### Viewing in 3D

Woo, Neider et Al., Chapter 3

**OpenGL®**

### The Camera Analogy

<table>
<thead>
<tr>
<th>tripod</th>
<th>view</th>
<th>tripod</th>
<th>view</th>
<th>tripod</th>
<th>view</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position the viewing-volume in the world</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>model</td>
<td>modeling</td>
<td>model</td>
<td>modeling</td>
<td>model</td>
<td>modeling</td>
</tr>
<tr>
<td>Positioning the Models in the world</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lens</td>
<td>projection</td>
<td>lens</td>
<td>projection</td>
<td>lens</td>
<td>projection</td>
</tr>
<tr>
<td>Determining shape of viewing-volume</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>photograph</th>
<th>viewport</th>
<th>photograph</th>
<th>viewport</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Vertex Transformation Pipeline

- **Vertex coordinates**
- **Eye coordinates**
- **Clip coordinates**
- **Normalized device coordinates**
- **Window coordinates**

**Note:** Even when (x, y) coordinates are sufficient to determine which pixels are set, all transformation are performed on “z” as well.

### General Purpose Commands

- **glMatrixMode**(mode): mode being GL_MODELVIEW, GL_PROJECTION or GL_TEXTURE
- **glLoadIdentity():** Load the current transformation matrix with I
- **glLoadMatrix*(m):** Load the CTM with matrix m (m is 4x4)
- **glMultMatrix*(m):** Multiply the CTM by m (m is 4x4)

**Note:** Better to declare m as m[16], since in C m[4][4] is filled by rows and not by columns.

\[
m = \begin{pmatrix}
m_1 & m_4 & m_9 & m_{13} \\
m_2 & m_5 & m_{10} & m_{14} \\
m_3 & m_6 & m_{11} & m_{15} \\
m_4 & m_8 & m_{12} & m_{16}
\end{pmatrix}
\]

### Order of Transformation

Since the application of a transformation multiplies the matrix on top of the stack, the transformation specified most recently is applied first.

**Example:**

```c
glMatrixMode(GL_MODELVIEW);  // CTM is Modelview
glLoadIdentity();           // CTM = I

glRotatef(45.0f, 1.0f, 0.0f, 0.0f);  // CTM = I * N
glTranslatef(4.0f, 0.0f, 0.0f);     // CTM = I * N * M

glBegin(GL_POINTS);
    glVertex3f(0.0f, -2.0f, 0.0f);
    glVertex3f(0.0f, 2.0f, 0.0f);
    glVertex3f(4.0f, 0.0f, 0.0f);
    glVertex3f(-4.0f, 0.0f, 0.0f);
glEnd();
```

### Meaning of Transformations

**The most intuitive way:** Moving a local coordinate-system.

A local coordinate-system is tied to the object. All operations occur relative to this changing coordinate-system.

**Example:**

```c
glMatrixMode(GL_MODELVIEW);
gLoadIdentity();
draw_flowerpot();

glTranslatef(4.0f, 0.0f, 0.0f);  // CTM = I * N

glTranslatef(4.0f, 0.0f, 0.0f);  // CTM = I * N * M

glScalef(0.5f, 0.5f, 0.5f);      // CTM = I * N * M * L

draw_flowerpot();
```
Manipulating the Matrix Stacks

A stack of matrices is useful for constructing hierarchical models, in which complicated objects are constructed from simpler ones.

Example – Stack Usage

```c
void gluLookAt(GLdouble ex, GLdouble ey, GLdouble ez,
               GLdouble cx, GLdouble cy, GLdouble cz,
               GLdouble ux, GLdouble uy, GLdouble uz);
```

Projections

- The projection transformation is analogous to the camera lens. It determines the field of view and the viewing volume.
- OpenGL projections can be perspective or orthographic.
- Before any projection transformation command:
  ```c
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  ```
- NOTE: In the OpenGL perspective parameters, near and far are always positive.
**Perspective Projection**

void glFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);

- `glFrustum()` is not very intuitive, so GLUT provides a simpler interface:
  
  void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble near, GLdouble far);

**Aspect Ratio**

The aspect-ratio of a viewport should generally be equal to the aspect-ratio of the viewing-volume. If the 2 ratios are different, the projected image will be distorted.

**Orthographic Projection**

void glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);

**Troubleshooting Transformations**

- Make sure the objects are drawn with a color which is different from the background.
- Remember that “near” & “far” measure distance from the viewpoint, and that (by default) you are looking down the negative Z, so if near=1 and far=3, objects Z must be [-1, .. , -3] to be visible.
- Determine where the viewpoint is, in which direction you are looking, and where your objects are (try by creating a real 3D space).
- Make sure you know the rotation axes.

**Additional Clipping Planes**

void glClipPlane(Glenum plane, const GLdouble *equation);

- Defines an extra clipping plane.
- The parameter `plane` is `GL_CLIP_PLANExi`.
- Clipping can be enabled/disabled with `glEnable(GL_CLIP_PLANExi));
- The parameter `equation` points to the 4 coefficients of the plane-equation: $Ax + By + Cz + D = 0$. 
Vertex Arrays

Standard OpenGL drawing:
• Requires many function calls to render geometric primitives
• Redundant processing of shared vertices

Example: The cube has 6 faces and 8 shared vertices. Using the standard method, each vertex is specified 3 times, so 24 vertices would be processed, though 8 would be enough

Cube: 6 sides, 8 shared vertices

Steps for using vertex arrays:
• Enable (up to 6) arrays, each to store a different type of data (vertices, colors, normals, edge flags, etc.)
• Specify data for the array(s)
• Draw geometry with the data. OpenGL obtains the data from all activated arrays by dereferencing the pointers

NOTE: Arrays are stored in the client

Enabling Arrays

void glEnableClientState(GLenum array)

Specifies the array to enable. GL_VERTEX_ARRAY, GL_COLOR_ARRAY, GL_NORMAL_ARRAY, etc., are all acceptable parameters

When using lighting, we will usually do:

   glEnableClientState(GL_NORMAL_ARRAY);
   glEnableClientState(GL_VERTEX_ARRAY);

If we turn off lighting:

   glDisableClientState(GL_NORMAL_ARRAY);

To specify data: one function for each array type:
glVertexPointer, glColorPointer, glIndexPointer, glNormalPointer, glTexCoordPointer, glEdgeFlagPointer

Example:

   void glVertexPointer(GLint size, GLenum type, GLsizei stride, const GLvoid *pointer);

   Pointer: memory address of the first coordinate of the first vertex

   Type: GL_SHORT, GL_INT, GL_FLOAT or GL_DOUBLE

   Size: the number of coordinates per-vertex

   Stride: byte offset between consecutive vertexes

Example: enabling arrays and loading data

   static GLint vertices[ ] = {25, 25, 100, 325, 175, 25, 175, 325, 250, 25, 325, 325};
   static GLfloat colors[ ] = {1.0, 0.2, 0.2, 0.2, 0.2, 1.0, 0.8, 1.0, 0.2, 0.75, 0.75, 0.75, 0.35, 0.35, 0.35, 0.5, 0.5, 0.5};

   glEnableClientState(GL_COLOR_ARRAY);
   glEnableClientState(GL_VERTEX_ARRAY);
   glColorPointer(3, GL_FLOAT, 0, colors);
   glVertexPointer(2, GL_INT, 0, vertices);

Basic ways of accessing array data:

• Single array-element:
  void glArrayElement( ... )

• List of array-elements
  void glDrawElements( ... )

• Sequence of array elements
  void glDrawArrays( ... )
**Vertex Arrays**

```c
void glArrayElement(GLint ith)
```

Dereferences an array element to obtain vertex data from all currently enabled arrays.

**Example:**

```c
glBegin(GL_TRIANGLES);
glArrayElement(2);
glArrayElement(3);
glArrayElement(5);
glEnd();
```

**Same effect (with less function calls) as:**

```c
glBegin(GL_TRIANGLES);
COLORfv(colors+(2*3*sizeof(GLfloat));
Vertexfv(vertices+(2*2*sizeof(GLint));
COLORfv(colors+(3*3*sizeof(GLfloat));
Vertexfv(vertices+(3*2*sizeof(GLint));
COLORfv(colors+(5*3*sizeof(GLfloat));
Vertexfv(vertices+(5*2*sizeof(GLint));
glEnd();
```

**Vertex Arrays**

```c
void glDrawElements(GLenum mode,
                    GLsizei count, GLenum type,
                    void *indices)
```

Dereferences a list of array elements in order to define a sequence of geometric primitives.

**Mode:** geometric primitive

**Count:** number of elements

**Type:** data type of indices

**Indices:** indices of the elements in the arrays

**Equivalent to (with less function calls):**

```c
glBegin(mode);
for (int i = 0; i < count; i++)
glArrayElement(indices[i]);
glEnd();
```

**Vertex Arrays**

```c
static Glubyte frontIndices = {4, 5, 6, 7};
static Glubyte rightIndices = {1, 2, 6, 5};
static Glubyte bottomIndices = {0, 1, 5, 4};
static Glubyte backIndices = {0, 3, 2, 1};
static Glubyte leftIndices = {0, 4, 7, 3};
static Glubyte topIndices = {2, 3, 7, 6};
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, frontIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, rightIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, bottomIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, backIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, leftIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, topIndices);
```

**Or, better, crunch all indices together:**

```c
static Glubyte allIndices = {4, 5, 6, 7, 1, 2, 6, 5, 0, 1, 5, 4, 0, 3, 2, 1, 0, 4, 7, 3, 2, 3, 7, 6};
glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, allIndices);
```

**Vertex Arrays**

```c
void glDrawArrays(GLenum mode, GLint first,
                  GLsizei count);
```

Constructs a sequence of geometric primitives of type **mode** by using array elements that start at **first** and end at **first**+**count**-1 of each enabled array.

**Effect similar to:**

```c
glBegin(mode);
for (int i = 0; i < count; i++)
glArrayElement(first + i);
glEnd();
```

**Extruded Shapes**

A large class of 3D shapes can be generated by **extruding** or **sweeping**, a 2D shape through space.

**Example:** Creating prisms

Let the prism’s base be a polygon P with N vertices \((x_i, y_i)\)

We number the base vertices 0, …, N-1, and those of the cap N, …, 2N-1. An edge joins \(v(i)\) and \(v(i+n)\)

**Vertex-list:** \((x_i, y_i, 0)\) and \((x_i, y_i, H)\) for \(i = 1, …, N-1\)

**Face-list:** First we make the walls. The \(j\)-th wall, \(j=0, …, N-1\) is:

\[
\text{v}(j), \text{v}(j+N), \text{v}(\text{next}(j) + N), \text{v}(%(j))
\]

Where: \(\text{next}(j) = (j+1) \text{ modulo } N\)

Finally we add “base” & “cap” polygons to complete the list...