Introduction to 3D Graphics Camera, Viewing and Movement

January 23rd and 26th 2009

MAE 574, Virtual Reality Applications and Research

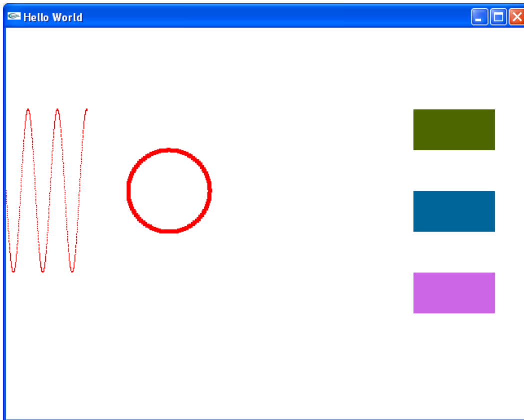
Instructor: Govindarajan Srimathveeravalli

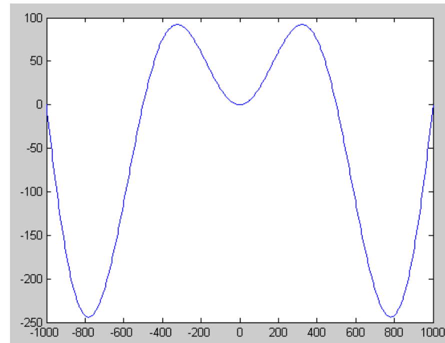
HW 2: There is no escaping, homework, death or taxes. Due: 1/30

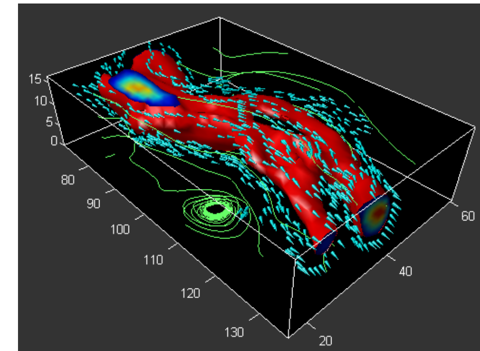
1. Implement the following for polygons:
   - A structure that will allow you to store all basic information associated with a polygon (vertices, connectivity or edge information and normal)
   - Write a function that will render polygons correctly using this information
   - Make special provision in this function to accommodate triangles.
2. Implement algorithms that will
   - Given polygon vertices and connectivity information (based on vertex index values)
     - Will determine if the polygon is convex
     - Calculate its normal
       - Normalize!
     - Determine if a given point lies within a polygon or not
       - Extra credit for implementing the ray casting method
     - Find the convex hull of a concave polygon

- Follow same submission procedure as in last homework. Part 1 of homework has to be done in C++, using OpenGL. Part 2 can be written in either C++ or Matlab

Screen vs World







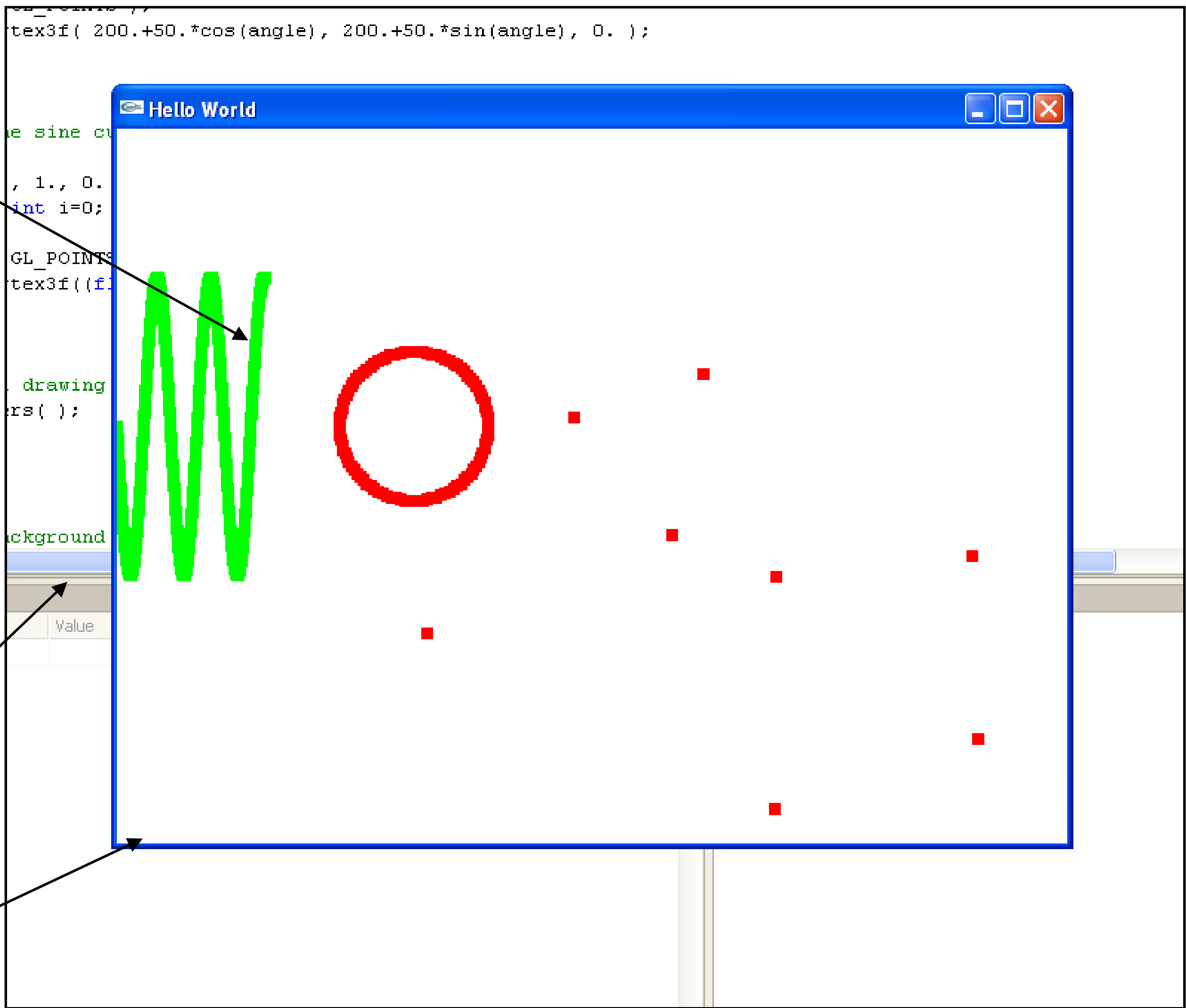
Screen Coordinates

1. Measurement in pixels
2. Origin located in the top left hand corner or bottom left
3. X positive to the right
4. Y positive to the bottom or from top
5. Technically, no negative values
6. Length varies from zero to some screen width (again defined in pixels)
7. All picking operations (i.e. interactions with mouse) have values defined in screen coordinates.

1. World is where all "the objects" to be drawn are located
2. Arbitrary sizes and lengths
   - On the same screen need to represent things in nano-level to world sized one

- Arbitrary scaling, f.e. for graph x ranges in 1000's, y ranges in 100's
- Part of the world to be captured from the window we look into it
- Mapping is required

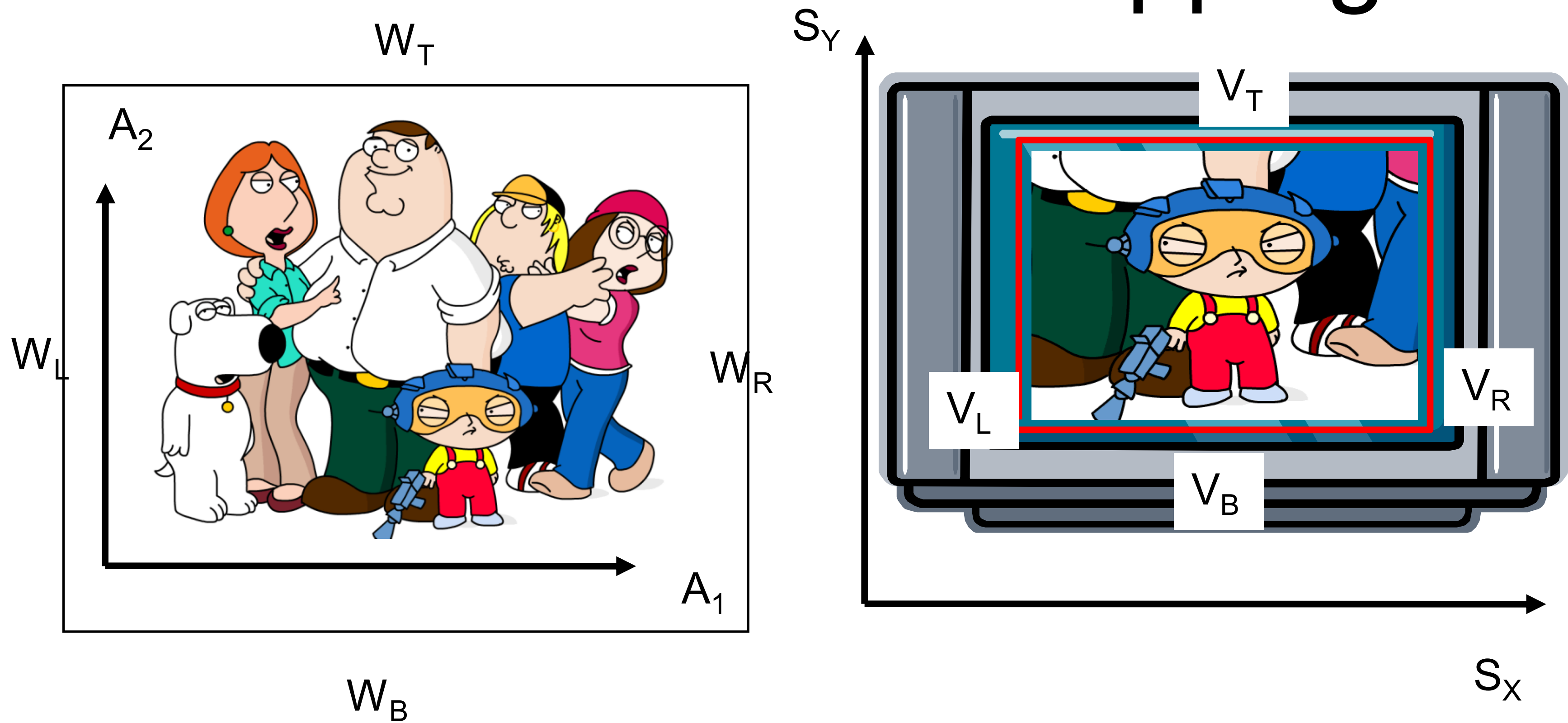
Viewport



Screen

Window

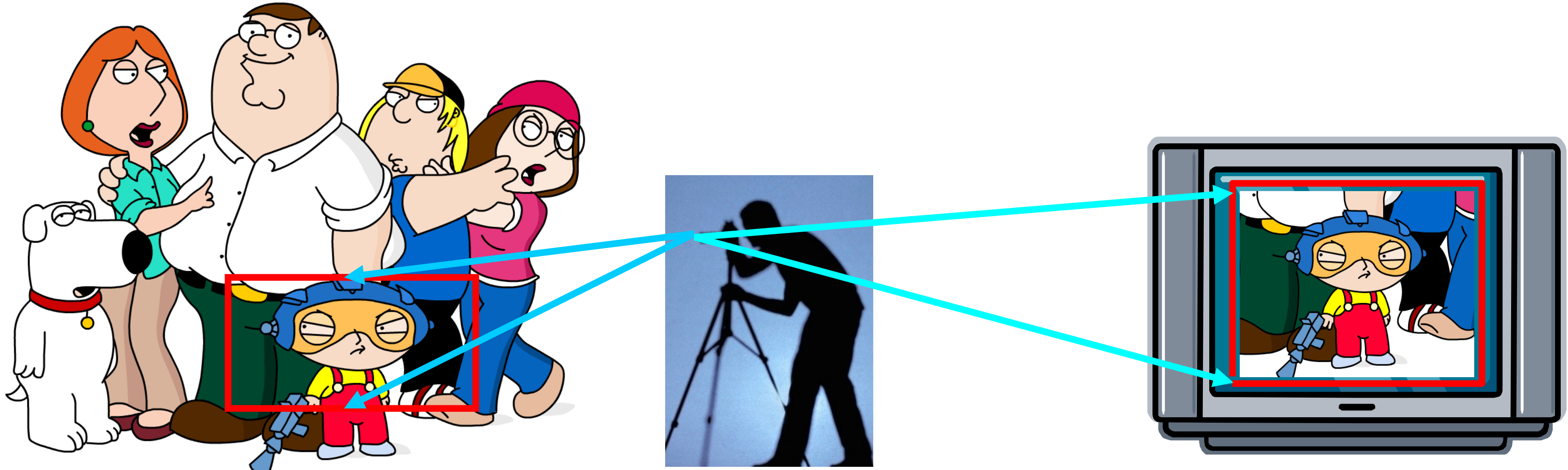
World to Screen Mapping



- Each window is defined by four values (left (W_L and $V_{L.}$), right (W_R and V_R.), bottom (W_B and V_B.) and top (W_T and V_T.),)
- By choice of bottom and top we can arbitrarily allocate the "origin" for each case
- Since there is a mapping from world to screen, scaling and adjustment for size needs to be done

The Scaling

Look through a “*window*” to the world and map it onto a portion of the screen (*viewport*)



$$S_X = A * X + C$$

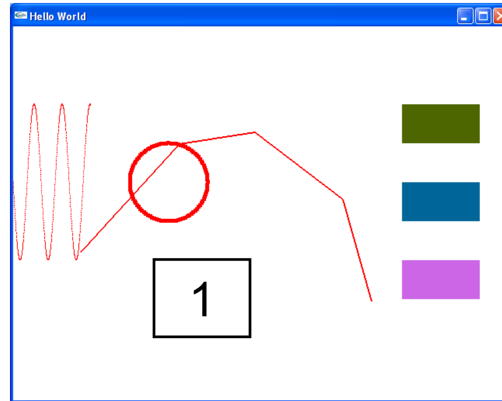
$$A = \frac{(V_R - V_L)}{(W_R - V_L)} * X$$

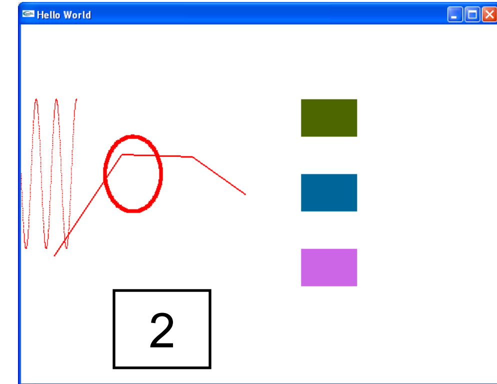
$$C = (V_L - A * V_L)$$

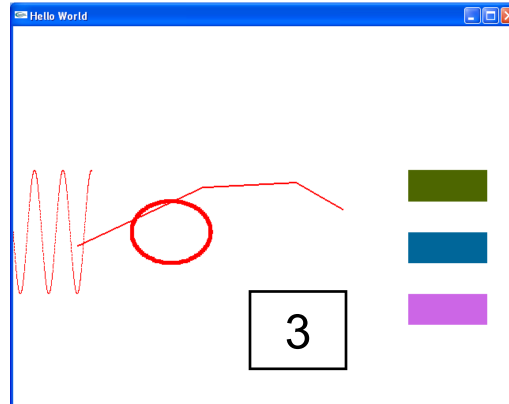
- Formula for scaling and mapping in 2D
- Needed for “*correct*” graphing or mapping in 2D without distortion
- Looking at gluOrtho2D(left, right, bottom, top)
    - Determines the actual size and scaling of things in the world
- Also, glViewport(lower left x, lower right y, width, height)
    - –Determines the amount of screen space allotted to the given scene from world space
- How does viewport and window settings affect mouse picking ?

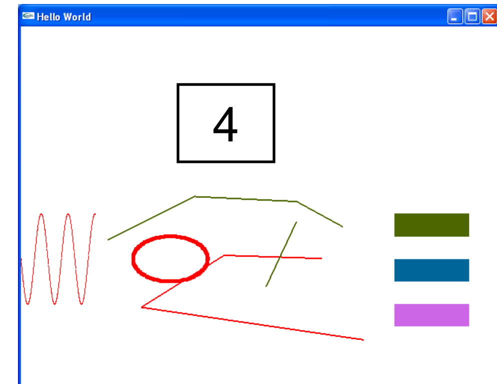
Scaling Things

Standard	Our Value
4:3	640x480 (default)
1:1	480x480
16:9	640x376
221:100	640x290



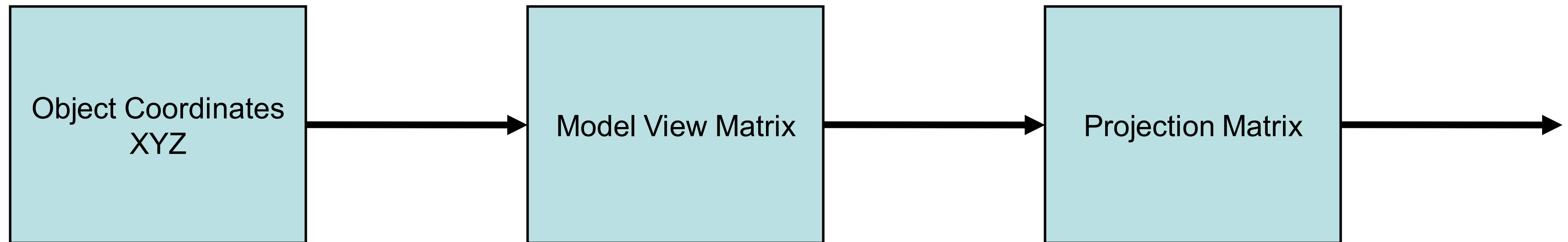






- Apparently it is ideal to have the viewport to have the closest ratio to the window and the world
- What needs to be done for a 1:1 correspondence between world and screen ?
- Note:!! In 3D aspect ratio of viewing volume and viewport must be “close” to each other

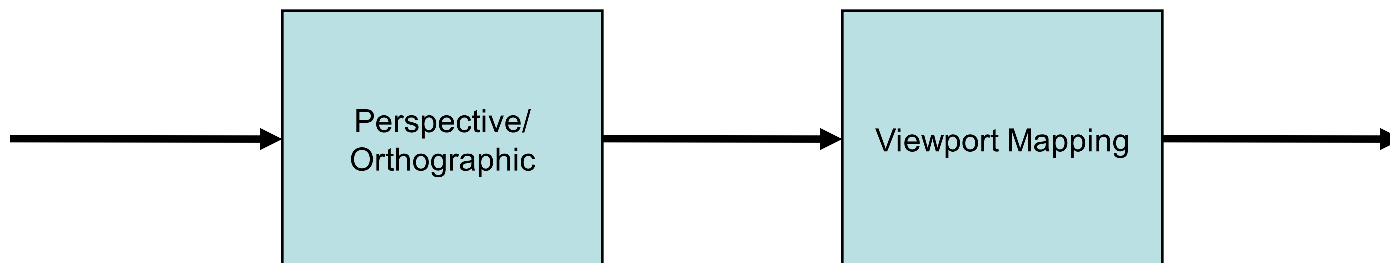
Before 3D



Vertices

Eye Coords

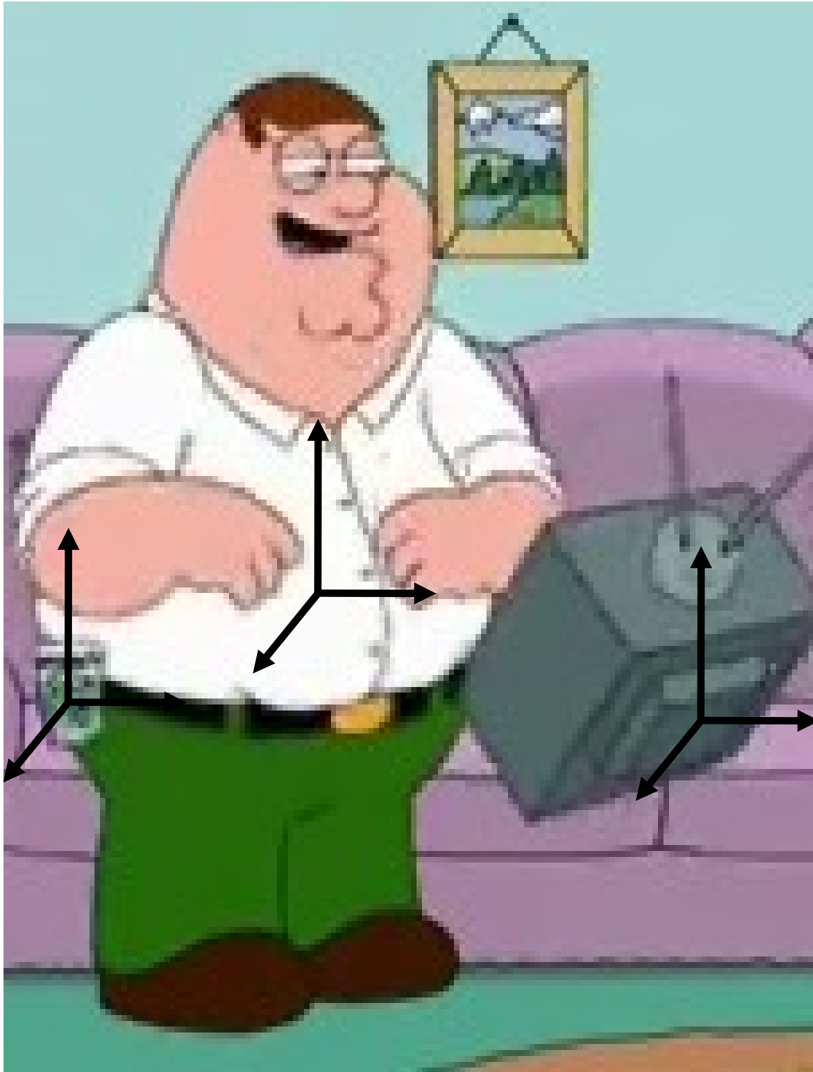
Clip Coords



Normalized
Device Coords

Window Coords

Object and the World, Rules



- All objects in our world are defined by their vertices
- All objects exist in a "world" (Peter's living room)
- The world has one single fixed reference coordinate frame that does not move (Peter)
- All objects in the world have their own "*local*" coordinate frames
- The movement of the objects can be either defined in terms of the world coordinates (revolutions) and/or in terms of their own coordinates (rotation about their selves)

Generic Matrix Operations

$$p \leftarrow \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Any point can be represented as a vector shown here. The last value in the vector is added to represent homogenous coordinates and "*need not*" be unity in value.

$$p' \leftarrow M \cdot p$$

The following operation "*pre*" multiplies a given point by a matrix *M,* causing some transformation to occur.

$$p'' \leftarrow N \cdot p'$$

If we were to define one more operation on the transformed point as given, here then both can be combined as shown below.

$$p'' \leftarrow N \cdot M \cdot p$$

Notice how multiplication goes from right to left!

$$M \cdot N \neq N \cdot M$$

Matrix multiplication is not commutative

Images courtesy, Howard and Murta, University of Manchester

More definitions

- Affine Transformations : Edges retain their geometric properties relative to each other (ex. Parallel)
  - This is one of the primary properties of homogenous transformations
- Homogenous coordinates: Representation of all transformation operations as multiplications
  - F.e. a vector is represented by 4 values $[\ x\ y\ z\ w\ ]^T$
  - Developed to “support” affine transformations
  - This representation allows us to denote all transformations on the vector as matrix multiplications.
  - In a way this system also lends itself to projections such as perspective projection
  - They allow all affine transformation to be represented by multiplication (operation by a common operator)

Transformation

Transformations are defined by the following operations

1. Translation - glTranslatef(x, y, z)
2. Rotation - glRotatef(angle[degs], x, y, z) {remember axis is centered around origin)
3. Scaling - glScale(x, y, z)
4. Generic glMultMatrixf
5. Prior to any transformation we need to set the matrix mode
   - glMatrixMode() , can be either GL_PROJECTION or GL_MODELVIEW (GL_TEXTURE)
6. Also, a fresh matrix needs to be used for all operations. A new identity matrix can be loaded using
   - glLoadIdentity()
7. OpenGL works by post multiplication, so transformation have to be reverse ordered
   1. Same applies for viewing commands (in the pipeline, viewing commands come after tranform, so in code they should come before!)
8. If using MultMatrix
   1. Use glLoadMatrix and glMultMatrix
9. Pls Note! OGL uses column-row format and not row-column format

Sundry Transformation Matrices

$$\begin{bmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Translation

$$\begin{bmatrix} sx & 0 & 0 & 1 \\ 0 & sy & 0 & 1 \\ 0 & 0 & sz & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

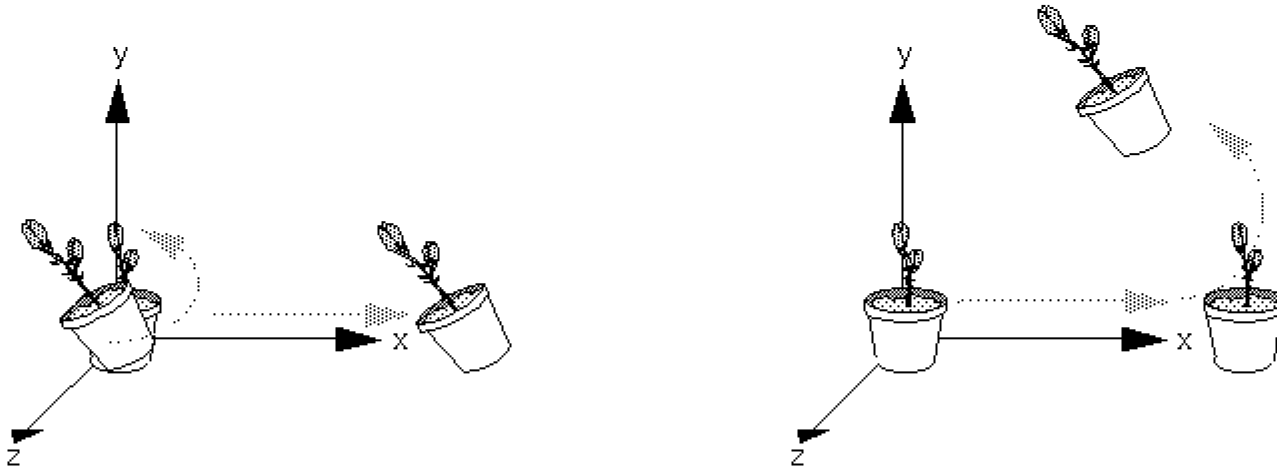
Scaling

Rotation

$$\mathcal{R}_x(\theta_x) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x \\ 0 & \sin\theta_x & \cos\theta_x \end{bmatrix} \quad \mathcal{R}_y(\theta_y) = \begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y \\ 0 & 1 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y \end{bmatrix} \quad \mathcal{R}_z(\theta_z) = \begin{bmatrix} \cos\theta_z & -\sin\theta_z & 0 \\ \sin\theta_z & \cos\theta_z & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Images courtesy wikipedia

Translation and Rotations



- As mentioned earlier, matrix multiplication is not commutative
- Rotations counter clockwise by default
- Also, in OpenGL the transformation effect is in the reverse order of code. That is the last transformation in code has the first effect on the object!! (Where have we seen this before?)

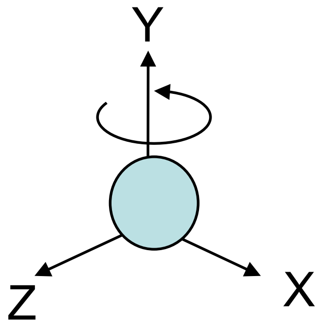
Two ways to do the same thing

- Consider the revolution and rotation of planets
  - Can be achieved as rotation about Y – translation to desired radius and rotation abou Y
  - Or can be written as rotation about Y and translation along locus of circle

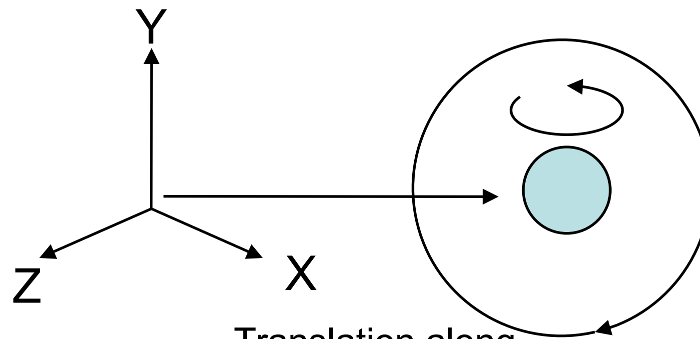
```
// Jupiter 5.2 AU
drawCircle( 52. );
glPushMatrix();
    //glTranslatef( 52.*cos(earthAt/12), 0.,  52.*sin(earthAt/12) );
    //glRotatef( earthAt*3, 0., 1., 0. );
    glRotatef( earthAt/12, 0., 1., 0. );
    glTranslatef( 52., 0., 0. );
    glRotatef( earthAt*3, 0., 1., 0. );
    glutSolidSphere( 12., 15, 15 );
glPopMatrix();
```

- What about the moon ?
  - It involves revolutions (revolution around a moving axis, which in itself has circular locus) and a rotation.
  - Rotation about own axis, followed by revolution about origin with a radius equal to distance from earth and then move it to earth's orbit

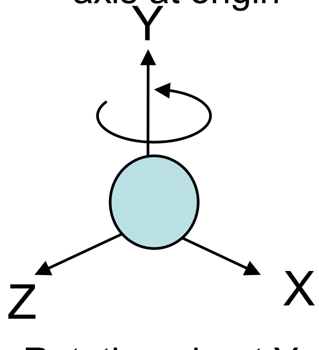
```
// Moon at 3 AU (random, not accurate)
glPushMatrix();
    glTranslatef( 10*cos(earthAt), 0., 10*sin(earthAt));
    glRotatef( earthAt*12, 0., 1., 0. );
    glTranslatef( 3., 0., 0. );
    glRotatef( earthAt, 0., 1., 0. );
    glutSolidSphere( 0.25, 15, 15 );
glPopMatrix();
```



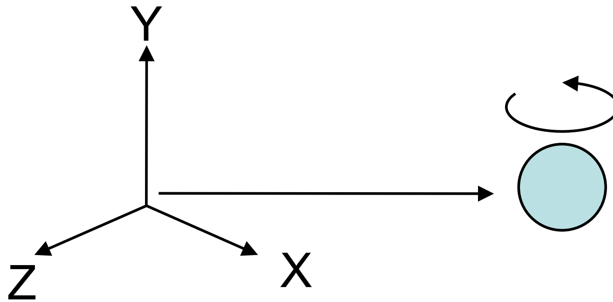
Rotation about Y axis at origin



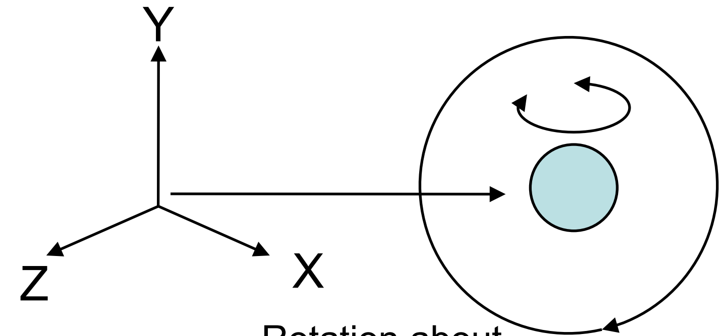
Translation along the locus of a circle



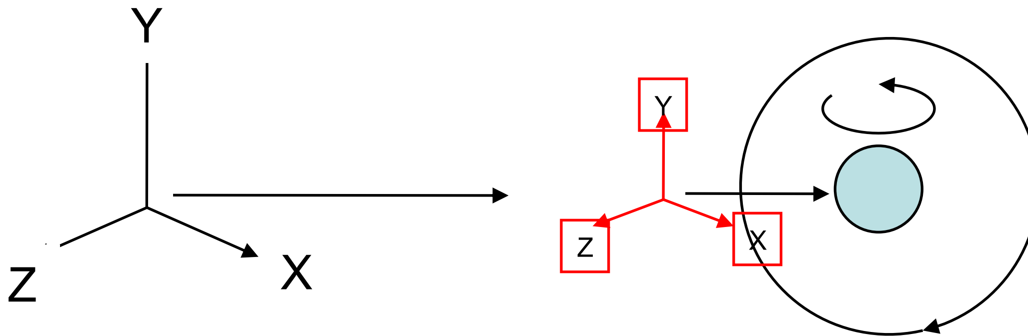
Rotation about Y axis at origin



Translation to the desired point

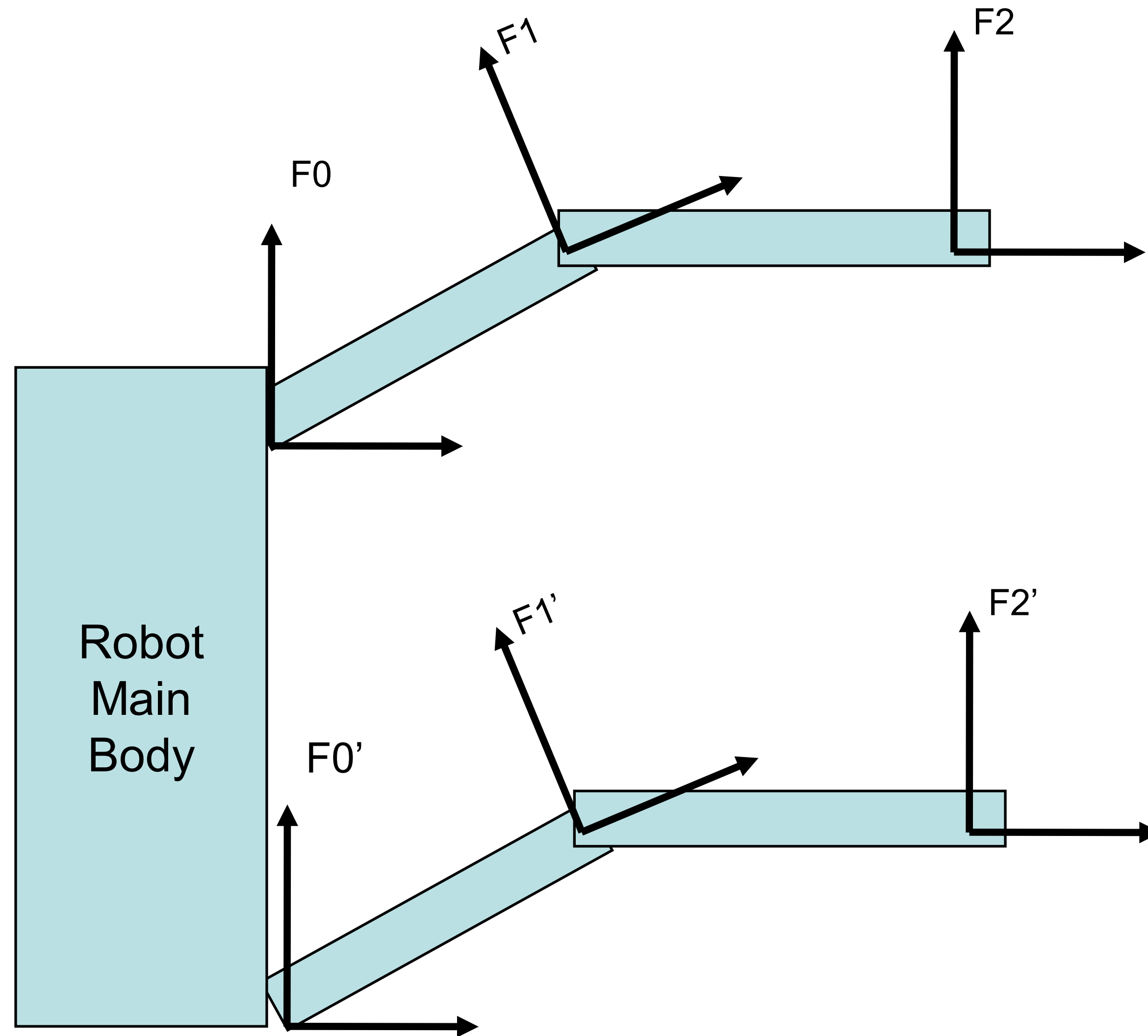


Rotation about the global axis



Final translation effectively changes the axis about which rotation takes place

Hierarchical Transformations and Object Reuse

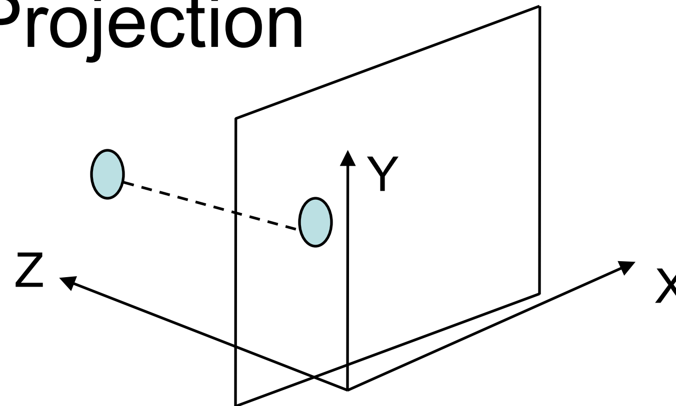



```
drawArm()
{ Arm drawing code }

Display() {

Push
   Necessary
Transformations
   drawArm() Right
   Pop

Push
   Necessary
Transformations
   drawArm() Left
   Pop
}
```

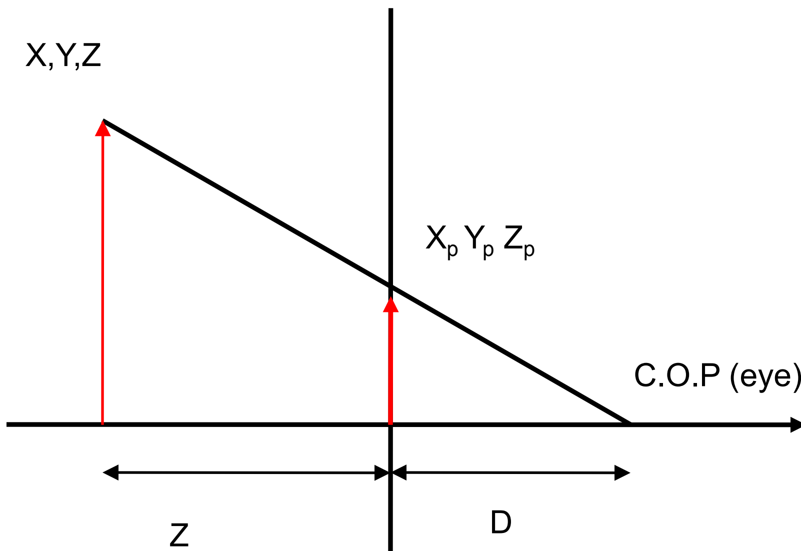
```
drawArm(){

Rotation( Shoulder angle )
Draw Shoulder

Translation( fixed value )
Rotation( arm angle )
Draw Arm

Translation( fixed value )
Rotation( wrist angle )
Draw Wrist }
```

Projection

- Two types
  - Parallel
    - Orthographic
      - Direction of projection perpendicular to projection plane
    - Oblique
      - Otherwise
    - Isometric
  - Perspective



Example of Parallel projection, the blue point is projected onto a plane on the Z axis by lines parallel to itself. The projectors (line tracing the projection from point to projection plane) are perpendicular to the projection plane.



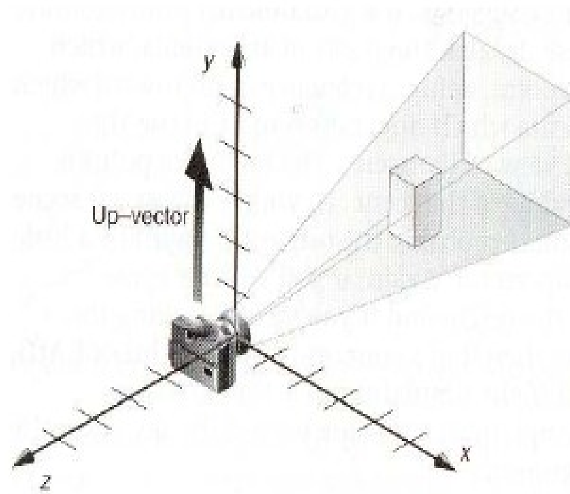
For perspective projection

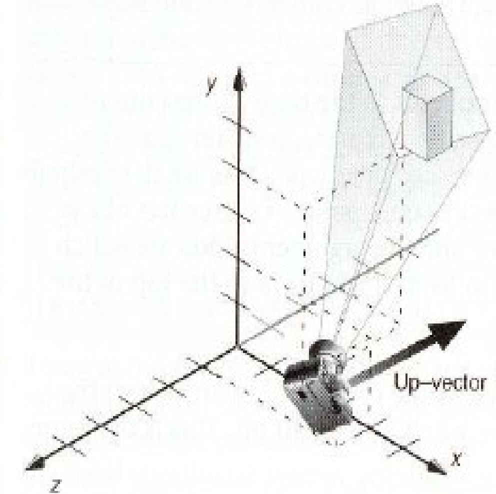
$$\frac{x}{z+d} = \frac{xp}{d} \qquad xp = \frac{x}{(\frac{z}{d}+1)}$$

Same follows for rest of the axis, and can be represented by the matrix notation. Parallel lines not parallel to projection plane will tend to infinity, giving rise to vanishing points

Jumping to 3D – Viewing Your World

- Need to position and aim your camera to get the "*correct*" view of the world
- Three parameters needed for this
  - Camera/Eye position
  - Where is it looking at
  - What is the up direction
- Easy way to do this,
  - gluLookAt()





Perspective and Orthographic Projections

- gluLookAt(camera posn, looking at, up direction)
- glFrustum(left, right, bottom, top, near, far)
- gluPerpective(field of view, aspect, near, far)
- glOrtho(left, right, bottom, top, near, far)
- Matrix operations to come later