Week 5 – 3D transformations, Views and Quaternions

This week we move to 3D Geometric transformations and views. We do have to work through some math and 3D view details to make the jump from 2D to 3D.

3D Geometric Transformations

The rotation matrixes for each of the axis in 4x4 form are:

\[
R(x) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\]

\[
R(y) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\]

\[
R(z) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\]

If you perform the matrix multiplication with vector \([x \ y \ z \ 1]\) by each of these you will obtain the rotated positions as follows:

**R(x):**
\[x' = x;\]
\[y' = y \cos(\theta) - z \sin(\theta);\]
\[z' = y \sin(\theta) + z \cos(\theta);\]

**R(y):**
\[x' = z \sin(\theta) + x \cos(\theta);\]
\[y' = y;\]
\[z' = z \cos(\theta) - x \sin(\theta);\]

**R(z):**
\[x' = x \cos(\theta) - y \sin(\theta);\]
\[y' = x \sin(\theta) + y \cos(\theta)\]
\[z' = z;\]

The translation matrix is used to translate across the viewing area as well as translate an object back to the coordinate axis prior to rotation or scaling.
Translation with x, y and z components become:

\[
T(x,y,z) = \begin{bmatrix}
1 & 0 & 0 & tx \\
0 & 1 & 0 & ty \\
0 & 0 & 1 & tz \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Scale with x, y and z components become:

\[
S(x,y,z) = \begin{bmatrix}
sx & 0 & 0 & 0 \\
0 & sy & 0 & 0 \\
0 & 0 & sz & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Matrix math for 4x4 equations work similarly to 3x3 equations. You just have one row and column to take into account. For example, to determine the translation of -10, 20 and 30 for x, y and z components respectively, for a starting point of (100, 50, 20) results in this calculation:

Translation of (-10,20,30)

\[
\begin{bmatrix}
1 & 0 & 0 & -10 \\
0 & 1 & 0 & 20 \\
0 & 0 & 1 & 30 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
100 \\
50 \\
20 \\
1
\end{bmatrix} =
\begin{bmatrix}
90 \\
70 \\
50 \\
0
\end{bmatrix}
\]

If we take the output from this translation and rotation of \(\pi/2\) about the z-axis the calculations become:

Note: \(\sin(\pi/2) = 1\) and the \(\cos(\pi/2) = 0\)

\[
\begin{bmatrix}
0 & -1 & 0 & 0 \\
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
90 \\
70 \\
50 \\
0
\end{bmatrix} =
\begin{bmatrix}
-70 \\
90 \\
50 \\
0
\end{bmatrix}
\]

If we take the output from this rotation and apply a scale of 2 for the x component, 1.5 for the y and 3 for the z components the calculations become:

\[
\begin{bmatrix}
2 & 0 & 0 & 0 \\
0 & 1.5 & 0 & 0 \\
0 & 0 & 3 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
-70 \\
90 \\
50 \\
0
\end{bmatrix} =
\begin{bmatrix}
-140 \\
135 \\
150 \\
0
\end{bmatrix}
\]

Combining the 3 transformations into a single composite matrix results in:
Just as within the 3x3 matrix math, composite matrices can be formed that comprise all steps in a series of geometric transformations.

**Quaternions**

Quaternions become interesting for 3D rotations as they can reduce the amount of math computations required in a 4x4 matrix. Although the math for a quaternion can look a bit intimidating at first think of the complex components as representing rotations about the x, y and z axis. The cosine and sine parts of a quaternion represent the scalar and complex components of a quaternion. To better visualize this and the relationship to the rotation about each axis consider the following equation:

\[
q = \cos(\Theta/2) + i (x \cdot \sin(\Theta/2)) + j (y \cdot \sin(\Theta/2)) + k (z \cdot \sin(\Theta/2))
\]

Here the i, j and k complex terms represent the rotations for a given angle \( \Theta \) for a vector \( x, y, z \). In addition \( x, y \) and \( z \) should be a unit vector such that the magnitude of the vector = 1 (e.g. \( (1,0,0), (0,1,0), \) \((0,0,1)\)). These represent the x, y and z axis.

Note that a quaternion has the general form of:

\[
q = a + i b + j c + k d;
\]

Once you have determined the quaternion you can use it to determine the resulting points for rotation, scale and even translation geometric transformations. Here is short example using quaternions to determine the \( P' \) based on an input point \( P \) and the quaternion \( (q) \) and inverse of the quaternion \( (q^{-1}) \).

Note: for larger, more complex (and time-consuming) problems, the approach is identical.

To help with this example, recall your complex math operations

\[
i^2 = j^2 = k^2 = -1;
ixj = -jxi = k;
jk = -kxj = i;
kxi = -ixk = j;
\]

To further clarify these complex components, \( ixj = k \) and \( jxi = -k \). Also \( jk = i \) and \( kxi = -i \); Finally, \( kxi = j \) and \( ixk = -j \). These properties are critical in helping to reduce the complexity of the algebraic expressions when you expand the equations.
Another key to success is taking your time. One small math error can propagate and cause results to not be correct. Take your time with the math and the use the properties above to simplify the equations.

**Example: Calculate the Quaternion for a 90-degree z-axis rotation and P'.**

Given a Point P= (1,0,0) and a z-rotation of 90 degrees calculate the quaternion and P'.

P = (1,0,0);
\( \Theta = \pi/2 \) radians (90 degrees);

**Step1 : calculate q for a 90 degree rotation (counterclockwise) about the z-axis**

90 degrees = \( \pi/2 \) in radians. Since it is a z-rotation both i and j components are zero and \( z = 1 \).

\[
q = \cos(\Theta/2) + i (x \cdot \sin(\Theta/2)) + j (y \cdot \sin(\Theta/2)) + k (z \cdot \sin(\Theta/2));
\]

// Substitute \( \pi/2 \) for \( \Theta \)
\[
q = \cos(\pi/4) + i (0 \cdot \sin(\pi/4)) + j (0 \cdot \sin(\pi/4)) + k (1 \cdot \sin(\pi/4));
\]

// Solve
\[
q = 0.7071 + i 0 + j 0 + k 0.7071;
\]

// Simplify
\[
q = 0.7071 + k 0.7071;
\]

**Step 2: Determine P' using P' = q P q';**

Note that \( q^{-1} \) is \((s,-v)\) which just means keep the scalar component as is and negate the complex component. Also, a point \( P = (1,0,0) \) can be represented as \( 0 + i 1 + j 0 + k 0 \) since \( x,y,z \) represent \( i,j \) and \( k \) respectively.

So in our case:

\[
q^{-1} = 0.7071 - k 0.7071;
\]
\[
P = 0 + i 1 + j 0 + k 0 = i;
\]

//Substituting into the formula
\[
P' = q P q';
\]
\[
P' = (0.7071 + k 0.7071) (i) (0.7071 - k 0.7071);
\]
\[
P' = (i 0.7071 + (k x i) 0.7071) (0.7071 - k 0.7071);
\]
\[
P' = (i 0.7071 + j 0.7071) (0.7071 - k 0.7071);
\]
\[
P' = i 0.5 + j 0.5 + j 0.5 - i 0.5;
\]
\[
P' = j 1 = (0, 1, 0);
\]

**Scaling and translation with Quaternions**

We can scale and translate using quaternions and the following equations:

\[
P'(scale) = \text{scalar} \times P;
\]
\[ P'(\text{translate}) = P + q; \]

**Example: Use quaternions to Calculate the \( P' \) for a point \( P=(2,3,4) \) for a translation of 3 in the x direction, -2 in the y-direction and 20 in the z-direction.**

Solving we first determine \( q, P \) in quaternion space
\[
q (3,-2,20) = 0 + i 3 - j 2 + k 20;
\]
\[
P(2,3,4) = 0 + i 2 + j 3 + k 4;
\]

Substituting and solving:
\[
P' = P + q = (0 + i 2 + j 3 + k 4) + (0 + i 3 - j 2 + k 20) = 0 + i 5 + j 1 + k 24;
\]
\[
P' = (5,1,24);
\]

**Example: Use quaternions to Calculate the \( P' \) for a point \( P=(2,3,4) \) for a scale of 7**

Solving we first determine \( q, P \) in quaternion space
\[
q (2,1,4) = 0 + i 3 - j 2 + k 20;
\]

Substituting and solving:
\[
P' = s P = 7 * (0 + i 2 + j 3 + k 4) = 0 + i 14 + j 21 + k 28;
\]
\[
P' = (14,21,28);
\]

Combining multiple geometric transformation using quaternions is also possible. For example, performing a rotation of 90 degrees in the z direction and then 90 degrees in the y direction is accomplished using the compound approach:

For the first rotation:
\[
P' = q_1 P q_1^{-1};
\]

For the Second rotation:
\[
P' = q_2 (q_1 P q_1^{-1}) q_2^{-1};
\]

One of your homework problems this week will be to apply compound geometric transformations using quaternions.
Viewing 3D Objects with OpenGL

Processes used for viewing 3D objects are similar to those we have used for viewing 2D objects such as the use of clipping and windowing regions. However; projections are needed with 3D objects to allow viewing of the scene on a planar surface. In addition, lighting effects and surface characteristics may be needed to provide more realistic views.

To view a 3D world coordinate scene, we set up camera parameters. These parameters define the position and orientation for the viewing plane. Objects are then projected onto the viewing plane. Perspective projection can then be used by projecting the points to the view plane along converging paths. One advantage of using Perspective projection is objects further from the viewing position are displayed smaller than those closer.

Depth information is critical in 3D scenes as this allows the viewer to differentiate the front from the back an object. Light plays an important part in providing depth information. OpenGL has different lighting and shading models that allow the front side (the closer side) to have a strong intensity than the back side.

To better view the perspective projection, consider the following figure adapted from this resource: http://www.gamedev.net/page/resources/_/technical/directx-and-xna/directx-11-c-game-camera-r2978.
The viewing frustum includes all possible viewing from the eye position to the far plane. The view plane (or projection screen) represents the camera’s position. This diagram will be useful when you review the gluPerspective() parameters in OpenGL as each parameter is set to optimize the projection of the 3D scene onto the projection screen. For example, gluPerspective(45.0f, aspect, 0.1f, 100.0f); sets the view angle from the bottom-plane to the top-plane, aspect ratio (width/height), near clip and far clip plane values.

An OpenGL code example of a 3D scene and the resulting perspective projection may best illustrate the theory and applied results. For a simple example, we can create a 3D wireframe of a sphere using the OpenGL function glutWireSphere(); This function takes 3 parameters include the radius of the sphere and the number of longitude and latitude subdivisions. For 3D objects, use of the model view enables the creation of unit scales objects which we can create in model view and translate to World coordinates. This process allows for the creation of complex and detailed 3D scenes. We can use glTranslatef() to translate from model coordinates to world coordinates as illustrated in the figure below:
In addition, we can use scale and even rotate OpenGL to further enhance the view. The following code uses the GL_MODELVIEW, clears the matrix operations with glLoadIdentity() and creates a WireSphere, scaled by 50% and translates into World coordinates with move of 1.5 to the right (x coordinate) 1.5 up (y coordinate) and 7 out (z coordinate). The glutSwapBuffers() call provides us with double buffering which will come in handy once we start performing animations.

```c
void displayfcn() {
    // Clear color and depth buffers
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // Model view matrix mode
    glMatrixMode(GL_MODELVIEW);

    // Reset the model-view matrix
    // This is done each time you need to clear the matrix operations
    // This is similar to glPopMatrix()
    glLoadIdentity();

    // Set color to green
    glColor3f(0.0f, 1.0f, 0.0f);
    // Since we are in ModelView we need to move to World coordinates
    // Move x to right by 1.5, y down by -1.5 and z out by -7
    glTranslatef(1.5f, -1.5f, -7.0f);
    // Scale x,y and z by 50%
    glScalef(0.5f, 0.5f, 0.5f);
    // Add a wireSpere mesh with 1.5 radius and 15x15 gridline
    glutWireSphere(1.5, 15, 15);

    // Double buffering
    glutSwapBuffers();
}
```
In the windows reshape function, the Projection mode is needed as we will set a viewport encompassing the entire width and height of projection screen. The gluPerspective() function is called to provide optimal viewing of the 3D scene.

```c
// Windows redraw function
void winReshapeFcn(GLsizei width, GLsizei height) {
    // Compute aspect ratio of the new window
    GLfloat aspect = (GLfloat)width / (GLfloat)height;
    // Set the viewport
    // Allows for resizing without major display issues
    glViewport(0, 0, width, height);
    // Set the aspect ratio of the clipping volume to match the viewport
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    // Enable perspective projection with fovy, aspect, zNear and zFar
    // This is the camera view and objects align with view frustrum
    gluPerspective(45.0f, aspect, 0.1f, 100.0f);
}
```

The init() function changes slightly to enable OpenGL Depth perception and appropriate shading models to enable smooth color transition.

```c
void init() {
    // Get and display your OpenGL version
    const GLubyte *Vstr;
    Vstr = glGetString(GL_VERSION);
    fprintf(stderr, "Your OpenGL version is %s\n", Vstr);
    // Set background color to black
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    // Maximize depth display
    glClearDepth(1.0f);
    // Enable depth testing for z-culling
    glEnable(GL_DEPTH_TEST);
    // Set the type of depth-test
    glDepthFunc(GL_LEQUAL);
    // Provide smooth shading
    glShadeModel(GL_SMOOTH);
}
```

The main() method is similar to previous examples, but notice the additional of the GLUT_DOUBLE display mode.

```c
void main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE);
    // Set the window's initial width & height
    glutInitWindowSize(winWidth, winHeight);
    // Position the window's initial top-left corner
    glutInitWindowPosition(50, 50);
    // Create window with the given title
```
glutCreateWindow(title);
// Display Function
glutDisplayFunc(displayfcn);
// Reshape function
glutReshapeFunc(winReshapeFcn);
// Initialize
init();
// Process Loop
glutMainLoop();
}

Running the code found in CMSC405Week5_1.cpp results in the following graphic:

Experimenting with the glTranslatef() function provides for fine tuning of the viewing plane. For example, if change the x and y parameters to -1.5f and 1.5f respectively (e.g. glTranslatef(-1.5f, 1.5f, -7.0f)), the following graphic will result:
Similarly, modifying the z parameter causes additional viewing plane changes. If we increased the value of z from -7.0f to -4.0f, the following graphic results:

Notice moving the projection screen closer to the object results in a larger display but also, we begin to lose the image from the scene. Adjustments to the x and y parameters may be needed. If we set the z value further then the object appears smaller. The following screen shows the results when z was set to -20.f.
The following illustrates adding additional OpenGL 3D shapes and using the glTranslatef(), glScalef() and glRotatef() functions. The code adds different color shapes at different locations, scales and rotations.

```c
void displayfcn() {
    // Clear color and depth buffers
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // Model view matrix mode
    glMatrixMode(GL_MODELVIEW);

    // Reset the model-view matrix
    // This is done each time you need to clear the matrix operations
    // This is similar to glPopMatrix()
    glLoadIdentity();

    // Set color to green
    glColor3f(0.0f, 1.0f, 0.0f);
    // Since we are in ModelView we need to move to World coordinates
    // Move x to right by 1.5, y down by -1.5 and z out by -7
    glTranslatef(1.5f, -1.5f, -7.0f);
    // Scale x, y and z by 50%
    glScalef(0.5f, 0.5f, 0.5f);
    // Add a wireSphere mesh with 1.5 radius and 15x15 gridline
    glutWireSphere(1.5, 15, 15);

    // Set color to Yellow
    glColor3f(1.0f, 1.0f, 0.0f);
    glLoadIdentity();
    // Move x to right by 1.5, y up by 1.5 and z out
    glTranslatef(1.5f, 1.5f, -7.0f);
    glScalef(0.5f, 0.5f, 0.5f);
    // Add a SolidSphere mesh
    glutSolidSphere(1.5, 15, 15);
}
```
glColor3f(0.0f, 0.0f, 1.0f);
// Reload the identify matrix
glLoadIdentity();
// Translate and add a Solidcube:
glTranslatef(-1.5f, 1.5f, -6.0f);
glScalef(0.5f, 0.5f, 0.5f);
glutSolidCube(1.5);

glColor3f(1.0f, 0.0f, 0.0f);
// Reload the identify matrix
glLoadIdentity();
// Translate and add a wirecube:
glTranslatef(-1.5f, -1.5f, -6.0f);
glScalef(0.5f, 0.5f, 0.5f);
glutWireCube(1.5);

// Add a Cyan wire teapot
glColor3f(0.0f, 1.0f, 1.0f);
// Translate to center
GLfloat Center = 0.0f, 0.0f, -6.0f;
glScalef(0.5f, 0.5f, 0.5f);
// Rotate 45 degree steps across z,y and x.
glRotatef(45.0f, 0, 0, 1.0);
glRotatef(45.0f, 0, 1.0, 0.0);
glRotatef(45.0f, 1.0, 0.0, 0.0);
glutWireTeapot(1.5);

// Reset Matrix and then translate and scale a Cube
glLoadIdentity();
glTranslatef(-2.5f, -0.0f, -6.0f);
glScalef(0.25f, 0.25f, 0.25f);
glRotatef(45.0f, 1.0f, 0.0f, 0.0f);
// Cube - 6 sides different colors
glBegin(GL_QUADS);
// Top Green
GLfloat Topgreen = 1.0f, 1.0f, 1.0f;
glVertex3f(Topgreen, Topgreen, -1.0f);
glVertex3f(-Topgreen, Topgreen, -1.0f);
glVertex3f(-Topgreen, -Topgreen, -1.0f);
glVertex3f(Topgreen, -Topgreen, -1.0f);
// Bottom Orange
GLfloat Bottomorange = 1.0f, 0.5f, 0.0f;
glVertex3f(Bottomorange, Bottomorange, 1.0f);
glVertex3f(-Bottomorange, Bottomorange, 1.0f);
glVertex3f(-Bottomorange, -Bottomorange, 1.0f);
glVertex3f(Bottomorange, -Bottomorange, 1.0f);
// Front Red
GLfloat Frontred = 1.0f, 0.0f, 0.0f;
glVertex3f(Frontred, Frontred, Frontred);
glVertex3f(-Frontred, Frontred, Frontred);
glVertex3f(-Frontred, -Frontred, Frontred);
glVertex3f(Frontred, -Frontred, Frontred);
// Back Yellow
GLfloat Backyellow = 1.0f, 1.0f, 0.0f;
glVertex3f(Backyellow, Backyellow, Backyellow);
glVertex3f(-Backyellow, Backyellow, Backyellow);
glVertex3f(-Backyellow, -Backyellow, Backyellow);
glVertex3f(Backyellow, -Backyellow, Backyellow);
glVertex3f(1.0f, -1.0f, -1.0f);
glVertex3f(-1.0f, -1.0f, -1.0f);
glVertex3f(-1.0f, 1.0f, -1.0f);
glVertex3f(1.0f, 1.0f, -1.0f);

// Left Blue
glColor3f(0.0f, 0.0f, 1.0f);
glVertex3f(-1.0f, 1.0f, 1.0f);
glVertex3f(-1.0f, 1.0f, -1.0f);
glVertex3f(-1.0f, -1.0f, 1.0f);
glVertex3f(-1.0f, -1.0f, -1.0f);

// Right Cyan
glColor3f(1.0f, 1.0f, 0.0f);
glVertex3f(1.0f, 1.0f, -1.0f);
glVertex3f(1.0f, 1.0f, 1.0f);
glVertex3f(1.0f, -1.0f, 1.0f);
glVertex3f(1.0f, -1.0f, -1.0f);
glEnd();

// Double buffering
glutSwapBuffers();
}

Running the code found in CMSC405Week5_2.cpp will result in this 3D scene.

![3D Scene]

Experimenting and analyzing with the code examples from this week is critical to understanding how to set-up and project 3D scenes onto a plane. Your homework for this week will also give you practice creating your own unique 3D scene.