Lecture 11:
Recursive Ray Tracer

COMP 175: Computer Graphics
April 9, 2018
Note on using Libraries

- **C++ STL**
  - Does not always have the same performance.
  - Interface is (mostly) the same, but implementations differ and compilations also differ.
  - Generally very good, but you don’t always know what’s happening under the hood.

- **OpenGL:**
  - Same idea... The important thing is that interface is the same.
Summary

*Simple, non-recursive raytracer*

\[ P = \text{eyePt} \]

**for each** sample of image

Compute \( d \)

**for each** object

Intersect ray \( P+td \) with object

Select object with smallest non-negative \( t \)-value (visible object)

For this object, find object space intersection point

Compute normal at that point

Transform normal to world space

**for each** light

Use world space normal for lighting computations
Recall that:

- $I$ is intensity (for both light and the “final color”, which appears on the left side of the equation)
- $O$ is for object constant
- $k$ is for light constant
- $f_{att}$ is for attenuation
- $f$ is the amount of specular (how broad/narrow the specular region is)
- $n, l, r, v$ are normal that represent normal, light direction, reflective ray, and view vector respectively

The lighting is summed over all lights in the scene ($m$ lights in the equation) in a linear fashion.
Shadows

In the intersect assignment, we did the following things:

1. Created rays
2. Intersected rays with objects in the scene
   1. Found the nearest object
   2. Identified the intersection point and the normal
3. For the intersection point, solved the lighting equation

What about shadows?

How would you modify your intersect code to render shadows?
Shadows

- Review our previous lighting equation:

\[
\text{objectIntensity}_\lambda = \text{ambient} + \sum_{\text{light} = 1}^{\text{numLights}} \text{attenuation} \cdot \text{lightIntensity}_\lambda \cdot [\text{diffuse} + \text{specular}]
\]

\[
I = I_a k_a O_a + \sum_{m-\text{lights}} f_{\text{att}} I_m \left[ k_d O_d (n \cdot l) + k_s O_s (r \cdot v)^f \right]
\]
Shadows

- Each light in the scene contributes to the color and intensity of a surface, but **if and only if** it reaches the object!
  - could be occluded by other objects in scene
  - could be self-occluding
- Construct a ray from the surface intersection to each light
- Make sure light is first object intersected
  - if first object intersected **is** light, **count** light’s full contribution
  - if first object intersected **is not** light, **do not count** (ignore) light’s contribution
  - this method causes hard shadows; soft shadows are harder to compute
Recursive Ray Tracing Example

- Ray traced image with recursive ray tracing: transparency and refractions
Recursive Ray Tracing (1/4)

Simulating global lighting effects (Whitted, 1979)
Recursive Ray Tracing

Simulating global lighting effects (Whitted, 1979)

- By recursively casting new rays into scene, we can look for more information
- Start from point of intersection
- We'd like to send rays in all directions, but that's too hard/computationally taxing
- Send rays in directions likely to contribute most:
  - toward lights (blockers to lights create shadows for those lights)
  - specular bounce off other objects to capture specular inter-object reflections
  - use ambient hack to capture diffuse inter-object reflection
  - through object (transparency/refraction)
Recursive Ray Tracing (2/4)

- Trace “secondary” rays at intersections:
  - **light**: trace a ray to each light source. If light source is blocked by an opaque object, it does not contribute to lighting.
  - **specular reflection**: trace reflection ray (i.e., about normal vector at surface intersection).
  - **refractive transmission/transparency**: trace refraction ray (following Snell’s law).
  - Recursively spawn new light, reflection, and refraction rays at each intersection until contribution negligible / max recursion depth reached.

- Limitations
  - Recursive inter-object reflection is strictly specular.
  - Diffuse inter-object reflection is handled differently.

- Oldies-but-goodies
  - *Ray Tracing Silent Film, A Long Ray’s Journey into Light* (http://www.youtube.com/watch?v=b_UqzLBFz4Y)
Recursive Ray Tracing

- Your new lighting equation:

\[
I = I_a k_a O_a + \sum_{m\text{-lights}} f_{att} I_m [k_d O_d (n \cdot l) + k_s O_s (r \cdot v)^f] + k_s O_s I_r + k_t O_t I_t
\]

- \( I_r \) is the light intensity that comes from the secondary reflective ray
  - Finding the value of \( I_r \) is the “recursive” part of a recursive ray tracer
  - You stop the recursion when either:
    - Maximum recursive level is reached (defined by the user, e.g., 3)
    - The global contribution falls below a threshold (notice that \( I_r \) is multiplied by \( K_s \), which is the light’s specular constant and should be \( \leq 1 \))

- \( I_t \) is for transmitted rays (for refraction)
Transparent Surfaces (Transmitted Rays)

- For a partially transparent surface

\[ I_\lambda = (1 - k_{t1})I_{\lambda 1} + k_{t1}I_{\lambda 2} \]

- \( I_\lambda \) = the final intensity (\( \lambda \) denotes a color channel, r, g, or b)
- \( k_{t1} \) = the transparency value of surface 1 (0 = opaque, 1 = fully transparent)
- \( I_{\lambda 1} \) = the intensity calculated at surface 1
- \( I_{\lambda 2} \) = the intensity calculated at surface 2
Transparent Surface (Refraction)

- Refraction is modeled using Snell’s Law

\[
\frac{\sin(\theta_r)}{\sin(\theta_i)} = \frac{\eta_r}{\eta_i}
\]

- \( r \) = refraction, \( i \) = incident
- \( \eta_r \)=index of refraction for medium 2
- \( \eta_i \)=index of refraction for medium 1
- Note that we need to model each color channel (R, G, B) independently, so we could re-write the equation as:

\[
\frac{\sin(\theta_r)}{\sin(\theta_i)} = \frac{\eta_{r\lambda}}{\eta_{i\lambda}}
\]
Reflection + Transparency

- Remember that both $I_R$ and $I_T$ contribute to the final intensity of the light ($I$)
Reflection + Transparency + Direct Illumination

- With direct illumination, be mindful of potential shadows
  - That is, remember to compute if a light source can reach the intersection point (isect 1)
Reflection + Transparency + Direct Illumination + Recursion

- In this particular case, if we examine the contribution of $I_R$ at the recursive depth of 1 (that is, no more secondary rays), $I_R$ should contribute no light.
Recap of Recursive Ray Tracing

- Again, there are 3 types of secondary rays:
  - “Shadow Check”, Reflection, Refraction

- Controlling the recursion:
  - Recursively spawn secondary rays until lighting contribution falls below a certain threshold OR a max recursive depth is reached
  - Skip reflection rays if the material properties of the object is not reflective
  - Skip refraction rays if the object is opaque
“Tree of Light Rays”

- (Note we’re not showing the “shadow checks” in these images)
- T is for Transmitted Rays (refraction), and R is for Reflective Rays (reflection)
Programming Tip!

- Once we find an intersection (P) and cast “shadow check” rays against light sources (L1 and L2), we need to intersect the rays with the object of which P is on.

- In checking with L1, this works great. We find that there is an intersection between P and L1, and the intersection occurs at t=0 (where P is, that is, starting point of the ray)

- In checking with L2, this approach falls apart. **We will also find that an intersection occurs at t=0!!**

- Solution: move the intersection out by epsilon amount…
Questions?
Texture Mapping with Ray Tracer

- Texture mapping is supported by OpenGL.
  - The general idea is to “wrap” a texture around a geometric surface

- It can be incorporated into a Ray Tracer, which will allow for additional lighting effects (diffuse, ambient, specular, transparency, shadows, etc.)
Texture Mapping with Ray Tracer
Texture Mapping

- In general, we can think of texture mapping as a function that “maps” between two domains, position on the surface of an object (domain), and a pixel value from the texture image (co-domain).

This is typically done in two steps:

1. Map a point on the geometry to a point on a unit square
2. Map the unit square onto a texture image
2. Map from Unit Square to Texture Image

- Map a point in the unit (u, v) square to a texture of arbitrary dimension
- This can be done by linear interpolation between the coordinate space of the unit square to the texture
  - Unit Square coordinate: u is from 0.0-1.0, v is also from 0.0-1.0
  - Texture coordinate: w is from 0-width pixels, and h is from 0-height pixels

- In the above example:
  - (0.0, 0.0)->(0, 0); (1.0, 1.0)->(200, 100); (0.7, 0.45)->(140, 45)
- Note that the coordinates in (u, v) might not map perfectly to integer values that correspond to pixels. Need to do some interpolation (filter)
1. Map a point on the geometry to a point on a unit square

- **We have 4 geometric objects to consider in our ray tracer:**
  - Cube
  - Sphere
  - Cone
  - Cylinder

- **Cube is pretty easy...** The faces of the cube map nicely to a unit square
Mapping a Cube - Tiling

- Note that we can allow for tiling if the face of a cube is too large
  - If we map a single texture, it could stretch and look terrible
  - So a possible alternative is to use tiling
Tiling Example

Texture

Without Tiling

With Tiling
Mapping a Cylinder / Cone

- Recall: the goal is to map a point in \((x, y, z)\) into \((u, v)\)
- For a cylinder, we can break down the object into two parts:
  - The cap, which we will treat as a square surface
  - The body, which we will unroll into a square (see below)
- Cone is a special case of a cylinder; need to interpolate as we go up from the base to the tip

\[
P_x = u = .85 \\
P_y = v = .4
\]
Mapping a Cylinder / Cone

- Help on computing the u value:

- We need to map all points on the surface to [0, 1]

- The easiest way is to say \( u = \frac{\theta}{2\pi} \), and use the dot product to find \( \theta \), but it’s hard to determine the signs (e.g. \( \text{acos} \) returns a value between 0 and \( \pi \), same for \( \text{atan} \))
  - For example, \( \text{atan}(1, 1) = \text{atan}(-1, -1) = \frac{\pi}{2} \)

- So instead we use \( \text{atan2}(x, y) \), which returns a values between \(-\pi\) and \(\pi\). (Notice the discontinuity at \( u = 0.5 \))
Mapping a Sphere

- **Find \((u, v)\) coordinate for \(P\)**
  - Find \(u\) the same way as before for cylinders
  - \(v\) maps to the “latitude” of the sphere between 0 and 1 (the two caps)
    - At \(v=0\) and \(v=1\), there is a singularity. So set \(u\) a pre-defined value (e.g., 0.5)
  - \(v\) is a function of the latitude of \(P\):

\[
\phi = \sin^{-1} \frac{p_y}{r} \quad -\frac{\pi}{2} \leq \phi < \frac{\pi}{2} \quad r = \text{radius}
\]

\[
v = \frac{\phi}{\pi} + .5
\]
Questions?
Texture Mapping Complex Geometry

- Sometimes, texture mapping the polygons of an object doesn’t get what you are looking for.

Original Geometry

Texture each face separately (notice discontinuities)

Texture the object as a continuous object
Basic Idea

- Use a bounding sphere...
  - Find the ray’s intersection (in object space) with a bounding sphere, called P
  - Find P’s coordinate in the texture map’s (u, v) coordinate
  - Apply the texture to the point on the underlying geometry (the house)
Slightly More Advanced

- Turns out that we don’t have to use the bounding sphere at all.
  - Just intersect the geometry (house) at point $P'$, and assume that $P'$ lies on a sphere.
  - Same result, but need to find the radius at different parts of the geometry
  - Compute a new radius for each intersected point by finding the center of the geometry (house) and connect the center to the intersection point
Slightly More Advanced

- Turns out that you don’t have to use a sphere as a bounding surface

- You can use a cylinder or planar mappings for complex objects. Each has drawbacks:
  - Sphere: warping at the “poles” of the object
  - Cylinder: discontinuities between cap and body of the cylinder
  - Planar: one dimension needs to be ignored
    - But can do cool tricks with this...

- The problem is kind of hard. Since the object is in 3D, mapping it to 2D usually means some drawback
Questions?
Supersampling

- Notice the jaggies in this (recursively) ray-traced image

- What’s wrong with it?

- How can we fix it?
Supersampling

- Left image: one ray per pixel (through pixel center)
- Right image: 5 rays per pixel (corners + center)
  - Do weighted average of the rays to color in the pixel

Adaptive sampling

- Supersampling: more samples where we need it (e.g., where geometry or lighting changes drastically)
- Subsampling: fewer samples where we don’t need detail (faster computation)
  - Beam tracing: track a bundle of neighboring rays together
Supersampling

With Supersampling

Without Supersampling