CPSC/ECE 4780/6780

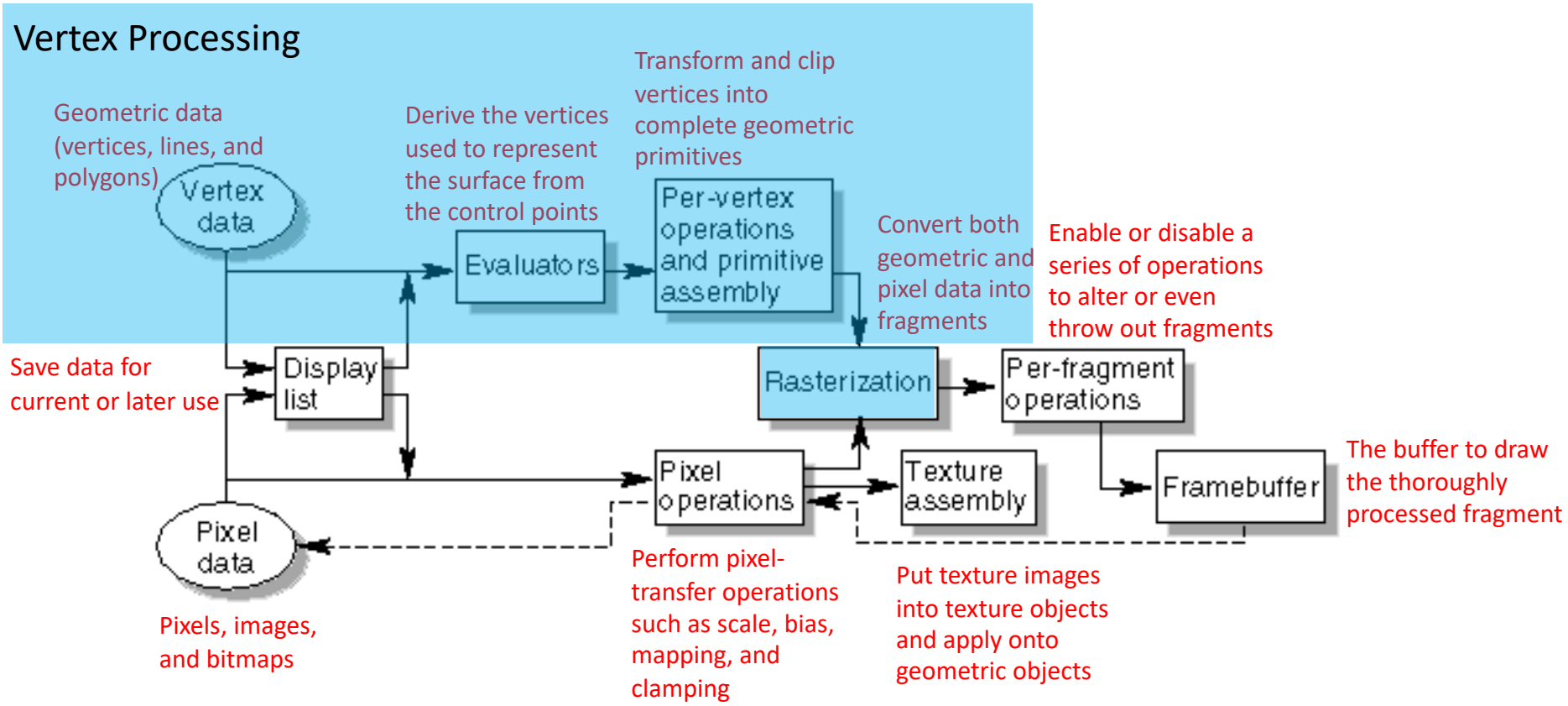
General-Purpose Computation on Graphical Processing Units (GPGPU)

Lecture 17: (OpenGL) Transformations and Projections

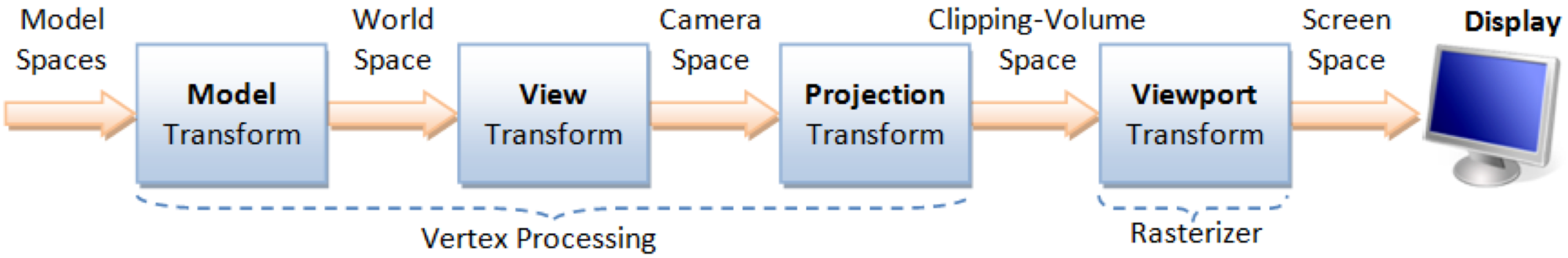
Recaps of Last Lecture

- What is OpenGL?
- What are the features of OpenGL?
- What are the components of the OpenGL rendering pipeline?
- OpenGL-Related libraries
- Geometric primitives
- Display of geometric primitives

Revisit OpenGL Rendering Pipeline



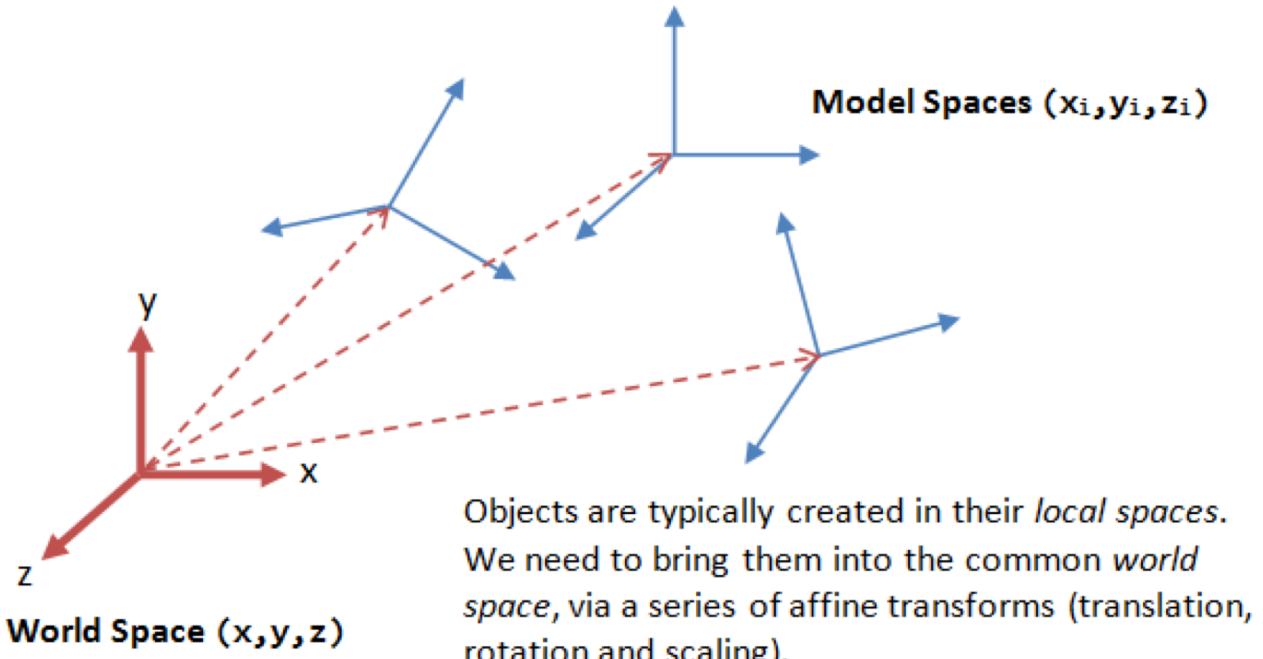
Coordinates Transform Pipeline



- A transform converts a vertex *V* from one space to another space *V'*
- In computer graphics, transform is carried by multiplying the vector with a *transformation matrix*, i.e., ***V' = M V***.

Model Transform

- Each object in a 3D scene is typically drawn in its own coordinate system (model space)
- World transform
  - Transforms the vertices from their local spaces to the world space
  - Consists of scaling, rotation, and translation



Model Transform – Scaling

- A vertex $\boldsymbol{V}$ at (x, y, z) is represented as a 3x1 column vector : $\boldsymbol{V} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$
- 3D scaling can be represented in 3x3 matrix: $\boldsymbol{S}(\alpha_x, \alpha_y, \alpha_z) = \begin{bmatrix} \alpha_x & 0 & 0 \\ 0 & \alpha_y & 0 \\ 0 & 0 & \alpha_z \end{bmatrix}$

  where α_x, α_y and α_z represent the scaling factors in x, y and z direction
- We can obtain the transformed result $\boldsymbol{V'}$ of vertex $\boldsymbol{V}$ via matrix multiplication:

$$\boldsymbol{V'} = \boldsymbol{S}\,\boldsymbol{V} = \begin{bmatrix} \alpha_x & 0 & 0 \\ 0 & \alpha_y & 0 \\ 0 & 0 & \alpha_z \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \alpha_x x \\ \alpha_y y \\ \alpha_z z \end{bmatrix}$$

- OpenGL function: **glScale,** for example glScalef(0.5, 0.5, 0.5) would cause all objects drawn subsequently to be half as big

Model Transform – Rotation

- 2D rotation operates about a center of rotation
- 3D rotation operates about an axis direction (*x*, *y*, *z)* of rotation through an angle θ (counter-clockwise):

$$\boldsymbol{R_z}(\theta) = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}, \boldsymbol{R_x}(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\theta & -sin\theta \\ 0 & sin\theta & cos\theta \end{bmatrix}, \boldsymbol{R_y}(\theta) = \begin{bmatrix} cos\theta & 0 & sin\theta \\ 0 & 1 & 0 \\ -sin\theta & 0 & cos\theta \end{bmatrix}$$

- OpenGL function: **glRotate**, for example glRotatef(30.0, 0.0, 1.0, 0.0) rotates by 30° about y-axis
- Note that glRotate wants angles in degrees
  - C math library (sin, cos, etc.) wants angles in radians
  - *degs* = *rads* * 180/π; *rads* = *degs* * π / 180
- Rotations do not commute
  - The order that transformations are done matters

Model Transform – Translation

- Translation is modeled via a vector addition:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix} = \begin{bmatrix} x + d_x \\ y + d_y \\ z + d_z \end{bmatrix}, \text{where } \begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix} \text{ is the tranlational vector}$$

- Using the 4-component homogeneous coordinates, translation can be

$$\boldsymbol{T}(d) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{where } d = \begin{bmatrix} d_x \\ d_y \\ d_z \\ 1 \end{bmatrix} \text{ is the tranlational vector}$$

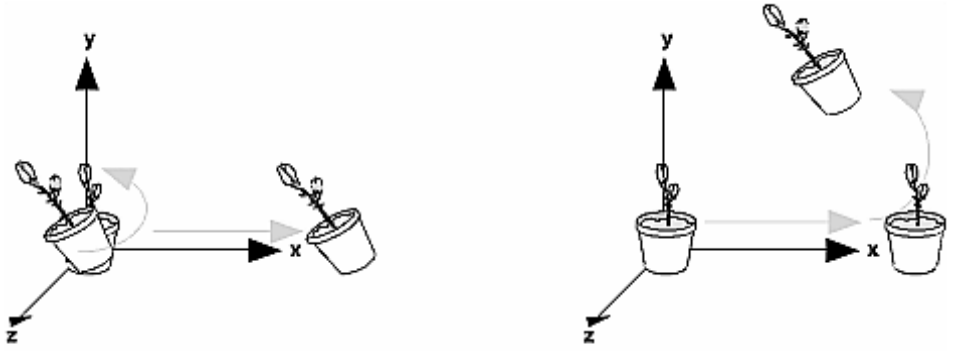
- The transformed vertex ***V'*** can again be computed via matrix multiplication:

$$\boldsymbol{V}' = \boldsymbol{T}(d)\,\boldsymbol{V} = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + d_x \\ y + d_y \\ z + d_z \\ 1 \end{bmatrix}$$

- OpenGL function: glTranslate, for example glTranslatef(0.0, 0.0, -0.5)

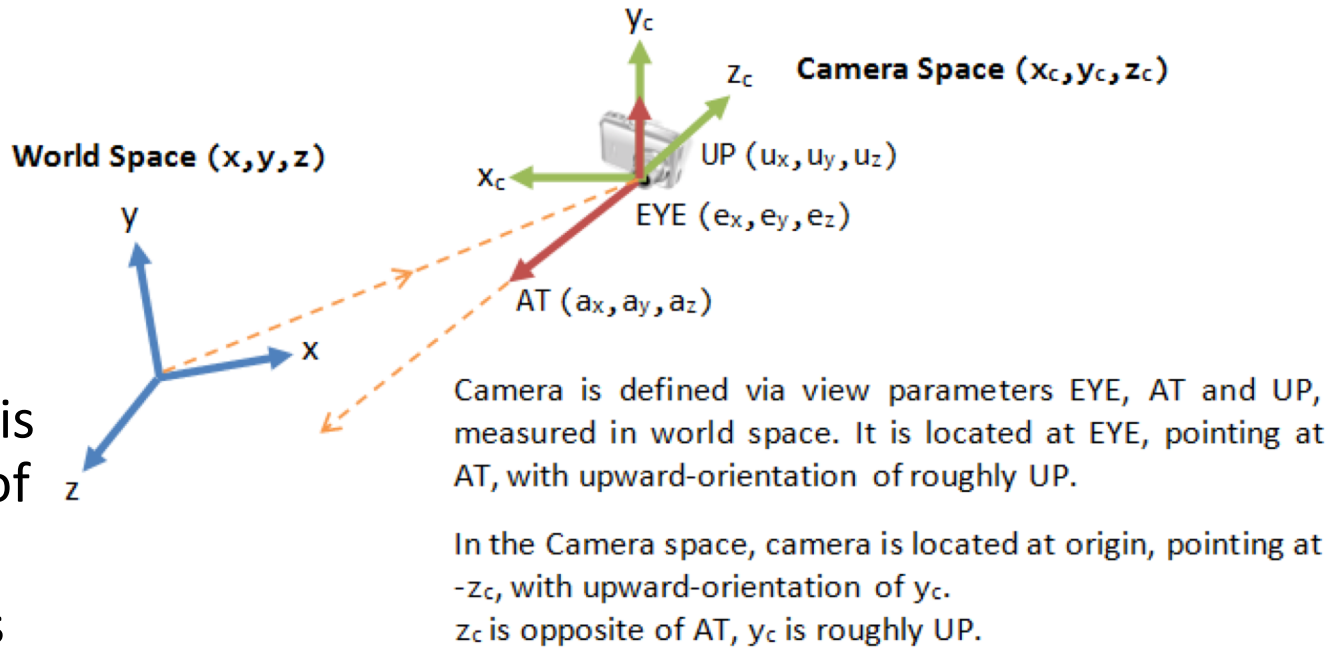
More on Model Transform

- Successive transforms
  - A series of successive affine transforms (T_1, T_2, T_3, ...) operating on a vertex V can be computed via concatenated matrix multiplications $V' = \ldots T_3 T_2 T_1 V$
  - The matrices can be combined before applying to the vertex because matrix multiplication is associative, i.e., $T_3 (T_2 (T_1 V)) = (T_3 T_2 T_1) V$
- Order of transformations
  - Transformations are cumulative and the order matters
  - For each object, the usual sequence is:
    - Translate (move the origin to the right location)
    - Rotate (orient the coordinate axes right)
    - Scale (get the object to the right size)



View Transform

- Position the camera by specifying three view parameters:
  - The point EYE(e_x, e_y, e_z) defines the location of the camera
  - The vector AT(a_x, a_y, a_z) denotes the direction where the camera is aiming at, usually at the center of the world or an object
  - The vector UP(u_x, u_y, u_z) denotes the upward orientation of the camera roughly



View Transform – OpenGL Functions

- In OpenGL, we can use the GLU function gluLookAt() to position the camera:

```
void gluLookAt(GLdouble xEye, GLdouble yEye, GLdouble zEye,
               GLdouble xAt, GLdouble yAt, GLdouble zAt,
               GLdouble xUp, GLdouble yUp, GLdouble zUp)
```

- The default settings of gluLookAt() is:

  gluLookAt(0.0, 0.0, 0.0, 0.0, 0.0, -100.0, 0.0, 1.0, 0.0)

  - The camera is positioned at the origin (0, 0, 0)
  - Aimed into the screen (negative z_axis)
  - Faced upwards (positive y_axis)

View Transform – Compute the Camera Coordinates (x_c, y_c, z_c)

- Fix z_c to be the opposite of AT

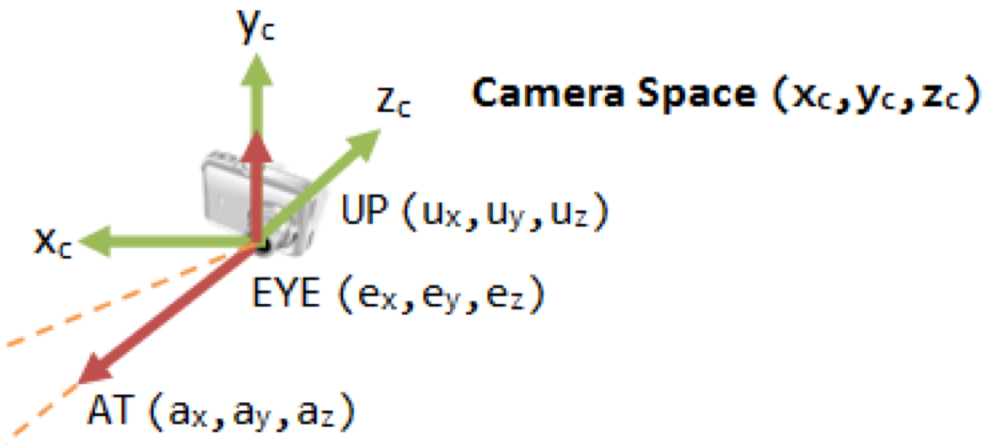
$$z_c = \frac{EYE - AT}{\|EYE - AT\|}$$

- Obtain the direction of x_c by taking the cross-product of AT and UP

$$x_c = \frac{UP \times z_c}{\|UP \times z_c\|}$$

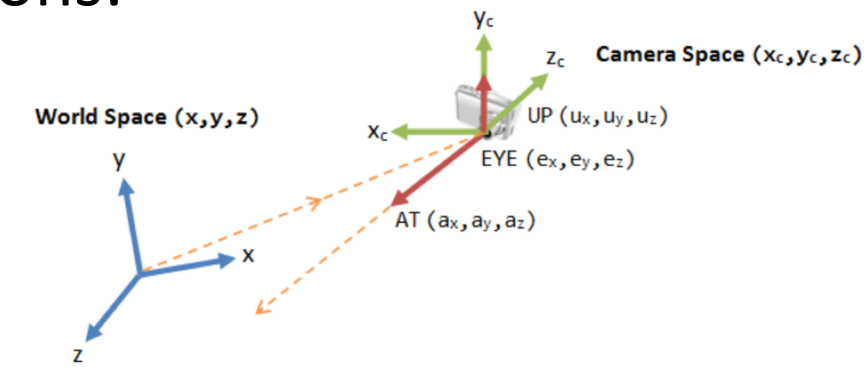
- Get the direction of y_c by taking the cross-product of x_c and z_c

$$y_c = z_c \times x_c$$



View Transform – View Matrix

- The view transform consists of two operations:
  - A translation (for moving EYE to the origin)



$$\text{Translation: } \boldsymbol{T}(-EYE) = \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

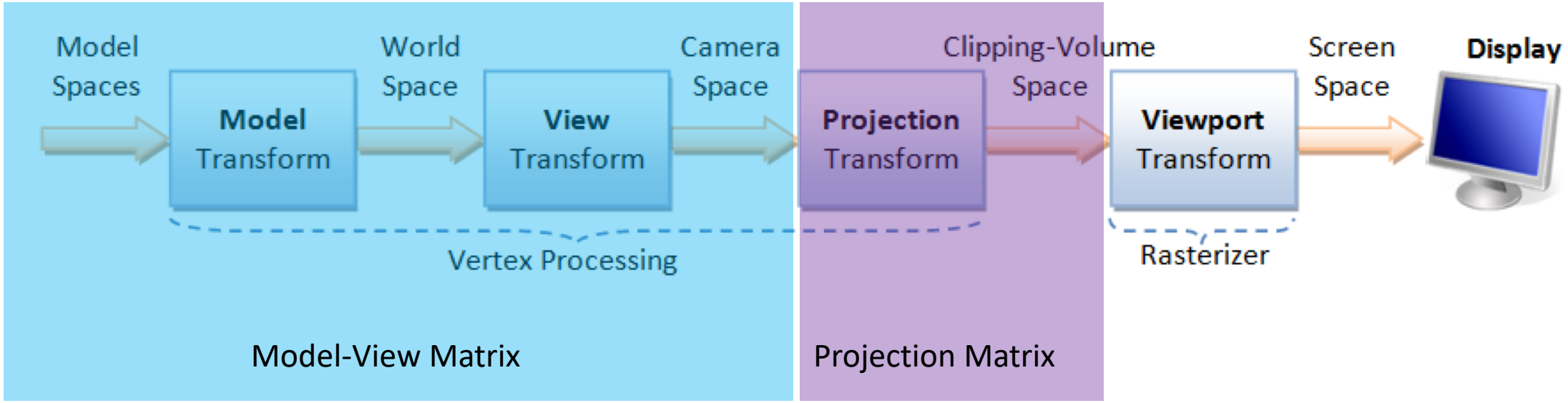
  - A rotation (to axis the axes)

$$\text{Rotation: } \boldsymbol{R} = \begin{bmatrix} x_x^c & x_y^c & x_z^c & 0 \\ y_x^c & y_y^c & y_z^c & 0 \\ z_x^c & z_y^c & z_z^c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \textit{where } x_c = \begin{bmatrix} x_x^c \\ x_y^c \\ x_z^c \end{bmatrix}, y_c = \begin{bmatrix} y_x^c \\ y_y^c \\ y_z^c \end{bmatrix}, z_c = \begin{bmatrix} z_x^c \\ z_y^c \\ z_z^c \end{bmatrix}$$

- We can combine the two operations into one single View Matrix:

$$\text{View Matrix: } \boldsymbol{M_{view}} = \boldsymbol{R}\,\boldsymbol{T} = \begin{bmatrix} x_x^c & x_y^c & x_z^c & 0 \\ y_x^c & y_y^c & y_z^c & 0 \\ z_x^c & z_y^c & z_z^c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} x_x^c & x_y^c & x_z^c & -e_x \cdot x_c \\ y_x^c & y_y^c & y_z^c & -e_y \cdot y_c \\ z_x^c & z_y^c & z_z^c & -e_z \cdot z_c \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Model-View Transform




```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

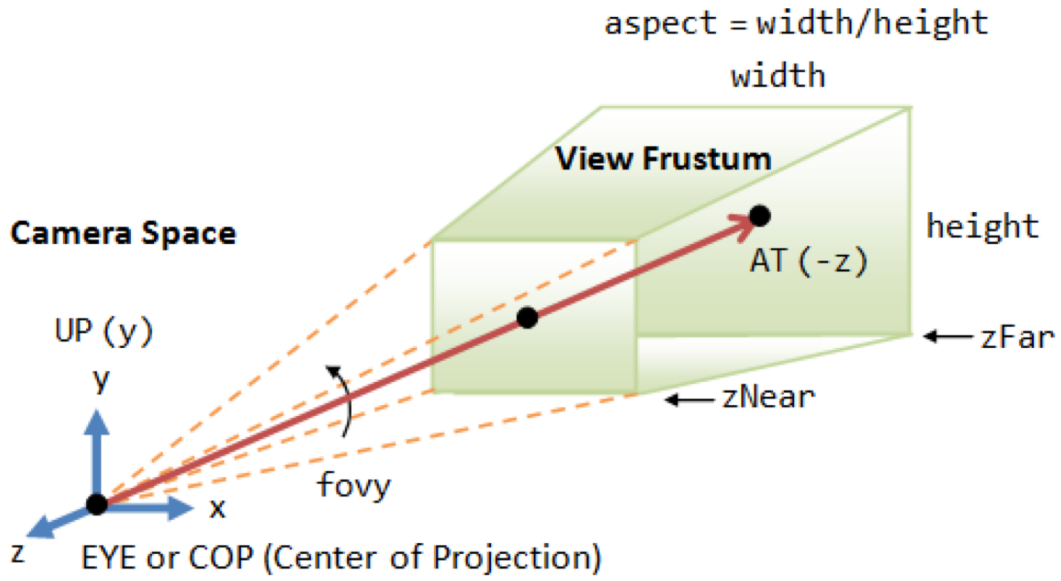
```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
```

Projection Transform

- Projection transformation defines a viewing volume to
  - Determines how an object is projected on to the screen
    - Perspective projection
    - Orthographic projection
  - Defines which objects or potions of objects are clipped out of the final image

Perspective Projection

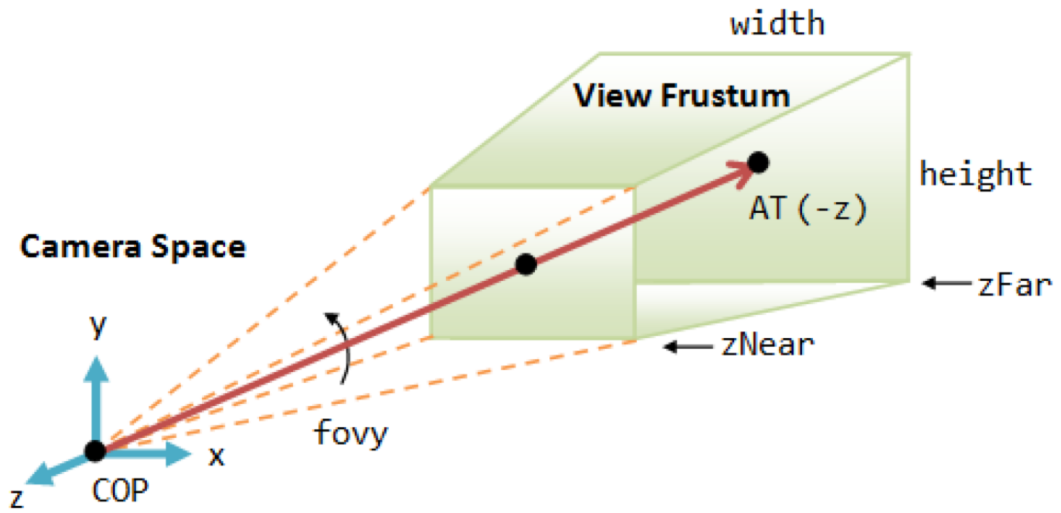
- The viewing volume for a perspective projection is a frustum of a pyramid
- The camera's view frustum is specified via 4 view parameters:
  - Fovy: specify the total vertical angle of view in degrees
  - Aspect: the ratio of width vs. height
  - zNear; the near plane
  - zFar: the far plane
- View-frustum culling: an object outside the view frustum is not visible to the camera

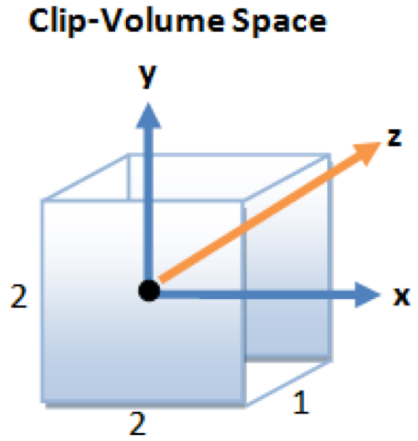


Perspective Projection – OpenGL Functions

- A GLU function to choose the perspective projection:
  - void **gluPerspective**(GLdouble fovy, GLdouble aspectRatio, GLdouble zNear, GLdouble zFar);
    - fovy is the angle [0.0,180.0]) between the bottom and top of the projectors
    - aspect ratio is the ratio of width and height of the front (and also back) clipping plane
    - zNear and zFar specify the front and back clipping planes
- A GL function to set clipping volume:
  - void **glFrustum**(GLdouble xLeft, GLdouble xRight, GLdouble yBottom, GLdouble yTop, GLdouble zNear, GLdouble zFar);
    - xLeft, xRight, yBottom and yTop specifies the front clipping plane
    - zNear and zFar specify the positions of the front and back clipping planes

Perspective Projection – Projection Matrix



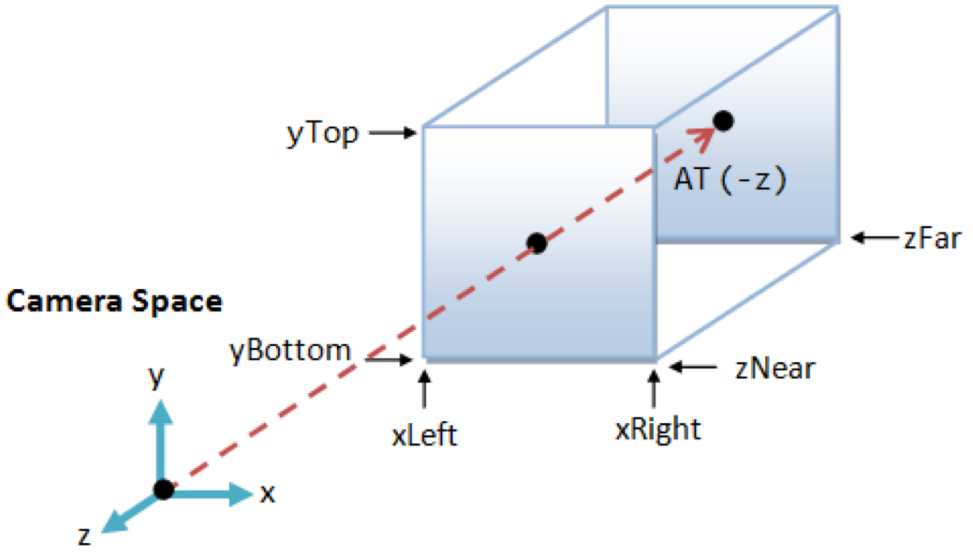


Clipping Volume
(2x2x1 Cuboid)

Projection Matrix: $\boldsymbol{M_{proj}} = \begin{bmatrix} \frac{cot\ (fovy/2)}{aspect} & 0 & 0 & 0 \\ 0 & cot\ (fovy/2) & 0 & 0 \\ 0 & 0 & -\frac{zFar}{zFar - zNear} & -\frac{zNear \times zFar}{zFar - zNear} \\ 0 & 0 & -1 & 0 \end{bmatrix}$

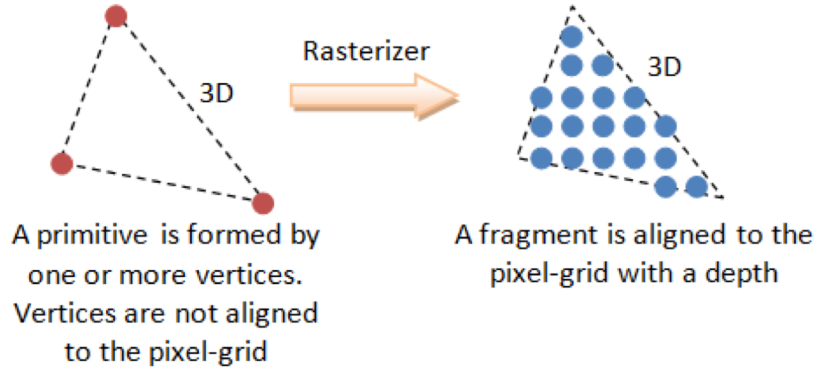
Orthographic Projection

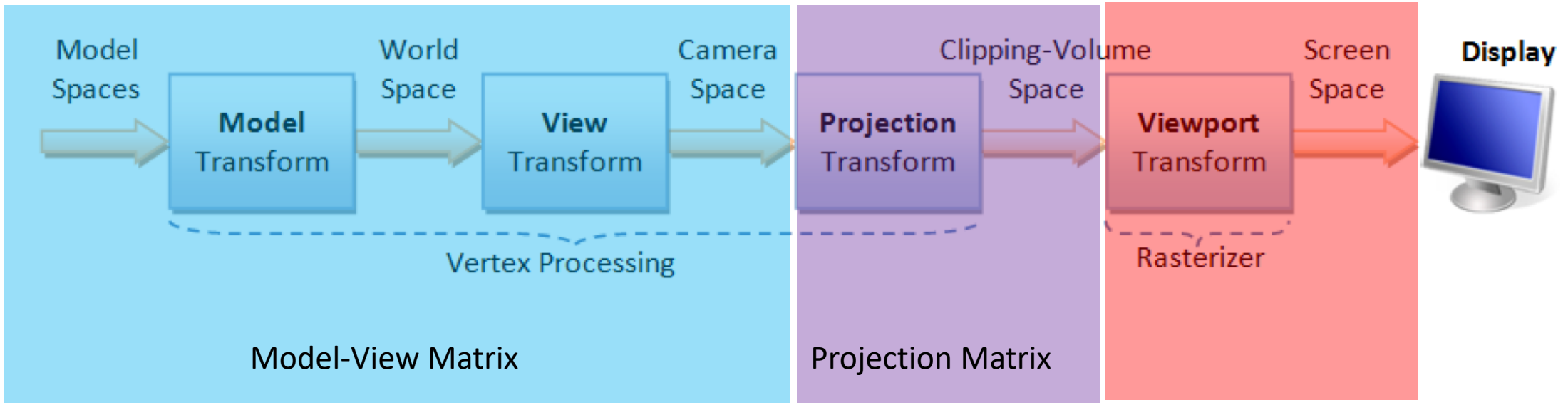
- The command glOrtho() creates an orthographic parallel viewing volume by specifying the corners of the near clipping plane and the distance to the far clipping plane:
  - void **glOrtho**(GLdouble xLeft, GLdouble xRight, GLdouble yBottom, GLdouble yTop, GLdouble zNear, GLdouble zFar);
  - void **gluOrtho2D**(GLdouble xLeft, GLdouble xRight, GLdouble yBottom, GLdouble yTop);
- The default 3D projection in OpenGL is the orthographic with parameters (-1.0, 1.0, -1.0, 1.0, -1.0, 1.0)



Orthographic Projection: Camera positioned infinitely far away at $z = \infty$

Rasterization



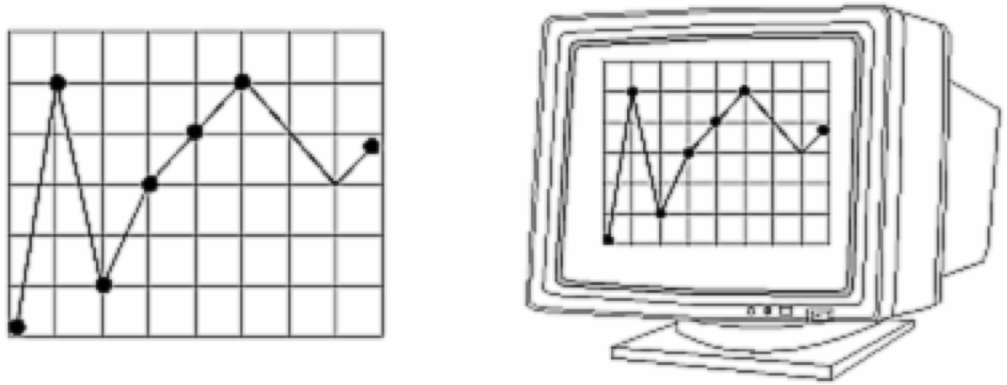



```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
```

Viewport Transform

- The viewport is the rectangular region of the window where the image is drawn, measured in screen's coordinates
- A viewport defines the size and shape of the display area to map the projected scene captured by the camera onto the application window



Viewport Transform – OpenGL Functions

- void **glViewport**(GLint x, GLint y, GLsizei width, GLsizei height);
  - Defines a pixel rectangle in the window into which the final image is mapped
  - The (x, y) parameter specifies the lower-left corner of the viewport rectangle in pixel
  - The width and height are the size of the viewport rectangle
  - By default, the initial viewport values are (0, 0, winWidth, winHeight), where winWidth and winHeight are the size of the window
- void **glDepthRange**(GLclampd nearVal, Glclampd farVal);
  - Sets the z-range of viewport
  - nearVal and farVal values represent adjustments to the minimum and maximum values that can be stored in the depth buffer

Viewport Transform – Viewport Matrix

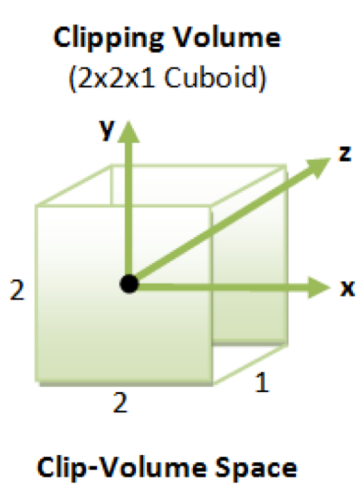
- Viewport transform consists of a series of reflection, scaling, and translation:

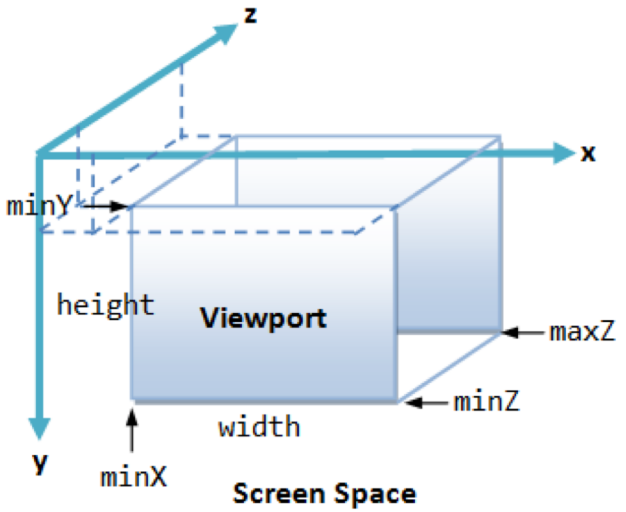
Reflection of y-axis: $\boldsymbol{M_1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Scaling of x, y, z axes: $\boldsymbol{M_2} = \begin{bmatrix} w/2 & 0 & 0 & 0 \\ 0 & h/2 & 0 & 0 \\ 0 & 0 & maxZ - minZ & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Translation of origin: $\boldsymbol{M_3} = \begin{bmatrix} 1 & 0 & 0 & minX + w/2 \\ 0 & 1 & 0 & minY + h/2 \\ 0 & 0 & 1 & minZ \\ 0 & 0 & 0 & 1 \end{bmatrix}$

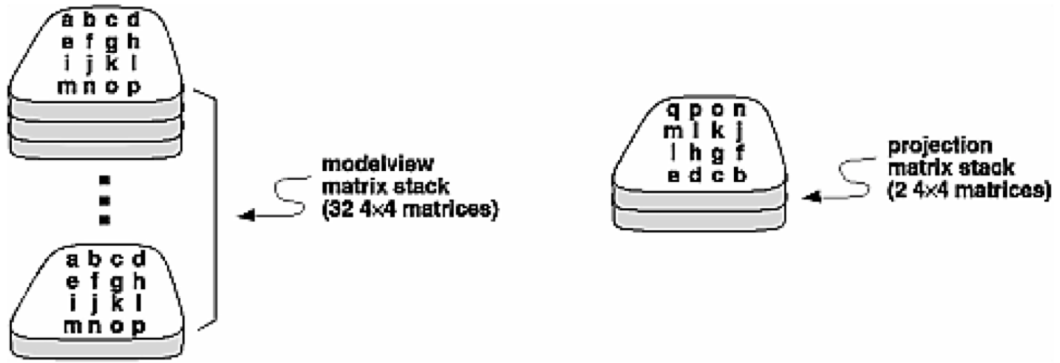
Viewport Matrix: $\boldsymbol{M_{viewport}} = \boldsymbol{M_3 M_2 M_1} = \begin{bmatrix} w/2 & 0 & 0 & minX + w/2 \\ 0 & -h/2 & 0 & minY + h/2 \\ 0 & 0 & maxZ - minZ & minZ \\ 0 & 0 & 0 & 1 \end{bmatrix}$





Matrix Stacks

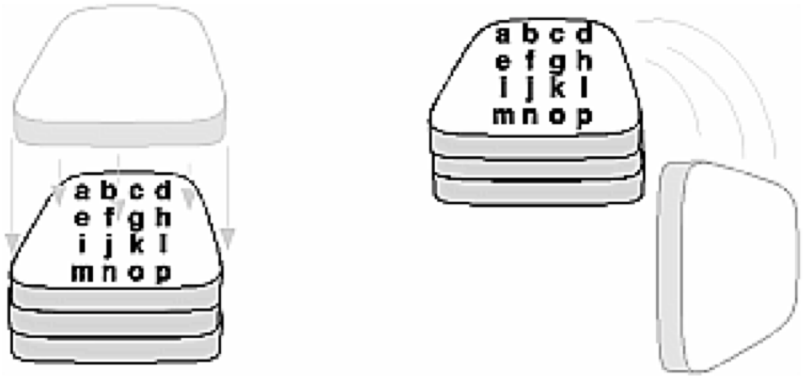
- All the modelview and projection transformations are stored as matrices
- Each of these matrices is actually the topmost member of a stack of matrices
- Commands that create specific transformation matrices deal with the current matrix on the top of the stack



Modelview and Projection Matrix Stacks

Manipulating the Matrix Stacks

- We can control which matrix is on top with the commands that perform stack operations:
  - void **glPushMatrix**(void);
    - Pushes all matrices in the current stack down one level
    - The topmost matrix is copied, so its contents are duplicated in both the top and second-from-the-top matrix
  - void **glPopMatrix**(void);
    - Pops the top matrix off the stack
    - Destroying the contents of the popped matrix. What was the second-from-the-top matrix becomes the top matrix
    - If the stack contains a single matrix, calling glPopMatrix() generates an error



Pushing and Popping the Matrix Stacks

Manipulating the Matrix Stacks Example

- Suppose you're drawing an automobile that has four wheels, each of which is attached to the car with five bolts

```
draw_body_and_wheel_and_bolts()
{
   draw_car_body();
   glPushMatrix();
      glTranslatef(40,0,30);     /*move to first wheel position*/
      draw_wheel_and_bolts();
   glPopMatrix();
   glPushMatrix();
      glTranslatef(40,0,-30);    /*move to 2nd wheel position*/
      draw_wheel_and_bolts();
   glPopMatrix();
   ...                        /*draw last two wheels similarly*/
}
```

Draw the car body, remember where you are, and translate to the right front wheel. Draw the wheel and throw away the last translation so your current position is back at the origin of the car body. Remember where you are, and translate to the left front wheel....

```
draw_wheel_and_bolts()
{
   long i;

   draw_wheel();
   for(i=0;i<5;i++){
      glPushMatrix();
         glRotatef(72.0*i,0.0,0.0,1.0);
         glTranslatef(3.0,0.0,0.0);
         draw_bolt();
      glPopMatrix();
   }
}
```

Draw the wheel, remember where you are, and successively translate to each of the positions that bolts are drawn, throwing away the transformations after each bolt is drawn.

Example: Transformed Cube

```
1  #include <GL/gl.h>
2  #include <GL/glu.h>
3  #include <GL/glut.h>
4
5  void init(void)
6  {
7     glClearColor (0.0, 0.0, 0.0, 0.0);
8     glShadeModel (GL_FLAT);
9  }
10
11 void display(void)
12 {
13    glClear (GL_COLOR_BUFFER_BIT);
14    glColor3f (1.0, 1.0, 1.0);
15    glLoadIdentity ();
16    gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
17    glScalef (1.0, 2.0, 1.0);
18    glutWireCube (1.0);
19    glFlush ();
20 }
21
22 void reshape (int w, int h)
23 {
24    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
25    glMatrixMode (GL_PROJECTION);
26    glLoadIdentity ();
27    glFrustum (-1.0, 1.0, -1.0, 1.0, 1.5, 20.0);
28    glMatrixMode (GL_MODELVIEW);
29 }
30
```

Set the current matrix to the identity matrix

Specify viewing transformation

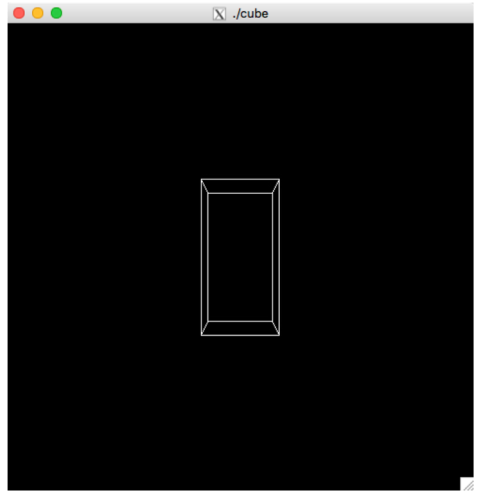
Specify modeling transformation

Specify viewport transformation

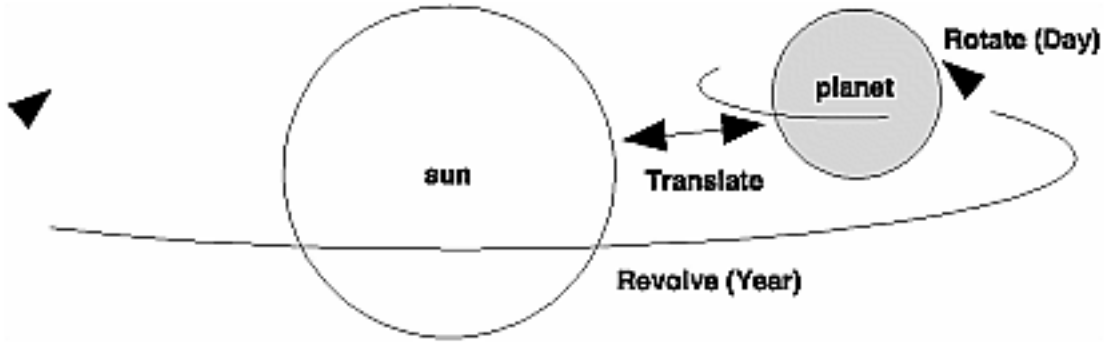
Initialize the current projection matrix

Specify projection transformation

```
32 int main(int argc, char** argv)
33 {
34    glutInit(&argc, argv);
35    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
36    glutInitWindowSize (500, 500);
37    glutInitWindowPosition (100, 100);
38    glutCreateWindow (argv[0]);
39    init ();
40    glutDisplayFunc(display);
41    glutReshapeFunc(reshape);
42    glutMainLoop();
43    return 0;
44 }
```



Example: Planet and Sun

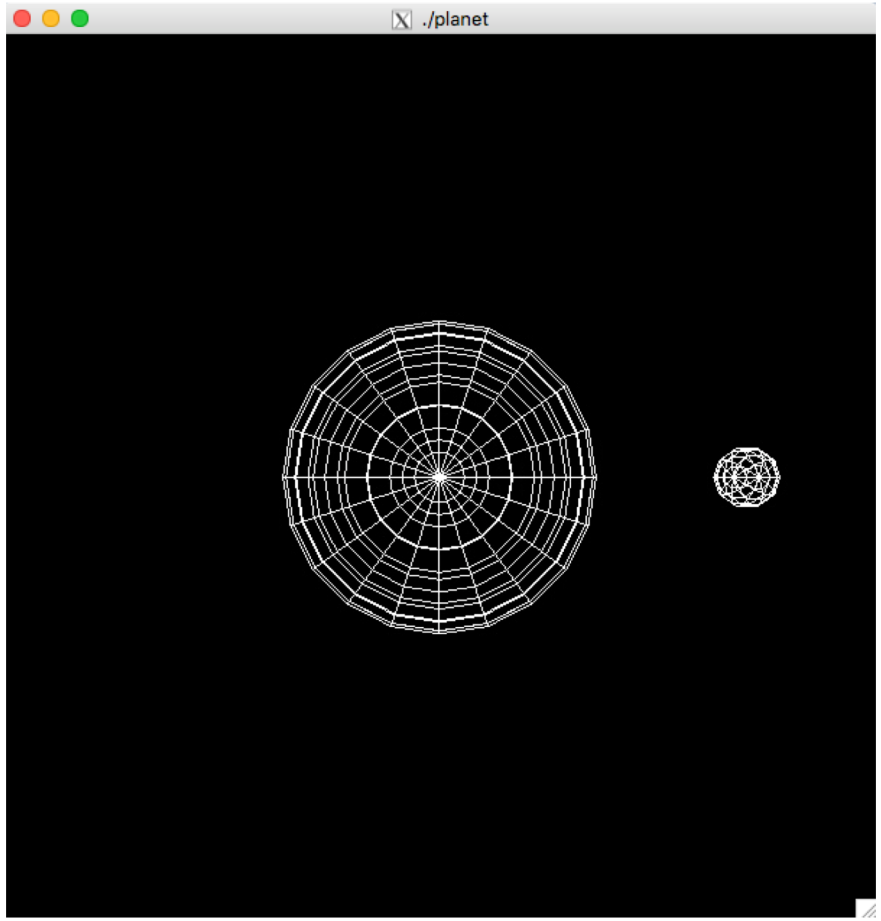


- Draw the sun at the origin of the fixed coordinate system
- Draw a planet rotating around the sun
  - Rotates about its own axis once a day
  - Completes one revolution around the sun once a year

Example: Planet and Sun (Cont.)

```c
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>

static int year = 0, day = 0;

void init(void)
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL_FLAT);
}

void display(void)
{
   glClear (GL_COLOR_BUFFER_BIT);
   glColor3f (1.0, 1.0, 1.0);

   glPushMatrix();
   glutWireSphere(1.0, 20, 16);   /* draw sun */
   glRotatef ((GLfloat) year, 0.0, 1.0, 0.0);
   glTranslatef (2.0, 0.0, 0.0);
   glRotatef ((GLfloat) day, 0.0, 1.0, 0.0);
   glutWireSphere(0.2, 10, 8);    /* draw smaller planet */
   glPopMatrix();
   glutSwapBuffers();
}

void reshape (int w, int h)
{
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();
   gluPerspective(60.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
}

void keyboard (unsigned char key, int x, int y)
{
   switch (key) {
      case 'd':
         day = (day + 10) % 360;
         glutPostRedisplay();
         break;
      case 'D':
         day = (day - 10) % 360;
         glutPostRedisplay();
         break;
      case 'y':
         year = (year + 5) % 360;
         glutPostRedisplay();
         break;
      case 'Y':
         year = (year - 5) % 360;
         glutPostRedisplay();
         break;
      default:
         break;
   }
}

int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
   glutInitWindowSize (500, 500);
   glutInitWindowPosition (100, 100);
   glutCreateWindow (argv[0]);
   init ();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutMainLoop();
   return 0;
}
```

Example: Planet and Sun (Cont.)



Summary

- Viewing and modeling transformations: orient the model and the camera relative to each other to obtain the desired final image
- Projection transformations: specify the shape and orientation of the viewing volume, which determines how a scene is projected onto the screen (with a perspective or orthographic projection) and which objects or parts of objects are clipped out of the scene
- Viewport transformation: control the conversion of three-dimensional model coordinates to screen coordinates
- Manipulating the matrix stacks: save and restore certain transformations