Viewing in OpenGL

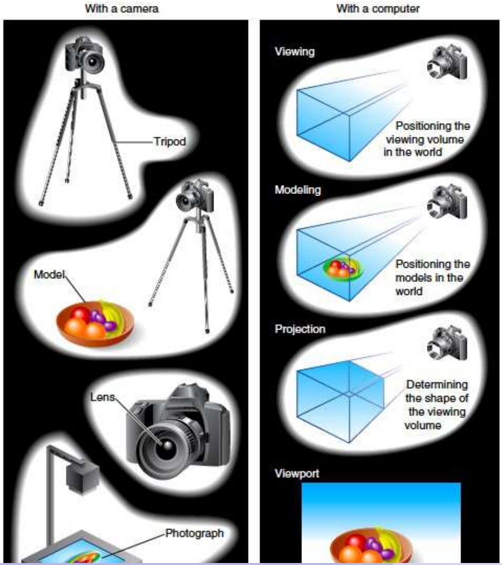
Professor Raymond Zavodnik

Fachhochschule Regensburg, Germany, May 9, 2012

Motivating Example

To take a photograph of a scene:

- Set up your tripod and point camera at the scene (Viewing Transformation)
- Position objects within the scene (Modeling Transofrmation)
- Choose an appropriate camera lens (Projection Transofrmation)
- Choose the size of the photo (Viewport Transformation


With a camera
Tripod
Model
Lens
Photograph
With a computer
Viewing
Positioning the viewing volume in the world
Modeling
Positioning the models in the world
Projection
Determining the shape of the viewing volume
Viewport

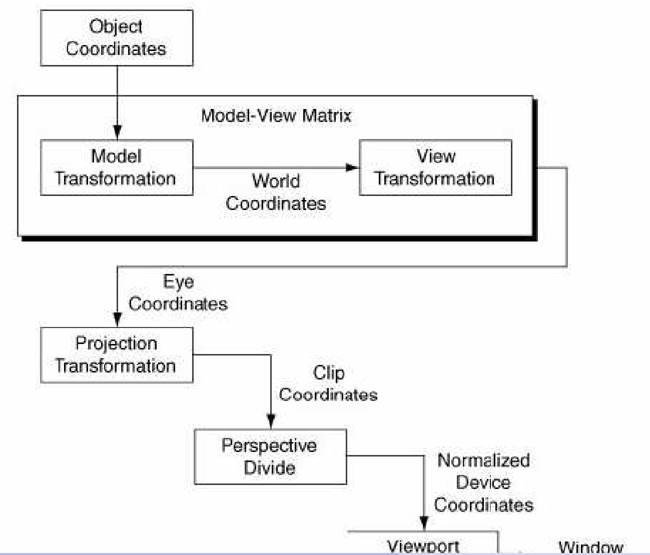
The Graphics Pipeline in OpenGL

The ModelView phase

- Start by defining an object in its own **local coordinate system**
- The place it in the scene
  - The object now has standardised scene coordinates, called **eye coordinates**
  - The camera is at the origin and pointing down the negative z-axis
  - The “up” direction is the positive y-axis
- the placement happened by a sequence of **rotations** and **translations**
- The order is important!!

Pojection

- Now project the objects in the scene onto the Projection Plane
  - For perspective it is necessary to do this in two steps
  - Map the entire space by an appropriate matrix, clipping away those portions of the objects that are outside of the viewing volume. Result is **Clipping Coordinates**
  - Do the nonlinear part (perspective divide) to get a nonlinear projection onto the projection place. Result is **Normalized Device Coordinates** or **NDC**. These coordinates are still floating point numbers.
  - Now map the resultant vertices to your computer screen ("rasterization") The resulting pixels are called **Window Coordinates**


Object Coordinates
Model-View Matrix
Model Transformation
World Coordinates
View Transformation
Eye Coordinates
Projection Transformation
Clip Coordinates
Perspective Divide
Normalized Device Coordinates
Viewport
Window

Homogeneous Coordinates

- A *3 D point* is given by the 4 dimensional column vector

$$P = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

- The last coordinate w is usually 1.0
- If $w \neq 1.0$, x, y and z in 3 D space can be recovered by dividing by w
- This is only needed for the perspective transformation

3-D Transformations

- Use 4×4 matrices stored in 1-dimensional arrays in *column order*
- This the array `GLfloat A[16]` is equivalent to the array

$$M = \begin{pmatrix} A[0] & A[4] & A[8] & A[12] \\ A[1] & A[5] & A[9] & A[13] \\ A[2] & A[6] & A[10] & A[14] \\ A[3] & A[7] & A[11] & A[15] \end{pmatrix}$$

- Transformation composition is through *post multiplication*
- The last matrix specified is the first applied

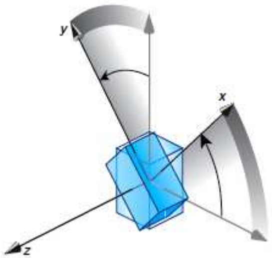
OpenGL State: Matrix Stacks

- OpenGL keeps two stacks of matrices as part of its state: Modelview matrices (with max 32 entries) and Projection (2 entries)
- A matrix state is changed when a matrix multiplies the top of the stack for that matrix type
- The matrix on top of each respective is called **Current Matrix** if there is no confusion
- Applications can push matrices onto the matrix stack, alter it and return by popping it
- OpenGL initializes matrix stacks with the identity matrix
- Thus the commands `glPushMatrix()` and `glPopMatrix()`

Matrix Procedures

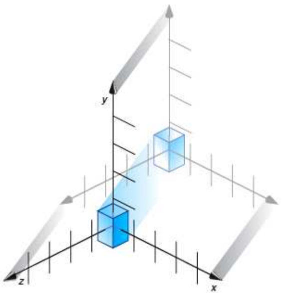
- `glLoadIdentity()`
- `glLoadMatrix const TYPE *m)`
- `glMultMatrix(const TYPE *m)`
- `glRotate[fd](TYPE theta, TYPE x, TYPE y, TYPE z)`
- `glTranslate[fd](TYPE , TYPE y, TYPE z)`
- `glScale[fd](TYPE x, TYPE y, TYPE z)`

Rotation About An Axis



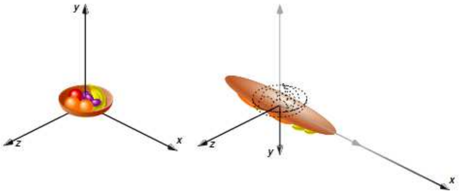
- Multiply the current matrix with this one, which rotates an object (or a local coordinate system) counterclockwise `theta` degrees about the axis defined by the point (x, y, z)
- Replace the matrix on top of the matrix stack by this result

Translation by a Vector



- Multiply the current matrix with this one, which translates an object by the vector (x, y, z)
- Replace the current matrix on top of the matrix stack by this result

Scaling an Object



- Multiply the current matrix with this one, which scales an object in each of the x, y and z direction by the respective quantities x, y, z
- Replace the current matrix on top of the matrix stack with this result
- Note that negative scalars result in reflections
- Positive scalars stretch, negative ones shrink

Properties of OpneGL Matrices

- Matrix multiplication is not commutative.
- Thus, if R is a rotation matrix and T is a translation matrix $RL \neq LR$ in general
- Consider the code

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(1.0,1.0,0.0);
glRotatef(45, 0.0,0.0,1.0);
```

- Postmultiply first a translation to the (x, y) point and *then* rotate this point 45 degrees about the z axis
- This is not the same as rotating 45 degrees about the z axis and *then* translating by $(1, 1)$

The Modelling Transformation

- The whole process starts with **Object Coordinates**
- The **Modelling Transformation** is applied to objects to give us **World Coordinates**, which are not used in OpenGL
- It main purpose is to position and orient the model
- This is usually a concatanation of rotation, translation and scaling
- Consider the program `cube.c`
- In this case we use `glScalef(1.0,2.0,1.0)`, stretching the cube in the *y*-direction by a factor of 2

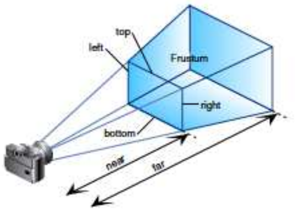
The Viewing Transformation

- This is analogous to positioning and aiming the camera
- See again `cube.c`
- Here we used `glLookat()` from the GLU library:
- `glLookat(`$\overbrace{ex, ey, ez}^{\texttt{Eye}}, \overbrace{rx, ry, rz}^{\texttt{Reference}}, \overbrace{ux, uy, uz}^{\texttt{Up}}$`)`
- Effect: Map the Reference Point to the negative z-axis, the Eyepoint Eye to the origin and the Up Vector to the positive y-axis
- The result is **Eye Coordinates**
- Consider the program `cube.c`
- Note: `glLookat()` comes before `glScale()` in the program text, although it is applied *after* the scaling

Remarks

- The net effect of both transformations is to map object coordinates into eye coordinates
- These are usually considered together and called **Modelview**
- If this needs to be specified write `glMatrixMode(GL_MODELVIEW)` to distinguish it from a projection

The Projection Transformation



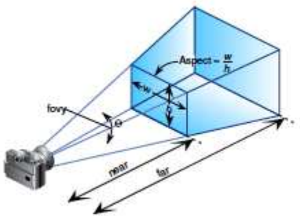
- Analogous to choosing a lens for the camera
- Can be one of two types:
  - **Orthogonal**, which induces no foreshortening
  - **Perspective**, which does
- In the program we used the perspective projection

$$\texttt{glFrustum}(\overbrace{\mathit{left}, \mathit{right}, \mathit{bottom}, \mathit{top}, \mathit{near}, \mathit{far}}^{\mathit{ClippingPlanes}})$$

The Projection Transformation

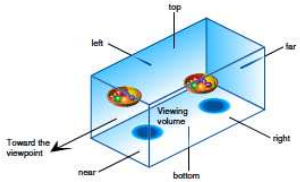
- Notice the implicit camera position

The Projective Transformation



- There is another more camera-oriented approach
  - Consider the angle *fovy* defining outgoing vertical rays from the viewer
  - Determine the **aspect ratio** of the viewing window's width to height
- We obtain: `gluPerspective(fovy, aspect, near, far)`

Orthographic Projection



- With **Orthographic** Projection a rectangular viewing volume is specified with 6 clipping planes, but there is no viewer as in perspective
- There is simply a **Direction of Projection** and a **Projection Plane**
- There is no foreshortening, so line segment lengths and angles are preserved

Orthographic Projection

- Thus `glOrtho(left,right,bottom,top,near,far)`
- The preferred projection in Cad/Cam programs
- Note the 2-D Version `glOrtho2D()`

Projection and Clipping

- Objects that are outside the Viewing Volume must be clipped, i.e. eliminated
- Objects that are partially outside, partially inside must also be clipped, but only those portions lying outside
- Objects completely within the Viewing Volume are untouched
- This process is carried during the Projective Transformaton, before the Projective Divide (thus “Clipping Coordinates”)

The Viewport Transformation

- This is where the size of the photograph is specified
- The net result of applying the Projective Transformation is a standard two dimensional region (of type `GLdouble`) (specifically $\{(u,v) \mid - \leq u, v + 1\}$) called **Normalized Device Coordinates** or NDC.
- Now object vertices in these coordinates must be mapped to **Pixels** or**Window Coordinates**, which are of type `GLint`)
- Thus

  $$\texttt{glViewport(}\overbrace{GLintx, GLinty}^{\texttt{Origin}}, overbraceGlsizeiwidth, GLsizeiheig$$
- Note If there is to be no distortion, the proprtions of the window width to height must be the same as the viewport's proportion of width to height

The Viewport Transformation

- See `cube.c`
- Thus the Perspective and Viewport transformations are reset in `reshape()`

Some General Remarks

- The Modelview Transformation: It is mathematically irrelevant whether thecamera is moved towards or away from a fixed scene or whether the scene is moved in the opposite direction
- Moving the camera 5 units up the z-axis (away from the scene) is thus `glTranslatef(0,0,-5)`, thinking the scene objects stationary
- OpenGL does not use **World Coordinates**. Thinking in terms of World Coordinates requires adjusting to OpenGL conventions, as the order of transforms is opposite to this paradigm (see above)

Hierarchical Modeling Systems

- Recall that moving the camera back 5 units is given by `glTranslatef(0,0,-5)`
- This corresponds to moving the origin to $(0, 0, 5)$, thus translates the local coordinate system by 5 units
- The command `glTranslatef(0,0,-5)` can also be thought of as applying to an object, in which case it is moved 5 units down the negative z-axis. The camera is considered stationary, i.e. at the origin
- The result is mathematically the same
- Now use this to advantage

Hierarchical Modeling Systems

- Suppose you want to make a (complex) model of a car
- Suppose you have drawn the chassis
- To draw a wheel, remember where you are and then change coordinate systems to the point where you want to affix the wheel, orienting it the way you want to draw the wheel
- Draw the wheel
- Go back to where you were and do the same for the other three wheels
- Here “remember” means note the top of the Modelview stack.
- Change coordinate systems means pushing the corresponding matrix onto the stack, to obtain a new coordinate system