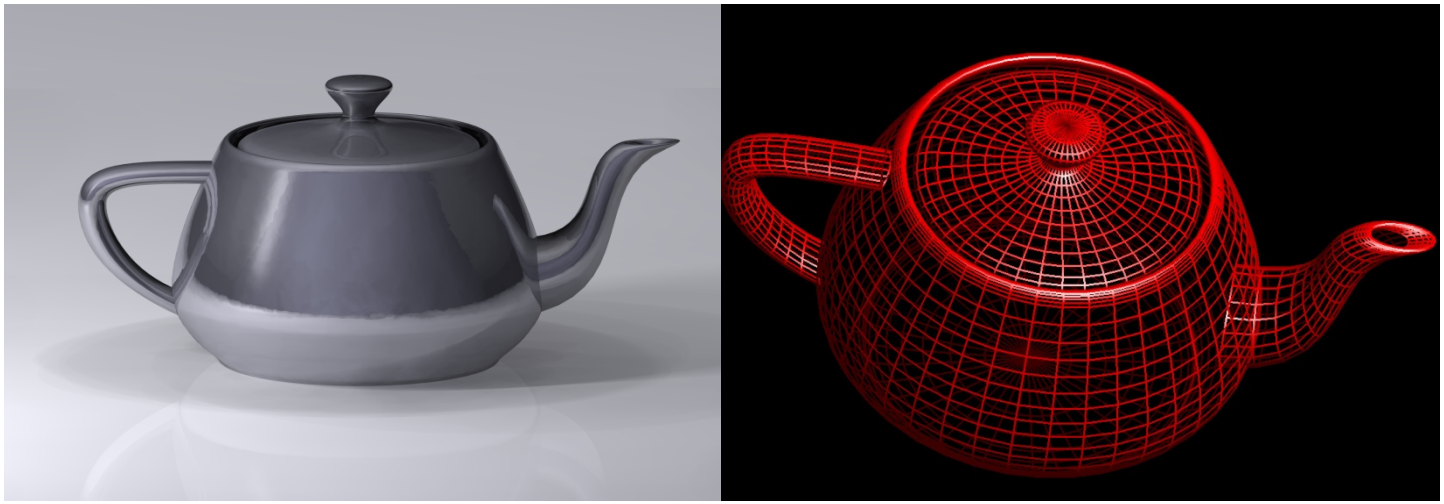


Foundations of Computer Graphics

Saurabh Ray

We will start at 16:07.

Tuesday, April 7.

The exam will be an assignment on NYU classes that starts at 16:00 and closes at 17:25.

You can submit pictures of your handwritten solutions.

Topics. Everything until lecture 14 except 'procedural noise'.

Please practice the exams from past offerings (on NYU classes).

I don't have solutions but I am happy answer questions during office hours.

Office hours. Today and on Monday from 17:30 - 19:00 on Zoom.

Today:.

- Ray Tracing.

  Reading: Sections 12.1 - 12.3 of the textbook.

- Constructive Solid Geometry.

  Reading: Section 9.10.1 of the textbook.

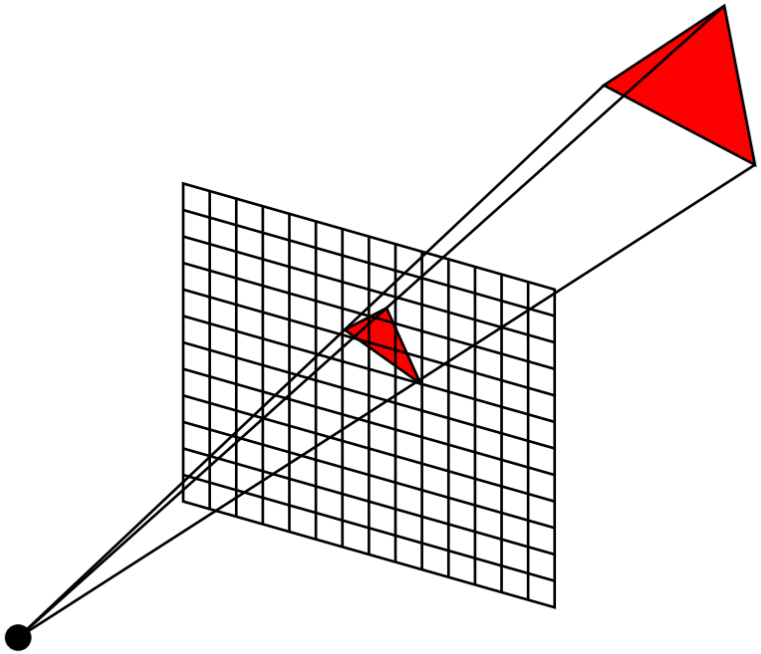
Projective Methods

- 3D triangles project to 2D triangles

- Project the three vertices

- Find pixels covered by 2D triangle

- Shade these pixels



What if > 1 triangles project onto the same pixel?

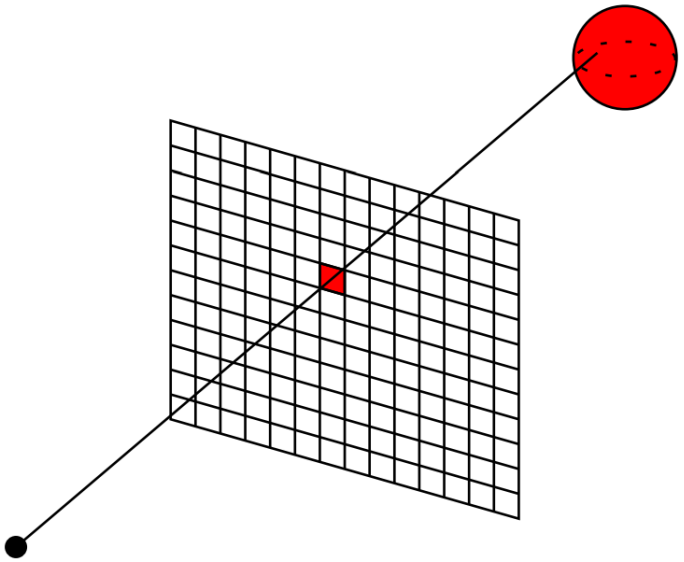
Used by OpenGL and DirectX. Fast, hardware supported, real-time.

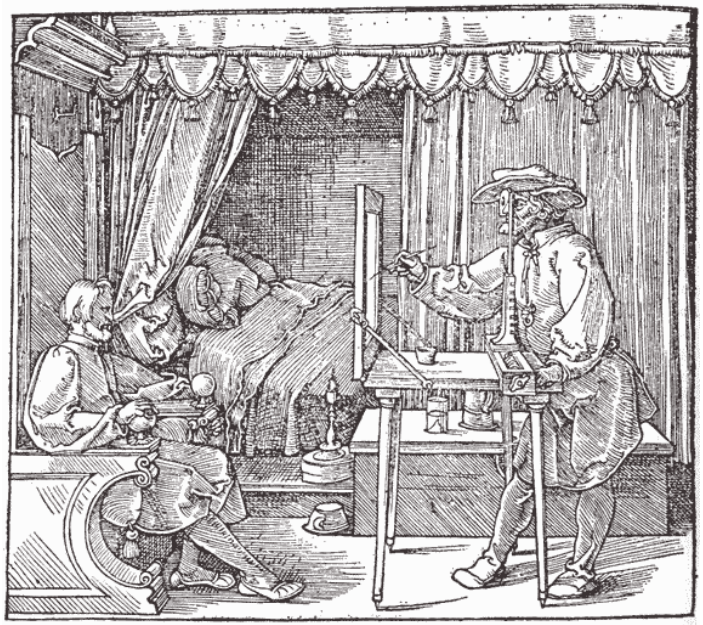
Ray Tracing

- Compute ray from the view point through the pixel center

- Determine the intersection point of the first object hit by the ray

- Calculate shading for the pixel (possibly with recursion)

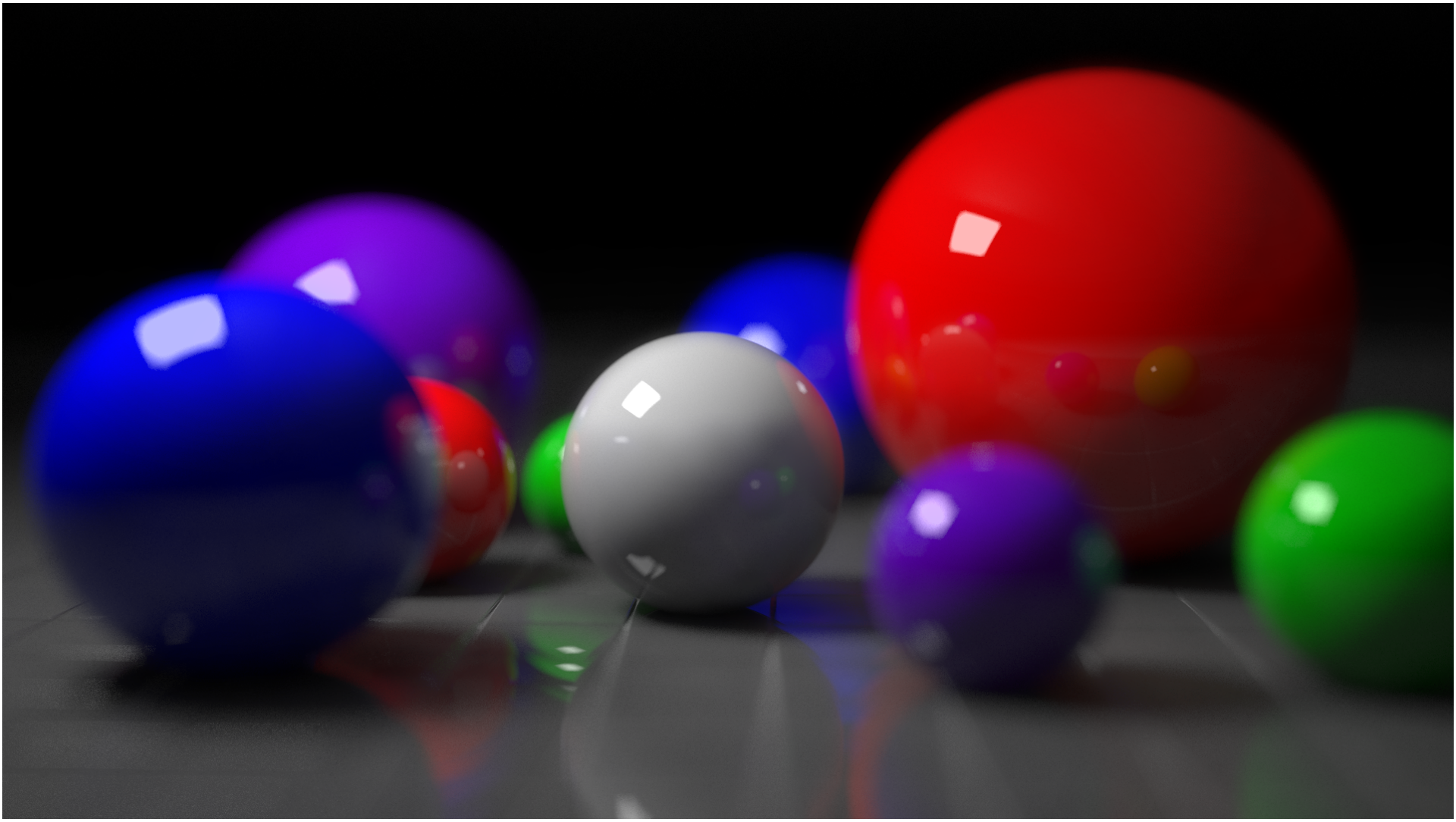




Alberti's veil

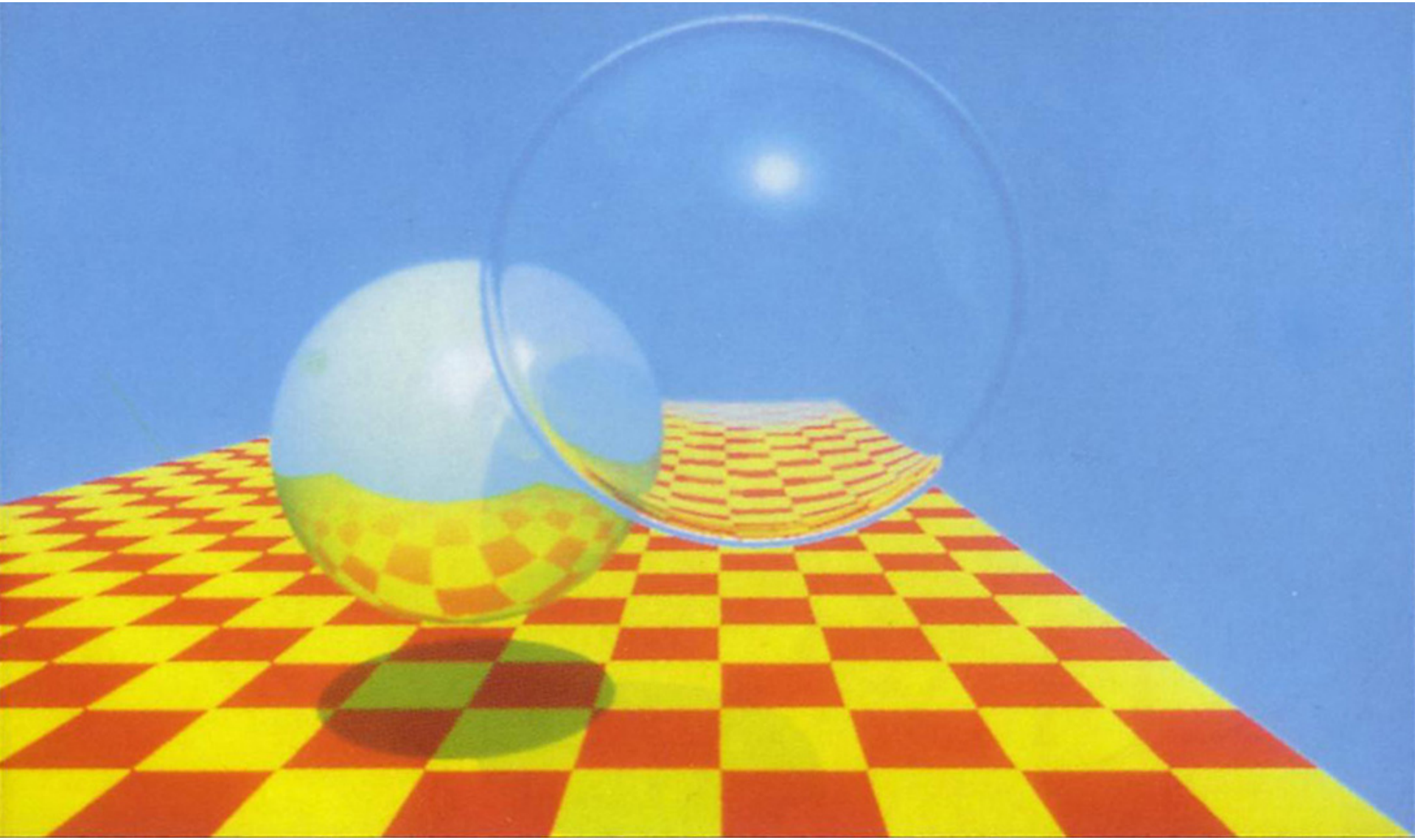
Ray Tracing

Ray Tracing



Ray Tracing

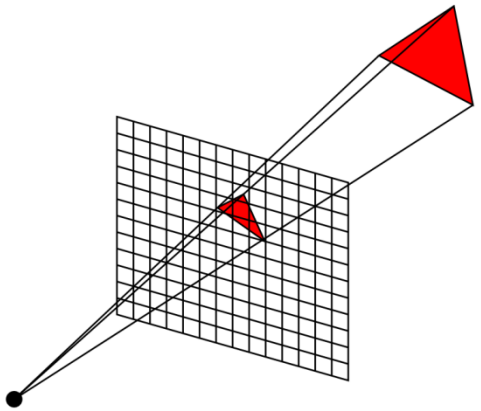
Ray Tracing



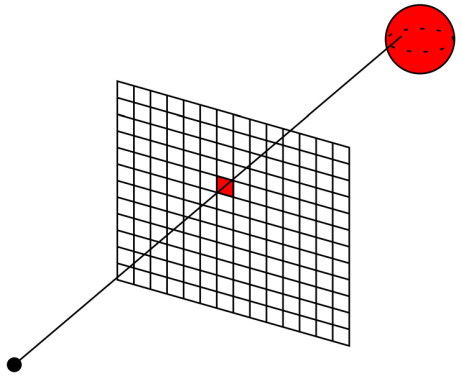
Recursive Ray Tracing, Turner Whitted, 1980

Object Order vs Image Order Rendering

```
for (each object) {
      find and update all pixels
      that it influences ;
}
```



```
for (each pixel) {
    find all objects influencing it ;
    update pixel accordingly ;
}
```



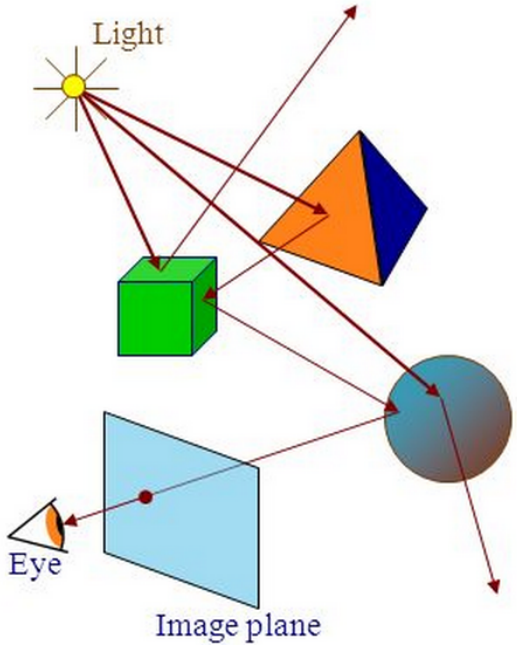
Forward vs Backward Ray Tracing

Why don't we trace the rays emanating from the light source through the various scene interactions and see how each pixel is affected?

Most rays don't even get close to the eye!

Wasted computation.

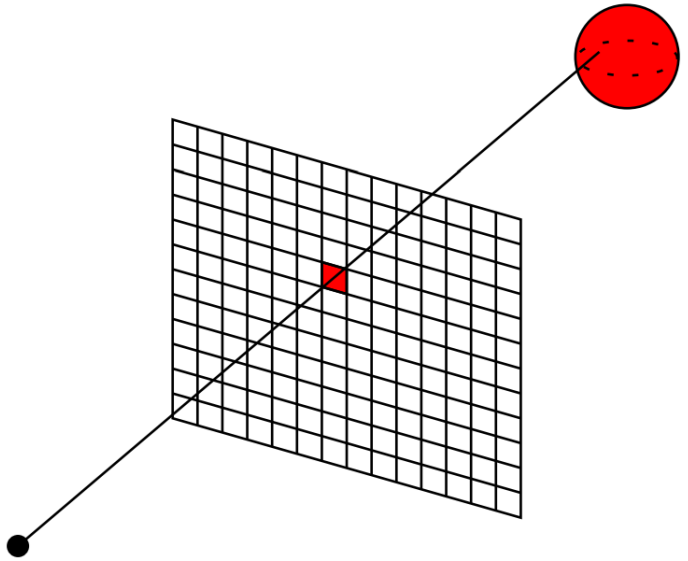
This is why we do backward ray-tracing.



Ray Casting

Basic Ray Tracing Algorithm:

```
for (each pixel) {
  - find 1st object hit by
    ray and surface normal n⃗
  - set pixel color to value
    computed from hit point,
    light, and n⃗
}
```



these rays are called **cast rays**

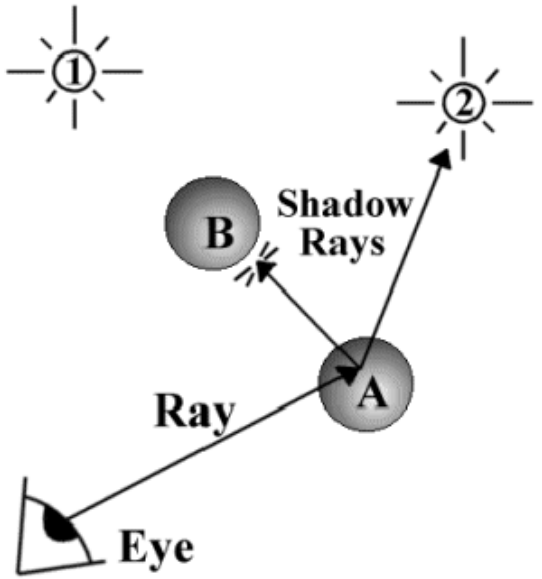
Ray Casting

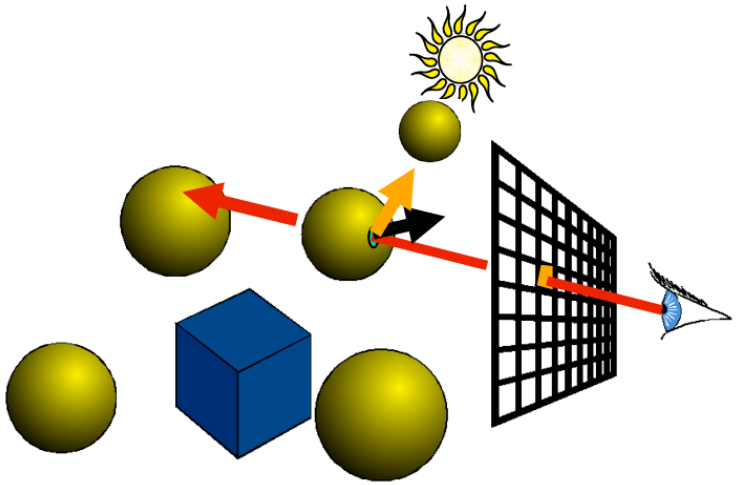
A naive implementation:

```
for (every pixel) {

     Construct a ray from the eye;

     for(every object){

          Find intersection with the ray;
          Keep if closest;
     }

     Shade depending on light and normal vector
}
```

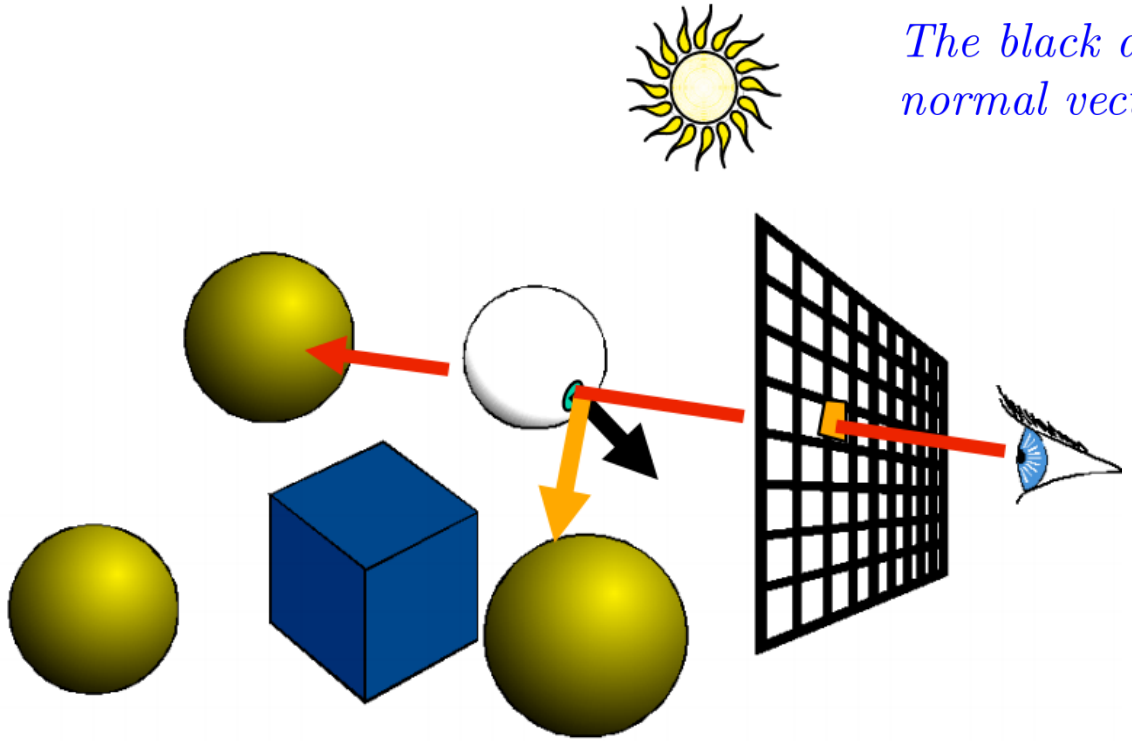
Shadows

```
Find the point to be shaded;

for (every light source) {

   Construct a ray from the point to the light source;

   for (every object) {
       Find the intersection of the ray with the object;
   }

   if (no object between point and light source){
       Add contribution from the light source;
   }
}
```





Reflection

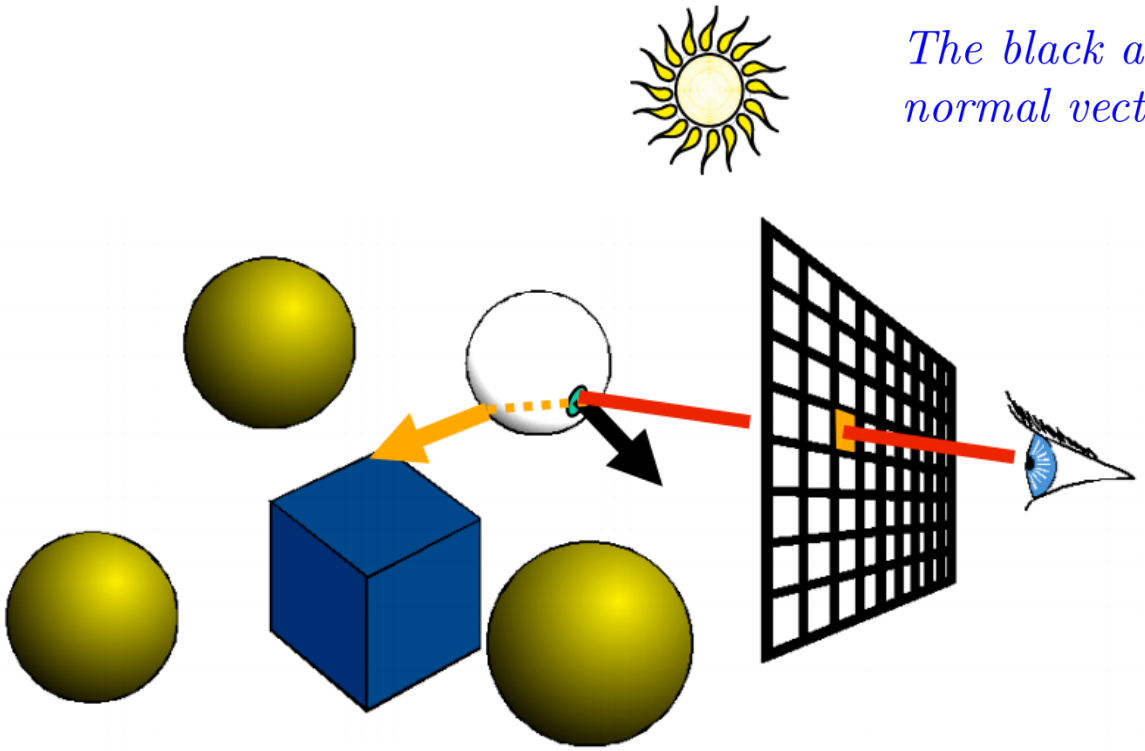


The black arrow is the normal vector.

Construct reflected ray (symmetric to the normal)

Multiply by reflection coefficient

Refraction



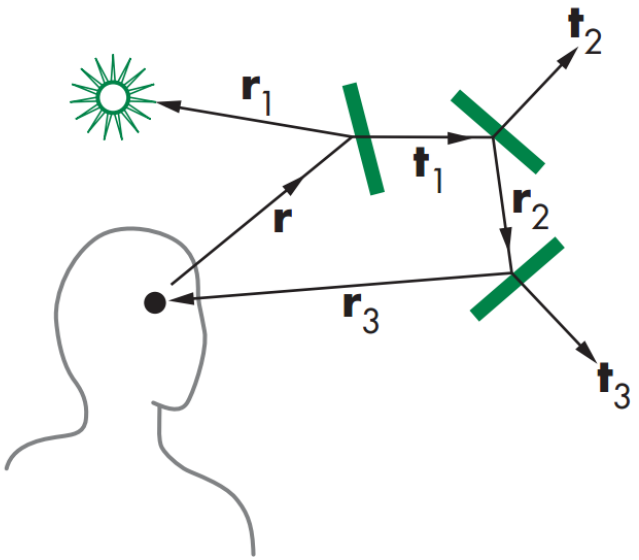
The black arrow is the normal vector.

Construct refracted ray (using refraction coefficient)

Multiply by transparency coefficient

Ray Tracing

Some surfaces can both reflect and refract.



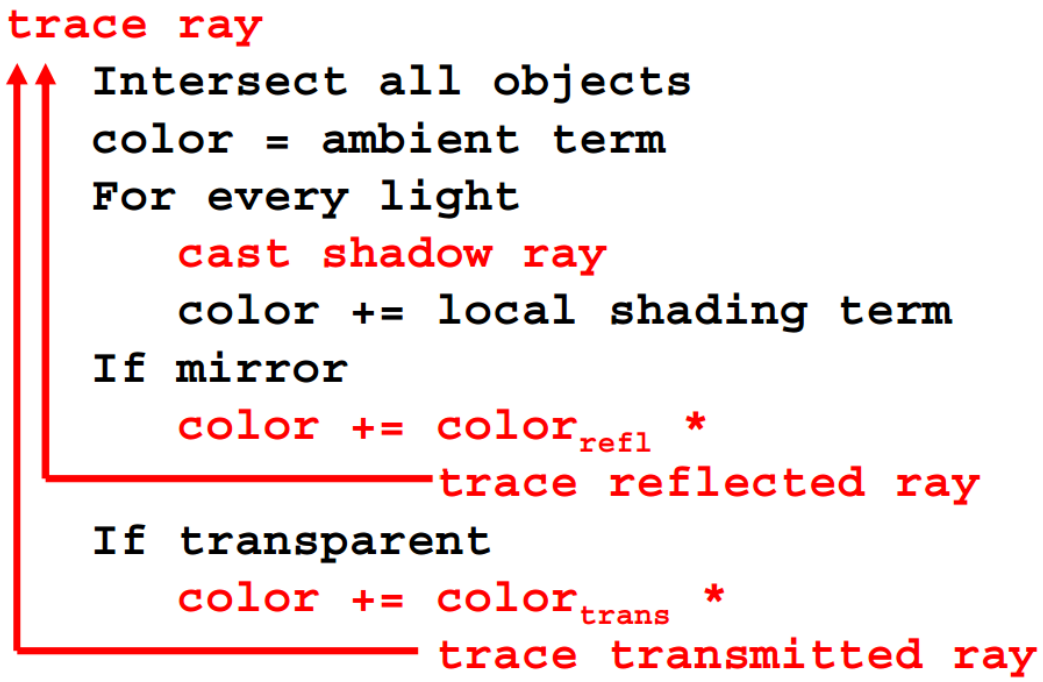
At every incidence, some part of the ray is:

- absorbed
- scattered (diffuse component)
- reflected
- refracted

We don't trace the scattered rays: too many rays!

These are traced recursively

Recursive Ray Tracing



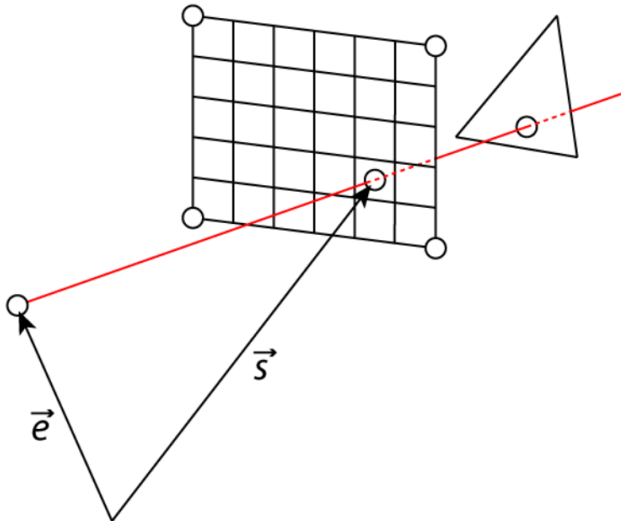
Stopping Criteria:

- *Recursion depth (stop after a fixed no. of bounces)*
- *Ray contribution (stop when the reflected or transmitted contribution is too small)*

Ray Tracing

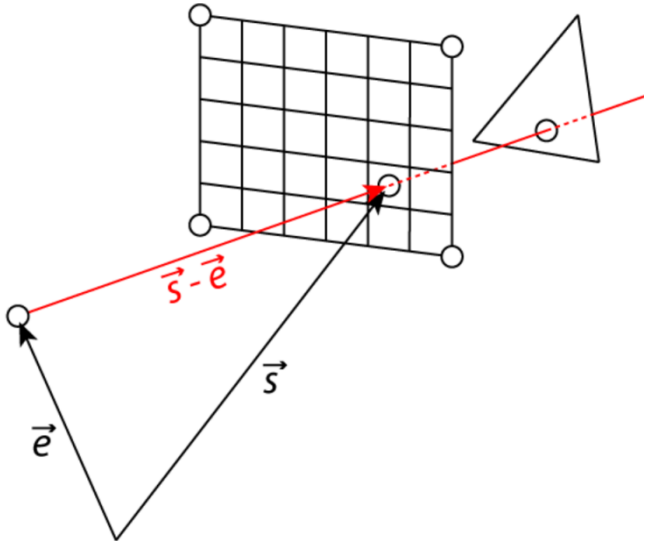
We need to shoot a ray

- from the view point $\vec{e}$
- through a pixel $\vec{s}$
- compute where the ray intersects the objects



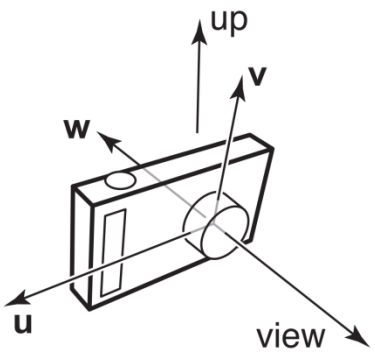
Parametric equation of the ray?

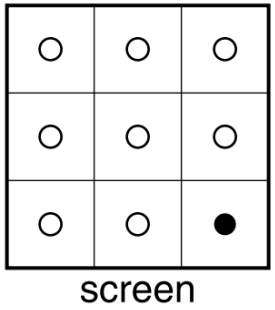
$$\vec{p}(t) = \vec{e} + t\ (\vec{s} - \vec{e})$$

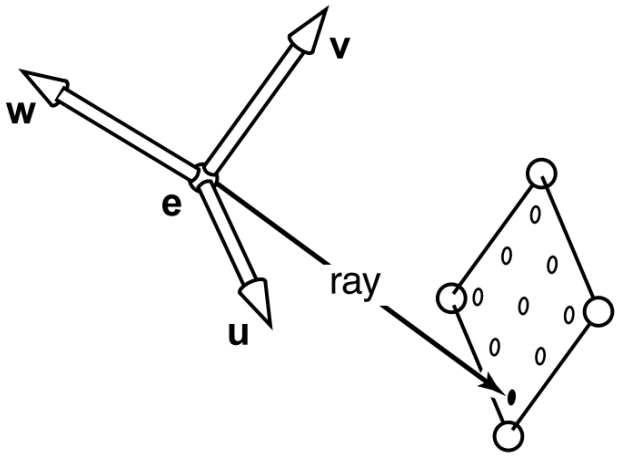


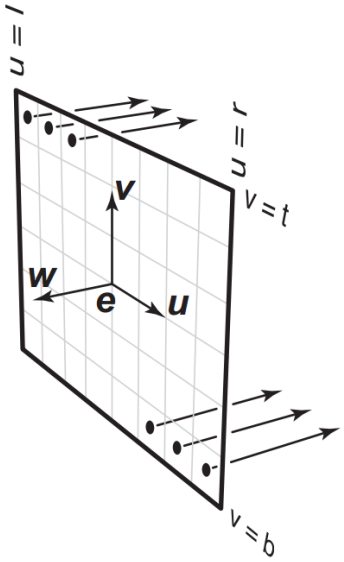
Ray Tracing

Things are simpler if we use the camera coordinate system basis vectors.

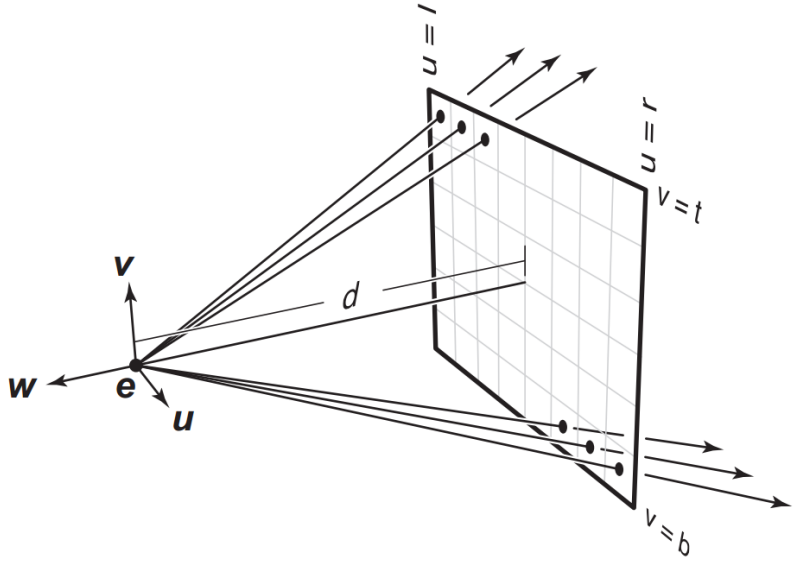








Parallel projection
same direction, different origins



Perspective projection
same origin, different directions

Orthographic View

Image is $n_x \times n_y$ pixels

- All rays have direction $-\vec{w}$

- For $(i, j)^{th}$ pixel:

  Position in basis $(\vec{u}, \vec{v})$:

$$u = l + (r - l)(i + 0.5)/n_x$$
$$v = b + (t - b)(j + 0.5)/n_y$$

  Ray origin: $\vec{e} + u\vec{u} + v\vec{v}$

  Ray: $\vec{e} + u\vec{u} + v\vec{v} + \alpha(-\vec{w})$

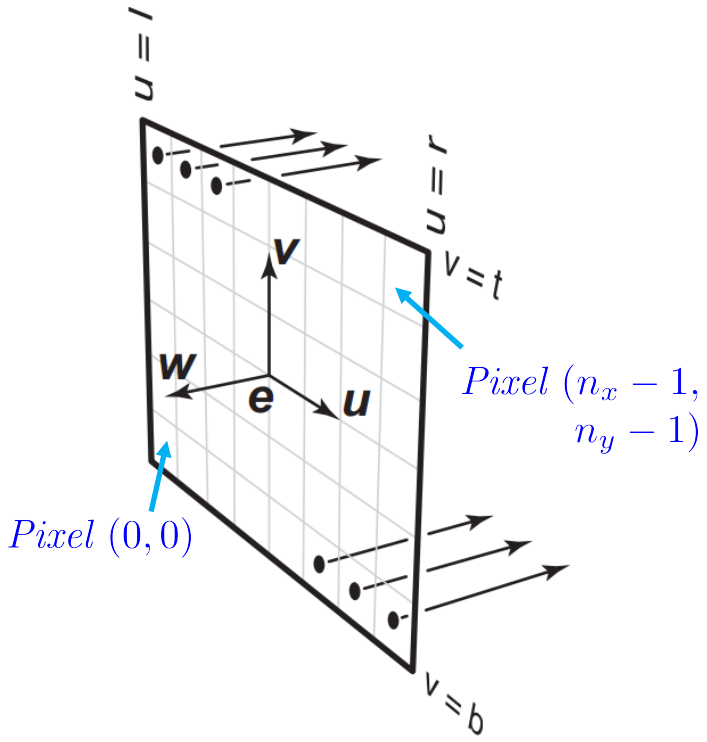


Image is $n_x \times n_y$ pixels

Perspective View

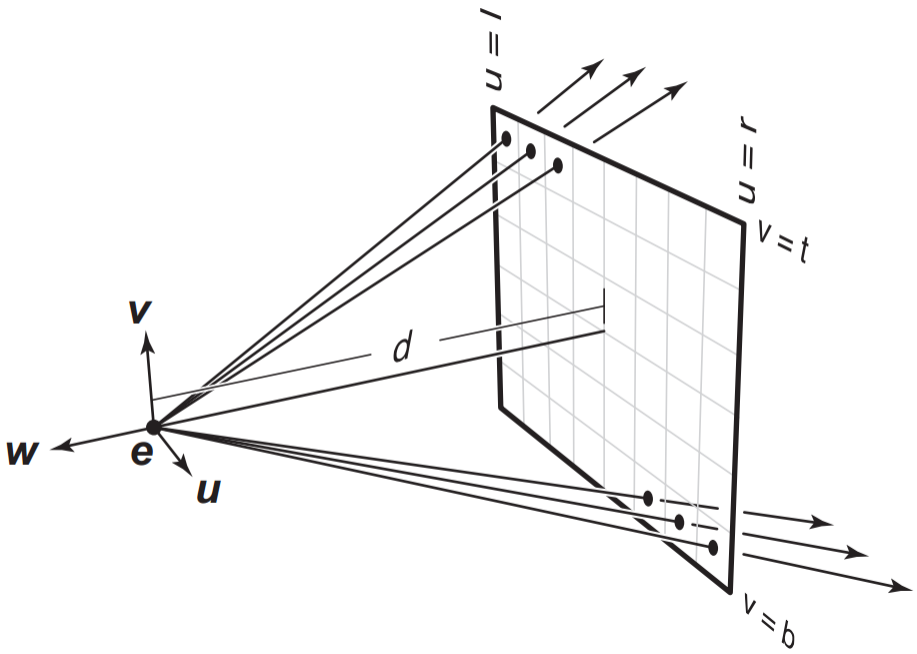
- Ray Origin: $\vec{e}$ for all pixels

- For $(i, j)^{th}$ pixel:

$$u = l + (r - l)(i + 0.5)/n_x$$
$$v = b + (t - b)(j + 0.5)/n_y$$
$$w = -d$$

Ray direction: $u\vec{u} + v\vec{v} - d\vec{w}$

Ray: $\vec{e} + \alpha(u\vec{u} + v\vec{v} - d\vec{w})$

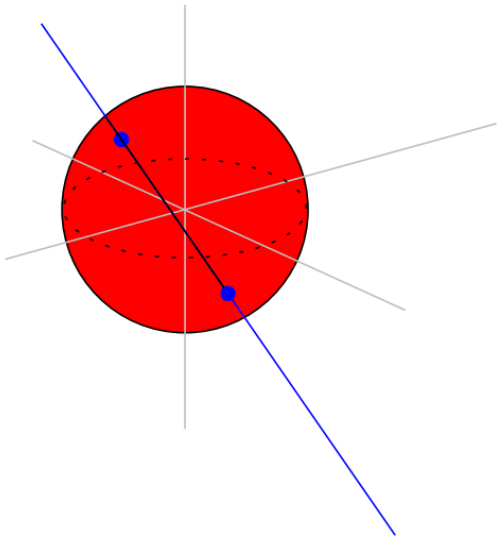


Ray Tracing

Once we have generated the ray, we need to figure out where it intersects objects.

Suppose that we have a sphere with center $\vec{c} = (x_c, y_c, z_c)$ and radius R.

A point $\vec{p}$ lies on the sphere iff its distance from the center is R.



So, the implicit equation is: $(\vec{p} - \vec{c}) \cdot (\vec{p} - \vec{c}) = R^2$.

Equation of ray: $\vec{e} + t\vec{d}$.

Intersection points satisfy:

$$(\vec{e} + t\vec{d} - \vec{c}) \cdot (\vec{e} + t\vec{d} - \vec{c}) = R^2.$$

Ray Tracing

Intersection points satisfy:

$$(\vec{e} + t\vec{d} - \vec{c}) \cdot (\vec{e} + t\vec{d} - \vec{c}) = R^2.$$

Same as:

$$(\vec{d} \cdot \vec{d})t^2 + 2\vec{d} \cdot (\vec{e} - \vec{c})t + (\vec{e} - \vec{c}) \cdot (\vec{e} - \vec{c}) - R^2 = 0.$$

Quadratic equation in t. $0, 1$ *or* 2 *real solutions.*

We consider only the solutions with $t \geq 0$.

Ray Tracing

How do we figure out the intersection of the ray and a triangle?

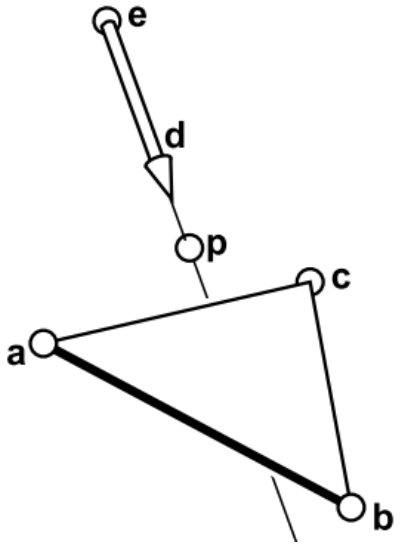
Intersection point $\vec{p} = \vec{e} + t\vec{d}$ satisfies:

$$\vec{e} + t\vec{d} = \vec{a} + \beta(\vec{b} - \vec{a}) + \gamma(\vec{c} - \vec{a})$$

$$\begin{aligned} x_e + tx_d &= x_a + \beta(x_b - x_a) + \gamma(x_c - x_a), \\ y_e + ty_d &= y_a + \beta(y_b - y_a) + \gamma(y_c - y_a), \\ z_e + tz_d &= z_a + \beta(z_b - z_a) + \gamma(z_c - z_a). \end{aligned}$$

$$\underbrace{\begin{bmatrix} x_a - x_b & x_a - x_c & x_d \\ y_a - y_b & y_a - y_c & y_d \\ z_a - z_b & z_a - z_c & z_d \end{bmatrix}}_{A} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} x_a - x_e \\ y_a - y_e \\ z_a - z_e \end{bmatrix}$$

Various ways of solving this.



The ray hits the plane containing the triangle at a point p

$$\beta = \frac{\begin{vmatrix} x_a - x_e & x_a - x_c & x_d \\ y_a - y_e & y_a - y_c & y_d \\ z_a - z_e & z_a - z_c & z_d \end{vmatrix}}{|\mathbf{A}|}$$

$$\gamma = \frac{\begin{vmatrix} x_a - x_b & x_a - x_e & x_d \\ y_a - y_b & y_a - y_e & y_d \\ z_a - z_b & z_a - z_e & z_d \end{vmatrix}}{|\mathbf{A}|}$$

$$t = \frac{\begin{vmatrix} x_a - x_b & x_a - x_c & x_a - x_e \\ y_a - y_b & y_a - y_c & y_a - y_e \\ z_a - z_b & z_a - z_c & z_a - z_e \end{vmatrix}}{|\mathbf{A}|}$$

$|M|$ denotes the determinant of the matrix M

Once we computed these, how do we know that the intersection is inside the triangle?

It is inside the triangle iff : $t, \beta, \gamma \geq 0$ and $\beta + \gamma \leq 1$

Ray Tracing

POV Ray: Persistence of Vision Ray Tracer

Scene is described by a Scene Description Language (SDL)

Generally uses mathematical definitions of objects rather than meshes.

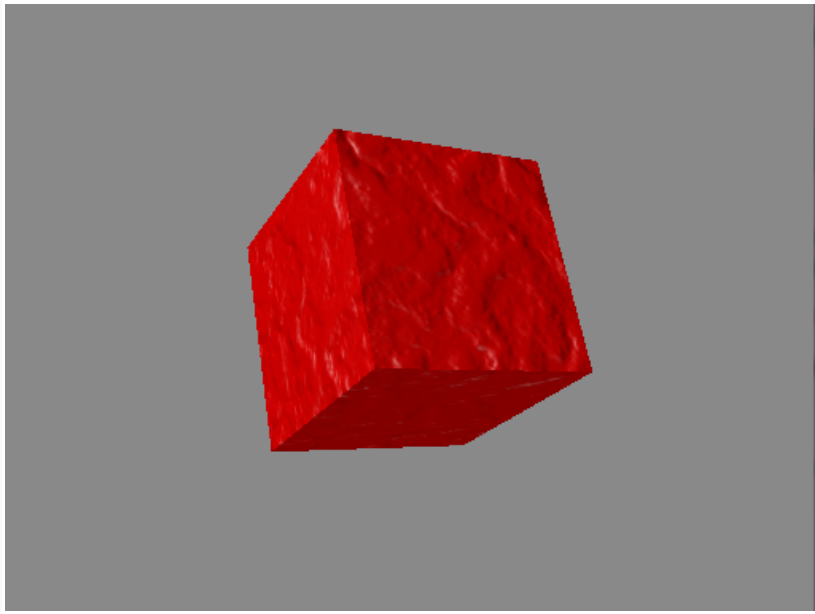
E.g. A sphere is described by a center and a radius instead of a bunch of triangles.

It does allow meshes.

Ray Tracing

```
#version 3.6;
//Includes a separate file defining a number of common colours
#include "colors.inc"
global_settings { assumed_gamma 1.0 }

//Sets a background colour for the image (dark grey)
background   { color rgb <0.25, 0.25, 0.25> }

//Places a camera
//direction : Sets, among other things, the field of view of the camera
//right: Sets the aspect ratio of the image
//look_at: Tells the camera where to look
camera       { location  <0.0, 0.5, -4.0>
               direction 1.5*z
               right     x*image_width/image_height
               look_at   <0.0, 0.0, 0.0> }

//Places a light source
//color : Sets the color of the light source (white)
//translate : Moves the light source to a desired location
light_source { <0, 0, 0>
               color rgb <1, 1, 1>
               translate <-5, 5, -5> }
//Places another light source
//color : Sets the color of the light source (dark grey)
//translate : Moves the light source to a desired location
light_source { <0, 0, 0>
               color rgb <0.25, 0.25, 0.25>
               translate <6, -6, -6> }

//Sets a box
//pigment : Sets a color for the box ("Red" as defined in "colors.inc")
//finish  : Sets how the surface of the box reflects light
//normal  : Sets a bumpiness for the box using the "agate" in-built model
//rotate : Rotates the box
box          { <-0.5, -0.5, -0.5>,
               <0.5, 0.5, 0.5>
               texture { pigment { color Red }
                         finish  { specular 0.6 }
                         normal  { agate 0.25 scale 1/2 }
                       }
               rotate <45,46,47> }
```

Example Scene



Instancing

Object —*Transformation M*→ Transformed object

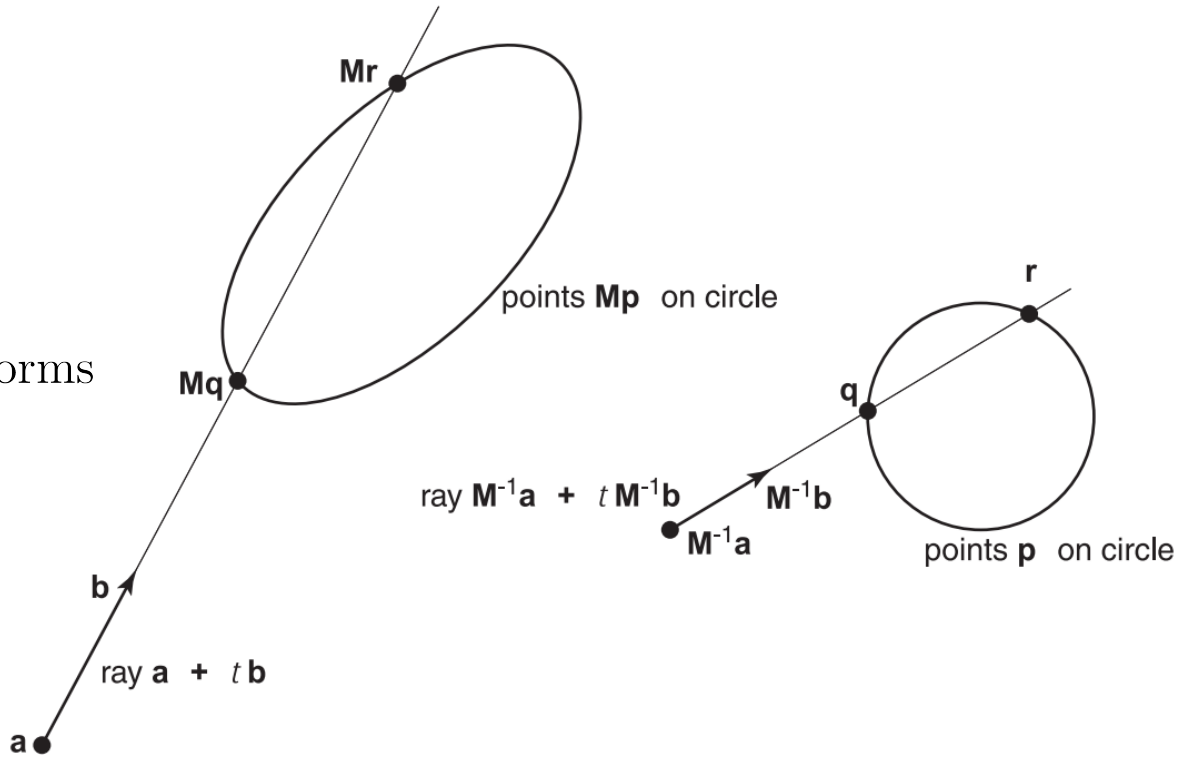
We need to intersect the transformed object with a ray r.

Instead we intersect the untransformed object with an *inverse transformed ray*.

Advantages:

- Simpler intersection routine

- Many objects may be transforms of the same simple object

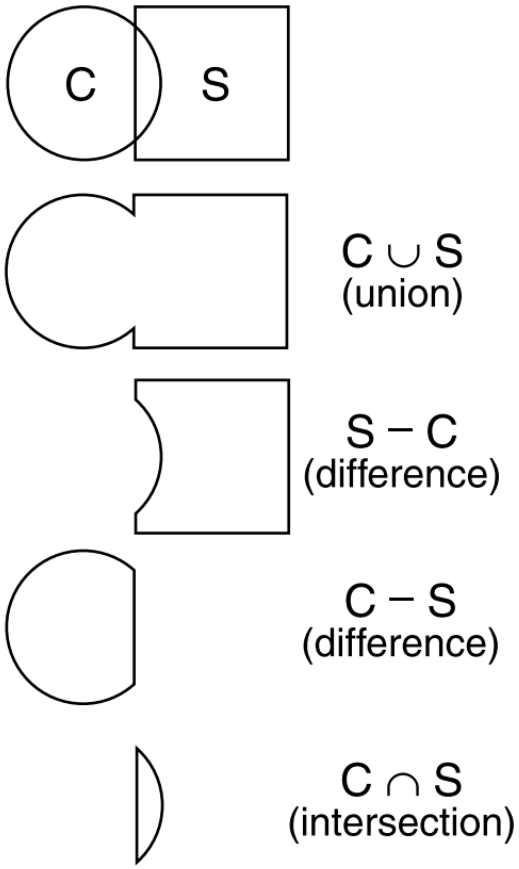


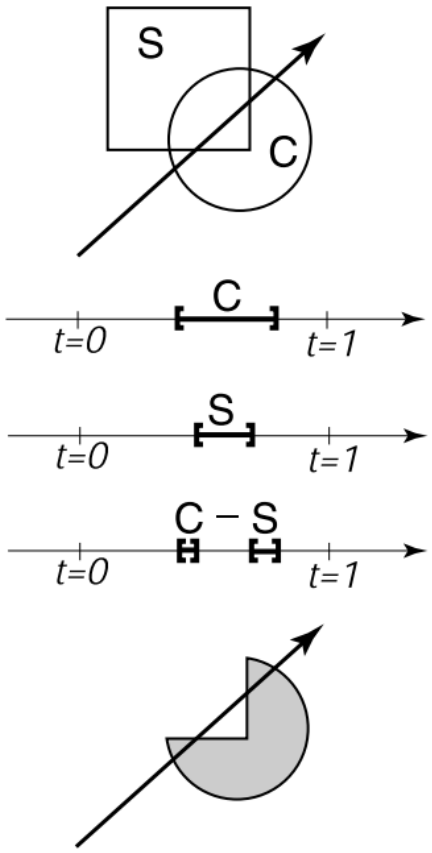
Constructive Solid Geometry

Use set operations to combine solid shapes.

We do not explicitly change the model.

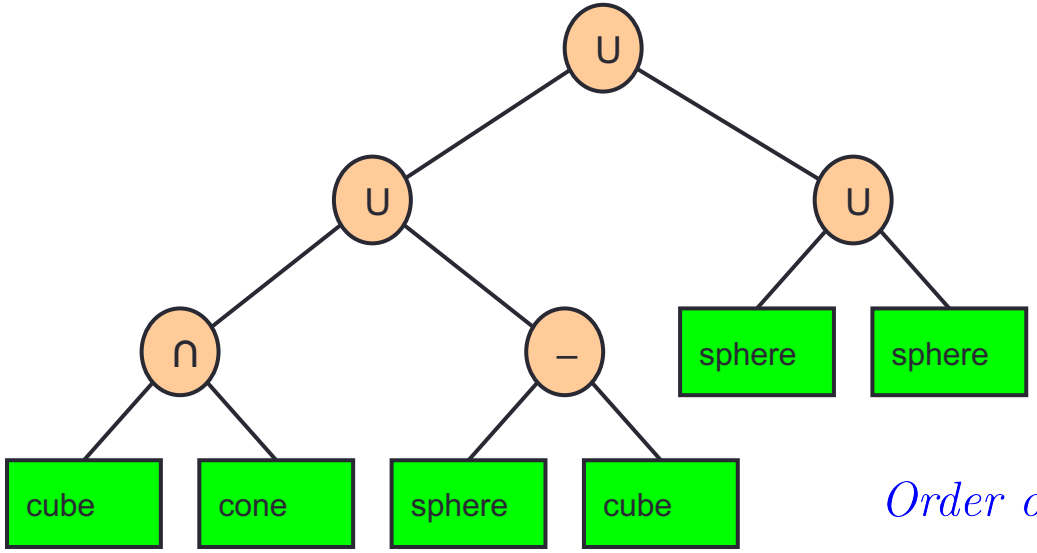
We do set operations on the intersections of a ray with the model.





Constructive Solid Geometry

An object can be represented using a tree in which the interior nodes denote set operations:

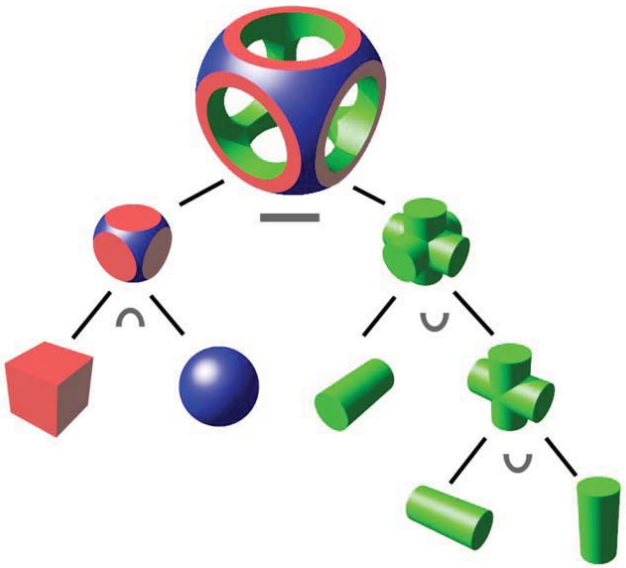


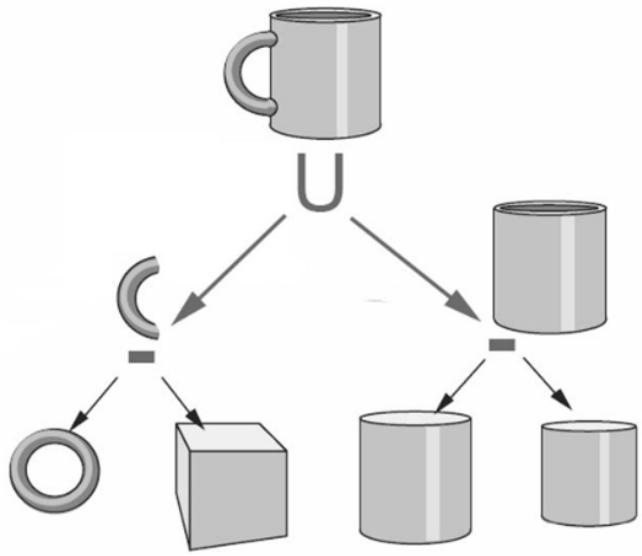
Order of children matters.

We compute the intervals for the leaves and do the set operations given by the tree on these intervals.

How do we compute the normal vector at a point on the object ?

Examples of CSG Trees:





Constructive Solid Geometry

Example CSG Model: