Nicholas Lloyd
Math 2270

OpenGL Matrix/Vector Manipulation using SFML

Open Graphics Library (hereafter referred to as OpenGL) is a graphics library primarily used to compute polygons on a computer graphics unit. It allows for parallelization of simple matrix computations, freeing up precious processor power by offloading to the GPU. This project is a look at how those manipulations are done, and how linear algebra is used to make these computations happen. Code examples will be given in C++, using the Simple and Fast Multimedia Library for OpenGL abstraction. Below is a representation of the graphics pipeline process to better understand how OpenGL works.

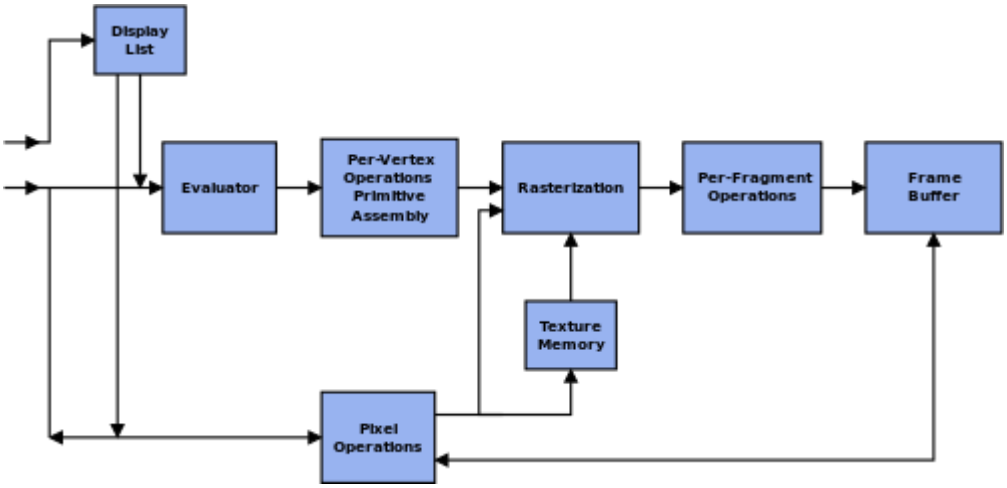

Figure 1 - via Wikipedia's OpenGL entry

At a base level, OpenGL holds vectors which define polygons in three dimensions. It also holds textures, which are pixel mappings for colors. Textures allow for polygons to have images or colors placed on them. OpenGL holds the Polygons in a Display List, and when called, can evaluate these objects and perform linear algebra operations on them through “Per-Vortex Operations”. Using SFML, polygons are defined using Shapes, which hold vectors for each point

in the polygon. For simplicities sake, all examples will be given in two dimension. When defining a polygon, it is important to lay out the polygon with no convex points. The reason for this is that it complicates the calculations to have convex polygon points. The following bit of code shows how randomized, non-convex points can be entered into OpenGL in a clockwise manner.

```
std::vector<b2Vec2> localPoly = polygonPoints.toStdVector();

// Create highs and lows that will be overwritten on the first round.
        // Find the greatest and least x & y coordinates
        int xLow = Math::inf;
        int xHigh = Math::inf;
        int yLow = Math::inf;
        int yHigh = Math::inf;
        for (int i = 0; i < polygonPoints.size(); i++) {
           localPoly[i] = polygonPoints.at(i);
           if (polygonPoints.at(i).x < xLow) xLow =
polygonPoints.at(i).x;
           if (polygonPoints.at(i).x > xHigh) xHigh =
polygonPoints.at(i).x;
           if (polygonPoints.at(i).y < yLow) yLow =
polygonPoints.at(i).y;
           if (polygonPoints.at(i).y > yHigh) yHigh =
polygonPoints.at(i).y;
        }

        // Now find the global origin of the local object
        int xMid = (xHigh + xLow)/2;
        int yMid = (yHigh + yLow)/2;

        // Use the global difference to the origin to calculate local
coordinates
        for (int i = 0; i < localPoly.size(); i++) {
           localPoly[i].x = localPoly[i].x - xMid;
           localPoly[i].y = localPoly[i].y - yMid;
        }

        // Make the points clockwise
        int n = localPoly.size();
        int d;
        int i1 = 1;
        int i2 = n - 1;
        std::vector<b2Vec2> clockwisePoly(n);
        b2Vec2 C;
        b2Vec2 D;
        for (int k = 0; k < localPoly.size(); k++) {
           for (int j = k+1; j < localPoly.size(); j++) {
              if (localPoly[j].x < localPoly[k].x) {
                 b2Vec2 temp = localPoly[k];
                 localPoly[k] = localPoly[j];
                 localPoly[j] = temp;
              }
           }
        }
clockwisePoly[0] = localPoly[0];
        C = localPoly[0];
        D = localPoly[n-1];

        for (int i = 1; i < n - 1; i++) {
           d =
determinant(C.x,C.y,D.x,D.y,localPoly[i].x,localPoly[i].y);
           if (d<0) {
              clockwisePoly[i1++]=localPoly[i];
           } else {
              clockwisePoly[i2--]=localPoly[i];
           }
        }
        clockwisePoly[i1] = localPoly[n-1];

        std::vector<b2Vec2> points(clockwisePoly.size());
        for (int i = 0; i < clockwisePoly.size(); i++) {
           points[i] = b2Vec2(clockwisePoly[i].x/SCALE,-
clockwisePoly[i].y/SCALE);
        }
        std::vector<b2Vec2> holdPoints(clockwisePoly);
        b2Vec2 *arrayPoints = &points[0];
        b2PolygonShape shape;
        shape.Set(arrayPoints,points.size());
        b2Vec2 center =
b2Vec2(shape.m_centroid.x*SCALE,shape.m_centroid.y*(-SCALE));
        center.x = center.x + xMid;
        center.y = -center.y + yMid;

        for (int i = 0; i < clockwisePoly.size(); i++) {
           qDebug() << "Point" << i << "is:" << clockwisePoly[i].x <<
clockwisePoly[i].y;
        }

        b2Vec2 *holdingPoints = &holdPoints[0];
```

The above code was written by Nicholas Lloyd using C++, SFML,

and Box2D

The above example will now be used to make a few points about the calculations going on. First, polygons are defined in their own local coordinate system, then translated by OpenGL into a single “world” if you will with its own coordinate system. Several translations are actually going on here, as computer displays are calibrated to layout from the top left of the screen, making them a coordinate system existing solely in quadrant four of the two dimensional plane. On the other hand, OpenGL generally origins from the center. This is so that manipulations can be done consistently throughout a space that is not being viewed from the same angle or depth all the time. Therefore, the points above are being translated into their own coordinate system, then entered into a polygon.

The other important thing to note about the above example is the way the points are being placed into clockwise order. The following helper method might help make better sense of this:

```
int determinant(int x1, int y1, int x2, int y2, int x3, int y3) {
        return x1*y2+x2*y3+x3*y1-y1*x2-y2*x3-y3*x1;
}
```

As seen, the determinant of a pairwise set of vectors is being calculated to evaluate its position relative to the group. Now that the set of vectors has been placed in a clockwise ordering, we can enter it into OpenGL as a defined polygon.

Now that polygons have been defined in the OpenGL world system, any way to view them must be defined as its own “projection” matrix. The diagram below shows the ordering of steps.

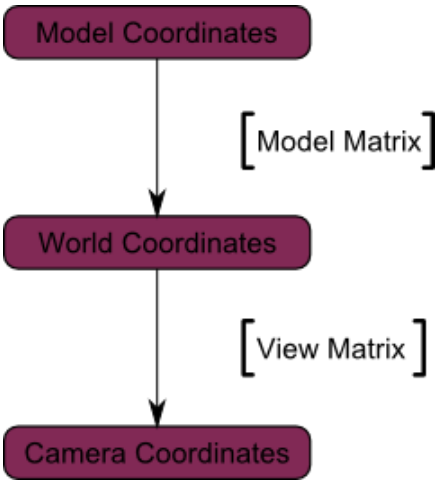

Figure 2 - http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/

The position is being defined relative to a “Camera” or “Viewer” coordinate system. This is done so that the view can be altered using a simple transformation of all points within the system.

The two following graphics will help to demonstrate this.

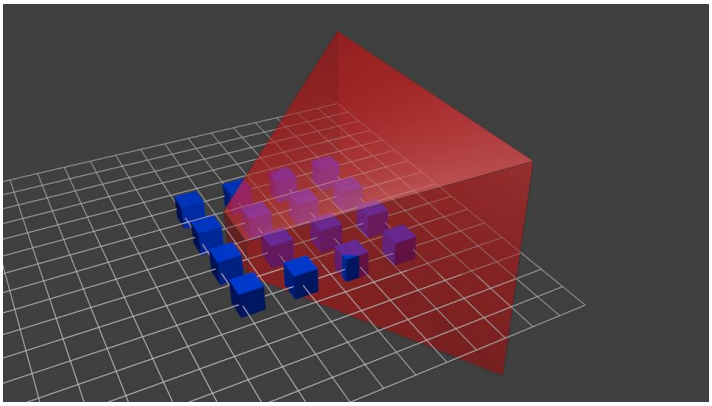

Figure 3 - http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/

This is the “camera” relative to defined objects

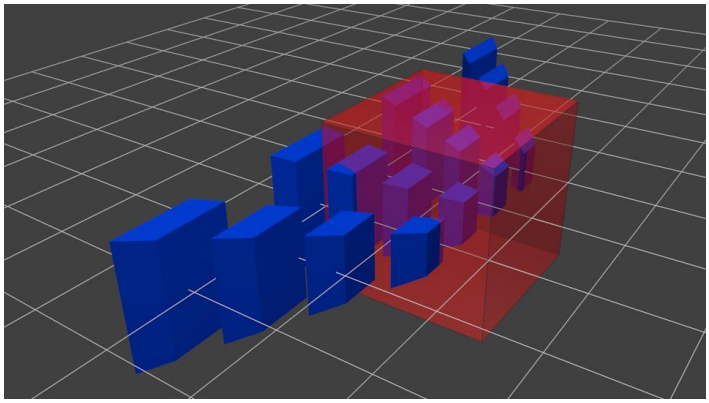

Figure 4 - http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/

This is after the objects are multiplied by the Projection Matrix

The following matrices are the orthographic projection matrix, and the perspective matrix.

$$\begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \frac{2\cdot near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2\cdot near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2\cdot far\cdot near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Next, it is important to be able to manipulate objects in the world. All manipulations to polygons are done point by point in a parallel system on the graphics card using a transformation Matrix. This is shown below:

$$\begin{bmatrix} 1 & 0 & 0 & X \\ 0 & 1 & 0 & Y \\ 0 & 0 & 1 & Z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + X \cdot 1 \\ y + Y \cdot 1 \\ z + Z \cdot 1 \\ 1 \end{bmatrix}$$

As shown, this must be done using a four by four translation matrix. Further, if one wishes to scale an object, it is done in a similar fashion, using a different translation matrix.

$$\begin{bmatrix} SX & 0 & 0 & 0 \\ 0 & SY & 0 & 0 \\ 0 & 0 & SZ & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} SX \cdot x \\ SY \cdot y \\ SZ \cdot z \\ 1 \end{bmatrix}$$

And lastly, rotation is done using a transformation around an origin. The below rotation demonstrates a rotation around the X-axis.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ cos\theta \cdot y - sin\theta \cdot z \\ sin\theta \cdot y + cos\theta \cdot z \\ 1 \end{bmatrix}$$

These transformations in OpenGL are accomplished using the OpenGL Math library GLM. This is a far from complete list of linear algebra operation performed

Sources:

OpenGL Matrices tutorial. (n.d.). Retrieved from http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/

Transformations. (n.d.). Retrieved from https://open.gl/transformations

OpenGL 101: Matrices - projection, view, model. (n.d.). Retrieved from https://solarianprogrammer.com/2013/05/22/opengl-101-matrices-projection-view-model/