Now Playing:

Quicksand Under Carpet
New Radiant Storm King
from *Leftover Blues: 1991-2003*
Released 2004

Movie:
Geri's Game

Pixar, 1997
Academny Award Winner, Best Short Film

Ray Casting 2

Rick Skarbez, Instructor
COMP 575
October 30, 2007

Announcements

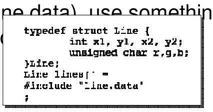
- Programming Assignment 3 (Rasterization) is out
  - Due Saturday, November 3 by 11:59pm
    - If you do hand in by Thursday midnight, +10 bonus points
- Remember that you need to talk to me about your final project
  - Send an email to schedule a meeting, or come by office hours

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 co



```
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.

Last Time

- Derived the math for ray casting
  - Intersecting rays and objects
  - Generating rays
    - aka generating a camera matrix
  - Coloring pixels
    - Phong shading for each ray

Today

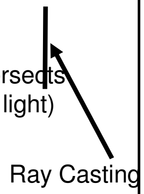
- Extending some of the ideas from last time
- Briefly discussing the software architecture of a raycaster
- Optional course feedback survey

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
```



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

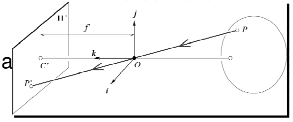
Computing Ray-Object Intersections

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

$r(t) = \mathbf{p} + t\mathbf{d}$

Generating Rays

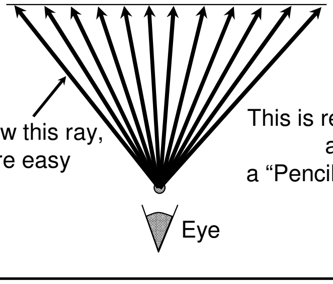
- Now, given a ray, we know how to test if it intersects an object
  - But we don’t yet know how to generate the rays
- We talked a bit about lenses last time, but an ideal pinhole camera is still the simplest model
  - So let’s assume tha



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

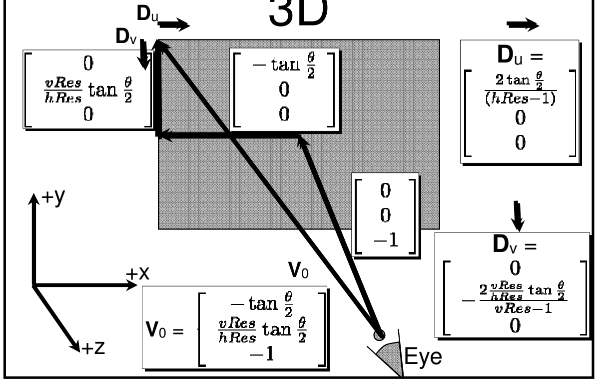
Generating Rays in 2D



Once we know this ray, the rest are easy

This is referred to as a “Pencil of Rays”

Generating Rays in 3D



$$\begin{bmatrix} 0 \\ \frac{vRes}{hRes}\tan\frac{\theta}{2} \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} -\tan\frac{\theta}{2} \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}$$

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

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

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

Generating Rays in 3D

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

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

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

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

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)

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

Doing it Better

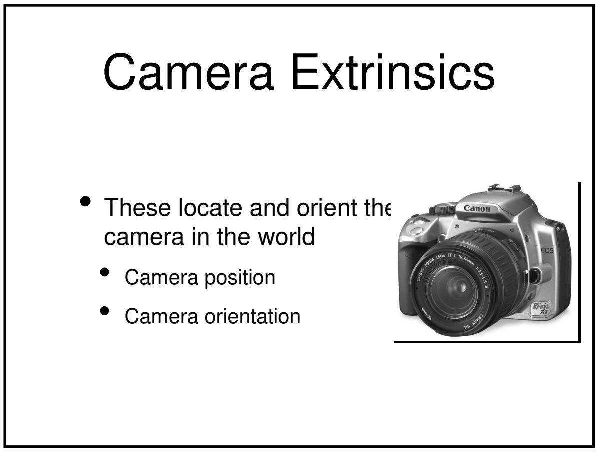
- We now know how to generate a simple raycasted image
- However, we've assumed only a very simple/limited camera definition
- Now we're going to extend our notion of cameras

Camera Intrinsics and Extrinsics

- We normally divide camera properties into two classes: intrinsic and extrinsic
  - *Intrinsic* properties are those belonging to the camera itsel
    - *Intrinsic* properties are *insi* the camera
  - *Extrinsic* properties define how the camera is situated in the world

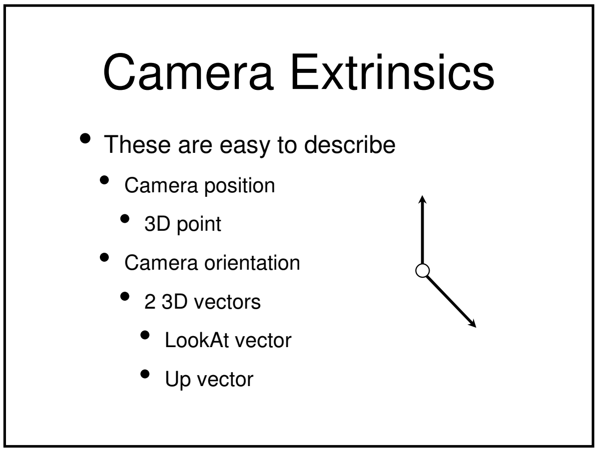
Camera Intrinsics

- These describe the behavior of the camera
  - Focal length
  - Aspect ratio
  - Resolution
  - Aperture
  - Shutter speed
  - etc.



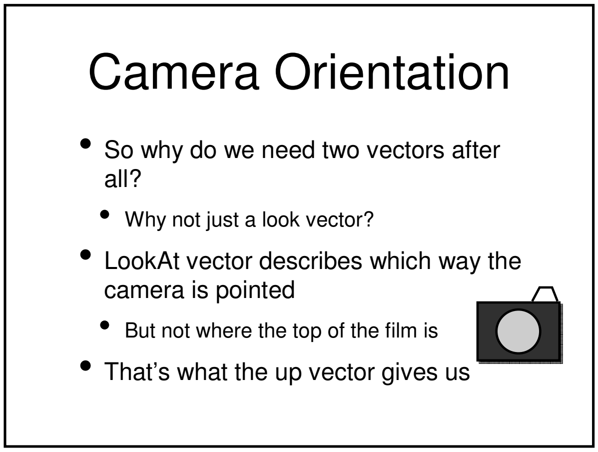
Camera Extrinsics

- These locate and orient the camera in the world
  - Camera position
  - Camera orientation



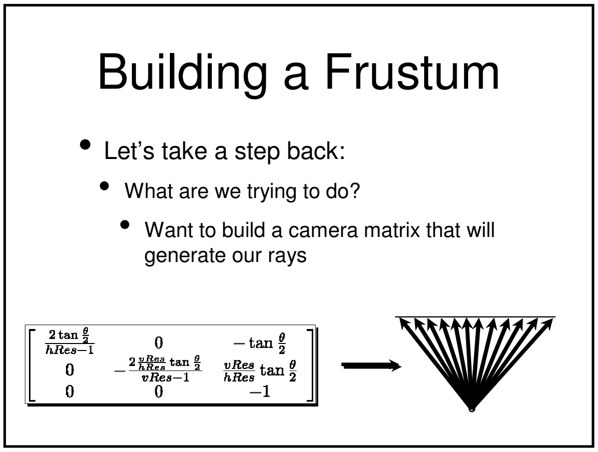
Camera Extrinsics

- These are easy to describe
  - Camera position
    - 3D point
  - Camera orientation
    - 2 3D vectors
      - LookAt vector
      - Up vector



Camera Orientation

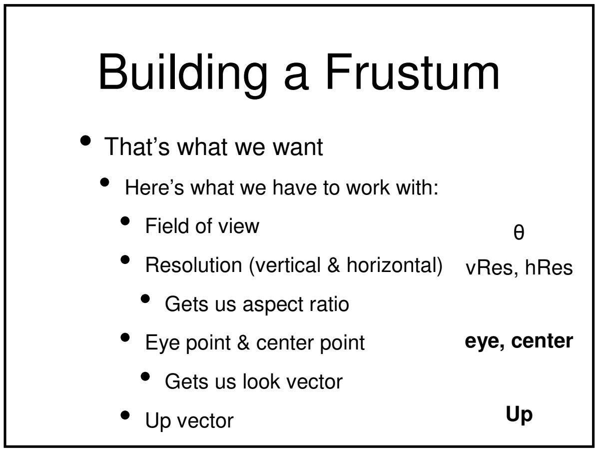
- So why do we need two vectors after all?
  - Why not just a look vector?
- LookAt vector describes which way the camera is pointed
  - But not where the top of the film is
- That's what the up vector gives us



Building a Frustum

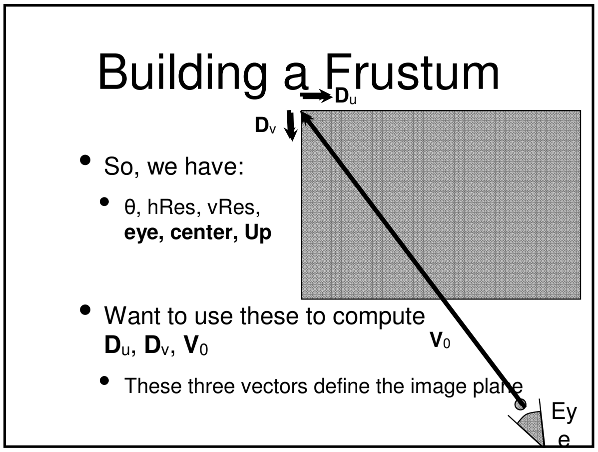
- Let's take a step back:
  - What are we trying to do?
    - Want to build a camera matrix that will generate our rays

$$\begin{bmatrix} \frac{2\tan\frac{\theta}{2}}{hRes-1} & 0 & -\tan\frac{\theta}{2} \\ 0 & -\frac{2\frac{vRes}{hRes}\tan\frac{\theta}{2}}{vRes-1} & \frac{vRes}{hRes}\tan\frac{\theta}{2} \\ 0 & 0 & -1 \end{bmatrix}$$



Building a Frustum

- That's what we want
  - Here's what we have to work with:
    - Field of view θ
    - Resolution (vertical & horizontal) vRes, hRes
      - Gets us aspect ratio
    - Eye point & center point **eye, center**
      - Gets us look vector
    - Up vector **Up**



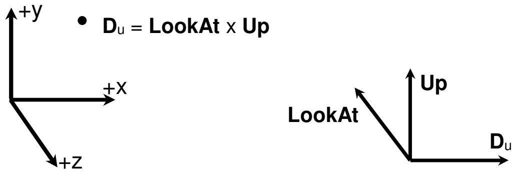
Building a Frustum

- So, we have:
  - θ, hRes, vRes, **eye, center, Up**
- Want to use these to compute **D**$_u$, **D**$_v$, **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



Finding $\mathbf{D}_u$

- So now we have a vector that points in the correct direction
  - But we originally said that $\mathbf{D}_u$ was one pixel width long
  - If **LookAt** and **Up** are unit vectors, $\mathbf{D}_u$ now has length 1
    - Too long
- We’ll need to rescale our vectors

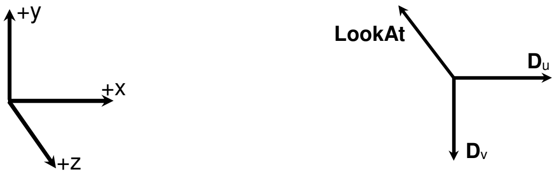
The “Down” Vector

- We also need a vector in the $\mathbf{D}_v$ direction
  - That is, we need a vector perpendicular to $\mathbf{D}_u$ and **LookAt**
  - Could we just use -**Up**?
    - Not necessarily
      - We have not required **Up** to be perpendicular to **LookAt**

Finding $\mathbf{D}_v$

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

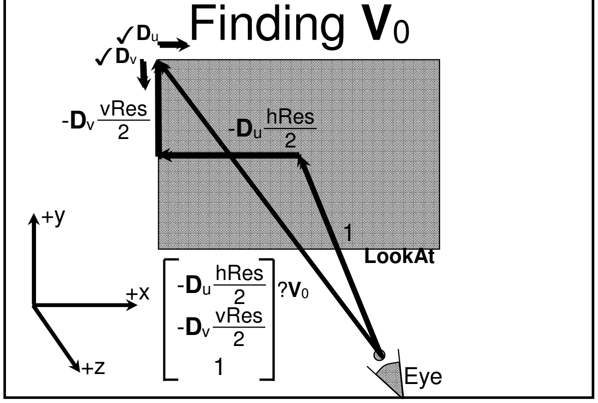
Note that if **LookAt** and **Up** are both unit vectors, then $\mathbf{D}_u$ and $\mathbf{D}_v$ are also unit vectors



Finding $\mathbf{V}_0$

- So now we have 2 out of our 3 vectors
  - Need to find the “origin” vector

Finding $\mathbf{V}_0$



$$\begin{bmatrix} -\mathbf{D}_u\frac{hRes}{2} \\ -\mathbf{D}_v\frac{vRes}{2} \\ 1 \end{bmatrix} ?\mathbf{V}_0$$

Focal Length

- Remember this equation? $\frac{1}{z'} - \frac{1}{z} = \frac{1}{f}$

Distance to film

Focal Length

Distance to scene

- We want to know how far along the **LookAt** vector the image plane lies
  - Before, we assumed it was 1, so the distance from the center of the image to the left edge was just tan (θ/2)
  - Now it is hRes/2

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

Complete Frustum Specification

Given Points: **eye, center**

Given Unit Vector: **Up**

Given FOV Angle: θ

Given Dimensions: vRes, hRes

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

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

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

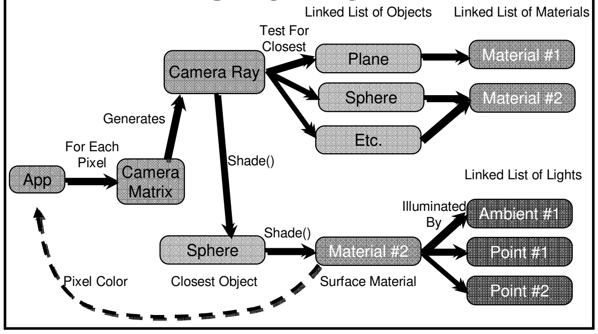
NOTE:
Normalize $\mathbf{D}_u$ and $\mathbf{D}_v$!

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

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

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

Raycaster System Overview



Next Time

- Figuring out what raycasting/raytracing really buys us over OpenGL
  - Shadows
  - Reflection
  - Refraction
    - i.e. Real Ray Tracing

Course Feedback Survey