Computer Graphics
- Transformations in OpenGL

Junjie Cao @ DLUT

Spring 2016

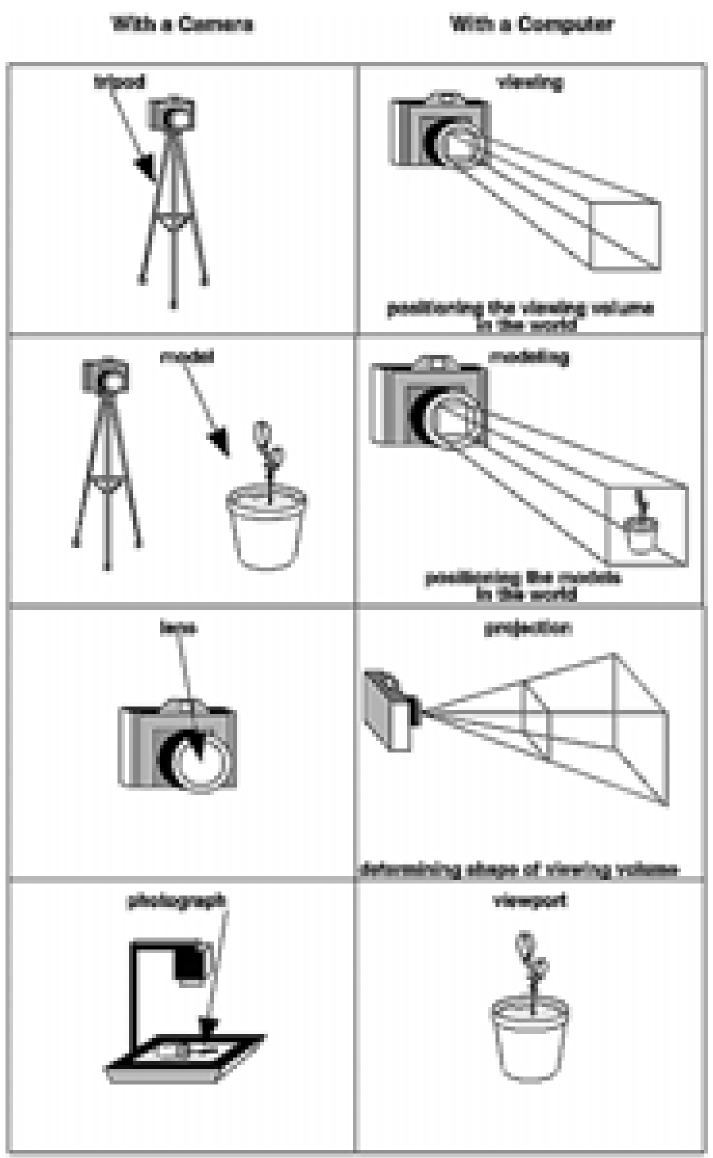
http://jjcao.github.io/ComputerGraphics/

Camera Analogy

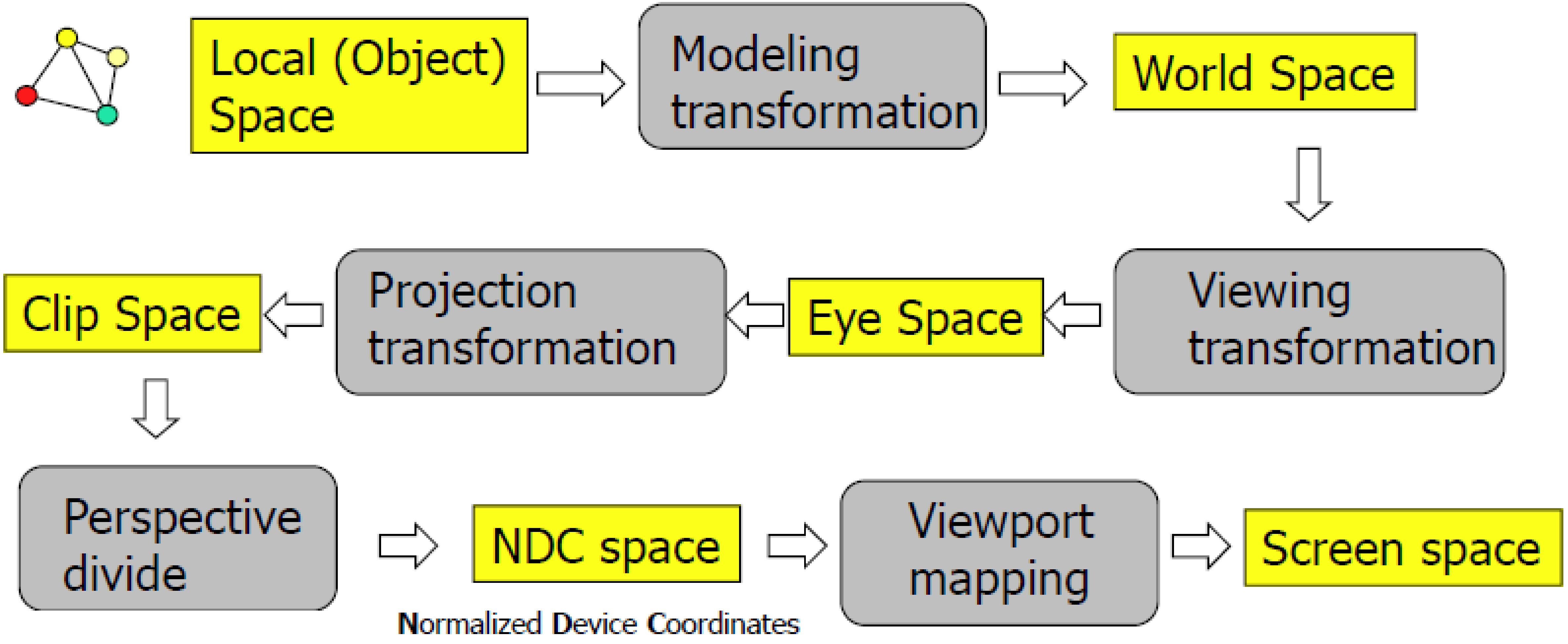
- OpenGL coordinate system has different origin (lower-left corner) from the window system (upper-left corner)

- The transformation process to produce the desired scene for viewing is analogous to taking a photograph with a camera

- The steps with a camera (or a computer) might be the following:
    - Arrange the scene to be photographed into the desired composition (modelling transformation)
    - Set up your tripod and pointing the camera at the scene (viewing transformation).
    - Choose a camera lens or adjust the zoom (projection transformation)
    - Determine how large you want the final photograph to be (viewport transformation)

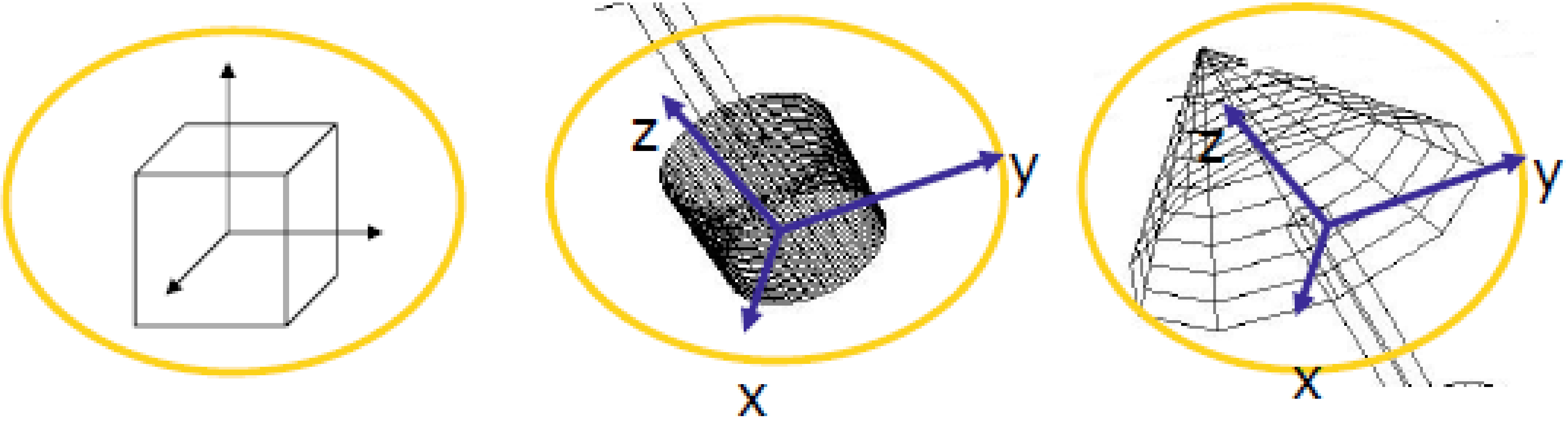


Transformation Pipeline



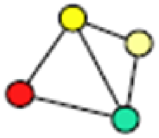
Local Coordinate System

- When you load a file containing a 3d object, its vertices stores coordinates in local CS.
- Assuming obj1, obj2 & obj3 are loaded.
  - Normally, their centers are the origins if they are actually created by code or hand.
  - Sometimes, their centers are not the origins of their local CS respectively if they are results of 3D scanning, etc.
  - Anyway, they are treated as local CS



World Coordinate System

- When the obj is just loaded, its local CS is used as WCS.
- To place multiple objs in your WCS, you need specify position, size, orientation of them
- Transformations need to be performed to position the object in WCS



Local (Object) Space ⇒ Modeling transformation ⇒ World Space

- A modeling transformation is a sequence of translations, rotations, scalings (in arbitrary order) matrices multiplied together

$$\begin{aligned} \mathbf{x}' &= \mathbf{m_{11}x} + \mathbf{m_{12}y} + \mathbf{m_{13}z} \\ \mathbf{y}' &= \mathbf{m_{21}x} + \mathbf{m_{22}y} + \mathbf{m_{23}z} \\ \mathbf{z}' &= \mathbf{m_{31}x} + \mathbf{m_{32}y} + \mathbf{m_{33}z} \end{aligned}$$

or

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \underbrace{\begin{pmatrix} m_{11} & m_{12} & m_{13} & 0 \\ m_{21} & m_{22} & m_{23} & 0 \\ m_{31} & m_{32} & m_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{Modeling transformation matrix}} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

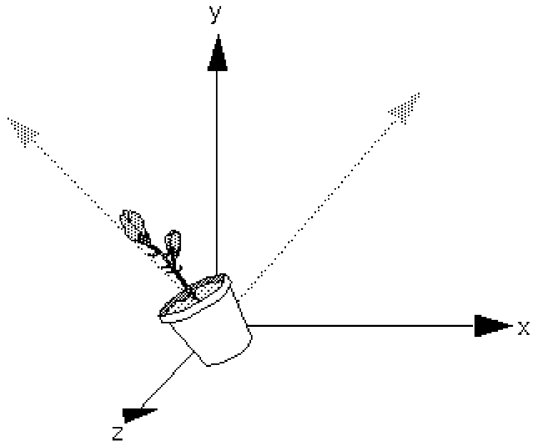
Modeling Transformations

- The three OpenGL routines for modeling transformations are:
  - glTranslate*(),
  - glScale*()
  - void **glRotate**{fd}(TYPE *angle*, TYPE *x*, TYPE *y*, TYPE *z*);
  - **glRotatef(**45.0, 0.0, 0.0, 1.0**)**

  deprecated

- These routines **transform an object (or coordinate system**, if you're thinking of it that way) by moving, rotating, stretching, shrinking, or reflecting it

- All three commands are **equivalent** to producing an appropriate translation, rotation, or scaling **matrix**, and then calling **glMultMatrix***() with that matrix as the argument



- OpenGL **automatically** computes the matrices for you

Modeling Transformations (cont)

- Each of these postmultiplies the *current matrix*
  - E.g., if current matrix is **C**, then **C=CS**

- The current matrix is either the **modelview** matrix or the projection matrix (also a texture matrix, won't discuss)
  - Set these with glMatrixMode(), e.g.:

    glMatrixMode(GL_MODELVIEW);

    glMatrixMode(GL_PROJECTION);

- **WARNING: common mistake ahead!**
  - Be sure that you are in **GL_MODELVIEW** mode before making modeling or viewing calls!
  - Ugly mistake because it can appear to work, at least for a while…, see https://sjbaker.org/steve/omniv/projection_abuse.html

Example for Modeling Transformation 1

```
void display() {
glClearColor(0,0,1,1);
glClear(GL_COLOR_BUFFER_BIT);
glColor4f(1,1,0,1); //glColor* have been deprecated in OpenGL 3

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glRotatef(45, 0,0,1);

glBegin(GL_TRIANGLES); //glBegin/End have been deprecated in OpenGL 3
glColor4f(1,1,0,1);
glVertex4f(vertices[0], vertices[1], vertices[2], vertices[3]);
…
```

More details will be explained

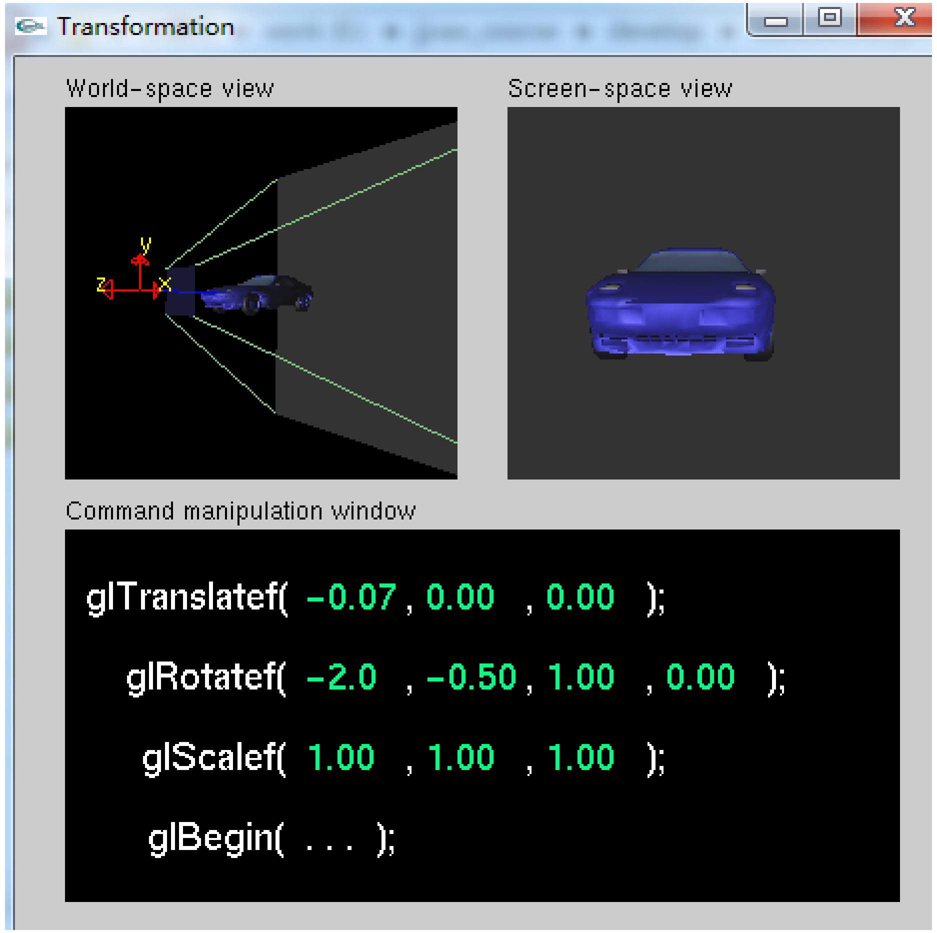
```
float vertices[] = {-0.5, -0.5, 0.0, 1.0,  // first triangle
                    -0.5, 0.5, 0.0, 1.0,
                     0.5, 0.5, 0.0, 1.0,
                     0.5, 0.5, 0.0, 1.0,   // second triangle
                     0.5, -0.5, 0.0, 1.0,
                    -0.5, -0.5, 0.0, 1.0};
```

Example for Modeling Transformation 2

```
glVertex4f(vertices[4], vertices[5], vertices[6], vertices[7]);
glVertex4f(vertices[8], vertices[9], vertices[10], vertices[11]);
glColor4f(1,0,0,1);
glVertex4f(vertices[12], vertices[13], vertices[14], vertices[15]);
glVertex4f(vertices[16], vertices[17], vertices[18], vertices[19]);
glVertex4f(vertices[20], vertices[21], vertices[22], vertices[23]);
glEnd();
glutSwapBuffers();
}
```

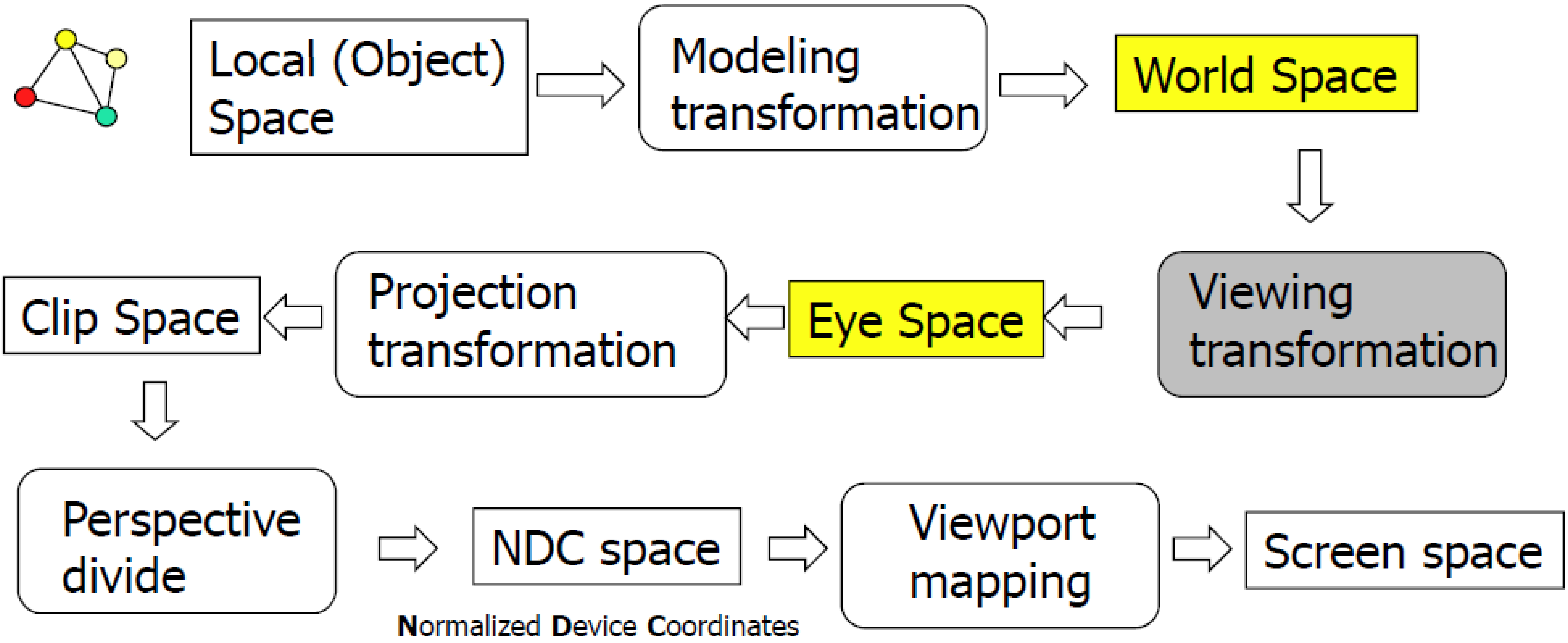
```
float vertices[] = {-0.5, -0.5, 0.0, 1.0,  // first triangle
                    -0.5, 0.5, 0.0, 1.0,
                     0.5, 0.5, 0.0, 1.0,
                     0.5, 0.5, 0.0, 1.0,   // second triangle
                     0.5, -0.5, 0.0, 1.0,
                    -0.5, -0.5, 0.0, 1.0};
```

Modeling Transformations (cont)



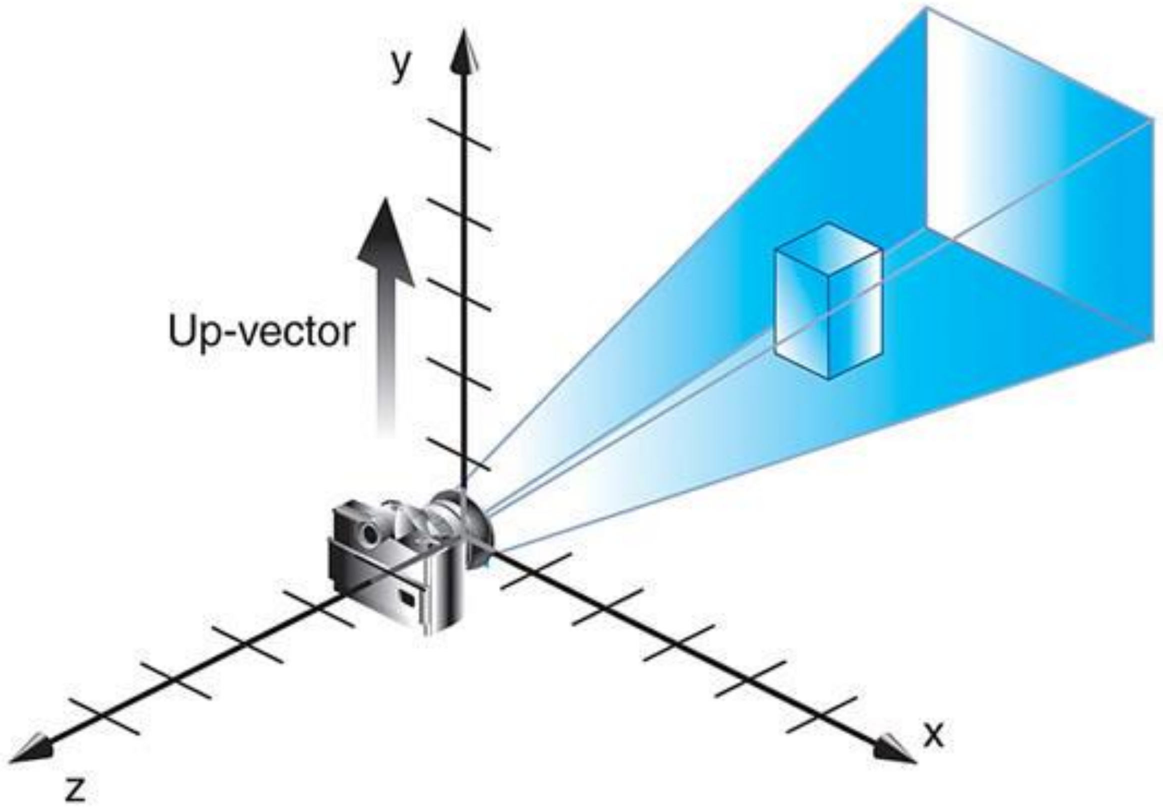
Nate_Robins_tutorials: Transformation

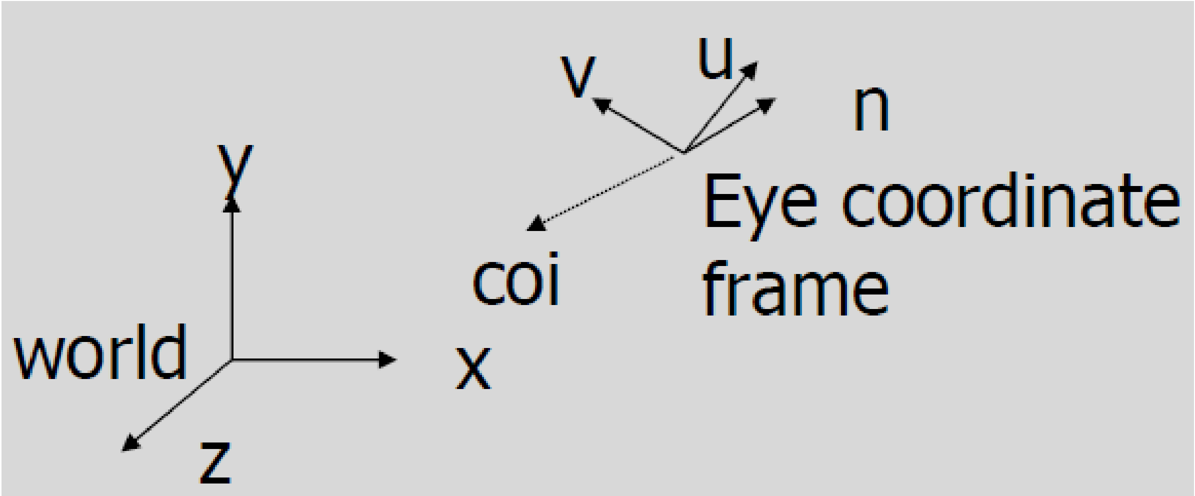
Viewing transformation



Viewing Transformation

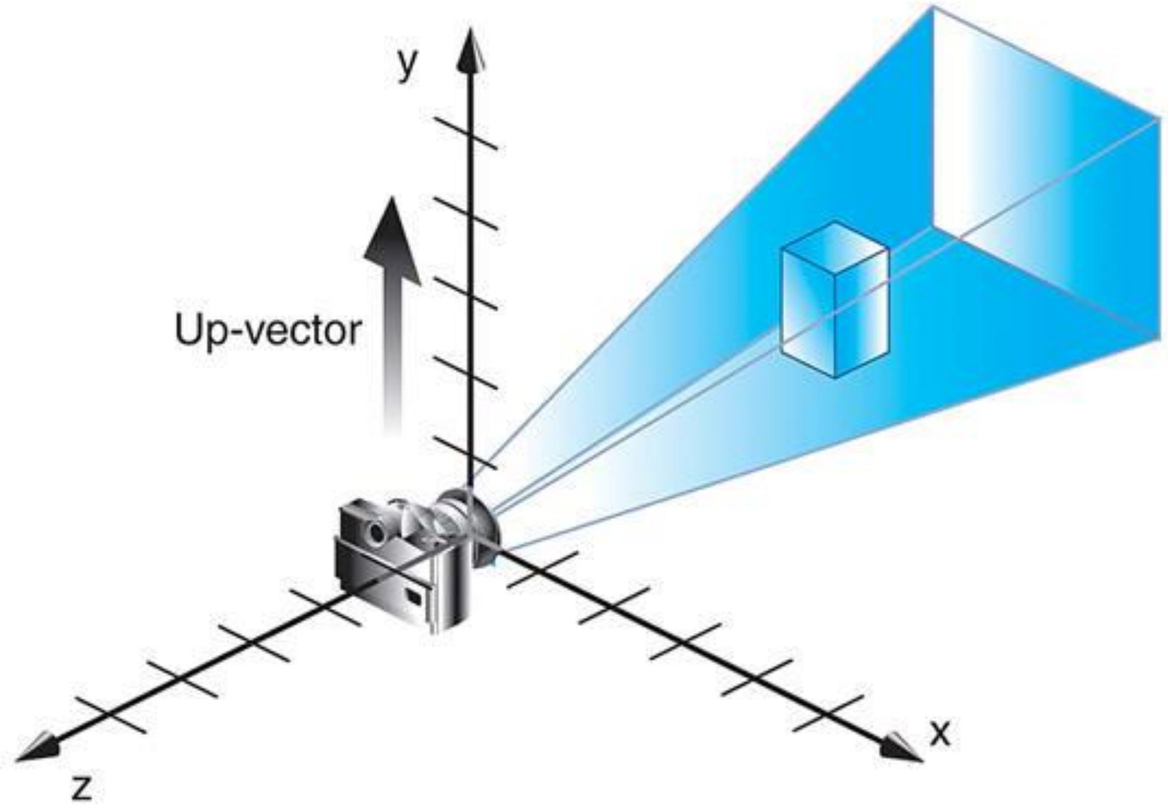
- Convert from WCS to the camera (eye) coordinate sys
- The camera position is the origin initially.
- The objs are also in the origin mostly. Or have been placed well in WCS
- Anyway, we need move the camera to see what we wish to see

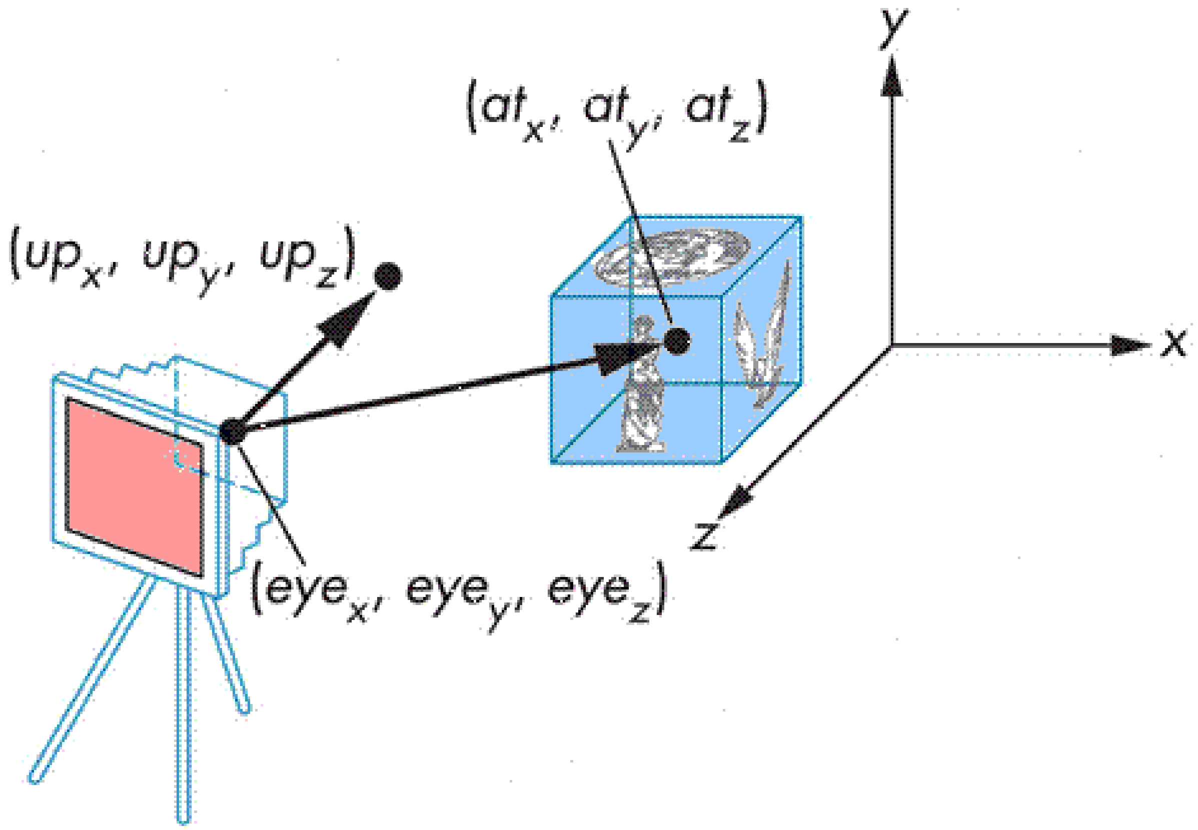




Viewing Transformation

- void **gluLookAt**(*eyeX*, *eyeY*, *eyeZ*, *centerX*, *centerY*, *centerZ*, *upX*, *upY*, *upZ*);





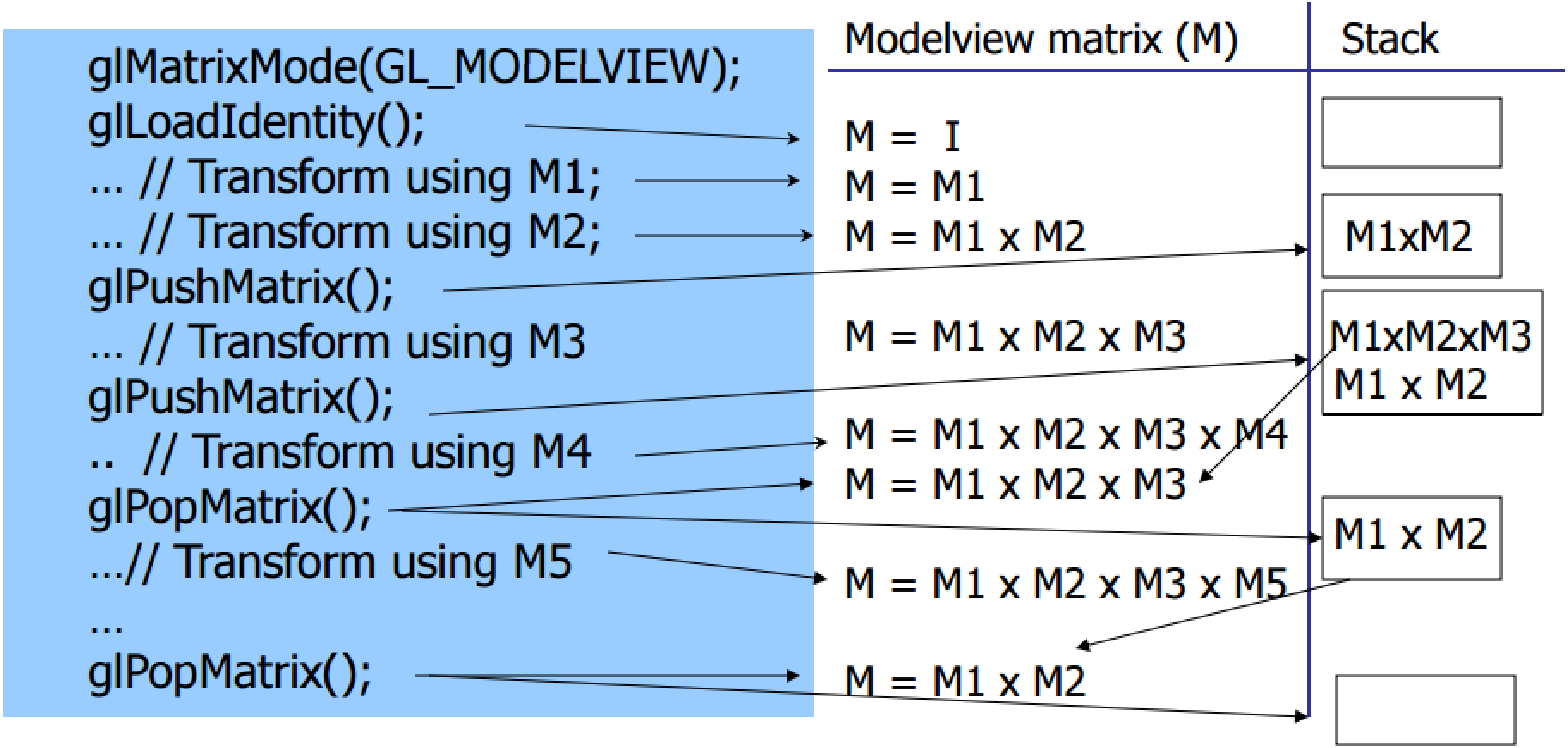
Example: modeling + viewing transformation

- With all this, we can give an outline for a typical display routine for drawing an image of a 3D scene with OpenGL 1.1:

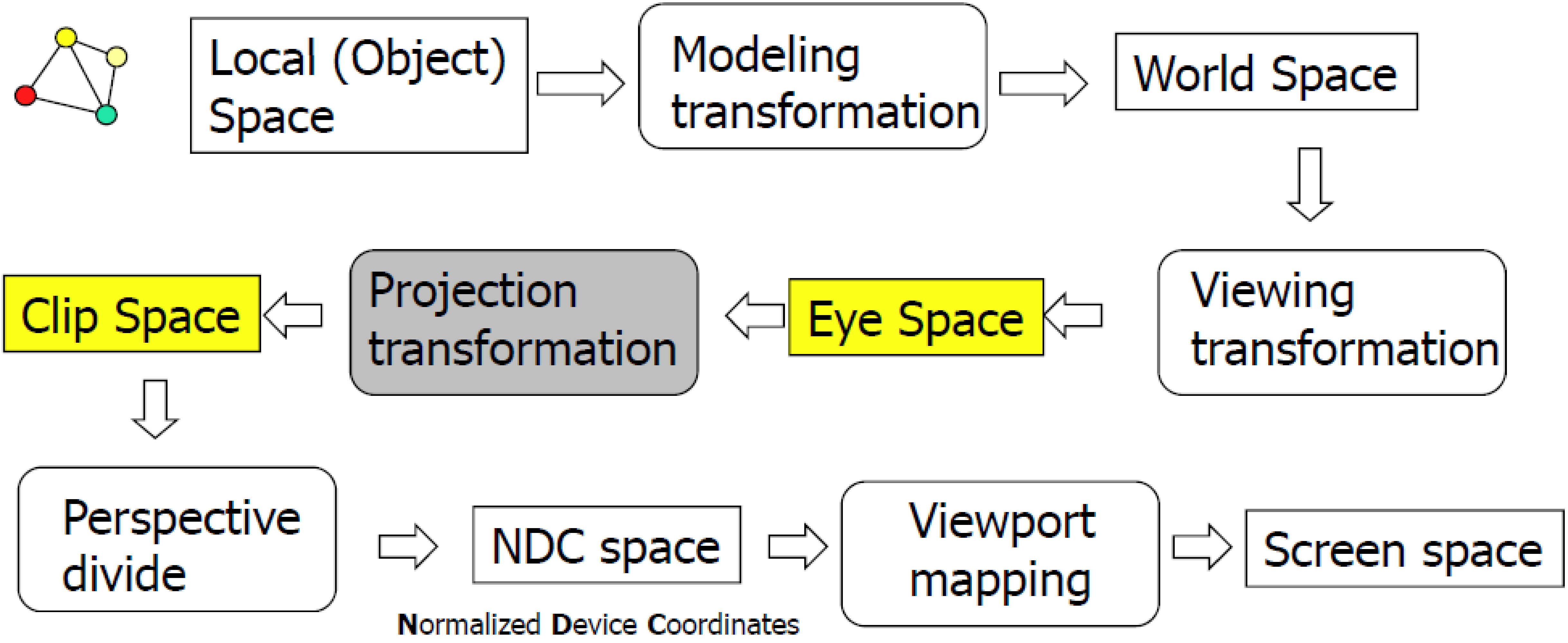
```
// possibly set clear color here, if not set elsewhere
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

// possibly set up the projection here, if not done elsewhere
glMatrixMode( GL_MODELVIEW ); glLoadIdentity();
gluLookAt( eyeX,eyeY,eyeZ, refX,refY,refZ, upX,upY,upZ ); // Viewing transform

glPushMatrix();
. .. // apply modeling transform and draw an object
glPopMatrix();

glPushMatrix();
. .. // apply another modeling transform and draw another object
glPopMatrix()
…
```

Copy the current matrix and push it onto a stack: **glPushMatrix()**
Discard the current matrix and replace it with whatever's on top of the stack: **glPopMatrix()**

Push and Pop Matrix Stack

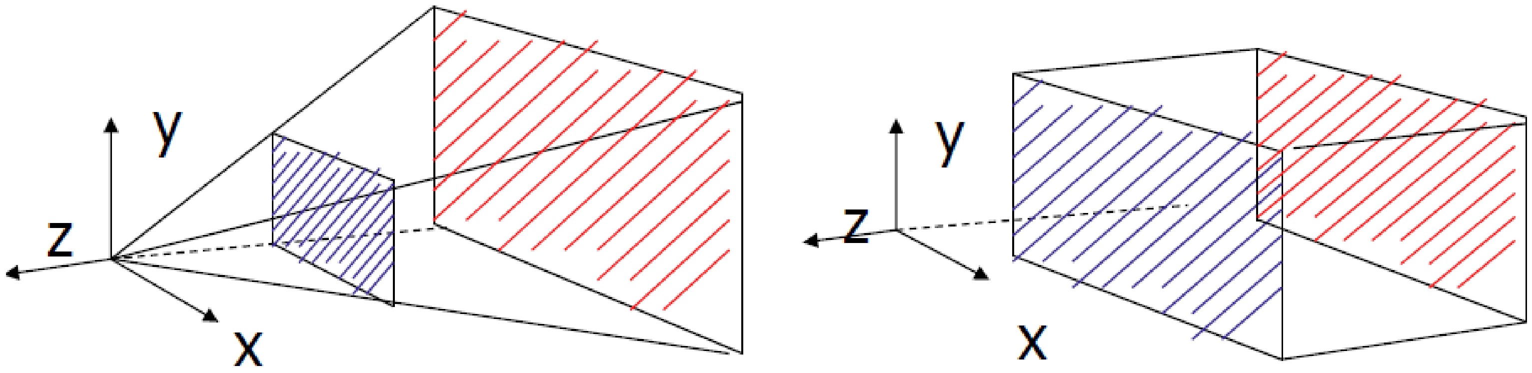


Transformation Pipeline



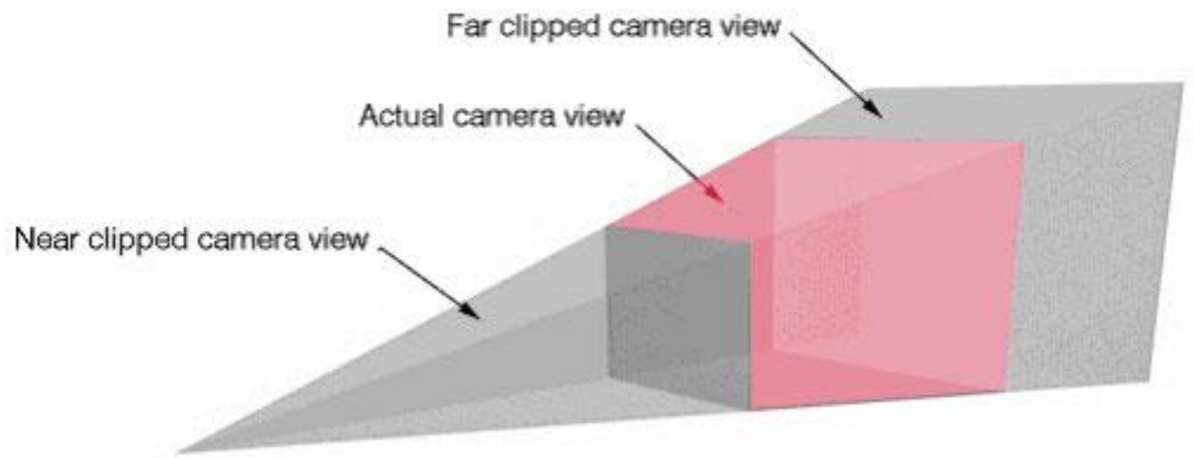
Projection Transformation

- Specifying PT is like choosing a **lens for a camera**
- The purpose of PT is to define a **viewing volume**, which is used in two ways.
  - The viewing volume determines **how an object is projected** onto the screen (that is, by using a perspective or an orthographic projection), and
  - Defines **which objects or portions of objects are clipped out** of the final image
- Need to establish the appropriate mode for constructing the viewing transformation, or in other words select the projection mode
  - **glMatrixMode(GL_PROJECTION);**
- This designates the projection matrix as the current matrix, which is originally set to the identity matrix



Perspective: **gluPerspective()** Parallel: **glOrtho()**

Perspective Projection



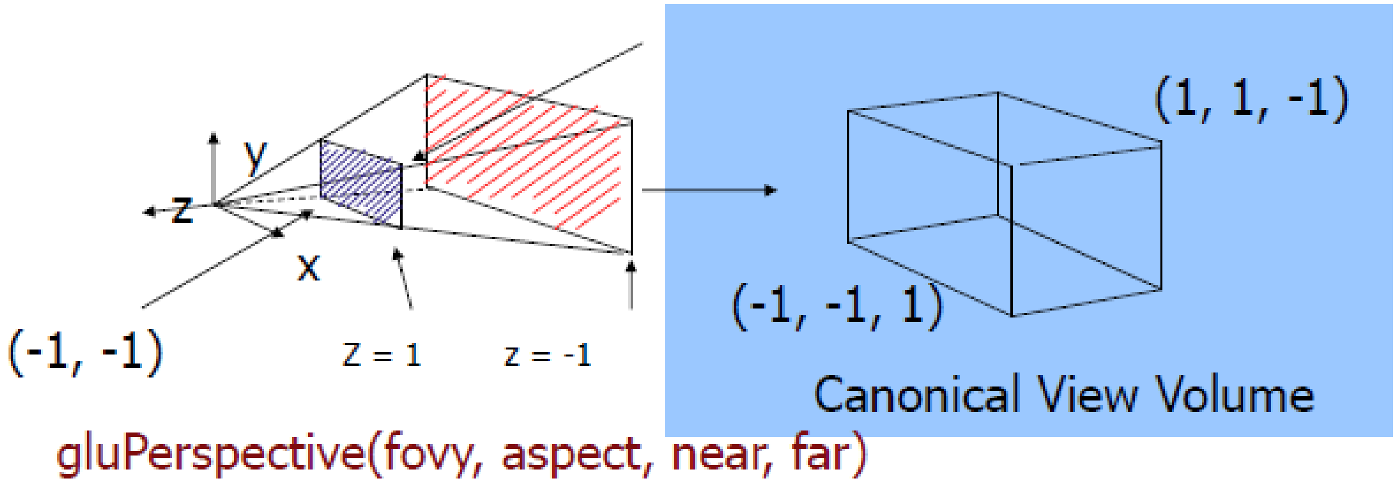
- The command to define a frustum, glFrustum(), calculates a perspective projection matrix and multiplies the current projection matrix (typically the identity matrix) by it
    - **glFrustum(xmin, xmax, ymin, ymax, N, F)** N = near plane, F = far plane ar);

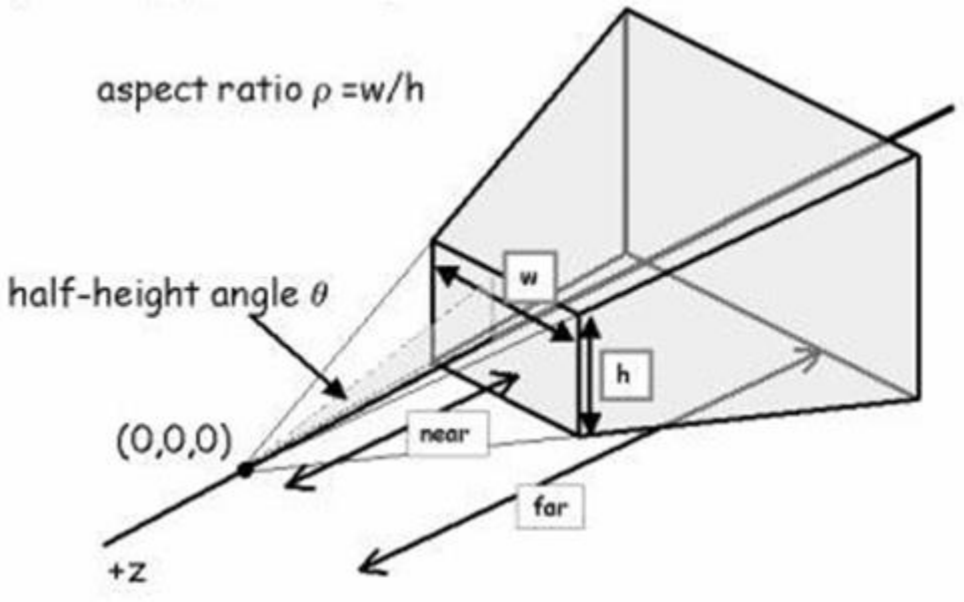
$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} 2N/(xmax-xmin) & 0 & (xmax+xmin)/(xmax-xmin) & 0 \\ 0 & 2N/(ymax-ymin) & (ymax+ymin)/(ymax-ymin) & 0 \\ 0 & 0 & -(F+N)/(F-N) & -2F*N/(F-N) \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Projection Matrix

gluPerspective

- glFrustum() isn't intuitive to use so can use **gluPerspective** to specify
  - Fovy: the angle of the field of view in the y direction
  - Aspect: the aspect ratio of the width to height (x/y)
  - Near & far: distance between the viewpoint and the near and far clipping planes
- Note that gluPerspective() is limited to creating frustums that are symmetric in both the x- and y-axes along the line of sight

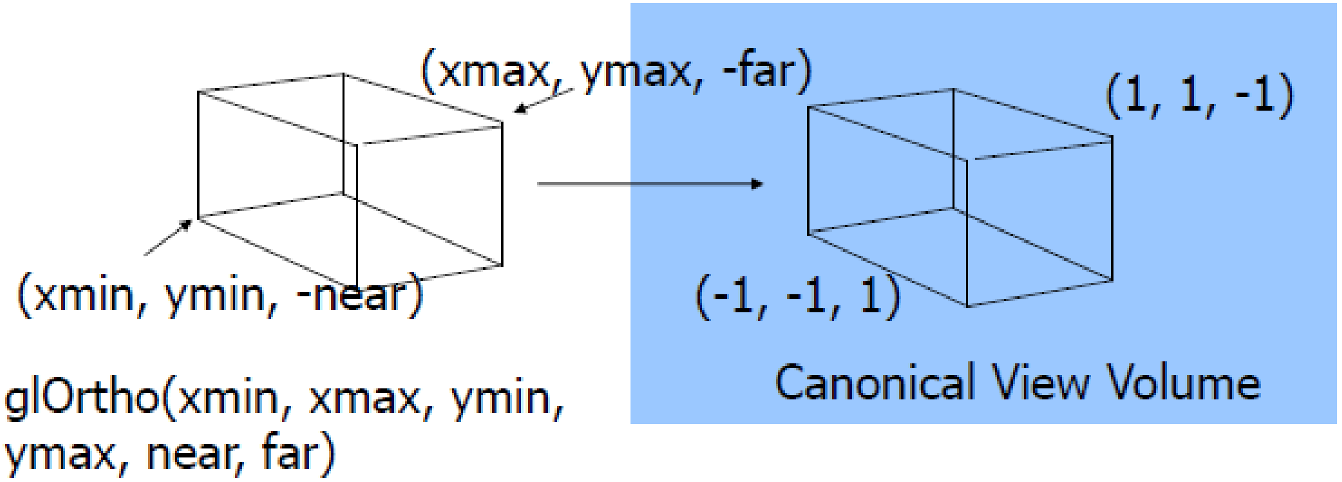


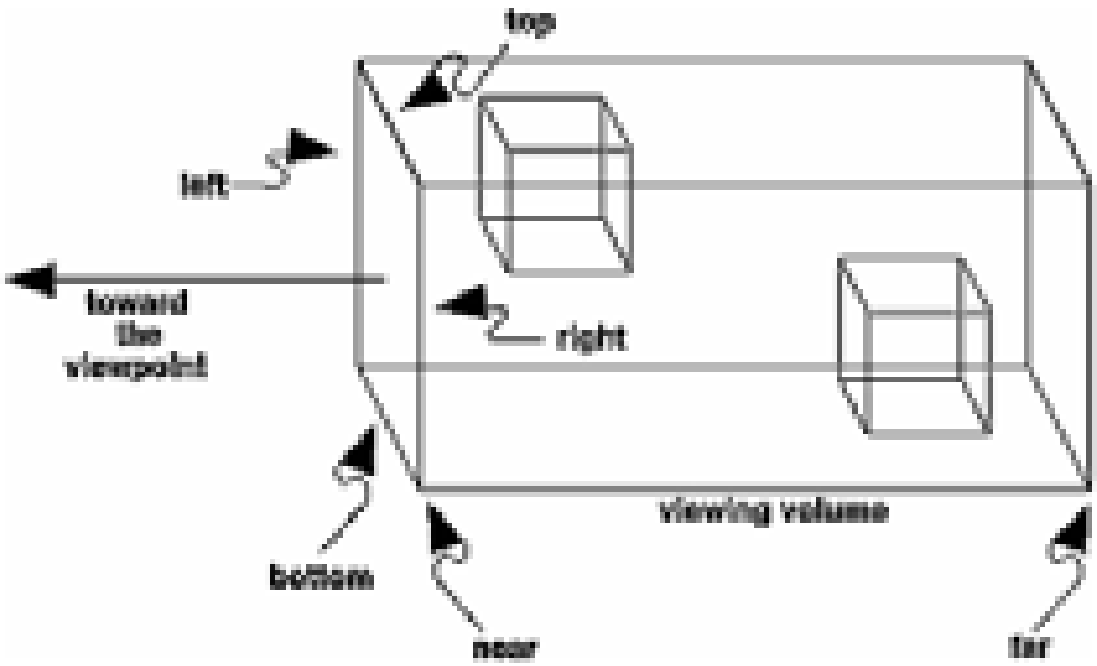


Maps (projects) everything in the visible volume into a canonical view volume

Orthographic Projection

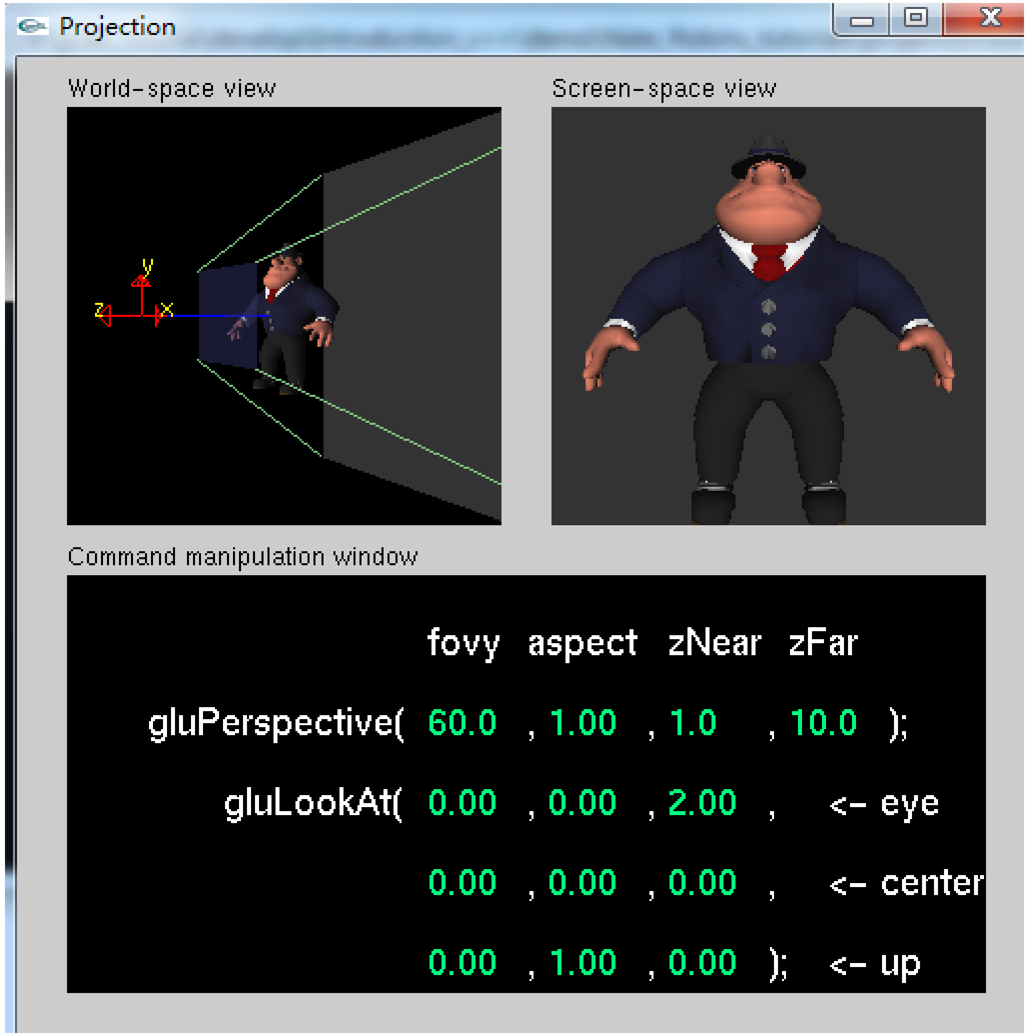
- Ortographic projection is used for applications such as creating architectural blueprints and computer-aided design, where it's crucial to maintain the actual sizes of objects and angles between them
  - void **gluOrtho2D** (left, right, bottom, top);
  - void **glOrtho** (left, right, bottom, top, near, far);





Maps (projects) everything in the visible volume into a canonical view volume

Projection & Viewpoint (cont)

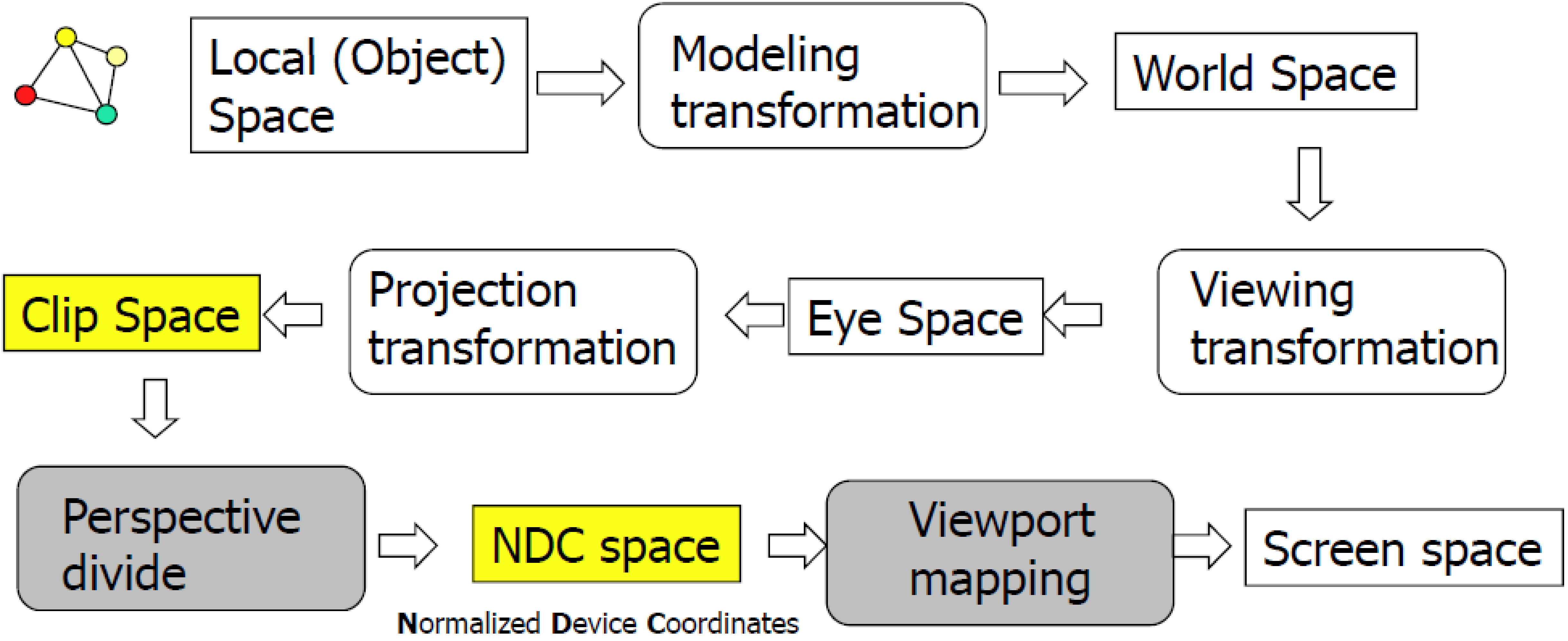


Nate_Robins_tutorials: Projection

The Golden Rule

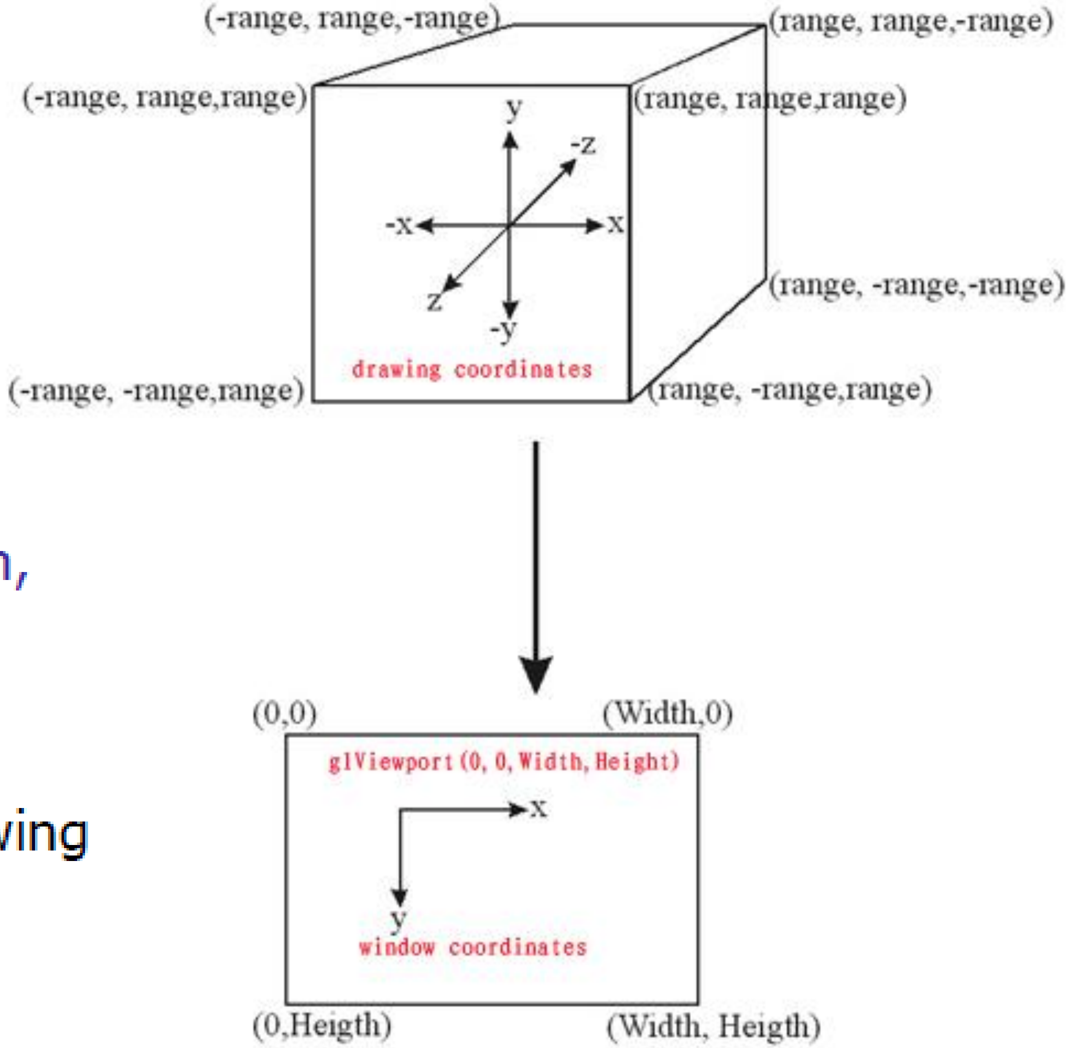
- Modeling transformation
  - glMatrixMode(GL_MODELVIEW); glRotate3f?
- Viewing transformation
  - glMatrixMode(GL_MODELVIEW); gluLookAt()
- Projection transformation
  - glMatrixMode(GL_PROJECTION);
  - glLoadIdentity - to initialise the stack.
  - **gluPerspective/glFrustum/glOrtho/gluOrtho2 - to set the appropriate projection onto the stack.**
  - You *could* use glLoadMatrix to set up your own projection matrix (if you understand the restrictions and consequences) - but I'm told that this can cause problems for some OpenGL implementations which rely on data passed to glFrustum, etc to determine the near and far clip planes.

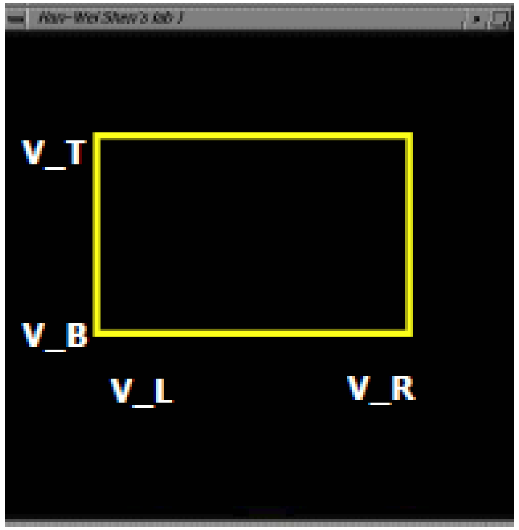
Transformation Pipeline



Viewpoint transformation

- The **viewport** maps the **drawing coordinates** to **window coordinates** and therefore defines the region of the scene, which can be seen. If the user resizes the window, we have to adjust the **viewport** and correct the aspect ratio.
-



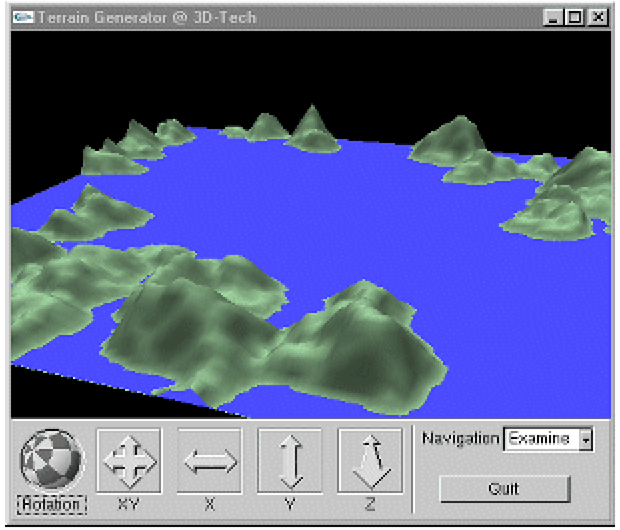



```
glViewport(int  left, int  bottom,
           int  (right-left),
           int (top-bottom));
```

call this function before drawing (calling glBegin() and glEnd())

OpenGL Terrain Generator

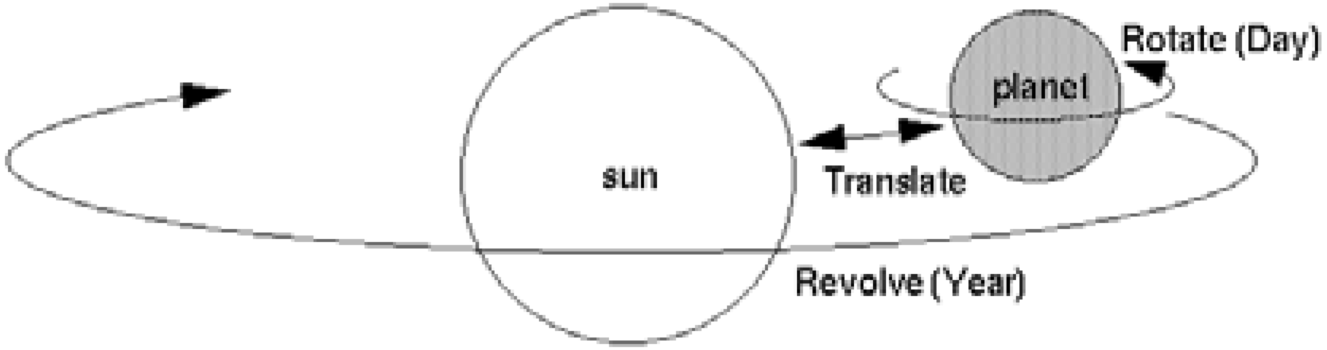
- An example of OpenGL terrain generator developed by António Ramires Fernandes can be found in: http://www.lighthouse3d.com/opengl/appstools/tg/



- Terrain generation from an image, computing normals and simulating both directional and positional lights

Laboratory Sessions

- Assignment: Building the solar system



- You will need to write from scratch a complete OpenGL programme that renders a Sun with an orbiting planet and a moon orbiting the planet

Assignment Basic Implementation

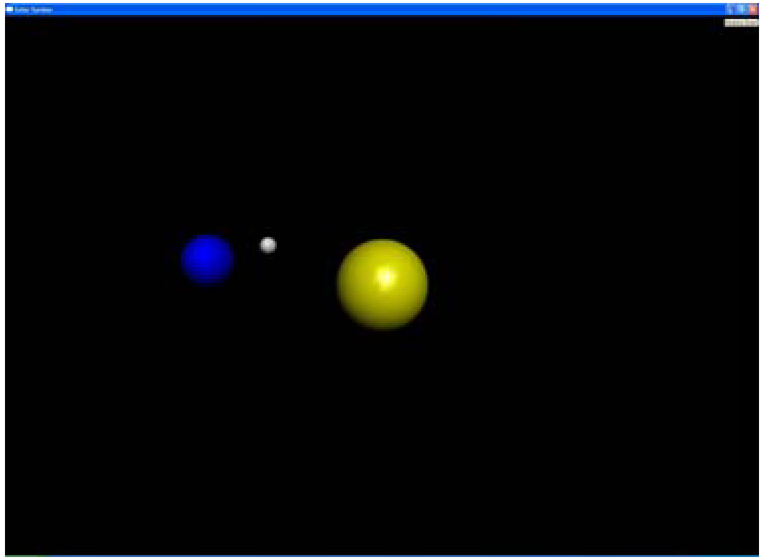
The basic implementation includes the following:

- Add a sphere representing the sun planet
- Make the sun planet to rotate around itself
- Add another sphere representing the earth
- Make the earth planet to rotate around itself
- Make the earth planet to rotate around sun
- Add another sphere representing the moon
- Make the moon planet to rotate around itself
- Make the moon planet to rotate around the earth
- Control the camera position using the keyboard
- Control the camera position using widget menus
- Add a light source
- Add shading to the planets
- Add material properties to the planets (you have to check this out
- yourselves)

Assignment Advanced Implementation

Recommended Implementation

- Add more planets, e.g. if you are quick enough you could create the complete solar system
- Add more light sources (OpenGL supports up to 8 lights)
- Have planets counter rotating
- Add more moons to planets
- Add stars to the planetary system
- Add spaceships



Summary

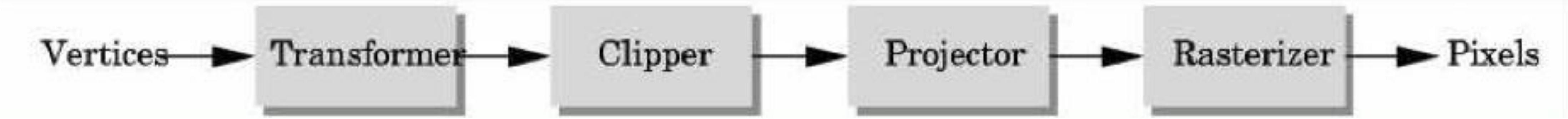
- **A Graphics Pipeline**

- The OpenGL **API**

- **Primitives**: vertices, lines, polygons

- **Attributes**: color

- Example: drawing a **shaded triangle**



Suggestions

- Most people do old OGL because they found an out of date tutorial online.
- Modern OpenGL (Shaders & VBOs [Vertex Buffer Objects])