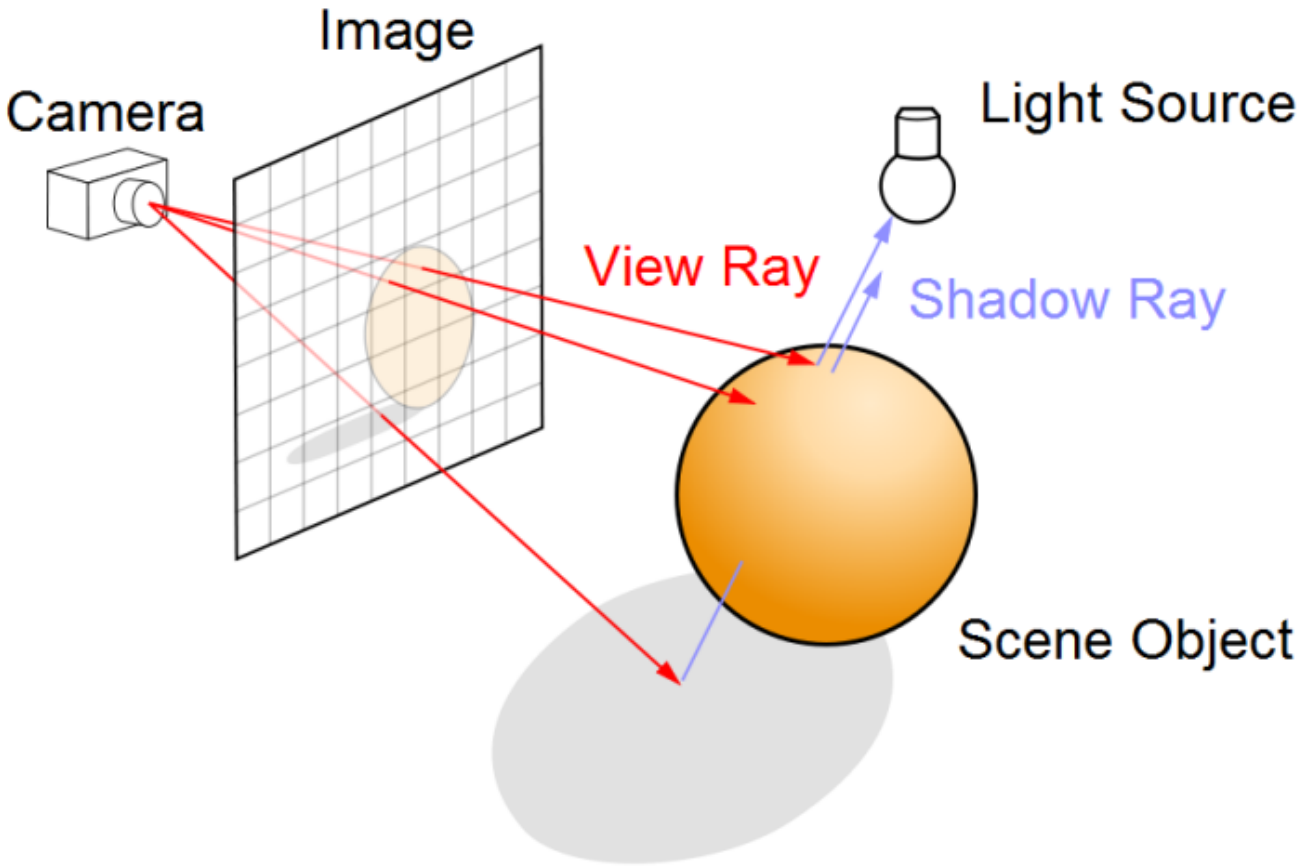
GRAPHICS GROUP

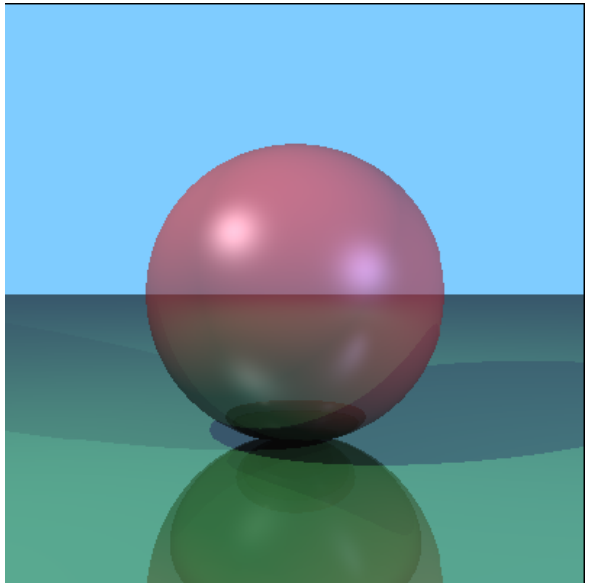
Computer Graphics and Image Processing Ray Tracing III

Part 1 – Lecture 11

Today's Outline

- Ray Tracing Reflections
- Ray Tracing Transformed Primitives
- Speeding Up Ray Tracing



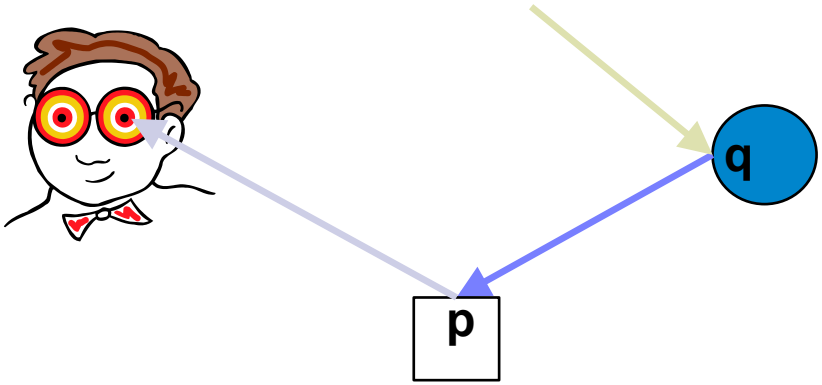


RAY TRACING REFLECTIONS

Diagram thanks to Henrik

Ray Tracing Reflections

Idea: the color of a point is influenced by the color that the ray carries over from the previous reflection



Ray is reflected at **q** (blue sphere) before being reflected at **p** (white box)
→ ray has bluish color when it hits the box

Reflectivity: fraction of incident radiation reflected by a surface (between 0 and 1)

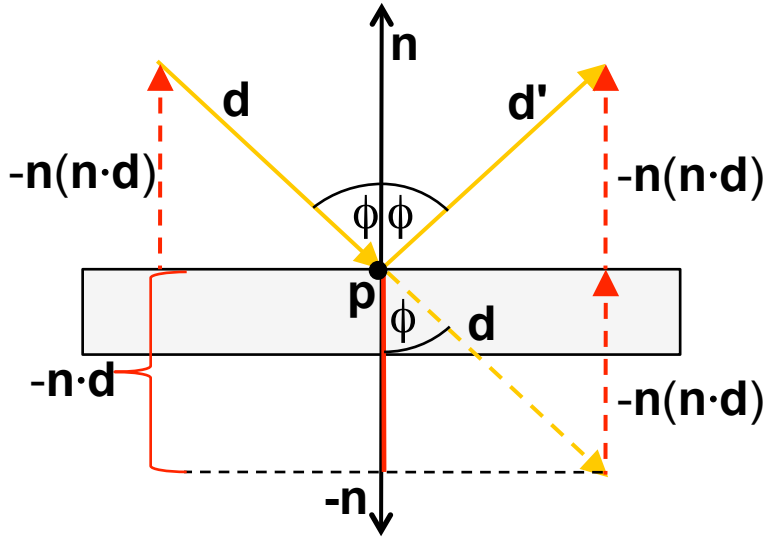
Add the fraction of light reflected from **q** to the reflection at **p**:

$$R_{\mathbf{p}} = R_{\text{ambient},\mathbf{p}} + R_{\text{diffuse},\mathbf{p}} + R_{\text{specular},\mathbf{p}} + \underline{\text{reflectivity}}_{\mathbf{p}} R_{\mathbf{q}}$$

Perfect Ray Reflection

- **Given**: incoming ray direction **d**
- **Wanted**: outgoing ray direction **d'**
- Reflection rule: incoming angle = outgoing angle (both are ϕ)
- In diagram:
  - Horizontal component of **d** stays the same
  - Only vertical component is reversed (ray bounces off)
  - Use dot product to get the vertical component ($-\mathbf{n} \cdot \mathbf{d} = \cos(\phi) \cdot |\mathbf{d}| \cdot |\mathbf{n}|$, $|\mathbf{n}|=1$)

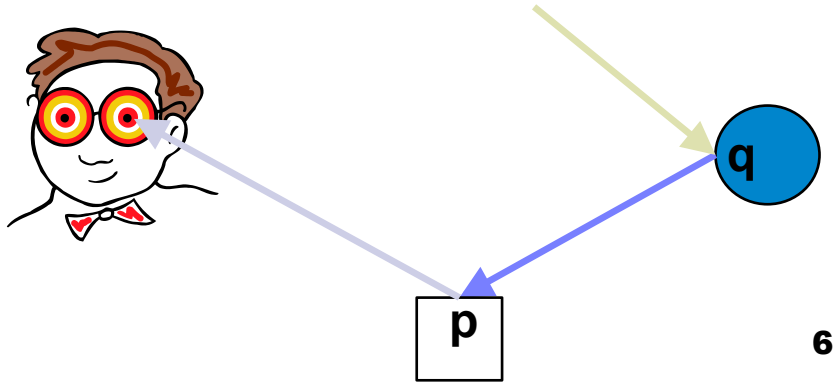
$$\mathbf{d}' = \mathbf{d} - 2\mathbf{n}(\mathbf{n} \cdot \mathbf{d})$$

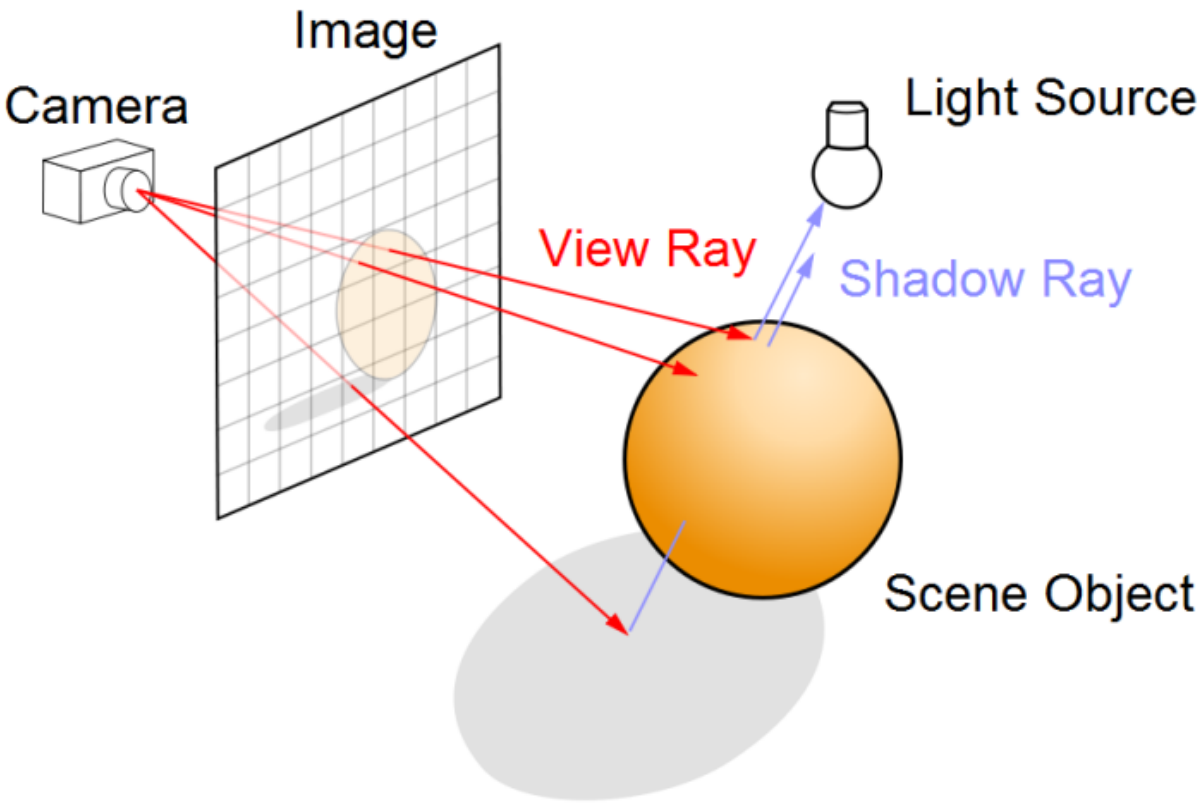


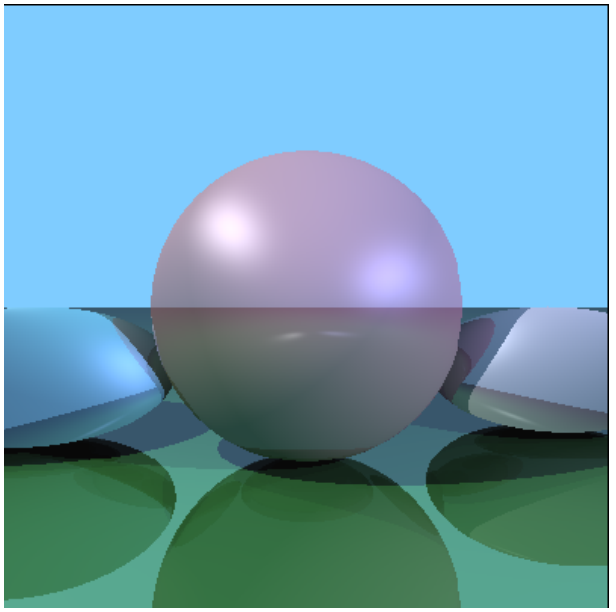
Adding Reflections to `shade`

```
Color shade(Hit hit, int reflectionNo) {  …
    // if no hit, return background color…
    // calculate p and m…

    for(int i=0; i<numLights; i++) {
        // ambient, diffuse, specular reflection
    }

    // ray reflection
    if("not too many reflections"
        && "reflectivity high enough") {
      Hit reflection = intersect( ? , ? );
      color = color +
          shade(reflection, reflectionNo+1)
              * hit.object->reflectivity;
    }
    return color;
}
```

- Make sure that there is a maximum number of reflections
- Calculate reflection only for fairly reflective surfaces
- Cast reflection ray using `intersect`
- Add light coming from reflection ray (attenuated by reflectivity) to the color (calling `shade` recursively)



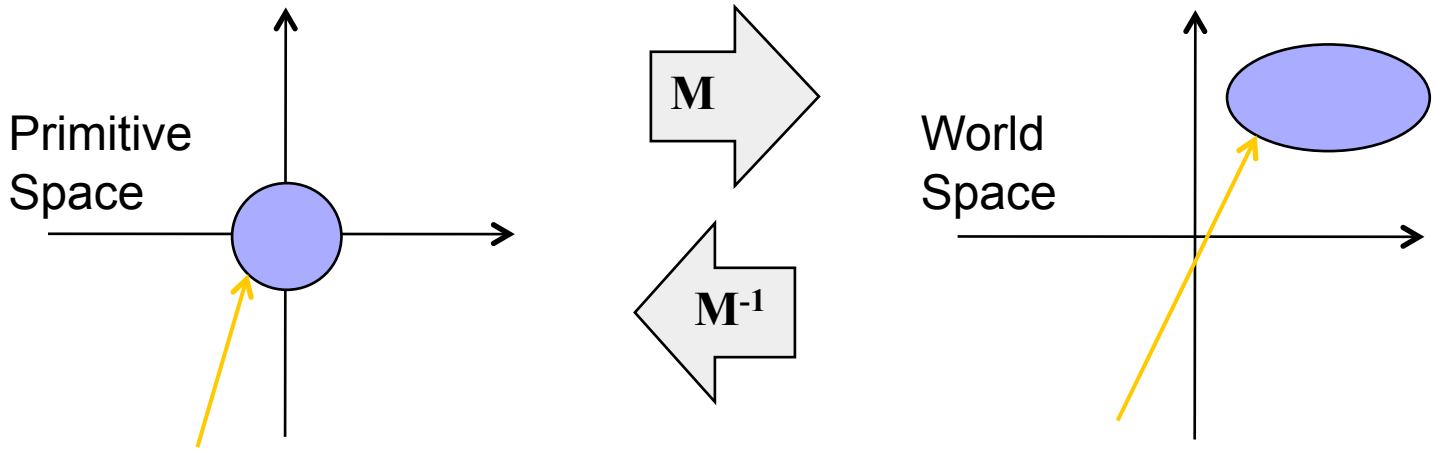




RAY TRACING TRANSFORMED PRIMITIVES

Transformed Primitives

Problem: How to intersect with transformed primitives?
(e.g. scaled and translated unit sphere)

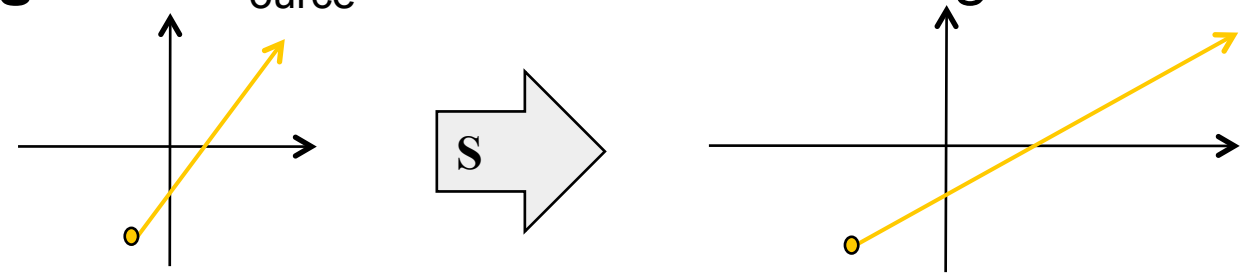


Solution: intersection of ray with transformed primitive is the same as intersection with inversely transformed ray and primitive

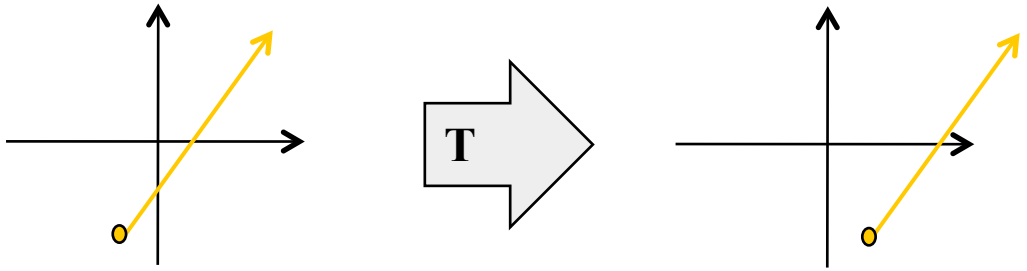
- Intersect with transformed ray $\tilde{\mathbf{s}}_{\text{ource}} + \tilde{\mathbf{d}}t$
  where $\tilde{\mathbf{s}}_{\text{ource}} = \mathbf{M}^{-1}\mathbf{s}_{\text{ource}}$ and $\tilde{\mathbf{d}} = \mathbf{M}^{-1}\mathbf{d}$
- t for the intersection is the same in world and primitive space

Transforming Rays

- Ray has position vector (point) $\mathbf{s}_{ource}$ and direction vector **d**
- **Scaling**: both $\mathbf{s}_{ource}$ and direction **d** change



- **Translation**: $\mathbf{s}_{ource}$ changes, but the direction **d** does not (point **source** has w=1, but direction vector **d** has w=0)



- If $\mathbf{M}=\mathbf{T}\,\mathbf{S}$ then inverse ray transformation is:

$$\tilde{\mathbf{s}}_{ource} = \mathbf{S}^{-1}\mathbf{T}^{-1}\mathbf{s}_{ource} \text{ and } \tilde{\mathbf{d}} = \mathbf{S}^{-1}\mathbf{d}$$

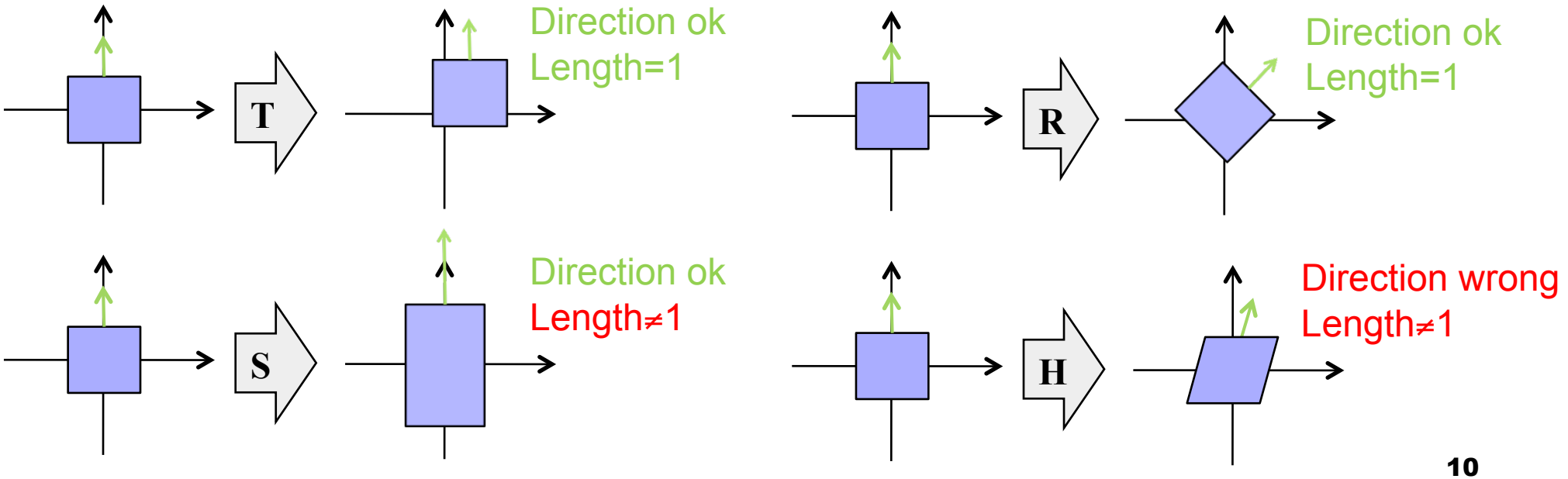
Surface Normals

Problem:

After transforming a vertex, its surface normal needs to be adjusted

1. The direction might be wrong
2. The length might not be 1 anymore

Examples:



Transformation of Surface Normals

How to adjust surface normal **n** after arbitrary transformation **M** ?

Answer: adjust by transforming with $\mathbf{Q} = (\mathbf{M}^{-1})^T = (\mathbf{M}^T)^{-1} = \mathbf{M}^{-T}$

Proof: let $\mathbf{p}_1$ and $\mathbf{p}_2$ be two points on a polygon with normal **n**

1. $\mathbf{n}\cdot(\mathbf{p}_2 - \mathbf{p}_1) = 0 \Leftrightarrow \mathbf{n}^T(\mathbf{p}_2 - \mathbf{p}_1) = 0$ (**n** perpendicular to polygon)
2. This has also to be true after transforming $\mathbf{p}_1$ and $\mathbf{p}_2$ by **M** and **n** by **Q**, i.e. $(\mathbf{Qn})^T(\mathbf{M}(\mathbf{p}_2 - \mathbf{p}_1)) = 0$
3. Apply rule from matrix algebra: $(\mathbf{Qn})^T = \mathbf{n}^T\mathbf{Q}^T$: $\mathbf{n}^T\mathbf{Q}^T\mathbf{M}(\mathbf{p}_2 - \mathbf{p}_1) = 0$ whenever $\mathbf{n}^T(\mathbf{p}_2 - \mathbf{p}_1) = 0$
4. Solution is $\mathbf{Q}^T\mathbf{M} = \mathbf{I} \Rightarrow \mathbf{Q}^T = \mathbf{M}^{-1} \Rightarrow \mathbf{Q} = (\mathbf{M}^{-1})^T = \mathbf{M}^{-T}$

Note: the adjusted normal is not always normalized

Normals for Transformed Primitives

- Recap: given a normal **n**, after a transformation **M** the new normal is **n'** with **n'** = $\mathrm{normalize}(\mathbf{M}^{-T}\,\mathbf{n})$
- Normals are direction vectors (i.e. not affected by translation of the object, w=0)
- For normal **n** and object transformation **M**=**T** **S** the adjusted normal is **n'** = $\mathrm{normalize}(\mathbf{S}^{-1}\,\mathbf{n})$

Sphere normal in our implementation:

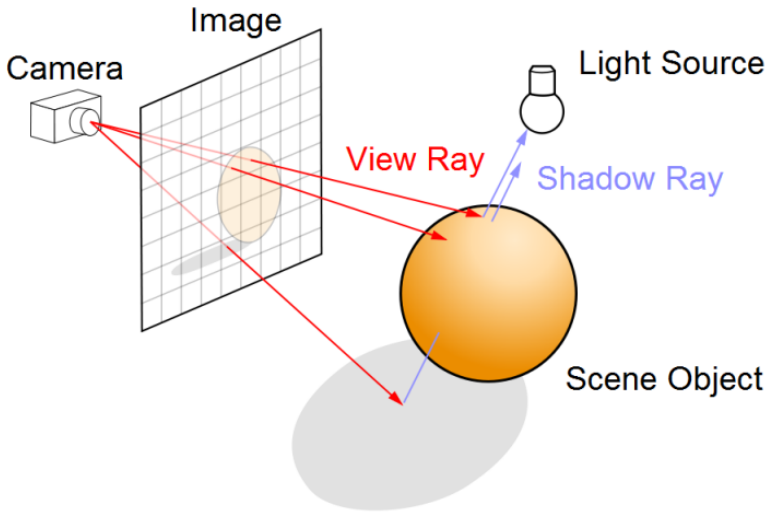
- Calculated from point **p** on the transformed sphere
- In order to get the adjusted normal **n'**:
  1. Calculate corresponding point $\mathbf{p}_{pr}$ on primitive sphere: $\mathbf{p}_{pr} = \mathbf{S}^{-1}\,\mathbf{T}^{-1}\,\mathbf{p}$
  2. Calculate corresponding normal $\mathbf{n}_{pr}$ for the primitive sphere
  3. Return adjusted $\mathbf{n}_{pr}$

Using Transformed Rays

```
Hit intersect(Vector source, Vector d) {
    Hit hit = Hit( source, d, -1, NULL );
    for( int i = 0; i < numObjects; i++) {
        // inversely transform ray with
        // object modeling transformation
        Vector source2 = ? ;
        Vector d2 = ? ;

        float t = objects[i]->Intersect(
                                    source2, d2);

        if ( t > 0.00001
            && ( hit.object == NULL || t < hit.t ) )
            hit = Hit( source, d, t, objects[i] );
    }
   return hit;
}
```

```
Vector Sphere::Normal( Vector p ) {
    // get corresponding point p2 on primitive
    // sphere by inverting modeling transform
    Vector p2 = ? ;
    // adjust primitive normal with M-T
    return ? ;
}

Vector Plane::Normal( Vector p ) {
    // adjust primitive normal n with M-T
    return ? ;
}
```

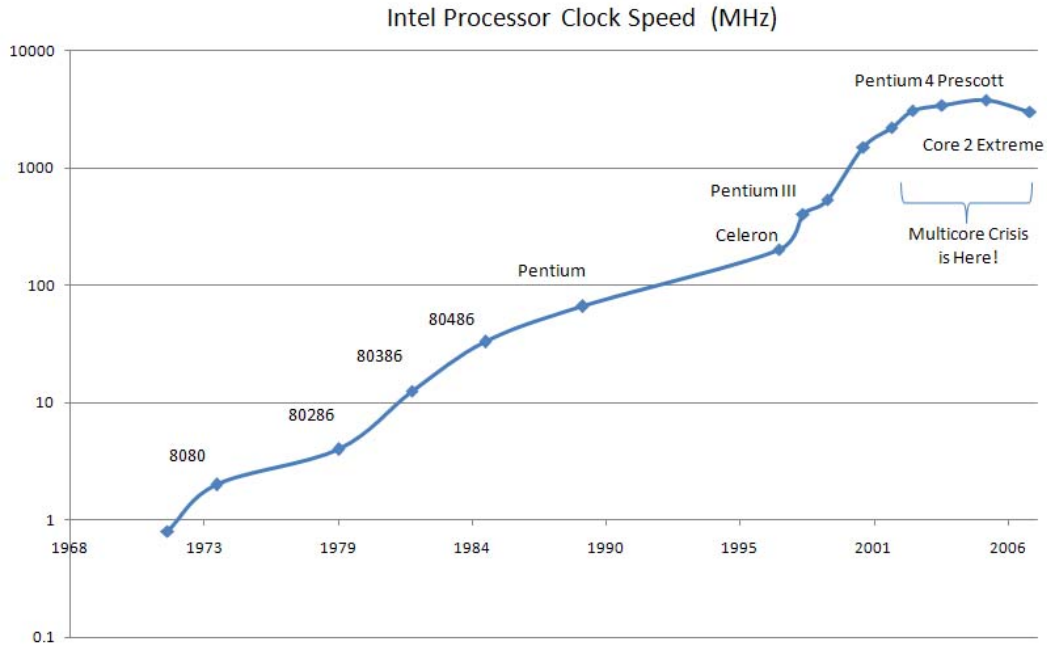
- Use transformed ray (source2, d2) to get *t* ; then
- Use *t* with original ray (source, d)



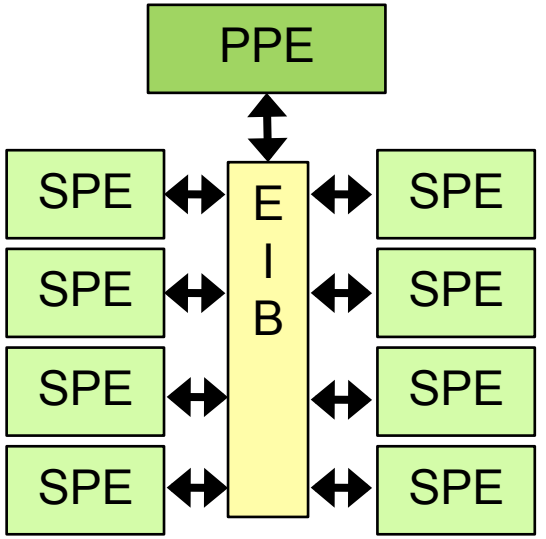
SPEEDING UP RAY TRACING

Tracing Rays in Parallel

- Tracing one ray after the other is slow
- **Observation**: calculations for different primary rays are independent
- **Idea**: trace primary rays in parallel
- For *n* pixels and *m* processors, each processor traces only *n/m* pixels

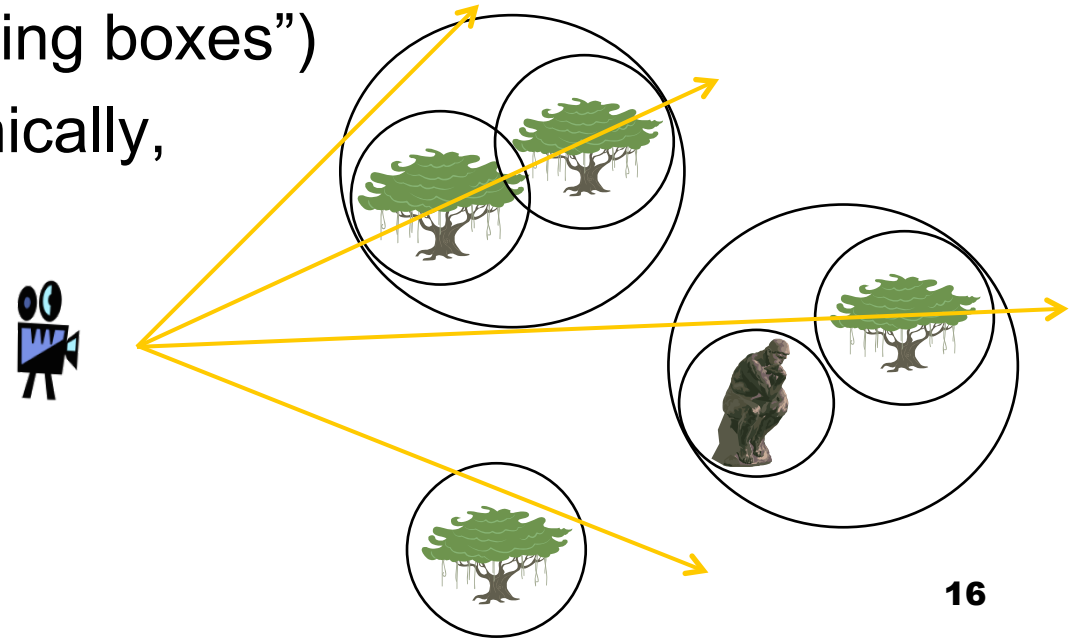


Example: Cell Processor



Object Extents

- Without optimization: each ray must be tested for intersection with every object
- **Extent**: simple shape that encloses one or more objects
- Helps to rule out intersections: if ray does not hit extent, then it also does not hit contained objects
- Typical extents: spheres (“bounding spheres”), boxes aligned with coordinate axes (“bounding boxes”)
- Extents can be used hierarchically, i.e. extents nested in extents



Using Spheres as Extents

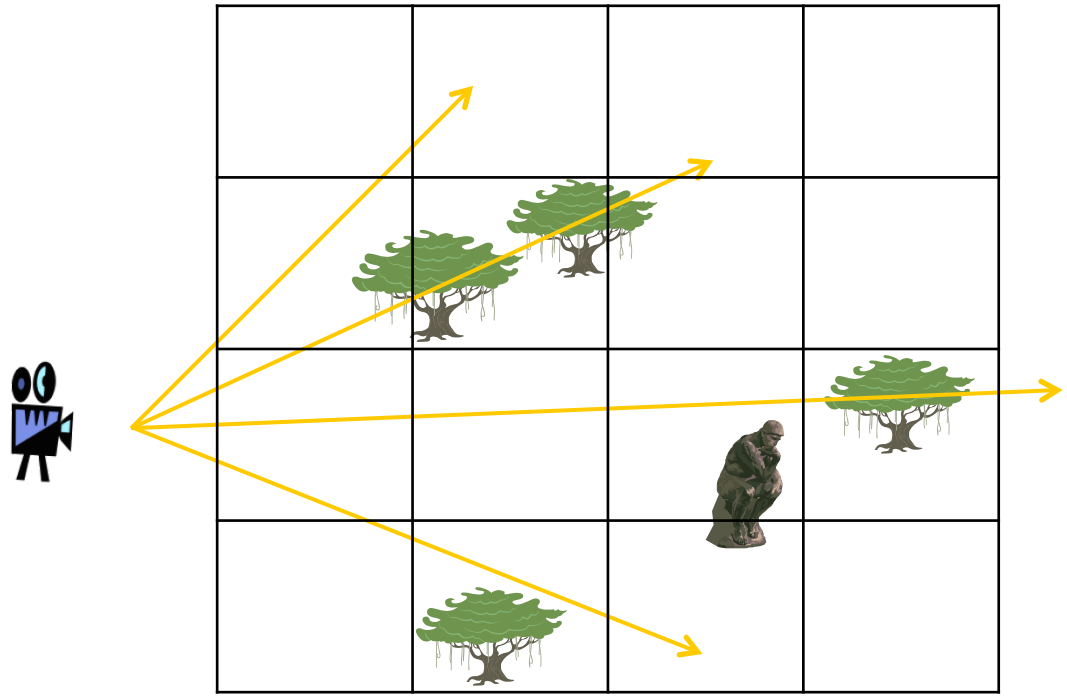
- We know how to intersect a ray (**eye**, **d**) with a (primitive) sphere:

$$t_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \qquad \textit{with} \quad \begin{aligned} a &= \mathbf{d}\cdot\mathbf{d} \\ b &= 2\,\mathbf{e}_{ye}\cdot\mathbf{d} \\ c &= \mathbf{e}_{ye}\cdot\mathbf{e}_{ye} - 1 \end{aligned}$$

- Interesting case for use as extent:
  if $(b^2 - 4ac) < 0$ then ray misses sphere (fast to compute)
- The more objects are in a bounding sphere, the less intersection tests are necessary if the ray does not hit it
- **Research problem**: how do we place hierarchical bounding spheres automatically? (also for other extent types)

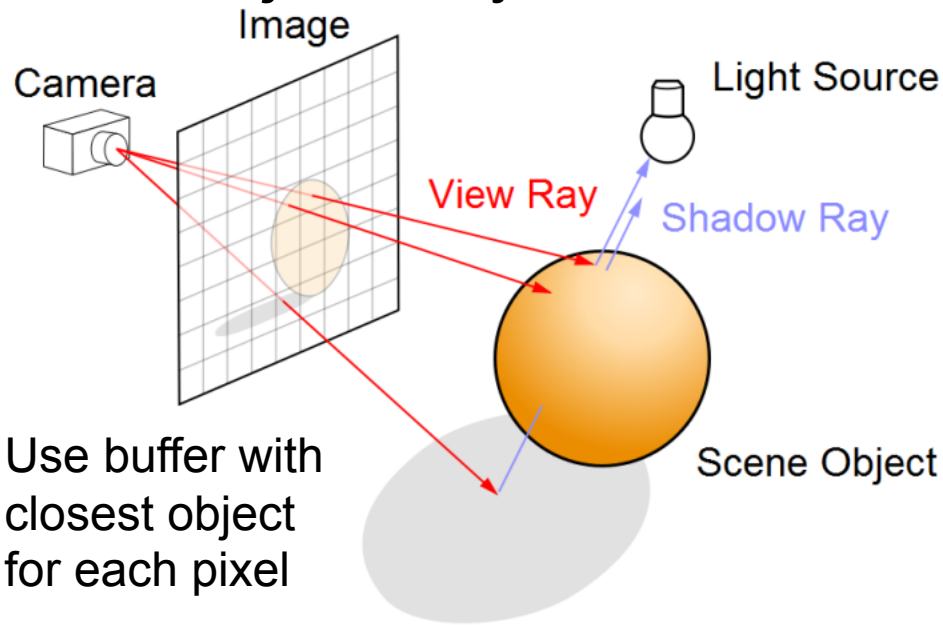
Space Division

- **Idea**: subdivide the world into subspaces
- Speedup by excluding some subspaces (and their objects)
- Subdivision can be done recursively
- Examples: division into cubic boxes, binary space division (BSP) trees



Item Buffer

- **Idea**: for each pixel, store which object is visible (similar to depth buffer)
- Item buffer can be generated quickly by iterating over objects, with techniques from polygon rendering
- For primary rays (those going through the pixels) we know immediately which object they hit



SUMMARY

Summary

- Ray tracing reflections
  - Construct reflection ray and call `shade` recursively
  - Add reflectivity times color from previous reflection to current color
- Ray tracing transformed primitives
  - Intersect inversely transformed ray with primitive, get t
  - Adjust primitive normal with $\mathbf{M}^{-T}$
  - Note: direction vectors are not translated
- Speeding up ray tracing: extents, space division, item buffer

References:

- Ray Tracing Reflections: Hill, Chapter 12.12
- Intersection with Transformed Objects: Hill, Chapter 12.4.3
- Using Extents: Hill, Chapter 12.10

Quiz

1. How do we consider light reflected from another surface?
2. Given a modelling transformation $\mathbf{M}=\mathbf{TS}$, how do we transform a ray ($\mathbf{s}_{ource}$, $\mathbf{d}$) with $\mathbf{M}^{-1}$?
3. What is an extent? Why is it useful?

Stare at the black lightbulb
for at least 30 seconds,
then immediately stare at
the white area on the screen.