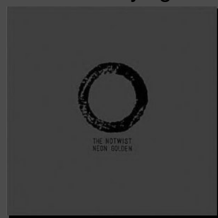
Now Playing:



Pilot
The Notwist
from *Neon Golden*
Released February 25, 2003

Movie: Boundin'

Pixar, 2003

Ray Tracing II

Rick Skarbez, Instructor
COMP 575
November 1, 2007

Announcements

- Assignment 3 (texture mapping and ray tracing) is out, due Thursday by the end of class
- Also due on Thursday is your project proposal
  - Make sure you meet with me if you haven't already
- Programming Assignment 4 (Ray tracer) is out today, due Tuesday 11/20 by 11:59pm

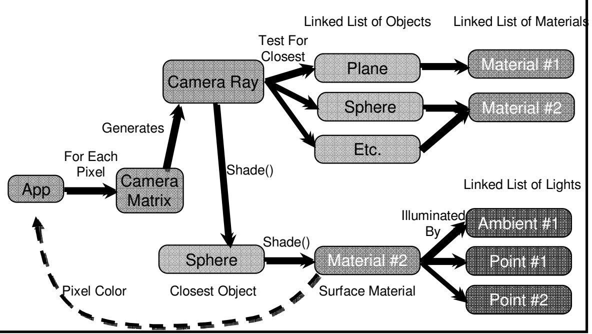
Assignment 3

- Homework 3 is due next time
  - Texture mapping
  - Ray generation
  - Ray-object intersection
  - Refraction
- Any questions?

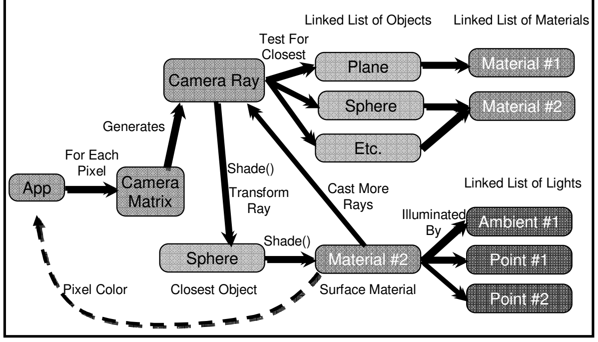
Programming Assignment 4

- Build a ray tracer
- Components:
  - Handle file output
    - You will be storing your images to disk
  - Generate ray casted images
  - Generate ray traced images

Ray Caster Overview



Ray Tracer Overview



What I will give you

- FSF image format specification
- FSF Viewer
- Matrix / Vector / Ray classes
- .ray file format specification
- Some sample .ray files
  - All available on the website

.FSF File Format

- You will be outputting your result images in this format
- 32-byte (RGBA) uncompressed ASCII format
- This was developed by Eric Bennett for COMP 575 last year
  - Info is online on his website: http://www.ericpbennett.com/COMP575/FSF.htm

.FSF File Format

```
32-bit Unsigned Integers
        Value: 575
        Value: Width
        Value: Height
        Value: Number of Frames (1 implies a still image)

Repeat for each image {
        Repeat for each scanline {
                Repeat for each pixel (left to right) {
                        8-bit Unsigned Chars
                                Value: Red
                                Value: Green
                                Value: Blue
                                Value: Alpha (0 is transparent, 255 is opaque)
                }
}
}
```

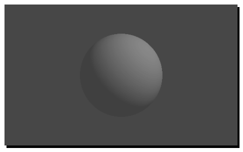
.RAY File Format

- Adapted from the scene descriptions used by Prof. Leonard McMillan and Eric Bennett
- A plain text file specifying the scene
- Each line is a command

Example Scene

```
eye 0 0 5
lookat 0 0 0
up 0 1 0
fov 60
background .5 0 0.5

material 0 1 0 .3 .7 0 0 0 0 0

sphere

light 1 1 1 ambient
light 0.6 0.6 0.6 point 3 3 3
```

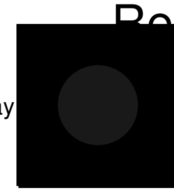
test2.ray



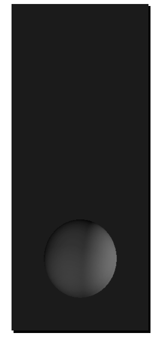
500x300

Expected Raycasting Results

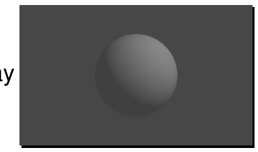
test1.ray



test3.ray



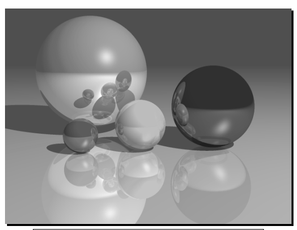
test2.ray



Example Scene

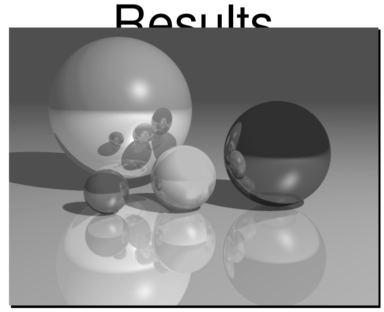
```
eye -2 1.5 10
lookat 0 0 0
up 0 1 0
fov 45
background 0.078 0.361 0.753

material 1 0 0 0.3 .7 .5 100 .5 0 0

reset
scale .5 .5 .5
translate -2 -.5 0
sphere

material 0 1.0 0 0.3 .7 .5 100 .5 0 0

reset
scale .75 .75 .75
translate -.25 -.25 0
sphere

material 0 0 1.0 0.3 .7 .5 100 .5 0 0

reset
scale 1.25 1.25 1.25
translate 2 .25 0
sphere

material 1 1 1 0.1 .3 .3 100 .8 0 0

reset
scale 2 2 2
translate -1.5 1.25 -3
sphere

material .6 .6 .6 0.3 .7 1.0 50 .5 0 0

reset
translate  0 -1 0
plane

light 1 1 1 ambient
light 1.0 1.0 1.0 point 5 9 10
```

reflect.ray



You may not get a perfect match, but it should look very similar

Expected Raytracing Results



reflect.ray

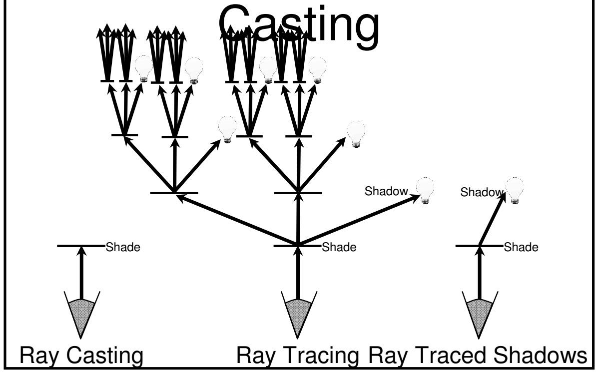
Last Time

- Discussed how to implement
  - Shadows
  - Reflection
  - Refraction

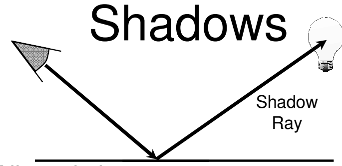
Ray-Tracing Algorithm

- 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

Recursive Ray Casting

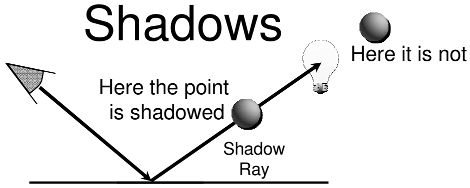


Implementing Shadows



- 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

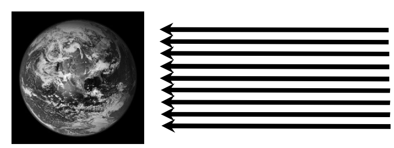
Implementing Shadows



- 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

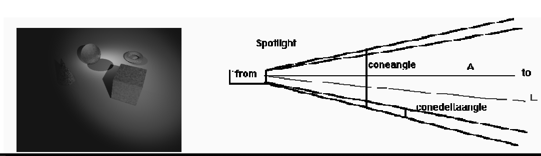
Directional Lights

- 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)

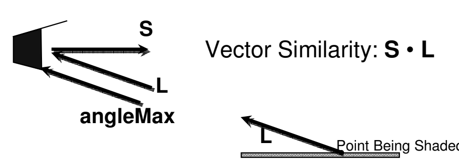


Spot Lights

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



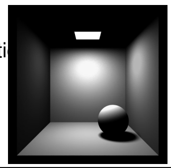
Spot Lights



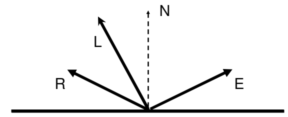
- Can test your shadow ray against the extents of your spotlight
  - If |**S • L**| <= |**S • angleMax**|, go ahead

Area Lights

- The most difficult case
  - No longer just one shadow ray
    - Really, infinitely many 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



Ray Reflection



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

- 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

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

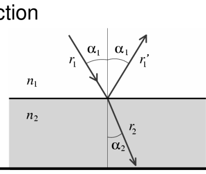
Refraction

- 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

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



Review Over

- Any questions?

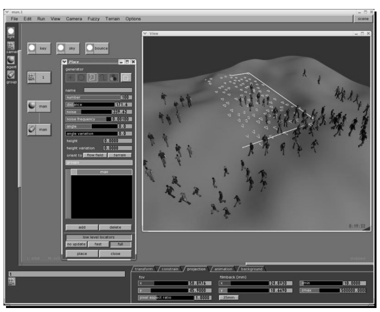
Today

- Cover "the rest" of our ray tracer
  - Talk about instantiation of multiple objects
  - Address some potential problems
  - Talk about data structures
  - Talk about optimizations

Instantiation

- We know how to handle canonical versions of objects
  - Say, a unit sphere centered at the origin
  - Or an infinite plane at y = 0
- How do we handle multiple objects?
- Or objects with different sizes/shapes?
  - These are all part of instantiation

The Power of Instantiation



MASSIVE Crowd Simulator

Carlton Draught's "Big Ad"

George Patterson & Partners, 2005

Transforming Objects

- We talked extensively about transforms earlier in the class
  - Translation
  - Rotation
  - Scaling
- We're going to be using them here, but now we have to build the matrices ourselves
  - Let's review

Translation in 3D

- We will represent translation with a matrix of the following form:

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & t \\ 0 & 1 & 0 & u \\ 0 & 0 & 1 & v \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

t is the x-offset
u is the y-offset
v is the z-offset

Scaling in 3D

- We will represent scaling with a matrix of the following form:

$$\mathbf{M} = \begin{bmatrix} \alpha & 0 & 0 & 0 \\ 0 & \beta & 0 & 0 \\ 0 & 0 & \gamma & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

α is the scale factor in the x-direction
β is the scale factor in the y-direction
γ is the scale factor in the z-direction

Rotation in 3D

Rotation About The Z-Axis

$$\mathbf{M} = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation About The X-Axis

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

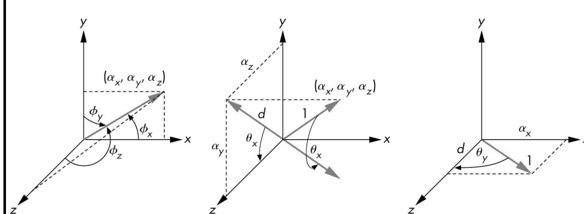
Rotation About The Y-Axis

$$\mathbf{M} = \begin{bmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation about any axis in 3D

- How can we extend this to rotation about *any* axis, not just the principle axes?
  - Need to move the axis we want to rotate about to one of the principle axes (say, the z axis)
  - First, apply a rotation about x, to move the axis into the yz-plane (R_x)
  - Then, apply a rotation about y, to move the axis onto the z-axis (R_y)
  - Then apply your desired rotation, followed by the inverses of the other

Rotation about any axis in 3D



$R_{total} = R_x^{-1} R_y^{-1} R_z R_y R_x$

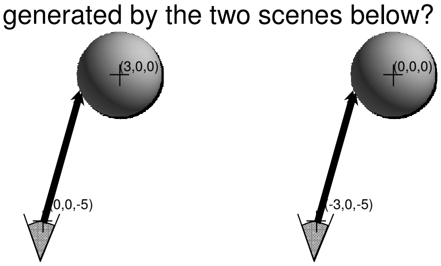
Rotation about any point and any axis in 3D

- To rotate about a non-origin point, extend in the same way as in 2D
  - First, translate to the origin (T_{xyz}^{-1})
  - Then apply your rotation (in the most general case, $R_x^{-1} R_y^{-1} R_z R_y R_x$)
  - Then translate back (T_{xyz})

$R_{total} = T_{xyz} R_x^{-1} R_y^{-1} R_z R_y R_x T_{xyz}^{-1}$

One More Thing...

What is the difference in the images generated by the two scenes below?



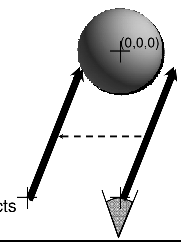
As it turns out, there isn't any

Remember This?

- We talked about how applying a transformation to the world is the same as applying the inverse transformation to the camera
  - Why might this be useful?

Transforming Rays

- Instead of transforming objects, we will apply the inverse transforms to our rays
- Why?
  - We can write really really fast code to intersect rays only with canonical objects without worrying about size, shape, location, etc.
  - We have a standard process
    - Transform rays
    - Intersect with canonical unit objects



Inverse Transforms

For all of our transforms, changing their direction generates the inverse matrix

$$[Translate(x,y,z)]^{-1} = Translate(-x,-y,-z)$$

$$[Scale(x,y,z)]^{-1} = Scale(\frac{1}{x},\frac{1}{y},\frac{1}{z})$$

$$[Rotate(\theta)]^{-1} = Rotate(-\theta)$$

This conveniently saves us the trouble (and cost) of implementing matrix inversion

Inverting Composed Transforms

- Remember that one of the benefits of using matrices for transforms was that we could compose many transforms into one matrix
  - Can we easily get the inverse of this composed matrix?

Inverting Composed Transforms

- Answer: Yes!
- Lucky for us,

$$(\mathbf{ABC})^{-1} = \mathbf{C}^{-1}\mathbf{B}^{-1}\mathbf{A}^{-1}$$

- Note that the order of transforms gets reversed
- Now the operator that gets applied first is the leftmost

Transforming Rays

- So, now we know how to invert our transforms
  - And we know that we can use these to transform the camera
- But how do these affect our rays?
  - Remember: A ray is a point and a vector
    - The point is affected by translation
    - The ray is affected by rotation
    - Both are affected by scaling

Transforming Rays

- The point of origin is a point, so it gets transformed as a homogeneous point
- The direction is a vector, so it gets transformed as a vector

Untransformed ray:	Ray equivalent to the transform **M** being applied to the world:
$r(t) = \mathbf{S} + t\mathbf{V}$	$r'(t) = \mathbf{M}^{-1}\mathbf{S} + t\mathbf{M}^{-1}\mathbf{V}$

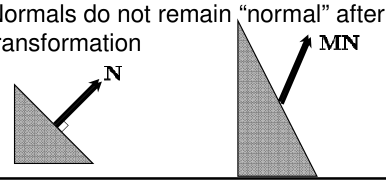
Object Intersections with Transformed Rays

- Once the ray is transformed, just intersect it with your canonical objects as normal
- The resulting t value can be plugged into the original untransformed ray to find the point of intersection in world space

Caution: Do not normalize the vector in the ray after transformation *r'*, or else values of t will not be comparable to each other

You didn't really think it would be that easy…

- That gets us the new ray to the eye
  - But that isn't the only thing we need for shading
  - What about the normal vector?
- Normals do not remain "normal" after transformation



Finding the New Normal Vector

- For just a minute, let's pretend that we're doing it the old way
  - Transforming the world, not the ray
- Before the transform
  - $\mathbf{N} \cdot \mathbf{T} = 0$ (**N:** normal vector, **T**: tangent vector)
- After the transform
  - $\mathbf{T}' = \mathbf{MT}$ (tangent vectors remain tangent)
  - $\mathbf{N}' \cdot \mathbf{T}' = 0$
    - So, what is **N**'?

Finding the New Normal Vector

- Let's denote the unknown transform as **G**
  - $\mathbf{N}' = \mathbf{GN}$ and $\mathbf{T}' = \mathbf{MT}$, and $\mathbf{N}' \cdot \mathbf{T}' = 0$
    - ➡ $\mathbf{GN} \cdot \mathbf{MT} = 0$
    - ➡ $(\mathbf{GN})^T\mathbf{MT} = 0$
    - ➡ $\mathbf{N}^T\mathbf{G}^T\mathbf{MT} = 0$
  - With the original normal, $\mathbf{N}^T\mathbf{T} = 0$
    - ➡ $\mathbf{G}^T\mathbf{M} = \text{Identity}$
    - ➡ $\mathbf{G}^T = \mathbf{M}^{-1}$ □ $\mathbf{G} = (\mathbf{M}^{-1})^T$

Finding the New Normal Vector

- So, in the end, the new normal vector is given by
  - $\mathbf{N}' = (\mathbf{M}^{-1})^T\mathbf{N}$
- Since we already know how to compute the inverse of the transform matrix
  - All that is left to do is transpose it!

Putting it All Together: Applying Ray Transforms

- For each ray-object intersection
  - Apply the inverse of any object transforms to the ray
  - Intersect the resulting ray with the canonical object
  - If there is a valid intersection
    - Plug t into the original ray equation to get the location of the intersection in world space
    - Get the correct normal as shown on the last slide

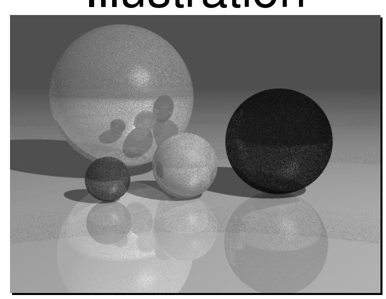
So why do things this way?

- Only need to store a single model of an object
- For each instance of it, maintain
  - Material properties
  - Object transform
    - Can precompute inverse and inverse transpose for improved performance

Potential Problem: Re-Intersection

- This could be a tricky problem
- Consider this situation:
  - We intersect a ray with a mirrored sphere
  - We find the reflection ray
  - We intersect that ray with all objects
    - It, by definition, intersects the sphere at the exact same point!

Re-Intersection Illustration



A ray tracer without any re-intersection handling

Why does this happen?

- Short answer: numerical precision issues
  - Sequences of floating point multiplies (accumulated in our transforms) result in small inaccuracies
    - It is essentially random whether a ray from any given point will work correctly (because the point is at t=0 or just behind it) or fail (because the point is at t>0)
- Note that this is a problem for shadow rays too

Solutions

- Solution #1
  - Simply do not allow intersections for values of t < ε
    - Where ε is a very small number, like .0001
- Solution #2
  - When a new ray is generated, offset it’s origin point by ε in the direction of the surface normal

Next Time

- Covering whatever raytracer implementation details we didn't get through today
- Discussing some advanced raytracer functionality
  - Acceleration data structures
  - Monte Carlo sampling for various effects