MIT EECS 6.837 Computer Graphics

Ray Casting II

Henrik Wann Jensen

C++

- 3 ways to pass arguments to a function
  - by value, e.g. float f(float x)
  - by reference, e.g. float f(float &x)
    - f can modify the value of x
  - by pointer, e.g. float f(float *x)
    - x here is a just a memory address
    - motivations:
      less memory than a full data structure if x has a complex type
      dirty hacks (pointer arithmetic),but just do not do it
    - clean languages do not use pointers
    - kind of redundant with reference
    - arrays are pointers

Pointers

- Can get it from a variable using &
  - often a BAD idea. see next slide
- Can be dereferenced with *
  - float *px=new float; // px is a memory address to a float
  - *px=5.0; //modify the value at the address px
- Should be instantiated with new. See next slide

Pointers, Heap, Stack

- Two ways to create objects
  - The BAD way, on the stack
    - myObject *f() {
      - myObject x;
      - ...
      - return &x
    - will crash because x is defined only locally and the memory gets de-allocated when you leave function f
  - The GOOD way, on the heap
    - myObject *f() {
      - myObject *x=new myObject;
      - ...
      - return x
    - but then you will probably eventually need to delete it

Segmentation Fault

- When you read or, worse, write at an invalid address
- Easiest segmentation fault:
  - float *px; // px is a memory address to a float
  - *px=5.0; //modify the value at the address px
  - Not 100% guaranteed, but you haven't instantiated px, it could have any random memory address.
- 2nd easiest seg fault
  - Vector<float> vx(3);
  - vx[9]=0;

Segmentation Fault

- TERRIBLE thing about segfault: the program does not necessarily crash where you caused the problem
- You might write at an address that is inappropriate but that exists
- You corrupt data or code at that location
- Next time you get there, crash

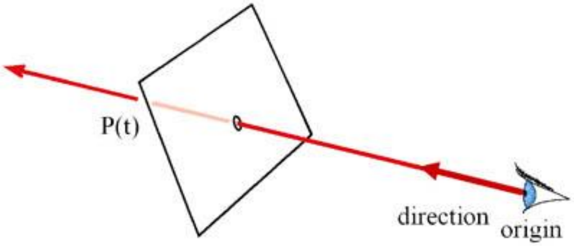
- When a segmentation fault occurs, always look for pointer or array operations before the crash, but not necessarily at the crash

Debugging

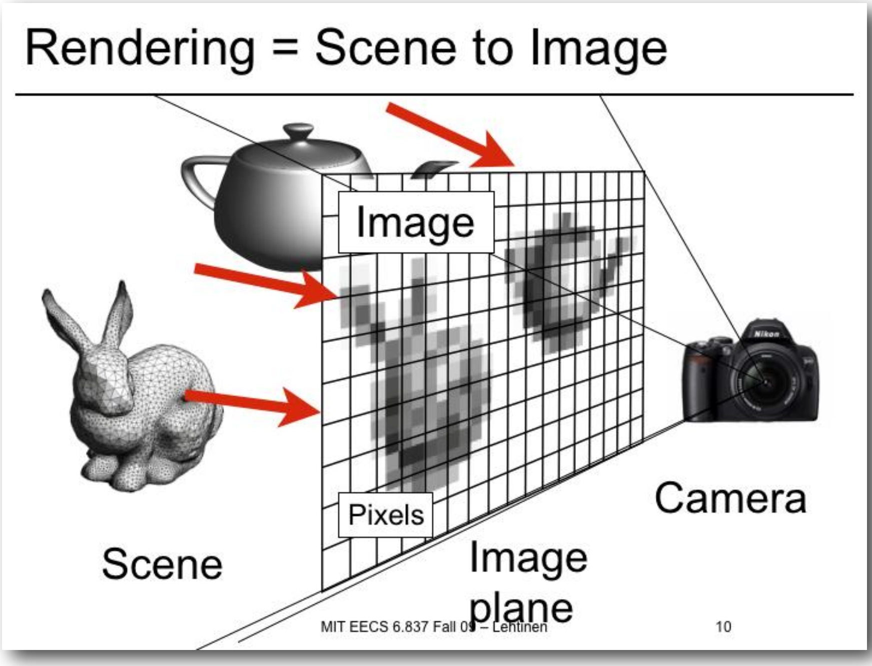
- Display as much information as you can
  - image maps (e.g. per-pixel depth, normal)
  - OpenGL 3D display (e.g. vectors, etc.)
  - cerr<< or cout<< (with intermediate values, a message when you hit a given if statement, etc.)
- Doubt everything
  - Yes, you are sure this part of the code works, but test it nonetheless
- Use simple cases
  - e.g. plane z=0, ray with direction (1, 0, 0)
  - and display all intermediate computation

Questions?

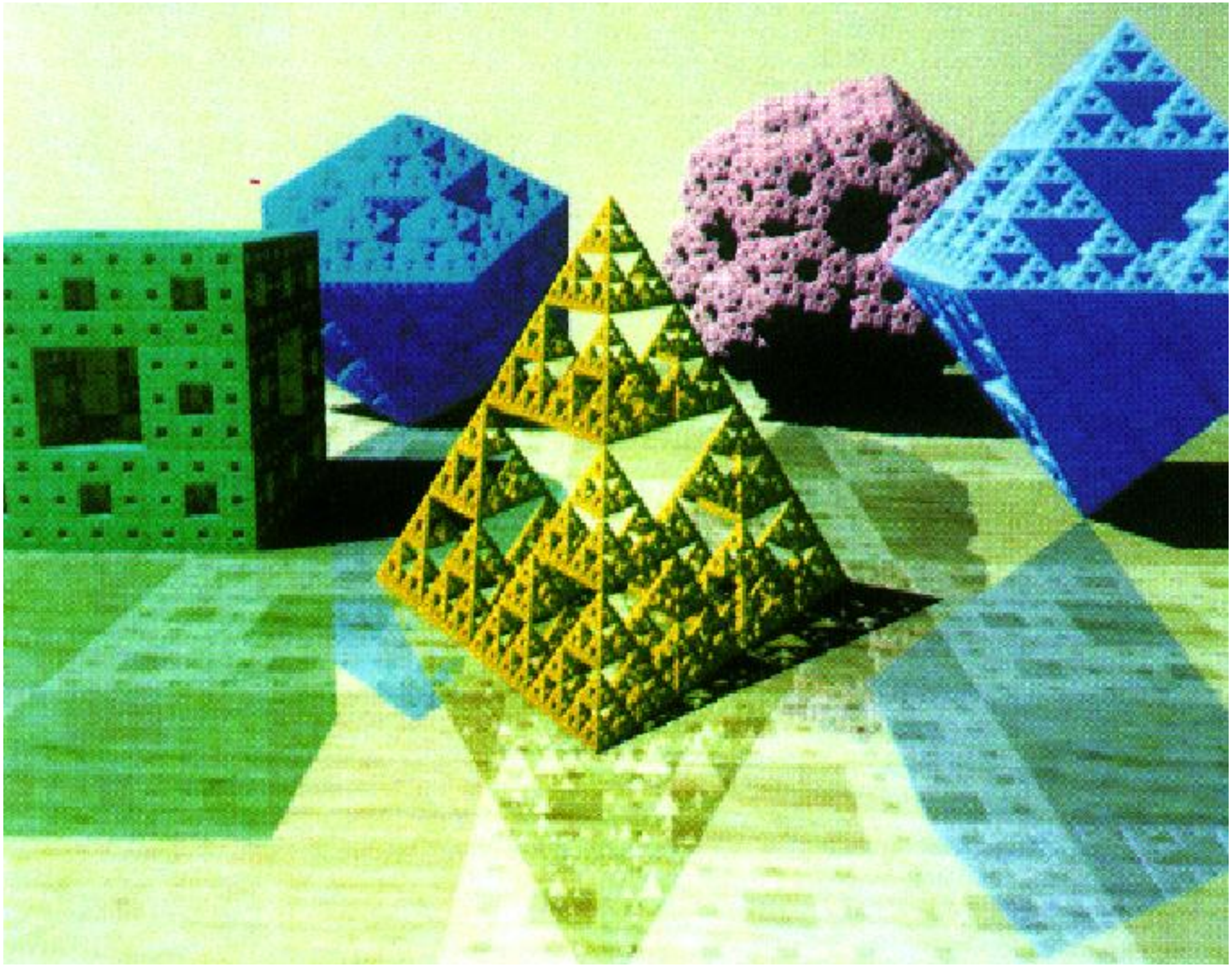
Thursday Recap



- Intro to rendering
  - Producing a picture based on scene description
  - Main variants: Ray casting/tracing vs. rasterization
  - Ray casting vs. ray tracing (secondary rays)
- Ray Casting basics
  - Camera definitions
    - Orthographic, perspective
  - Ray representation
    - P(*t*) = origin + *t* * direction
  - Ray generation
  - Ray/plane intersection
  - Ray-sphere intersection

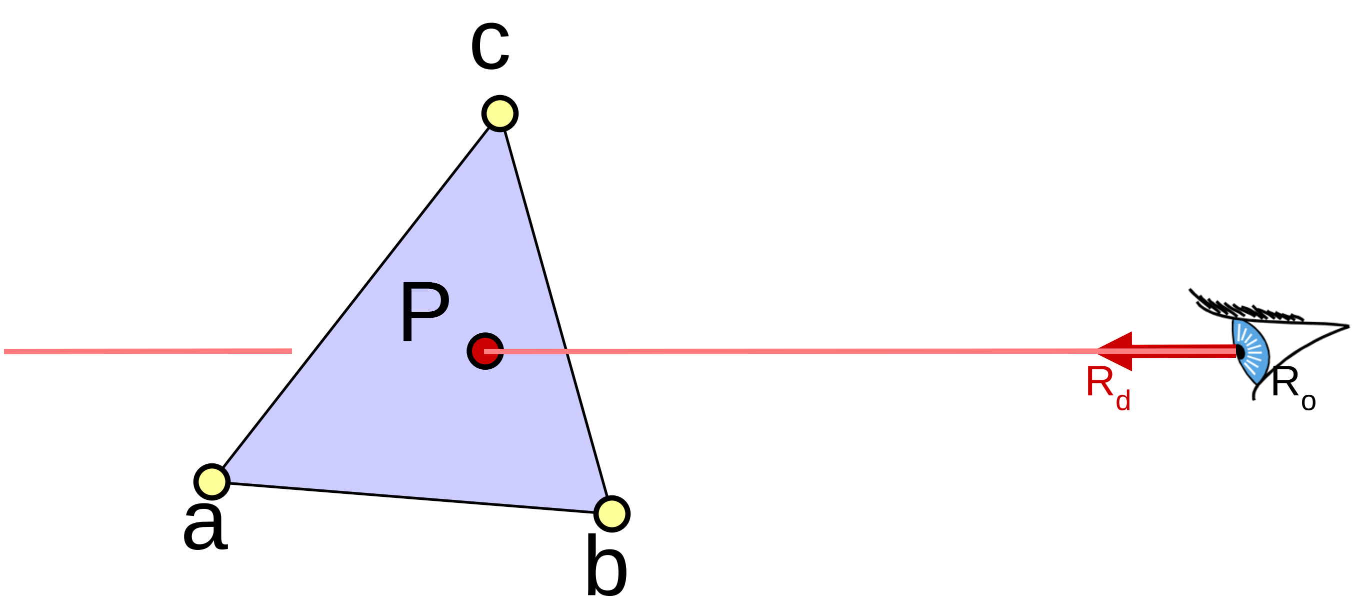


Questions?



Ray-Triangle Intersection

- Use ray-plane intersection followed by in-triangle test
- Or try to be smarter
  - Use barycentric coordinates

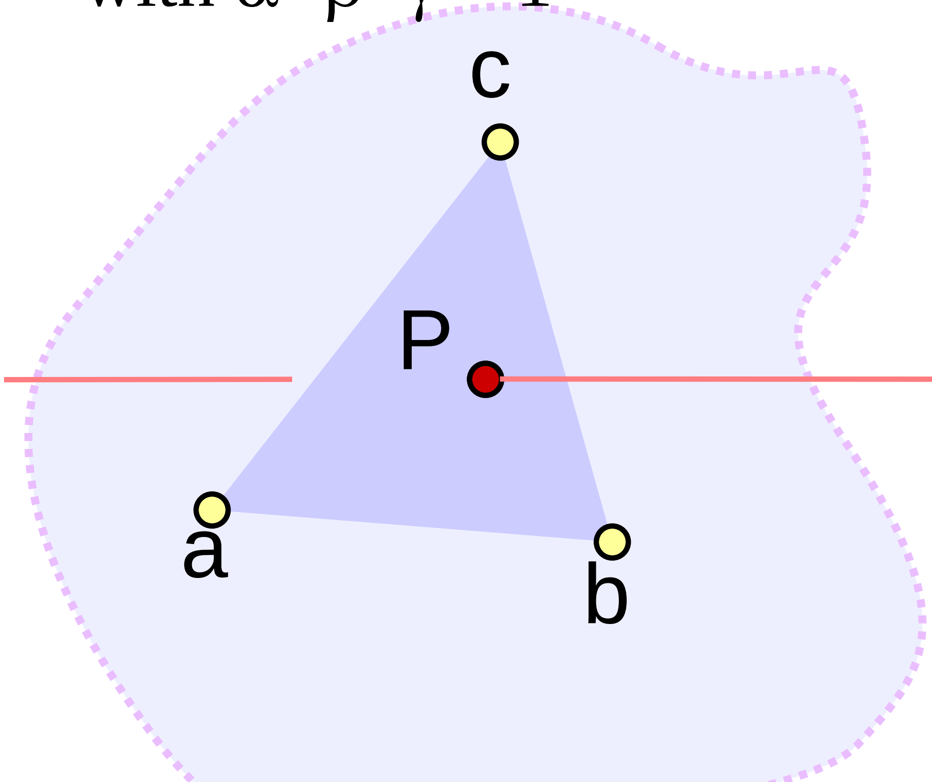


Barycentric Definition of a Plane

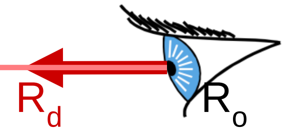
- A (non-degenerate) triangle (**a**,**b**,**c**) defines a plane
- Any point **P** on this plane can be written as

  $\mathbf{P}(\alpha,\beta,\gamma) = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$,

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



Why? How?



[Möbius, 1827]

Barycentric Coordinates

- Since $\alpha+\beta+\gamma = 1$, we can write $\alpha = 1-\beta-\gamma$

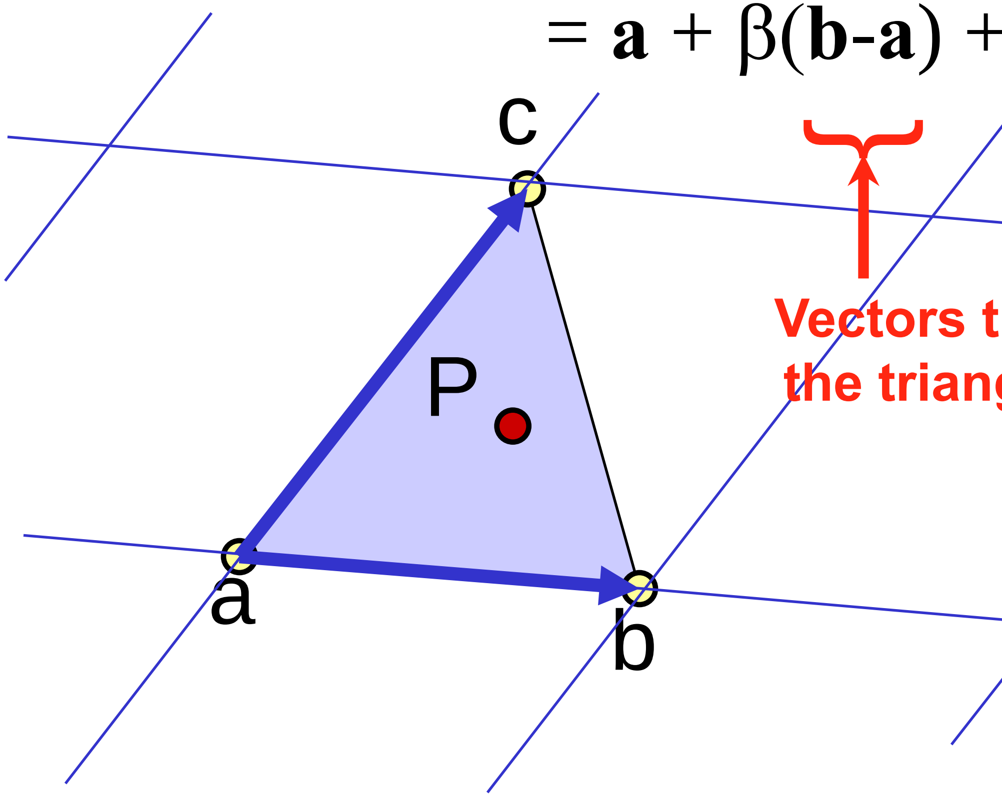
$$\mathbf{P}(\alpha,\beta,\gamma) = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$$

rewrite

$$\mathbf{P}(\beta,\gamma) = (1-\beta-\gamma)\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$$

$$= \mathbf{a} + \beta(\mathbf{b}-\mathbf{a}) + \gamma(\mathbf{c}-\mathbf{a})$$

Vectors that lie on the triangle plane

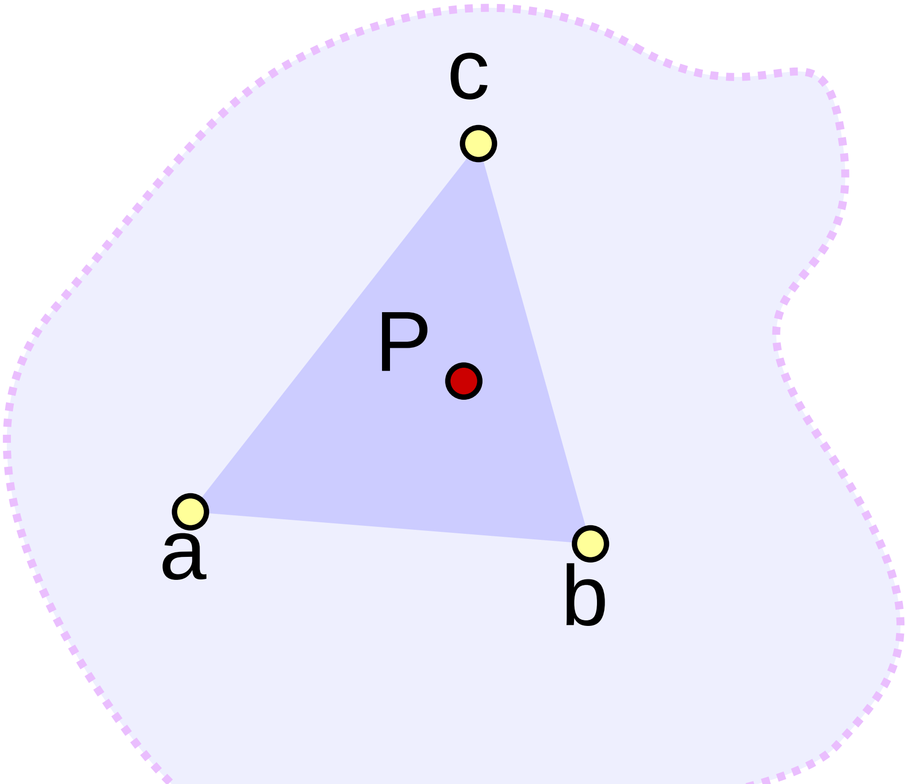


Non-orthogonal coordinate system on the plane!

Barycentric Definition of a Plane

[Möbius, 1827]

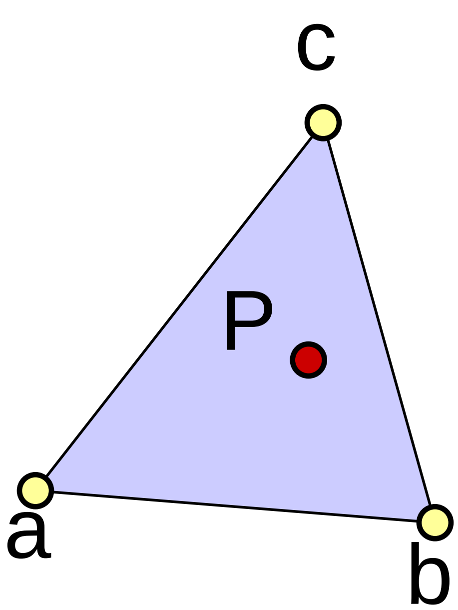
- $\mathbf{P}(\alpha,\beta,\gamma) = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$
  with $\alpha+\beta+\gamma = 1$
- Is it explicit or implicit?



Fun to know:
P is the **barycenter**, the single point upon which the triangle would balance if weights of size α, β, & γ are placed on points **a**, **b** & **c**.

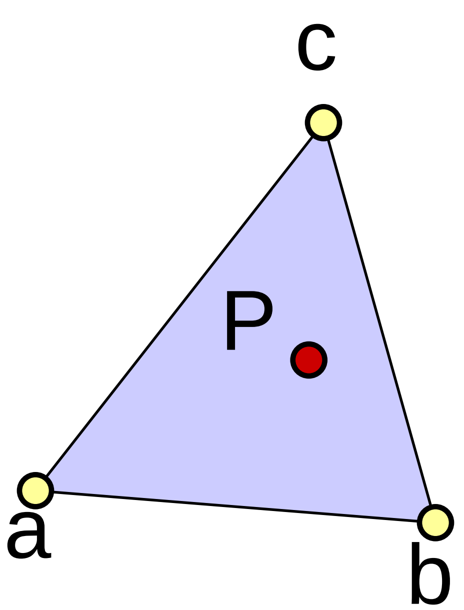
Barycentric Definition of a Triangle

- $\mathbf{P}(\alpha,\beta,\gamma) = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$
  with $\alpha+\beta+\gamma = 1$ parameterizes the entire plane



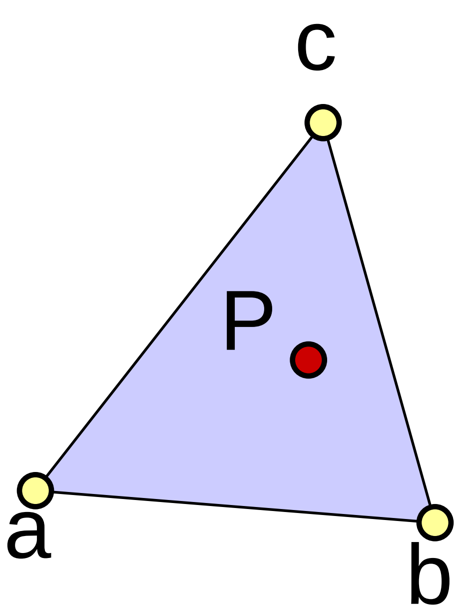
Barycentric Definition of a Triangle

- $\mathbf{P}(\alpha,\beta,\gamma) = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$
  with $\alpha+\beta+\gamma = 1$ parameterizes the entire plane
- If we require in addition that
  $\alpha, \beta, \gamma >= 0$, we get just the triangle!
  - Note that with $\alpha+\beta+\gamma = 1$ this implies
    $0 \leq \alpha \leq 1 \quad \& \quad 0 \leq \beta \leq 1 \quad \& \quad 0 \leq \gamma \leq 1$
  - Verify:
    - $\alpha = 0 \Rightarrow$ **P** lies on line **b-c**
    - $\alpha, \beta = 0 \Rightarrow \mathbf{P} = \mathbf{c}$
    - etc.



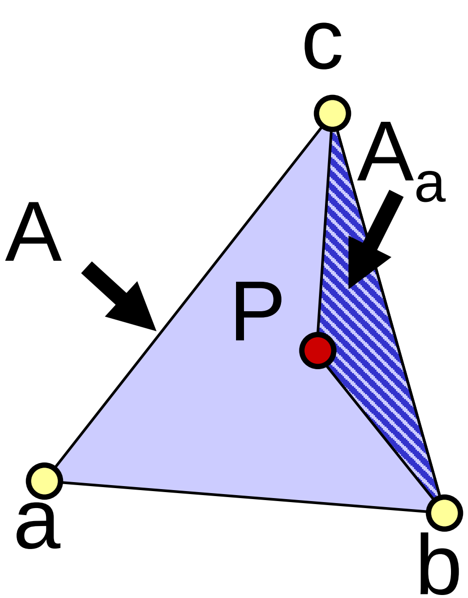
Barycentric Definition of a Triangle

- $\mathbf{P}(\alpha,\beta,\gamma) = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$
- Condition to be barycentric coordinates:
  $\alpha+\beta+\gamma = 1$
- Condition to be inside the triangle:
  $\alpha, \beta, \gamma \geq 0$



How Do We Compute α, β, γ ?

- Ratio of opposite sub-triangle area to total area
  - $\alpha = A_a/A \qquad \beta = A_b/A \qquad \gamma = A_c/A$
- Use signed areas for points outside the triangle



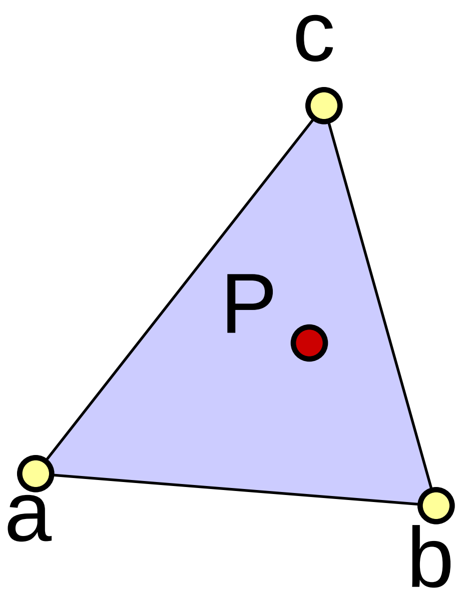
How Do We Compute α, β, γ ?

- Or write it as a 2×2 linear system
- $\mathbf{P}(\beta, \gamma) = \mathbf{a} + \beta\mathbf{e}_1 + \gamma\mathbf{e}_2$

  $\mathbf{e}_1 = (\mathbf{b}\text{-}\mathbf{a})$, $\mathbf{e}_2 = (\mathbf{c}\text{-}\mathbf{a})$

$$\boldsymbol{a} + \beta\boldsymbol{e}_1 + \gamma\boldsymbol{e}_2 - \boldsymbol{P} = 0$$

This should be zero



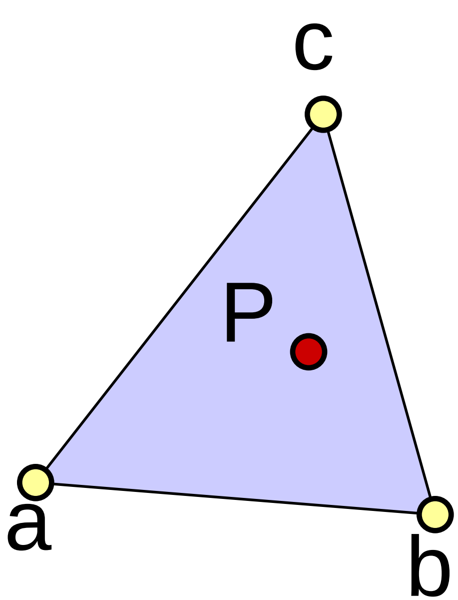
How Do We Compute α, β, γ ?

- Or write it as a 2×2 linear system
- $\mathbf{P}(\beta, \gamma) = \mathbf{a} + \beta\mathbf{e}_1 + \gamma\mathbf{e}_2$

  $\mathbf{e}_1 = (\mathbf{b}-\mathbf{a})$, $\mathbf{e}_2 = (\mathbf{c}-\mathbf{a})$

$$\boldsymbol{a} + \beta\boldsymbol{e}_1 + \gamma\boldsymbol{e}_2 - \boldsymbol{P} = 0$$

This should be zero



Something's wrong... This is a linear system of 3 equations and 2 unknowns!

How Do We Compute α, β, γ ?

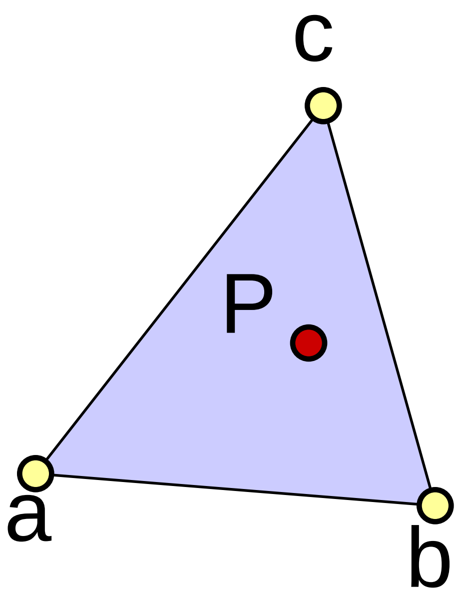
- Or write it as a 2×2 linear system
- $\mathbf{P}(\beta, \gamma) = \mathbf{a} + \beta\mathbf{e}_1 + \gamma\mathbf{e}_2$

$\mathbf{e}_1 = (\mathbf{b}-\mathbf{a})$, $\mathbf{e}_2 = (\mathbf{c}-\mathbf{a})$

$$\langle \boldsymbol{e}_1,\ \boldsymbol{a} + \beta\boldsymbol{e}_1 + \gamma\boldsymbol{e}_2 - \boldsymbol{P} \rangle = 0$$

$$\langle \boldsymbol{e}_2,\ \boldsymbol{a} + \beta\boldsymbol{e}_1 + \gamma\boldsymbol{e}_2 - \boldsymbol{P} \rangle = 0$$

These should be zero



Ha! We'll take inner products of this equation with $\mathbf{e}_1$ & $\mathbf{e}_2$

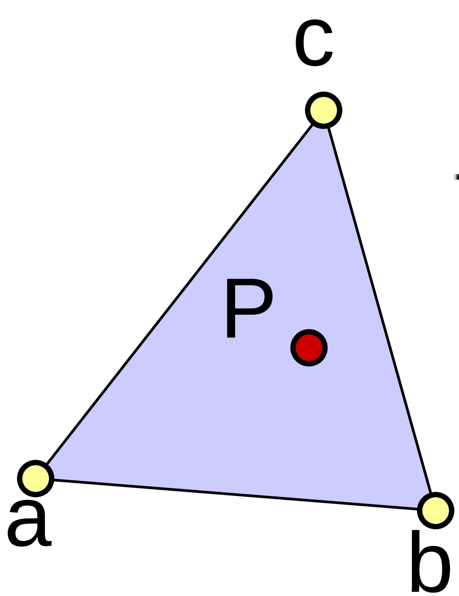
How Do We Compute α, β, γ ?

- Or write it as a 2×2 linear system
- $\mathbf{P}(\beta, \gamma) = \mathbf{a} + \beta\mathbf{e}_1 + \gamma\mathbf{e}_2$

  $\mathbf{e}_1 = (\mathbf{b}\text{-}\mathbf{a})$, $\mathbf{e}_2 = (\mathbf{c}\text{-}\mathbf{a})$

$$\langle \boldsymbol{e}_1,\ \boldsymbol{a} + \beta\boldsymbol{e}_1 + \gamma\boldsymbol{e}_2 - \boldsymbol{P}\rangle = 0$$

$$\langle \boldsymbol{e}_2,\ \boldsymbol{a} + \beta\boldsymbol{e}_1 + \gamma\boldsymbol{e}_2 - \boldsymbol{P}\rangle = 0$$



$$\Leftrightarrow \boxed{\begin{pmatrix} \langle \boldsymbol{e}_1, \boldsymbol{e}_1\rangle & \langle \boldsymbol{e}_1, \boldsymbol{e}_2\rangle \\ \langle \boldsymbol{e}_2, \boldsymbol{e}_1\rangle & \langle \boldsymbol{e}_2, \boldsymbol{e}_2\rangle \end{pmatrix} \begin{pmatrix} \beta \\ \gamma \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}}$$

where $\begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} \langle (\boldsymbol{P} - \boldsymbol{a}), \boldsymbol{e}_1\rangle \\ \langle (\boldsymbol{P} - \boldsymbol{a}), \boldsymbol{e}_2\rangle \end{pmatrix}$

and <a,b> is the dot product.

How Do We Compute α, β, γ ?

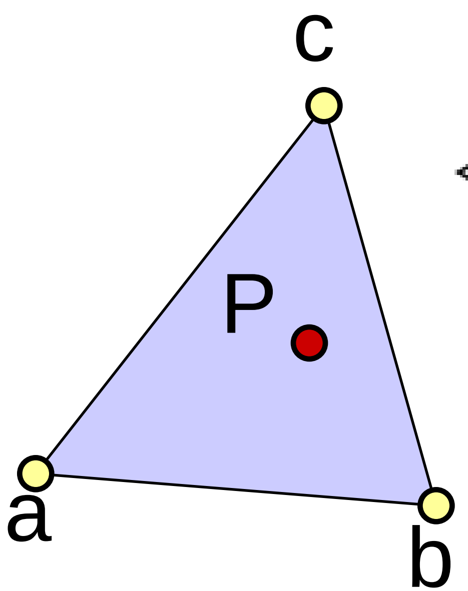
- Or write it as a 2×2 linear system
- $\mathbf{P}(\beta, \gamma) = \mathbf{a} + \beta \mathbf{e}_1 + \gamma \mathbf{e}_2$

  $\mathbf{e}_1 = (\mathbf{b}-\mathbf{a})$, $\mathbf{e}_2 = (\mathbf{c}-\mathbf{a})$

Questions?

$$\langle \boldsymbol{e}_1,\ \boldsymbol{a} + \beta\boldsymbol{e}_1 + \gamma\boldsymbol{e}_2 - \boldsymbol{P} \rangle = 0$$

$$\langle \boldsymbol{e}_2,\ \boldsymbol{a} + \beta\boldsymbol{e}_1 + \gamma\boldsymbol{e}_2 - \boldsymbol{P} \rangle = 0$$



$$\Leftrightarrow \boxed{\begin{pmatrix} \langle \boldsymbol{e}_1, \boldsymbol{e}_1 \rangle & \langle \boldsymbol{e}_1, \boldsymbol{e}_2 \rangle \\ \langle \boldsymbol{e}_2, \boldsymbol{e}_1 \rangle & \langle \boldsymbol{e}_2, \boldsymbol{e}_2 \rangle \end{pmatrix} \begin{pmatrix} \beta \\ \gamma \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}}$$

where $$\begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} \langle (\boldsymbol{P} - \boldsymbol{a}), \boldsymbol{e}_1 \rangle \\ \langle (\boldsymbol{P} - \boldsymbol{a}), \boldsymbol{e}_2 \rangle \end{pmatrix}$$

and <a,b> is the dot product.

Intersection with Barycentric Triangle

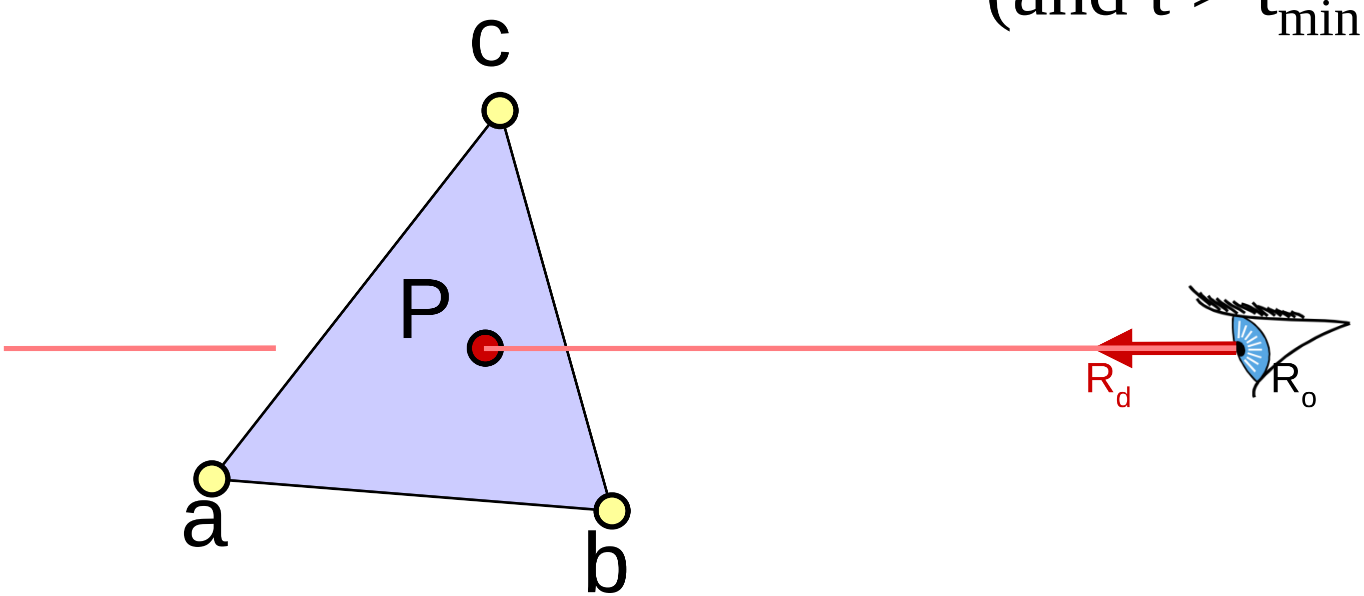
- Again, set ray equation equal to barycentric equation

$$\mathbf{P}(t) = \mathbf{P}(\beta, \gamma)$$

$$\mathbf{R}_o + t * \mathbf{R}_d = \mathbf{a} + \beta(\mathbf{b}-\mathbf{a}) + \gamma(\mathbf{c}-\mathbf{a})$$

- Intersection if $\beta + \gamma \leq 1$ & $\beta \geq 0$ & $\gamma \geq 0$

  (and $t > t_{min}$ …)



Intersection with Barycentric Triangle

- $\mathbf{R}_o + t * \mathbf{R}_d = \mathbf{a} + \beta(\mathbf{b}-\mathbf{a}) + \gamma(\mathbf{c}-\mathbf{a})$

$$\left.\begin{aligned}
R_{ox} + tR_{dx} &= a_x + \beta(b_x-a_x) + \gamma(c_x-a_x) \\
R_{oy} + tR_{dy} &= a_y + \beta(b_y-a_y) + \gamma(c_y-a_y) \\
R_{oz} + tR_{dz} &= a_z + \beta(b_z-a_z) + \gamma(c_z-a_z)
\end{aligned}\right\}\ \text{3 equations, 3 unknowns}$$

- Regroup & write in matrix form **Ax=b**

$$\begin{bmatrix} a_x - b_x & a_x - c_x & R_{dx} \\ a_y - b_y & a_y - c_y & R_{dy} \\ a_z - b_z & a_z - c_z & R_{dz} \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} a_x - R_{ox} \\ a_y - R_{oy} \\ a_z - R_{oz} \end{bmatrix}$$

Cramer's Rule

- Used to solve for one variable at a time in system of equations

$$\beta = \frac{\begin{vmatrix} a_x - R_{ox} & a_x - c_x & R_{dx} \\ a_y - R_{oy} & a_y - c_y & R_{dy} \\ a_z - R_{oz} & a_z - c_z & R_{dz} \end{vmatrix}}{|A|} \qquad \gamma = \frac{\begin{vmatrix} a_x - b_x & a_x - R_{ox} & R_{dx} \\ a_y - b_y & a_y - R_{oy} & R_{dy} \\ a_z - b_z & a_z - R_{oz} & R_{dz} \end{vmatrix}}{|A|}$$

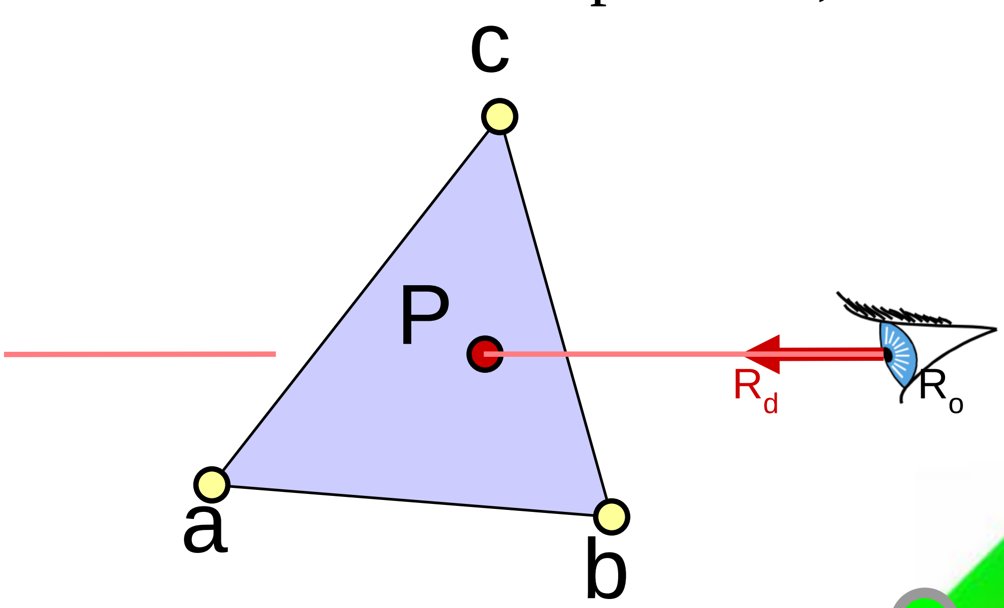
$$t = \frac{\begin{vmatrix} a_x - b_x & a_x - c_x & a_x - R_{ox} \\ a_y - b_y & a_y - c_y & a_y - R_{oy} \\ a_z - b_z & a_z - c_z & a_z - R_{oz} \end{vmatrix}}{|A|}$$

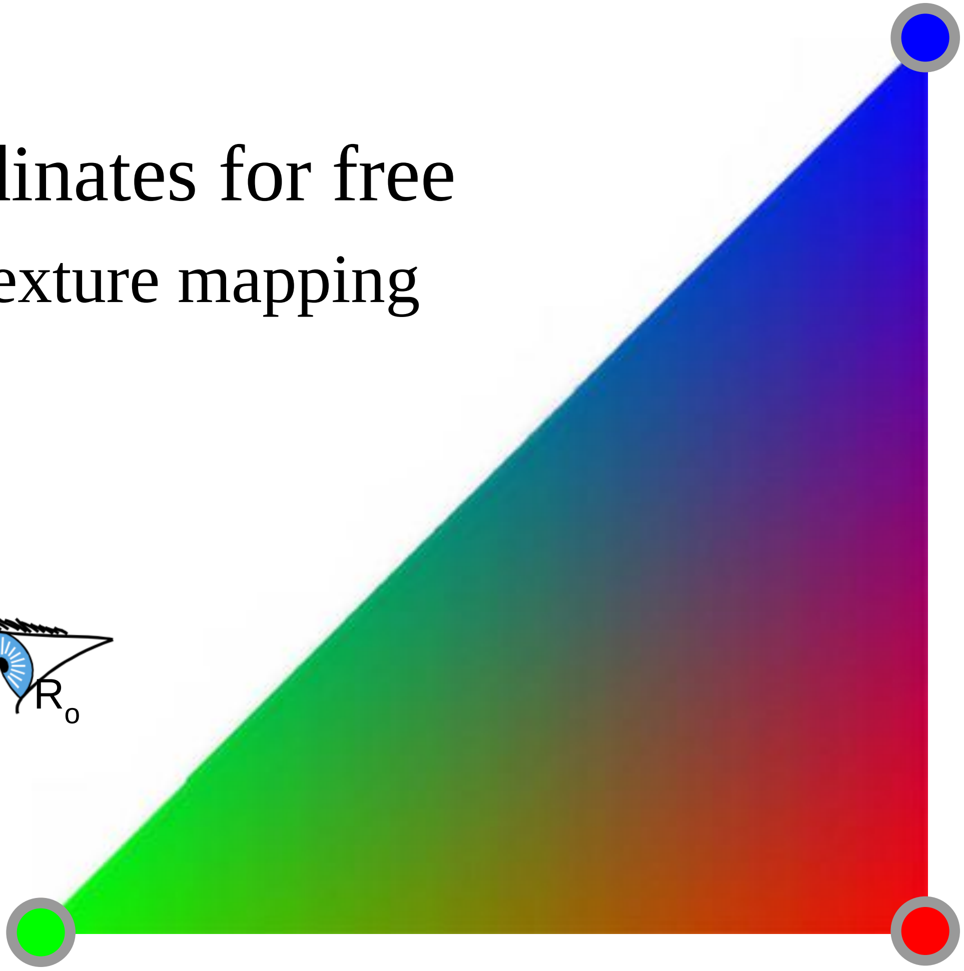
| | denotes the determinant

Can be copied mechanically into code

Barycentric Intersection Pros

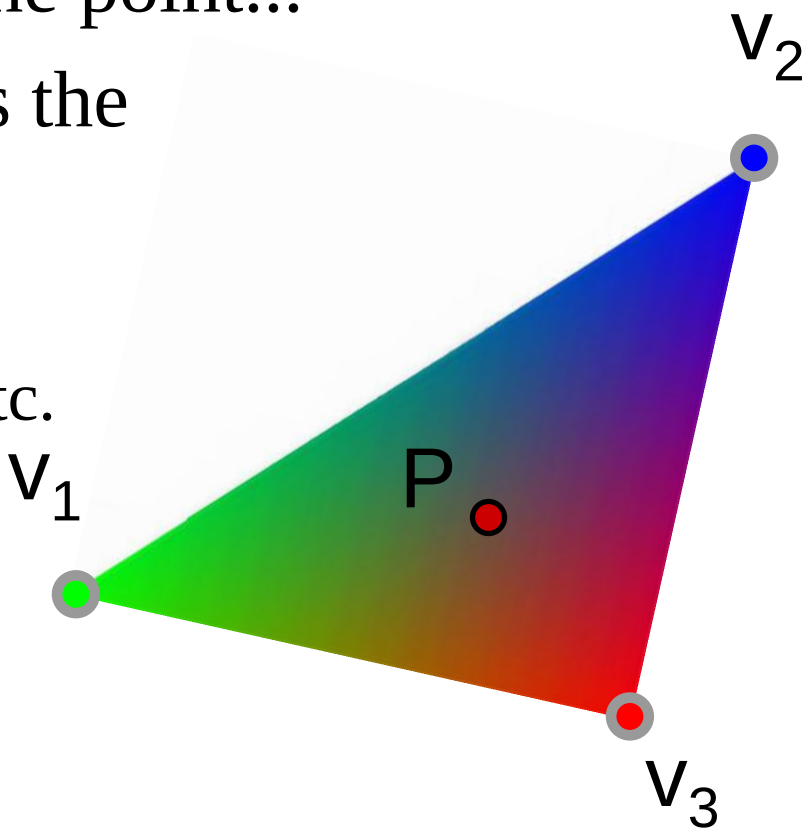
- Efficient
- Stores no plane equation
- Get the barycentric coordinates for free
  - Useful for interpolation, texture mapping





Barycentric Interpolation

- Values v_1, v_2, v_3 defined at **a**, **b**, **c**
  - Colors, normal, texture coordinates, etc.
- $\mathbf{P}(\alpha,\beta,\gamma) = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$ is the point...
- $v(\alpha,\beta,\gamma) = \alpha v_1 + \beta v_2 + \gamma v_3$ is the barycentric interpolation of v_1,v_2,v_3 at point **P**
  - Sanity check: $v(1,0,0) = v_1$, etc.
- I.e, once you know α,β,γ you can interpolate values using the same weights.
  - Convenient!

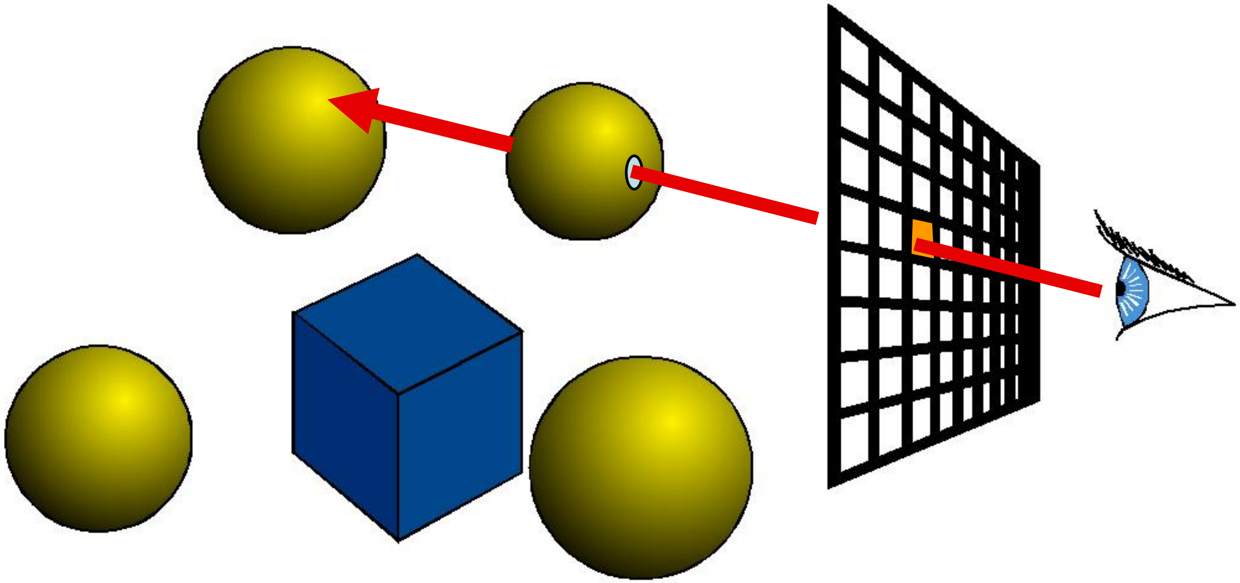


Questions?

- Image computed using the RADIANCE system by Greg Ward

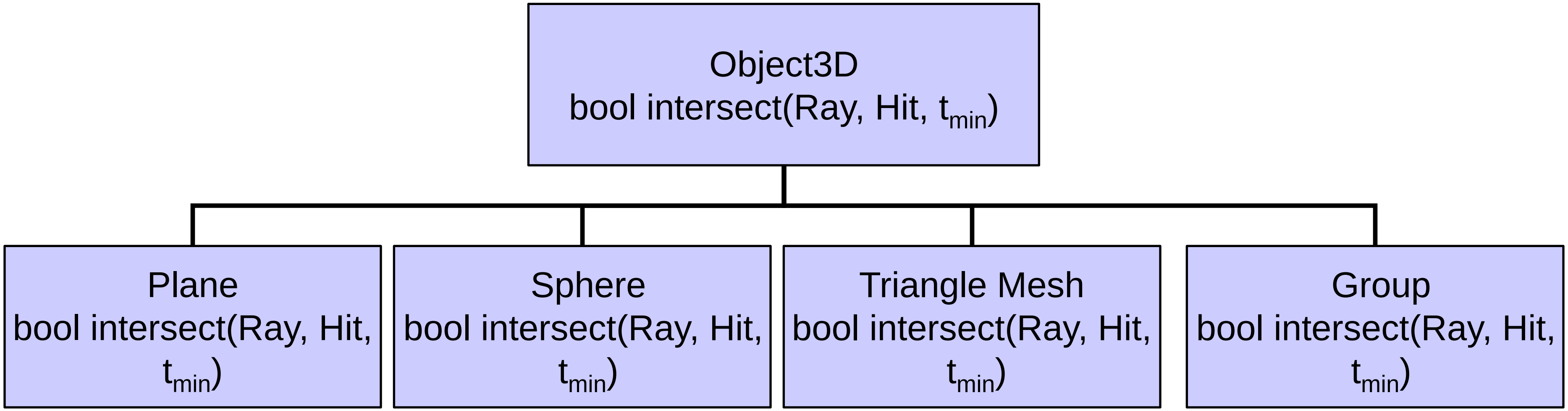
Ray Casting: Object Oriented Design

```
For every pixel
  Construct a ray from the eye
  For every object in the scene
     Find intersection with the ray
     Keep if closest
```



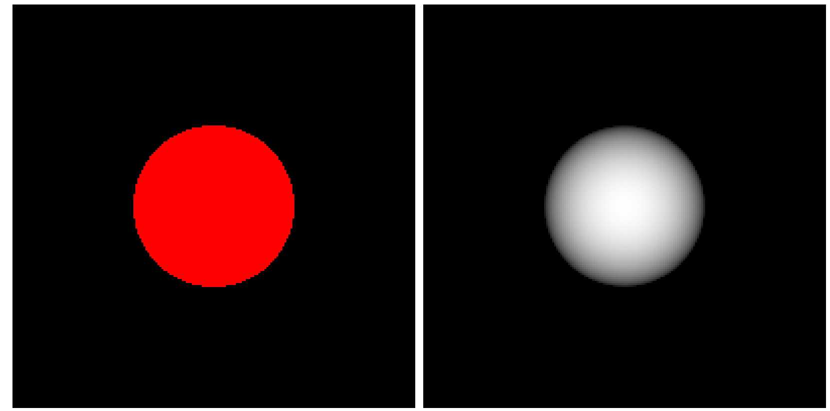
Object-Oriented Design

- We want to be able to add primitives easily
  - Inheritance and virtual methods
- Even the scene is derived from Object3D!

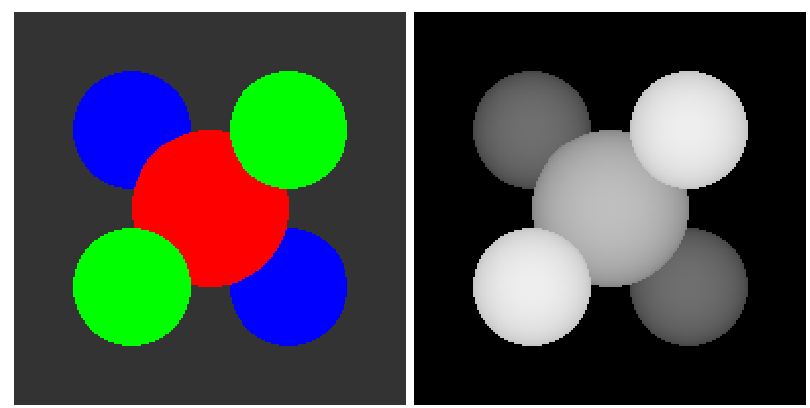


- Also cameras are abstracted (perspective/ortho)
  - Methods for generating rays for given image coordinates

Assignment 4 & 5: Ray Casting/Tracing



- Write a basic ray caster
  - Orthographic and perspective cameras
  - Spheres and triangles
  - 2 Display modes: color and distance
- We provide classes for
  - Ray: origin, direction
  - Hit: t, Material, (*normal*)
  - Scene Parsing
- You write ray generation, hit testing, simple shading

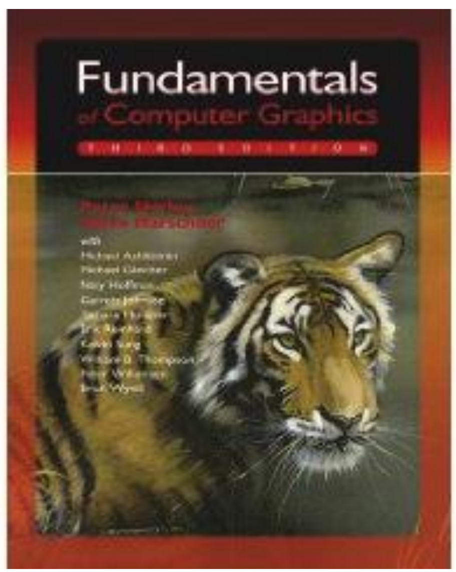


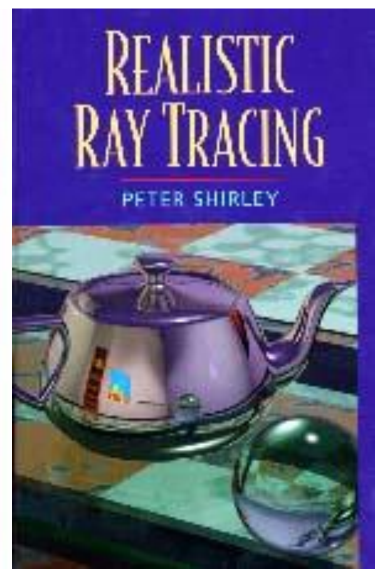
Books

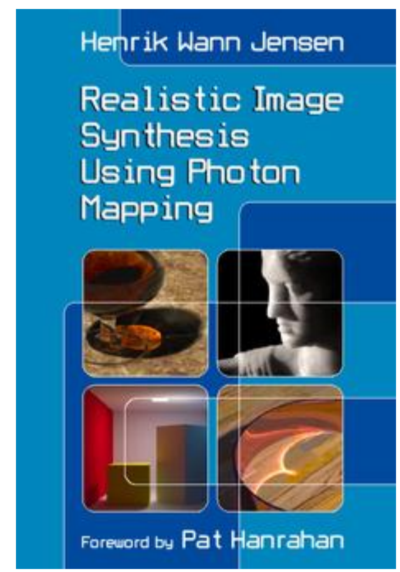
- Peter Shirley et al.: *Fundamentals of Computer Graphics* AK Peters

Remember the ones at books24x7 mentioned in the beginning!

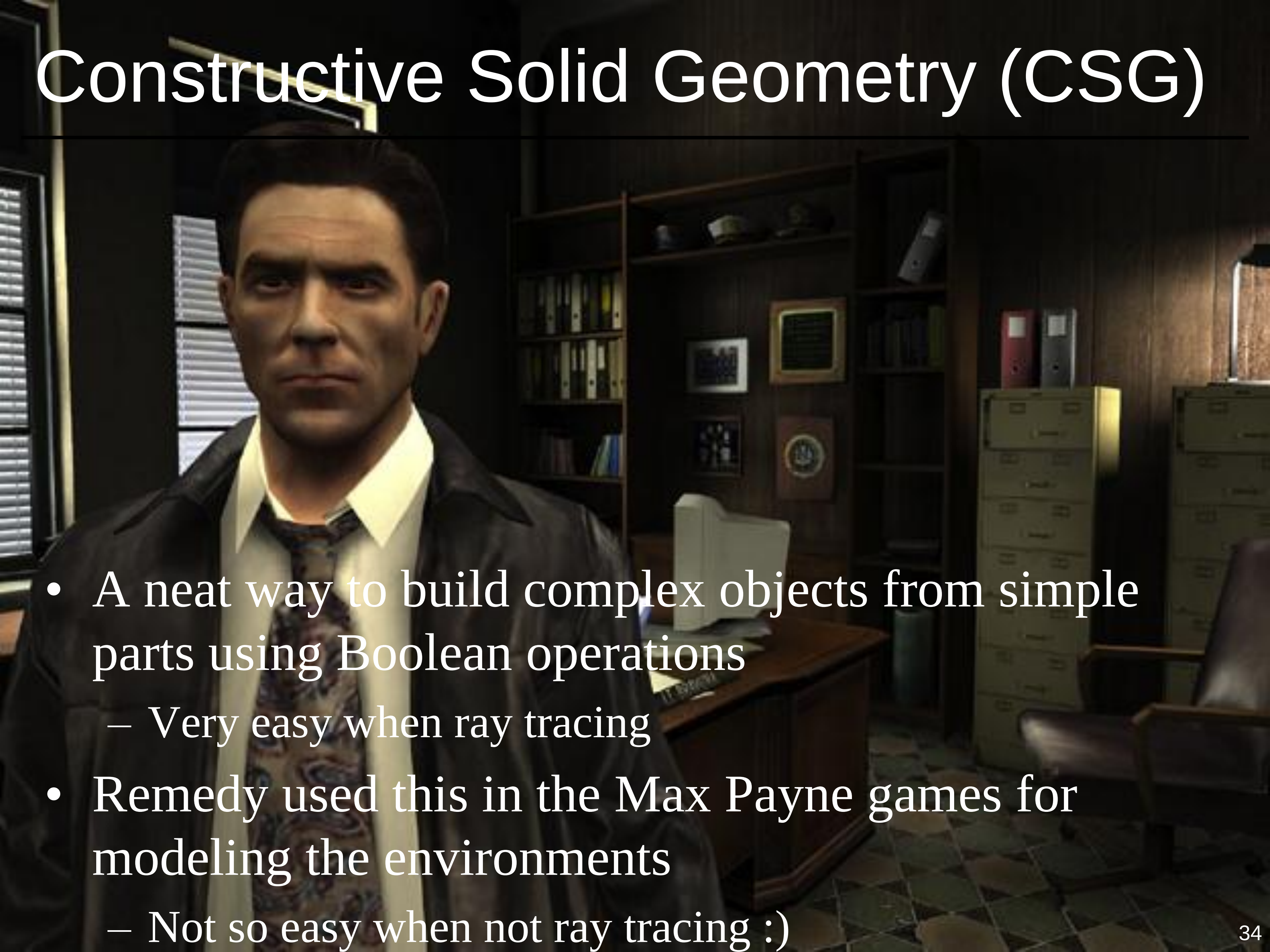
- Ray Tracing
  - Jensen
  - Shirley
  - Glassner





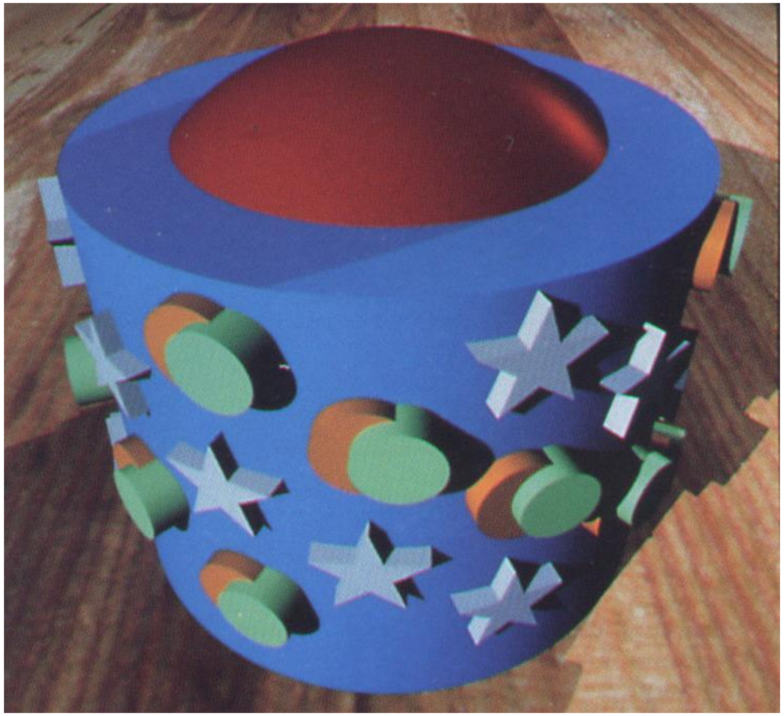


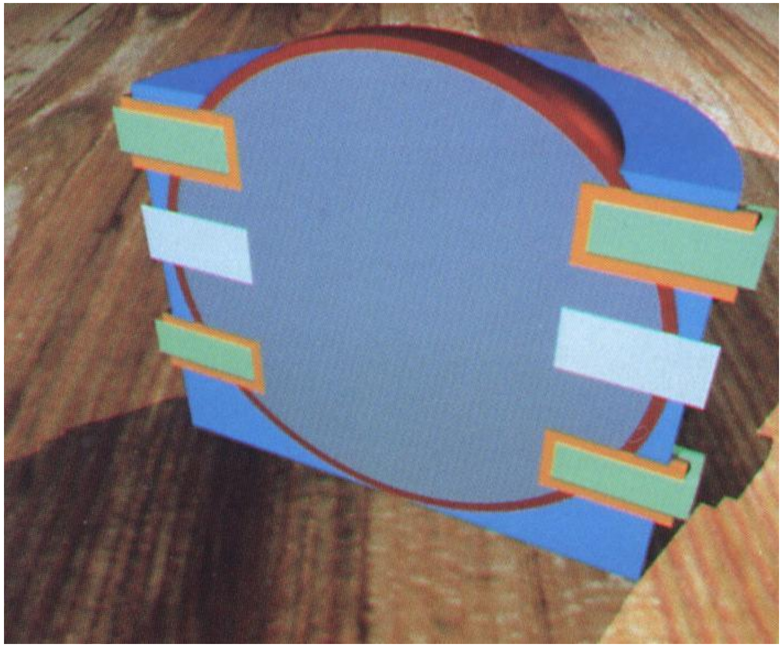
Constructive Solid Geometry (CSG)



- A neat way to build complex objects from simple parts using Boolean operations
  - Very easy when ray tracing
- Remedy used this in the Max Payne games for modeling the environments
  - Not so easy when not ray tracing :)

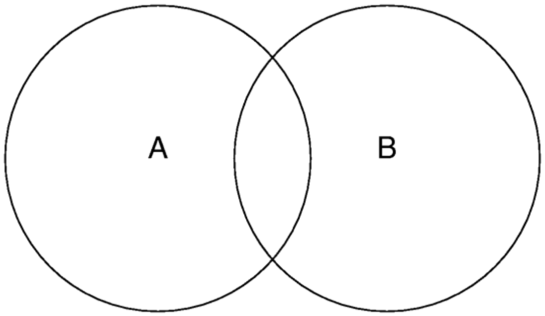
CSG Examples



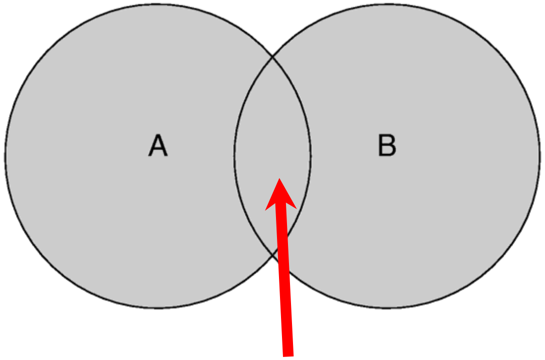


Constructive Solid Geometry (CSG)

Given overlapping shapes A and B:

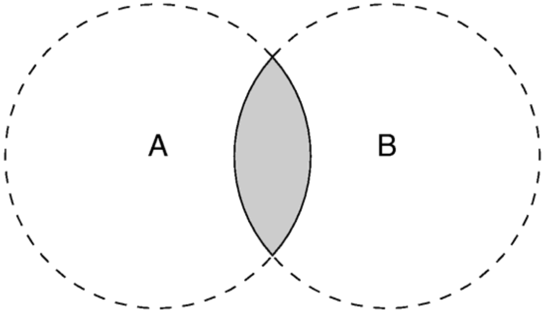


Union

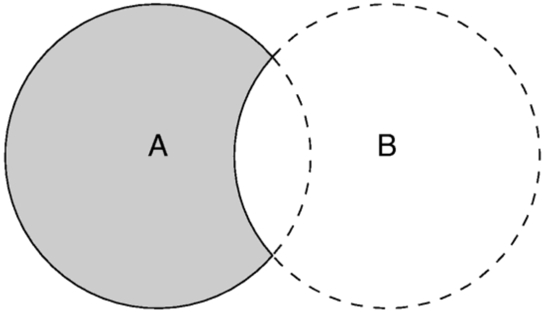


Should only "count" overlap region once!

Intersection

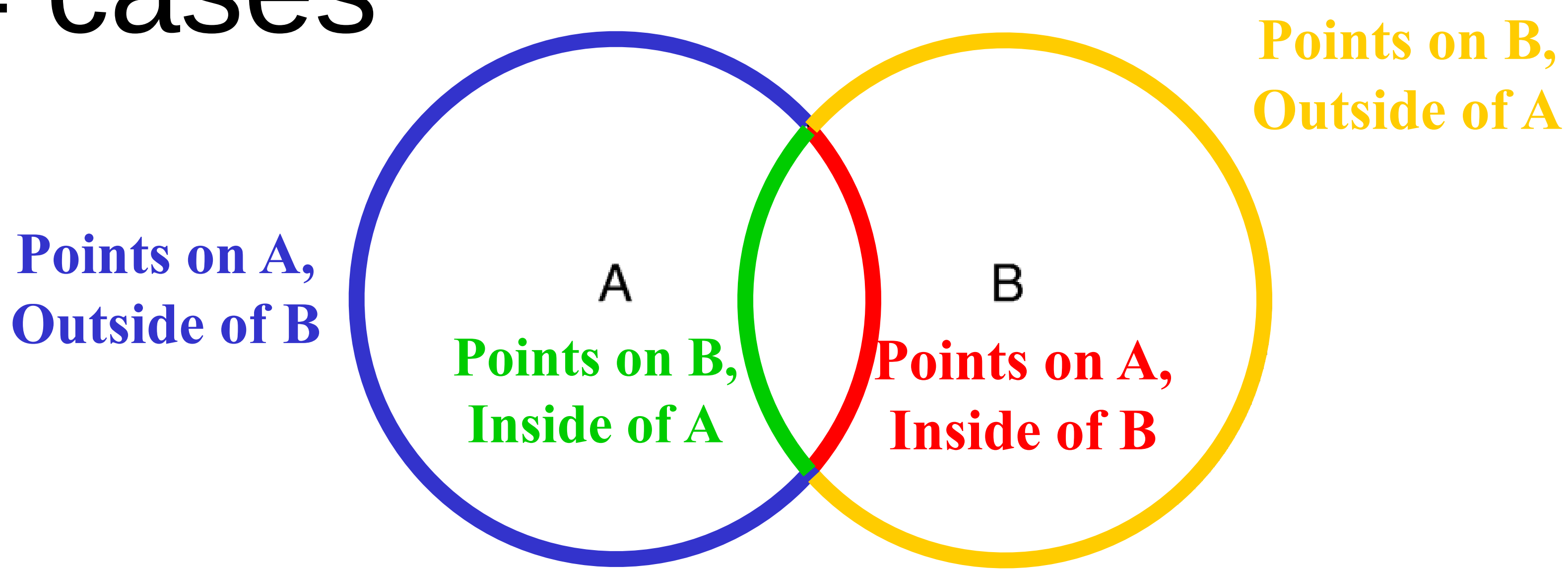


Subtraction

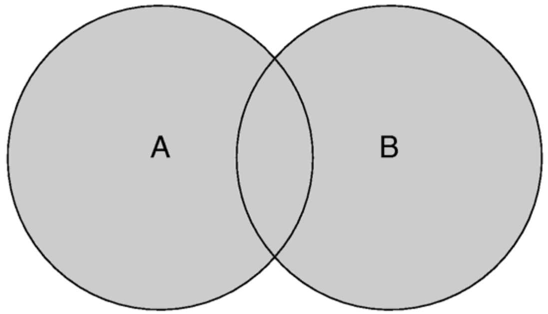


How Can We Implement CSG?

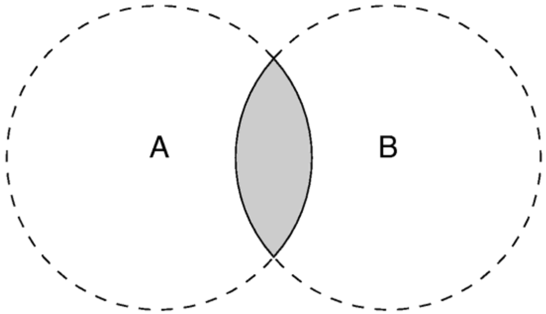
4 cases



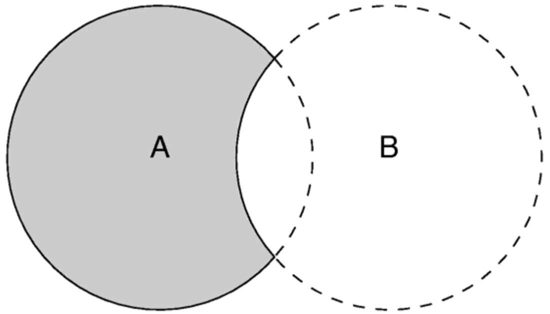
Union



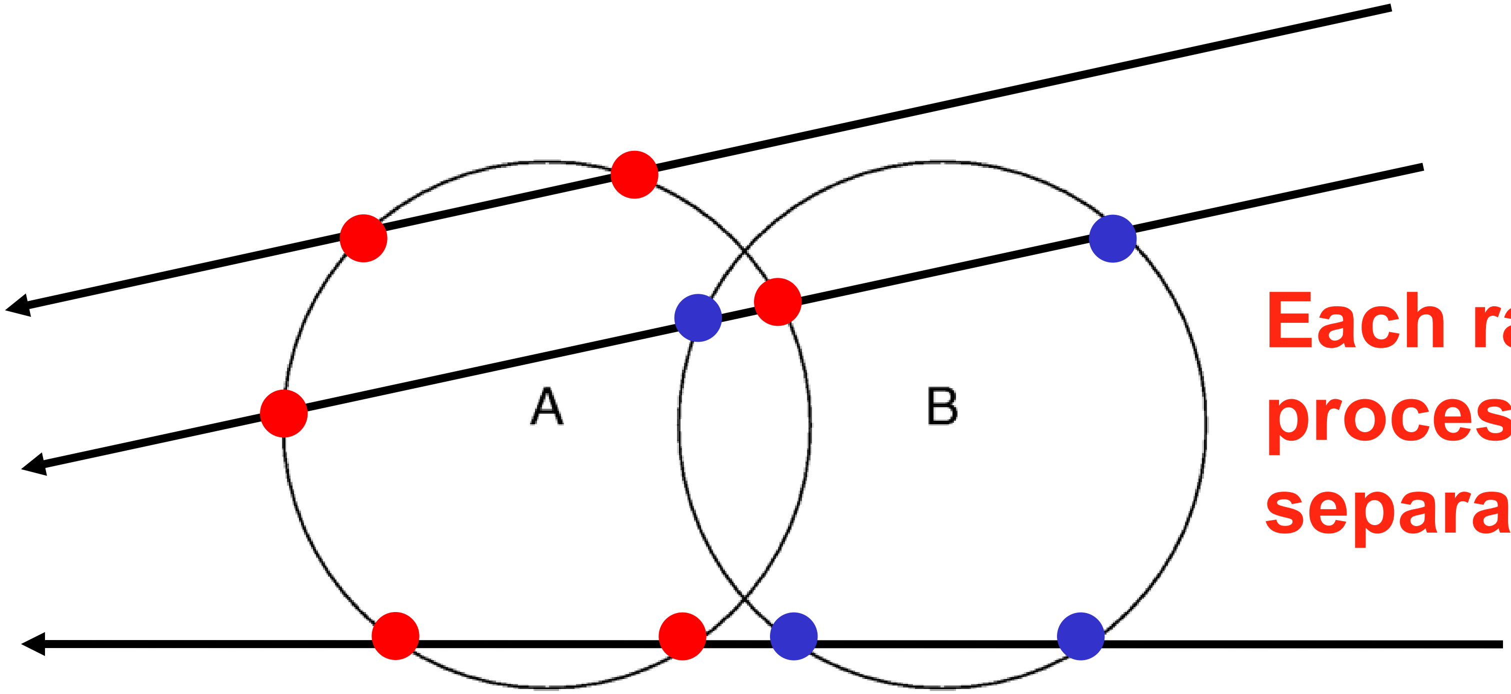
Intersection



Subtraction

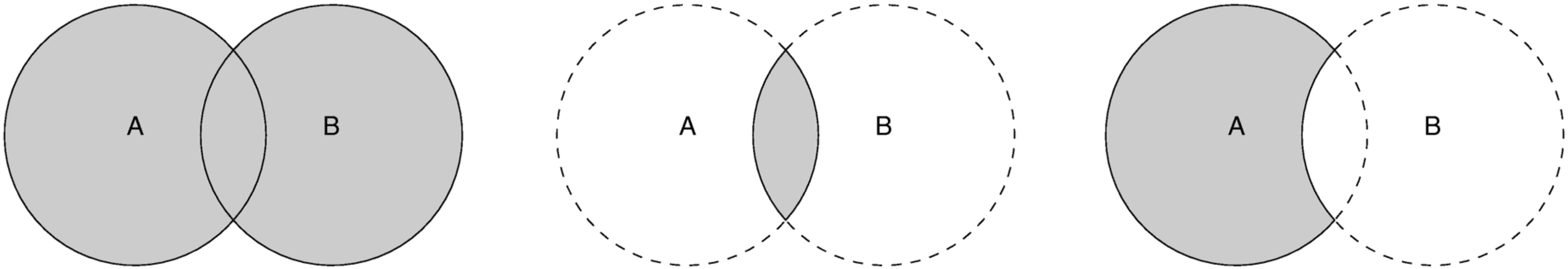


Collect Intersections



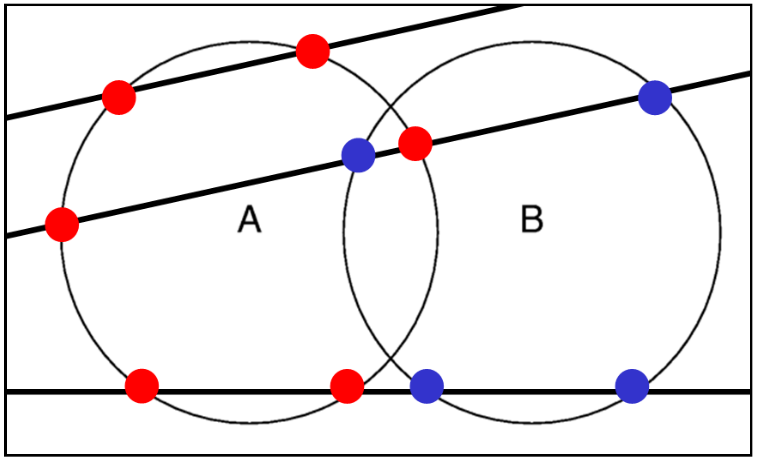
Each ray processed separately!

Union Intersection Subtraction



Implementing CSG

1. Test "inside" intersections:
   - Find intersections with A,
     test if they are inside/outside B
   - Find intersections with B,
     test if they are inside/outside A

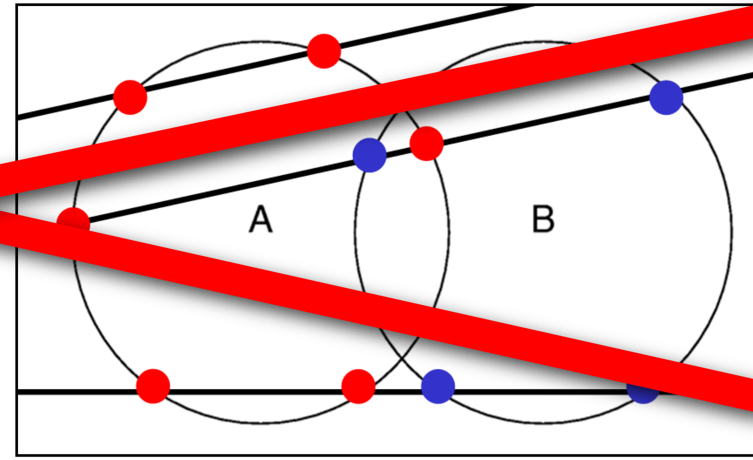


This would certainly work, but would need to determine if points are inside solids...

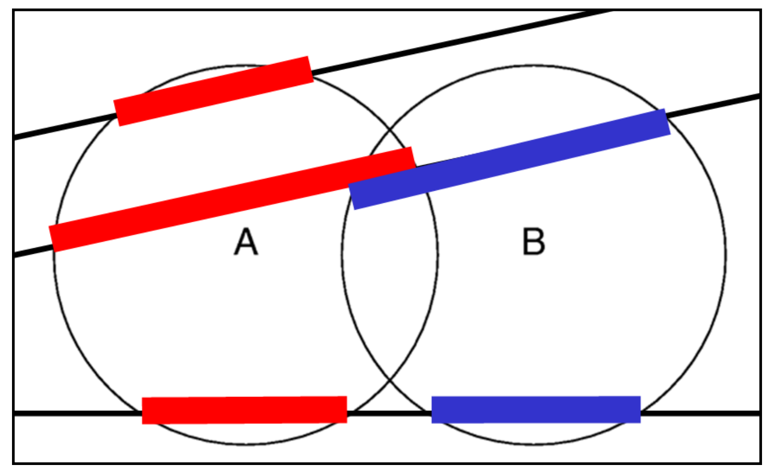
:-(

Implementing CSG

1. ~~Test "inside" intersections:~~
   - ~~Find intersections with A, test if they are inside/outside B~~
   - ~~Find intersections with B, test if they are inside/outside A~~



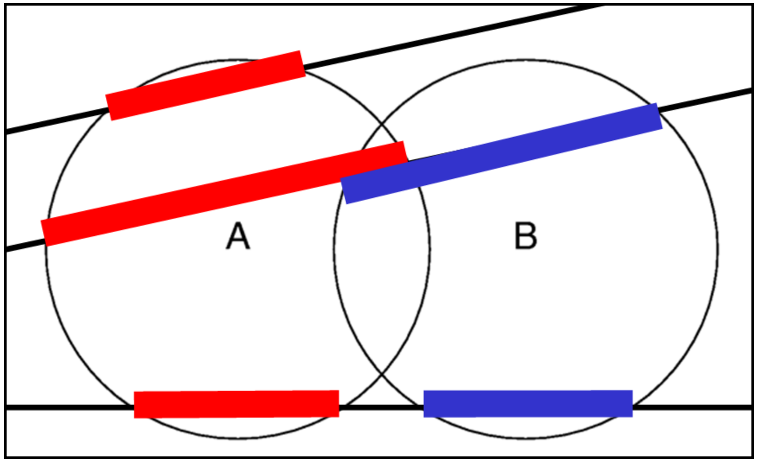
2. Overlapping intervals:
   - Find the intervals of "inside" along the ray for A and B
   - How? Just keep an “entry” / “exit” bit for each intersection
     - Easy to determine from intersection normal and ray direction
   - Compute union/intersection/subtraction of the intervals



Implementing CSG

Problem reduces to 1D for each ray

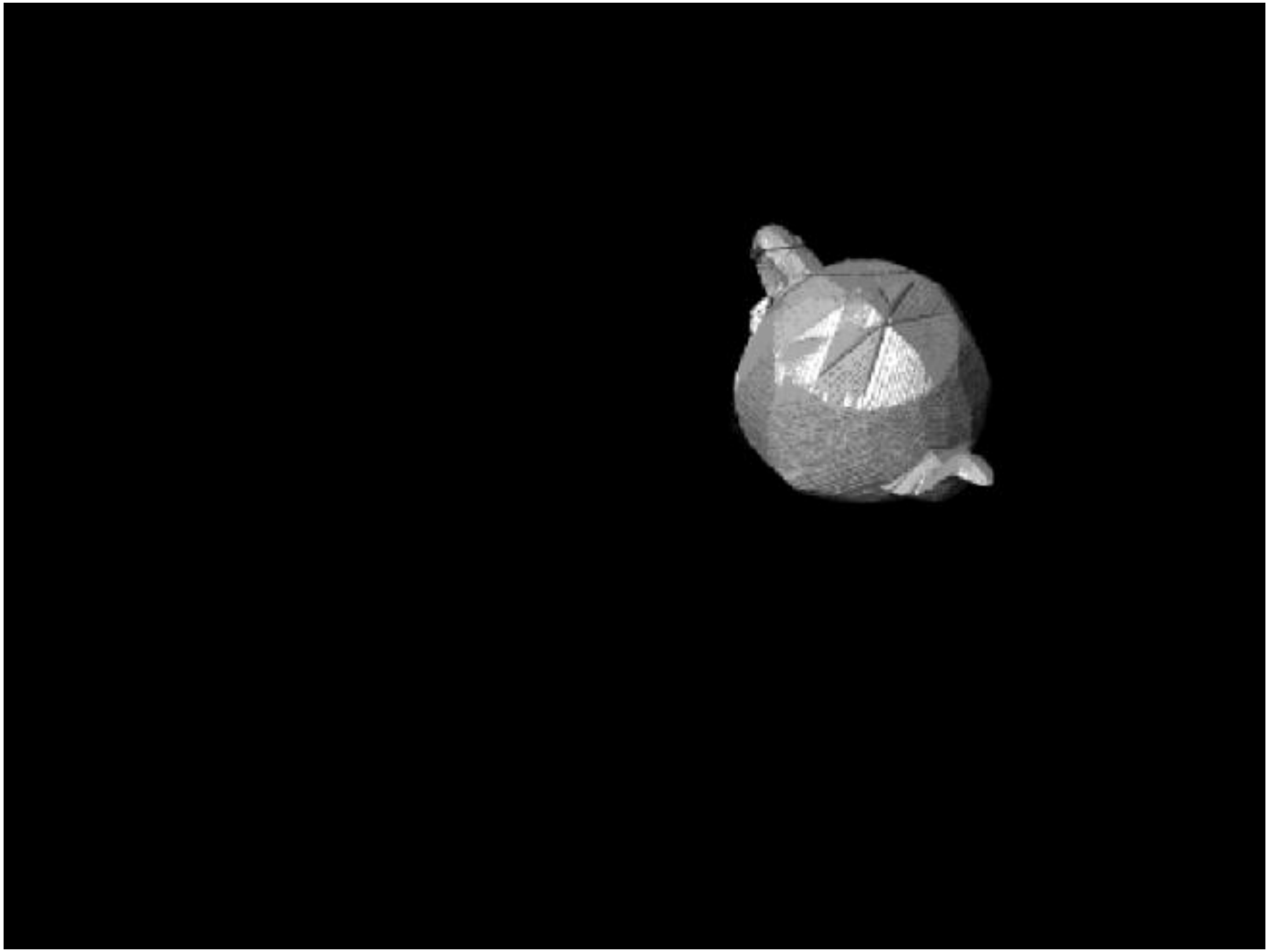
2. Overlapping intervals:
   - Find the intervals of "inside" along the ray for A and B
   - How? Just keep an “entry” / “exit” bit for each intersection
     - Easy to determine from intersection normal and ray direction
   - Compute union/intersection/subtraction of the intervals



CSG is Easy with Ray Casting...

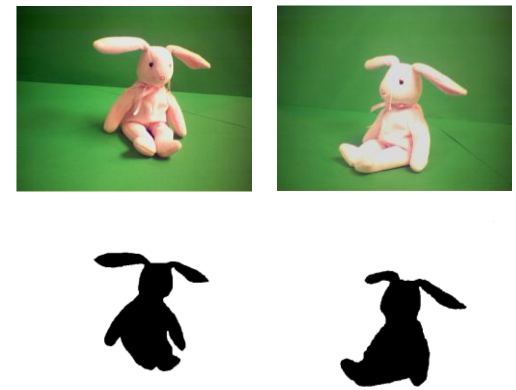
- ...but **very hard** if you actually try to compute an explicit representation of the resulting surface as a triangle mesh
- In principle very simple,
  but floating point numbers are not exact
  - E.g., points do not lie exactly on planes...
  - Computing the intersection A vs B is not necessarily the same as B vs A...
  - The line that results from intersecting two planes does not necessarily lie on either plane...
  - etc., etc.

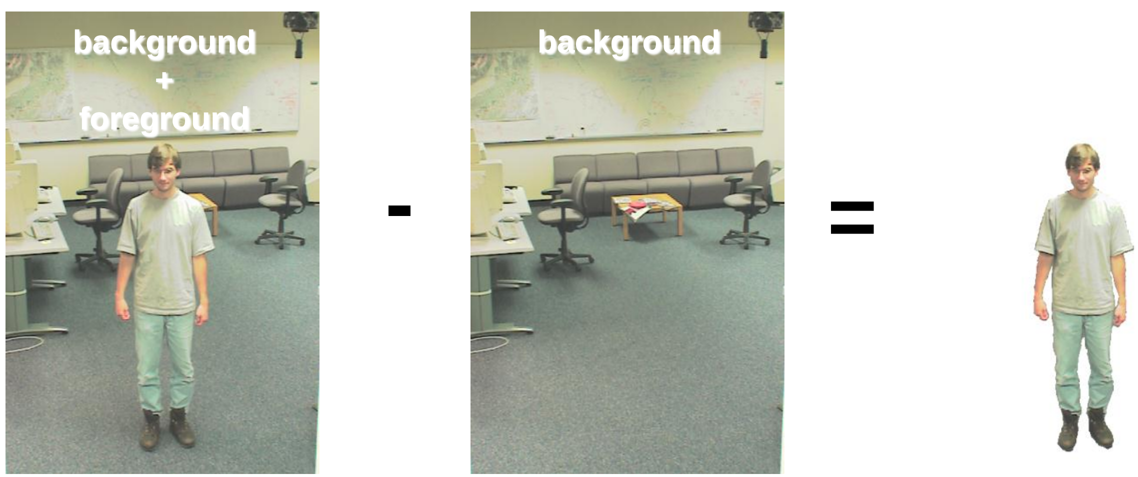
What is a Visual Hull?



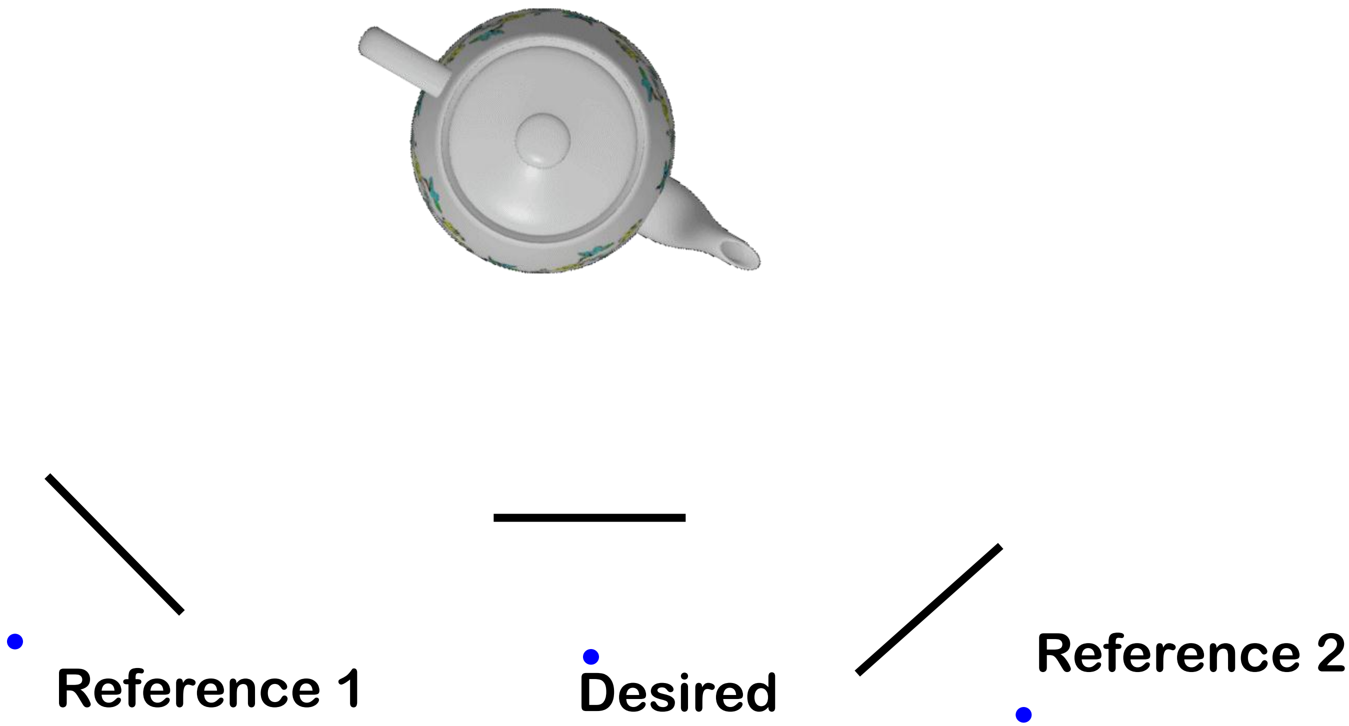
Why Use a Visual Hull?

- Can be computed robustly
- Can be computed efficiently

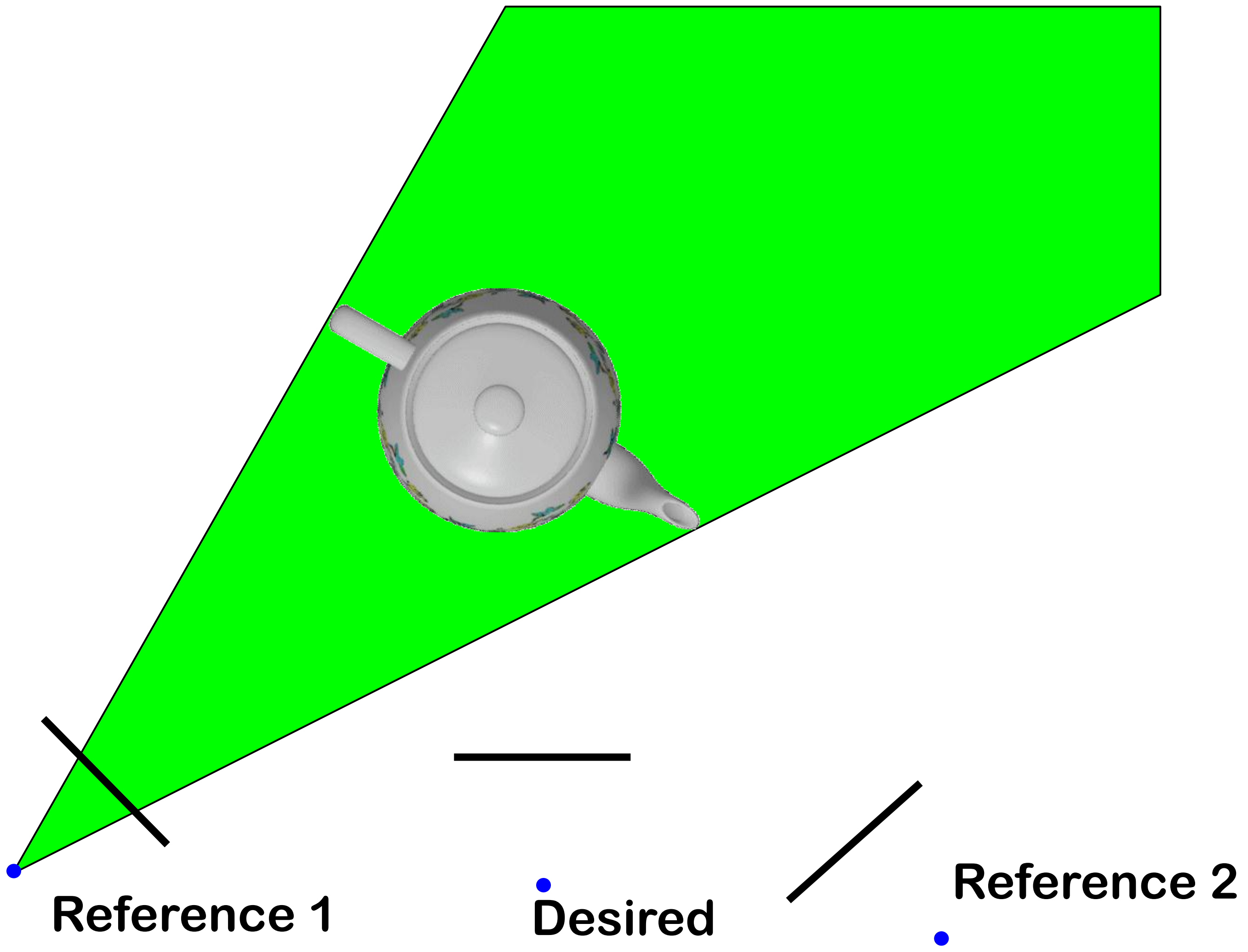




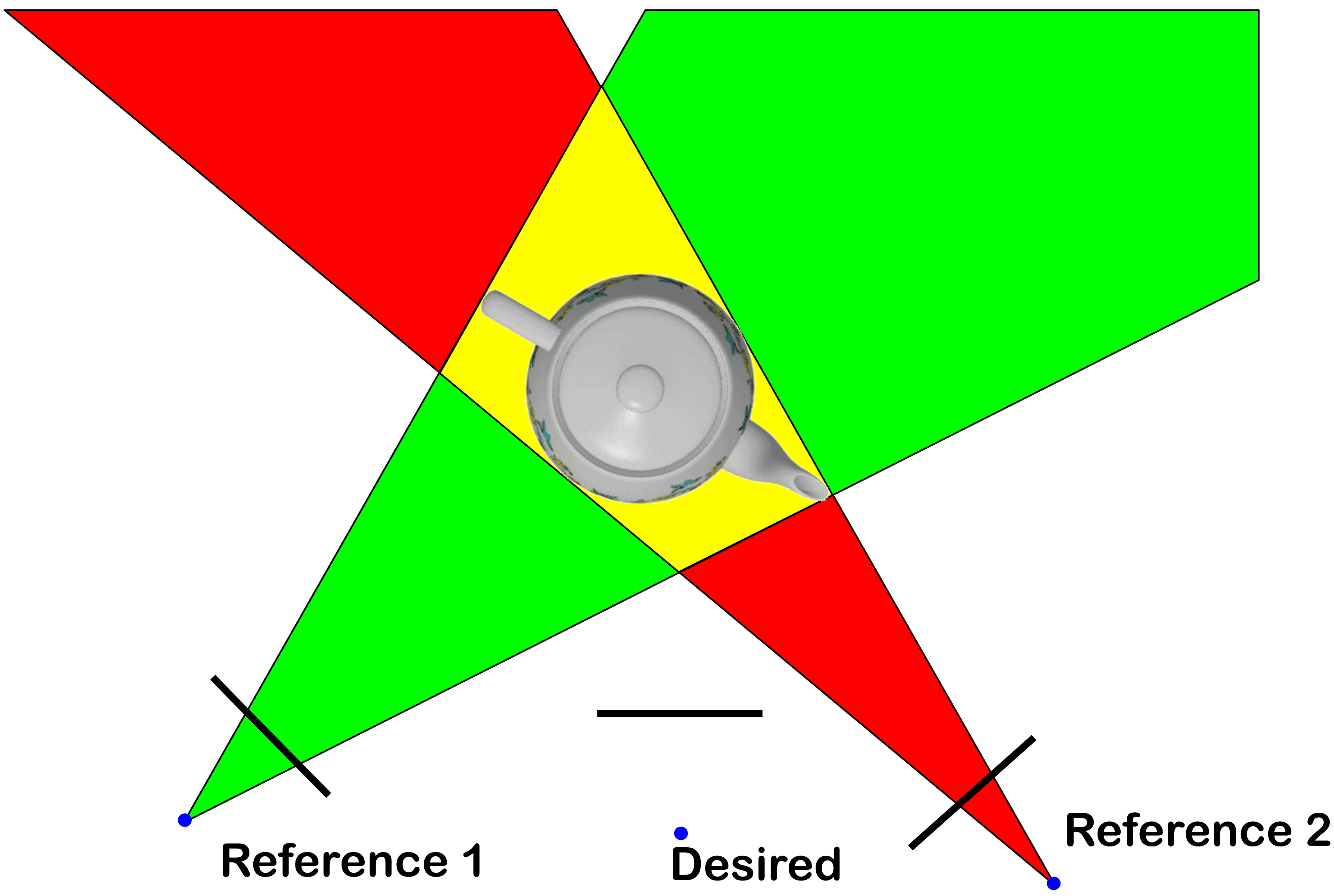
Rendering Visual Hulls



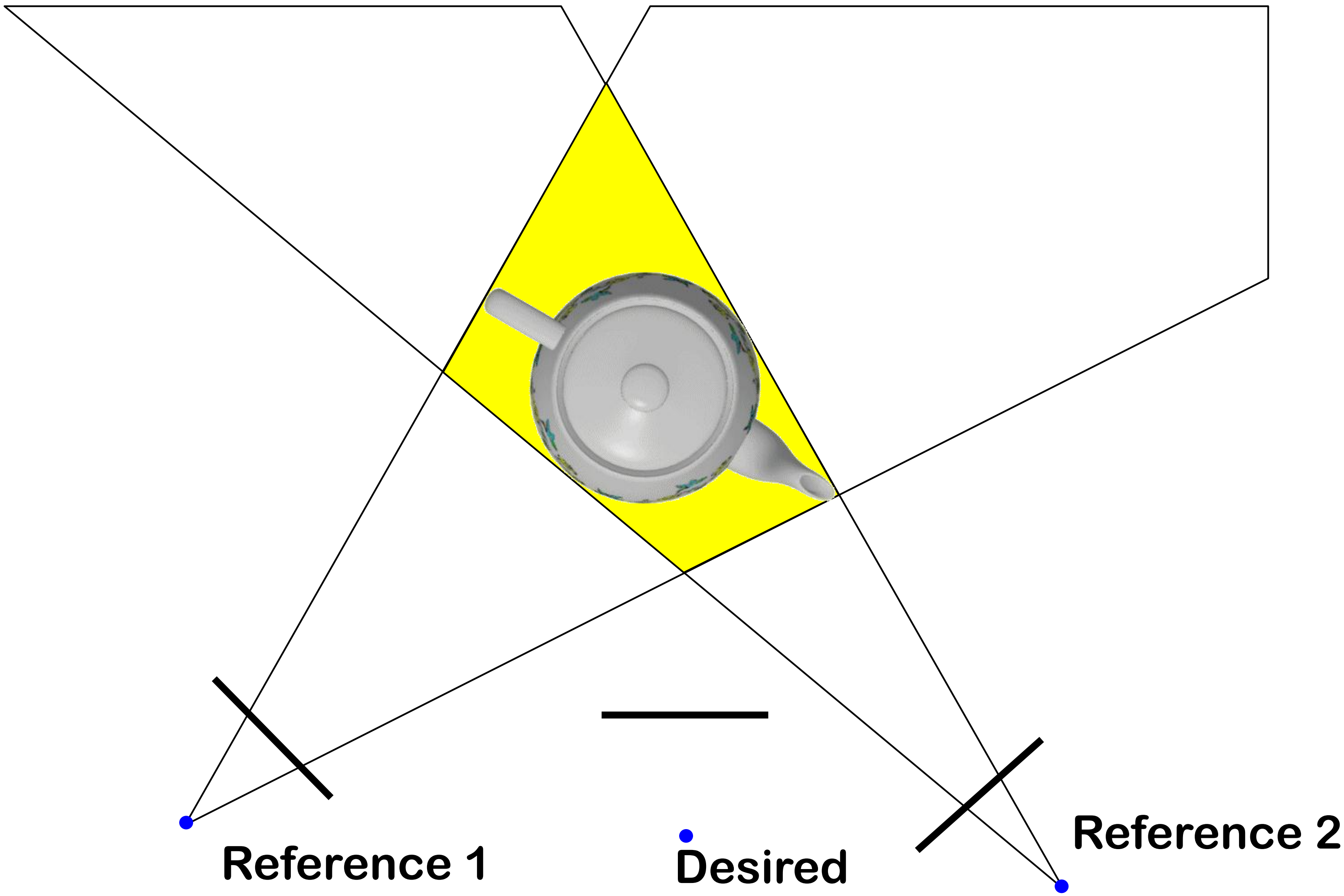
CSG then Ray Casting



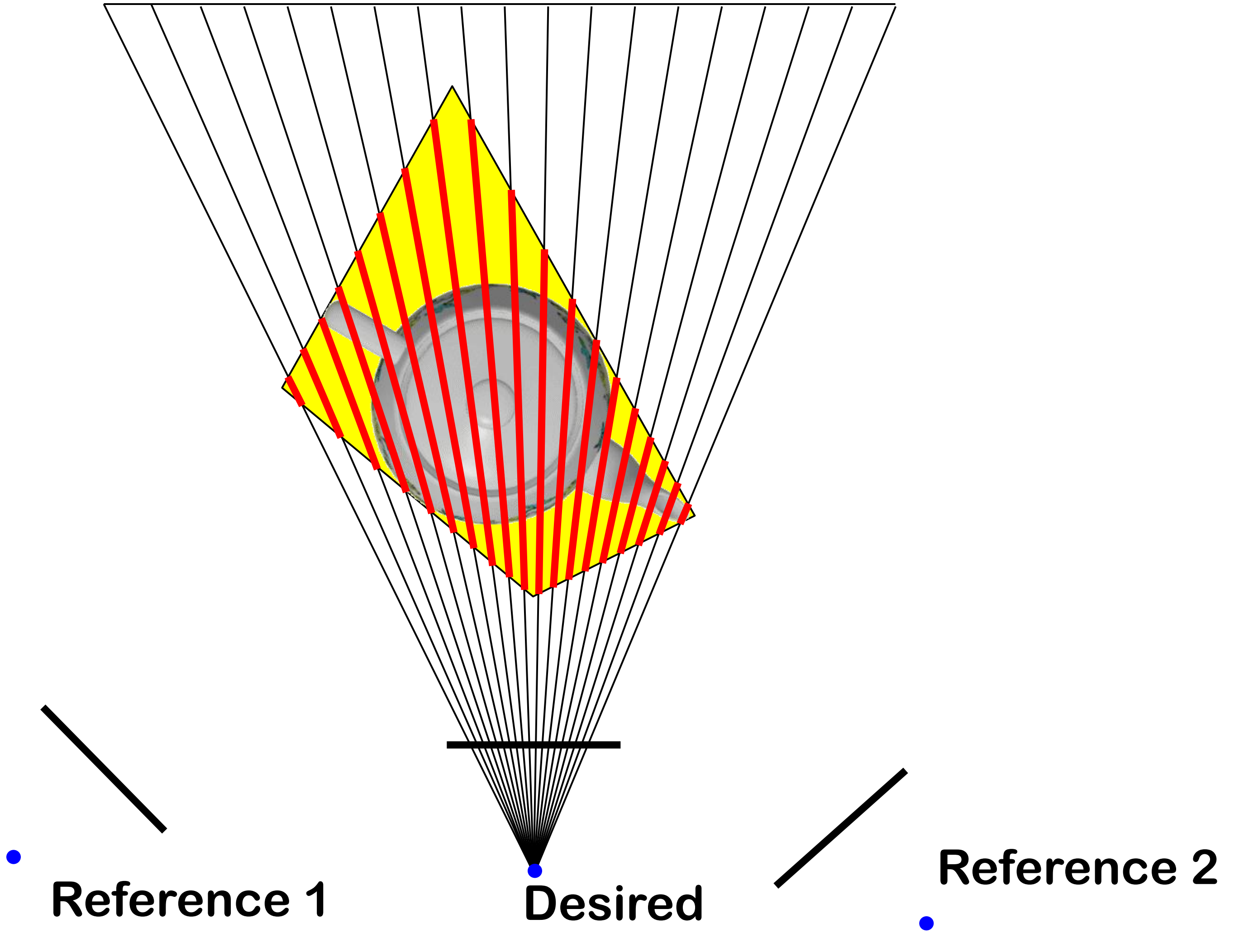
CSG then Ray Casting



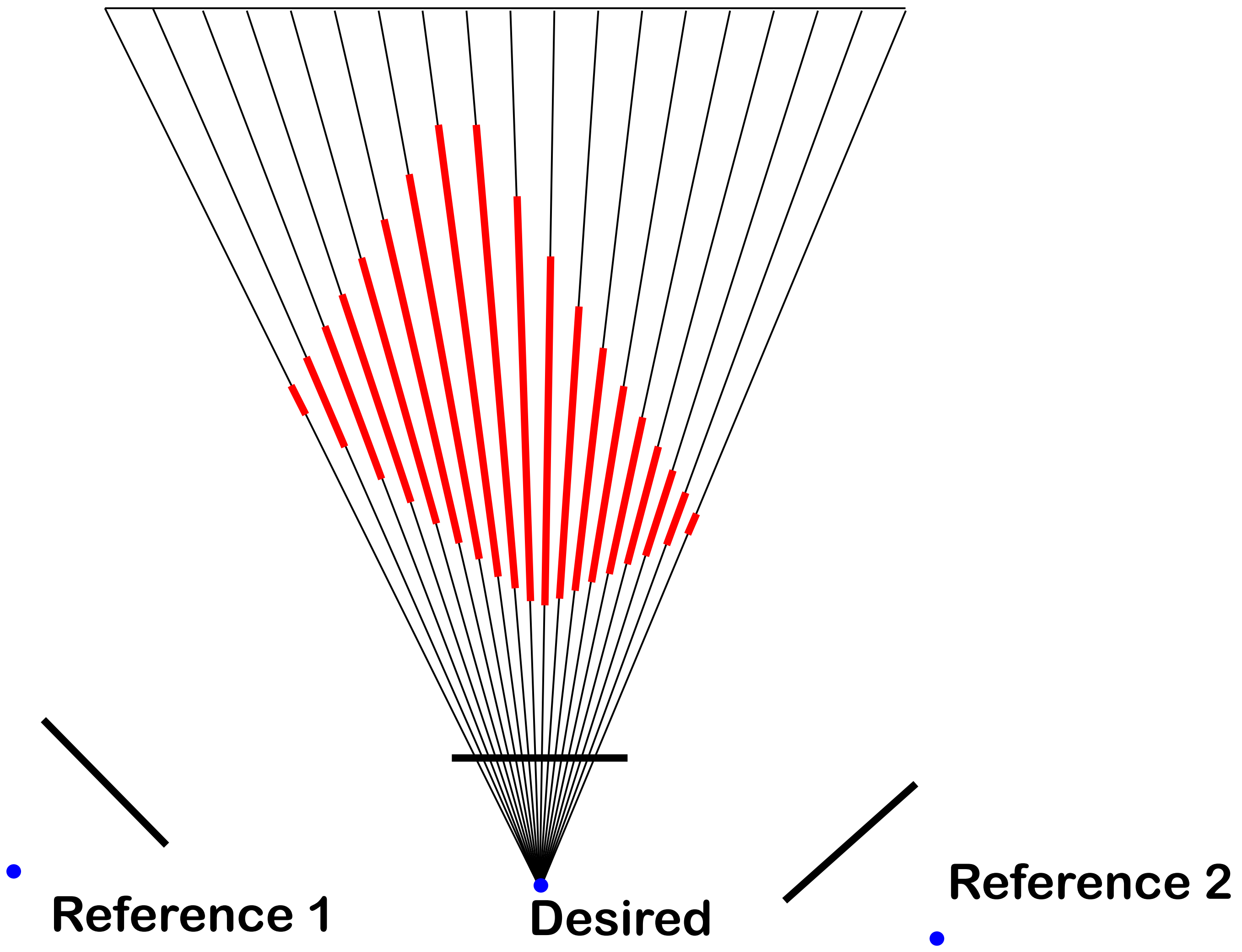
CSG then Ray Casting



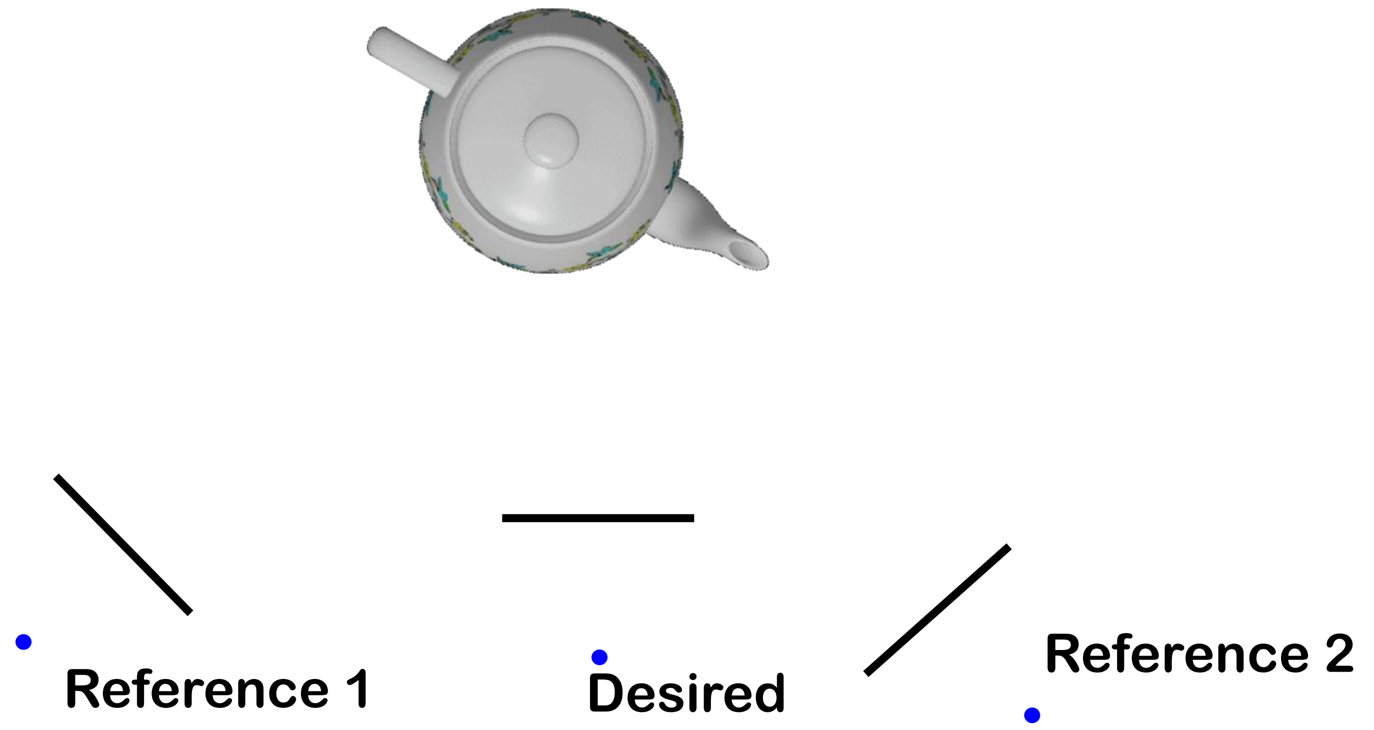
CSG then Ray Casting



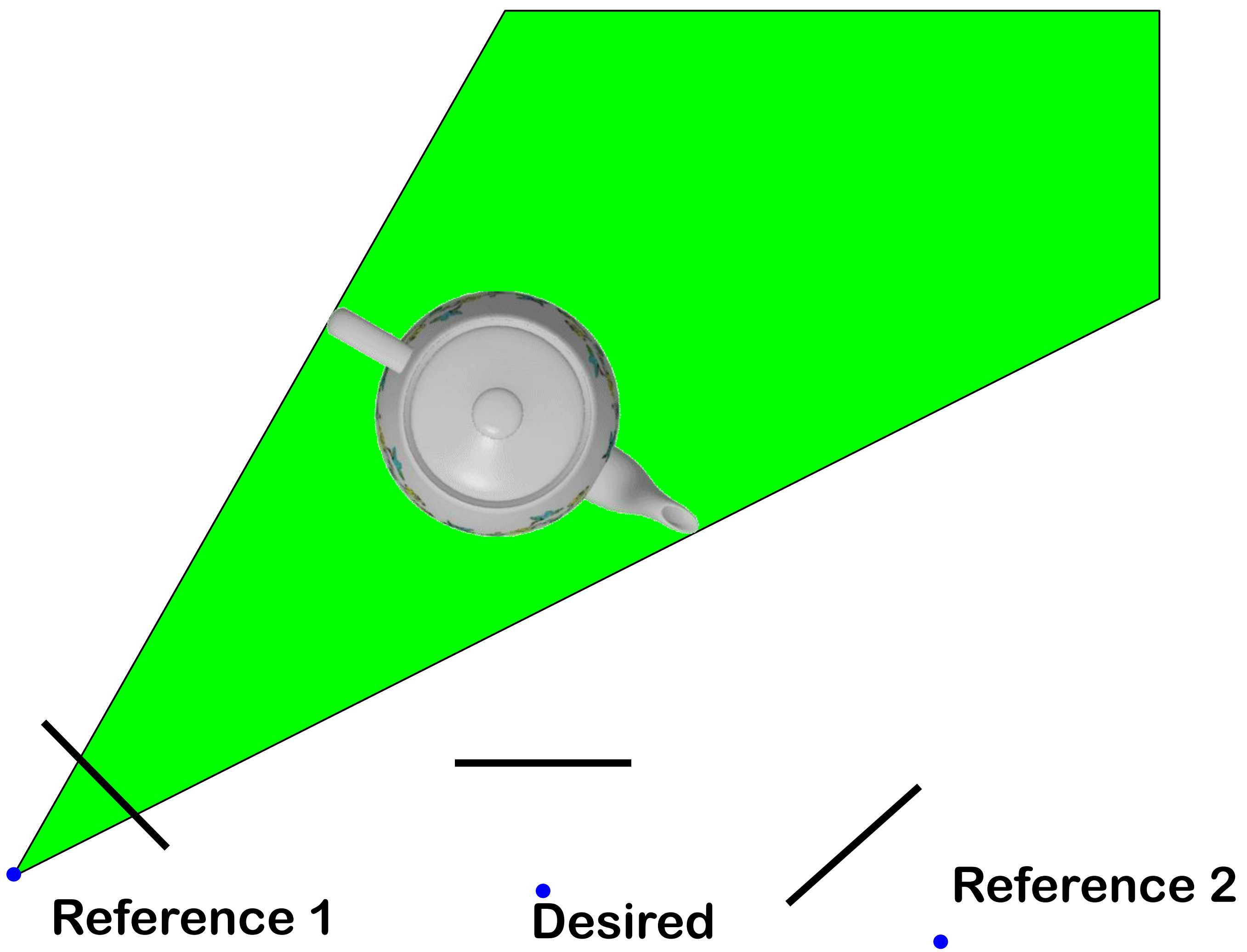
CSG then Ray Casting



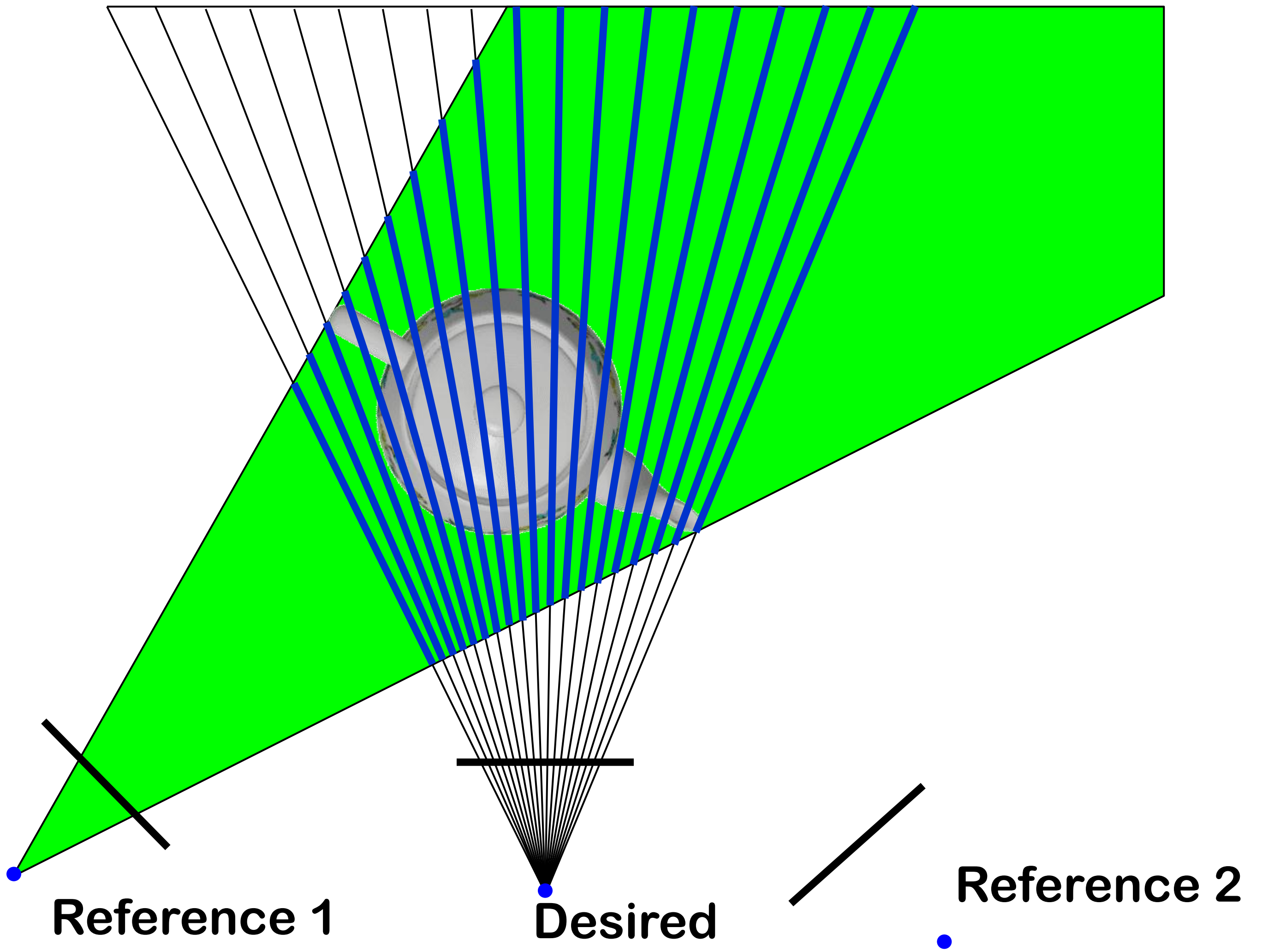
Ray Casting then Intersection



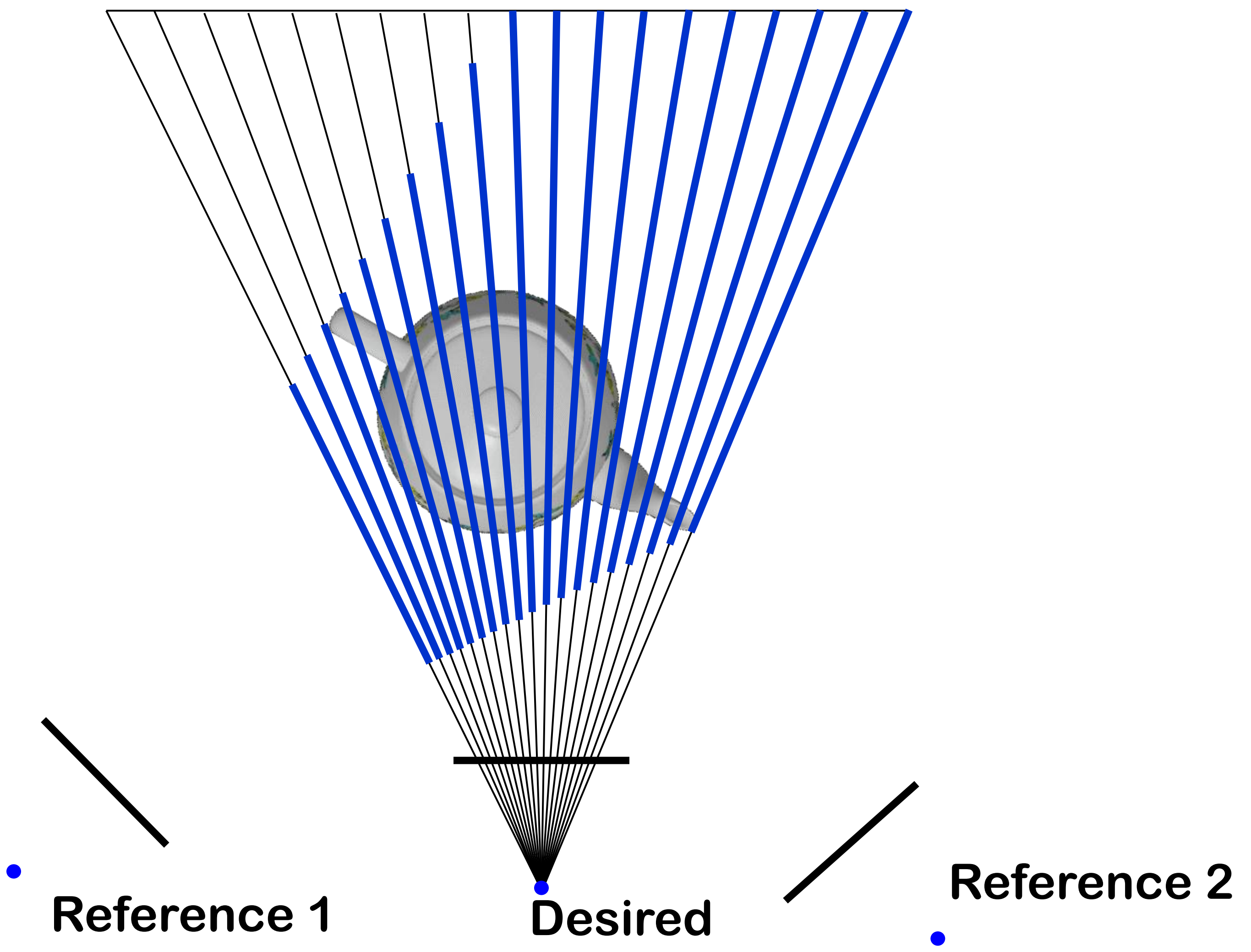
Ray Casting then Intersection



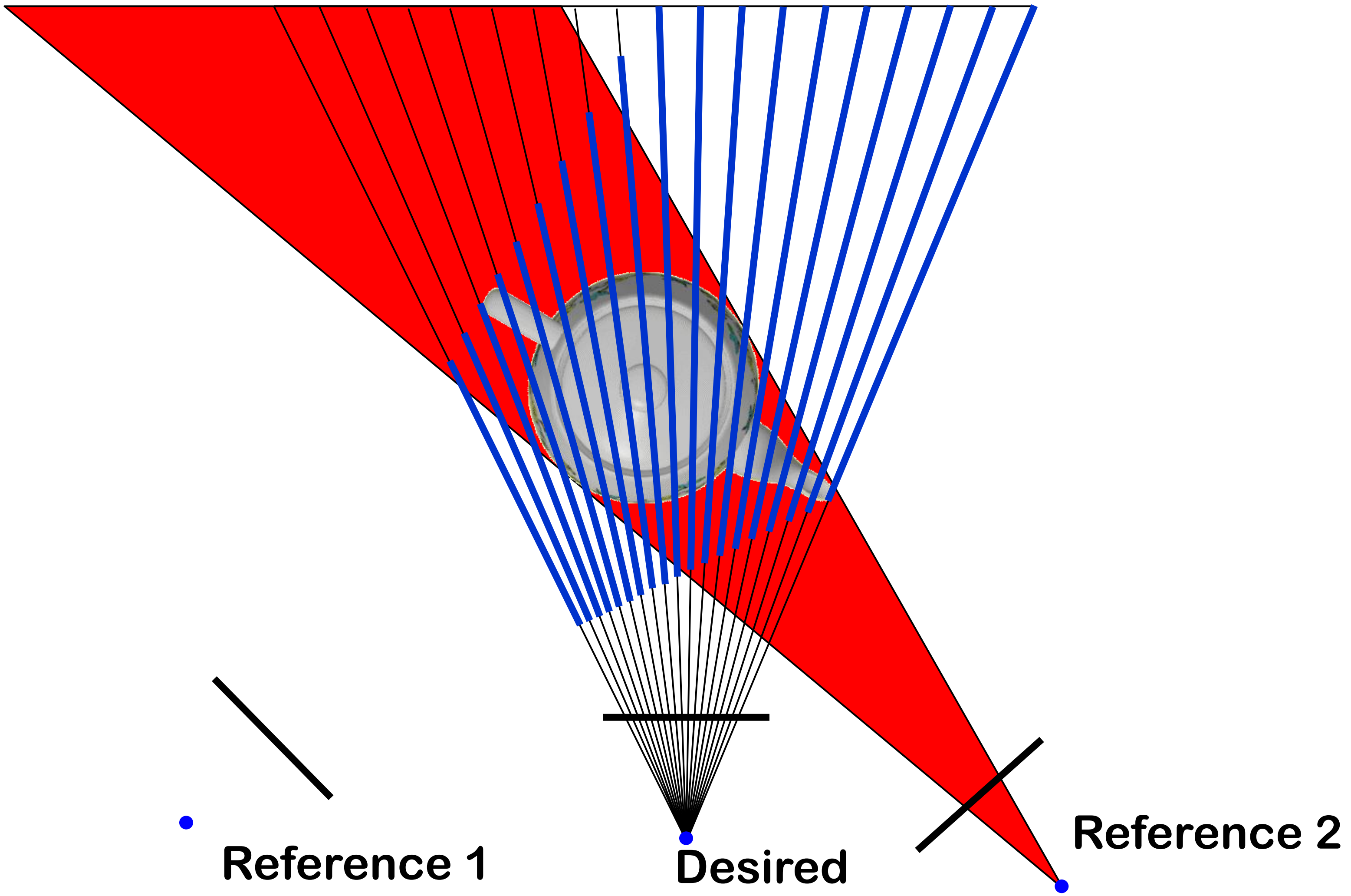
Ray Casting then Intersection



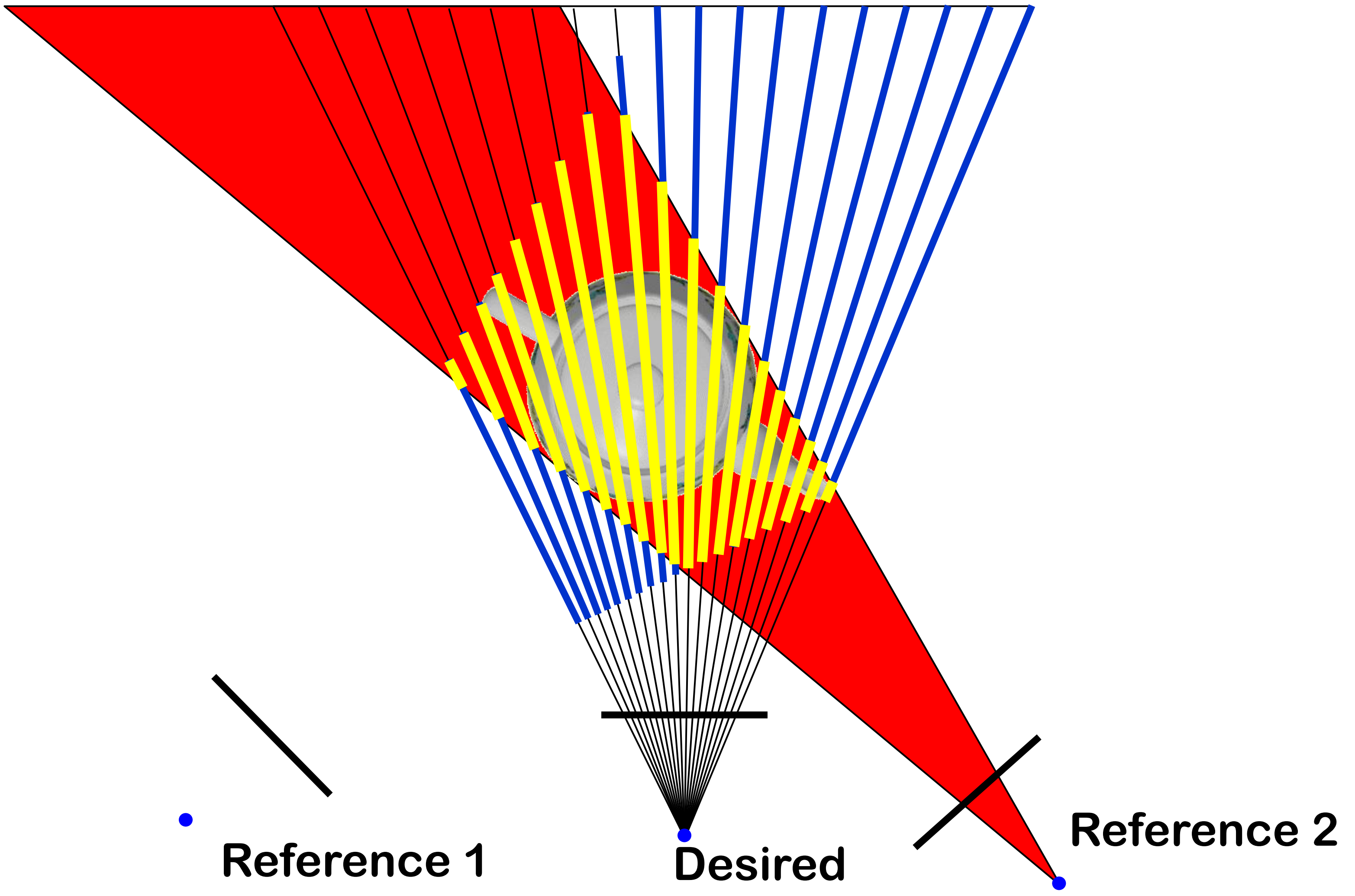
Ray Casting then Intersection



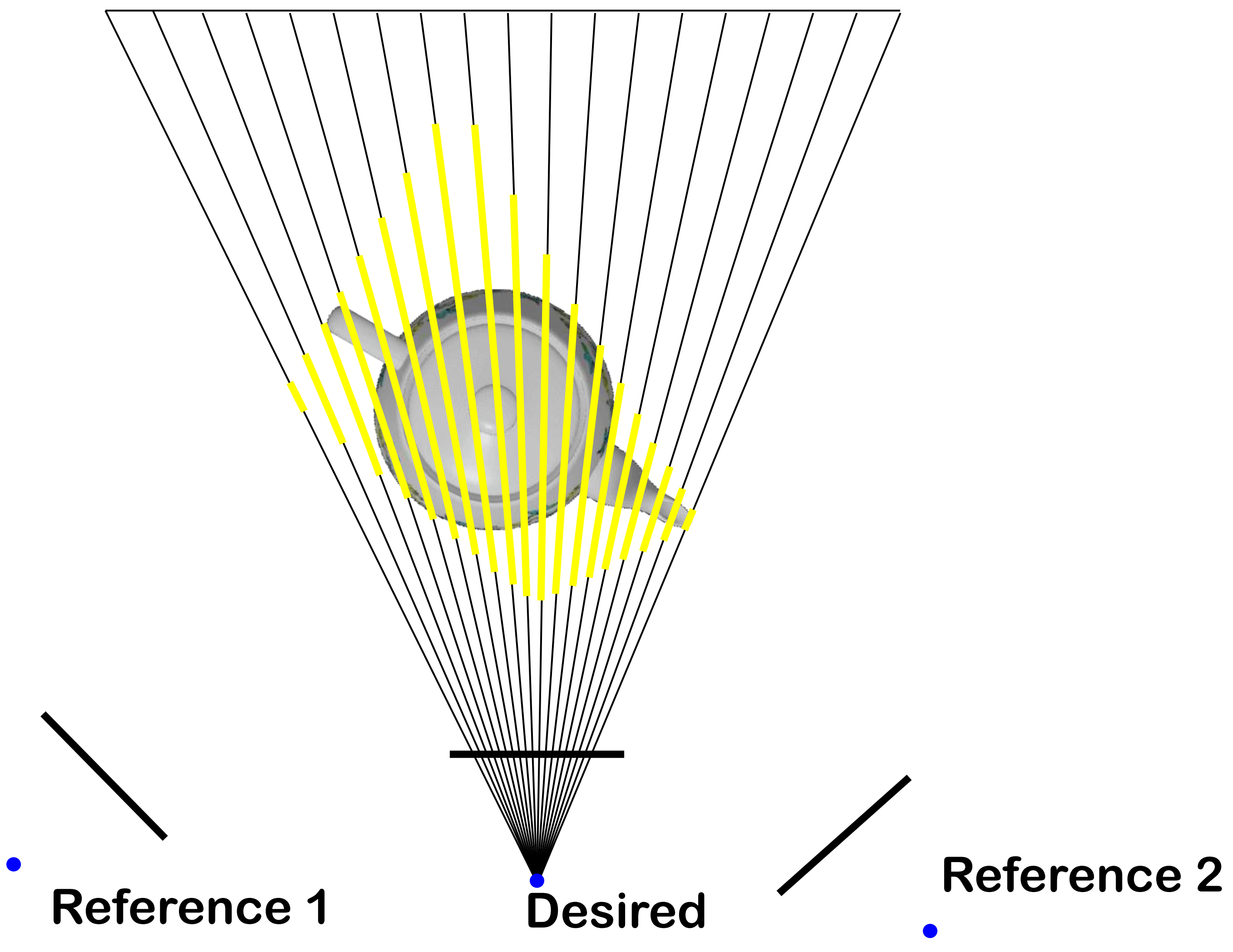
Ray Casting then Intersection



Ray Casting then Intersection



Ray Casting then Intersection



Ray Casting then Intersection

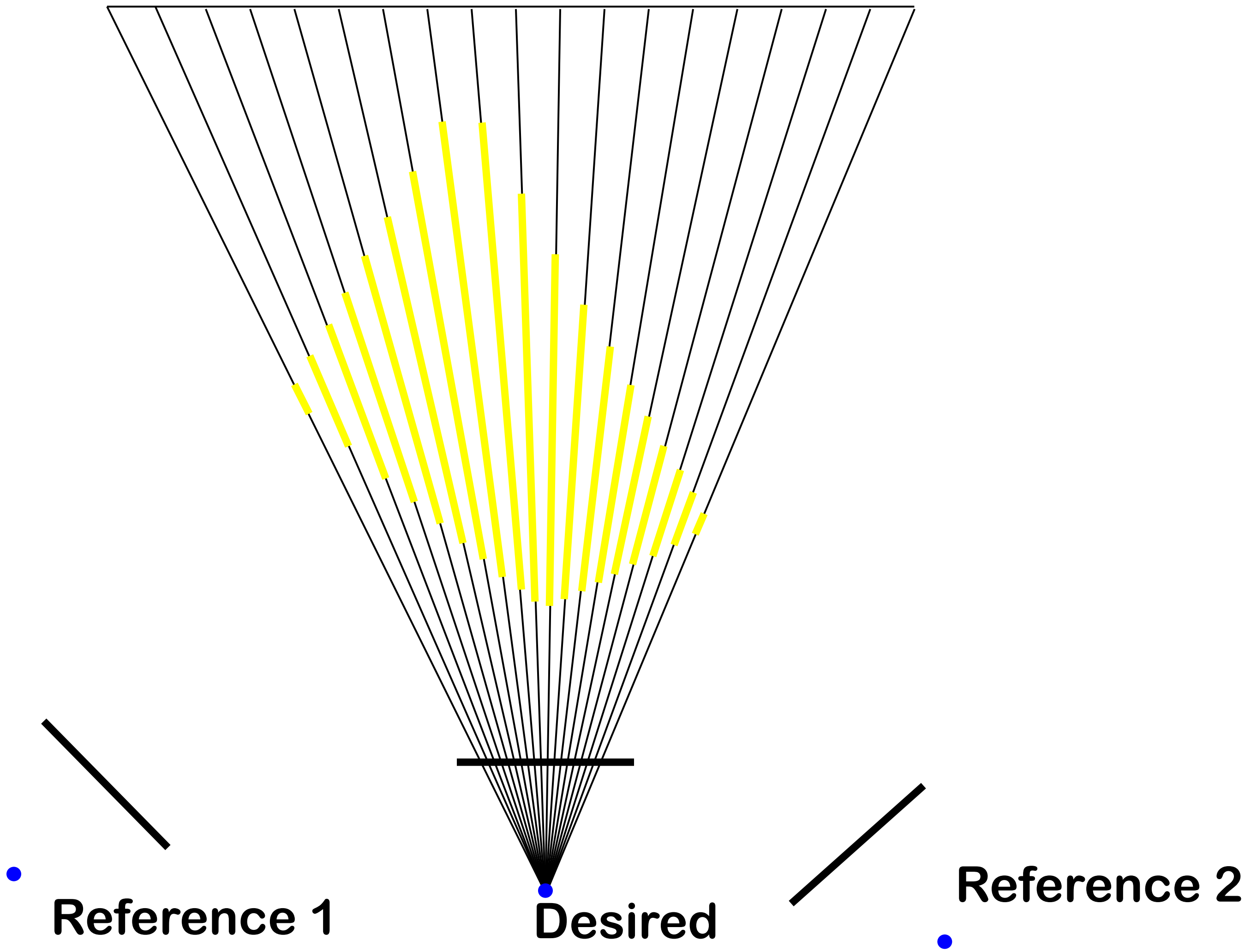


Image Based (2D) Intersection

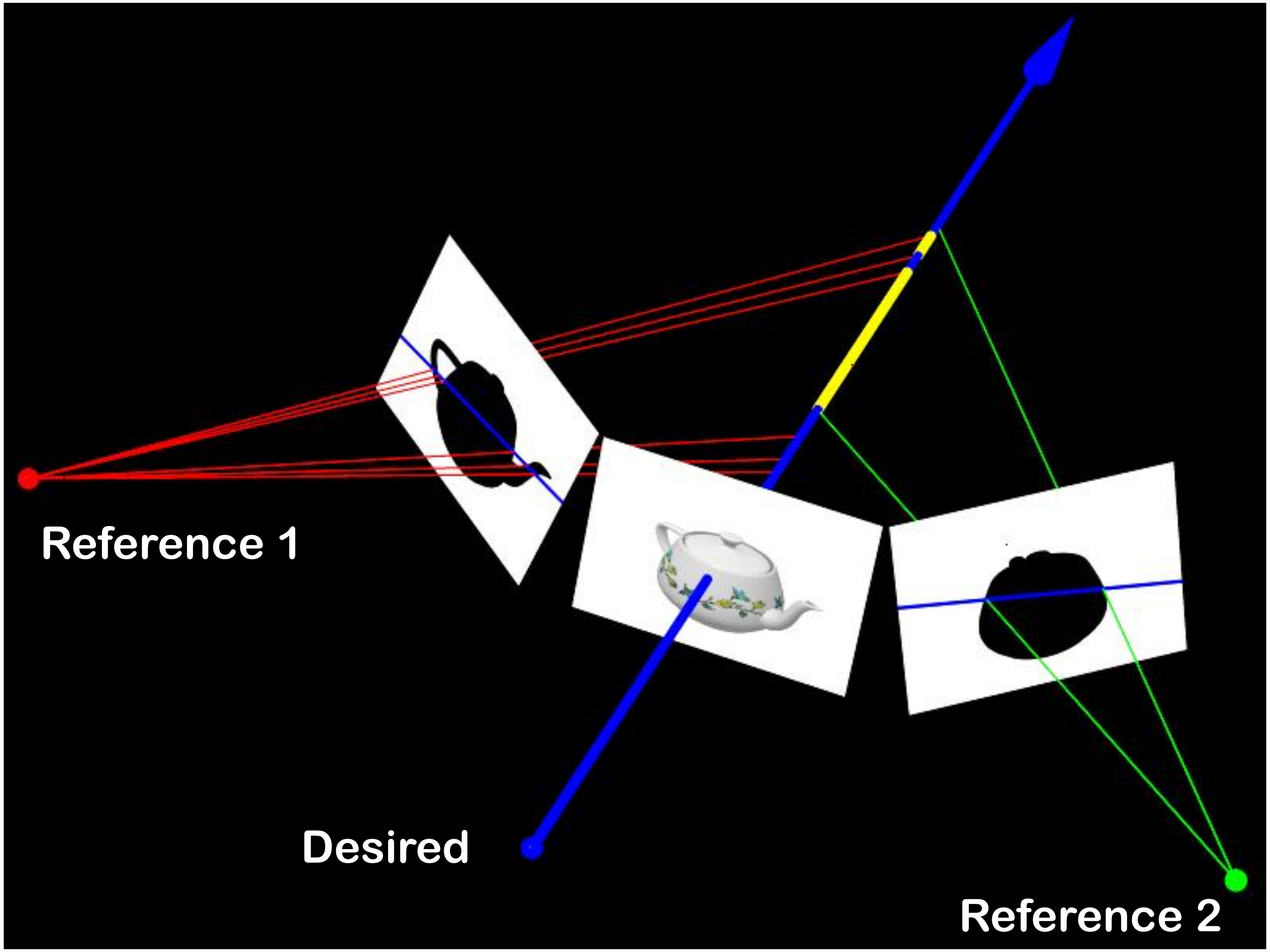
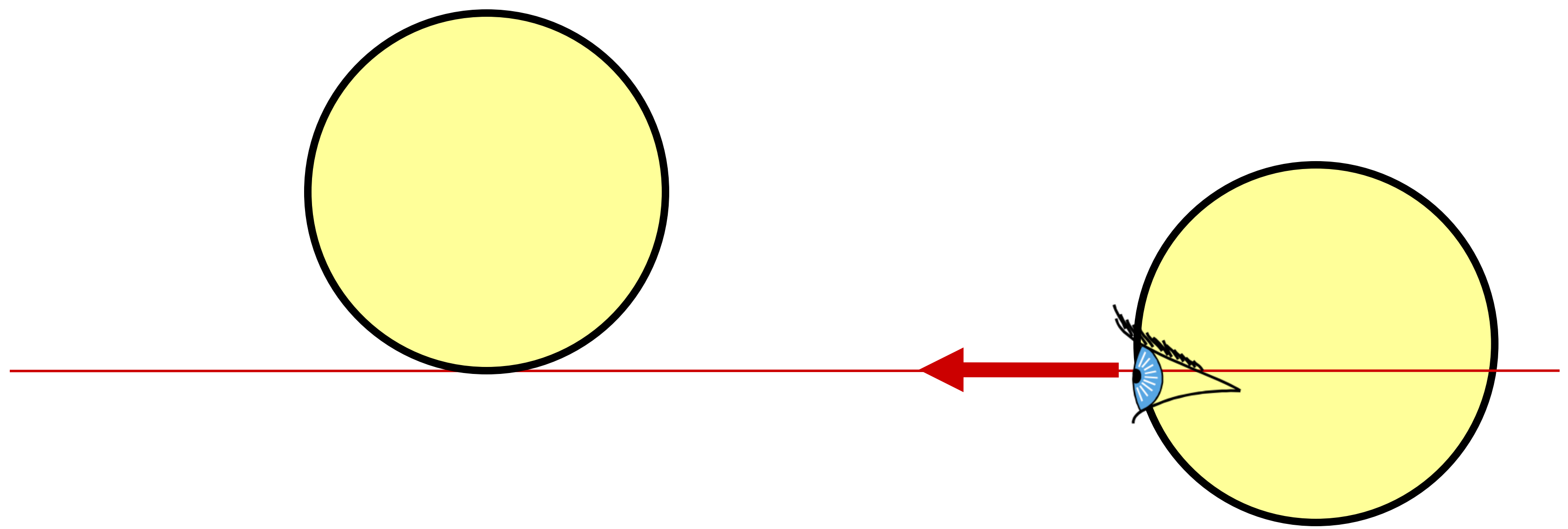


Image Based Visual Hulls

Questions?

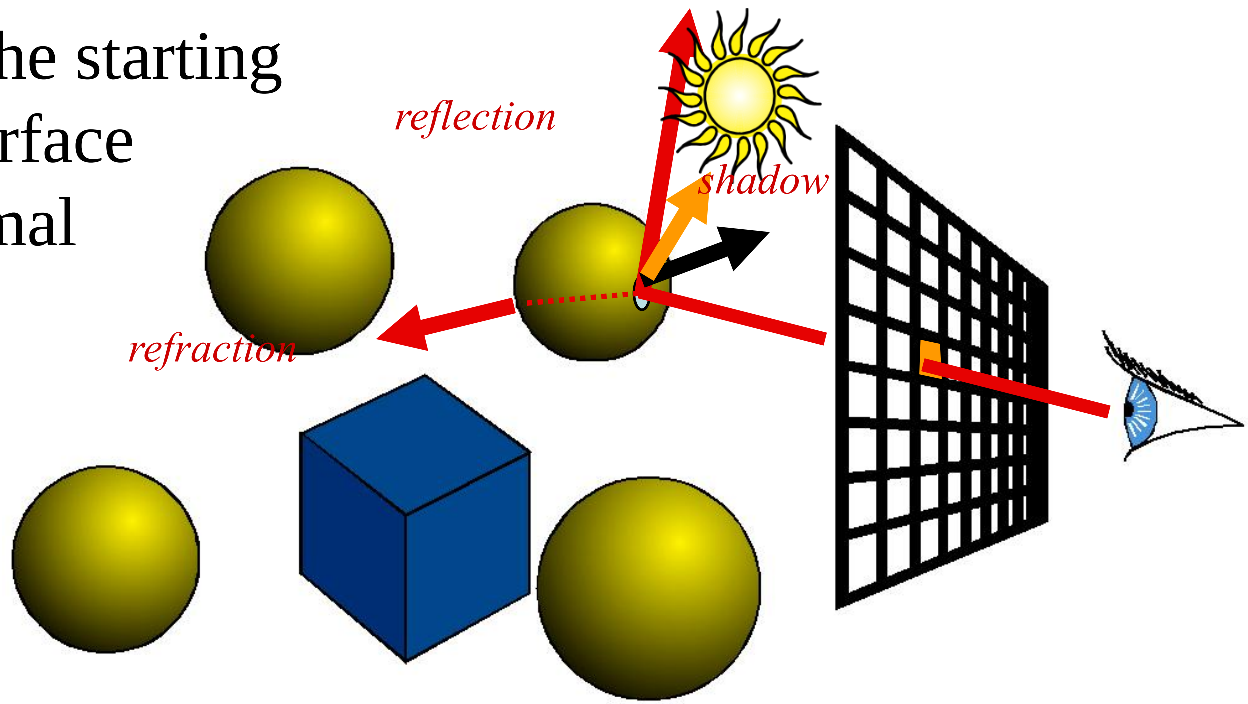
Precision

- What happens when
  - Ray Origin lies on an object?
  - Grazing rays?
- Problem with floating-point approximation



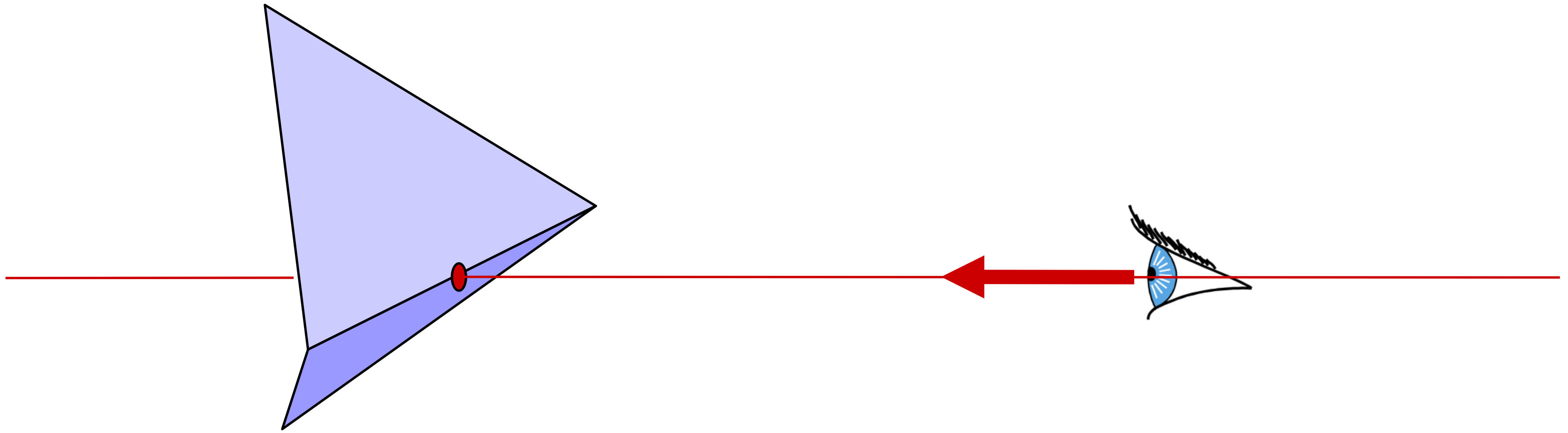
The Evil ε

- In ray tracing, do NOT report intersection for rays starting on surfaces
  - Secondary rays start on surfaces
  - Requires epsilons
  - Best to nudge the starting point off the surface e.g., along normal



The Evil ε

- Edges in triangle meshes
  - Must report intersection (otherwise not watertight)
  - Hard to get right



Questions?

Image by Henrik Wann Jensen

Transformations and Ray Casting

- We have seen that transformations such as affine transforms are useful for modeling & animation
- How do we incorporate them into ray casting?

Incorporating Transforms

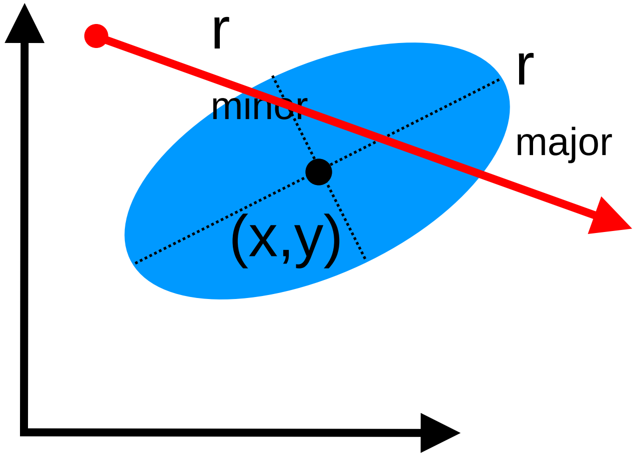
1. Make each primitive handle any applied transformations and produce a camera space description of its geometry

```
Transform {
    Translate { 1 0.5 0 }
    Scale { 2 2 2 }
    Sphere {
        center 0 0 0
        radius 1
    }
}
```

2. ...Or Transform the Rays

Primitives Handle Transforms

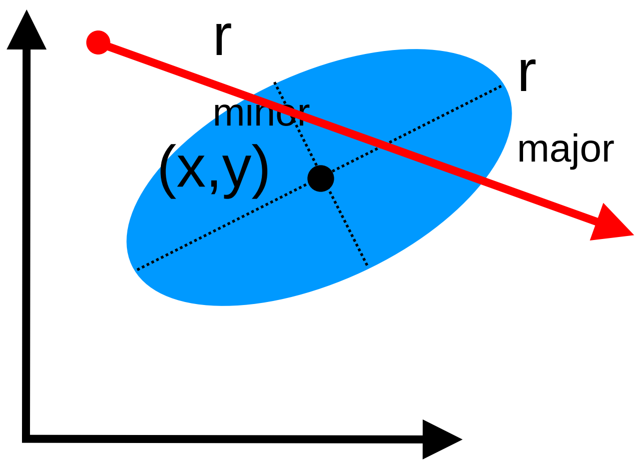
```
Sphere {
    center 3 2 0
    z_rotation 30
    r_major 2
    r_minor 1
}
```



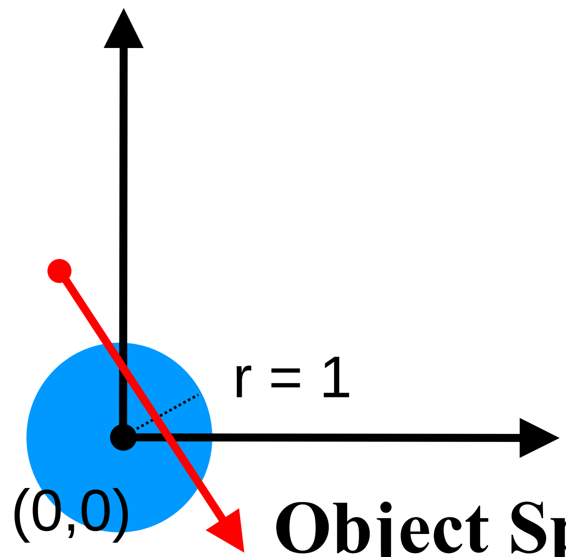
- Complicated for many primitives

Transform Ray

- Move the ray from *World Space* to *Object Space*



World Space



Object Space

$$p_{WS} = \mathbf{M} \; p_{OS}$$

$$p_{OS} = \mathbf{M}^{-1} \; p_{WS}$$

Transform Ray

- New origin:

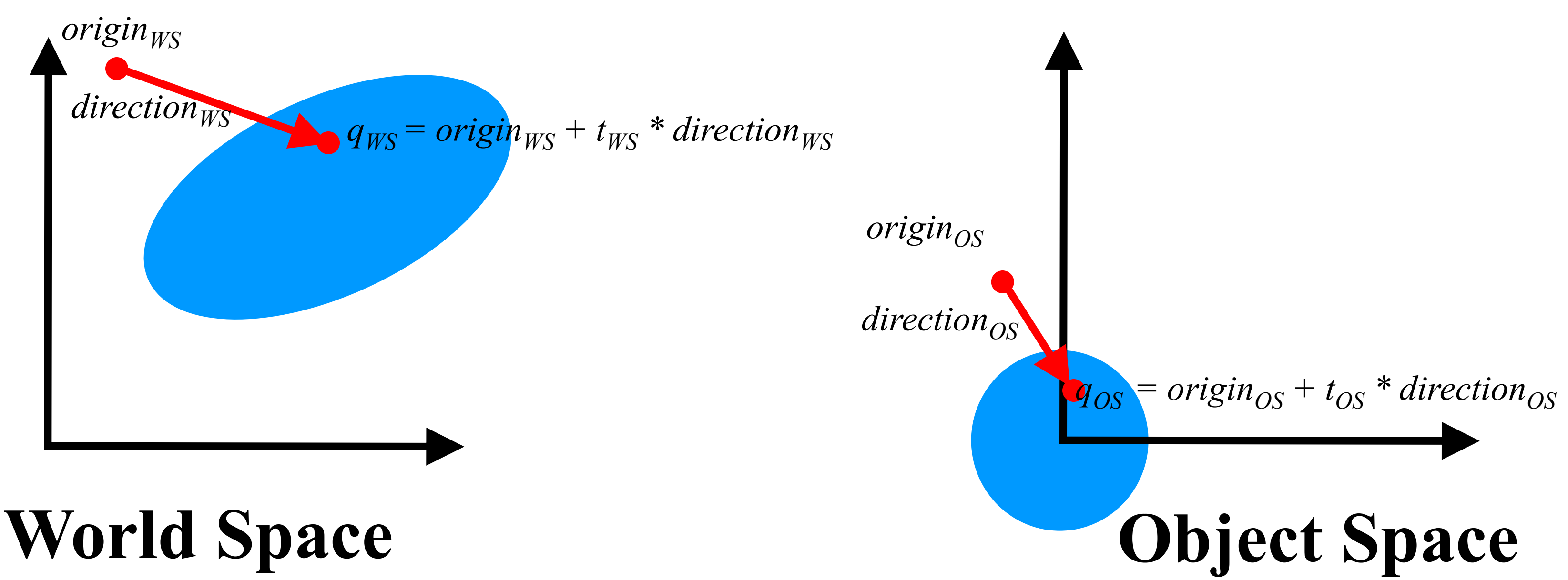
$$origin_{OS} = \mathbf{M}^{-1}\ origin_{WS}$$

- New direction:

$$direction_{OS} = \mathbf{M}^{-1}\ (origin_{WS} + 1 * direction_{WS}) \ - \ \mathbf{M}^{-1}\ origin_{WS}$$

$$direction_{OS} = \mathbf{M}^{-1}\ direction_{WS}$$

Note that the w component of direction is 0



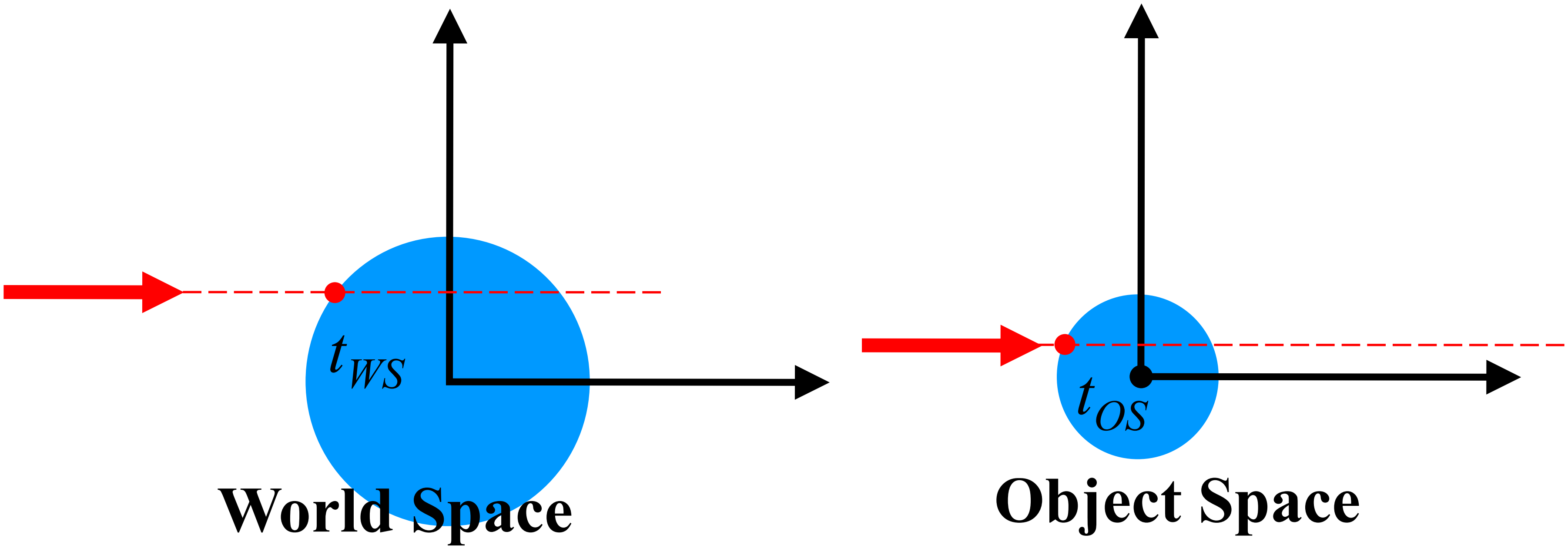
What About *t* ?

- If **M** includes scaling, $direction_{\mathrm{OS}}$ ends up NOT be normalized after transformation

- Two solutions
  - Normalize the direction
  - Do not normalize the direction

1. Normalize Direction

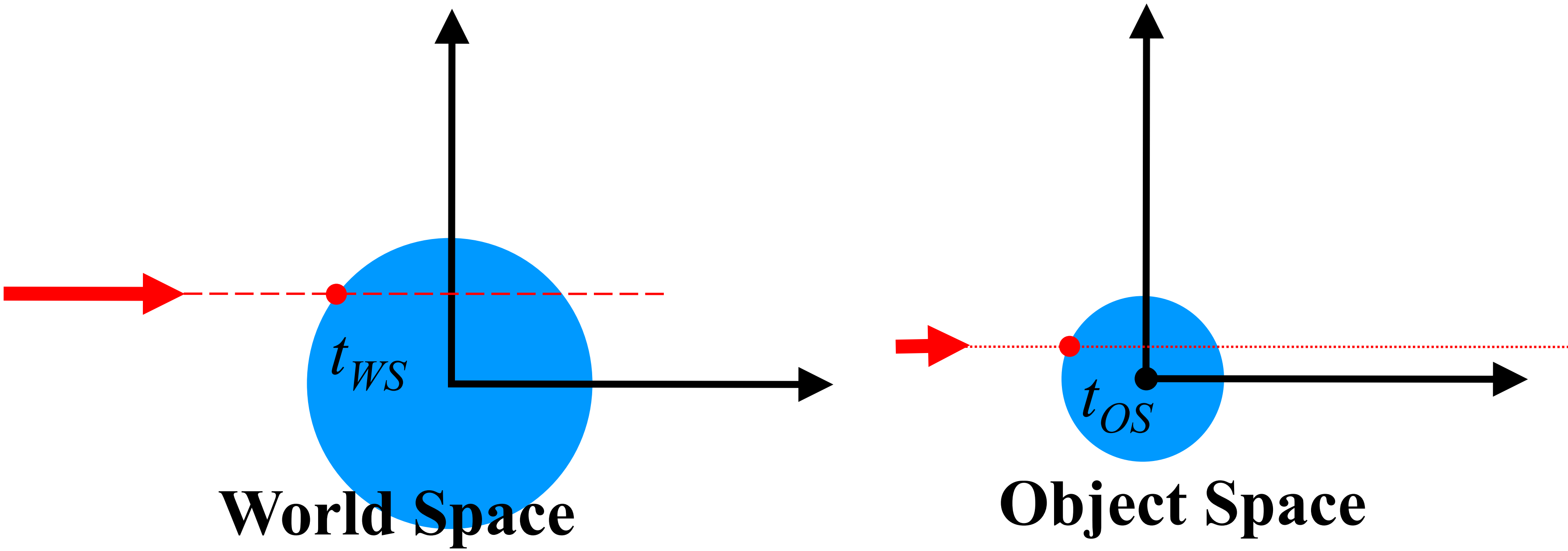
- $t_{OS} \neq t_{WS}$
  and must be rescaled after intersection
  ==> One more possible failure case...



2. Do Not Normalize Direction

Highly recommended

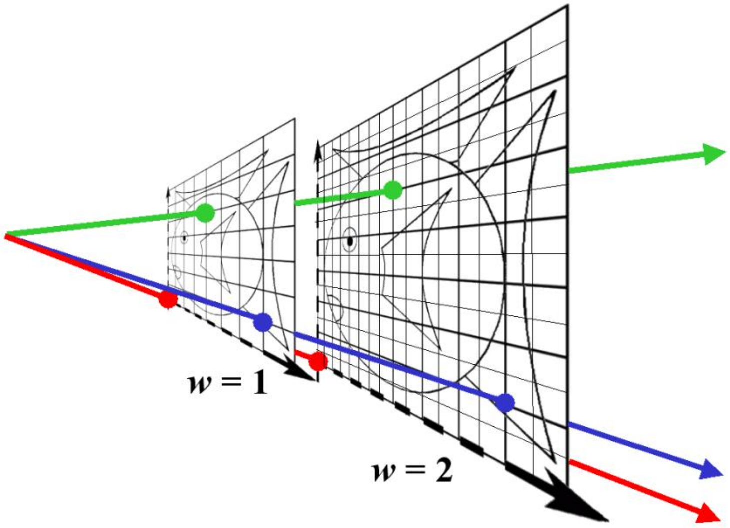
- $t_{OS} = t_{WS}$ → convenient!
- But you should not rely on t_{OS} being true distance in intersection routines (e.g. a≠1 in ray-sphere test)



Transforming Points & Directions

- Transform point

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} ax+by+cz+d \\ ex+fy+gz+h \\ ix+jy+kz+l \\ 1 \end{pmatrix}$$



- Transform direction

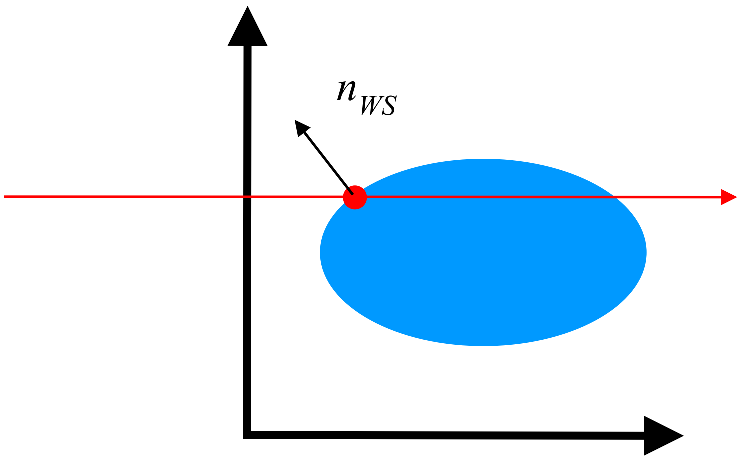
$$\begin{pmatrix} x' \\ y' \\ z' \\ 0 \end{pmatrix} = \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix} = \begin{pmatrix} ax+by+cz \\ ex+fy+gz \\ ix+jy+kz \\ 0 \end{pmatrix}$$

Homogeneous Coordinates: (x,y,z,w)

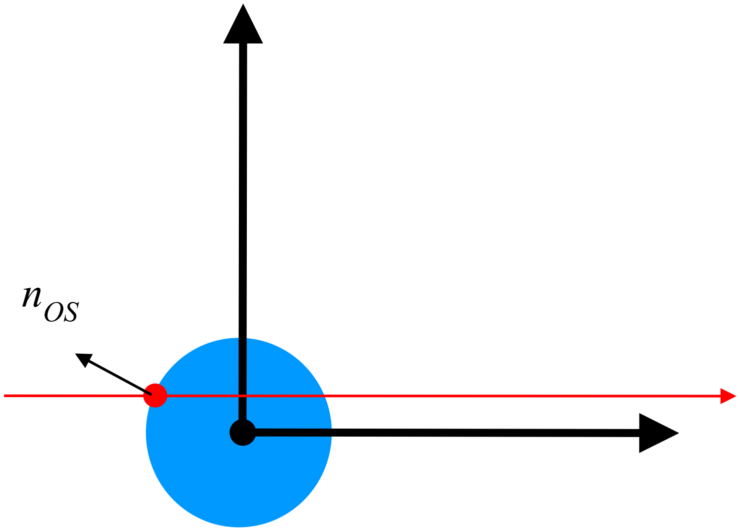
$w = 0$ is a point at infinity (direction)

- If you do not store w you need different routines to apply **M** to a point and to a direction ==> Store everything in 4D!

Recap: How to Transform Normals?



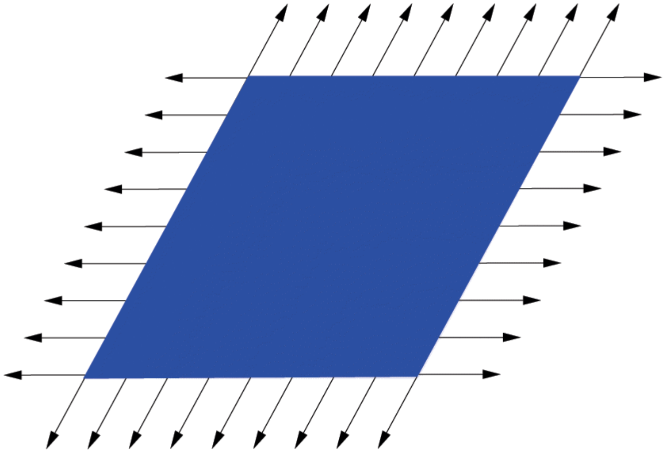
World Space

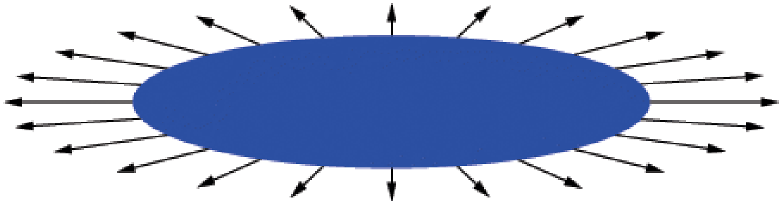


Object Space

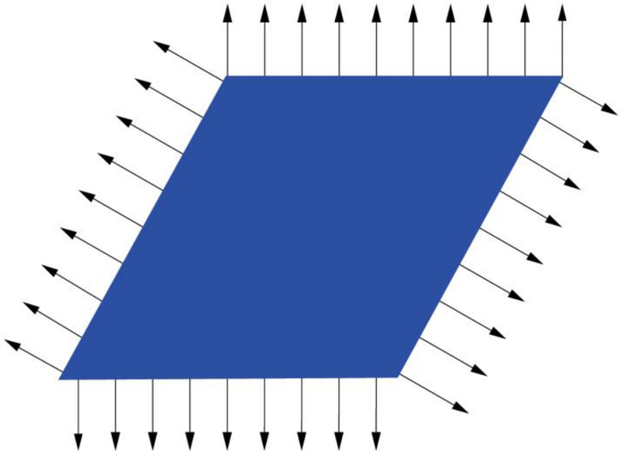
Transformation for Shear and Scale

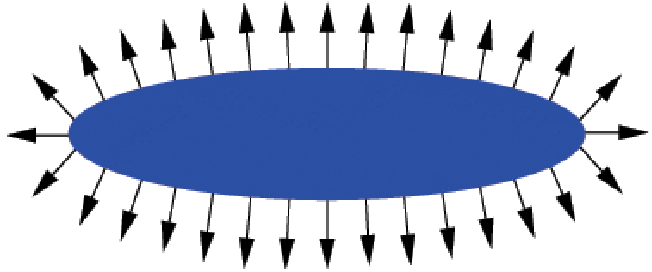
Incorrect
Normal
Transformation





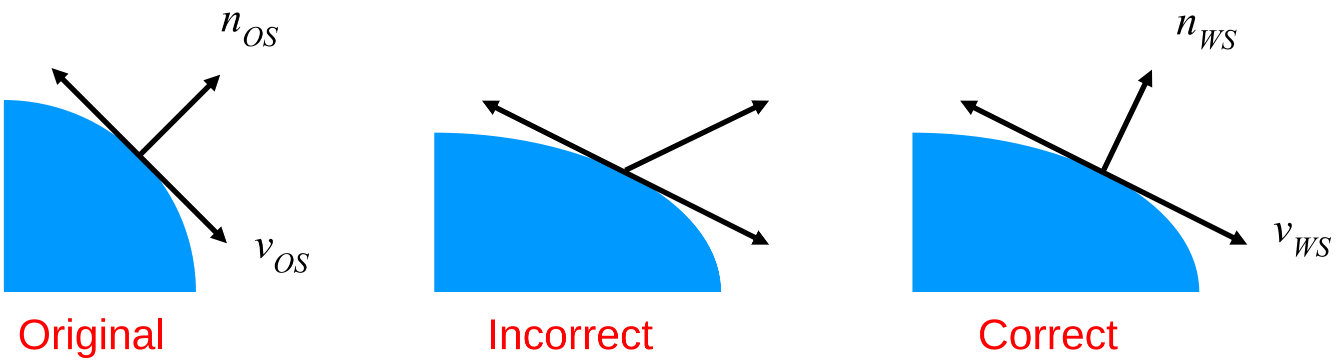
Correct
Normal
Transformation





So How Do We Do It Right?

- Think about transforming the *tangent plane* to the normal, not the normal *vector*



Pick any vector v_{OS} in the tangent plane, how is it transformed by matrix $\mathbf{M}$?

$$v_{WS} = \mathbf{M} \; v_{OS}$$

Transform Tangent Vector v

v is perpendicular to normal n:

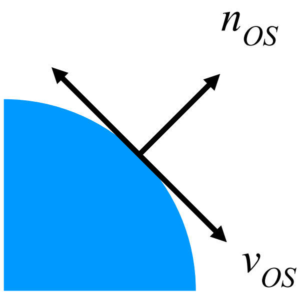
Dot product

$$n_{OS}^{\mathbf{T}}\, v_{OS} = 0$$

$$n_{OS}^{\mathbf{T}}\, (\mathbf{M}^{-1}\ \mathbf{M})\, v_{OS} = 0$$

$$(n_{OS}^{\mathbf{T}}\ \mathbf{M}^{-1})\, (\mathbf{M}\ \ v_{OS}) = 0$$

$$(n_{OS}^{\mathbf{T}}\ \mathbf{M}^{-1})\, v_{WS} = 0$$

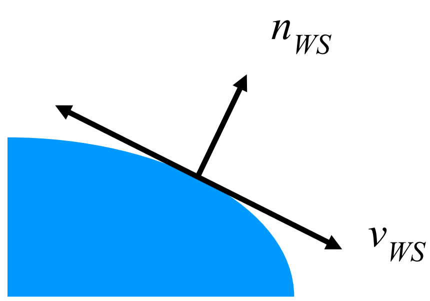


v_{WS} is perpendicular to normal n_{WS}:

$$n_{WS}^{\mathbf{T}}\, v_{WS} = 0$$

$$n_{WS}^{\mathbf{T}} = n_{OS}^{\mathbf{T}}\, (\mathbf{M}^{-1})$$

$$\boxed{n_{WS} = (\mathbf{M}^{-1})^{\mathbf{T}}\, n_{OS}}$$

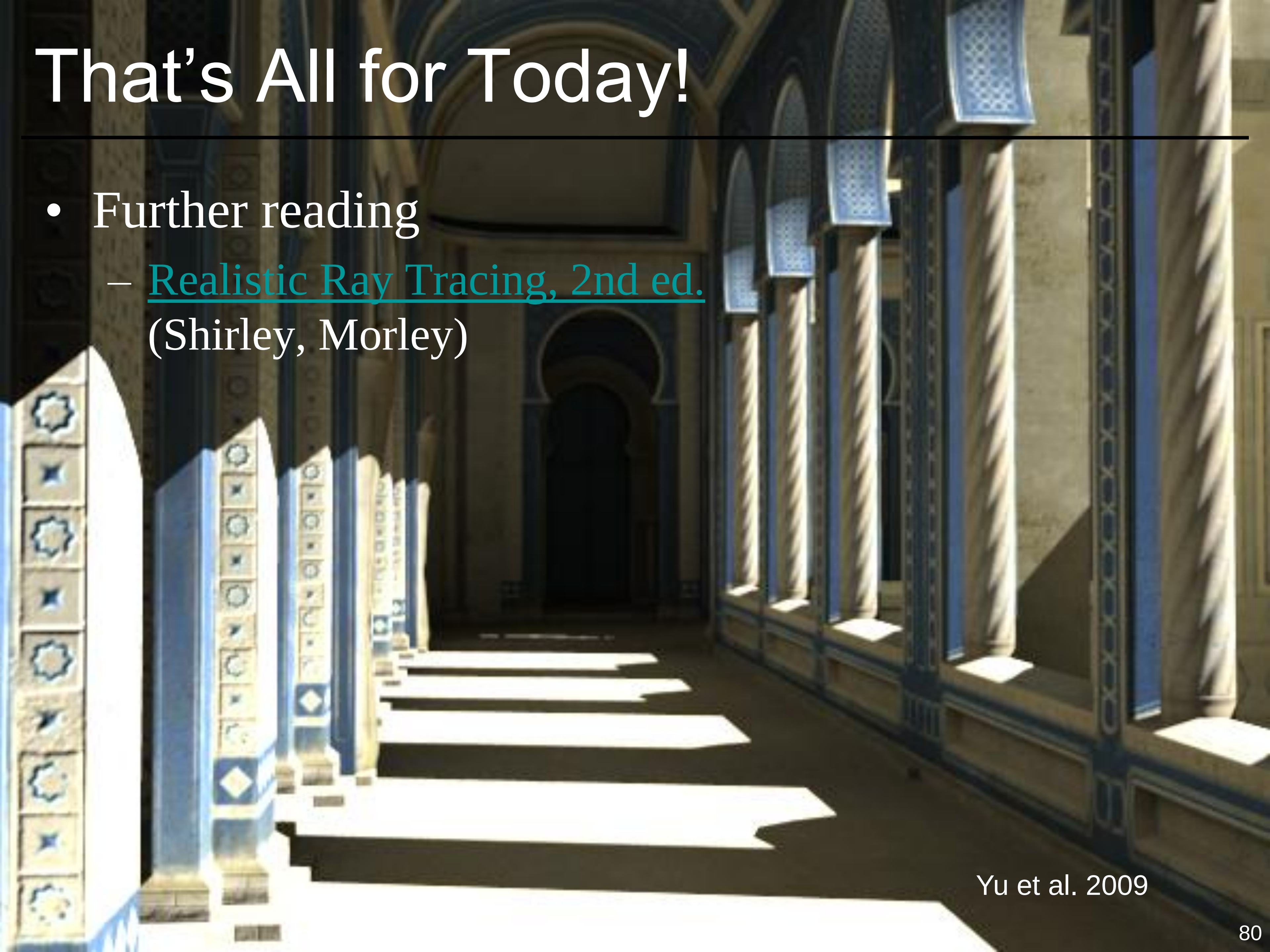


Position, Direction, Normal

- Position
  - transformed by the full homogeneous matrix **M**
- Direction
  - transformed by **M** except the translation component
- Normal
  - transformed by $\mathbf{M}^{-T}$, no translation component

That's All for Today!

- Further reading
  - Realistic Ray Tracing, 2nd ed. (Shirley, Morley)



Yu et al. 2009