Foundations of Computer Graphics  
(Spring 2010)  
CS 184, Lectures 16, 17:  
Nuts and bolts of Ray Tracing  
Ravi Ramamoorthi  
http://inst.eecs.berkeley.edu/~cs184

Acknowledgements: Thomas Funkhouser and Greg Humphreys

Midterm Brief Discussion

To Do

- Finish homework 3
- Start early on raytracer assignment (assn 4)

Outline

- Camera Ray Casting (choose ray directions) [2.3]
- Ray-object intersections [2.4]
- Ray-tracing transformed objects [2.4]
- Lighting calculations [2.5]
- Recursive ray tracing [2.6]

Outline in Code

```java
Image Raytrace(Camera cam, Scene scene, int width, int height)
{
    Image image = new Image (width, height) ;
    for (int i = 0 ; i < height ; i++) {
        for (int j = 0 ; j < width ; j++) {
            Ray ray = RayThruPixel (cam, i, j) ;
            Intersection hit = Intersect (ray, scene) ;
            image[i][j] = FindColor (hit) ;
        }
    }
    return image ;
}
```

Heckbert's Business Card Ray Tracer

```java
void drawr(double x, y, z, Image img, Scene scene, Color color, double rad, double phi, double theta, Camera cam) {
    for (int i = 0 ; i < 100 ; i++) {
        Ray ray = RayThruPixel (cam, i, j) ;
        Intersection hit = Intersect (ray, scene) ;
        if (hit != null) {
            color = scene.FindColor (hit) ;
            img[i][j] = color ;
        }
    }
}
```
Ray Casting

Virtual Viewpoint

Virtual Screen

Objects

Ray Casting

Finding Ray Direction

- Goal is to find ray direction for given pixel i and j
- Many ways to approach problem
  - Objects in world coord, find dirn of each ray (we do this)
  - Camera in canonical frame, transform objects (OpenGL)
- Basic idea
  - Ray has origin (camera center) and direction
  - Find direction given camera params and i and j
- Camera params as in gluLookAt
  - Lookfrom[3], LookAt[3], up[3], fov

Similar to gluLookAt derivation

- gluLookAt(eyex, eyey, eyez, centerx, centery, centerz, upx, upy, upz)
- Camera at eye, looking at center, with up direction being up

Camera coordinate frame

\[ w = \frac{a}{\|a\|}, \quad u = \frac{b \times w}{\|b \times w\|}, \quad v = w \times u \]

- We want to position camera at origin, looking down –Z dirn
- Hence, vector a is given by eye – center
- The vector b is simply the up vector

Canonical viewing geometry

\[ \alpha = \tan\left(\frac{\text{fovx}}{2}\right) \left(\frac{j - \text{width}/2}{\text{width}/2}\right), \quad \beta = \tan\left(\frac{\text{fovy}}{2}\right) \left(\frac{i - \text{height}/2}{\text{height}/2}\right) \]

\[ \text{ray} = \text{eye} + \frac{\alpha u + \beta v - w}{\|\alpha u + \beta v - w\|} \]
Outline

- Camera Ray Casting (choosing ray directions) [2.3]
- Ray-object intersections [2.4]
- Ray-tracing transformed objects [2.4]
- Lighting calculations [2.5]
- Recursive ray tracing [2.6]

Outline in Code

```
Image Raytrace(Camera cam, Scene scene, int width, int height)
{
    Image image = new Image (width, height)
    for (int i = 0 ; i < height ; i++)
        for (int j = 0 ; j < width ; j++)
        {
            Ray ray = RayThruPixel (cam, i, j)
            Intersection hit = Intersect (ray, scene)
            image[i][j] = FindColor (hit)
        }
    return image
}
```

Ray-Sphere Intersection

**Ray**: \( \mathbf{r} = \mathbf{P} - \mathbf{P}_t \)

**Sphere**: \((\mathbf{P} - \mathbf{C})(\mathbf{P} - \mathbf{C}) - r^2 = 0\)

Solve quadratic equations for \( t \)

- 2 real positive roots: pick smaller root
- Both roots same: tangent to sphere
- One positive, one negative root: ray origin inside sphere (pick + root)
- Complex roots: no intersection (check discriminant of equation first)

Ray-Sphere Intersection

**Ray**: \( \mathbf{r} = \mathbf{P} - \mathbf{P}_t \)

**Sphere**: \((\mathbf{P} - \mathbf{C})(\mathbf{P} - \mathbf{C}) - r^2 = 0\)

Substitute

**Ray**: \( \mathbf{r} = \mathbf{P} - \mathbf{P}_t \)

**Sphere**: \((\mathbf{P}_0 + \mathbf{P}_t - \mathbf{C})(\mathbf{P}_0 + \mathbf{P}_t - \mathbf{C}) - r^2 = 0\)

Simplify

\[ t^2 (\mathbf{P}_0 \cdot \mathbf{P}_t) + 2t (\mathbf{P}_0 - \mathbf{C}) \cdot (\mathbf{P}_0 - \mathbf{C}) - r^2 = 0 \]

**Intersection point**: \( \mathbf{r} = \mathbf{P} - \mathbf{P}_t \)

**Normal** (for sphere, this is same as coordinates in sphere frame of reference, useful for other tasks)

\[ \text{normal} = \frac{\mathbf{P} - \mathbf{C}}{|\mathbf{P} - \mathbf{C}|} \]
Ray-Triangle Intersection

- One approach: Ray-Plane intersection, then check if inside triangle
- Plane equation:
  \[ \begin{vmatrix} \mathbf{a} - \mathbf{b} \cdot \mathbf{n} \end{vmatrix} = 0 \]

Ray inside Triangle

- Once intersect with plane, still need to find if in triangle
- Many possibilities for triangles, general polygons (point in polygon tests)
- We find parametrically (barycentric coordinates). Also useful for other applications (texture mapping)

Other primitives

- Much early work in ray tracing focused on ray-primitive intersection tests
- Cones, cylinders, ellipsoids
- Boxes (especially useful for bounding boxes)
- General planar polygons
- Many more
- Consult chapter in Glassner (handed out) for more details and possible extra credit

Ray Scene Intersection

```c
Intersection FindIntersection(Ray ray, Scene scene) {
    min_t = infinity
    min_primitive = NULL
    For each primitive in scene {
        t = Intersect(ray, primitive);
        if (t > 0 & & t < min_t) then
            min_primitive = primitive
            min_t = t
    } 
    return Intersection(min_t, min_primitive)
} 
```
Outline

- Camera Ray Casting (choosing ray directions) [2.3]
- Ray-object intersections [2.4]
- Ray-tracing transformed objects [2.4]
- Lighting calculations [2.5]
- Recursive ray tracing [2.6]

Transformed Objects

- E.g. transform sphere into ellipsoid
- Could develop routine to trace ellipsoid (compute parameters after transformation)
- May be useful for triangles, since triangle after transformation is still a triangle in any case
- But can also use original optimized routines

Transformed Objects

- Consider a general 4x4 transform M
  - Will need to implement matrix stacks like in OpenGL
- Apply inverse transform $M^{-1}$ to ray
  - Locations stored and transform in homogeneous coordinates
  - Vectors (ray directions) have homogeneous coordinate set to 0 [so there is no action because of translations]
- Do standard ray-surface intersection as modified
- Transform intersection back to actual coordinates
  - Intersection point $p$ transforms as $Mp$
  - Distance to intersection if used may need recalculation
  - Normals $n$ transform as $M^{-1}n$. Do all this before lighting

Outline in Code

Image Raytrace (Camera cam, Scene scene, int width, int height)
{
  Image image = new Image (width, height) ;
  for (int i = 0 ; i < height ; i++)
    for (int j = 0 ; j < width ; j++) {
      Ray ray = RayThruPixel (cam, i, j) ;
      Intersection hit = Intersect (ray, scene) ;
      image[i][j] = FindColor (hit) ;
    }
  return image ;
}
Shadows: Numerical Issues

- Numerical inaccuracy may cause intersection to be below surface (effect exaggerated in figure)
- Causing surface to incorrectly shadow itself
- Move a little towards light before shooting shadow ray

Lighting Model

- Similar to OpenGL
- Lighting model parameters (global)
  - Ambient \( r \ g \ b \) (no per-light ambient as in OpenGL)
  - Attenuation constant linear quadratic (like in OpenGL)
  \[
  L_i = \frac{L}{\text{const} + \text{lin} \cdot d + \text{quad} \cdot d^2}
  \]
- Per light model parameters
  - Directional light (direction, RGB parameters)
  - Point light (location, RGB parameters)

Material Model

- Diffuse reflectance \( r \ g \ b \)
- Specular reflectance \( r \ g \ b \)
- Shininess \( s \)
- Emission \( r \ g \ b \)
- All as in OpenGL

Shading Model

\[
I = K_a + K_e + \sum_{i=1}^{n} L_i \left( K_d \max (\langle i \cdot n, 0 \rangle) + K_s (\max (h \cdot n, 0))^s \right)
\]

- Global ambient term, emission from material
- For each light, diffuse specular terms
- Note visibility/shadowing for each light (not in OpenGL)
- Evaluated per pixel per light (not per vertex)

Outline

- Camera Ray Casting (choosing ray directions) [2.3]
- Ray-object intersections [2.4]
- Ray-tracing transformed objects [2.4]
- Lighting calculations [2.5]
- Recursive ray tracing [2.6]

Mirror Reflections/Refractions

Generate reflected ray in mirror direction, Get reflections and refractions of objects
**Basic idea**

For each pixel
- Trace Primary Eye Ray, find intersection
- Trace Secondary Shadow Ray(s) to all light(s)
  - Color = Visible ? Illumination Model : 0 ;
- Trace Reflected Ray
  - Color += reflectivity * Color of reflected ray

---

**Recursive Shading Model**

\[ I = K_a + K_f + \sum_{i \in I} I_i (K_r \max (l_i \cdot n, 0) + K_t (\max (l_i \cdot n, 0))^2) + K_s I_s + K_l I_l \]

- Highlighted terms are recursive specularities [mirror reflections] and transmission (latter is extra credit)
- Trace secondary rays for mirror reflections and refractions, include contribution in lighting model
- GetColor calls RayTrace recursively (the I values in equation above of secondary rays are obtained by recursive calls)

---

**Problems with Recursion**

- Reflection rays may be traced forever
- Generally, set maximum recursion depth
- Same for transmitted rays (take refraction into account)

---

**Effects needed for Realism**

- (Soft) Shadows
- Reflections (Mirrors and Glossy)
- Transparency (Water, Glass)
- Interreflections (Color Bleeding)
- Complex Illumination (Natural, Area Light)
- Realistic Materials (Velvet, Paints, Glass)

Discussed in this lecture so far
Not discussed but possible with distribution ray tracing
Hard (but not impossible) with ray tracing; radiosity methods

---

**Some basic add ons**

- Area light sources and soft shadows: break into grid of n x n point lights
  - Use jittering: Randomize direction of shadow ray within small box for given light source direction
  - Jittering also useful for antialiasing shadows when shooting primary rays
- More complex reflectance models
  - Simply update shading model
  - But at present, we can handle only mirror global illumination calculations
Acceleration

Testing each object for each ray is slow
- Fewer Rays
  - Adaptive sampling, depth control
- Generalized Rays
  - Beam tracing, cone tracing, pencil tracing etc.
- Faster Intersections
  - Optimized Ray-Object Intersections
  - Fewer Intersections

Acceleration Structures

Bounding boxes (possibly hierarchical)
If no intersection bounding box, needn’t check objects

Bounding Volume Hierarchies

- Build hierarchy of bounding volumes
  - Bounding volume of interior node contains all children

- Use hierarchy to accelerate ray intersections
  - Intersect node contents only if hit bounding volume

Bounding Volume Hierarchies 1

Bounding Volume Hierarchies 2

Bounding Volume Hierarchies 3

Sort hits & detect early termination

```
FindIntersection(Ray ray, Node node)
{
    // Find intersections with child node bounding volumes
    // Sort intersections from to back
    // Process intersections (checking for early termination)
    min_t = infinity;
    for each intersected child i {
        if (min_t < b_.t) break;
        shape_t = FindIntersection(ray, child);
        if (shape_t < min_t) {
            min_t = shape_t;
        }
    }
    return min_t;
}
```
Uniform Grid: Problems

- Potential problem:
  - How choose suitable grid resolution?

  ![Uniform Grid Diagram]

  - Too little benefit if grid is too coarse
  - Too much cost if grid is too fine

Octree

- Construct adaptive grid over scene
  - Recursively subdivide box-shaped cells into 8 octants
  - Index primitives by overlaps with cells

  ![Octree Diagram]

  - Generally fewer cells

Octree traversal

- Trace rays through neighbor cells
  - Fewer cells
  - More complex neighbor finding

  ![Octree Traversal Diagram]

  - Trade-off fewer cells for more expensive traversal

Other Accelerations

- Screen space coherence
  - Check last hit first
  - Beam tracing
  - Pencil tracing
  - Cone tracing

- Memory coherence
  - Large scenes

- Parallelism
  - Ray casting is "embarrassingly parallelizable"

  - etc.

Interactive Raytracing

- Ray tracing historically slow
- Now viable alternative for complex scenes
  - Key is sublinear complexity with acceleration; need not process all triangles in scene
- Allows many effects hard in hardware
- OpenRT project real-time ray tracing (http://www.openrt.de)

Raytracing on Graphics Hardware

- Modern Programmable Hardware general streaming architecture
- Can map various elements of ray tracing
- Kernels like eye rays, intersect etc.
- In vertex or fragment programs
- Convergence between hardware, ray tracing

[Purcell et al. 2002, 2003]

http://graphics.stanford.edu/papers/photongfx