Ray Tracing

Week 3

Acknowledgement: The course slides are adapted from the slides prepared by Steve Marschner of Cornell University
Ray tracing idea
Ray tracing idea
Ray tracing idea
Ray tracing idea
Ray tracing algorithm

\textbf{for} each pixel { 
  compute viewing ray 
  intersect ray with scene 
  compute illumination at visible point 
  put result into image 
}\}
Generating eye rays

- Use window analogy directly
Generating eye rays

- Use window analogy directly
Generating eye rays

- Use window analogy directly
Generating eye rays

• Use window analogy directly
Durer’s Ray casting machine

• Albrecht Durer, 16th century

Source: F. Durand
Durer’s Ray casting machine

- Albrecht Durer, 16th century

Source: F. Durand
Generating eye rays

- Use window analogy directly

ORTHOGRAPHIC

PERSPECTIVE

view rect

viewpoint

pixel position

viewing ray
Vector math review

- Vectors and points
- Vector operations
  - addition
  - scalar product
- More products
  - dot product
  - cross product
- Bases and orthogonality
Generating eye rays—orthographic

• Just need to compute the view plane point \( s \):

\[
p = s; \quad d = d_v \\
r(t) = p + td
\]

– but where exactly is the view rectangle?
Generating eye rays—orthographic

- Positioning the view rectangle
  - establish three vectors to be *camera basis*: \( \mathbf{u}, \mathbf{v}, \mathbf{w} \)
  - view rectangle is in \( \mathbf{u} - \mathbf{v} \) plane, specified by \( l, r, t, b \)
  - now ray generation is easy:

\[
\begin{align*}
  \mathbf{s} &= \mathbf{e} + u\mathbf{u} + v\mathbf{v} \\
  \mathbf{p} &= \mathbf{s}; \quad \mathbf{d} = -\mathbf{w} \\
  \mathbf{r}(t) &= \mathbf{p} + t\mathbf{d}
\end{align*}
\]
Generating eye rays—perspective

- View rectangle needs to be away from viewpoint
- Distance is important: “focal length” of camera
  - still use camera frame but position view rect away from viewpoint
  - ray origin always $e$
  - ray direction now controlled by $s$

\[ d = s - e \]

\[ p = e \]

\[ r(t) = p + td \]
Generating eye rays—perspective

- Compute $s$ in the same way; just subtract $dw$
  - coordinates of $s$ are $(u, v, -d)$

\[ s = e + uu + vv - dw \]

\[ p = e; \ d = s - e \]

\[ r(t) = p + td \]
Pixel-to-image mapping

- One last detail: \((u, v)\) coords of a pixel

\[
u = l + (r - l)(i + 0.5)/n_x\]
\[
v = b + (t - b)(j + 0.5)/n_y\]
Ray intersection
Ray: a half line

- Standard representation: point \( p \) and direction \( d \)

\[
\mathbf{r}(t) = p + td
\]

- this is a \textit{parametric equation} for the line
- lets us directly generate the points on the line
- if we restrict to \( t > 0 \) then we have a ray
- note replacing \( d \) with \( a\mathbf{d} \) doesn’t change ray \((a > 0)\)
Ray-sphere intersection: algebraic

- Condition 1: point is on ray
  \[ r(t) = p + td \]

- Condition 2: point is on sphere
  - assume unit sphere; see Shirley or notes for general
    \[ \|x\| = 1 \Leftrightarrow \|x\|^2 = 1 \]
    \[ f(x) = x \cdot x - 1 = 0 \]

- Substitute:
  \[ (p + td) \cdot (p + td) - 1 = 0 \]
  - this is a quadratic equation in \( t \)
Ray-sphere intersection: algebraic

- Solution for $t$ by quadratic formula:

$$t = \frac{-d \cdot p \pm \sqrt{(d \cdot p)^2 - (d \cdot d)(p \cdot p - 1)}}{d \cdot d}$$

$$t = -d \cdot p \pm \sqrt{(d \cdot p)^2 - p \cdot p + 1}$$

- simpler form holds when $d$ is a unit vector but we won’t assume this in practice (reason later)
- I’ll use the unit-vector form to make the geometric interpretation
Ray-sphere intersection: geometric

\[\begin{align*}
t_m &= -p \cdot d \\
l_m^2 &= p \cdot p - (p \cdot d)^2 \\
\Delta t &= \sqrt{1 - l_m^2} \\
&= \sqrt{(p \cdot d)^2 - p \cdot p + 1} \\
t_{0,1} &= t_m \pm \Delta t = -p \cdot d \pm \sqrt{(p \cdot d)^2 - p \cdot p + 1}
\end{align*}\]
Ray-box intersection

- Could intersect with 6 faces individually
- Better way: box is the intersection of 3 slabs
Ray-box intersection

- Could intersect with 6 faces individually
- Better way: box is the intersection of 3 slabs
Ray-box intersection

- Could intersect with 6 faces individually
- Better way: box is the intersection of 3 slabs
Ray-box intersection

- Could intersect with 6 faces individually
- Better way: box is the intersection of 3 slabs
Ray-slab intersection

- 2D example
- 3D is the same!
Ray-slab intersection

- 2D example
- 3D is the same!
Ray-slab intersection

- 2D example
- 3D is the same!

\[
p_x + t_{x_{\text{min}}} d_x = x_{\text{min}} \\
t_{x_{\text{min}}} = \frac{(x_{\text{min}} - p_x)}{d_x}
\]
Ray-slab intersection

- 2D example
- 3D is the same!

\[ p_x + t_{x_{\text{min}}} d_x = x_{\text{min}} \]
\[ t_{x_{\text{min}}} = \frac{(x_{\text{min}} - p_x)}{d_x} \]
\[ p_y + t_{y_{\text{min}}} d_y = y_{\text{min}} \]
\[ t_{y_{\text{min}}} = \frac{(y_{\text{min}} - p_y)}{d_y} \]
Intersecting intersections

- Each intersection is an interval
- Want last entry point and first exit point

\[
\begin{align*}
t_{\text{min}} &= \max(t_{x\text{min}}, t_{y\text{min}}) \\
t_{\text{max}} &= \min(t_{x\text{max}}, t_{y\text{max}})
\end{align*}
\]

Shirley fig. 10.16
Ray-triangle intersection

- Condition 1: point is on ray
  \[ r(t) = p + td \]

- Condition 2: point is on plane
  \[ (x - a) \cdot n = 0 \]

- Condition 3: point is on the inside of all three edges

- First solve 1&2 (ray–plane intersection)
  - substitute and solve for \( t \):
  \[
  (p + td - a) \cdot n = 0
  \]
  \[
  t = \frac{(a - p) \cdot n}{d \cdot n}
  \]
Ray-triangle intersection

- In plane, triangle is the intersection of 3 half spaces
Ray-triangle intersection

- In plane, triangle is the intersection of 3 half spaces
Ray-triangle intersection

- In plane, triangle is the intersection of 3 half spaces
Ray-triangle intersection

- In plane, triangle is the intersection of 3 half spaces
Inside-edge test

- Need outside vs. inside
- Reduce to clockwise vs. counterclockwise
  - vector of edge to vector to $x$
- Use cross product to decide
Ray-triangle intersection

\[(b - a) \times (x - a) \cdot n > 0\]
\[(c - b) \times (x - b) \cdot n > 0\]
\[(a - c) \times (x - c) \cdot n > 0\]
Ray-triangle intersection

- a more efficient method
  - use barycentric coordinates

Source: F. Durand
Barycentric definition of a plane

- $P(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c$
- with $\alpha + \beta + \gamma = 1$

Source: F. Durand
Barycentric definition of a triangle

- \( P(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c \)
  with \( \alpha + \beta + \gamma = 1 \)

- \( 0 < \alpha < 1 \)
  \( 0 < \beta < 1 \)
  \( 0 < \gamma < 1 \)

Source: F. Durand
Given P, how can we compute $\alpha$, $\beta$, $\gamma$?

- Compute the areas of the opposite subtriangle
  - Ratio with complete area
    \[
    \alpha = \frac{A_a}{A}, \quad \beta = \frac{A_b}{A} \quad \gamma = \frac{A_c}{A}
    \]

Use signed areas for points outside the triangle

Source: F. Durand
Intuition behind area formula

- P is barycenter of a and Q
- A is the interpolation coefficient on aQ
- All points on line parallel to bc have the same $\alpha$
- All such Ta triangles have same height/area

Source: F. Durand
Simplify

- Since $\alpha + \beta + \gamma = 1$
  we can write $\alpha = 1 - \beta - \gamma$
- $P(\beta, \gamma) = (1 - \beta - \gamma) a + \beta b + \gamma c$

Source: F. Durand
Simplify

- \( P(\beta, \gamma) = (1 - \beta - \gamma) a + \beta b + \gamma c \)
- \( P(\beta, \gamma) = a + \beta(b-a) + \gamma(c-a) \)
- Non-orthogonal coordinate system of the plane

Source: F. Durand
How do we use it for intersection?

Insert ray equation into barycentric expression of triangle

\[ P(t) = a + \beta (b-a) + \gamma (c-a) \]

Intersection if \( \beta + \gamma < 1; \quad 0 < \beta \) and \( 0 < \gamma \)

Source: F. Durand
Intersection

\[ R_x + tD_x = a_x + \beta (b_x - a_x) + \gamma (c_x - a_x) \]
\[ R_y + tD_y = a_y + \beta (b_y - a_y) + \gamma (c_y - a_y) \]
\[ R_z + tD_z = a_z + \beta (b_z - a_z) + \gamma (c_z - a_z) \]

Source: F. Durand
Matrix form

\[
\begin{bmatrix}
  a_x - b_x & a_x - c_x & D_x \\
  a_y - b_y & a_y - c_y & D_y \\
  a_z - b_z & a_z - c_z & D_z \\
\end{bmatrix}
\begin{bmatrix}
  \beta \\
  \gamma \\
  t \\
\end{bmatrix}
= 
\begin{bmatrix}
  a_x - R_x \\
  a_y - R_y \\
  a_z - R_z \\
\end{bmatrix}
\]

Source: F. Durand
Cramer's rule

\[
\beta = \frac{\begin{vmatrix}
 a_x - R_x & a_x - c_x & D_x \\
 a_y - R_y & a_y - c_y & D_y \\
 a_z - R_z & a_z - c_z & D_z \\
\end{vmatrix}}{|A|}
\]

\[
\gamma = \frac{\begin{vmatrix}
 a_x - b_x & a_x - R_x & D_x \\
 a_y - b_y & a_y - R_y & D_y \\
 a_z - b_z & a_z - R_z & D_z \\
\end{vmatrix}}{|A|}
\]

\[
t = \frac{\begin{vmatrix}
 a_x - b_x & a_x - c_x & a_x - R_x \\
 a_y - b_y & a_y - c_y & a_y - R_y \\
 a_z - b_z & a_z - c_z & a_z - R_z \\
\end{vmatrix}}{|A|}
\]

Source: F. Durand
Advantage

Efficient

Store no plane equation

Get the barycentric coordinates for free

Useful for interpolation, texture mapping

Source: F. Durand
Image so far

- With eye ray generation and sphere intersection

Surface s = new Sphere((0.0, 0.0, 0.0), 1.0);
for 0 <= iy < ny
    for 0 <= ix < nx {
        ray = camera.getRay(ix, iy);
        hitSurface, t = s.intersect(ray, 0, +inf)
        if hitSurface is not null
            image.set(ix, iy, white);
    }
Intersection against many shapes

• The basic idea is:

```java
Group.intersect (ray, tMin, tMax) {
    tBest = +inf; firstSurface = null;
    for surface in surfaceList {
        hitSurface, t = surface.intersect(ray, tMin, tBest);
        if hitSurface is not null {
            tBest = t;
            firstSurface = hitSurface;
        }
    }
    return hitSurface, tBest;
}
```

– this is linear in the number of shapes but there are sublinear methods (acceleration structures)
Image so far

- With eye ray generation and scene intersection

```java
for 0 <= iy < ny
    for 0 <= ix < nx {
        ray = camera.getRay(ix, iy);
        c = scene.trace(ray, 0, +inf);
        image.set(ix, iy, c);
    }

... 

Scene.trace(ray, tMin, tMax) {
    surface, t = surfs.intersect(ray, tMin, tMax);
    if (surface != null) return surface.color();
    else return black;
}
```
Shading

• Compute light reflected toward camera

• Inputs:
  – eye direction
  – light direction
    (for each of many lights)
  – surface normal
  – surface parameters
    (color, shininess, …)
Diffuse reflection

- Light is scattered uniformly in all directions
  - the surface color is the same for all viewing directions
- Lambert's cosine law

\[ \text{Light per unit area is proportional to } \cos \theta = l \cdot n \]
Lambertian shading

• Shading independent of view direction

\[ L_d = k_d I \max(0, \mathbf{n} \cdot \mathbf{l}) \]

illumination from source
diffuse coefficient
diffusely reflected light
Lambertian shading

- Produces matte appearance
Diffuse shading
### Image so far

Scene.trace(Ray ray, tMin, tMax) {
    surface, t = hit(ray, tMin, tMax);
    if surface is not null {
        point = ray.evaluate(t);
        normal = surface.getNormal(point);
        return surface.shade(ray, point,
            normal, light);
    }
    else return backgroundColor;
}

...

Surface.shade(ray, point, normal, light) {
    v = −normalize(ray.direction);
    l = normalize(light.pos − point);
    // compute shading
}
Shadows

- Surface is only illuminated if nothing blocks its view of the light.
- With ray tracing it’s easy to check
  - just intersect a ray with the scene!
Surface.shade(ray, point, normal, light) {
    shadRay = (point, light.pos – point);
    if (shadRay not blocked) {
        v = –normalize(ray.direction);
        l = normalize(light.pos – point);
        // compute shading
    }
    return black;
}
Multiple lights

• Important to fill in black shadows
• Just loop over lights, add contributions
• Ambient shading
  – black shadows are not really right
  – one solution: dim light at camera
  – alternative: add a constant “ambient” color to the shading…
Image so far

shade(ray, point, normal, lights) {
    result = ambient;
    for light in lights {
        if (shadow ray not blocked) {
            result += shading contribution;
        }
    }
    return result;
}
Specular shading (Blinn-Phong)

- Intensity depends on view direction
  - bright near mirror configuration
Specular shading (Blinn-Phong)

- Close to mirror ⇔ half vector near normal
  - Measure “near” by dot product of unit vectors

\[
\begin{align*}
h &= \text{bisector}(v, l) \\
   &= \frac{v + l}{\|v + l\|} \\
L_s &= k_s I \max(0, \cos \alpha)^p \\
   &= k_s I \max(0, n \cdot h)^p
\end{align*}
\]
Phong model—plots

• Increasing $n$ narrows the lobe

Fig. 16.9 Different values of $\cos^n \alpha$ used in the Phong illumination model.
Specular shading

\( k_s \)

\( p \)
Diffuse + Phong shading
Ambient shading

- Shading that does not depend on anything
  - add constant color to account for disregarded illumination and fill in black shadows

\[ L_a = k_a I_a \]

reflected ambient light

ambient coefficient
Putting it together

• Usually include ambient, diffuse, Phong in one model

\[ L = L_a + L_d + L_s \]
\[ = k_a I_a + k_d I \max(0, n \cdot l) + k_s I \max(0, n \cdot h)^p \]

• The final result is the sum over many lights

\[ L = k_a I_a + \sum_{i=1}^{N} \left[ (L_d)_i + (L_s)_i \right] \]
\[ = k_a I_a + \sum_{i=1}^{N} \left[ k_d I_i \max(0, n \cdot l_i) + k_s I_i \max(0, n \cdot h_i)^p \right] \]
Mirror reflection

• Consider perfectly shiny surface
  – there isn’t a highlight
  – instead there’s a reflection of other objects

• Can render this using recursive ray tracing
  – to find out mirror reflection color, ask what color is seen from surface point in reflection direction
  – already computing reflection direction for Phong…

• “Glazed” material has mirror reflection and diffuse

\[ L = L_a + L_d + L_m \]

  – where \( L_m \) is evaluated by tracing a new ray
Mirror reflection

• Intensity depends on view direction
  – reflects incident light from mirror direction

\[
r = v + 2((n \cdot v)n - v)
= 2(n \cdot v)n - v
\]
Diffuse + mirror reflection (glazed)

(glazed material on floor)
Ray tracer architecture 101

• You want a class called Ray
  – point and direction; evaluate(t)
  – possible: tMin, tMax

• Some things can be intersected with rays
  – individual surfaces
  – groups of surfaces (acceleration goes here)
  – the whole scene
  – make these all subclasses of Surface
  – limit the range of valid t values (e.g. shadow rays)

• Once you have the visible intersection, compute the color
  – may want to separate shading code from geometry
  – separate class: Material (each Surface holds a reference to one)
  – its job is to compute the color
Architectural practicalities

• Return values
  – surface intersection tends to want to return multiple values
    • $t$, surface or shader, normal vector, maybe surface point
  – in many programming languages (e.g. Java) this is a pain
  – typical solution: an intersection record
    • a class with fields for all these things
    • keep track of the intersection record for the closest intersection
    • be careful of accidental aliasing (which is very easy if you’re new to Java)

• Efficiency
  – what objects are created for every ray? try to find a place for them where you can reuse them.
  – Shadow rays can be cheaper (any intersection will do, don’t need closest)
  – but: “First Get it Right, Then Make it Fast”