10. Ray Tracing

- Generate a ray for each pixel and trace the ray backwards to its origin.
Sample Result: back side and right side of the room are both mirrors. So you would see reflections of the room on both sides.
10.1 Basic Ray Tracing Algorithm

- Trace each ray to FIRST object hit by ray
- Structure of basic ray tracing program:

  Compute $u$, $v$, $w$ basis vectors
  for each pixel do
    compute viewing ray
    find first object hit by ray and its surface normal $n$
    use material, light, and $n$ to compute pixel color
10.2 Computing Viewing Rays

- Parametric representation of a ray:
  \[ P(t) = e + t(s - e), \]
  where \( e \) is eye point and \( s \) is a point on the screen.
10.2 Computing Viewing Rays

Computation of $\mathbf{s}$:

if resolution of the screen is $n_x \times n_y$ and coordinates of the 3D window on the view plan ($w = n$) are ($l$, $b$, $n$) and ($r$, $t$, $n$) defined in the uvw coordinate system, then

$$
\mathbf{s} = \mathbf{e} + u_s \mathbf{u} + v_s \mathbf{v} + w_s \mathbf{w}
$$

where

$$
w_s = n \quad u_s = l + (r - l) \frac{i + 0.5}{n_x} \quad v_s = b + (t - b) \frac{j + 0.5}{n_y}
$$

and ($i$, $j$) are pixel indices satisfying

$$
0 \leq i \leq n_x - 1 \quad 0 \leq j \leq n_y - 1
$$
10.2 Computing Viewing Rays

Why?

(i, j)  \quad (l, b, n)  \quad (r, t, n)
10.2 Computing Viewing Rays

- in matrix form:

\[
\begin{bmatrix}
  x_s \\
  y_s \\
  z_s \\
  1
\end{bmatrix} = \begin{bmatrix}
  1 & 0 & 0 & x_e \\
  0 & 1 & 0 & y_e \\
  0 & 0 & 1 & z_e \\
  0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
  x_u & x_v & x_w & 0 \\
  y_u & y_v & y_w & 0 \\
  z_u & z_v & z_w & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
  u_s \\
  v_s \\
  w_s \\
  1
\end{bmatrix}
\]
10.3 Ray-Object Intersection: ray-sphere intersection

Ray: \( \mathbf{p}(t) = \mathbf{e} + t\mathbf{d} \)

Sphere: \( f(\mathbf{p}) = (x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 - R^2 = 0 \)

Set \( f(\mathbf{p}(t)) = f(\mathbf{e} + t\mathbf{d}) = 0 \). We get

\[
(\mathbf{p} - \mathbf{c}) \cdot (\mathbf{p} - \mathbf{c}) - R^2 = 0
\]

\[
(\mathbf{e} + t\mathbf{d} - \mathbf{c}) \cdot (\mathbf{e} + t\mathbf{d} - \mathbf{c}) - R^2 = 0
\]

\[
(\mathbf{d} \cdot \mathbf{d})t^2 + 2\mathbf{d} \cdot (\mathbf{e} - \mathbf{c})t + (\mathbf{e} - \mathbf{c}) \cdot (\mathbf{e} - \mathbf{c}) - R^2 = 0
\]

a quadratic equation in \( t \). Can solve it for \( t \).

Sphere normal at \( \mathbf{p} \) is \( \mathbf{n} = 2(\mathbf{p} - \mathbf{c}) \)
10.3 Ray-Object Intersection: ray-triangle intersection

Ray: \( \mathbf{p}(t) = \mathbf{e} + td \)

Plane containing triangle:

\[ f(\beta, \gamma) = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a}) \]

where \( \mathbf{a}, \mathbf{b} \) and \( \mathbf{c} \) are vertices of the triangle

Set

\[ \mathbf{e} + td = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a}) \] (*)

The hit point is in the triangle iff \( \beta > 0, \gamma > 0 \) and \( \beta + \gamma < 1 \).
10.3 Ray-Object Intersection:
ray- triangle intersection

Ray: \( p(t) = e + td \)

Plane containing triangle:

\[
f(\beta, \gamma) = a + \beta(b - a) + \gamma(c - a)
\]

where \( a, b \) and \( c \) are vertices of the triangle

Rewrite (*) as

\[
\begin{bmatrix}
x_a - x_b & x_a - x_c & x_d \\
y_a - y_b & y_a - y_c & y_d \\
z_a - z_b & z_a - z_c & z_d
\end{bmatrix}
\begin{bmatrix}
\beta \\
\gamma \\
t
\end{bmatrix} =
\begin{bmatrix}
x_a - x_e \\
y_a - y_e \\
z_a - z_e
\end{bmatrix}
\]

then use Cramer’s rule to compute \( \beta, \gamma \text{ and } t \).
10.3 Ray-Object Intersection: ray- polygon intersection

Ray: \( p(t) = e + td \)

Polygon: \( p_1, p_2, \ldots, p_m, \) and polygon normal \( n \)

First, compute the intersection point between the ray and the plane containing the polygon

\[ f(p) = (p - p_1) \cdot n = 0 \]

Set \( f(e + td) = 0 \) and solve for \( t \). We get

\[ t = \frac{(p_1 - e) \cdot n}{d \cdot n} \]

Use this \( t \) we can compute \( p \). Then determine if \( p \) is inside the polygon.
How do you tell if a point $P$ is inside a 2D polygon???
Assuming $P$ is in the same plane as the polygon
10.4 A Ray-Tracing Program

- **Using ray-tracing technique to produce images similar to the z-buffer or BSP-tree codes**

---

```plaintext
for each pixel do
    compute viewing ray;
    if (ray hits an object with \( t \in [0, \infty) \)) then
        compute \( n \);
        evaluate lighting equation and set pixel to that color;
    else
        set pixel color to background color;
```

---
If ray hits an object ... 

- The statement “if ray hits an object” can be implemented as a (boolean) function that tests for hits in the interval \([t_0, t_1]\) as follows:

```plaintext
hit = false
for each object o do
    if (object is hit at ray parameter \(t\) & \(t \in [t_0, t_1]\)) then
        hit = true
        hitobject = o
        \(t_1 = t\)
return hit
```
If ray hits an object …

- In actual implementation, we should return a reference to the object that is hit. This can be done by passing a record/structure such as

  \[ \text{hit-record } \text{rec} \]

- Should define a super class “surface” with sub-classes such as \textit{sphere, cone, cube}, … to cover all the objects that could be intersected by a ray

- Anything that a ray can intersect would be under that class
Object-Oriented Design for a Ray Tracer

- Develop a class hierarchy:
  - that not only include everything that can be hit by a ray, but also efficiency structures, such as bounding volume hierarchies

- Definition of the class **surface**:
  ```
  class surface
  virtual bool hit(ray e + td, real t0, real t1, hit-record rec)
  virtual box bounding-box()
  ...
  (t0, t1) is the interval on the ray where hits will be returned.
  ```
Object-Oriented Design for a Ray Tracer

- Implementation of "bounding-box()" for a sphere:

```cpp
box sphere::bounding-box()
vector3 min = center - vector3(radius, radius, radius)
vector3 max = center + vector3(radius, radius, radius)
return box(min, max)
```

- Should also define a class called "material" to hold properties of an object and build links between objects and materials.
10.5 How to Generate Shadows?

- Use *shadow rays* (different from *viewing rays*) to determine if a point is in shadow.
implement shadow rays for Phong lighting:

```c
function raycolor( ray e + td, real t0, real t1 )
hit-record rec, srec
if ( scene→hit(e + td, t0, t1, rec ) ) then
    p = e + rec. td
color I = I_a rec. k_d
    if (not scene→hit(p + s l, ε, ∞, srec )) then
        vector3 h = normalized( normalized(l) + normalized(-d))
        I = I + I_p rec. k_d max(0, rec. n ⋅ l) + I_p rec. W(θ)(h ⋅ rec. n)^rec.q
    return I
else
    return background-color
```
10.6 Specular Reflection

- Add specular reflection to ray tracing
- Two approaches: use $\alpha$ or $\beta$

\[ h = \frac{-d + l}{|| -d + l ||} \]
Add specular reflection to ray tracing

- For the view direction \( \mathbf{d} \), compute its specular reflection \( \mathbf{r} \)
- Use \( \mathbf{r} \) to recursively compute contribution from specular reflection (see next page)
- To avoid an infinite loop in the above recursive call, set up a maximum recursive depth

\[
\mathbf{r} = \mathbf{d} + 2(\mathbf{d} \cdot \mathbf{n})\mathbf{n}
\]
Add specular reflection to ray tracing:

Use $r$ to recursively compute contribution from specular reflection

$$
\text{color } I = I + k_s \ \text{raycolor} ( \ p + sr, \ v, \ \text{infinity})
$$

Where $k_s$ is the specular RGB color reflection coefficient.
Include specular reflection in "raycolor" (making it a real ray-tracing process):

```c
function raycolor( ray e + td, real t0, real t1 )
hit-record rec, srec
if ( scene→hit(e + td, t0, t1, rec ) then
    p = e + rec. td
    color I = I_a rec. k_d
    if (not scene→hit(p + sl, ε, ∞, srec )) then
        vector3 h = normalized( normalized(l) + normalized(−d))
        I = I + I_p rec. k_d max(0, rec. n · l) + I_p rec. W(θ)(h · rec. n)_{rec.q}
        r = −rec. d + 2(rec. d · rec. n)rec. n
        I = I + k_s raycolor(p + sr, ε, ∞)
    return I
else
    return background-color
```
Illustration:
10.7 Refraction

- Add refraction to ray tracing

- Light is refracted when it enters a dielectric (transparent material that refracts light). The refraction follows Snell’s law

\[ n \sin \theta = n_t \sin \phi \]

where \( n \) and \( n_t \) are the refractive indices of the objects
10.7 Refraction

\[ n \phi = n_t \phi \]

\[ h (=d) \]
10.7 Refraction

\[ n \sin \theta = n_t \sin \phi \]
\[ n^2(1 - \cos^2 \theta) = n_t^2(1 - \cos^2 \phi) \]

- \( \cos \phi \) can be computed as follows:

\[ \cos^2 \phi = 1 - \frac{n^2(1 - \cos^2 \theta)}{n_t^2} \]

- The refraction vector \( t \) can be computed as follows:

\[ t = \frac{n(d - n(d \cdot n))}{n_t} - n \sqrt{1 - \frac{n^2(1 - (d \cdot n)^2)}{n_t^2}} \]

when the number under the square root is negative, there is no refracted ray and all the energy is reflected.
10.7 Refraction

\[ g = d + n(-d \cdot n) \]

\[ = d - n(d \cdot n) \]

\[ b = \frac{g}{\sin \theta} \]
10.7 Refraction

- How should *refraction* be included in "raycolor"? Conceptually (see figure on next page),
  1) Compute the vector $t$
  2) Compute the intersection point $q$
  3) Compute the vector $h$
  4) Call "raycolor" with the ray $q + uh$ as the parameter to compute the intensity $I'$ of the object at $q$
  5) Combine $I'$ with $I$ to get the final color for $s$
Include refraction in "raycolor" (making it a 100% raytracing process):

```plaintext
function raycolor( ray e + t*d, real t0, real t1 )
hit-record rec, srec
if ( scene→hit(e + td, t0, t1, rec) then
    p = e + rec.td
    color I = I_a rec.k_a
    if (not scene→hit(p + s1, ε, ∞, srec )) then
        vector3 h = normalized( normalized(l) + normalized(-d))
        I = I + I_p rec.k_d max(0, rec.n · l) + I_p rec.W(θ)(h · rec.n)^rec.q
    r = -rec.d + 2(rec.d · rec.n)rec.n
    I = I + k_s raycolor(p + sr, ε, ∞)
compute the vector t
and the intersection point q
compute the vector h
I = (1 - k_r)I + k_r raycolor(q + uh, ε, ∞)
return I
else
    return background-color
```

10.8 Instancing

• How should we ray trace an instance of an object transformed by a matrix $M$?
• The matrix could be the accumulation of several transformations
• Ray tracing the transformed object can be done in the space of the untransformed object
10.8 Instancing

- One needs to transform the ray to the space of the untransformed object, do the tracing there, then transform the results to the space where the transformed object is in
- If we define an instance class of type *surface*, we need to create a hit function:

```c
instance::hit(ray a + tb, real t0, real t1, hit-record rec)
ray r' = M^{-1}a + tM^{-1}b
if (base-object→hit(r', t0, t1, rec)) then
  rec.n = (M^{-1})^T rec.n
  return true
else
  return false
```
10.9 Sub-Linear Ray-Object Intersection

- Three techniques to speed up ray-tracing:
  - bounding volume hierarchies
  - uniform spatial subdivision
  - binary space partitioning
10.9.1 Bounding Boxes

- Only need to know if the ray hits the box; do not need to know where
- Bounding box: \([x_{\text{min}}, x_{\text{max}}] \times [y_{\text{min}}, y_{\text{max}}]\)
- Ray: \(e + td\)
- Compute the ray parameters: where the ray hits the line

\[
\begin{align*}
x &= x_{\text{min}}, & x &= x_{\text{max}}, & y &= y_{\text{min}}, & y &= y_{\text{max}},
\end{align*}
\]

respectively. The ray hits the box iff the intervals \([t_{x_{\text{min}}}, t_{x_{\text{max}}}]\) and \([t_{y_{\text{min}}}, t_{y_{\text{max}}}]\) overlap.
10.9.2 Hierarchical Bounding Boxes

- Bounding boxes can be nested by creating boxes around subsets of the node
- Use top-down ray-box testing and bottom-up parameter-merging
End of 10.9