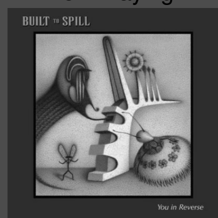
Now Playing:



Liar
Built To Spill
from *You In Reverse*
Released April 11, 2006

Movie: For The Birds

Pixar, 2000

Ray Tracing 1

Rick Skarbez, Instructor
COMP 575
November 1, 2007

Announcements

- Programming Assignment 3 (Rasterization) is due THIS Saturday, November 3 by 11:59pm
  - If you do hand in by tonight, +10 bonus points
- Assignment 3 (texture mapping and ray tracing) is out, due next Thursday by the end of class
- Remember that you need to talk to me about your final project

Programming 2 Recap

- Spherical Coordinates
  - Demo on board
- Per-Vertex Normals
  - Demo on board

Programming 3 Info

- Test data for part 1 (Lines) is available
  - As C/C++ array, or just as a text file
    - In both cases, each line has 7 parameters
      - (x_1, y_1, x_2, y_2, R, G, B)
    - This data set anticipates a 512x512 window
  - To read the array (line data), use something like the following code

```
typedef struct Line {
        int x1, y1, x2, y2;
        unsigned char r,g,b;
}Line;
Line lines[] =
#include "line.data"
;
```

Programming 3 Info

- For parts 2 and 3, the program should respond to user input
  - Can do this several ways
    - Accept coordinates as command line input
    - Prompt for user input while running
    - Allow user to click and choose points (like polygon creation in assignment 1

Programming 3 Info

- For part 3 (line clipping), should display a window bigger than the clip window
  - i.e.

Assignment 3 Overview

Last Time

- Extended our “camera” to be much more general
  - Arbitrary position / orientation / focal length
- Briefly discussed the software architecture of a raycaster
- Took a short course feedback survey
  - Thanks very much to everyone who participated!

Today

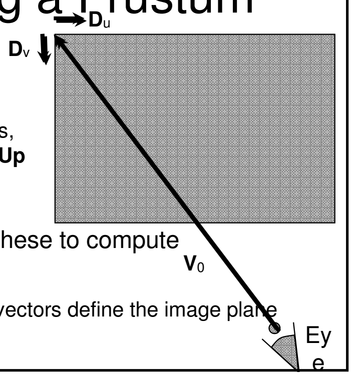
- Discussing how to implement shadows and reflections in a raytracer

Ray-Tracing Algorithm

- for each pixel / subpixel
  shoot a ray into the scene
  find nearest object the ray intersects
  if surface is (nonreflecting OR light)
  color the pixel
  else
  calculate new ray direction
  recurse

Building a Frustum

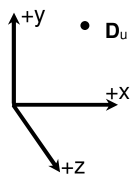
- So, we have:
  - θ, hRes, vRes, **eye, center, Up**
- Want to use these to compute $\mathbf{D}_u$, $\mathbf{D}_v$, $\mathbf{V}_0$
  - These three vectors define the image plane

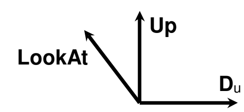


The "Right" Vector

- Need a vector that points in the "$\mathbf{D}_u$ direction"
  - Any ideas?
- Cross the look vector and the up vector
  - $\mathbf{D}_u$ = **LookAt** x **Up**

Note that **LookAt** and **Up** should be unit vectors

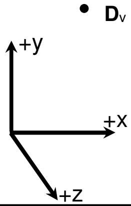


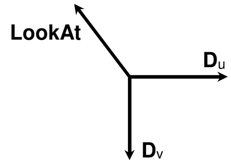


The "Down" Vector

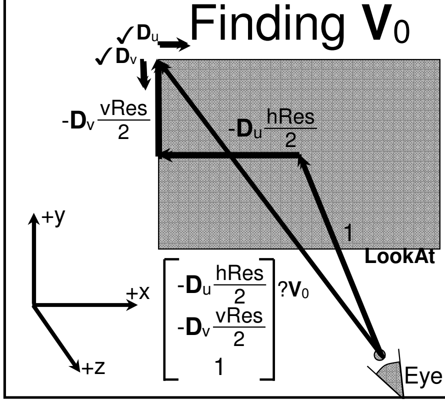
- So how to we find a vector perpendicular to two other vectors
  - Cross product
  - $\mathbf{D}_v$ = **LookAt** x $\mathbf{D}_u$

Note that if **LookAt** and **Up** are both unit vectors, then $\mathbf{D}_u$ and $\mathbf{D}_v$ are also unit vectors





Finding $\mathbf{V}_0$



Complete Frustum Specification

Given Points: **eye, center**
Given Unit Vector: **Up**
Given FOV Angle: θ
Given Dimensions: vRes, hRes

$$\textbf{LookAt} = \frac{\textbf{center} - \textbf{eye}}{||\textbf{center} - \textbf{eye}||}$$

$\mathbf{D}_u$ = **LookAt** x **Up**
$\mathbf{D}_v$ = **LookAt** x $\mathbf{D}_u$

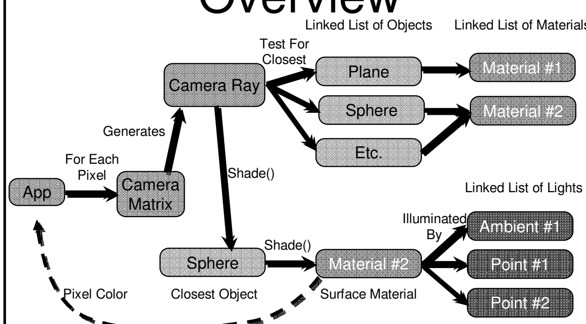
NOTE: Normalize $\mathbf{D}_u$ and $\mathbf{D}_v$!

$$FocalLength = \frac{hRes}{2\tan\frac{\theta}{2}}$$

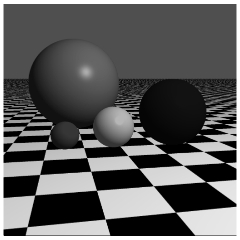
$$\mathbf{Vp} = FocalLength(\mathbf{Look}) - \frac{hRes}{2}(\mathbf{Du}) - \frac{vRes}{2}(\mathbf{Dv})$$

$$\mathbf{V} = \begin{bmatrix} \mathbf{Du} & \mathbf{Dv} & \mathbf{Vp} \end{bmatrix} \begin{bmatrix} i \\ j \\ 1 \end{bmatrix}$$

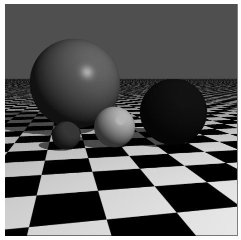
Raycaster System Overview



Shadows



Standard Scene

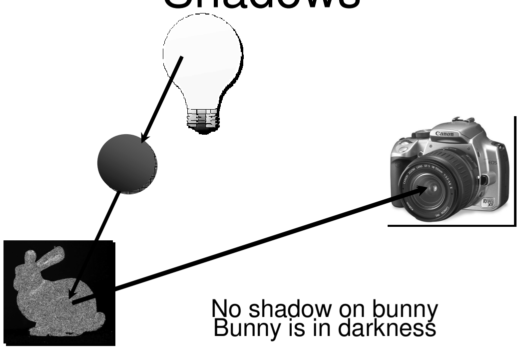


Scene With Shadows

Shadows

- What causes shadows?
  - An object lies between the shadowed surface and the light source
    - That is, the second object blocks photons/rays from reaching the first object

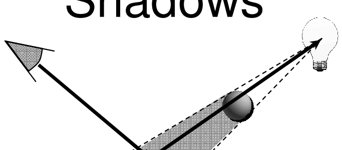
Shadows



Shadows

- So finding shadows is easy if all rays are starting at the light sources
  - If the ray does not get to a surface, that surface is in shadow
- But, remember, we're tracing rays backward
  - Starting at the camera
    - This complicates matters a bit

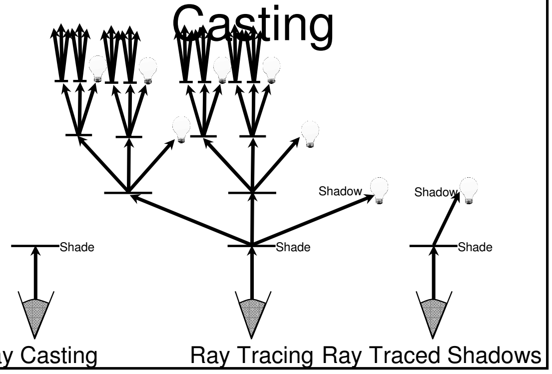
Shadows



- Assume we did our raycasting, and found that the ray intersected the plane
- Now we want to shade the point
  - Which includes determining if it is in shadow
- We can test this with a ray from the point to the light

And so we begin ray tracing...

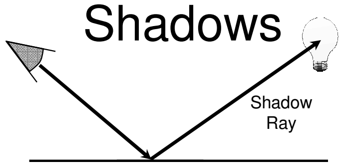
Recursive Ray Casting



Recursive Ray Tracing

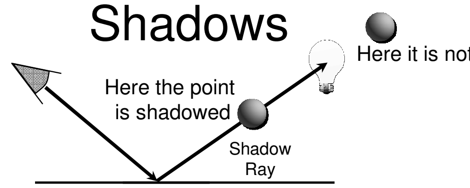
- Same idea as recursive functions
  - To solve one function/ray, do some simple work to generate inputs for a new function/ray
    - Recurse 'til you're done
- And just like recursive functions, it means we have to be very careful programmers
  - A small error in calculation or memory management can lead to catastrophe
- That said, now we can do some really cool stuff

Implementing Shadows



- All we do is generate a new ray, starting at the point and directed along the light vector
  - Test it just like any other ray
  - If an intersection occurs, then the point may be shadowed

Implementing Shadows



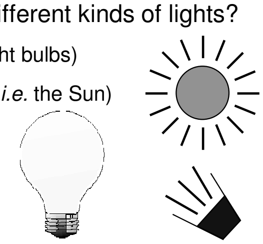
- To be thorough, we need to check the distance on the intersection
  - The object is only in shadow if the t value for the intersection is less than the t value of the light

Shadows Summary

- Can check if a point is in shadow by drawing a ray from that point to a light
  - If that ray hits an object, the point is in shadow
- This is our first baby step into real ray tracing
  - Shadows are EASY
    - Already know the point and vector of the new ray
    - Can use the existing intersection code

Lights and Shadows

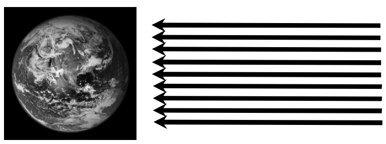
- Remember our different kinds of lights?
  - Point lights (*i.e* light bulbs)
  - Directional lights (*i.e.* the Sun)
  - Spot lights
  - Area lights



How do these cause problems for us when doing shadows?

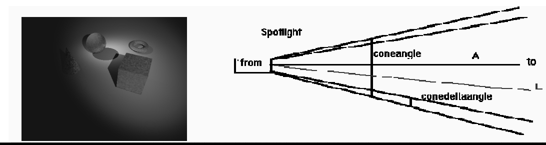
Directional Lights

- Point light sources at an infinite (or near infinite) distance
  - How does this affect our shadow rays?
    - Any intersection with a positive t is valid (generates a shadow)

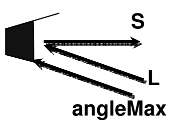


Spot Lights

- Similar to point lights, but intensity of emitted light varies by direction
  - Need to make sure that the shadow ray is inside the cone



Spot Lights

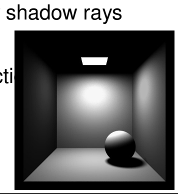


Vector Similarity: **S • L**

- Can test your shadow ray against the extents of your spotlight
  - If |**S • L**| <= |**S • angleMax**|, go ahead

Area Lights

- The most difficult case
  - No longer just one shadow ray
    - Really, <u>infinitely many</u> shadow rays
  - Can address by shooting many shadow rays for each light
    - This is a sampling/reconstruction problem
    - We'll come back to it later



Lights and Shadows Summary

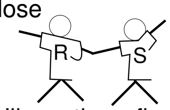
- Can still use the shadow ray technique with all the kinds of lights we consider
- Need to do a little bit more work for some
  - Directional lights: intersections at any distance
  - Spot lights: make sure ray is inside cone
  - Area lights: need to shoot a whole mess of rays

Reflection

- Now we're going to learn how to do reflections in our ray tracer
  - This is one of the classic benefits of ray tracing
    - Why do you think all these images have mirrored spheres in them?
  - Most every other rendering technique has to use hacks for this

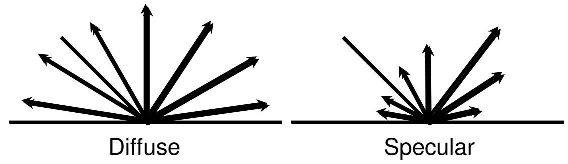
Reflection and Specularity

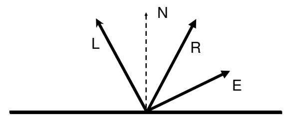
- Reflection and specularity are really close



- We'll use the reflection vector we computed for our specularity calculation as the new ray direction

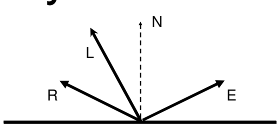
Reflection Vector





$$\mathbf{R} = -\mathbf{L} + \mathbf{2N}\,(\mathbf{L} \cdot \mathbf{N})$$

Ray Reflection



$$\boxed{\mathbf{R} = -\mathbf{E} + \mathbf{2N}(\mathbf{E} \cdot \mathbf{N})}$$

- Define a ray with
  - P = intersection point
  - V = reflection vector
    - Reflection of the eye vector, to be clear

How to Integrate This?

- This was our shading equation before:

Ambient Diffuse

- $I = (1 - r)\sum_{\text{Lights}}[I_a(R_a, L_a) + I_d(\mathbf{n}, \mathbf{l}, R_d, L_d, a, b, c, d) + I_s(\mathbf{r}, \mathbf{v}, R_s, L_s, n, a, b, c, d)]$

Specular

- Add another term, say r * (refColor)
  - Where r is how reflective the surface is
    - [0, 1]
  - And refColor is the color from the reflection ray

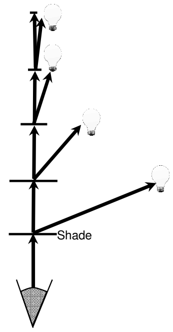
Shading + Reflection

- So now we have:
- $I = (1 - r)\sum_{\text{Lights}}[I_a(R_a, L_a) + I_d(\mathbf{n}, \mathbf{l}, R_d, L_d, a, b, c, d) + I_s(\mathbf{r}, \mathbf{v}, R_s, L_s, n, a, b, c, d)] + r(\text{refColor})$

Shading the Reflection Ray

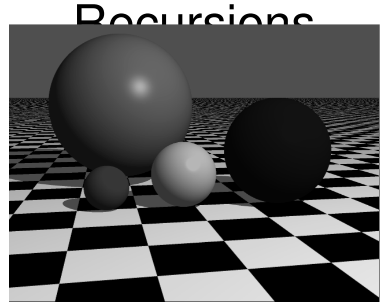
- So how do we determine the value of refColor?
  - Just treat it exactly like a camera ray
    - See if it intersects anything
      - If so, shade as normal and, if necessary, reflect again
      - If not, return the background color

Recursion



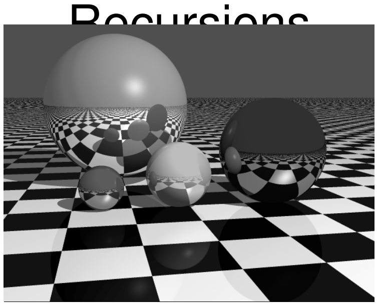
Ray Tracing

Reflection Recursions



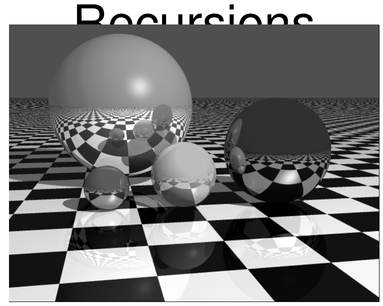
No Recursion (But With Shadows)

Reflection Recursions



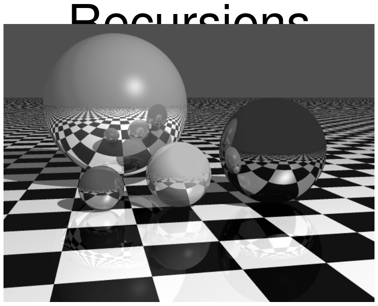
1 Recursion

Reflection Recursions



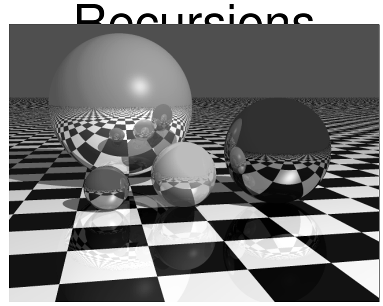
2 Recursions

Reflection Recursions



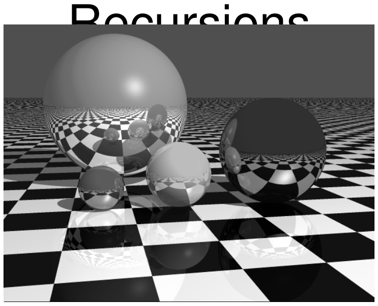
3 Recursions

Reflection Recursions



4 Recursions

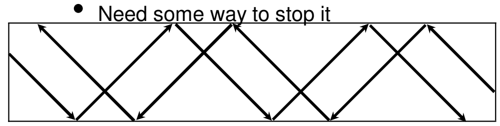
Reflection Recursions



5 Recursions

Stopping the Madness

- Does anyone see the problem here?
  - This could go on forever
    - Think of 2 mirrors reflecting each other
      - This would result in stack overflow and terribleness
    - Need some way to stop it

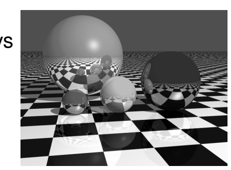


Stopping the Madness

- Solution: Put a depth limit on the recursion
  - Initialize each camera ray to have a depth of 0
  - Every "child" ray has depth = (parent's depth + 1)
    - Do not allow any new rays to be created with depth > maxDepth
- Also, there's obviously no need to cast new rays if the reflection coefficient is 0

Reflection Summary

- Reflection adds a great deal of realism to rendered scenes
- We discussed:
  - Generating reflection rays
    - Similar to specularity calculation
  - Shading with reflection
    - Just add another term
  - Preventing infinite recursion



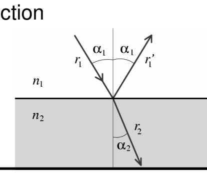
Refraction

- Refraction works just like reflection
  - When a ray hits a surface
    - Shade as normal
    - Figure out if you need to cast a refraction ray
      - If so, calculate the new ray
        - Shade it as normal, and add it as yet another term to our shading equation

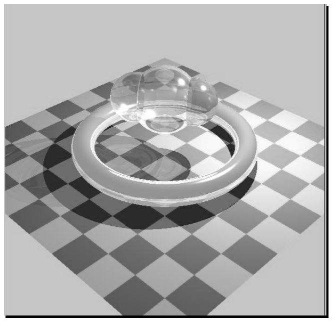
Refraction Rays

- Need to store the index of refraction and a transparency coefficient or each material
  - If the object is transparent, generate a new ray using Snell's law
    - Continue just as in reflection

$n_1 \sin \alpha_1 = n_2 \sin \alpha_2$



Refraction Example



Next Time

- Filling in some of the gaps for how to build a real ray tracer
  - Instantiation of multiple objects
  - Some acceleration tricks and optimizations
  - Identifying and fixing some tricky bits