INFOGR – Graphics

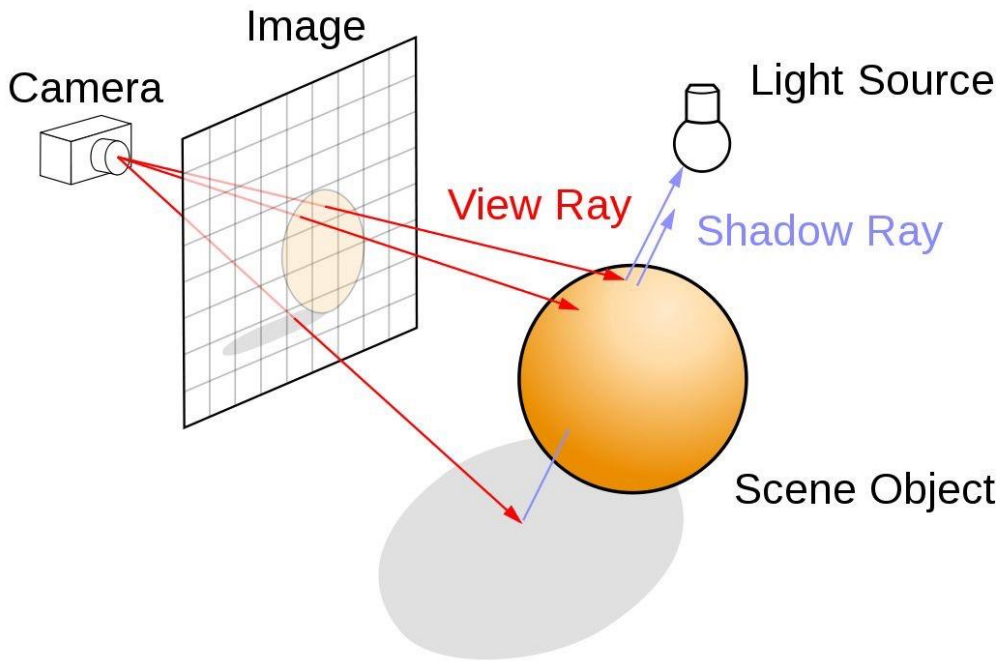
Peter Vangorp

Lecture 04: Ray Tracing

Today's Agenda:

- Rasters
- Colors
- Projection
- Ray Tracing

Rendering



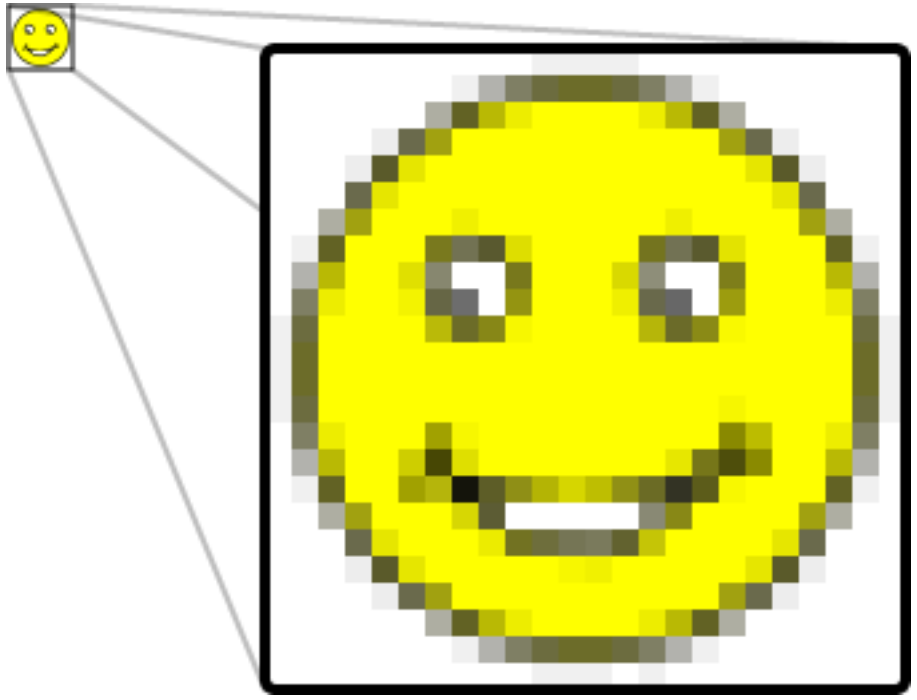
Rendering is the process of generating a photorealistic or non-photorealistic image from a 2D or 3D model by means of a computer program

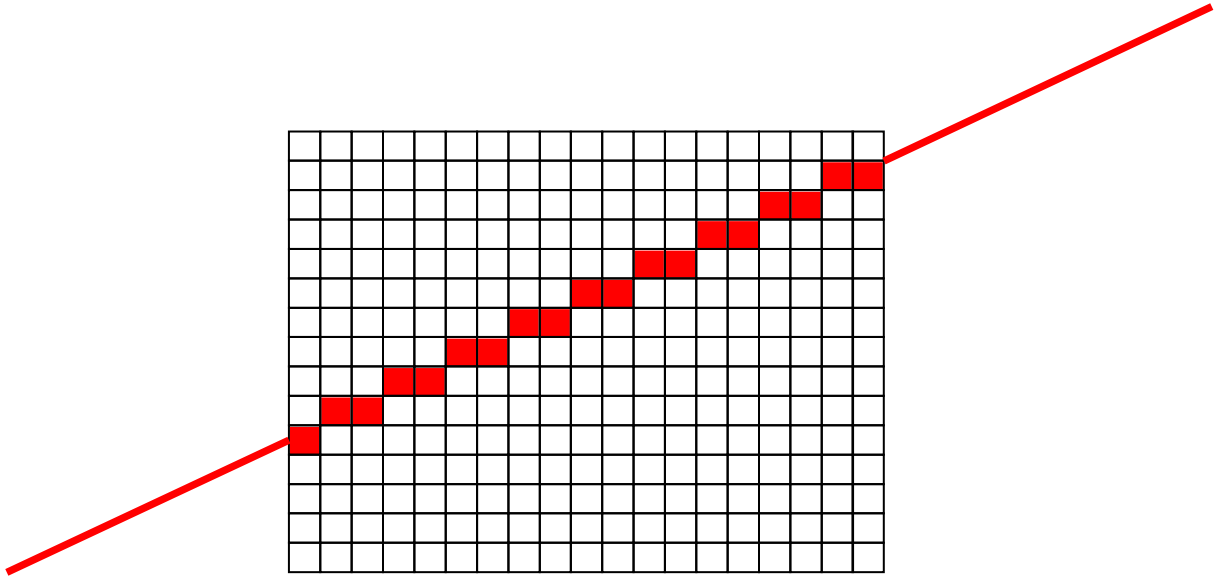
(Wikipedia)

Two main rendering methods:

1. **Ray tracing:** for each pixel, which object covers it?
2. **Rasterization:** for each object, which pixels does it cover?

Raster Image





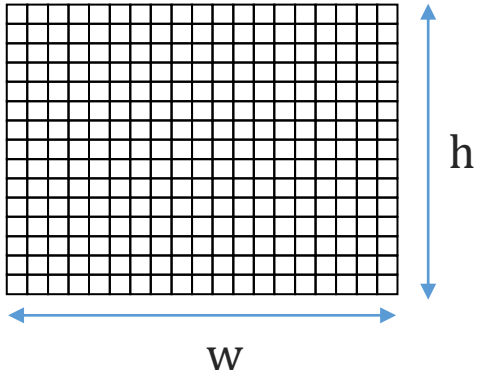
Raster image: grid of individual pixels that collectively compose an image

Pixel: picture element with a color

Discretization: turning a continuous model into a discrete representation

- Analog sound into a digital stream
- Scene description into a raster image

Raster Image



w × h- pixel resolution

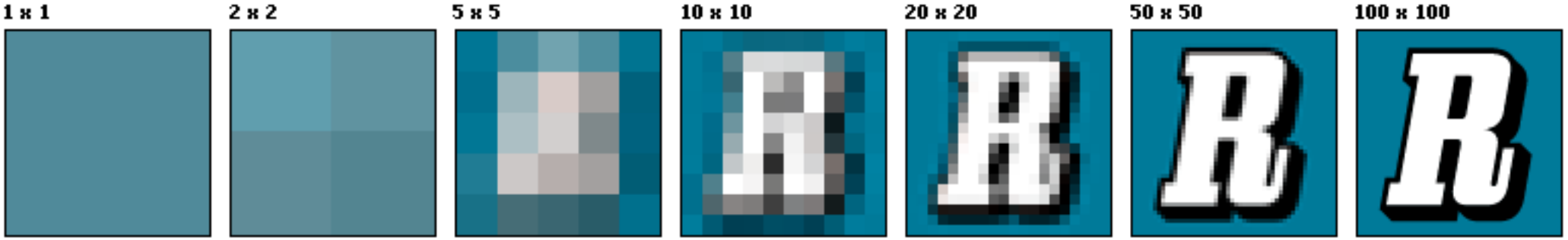
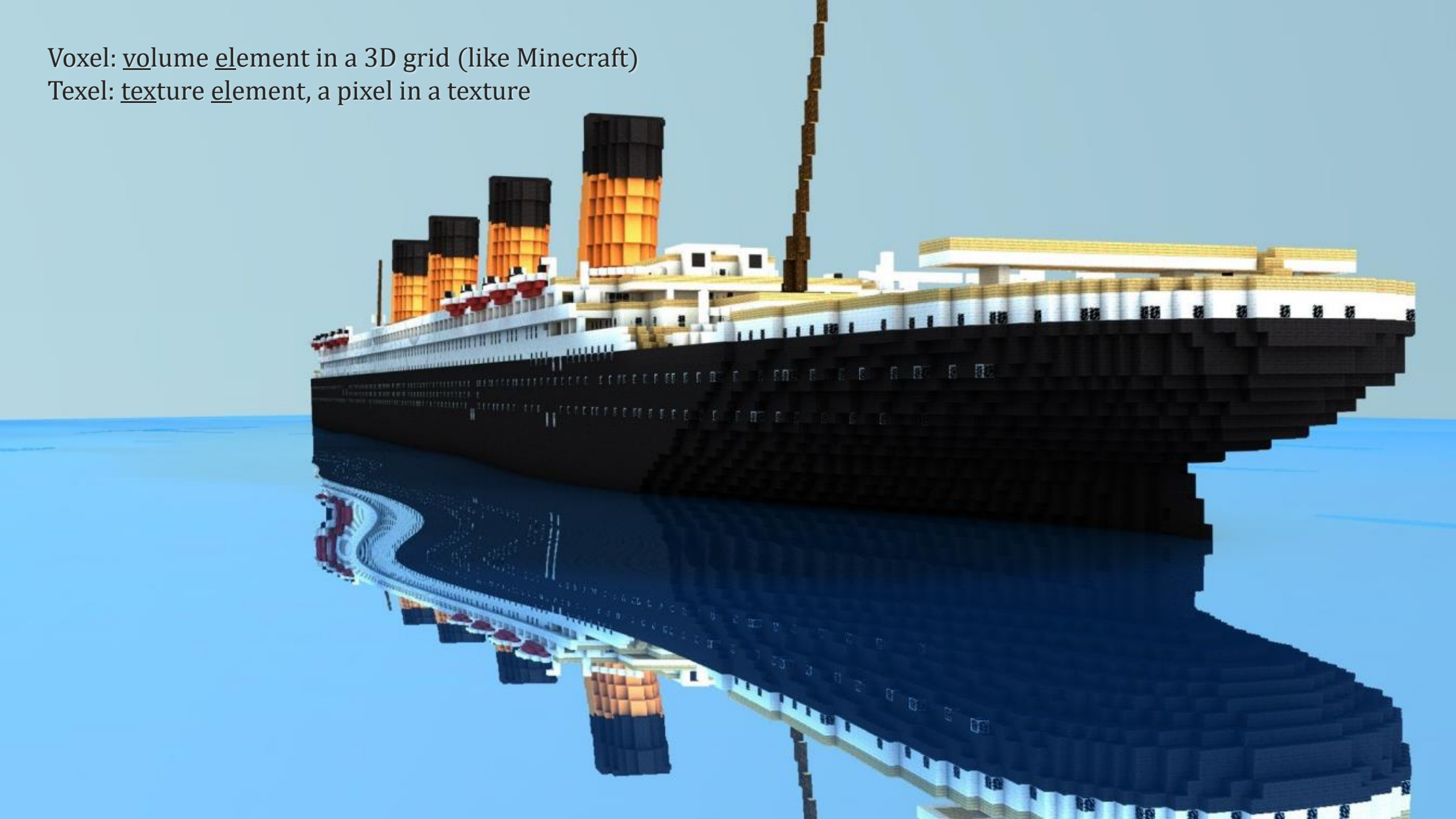


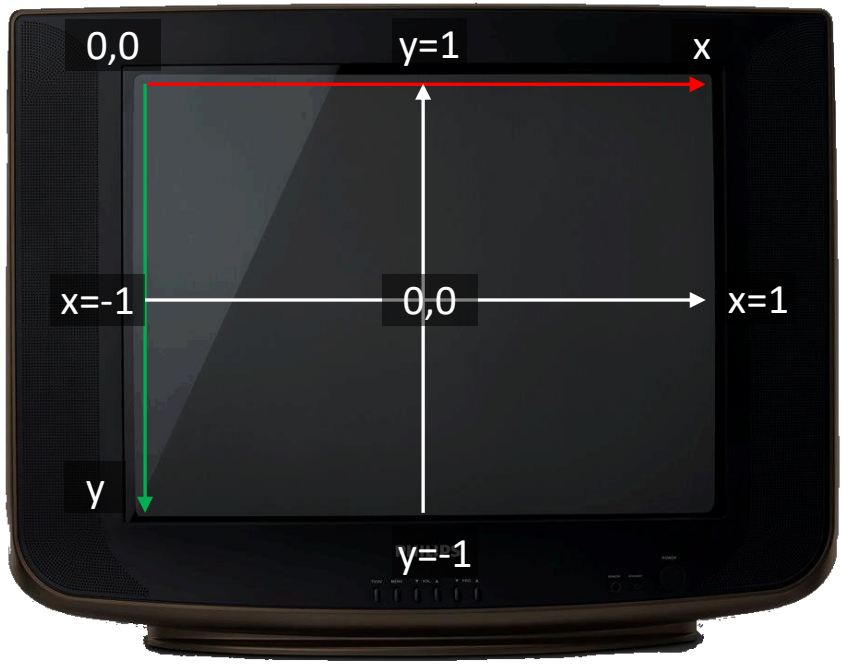
Illustration of how the same image might appear at different pixel resolutions

(Wikipedia)

Voxel: volume element in a 3D grid (like Minecraft)
Texel: texture element, a pixel in a texture



Display



Common raster display resolutions:

- 1920×1080 (Full HD, 1080p)
- 1366×768 (“HD ready”)
- 2532×1170 (iPhone 12)
- …

Screen coordinates

- Pixel coordinates are only relevant for the final step: plotting pixels
- Decouple the 2D screen coordinates in your game / app from the physical mapping

Many possible coordinate systems

- For math, 3D modeling, screen, etc.
- Different origin position (e.g., bottom-left, top-left, center)
- Different axis directions (e.g., y-axis up or down)
- Different range (e.g., -1..1 or 0..1920)

Today's Agenda:

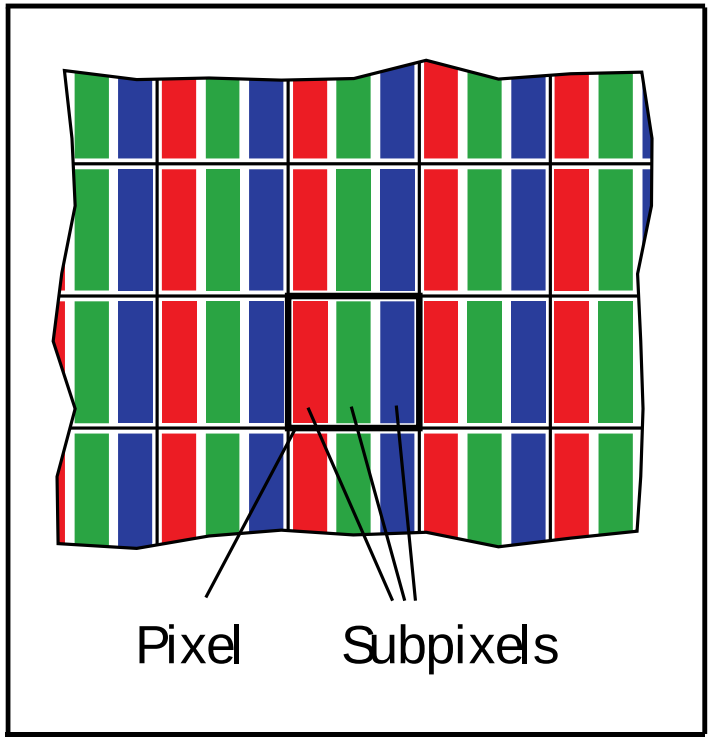
- Rasters
- Colors
- Projection
- Ray Tracing

Colors

Color representation

Computer pixel consists of three subpixels: red, green, and blue (RGB)

By additively mixing these, we can produce most colors

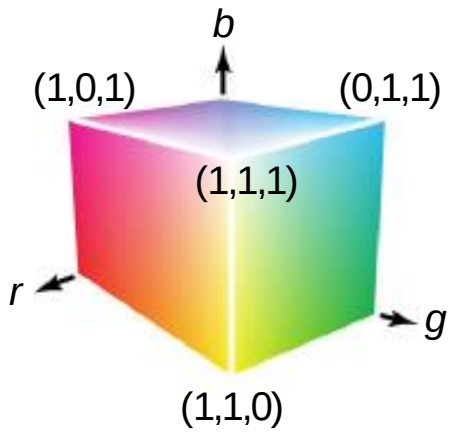


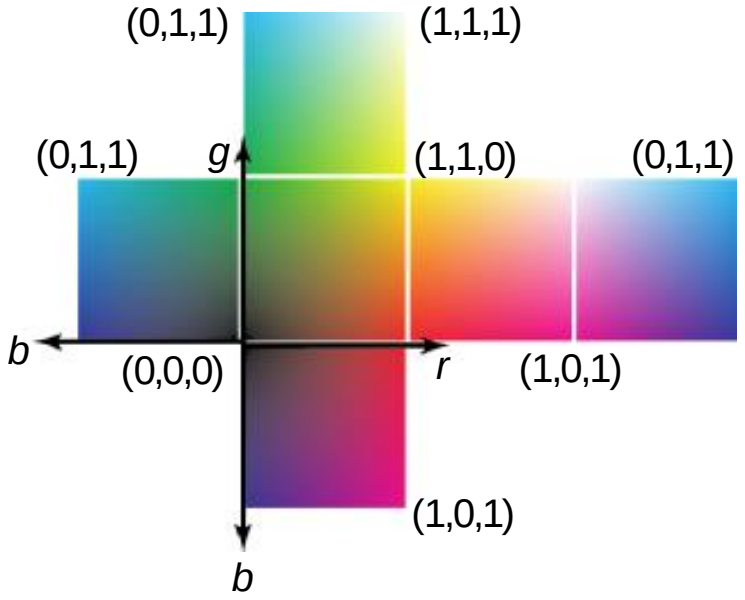
Colors

RGB

- The most used format
- Three values: for red, green, and blue

Black	= (0,0,0)
White	= (1,1,1)
Red	= (1,0,0)
Green	= (0,1,0)
Blue	= (0,0,1)





Storage

- 8 bits for each channel
  $2^8 = 256$ possible values (0..255)
  $0 = {}^{0}/_{255}\ ,\ {}^{1}/_{255}, \dots ,\ {}^{255}/_{255} = 1$
- RGB: 3 channels, $3 \times 8 = 24$ bits total

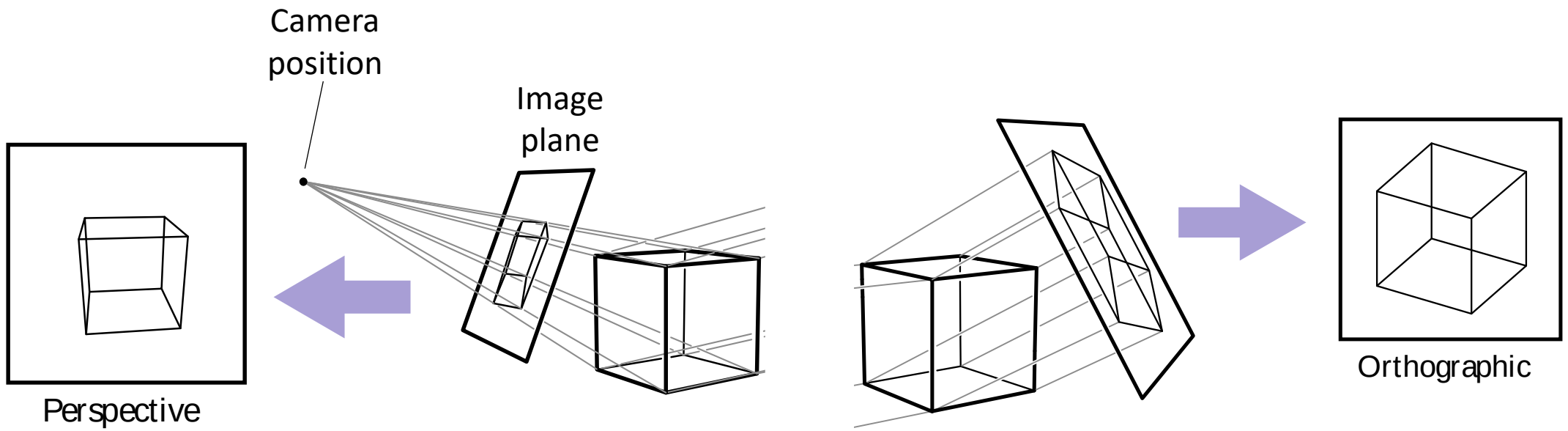
Internal representation:

- use `float` type for calculating colors in range 0..1
- discretize them to an integer type for the final image result

Today's Agenda:

- Rasters
- Colors
- Projection
- Ray Tracing

Projections



We need to project a 3D object/scene onto a 2D image/screen
Our eyes and (real) cameras use perspective projection, so we will emulate that in graphics with a virtual eye / camera and a virtual image plane

Today's Agenda:

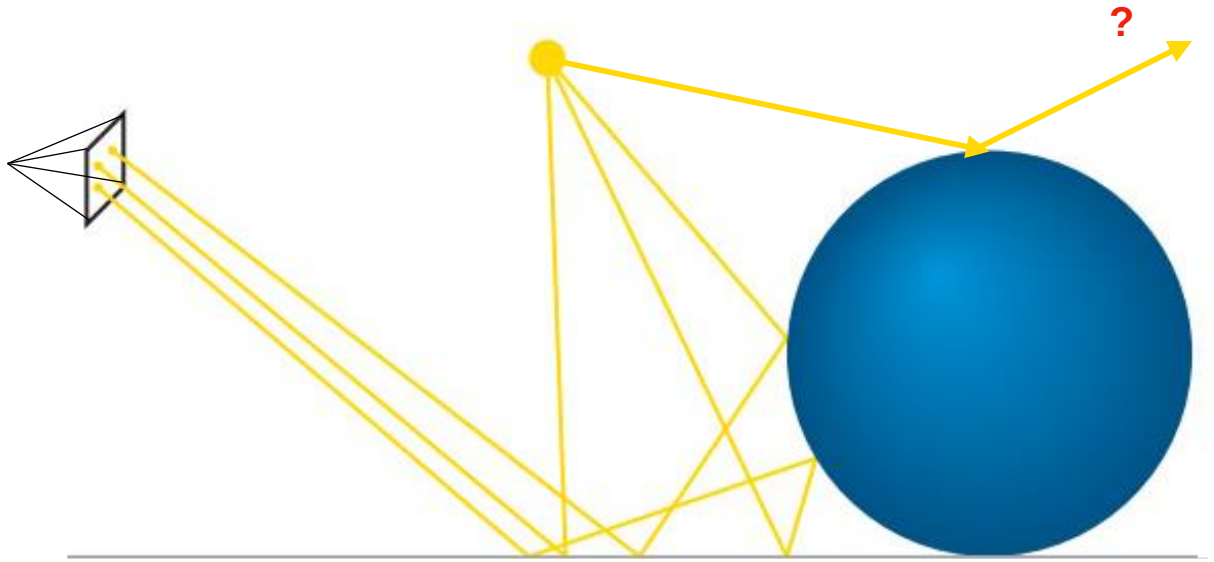
- Rasters
- Colors
- Projection
- Ray Tracing

Light rays in nature

- Light sources emit rays of light
- Light rays travel in straight lines

  (in a vacuum or in some media such as air)
- Rays bounce off the objects in the scene

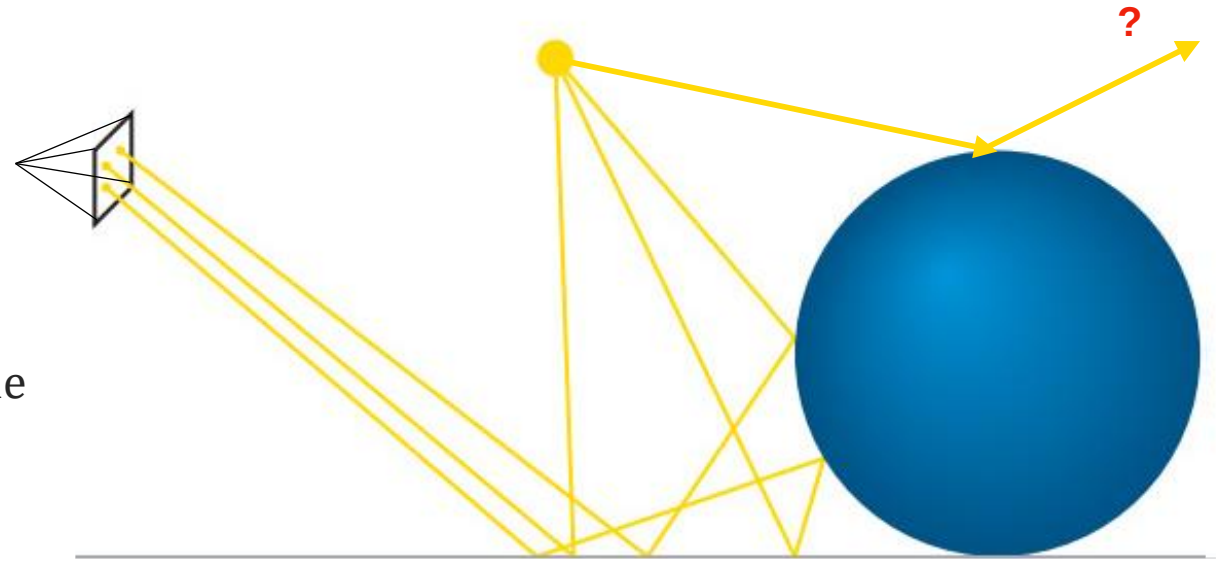
  (possibly many times off different objects)
- Some of the light hits the eye / camera and forms an image



Light rays in graphics

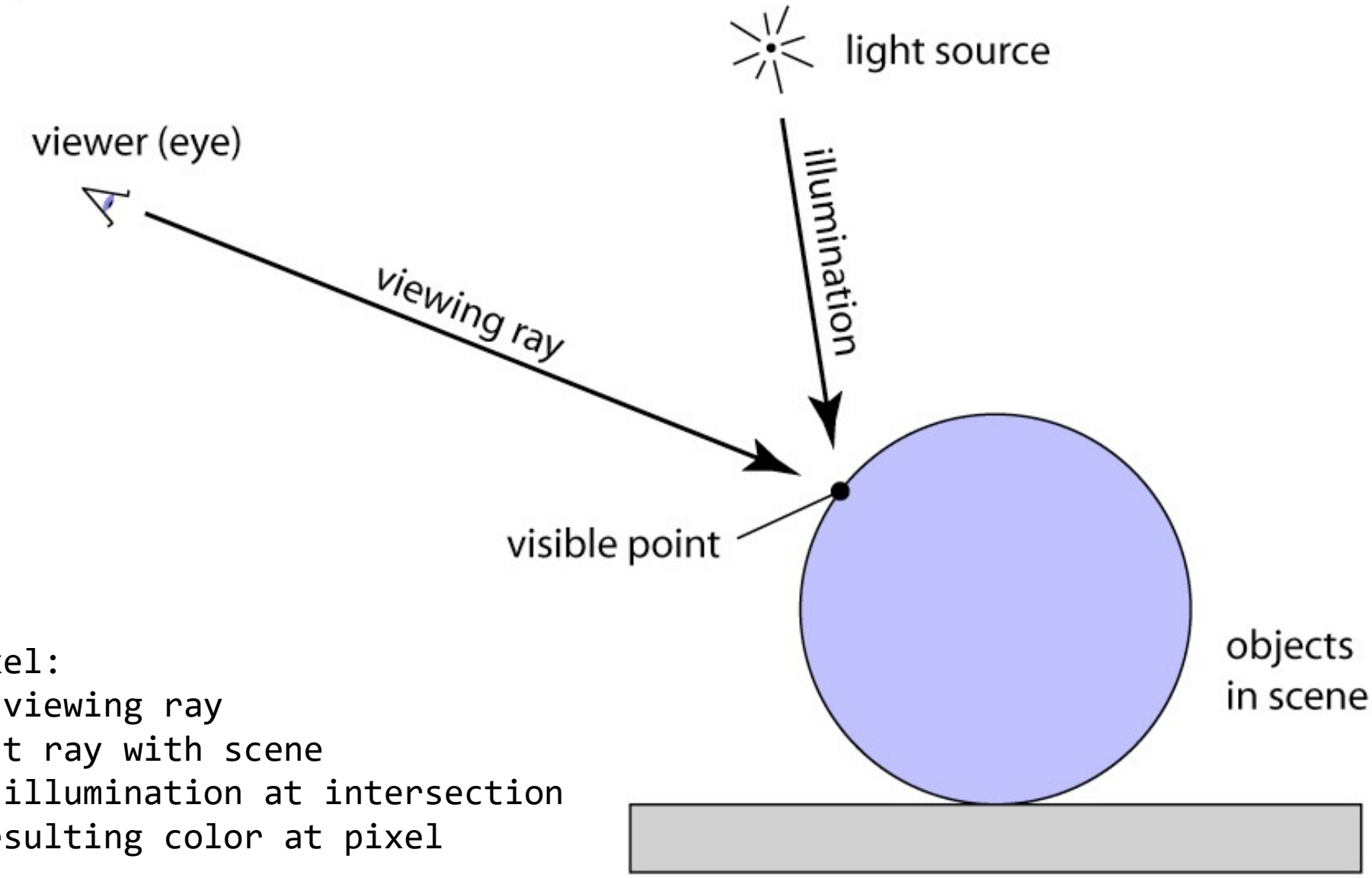
- Light sources emit rays of light that travel in straight lines
- Rays bounce off the objects in the scene

  (possibly many times off different objects)
- At each bounce we check if that piece of surface is visible by tracing a ray to the camera
  - Some reach the camera through the image plane
  - But most do not! Computationally wasteful!



- **Better idea:** Trace rays from camera into the scene (“backward ray tracing”) and at each bounce check if that piece of surface is illuminated
  - More likely to be illuminated: light sources usually aren’t limited to a narrow access like the image plane

Basic Ray Tracer Algorithm



Pseudocode:

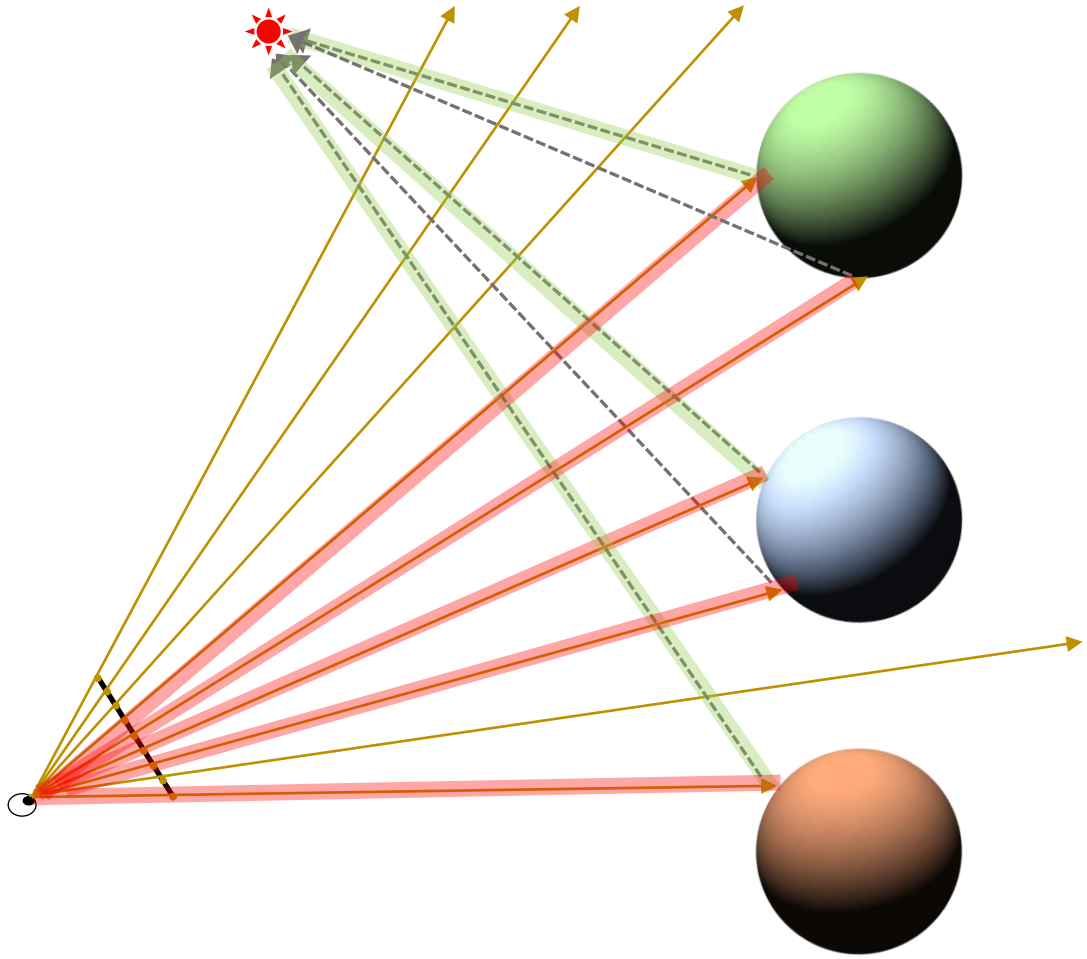
```
for each pixel:
    compute viewing ray
    intersect ray with scene
    compute illumination at intersection
    store resulting color at pixel
```

Ray Tracing

Given
- 3D models
- Eye
- Image plane
- Image pixels (their number depends on the desired picture resolution)

Compute
- Viewing rays aka primary rays
  - Intersections with 3D models
- Point light(s)
- Shadow rays
  - Intersections with 3D models

Note
- For now, a piece of surface is either illuminated or not
- More when we talk about shading

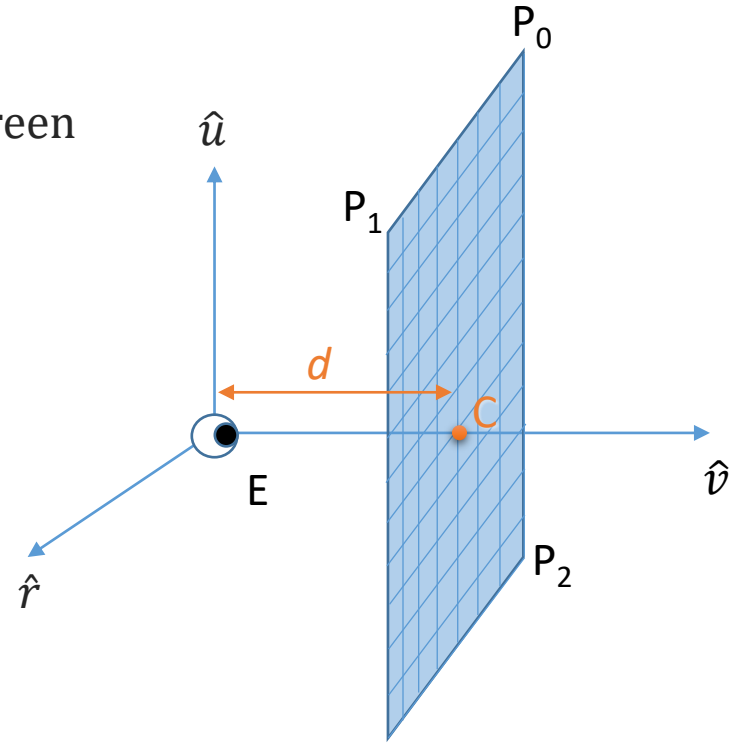


Camera setup

Camera position: E
View direction: $\hat{v}$ (through the center of the image)
Up direction: $\hat{u}$
Right direction: $\hat{r}$ ($\hat{v}$, $\hat{u}$, and $\hat{r}$ should form an orthonormal basis)
Image plane center: $C = E + d\hat{v}$
Image plane corners: $P_0 = C + \hat{u} - a\hat{r}$, $P_1 = C + \hat{u} + a\hat{r}$, $p_2 = C - \hat{u} - a\hat{r}$
where a is the aspect ratio of the image
e.g., $a = 1$ for a square image, or $a = 16/9$ for widescreen
Pixel positions: depend on the desired picture resolution

From here:

- Change field of view (FOV) / focal length / zoom by altering d
- Move the camera by changing E
- Rotate the camera by changing $\hat{v}$, $\hat{u}$, and $\hat{r}$ (but keep them orthonormal!)

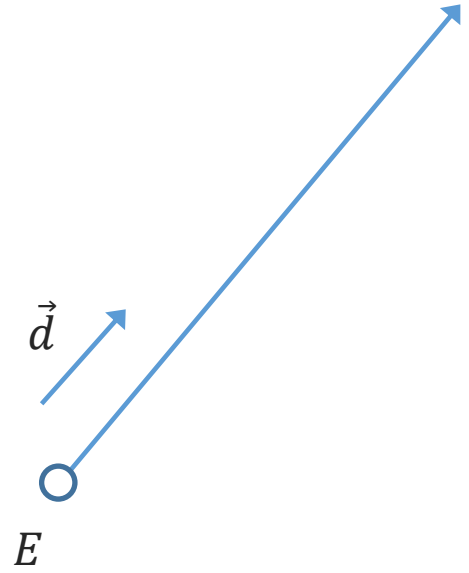


Ray setup (in code)

A ray is an infinite line with a start point:

$P(t) = E + t\vec{d}$, where $t > 0$ (parametric equation)

The ray direction is generally *normalized.*

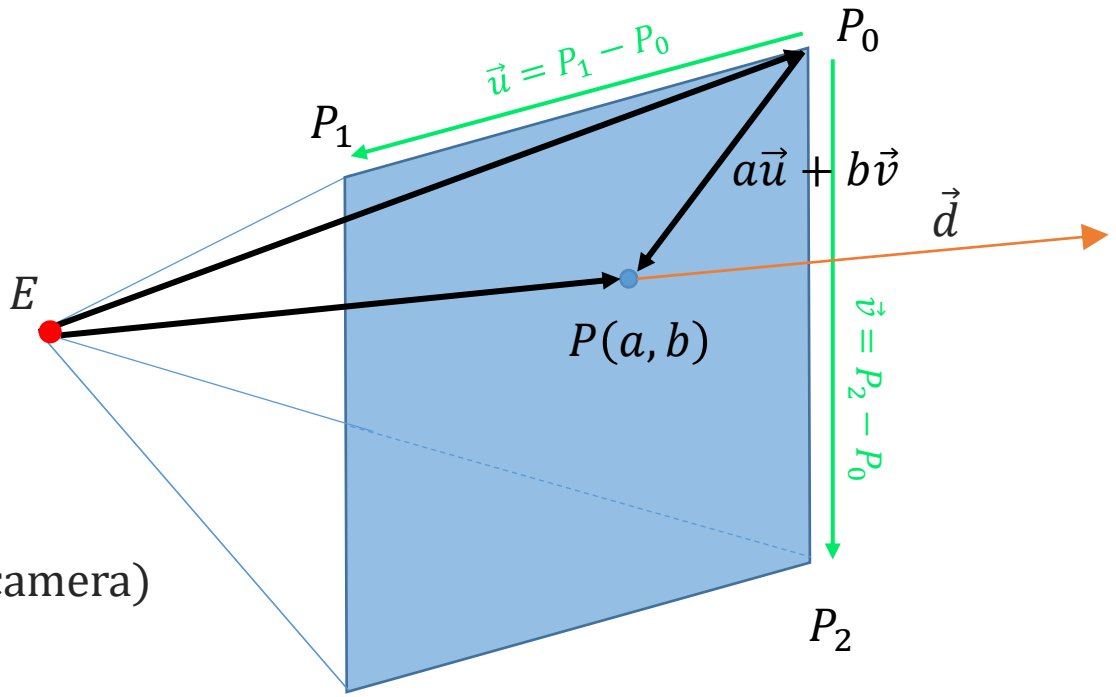


Suggested UML class diagram:

Ray
origin *E* : Vector3 direction *d* : Vector3 intersection distance *t* : float

Ray setup (in the scene)

Screen plane orthogonal basis: $\vec{u} = P_1 - P_0, \vec{v} = P_2 - P_0$



Point on the screen:

$P(a,b) = P_0 + a\vec{u} + b\vec{v}$ where $a, b \in [0,1]$
where pixel (x, y) in an image of resolution $w \times h$ has coordinates $a = {}^{x}/_{w}, b = {}^{y}/_{h}$

Ray direction:

$\vec{d} = P(a,b) - E$ where E is the ray origin (the eye / camera)

$\hat{d} = \frac{\vec{d}}{\|\vec{d}\|}$ (normalization)

$P(t) = E + t\hat{d}$, where $t > 0$

Ray intersection with a 3D model: plane

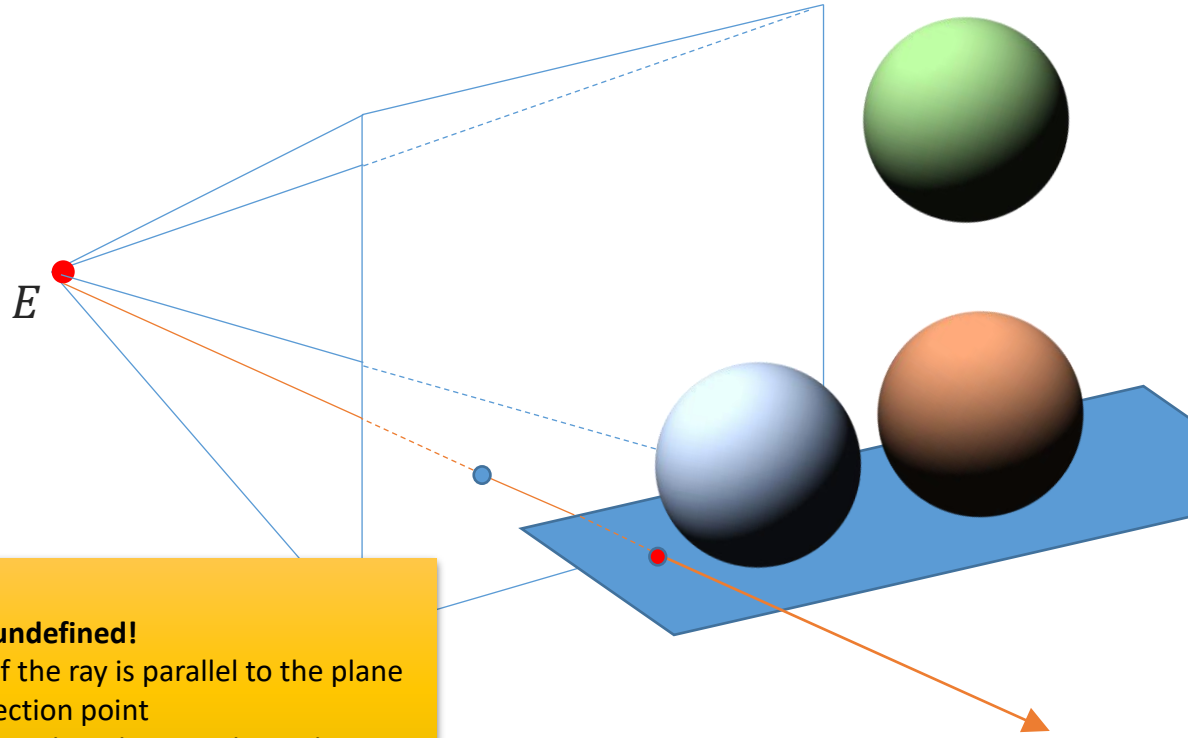
Given a ray $P(t) = E + t\vec{d}$, we determine the smallest positive intersection distance t by intersecting the ray with each of the primitives in the scene

Plane:
Normal vector $\hat{n}$, through point P_0

Intersection equation: $(P(t) - P_0) \cdot \hat{n} = 0$

$$\left(E + t\vec{d} - P_0\right) \cdot \hat{n} = 0$$

$$t = \frac{(P_0 - E) \cdot \hat{n}}{\vec{d} \cdot \hat{n}}$$



WARNING
If $\vec{d} \cdot \hat{n} = 0$ then t is undefined!
Geometric meaning: If the ray is parallel to the plane then there’s no intersection point
(or every point on the ray is an intersection point but we treat that as no intersection point…)

NOTICE
Math with vectors instead of separate x, y, z is simpler, clearer, and reveals the geometric meaning.
Also use vector math in your code if possible!

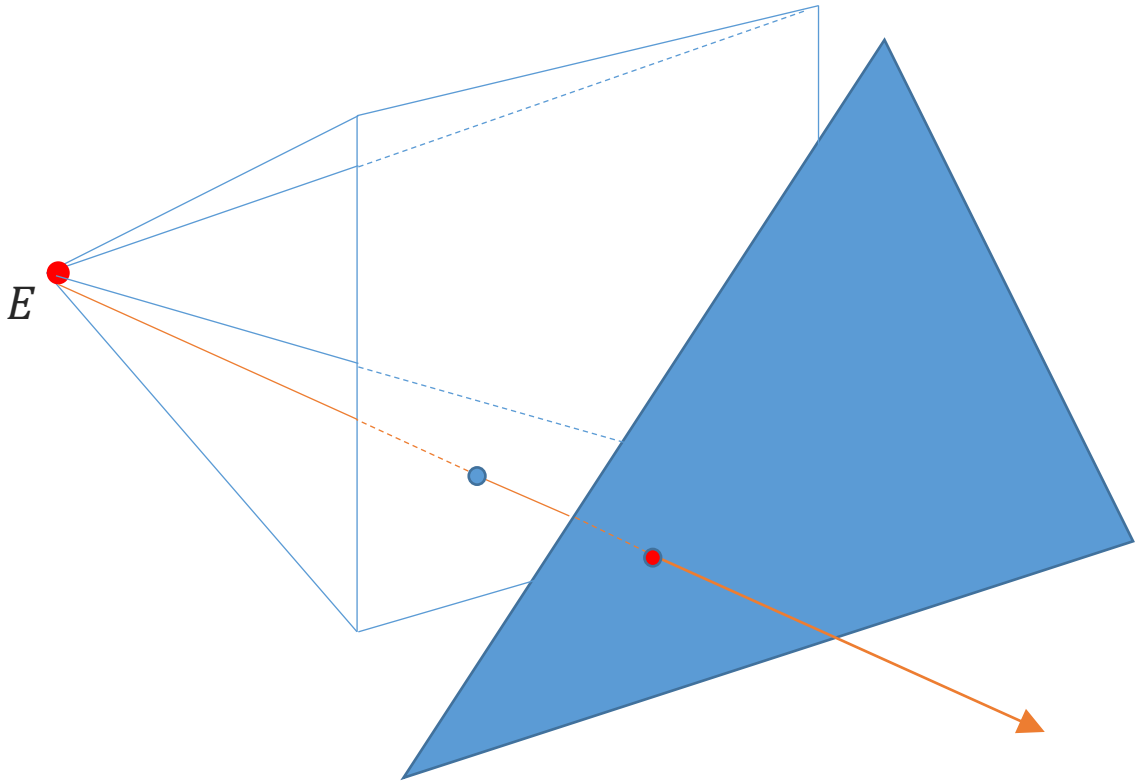
Update the ray’s t value to keep the smallest positive t among all the primitives

Ray intersection with a 3D model: triangle

1. Find the intersection point $P(t)$
   of the ray and the *plane* in which the triangle lies
   (see previous slide)

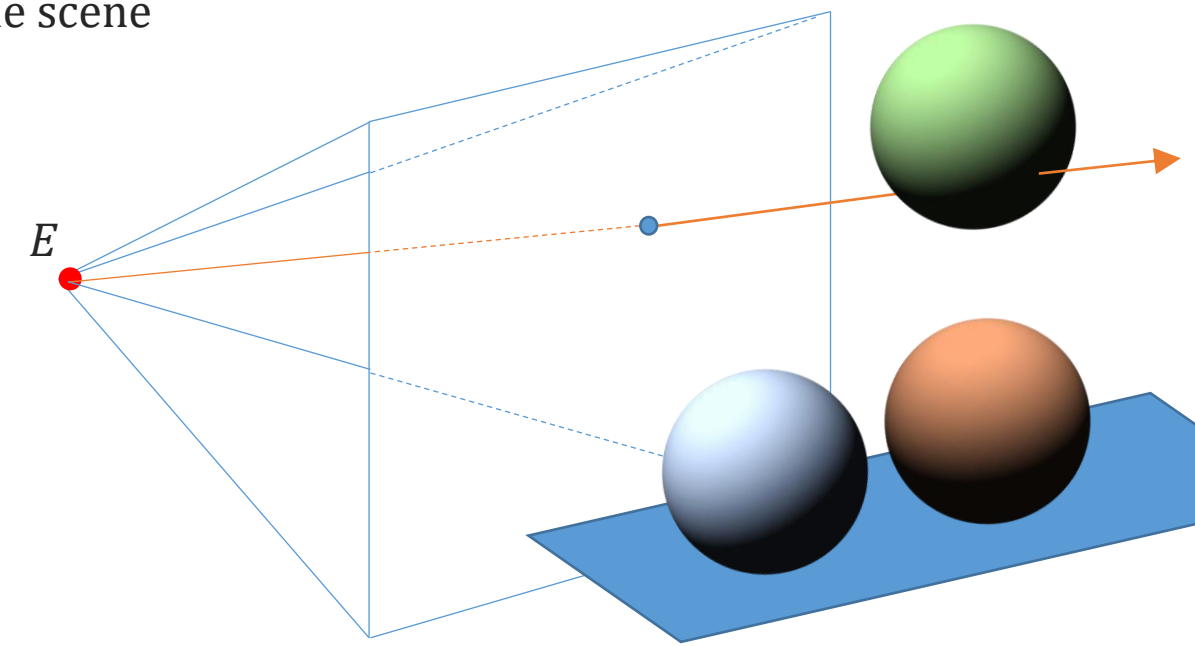
2. Check whether $P(t)$ is inside the triangle

[Optional for the practical assignment
so further details omitted until a later lecture]



Ray intersection with a 3D model: sphere

Given a ray $P(t) = E + t\vec{d}$, we determine the smallest positive intersection distance t
by intersecting the ray with each of the primitives in the scene



Sphere:
Center C and radius r

Intersection equation (implicit form)

$$\|P(t) - C\|^2 = r^2$$
$$(P(t) - C) \cdot (P(t) - C) = r^2$$
$$\left(E + t\vec{d} - C\right) \cdot \left(E + t\vec{d} - C\right) = r^2$$

Solve for the values of t

$$at^2 + bt + c = 0 \qquad t_{1,2} = \frac{-b \pm \sqrt{b^2 + 4ac}}{2a}$$

Find the value of t that corresponds to the visible intersection
(smallest positive t) and update the ray’s t value

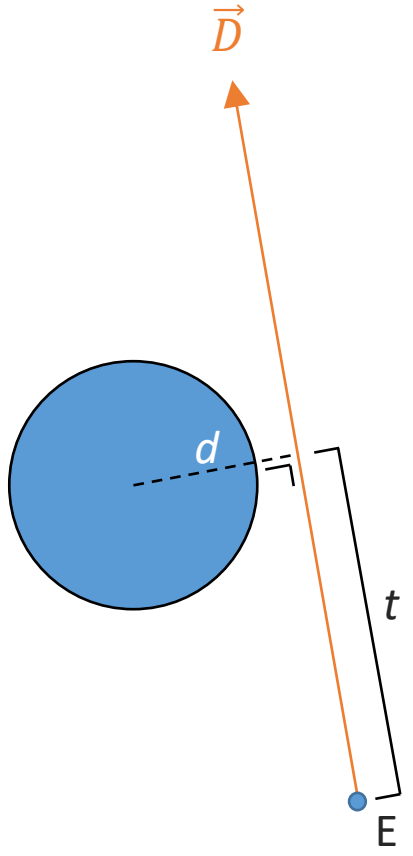
Ray intersection with a 3D model: sphere

Ray–sphere intersection:

a) Implement the math from the previous slide, carefully:
   Divide by a: Can $a = 0$? When? What to do?
   Square root of discriminant: Can discriminant < 0? When? What to do?
   – or –
b) Implement the following geometric interpretation

```
t = t-value of orthogonal projection of sphere center onto ray
d = shortest distance between ray and sphere center
if d > sphere radius:
    no intersection
else:
    t = t – a small step back
    if t > 0 and t is closer than any other intersection:
        t is the new closest intersection
```

Note: This pseudocode only works for rays that start outside the sphere. Why? How to fix that?

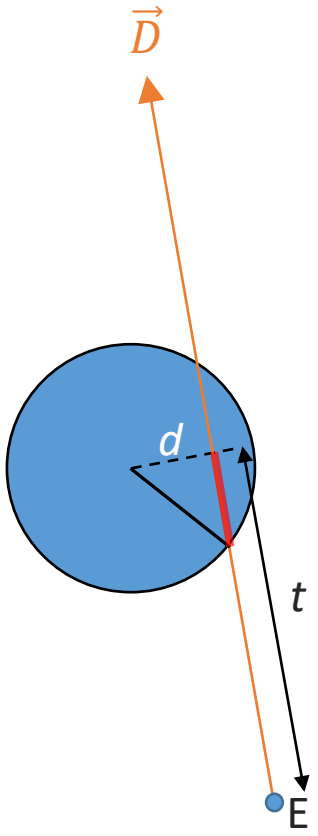


Ray intersection with a 3D model: sphere

Ray–sphere intersection:

a) Implement the math from the previous slide, carefully:
   Divide by a: Can a = 0? When? What to do?
   Square root of discriminant: Can discriminant < 0? When? What to do?
   – or –
b) Implement the following geometric interpretation

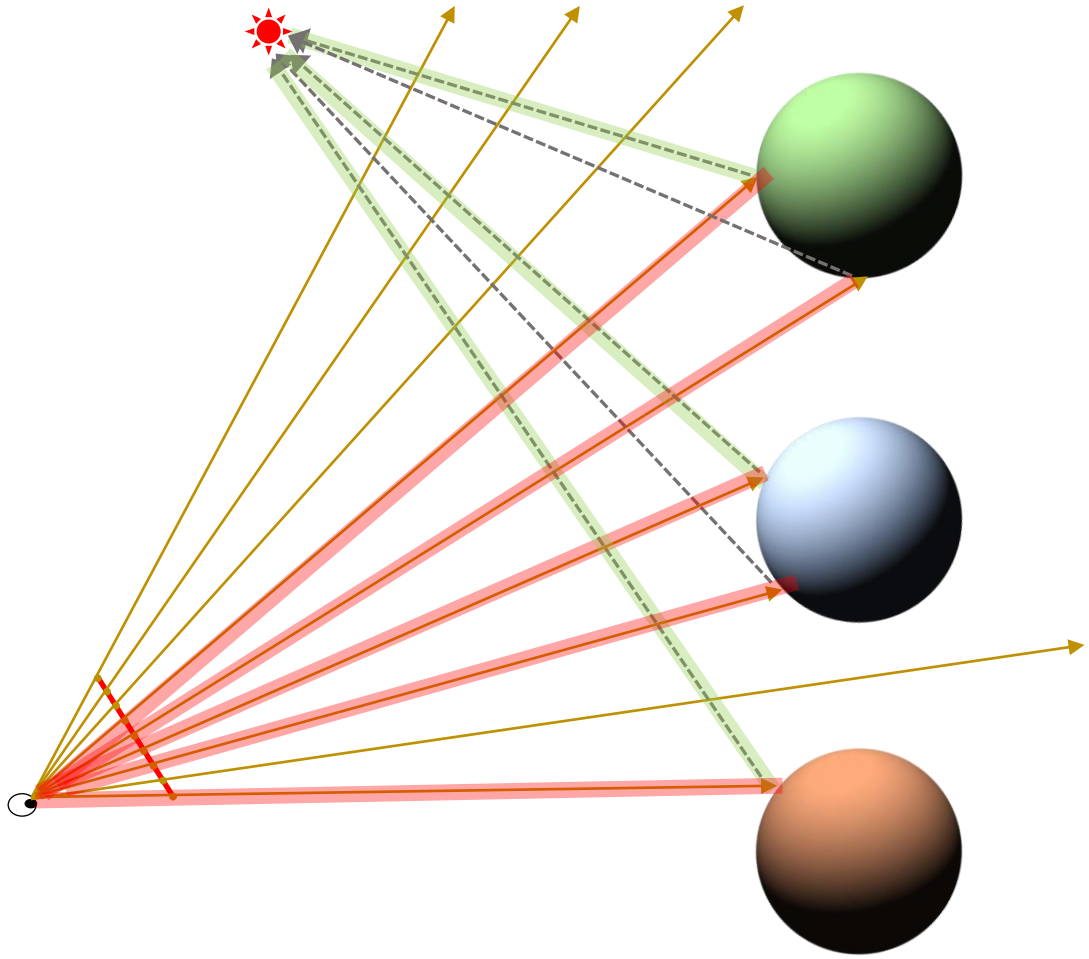
```
t = t-value of orthogonal projection of sphere center onto ray
d = shortest distance between ray and sphere center
if d > sphere radius:
    no intersection
else:
    t = t – a small step back (Pythagoras)
    if t > 0 and t is closer than any other intersection:
        t is the new closest intersection
```



Note: This pseudocode only works for rays that start outside the sphere. Why? How to fix that?

Today's Agenda:

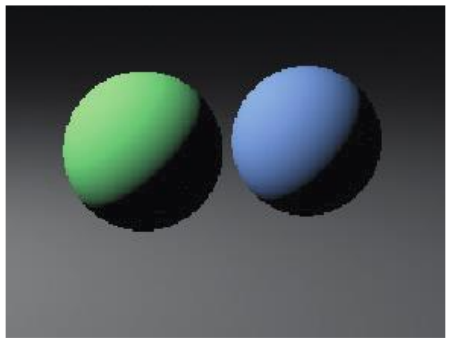
- Rasters (Sections 3.1 and 3.2)
- Colors (Section 3.3)
- Ray Tracing (Section 4.1 – 4.4)
  - Camera setup
  - Primary rays
  - Shadow rays
  - Intersection with planes and spheres



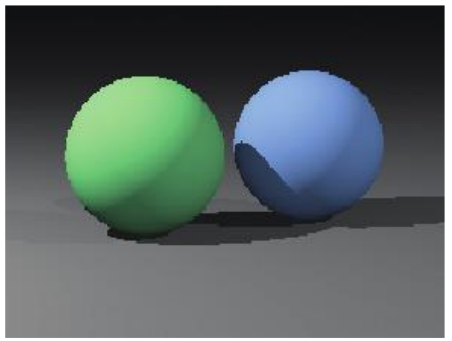
* book sections in Fundamentals of Computer Graphics, editions 3 – 5

Your ray tracer this week:

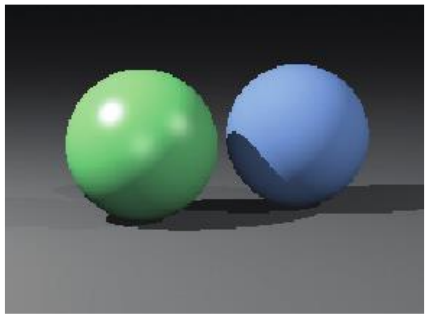
Your ray tracer next week:



Reflection: Diffuse



Shadows



Reflection: Specular

Materials used

- CSC 433 Computer Graphics at University of Arizona, Joshua Levine
  https://jalevine.bitbucket.io/csc433/
- “Fundamentals of Computer Graphics” Steve Marschner, Peter Shirley

Recommended tutorials

- Ray Tracing Topics & Techniques
  https://www.flipcode.com/archives/Raytracing_Topics_Techniques-Part_1_Introduction.shtml
- Ray Tracing in One Weekend
  https://raytracing.github.io/