Ray tracing

Trace rays out from light sources, bounce them around the scene, record the ones that enter the eye.

Highly impractical!*

Trace “light gathering rays” out from eye, record light that definitely contributes to the scene.

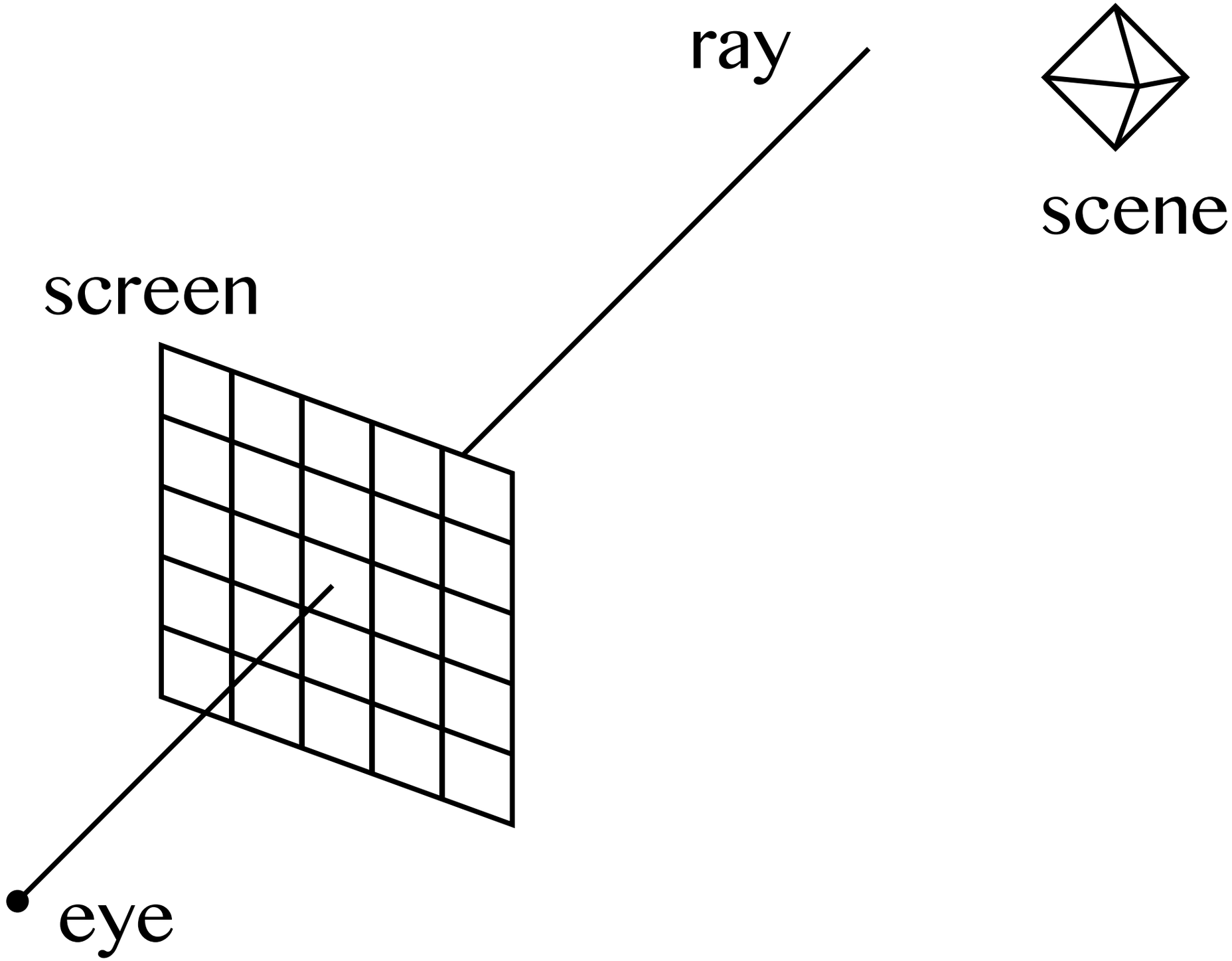
Backwards ray tracing

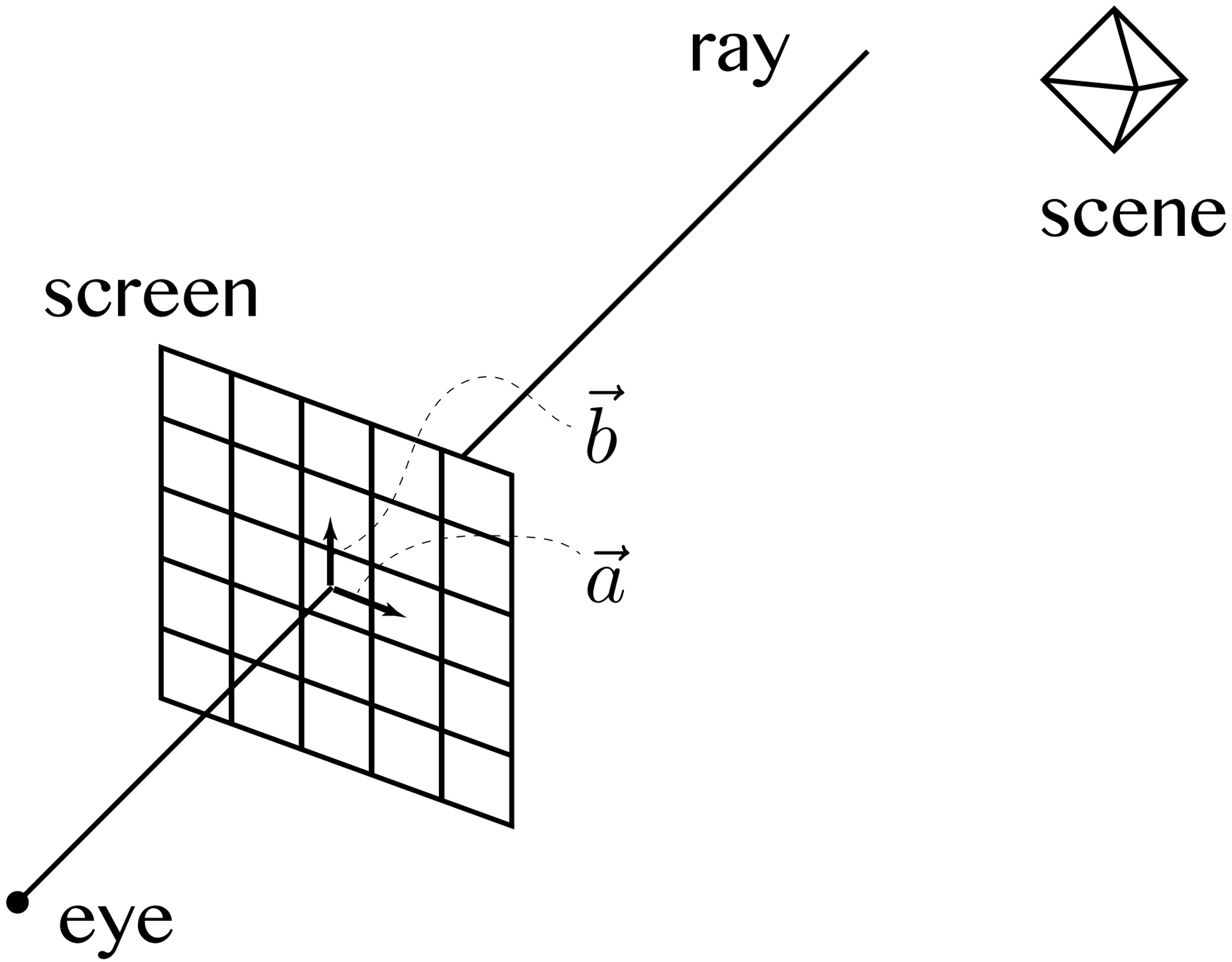
```
For every pixel (x,y) in scene:
  r = makePrimaryRay( x, y )
  i = scene.intersect( r )
  writePixel( x, y, illuminate( i ) )
```

Generating primary rays

Forget about P and V — we're not going to use them!

Imagine image plane floating in front of camera, compute "pixel-to-pixel vectors" on it.


ray
scene
screen
eye


ray
scene
screen
$\vec{b}$
$\vec{a}$
eye

Intersection computation

Like clipping!

Ray: $r(t) = E + t\vec{d}$

Primitive: $f(Q) = 0$

$$f(r(t)) = 0$$

Ray-Sphere

$$f(Q) = ||Q - C|| - r$$

Substituting $f(r(t)) = 0$ yields a quadratic equation in *t*.

Many other algebraic surfaces work similarly.

Ross on flickr

ShiftedReality

Ray-Triangle

Intersect ray with the support plane of the polygon.

Then check whether the point of intersection lies within the polygon.

Ray-Box

Could treat box as just another mesh, but there are more elegant approaches.

Treat cube as intersection of three “slabs”, intersect ray with each slab in turn.

Debugging

Visualize per-pixel behaviour

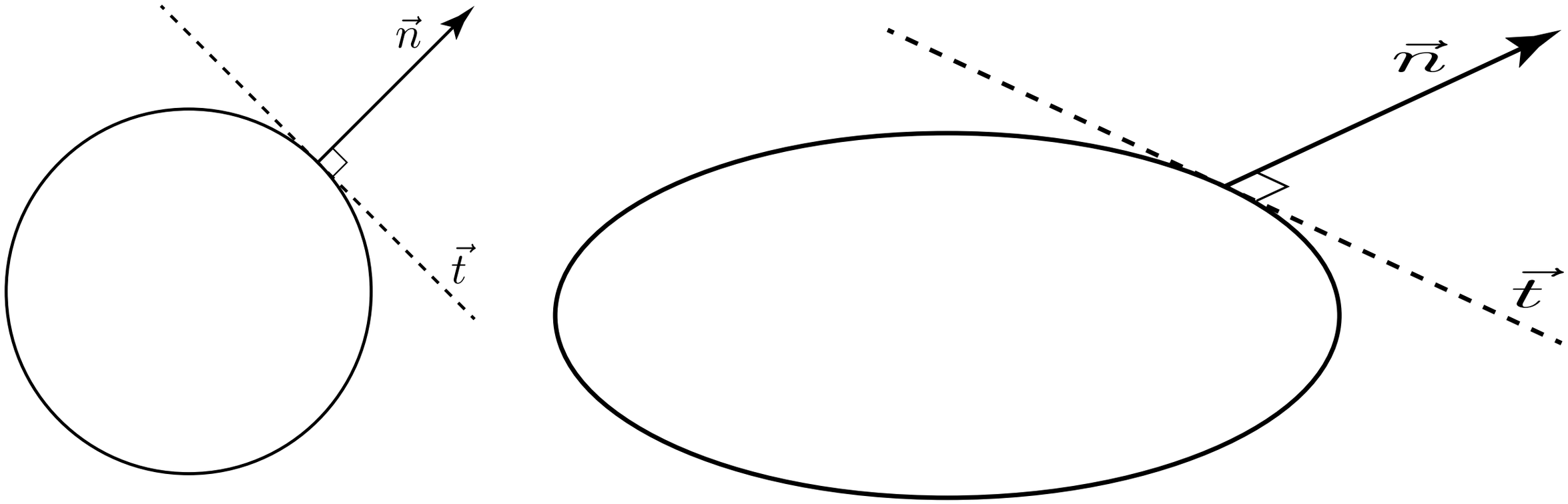
Isolate a single pixel

Ray tracing and hierarchical modelling

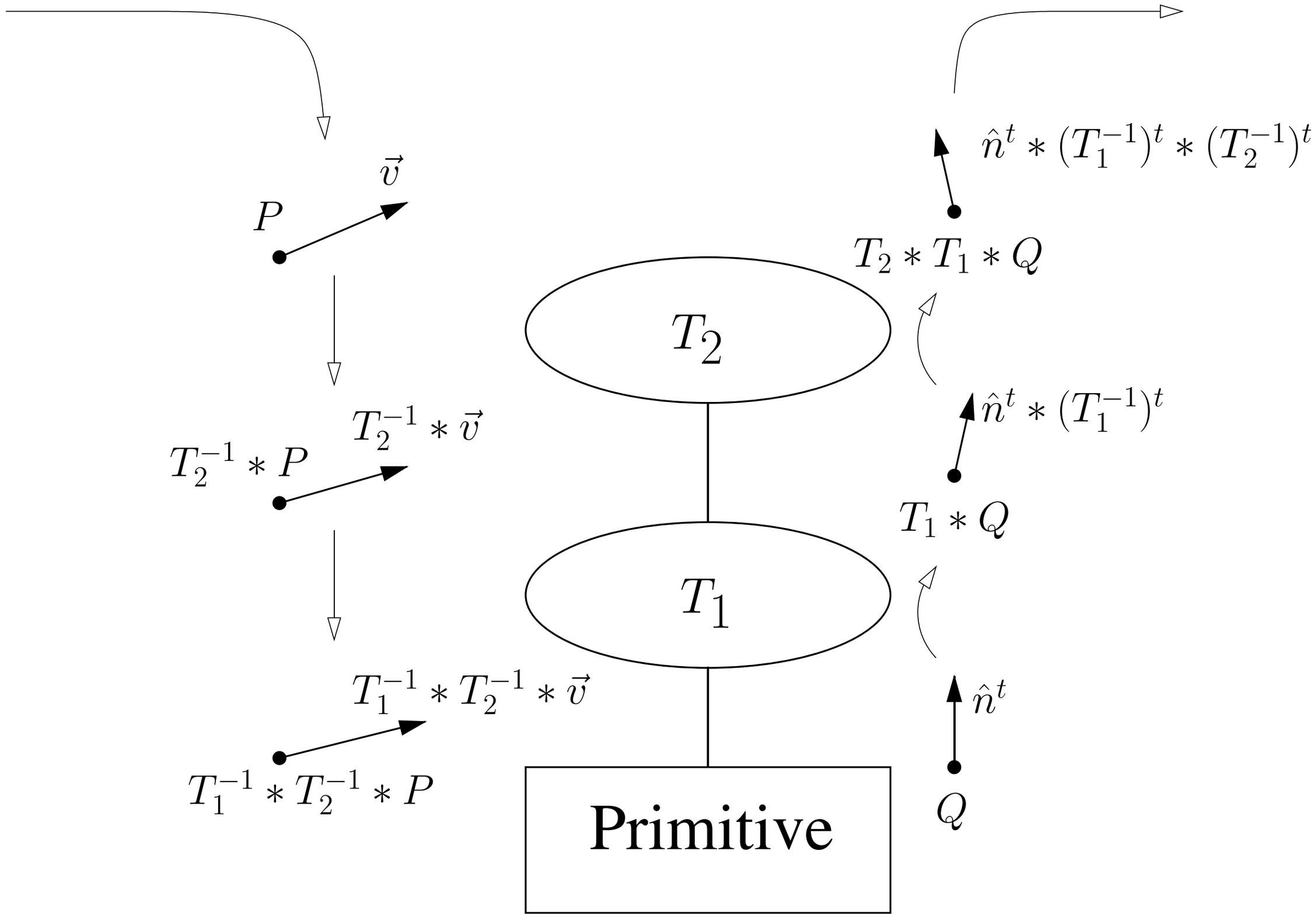
Transforming arbitrary primitives into world coordinates can be difficult.

So transform the ray into modelling coordinates, then transform the intersection result back into world coordinates.

But take special care with normals!



Normals don’t transform nicely (especially nonuniform scaling). *Tangents* do.


$\vec{v}$
P
$T_2^{-1} * \vec{v}$
$T_2^{-1} * P$
$T_1^{-1} * T_2^{-1} * \vec{v}$
$T_1^{-1} * T_2^{-1} * P$
T_2
T_1
Primitive
$\hat{n}^t * (T_1^{-1})^t * (T_2^{-1})^t$
$T_2 * T_1 * Q$
$\hat{n}^t * (T_1^{-1})^t$
$T_1 * Q$
$\hat{n}^t$
Q

Illumination

Ray-scene intersection yields:

- World position of intersection
- World normal vector at intersection
- Material info of intersected primitive

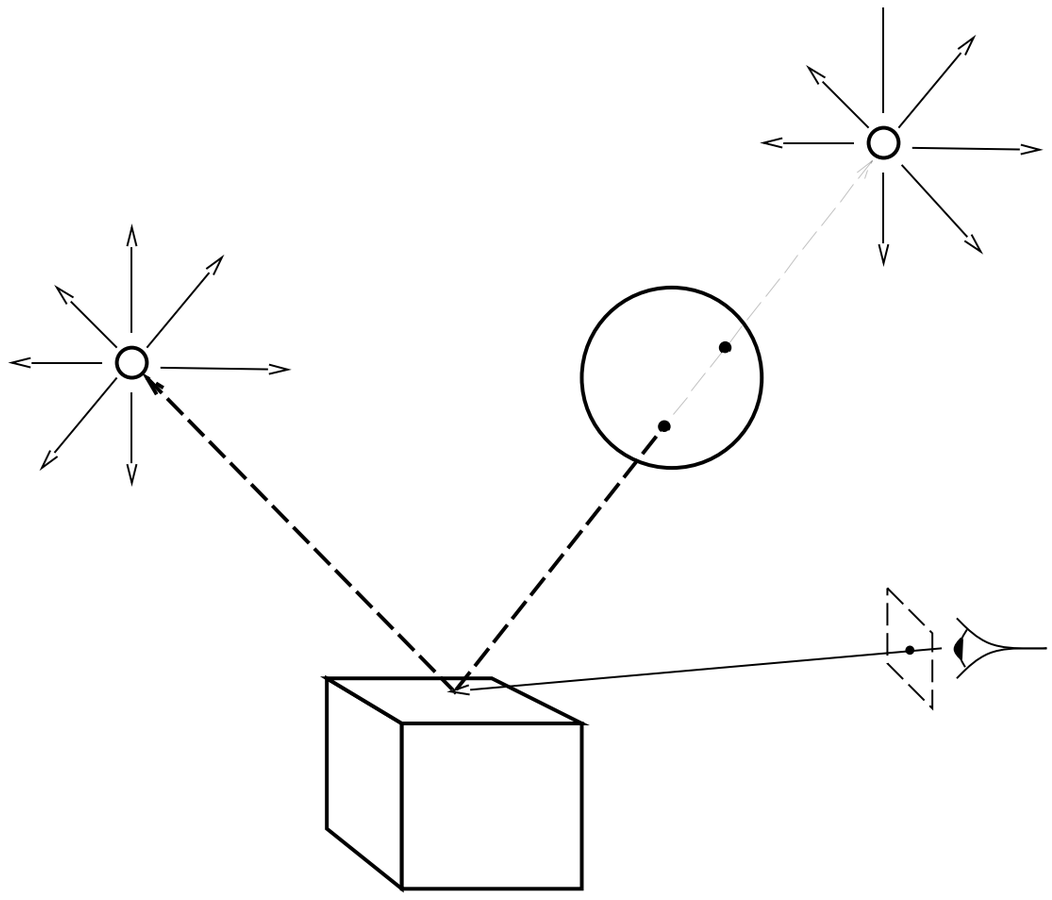
…so perform (Phong) illumination there.

Illumination

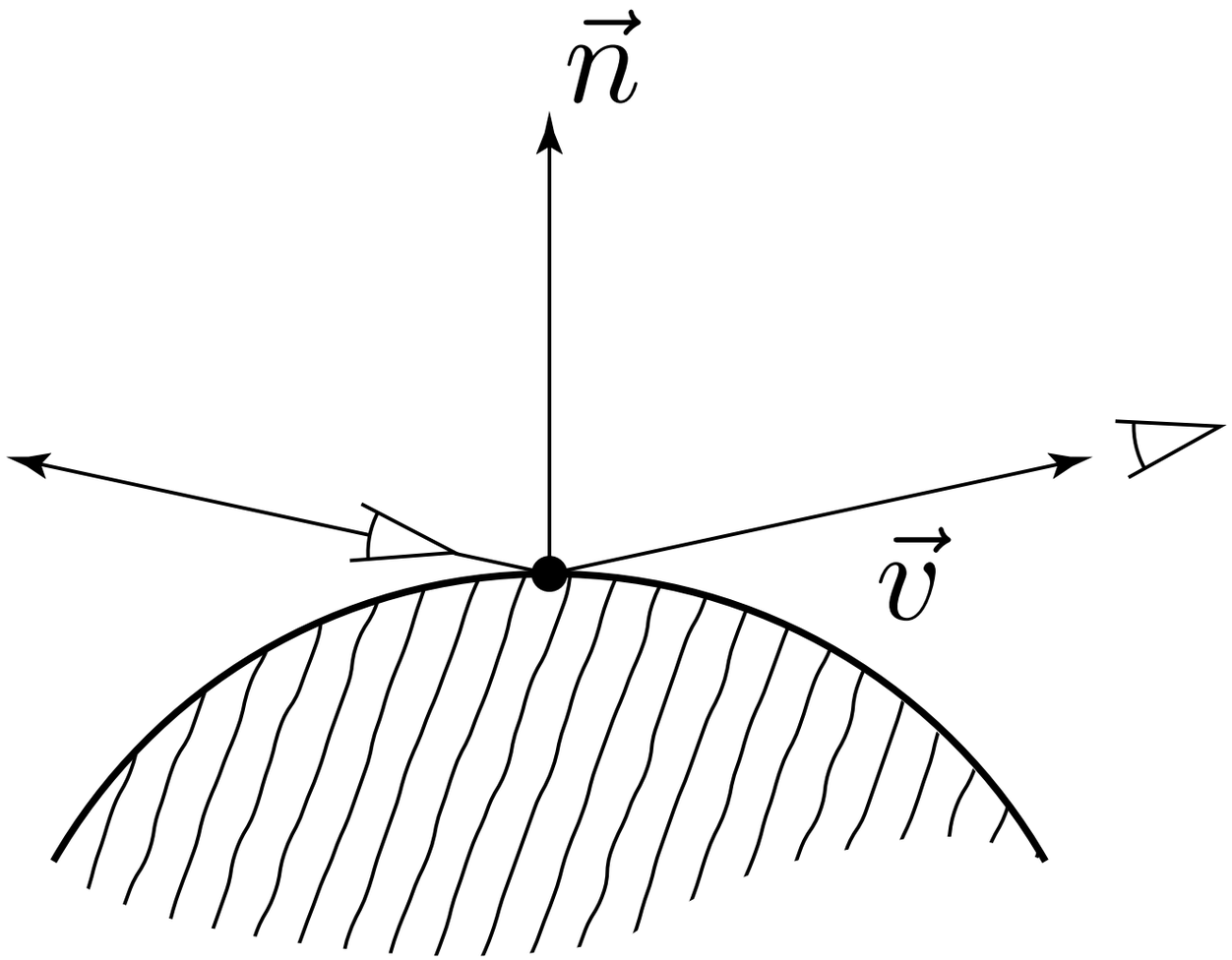
But a given light source isn't necessarily visible from the point of intersection.

So trace a *shadow ray* back to the light source!

If you hit any other objects before the light, the object is in shadow.

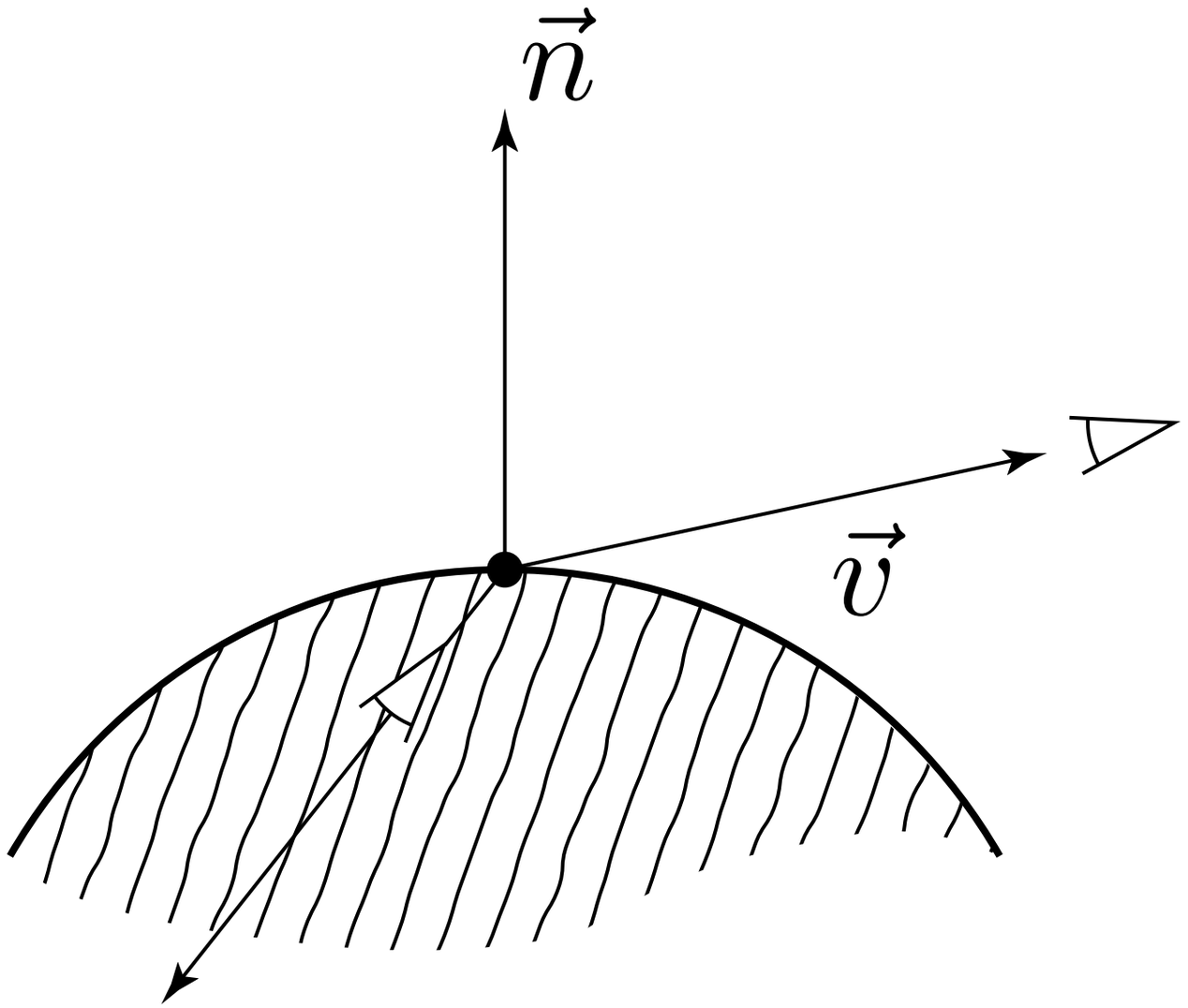


More secondary rays

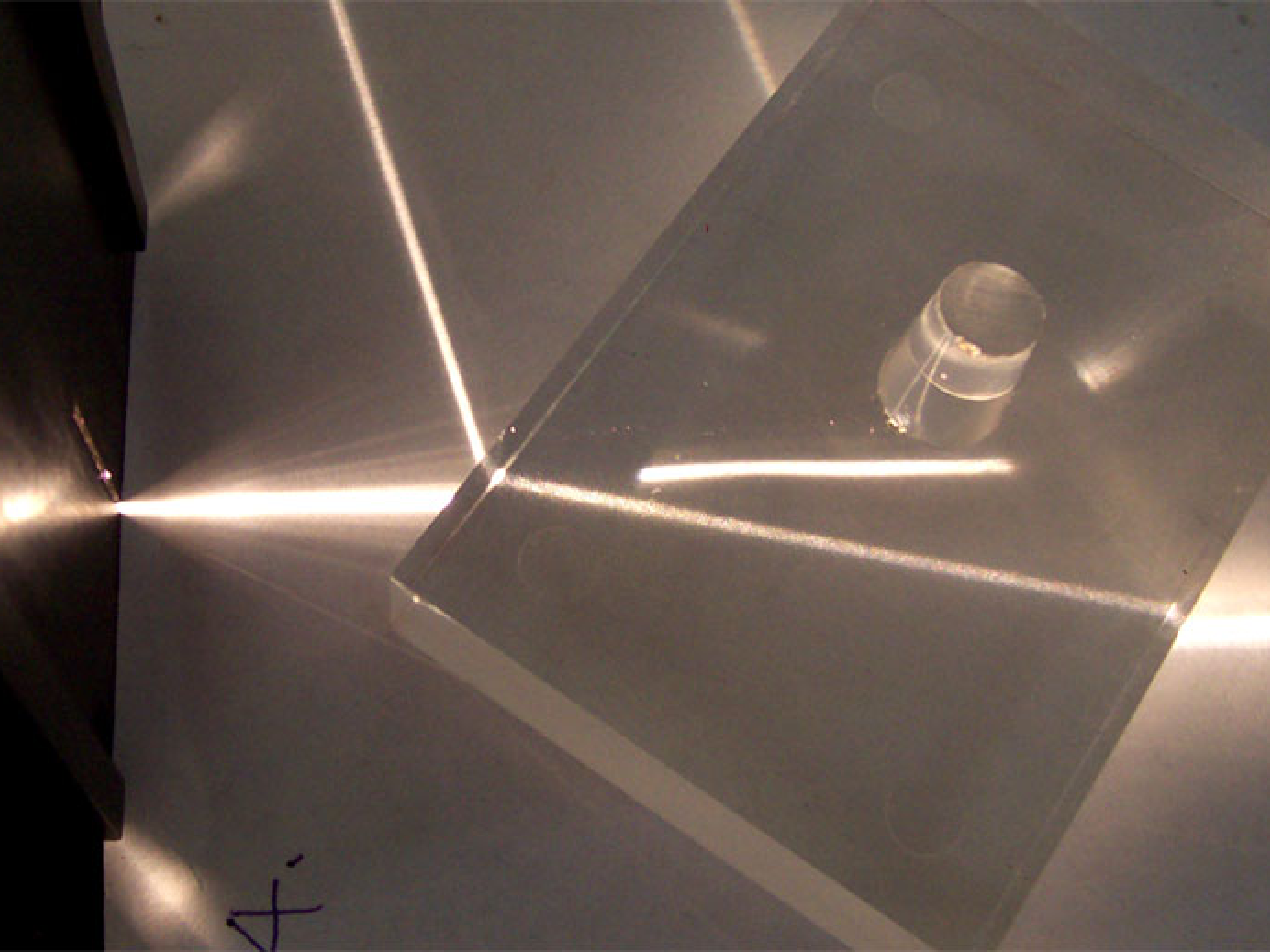


Generate a recursive ray in the direction of mirror *reflection*, add its (weighted) contribution to result.

More secondary rays



Generate a recursive ray in the direction of specular *refraction*, add its (weighted) contribution to result.



Total internal reflection

Recursive ray tracing

How many levels of recursion?

- Fixed upper limit on recursion
- Threshold for contribution to final scene

Acceleration

Preprocessing time

Runtime overhead

Savings

Obvious speedups

Smarter coding

“fail fast” intersection tests

Low-level tricks (byte alignment, cache coherency, etc.)

Organizing primitives

Bounding volumes

Bounding volume hierarchies

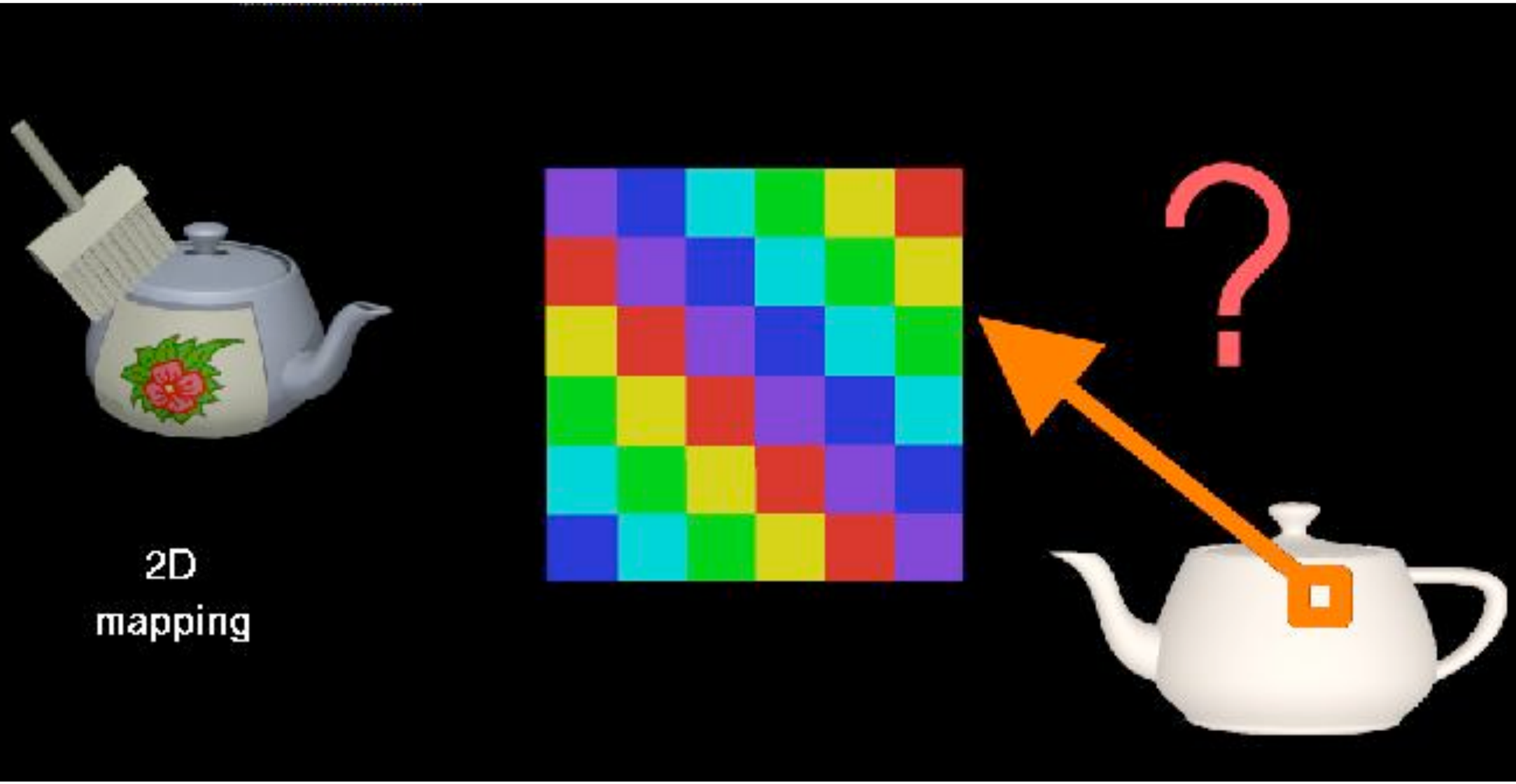
Protip: use special-purpose intersection tests

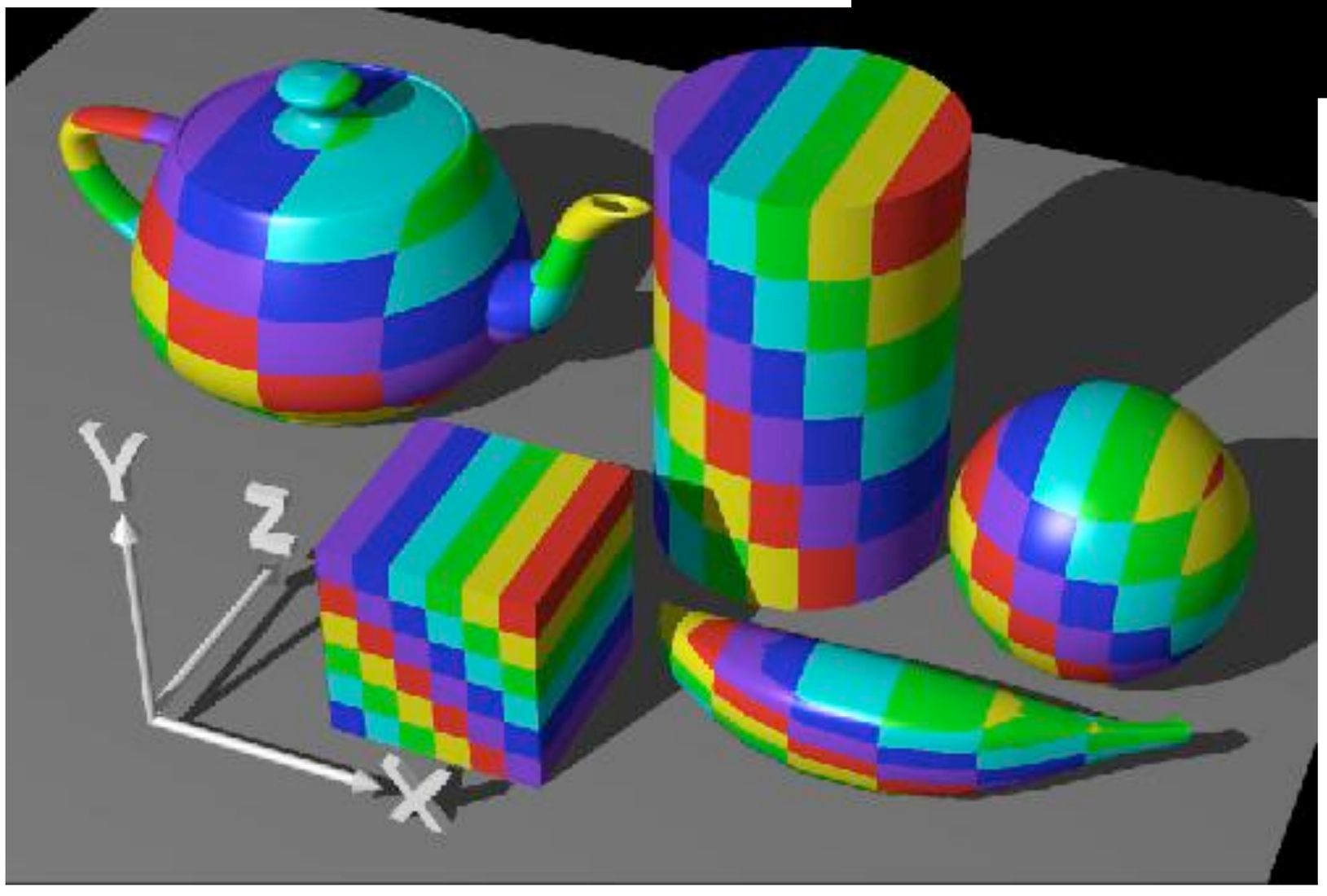
Organizing space

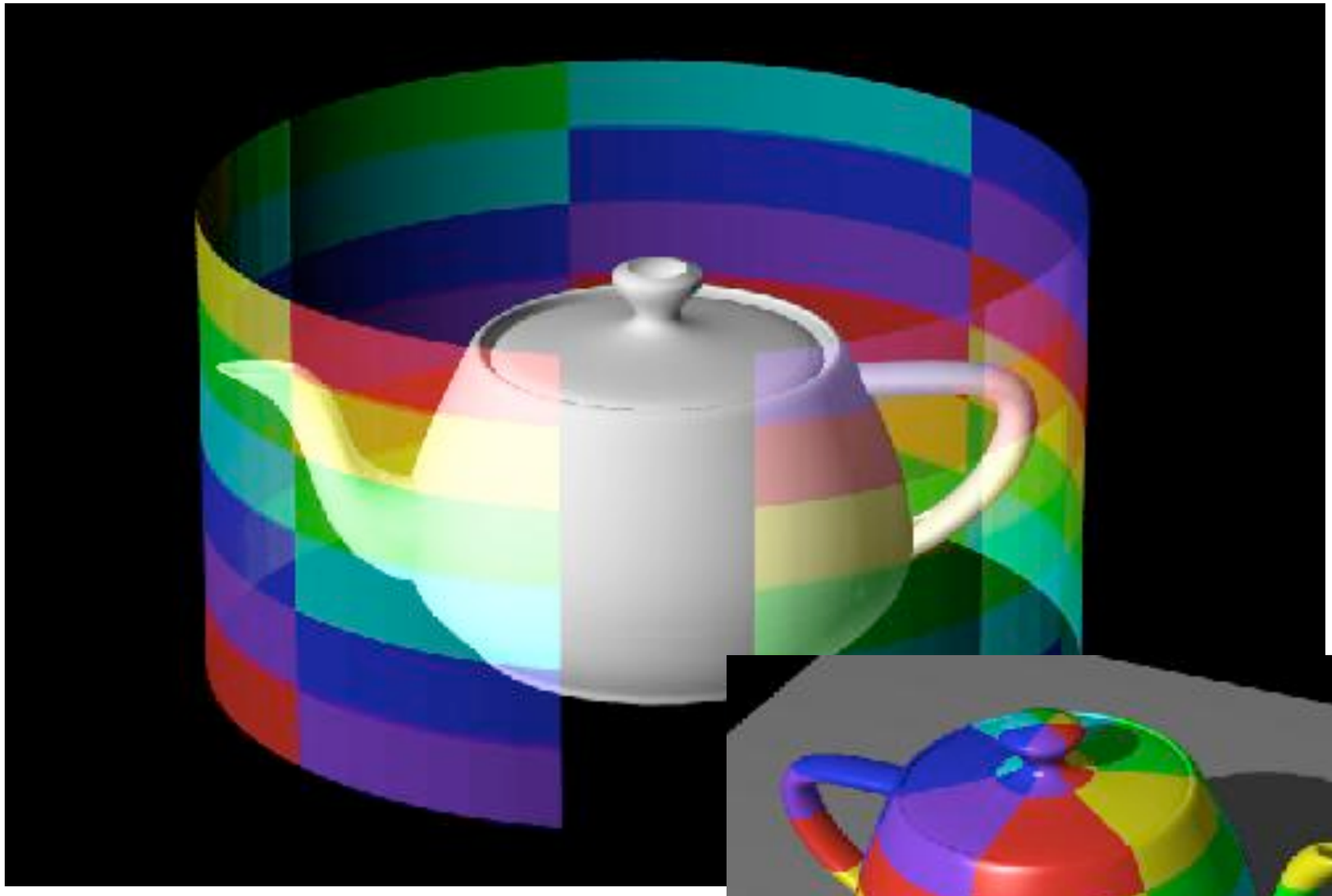
Uniform spatial subdivision

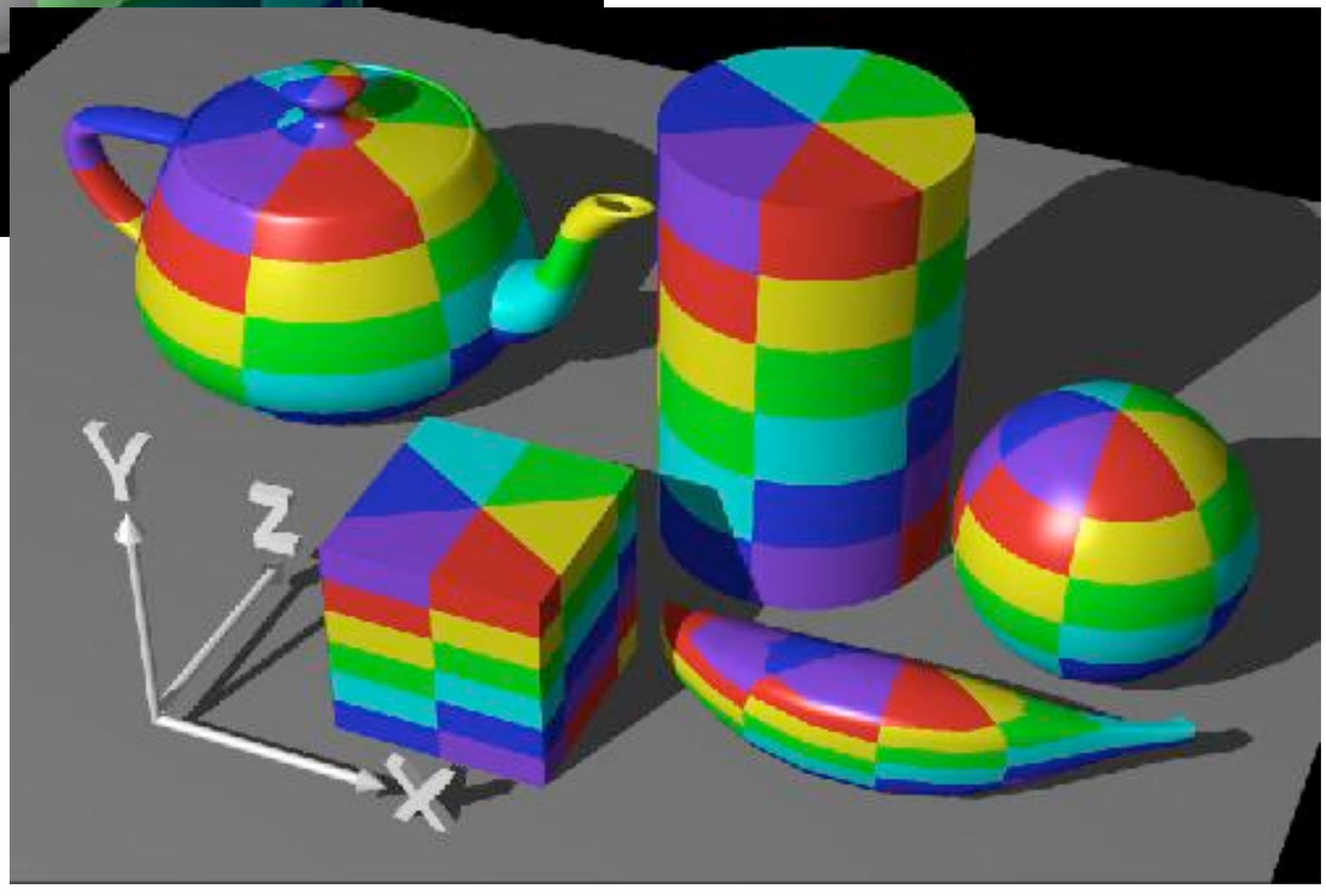
Non-uniform spatial subdivision

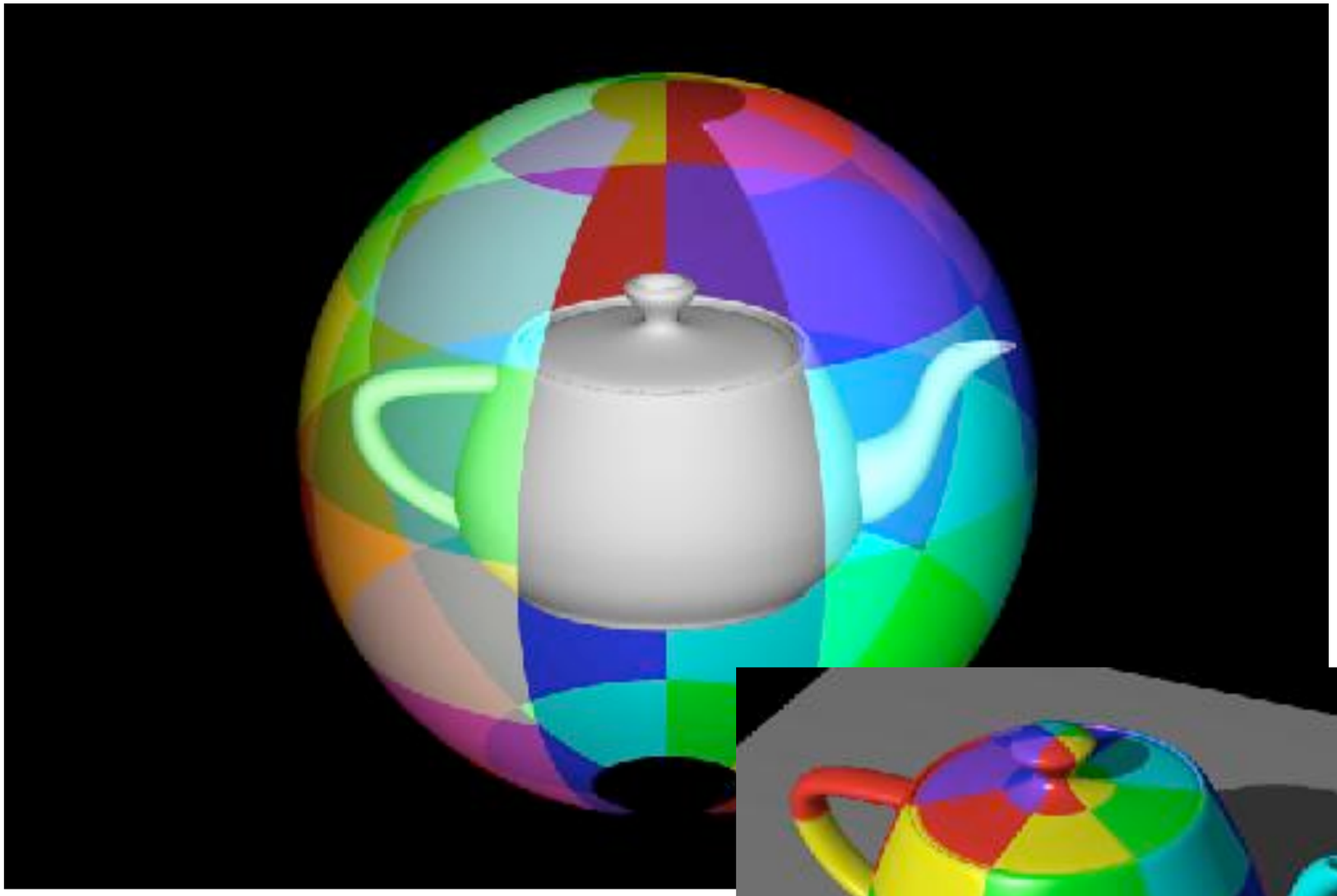
Need a more complicated ray traversal method.

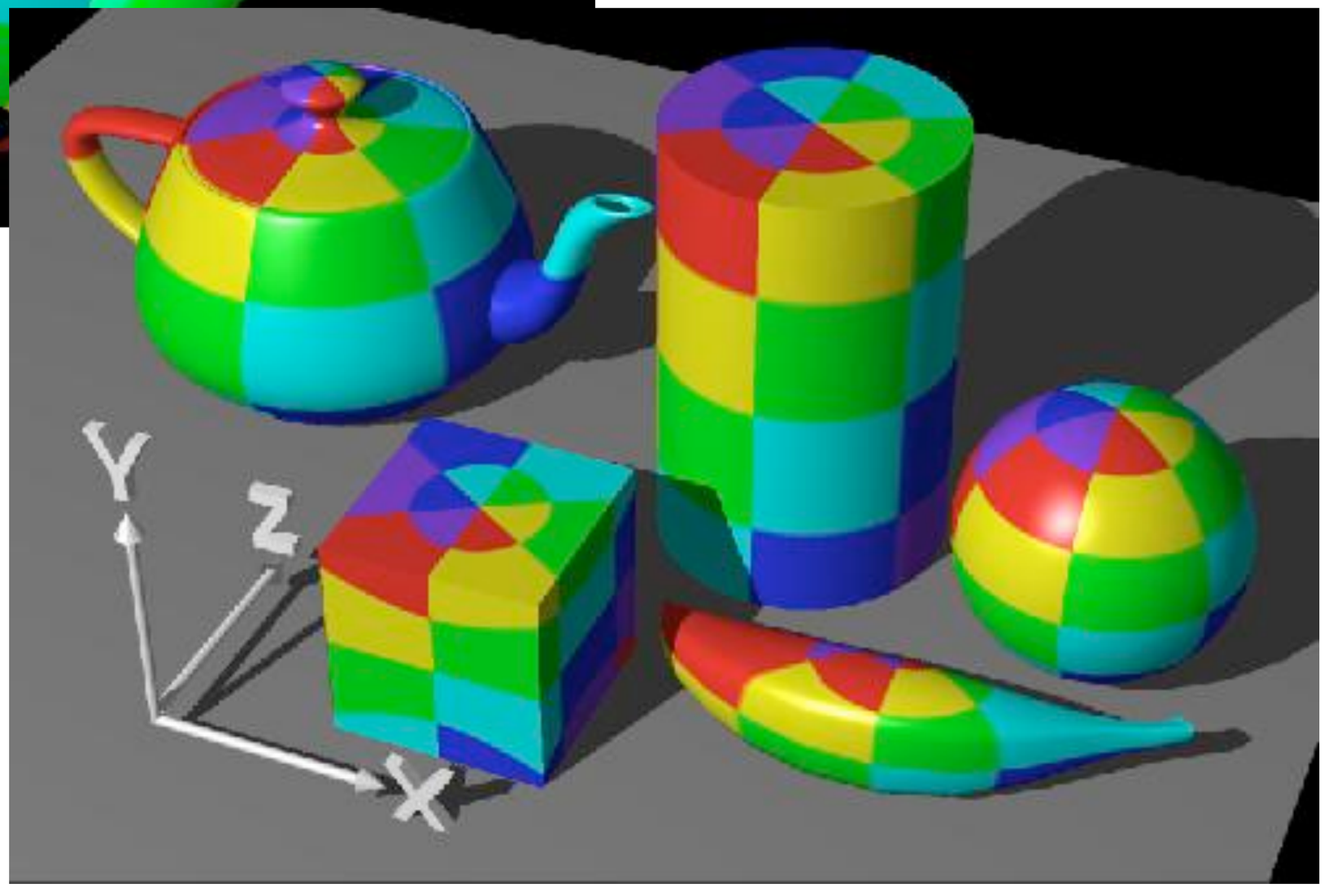

2D
mapping
?

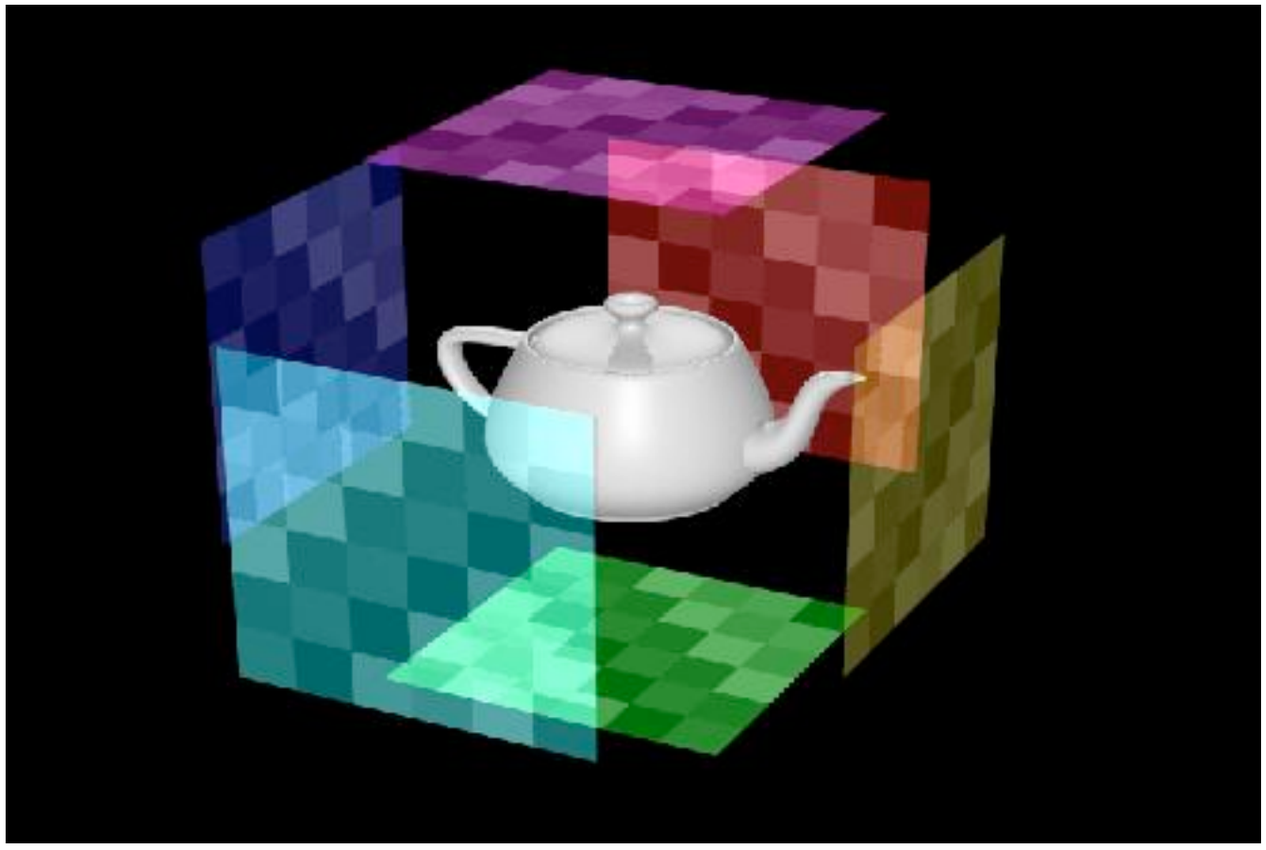

Y
Z
X
Y
Z
X

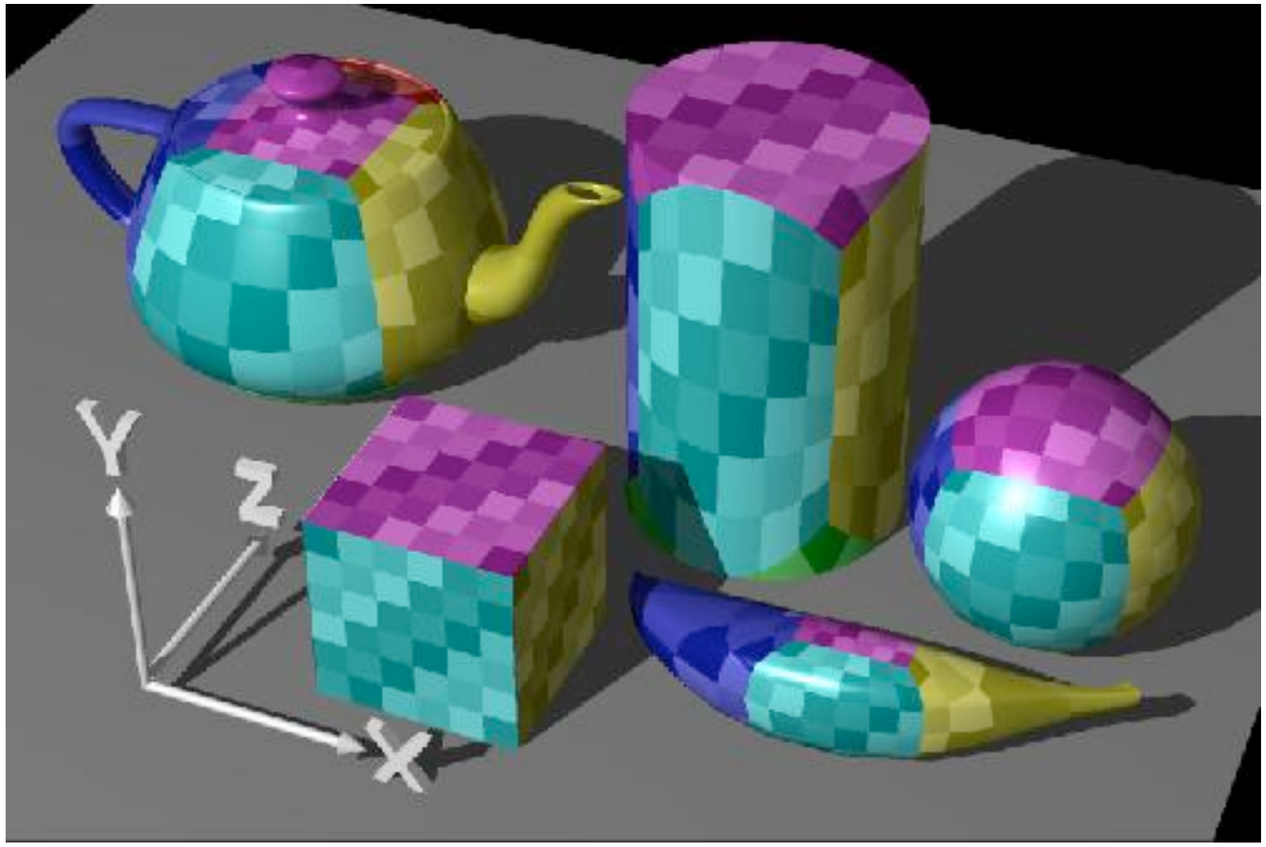


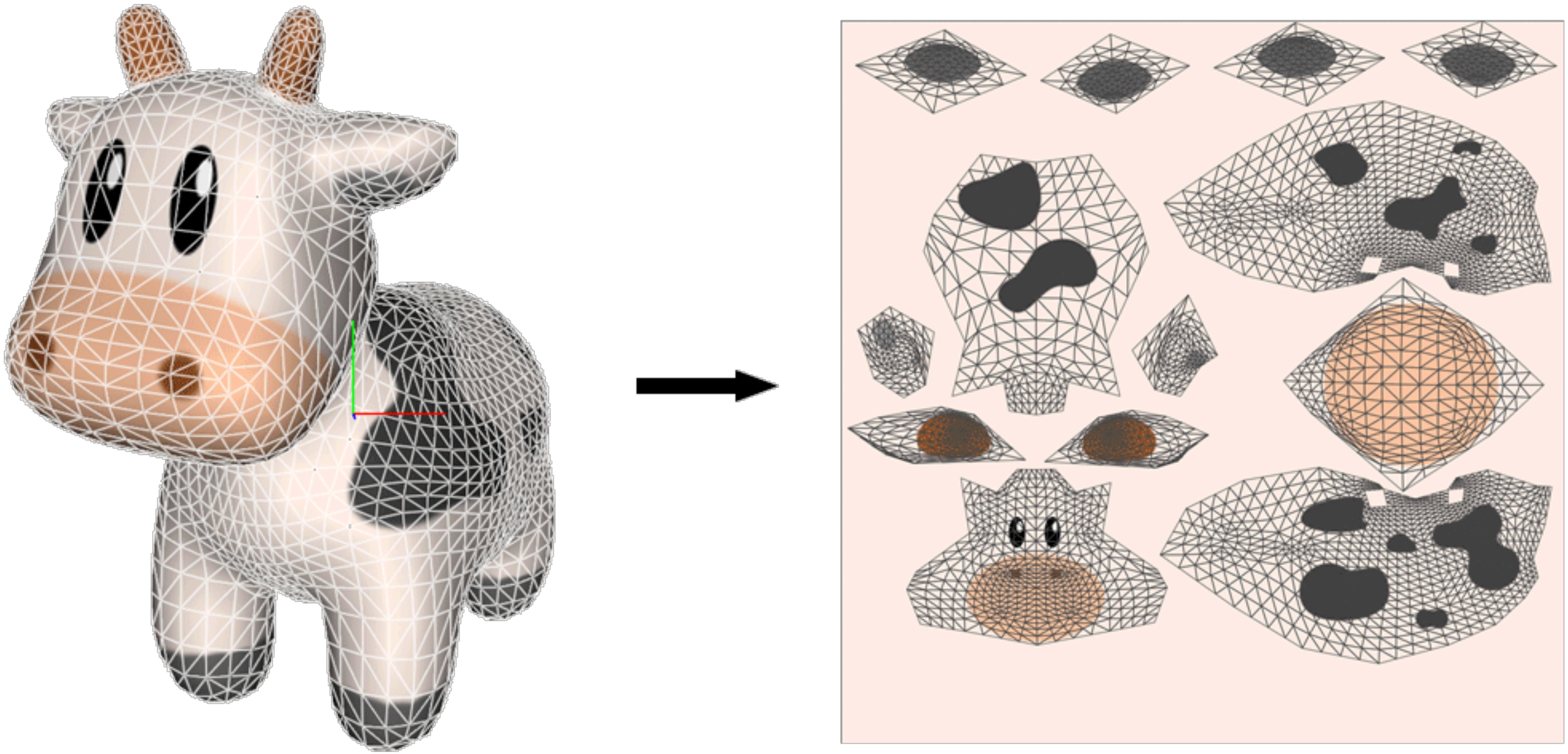

Y
Z
X



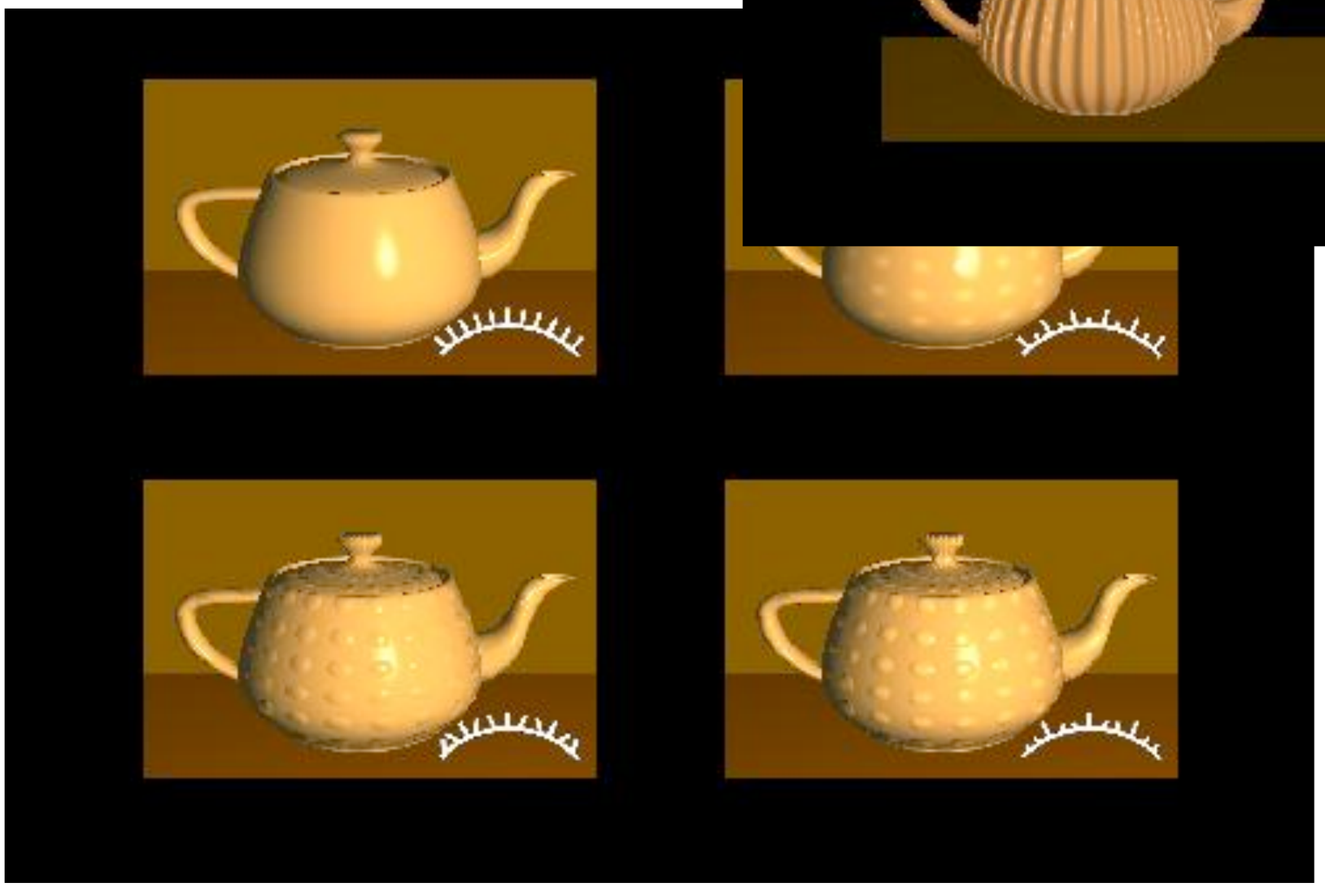

Y
Z
X




Y
Z
X

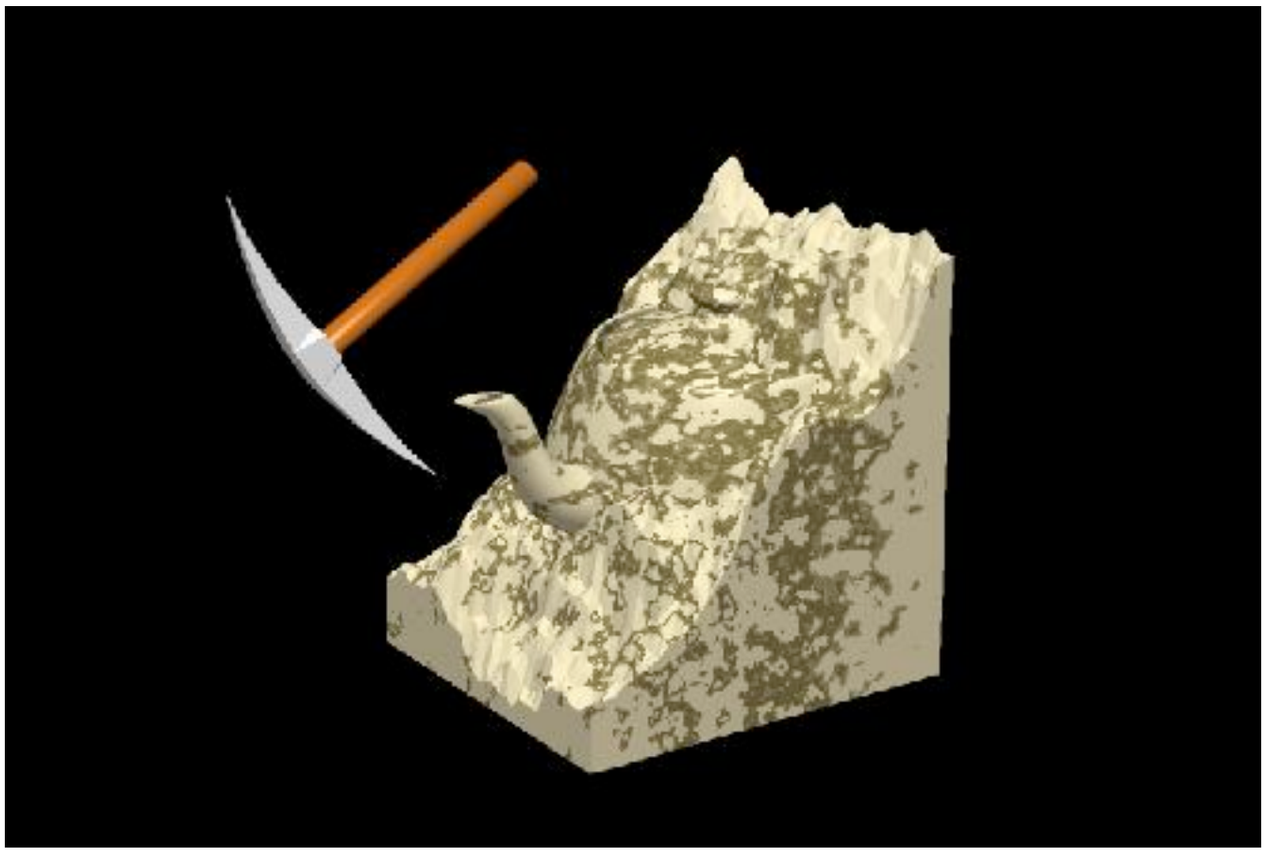


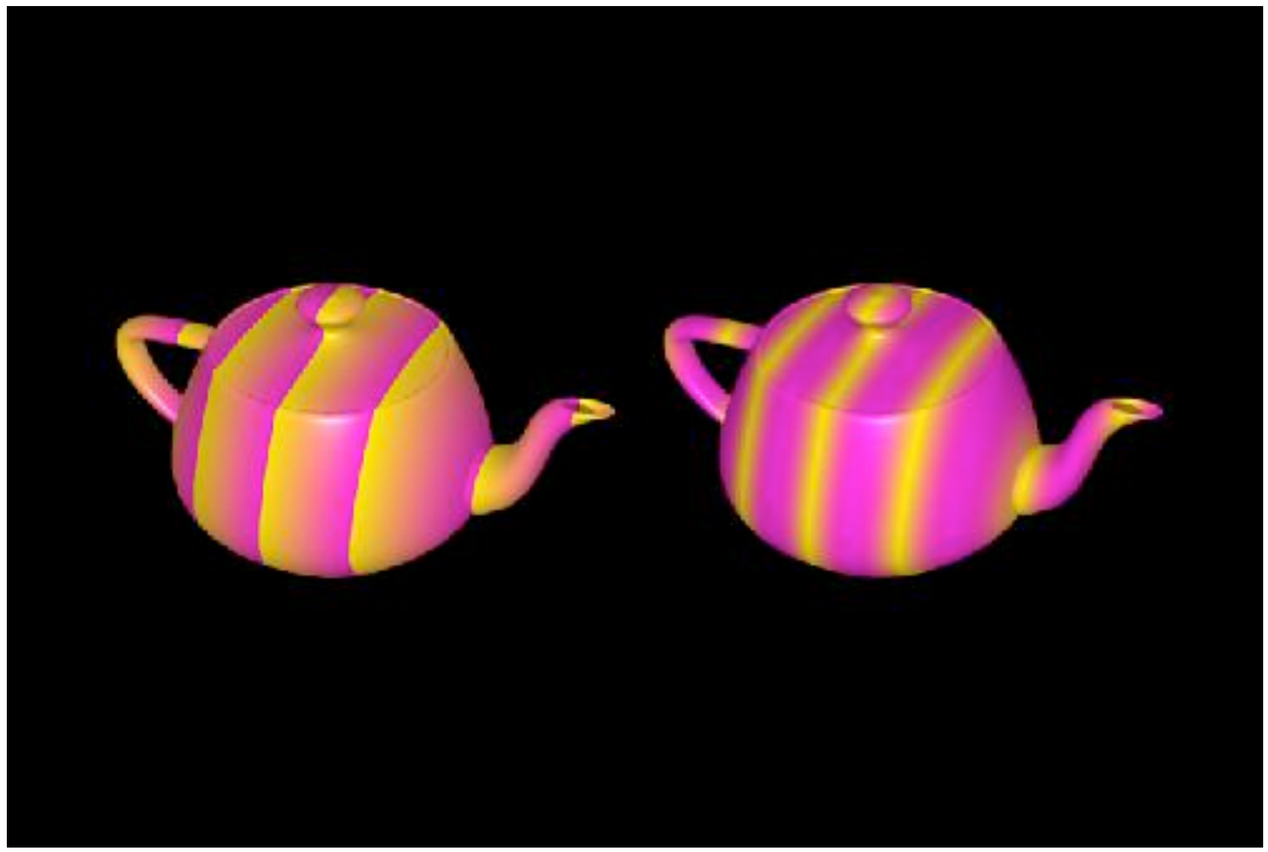
metalbyexample.com

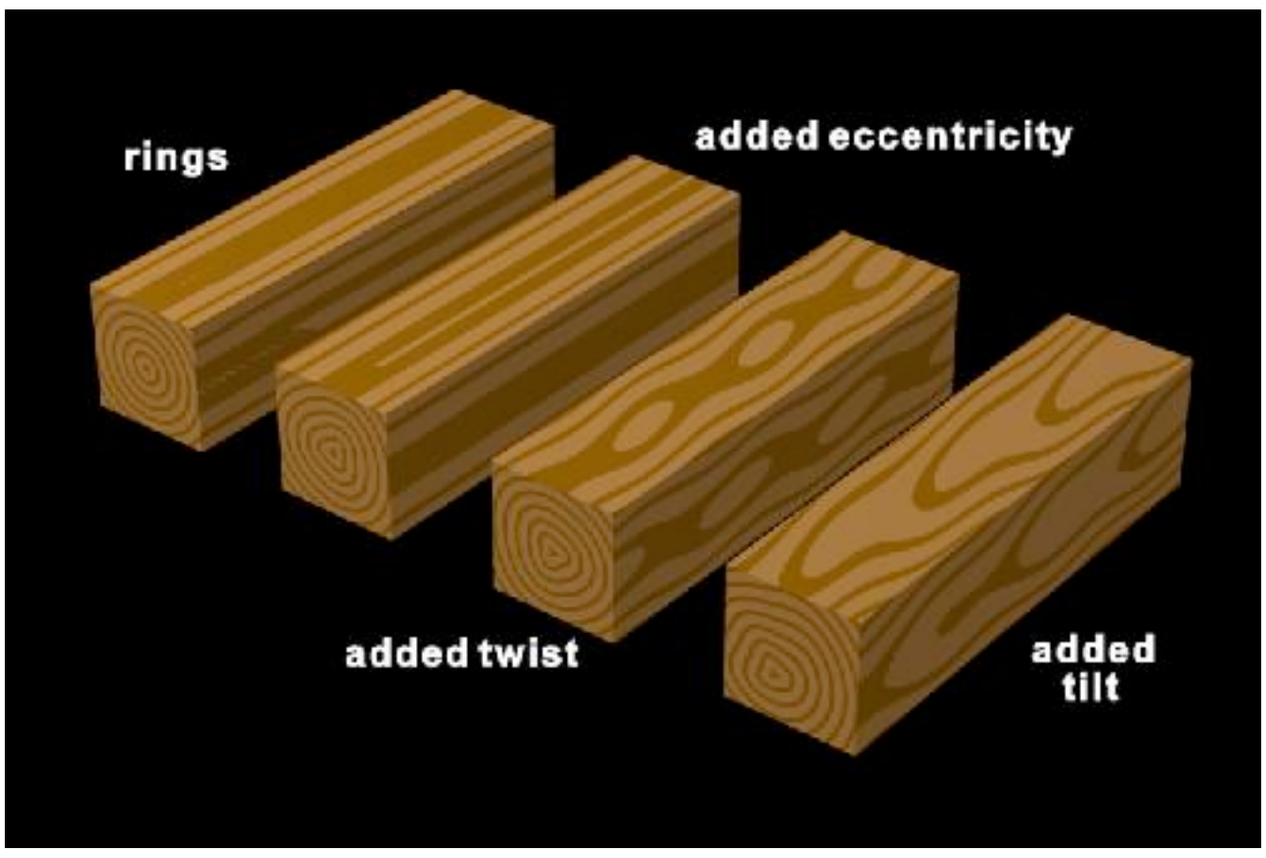


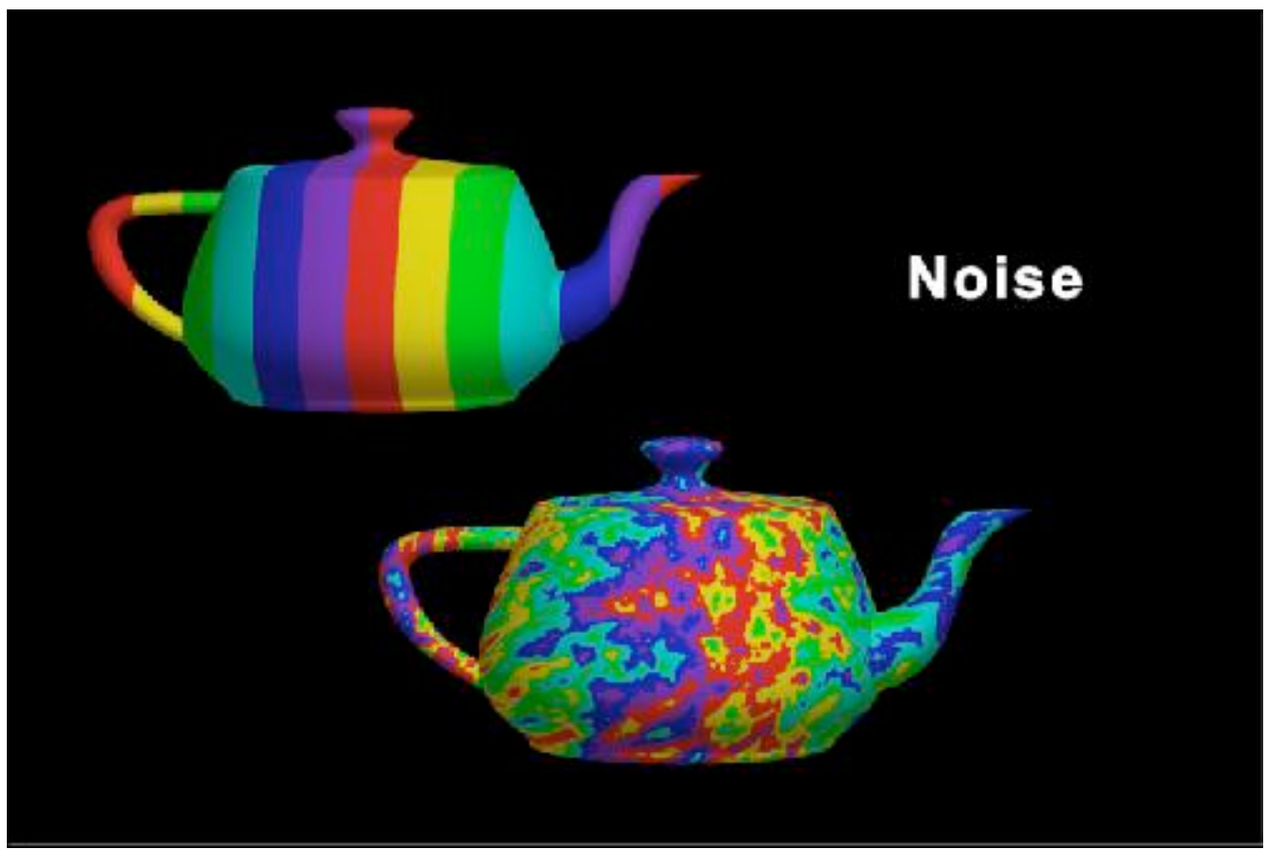
Bump mapping

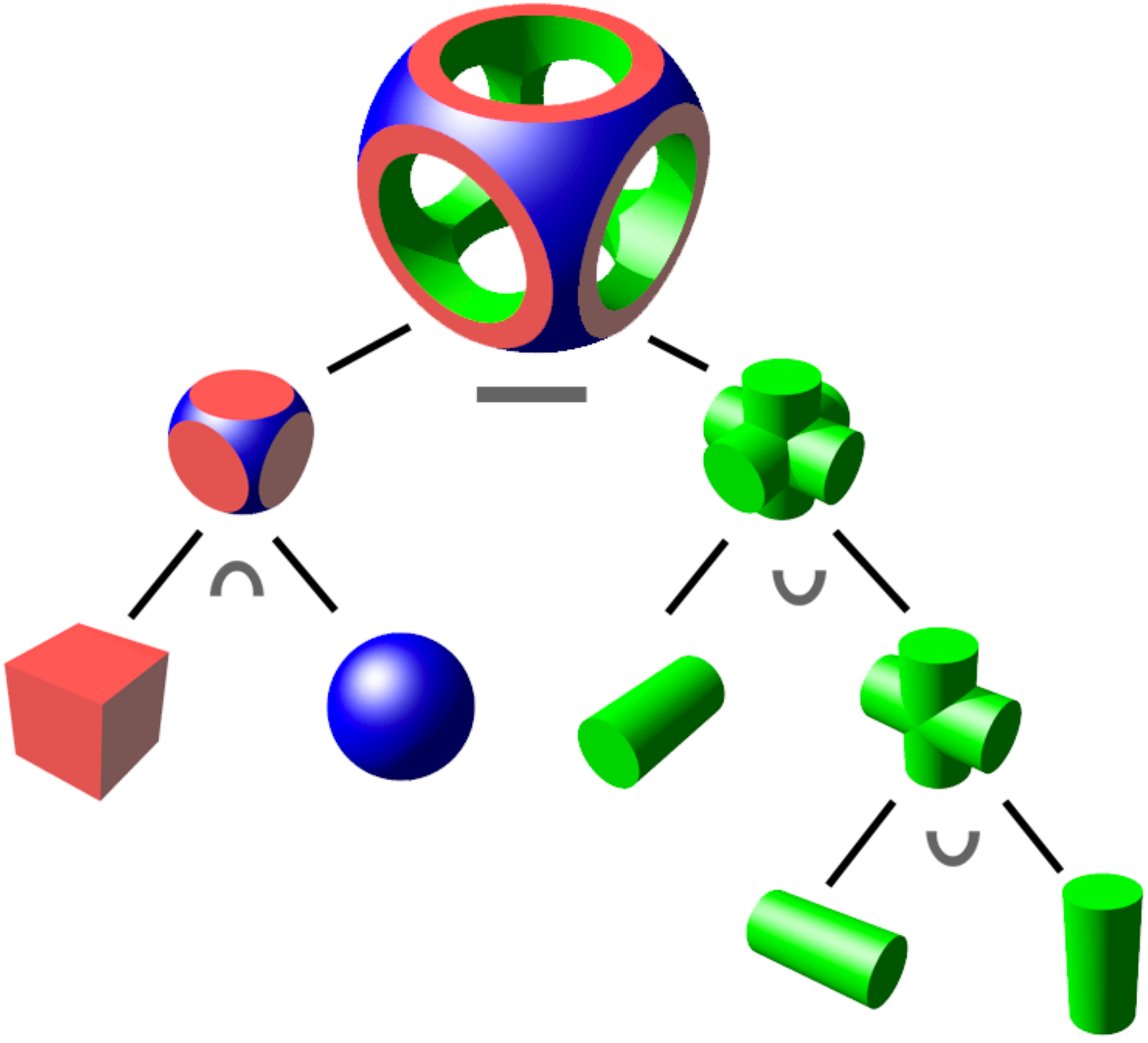
Displacement mapping

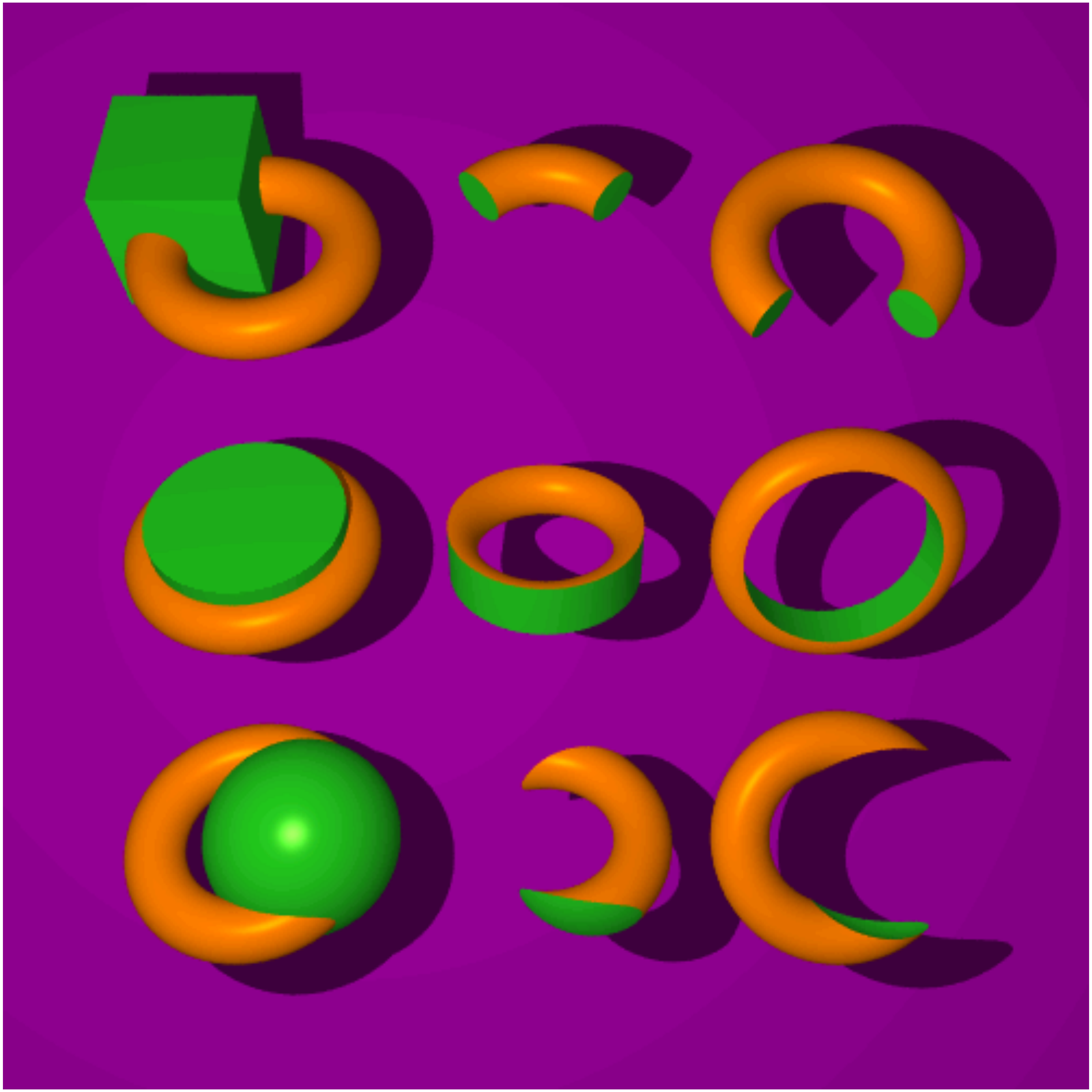





rings
added eccentricity
added twist
added
tilt


Noise





The “Little Dipper”