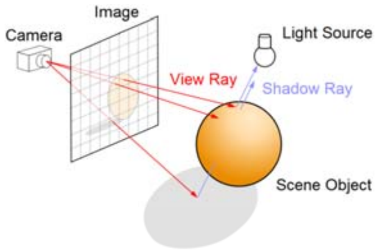
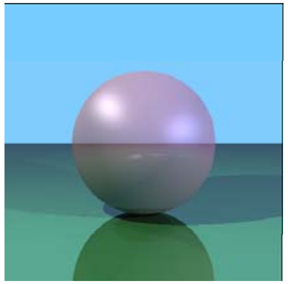
Computer Graphics: Ray Tracing III

Part 2 – Lecture 9

Today's Outline

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



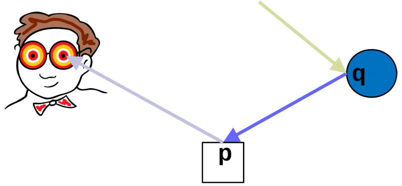


RAY TRACING REFLECTIONS

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

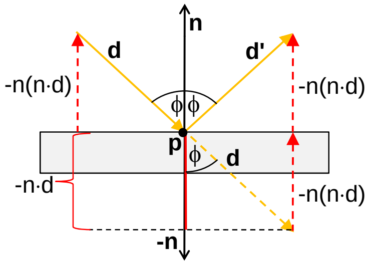
Reflectivity: fraction of incident radiation reflected by a surface (between 0 and 1)

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

$$R_p = R_{ambient,p} + R_{diffuse,p} + R_{specular,p} + reflectivity_p \; R_q$$

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 (-**n** · **d** = cos(ϕ) · |**d**| · |**n**|, |**n**|=1)

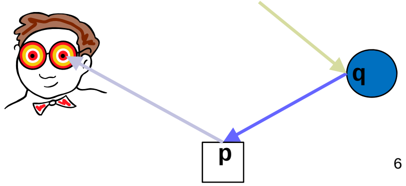


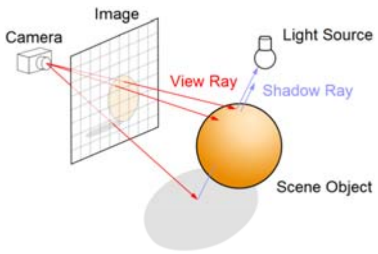
$$d' = d - 2\ n(n \cdot d)$$

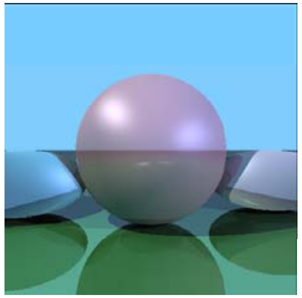
Adding Reflections to shade

```
Color shade(Hit hit, int reflectionNo) {  …
   for(int i=0; i<numLights; i++) {
      color = color + … ;     // ambient reflect.

       // cast a "shadow feeler“
       Hit feeler = intersect( … );
       if( … ) { … }

       // 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)



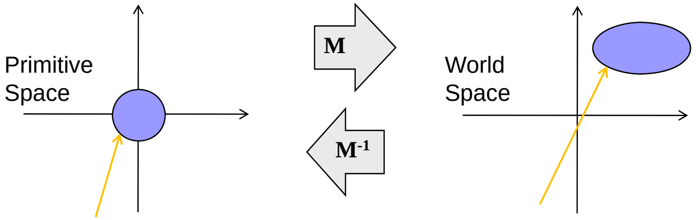




RAY TRACING
TRANSFORMED PRIMITIVES

Transformed Primitives

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

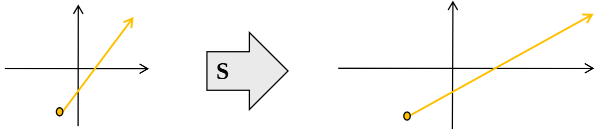


Solution: intersection of ray with transformed primitive is the same as intersection with inversely transformed ray and primitive

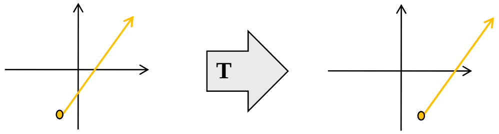
- Intersect with transformed ray (**source’** + **d’** *t*)
  i.e. **source’** = $\mathbf{M}^{-1}$ **source** and **d’** = $\mathbf{M}^{-1}$ **d**
- *t* for the intersection is the same in world and primitive space

Transforming Rays

- Ray has position vector (point) **source** and direction vector **d**
- **Scaling**: both **source** and direction **d** change



- **Translation**: **source** changes, but the direction **d** does not (point **source** has w=1, but direction vector **d** has w=0)



- If **M**=**T S** then inverse ray transformation is:
  source' = $\mathbf{S}^{-1}\,\mathbf{T}^{-1}$ **source** and **d'** = $\mathbf{S}^{-1}$ **d**

Normals for Transformed Primitives

- Recap: given a normal **n**, after a transformation **M** the new normal is **n'** with **n'** = normalize($\mathbf{M}^{-T}$ **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'** = normalize($\mathbf{S}^{-1}$ **n**)

Sphere normal in our implementation:

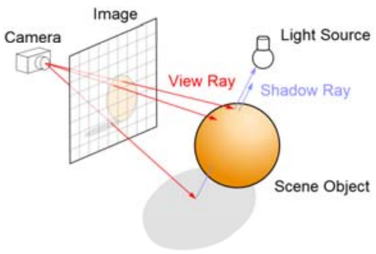
- Calculated from point **p** on the transformed sphere
- In order to get the adjusted normal **n'**:
  1. Calculate corresponding point $\mathbf{p}_{pr}$ on primitive sphere: $\mathbf{p}_{pr} = \mathbf{S}^{-1}\,\mathbf{T}^{-1}\,\mathbf{p}$
  2. Calculate corresponding normal $\mathbf{n}_{pr}$ for the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

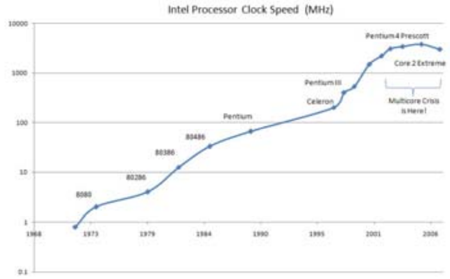
- Use transformed ray (source2, d2) to get t
- Then use t with original ray (source, d)



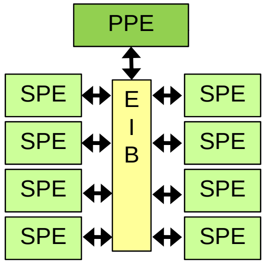
SPEEDING UP RAY TRACING

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


PPE

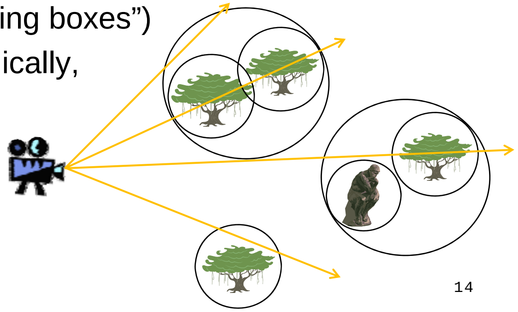
SPE SPE SPE SPE

E I B

SPE SPE SPE SPE

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

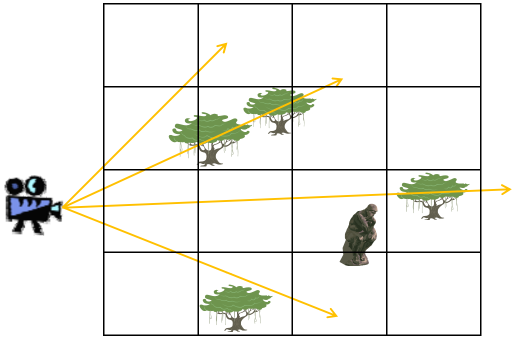
- We know how to intersect a ray (**eye**, **d**) with a (primitive) sphere:

$$t_{1,2} = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A} \quad \textit{with} \quad \begin{array}{l} A=\mathbf{d}\cdot\mathbf{d} \\ B=2\,\mathbf{eye}\cdot\mathbf{d} \\ C=\mathbf{eye}\cdot\mathbf{eye}-1 \end{array}$$

- 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)

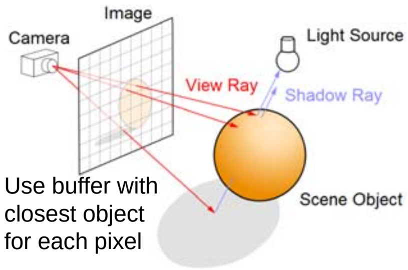
Space Division

- **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



Use buffer with
closest object
for each pixel

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 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 modeling transformation $\mathbf{M}=\mathbf{TS}$, how do we transform a ray (**source**, **d**) with $\mathbf{M}^{-1}$?
3. What is an extent? Why is it useful?