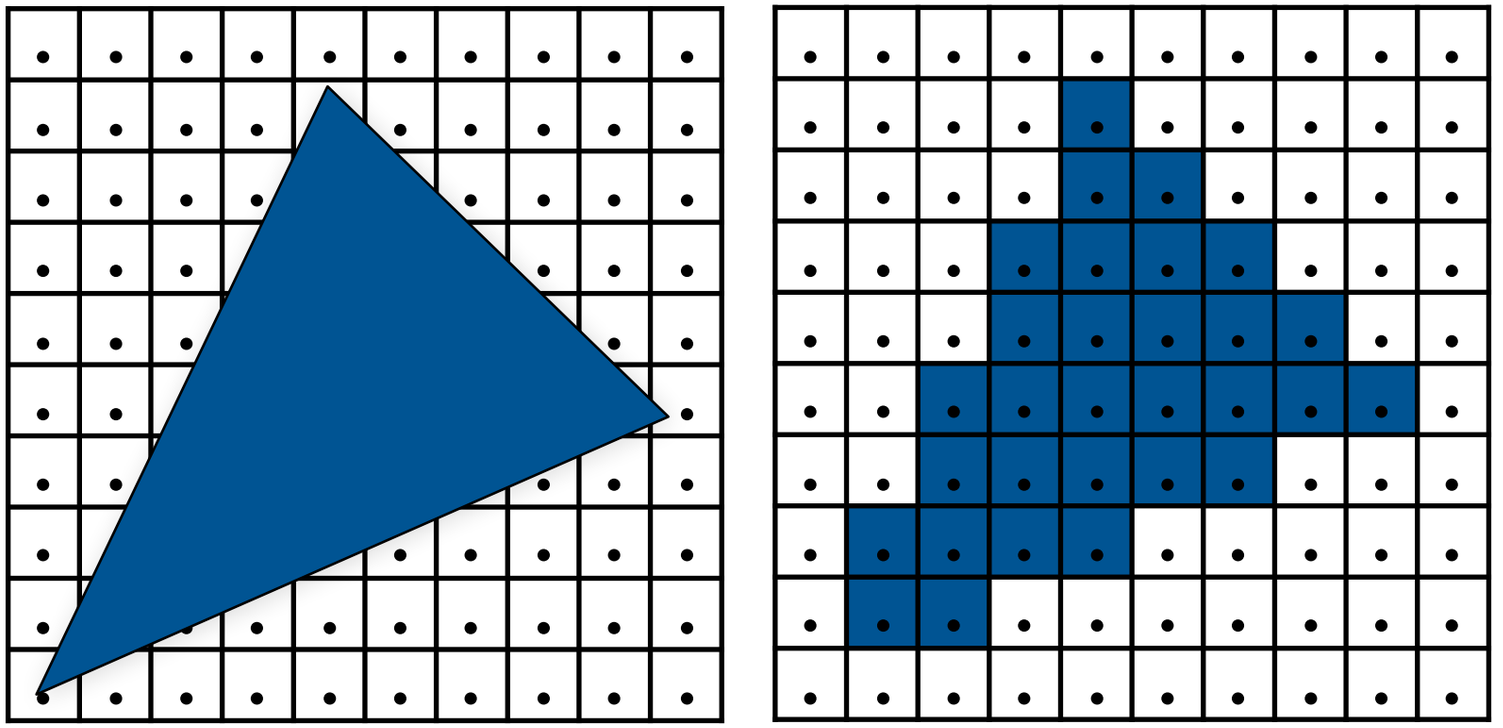
ECSE 446/546

IMAGE SYNTHESIS

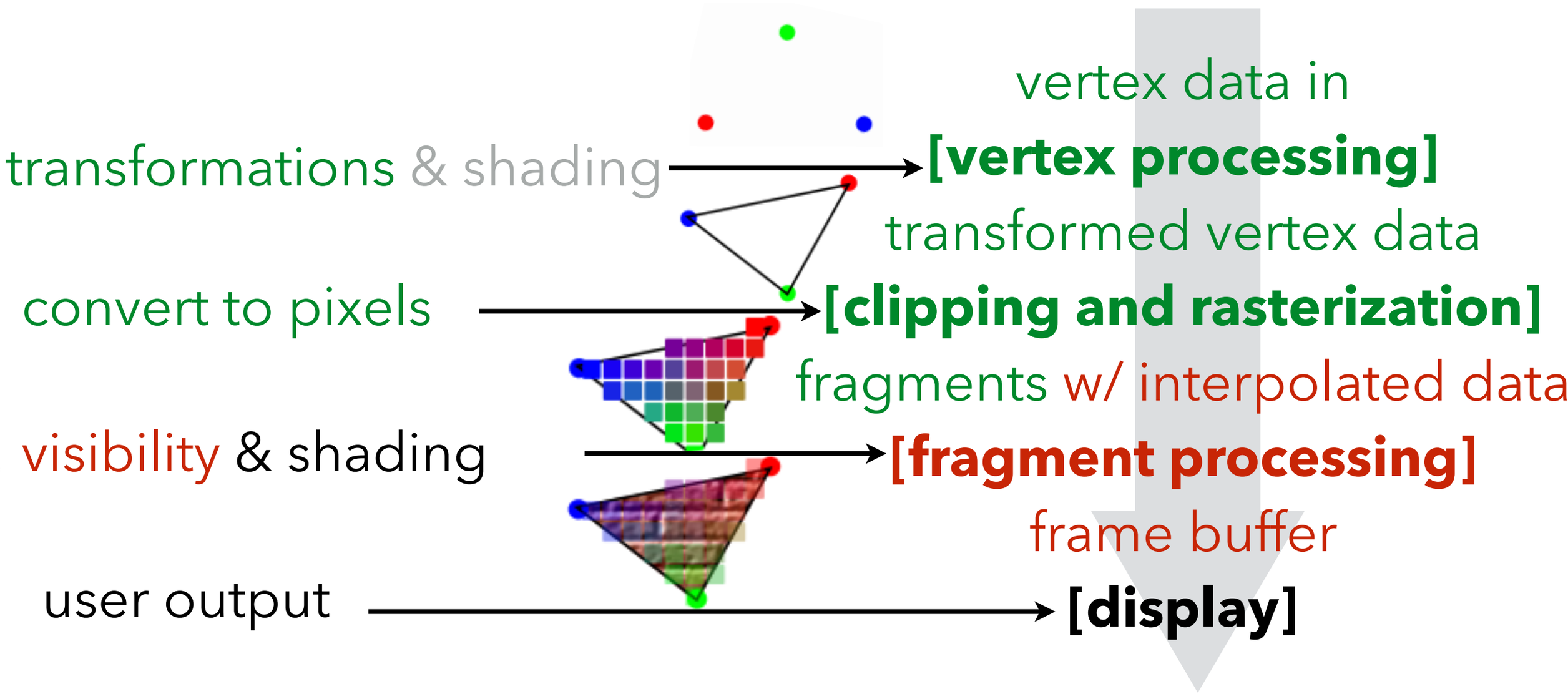


SYSTEMS 1 – RASTERIZATION

Prof. Derek Nowrouzezahrai

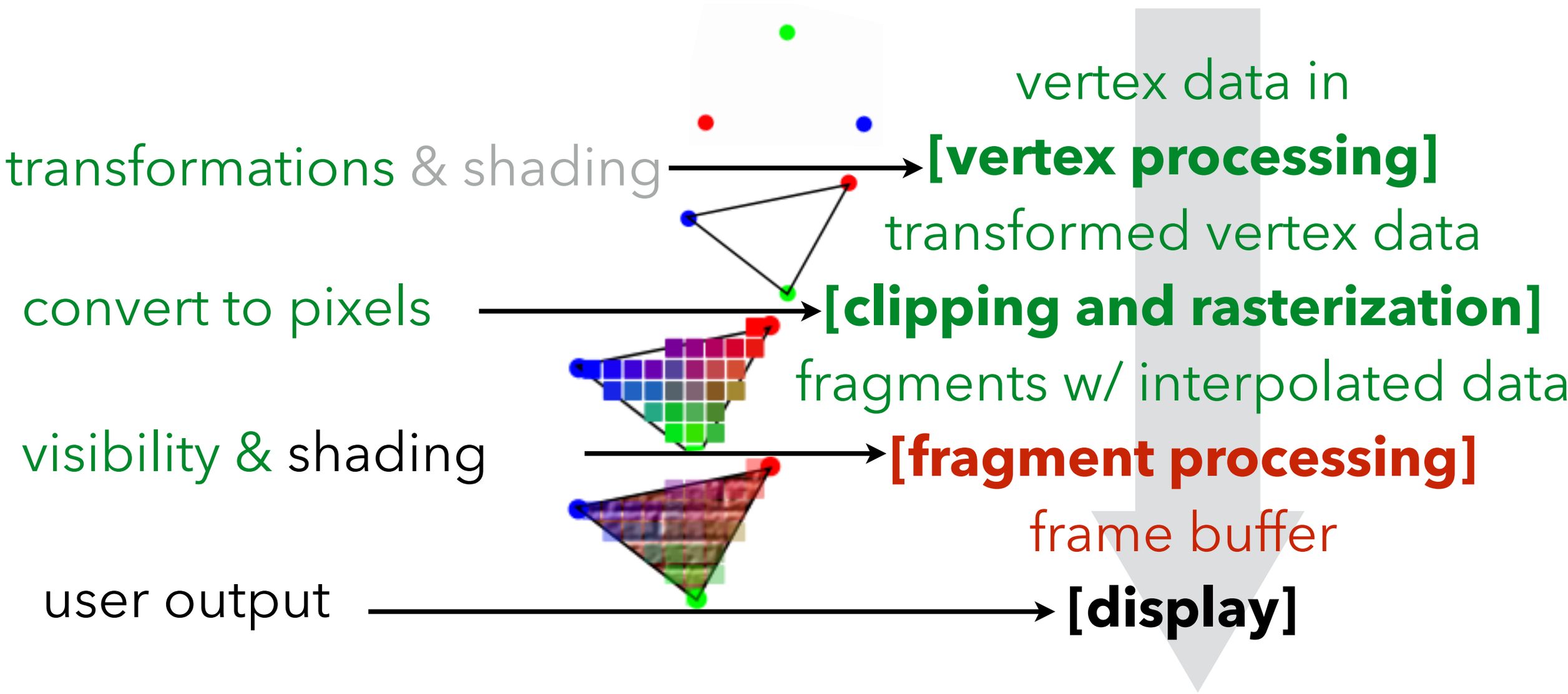
derek@cim.mcgill.ca

Programmable Rasterization Pipeline™



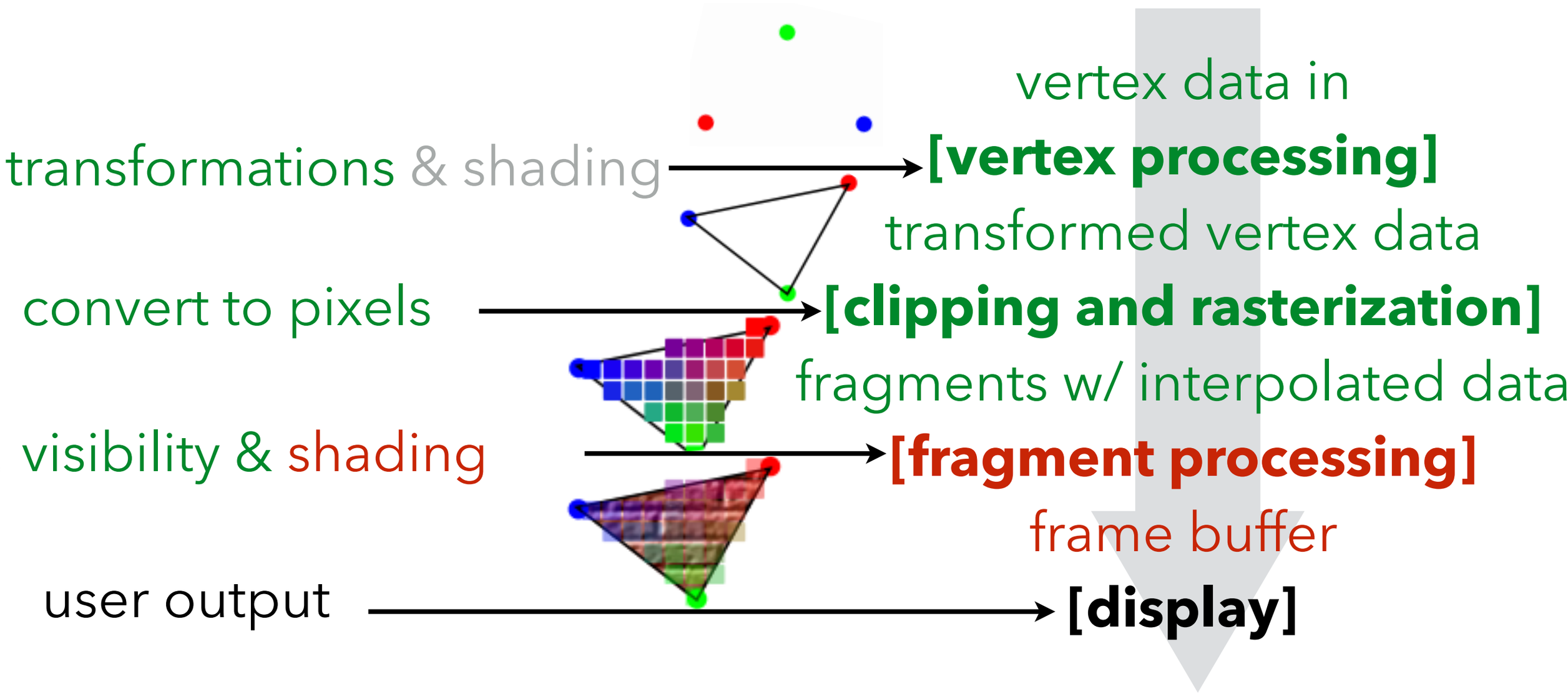
After a slide by Steve Marschner

Programmable Rasterization Pipeline™



After a slide by Steve Marschner

Programmable Rasterization Pipeline™



After a slide by Steve Marschner

Programmable Rasterization Pipeline™

```
for(each triangle)
    transform vertices into eye space
    project vertices to image space
    for(each pixel x,y)
        if(x,y in triangle)
            compute z
            if(z < zbuffer[x,y])
                zbuffer[x,y] = z
                framebuffer[x,y] = shade()
```

Flat Shading

Shade using the real normal of the triangle

- leads to constant shading and faceted appearance
- “truest” view of the mesh geometry

After a slide by Steve Marschner

[Foley et al.]

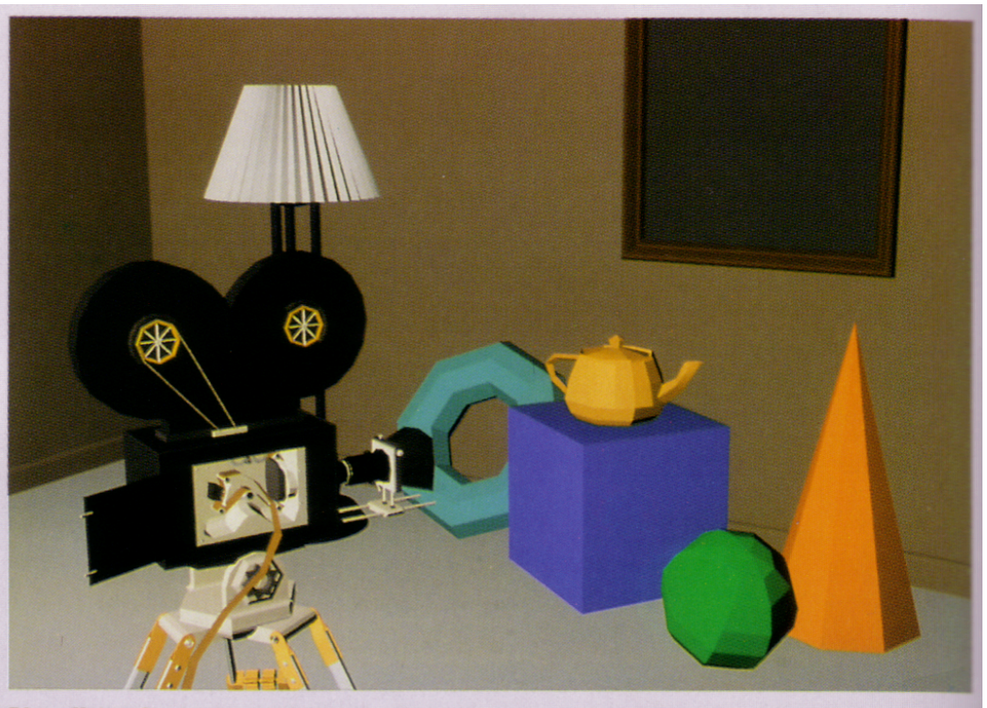


Plate II.29 *Shutterbug.* Individually shaded polygons with diffuse reflection (Sections 14.4.2 and 16.2.3). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

Pipeline for Flat Shading

Vertex stage (input: position / vtx; color and normal / tri)

- transform position & **per-face** normal (object to eye space)
- compute shaded color **per-face** using normal
- transform position (eye to screen space)

Rasterizer

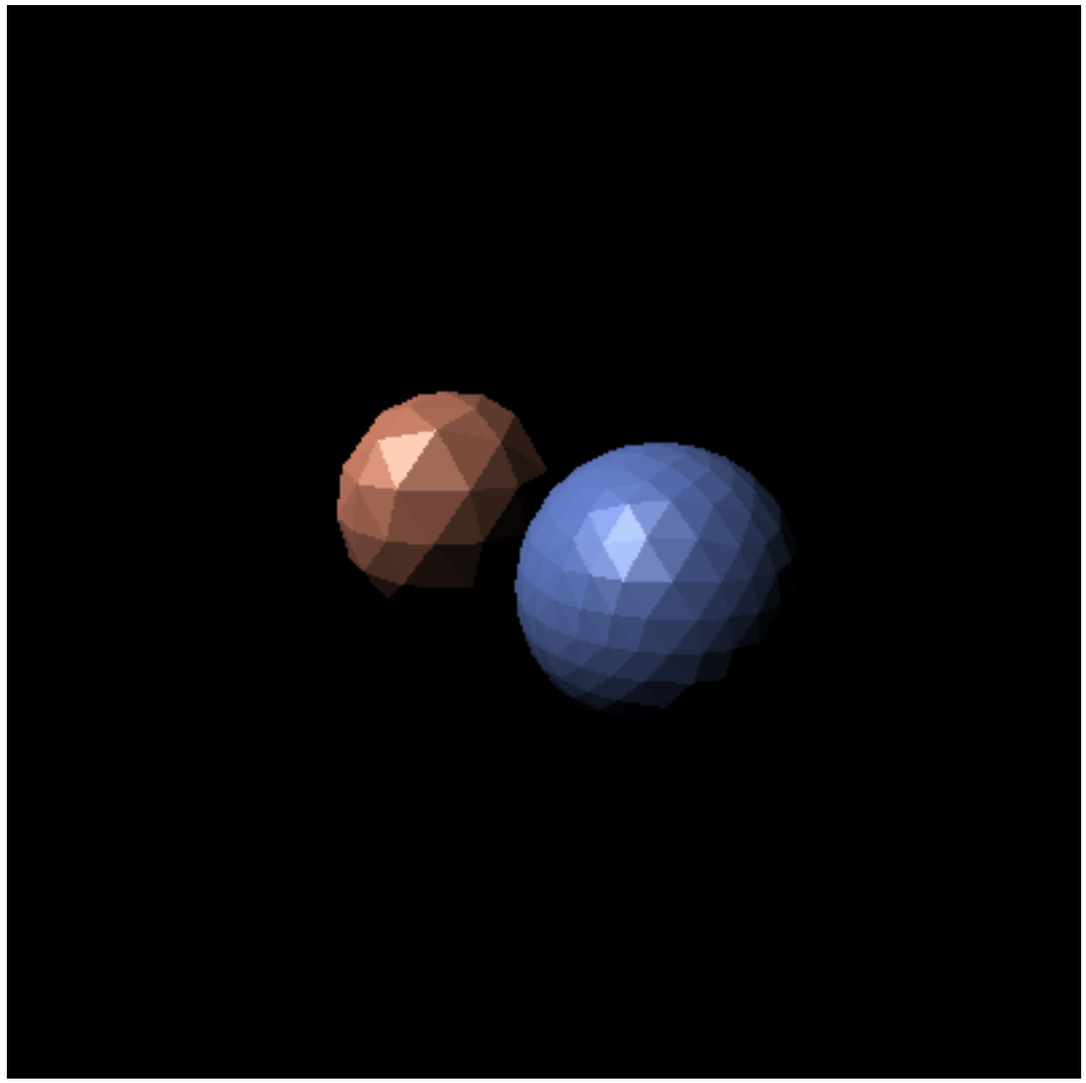
- interpolated parameters: z′ (screen z)
- pass through color

Fragment stage (output: color, z′)

- write to frame buffer if z-test passes:
  - interpolated z′ < current z′

After a slide by Steve Marschner

Result of Flat Shading Pipeline



After a slide by Steve Marschner

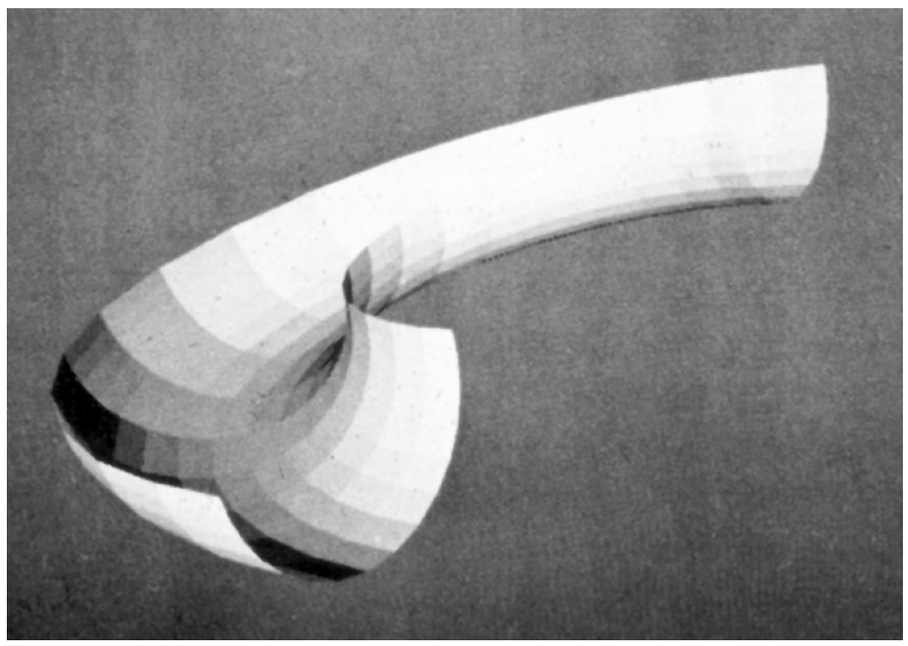
Gouraud Shading

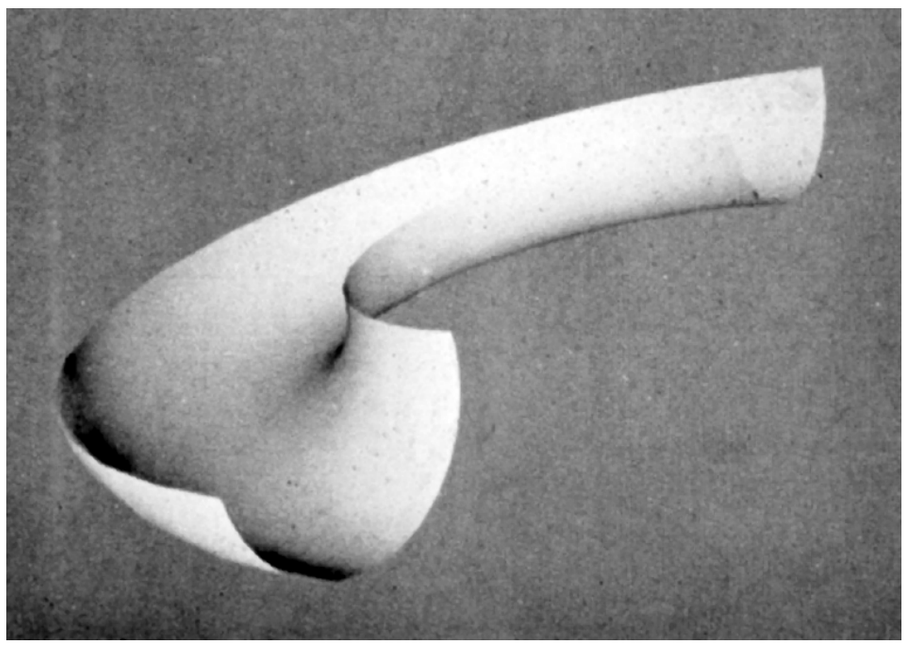
Often we're trying to draw smooth surfaces, so facets are an artifact

- compute colors at vertices w/ **vertex normals**
- interpolate colors across triangles

After a slide by Steve Marschner

- "Gouraud shading", "smooth shading"





[Gouraud thesis]

Gouraud Shading

Often we're trying to draw smooth surfaces, so facets are an artifact

- "Gouraud shading", "smooth shading"

After a slide by Steve Marschner

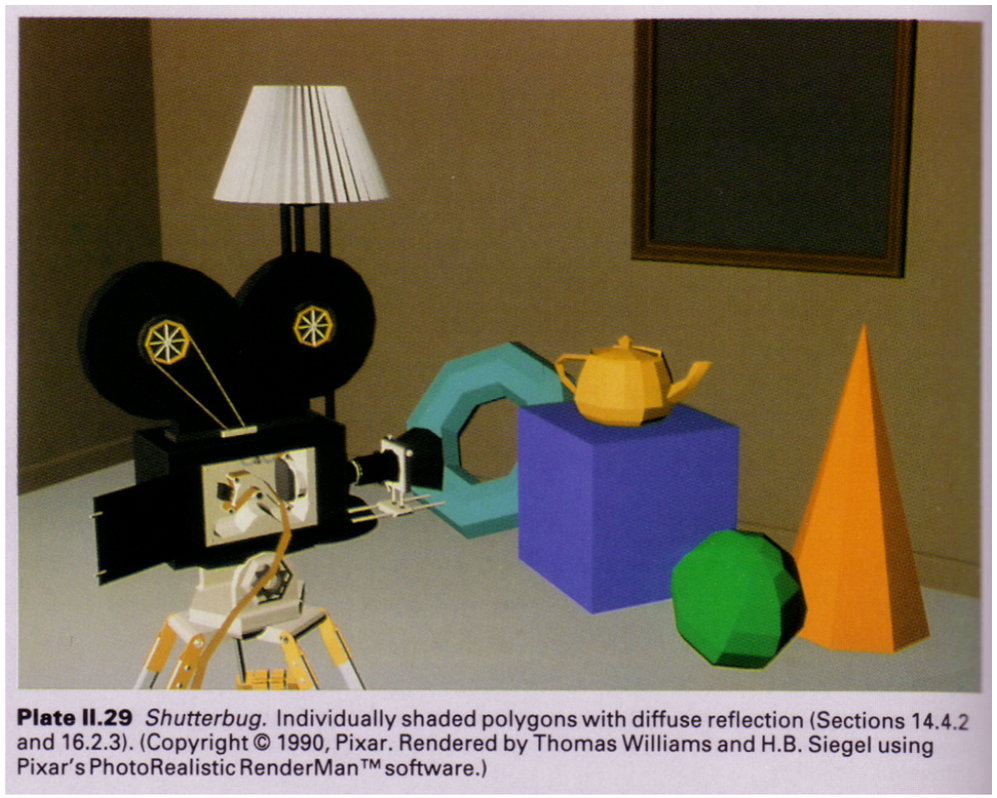


Plate II.29 *Shutterbug.* Individually shaded polygons with diffuse reflection (Sections 14.4.2 and 16.2.3). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

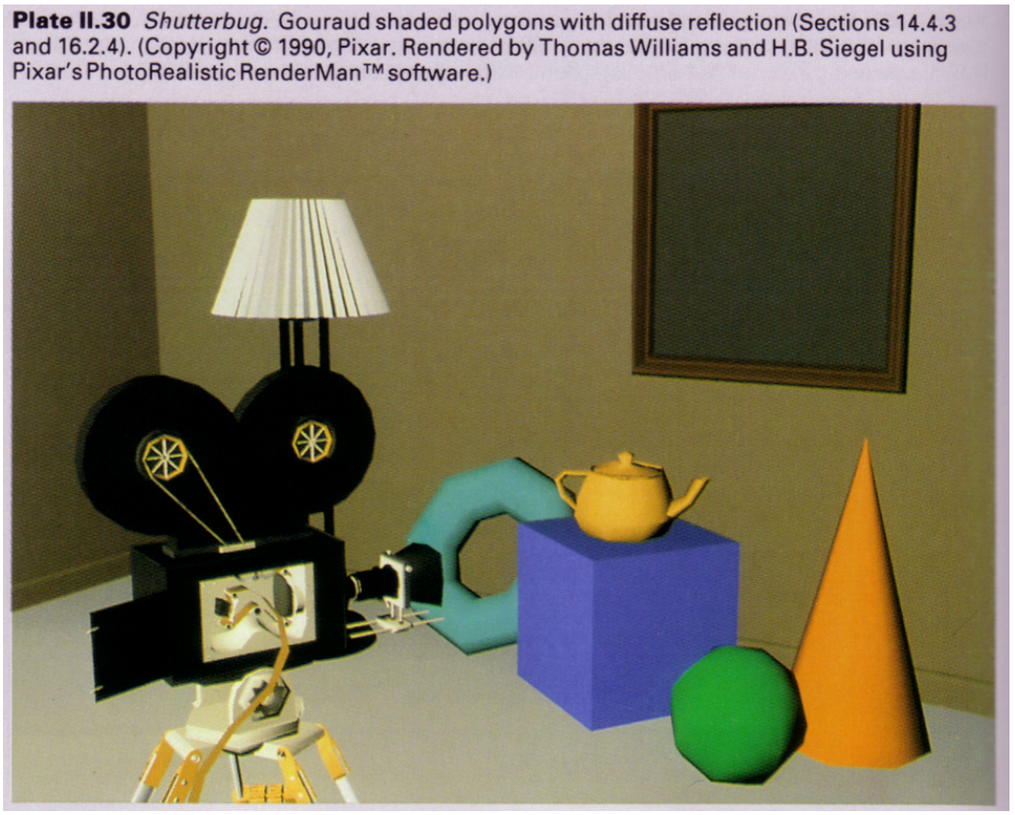


Plate II.30 *Shutterbug.* Gouraud shaded polygons with diffuse reflection (Sections 14.4.3 and 16.2.4). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

[Foley et al.]

Vertex Normals

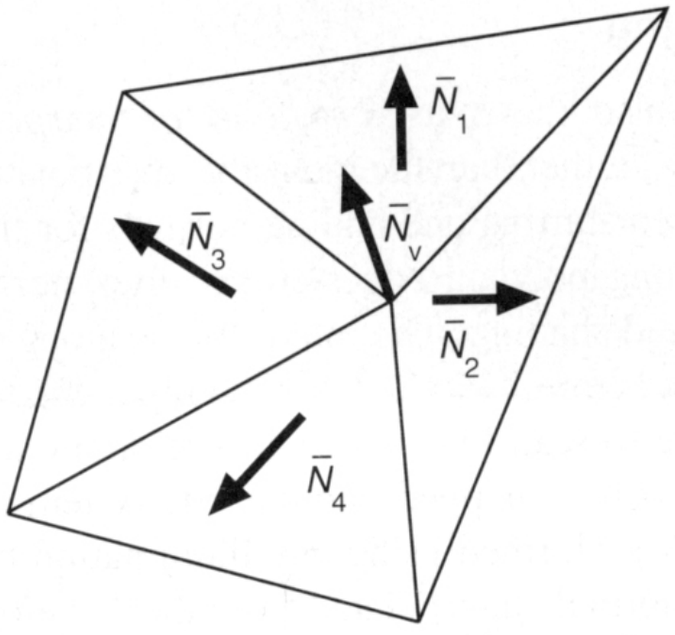
Gouraud shading requires per-vertex normals

- could obtain vertex normals by sampling the underlying geometry
  - e.g., spheres

Otherwise have to infer vertex normals from triangles

- simple scheme: average surrounding face normals

$$N_v = \frac{\sum_i N_i}{\|\sum_i N_i\|}$$



[Foley et al.]

After a slide by Steve Marschner

Pipeline for Gouraud Shading

Vertex stage (in: position, color & normal / vtx)

- transform position & normal (object to eye space)
- compute shaded color per vertex
- transform position (eye to screen space)

Rasterizer

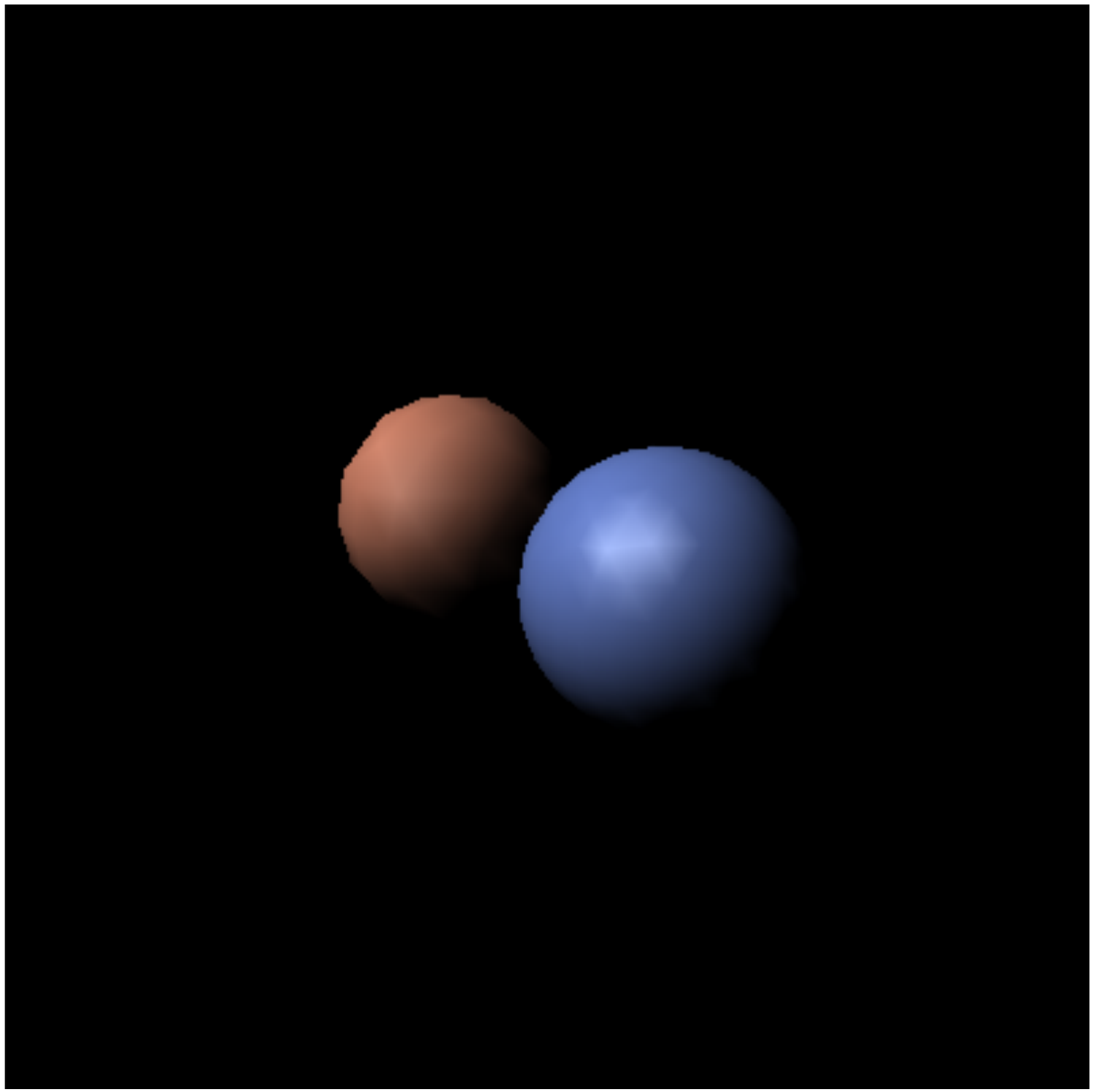
- interpolated parameters: z′ (screen z); **RGB** color

Fragment stage (output: color, z′)

- write to color plane only if z-test passes:
  - interpolated z′ < current z′

After a slide by Steve Marschner

Result of Gouraud Shading Pipeline



After a slide by Steve Marschner

Gouraud Shading – Specular/Glossy

Can apply Gouraud to any shading model

- it's just an interpolation method

- not great when shading variations are smaller than the underlying surface/triangle resolution

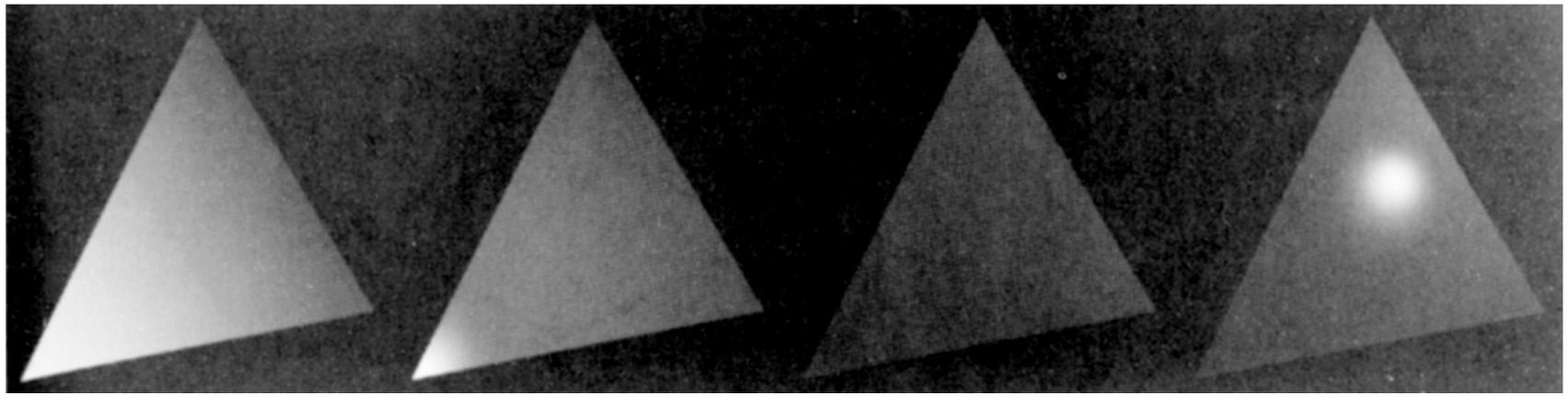
After a slide by Steve Marschner

- specular/glossy shading; "small" highlights

Phong Shading

Can increase shading quality by interpolating the normals per pixel

- just as easy as interpolating the color
  - but now, evaluate shading per pixel rather than per vertex (normalize the normal, first!)
  - in pipeline, this means we are moving illumination from the vertex processing stage to the fragment processing stage



[Foley et al.]

After a slide by Steve Marschner

Phong Shading

vertex data in

transformations & **shading** → **[vertex processing]**

transformed vertex data

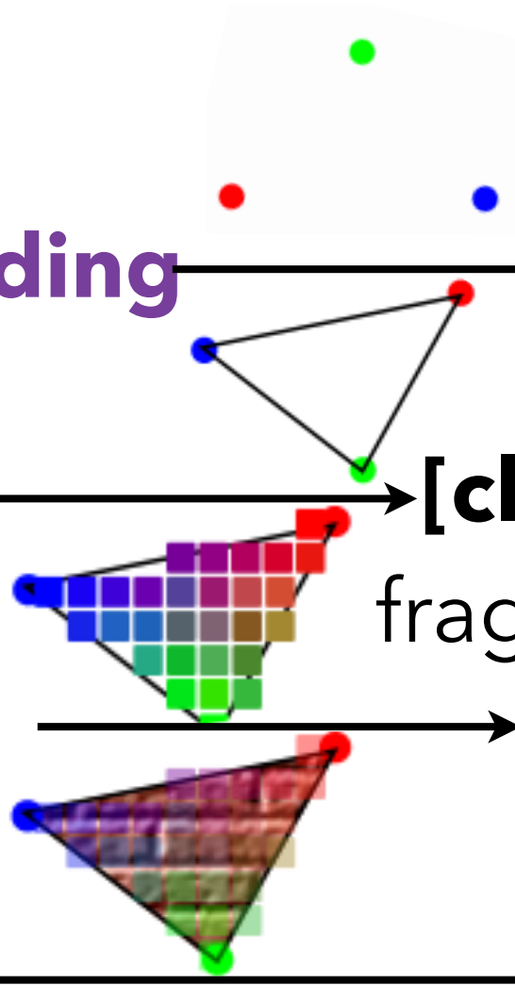
convert to pixels → **[clipping and rasterization]**

fragments w/ interpolated data

visibility & **shading** → **[fragment processing]**

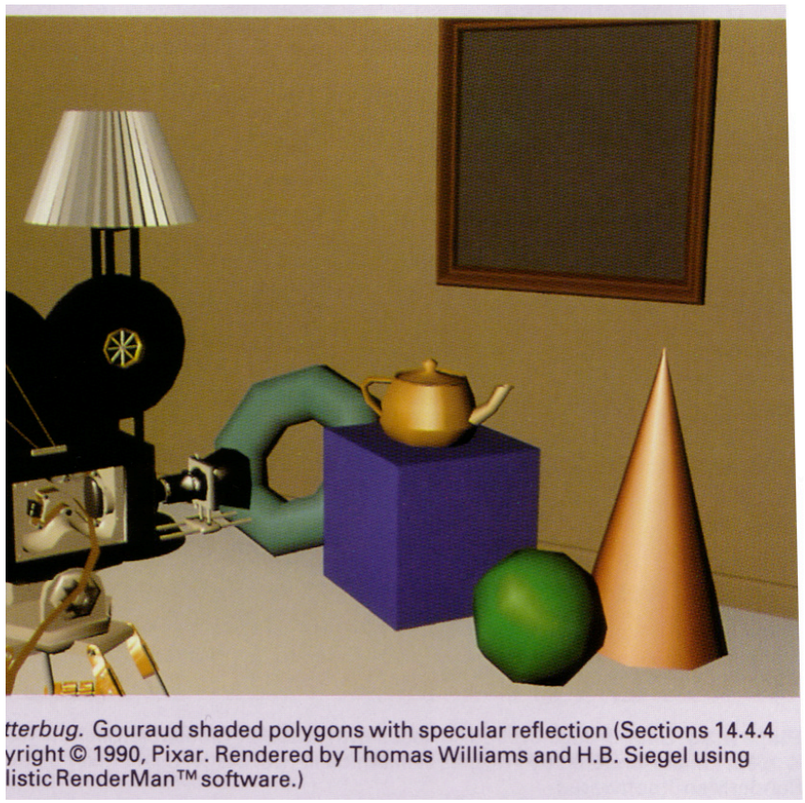
frame buffer

user output → **[display]**



Phong Shading

Bottom line: produces much better highlights



tterbug. Gouraud shaded polygons with specular reflection (Sections 14.4.4
yright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using
listic RenderMan™ software.)

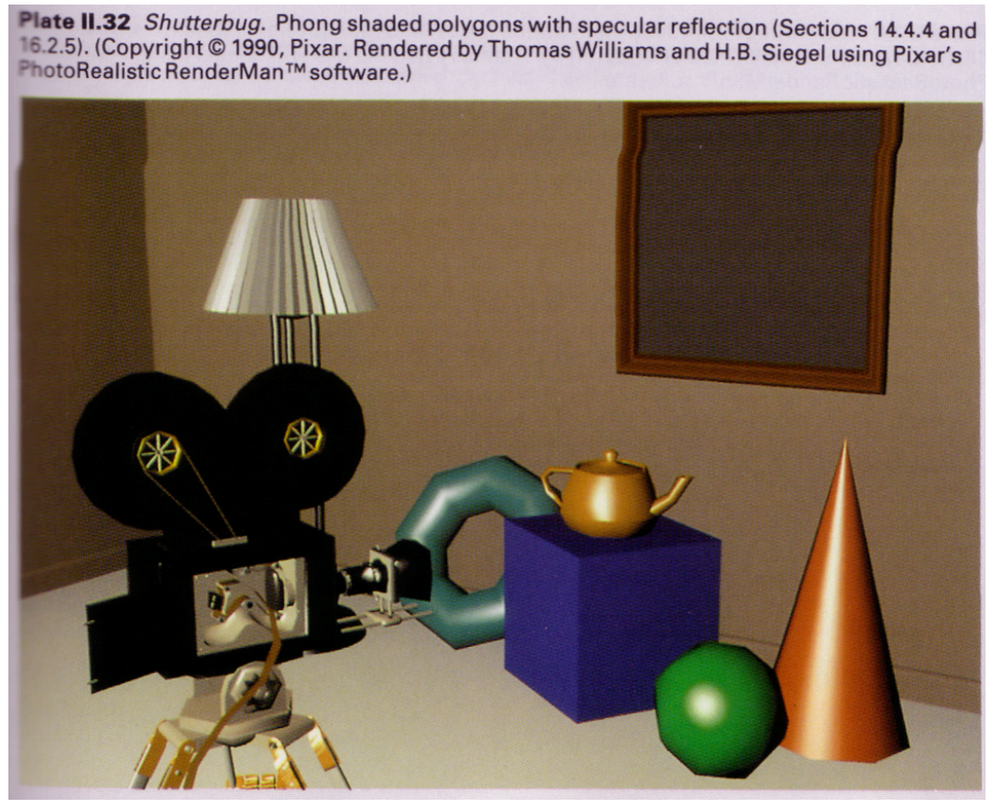


Plate II.32 *Shutterbug.* Phong shaded polygons with specular reflection (Sections 14.4.4 and 16.2.5). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

After a slide by Steve Marschner

[Foley et al.]

Pipeline for Phong shading

Vertex stage (input: position, color & normal / vtx)

- transform position and normal (object to eye space)
- transform position (eye to screen space)
- pass through color

Rasterizer

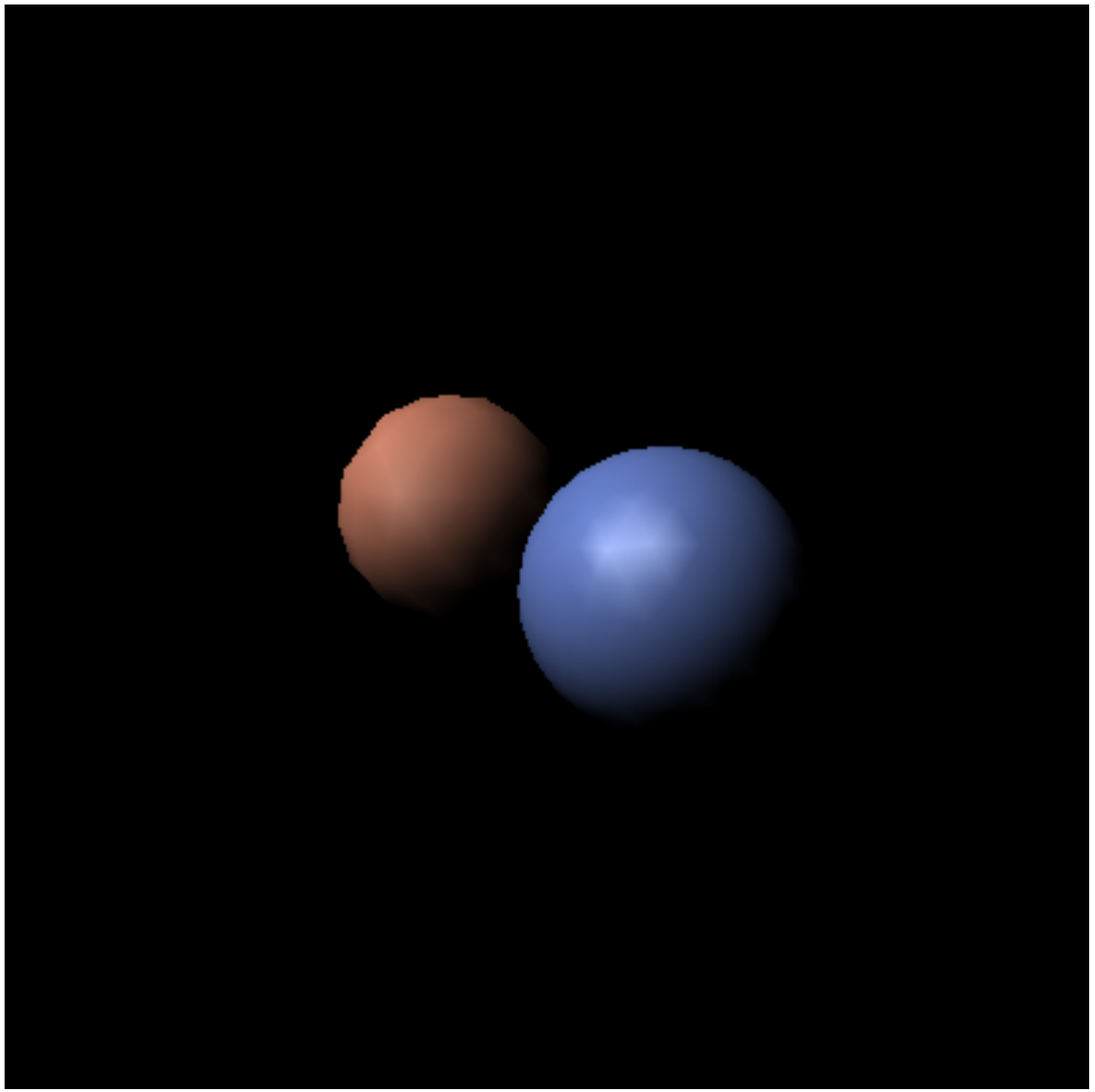
- interpolated parameters: z′ (screen z); **RGB** color; **x, y, z normal**

Fragment stage (output: color, z′)

- compute shading using interpolated color & normal
- write to color planes only if z-test passes
  - interpolated z′ < current z′

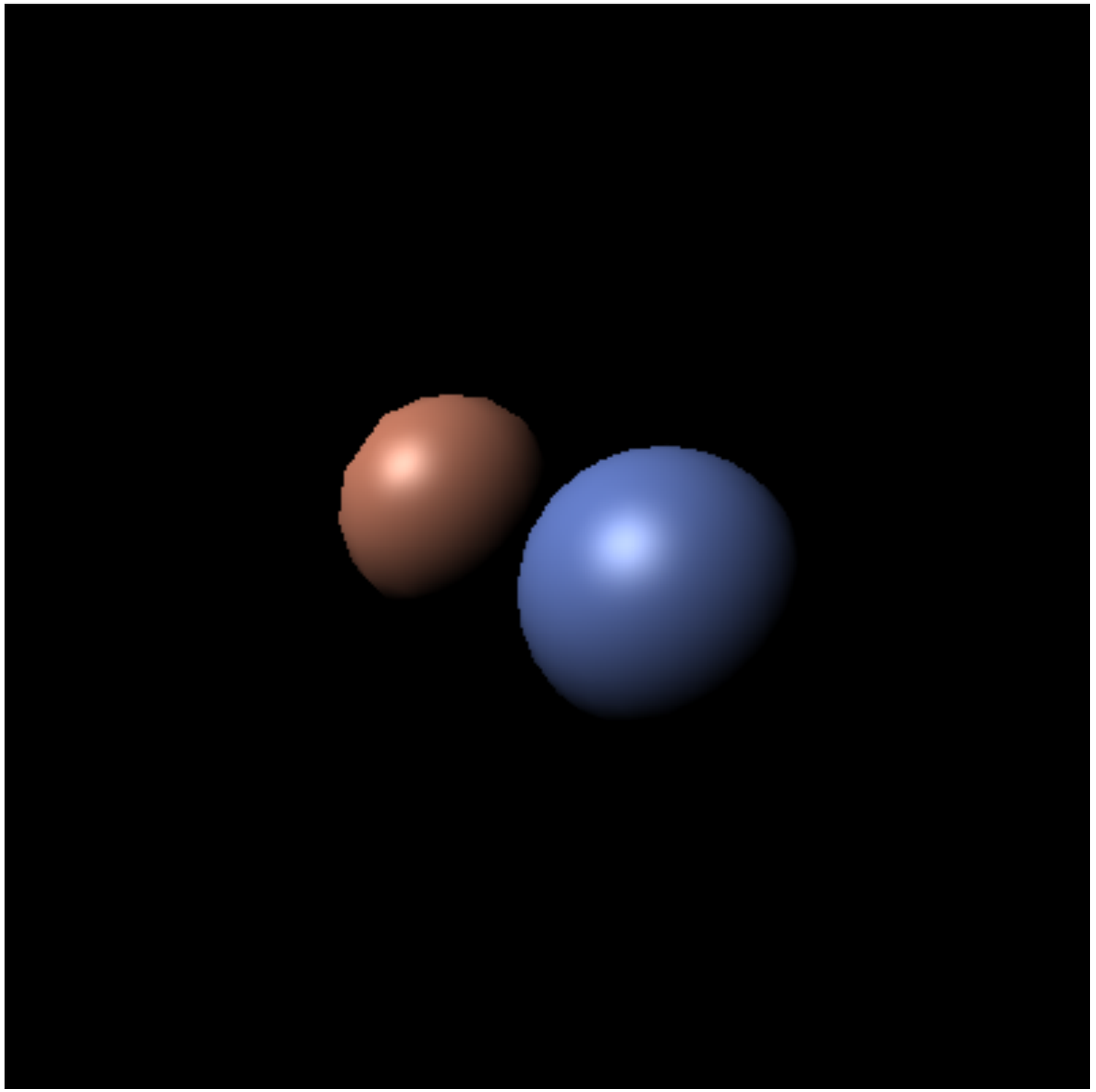
After a slide by Steve Marschner

Result of Gouraud Shading Pipeline



After a slide by Steve Marschner

Result of Phong Shading Pipeline

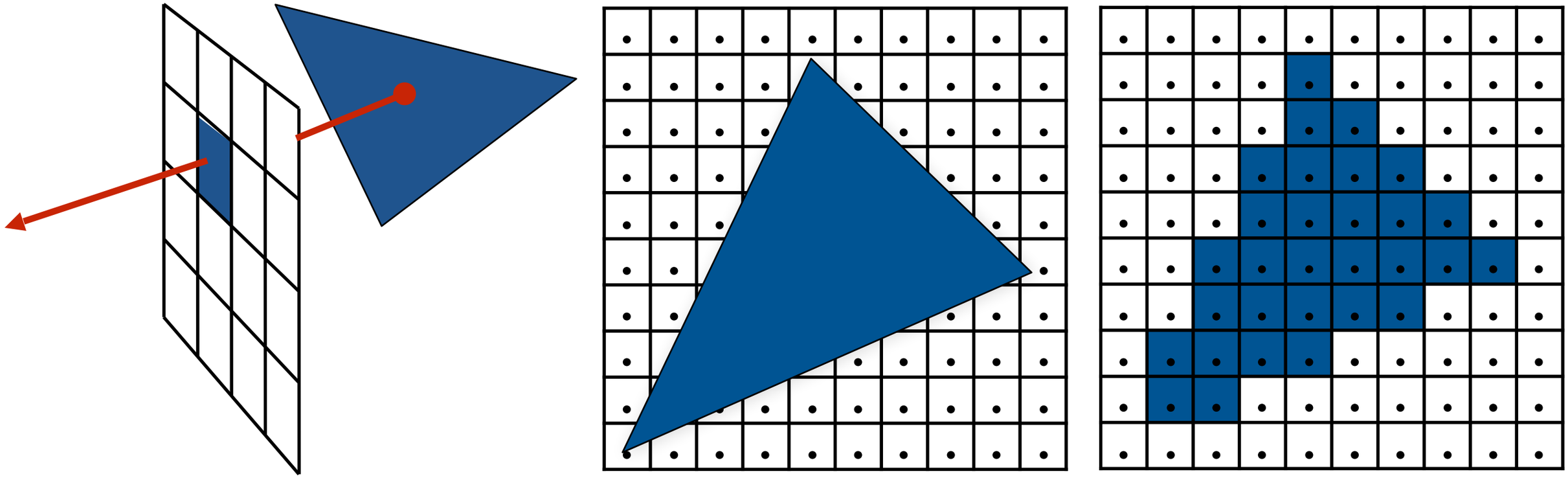


After a slide by Steve Marschner

Questions?

ECSE 446/546

IMAGE SYNTHESIS



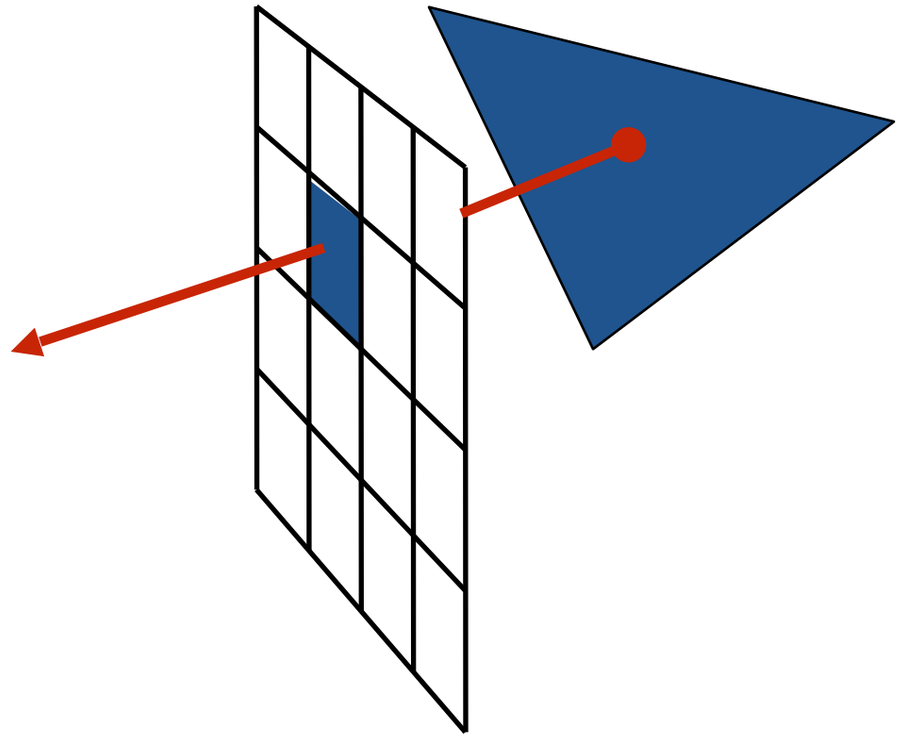
RENDERING SYSTEMS

Prof. Derek Nowrouzezahrai

derek@cim.mcgill.ca

ECSE 446/546

IMAGE SYNTHESIS



SYSTEMS 2 – RAY TRACING

Prof. Derek Nowrouzezahrai

derek@cim.mcgill.ca

Ray Tracing Today

Raytracing – Movie Production

Arnold Renderer

SOLIDANGLE

PIXAR's RenderMan

Hyperion

Surface Light Transport Assumptions

Geometric optics:

- no diffraction, no polarization, no interference

Light travels in a straight line (in a vacuum)

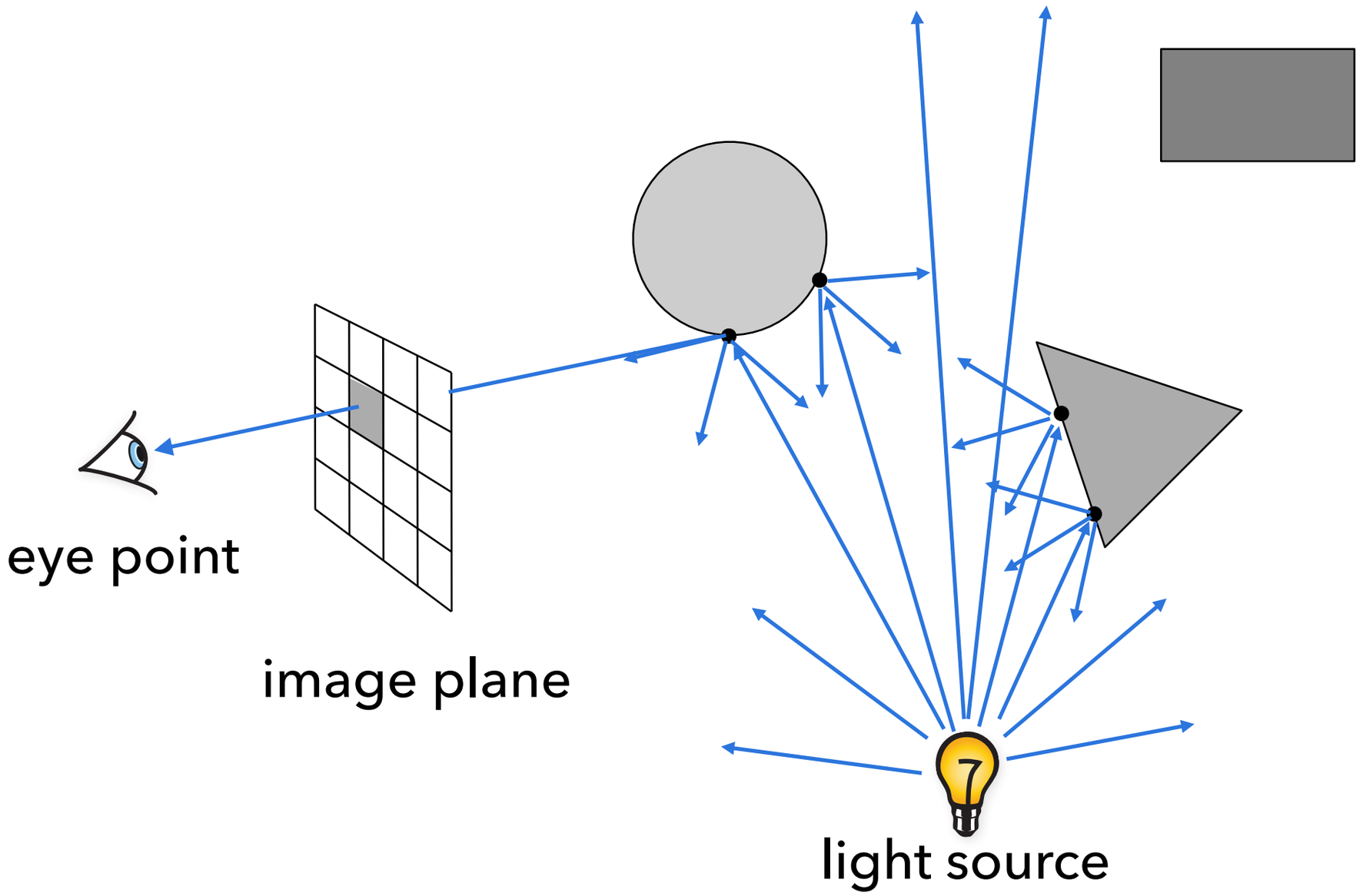
- no atmospheric scattering or refraction
- no gravity effects

Superposition

- no non-linear reflecting materials

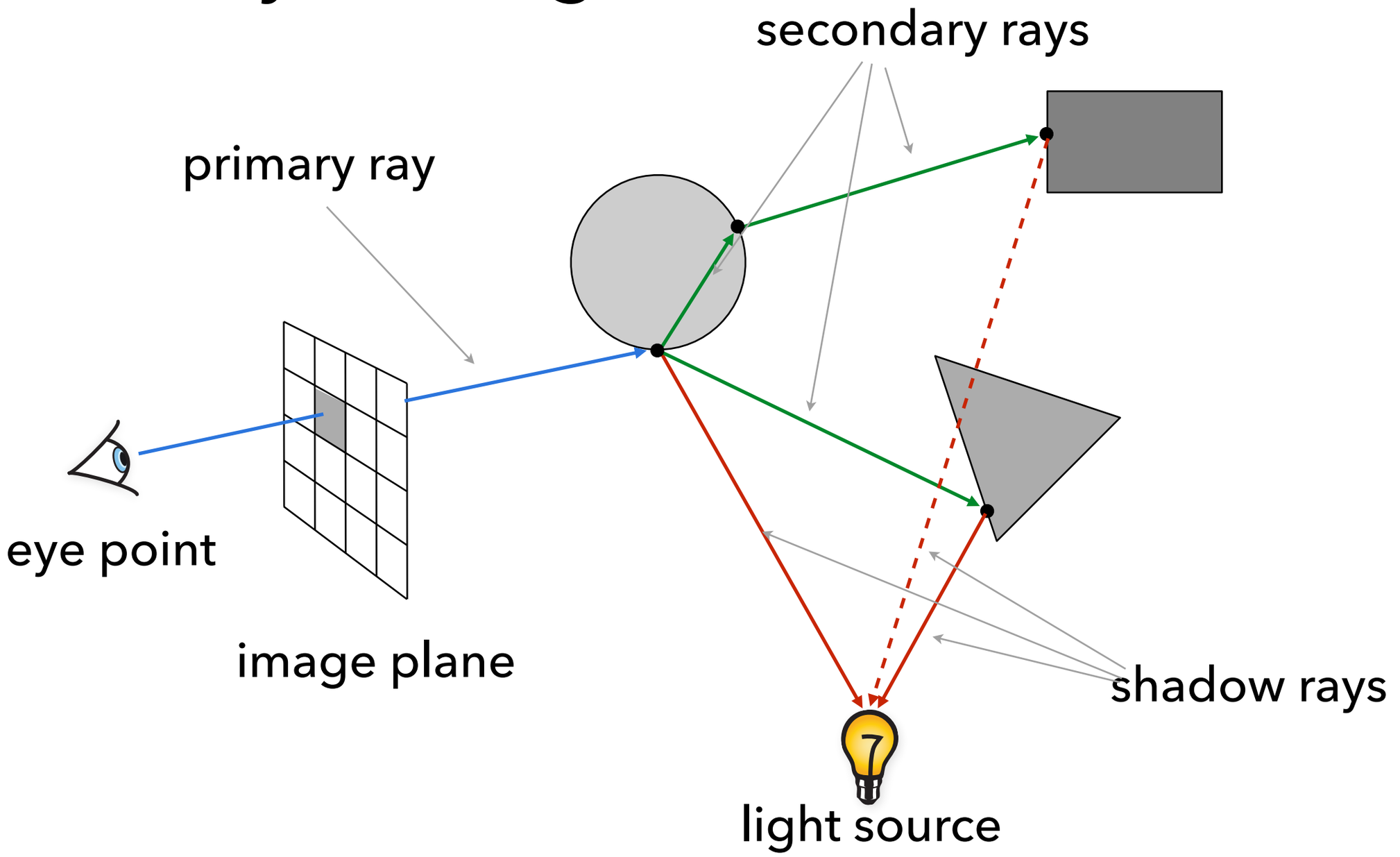
Ray Tracing – Overview

Forward Ray Tracing

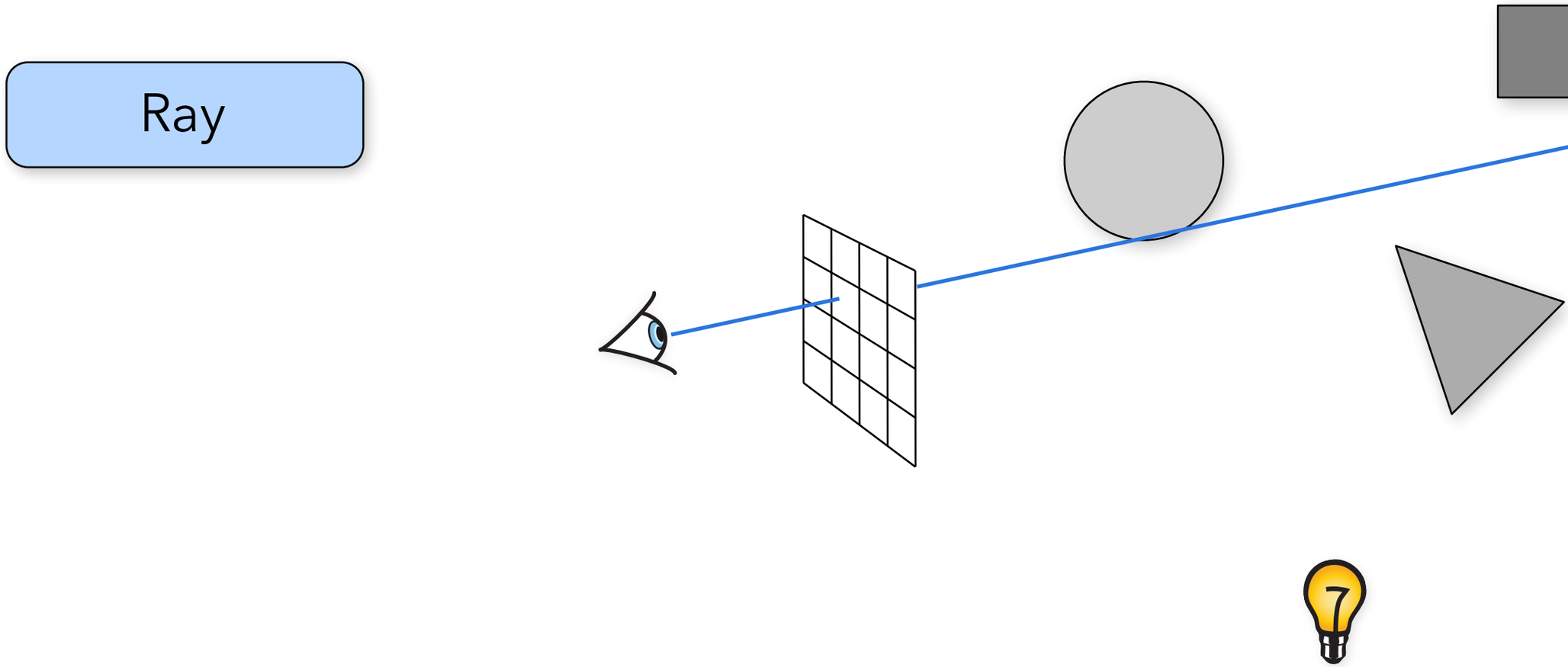


Ray Tracing – Overview

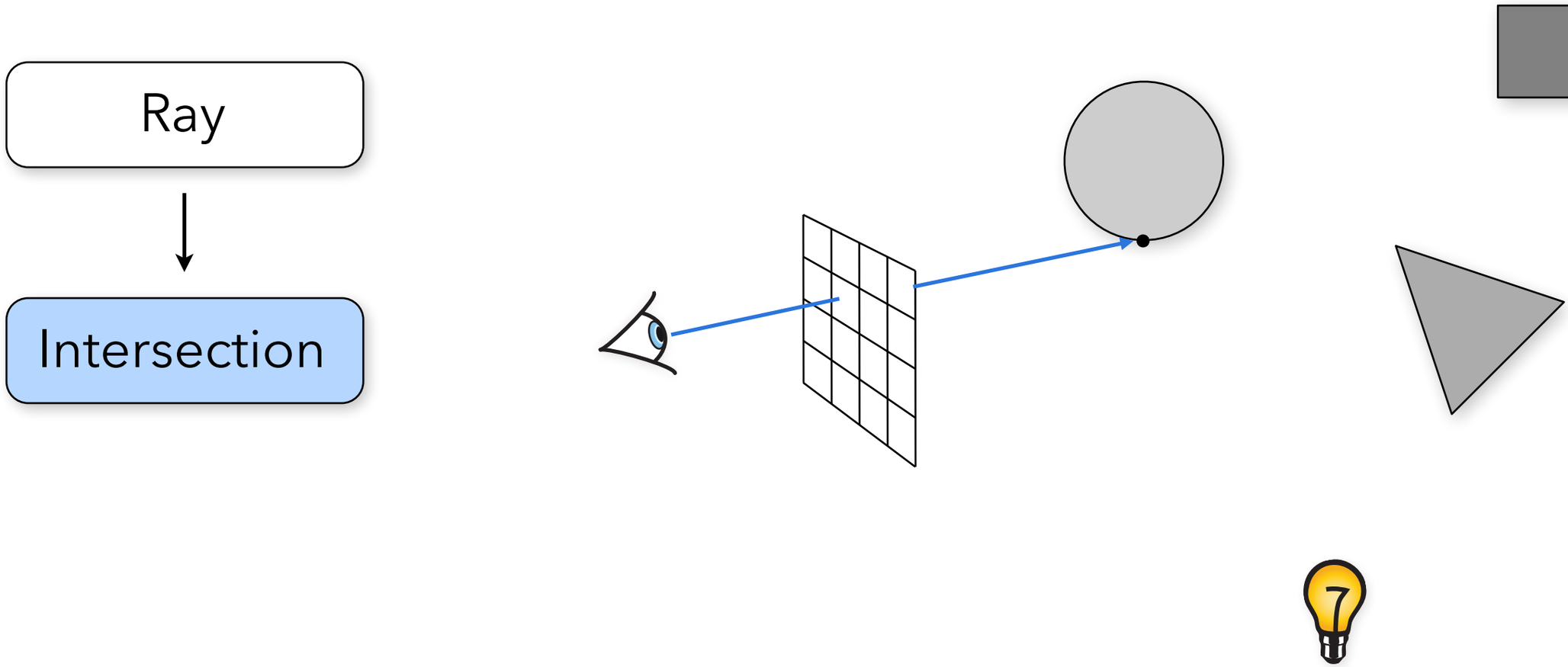
Backward Ray Tracing



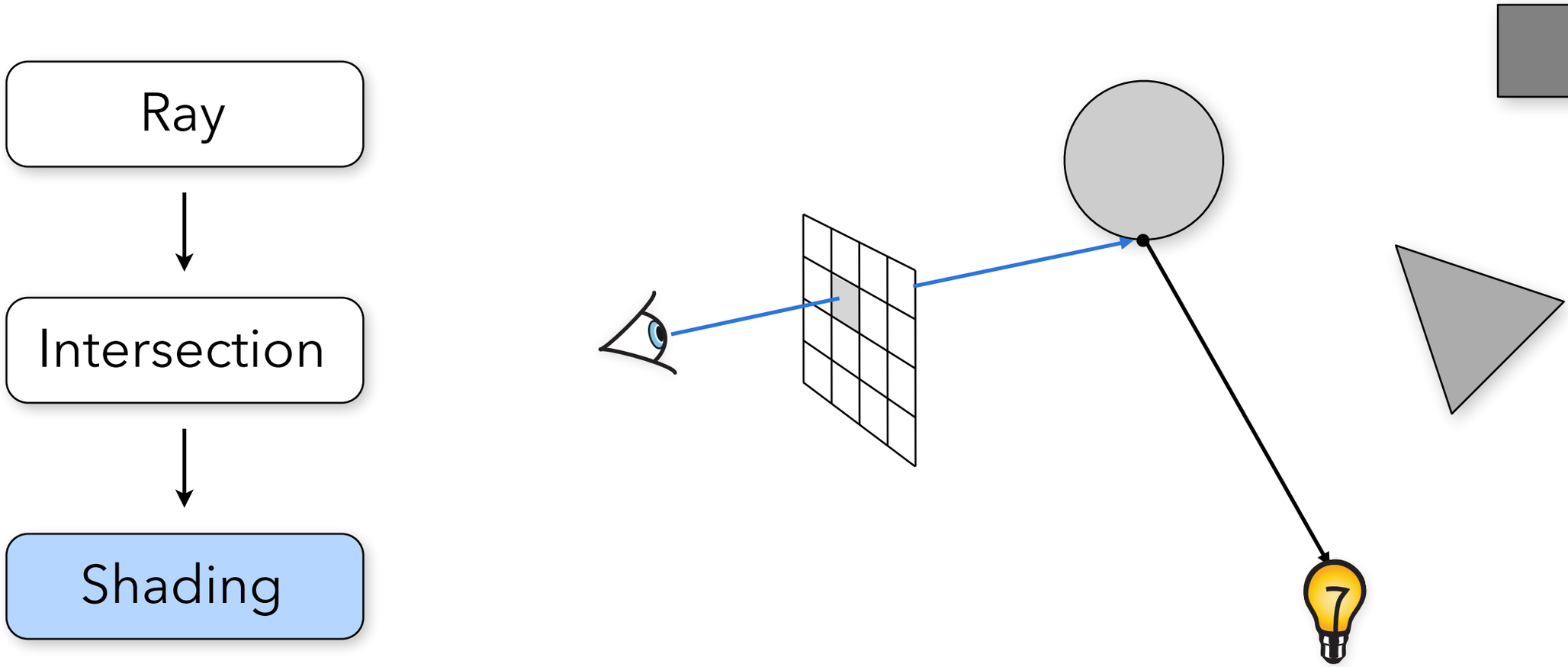
Basic Ray Tracing Pipeline



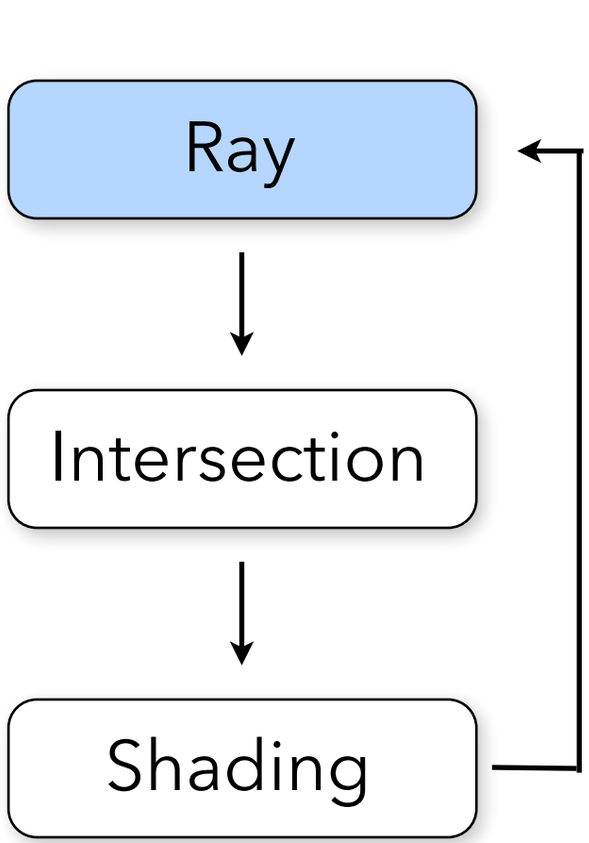
Basic Ray Tracing Pipeline

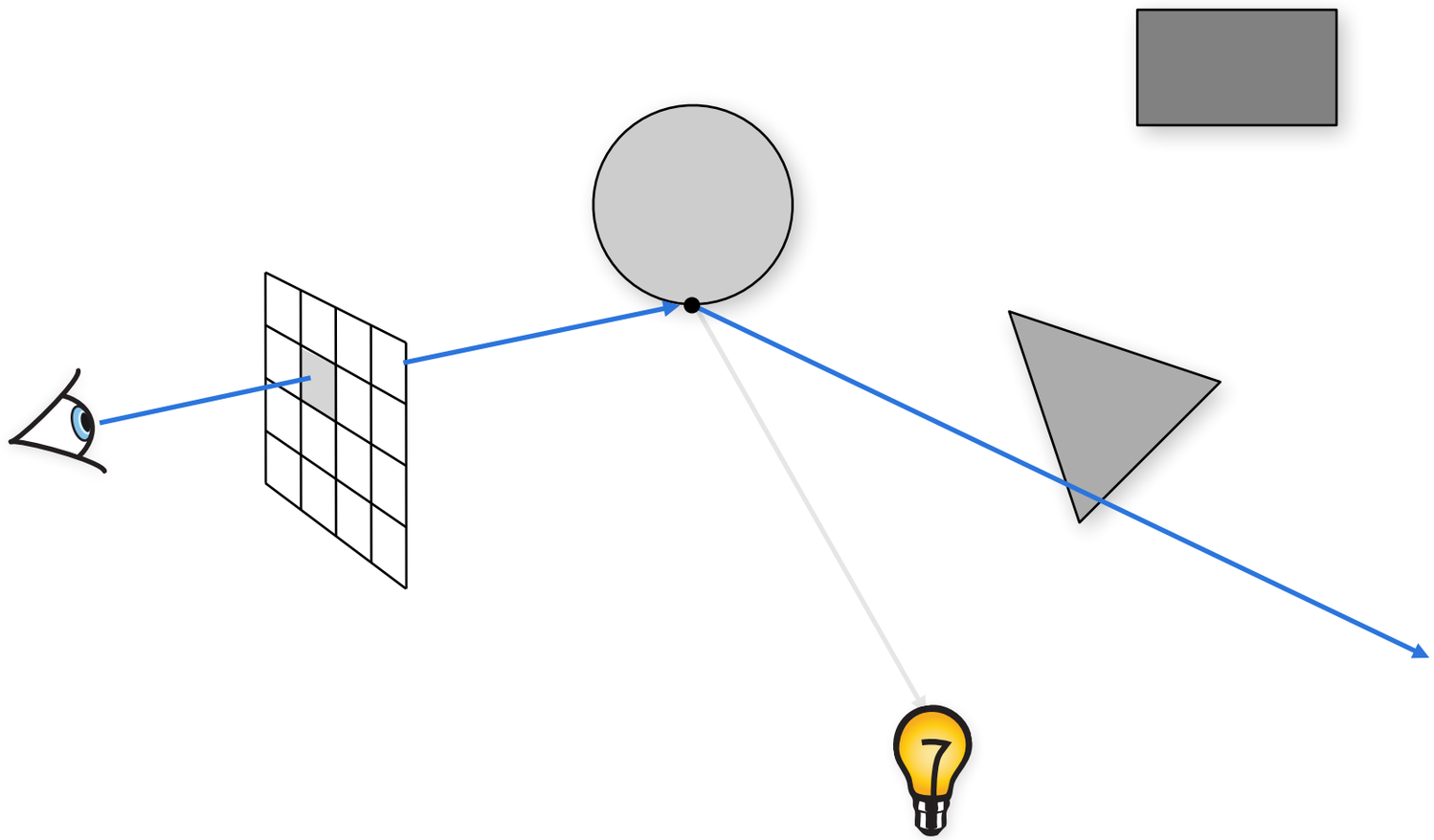


Basic Ray Tracing Pipeline

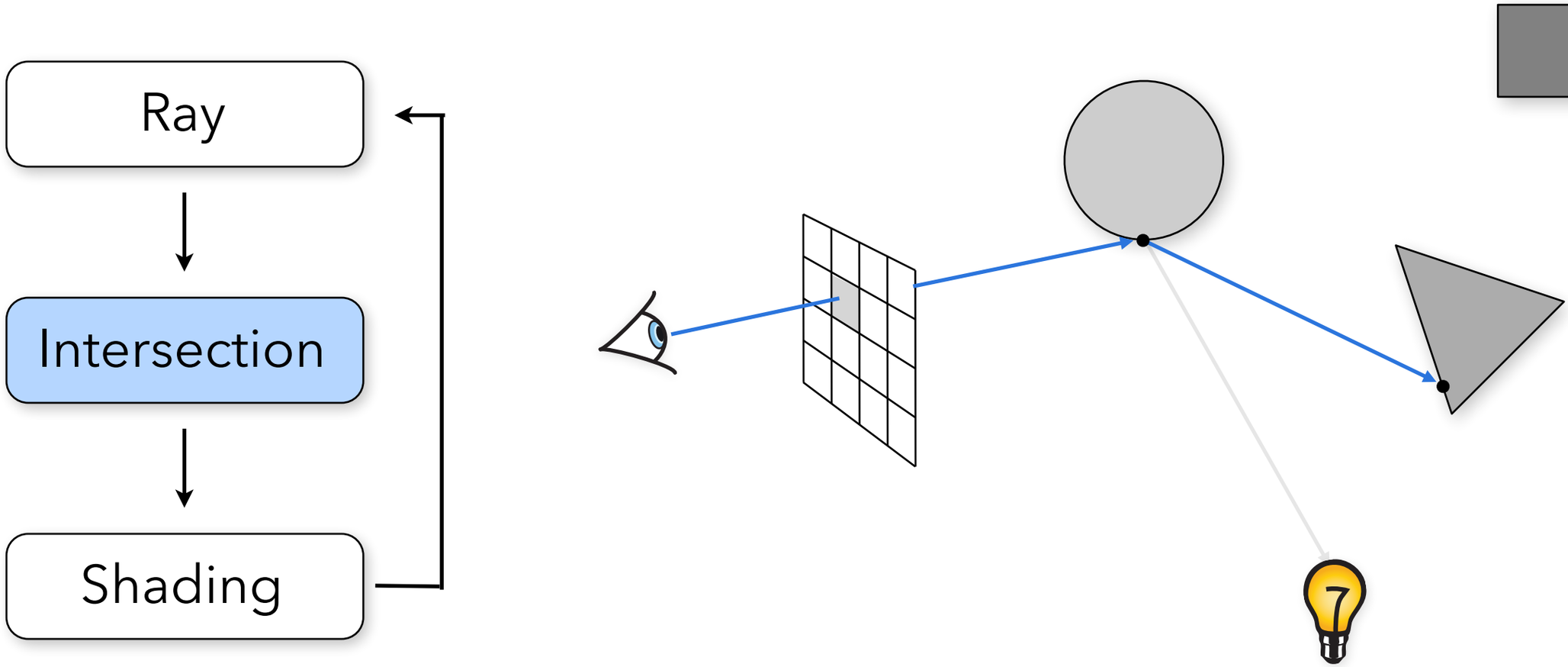


Basic Ray Tracing Pipeline

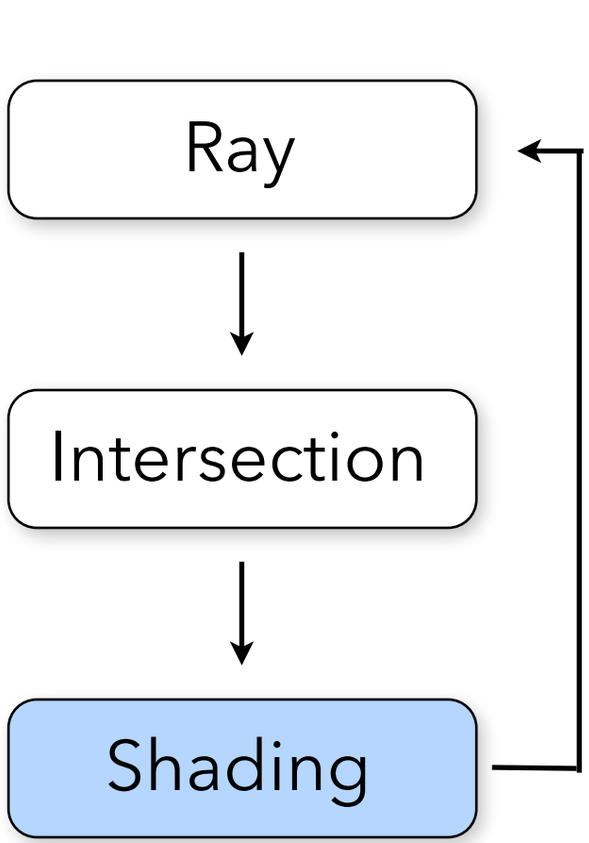


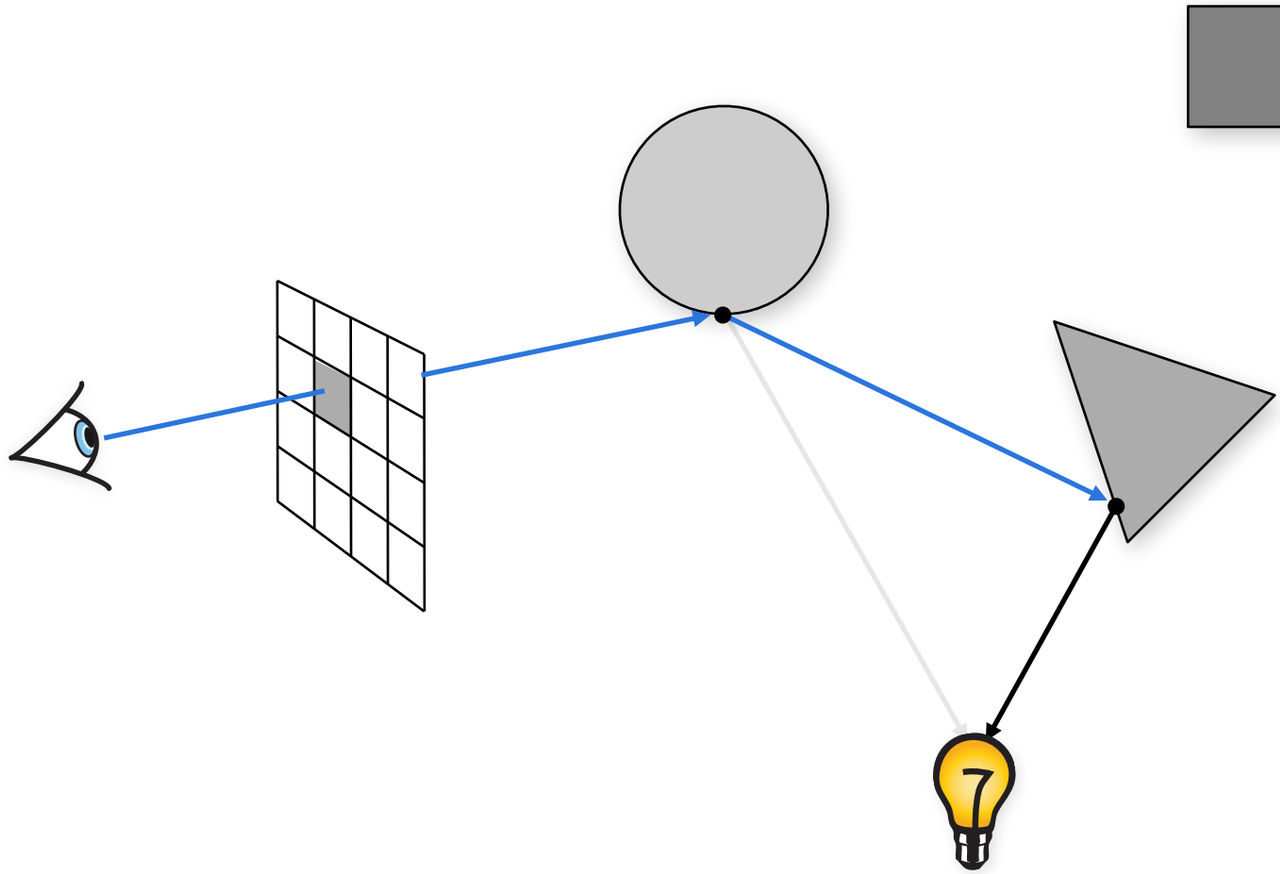


Basic Ray Tracing Pipeline



Basic Ray Tracing Pipeline





Ray Tracing Pseudocode

```
rayTraceImage()
{
    parse scene description

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

wait, what is a ray?

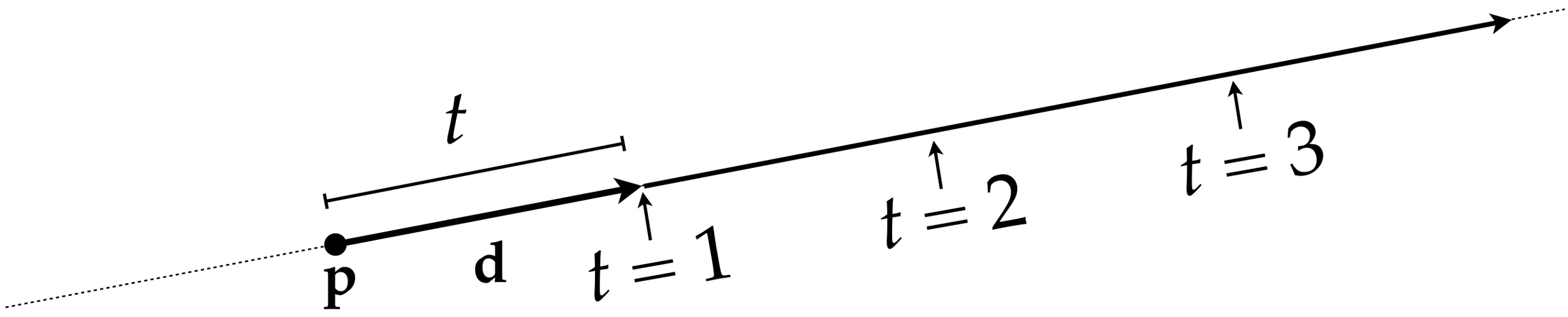
in backward ray tracing, the first ray is seeded from the eye

Ray – a directed half-line

Standard representation: point $\mathbf{p}$ and direction $\mathbf{d}$

$$\mathbf{r}(t) = \mathbf{p} + t\mathbf{d}$$

- this is a parametric equation for the line
- lets us directly generate the points on the line
- if we restrict to $t > 0$ then we have a ray
- note replacing $\mathbf{d}$ with $a\mathbf{d}$ doesn't change ray (for $a > 0$)



Pinhole Camera (Camera Obscura)

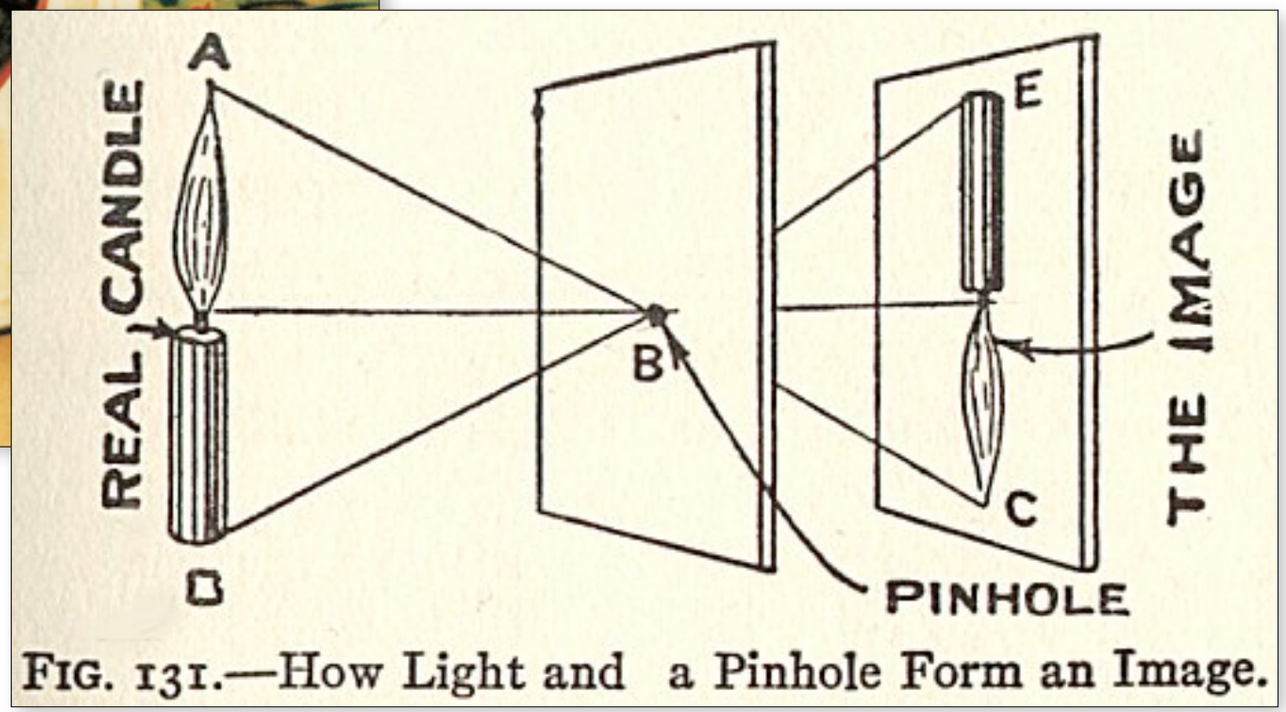
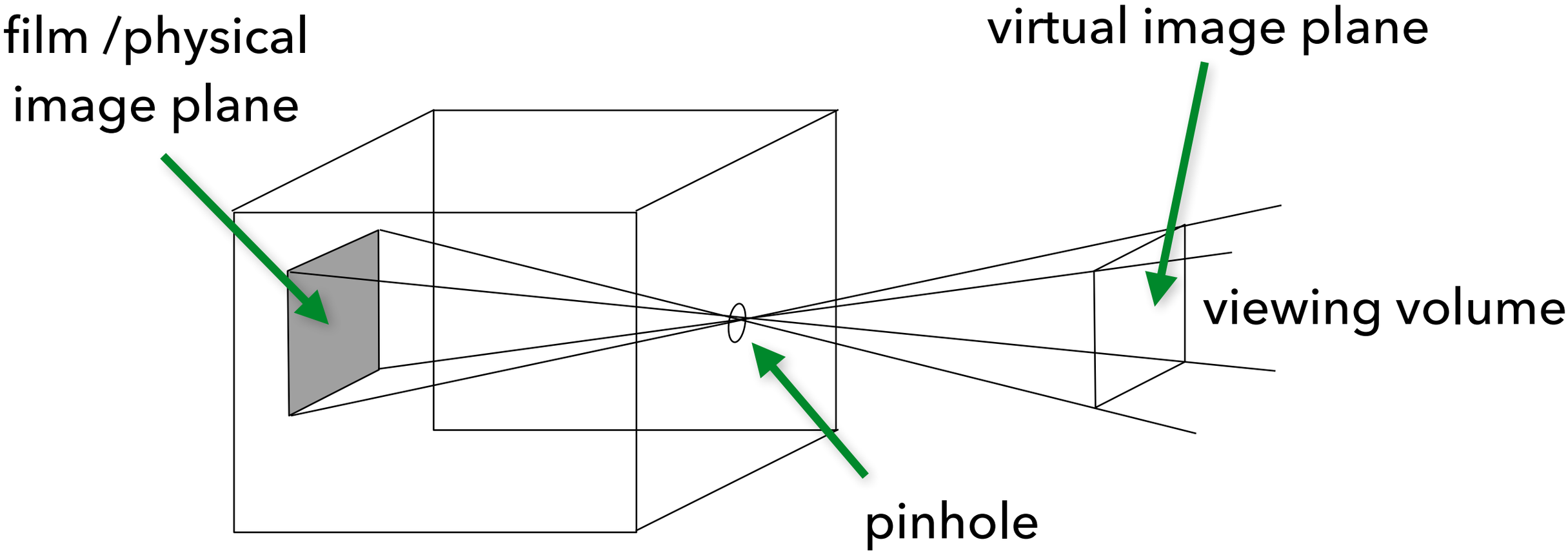
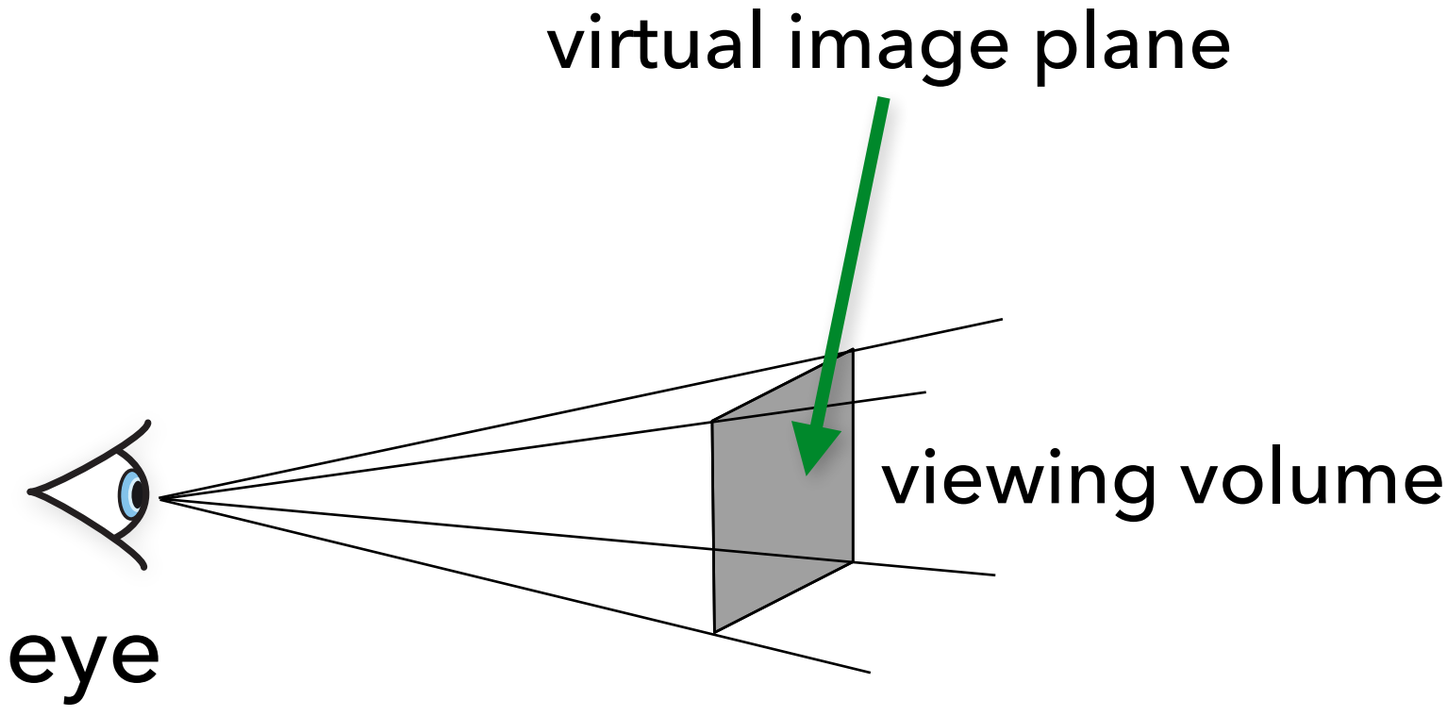


FIG. 131.—How Light and a Pinhole Form an Image.

Pinhole Camera



Pinhole Camera



Ray Tracing Pseudocode

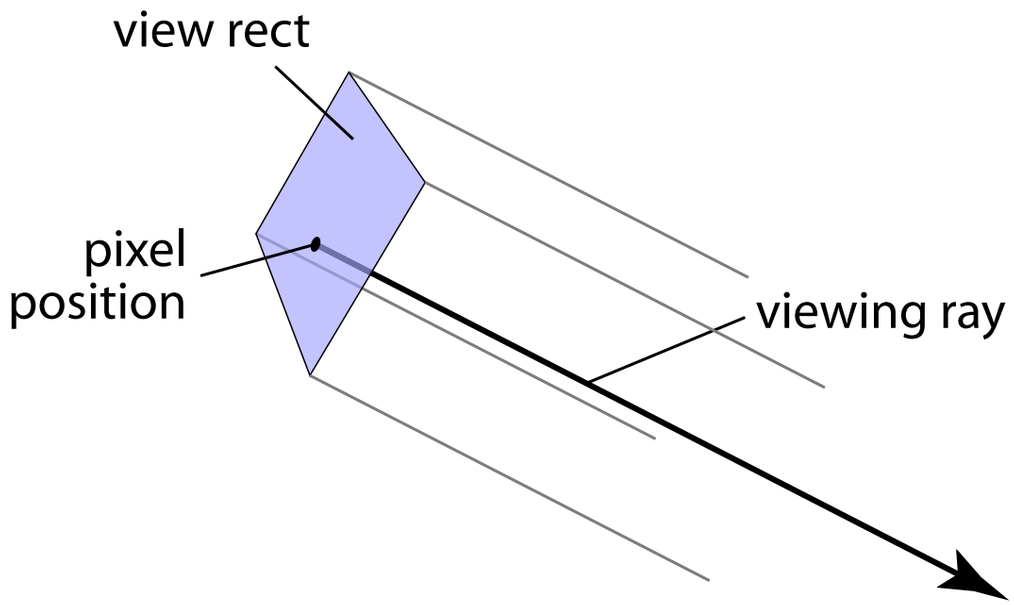
```
rayTraceImage()
{
    parse scene description

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

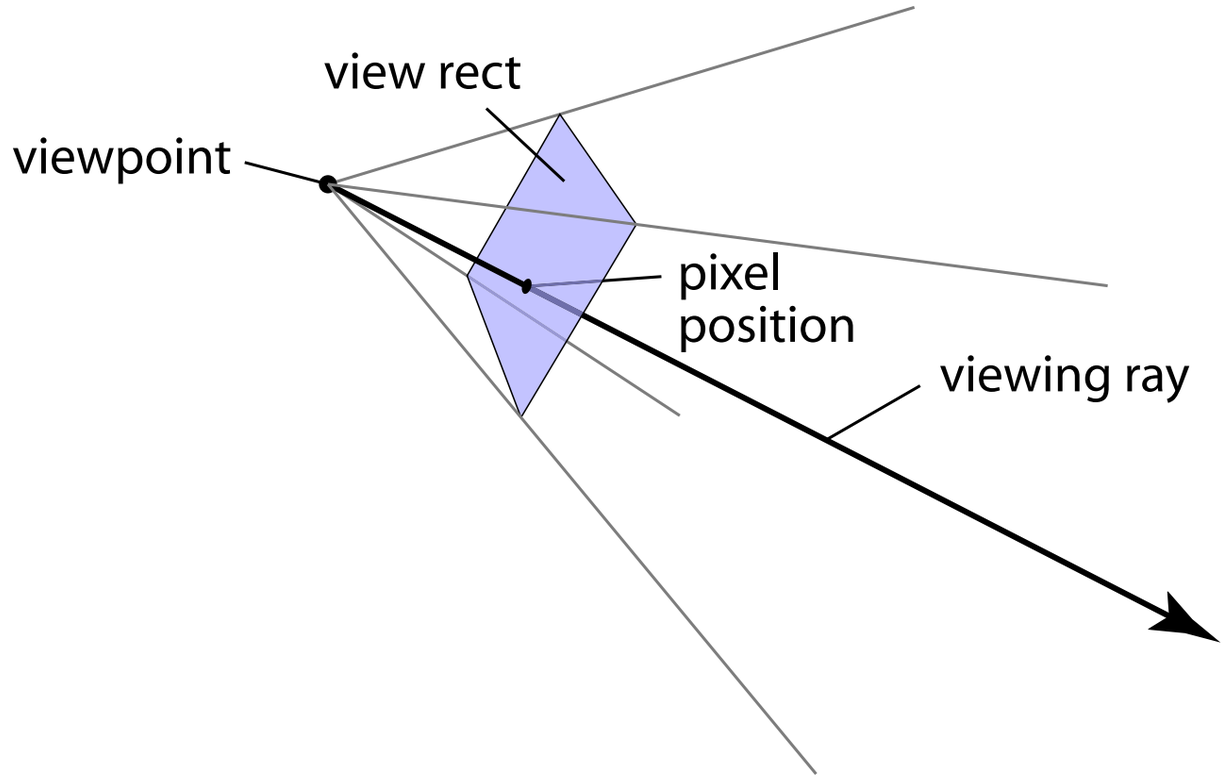
Generating Eye Rays

Use window analogy directly

Orthographic

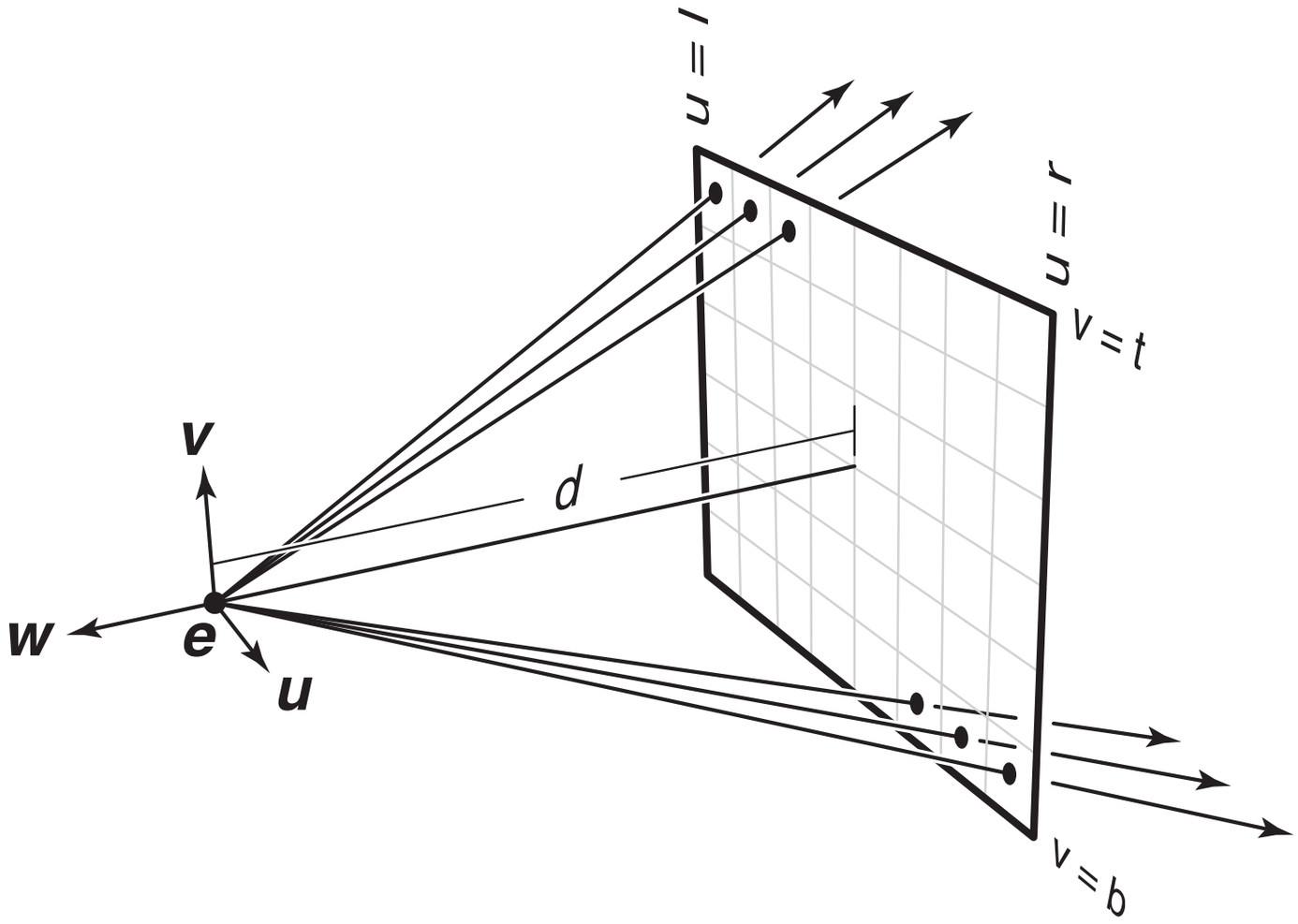


Perspective



After a slide by Steve Marschner

Generating Eye Rays



Ray Tracing Pseudocode

```
rayTraceImage()
{
    parse scene description

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

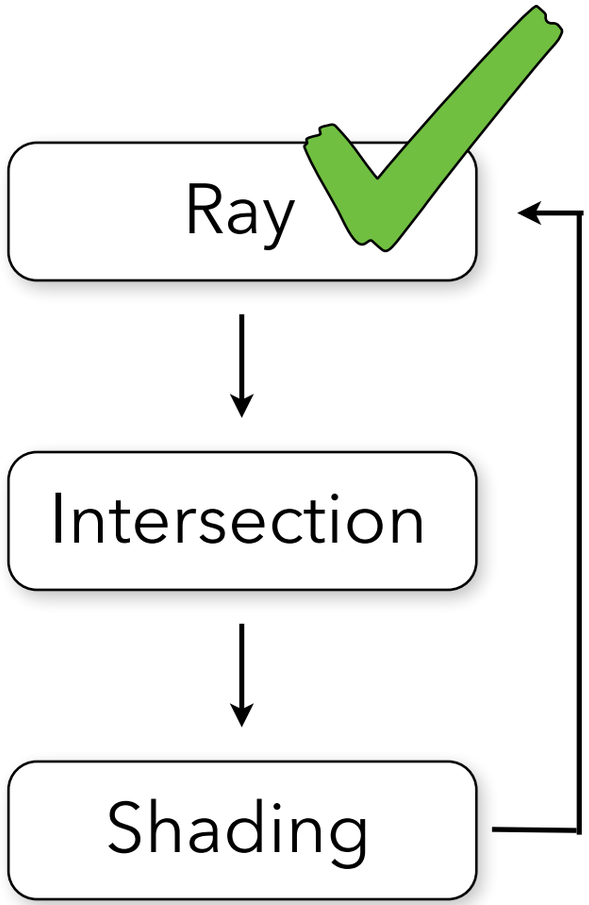
Ray Tracing Pseudocode

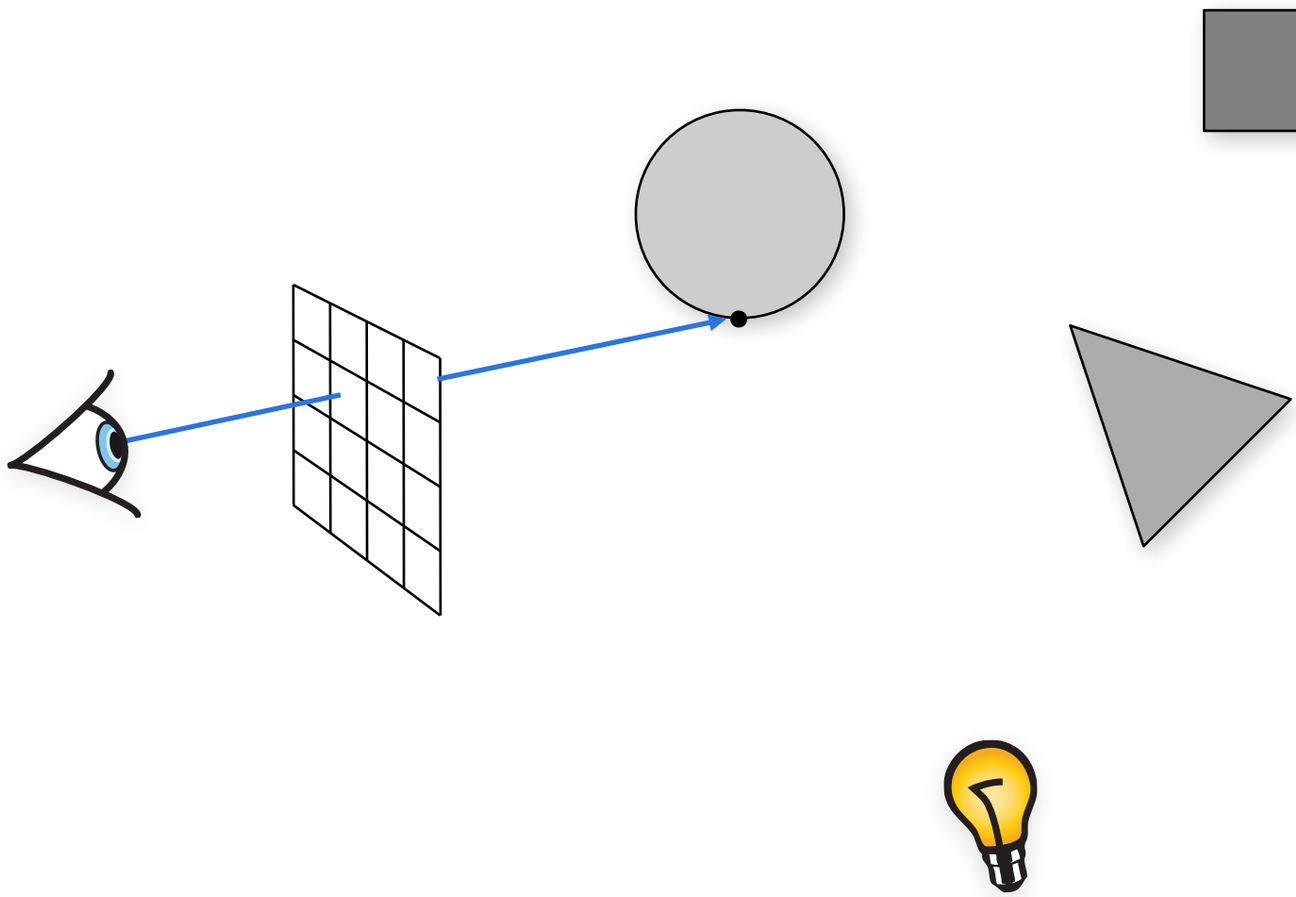
```
trace(ray)
{
    hit =  find first intersection
           with scene objects

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

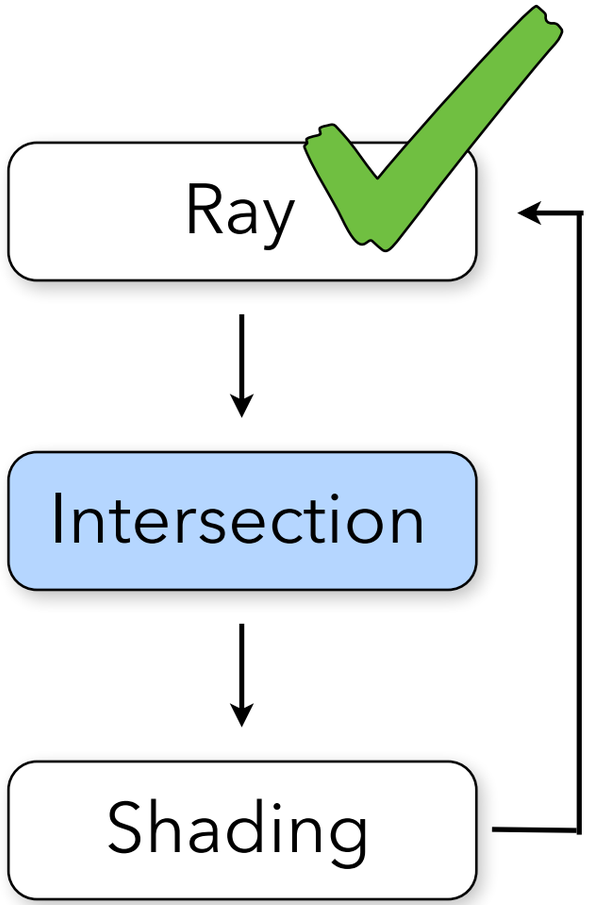
might **`trace`** more rays (recursive)

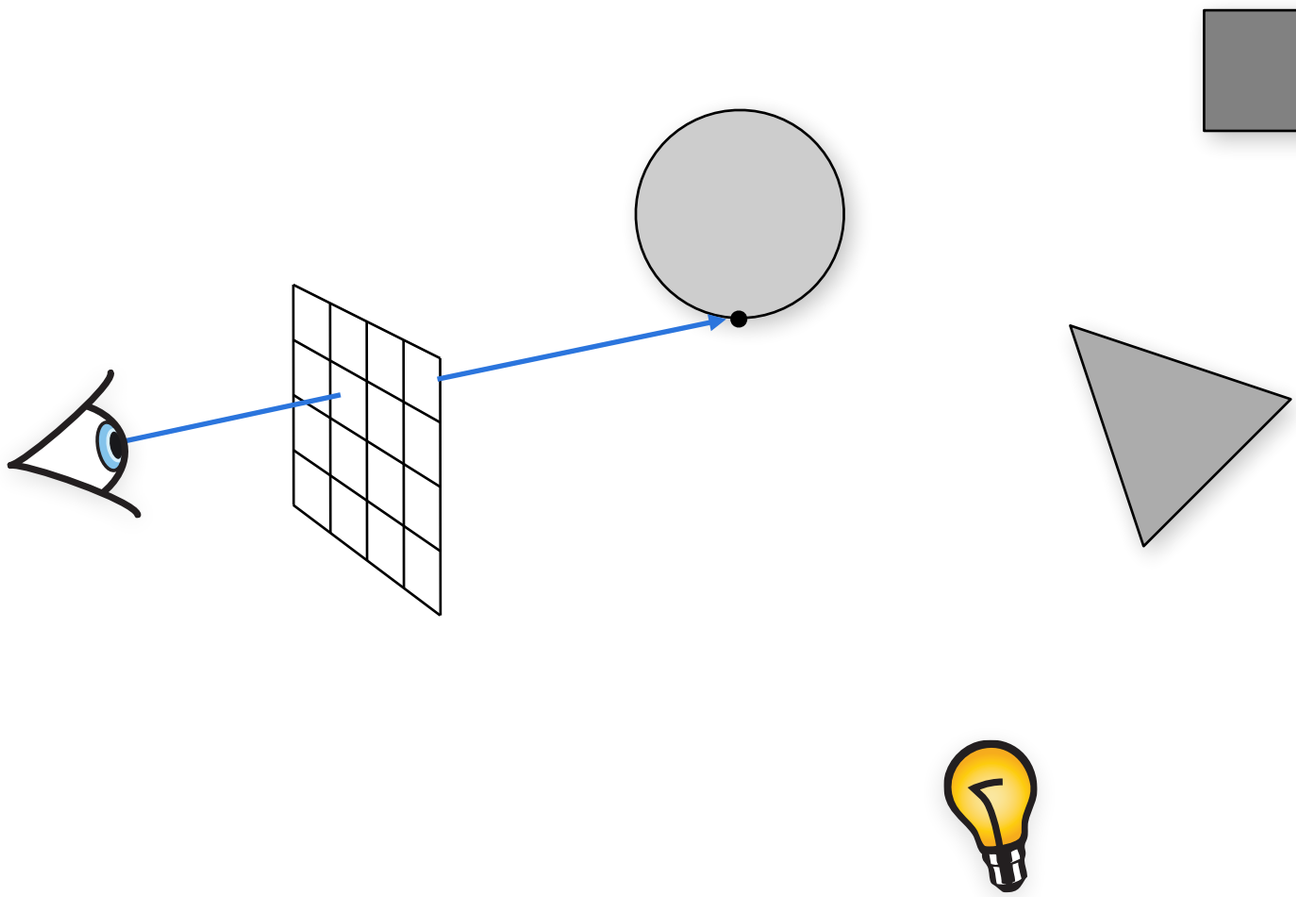
Ray-Surface Intersections





Ray-Surface Intersections





Ray-Surface Intersections

Surface primitives

- spheres
- planes
- triangles
- general implicits
- etc.

Ray-Surface Intersections

Surface primitives

- **spheres**
- planes
- triangles
- general implicits
- etc.

Ray-Sphere Intersection

Parametric equation of a sphere:

$$\|\mathbf{x} - \mathbf{c}\|^2 - r^2 = 0$$

Substitute $\mathbf{x}$ for $\mathbf{r}(t)$ and solve for t :

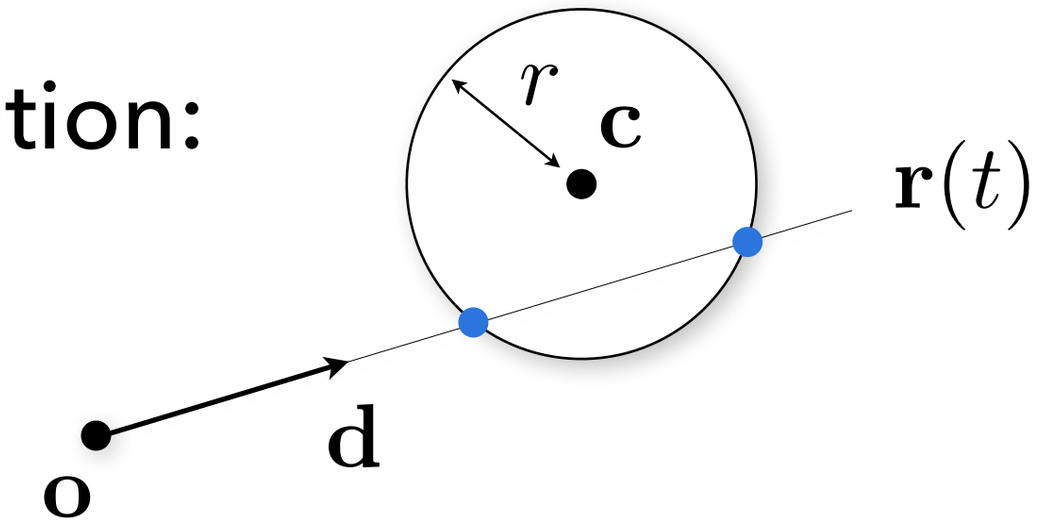
$$\|\mathbf{o} + t\mathbf{d} - \mathbf{c}\|^2 - r^2 = 0 \longrightarrow \begin{aligned}&(\mathbf{o}_x + t\mathbf{d}_x - \mathbf{c}_x)^2 + \\ &(\mathbf{o}_y + t\mathbf{d}_y - \mathbf{c}_y)^2 + \\ &(\mathbf{o}_z + t\mathbf{d}_z - \mathbf{c}_z)^2 - r^2 = 0\end{aligned}$$

which reduces to:

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

Solve for t using quadratic equation:

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



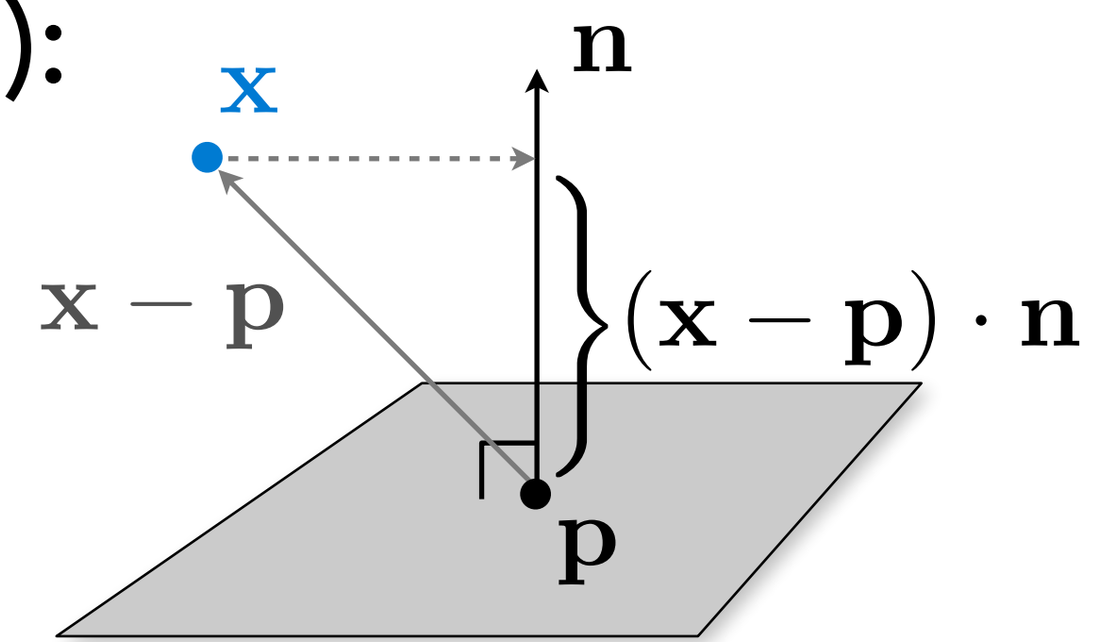
Ray-Surface Intersections

Surface primitives

- spheres
- **planes**
- triangles
- general implicits
- etc.

Ray-Plane Intersection

Plane equations (implicit):



Algebraic form:

$$ax + by + cz + d = 0$$

Geometric (vector) form:

$$(\mathbf{x} - \mathbf{p}) \cdot \mathbf{n} = 0$$

point of interest

point on plane

plane normal

$$\mathbf{x} \cdot \mathbf{n} - \mathbf{p} \cdot \mathbf{n} = 0$$

$$[a, b, c] = \mathbf{n}$$

$$-\mathbf{p} \cdot \mathbf{n} = d$$

Ray-Plane Intersection

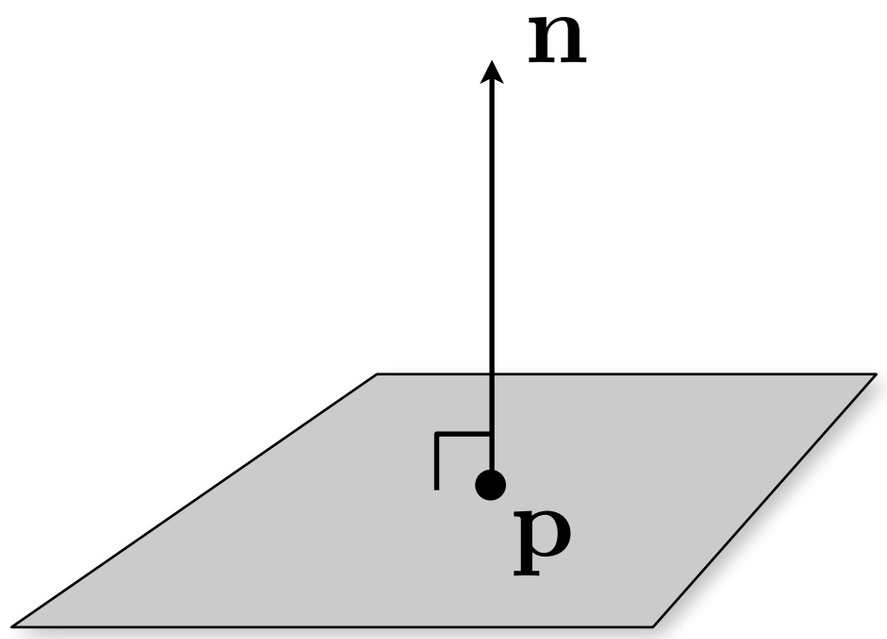
Plane equations (implicit)

$$(\mathbf{x} - \mathbf{p}) \cdot \mathbf{n} = 0$$

point of interest

point on plane

plane normal

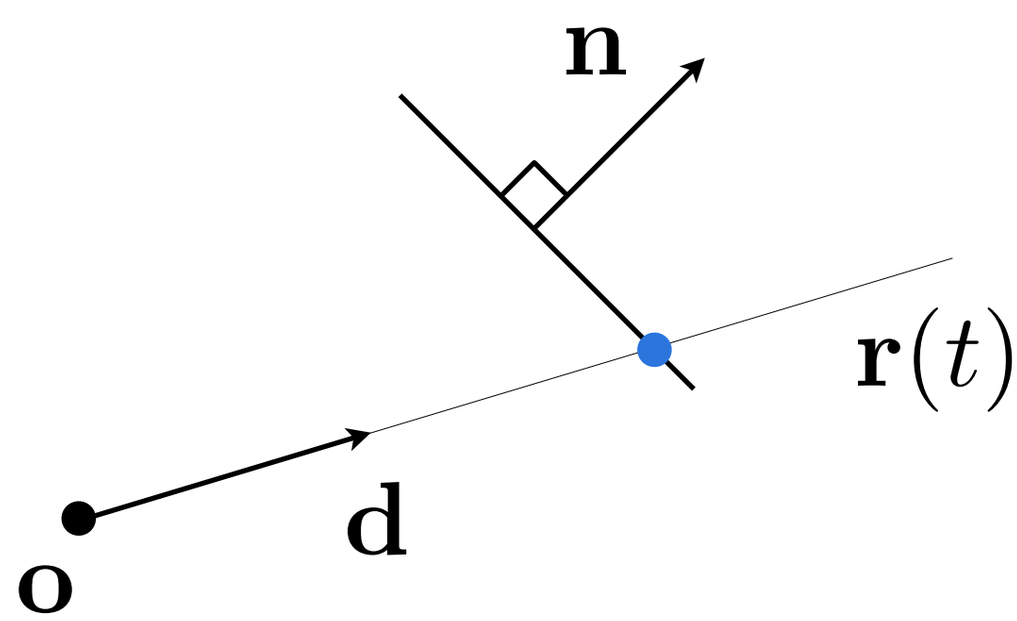


substitute ray equation for $\mathbf{x}$ and solve for t

$$(\mathbf{o} + t\mathbf{d} - \mathbf{p}) \cdot \mathbf{n} = 0$$

$$t\mathbf{d} \cdot \mathbf{n} + (\mathbf{o} - \mathbf{p}) \cdot \mathbf{n} = 0$$

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



Ray-Surface Intersections

Surface primitives

- spheres
- planes
- **triangles**
- general implicits
- etc.

Ray-Triangle intersection

Conditions

1. point is on the ray: $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$
2. point is on plane: $(\mathbf{x} - \mathbf{p}) \cdot \mathbf{n} = 0$
3. point is inside of all three edges

First solve 1 & 2 (ray–plane intersection); substitute and solve for t: $(\mathbf{o} + t\mathbf{d} - \mathbf{p}) \cdot \mathbf{n} = 0$

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

Several options for #3...

After a slide by Steve Marschner

Barycentric Coordinates

Barycentric coordinates:

$$\mathbf{x}(\alpha, \beta, \gamma) = \alpha\mathbf{p}_1 + \beta\mathbf{p}_2 + \gamma\mathbf{p}_3$$

Inside triangle conditions:

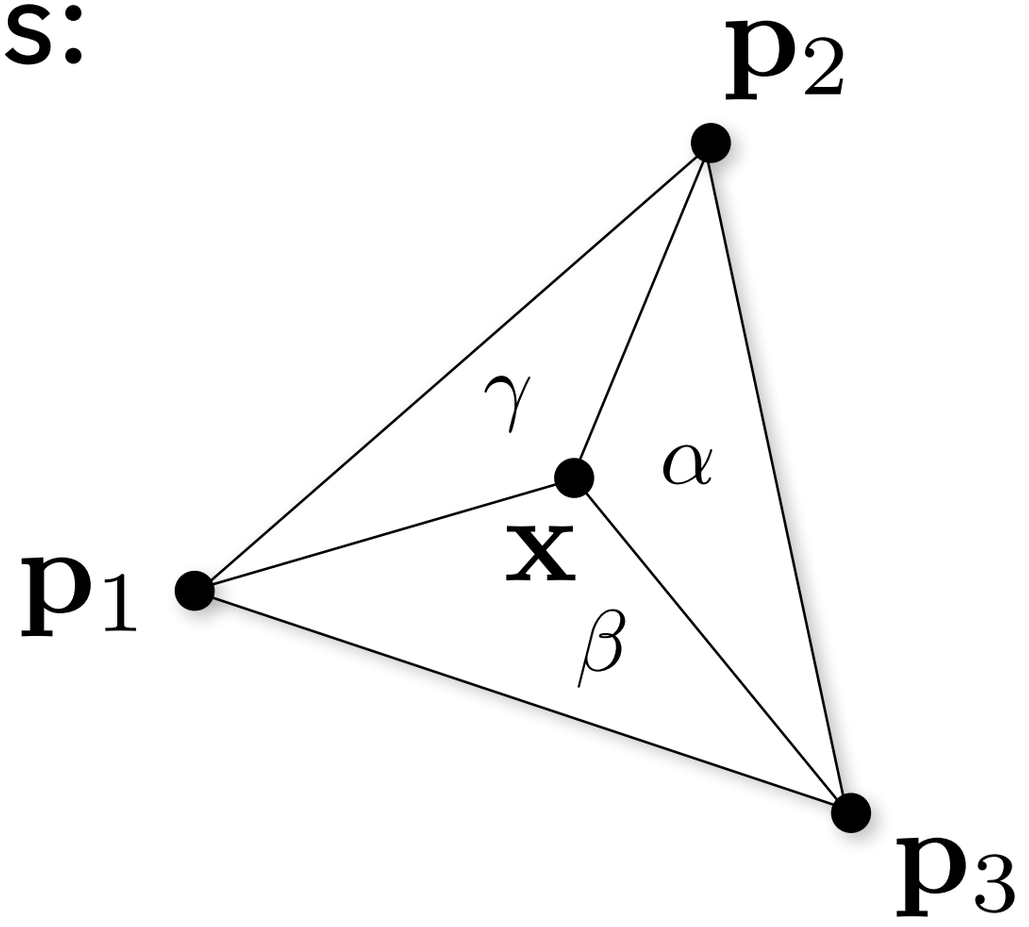
$$\alpha + \beta + \gamma = 1$$

$$\gamma = 1 - \alpha - \beta$$

$$0 \leq \alpha \leq 1$$

$$0 \leq \beta \leq 1$$

$$0 \leq \gamma \leq 1$$



Ray-Triangle Intersection

Insert ray equation:

$$\alpha\mathbf{p}_1 + \beta\mathbf{p}_2 + (1-\alpha-\beta)\mathbf{p}_3 = \mathbf{o} + t\mathbf{d}$$

$$\alpha(\mathbf{p}_1 - \mathbf{p}_3) + \beta(\mathbf{p}_2 - \mathbf{p}_3) + \mathbf{p}_3 = \mathbf{o} + t\mathbf{d}$$

$$\alpha(\mathbf{p}_1 - \mathbf{p}_3) + \beta(\mathbf{p}_2 - \mathbf{p}_3) - t\mathbf{d} = \mathbf{o} - \mathbf{p}_3$$

$$\alpha\mathbf{a} + \beta\mathbf{b} - t\mathbf{d} = \mathbf{e}$$

Solve system of equations:

$$\begin{bmatrix} -\mathbf{d} & \mathbf{a} & \mathbf{b} \end{bmatrix} \begin{bmatrix} t \\ \alpha \\ \beta \end{bmatrix} = \mathbf{e}$$

Ray-Surface Intersections

Other primitives

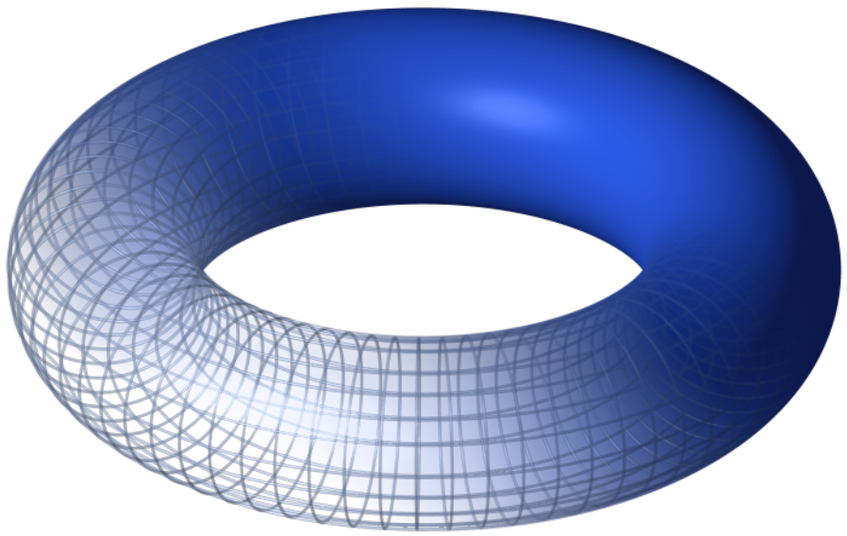
- cylinder
- cone, paraboloid, hyperboloid
- torus
- disk
- etc.

For Example – Torus

Can, once again, start from the implicit equation for a torus:

$$\left(R - \sqrt{\mathbf{x}^2 + \mathbf{y}^2}\right)^2 + \mathbf{z}^2 = r^2$$

Substitute the parametric equation of a ray and solve for t



source: wikipedia

Intersection – Coordinate Systems

Consider intersecting a transformed sphere

- given, say, a model-to-world transform matrix

Option 1:

- transform sphere into world coordinates
- write code to intersect arbitrary ellipsoids

Option 2:

- (inverse) transform ray into sphere's object space
- simply apply canonical sphere intersection routine