3D Viewing
Camera Transformations

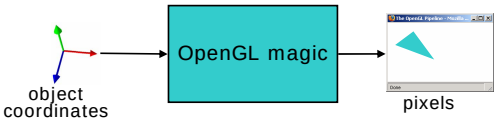
CS148: Intro to CG
Instructor: Dan Morris
TA: Sean Walker
July 7, 2005

Outline for today

- Overview: 3D → 2D
- Viewing
- Video break
- Projection
- The depth buffer

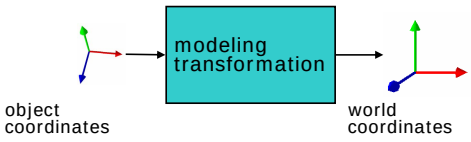
OpenGL in a nutshell

- We can build primitives from vertices
- We can build objects from primitives
- We usually model objects in a "convenient coordinate frame"
  - E.g. drawSqare(), glutSolidSphere()



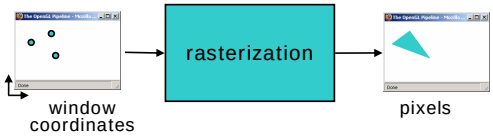
What have we done so far?

- We know how to take all points in our object and put them in a global frame
- Objects may have their own reference frames, but using GL transformations, every vertex ends up in the global frame



What have we done so far?

- We know how to turn primitives into pixels if we have their pixel locations



Working backwards

- We learned (almost) how to convert OpenGL coordinates into window coordinates:

  glViewport(x,y,w,h);

- What does glViewport() really do?
- From the documentation:

  Specifies the affine transformation of x and y from normalized device coordinates to window coordinates.

- What the $*%^ are normalized device coordinates?

Normalized Device Coordinates

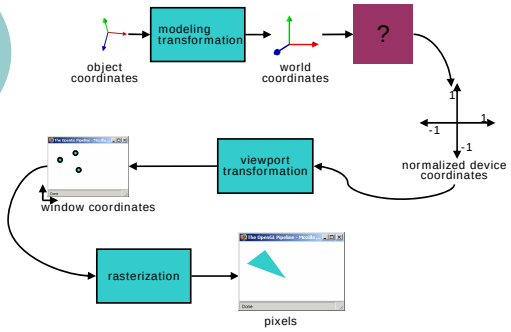
- On any computer in the world, with any window size, OpenGL maps the range [-1,1] to the window
- At the end of all my transformations:
  - (0,0) will be the center of the window,
  - (1,1) will be at the upper-right
- If I want a vertex to show up at the center of the screen, my transformations had better transform it to (0,0,z).

The Viewport Transformation

- We already know how to turn NDC into window coordinates
- Just call glViewPort()

What is the transformation set up by glViewport(x,y,w,h)?

What's left?

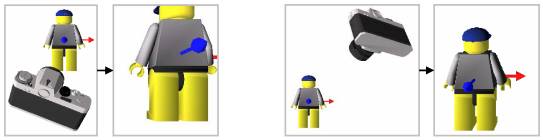


Outline for today

- Overview: 3D → 2D
- Viewing
- Video break
- Projection
- The depth buffer

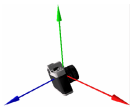
Viewing

- I have a world with objects in it
- Useful to think of a virtual camera somewhere in that world, looking at whatever I want the user to see



Eye Coordinates

- …but GL doesn't have a setCameraPosition() function*
- The OpenGL camera is always sitting at the origin and looking straight down the –z axis, with y pointed up
- We'll call this coordinate system "eye coordinates"



The Viewing Transformation

- In order to define our own camera, we'll apply a transformation that moves *everything* into view
- In other words, we'll translate our world coordinates into eye coordinates
- This is the "viewing transformation", and it's equivalent to placing our virtual camera in our virtual world

The ModelView Matrix Revisited

- To do this in OpenGL, the very first transform to go onto our modelview stack will be the viewing transform
- When you fire up project 2, the current coordinate frame is already *world* coordinates, because we defined a viewing transform for you
- The modelview matrix encompasses the modeling and viewing transformations

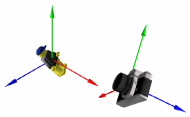
Defining a camera

- Let's assume we want the camera to be
  - Sitting at point E (eye) in world coords
  - Looking at point L (look) in world coords
- Does this define a unique coordinate system?
- Also need an 'up vector' that tells us how the camera is oriented around its "look axis"

Building a viewing matrix

- So let's think of our job as implementing the function:

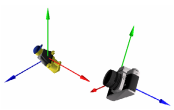
matrix buildViewMatrix(
  point E,
  point L,
  vector up);



E, L, and up are in the *world* frame…

How do we build a viewing matrix?

- The viewing matrix maps points in the world frame to points in the camera frame
- Two steps:
  - Apply a translation to put their origins at the same place
  - Apply a rotation to make their axes line up



Translation First

- We're good at translation…

What translation do I apply to put the camera at point (ex, ey, ez)?

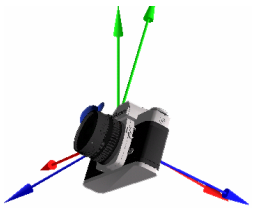
$$\begin{bmatrix} 1 & 0 & 0 & -ex \\ 0 & 1 & 0 & -ey \\ 0 & 0 & 1 & -ez \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

OpenGL, please translate all future vertices by (-ex,-ey,-ez)

- To "move the camera backward", we would move the whole world forward…

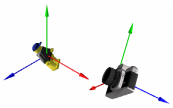
Now rotation

- Now the current origin is at the camera frame origin, but I need to align the axes…



Now rotation

matrix buildViewMatrix(
  point E,
  point L,
  vector up);



What vector in the *world* frame should match the *z* axis in the camera frame?

What vector in the *world* frame should match the y axis in the camera frame?

What vector in the *world* frame should match the x axis in the camera frame?

The magic vectors

- The z axis: E - L
  - The camera always looks down its -z axis, and we want to look from E to L
- The y axis: up
  - The camera is always oriented with y pointing up
- The x axis: up x (E-L)

- All three vectors are perpendicular…

The secrets of rotation matrices

$$\begin{bmatrix} r_{11} \\ r_{21} \\ r_{31} \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

WOW!

- Rotation matrices have some amazing properties…
  - All columns are unit vectors!
  - The columns of a rotation matrix define the vectors to which it rotates the coordinate axes!
  - $R^T = R^{-1}$

Hand-made rotation matrices

- If I want a rotation matrix that maps the x axis to the unit vector(a,b,c), what should the first column look like?

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} a & r_{12} & r_{13} \\ b & r_{22} & r_{23} \\ c & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Hand-made rotation matrices

- If I wanted to…
  - Rotate the z axis to the vector E-L
  - Rotate the y axis to the vector up
  - Rotate the x axis to the vector up x (E-L)
- Define unit vectors:
  - f = (E-L) / |E-L|
  - u = up / |up|
  - $s = u \times f$

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} s_x & u_x & f_x \\ s_y & u_y & f_y \\ s_z & u_z & f_z \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

But really that's not what I want…

- I *have* three unit vectors (f, u, and s) and I want to *rotate them* into the x, y, and z axes
- **What matrix will rotate f, u, and s into the x, y, and z axes?**

$$R^{-1} = R^T = \begin{bmatrix} s_x & u_x & f_x \\ s_y & u_y & f_y \\ s_z & u_z & f_z \end{bmatrix}^T = \begin{bmatrix} s_x & s_y & s_z \\ u_x & u_y & u_z \\ f_x & f_y & f_z \end{bmatrix}$$

Finally, the viewing matrix…

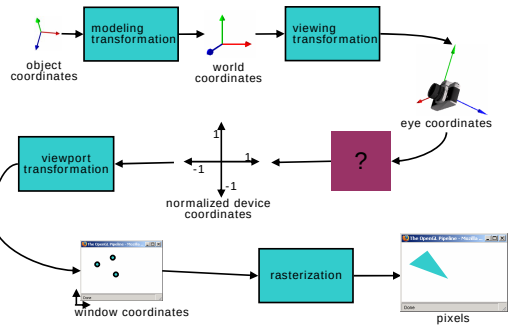
$$\begin{bmatrix} peye_x \\ peye_y \\ peye_z \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & s_y & s_z & 0 \\ u_x & u_y & u_z & 0 \\ f_x & f_y & f_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} pworld_x \\ pworld_y \\ pworld_z \\ 1 \end{bmatrix}$$

- Reminder: all this work was to transform points in *world* coordinates to points in *eye* coordinates

Viewing in OpenGL [cube.cpp]

- gluLookAt() does everything we just talked about
  - Multiplies the current transformation by the matrix we just built
  - Usually the current matrix is identity, since this is usually the first thing to happen on the modelview stack
- We even used the same notation as the gluLookAt() docs
- This is how you usually position a camera in OpenGL

What's left?



Video break: Final Fantasy Trailer

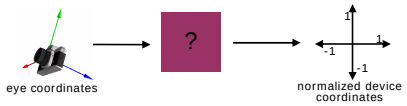
- CG pipeline for movies:
  - Modeling
  - Animation / MoCap / Simulation
  - Rendering
  - Compositing

Outline for today

- Overview: 3D → 2D
- Viewing
- Video break
- Projection
- The depth buffer

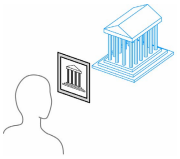
What have we done so far?

- All of our coordinates have been transformed into eye coordinates
- (0,0,0) is at the center of my world, the -z axis points away from me
- Still need to map 3D eye coordinates to 2D normalized device coordinates

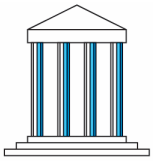


Projection

- Turning 3D eye coordinates into normalized device coordinates
- "Projecting" the world onto a 2D image plane



Types of projection



orthographic (parallel)

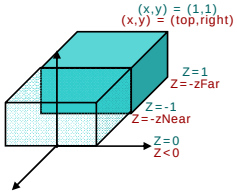
perspective

Orthographic projection

- Squeeze a certain range of what's in front of the camera into NDC without any distortion
- OpenGL will clip away everything outside x = [-1,1], y = [-1,1]
- What about z?
  - OpenGL will also clip away everything outside the range z = [-1,1]
  - Near and far "clip planes": the closest and farthest things that we want to see

Orthographic projection

- Can think of orthographic projection as defining a *view volume* that's shaped like a box



NDC's
Eye coordinates

Orthographic projection in GL

- glOrtho(left,right,bottom,top, zNear,zFar)
- From the docs:

```
multiply the current matrix with an
orthographic matrix
```

$$\begin{pmatrix} \frac{2}{right-left} & 0 & 0 & t_x \\ 0 & \frac{2}{top-bottom} & 0 & t_y \\ 0 & 0 & \frac{-2}{zFar-zNear} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

What can you tell me about this transformation just by looking at it?

Orthographic projection in GL

- Remember gluOrtho2D?
- From the docs:

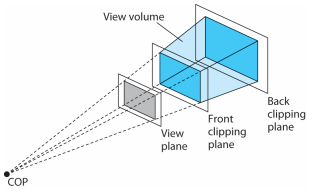
```
The gluOrtho2D function defines a 2-
D orthographic projection matrix.
This is equivalent to calling
glOrtho with near = -1 and far = 1.
```

Perspective projection

- Same deal... need to get eye coordinates into NDC's
- Stuff that's further away should look smaller
- E.g. a point at (1,1,-20) in eye coordinates should end up closer to the z-axis in NDC's than a point at (1,1,-5)

Perspective projection

- What is the shape of our view volume in a perspective projection?
- It's a frustum...
- What the $%^! is a frustum?



Perspective projection

- We need a matrix that will divide x and y by a value that depends on z
- Can't express this with a typical matrix multiplication...
- If only we had some way of dividing each point by a different value...

The perspective projection matrix

$$\begin{bmatrix} (sx)x_{eye}+tz \\ (sy)y_{eye}+ty \\ sz(z_{eye})+tz \\ z_{eye} \end{bmatrix} = \begin{bmatrix} sx & 0 & 0 & tx \\ 0 & sy & 0 & ty \\ 0 & 0 & sz & tz \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ 1 \end{bmatrix}$$

Holy non-zero homogeneous coordinates, batman!

- bigger z → bigger w → smaller x and y (after division by w)

Perspective projection in OpenGL

- glFrustum(left, right, bottom, top, near, far);
  - From the docs: multiply the current matrix by a perspective matrix
- gluPerspective(fovy, aspect, near, far)
  - From the docs: set up a perspective projection matrix
  - Takes a field-of-view angle in degrees and an aspect ratio (y/x)

Summary: Camera transforms in GL [cube.cpp]

- A typical 3D program sets up a perspective projection when it first starts up:

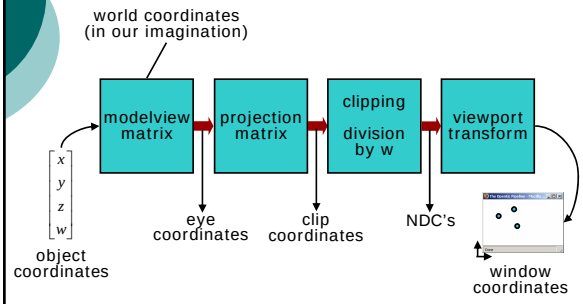
```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(45.0,1.5,0.2,10.0);
glMatrixMode(GL_MODELVIEW);
```

- A typical program sets up the camera every frame:

```
glLoadIdentity();
gluLookAt(ex,ey,ez,lookx,looky,lookz,
    upx,upy,upz);

// Do all my rendering...
```

The GL transformation pipeline



Outline for today

- Overview: 3D → 2D
- Viewing
- Video break
- Projection
- The depth buffer

What about z?

- After all this work, we're going to get vertices in window coordinates, then rasterize them to get pixels
- What happens when two primitives rasterize to the same pixel location?
- Intuitively, we want the pixel that was *closer* to the eye to get drawn, but not the other one

GL's solution: the depth buffer

- OpenGL keeps an extra array of floats, exactly the same size as the frame buffer
- This separate buffer stores the NDC **z** value for every pixel in the framebuffer

The depth test

- I'm busy rasterizing a triangle, and I decide to put a pixel at location (x,y) (just like pp1 in CS148)
- Before I sent a new color to the framebuffer, I say "is the new pixel's z-value less than the value in the depth buffer"?
- If not, I *cull* (throw away) this pixel because it "failed the depth test"

Interacting with the GL depth buffer

- You almost never have to work with the depth buffer directly… your only job is clearing it after every frame

```
glClear(GL_DEPTH_BUFFER_BIT |
  GL_COLOR_BUFFER_BIT);
```

In NDC's, what value should "clearing" the depth buffer set every pixel's depth value to?

Interacting with the GL depth buffer

- You do need to turn the depth test on at the beginning of your program:

```
glEnable(GL_DEPTH_TEST);
```

When might I want to turn depth-testing off?

Interacting with the GL depth buffer [cube.cpp] [broken tobor example]

What happens if I turn depth testing off for a typical scene?

What happens if I turn off the depth-buffer-clear for an entire scene?

What happens if I turn off the color-buffer-clear for an entire scene?

Next Time

- Polygonal meshes
- Surface normals