CS 283, Lecture 2: Basic Ray Tracing

Ravi Ramamoorthi

http://inst.eecs.berkeley.edu/~cs283/fa10

Basics
- Start working on raytracer assignment (if necessary)
- First 3 lectures cover basic topics
  - Rendering: Raytracing (required for homeworks 2, 3)
  - Fourier Analysis and Sampling
  - 3D objects and meshes
- Then we start main part of course
  - Meshes and assignment 1
- This lecture review for most of you
  - But needed to bring everyone up to speed
  - Will go through it fast since very basic material

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 Casting

- Virtual Viewpoint
- Virtual Screen
- Objects
- Ray misses all objects: Pixel colored black
- Ray intersects object: shade using color, lights, materials
- Multiple intersections: Use closest one (as does OpenGL)

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 dim 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

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 = a / |a|
u = b x w / |b x w|
v = w x u
```

Camera coordinate frame

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

\[ \alpha = \tan\left( \frac{\text{fov} \times width}{2} \right) \left( \frac{1 - (\text{width}/2)}{\text{width}/2} \right) \]

\[ \beta = \tan\left( \frac{\text{fov} \times height}{2} \right) \left( \frac{1 - (\text{height}/2)}{\text{height}/2} \right) \]

Outline

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

Outline in Code

```
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

\[ \text{ray} = \vec{P} = \vec{P}_0 + \vec{P}_t \]

\[ \text{sphere} = (\vec{P} - \vec{C}) \cdot (\vec{P} - \vec{C}) - r^2 = 0 \]

Substitute

\[ \text{ray} = \vec{P} = \vec{P}_0 + \vec{P}_t \]

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

Simplify

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

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

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

Ray-Triangle Intersection

- One approach: Ray-Plane intersection, then check if inside triangle
- Plane equation:
  \[ \text{plane} = \vec{n} \cdot \vec{A} - \vec{A} \cdot \vec{n} = 0 \]
- Combine with ray equation:
  \[ \vec{r}_{ray} = \vec{P} = \vec{P}_0 + \vec{P}_t t \]
  \[ (\vec{P}_0 + \vec{P}_t t) \cdot \vec{n} = \vec{A} \cdot \vec{n} \]
  \[ t = \frac{\vec{A} \cdot \vec{n} - \vec{P}_0 \cdot \vec{n}}{\vec{P}_t \cdot \vec{n}} \]

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)

\[ P = a \vec{A} + \beta \vec{B} + \gamma \vec{C} \]
\[ a \geq 0, \beta \geq 0, \gamma \geq 0 \]
\[ a + \beta + \gamma = 1 \]

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
- Many references. For example, chapter in Glassner introduction to ray tracing (see me if interested)
Ray Scene Intersection

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

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

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

Virtual Viewpoint
Virtual Screen
Objects

Light Source

Shadow ray to light is unblocked: object visible
Shadow ray to light is blocked: object in shadow

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

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

• 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
Generate reflected ray in mirror direction,
Get reflections and refractions of objects

Recursive Shading Model

\[ I = K_a + K_c + \sum_{i=1}^{s} L_i (K_s \max (l \cdot n, 0) + K_r (max(h \cdot n, 0))^2) + K_f I_f \]

- 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)

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 ^2 Color of reflected ray

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

```c
FindIntersection(Ray ray, Node node)
{
    for each intersected child i {
        if (min_t < t) break;
        shape_i = FindIntersection(ray, child_i);
        if (shape_i < min_t) min_t = shape_i;
    }
    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 with cells

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