Advanced Computer Graphics
(Fall 2009)
CS 294-13, Rendering Lecture 1: Introduction and
Basic Ray Tracing

Ravi Ramamoorthi
http://inst.eecs.berkeley.edu/~cs294-13/fa09

To Do
- Start working on raytracer assignment (if necessary)
- Start thinking about path tracer, final project

First Assignment
- In groups of two (find partners)
- Monte Carlo Path Tracer
- If no previous ray tracing experience, ray tracer first.
- See how far you go. Many extra credit items possible, fast multi-dim. rendering, imp. sampling...
- This lecture focuses on basic ray tracing
- Likely to be a review for most of you, go over fast

Ray Tracing History
- Appel 1968 - Ray casting
1. Generate an image by sending one ray per pixel
2. Check for shadows by sending a ray to the light

Ray Tracing in Computer Graphics
"An improved illumination model for shaded display."
T. Whitted, CACM 1980
Resolution: 512 x 512
Time: VAX 11/780 (1979) 74 min.
Spheres and Checkerboard, T. Whitted, 1979

Image courtesy Paul Heckbert 1983
Outline

- Camera Ray Casting (choosing ray directions)
- Ray-object intersections
- Ray-tracing transformed objects
- Lighting calculations
- Recursive ray tracing

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 ;
}

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, centersx, centersy, centersz, upx,upy, upz)
- Camera at eye, looking at center, with up direction being up

Constructing a coordinate frame?

We want to associate w with a, and v with b
- But a and b are neither orthogonal nor unit norm
- And we also need to find u

\[ w = \frac{a}{\|a\|} \]
\[ u = \frac{b \times w}{\|b \times w\|} \]
\[ v = w \times u \]
Camera coordinate frame

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

- We want to position camera at origin, looking down \(-Z\) dirn.
- Hence, vector \( a \) is given by \( \text{eye} - \text{center} \).
- The vector \( b \) is simply the up vector.

Canonical viewing geometry

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

Outline

- Camera Ray Casting (choosing ray directions)
- Ray-object intersections
- Ray-tracing transformed objects
- Lighting calculations
- Recursive ray tracing

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 ;
}
```

Ray-Sphere Intersection

\[ \text{ray} = \tilde{P} = \tilde{P}_0 + \tilde{P}_t \]
\[ \text{sphere} = (\tilde{P} - \tilde{C}) \cdot (\tilde{P} - \tilde{C}) - r^2 = 0 \]

Substitute

\[ \text{ray} = \tilde{P} = \tilde{P}_0 + \tilde{P}_t \]
\[ \text{sphere} = (\tilde{P} - \tilde{C}) \cdot (\tilde{P} - \tilde{C}) - r^2 = 0 \]

Simplify

\[ r^2 \cdot (\tilde{P} \cdot \tilde{P}) + 2t \cdot \tilde{P} \cdot (\tilde{P}_0 - \tilde{C}) + (\tilde{P}_0 - \tilde{C}) \cdot (\tilde{P}_0 - \tilde{C}) - r^2 = 0 \]
### Ray-Sphere Intersection

\[ t^2(\vec{P}_1 \cdot \vec{P}_1) + 2t \vec{P}_1 \cdot (\vec{P}_0 - \vec{C}) + (\vec{P}_0 - \vec{C}) \cdot (\vec{P}_0 - \vec{C}) - \rho^2 = 0 \]

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

- Intersection point: \( ray = \vec{P} = \vec{P}_0 + \vec{P}_1 t \)
- Normal (for sphere, this is same as coordinates in sphere frame of reference, useful other tasks)
  \[ normal = \frac{\vec{P} - \vec{C}}{|\vec{P} - \vec{C}|} \]

### Ray-Triangle Intersection

- One approach: Ray-Plane intersection, then check if inside triangle
- Plane equation:
  \[ plane = \vec{P} \cdot \vec{n} - \vec{A} \cdot \vec{n} = 0 \]

### Ray-Triangle Intersection

- One approach: Ray-Plane intersection, then check if inside triangle
- Plane equation:
  \[ plane = \vec{P} \cdot \vec{n} - \vec{A} \cdot \vec{n} = 0 \]
  \[ \vec{n} = \frac{(C - \vec{A}) \times (\vec{B} - \vec{A})}{|C - \vec{A}| \times (\vec{B} - \vec{A})} \]

- Combine with ray equation:
  \[ ray = \vec{P} = \vec{P}_0 + \vec{P}_1 t \]
  \[ (\vec{P}_0 + \vec{P}_1 t) \cdot \vec{n} = \vec{A} \cdot \vec{n} \]
  \[ t = \frac{\vec{A} \cdot \vec{n} - \vec{P}_0 \cdot \vec{n}}{\vec{P}_1 \cdot \vec{n}} \]

### 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, ellipsoides
- Boxes (especially useful for bounding boxes)
- General planar polygons
- Many more
- Many references. For example, chapter in Glassner introduction to ray tracing (see me if interested)

Ray Scene Intersection

Outline

- Camera Ray Casting (choosing ray directions)
- Ray-object intersections
- Ray-tracing transformed objects
- Lighting calculations
- Recursive ray tracing

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

- Camera Ray Casting (choosing ray directions)
- Ray-object intersections
- Ray-tracing transformed objects
- Lighting calculations
- Recursive ray tracing
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 ;
}
```

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 const linear quadratic (like in OpenGL)
    \[ L = \frac{L_a}{\text{const} + \text{lin} \cdot d + \text{quad} \cdot d^2} \]
- 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
\[ I = K_e + K_s + \sum_{i=1}^{n} L_i (K_d \max (I_i \cdot n, 0) + K_s (\max(h_i \cdot n, 0))^s) \]
- 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)
- Ray-object intersections
- Ray-tracing transformed objects
- Lighting calculations
- Recursive ray tracing

Mirror Reflections/Refractions

Virtual Viewpoint
Virtual Screen
Objects

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

\[ I = K_c + K_s + \sum L \frac{1}{n} (K_r \text{max}(l \cdot n, 0) + K_t (\text{max}(l \cdot n, 0)^2) + K_a I_a + K_f I_f) \]

- Highlighted terms are recursive specularities [mirror reflections] and transmission
- 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

Spatial Hierarchies (Oct-trees, kd trees, BSP trees)

Bounding Volume Hierarchies 1

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

Bounding Volume Hierarchies 2

- Use hierarchy to accelerate ray intersections
  - Intersect node contents only if hit bounding volume
Bounding Volume Hierarchies 3

- Sort hits & detect early termination

```plaintext
FindIntersection(Ray ray, Node node)

  // Find intersections with child node bounding volumes
  // Sort intersections front to back
  // Process intersections (checking for early termination)
  min_t = infinity;
  for each intersected child i {
    if (min_t < t0[i]) break;
    shape_t = FindIntersection(ray, child);
    if (shape_t < min_t) min_t = shape_t;
  }
  return min_t;
```

Acceleration Structures: Grids

Uniform Grid: Problems

- Potential problem:
  - How choose suitable grid resolution?

Octree

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

Octree traversal

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

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
  - NVIDIA now has CUDA-based raytracing API!

[Purcell et al. 2002, 2003]
http://graphics.stanford.edu/papers/photongfx