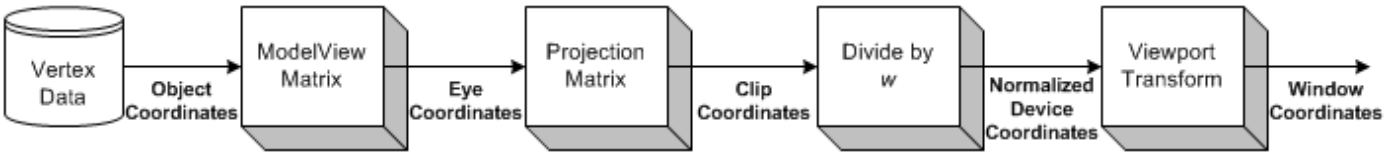
←Back

OpenGL Transformation

Related Topics: OpenGL Pipeline, OpenGL Projection Matrix, OpenGL Matrix Class
Download: matrixModelView.zip, matrixProjection.zip

Overview

Geometric data such as vertex positions and normal vectors are transformed via **Vertex Operation** and **Primitive Assembly** operation in OpenGL pipeline before raterization process.



OpenGL vertex transformation

Object Coordinates

It is the local coordinate system of objects and is initial position and orientation of objects before any transform is applied. In order to transform objects, use glRotatef(), glTranslatef(), glScalef().

Eye Coordinates

It is yielded by multiplying GL_MODELVIEW matrix and object coordinates. Objects are transformed from object space to eye space using GL_MODELVIEW matrix in OpenGL. **GL_MODELVIEW** matrix is a combination of Model and View matrices ($M_{\mathbf{view}} \cdot M_{\mathbf{model}}$). Model transform is to convert from object space to world space. And, View transform is to convert from world space to eye space.

$$\begin{pmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{pmatrix} = M_{modelView} \cdot \begin{pmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ w_{obj} \end{pmatrix} = M_{view} \cdot M_{model} \cdot \begin{pmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ w_{obj} \end{pmatrix}$$

Note that there is no separate camera (view) matrix in OpenGL. Therefore, in order to simulate transforming the camera or view, the scene (3D objects and lights) must be transformed with the inverse of the view transformation. In other words, OpenGL defines that the camera is always located at (0, 0, 0) and facing to -Z axis in the eye space coordinates, and cannot be transformed. *See more details of GL_MODELVIEW matrix in ModelView Matrix.*

Normal vectors are also transformed from object coordinates to eye coordinates for lighting calculation. Note that normals are transformed in different way as vertices do. It is mutiplying the tranpose of the inverse of GL_MODELVIEW matrix by a normal vector. *See more details in Normal Vector Transformation.*

$$\begin{pmatrix} nx_{eye} \\ ny_{eye} \\ nz_{eye} \\ nw_{eye} \end{pmatrix} = \left({M_{modelView}}^{-1} \right)^T \cdot \begin{pmatrix} nx_{obj} \\ ny_{obj} \\ nz_{obj} \\ nw_{obj} \end{pmatrix}$$

Clip Coordinates

The eye coordinates are now multiplied with **GL_PROJECTION** matrix, and become the clip coordinates. This GL_PROJECTION matrix defines the viewing volume (frustum); how the vertex data are projected onto the screen (perspective or orthogonal). The reason it is called *clip coordinates* is that the transformed vertex (x, y, z) is clipped by comparing with ±w.
See more details of GL_PROJECTION matrix in Projection Matrix.

$$\begin{pmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{pmatrix} = M_{projection} \cdot \begin{pmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{pmatrix}$$

Normalized Device Coordinates (NDC)

It is yielded by dividing the clip coordinates by *w*. It is called *perspective division*. It is more like window (screen) coordinates, but has not been translated and scaled to screen pixels yet. The range of values is now normalized from -1 to 1 in all 3 axes.

$$\begin{pmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \end{pmatrix} = \begin{pmatrix} x_{clip}/w_{clip} \\ y_{clip}/w_{clip} \\ z_{clip}/w_{clip} \end{pmatrix}$$

Window Coordinates (Screen Coordinates)

It is yielded by applying normalized device coordinates (NDC) to viewport transformation. The NDC are scaled and translated in order to fit into the rendering screen. The window coordinates finally are passed to the raterization process of OpenGL pipeline to become a fragment. **glViewport()** command is used to define the rectangle of the rendering area where the final image is mapped. And, **glDepthRange()** is used to determine the *z* value of the window coordinates. The window coordinates are computed with the given parameters of the above 2 functions;
glViewport(x, y, w, h);
glDepthRange(n, f);

$$\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} \frac{\mathrm{w}}{2} x_{ndc} + (\mathrm{x} + \frac{\mathrm{w}}{2}) \\ \frac{\mathrm{h}}{2} y_{ndc} + (\mathrm{y} + \frac{\mathrm{h}}{2}) \\ \frac{\mathrm{f}-\mathrm{n}}{2} z_{ndc} + \frac{\mathrm{f}+\mathrm{n}}{2} \end{pmatrix}$$

The viewport transform formula is simply acquired by the linear relationship between NDC and the window coordinates;

$$\begin{cases} -1 & \to \mathrm{x} \\ 1 & \to \mathrm{x}+\mathrm{w} \end{cases} \quad \begin{cases} -1 & \to \mathrm{y} \\ 1 & \to \mathrm{y}+\mathrm{h} \end{cases} \quad \begin{cases} -1 & \to \mathrm{n} \\ 1 & \to \mathrm{f} \end{cases}$$

OpenGL Transformation Matrix

$$\begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix}$$

OpenGL Transform Matrix

OpenGL uses 4 x 4 matrix for transformations. Notice that 16 elements in the matrix are stored as 1D array in column-major order. You need to transpose this matrix if you want to convert it to the standard convention, row-major format.

OpenGL has 4 different types of matrices; **GL_MODELVIEW**, **GL_PROJECTION**, **GL_TEXTURE**, and **GL_COLOR**. You can switch the current type by using **glMatrixMode()** in your code. For example, in order to select GL_MODELVIEW matrix, use **glMatrixMode(GL_MODELVIEW)**.

Model-View Matrix (GL_MODELVIEW)

GL_MODELVIEW matrix combines viewing matrix and modeling matrix into one matrix. In order to transform the view (camera), you need to move whole scene with the inverse transformation. **gluLookAt()** is particularly used to set viewing transform.

The 3 matrix elements of the rightmost column (m_{12}, m_{13}, m_{14}) are for the translation transformation, **glTranslatef()**. The element m_{15} is the homogeneous coordinate. It is specially used for projective transformation.

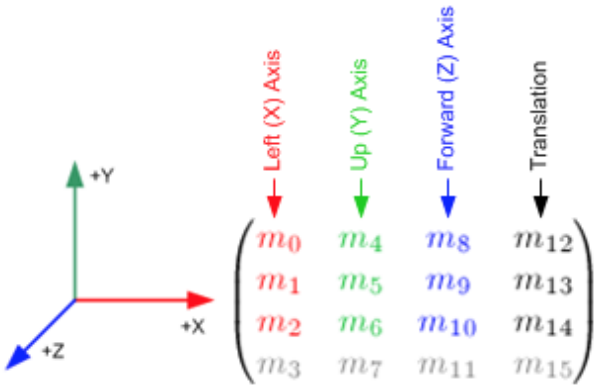
3 elements sets, (m_0, m_1, m_2), (m_4, m_5, m_6) and (m_8, m_9, m_{10}) are for Euclidean and affine transformation, such as rotation **glRotatef()** or scaling **glScalef()**. Note that these 3 sets are actually representing 3 orthogonal axes;

- (m_0, m_1, m_2) : +X axis, *left* vector, (1, 0, 0) by default
- (m_4, m_5, m_6) : +Y axis, *up* vector, (0, 1, 0) by default

- (m_8, m_9, m_{10}) : +Z axis, *forward* vector, (0, 0, 1) by default

We can directly construct GL_MODELVIEW matrix from angles or lookat vector without using OpenGL transform functions. Here are some useful codes to build GL_MODELVIEW matrix:

- Angles to Axes
- Lookat to Axes
- Matrix4 class



4 columns of GL_MODELVIEW matrix

Note that OpenGL performs matrices multiplications in reverse order if multiple transforms are applied to a vertex. For example, If a vertex is transformed by M_A first, and transformed by M_B second, then OpenGL performs M_B x M_A first before multiplying the vertex. So, the last transform comes first and the first transform occurs last in your code.

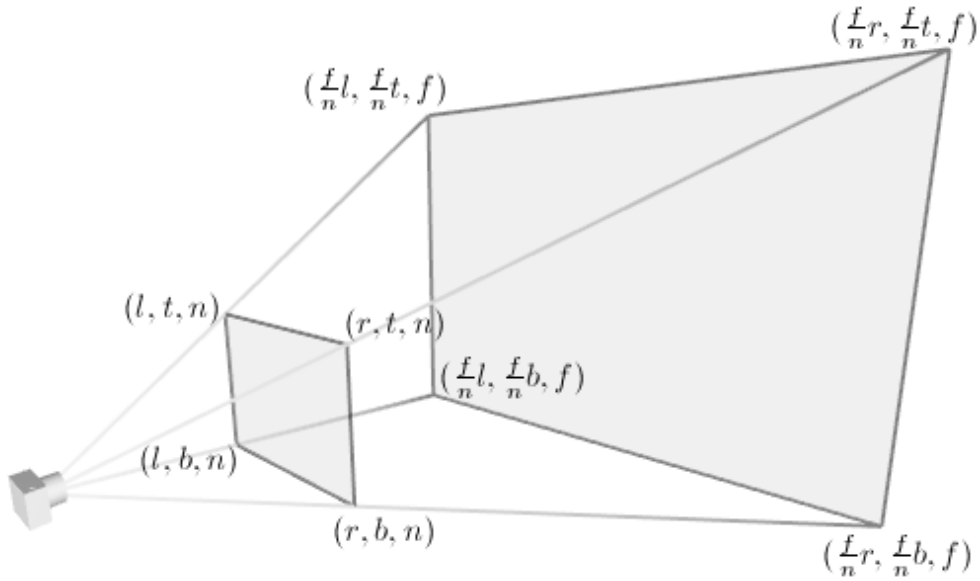
$$v' = M_B \cdot (M_A \cdot v) = (M_B \cdot M_A) \cdot v$$

```
// Note that the object will be translated first then rotated
glRotatef(angle, 1, 0, 0);   // rotate object angle degree around X-axis
glTranslatef(x, y, z);       // move object to (x, y, z)
drawObject();
```

Projection Matrix (GL_PROJECTION)

GL_PROJECTION matrix is used to define the frustum. This frustum determines which objects or portions of objects will be clipped out. Also, it determines how the 3D scene is projected onto the screen. *(Please see more details how to construct the projection matrix.)*

OpenGL provides 2 functions for GL_PROJECTION transformation. **glFrustum()** is to produce a perspective projection, and **glOrtho()** is to produce a orthographic (parallel) projection. Both functions require 6 parameters to specify 6 clipping planes; *left*, *right*, *bottom*, *top*, *near* and *far* planes. 8 vertices of the viewing frustum are shown in the following image.



OpenGL Perspective Viewing Frustum

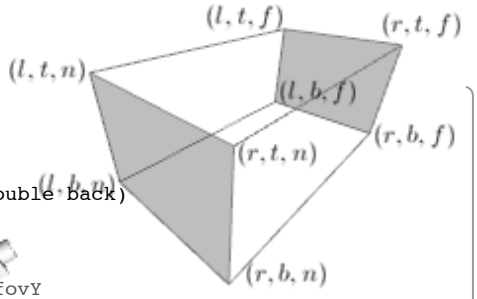
The vertices of the far (back) plane can be simply calculated by the ratio of similar triangles, for example, the left of the far plane is;

$$\frac{far}{near} = \frac{left_{far}}{left}, \quad left_{far} = \frac{far}{near} \cdot left$$

For orthographic projection, this ratio will be 1, so the *left*, *right*, *bottom* and *top* values of the far plane will be same as on the near plane.

You may also use gluPerspective() and gluOrtho2D() functions with less number of parameters. **gluPerspective()** requires only 4 parameters; vertical field of view (FOV), the aspect ratio of width to height and the distances to near

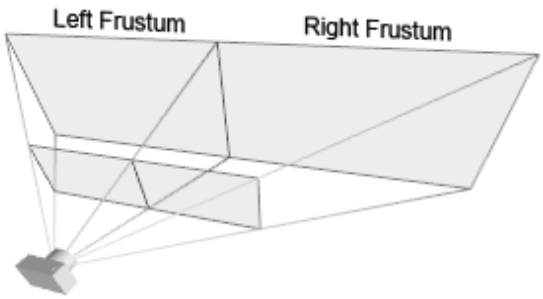
and far clipping planes. The equivalent conversion from gluPerspective() to glFrustum() is described in the following code.



OpenGL Orthographic Frustum

```
// This creates a symmetric frustum.
// It converts to 6 params (l, r, b, t, n, f) for glFrustum()
// from given 4 params (fovy, aspect, near, far)
void makeFrustum(double fovY, double aspectRatio, double front, double back)
{
    const double DEG2RAD = 3.14159265 / 180;

    double tangent = tan(fovY/2 * DEG2RAD);   // tangent of half fovY
    double height = front * tangent;          // half height of near plane
    double width = height * aspectRatio;      // half width of near plane

    // params: left, right, bottom, top, near, far
    glFrustum(-width, width, -height, height, front, back);
}
```

However, you have to use glFrustum() directly if you need to create a non-symmetrical viewing volume. For example, if you want to render a wide scene into 2 adjoining screens, you can break down the frustum into 2 asymmetric frustums (left and right). Then, render the scene with each frustum.



An example of an asymmetric frustum

Texture Matrix (GL_TEXTURE)

Texture coordinates (*s*, *t*, *r*, *q*) are multiplied by GL_TEXTURE matrix before any texture mapping. By default it is the identity, so texture will be mapped to objects exactly where you assigned the texture coordinates. By modifying GL_TEXTURE, you can slide, rotate, stretch, and shrink the texture.

```
// rotate texture around X-axis
glMatrixMode(GL_TEXTURE);
glRotatef(angle, 1, 0, 0);
```

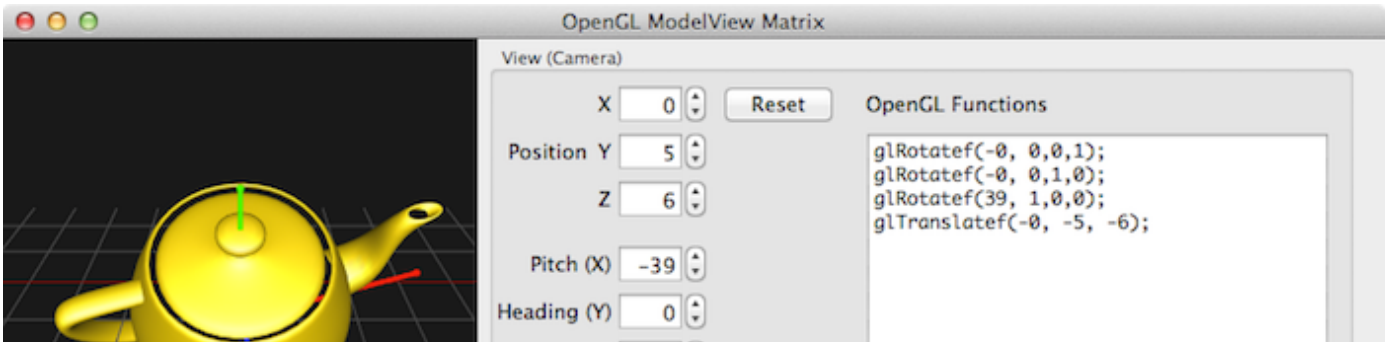
Color Matrix (GL_COLOR)

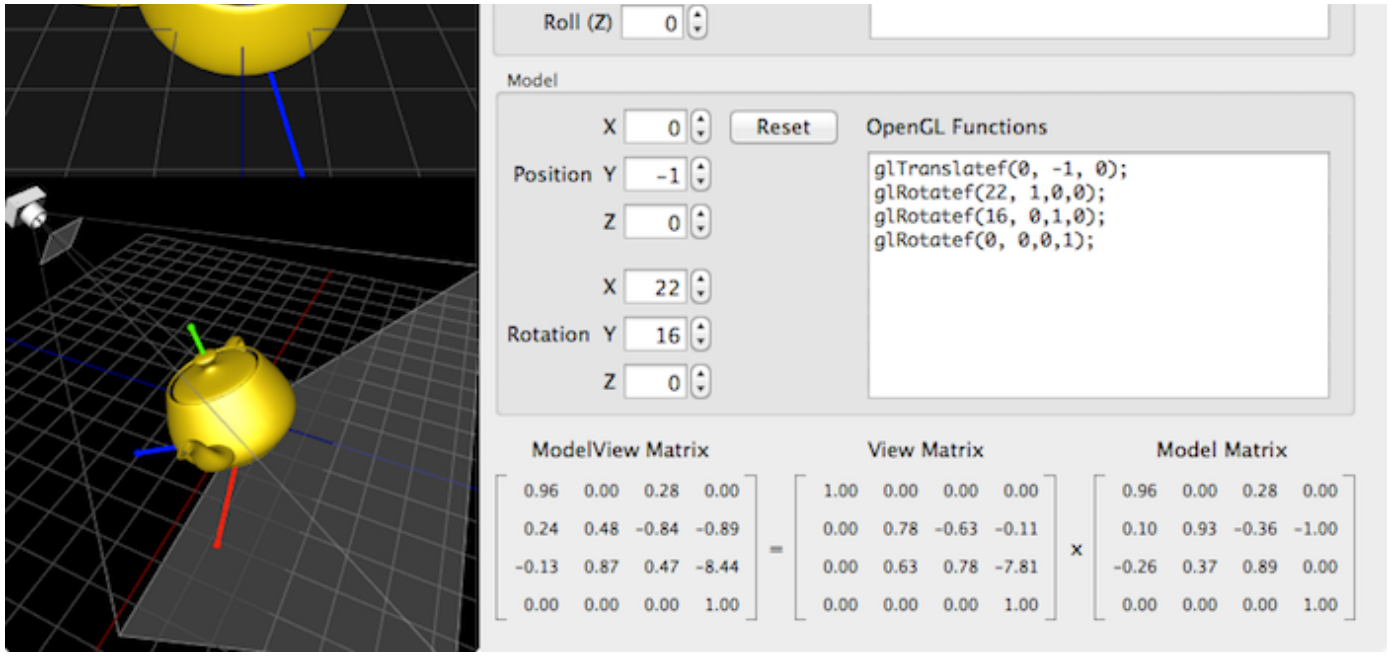
The color components (*r*, *g*, *b*, *a*) are multiplied by GL_COLOR matrix. It can be used for color space conversion and color component swaping. GL_COLOR matrix is not commonly used and is required **GL_ARB_imaging** extension.

Other Matrix Routines

glPushMatrix() : push the current matrix into the current matrix stack.
glPopMatrix() : pop the current matrix from the current matrix stack.
glLoadIdentity() : set the current matrix to the identity matrix.
glLoadMatrix{fd}(*m*) : replace the current matrix with the matrix *m*.
glLoadTransposeMatrix{fd}(*m*) : replace the current matrix with the row-major ordered matrix *m*.
glMultMatrix{fd}(*m*) : multiply the current matrix by the matrix *m*, and update the result to the current matrix.
glMultTransposeMatrix{fd}(*m*) :
multiply the current matrix by the row-major ordered matrix *m*, and update the result to the current matrix.
glGetFloatv(GL_MODELVIEW_MATRIX, *m*) : return 16 values of GL_MODELVIEW matrix to *m*.

Example: ModelView Matrix





This demo application shows how to manipulate GL_MODELVIEW matrix with glTranslatef() and glRotatef().

Download the source and binary:
(Updated: 2013-05-20)

matrixModelView.zip

matrixModelView_mac.zip *(OS X 10.6+)*

Note that all OpenGL function calls are implemented in *ModelGL.h* and *ModelGL.cpp* on both Mac and Windows versions, and these files are *identical* on both packages.

This demo application uses a custom 4x4 matrix class as well as default OpenGL matrix routines in order to specify model and camera transforms. There are 3 of matrix objects defined in ModelGL.cpp; matrixModel, matrixView and matrixModelView. Each matrix stores the pre-computed transformation and passes the matrix elements to OpenGL by using **glLoadMatrixf()**. The actual drawing routine looks like;

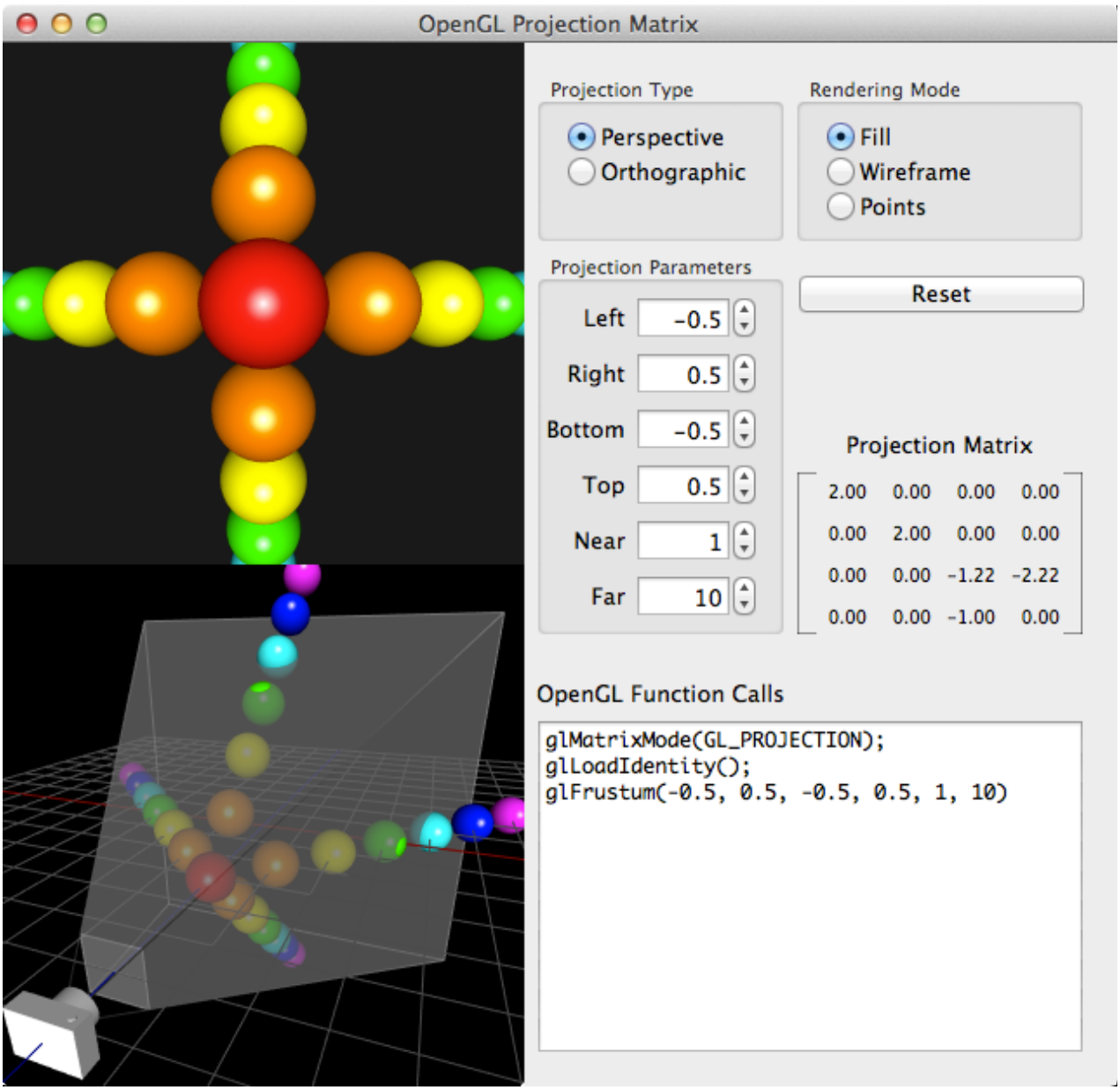
```
...
glPushMatrix();

// set view matrix for camera transform
glLoadMatrixf(matrixView.getTranspose());

// draw the grid at origin before model transform
drawGrid();

// set modelview matrix for both model and view transform
// It transforms from object space to eye space.
glLoadMatrixf(matrixModelView.getTranspose());

// draw a teapot after both view and model transforms
drawTeapot();

glPopMatrix();
...
```

The equivalent code using default OpenGL matrix functions is;

```
...
glPushMatrix();

// initialze ModelView matrix
glLoadIdentity();

// First, transform the camera (viewing matrix) from world space to eye space
// Notice all values are negated, because we move the whole scene with the
// inverse of camera transform
glRotatef(-cameraAngle[2], 0, 0, 1); // roll
glRotatef(-cameraAngle[1], 0, 1, 0); // heading
glRotatef(-cameraAngle[0], 1, 0, 0); // pitch
```

```
glTranslatef(-cameraPosition[0], -cameraPosition[1], -cameraPosition[2]);

// draw the grid at origin before model transform
drawGrid();

// transform the object (model matrix)
// The result of GL_MODELVIEW matrix will be:
// ModelView_M = View_M * Model_M
glTranslatef(modelPosition[0], modelPosition[1], modelPosition[2]);
glRotatef(modelAngle[0], 1, 0, 0);
glRotatef(modelAngle[1], 0, 1, 0);
glRotatef(modelAngle[2], 0, 0, 1);

// draw a teapot with model and view transform together
drawTeapot();

glPopMatrix();
...
```

Example: Projection Matrix



This demo application is to show how to manipulate the projection transformation with glFrustum() or glOrtho().

Download the source and binary:
(Updated: 2013-03-17)

matrixProjection.zip

matrixProjection_mac.zip *(OS X 10.6+)*

Again, *ModelGL.h* and *ModelGL.cpp* are exactly same files on both packages, and all OpenGL function calls are placed in these files.

ModelGL class has a custom matrix object, *matrixProjection*, and 2 member functions, *setFrustum()* and *setOrthoFrustum()*, which are equivalent to *glFrustum()* and *glOrtho()*.

```
///////////////////////////////////////////////////////////////////////////////
// set a perspective frustum with 6 params similar to glFrustum()
// (left, right, bottom, top, near, far)
// Note: this is for row-major notation. OpenGL needs transpose it
///////////////////////////////////////////////////////////////////////////////
void ModelGL::setFrustum(float l, float r, float b, float t, float n, float f)
{
    matrixProjection.identity();
    matrixProjection[0]  = 2 * n / (r - l);
    matrixProjection[2]  = (r + l) / (r - l);
    matrixProjection[5]  = 2 * n / (t - b);
    matrixProjection[6]  = (t + b) / (t - b);
    matrixProjection[10] = -(f + n) / (f - n);
    matrixProjection[11] = -(2 * f * n) / (f - n);
    matrixProjection[14] = -1;
    matrixProjection[15] = 0;
}

///////////////////////////////////////////////////////////////////////////////
// set a orthographic frustum with 6 params similar to glOrtho()
// (left, right, bottom, top, near, far)
// Note: this is for row-major notation. OpenGL needs transpose it
///////////////////////////////////////////////////////////////////////////////
void ModelGL::setOrthoFrustum(float l, float r, float b, float t, float n,
                              float f)
{
    matrixProjection.identity();
    matrixProjection[0]  = 2 / (r - l);
    matrixProjection[3]  = -(r + l) / (r - l);
    matrixProjection[5]  = 2 / (t - b);
    matrixProjection[7]  = -(t + b) / (t - b);
    matrixProjection[10] = -2 / (f - n);
    matrixProjection[11] = -(f + n) / (f - n);
}
...

// pass projection matrx to OpenGL before draw
glMatrixMode(GL_PROJECTION);
glLoadMatrixf(matrixProjection.getTranspose());
...
```

Constructing 16 elements of GL_PROJECTION matrix is explained here.

←Back