CS-C3100 Computer Graphics

12.1 Ray Tracing: Intersections

Jaakko Lehtinen
with lots of material from Frédo Durand

In this Video

- **Transformations & Ray Tracing**
- Object-oriented ray tracer design
- Precision issues
- Fun stuff: Constructive Solid Geometry (CSG)

Transformations and Ray Tracing

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

Incorporating Transforms

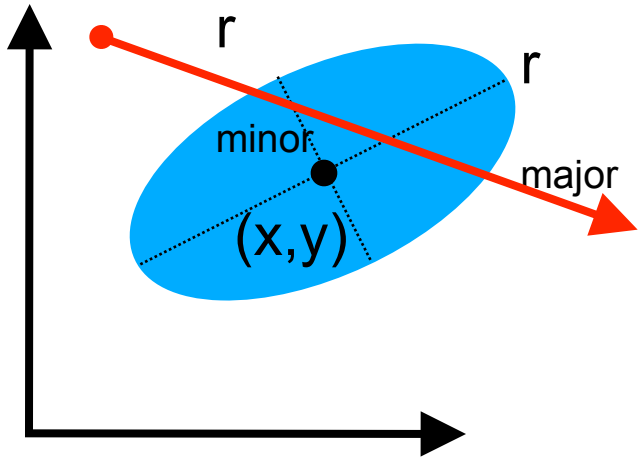
1. Make each primitive handle any applied transformations and produce a camera space description of its geometry

```
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?

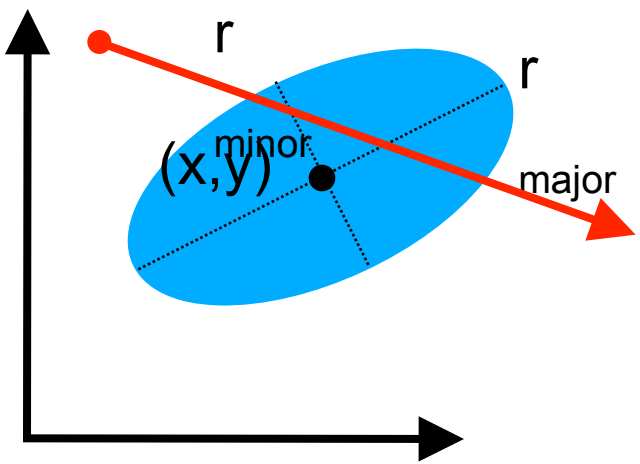
```
Sphere {
     center 3 2 0
     z_rotation 30
     r_major 2
     r_minor 1
}
```



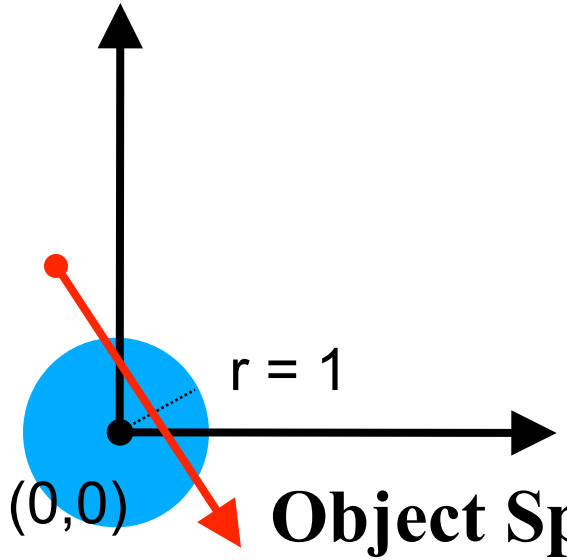
- Complicated for many primitives :(

Transform the Rays instead

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



World Space



Object Space

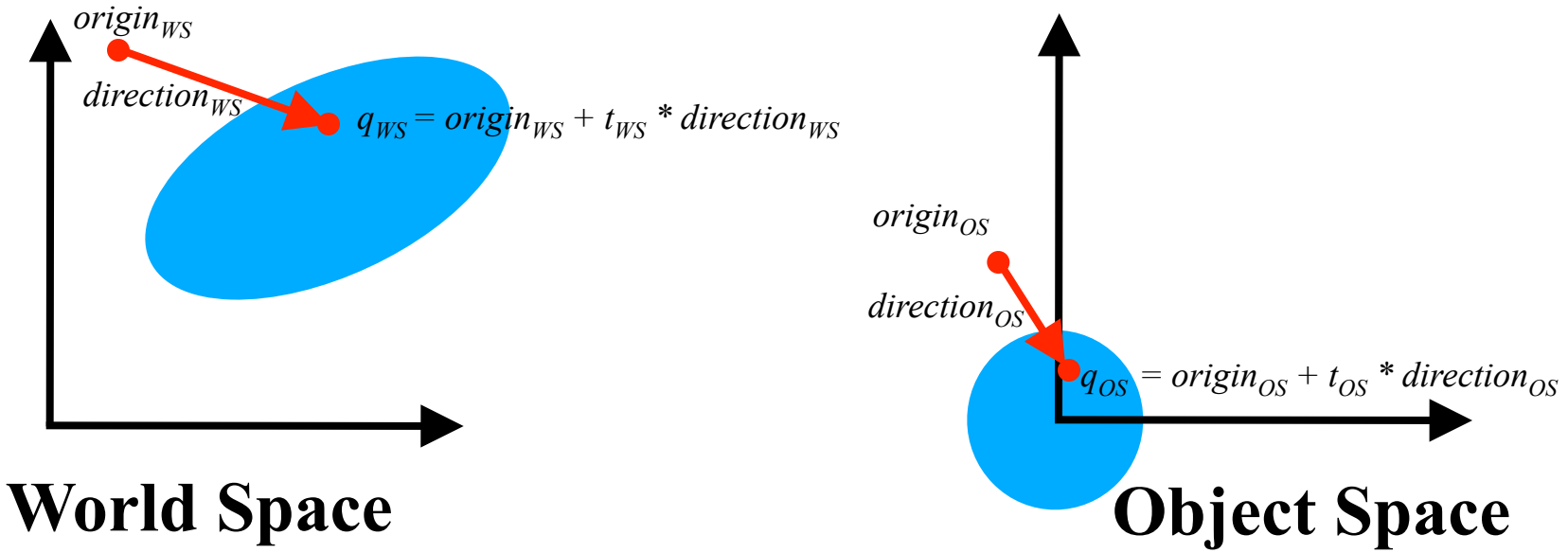
$$p_{WS} = \mathbf{M}\ p_{OS}$$

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

Transforming the Ray

- New origin:

- New direction:

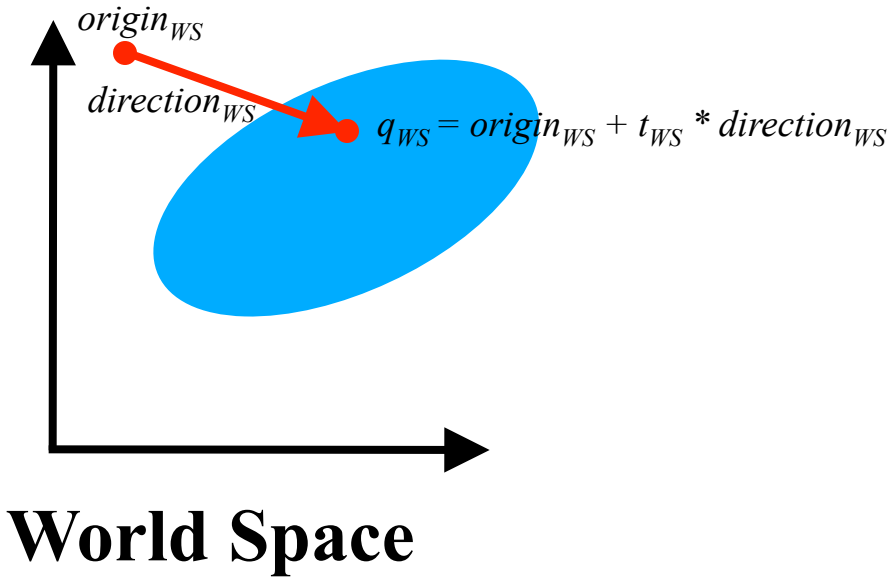


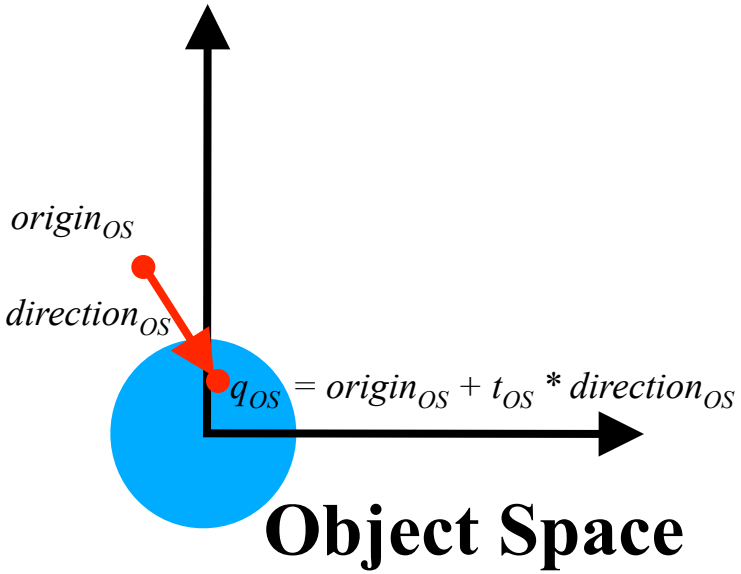
Transforming the Ray

- New origin:

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

- New direction:





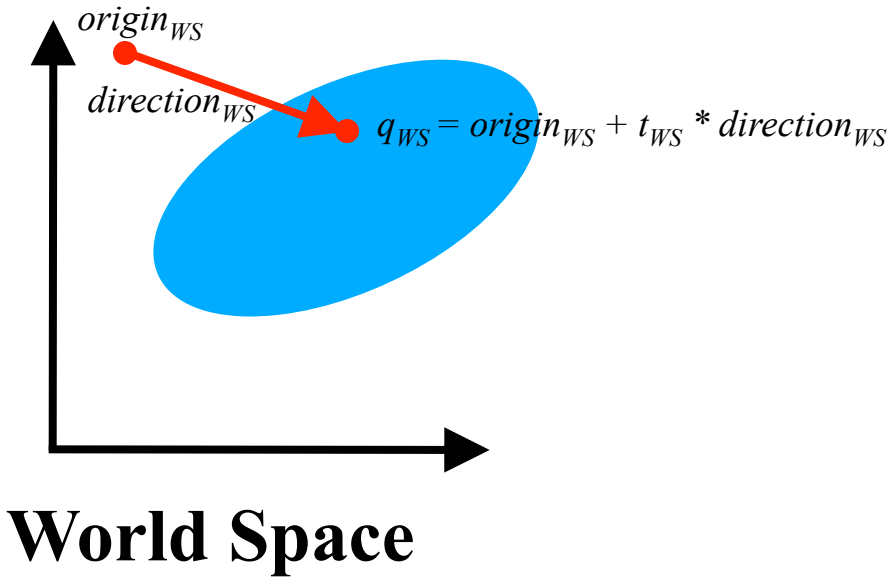
Transforming the Ray

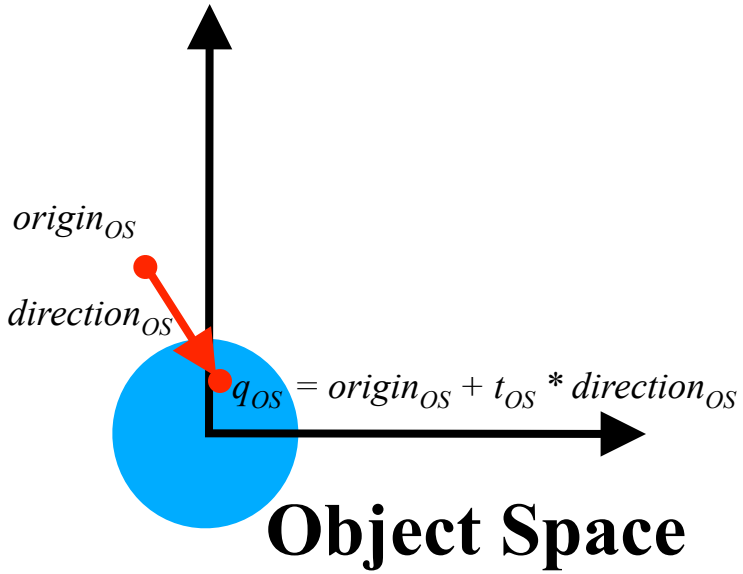
- New origin:

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

- New direction:

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





Transforming the Ray

- New origin:

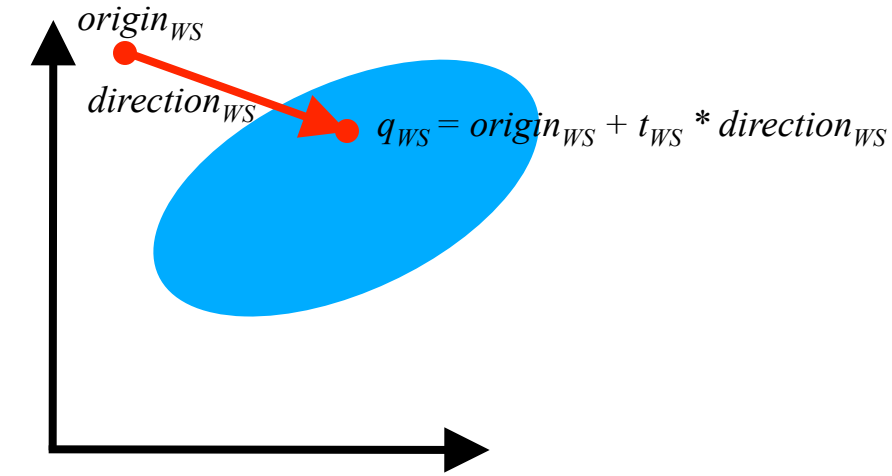
  $origin_{OS} = \mathbf{M}^{-1}\, origin_{WS}$

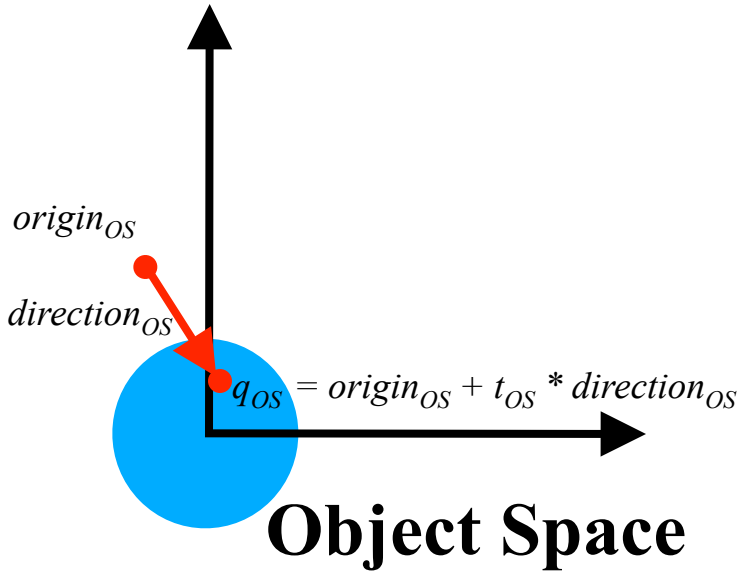
- New direction:

Note that the w component of direction is 0!

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

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



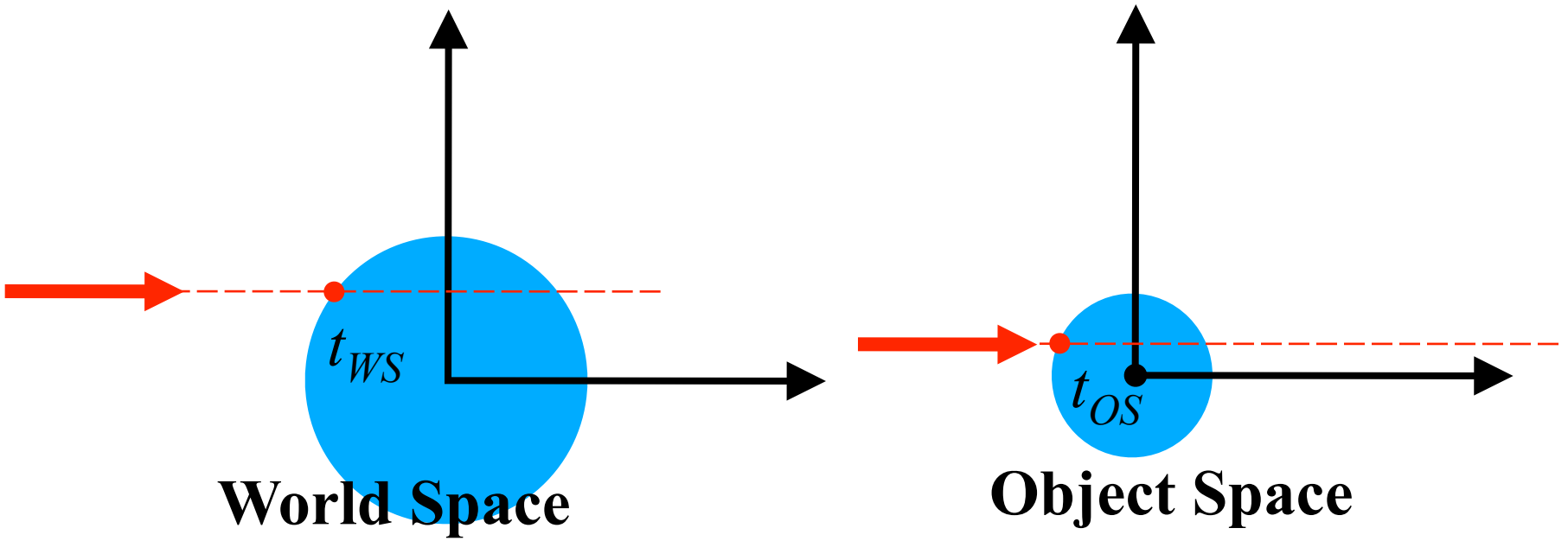


What about *t* ?

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

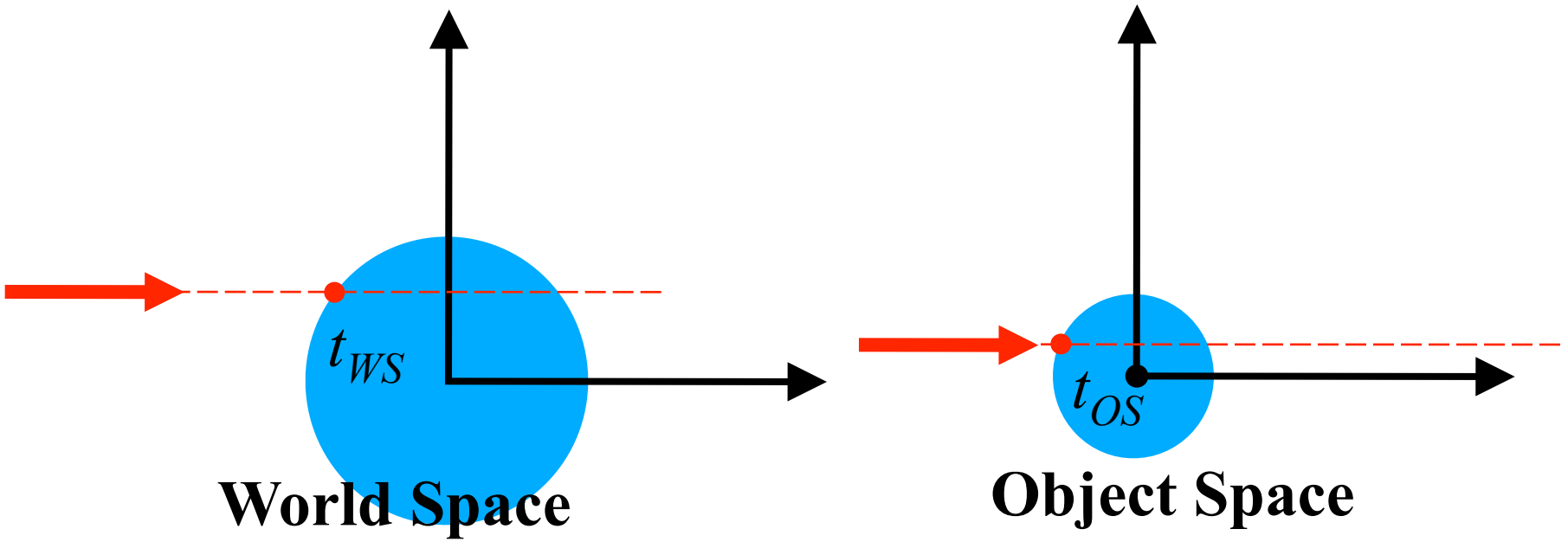
- Two solutions
  - Normalize the direction
  - Don't normalize the direction

1. Normalize direction?



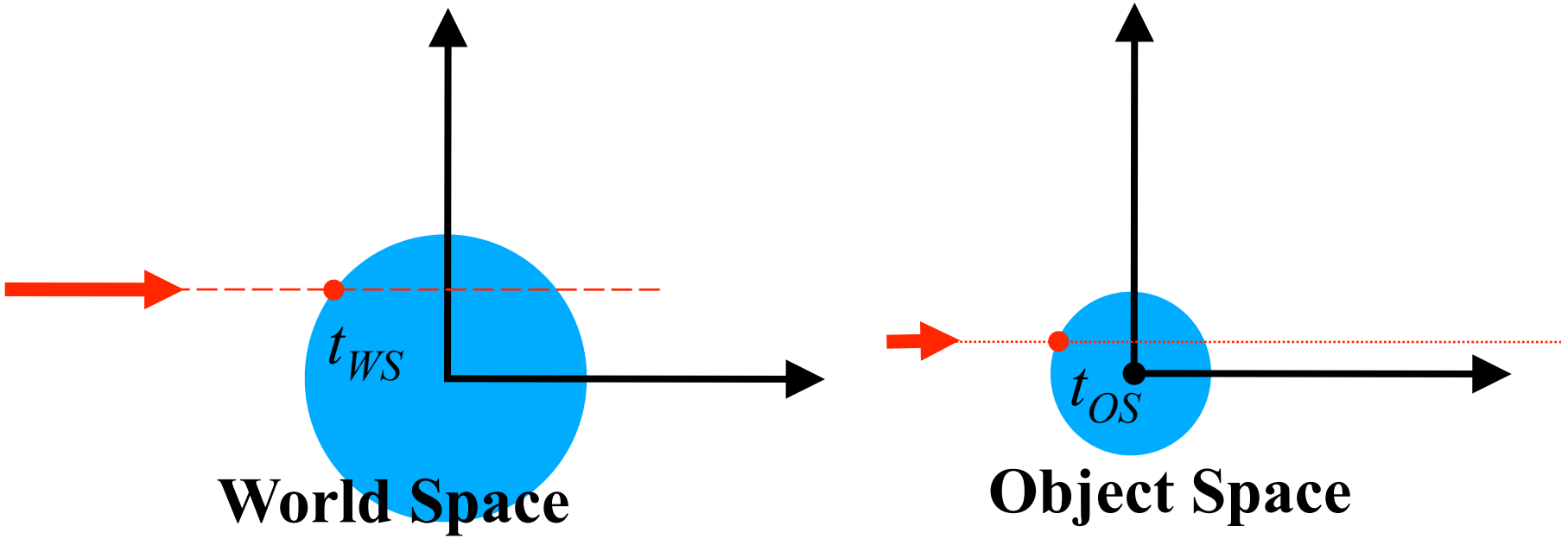
1. Normalize direction?

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



2. Don't normalize direction

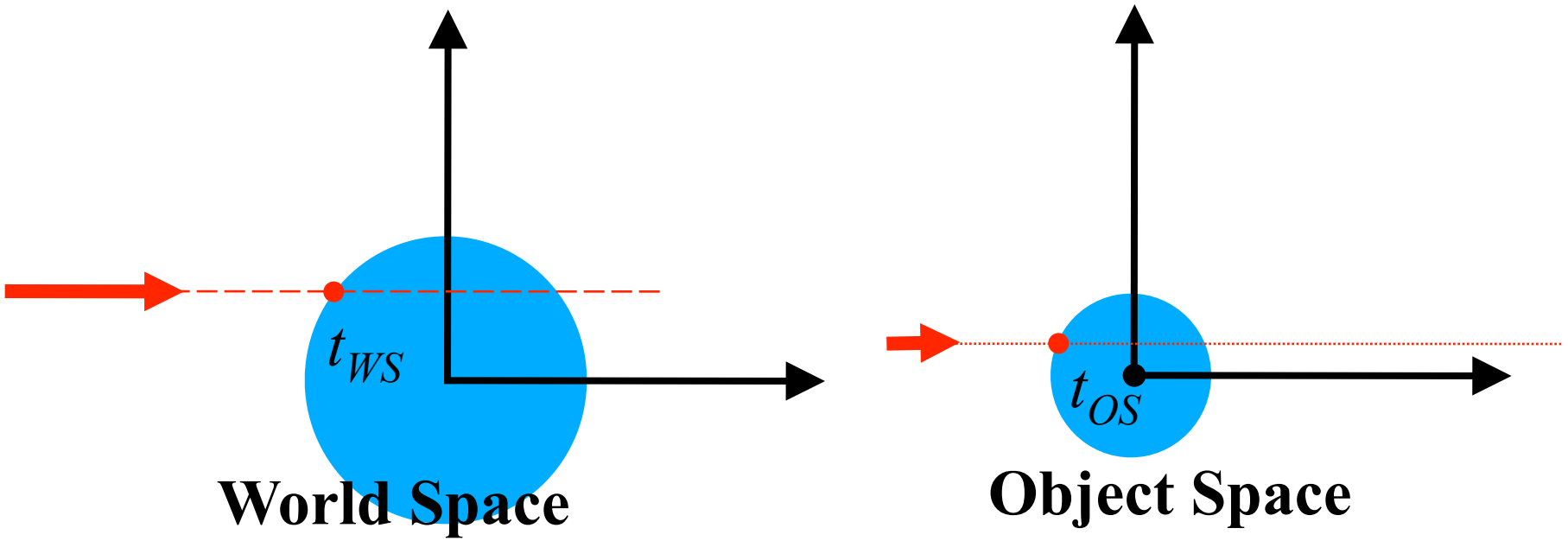
Highly recommended



2. Don't normalize direction

Highly recommended

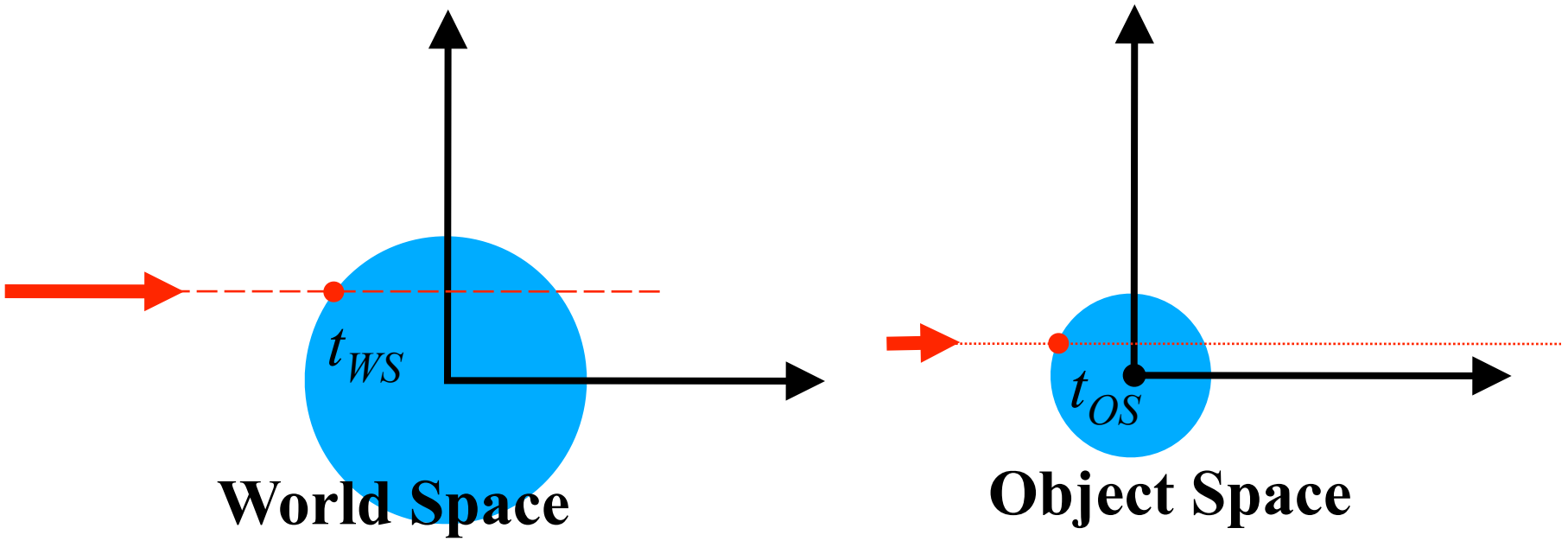
- $t_{OS} = t_{WS}$ → convenient!



2. Don't 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)



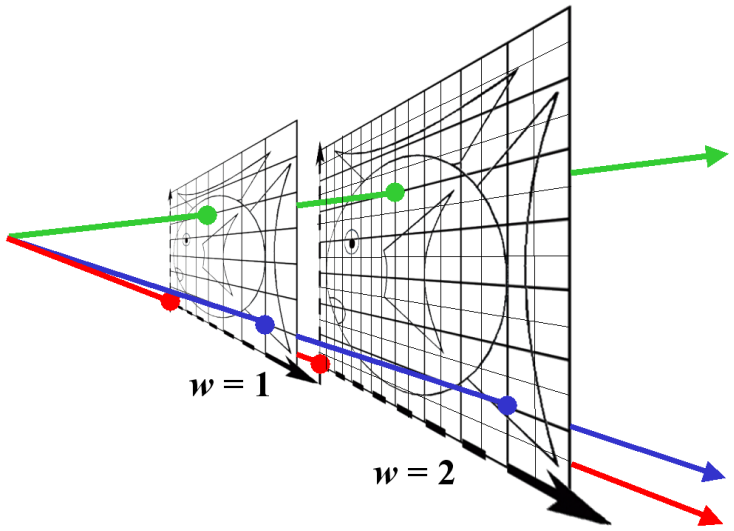
Transforming Points & Directions

- 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)

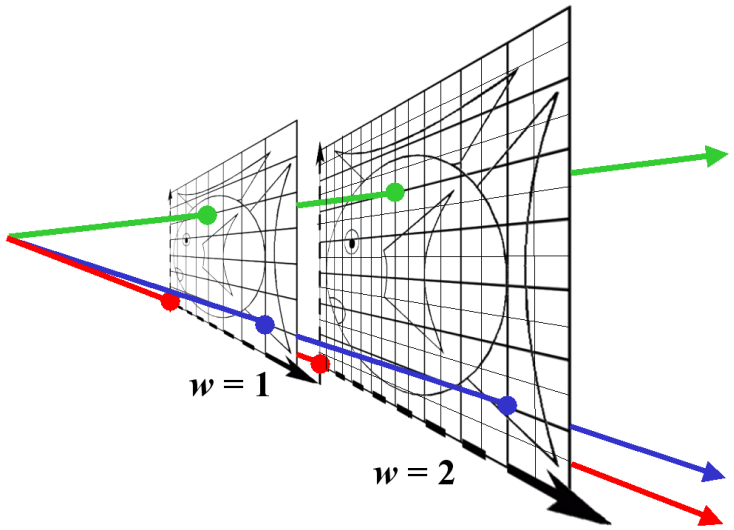
Transforming Points & Directions

- 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

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

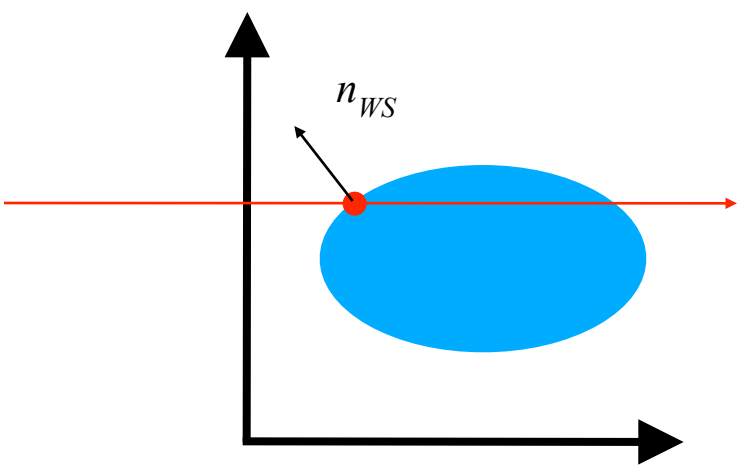


Homogeneous Coordinates: (x,y,z,w)

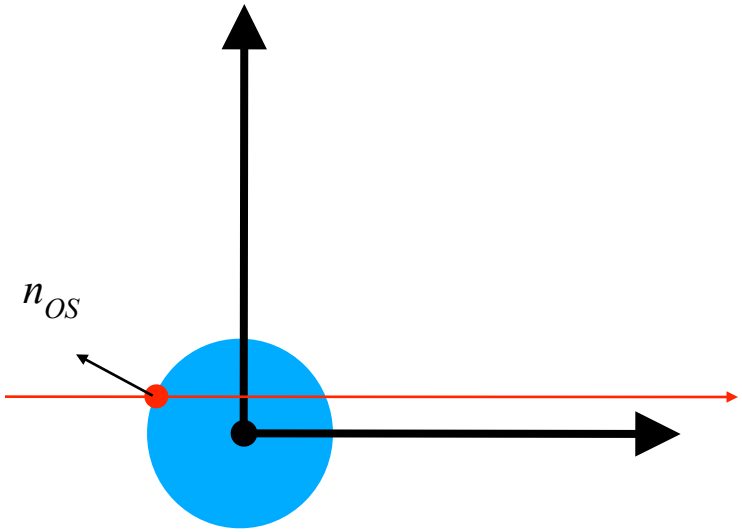
$w = 0$ is a point at infinity (direction)

• 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

Recap: Inverse Transpose for Normals

v is perpendicular to normal n:

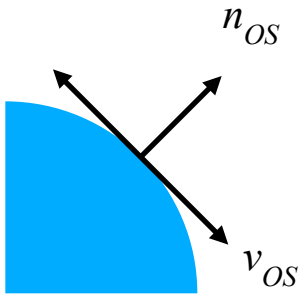
Dot product

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

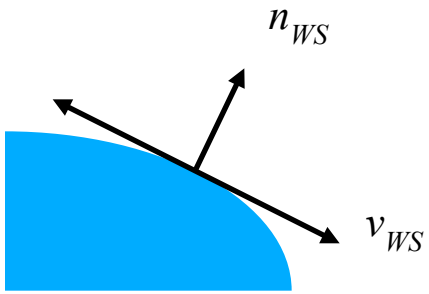
$$n_{OS}^{\mathrm{T}} \, (\mathbf{M}^{-1} \, \mathbf{M}) \, v_{OS} = 0$$

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

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



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



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

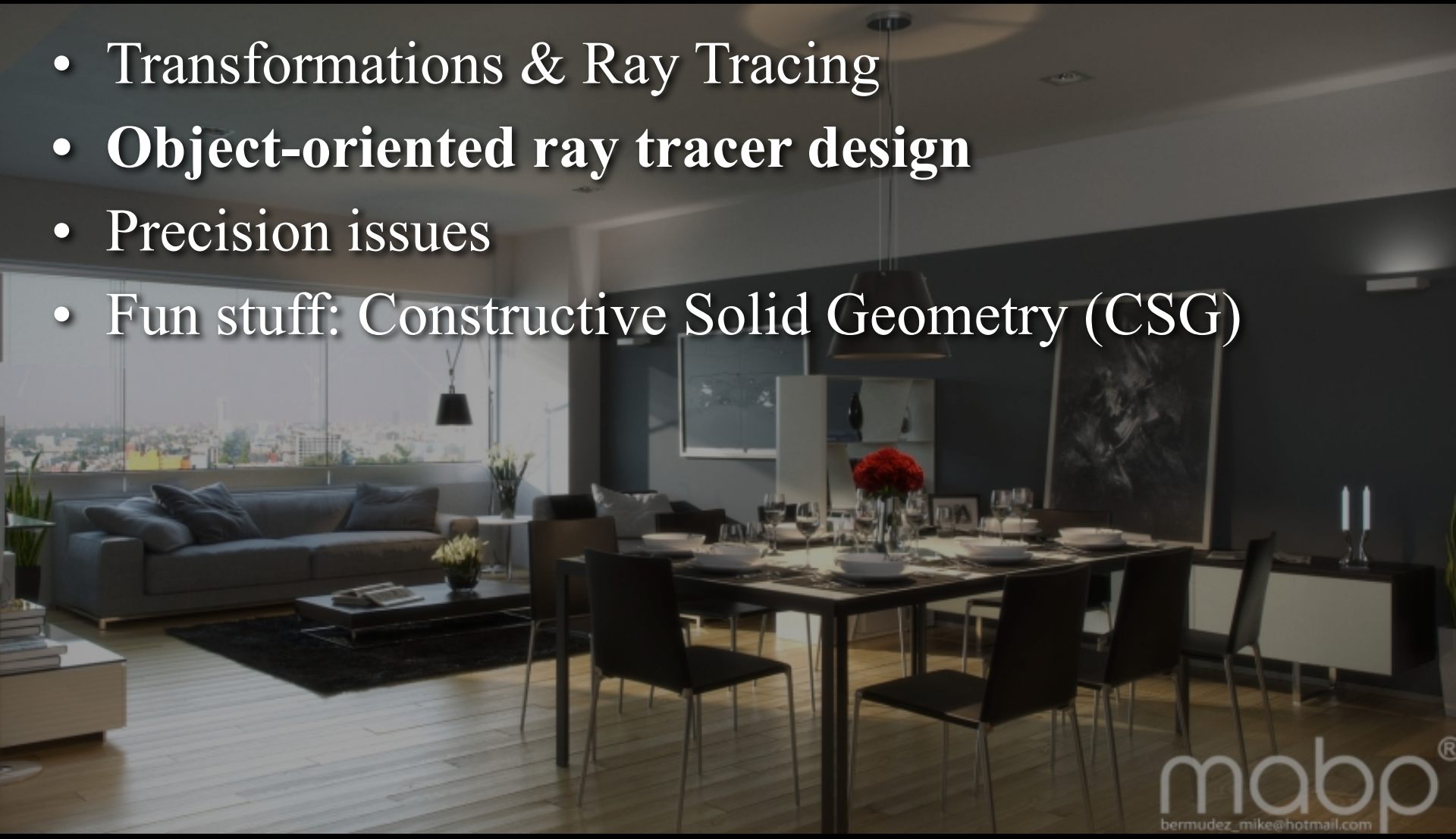
$$n_{WS} = (\mathbf{M}^{-1})^{\mathrm{T}} \, n_{OS}$$

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

Transformation Cheat Sheet

- Position
  - transformed by the full homogeneous matrix **M**
- Direction
  - transformed by **M** except the translation component (equiv. to w component of direction == 0)
- Normal
  - transformed by $\mathbf{M}^{-T}$, no translation component

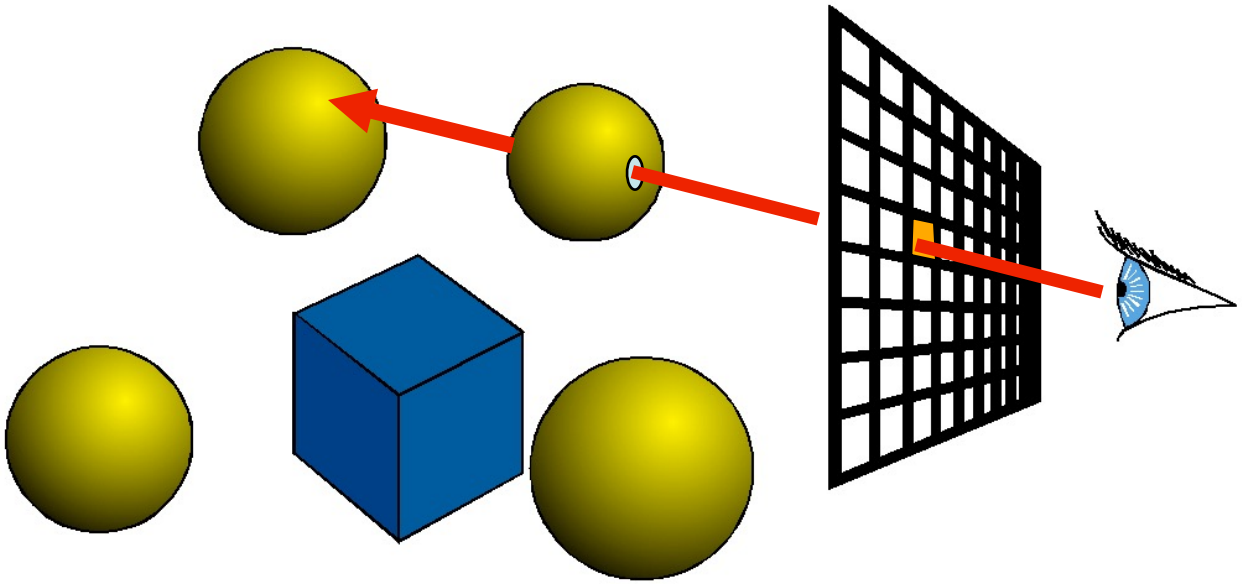
In this Video



- Transformations & Ray Tracing
- **Object-oriented ray tracer design**
- Precision issues
- Fun stuff: Constructive Solid Geometry (CSG)

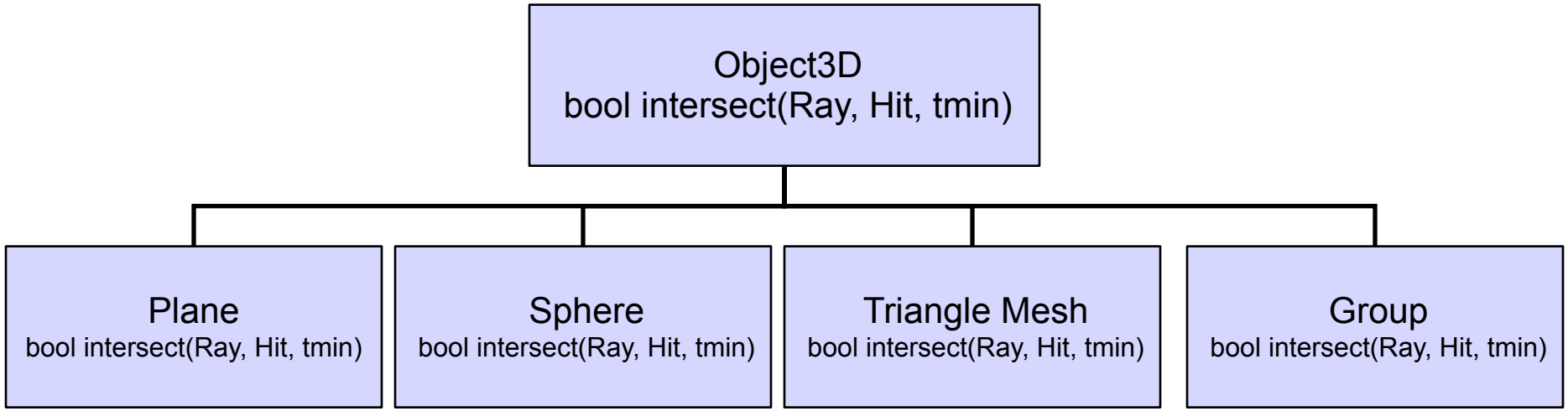
Ray Tracing: Object oriented design

```
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!



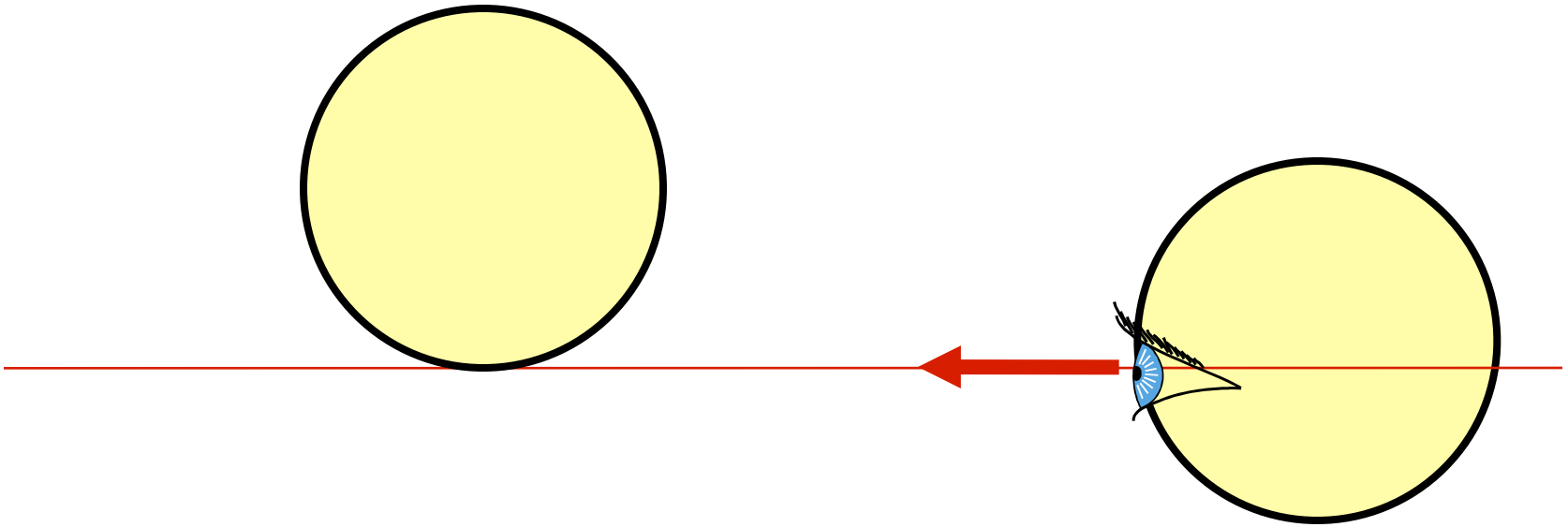
- Also cameras are abstracted (perspective/ortho)
  - Methods for generating rays for given image coordinates

In this Video

- Transformations & Ray Tracing
- Object-oriented ray tracer design
- **Precision issues**
- Fun stuff: Constructive Solid Geometry (CSG)

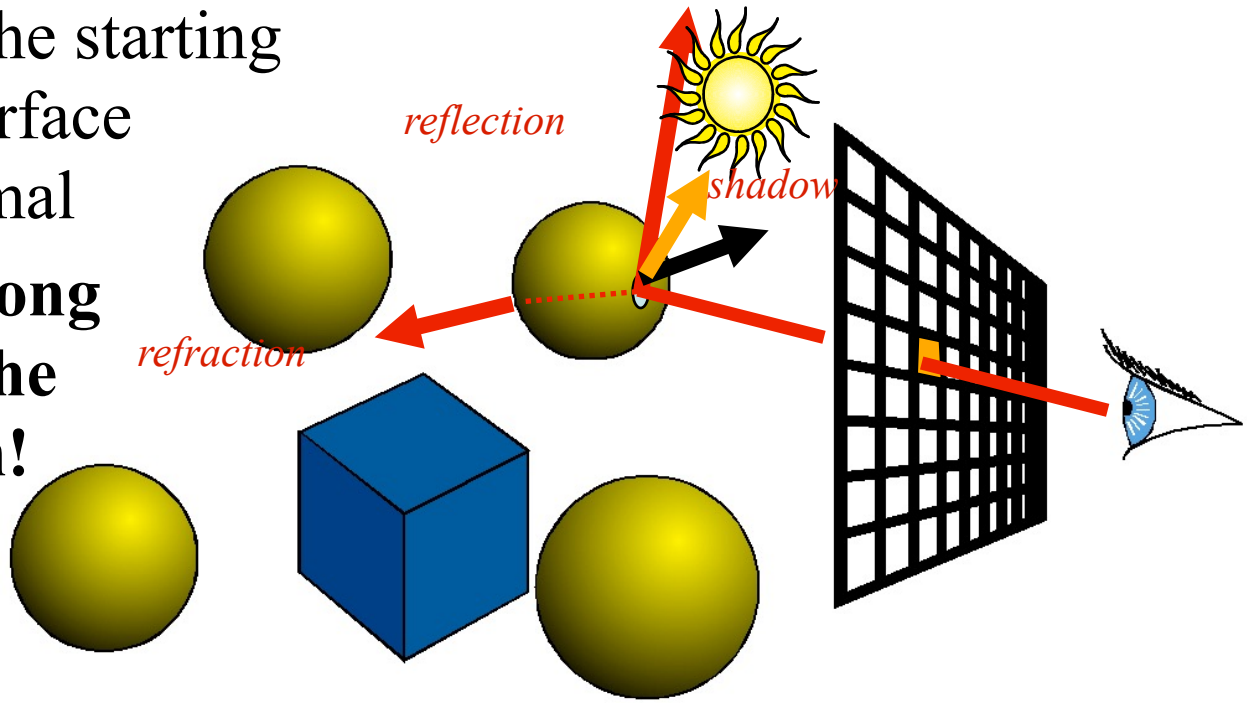
Precision

- 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 at the surface
  - Secondary rays will start at the surfaces
  - Requires epsilons
  - Best to nudge the starting point off the surface e.g., along normal
  - **Best: nudge along the direction the ray came from!**



The evil ε

- Edges in triangle meshes
  - Must report intersection (otherwise not watertight)
  - Hard to get right
  - See Woop, Wenthin, Wald, JCGT 2013

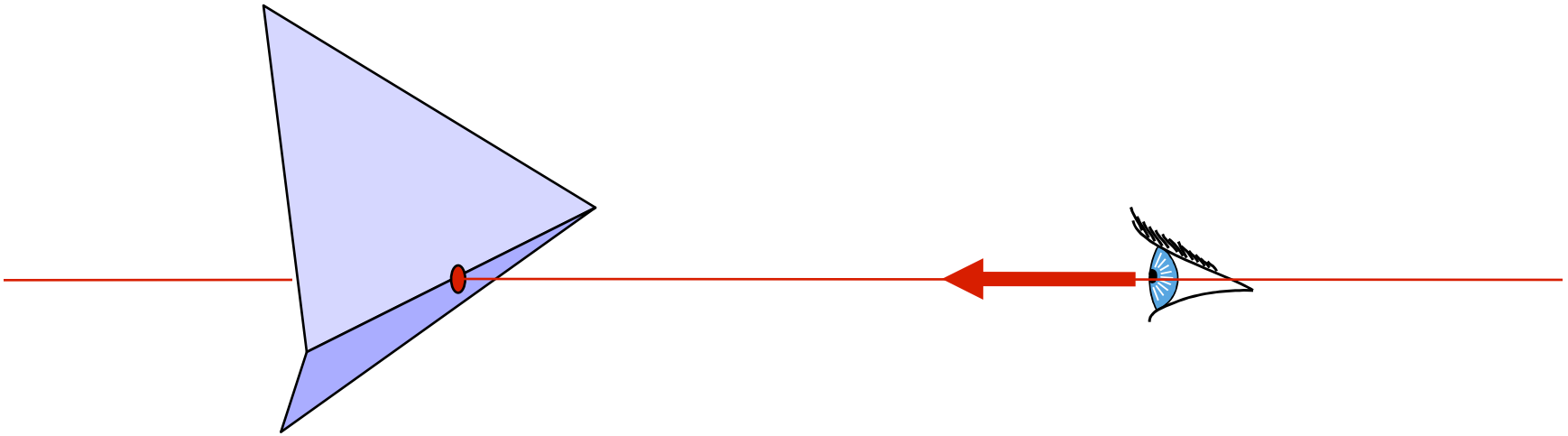
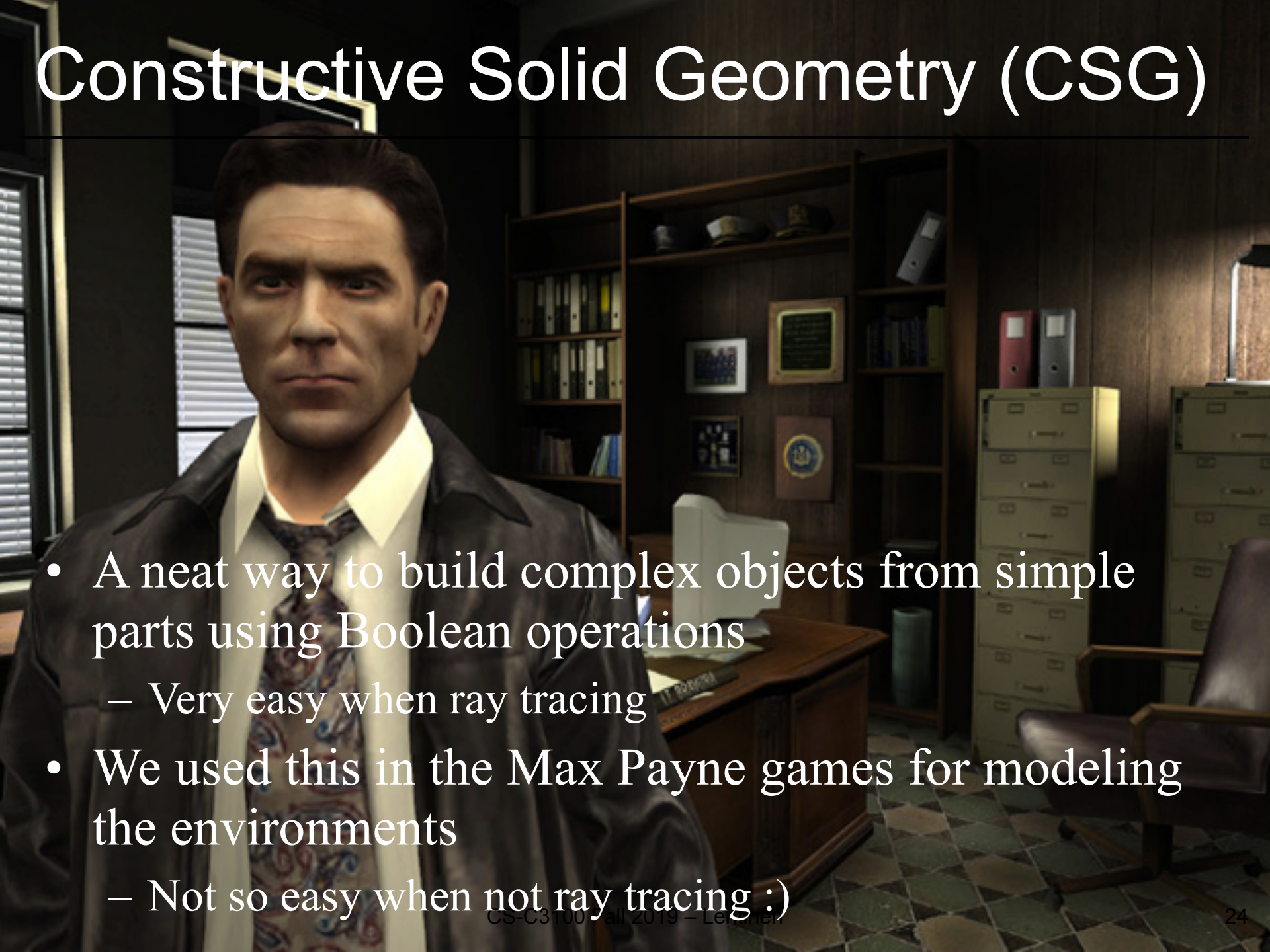


Image by Henrik Wann Jensen

In this Video

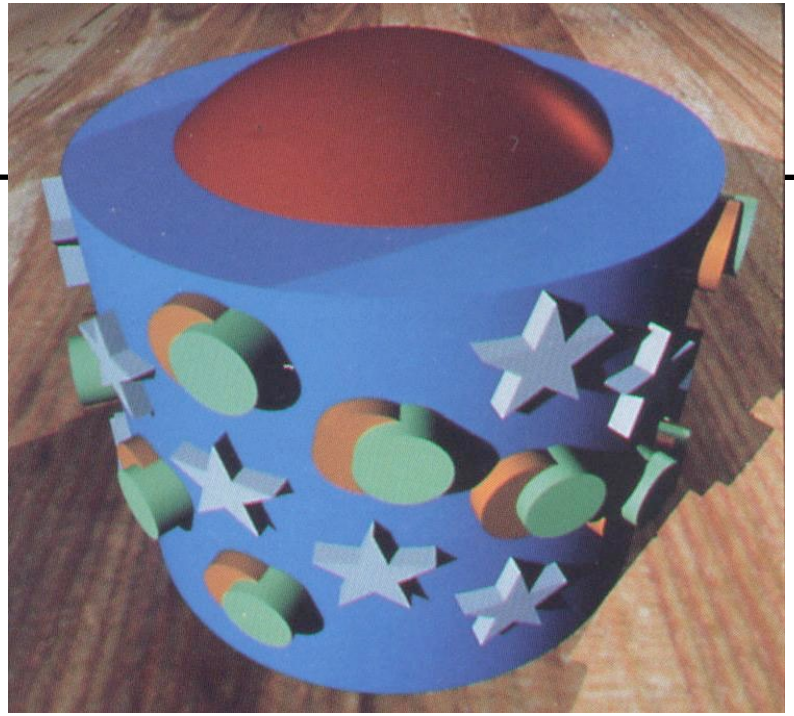
- Transformations & Ray Tracing
- Object-oriented ray tracer design
- Precision issues
- **Fun stuff: Constructive Solid Geometry (CSG)**

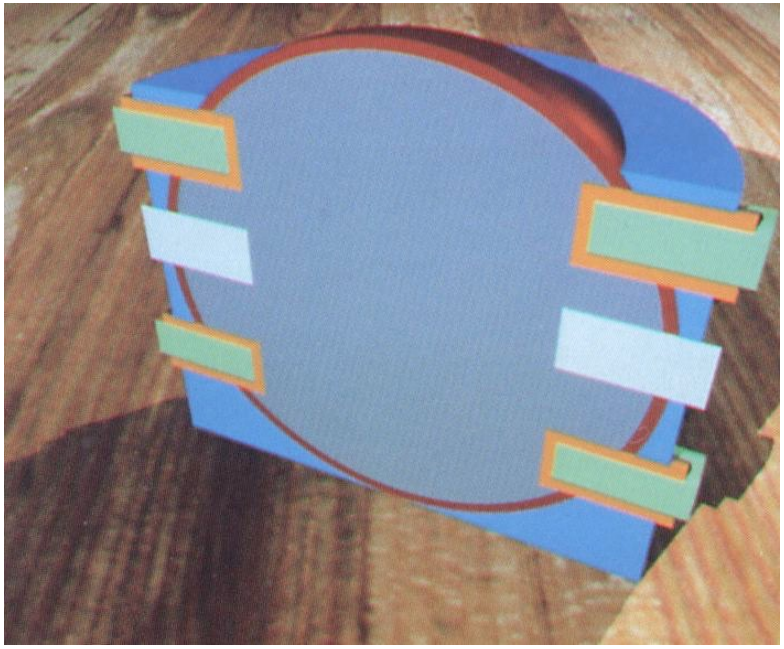
Constructive Solid Geometry (CSG)



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

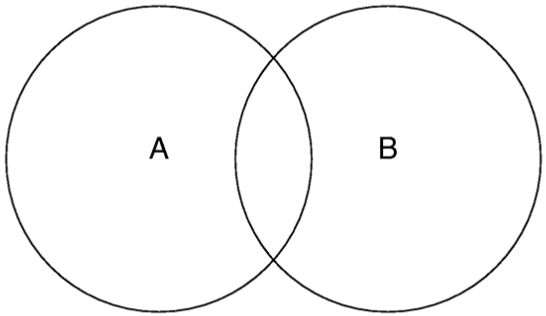
CSG Examples



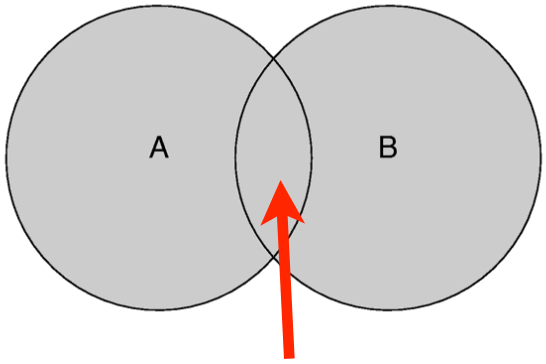


Constructive Solid Geometry (CSG)

Given overlapping shapes A and B:

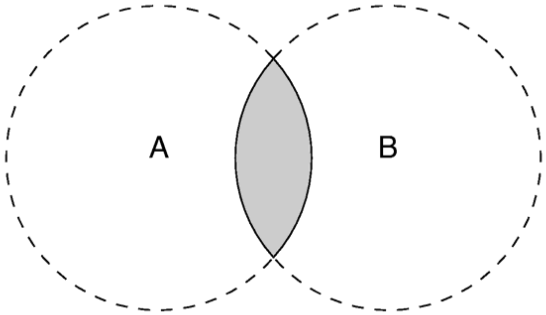


Union

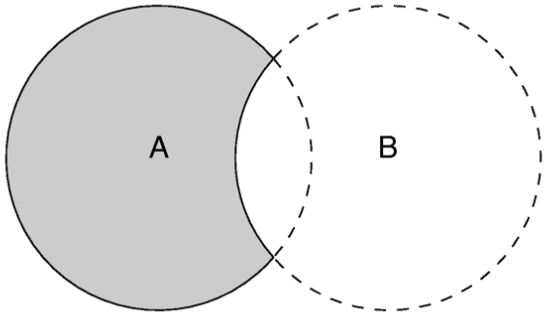


Should only "count" overlap region once!

Intersection

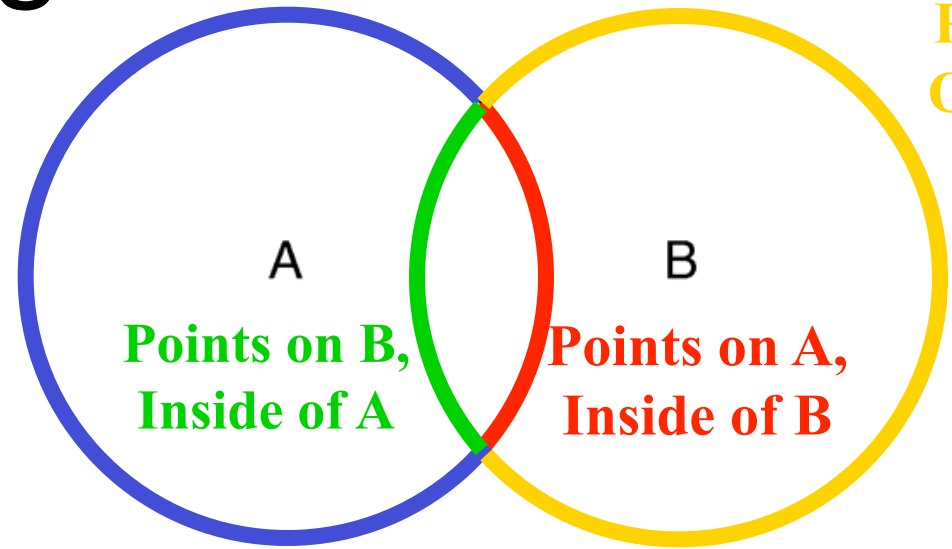


Subtraction

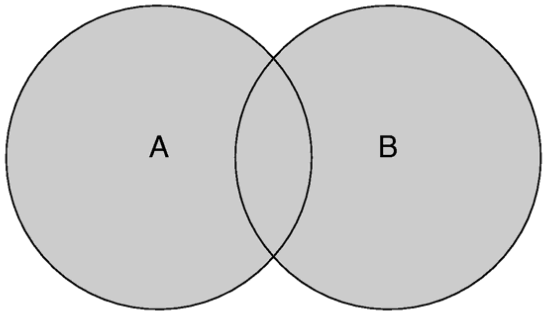


How can we implement CSG?

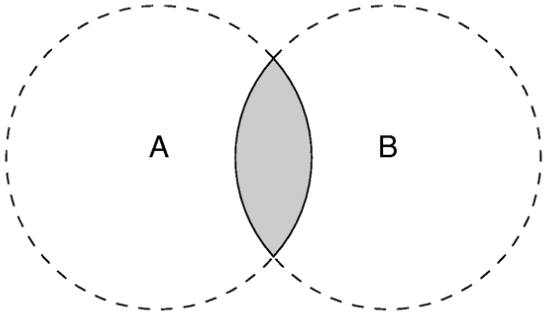
4 cases



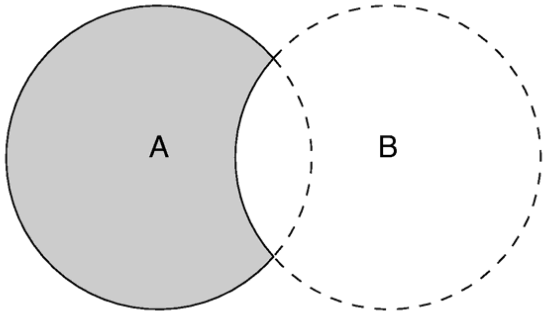
Union



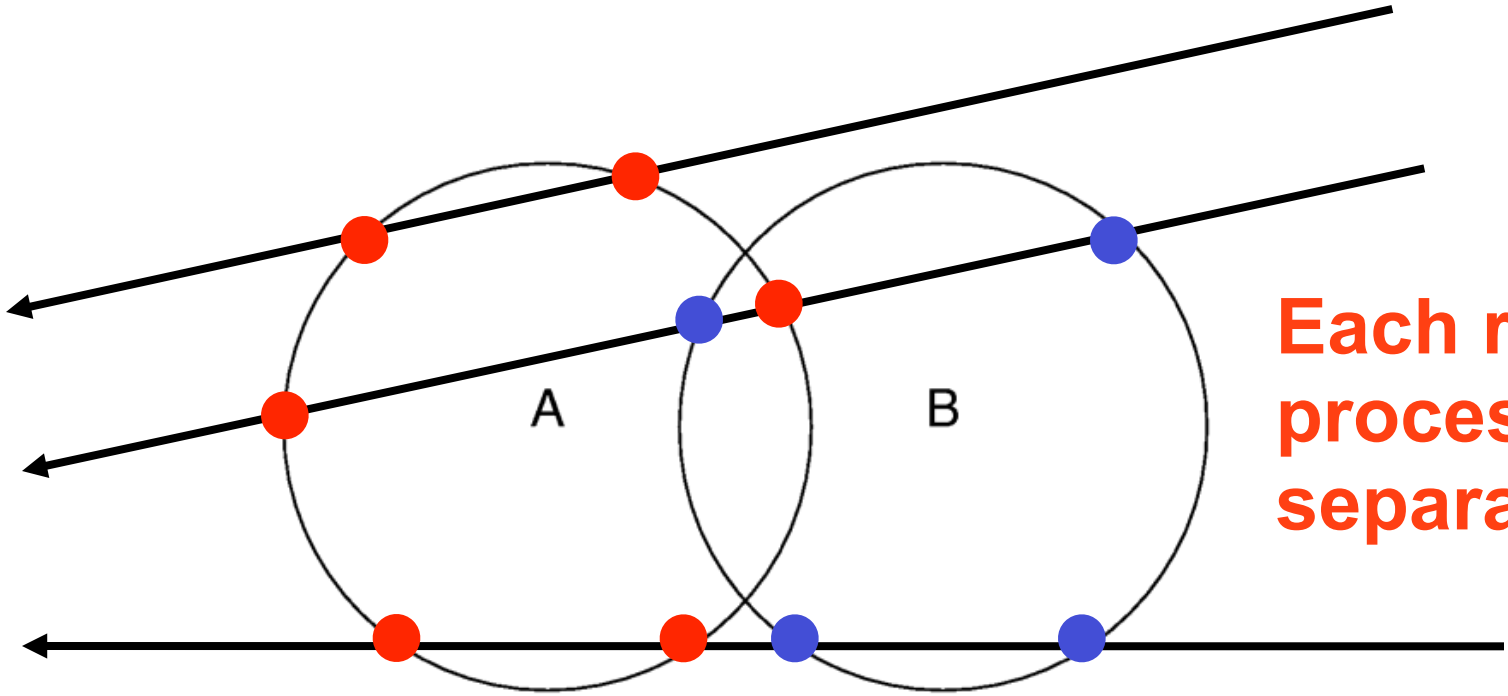
Intersection



Subtraction

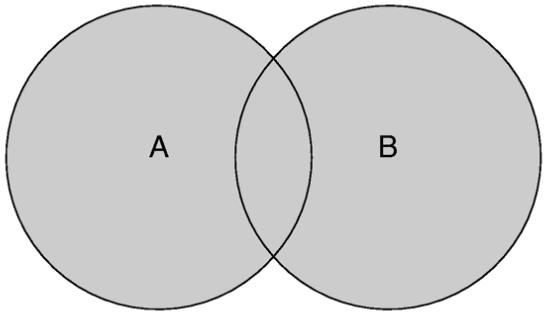


Collect Intersections

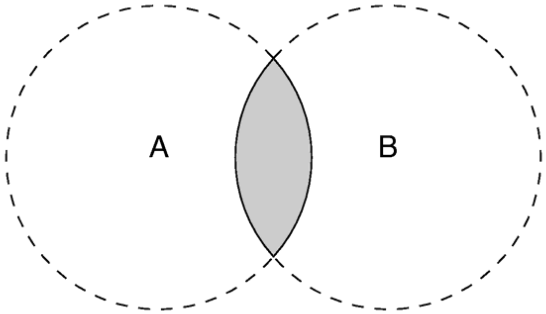


Each ray processed separately!

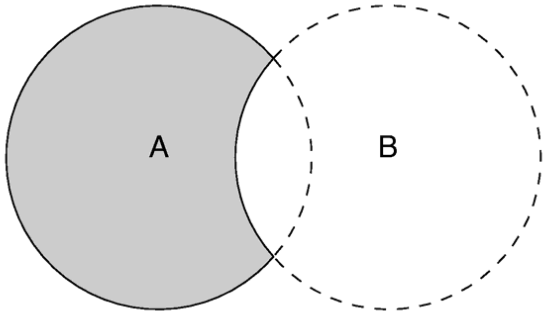
Union



Intersection

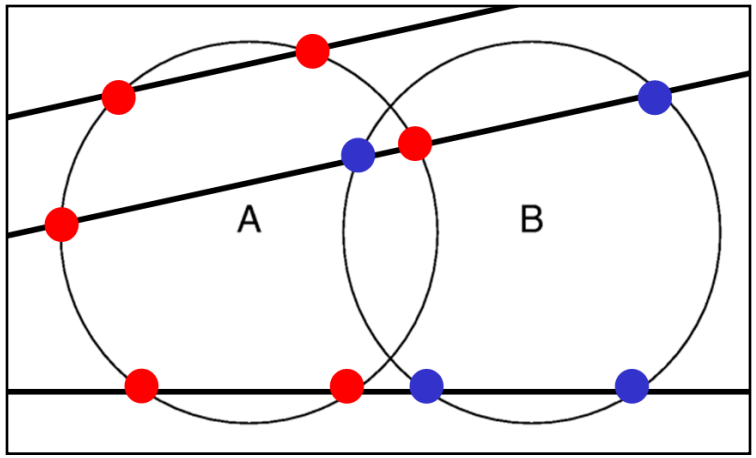


Subtraction



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

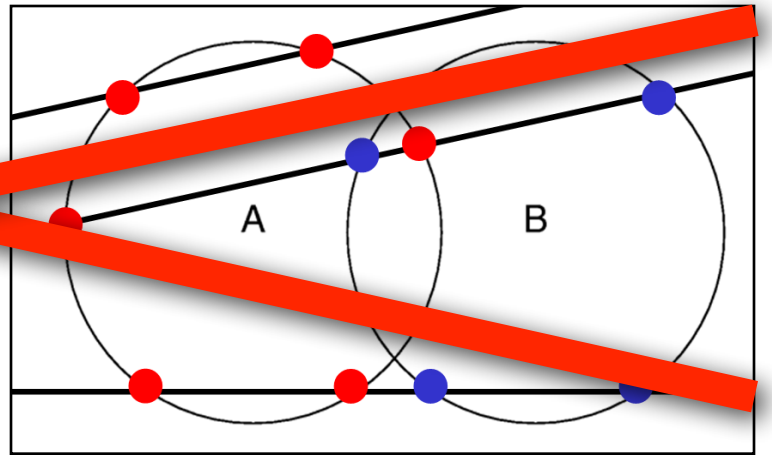


This would certainly work, but would need to determine if points are inside solids...

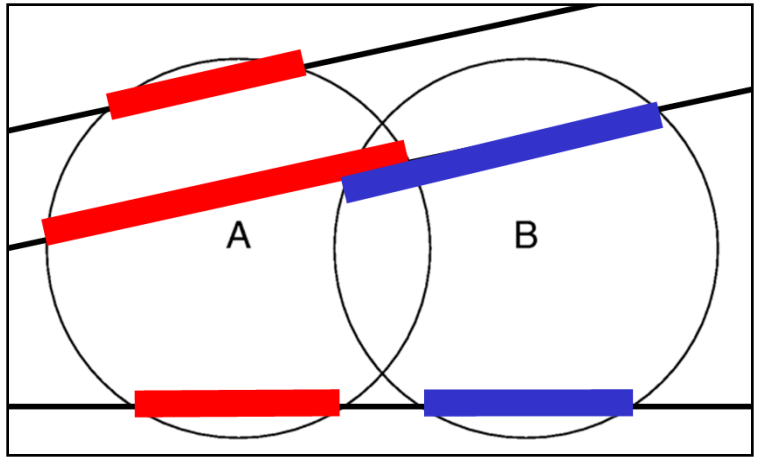
:-(

Implementing CSG

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



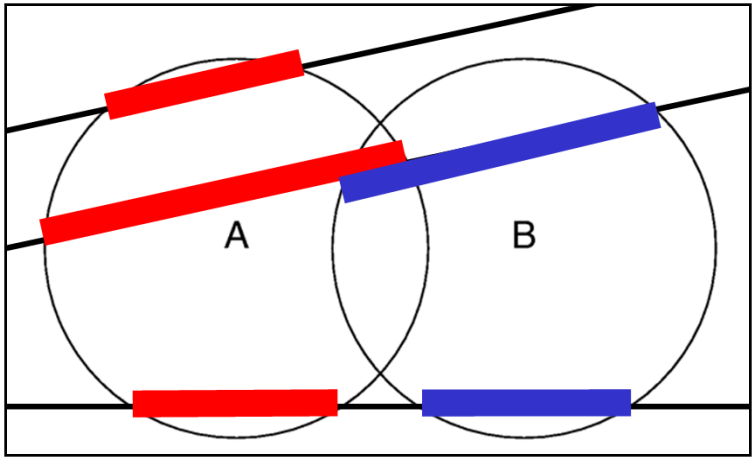
2. Overlapping intervals:
   - Find the intervals of "inside" along the ray for A and B
   - How? Just keep an “entry” / “exit” bit for each intersection
     - Easy to determine from intersection normal and ray direction
   - Compute union/intersection/ subtraction of the intervals



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



Implementing CSG

Problem reduc

Highly recommended, easy extra credit for Assn'5!

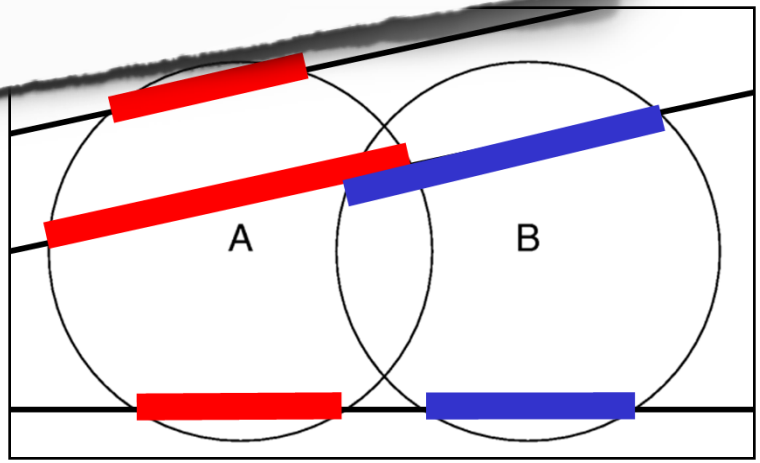
2.

"exit" bit

each intersection

- Easy to determine from intersection normal and ray direction

- Compute union/intersection/ subtraction of the intervals



CSG is Easy with Ray Tracing…

- ...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.

CSG is Easy with Ray Tracing…

- ...but **very hard** if you actual… compute an explicit representation… urface as a triangle mesh
- In princi…
  but flo…
  - E.g., …
  - Compu… rily the same as …
  - The line t… two planes does n… necessarily …
  - etc., etc.

Highly recommended, very hard extra credit for Assn'5! :)