Now Playing:

Go into the Water
Dethklok
from *Dethalbum*
Released September 25, 2007

Movie: One Man Band

Pixar, 2005

Programming 4 Tips and Advanced Ray Tracing

Rick Skarbez, Instructor
COMP 575
November 8, 2007

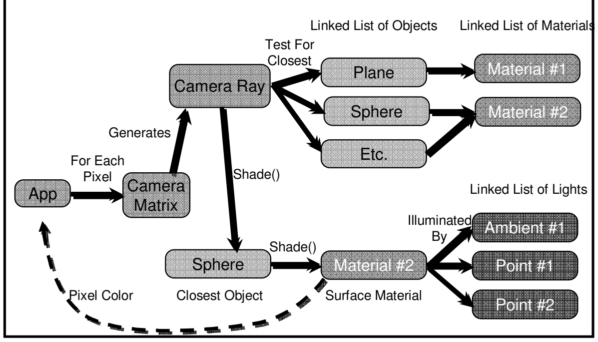
Announcements

- Programming Assignment 4 (Ray tracer) is out, due Tuesday 11/20 by 11:59pm
- If you haven't met with me yet to discuss your final project, you should really do that as soon as possible

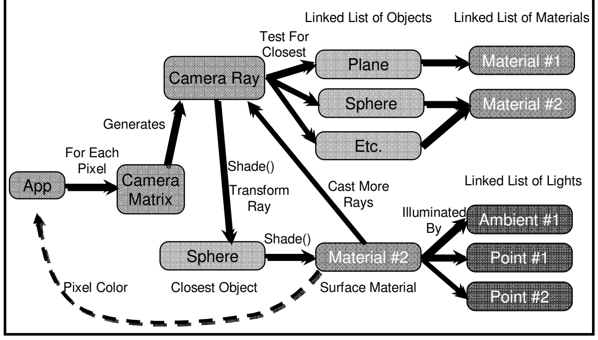
Programming Assignment 4

- 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



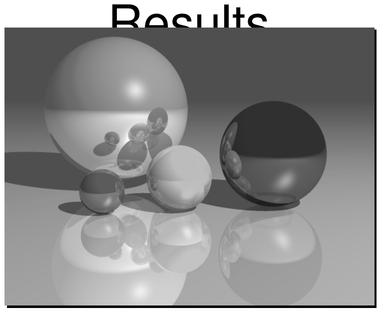
Ray Tracer Overview



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

Expected Raytracing Results



reflect.ray

I'd like to put your mind at ease...

- I know some (alright, probably all) of you are worried about this assignment
  - I am now going to try to convince you not to worry as much

Why this assignment shouldn't scare you

- Unlike OpenGL,
  - Don't have to handle user input
  - Don't have to have a moving camera or moving objects
  - Don't have to manage graphics state
    - All computation is just on the CPU

Why this assignment shouldn't scare you

- Unlike previous assignments, this one (for better or worse) is pretty straightforward
  - Just getting the ray tracer working is already good for full credit
    - Don't need to do a bunch of different programs
  - Test cases are readily available

Why this assignment shouldn't scare you

- Ray tracing is really a very simple algorithm
  - If you don't worry much about performance (which you don't have to), it might be the shortest program you write all semester
  - Let's go to the board

Code for you

- Now on the website:
  - 4x4 matrix class
    - matrix44.h and matrix44.cpp
  - 4-vector class (for homogeneous points and vectors)
    - vector3D.h and vector3D.cpp
  - Ray class
    - ray3D.h and ray3D.cpp

Deep Breaths...

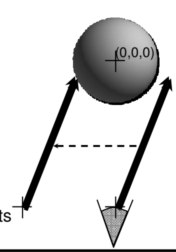
- I hope everyone is feeling a little bit better
- Are there any more questions?

Last Time

- Presented programming assignment 4
  - Already talked a bunch about that
- Talked about instantiating objects and implementing transforms in a ray tracer
- Talked about the "re-intersection problem", and how to avoid it

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

- Answer: Yes!
- Lucky for us,

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

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

Transforming Rays

- 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:
$r(\mathrm{t}) = \mathbf{S} + \mathrm{t}\mathbf{V}$	$r'(\mathrm{t}) = \mathbf{M}^{-1}\mathbf{S} + \mathrm{t}\mathbf{M}^{-1}\mathbf{V}$

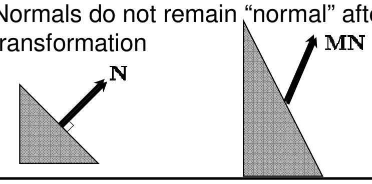
Object Intersections with Transformed Rays

- 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



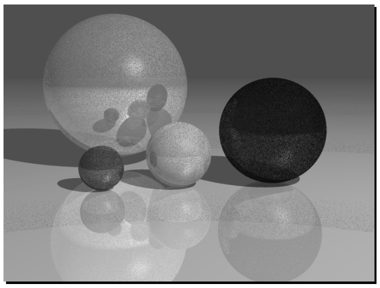
Finding the New Normal Vector

- 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

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

Re-Intersection 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

Today

- Talking about advanced ray tracing techniques
  - Acceleration data structures
    - To improve performance
  - Distributed ray tracing techniques
    - To achieve some neat visual effects

Must go faster...

- So what we've done so far works
  - We can render any scene just fine
    - At least, any scene that doesn't use additional effects
- But it's really, really slow:
  - Loop
    - For each pixel
      - For each object
        - For each light
  - Reflection/refraction/shadows make it even worse

Must go faster...

- Can't do anything about looping over each pixel
  - That we're stuck with
- But looping over every object and every light?
  - There we have some options

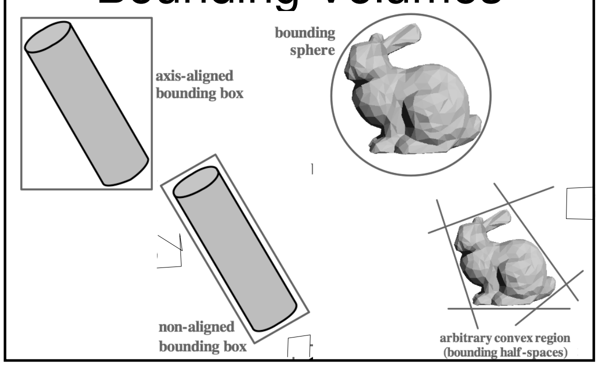
Reducing Intersections

- Note that this is also the biggest “bang for your buck” in terms of performance
  - Computing ray intersections is slow
- 2 main ways
  - Use bounding volumes
  - Use a spatial data structure

Bounding Volumes

- Here’s the idea:
  - Some shapes are harder to intersect with than others
    - Consider a box vs. a complex polygonal model
  - So, for every object, find the smallest simple object that encloses it
  - Test for intersection against the simple object
  - If there is one, only then do you test the original object

Bounding Volumes



Bounding Volumes

- What makes a good bounding volume?
  - Conservative
    - No false negatives
  - Tight to the object
    - No false positives
  - Fast to compute intersections with
- Often a tradeoff between the two
  - Spheres or axis-aligned bounding boxes (AABBs) are good places to start

Spatial Data Structures

- We already talked about binary space partitioning (BSP) trees
  - We said we’d come back to them
  - Now we are
- Not the only option, though
  - Grids
    - Adaptive (quad/octrees) or non-adaptive

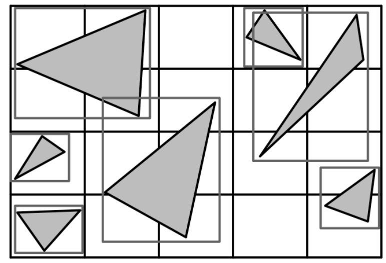
Spatial Data Structures

- The idea is that we only need to test if a ray hits an object if the ray passes through the region of space that the object is in
  - If a ray is going left, and the object is on the right, there is no need to test for intersection

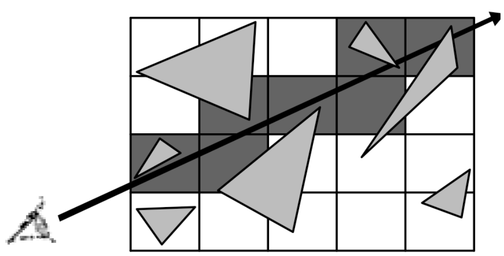
Regular Grids

- Simplest structure
  - Just divide space into a regular grid
    - Say, 1-unit axis aligned cubes
  - Only need to test against any objects inside a region if the ray hits it
  - Only need to test farther regions if no intersection in nearer regions
  - Can use 3D line drawing algorithms for fast cell traversal

Regular Grids



Regular Grids



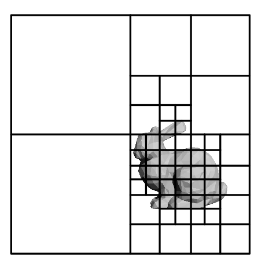
Adaptive Grids

- Several ways to do it
  - We'll talk about quadtrees / octrees
    - Quadtrees are 2D, octrees are 3D

Octrees

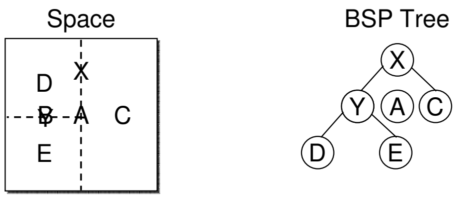
- Start out with a very coarse regular grid
  - Subdivide only in areas where there is geometry
    - If there is a primitive inside a grid cell, split that grid cell into 8 equal cells
    - Repeat until you've reached some maximum split depth, or a cell only contains a single primitive

Octree (well, really Quadtree) example



Remember this example?

- Let's see an example (in 2D, not 3D)
  - Here, a line divides the plane into two half-planes



BSP Trees

- Can create a BSP tree for your scene
- Then only have to test against objects that are on the "right" side of a split plane

Accelerating Lighting

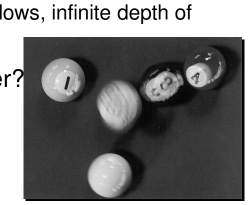
- The previous structures reduce ray intersections
  - Can we improve lighting, too?
- Sure!
  - One way: Do a pre-pass that determines if a light is visible from an object (or region of space...), and cache that information
  - Then only need to test against potentially visible lights

Acceleration Review

- We're stuck looping over each pixel
- But we can:
  - Test against fewer and/or simpler objects
    - Bounding volumes
    - Spatial data structures
  - Test against fewer lights
    - Do a potential visibility pre-pass

Stochastic Ray Tracing

- Cook argues that classical ray tracing (*i.e.* everything we've done so far) only represents <u>sharp</u> phenomena
  - Unrealistic sharp shadows, infinite depth of focus, etc.
- How can we do better?



Distributed Ray Tracing

Cook et al., 1984

- So what are some of the effects we can expect this way?
  - Antialiasing
    - Distribute rays across each pixel
  - Glossy reflections
    - Distribute multiple reflection rays instead of just one

Stochastic Ray Tracing

- So what are some of the effects we can expect this way? (cont'd)
  - Soft shadows
    - Distribute multiple rays to an area light source
  - Depth of field
    - Distribute rays across a lens
  - Motion blur
    - Distribute rays <u>over time</u>

Glossy Reflections

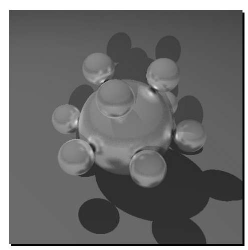
- Shooting a single reflection ray simulates perfect reflection
  - i.e. a mirror
- Many real surfaces are reflective, but not mirror-like
  - i.e. many metals
  - This is called *gloss*

Glossy Reflections

- To get glossy reflections, don't just shoot one ray
  - Shoot multiple rays, and perturb them slightly
    - This simulates taking the integral over a solid angle

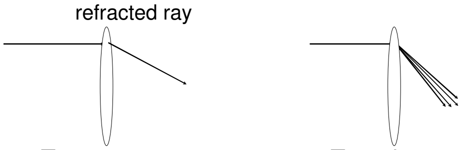
Mirror Glossy

Glossy Reflection Examples



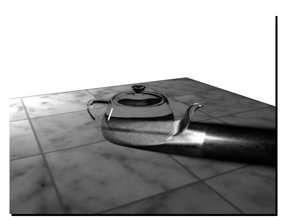
Translucency

- Translucency is sort of the dual of glossy reflection
  - Instead of distributing rays around the reflection ray, distribute them around the refracted ray



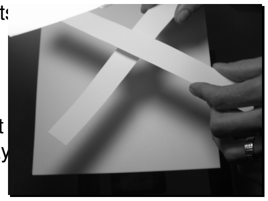
Transparent Translucent

Translucency Examples



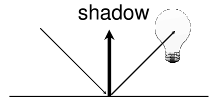
Soft Shadows

- In most graphics applications (and in our ray tracer so far), we've assumed point light sources
  - In the real world, lights have area
  - This leads to soft shadows in the real world, which we can't yet simulate in our ray tracer

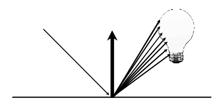


Soft Shadows

- To get soft shadows, don't just shoot one ray
  - Shoot multiple rays distributed across the surface of the light
  - Sum their contributions to find the amount of shadow

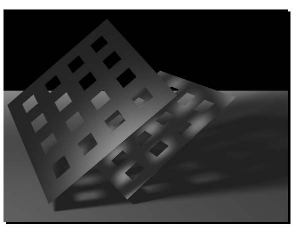


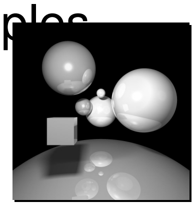
Hard Shadows

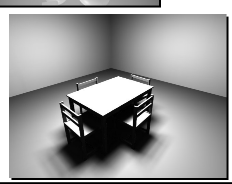


Soft Shadows

Soft Shadow Examples





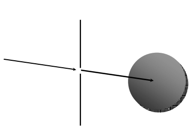


Depth of Field

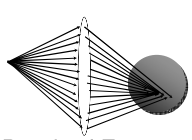
- Our ray tracer up to this point simulates a pinhole camera
  - Real world cameras have lens differing aperture sizes, differing exposure times, etc.
  - We're going to focus (no pun intended) on depth of field

Depth of Field

- To get depth of field, generate multiple rays for each pixel
  - Distribute them across the surface of the lens

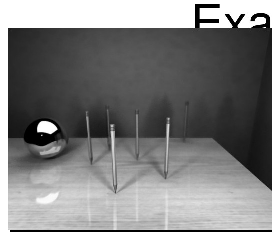


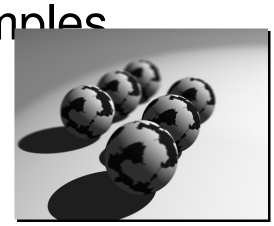
Perfect Focus

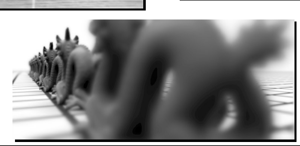


Depth of Focus

Depth of Field Examples







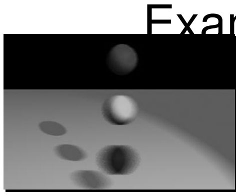
Motion Blur

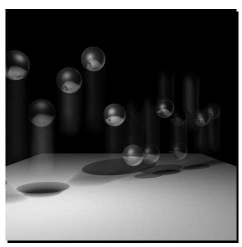
- Motion blur in the real world happens when objects are moving while the camera shutter is open
  - Effectively, the same point on the object is seen along multiple ra[ys by] the camera

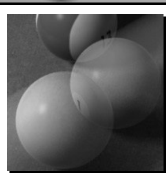
Motion Blur

- To get motion blur, you need to distribute your rays over time
  - As an object moves, it will get hit by different camera rays
  - Moving objects get averaged with the environment
- What happens to stationary objects?
  - Additional rays can still be used for antialiasing, depth of field effects, etc.

Motion Blur Examples







Next Time

- Leaving the world of standard ray tracing
- Introducing radiosity