Computer Graphics

Prof. Feng Liu

Fall 2018

http://www.cs.pdx.edu/~fliu/courses/cs447/

10/16/2018

Last time

- Compositing
- NPR
- 3D Graphics Toolkits
  - Transformations

Today

- 3D Transformations
- The Viewing Pipeline
- Mid-term: in class, Nov. 1
- Homework 3 available, due October 30, in class

Homogeneous Coordinates

- Use three numbers to represent a 2D point
- $(x,y)=(wx,wy,w)$ for any constant $w \neq 0$
  - Typically, (x,y) becomes $(x,y,1)$
  - To go backwards, divide by w
- Translation can now be done with matrix multiplication!

Basic Transformations

Translation: $$\begin{bmatrix} 1 & 0 & b_x \\ 0 & 1 & b_y \\ 0 & 0 & 1 \end{bmatrix}$$

Rotation: $$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

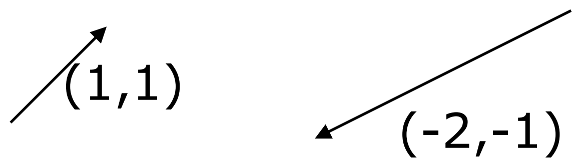
Scaling: $$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Homogeneous Transform Advantages

- Unified view of transformation as matrix multiplication
  - Easier in hardware and software
- To compose transformations, simply multiply matrices
  - Order matters: AB is generally not the same as BA
- Allows for non-affine transformations:
  - Perspective projections!

Directions vs. Points

- We have been talking about transforming points
- Directions are also important in graphics
  - Viewing directions
  - Normal vectors
  - Ray directions



- Directions are represented by vectors, like points, and can be transformed, but not like points

Transforming Directions

- Say I define a direction as the difference of two points: $d=a-b$
  - This represents the *direction* of the line between two points
- Now I translate the points by the same amount: $a'=a+t$, $b'=b+t$
- How should I transform d?

Homogeneous Directions

- Translation does not affect directions!
- Homogeneous coordinates give us a very clean way of handling this
- The direction *(x,y)* becomes the homogeneous direction *(x,y,0)*

$$\begin{bmatrix} 1 & 0 & b_x \\ 0 & 1 & b_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \end{bmatrix}$$

- The correct thing happens for rotation and scaling also
  - Uniform scaling changes the length of the vector, but not the direction

3D Transformations

- Homogeneous coordinates: $(x,y,z)=(wx,wy,wz,w)$
- Transformations are now represented as 4x4 matrices
- Typical graphics packages allow for specification of translation, rotation, scaling and arbitrary matrices
  - OpenGL: glTranslate[fd], glRotate[fd], glScale[fd], glMultMatrix[fd]

3D Translation

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

3D Rotation

- Rotation in 3D is about an *axis* in 3D space passing through the origin
- Using a matrix representation, any matrix with an *orthonormal* top-left 3x3 sub-matrix is a rotation
  - Rows are mutually orthogonal (0 dot product)
  - Determinant is 1
  - Implies columns are also orthogonal, and that the transpose is equal to the inverse

3D Rotation

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} r_{xx} & r_{xy} & r_{xz} & 0 \\ r_{yx} & r_{yy} & r_{yz} & 0 \\ r_{zx} & r_{zy} & r_{zz} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

and if

$$R = \begin{bmatrix} - & \mathbf{r}_1 & - & 0 \\ - & \mathbf{r}_2 & - & 0 \\ - & \mathbf{r}_3 & - & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \text{then } \mathbf{r}_1 \bullet \mathbf{r}_2 = 0, \mathbf{r}_1 \bullet \mathbf{r}_3 = 0, \mathbf{r}_2 \bullet \mathbf{r}_3 = 0, \mathbf{r}_1 \bullet \mathbf{r}_1 = 1, etc.$$

Problems with Rotation Matrices

- Specifying a rotation really only requires 3 numbers
  - Axis is a unit vector, so requires 2 numbers
  - Angle to rotate is third number
- Rotation matrix has a large amount of redundancy
  - Orthonormal constraints reduce degrees of freedom back down to 3
- Rotations are a very complex subject, and a detailed discussion is way beyond the scope of this course

Alternative Representations

- Specify the axis and the angle (OpenGL method)
- *Euler angles:* Specify how much to rotate about X, then how much about Y, then how much about Z
  - Hard to think about, and hard to compose
  - Any three axes will do e.g. X,Y,Z
- Specify the axis, scaled by the angle
  - Only 3 numbers, called the *exponential map*
- *Quaternions*

Quaternions

- 4-vector related to axis and angle, unit magnitude
  - Rotation about axis (n_x, n_y, n_z) by angle θ:

$$\left(n_x \cos(\theta/2), n_y \cos(\theta/2), n_z \cos(\theta/2), \sin(\theta/2)\right)$$

- Reasonably easy to compose
- Reasonably easy to go to/from rotation matrix
- Only normalized quaternions represent rotations, but you can normalize them just like vectors, so it isn't a problem
- Easy to perform spherical interpolation

Other Rotation Issues

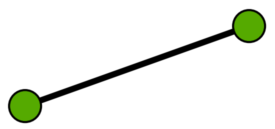
- Rotation is about an axis at the origin
  - For rotation about an arbitrary axis, use the same trick as in 2D: Translate the axis to the origin, rotate, and translate back again
- Rotation is not commutative
  - **Rotation order matters**
  - Experiment to convince yourself of this

Transformation Leftovers

- Scale, shear etc extend naturally from 2D to 3D
- Rotation and Translation are the *rigid-body transformations:*
  - Do not change lengths or angles, so a body does not deform when transformed

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)

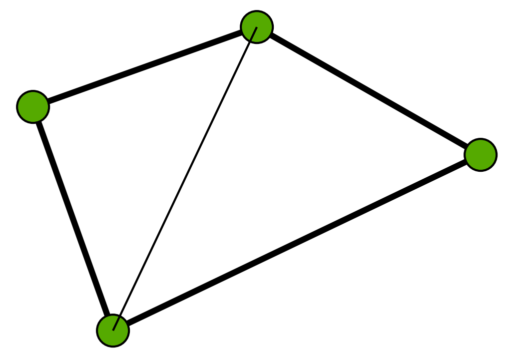


Image url: http://www.capcom-central.com/concept/re4/LeonWireFrame.jpg

Modeling and OpenGL

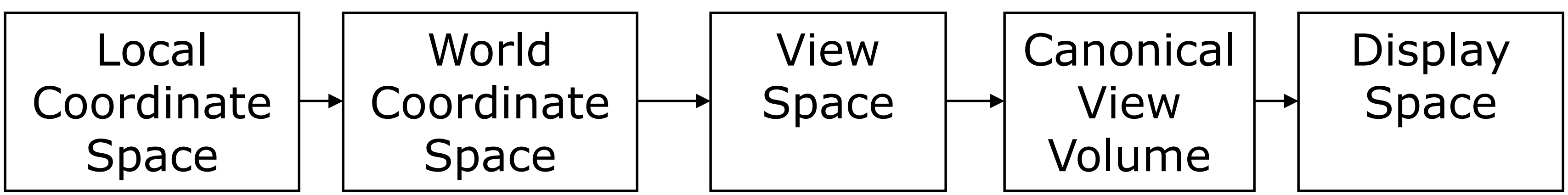
- In OpenGL, all geometry is specified by stating which type of object and then giving the vertices that define it
- `glBegin() …glEnd()`
- `glVertex[34][fdv]`
  - Three or four components (regular or homogeneous)
  - Float, double or vector (eg float[3])
- Chapter 2 of the OpenGL red book

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 pixels should be filled
  - *Rasterization*
- Determine which object is in front at each pixel
  - *Hidden surface elimination, Hidden surface removal, Visibility*
- Determine what color it is
  - *Lighting, Shading*

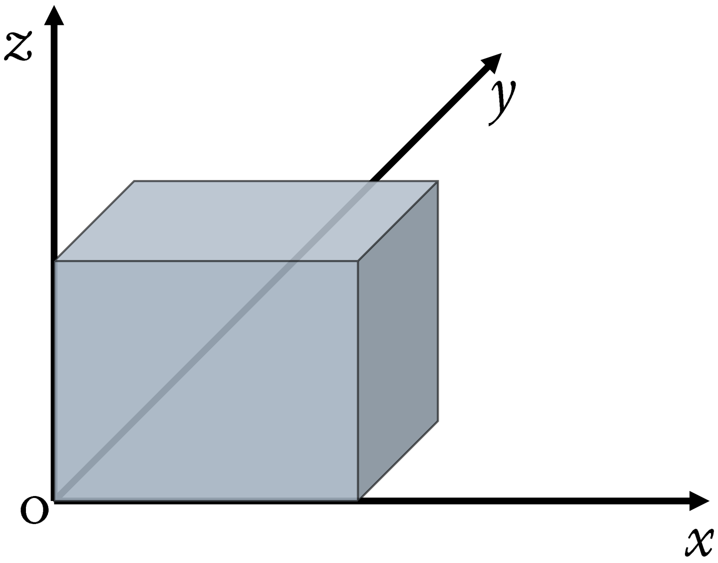
Graphics Pipeline

- 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

- 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 axes



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 axes
- 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

World Coordinate System

- *Everything* in the world is transformed into one coordinate system - the *world coordinate system*
  - It has an origin, and three coordinate directions, *x*, *y*, and *z*
- 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

- Define a coordinate system based on the *eye* and *image plane* – the *camera*
  - The *eye* is the center of projection, like the aperture in a camera
  - The *image plane* is the orientation of the plane on which the image should "appear," like the film plane of a camera
- Some camera parameters are easiest to define in this space
  - Focal length, image size
- Relative depth is captured by a single number in this space

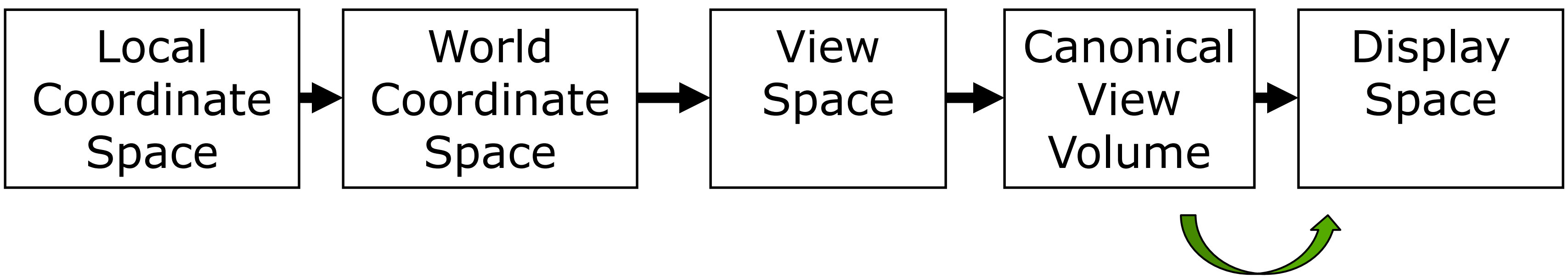
Canonical View Volume

- **Canonical View Space: A cube, with the origin at the center, the viewer looking down –z, x to the right, and y up**
  - Canonical View Volume is the cube: $[-1,1]\times[-1,1]\times[-1,1]$
  - Variants (later) with viewer looking down +z and z from 0-1
  - Only things that end up inside the canonical volume can appear in the window
- Tasks: Parallel sides and unit dimensions make many operations easier
  - Clipping – decide what is in the window
  - Rasterization - decide which pixels are covered
  - Hidden surface removal - decide what is in front
  - Shading - decide what color things are

Window Space

- **Window Space: Origin in one corner of the "window" on the screen, x and y match screen x and y**
- Windows appear somewhere on the screen
  - Typically you want the thing you are drawing to appear in your window
  - But you may have no control over where the window appears
- You want to be able to work in a standard coordinate system – **your code should not depend on *where* the window is**
- You target Window Space, and the windowing system takes care of putting it on the screen

Graphics Pipeline



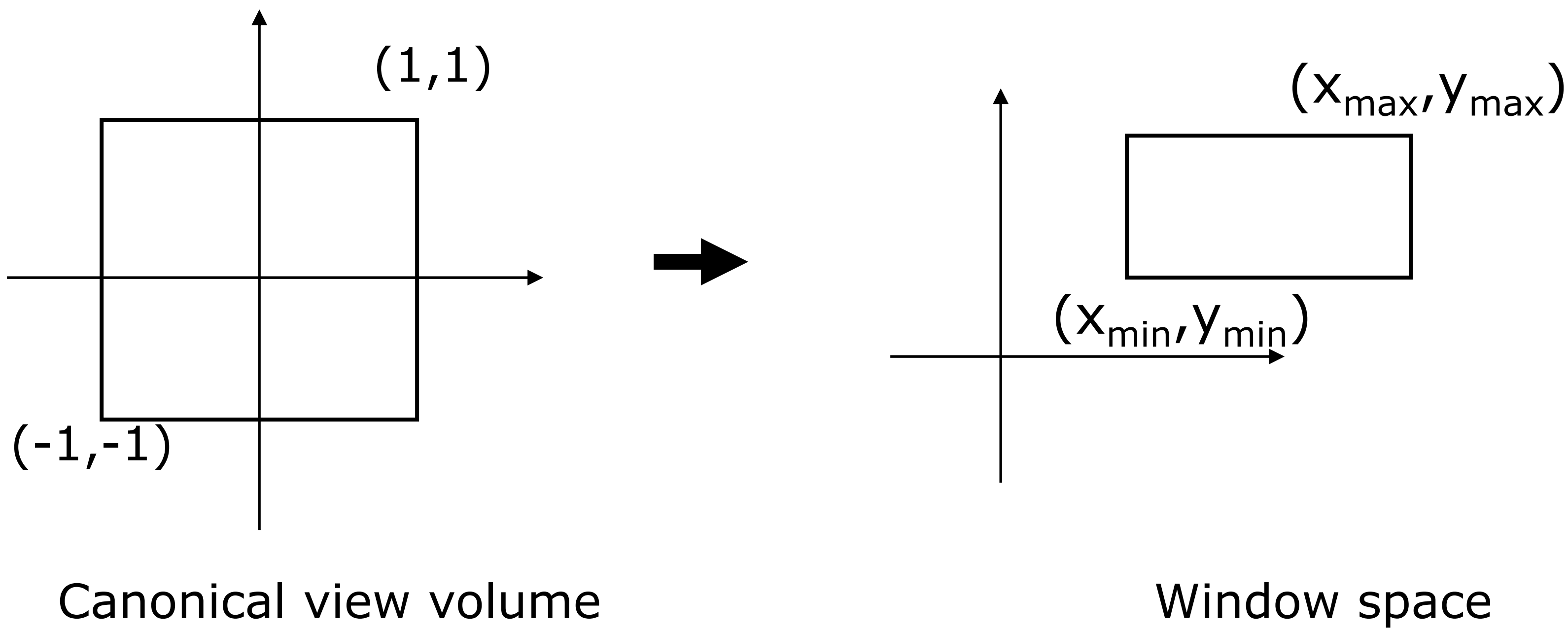
Canonical → Window Transform

- Problem: Transform the Canonical View Volume into Window Space (real screen coordinates)
  - Drop the depth coordinate and translate
  - The graphics hardware and windowing system typically take care of this – but we'll do the math to get you warmed up
- The windowing system adds one final transformation to get your window on the screen in the right place

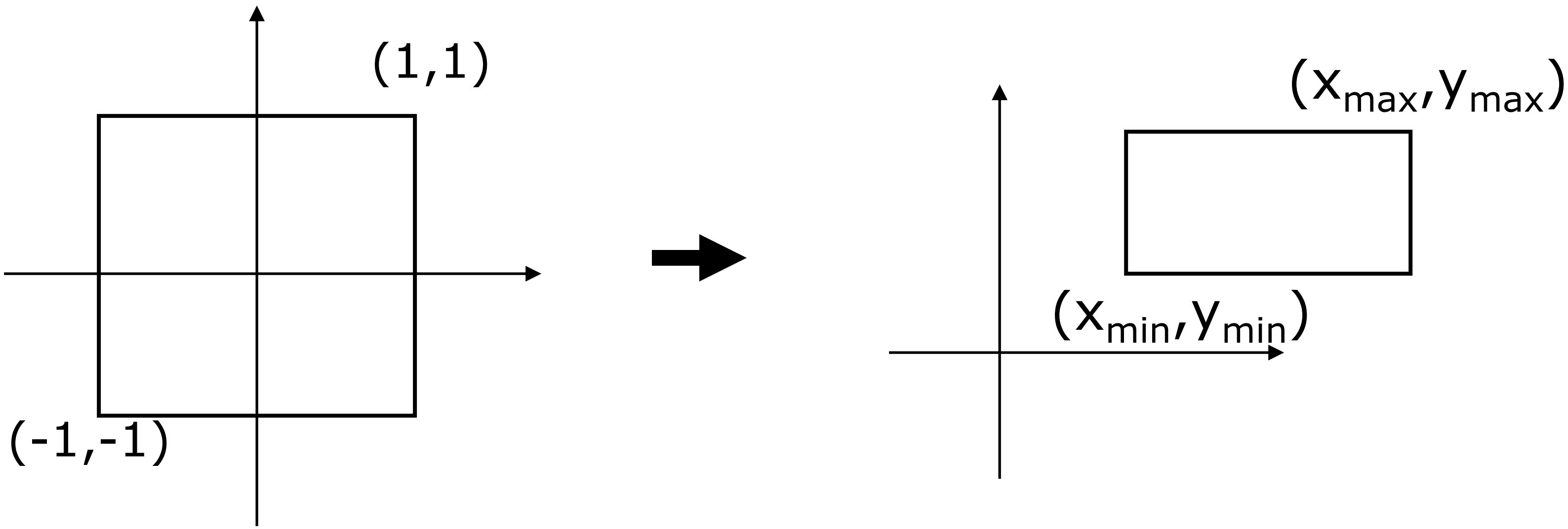
Canonical $\rightarrow$ Window Transform

- Typically, windows are specified by a corner, width and height
  - Corner expressed in terms of screen location
  - This representation can be converted to (x_{min},y_{min}) and (x_{max},y_{max})
- We want to map points in Canonical View Space into the window
  - Canonical View 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?

Canonical → Window Transform



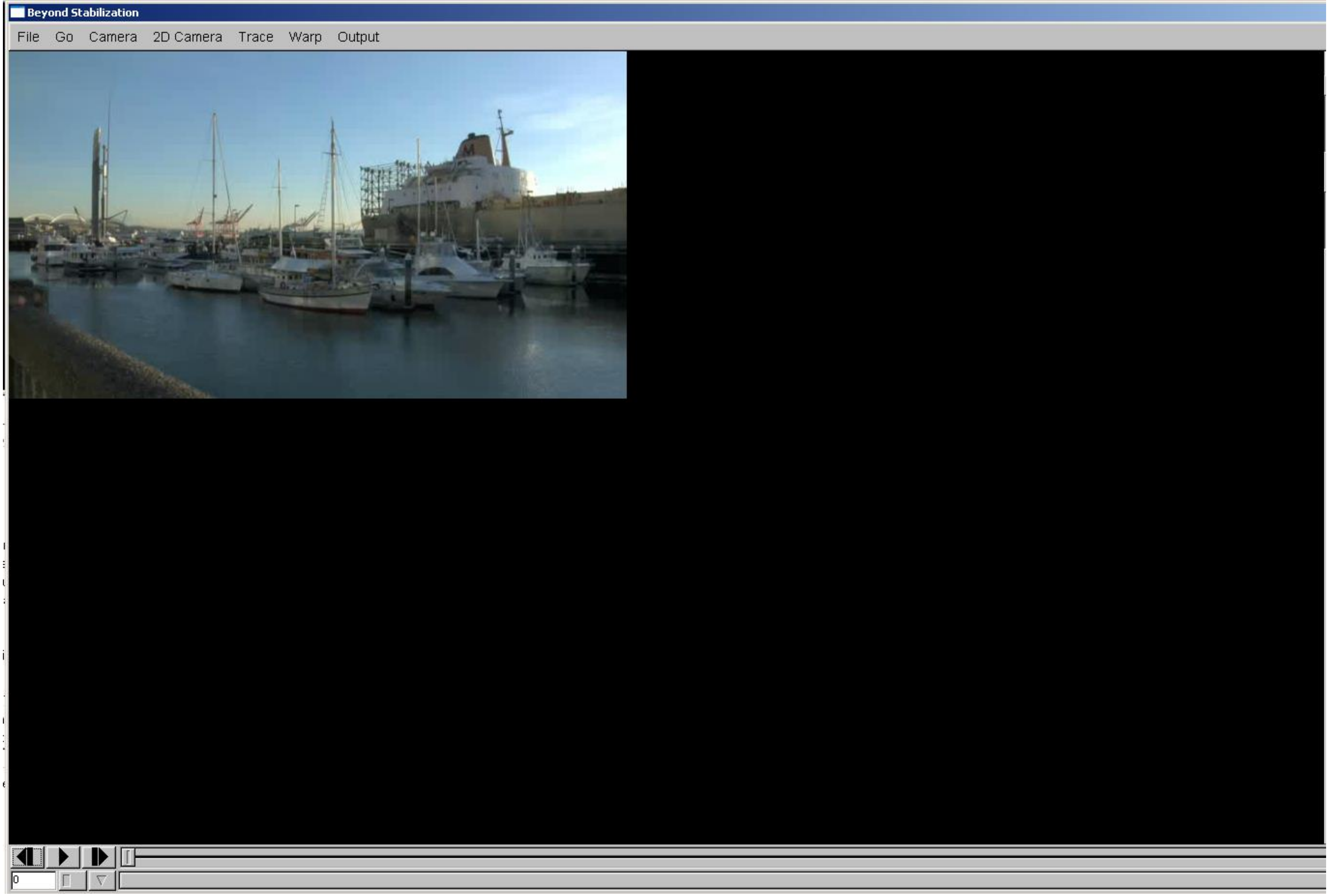
Canonical → Window Transform



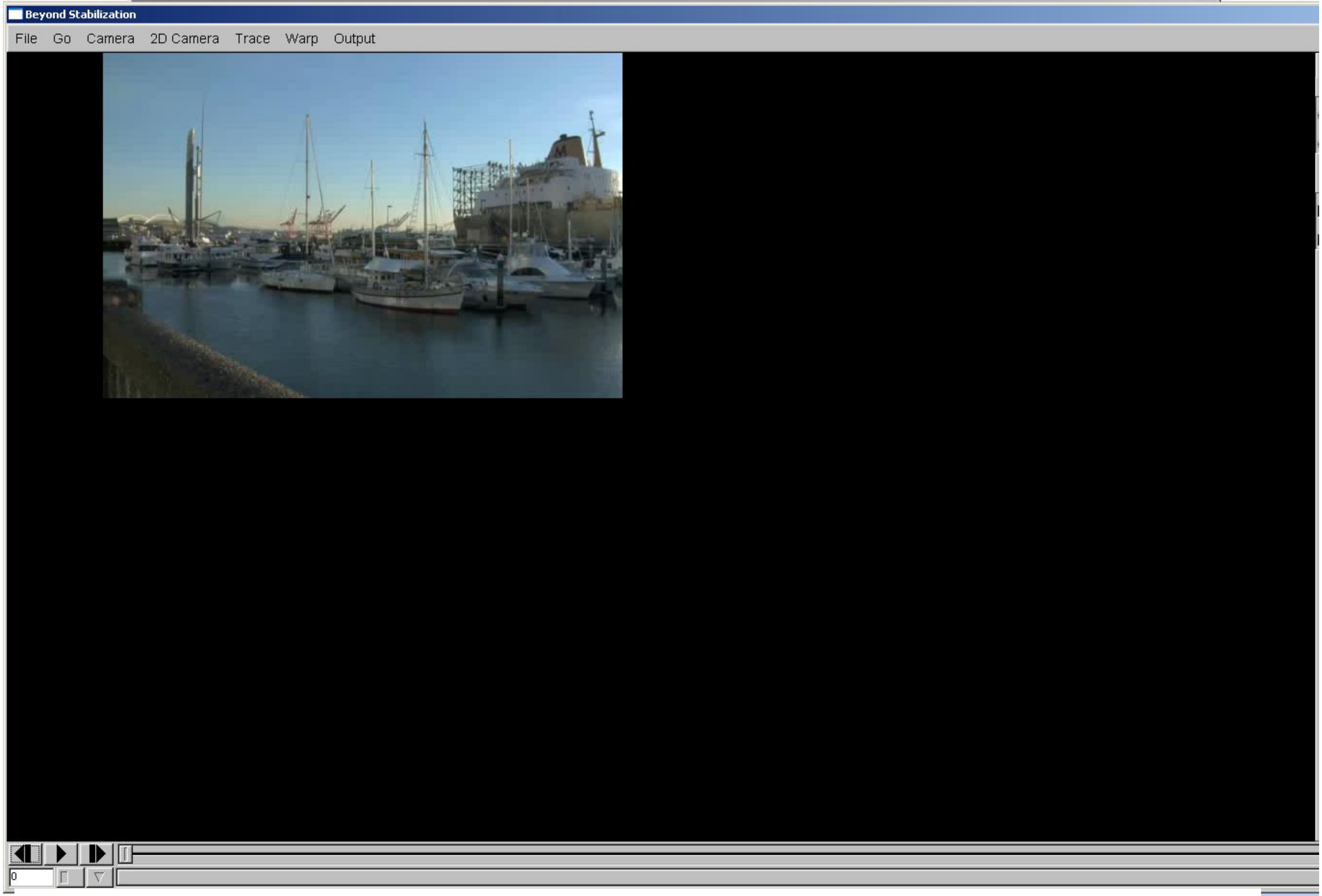
$$\begin{bmatrix} x_{pixel} \\ y_{pixel} \\ z_{pixel} \\ 1 \end{bmatrix} = \begin{bmatrix} (x_{\max} - x_{\min})/2 & 0 & 0 & (x_{\max} + x_{\min})/2 \\ 0 & (y_{\max} - y_{\min})/2 & 0 & (y_{\max} + y_{\min})/2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{canonical} \\ y_{canonical}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

Canonical → Window Transform

- You almost never have to worry about the canonical to window transform
- In OpenGL, you tell it which part of **your window** to draw in – relative to the window's coordinates
  - That is, you tell it where to put the canonical view volume
  - You must do this whenever the window changes size
  - Window (not the screen) has **origin at bottom left**
  - `glViewport(minx, miny, maxx, maxy)`
  - Typically: `glViewport(0, 0, width, height)` fills the entire *window* with the image
- Some textbook derives a different transform, but the same idea

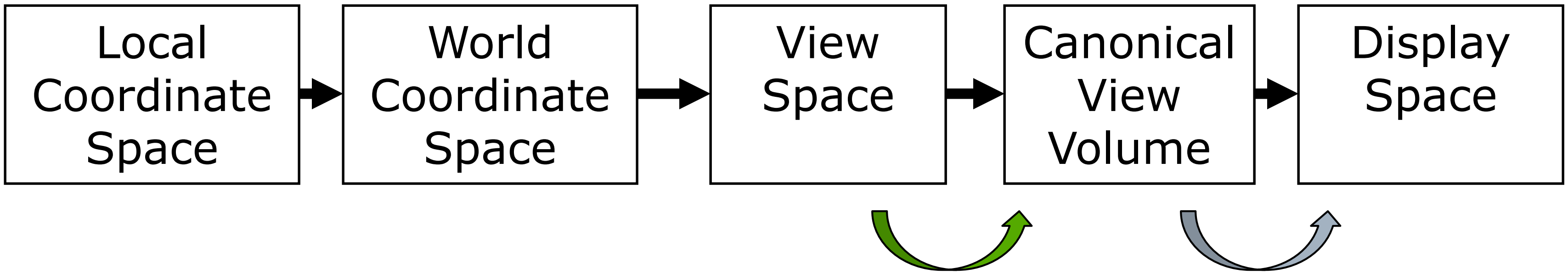



```
glViewport(0, 0, width, height)
```




```
glViewport(100, 0, width, height)
```

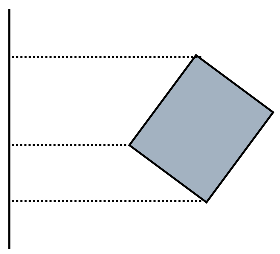
Graphics Pipeline

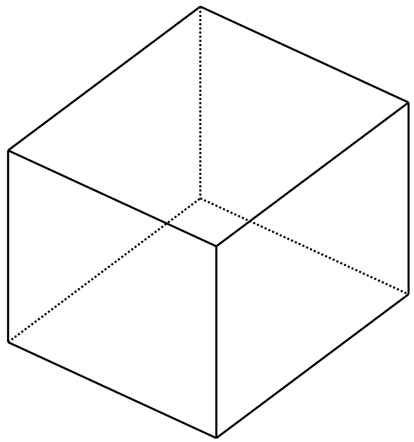


View Volumes

- Only stuff inside the Canonical View Volume gets drawn
  - Points too close or too far away will not be drawn
  - But, it is inconvenient to model the world as a unit box
- A **view volume** is the region of space we wish to *transform into* the Canonical View Volume for drawing
  - Only stuff inside the view volume gets drawn
  - **Describing the view volume is a major part of defining the view**

Orthographic Projection

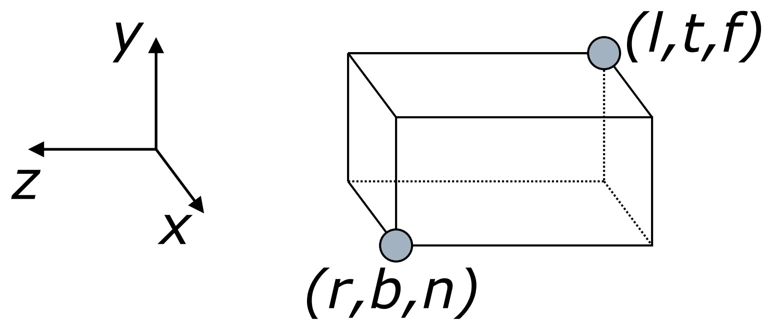




- Orthographic projection projects all the points in the world along parallel lines onto the image plane
  - Projection lines are perpendicular to the image plane
  - Like a camera with infinite focal length
- The result is that *parallel lines in the world project to parallel lines in the image*, and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

Orthographic View Space

- **View Space: a coordinate system with the viewer looking in the -*z* direction, with *x* horizontal to the right and *y* up**
  - A right-handed coordinate system! All ours will be
- The view volume is a *rectilinear box* for orthographic projection
  - 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 find out where points end up on the screen, we must transform View Space into Canonical View Space
  - We know how to draw Canonical View Space on the screen
- This transformation is "projection"
- The mapping looks similar to the one for Canonical to Window …

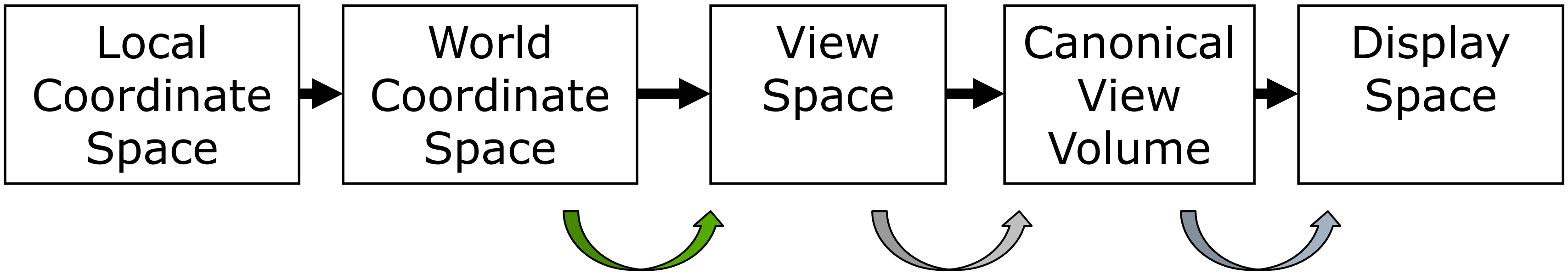
Orthographic Projection Matrix (Orthographic View to Canonical Matrix)

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

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

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

Graphics Pipeline



Next Time

- Perspective Projection
- Clipping
-