3D Viewing

- How do you take a picture with a camera?
  - Set up a scene
  - Grab a camera
  - Take a snapshot
    - Final print is a 2D representation
    - Of the 3D scene
    - Taken from a given perspective

Viewing Via Camera in the Real World

- Here are images taken by a camera:

Viewing Via Camera in Computer Graphics

- Just like in photography, the camera defines what part of the scene you can see.
- Based on:
  - Projection type used by camera
  - Location of camera
  - Direction of camera
  - Orientation of camera
  - “Range” of your camera
- All of the above will (almost) define a view volume
- All objects in the view volume are seen by the camera
- Camera “range” defined by near and far clipping planes
- Your thumbs never “get in the way” in graphics

Camera Coordinates

- Camera has its own 3D coordinate system based on its orientation
  - u, v, w
  - u corresponds to x (as seen by the camera)
  - v corresponds to y (as seen by the camera)
  - w corresponds to z (as seen by the camera)
  - Negative w is into the scene
- We define camera orientation in world coordinates
  - Provide the camera location (eyepoint)
  - Indicate what direction the camera is looking (lookat)
  - Give the “up” direction of the camera
  - Then
    - w = eyepoint – lookat (normalized)
    - u = w x lookat (normalized)
    - v = w x w

The camera’s up does not have to equal v
- u, v, and w need to be perpendicular to each other
Camera Coordinates

- Default camera orientation has the camera/viewing/eye coordinate system coincident with the world axes, i.e.,
  - Eyepoint at (0, 0, 0)
  - Looking at (0, 0, -1)
  - Up vector (0,1,0) anchored at the eye
  - Then
    - \( n = \text{eyepoint} - \text{lookat} \) is (0, 0, 1) (normalized)
    - \( u = \text{up} \times n \) is (1, 0, 0) (normalized)
    - \( v = \text{uv} \times n \) is (0, 1, 0)

- Remember:
  \[ \text{m1} \times \text{o2} = (\text{m1y}\text{o2z} - \text{m1z}\text{o2y}, \text{m1z}\text{o2x} - \text{m1x}\text{o2z}, \text{m2x}\text{o2y} - \text{m1y}\text{o2x}) \]
  Equivalent to a vector perpendicular to the plane of the 2 crossed vectors
  Magnitude is equal to the area of the parallelogram formed by 2 vectors
  Cross product of 2 parallel vectors is 0

What's different here?

- The role of cameras can be described as projecting a 3D scene onto a 2D plane

Projection - Terminology

- Center of projection
  - During the projection, points in the scene will converge to a given point.
  - This point is the center of projection
- Projection or view plane
  - 2D plane upon which the 3D scene is getting projected (In OpenGL, it's the front or near clipping plane)

A Hierarchy of Projections

- Sometimes called orthographic projection.
- Objects of equal size appear the same size after being projected, regardless of the distance they are from the viewing plane.
- The Center of Projection is at infinity

Parallel Projection

- Center of projection
- View plane
- Object in 3D scene
Perspective Projection

- Sometimes called frustum projection
- Objects closer to the view plane will appear larger when projected than objects of the same size that are farther from the view point.
- The Center of Projection is at camera location
- This is the projection used by “real” cameras

Perspective Projections – Vanishing Points

- Vanishing points can appear on more than one axis
- One point is what we’ve seen
  - View plane parallel to x and y
- Add a second point (two-point):
  - View plane parallel to y axis only
- Add a third point (three-point):
  - View plane not parallel to any axis

Perspective vs. Parallel Projection

- Orthographic best for:
  - Architectural drawings where line up/same size checking essential
  - Not trying to fly through scene
- Perspective/frustum best for:
  - Realism
  - Moving through scene
  - Aligning/measuring is not an issue

View Volumes

- Comparison of orthographic and perspective view volumes:
  - Parallel (Ortho) Projection
  - Perspective (Frustum) Projection
  - Field of view (FOV)
    - Angle subtended from center of projection to top and bottom of the view plane
  - Aspect ratio
    - Width/height of view plane
Camera Coordinates

- Coordinates of objects in the 3D scene must be converted to the coordinate system of the camera
- In fact, the whole image generation process is nothing more than a series of concatenated transformations

Graphics Pipeline

- So how is all this implemented?
  - Using 4-D homogeneous matrices

\[
\begin{bmatrix}
    x_v \\
    y_v \\
    z_v \\
    1
\end{bmatrix} =
\begin{bmatrix}
    m_{11} & m_{12} & m_{13} & m_{14} & x_o \\
    m_{21} & m_{22} & m_{23} & m_{24} & y_o \\
    m_{31} & m_{32} & m_{33} & m_{34} & z_o \\
    m_{41} & m_{42} & m_{43} & m_{44} & 1
\end{bmatrix}
\]

view plane  transformation  object

Homogeneous Matrices

- Transformations are expressed as 4D matrices
  - World → Camera
  - Projection
  - Object transformations
  - Object hierarchy transformation
  - These matrices can be multiplied together to create a single composite matrix that does all the transformations in one shot

Homogeneous Matrices

- OpenGL maintains 3 kinds of matrices:
  - Modelview
    - Handles all object transformations
    - Handles world to camera transformation
  - Projection
    - Handles projection
    - Equivalent to the camera view plane
  - Viewport
    - Handles view plane to view port (2D → 2D)
- All matrix operations in OpenGL are performed on the "current" matrix as defined by the OpenGL matrix mode
- glMatrixMode() is used to select the mode
Orthographic Cameras in OpenGL

\texttt{glOrtho(left, right, bottom, top, near, far)}

- Defines a view volume for an orthographic projection
  - All values can be positive or negative
  - Default is: \texttt{glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0)}
- This defines a projection
  - It should be applied to the OpenGL Projection Matrix

\begin{verbatim}
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(left, right, bottom, top, near, far);
\end{verbatim}

Perspective Cameras in OpenGL

\texttt{glFrustum(left, right, bottom, top, near, far)}

- Defines a view volume for a perspective projection
  - Near and far are measured in the \(-z\) axis and must be positive
- Again, this is applied to the OpenGL Projection Matrix

\begin{verbatim}
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum(left, right, bottom, top, near, far);
\end{verbatim}

Perspective Cameras in OpenGL

\texttt{gluPerspective(fov, aspect, near, far)}

- Defines a view volume for an orthographic projection
  using \(\text{fov} \) & \(\text{aspect}\) ratio
  - \(\text{Near and far are measured in the } -z \text{ axis and must be positive}\)
- Again, this is applied to the OpenGL Projection Matrix

\begin{verbatim}
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(fov, aspect, near, far);
\end{verbatim}

Camera Orientation in OpenGL

\texttt{gluLookAt(eye.x, eye.y, eye.z, lookat.x, lookat.y, lookat.z, up.x, up.y, up.z)}

- Defines the correct world-to-camera transformation
- This defines the world to camera transformation
  - It must be applied to the ModelView Matrix

\begin{verbatim}
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(eye.x, eye.y, eye.z, lookat.x, lookat.y, lookat.z, up.x, up.y, up.z);
\end{verbatim}

Object Transformations

- Dividing scenes into individual objects
- Why?
  - Model each individually
  - Reuse
  - Animation
- Objects are usually defined in their own coordinate system
- Then they are “transformed” and placed in their proper place in the scene

Transformations:
- Translation (moving from one spot to another)
- Rotating (around any of the axes)
- Scaling (making the object larger or smaller in any direction)
- There are 4D homogeneous matrices defined for each of these transformation operations.
Object Transformation Example

- Successive transformations are performed by multiplying matrices for individual transformations into a single transformation matrix.

- If you want to:
  - Rotate an object about the y axis by 30 degrees, then
  - Rotate an object about the z axis by 45 degrees, then
  - Scale the object to twice its size in each dimension, then
  - Translate the object to (1, 2, 3)

- You can create the matrix:

  \[
  M = T(1,2,3) \cdot S(2,2,2) \cdot R_z(45) \cdot R_y(30)
  \]

  - Then transform each point of your object by multiplying by \( M \)

Object Transformation

\[
\begin{bmatrix}
  x_t \\
  y_t \\
  z_t \\
  1
\end{bmatrix} =
\begin{bmatrix}
  m_{11} & m_{12} & m_{13} & m_{14} \\
  m_{21} & m_{22} & m_{23} & m_{24} \\
  m_{31} & m_{32} & m_{33} & m_{34} \\
  m_{41} & m_{42} & m_{43} & m_{44}
\end{bmatrix}
\begin{bmatrix}
  x_o \\
  y_o \\
  z_o \\
  1
\end{bmatrix}
\]

Complete Matrix Application

\[
\begin{bmatrix}
  x_o \\
  y_o \\
  z_o \\
  1
\end{bmatrix} =
\begin{bmatrix}
  1 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 \\
  0 & 0 & 1 & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix}
\]

Doing This in OpenGL

- OpenGL provides functions that will apply translations, rotations, and scaling transformations to a “current” matrix.

- These are applied to the ModelView Matrix
  - They are the Model part of this matrix

Doing This in OpenGL – Example

- Task:
  - Rotate an object about the y axis by 30 degrees, then
  - Rotate an object about the z axis by 45 degrees, then
  - Scale the object to twice its size in each dimension, then
  - Translate the object to (1, 2, 3)
Doing This in OpenGL – Example

```
glMatrixMode(GL_MODELVIEW);
gluLookAt(2.0, 2.0, 2.0, 0.0, 0.0, 0.0, 1.0, 0.0);

// Note that these are in reverse!
glTranslatef(1.0, 2.0, 3.0);
glScalef(2.0, 2.0, 2.0);
glRotatef(45.0, 0.0, 0.0, 1.0);
glRotatef(30.0, 0.0, 1.0, 0.0);
// code to draw stuff
```

OpenGL Matrix Stack

- Let's say we want to add another object independent of the one we just added.
- Problem: the last object's transformation has already been applied to the ModelView matrix.
- Ideal solution: localize transformations to apply just to one object
  - After drawing the object, throw away the transformation.

OpenGL Matrix Stack Example

- Task 1:
  - Rotate object1 about the y axis by 30 degrees, then
  - Rotate object1 about the z axis by 45 degrees, then
  - Scale object1 to twice its size in each dimension, then
  - Translate object1 to (1, 2, 3)
- Task 2:
  - Translate object2 to (10, 10, 10)
So you'd like to use multiple models...

- Display lists: a group of OpenGL commands that have been stored for later execution
- Some example code we could make into a list:

```c
glColor3f(1, 1, 1);
glutSolidTeapot(0.25);
gluMatrix();
gColor3f(1, 0, 0);
guTranslatef(-1.0, -0.5, 0.0);
glutSolidTeapot(0.25);
gPopMatrix();
guTranslatef(0.0, -0.5, 0.0);
gColor3f(0, 0, 1);
glutSolidTeapot(0.25);
guAttrib();
```

```c
glNewList(TEAPOTS, GL_COMPILE);
gColor3f(1, 1, 1);
glutSolidTeapot(0.25);
guMatrix();
gColor3f(1, 0, 0);
guTranslatef(-1.0, -0.5, 0.0);
glutSolidTeapot(0.25);
gPopMatrix();
guTranslatef(GL_ALL_ATTRIB_BITS);
guTranslatef(0.0, -0.5, 0.0);
gColor3f(0, 0, 1);
glutSolidTeapot(0.25);
guAttrib();
guAttrib();
```

- Can call list with: `glCallList(TEAPOTS)`

Other Helpful Commands

- `GLint gluGenLists(GLsizei range)` will give you a new range of ID's that haven't been used
- `GLboolean glIsList(GLuint listID)` tells you if this list index is used
- `GLvoid gluDeleteLists(GLuint firstlist, GLsizei range)` deletes a range of list indices starting with firstlist
- `GLvoid gluCallList(GLsizei n,GLenum type, GLvoid *lists_indices)` used to call several lists in a row (good for lists of lists)