Topic 11: More ray tracing

Some slides and figures courtesy of Wolfgang Hurst, Karan Singh
Some figures from Peter Shirley, “Fundamentals of Computer Graphics”, 3rd Ed.

Ideal specular or mirror reflection

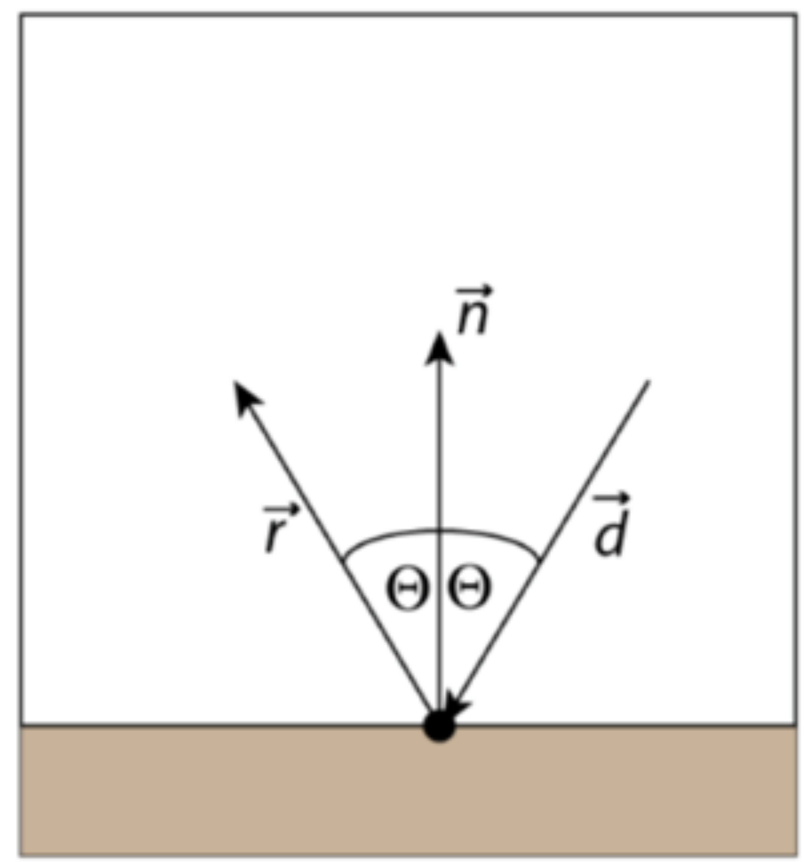
Key characteristic of a mirror:

If looking from direction $\vec{d}$ to a spot on the reflecting surface, the viewer sees the same image as if looking from the surface point in direction $\vec{r}$.

Hence, we need to calculate the reflection vector:

$$\vec{r} = \vec{d} - 2(\vec{d} \cdot \vec{n})\vec{n}$$

Then we shoot a ray in that direction.



Ideal specular or mirror reflection

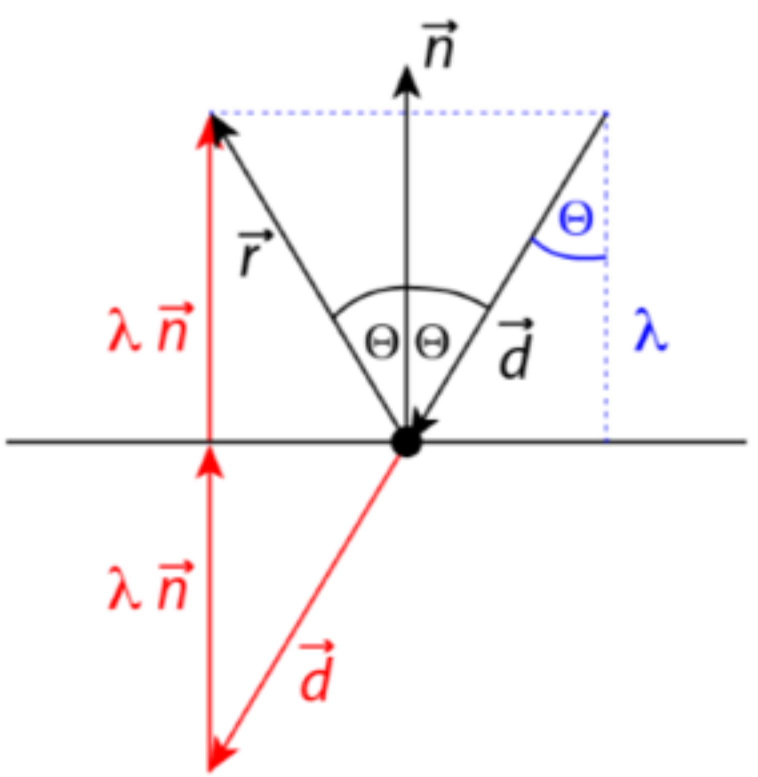
We know this from Phong reflection with light vector l:

$$\vec{r} = -\vec{l} + 2(\vec{l} \cdot \vec{n})\vec{n}$$

But be careful with the direction of $\vec{d}$ when calculating the reflection vector for mirroring:

$$\vec{r} = \vec{d} - 2(\vec{d} \cdot \vec{n})\vec{n}$$

Also: we need to include a max. recursion depth to avoid "infinite bouncing" of rays.

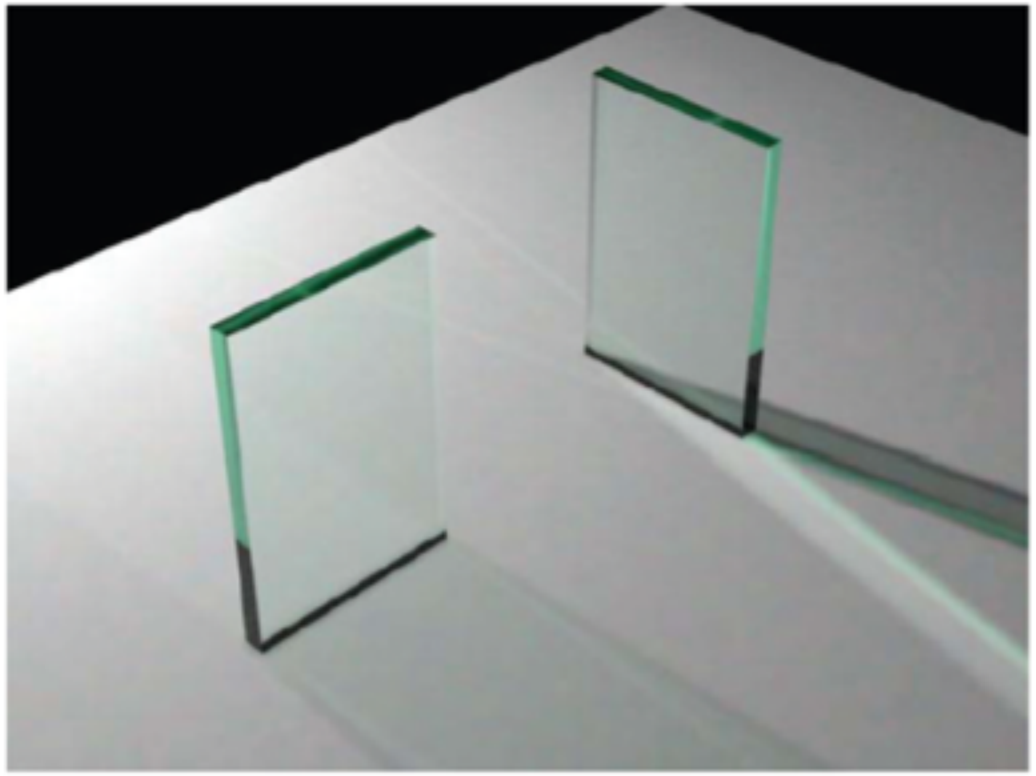


$$\lambda = \cos\theta \|\vec{n}\| \|\vec{d}\| = -\vec{d} \cdot \vec{n}$$

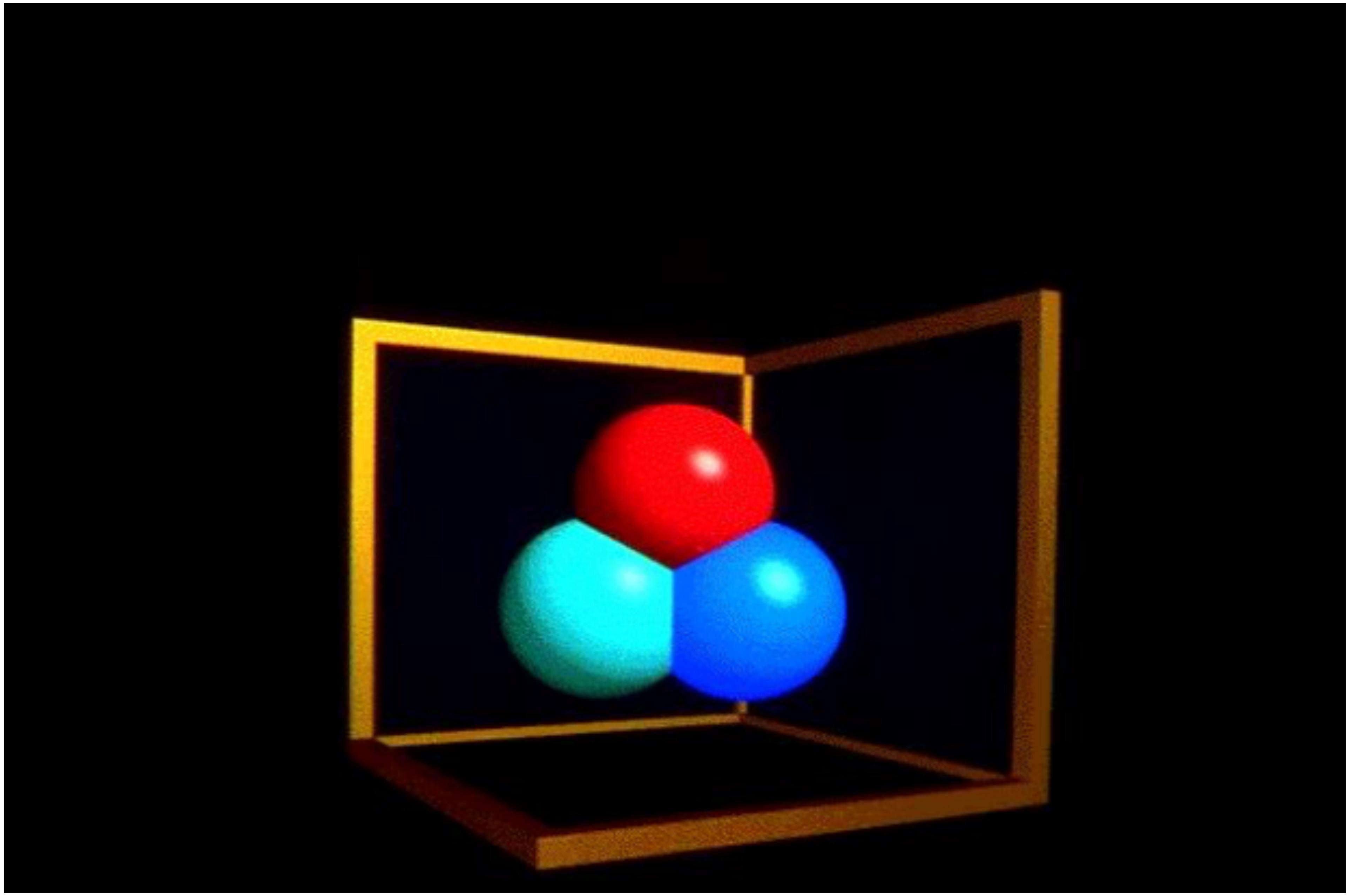
Refraction

Light traveling from one transparent medium into another one is refracted.

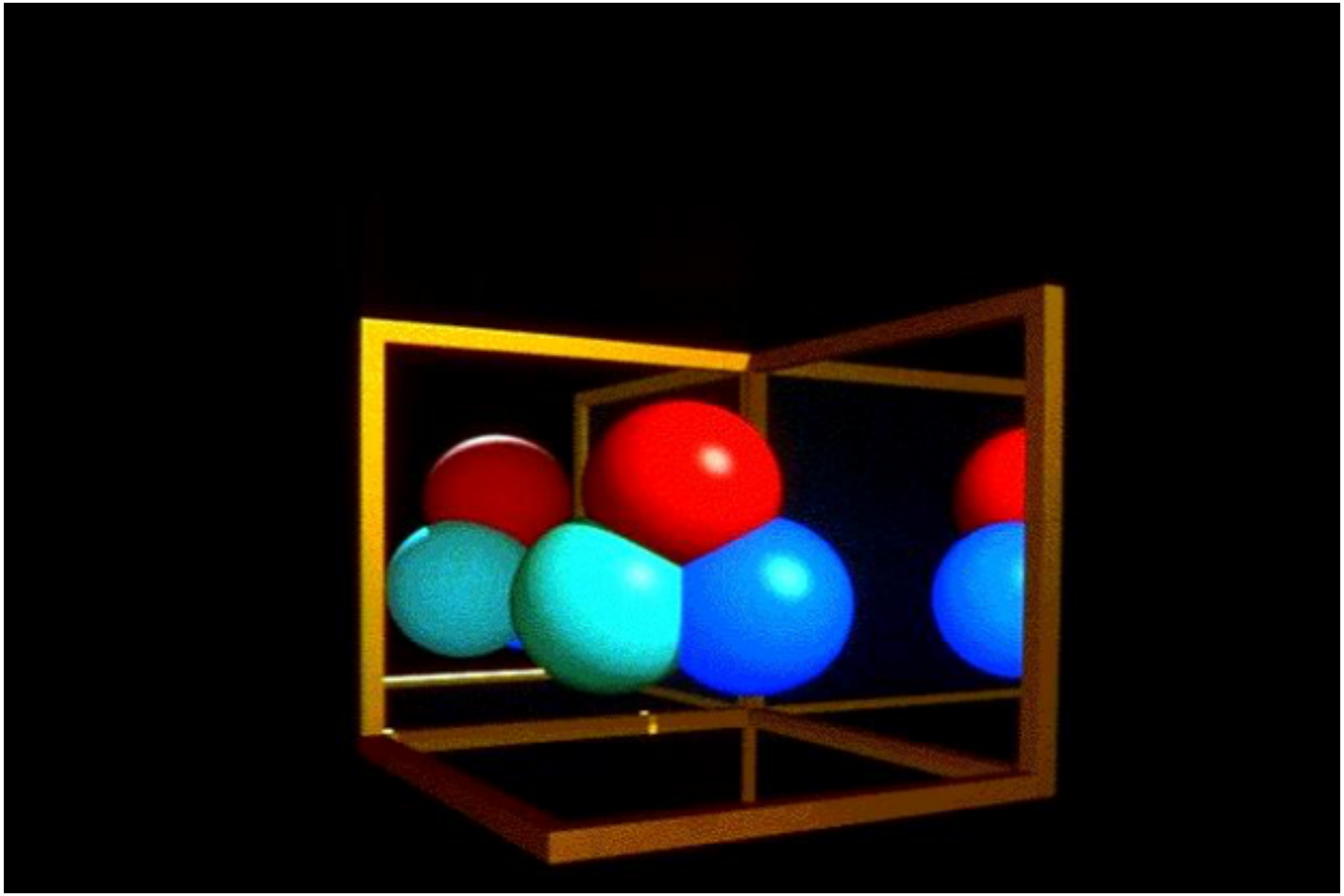
To consider this in ray tracing, we need to know the refraction angles.



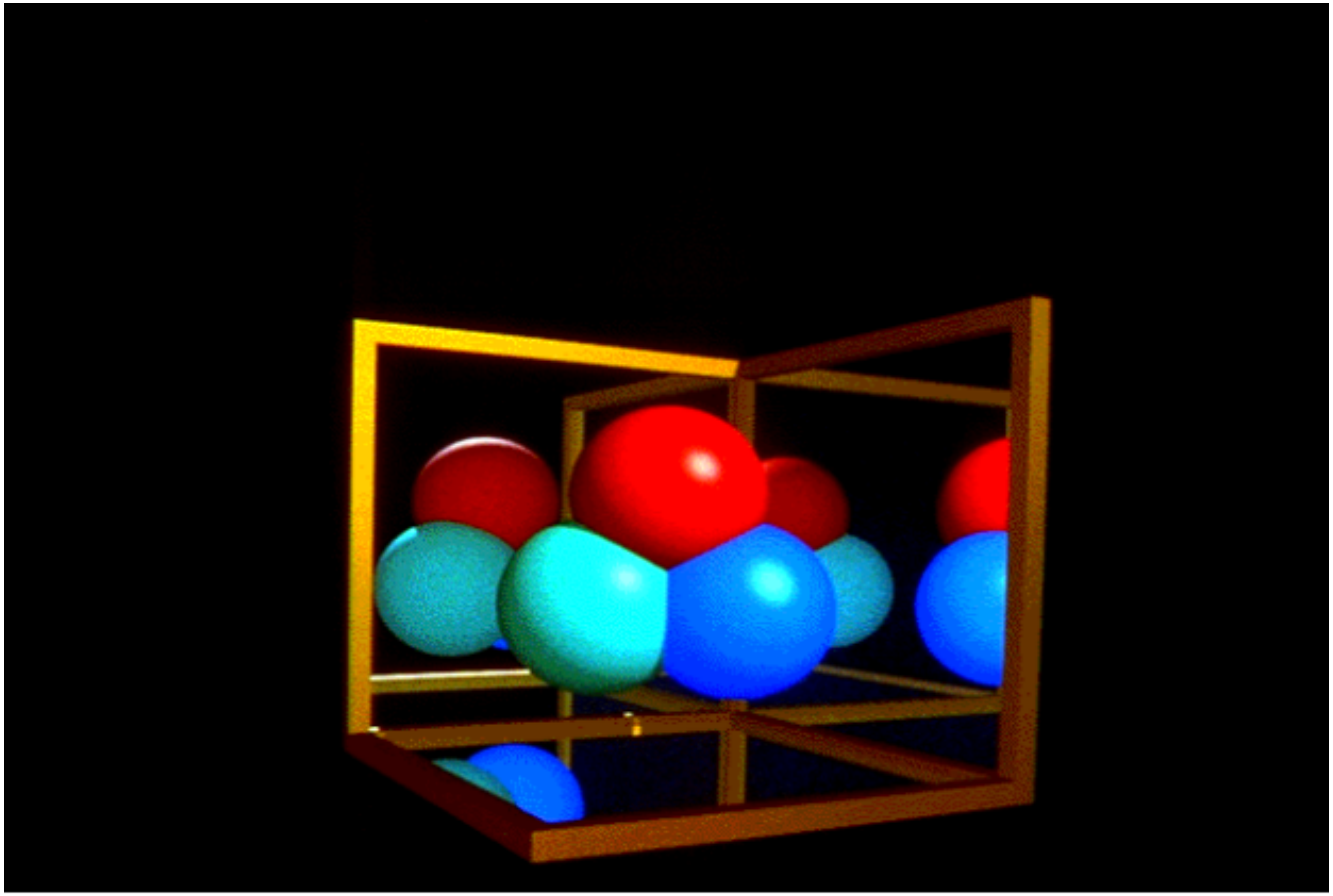
No reflection



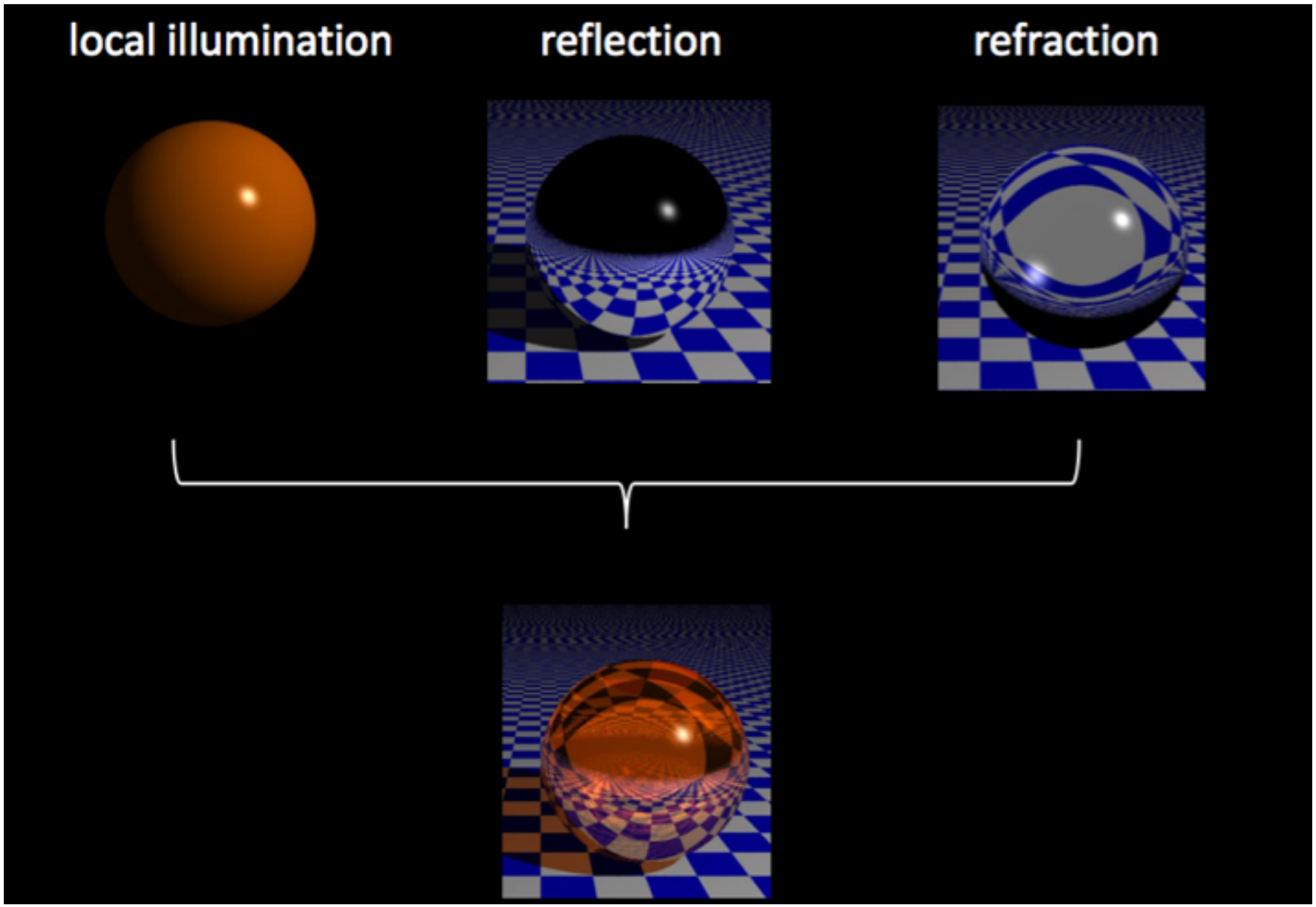
Single reflection



Double reflection



Components

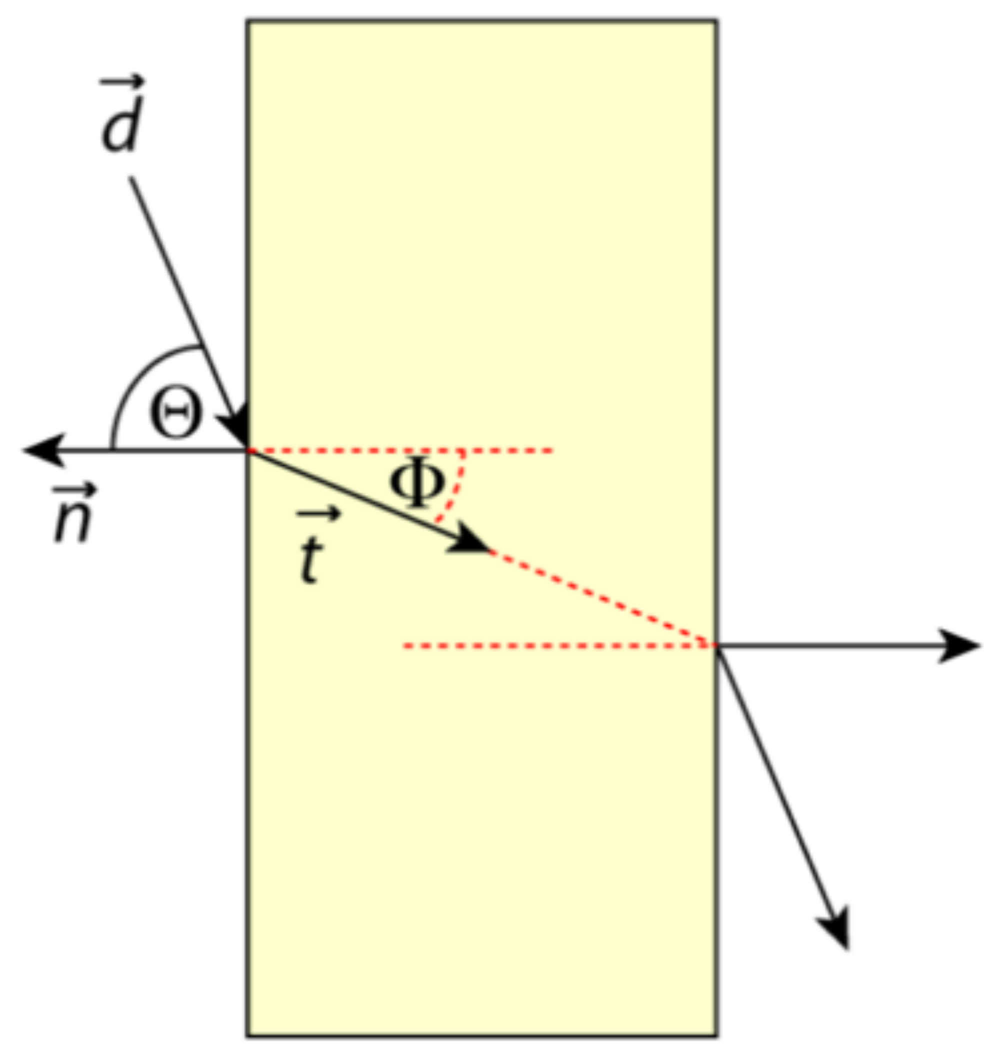


Snell's law

According to Snell's law, the angles before and after refraction are related as follows:

$$n \sin \theta = n_t \sin \phi$$

where n and n_t are the refractive indices of the source and target media, respectively, and θ and ϕ the angles indicated in the image.

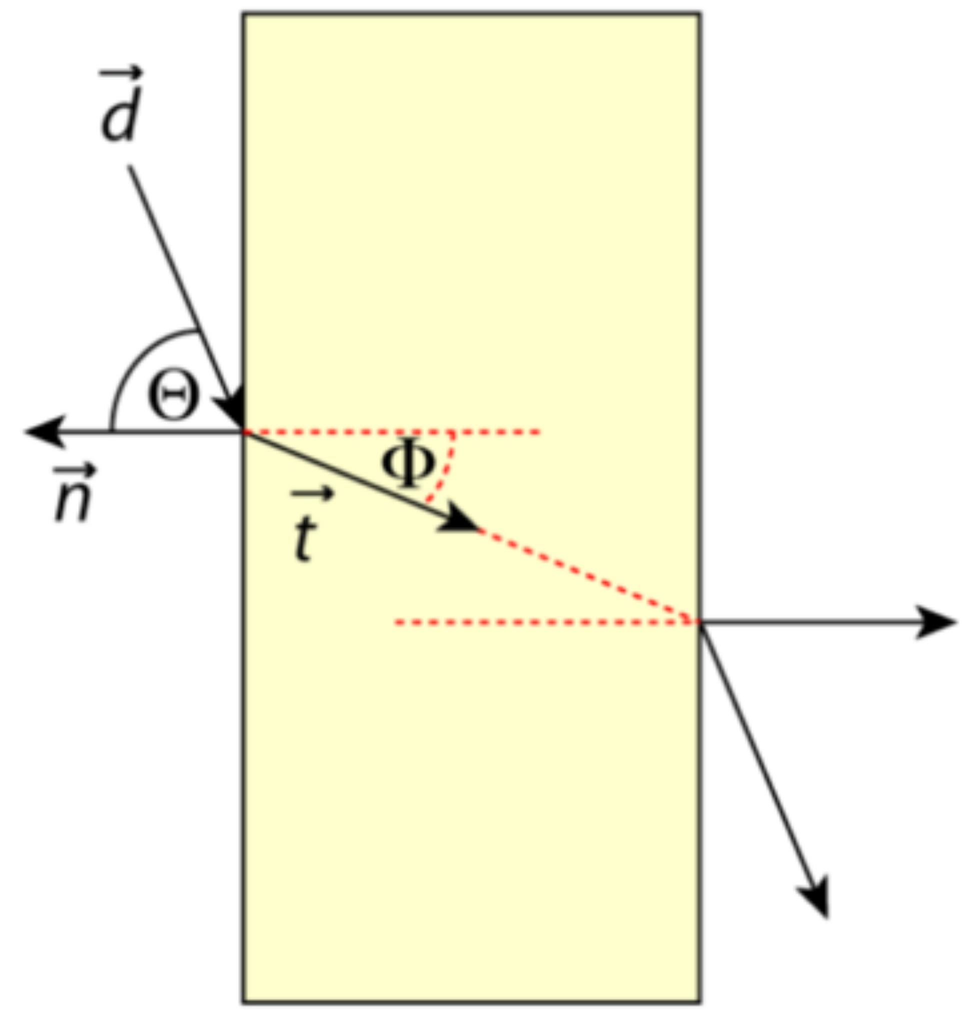


Getting rid of sines

An equation that relates sines of the angles θ and ϕ is not as convenient as an equation that relates the cosines of the angles.

Fortunately, we can eliminate the sines using the trigonometric identity:

$$\sin^2 \alpha + \cos^2 \alpha = 1$$



Getting rid of sines

Trigonometric identity: $\sin^2\alpha + \cos^2\alpha = 1$

$$\rightsquigarrow \sin^2\alpha = 1 - \cos^2\alpha \quad \textbf{(i)}$$

Snell's law: $n \sin\theta = n_t \sin\phi$

$$\rightsquigarrow \sin\phi = \tfrac{n}{n_t} \sin\theta$$

$$\rightsquigarrow \sin^2\phi = \tfrac{n}{n_t}^2 \sin^2\theta \quad \textbf{(ii)}$$

(i) with $\alpha = \phi$ in **(ii)** gives us:

$$\cos^2\phi = 1 - \tfrac{n}{n_t}^2 \sin^2\theta \quad \textbf{(iii)}$$

(i) with $\alpha = \theta$ in **(iii)** gives us:

$$\cos^2\phi = 1 - \tfrac{n}{n_t}^2 (1 - \cos^2\theta)$$

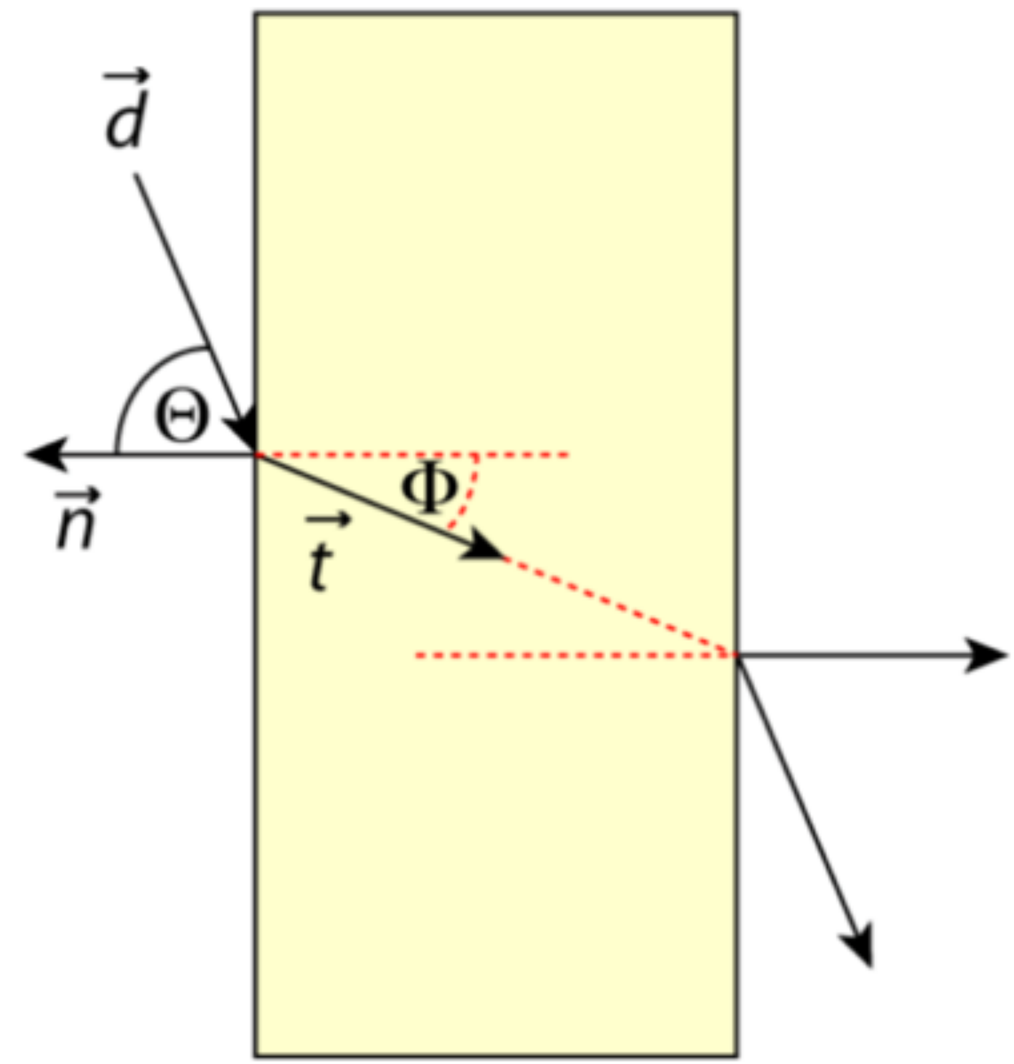
Getting rid of sines

Hence, we can rewrite Snell's law

$$n \sin \theta = n_t \sin \phi$$

with cosines instead of sines

$$\cos^2 \phi = 1 - \frac{n^2(1 - \cos^2 \theta)}{n_t^2}$$

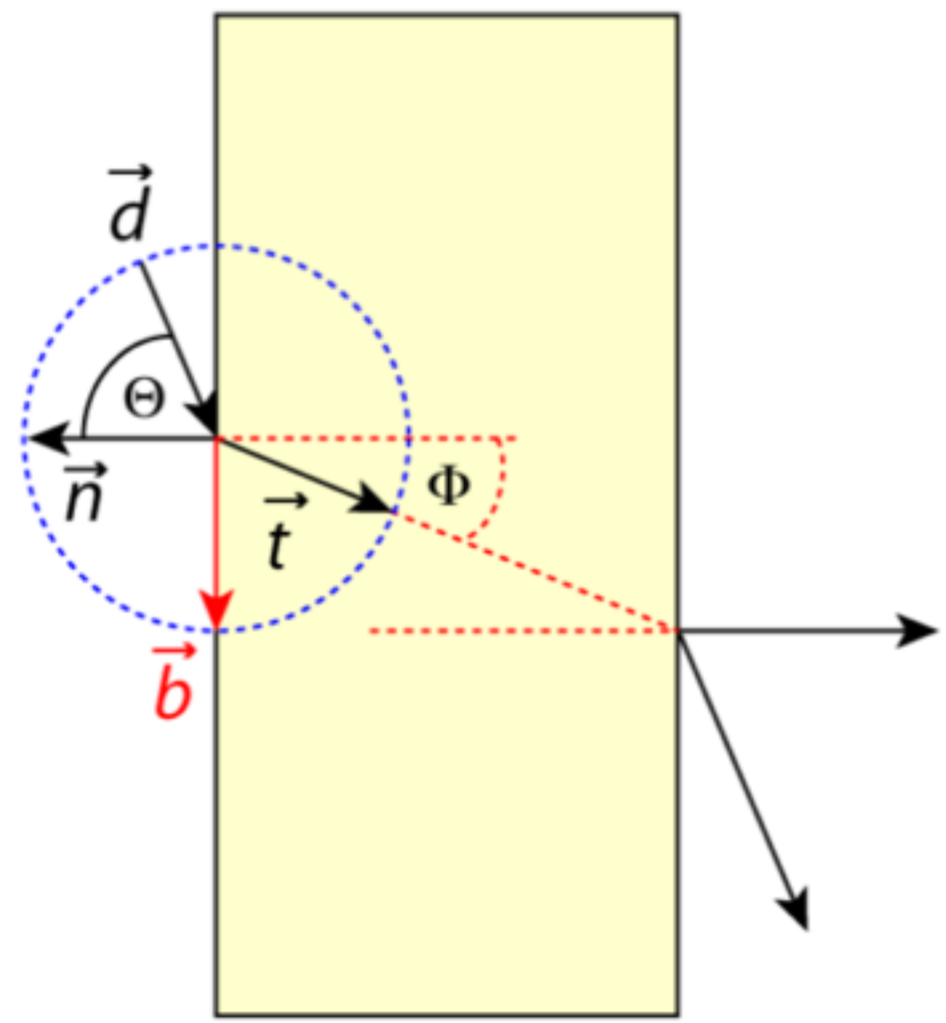


Constructing an orthonormal basis

How do we find the refracted vector $\vec{t}$?

First, notice that $\vec{t}$ lies in the plane spanned by $\vec{d}$ and $\vec{n}$.

If the incoming vector $\vec{d}$ and the normal $\vec{n}$ are normalized, we can express $\vec{t}$ in an orthonormal basis in this plane using an appropriate vector $\vec{b}$.



Finding the refraction vector

The refraction vector $\vec{t}$ is a linear combination of $\vec{b}$ and $-\vec{n}$:

$$\vec{t} = x_t \vec{b} + y_t(-\vec{n})$$

From the image we see that

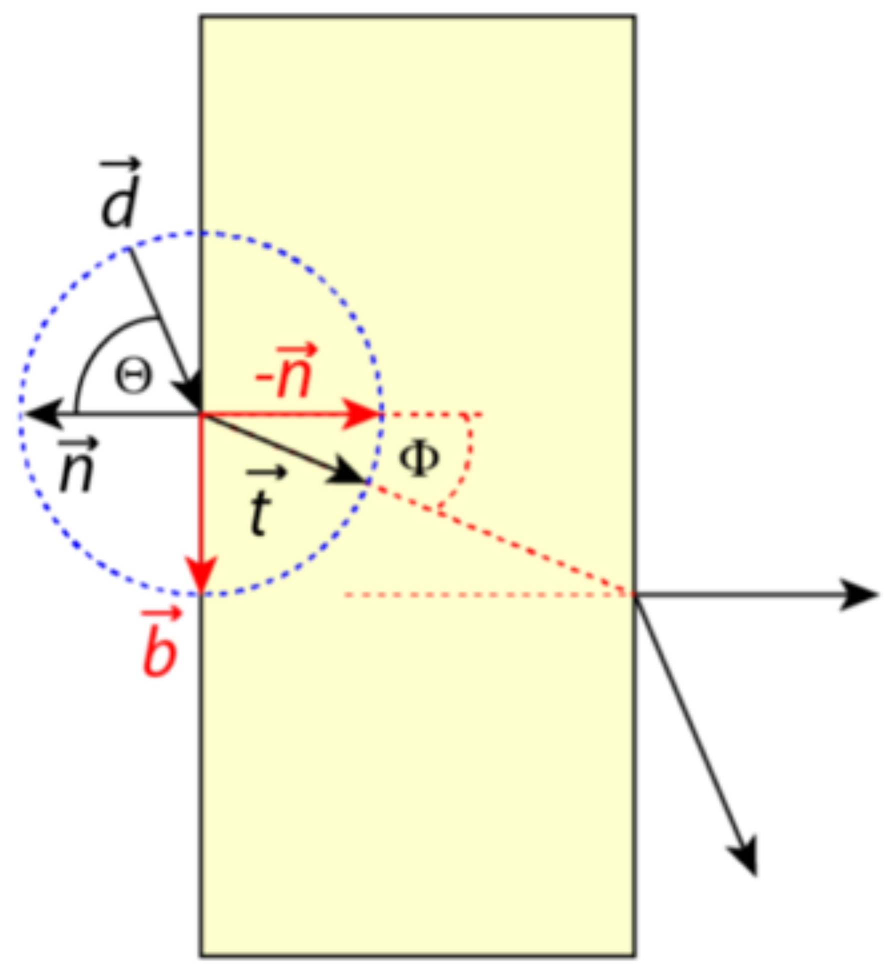
$$x_t = \sin\phi \text{ and } y_t = \cos\phi$$

Hence, we have

$$\vec{t} = \vec{b}\sin\phi - \vec{n}\cos\phi$$

Likewise, we get

$$\vec{d} = \vec{b}\sin\theta - \vec{n}\cos\theta$$



Finding the refraction vector

The refraction vector $\vec{t}$ is a linear combination of $\vec{b}$ and $-\vec{n}$:

$$\vec{t} = x_t\vec{b} + y_t(-\vec{n})$$

From the image we see that

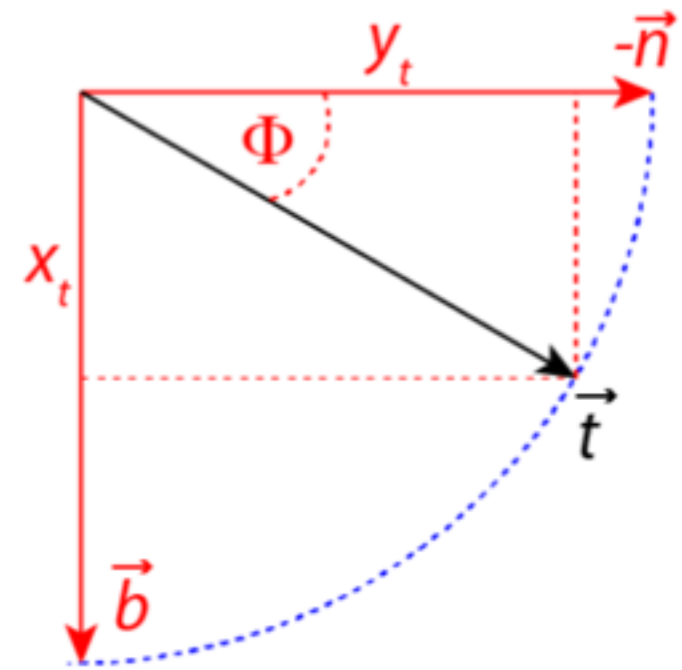
$$x_t = \sin\phi \text{ and } y_t = \cos\phi$$

Hence, we have

$$\vec{t} = \vec{b}\sin\phi - \vec{n}\cos\phi$$

Likewise, we get

$$\vec{d} = \vec{b}\sin\theta - \vec{n}\cos\theta$$



Finding the refraction vector

We have

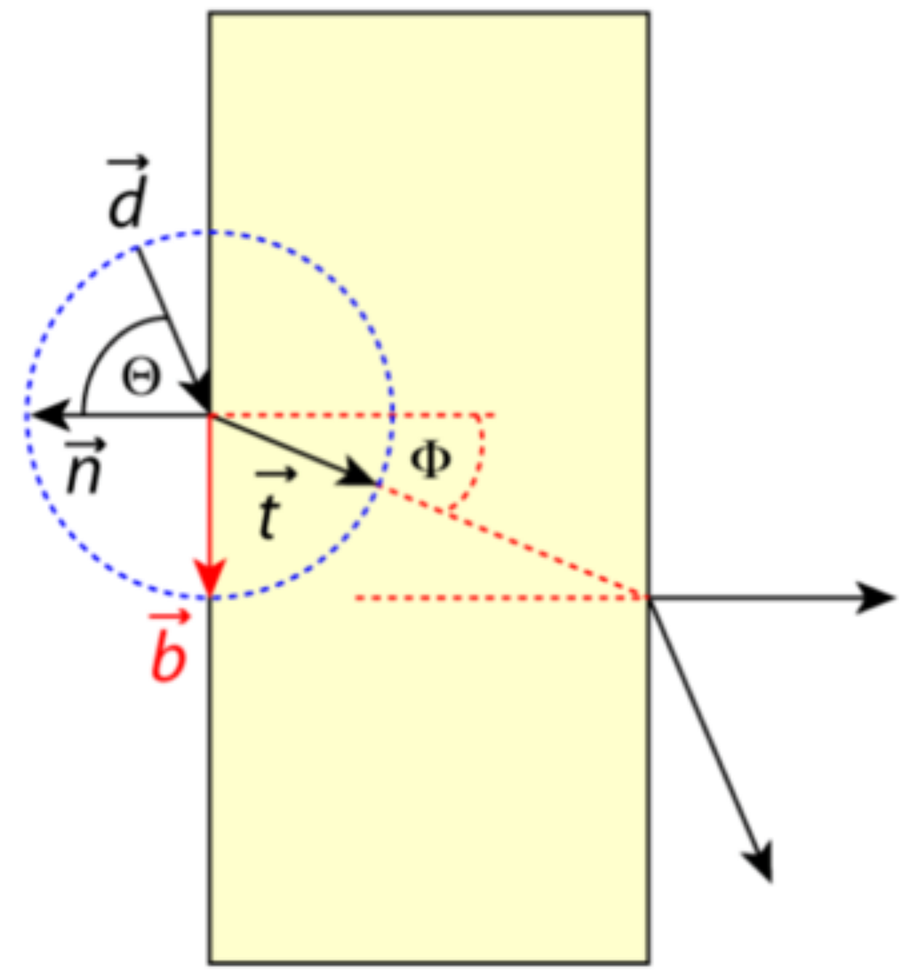
$$\begin{aligned}\vec{t} &= \vec{b}\sin\phi - \vec{n}\cos\phi \\ \vec{d} &= \vec{b}\sin\theta - \vec{n}\cos\theta\end{aligned}$$

So we can solve for $\vec{b}$:

$$\vec{b} = \frac{\vec{d}+\vec{n}\cos\theta}{\sin\theta}$$

and for $\vec{t}$:

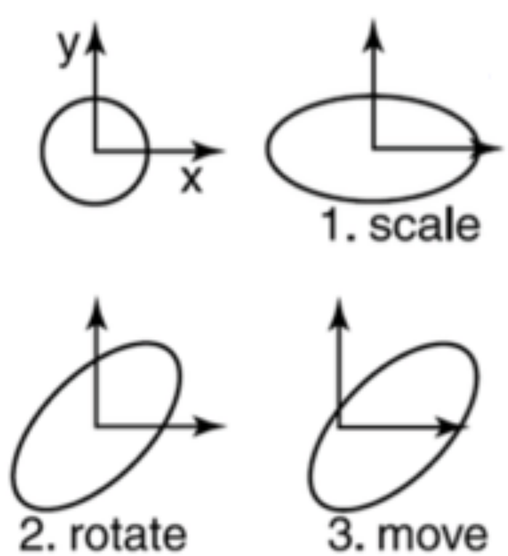
$$\begin{aligned}\vec{t} &= \frac{\sin\phi(\vec{d}+\vec{n}\cos\theta)}{\sin\theta} - \vec{n}\cos\phi \\ &= \frac{n(\vec{d}+\vec{n}\cos\theta)}{n_t} - \vec{n}\cos\phi \\ &= \frac{n(\vec{d}-\vec{n}(\vec{d}\cdot\vec{n}))}{n_t} - \vec{n}\sqrt{1 - \frac{n^2(1-(\vec{d}\cdot\vec{n})^2)}{n_t^2}}\end{aligned}$$

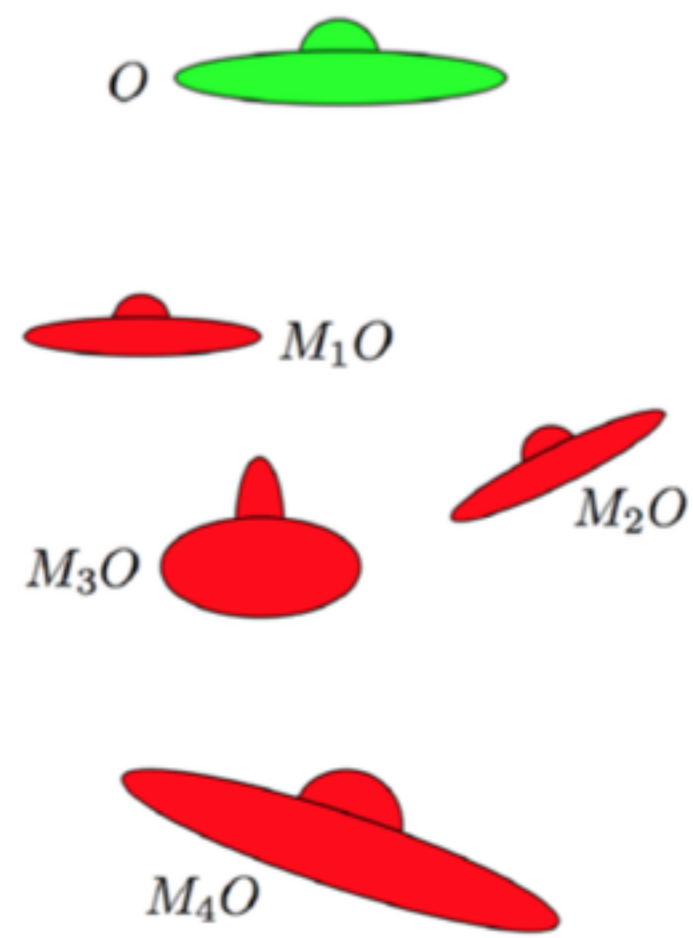


Copying and transforming objects

Instancing is an elegant technique to place various transformed copies of an object in a scene.

Expl.: circle → elipse

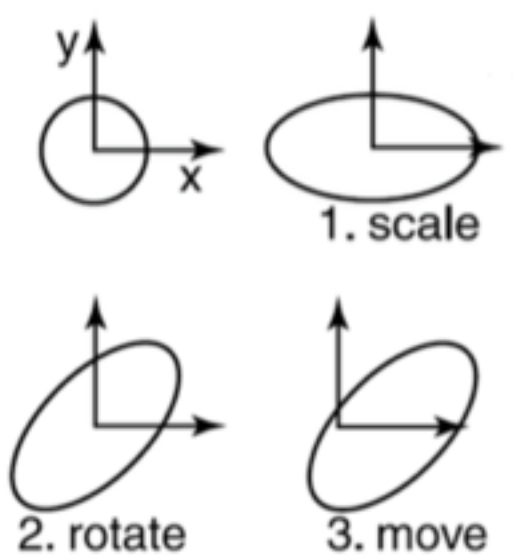


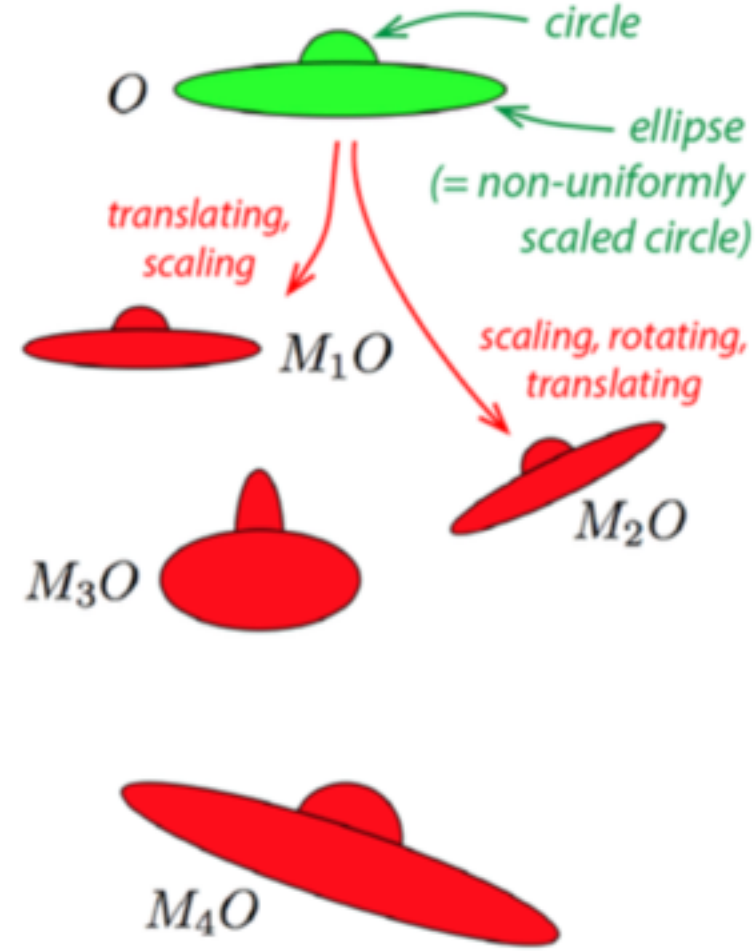


Copying and transforming objects

Instancing is an elegant technique to place various transformed copies of an object in a scene.

Expl.: circle → elipse



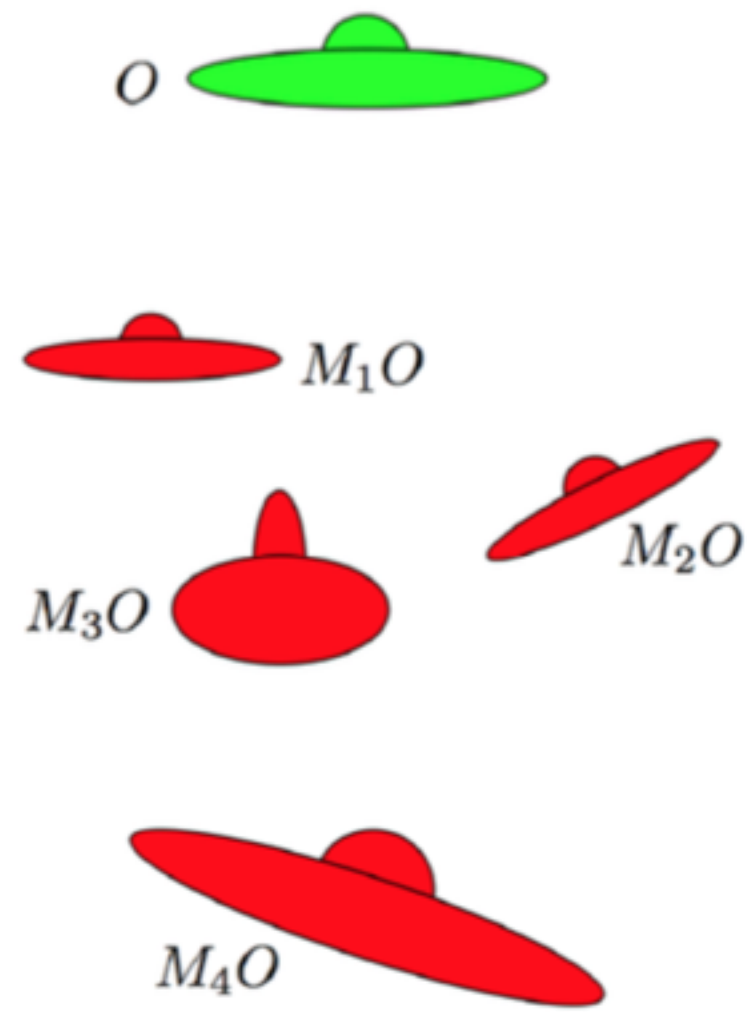


Copying and transforming objects

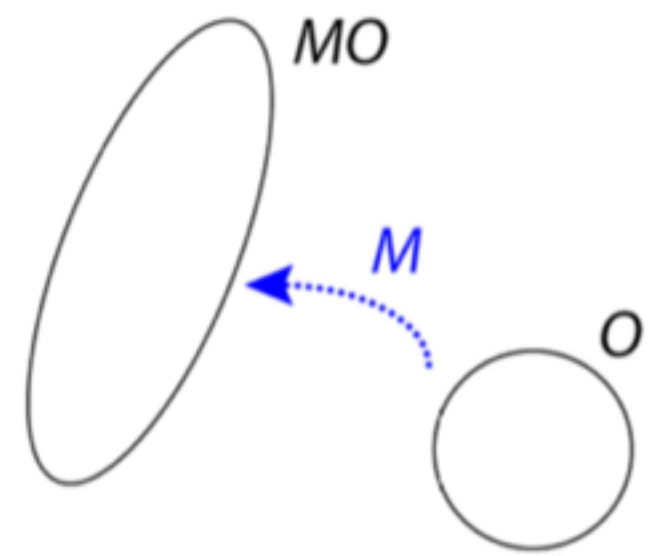
Instead of making actual copies, we simply store a reference to a base object, together with a transformation matrix.

That can save us lots of storage.

Hmm, but how do we compute the intersection of a ray with a randomly rotated elipse?



Ray-instance intersection

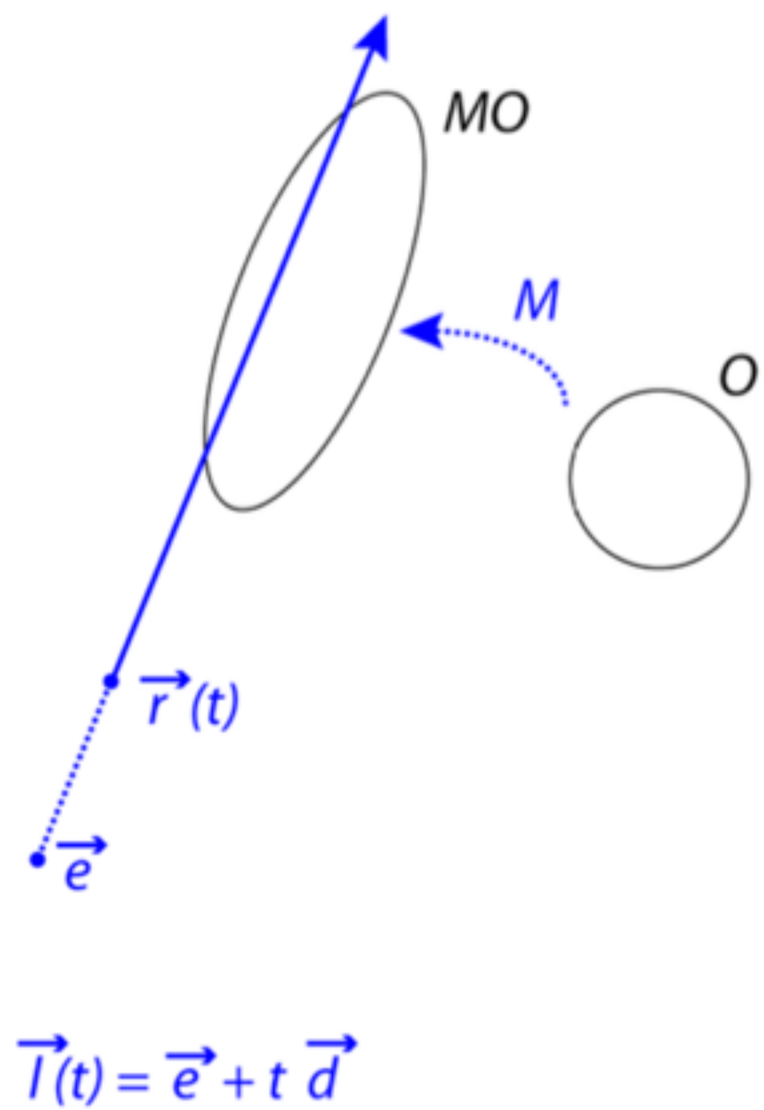


Assume an object O that is used to create an object MO via instancing.

Ray-instance intersection

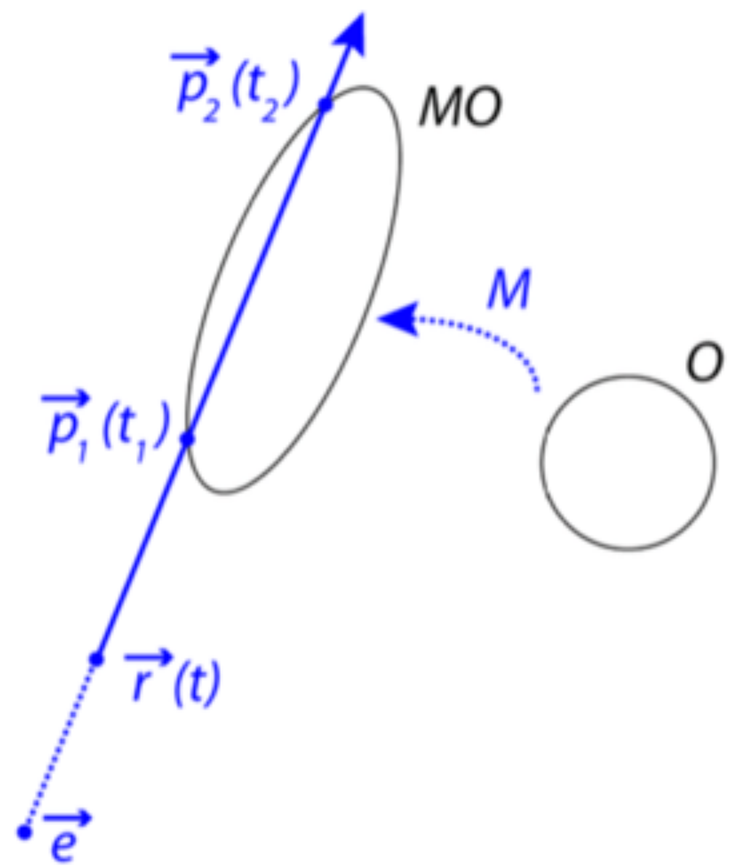
Now, we want to create the intersection of MO with the ray $\vec{r}(t)$, which in turn is defined by the line

$$\vec{l}(t) = \vec{e} + t\vec{d}.$$



Ray-instance intersection

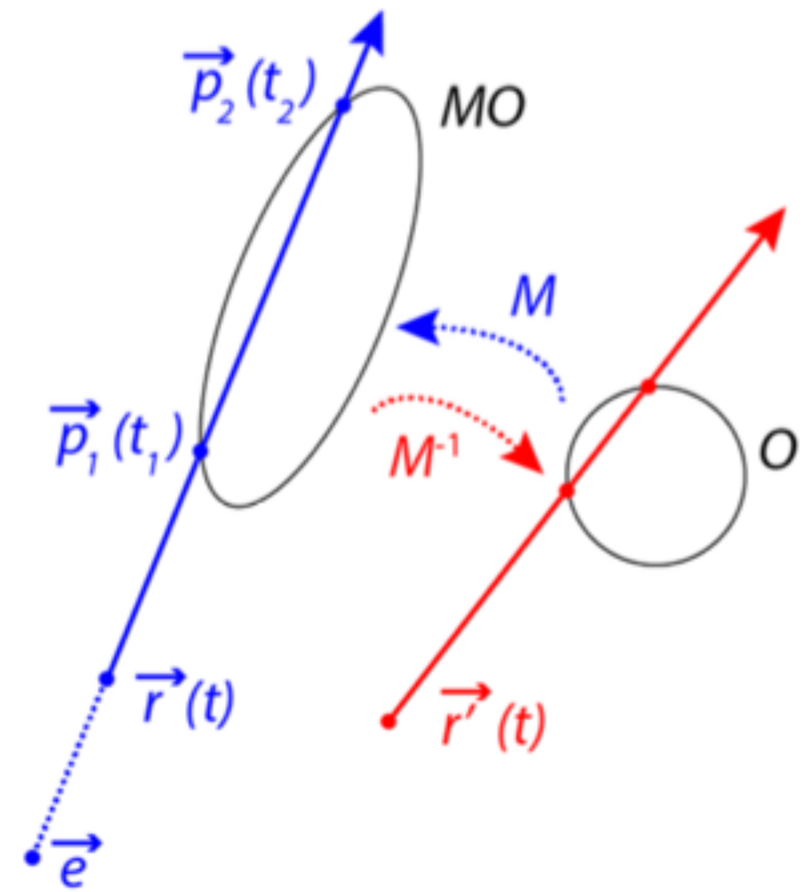
Fortunately, such complicated intersection tests (e.g. ray/ellipsoid) can often be replaced by much simpler tests (e.g. ray/sphere).



$\vec{l}(t) = \vec{e} + t\,\vec{d}$

Ray-instance intersection

To determine the intersections $\vec{p_i}$ of a ray $\vec{r}$ with the instance MO, we first compute the intersections $\vec{p_i'}$ of the inverse transformed ray $M^{-1}\vec{r}$ and the original object O.



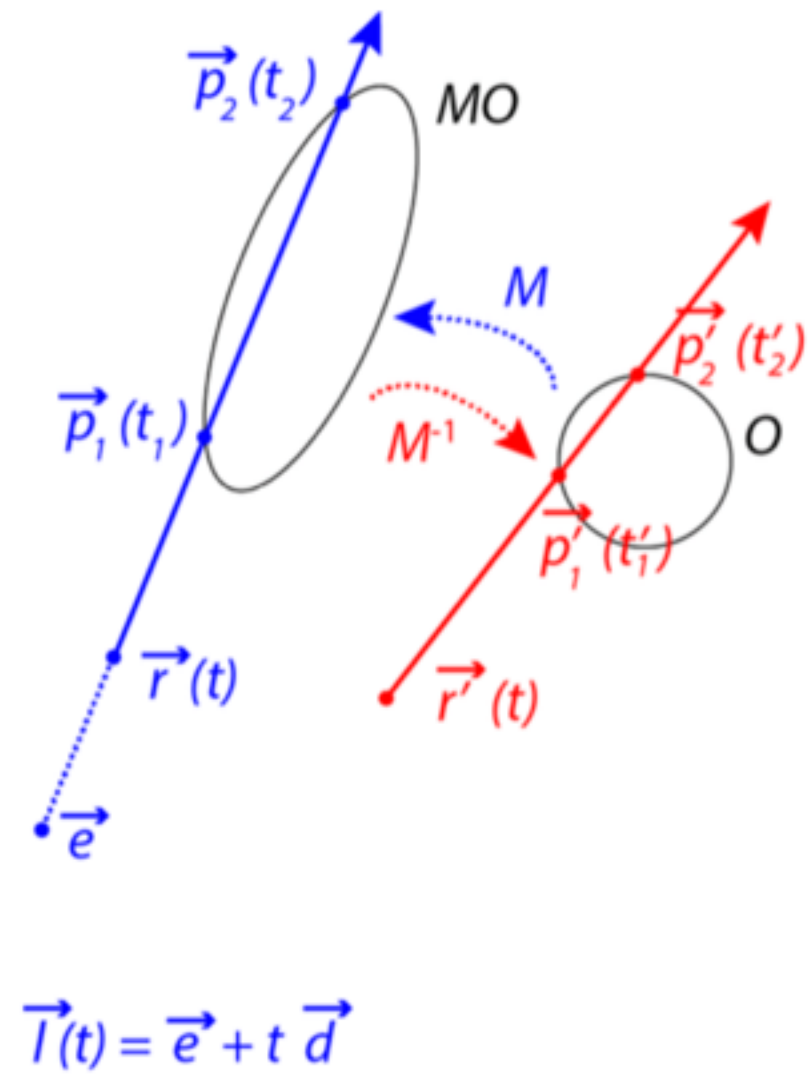
$$\vec{l}(t) = \vec{e} + t\,\vec{d}$$

Ray-instance intersection

The points $\vec{p_i}$ are then simply

$$M\vec{p_i'} \quad \text{or} \quad \vec{l}(t_i')$$

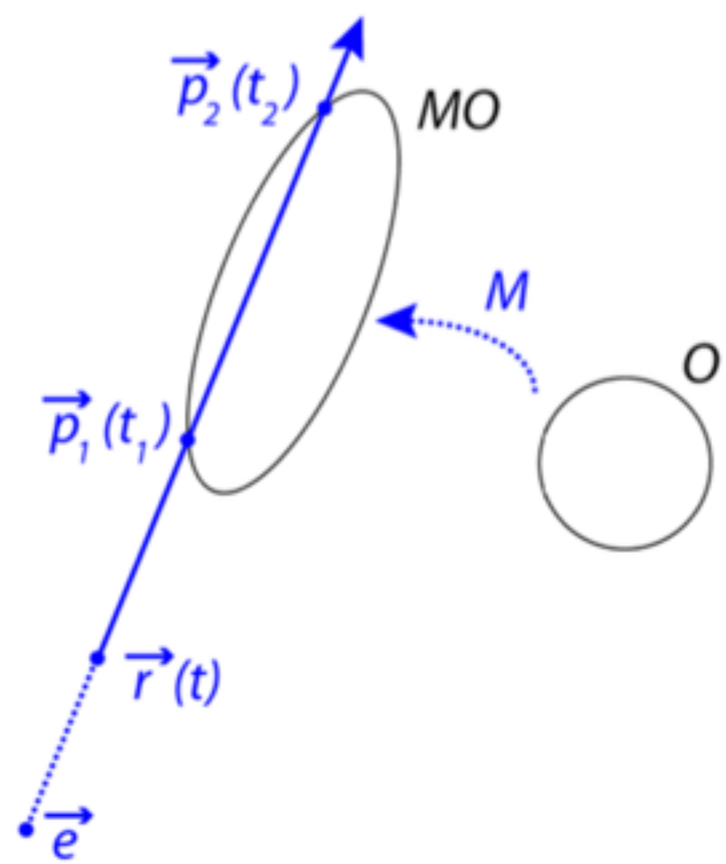
because the linear transformation preserves relative distances along the line.



Ray-instance intersection

Two pitfalls:

- The direction vector of the ray should *not* be normalized
- Surface normals transform differently!
  $\rightarrow$ use $(M^{-1})^T$ instead of M for normals



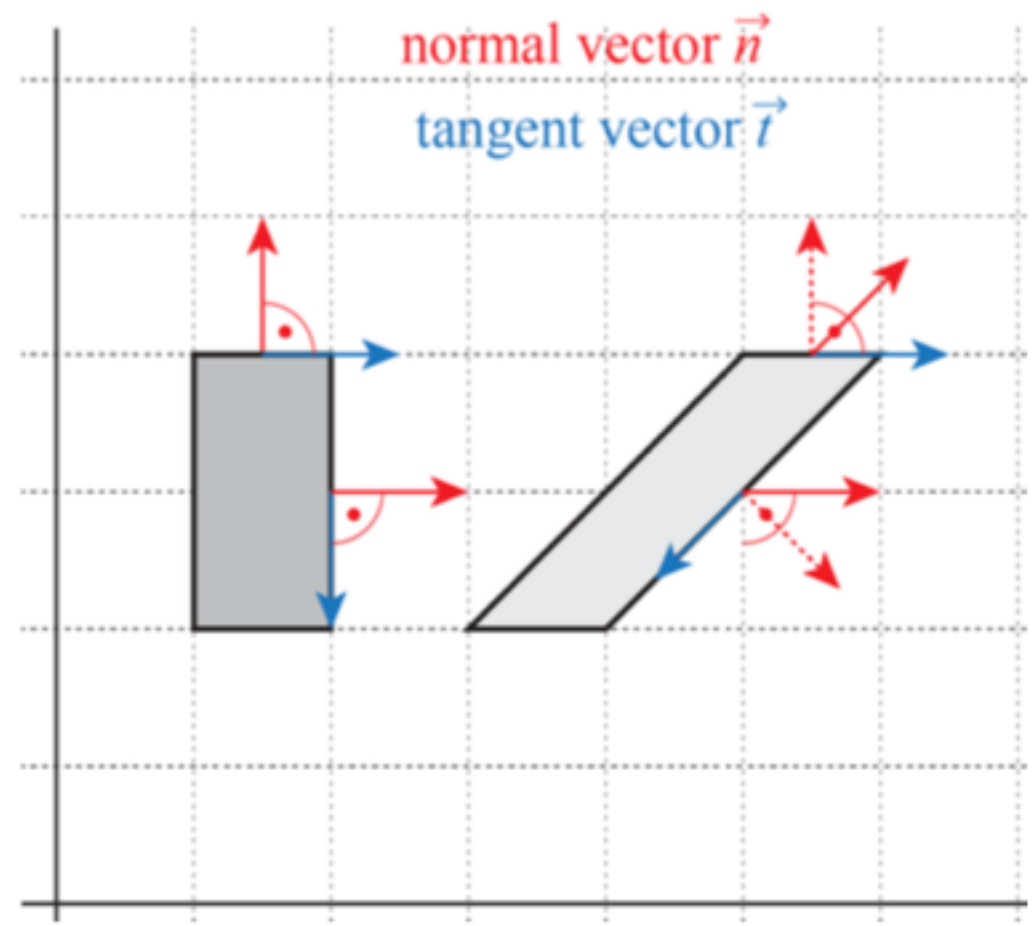
$\vec{l}(t) = \vec{e} + t\,\vec{d}$

Transforming normal vectors

Unfortunately, normal vectors are not always transformed properly.

E.g. look at shearing, where tangent vectors are correctly transformed but normal vectors not.

To transform a normal vector $\vec{n}$ correctly under a given linear transformation A, we have to apply the matrix $(A^{-1})^T$. Why?



Transforming normal vectors

We know that tangent vectors are tranformed correctly: $A\vec{t} = \vec{t_A}$.

But this is not necessarily true for normal vectors: $A\vec{n} \neq \vec{n_A}$.

Goal: find matrix N_A that transforms $\vec{n}$ correctly, i.e. $N_A\vec{n} = \vec{n_N}$ where $\vec{n_N}$ is the correct normal vector of the transformed surface.

Because our original normal vector $\vec{n}^T$ is perpendicular to the original tangent vector $\vec{t}$, we know that:

$$\vec{n}^T\vec{t} = 0.$$

This is the same as

$$\vec{n}^T I\vec{t} = 0$$

which is is the same as

$$\vec{n}^T A^{-1} A\vec{t} = 0$$

Transforming normal vectors

Because $A\vec{t} = \vec{t_A}$ is our correctly transformed tangent vector,
we have

$$\vec{n}^T A^{-1} \vec{t_A} = 0$$

Because their scaler product is 0, $\vec{n}^T A^{-1}$ must be orthogonal to it.
So, the vector we are looking for must be

$$\vec{n}_N^T = \vec{n}^T A^{-1}.$$

Because of how matrix multiplication is defined,
this is a transposed vector. But we can rewrite this to

$$\vec{n}_N = (\vec{n}^T A^{-1})^T.$$

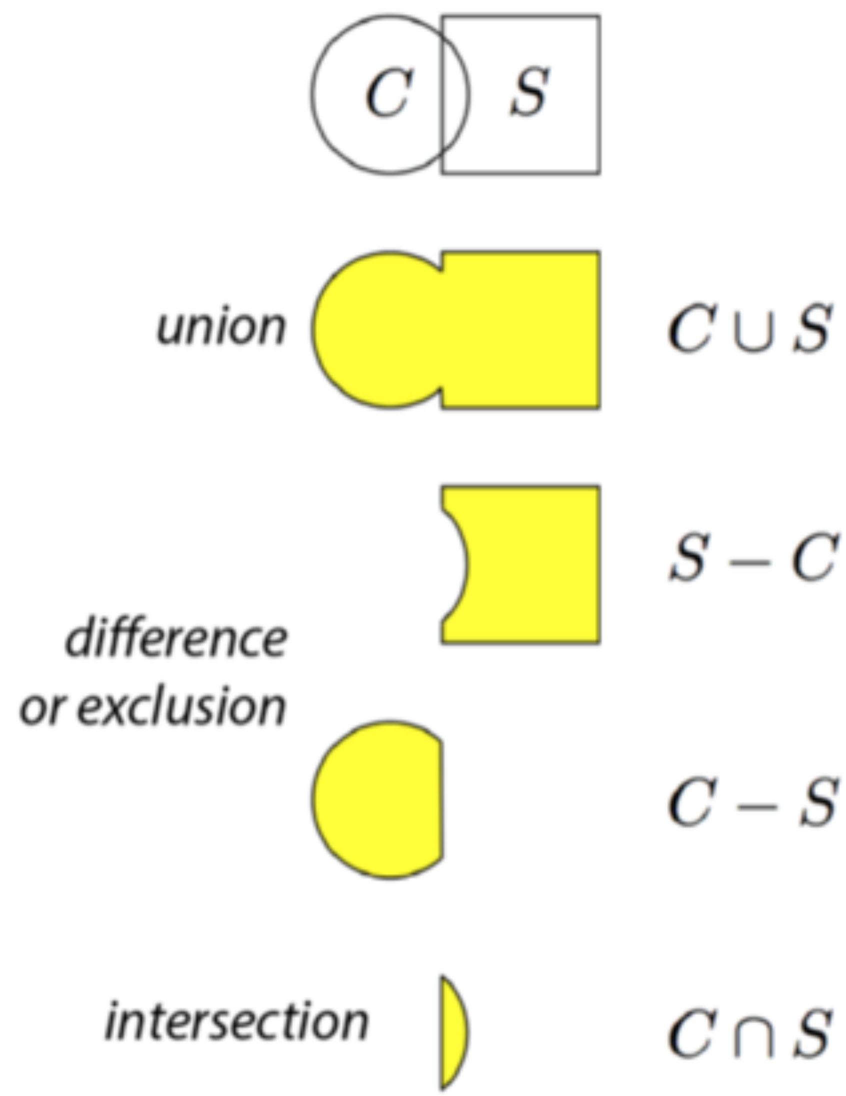
And if you remember that $(AB)^T = B^T A^T$, we get

$$\vec{n}_N = (A^{-1})^T \vec{n}$$

Constructive solid geometry

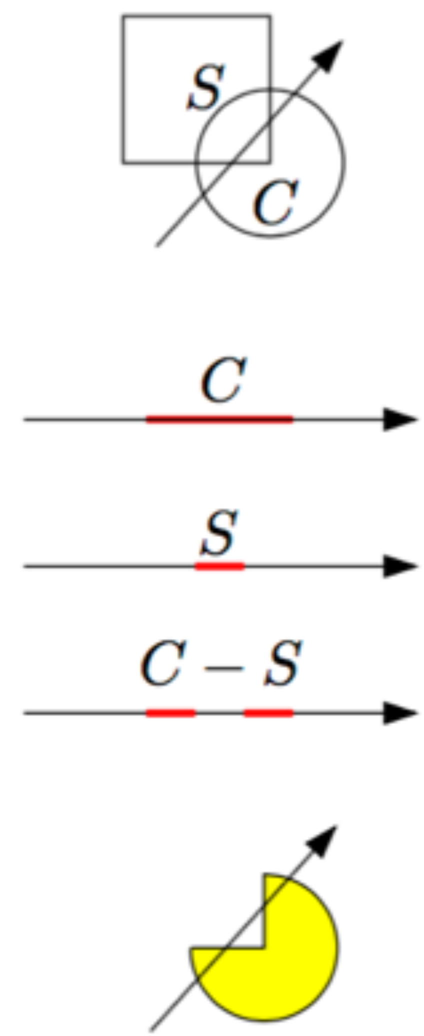
To model our scenes in ray tracing, we can basically use any object that allows us to calculate its intersection with a 3D line.

Using Constructive Solid Geometry (CSG), we can build complex objects from simple ones with set operations.



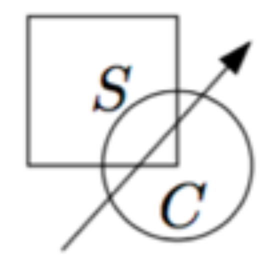
Intersection and CSG

Instead of actually constructing the objects, we can calculate ray-object intersections with the original objects and perform set operations on the resulting intervals.

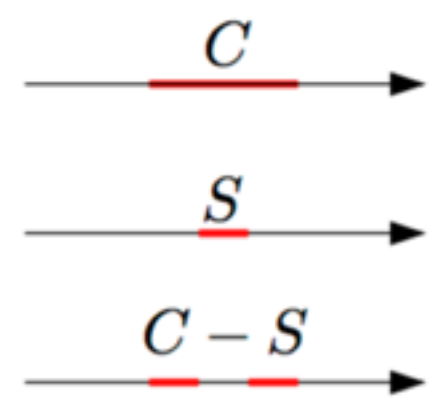


Intersection and CSG

For every base object, we maintain an interval (or set of intervals) representing the part of the ray inside the object.



The intervals for combined objects are computed with the same set operations that are applied to the base objects.



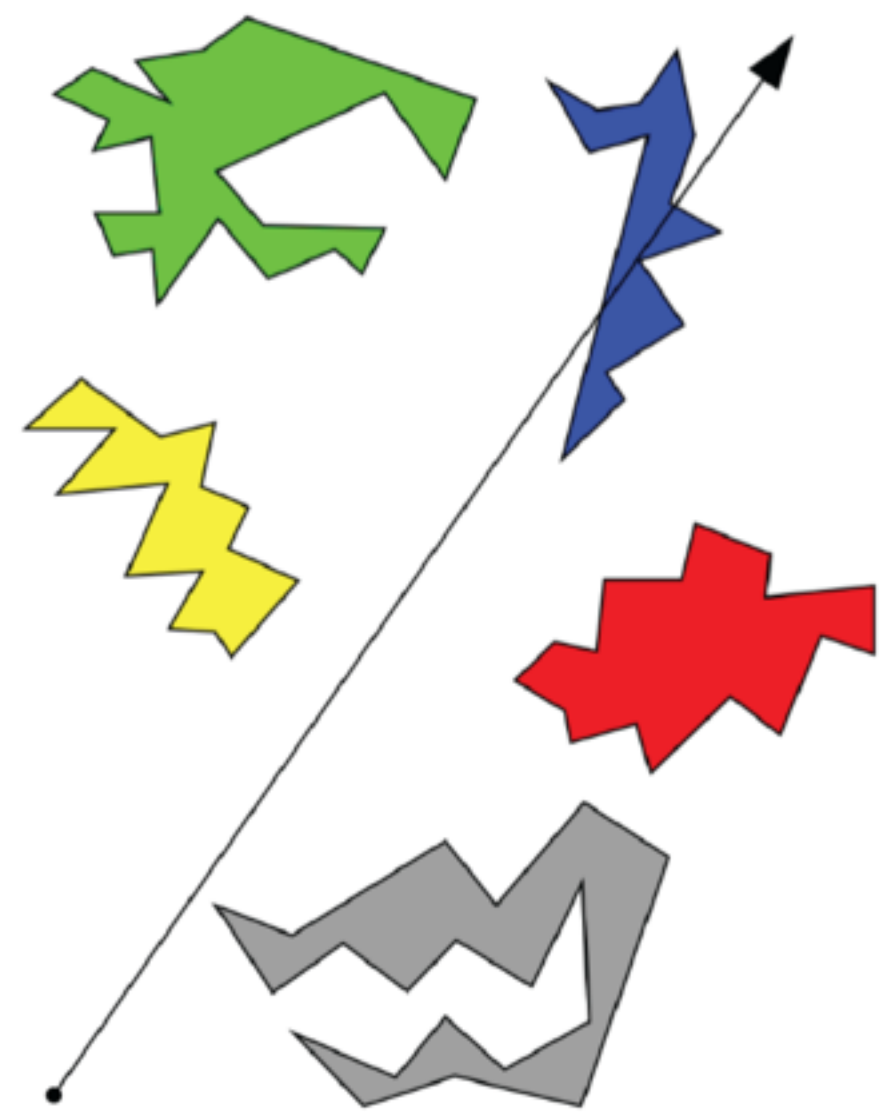
The borders of the resulting intervals are our intersection points.

The bottleneck in ray tracing

As said before: ray tracing is generally SLOW.

It is estimated that 75% to 95% of the time in ray/tracing is spent on ray/object intersections [Chang, 2001].

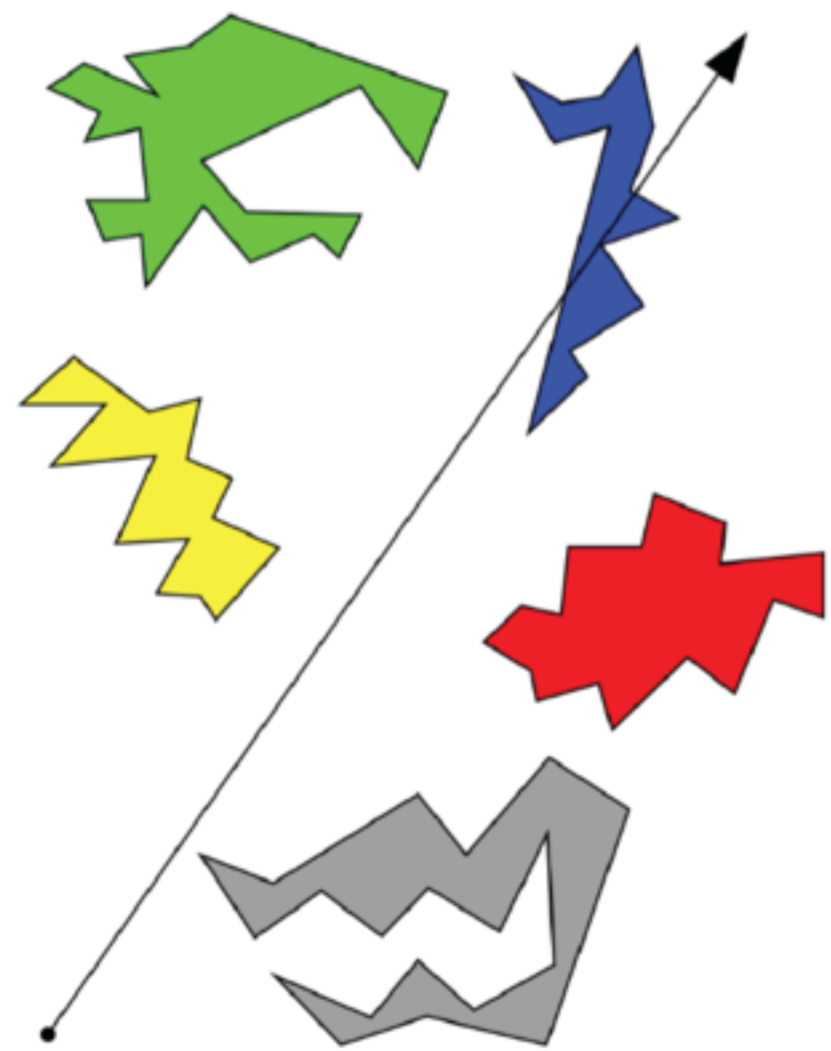
Hence, there are many approaches to increase calculation speed by reducing the number of necessary intersection tests.



Spatial data structures

Applied for

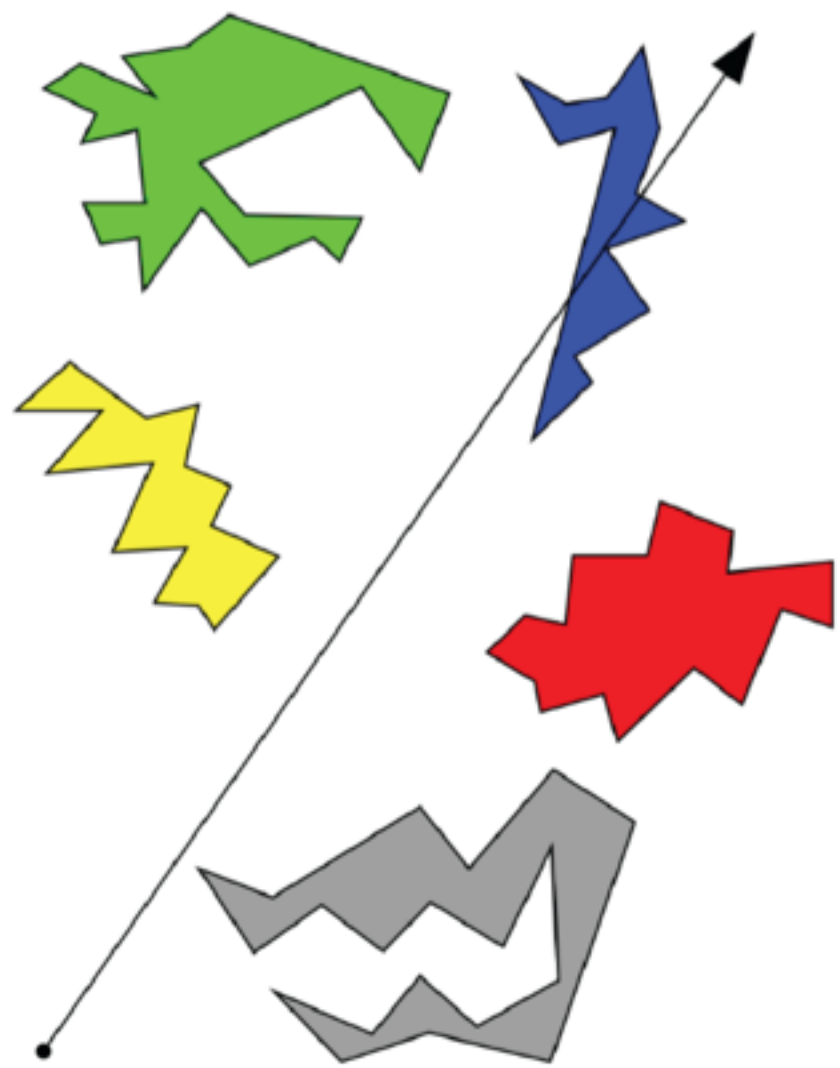
- ray tracing
- culling
- collision detection



Spatial data structures

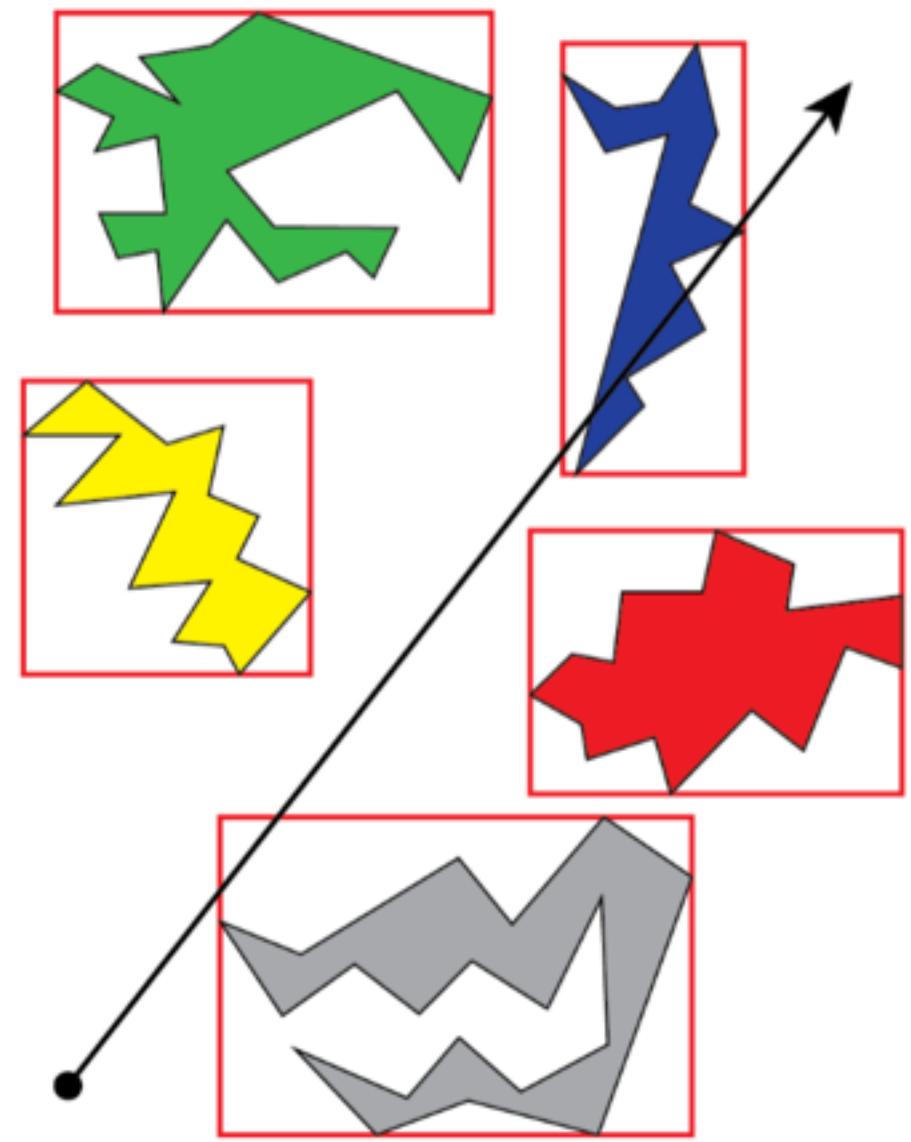
We distinguish between

- object partitioning schemes
- space partitioning schemes



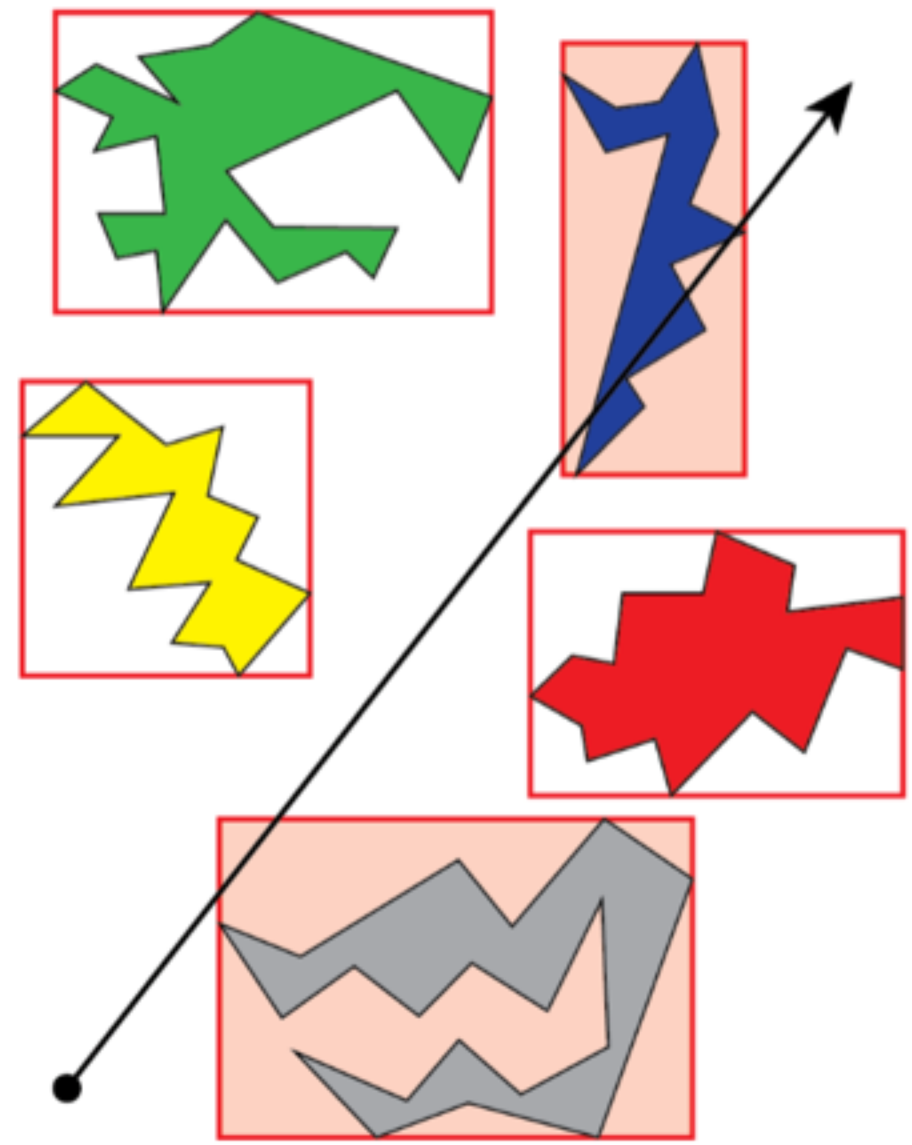
Ray Tracing Improvements: Bounding boxes

A common technique to improve ray/object intersection query times is the use of bounding boxes.



Bounding boxes

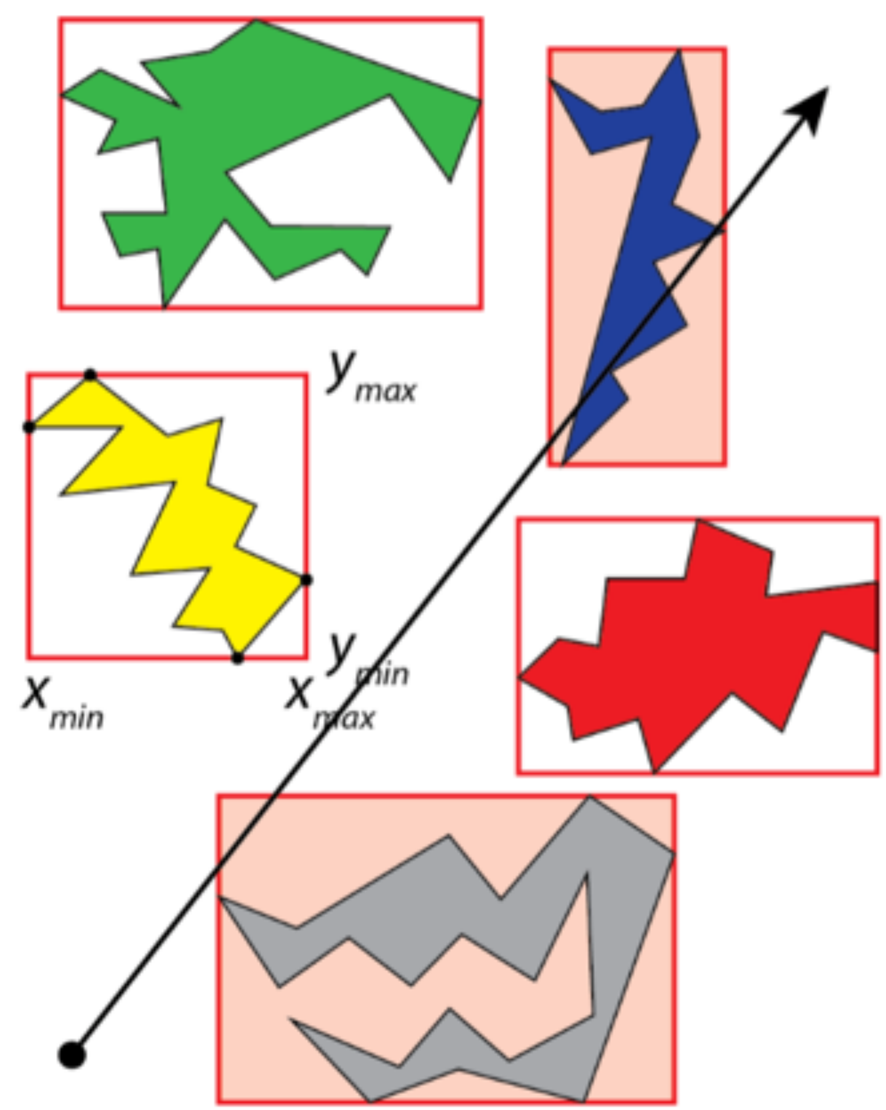
A common technique to improve ray/object intersection query times is the use of bounding boxes.



Bounding boxes

A common technique to improve ray/object intersection query times is the use of bounding boxes.

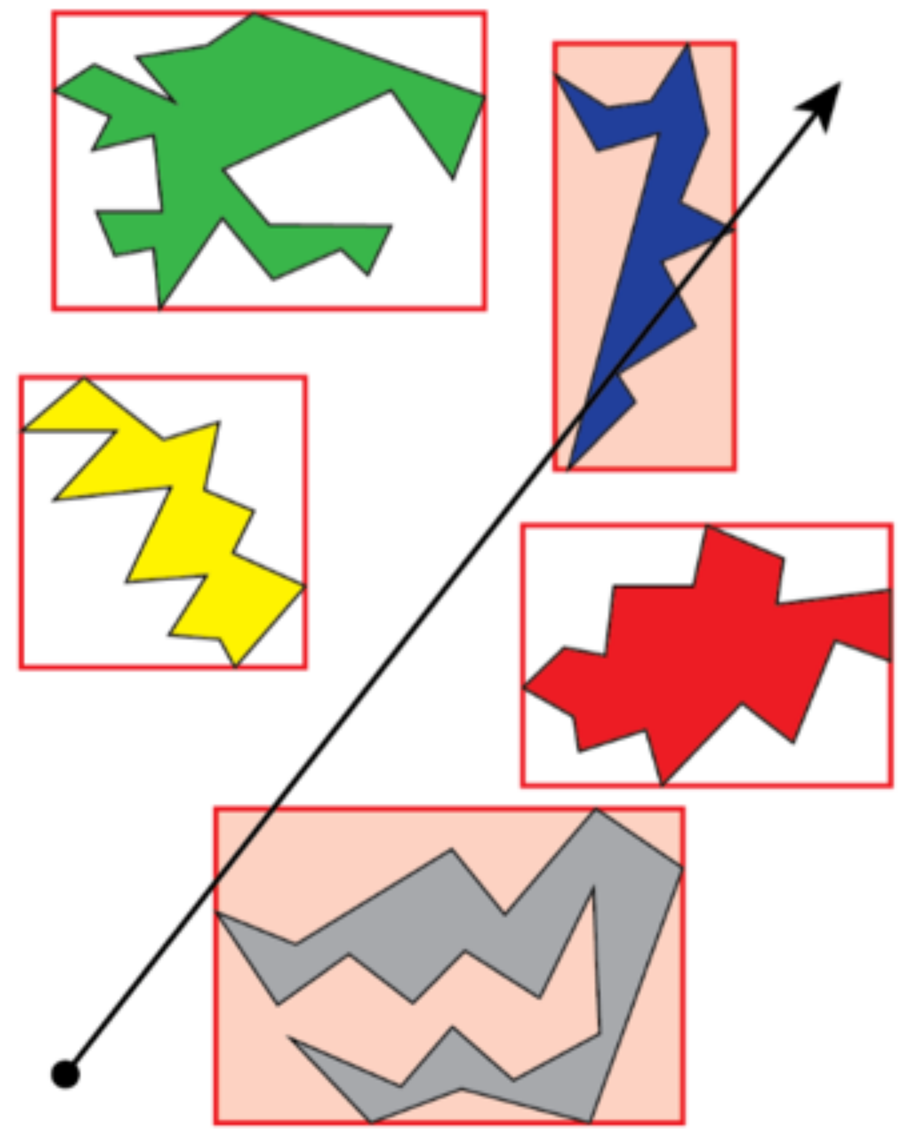
- $\vec{b}_0 = (\min\{x_i\}, \min\{y_j\})$
- $\vec{b}_1 = (\max\{x_i\}, \max\{y_j\})$



Bounding boxes

Note: We don't need the actual intersection point but just a yes or no answer to the intersection test.

How can we compute this easily?



Bounding boxes

First, we calculate the intersection of the ray $\vec{r}$ with the lines defined by the borders of the bounding box.

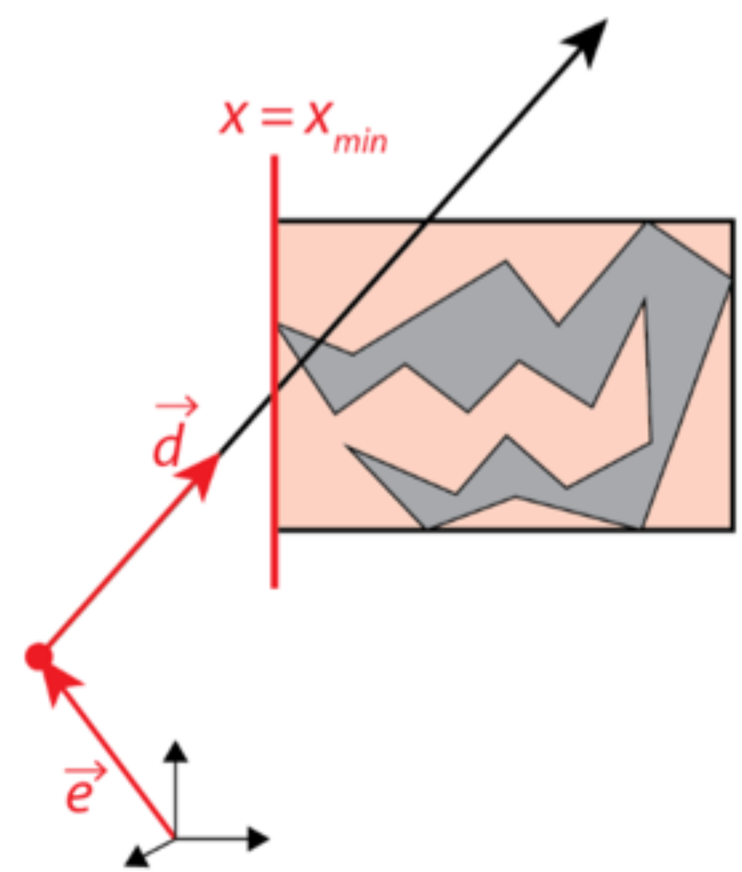
For example for the left border, we have

$$\vec{r}(t) = \begin{pmatrix} x_e \\ y_e \end{pmatrix} + t \begin{pmatrix} x_d \\ y_d \end{pmatrix} \text{ and } x = x_{min}$$

which gives us

$$x_e + t_{x_{min}} x_d = x_{min}$$

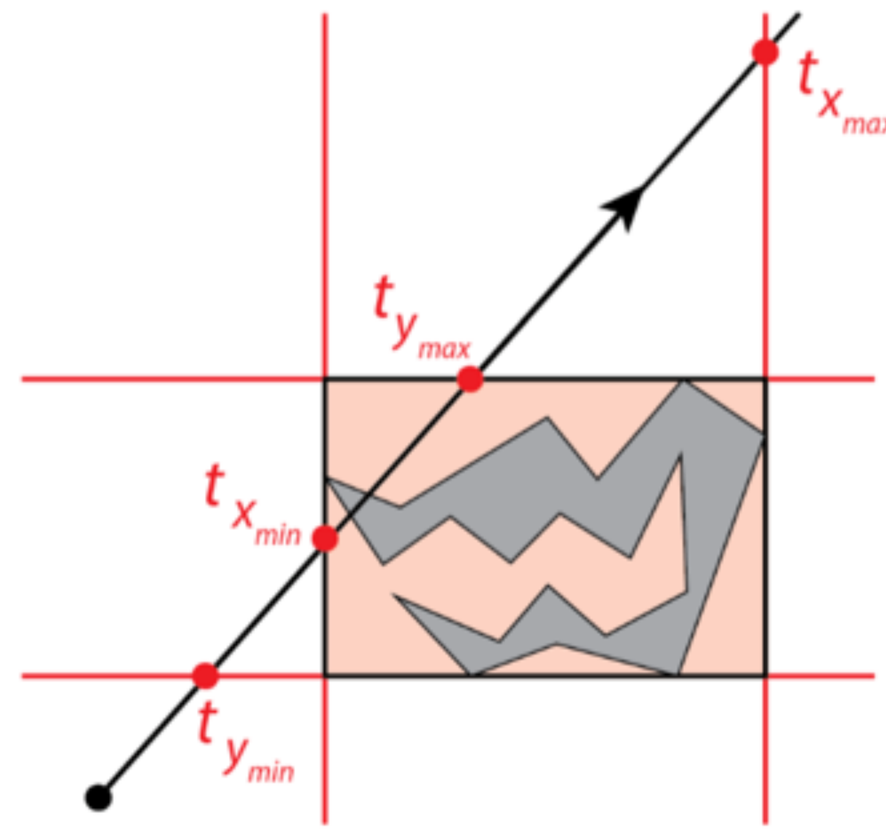
$$t_{x_{min}} = (x_{min} - x_e)/x_d$$



Bounding boxes

Likewise, we get the following values for the other borders of the bounding box:

$$
\begin{aligned}
t_{x_{min}} &= (x_{min} - x_e)/x_d \\
t_{x_{max}} &= (x_{max} - x_e)/x_d \\
t_{y_{min}} &= (y_{min} - y_e)/y_d \\
t_{y_{max}} &= (y_{max} - y_e)/y_d
\end{aligned}
$$



Notice that for simplicity we are assuming positive values for x_d and y_d here. The other cases are symmetrical (exercise!).

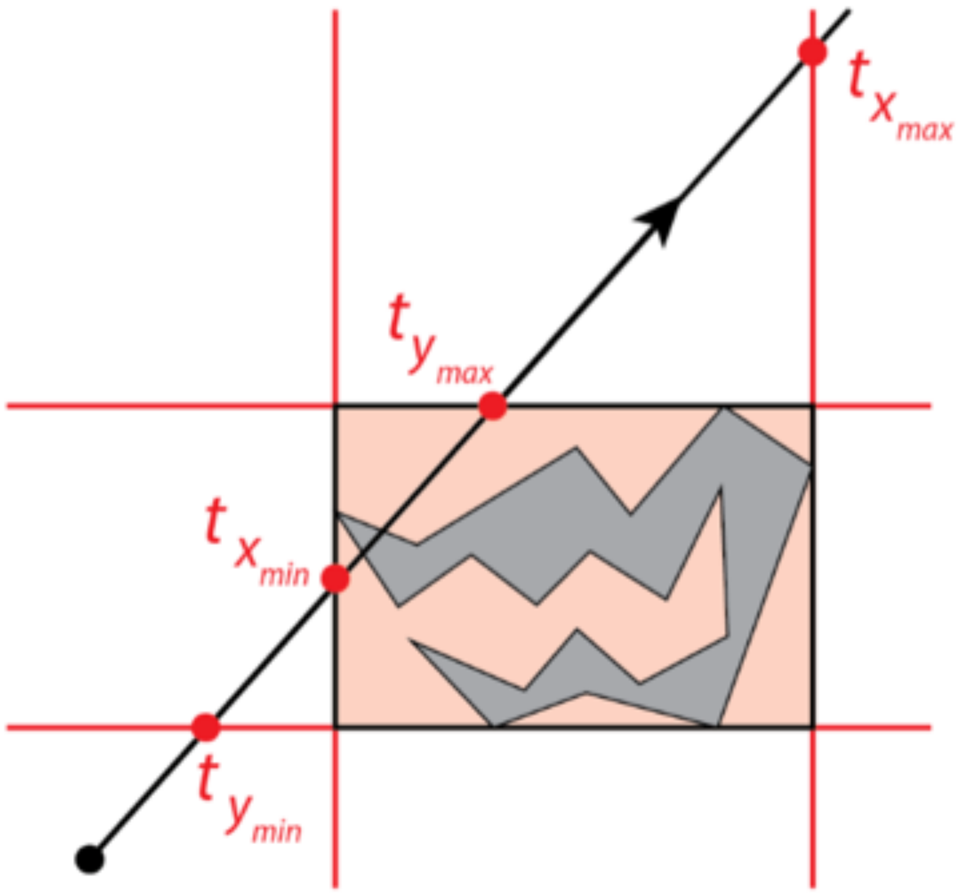
Bounding boxes

How does this help us in answering the question?

Hint: look at the intervals

$$[t_{x_{min}}, t_{x_{max}}] \quad \text{and} \quad [t_{y_{min}}, t_{y_{max}}]$$

Notice that for simplicity we are assuming positive values *for x_d and y_d here. The other cases are symmetrical (exercise!).*



Bounding boxes

By looking at the intervals

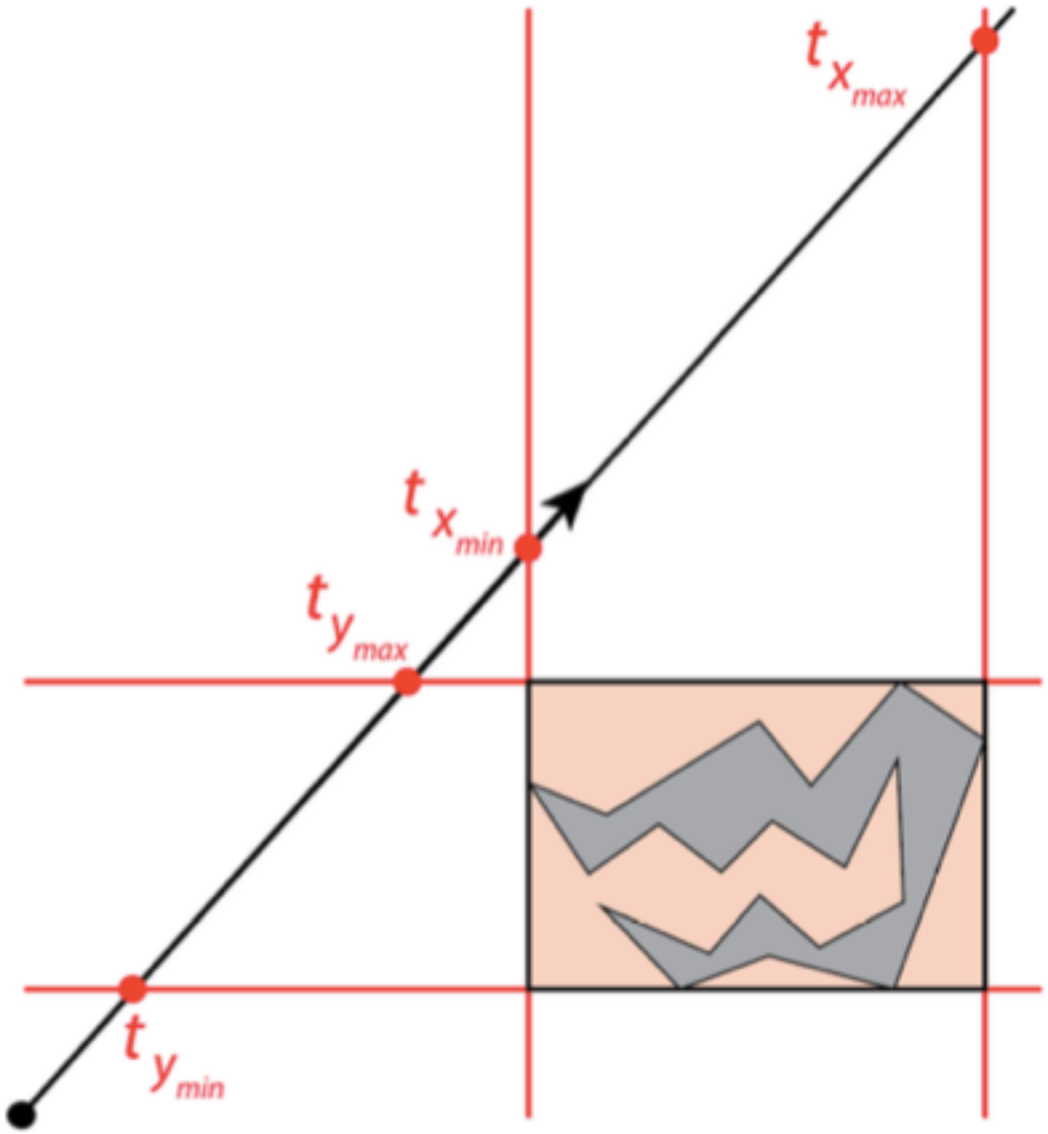
$$[t_{x_{min}}, t_{x_{max}}] \quad \text{and} \quad [t_{y_{min}}, t_{y_{max}}]$$

we see that the ray misses the box if and only if they don't overlap, i.e. if

$$t_{x_{min}} > t_{y_{max}}$$

or

$$t_{y_{min}} > t_{x_{max}}$$

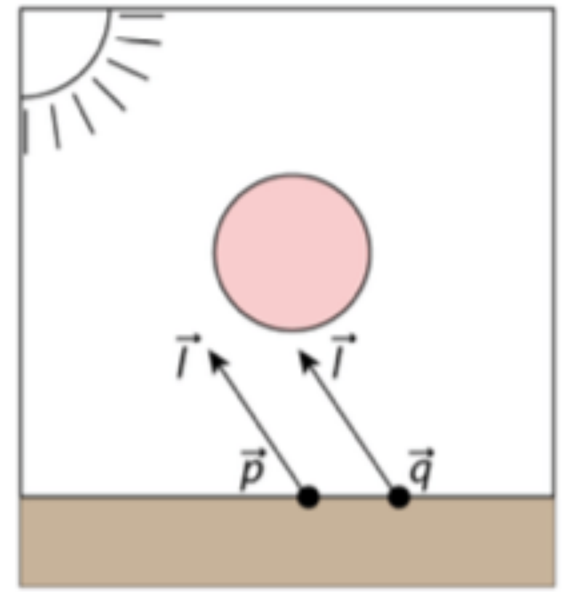


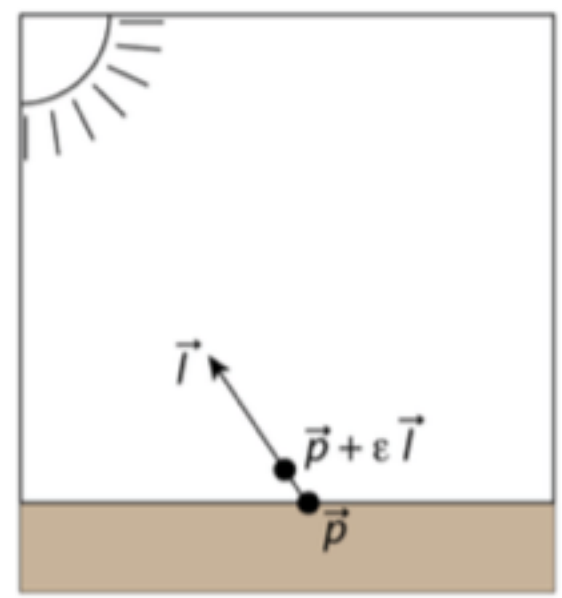
Shadow feelers

Shadows can be implemented fairly easy by so called shadow feelers / rays.

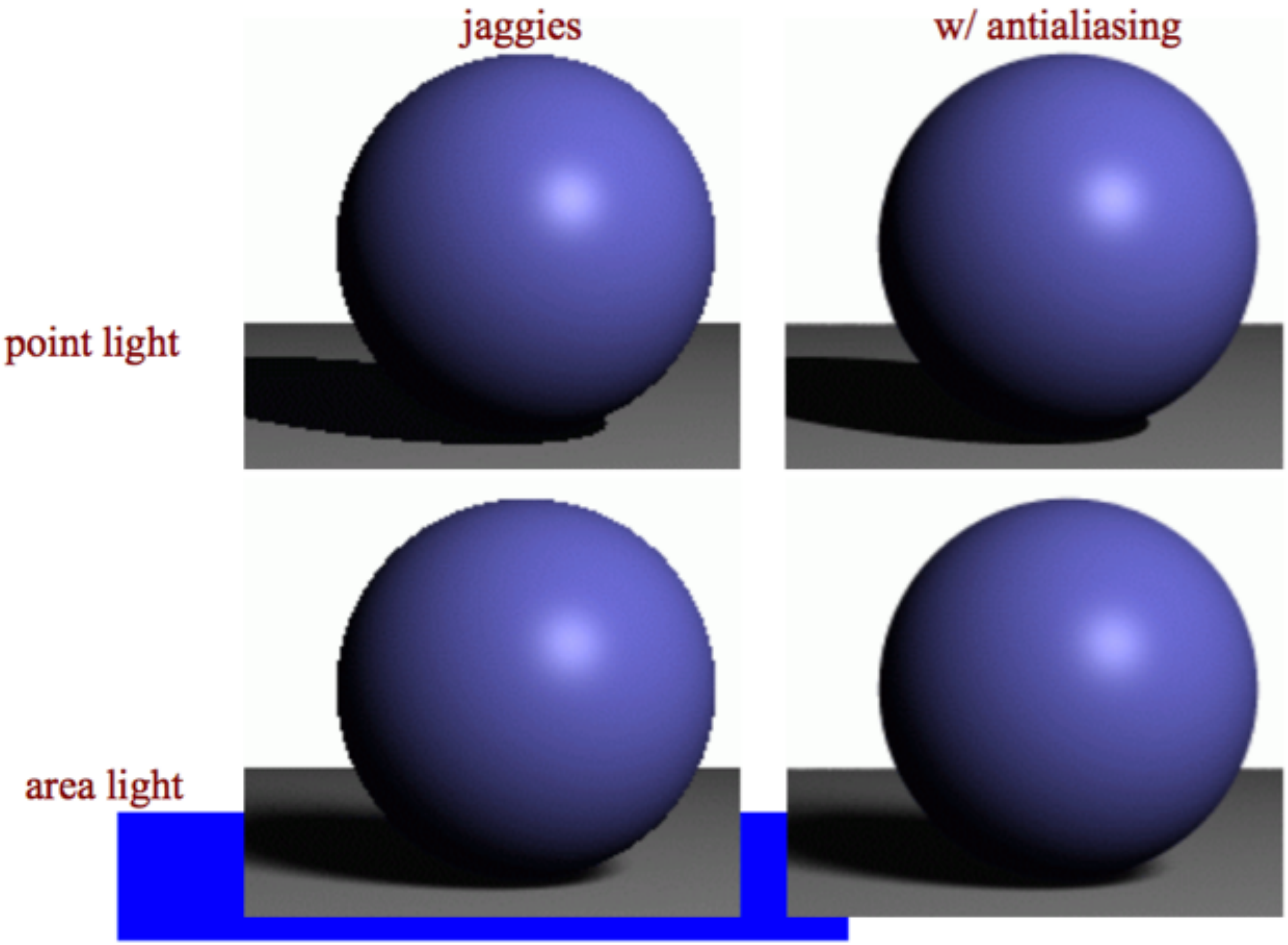
- Shoot a shadow ray $\vec{p} + t\vec{l}$ from a point $\vec{p}$ towards a light source $\vec{l}$.
- If ray hits an object, $\vec{p}$ is in the shadow. Otherwise it's not.

Because of potential precision issues, we often look at point $\vec{p} + \epsilon\vec{l}$ instead of $\vec{p}$.

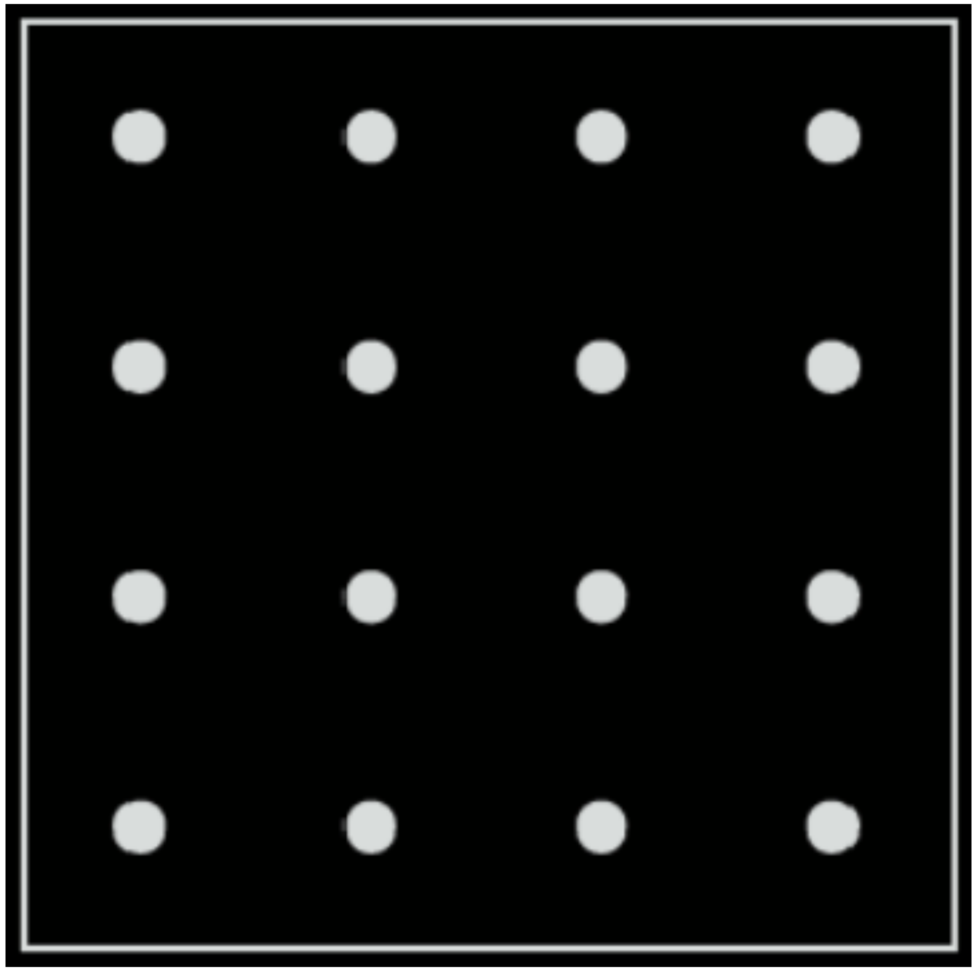




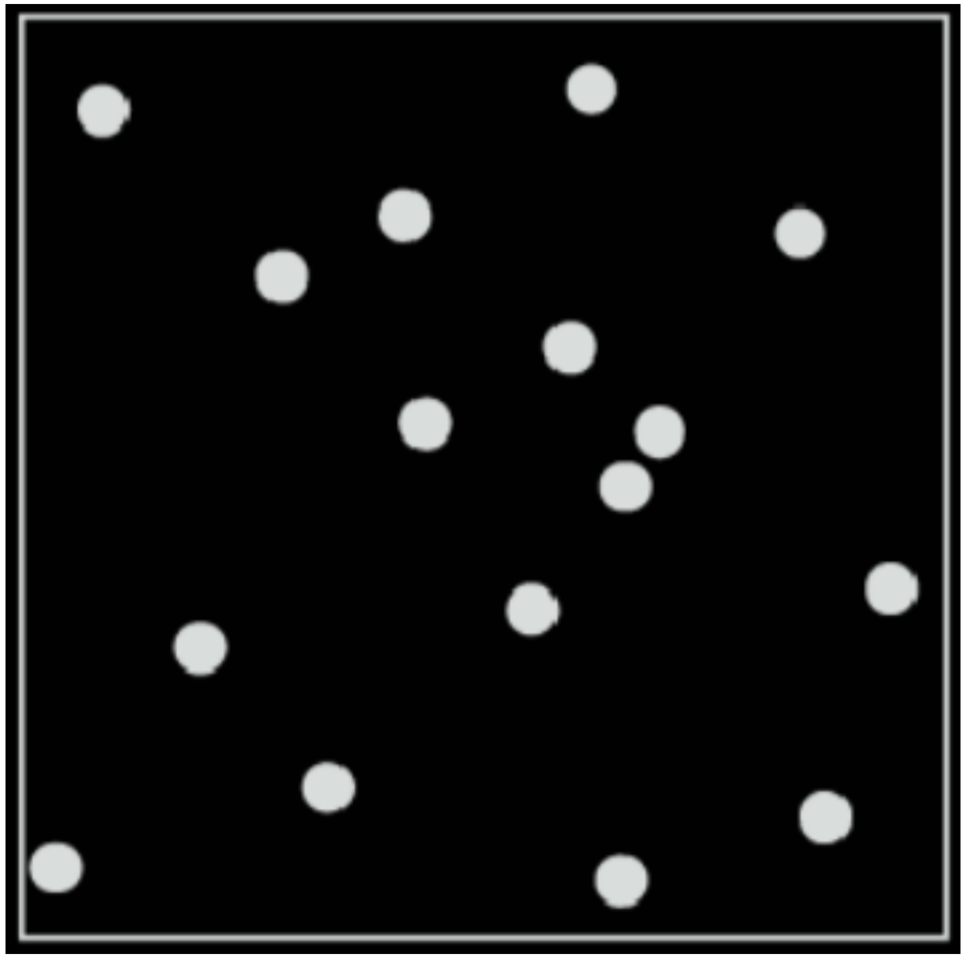
Antialiasing - Supersampling



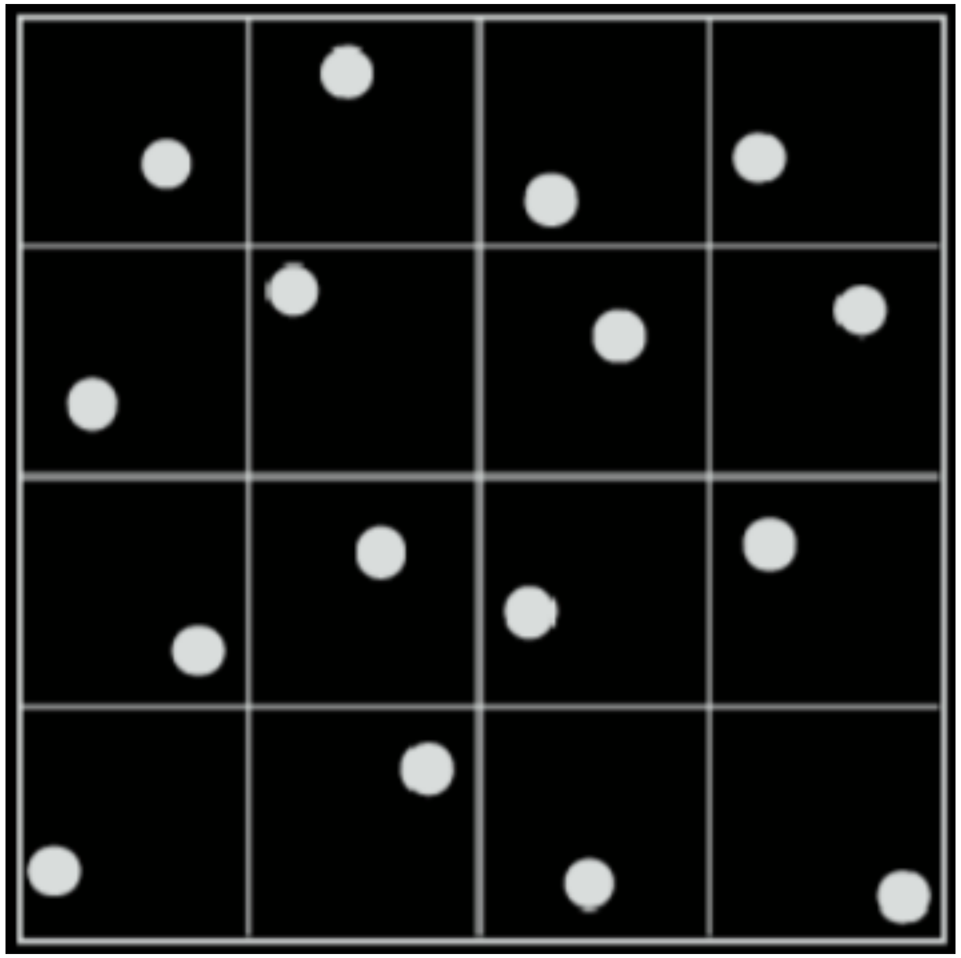
Antialiasing - Supersampling



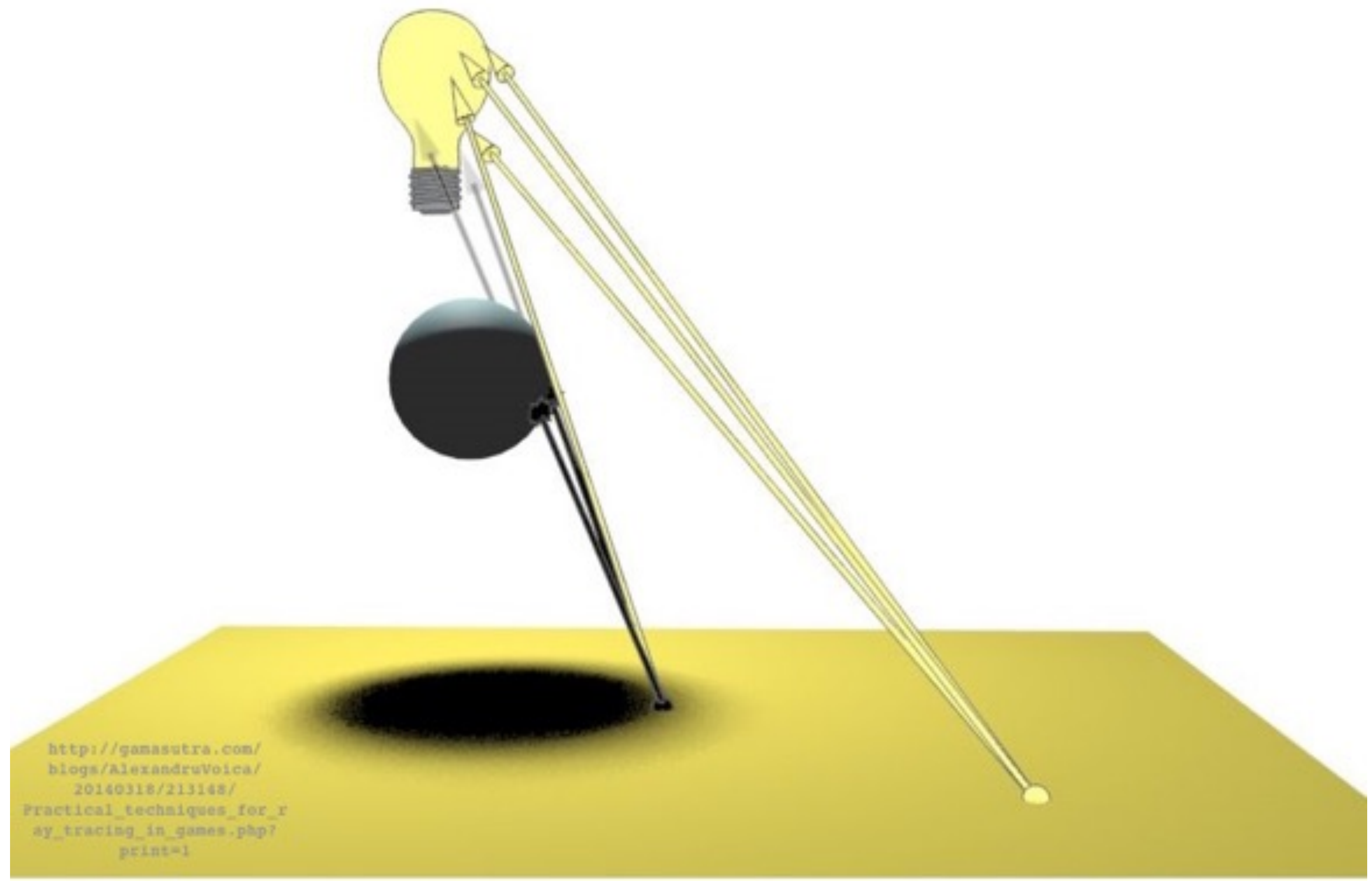
Antialiasing - Supersampling



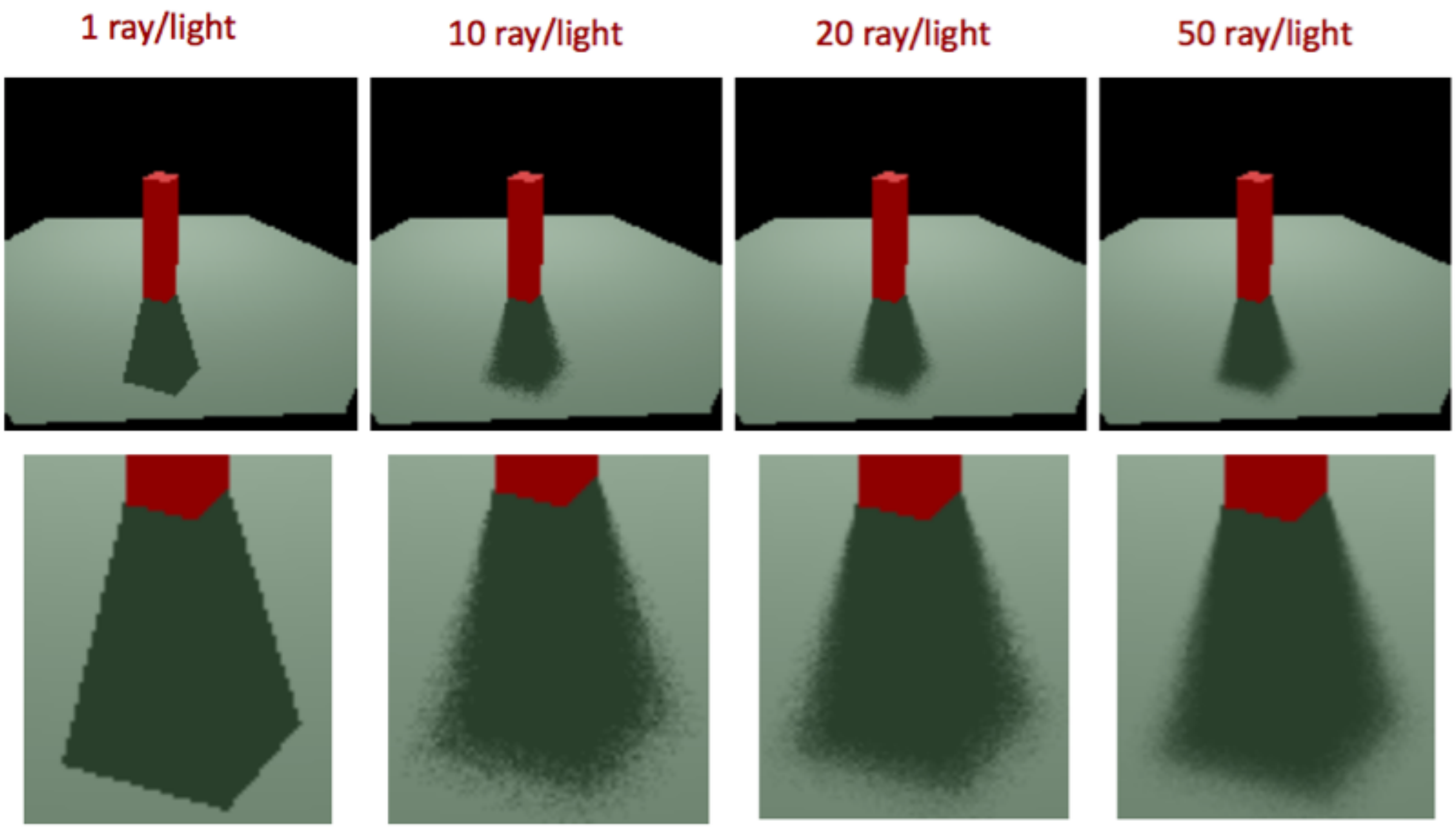
Antialiasing - Supersampling



Antialiasing - Supersampling



How many rays do you need?

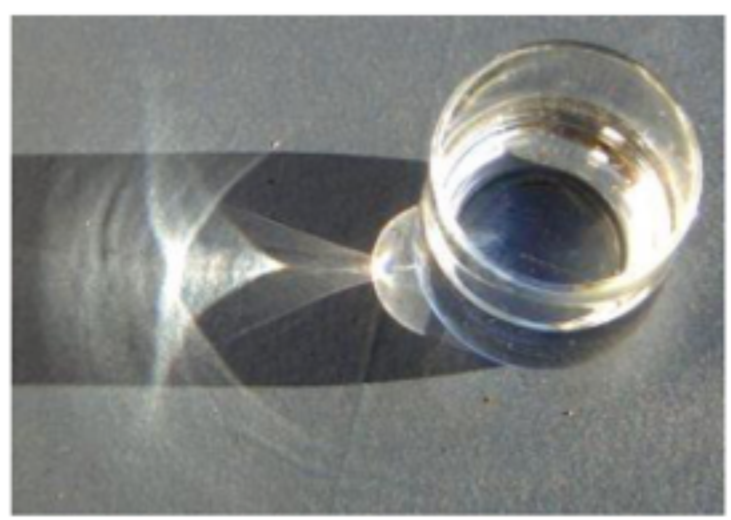


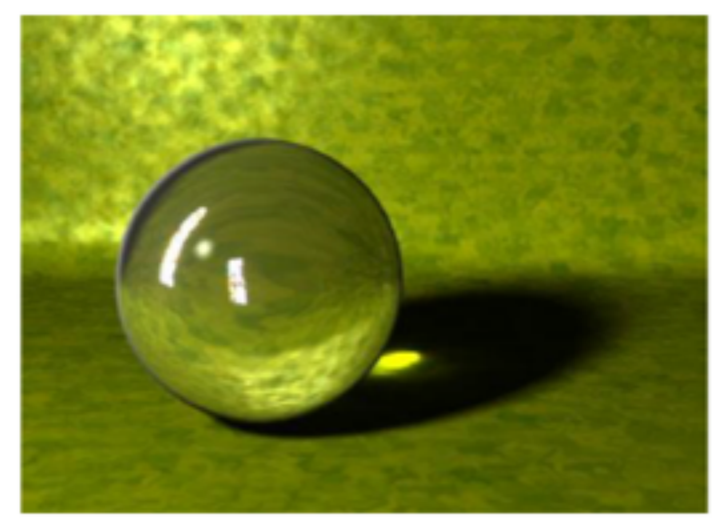
Images taken from http://web.cs.wpi.edu/~matt/courses/cs563/talks/dist_ray/dist.html

Ray Tracing Improvements: Image Quality

Backwards ray tracing

- Trace from the light to the surfaces and then from the eye to the surfaces
- “shower” scene with light and then collect it
- “Where does light go?” vs “Where does light come from?”
- Good for caustics
- Transport E – S – S – S - D – S – S – S - L





Ray tracing improvements: caustics

Ray Tracing Improvements: Image Quality

Cone tracing

- Models some dispersion effects

Distributed Ray Tracing

- Super sample each ray
- Blurred reflections, refractions
- Soft shadows
- Depth of field
- Motion blur

Stochastic Ray Tracing