Foundations of Computer Graphics

Online Lecture 10: Ray Tracing 2 – Nuts and Bolts

Camera Ray Casting

Ravi Ramamoorthi

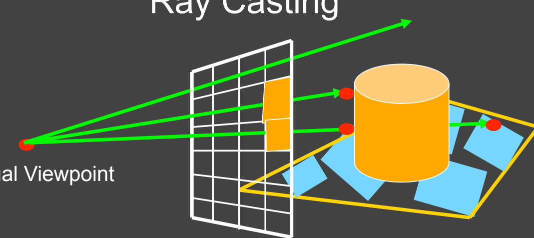
Outline

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

Outline in Code

```
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



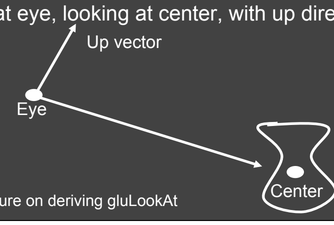
Multiple intersections: Use closest one (as does OpenGL)

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 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



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**

$$w = \frac{a}{\|a\|}$$

$$u = \frac{b \times w}{\|b \times w\|}$$

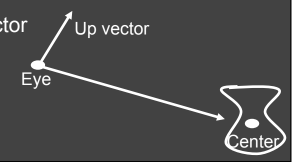
$$v = w \times u$$

From basic math lecture - Vectors: Orthonormal Basis Frames

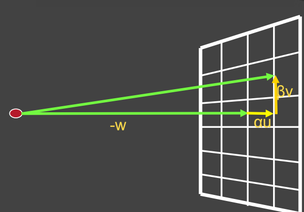
Constructing a coordinate frame

$$w = \frac{a}{\|a\|} \qquad u = \frac{b \times w}{\|b \times w\|} \qquad v = w \times u$$

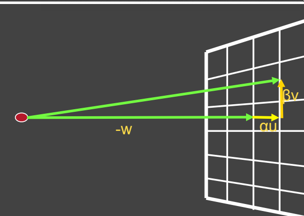
- We want to position camera at origin, looking down –Z dirn
- Hence, vector **a** is given by **eye** – **center**
- The vector **b** is simply the **up** vector



Canonical viewing geometry

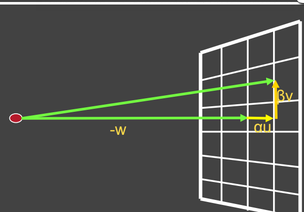


Canonical viewing geometry



$$\alpha = \tan\left(\frac{fovx}{2}\right) \times \left(\frac{j - (width/2)}{width/2}\right) \qquad \beta = \tan\left(\frac{fovy}{2}\right) \times \left(\frac{(height/2) - i}{height/2}\right)$$

Canonical viewing geometry



$$ray = eye + \frac{\alpha u + \beta v - w}{|\alpha u + \beta v - w|}$$

$$\alpha = \tan\left(\frac{fovx}{2}\right) \times \left(\frac{j - (width/2)}{width/2}\right) \qquad \beta = \tan\left(\frac{fovy}{2}\right) \times \left(\frac{(height/2) - i}{height/2}\right)$$

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

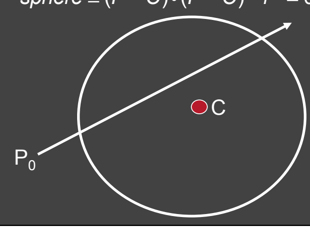
Outline in Code

```
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

$$ray \equiv \vec{P} = \vec{P}_0 + \vec{P}_1 t$$

$$sphere \equiv (\vec{P} - \vec{C}) \bullet (\vec{P} - \vec{C}) - r^2 = 0$$



Ray-Sphere Intersection

$$ray \equiv \vec{P} = \vec{P}_0 + \vec{P}_1 t$$

$$sphere \equiv (\vec{P} - \vec{C}) \bullet (\vec{P} - \vec{C}) - r^2 = 0$$

Substitute

Ray-Sphere Intersection

$$ray \equiv \vec{P} = \vec{P}_0 + \vec{P}_1 t$$

$$sphere \equiv (\vec{P} - \vec{C}) \bullet (\vec{P} - \vec{C}) - r^2 = 0$$

Substitute

$$ray \equiv \vec{P} = \vec{P}_0 + \vec{P}_1 t$$

$$sphere \equiv (\vec{P}_0 + \vec{P}_1 t - \vec{C}) \bullet (\vec{P}_0 + \vec{P}_1 t - \vec{C}) - r^2 = 0$$

Simplify

Ray-Sphere Intersection

$$ray \equiv \vec{P} = \vec{P}_0 + \vec{P}_1 t$$

$$sphere \equiv (\vec{P} - \vec{C}) \bullet (\vec{P} - \vec{C}) - r^2 = 0$$

Substitute

$$ray \equiv \vec{P} = \vec{P}_0 + \vec{P}_1 t$$

$$sphere \equiv (\vec{P}_0 + \vec{P}_1 t - \vec{C}) \bullet (\vec{P}_0 + \vec{P}_1 t - \vec{C}) - r^2 = 0$$

Simplify

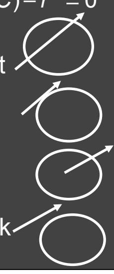
$$t^2 (\vec{P}_1 \bullet \vec{P}_1) + 2t\, \vec{P}_1 \bullet (\vec{P}_0 - \vec{C}) + (\vec{P}_0 - \vec{C}) \bullet (\vec{P}_0 - \vec{C}) - r^2 = 0$$

Ray-Sphere Intersection

$$t^2(\vec{P}_1 \bullet \vec{P}_1) + 2t\,\vec{P}_1 \bullet (\vec{P}_0 - \vec{C}) + (\vec{P}_0 - \vec{C}) \bullet (\vec{P}_0 - \vec{C}) - r^2 = 0$$

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)



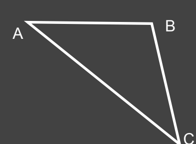
Ray-Sphere Intersection

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

$$normal = \frac{\vec{P} - \vec{C}}{\left|\vec{P} - \vec{C}\right|}$$

Ray-Triangle Intersection

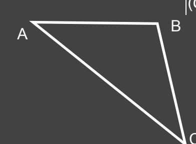
- One approach: Ray-Plane intersection, then check if inside triangle
- Plane equation:



Ray-Triangle Intersection

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

$$n = \frac{(C-A)\times(B-A)}{\left|(C-A)\times(B-A)\right|}$$

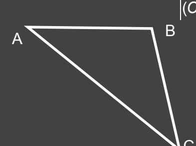


Ray-Triangle Intersection

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

$$n = \frac{(C-A)\times(B-A)}{\left|(C-A)\times(B-A)\right|}$$

$$plane \equiv \vec{P} \bullet \vec{n} - \vec{A} \bullet \vec{n} = 0$$



Ray-Triangle Intersection

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

$$n = \frac{(C-A)\times(B-A)}{\left|(C-A)\times(B-A)\right|}$$

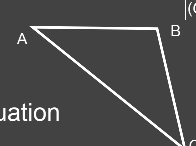
$$plane \equiv \vec{P} \bullet \vec{n} - \vec{A} \bullet \vec{n} = 0$$

- Combine with ray equation

$$ray \equiv \vec{P} = \vec{P}_0 + \vec{P}_1 t$$

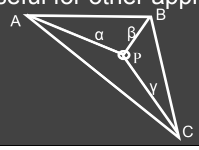
$$(\vec{P}_0 + \vec{P}_1 t) \bullet \vec{n} = \vec{A} \bullet \vec{n}$$

$$t = \frac{\vec{A} \bullet \vec{n} - \vec{P}_0 \bullet \vec{n}}{\vec{P}_1 \bullet \vec{n}}$$



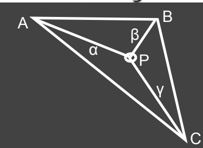
Ray inside Triangle

- 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)



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

Ray inside Triangle



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

$$P - A = \beta(B - A) + \gamma(C - A)$$
$$0 \leq \beta \leq 1 \;,\; 0 \leq \gamma \leq 1$$
$$\beta + \gamma \leq 1$$

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 $M^{-t}n$. Do all this before lighting

Foundations of Computer Graphics

Online Lecture 10: Ray Tracing 2 – Nuts and Bolts

Lighting Calculations

Ravi Ramamoorthi

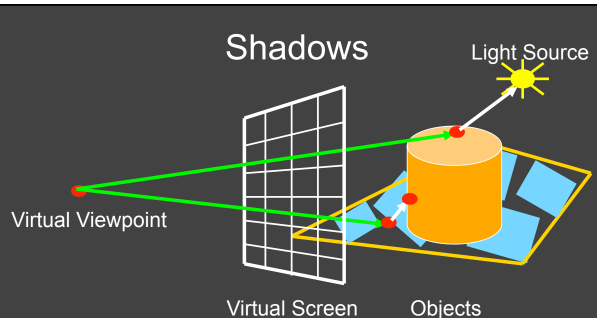
Outline

- 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 ;
}
```

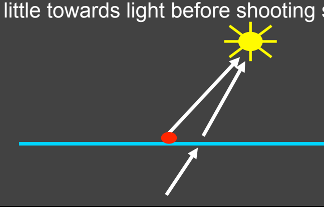
Shadows



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

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 $L = \frac{L_0}{const + lin * d + quad * d^2}$
- 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

$$I = K_a + K_e + \sum_{i=1}^{n} V_i L_i (K_d \max(l_i \bullet n, 0) + K_s (\max(h_i \bullet n, 0))^s)$$

- 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)

Foundations of Computer Graphics

Online Lecture 10: Ray Tracing 2 – Nuts and Bolts

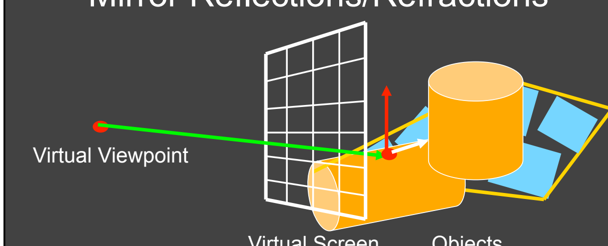
Recursive Ray Tracing

Ravi Ramamoorthi

Outline

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

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

$$I = K_a + K_e + \sum_{i=1}^{n} V_i L_i (K_d \max(l_i \bullet n, 0) + K_s(\max(h_i \bullet n, 0))^s) + K_s I_R + K_T I_T$$

- 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