Foundations of Computer Graphics
Online Lecture 10: Ray Tracing 2 – Nuts and Bolts

Camera Ray Casting

Ravi Ramamoorthi

Outline

- Camera Ray Casting (choose 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

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

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

From earlier lecture on deriving gluLookAt
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 $u$

\[
\begin{align*}
    w &= a \\
    u &= \frac{b \times w}{|b \times w|} \\
    v &= w \times u
\end{align*}
\]

From basic math lecture - Vectors: Orthonormal Basis Frames

Canonical viewing geometry

\[
\begin{align*}
    \alpha &= \tan\left(\frac{\text{fovx}}{2}\right) \times \left(1 - \left(\text{width} / 2\right) / \text{width} / 2\right) \\
    \beta &= \tan\left(\frac{\text{fovy}}{2}\right) \times \left(\text{height} / 2 - i\right) / \text{height} / 2
\end{align*}
\]

Foundations of Computer Graphics

Online Lecture 10: Ray Tracing 2 – Nuts and Bolts
Ray-Object Intersections

Ravi Ramamoorthi
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

\[ \text{ray} = P_0 + Pt \]
\[ \text{sphere} = (P - C) \cdot (P - C) - r^2 = 0 \]

Substitute

\[ \text{ray} = \hat{P} = P_0 + \hat{P}t \]
\[ \text{sphere} = (\hat{P} - C) \cdot (\hat{P} - C) - r^2 = 0 \]

Simplify

\[ t^2 (P_1 \cdot P_1) + 2t (P_1 \cdot (P_0 - C) + (\hat{P}_0 - C) \cdot (\hat{P}_0 - C) - r^2 = 0 \]
Ray-Sphere Intersection

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

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

Ray-Triangle Intersection

- One approach: Ray-Plane intersection, then check if inside triangle

Plane equation:

\[ \text{plane} = \vec{P} \cdot \vec{n} - \vec{A} \cdot \vec{n} = 0 \]

Ray-Triangle Intersection

- One approach: Ray-Plane intersection, then check if inside triangle

Plane equation:

\[ \vec{n} = \frac{(\vec{C} - \vec{A}) \times (\vec{B} - \vec{A})}{|| (\vec{C} - \vec{A}) \times (\vec{B} - \vec{A}) ||} \]

Ray-Triangle Intersection

- One approach: Ray-Plane intersection, then check if inside triangle

Plane equation:

\[ \text{plane} = \vec{P} \cdot \vec{n} - \vec{A} \cdot \vec{n} = 0 \]

Combine with ray equation

\[ \text{ray} = \vec{P}_0 + \vec{P}_1 t \]

\[ (\vec{P}_0 + \vec{P}_1) \cdot \vec{n} = \vec{A} \cdot \vec{n} \]
Ray inside Triangle

* Once intersect with plane, need to find if in triangle
* Many possibilities for triangles, general polygons
* We find parametrically [barycentric coordinates]. Also useful for other applications (texture mapping)

\[
P = \alpha A + \beta B + \gamma C \\
\alpha \geq 0, \beta \geq 0, \gamma \geq 0 \\
\alpha + \beta + \gamma = 1
\]

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

Ray Scene Intersection

Intersection (ray, scene) {
    mindist = infinity; hitobject = NULL;
    For each object in scene { // Find closest intersection; test all objects
        t = Intersect (ray, object) ;
        if (t > 0 && t < mindist) // closer than previous closest object
            mindist = t ; hitobject = object ;
    }
    return IntersectionInfo(mindist, hitobject) ; // may already be in Intersect()
}

Outline

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

Ray-Tracing Transformed Objects

We have an optimized ray-sphere test
* But we want to ray trace an ellipsoid...

Solution: Ellipsoid transforms sphere
* Apply inverse transform to ray, use ray-sphere
* Allows for instancing (traffic jam of cars)
* Same idea for other primitives
Transformed Objects

- Consider a general 4x4 transform \( M \) (matrix stacks)
- 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 \( Mp \)
  - 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

- 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
  - Attenuation const linear quadratic
  \[ L = \frac{L_0}{\text{const} + \text{lin} \cdot d + \text{quad} \cdot d^2} \]
- Per light model parameters
  - Directional light (direction, RGB parameters)
  - Point light (location, RGB parameters)
  - Some differences from HW 2 syntax

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

### Shading Model
\[ I = K_a + K_e + \sum_i V_i L_i (K_d \max (l_i \cdot n, 0) + K_s (\max (h_i \cdot n, 0))^s) \]
- 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

### Foundations of Computer Graphics
Online Lecture 10: Ray Tracing 2 – Nuts and Bolts

*Recursive Ray Tracing*

Ravi Ramamoorthi

---

### 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

\[ I = K_s + K_t + \sum \forall \mathbf{L}_i (K_d \max (I \cdot n, 0) + K_s \max (\mathbf{h} \cdot \mathbf{n}, 0)^s) + K_t + K_r \]

- Highlighted terms are recursive specularities [mirror reflections] and transmission (latter is extra)
- 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)

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