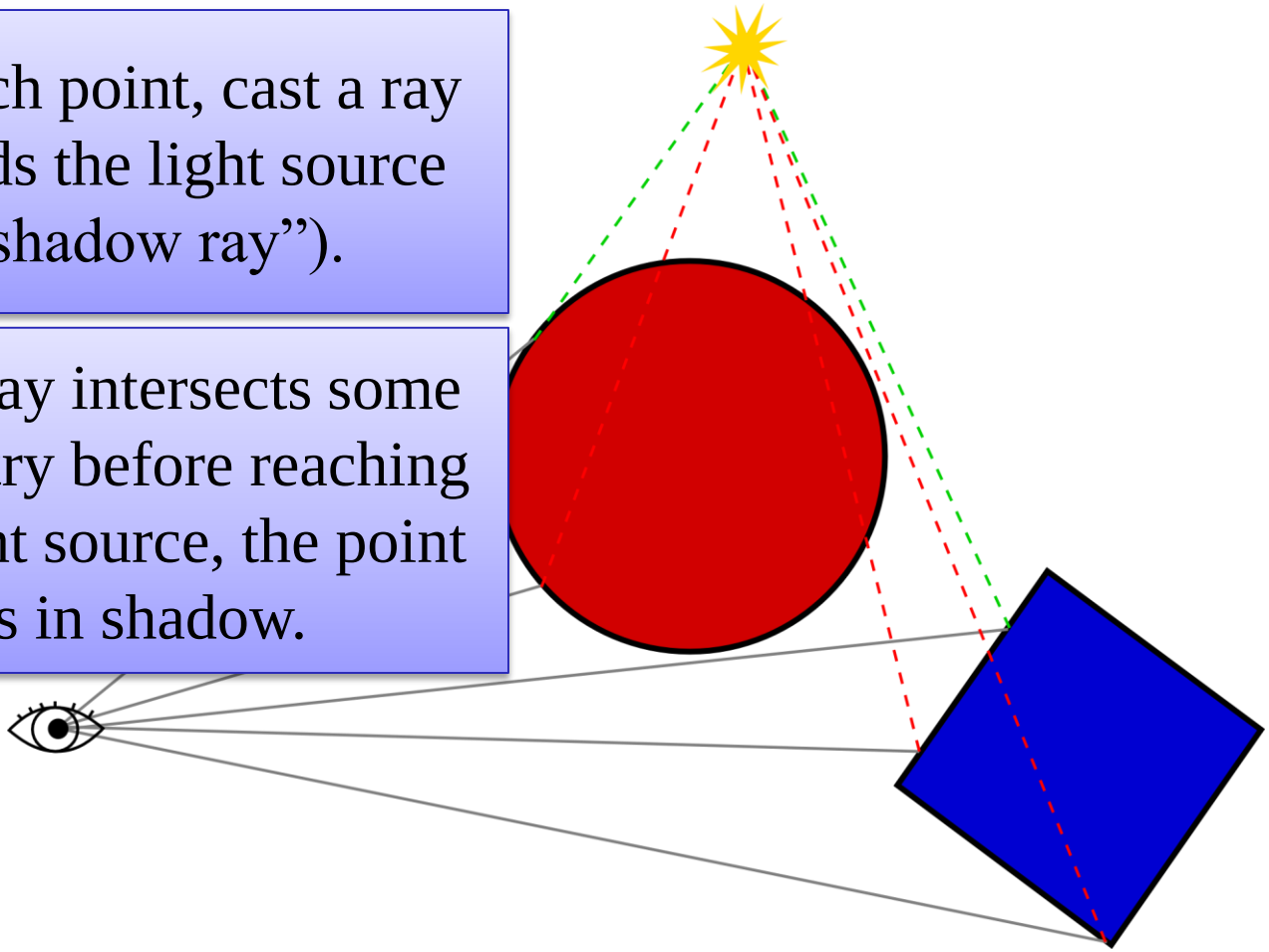
For each point, cast a ray towards the light source ("shadow ray").



Shadows via Raycasting

For each point, cast a ray towards the light source (“shadow ray”).

If the ray intersects some geometry before reaching the light source, the point is in shadow.



Reflections via Raycasting

- For each point, continue the ray in a **reflected** direction, and combine the local lighting of the point with whatever is hit by the reflected ray.

$$C_{local\ model} + k_{refl} C_{refl}$$

(where k_{refl} is the "reflectivity coefficient")

Refractions via Raycasting

- For each point, continue the ray in a **refracted** direction, and combine the local lighting of the point with whatever is hit by the reflected ray + refracted ray

$$C_{local\ model} + k_{refl} C_{refl} + k_{refr} C_{refr}$$

Recursive Raytracing

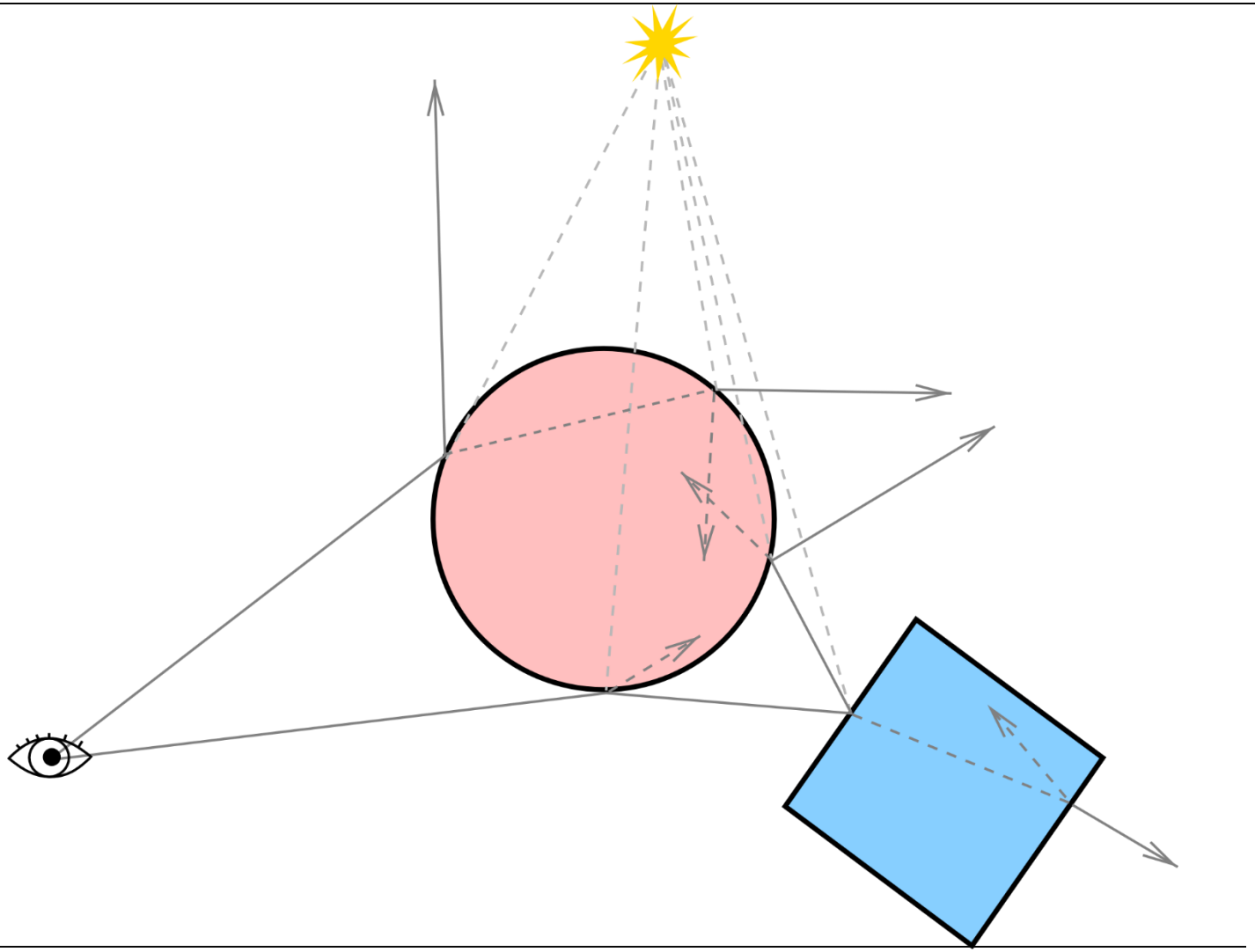


Illustration © Mark Fishel

Faster Raytracing

- **BVHs & BSPs**:
  - Quiz: What were those?

Faster Raytracing

- **BVHs & BSPs**:
  - Bounding Volume Hierarchies allow to quickly cull away irrelevant portions of the scene. Binary Space Partitioning trees let you seek for ray intersections in a front-to-back order.

Faster Raytracing

- "**Object z-buffer**":
  - Pre-render the scene using the Graphics pipeline, storing for each pixel the id of the topmost object rendered at that pixel.
  - This effectively replaces the first raycasting step with a graphics pipeline, making it faster.

Faster Raytracing

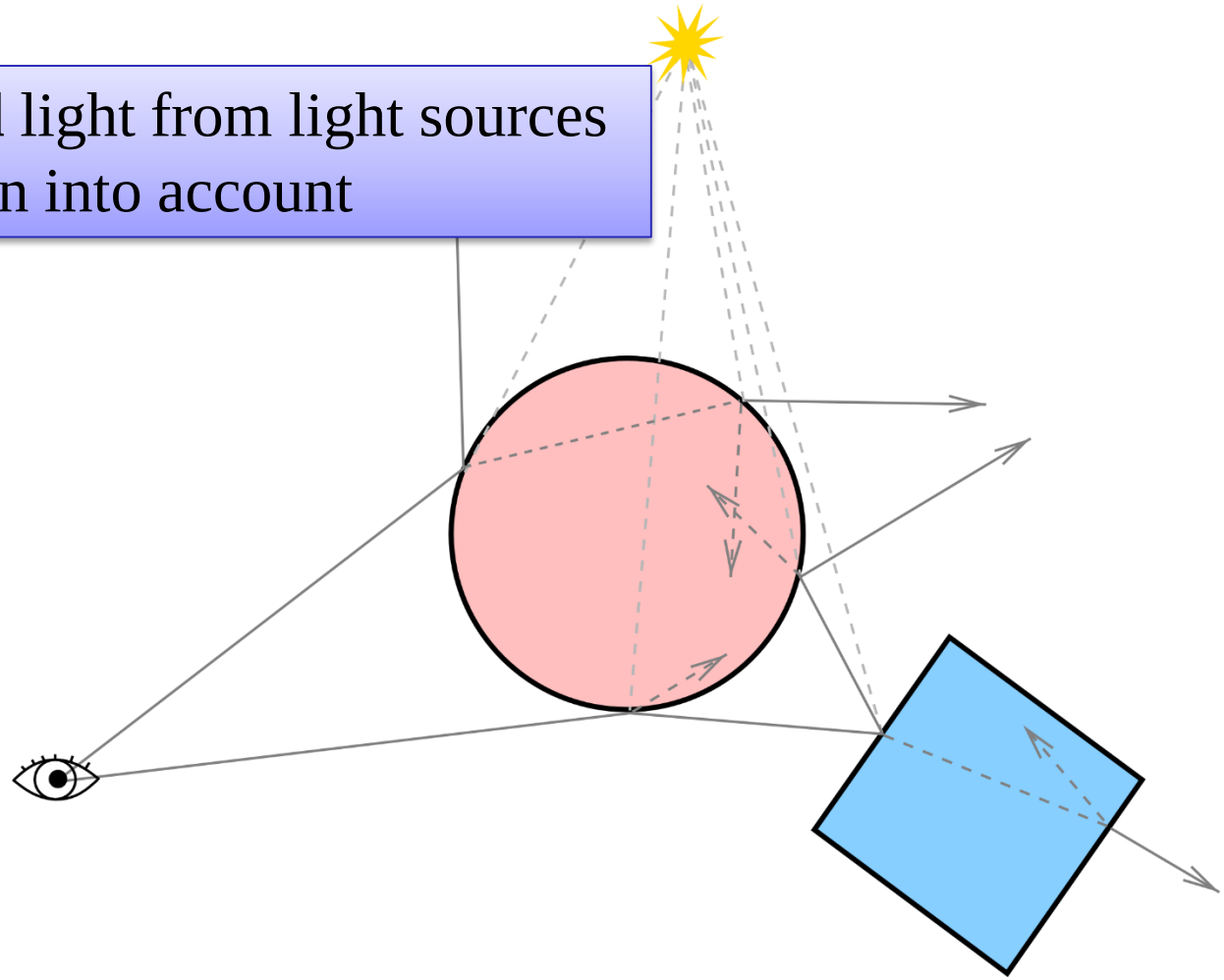
- **Adaptive recursion depth**:
  - Keep track, for each ray, what will be its final contribution to the pixel it originated from.
  - E.g. if after hitting a surface we reflect with reflection coefficient 0.2, then refract with coefficient 0.1, we may stop as further computations won’t contribute more than 0.02.

Faster Raytracing

- **Shadow mapping**:
  - Instead of casting each “shadow ray”, use a precomputed depth buffer like in shadow mapping.

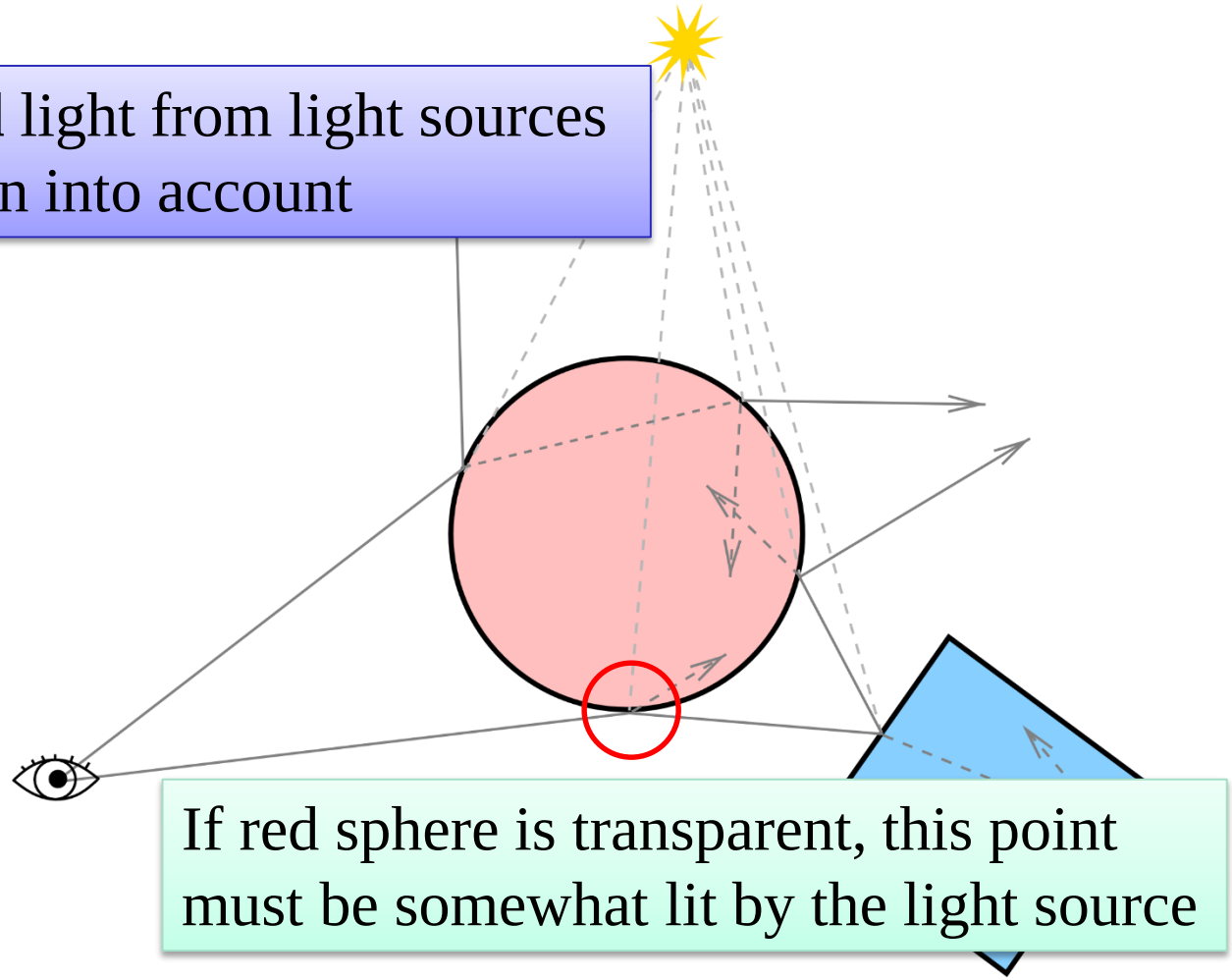
Refracted lighting problem

Refracted light from light sources is not taken into account



Refracted lighting problem

Refracted light from light sources is not taken into account



If red sphere is transparent, this point must be somewhat lit by the light source

Backwards Raytracing

- **Solution**:
  - Split geometry into patches.
  - Make a raytracing pass with rays originating from the light sources
  - For each patch store its lighting data.
  - Use this data in the normal forward pass.

Aliasing

- Aliasing can be a considerable problem for raytracing implementations.

- Solutions:
  - Cone tracing, Beam tracing, Pencil tracing
  - **Stochastic raytracing**

Stochastic raytracing

- Recall that a common way of dealing with aliasing artifacts is by using a convolution with a filter.

- In layman's terms: send multiple rays for each pixel, sampling slightly different directions, and average the results.

Basic antialiasing

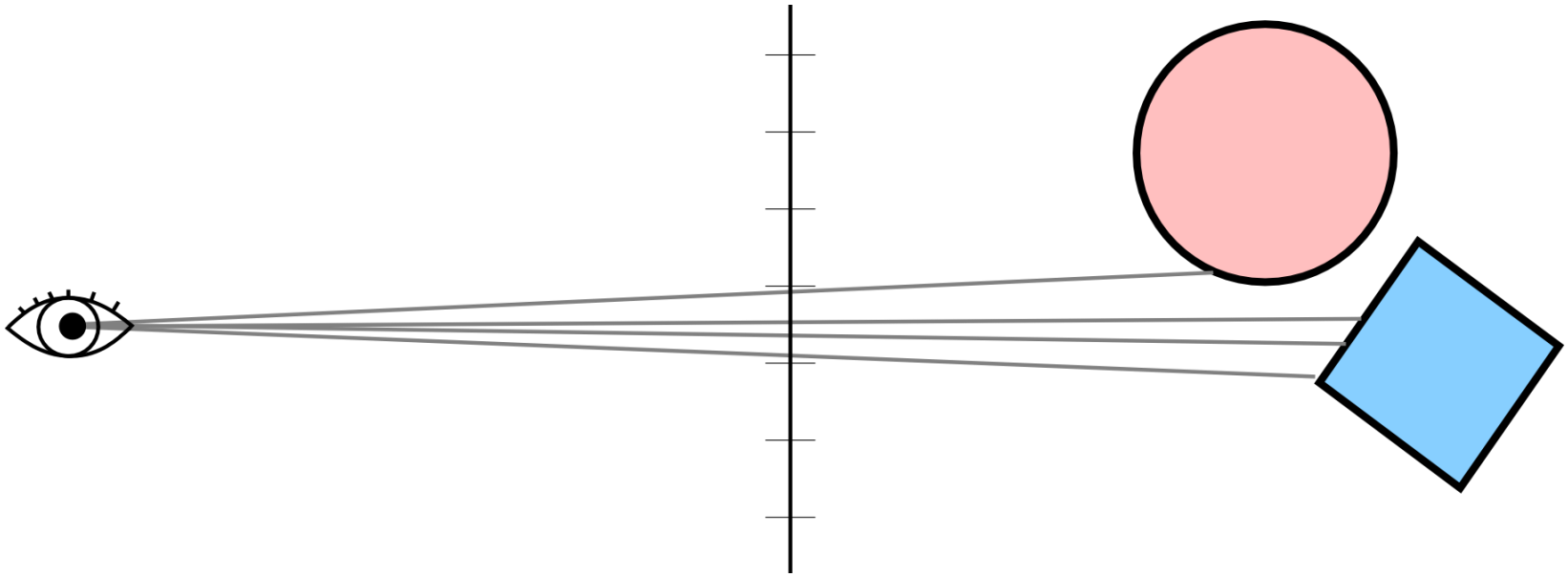
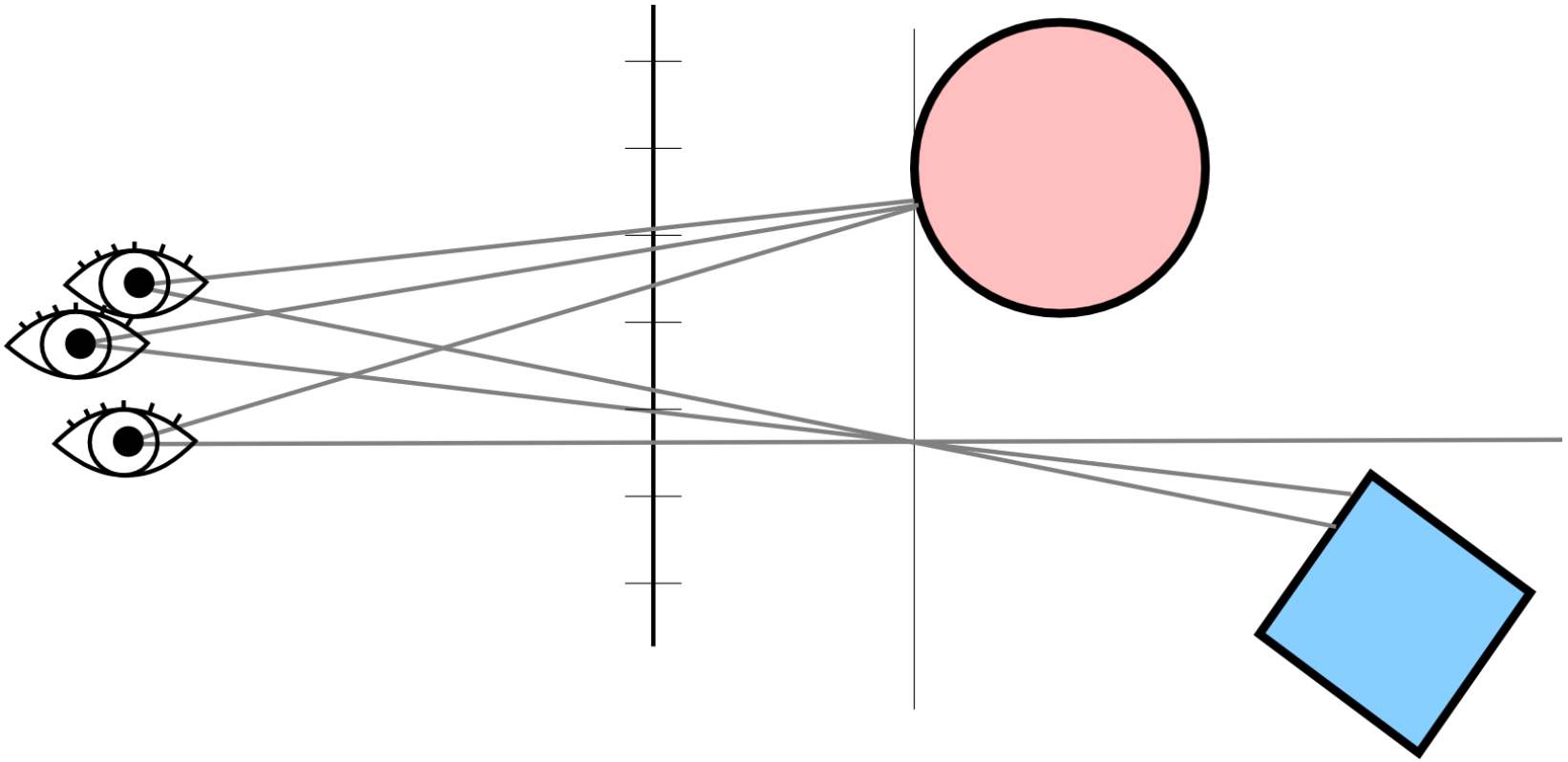


Illustration © Mark Fishel

Multiple rays per pixel

- For each ray we can vary many parameters:
  - Ray direction (basic antialiasing)
  - Ray origin (depth of field)
  - Object positions (motion blur)
  - Light source positions (soft shadows)
  - Ray light frequency (true refractions)

Depth of Field



True refraction

One ray per light frequency

Combined stochastic rays

- Suppose we want to do
  - 4x4-grid jittered antialiasing
  - + 4x4-grid jittered DOF.
  - + 10-frame motion blur

- How many rays per pixel should we send?

Combined stochastic rays

- Suppose we want to do
  - 4x4-grid jittered antialiasing
  - + 4x4-grid jittered DOF.
  - + 10-frame motion blur

- We do not need to send 16x16x10 = 2560 rays for all possible combinations!
- Choosing, say, 4x4 random parameter combinations can suffice.

Outline

- Raycasting
- Raytracing
- Raymarching / Sphere tracing

- Rendering equation solvers
  - Radiosity, Photon mapping, Path tracing

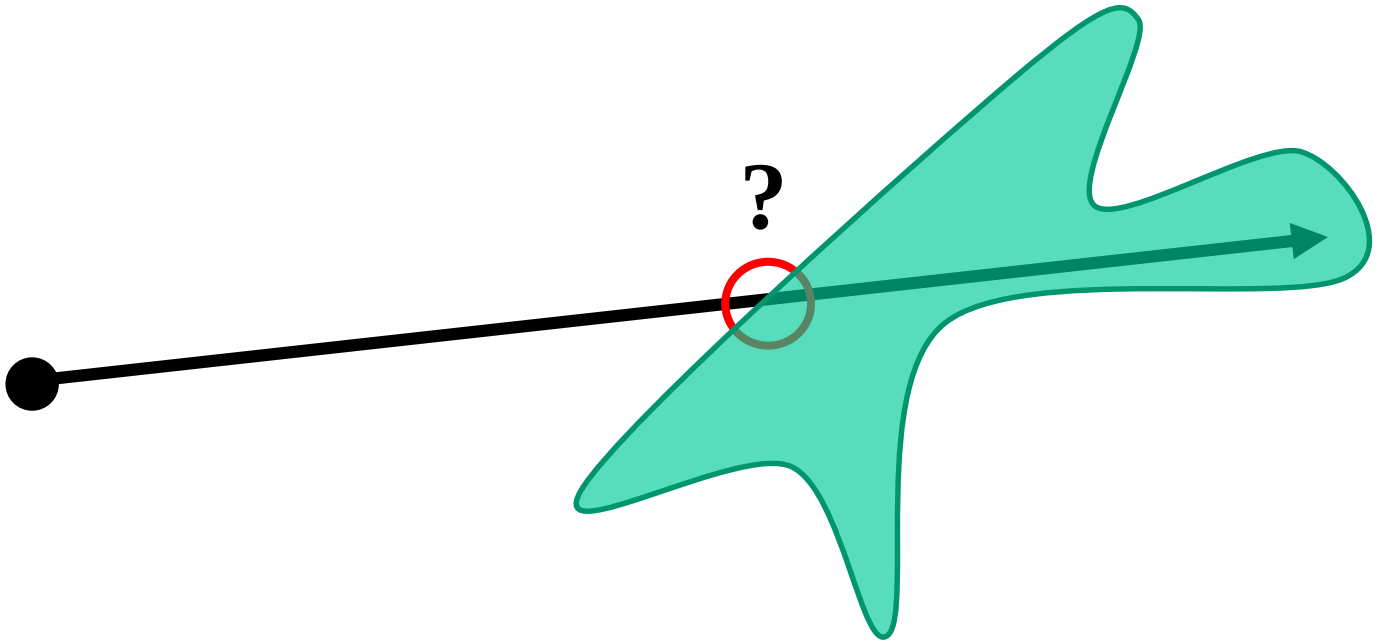
Outline

- Raycasting
- Raytracing
- Raymarching / Sphere tracing

- Rendering equation solvers
  - Radiosity, Photon mapping, Path tracing

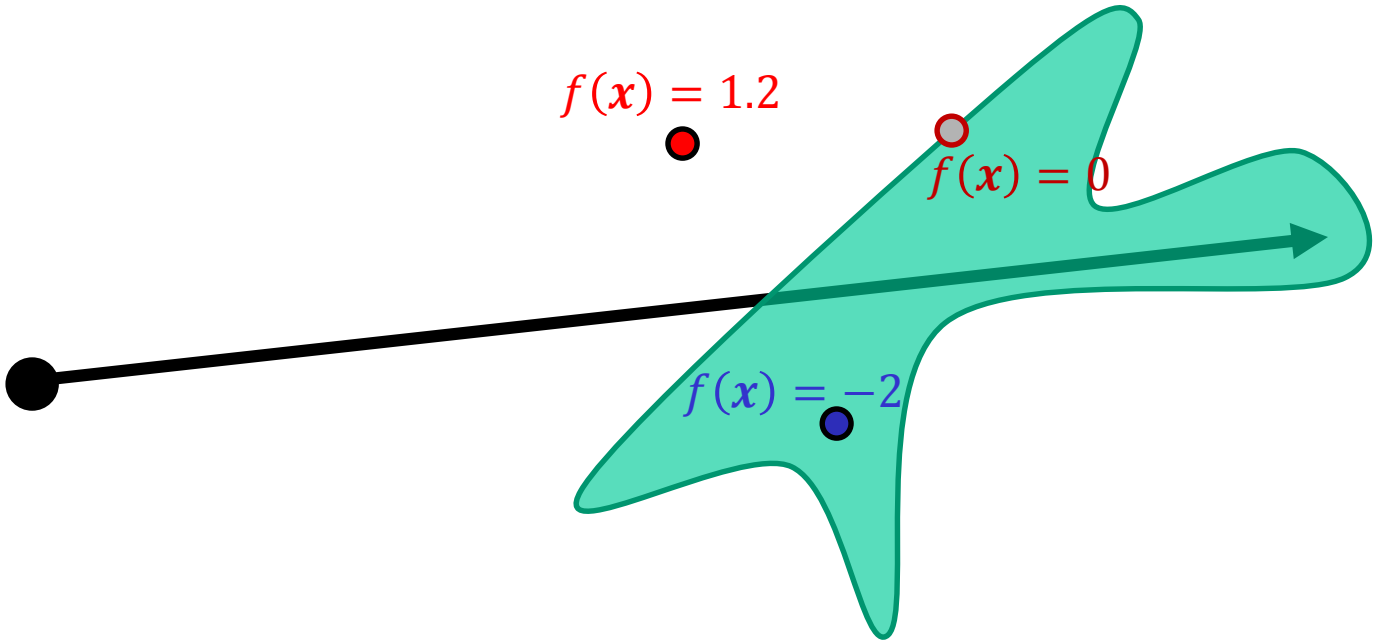
Problems with Raytracing

- Finding ray intersections with complex shapes can be hard, and not always analytically possible.



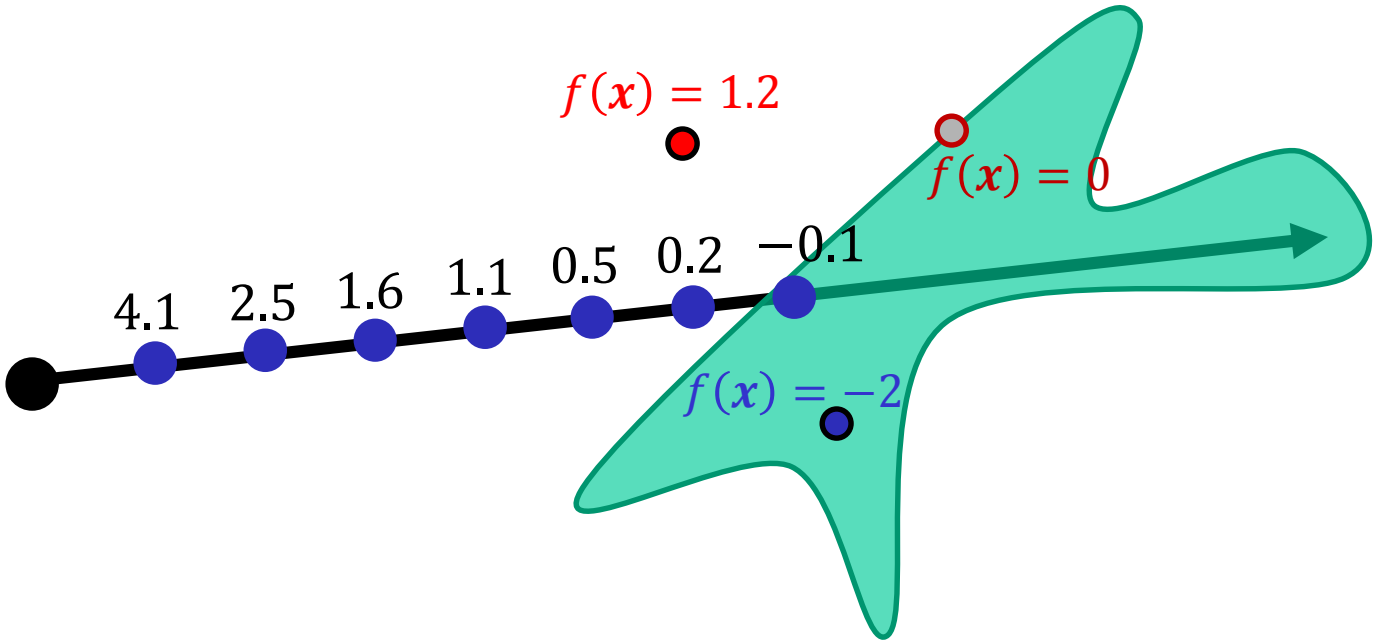
Raymarching

- Idea: use implicit surface representation:
  - $f(\boldsymbol{x}) = 0$, for points $\boldsymbol{x}$ on the surface.
  - $f(\boldsymbol{x}) > 0$, for points outside the shape.



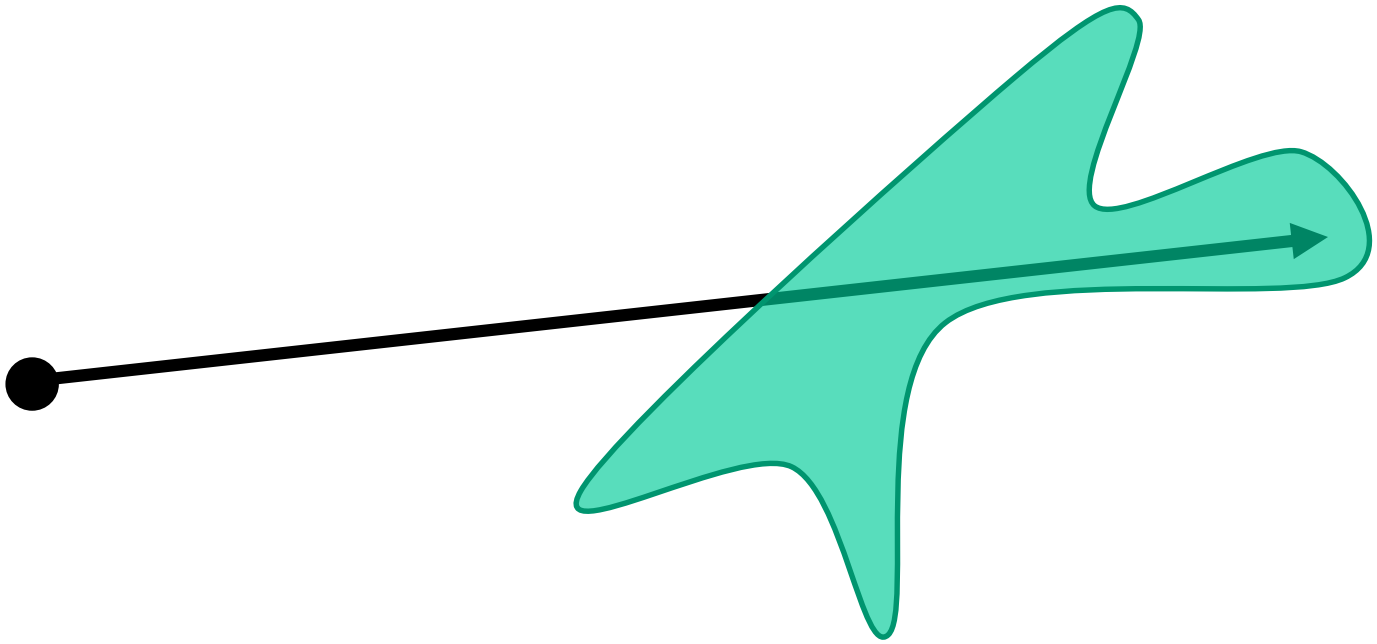
Raymarching

- Now we can “walk” along the ray in small steps, seeking for the point where $f(\boldsymbol{x}) = 0$



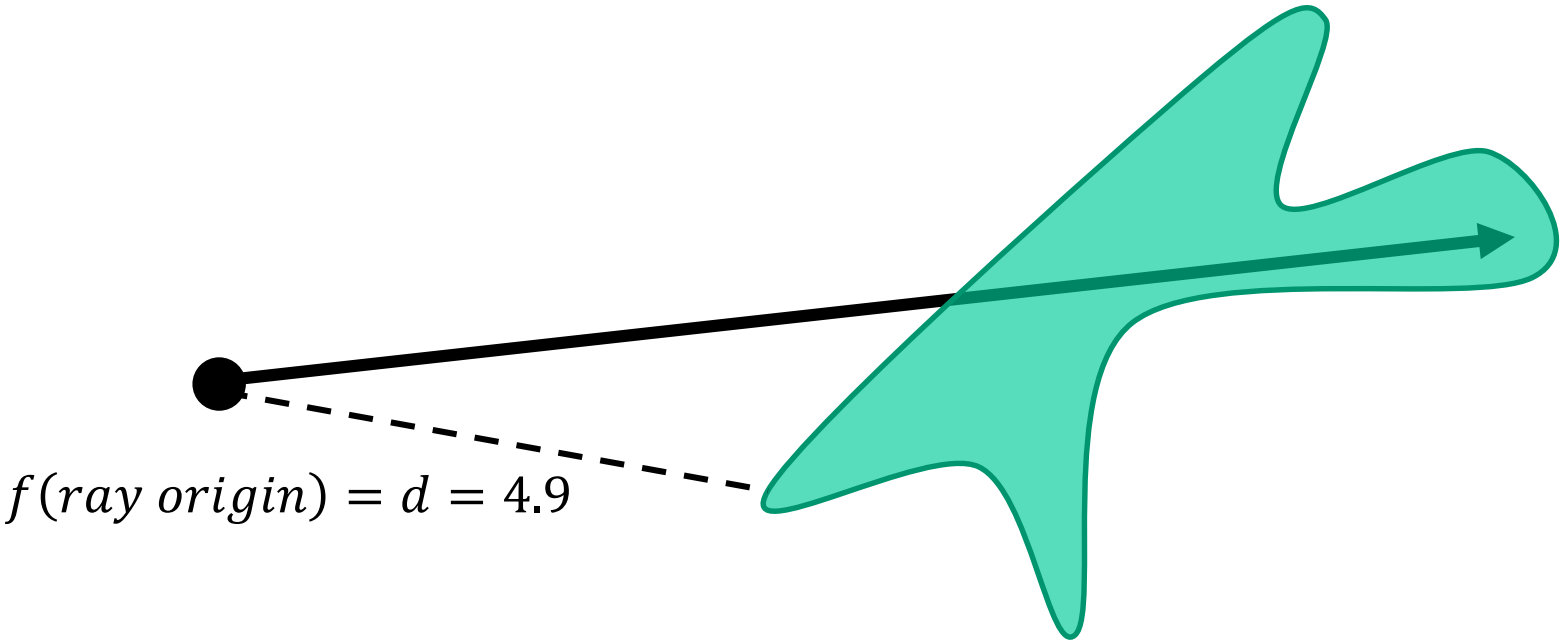
Sphere tracing

- Even better if $f(\boldsymbol{x})$ is a **lower bound on the distance** to any shape in the scene. Then we can safely use step size $f(\boldsymbol{x})$ along the ray.



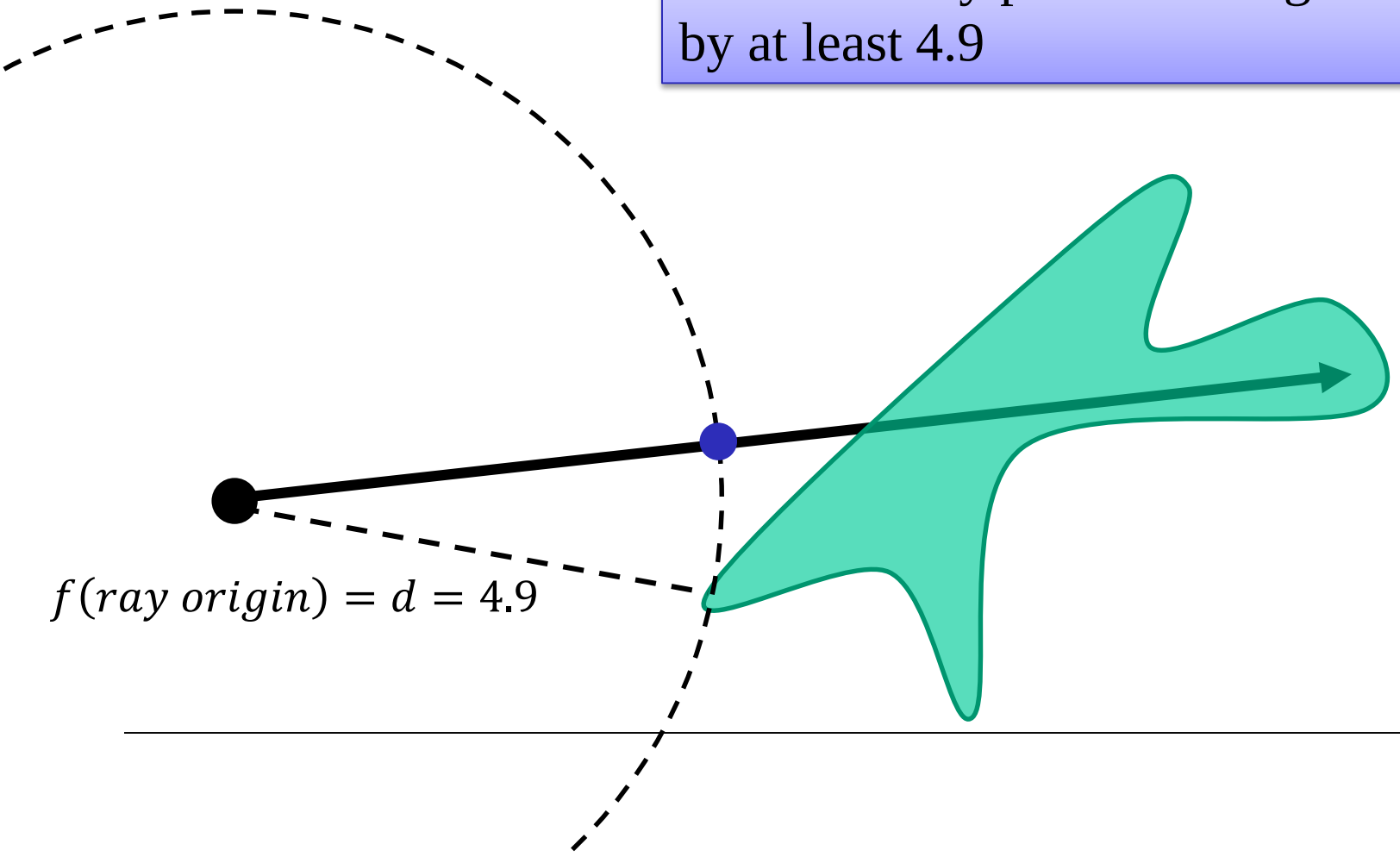
John C Hart. Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.87.339&rep=rep1&type=pdf

Sphere tracing

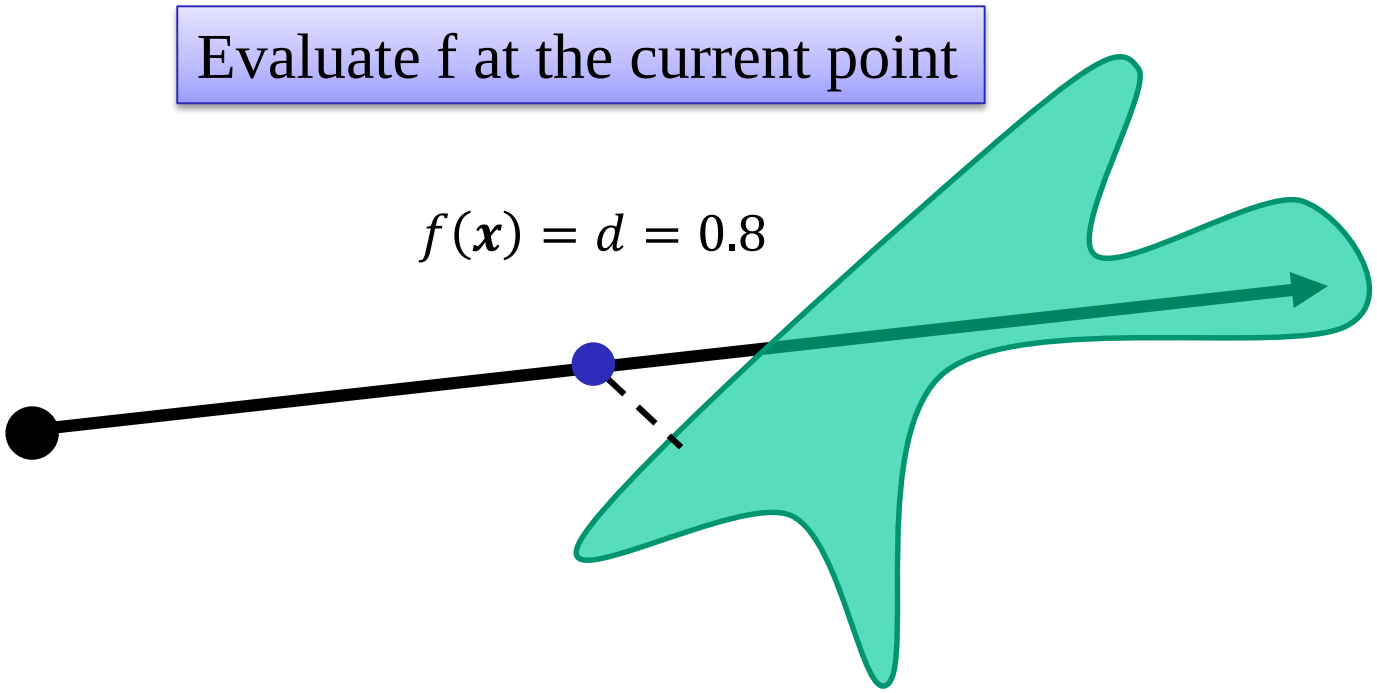


Sphere tracing

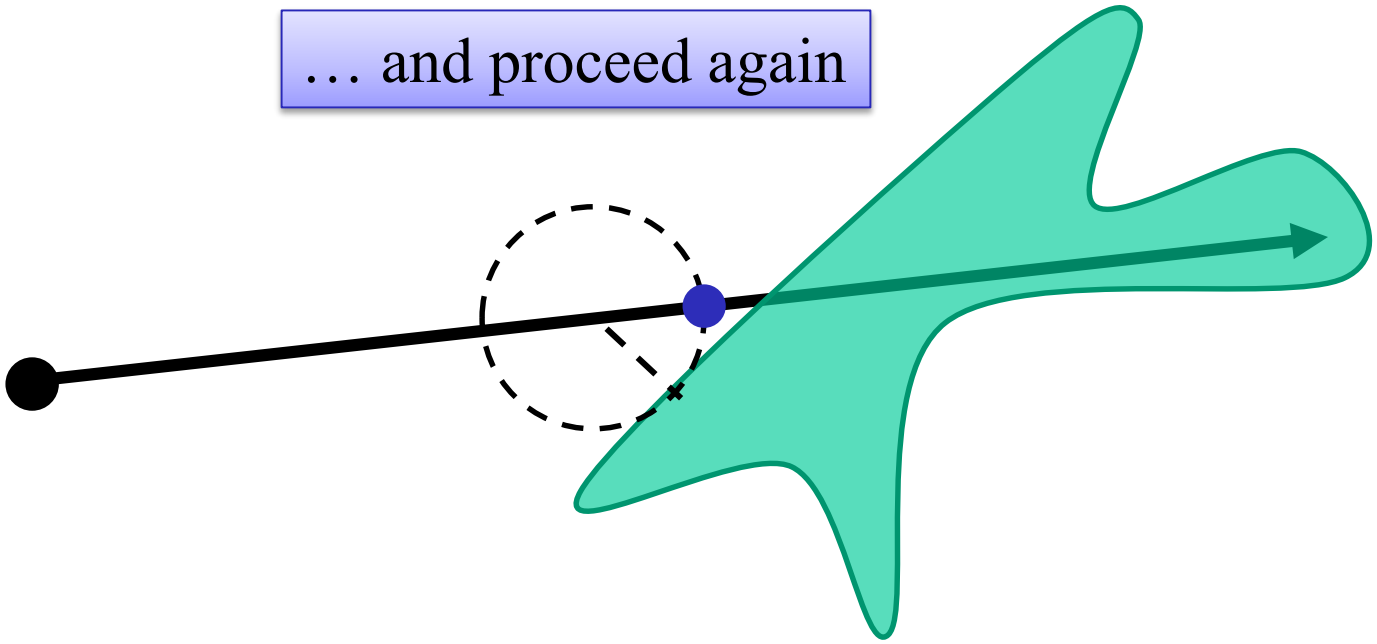
We can safely proceed along the ray by at least 4.9



Sphere tracing

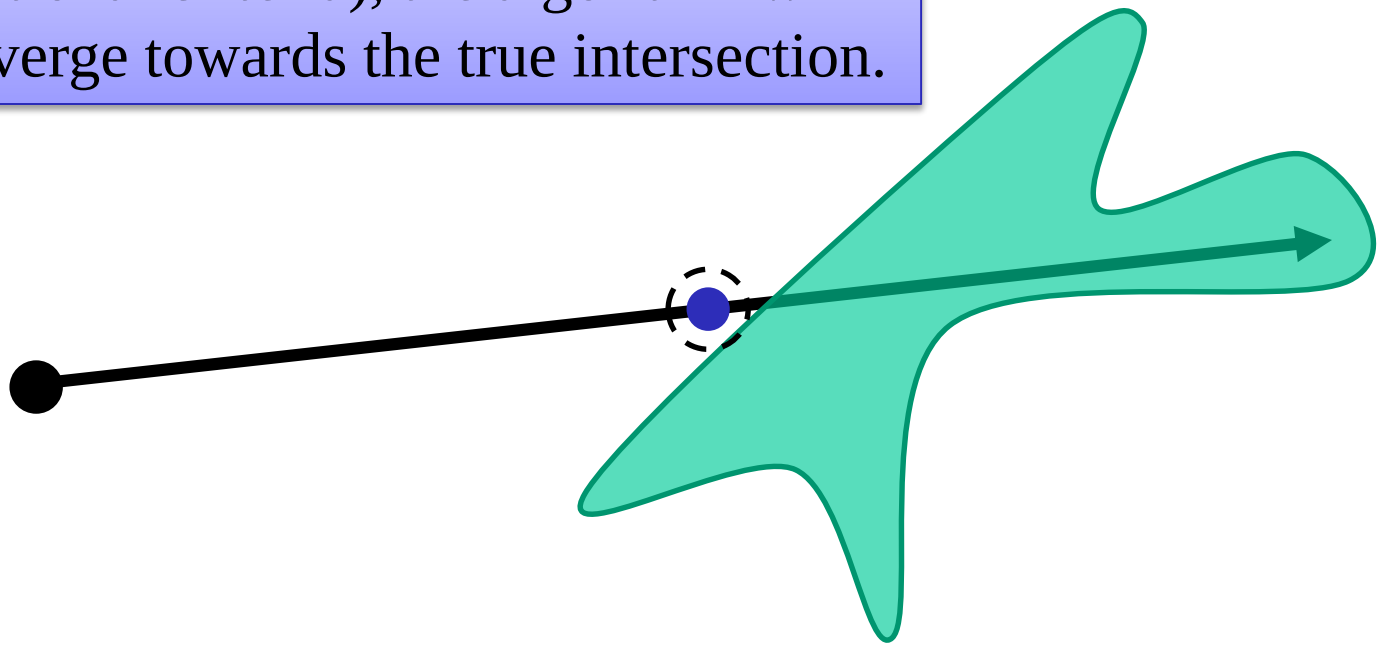


Sphere tracing



Sphere tracing

If $f(x)$ corresponds to an actual distance (or a lower bound with some additional criteria), the algorithm will converge towards the true intersection.



Sphere tracing / Raymarching

```
float raymarch(vec3 from, vec3 dir, function f)
{
   vec3 x = from;
   int steps = MAX_STEPS;
   while (f(x) > 0.0001 && steps-- > 0)
     x += f(x)*dir;
   return length(x - from);
}
```

Distance fields

- To do raymarching we have to model the whole scene in the form of a single *distance field function* $f(\boldsymbol{x})$.

- It turns out that distance fields let you model quite complicated shapes rather naturally.

Quiz

- Devise a distance function for a sphere of radius 1 centered at (0, 0, 0).

Quiz

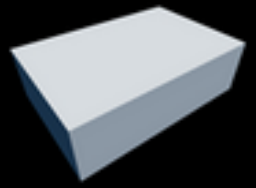
- Devise a distance function for a sphere of radius 1 centered at (0, 0, 0).

```
float sphere(vec3 x) {
    return length(x) - 1;
}
```

Sphere - signed

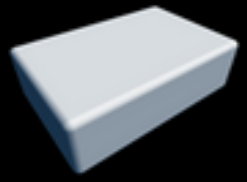
```
float sdSphere( vec3 p, float s )
{
  return length(p)-s;
}
```

Box - unsigned



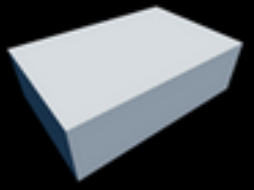
```
float udBox( vec3 p, vec3 b )
{
  return length(max(abs(p)-b,0.0));
}
```

Round Box - unsigned



```
float udRoundBox( vec3 p, vec3 b, float r )
{
  return length(max(abs(p)-b,0.0))-r;
}
```

Box - signed

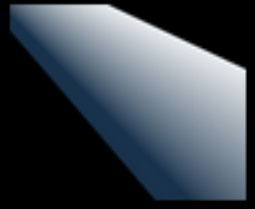


```
float sdBox( vec3 p, vec3 b )
{
  vec3 d = abs(p) - b;
  return min(max(d.x,max(d.y,d.z)),0.0) +
         length(max(d,0.0));
}
```

Torus - signed

```
float sdTorus( vec3 p, vec2 t )
{
  vec2 q = vec2(length(p.xz)-t.x,p.y);
  return length(q)-t.y;
}
```

Cylinder - signed



```
float sdCylinder( vec3 p, vec3 c )
{
  return length(p.xz-c.xy)-c.z;
}
```

Cone - signed

```
float sdCone( vec3 p, vec2 c )
{
    // c must be normalised
    float q = length(p.xy);
    return dot(c,vec2(q,p.z));
}
```

Plane - signed

```
float sdPlane( vec3 p, vec4 n )
{
  // n must be normalised
  return dot(p,n.xyz) + n.w;
}
```

 http://iquilezles.org/www/articles/distfunctions/distfunctions.htm

See also: http://www.viz.tamu.edu/faculty/ergun/research/implicitmodeling/papers/sm99.pdf

Quiz

- Given two distance functions corresponding to some shapes, devise a distance function that corresponds to the union of these shapes:

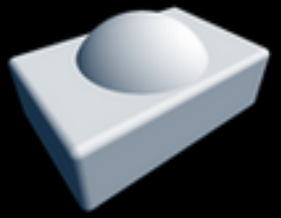
Quiz

- Given two distance functions corresponding to some shapes, devise a distance function that corresponds to the union of these shapes:

```
float union(vec3 x) {
      return min(f1(x), f2(x));
}
```

Constructive Solid Geometry

Union

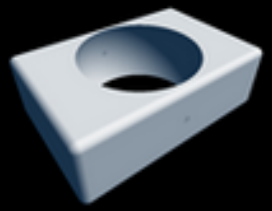


```
float opU( float d1, float d2 )
{
    return min(d1,d2);
}
```

Intersection

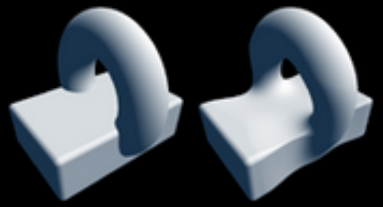
```
float opI( float d1, float d2 )
{
    return max(d1,d2);
}
```

Subtraction



```
float opS( float d1, float d2 )
{
    return max(-d1,d2);
}
```

Blend



```
float opBlend( vec3 p )
{
    float d1 = primitiveA(p);
    float d2 = primitiveB(p);
    return smin( d1, d2 );
}
```

http://iquilezles.org/www/articles/distfunctions/distfunctions.htm

Quiz

- Given a distance function describing a shape centered at (0, 0, 0). Devise a distance function for the same shape, centered at (1, 1, 1):

Quiz

- Given a distance function describing a shape centered at (0, 0, 0). Devise a distance function for the same shape, centered at (1, 1, 1):

```
float translate(vec3 x) {
    return f(x - vec3(1, 1, 1))
}
```

Arbitrary transformations

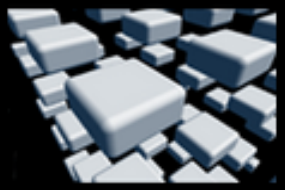
Rotation/Translation

```
vec3 opTx( vec3 p, mat4 m )
{
    vec3 q = invert(m)*p;
    return primitive(q);
}
```

Scale

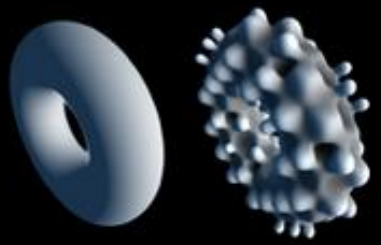
```
float opScale( vec3 p, float s )
{
    return primitive(p/s)*s;
}
```

Repetition



```
float opRep( vec3 p, vec3 c )
{
    vec3 q = mod(p,c)-0.5*c;
    return primitve( q );
}
```

Displacement

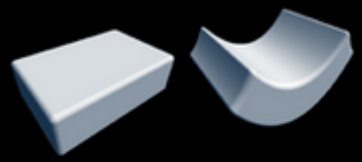


```
float opDisplace( vec3 p )
{
    float d1 = primitive(p);
    float d2 = displacement(p);
    return d1+d2;
}
```

Twist

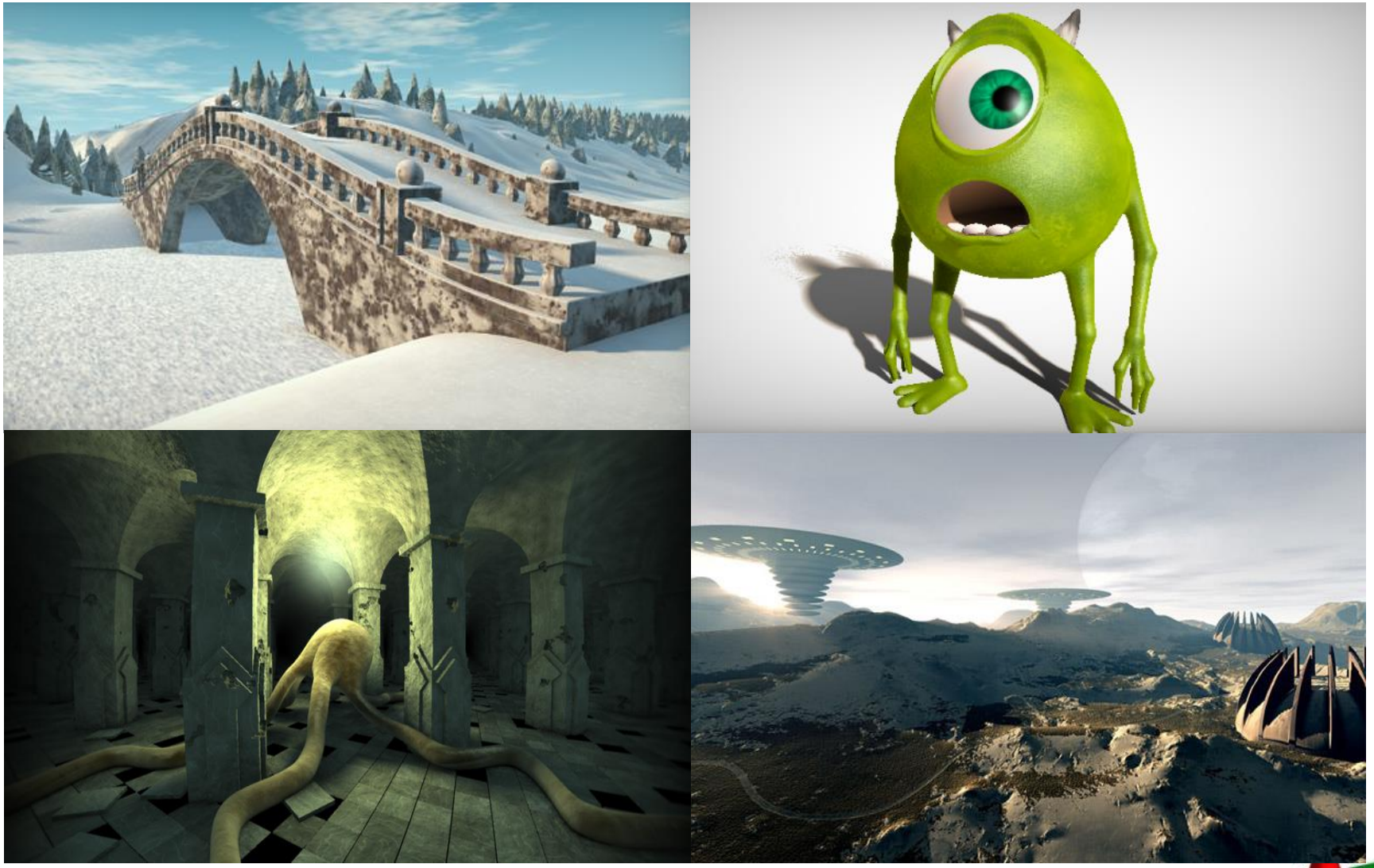
```
float opTwist( vec3 p )
{
    float c = cos(20.0*p.y);
    float s = sin(20.0*p.y);
    mat2   m = mat2(c,-s,s,c);
    vec3   q = vec3(m*p.xz,p.y);
    return primitive(q);
}
```

Cheap Bend



```
float opCheapBend( vec3 p )
{
    float c = cos(20.0*p.y);
    float s = sin(20.0*p.y);
    mat2   m = mat2(c,-s,s,c);
    vec3   q = vec3(m*p.xy,p.z);
    return primitive(q);
}
```

http://iquilezles.org/www/articles/distfunctions/distfunctions.htm



Next time

- Raycasting
- Raytracing
- Raymarching / Sphere tracing

- Rendering equation solvers
  - Radiosity, Photon mapping, Path tracing