Preview

• Basic ray tracing model
• Coordinate systems
• Ray-object intersections
• Lighting models for ray tracing
• Aliasing and anti-aliasing in ray tracing
• Texture mapping in a ray tracer
• Ray tracing optimization
Ray Tracing Overview

- Ray tracing algorithm simulates light rays *in reverse*.  
  - Rays go from eye through pixel into the scene
- Rays that hit *shiny* objects, propagate as a *reflection*.
- Rays that hit *semi-transparent* objects, propagate as *refraction*.
- *Shadows* can be computed by sending rays to each light source from each object hit.
- Pixel color is combination of all objects hit by rays spawned by initial sight ray.
- A *ray tree* is (conceptually) built while recursively generating rays
  - once all ray branches end, pixel value is built from the leaves up to the root

Ray Tracing Example - 1

Primary ray hits box

Ray Tree

Can’t color pixel yet, since don’t yet know what color the object will have at that point
Ray Tracing Example - 2

Box not transparent, no refractive ray (blue/gray)
Light ray (yellow/dotted) is blocked, box in shadow
Reflected ray (black) hits sphere

Ray Tracing Example - 3

Sphere not transparent; empty blue/gray link
Light ray not blocked
Sphere reflective; ray hits striped poly
Ray Tracing Example - 4

Striped object transmits light
Reflected ray hits nothing
Light ray not blocked

Ray Tracing Example - 5

Object not transparent
Reflected ray hits sphere
Light ray not blocked
Ray Tracing Example - 6

Sphere not transparent
Reflected ray hits bricks
Light ray blocked

Ray Tree

Ray Tracing Example - 7

Bricks not transparent
Reflected ray hits nothing
Light ray not blocked

Ray Tree

All rays taken to their ends
Now, build pixel value from bottom up
Ray Tracing Main Algorithm

Create mapping to eye coord system
Define resolution of viewport
Let RayHit class represent a ray/object intersection: object, ray, hit point, ...
for each pixel
P(t) = parametric ray from eye to pixel
RayHit closestHit = findClosest( P(t) )
if ( closest == null )
    color pixel with background
else
    pixel.color = shade( closestHit )

findClosest method

findClosest( Ray R(t) ):
closestT = +infinity
RayHit closeHit = new RayHit(inf,null)
foreach object
    RayHit hit = intersect( R, object )
    if hit.t < closeHit.t
        closeHit = hit
return closeHit
**intersect method**

**RayHit intersect(R(t), object):**

- Ray/object intersection is object dependent; in an OO world it is cleanest to have the objects implement the intersection. We’ll come back to this issue.
- Usually most efficient to do the intersection in object coordinates
  - Consider a rectangular box; probably defined in its coord system as axis aligned: makes ray intersections even easier
  - Consider a sphere; ray-sphere intersection is especially easy, but affine transform can map sphere to a sheared ellipsoid
- Need to map the Ray by the inverse modeling transform
  - But, that’s easy and cheap and a ray maps to a ray

**shade method**

```java
shade( RayHit hit )
Object objFace = hit.getHitFace
color = hit.ka * ambientLight( objFace )
color += hit.ks * specularReflection( hit )
color += hit.kt * specularTransmission( hit )
foreach light, L,
    color += directLight( L, hit )

ambientLight( objFace )
return objFace.color * ambientLight.color;
// * multiplies corr elements
```

CS770/870 Spring 2017 Bergeron
**specularReflection** method

```java
specularReflection( RayHit hit )
// specular reflection only occurs along ray;
//   there is no cos^n term as in direct light
Object objFace = hit.getHitFace
if objFace is shiny enough
    create S(t): dir of principal specular refl
    reflectHit = findClosest( S )
    if reflectHit != null
        return shade( reflectHit )
return 0
```

**specularTransmission** method

```java
specularTransmission( RayHit hit )
Object objFace = hit.getHitFace
if objFace is partially transparent
    create T(t): dir of transmission
    // β is refraction angle, a material prop
    transmitHit = findClosest( T )
    if transmitHit != null
        return shade( transmitHit )
return 0
```
**directLight** method

```java
directLight( Light L, RayHit hit )
Object objFace = hit.getHitFace
create L(t), vector from L to hit point
lightHit = findClosest( L(t) )
col = object color at hit point (rgb)
I_L = intensity of light L (rgb)
if ( lightHit == rayHit ) // object sees light
    directColor = kd * (col * I_L) * dot( L, N )
directColor += ks * (col * I_L) * (dot(H,N))^n
return directColor
else
    return 0
```

Note: rgb*rgb \(\Rightarrow (r_1*r_2, g_1*g_2, b_1*b_2)\)

---

**Ray Tracing Algorithm Notes**

- No explicit ray tree is built in this version
  - the recursive stack tree encapsulates the ray tree
- This is a high-level pedagogic presentation
  - real algorithms would make the shiny and transparent tests prior to the recursive calls
- Care must be taken in creating the rays to insure that you don’t hit yourself in the recursion
  - ray start must be just off the surface on the plus side for reflection and minus side for refraction
- Lots of room for efficiency improvements
Efficiency Options

- Ray-Object intersection is the major cpu load
- Two basic kinds of efficiencies (not really disjoint)
  - reduce number of intersection tests
  - reduce cost of the intersection tests
- Vast majority of ray-object intersect tests fail
  - i.e., ray fails to intersect the object
  - need ways to identify failure quickly
  - some of these can be interpreted as either reducing cost or count
- Reducing number of tests
  - Limit tree depth
  - Bounding volume tests
  - Space partitioning
- Reducing cost of tests

Limiting Tree Traversal

- Terminate tree depth after some fixed depth
  - what happens if there are 2 mirrors in the scene facing each other?
- Each color computation uses a $k$ coefficient that is less than one (and often much less than 1)
  - Each tree level, therefore, contributes only a fraction of its color to its parent
  - These fractions are multiplied and often get very small very fast

$$k_s = 0.6$$
$$k_t = 0.3$$
$$k_t = 0.2$$
$$k_t = 0.1$$

Node adds < 4% to pixel color
Node adds < 2% to pixel color
Bounding Volume Tests

- Many world objects are composed of many parts
  - objects are often fairly compact
  - or, are made of compact subparts
- Enclose the object (or its subparts) in bounding volumes, like a sphere
  - first test ray with bounding volume
  - if it misses, don’t need to test any of the parts
  - fast failure test
- Typical bounding volumes:
  - spheres
  - axis aligned boxes
  - parallel planes

Spatial Partitioning

- Partition world space into axis aligned regions
- Each region contains list of objects that have any part in that region
- Ray intersection proceeds along from region to region (very cheap computations)
  - once enter a region, only need to intersect with objects in the region
  - can dramatically reduce computations
Spatial Partitioning Examples

- \( R_1 \) goes through
  - partitions \([0,1], [1,1], [1,2],[2,2],[3,2]\)
  - only needs to test objects C and E before finding E
- \( R_2 \) goes through
  - \([0,0], [1,0], [2,0], [2,1], [3,1]\)
  - intersects A while in \([2,0]\)
    - but it’s not right, can tell since intersection is **not** in the partition
  - intersects B while in \([2,1]\), also wrong
  - intersects B and A in \([3,1]\), picks B
  - do not need to recompute intersections; can save them for current ray

Binary Space Partitioning

- Choose a plane that roughly partitions objects in the scene into two subtrees:
  - those on + side of plane and
  - those on − side of plane
  - this is root of a BSP tree
- for each subtree
  - choose a partitioning plane
- Ray/plane intersection is cheap
  - once ray crosses a plane, only test objects on that side
  - each tree node (ideally) cuts in \( \frac{1}{2} \) number of objects left to test.