Last Time

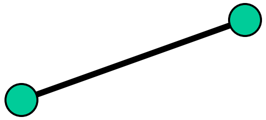
- Toolkits
- Transformations
  - Rotation is complex in 3D
  - Any rotation can be expressed with an axis and angle approach
  - Points on the axis do not move anywhere, points off the axis rotate around it
  - **The axis passes through the origin**

Today

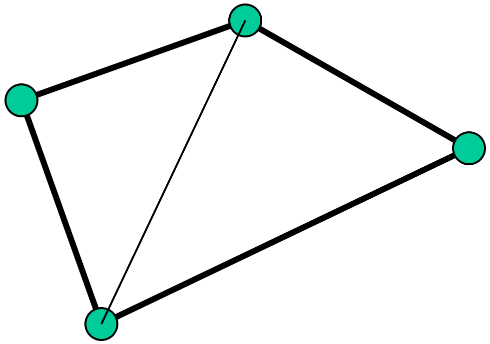
- Viewing
- Orthographic viewing
- Homework 3

Modeling 101

- For the moment assume that all geometry consists of points, lines and faces
- Line: A segment between two endpoints



- Face: A planar area bounded by line segments
  - Any face can be *triangulated* (broken into triangles)



Modeling and OpenGL

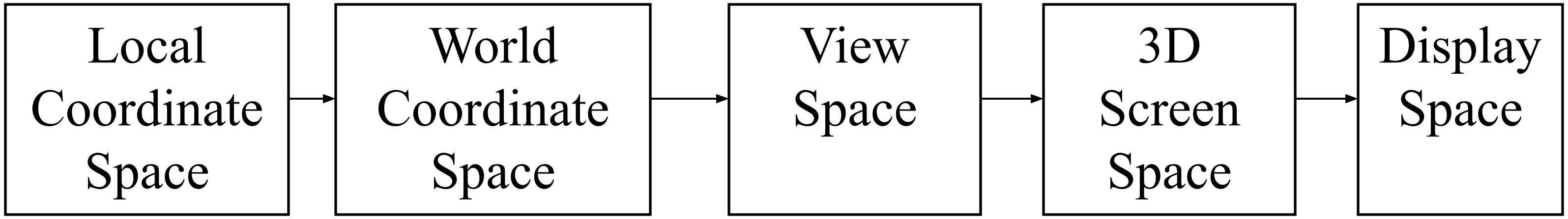
- In OpenGL, all geometry is specified by stating which type of object and then giving the vertices that define it
- `glBegin(…) …glEnd()`
- `glVertex[34][fdv]`
  - Three or four components (regular or homogeneous)
  - Float, double or vector (eg float[3])
- Chapter 2 of the red book

Rendering

- Generate an image showing the contents of some region of space
  - The region is called the *view volume,* and it is defined by the user
- Determine where each object should go in the image
  - *Viewing, Projection*
- Determine which object is in front at each pixel
  - *Hidden surface elimination, Hidden surface removal, Visibility*
- Determine what color it is
  - *Lighting, Shading*

Graphics Pipeline

- Graphics hardware employs a sequence of coordinate systems
  - The location of the geometry is expressed in each coordinate system in turn, and modified along the way
  - The movement of geometry through these spaces is considered a pipeline



Local Coordinate Space → World Coordinate Space → View Space → 3D Screen Space → Display Space

Local Coordinate Space

- It is easiest to define individual objects in a local coordinate system
  - For instance, a cube is easiest to define with faces parallel to the coordinate axis
- Key idea: Object instantiation
  - Define an object in a local coordinate system
  - Use it multiple times by copying it and transforming it into the global system
  - This is the only effective way to have libraries of 3D objects, and such libraries do exist

Global Coordinate System

- *Everything* in the world is transformed into one coordinate system - the *global coordinate system*
  - Actually, some things, like dashboards, may be defined in a different space, but we’ll ignore that
- Lighting is defined in this space
  - The locations, brightness’ and types of lights
- The camera is defined with respect to this space
- Some higher level operations, such as advanced visibility computations, can be done here

View Space

- Associate a set of axes with the *image plane*
  - The image plane is the plane in space on which the image should “appear,” like the film plane of a camera
  - One normal to the image plane
  - One up in the image plane
  - One right in the image plane
  - These three axes define a coordinate system (a rigid body transform of the world system)
- Some camera parameters are easiest to define in this space
  - Focal length, image size
- Depth is represented by a single number in this space
  - The “normal to image plane” coordinate

3D Screen Space

- Transform view space into a cube: [-1,1]×[-1,1]×[-1,1]
  - The cube is the *canonical view volume*
  - Parallel sides make many operations easier
- Tasks to do:
  - Clipping – decide what you can see
  - Rasterization - decide which pixels are covered
  - Hidden surface removal - decide what is in front
  - Shading - decide what color things are

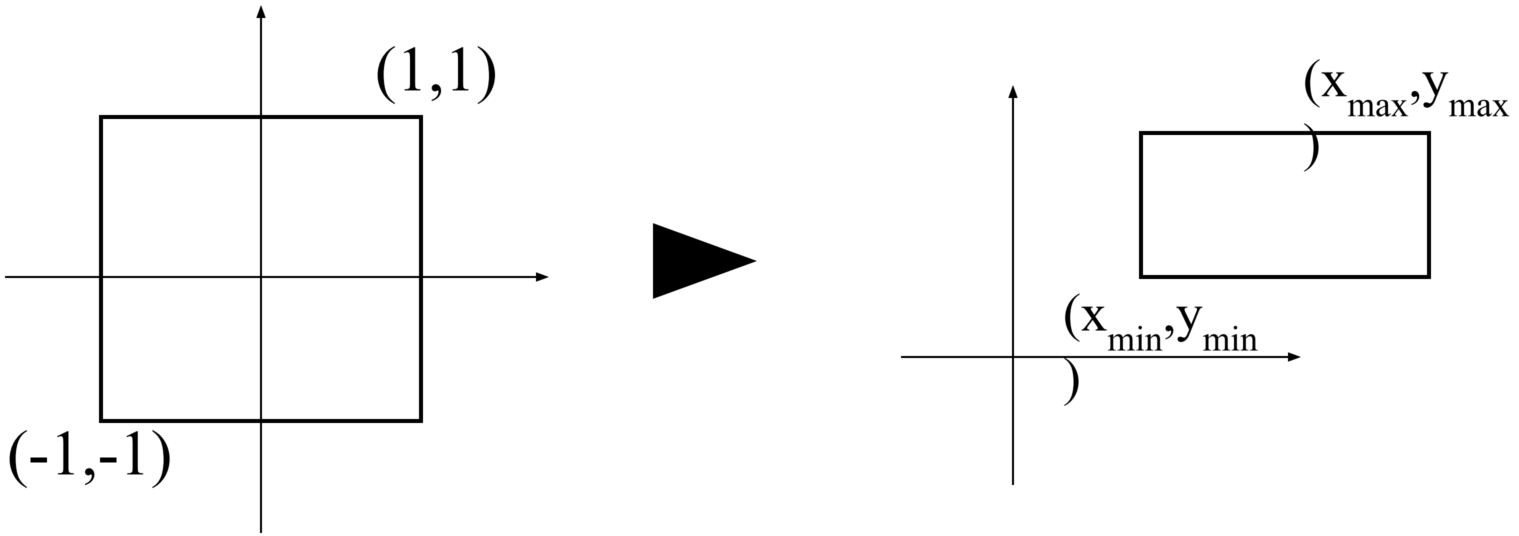
Window Space

- Also called screen space (confusing)
- Convert the virtual screen into real screen coordinates
  - Drop the depth coordinates and translate
- The windowing system takes care of this

3D Screen to Window Transform

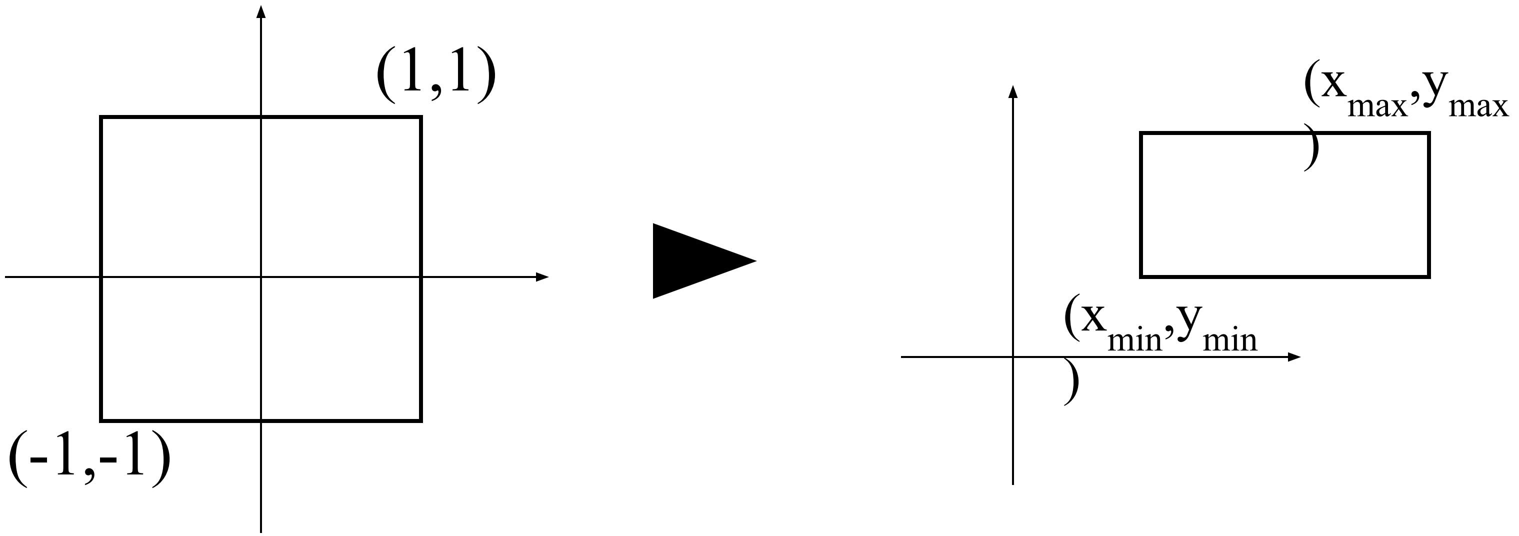
- Typically, windows are specified by an origin, width and height
  - Origin is either bottom left or top left corner, expressed as *(x,y)* on the total visible screen on the monitor or in the framebuffer
- This representation can be converted to (x_{min},y_{min}) and (x_{max},y_{max})
- 3D Screen Space goes from (-1,-1,-1) to (1,1,1)
  - Lets say we want to leave *z* unchanged
- What basic transformations will be involved in the total transformation from 3D screen to window coordinates?

3D Screen to Window Transform



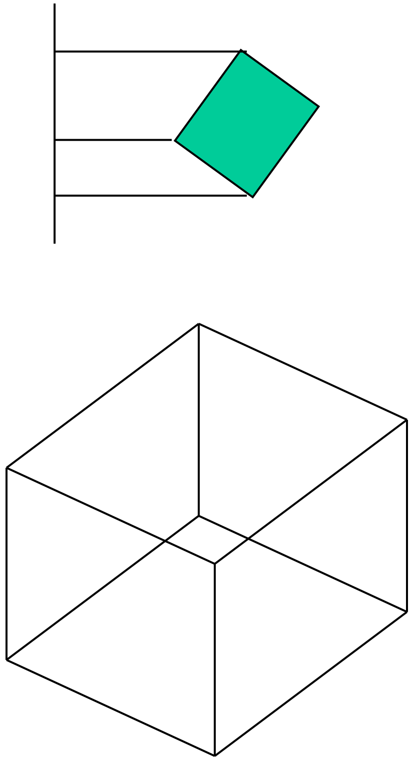
- How much do we translate?
- How much do we scale?

3D Screen to Window Transform



$$\begin{bmatrix} x_{pixel} \\ y_{pixel} \\ z_{pixel} \\ 1 \end{bmatrix} = \begin{bmatrix} (x_{\max} - x_{\min})/2 & 0 & 0 & (x_{\max} + x_{\min})/2 \\ 0 & (y_{\max} - y_{\min})/2 & 0 & (y_{\max} + y_{\min})/2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{screen} \\ y_{screen} \\ z_{screen} \\ 1 \end{bmatrix}$$

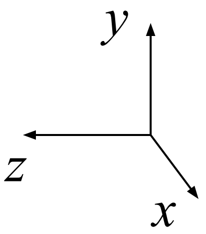
Orthographic Projection

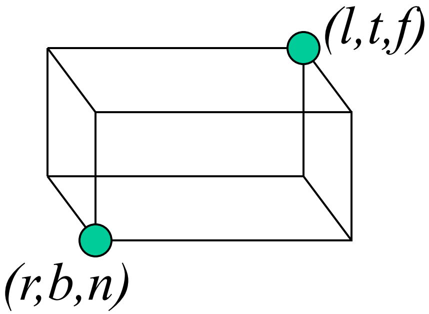


- Orthographic projection projects all the points in the world along parallel lines onto the image plane
  - Projection lines are perpendicular to the image plane
  - Like a camera with infinite focal length
- The result is that parallel lines in the world project to parallel lines in the image, and ratios of lengths are preserved
  - This is important in some applications, like medical imaging and some computer aided design tasks

Simple Orthographic Projection

- Specify the region of space that we wish to render as a *view volume*
- Assume that the viewer is looking in the $-z$ direction, with x to the right and y up
  - Assuming a right-handed coordinate system
- The view volume has:
  - a near plane at $z=n$
  - a far plane at $z=f$, $(f < n)$
  - a left plane at $x=l$
  - a right plane at $x=r$, $(r>l)$
  - a top plane at $y=t$
  - and a bottom plane at $y=b$, $(b<t)$





Rendering the Volume

- To project, map the view volume onto the canonical view volume
  - After that, we know how to map the view volume to the window
- The mapping looks just like the one for screen->window:

$$\begin{bmatrix} x_{screen} \\ y_{screen} \\ z_{screen} \\ 1 \end{bmatrix} = \begin{bmatrix} 2/(r-l) & 0 & 0 & -(r+l)/(r-l) \\ 0 & 2/(t-b) & 0 & -(t+b)/(t-b) \\ 0 & 0 & 2/(n-f) & -(n+f)/(n-f) \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{view} \\ y_{view} \\ z_{view} \\ 1 \end{bmatrix}$$

$$\mathbf{x}_{screen} = \mathbf{M}_{view->screen} \mathbf{x}_{view}$$

General Orthographic Projection

- We could look at the world from any direction, not just along $-z$
- The image could rotated in any way about the viewing direction: x need not be right, and y need not be up
- How can we specify the view under these circumstances?

Specifying a View

- The location of the image plane in space
  - A point in space for the center of the image plane, (c_x,c_y,c_z)
- The direction in which we are looking
  - Specified as a vector that points *back toward the viewer:* (d_x,d_y,d_z)
  - This vector will be normal to the image plane
- A direction that we want to appear *up* in the image
  - This vector does not have to be perpendicular to n
- We also need the size of the view volume – l,r,t,b,n,f
  - Specified with respect to the image plane, not the world

Getting there…

- We wish to end up in the “simple” situation, so we need a coordinate system with:
  - A vector toward the viewer
  - One pointing right in the image plane
  - One pointing up in the image plane
  - The origin at the center of the image
- We must:
  - Define such a coordinate system, *view space*
  - Transform points from the world space into view space
  - Apply our simple projection from before

View Space

- Given our camera definition:
  - Which point is the origin of view space?
  - Which direction is the normal to the view plane, *n*?
  - How do we find the right vector, *u*?
  - How do we find the up vector, *v*?
- Given these points, how do we do the transformation?

View Space

- The origin is at the center of the image plane: (c_x, c_y, c_z)
- The normal vector is the normalized viewing direction: $n = \hat{d}$
- We know which way up should be, and we know we have a right handed system, so $u = up \times n$, normalized: $\hat{u}$
- We have two vectors in a right handed system, so to get the third: $v = n \times u$

World to View

- We must translate the world so the origin is at (c_x,c_y,c_z)
- To complete the transformation we need to do a rotation
- After this rotation:
  - The direction *u* in world space should be the direction (1,0,0) in view space
  - The vector *v* should be (0,1,0)
  - The vector *n* should be (0,0,1)
- The matrix that does that is:

$$\begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

All Together

- We apply a translation and then a rotation, so the result is:

$$\mathbf{M}_{world->view} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & -c_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} u_x & u_y & u_z & -\mathbf{u} \bullet \mathbf{c} \\ v_x & v_y & v_z & -\mathbf{v} \bullet \mathbf{c} \\ n_x & n_y & n_z & -\mathbf{n} \bullet \mathbf{c} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- And to go all the way from world to screen:

$$\mathbf{M}_{world->screen} = \mathbf{M}_{view->screen} \mathbf{M}_{world->view}$$

$$\mathbf{x}_{screen} = \mathbf{M}_{world->screen} \mathbf{x}_{world}$$

OpenGL and Transformations

- OpenGL internally stores several matrices that control viewing of the scene
  - The MODELVIEW matrix is intended to capture all the transformations up to the view space
  - The PROJECTION matrix captures the view to screen conversion
- You also specify the mapping from the canonical view volume into window space
  - Directly through function calls to set up the window
- Matrix calls multiply some matrix M onto the current matrix C, resulting in CM
  - Set view transformation first, then set transformations from local to world space – last one set is first one applied

OpenGL Camera

- The default OpenGL image plane has u aligned with the x axis, v aligned with y, and n aligned with z
  - Means the default camera looks along the negative z axis
  - Makes it easy to do 2D drawing (no need for any view transformation)
- `glOrtho(…)` sets the view->screen matrix
  - Modifies the PROJECTION matrix
- `gluLookAt(…)` sets the world->view matrix
  - Takes an image center point, a point along the viewing direction and an up vector
  - Multiplies a world->view matrix **onto the current MODELVIEW matrix**
  - You could do this yourself, using `glMultMatrix(…)` with the matrix from the previous slides

Left vs Right Handed View Space

- You can define u as right, v as up, and n as toward the viewer: a right handed system $u \times v = n$
  - Advantage: Standard mathematical way of doing things
- You can also define u as right, v as up and n as into the scene: a left handed system $v \times u = n$
  - Advantage: Bigger n values mean points are further away
- OpenGL is right handed
- Many older systems, notably the Renderman standard developed by Pixar, are left handed