Ray tracing

CS 4300

Amit Shesh

Drawbacks of current “Pipeline” approaches

- Interactivity limits the realism of effects
- Shadows and reflections are not very easy
  - As we have seen...
- Transparency is...tricky too (beyond simple things)

What can a ray tracer do?

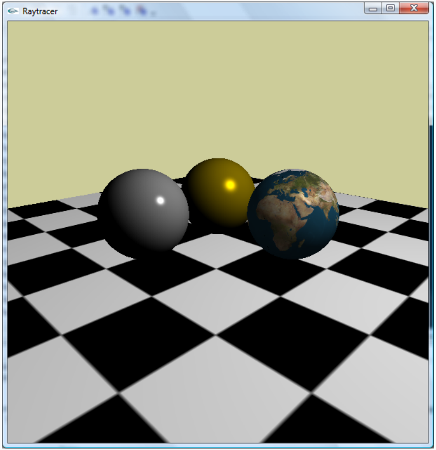

Figure 1: Ray tracer (comparable to your program)

What can a ray tracer do?

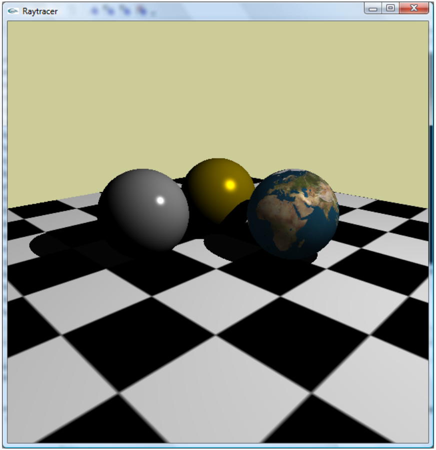

Figure 2: Ray tracer with shadows

What can a ray tracer do?

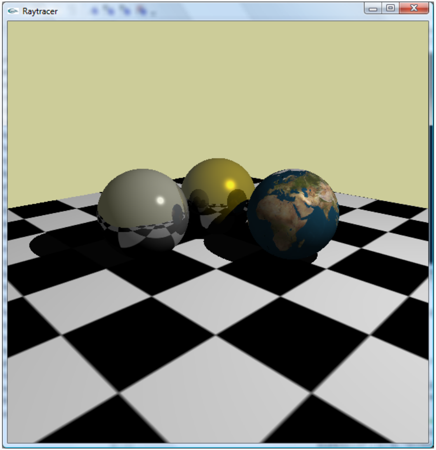

Figure 3: Ray tracer with shadows and reflections

What can a ray tracer do?

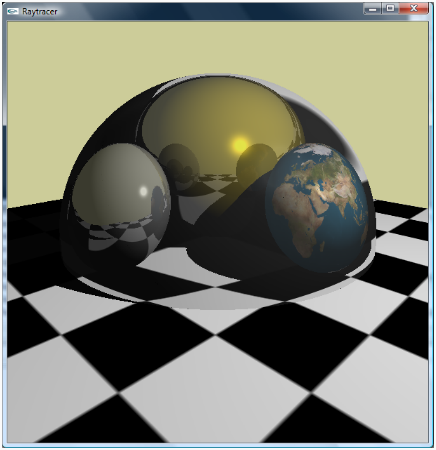

Figure 4: Ray tracer with shadows, reflections and transparency

What can a ray tracer do?

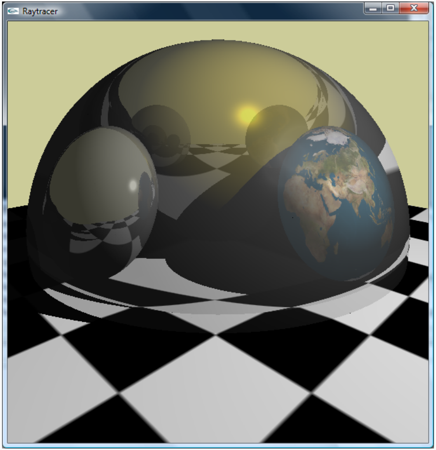

Figure 5: Ray tracer with shadows, reflections and multiple transparency

What could your ray tracer do?

- Graphics Hall of Fame
- Internet Ray Tracing Competition (archived)
- Results from POV-Ray

What is ray tracing?

- How the real-world physics works
- Ray tracing
- What is involved?
  - Math (we will go over this)
  - Minimal use of OpenGL

The ray-object intersection

- Fundamental to a ray tracer
- Basic question:
  - Given ray R and object O, does R hit O and if so, where?
- Math differs with each object

The ray-plane intersection

- Plane equation: $ax + by + cz + d = 0$
- Ray: $p = s + t\vec{v}$
- Problem: Find point of intersection, denoted by 't'

The ray-plane intersection

- Plane equation: $ax + by + cz + d = 0$
- Ray: $p = s + t\vec{v}$
- Problem: Find point of intersection, denoted by 't'
- Solution: $t = \frac{-(as_x + bs_y + cs_z + d)}{av_x + bv_y + cv_z}$

The ray-box intersection

- If ray intersects box, it will go "in" and then come "out" (2 intersections)
- Find intersection with 6 planes and choose
- Ray: $p = s + t\vec{v}$
- If it intersects the two "x" planes, then $-0.5 \leq s_x + tv_x \leq 0.5$

The ray-box intersection

- If ray intersects box, it will go “in” and then come “out” (2 intersections)
- Find intersection with 6 planes and choose
- Ray: $p = s + t\vec{v}$
- If it intersects the two “x” planes, then $-0.5 \leq s_x + tv_x \leq 0.5$
- Solution: $(t_{min}, t_{max}) = (\frac{-0.5 - s_x}{v_x}, \frac{0.5 - s_x}{v_x})$
  - What if $v_x = 0$?
- Find ranges for t using all three pairs, find their intersection

The ray-box intersection

- Normals:
  - From point of intersection,determine which face was hit
  - From hit face, compute normal directly
- Texture mapping:
  - From point of intersection,determine which face was hit
  - From hit face, one can compute the texture coordinates (try to reverse-engineer)

The ray-sphere intersection

- If ray intersects sphere, it will go "in" and then come "out" (2 intersections)
- Ray: $p = s + t\vec{v}$
- Sphere: $(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 = r^2$

The ray-sphere intersection

- If ray intersects sphere, it will go “in” and then come “out” (2 intersections)
- Ray: $p = s + t\vec{v}$
- Sphere: $(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 = r^2$
- Solution: $t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$
  - $A = {v_x}^2 + {v_y}^2 + {v_z}^2$
  - $B = 2(v_x(s_x - x_c) + v_y(s_y - y_c) + v_z(s_z - z_c))$
  - $C = (s_x - x_c)^2 + (s_y - y_c)^2 + (s_z - z_c)^2 - r^2$

The ray-sphere intersection

- Normals:
  - Determine normal from point of intersection and center of sphere
- Texture mapping:
  - From point of intersection, determine latitude ϕ and longitude θ
  - ϕ gives t and θ gives s

Transformations

- Ray and object must be in the same coordinate system
- t is independent of coordinate system
  - Applying t to ray in X coordinate system will give point of intersection in X
- Calculate t where the Math is the simplest
- Scenegraph:
  - Pass ray down the scene graph, similar to draw
  - In leaf, transform ray to leaf's coordinate system to do the Math

Computing “nearest” intersection

- Usually we want the “nearest” intersection, with respect to where the ray started
    - We want the smallest positive value of t (Why?)
- This simulates Z-buffer test in ray tracer
- Where to do this?
    - Compare with nearest intersection in the leaf itself
    - Take all ’t’ values and compare outside the scene graph

The overall setup

```
raytrace(width,height)
   Load world to view in modelview
   for y ← 0 to height do
      for x ← 0 to width do
         Create ray R from camera through pixel (x,y)
         color ← raycast(R,modelview)
         pixel(x, y) ← color
      end for
   end for
```

Ray casting

raycast(R,modelview)

Load world to view in modelview

(result,material) $\leftarrow$ closest_intersection(R,modelview,hitrecord)

if result $\leftarrow$ true **then**

  color $\leftarrow$ shade(R,hitrecord,lights,material)

else

  color $\leftarrow$ background-color

end if

Shadows

- Shadow is the “absence” of light
- Basic algorithm: for each point of intersection P, check if P can see light L_i
  - If it can, calculate shading for L_i else continue onto next light
- Shadow ray: from P in the direction towards the light
- Use ray cast to see if shadow ray hits something between P and the light
- Fudge shadow ray a bit in its direction to avoid precision issues

Reflections and Transparency

- To give material reflectivity and transparency, consider coefficients a, r, t
  - a: Absorption (0: not absorbent, 1: fully absorbent
  - r: Reflection (0: not reflective, 1: fully reflective
  - t: Transparency (0: not transparent, 1: fully transparent
  - Constrain $a + r + t = 0$
- Final color $= aC_a + rC_r + tC_t$
  - C_a: Color from the material itself (shading + texture mapping)
  - C_r: Color reflected by the object at point of evaluation
  - C_t: Color due to light passing through the object at point of evaluation

Reflections

- For a reflective object, consider only "specular" reflection (single direction of reflection)
- Law of reflection: Angle of reflection = Angle of incidence
- $\vec{R} = \vec{I} - 2(\vec{N} \circ \vec{I})\vec{N}$
  - $\vec{I}$: Direction of incoming ray
  - $\vec{N}$: Normal at point of evaluation
  - $\vec{R}$: Direction of reflected ray
- Find color reflected by this object at point of evaluation
  - Create reflection ray starting at point of evaluation in direction $\vec{R}$
  - Shoot the ray into the world and see what color it returns (i.e. raycast!)

Transparency

- Transparent objects “bend” light because speed of light changes
- Refractive index η: The ratio of speeds of light in vacuum to material (thus always $\geq$1)
- Snell’s law: $\frac{sin(\theta_i)}{sin(\theta_r)} = \frac{\eta_r}{\eta_i}$

Transparency

- $\vec{T} = \frac{\eta_i}{\eta_r}\vec{I} + (\frac{\eta_i}{\eta_r}\cos(\theta_i) - \cos(\theta_r))\vec{N}$
    - $\vec{T}$: Direction of transparency ray
    - $\vec{N}$: Normal at point of evaluation
    - $\vec{I}$: Direction of incoming ray
    - η_i: Refractive index of medium of incoming ray
    - η_r: Refractive index of medium of transparency ray
    - $\cos(\theta_i) = -(\vec{N} \circ \vec{I})$
    - $\sin(\theta_i) = \sqrt{1 - \cos(\theta_i)^2}$
    - $\sin(\theta_r) = \frac{\eta_i}{\eta_r}\sin(\theta_i)$
    - $\cos(\theta_r) = \sqrt{1 - \sin(\theta_r)^2}$
- Detecting total internal reflection: $sin(\theta_r)$ above is invalid, leading to invalid math for $cos(\theta_r)$

Transparency

- Find color refracted by this object at point of evaluation
  - Create refraction ray starting at point of evaluation in direction $\vec{T}$
  - Shoot the ray into the world and see what color it returns (i.e. raycast!)
- Must keep track of medium where the ray is
- What about intersecting transparent objects?
  - Option 1: Do not allow: ray is either inside a solid or out in air
  - Option 2: Objects are completed contained within others: can keep track on a stack
  - Option 3: Keep track of where you are and where you will be at all times (trickiest)