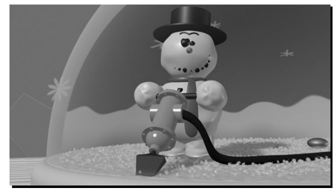
Now Playing:

Melody Day
Caribou
from *Andorra*
Released August 21, 2007

Movie: Knick Knack

Pixar, 1989



Ray Casting

Rick Skarbez, Instructor
COMP 575
October 25, 2007

Announcements

- Programming Assignment 2 (3D graphics in OpenGL) is due TONIGHT by 11:59pm
- Programming Assignment 3 (Rasterization) is out
  - Due NEXT Saturday, November 3 by 11:59pm
    - If you do hand in by Thursday midnight, +10 bonus points

Last Time

- Reviewed light transport
  - Lights
  - Materials
  - Cameras
- Talked about some features of real cameras
  - Lens effects
  - Film

Today

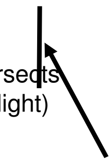
- Doing the math to cast rays

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

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



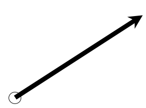
Ray Casting

- This is what we're going to discuss today
- As we saw on the last slide, ray casting is part of ray tracing
- Can also be used on its own
  - Basically gives you OpenGL-like results
    - No reflection/refraction

Generating an Image

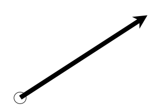
1. Generate the rays from the eye
   - One (or more) for each pixel
2. Figure out if those rays "see" anything
   - Compute ray-object intersections
3. Determine the color seen by the ray
   - Compute object-light interactions

Rays



- Recall that a ray is just a vector with a starting point
  - Ray = (Point, Vector)

Rays



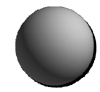
- Let a ray be defined by point **S** and vector **V**
- The parametric form of a ray expresses it as a function as some scalar t, giving the set of all points the ray passes through:
  - $r(t) = \mathbf{S} + t\mathbf{V}, 0 \leq t \leq \infty$
- This is the form we will use

Computing Ray-Object Intersections

- If a ray intersects an object, want to know the value of *t* where the intersection occurs:
  - t < 0: Intersection is behind the ray, ignore it
  - t = 0: Undefined
  - t > 0: Good intersection
- If there are multiple intersections, we want the one with the smallest t
  - This will be the closest surface

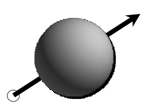
r(t) = **p** + t**d**

The Sphere



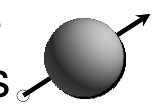
- For today's lecture, we're only going to consider one type of shape
  - The sphere
- The implicit equation for a sphere is:
  - $r^2 = (x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2$
  - If we assume it's centered at the origin:
    - $r^2 = x^2 + y^2 + z^2$

Ray-Sphere Intersections



- So, we want to find out where (or if) a ray intersects a sphere
  - Need to figure out what points on a ray represent valid solutions for the sphere equation

Ray-Sphere Intersections



Implicit Sphere

$$r^2 = x^2 + y^2 + z^2$$

Parametric Ray Equation

$$\mathbf{P}(t) = \mathbf{S} + t\mathbf{V}$$

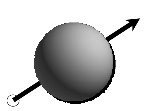
$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} S_x \\ S_y \\ S_z \end{bmatrix} + t \begin{bmatrix} V_x \\ V_y \\ V_z \end{bmatrix}$$

$$\begin{aligned} x &= S_x + tV_x \\ y &= S_y + tV_y \\ z &= S_z + tV_z \end{aligned}$$

Combined

$$r^2 = (S_x + tV_x)^2 + (S_y + tV_y)^2 + (S_z + tV_z)^2$$

Ray-Sphere Intersections



$$r^2 = (S_x + tV_x)^2 + (S_y + tV_y)^2 + (S_z + tV_z)^2$$

Want to solve for the value(s) of t that make this statement true:

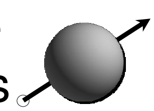
Expand

$$\begin{aligned} 0 \;=\; & S_x^2 + 2tS_xV_x + t^2V_x^2 \\ + \; & S_y^2 + 2tS_yV_y + t^2V_y^2 \\ + \; & S_z^2 + 2tS_zV_z + t^2V_z^2 \\ - \; & r^2 \end{aligned}$$

Rearrange

$$\begin{aligned} 0 \;=\; & t^2(V_x^2 + V_y^2 + V_z^2) \\ + \; & t(2S_xV_x + 2S_yV_y + 2S_zV_z) \\ + \; & S_x^2 + S_y^2 + S_z^2 - r^2 \end{aligned}$$

Ray-Sphere Intersections



$$\begin{aligned} 0 \;=\; & t^2(V_x^2 + V_y^2 + V_z^2) \\ + \; & t(2S_xV_x + 2S_yV_y + 2S_zV_z) \\ + \; & S_x^2 + S_y^2 + S_z^2 - r^2 \end{aligned}$$

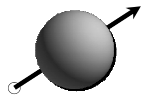
Note that this is in the form $at^2 + bt + c = 0$

Can solve with the quadratic formula:

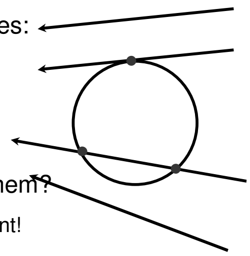
$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$\begin{aligned} a &= V_x^2 + V_y^2 + V_z^2 \\ b &= 2S_xV_x + 2S_yV_y + 2S_zV_z \\ c &= S_x^2 + S_y^2 + S_z^2 - r^2 \end{aligned}$$

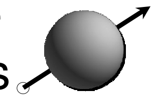
Ray-Sphere Intersections



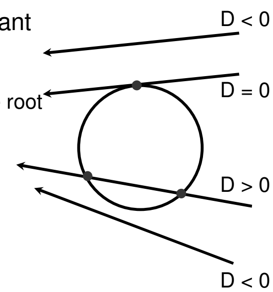
- There are three cases:
  - No intersection
  - 1 intersection
  - 2 intersections
- How do we detect them?
  - Check the discriminant!



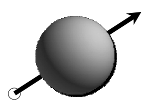
Ray-Sphere Intersections



- Using the discriminant
  - $D = b^2 - 4ac$
  - If D = 0, there is one root
  - If D > 0, there are 2 real roots
  - If D < 0, there are 2 imaginary roots



Ray-Sphere Intersections



- So, for the 3 cases
  - D < 0: Ray does not intersect the object
  - D = 0: One intersection; solve for t
  - D > 0: Two intersections
    - But we know we only want the closest
      - Can throw out the other solution

Ray-Object Intersections

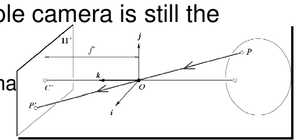
- We derived the math for sphere objects in detail
- The process is similar for other objects
  - Just need to work through the math
  - Using implicit surface definitions makes it easy

Generating an Image

1. Generate the rays from the eye
   - One (or more) for each pixel
~~2. Figure out if those rays “see” anythi~~ng
   - ~~Compute ray-object intersections~~
3. Determine the color seen by the ray
   - Compute object-light interactions

Generating Rays

- Now, given a ray, we know how to test if it intersects an object
  - But we don’t yet know how to generate the rays
- We talked a bit about lenses last time, but an ideal pinhole camera is still the simplest model
  - So let’s assume tha



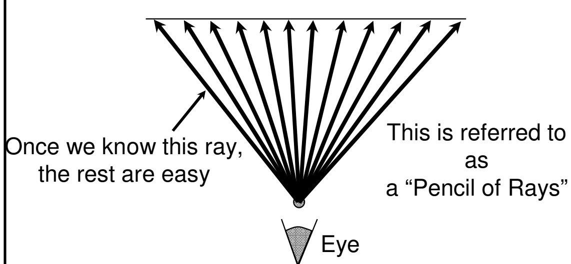
Generating Rays

- Recall the pinhole camera model
  - Every point **p** in the image is imaged through the center of projection **C** onto the image plane
    - Note that this means every point in the scene maps *to a ray*, originating at **C**
      - That is, r(t) = **C** + t**V**
      - **C** is the same for every ray, so just need to compute new **V**s

Generating Rays

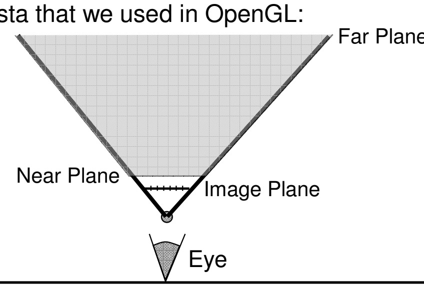
- Note that since this isn't a real camera, we can put the virtual image plane in front of the pinhole
  - This means we can solve for the ray directions and not worry about flipping the scene

Generating Rays in 2D



2D Frustum

- Note that this is the same idea as the frusta that we used in OpenGL:

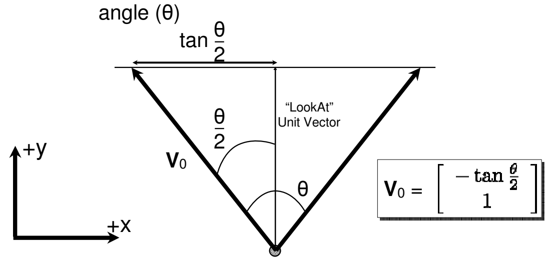


Building a Frustum

- So we need to know the same things that we did to build an OpenGL view frustum
  - Field of View
  - Aspect Ratio
  - Do we need near and far planes?
- Except now we need to build the camera matrix ourselves

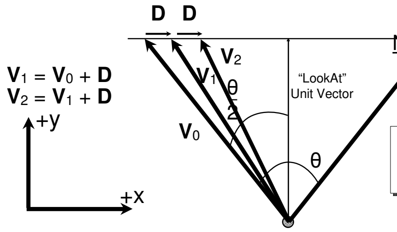
Field of View

- Recall that the field of view is how "wide" the view is
  - Not in terms of pixels, but in terms of viewing angle (θ)



Finding the Other Rays

- This tells us all we need to know
  - At least in 2D
  - All the other rays are just "offset" from the first



NOTE: *hRes* is the horizontal resolution

$$\mathbf{D} = \begin{bmatrix} \frac{2\tan\frac{\theta}{2}}{(hRes-1)} \\ 0 \end{bmatrix}$$

Generating Rays in 2D

- Note that we're assuming one ray per pixel
  - Can have more
- For all i from 0 to hRes:

$$\mathbf{V}_i = [\mathbf{D}\ \mathbf{V}_0] \begin{bmatrix} i \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} V_x \\ V_y \end{bmatrix} = \begin{bmatrix} \frac{2\tan\frac{\theta}{2}}{(hRes-1)} & -\tan\frac{\theta}{2} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ 1 \end{bmatrix}$$

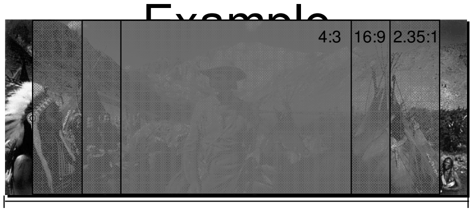
Extending to 3D

- So, this is all we need to know for 2D
  - Just generates a single row of rays
- For 3D, need to also know the vertical resolution
  - In the form of the aspect ratio

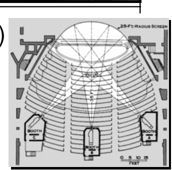
Quick Aside about Aspect Ratios

- With our virtual cameras, we can use any aspect ratio we want
- In the real world, though, some are most commonly used
  - 4:3 (standard video)
  - 16:9 (widescreen video)
  - 2.35:1 (many movies)

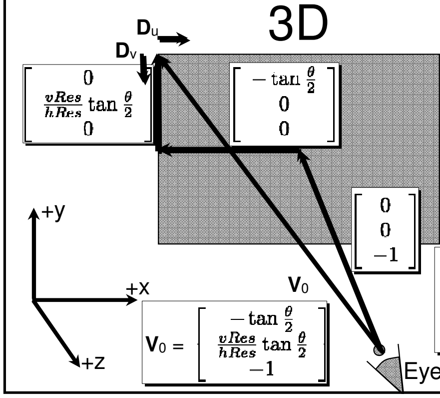
Aspect Ratios Example



Cinerama (2.59:1)



Generating Rays in 3D



$$\mathbf{D}_u = \begin{bmatrix} \frac{2\tan\frac{\theta}{2}}{(hRes-1)} \\ 0 \\ 0 \end{bmatrix}$$

$$\mathbf{D}_v = \begin{bmatrix} 0 \\ -\frac{2\frac{vRes}{hRes}\tan\frac{\theta}{2}}{vRes-1} \\ 0 \end{bmatrix}$$

$$\mathbf{V}_0 = \begin{bmatrix} -\tan\frac{\theta}{2} \\ \frac{vRes}{hRes}\tan\frac{\theta}{2} \\ -1 \end{bmatrix}$$

Generating Rays in 3D

$$\mathbf{V}_0 = \begin{bmatrix} -\tan\frac{\theta}{2} \\ \frac{vRes}{hRes}\tan\frac{\theta}{2} \\ -1 \end{bmatrix}$$

$$\mathbf{D}_u = \begin{bmatrix} \frac{2\tan\frac{\theta}{2}}{(hRes-1)} \\ 0 \\ 0 \end{bmatrix} \qquad \mathbf{D}_v = \begin{bmatrix} 0 \\ -\frac{2\frac{vRes}{hRes}\tan\frac{\theta}{2}}{vRes-1} \\ 0 \end{bmatrix}$$

$$\mathbf{V}_{i,j} = [\mathbf{D}_u\ \mathbf{D}_v\ \mathbf{V}_0] \begin{bmatrix} i \\ j \\ 1 \end{bmatrix}$$

A Basic 3D Camera Matrix

$$\begin{bmatrix} V_x \\ V_y \\ V_z \end{bmatrix} = \begin{bmatrix} \frac{2\tan\frac{\theta}{2}}{hRes-1} & 0 & -\tan\frac{\theta}{2} \\ 0 & -\frac{2\frac{vRes}{hRes}\tan\frac{\theta}{2}}{vRes-1} & \frac{vRes}{hRes}\tan\frac{\theta}{2} \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \\ 1 \end{bmatrix}$$

- Assumes:
  - Camera on the z-axis
  - Looking down -z
  - Ideal pinhole model
  - Fixed focal length (focal length = 1)

Generating an Image

~~1. Generate the rays from the eye~~
- ~~One (or more) for each pixel~~

~~2. Figure out if those rays "see" anything~~
- ~~Compute ray-object intersections~~

3. Determine the color seen by the ray
- Compute object-light interactions

Bonus Movie: Portal Half-Life 2 Mod

Available online:
http://www.youtube.com/watch?v=gKg3TUPQ8Sg

Determining Color

- Since we're not yet talking about tracing rays
  - Really just talking about OpenGL-style lighting and shading
    - Since surfaces are implicitly defined, can solve Phong lighting equation at every intersection

Review: Phong Lighting

Ambient, Diffuse, Specular

- $I = I_a(R_a, L_a) + I_d(\mathbf{n}, \mathbf{l}, R_d, L_d, a, b, c, d) + I_s(\mathbf{r}, \mathbf{v}, R_s, L_s, n, a, b, c, d)$
- $R_{something}$ represents how reflective the surface is
- $L_{something}$ represents the intensity of the light
- In practice, these are each 3-vectors
- One each for R, G, and B

Phong Reflection Model: Ambient Term

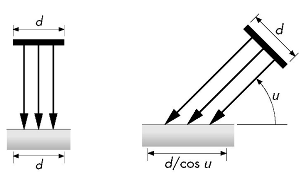
- Assume that ambient light is the same everywhere
  - Is this generally true?
- $I_a(R_a, L_a) = R_a * L_a$
  - The contribution of ambient light at a point is just the intensity of the light modulated by how reflective the surface is (for that color)

Phong Reflection Model: Diffuse Term

- $I_d(\mathbf{n}, \mathbf{l}, R_d, L_d, a, b, c, d) = (R_d / (a + bd + cd^2)) * max(\mathbf{l} \cdot \mathbf{n}, 0) * L_d$
- *a, b, c* : user-defined constants
- *d* : distance from the point to the light
- Let's consider these parts

Lambert's Cosine Law

- The incident angle of the incoming light affects its apparent intensity
  - Does the sun seem brighter at noon or 6pm?
- Why?



"Noon" "Evening"

Phong Reflection Model: Diffuse Term

- We already know how to get the cosine between the light direction and the normal
  - **n • l**
- What happens if the surface is facing away from the light?
  - That's why we use *max*(**n • l**, 0)
  - Why not just take |**n • l**|?

Phong Reflection Model: Diffuse Term

- In the real world, lights seem to get dimmer as they get further away
  - Intensity decreases with distance
- We can simulate that by adding an attenuation term
  - $(R_d / (a + bd + cd^2))$
  - User can choose the *a,b,c* constants to achieve the desired "look"

Phong Reflection Model: Specular Term

- $I_s(\mathbf{r}, \mathbf{v}, R_s, L_s, n, a, b, c, d) = (R_s / (a + bd + cd^2)) * max(\mathbf{r} \cdot \mathbf{v}, 0)^n * L_s$
- Why **r • v**?
  - Reflection is strongest in the direction of the reflection vector
  - **r • v** is maximized when the viewpoint vector (or really the vector to the viewpoint) is in the same direction as **r**
- What is n?

Generating an Image

~~1. Generate the rays from the eye~~
- ~~One (or more) for each pixel~~

~~2. Figure out if those rays "see" anything~~
- ~~Compute ray-object intersections~~

~~3. Determine the color seen by the ray~~
- ~~Compute object-light interactions~~

Review

- Reviewed the basic ray tracing algorithm
  - Talked about how ray casting is used
- Derived the math for generating camera rays
- Derived the math for computing ray intersections
  - For a sphere

Next Time

- Extending the camera matrix to be more general
- Covering some software engineering notes relating to building a ray tracer