Ex. No. 8	**3D Transformations**	**Karthik D**
October 19, 2022	**UCS1712 - Graphics and Multimedia Lab**	**195001047**

Aim

To develop a C++ program using the OpenGL framework to implement the 3D transformation algorithms, and demonstrate all its output cases.

Question

To apply the following 2D transformations on objects and to render the final output along with the original object:

1. Translation
2. Rotation
3. Scaling

Note: Use Homogeneous coordinate representations and matrix multiplication to perform transformations. Divide the output window into four quadrants. (Use LINES primitive to draw the x, y and z axes).

3D Transformation Algorithms

// assume transformations for a triangle
Procedure plot3DTransformations(x1, x2, x3, y1, y2, y3, z1, z2, z3);

var
tx, ty, theta, xr, yr, xref, yref: integer;
shx, shy, sx, sy: float;
(accept parameters from user)

Begin

triangle_mat := [[x1, x2, x3], [y1, y2, y3], [z1, z2, z3], [1, 1, 1]]

// translation
translation_mat := [[1 0 0 tx], [0, 1, 0, ty], [0, 1, 0, tz], [0, 0, 0, 1]]
translated_triangle := translation_mat * triangle_mat

// rotation
rotationz_mat := [[cos(theta) -sin(theta) 0, 0], [sin(theta), cos(theta), 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]]

rotationy_mat := [[1, 0, 0, 0], [0, cos(theta), -sin(theta), 0], [0, sin(theta), cos(theta) 0], [0, 0, 0, 1]]
rotationx_mat := [[cos(theta), 0, sin(theta), 0], [0, 1, 0, 0], [-sin(theta), 0, cos(theta), 0], [0, 0, 1]]
rotated_triangle := rotationz_mat * rotationy_mat * rotationx_mat * triangle_mat

// scaling
scaling_mat := [[sx 0 0 0], [0, sy, 0, 0], [0, 0, sz, 0], [0, 0, 0, 1]]
scaled_triangle := scaling_mat * triangle_mat

End {plot3DTransformations}

Implementation using C++ Program Code

1. main.cpp - Driver and Handler to render all 2D tranformations

```
#include <GL/glut.h>
#include <stdio.h>
#include <math.h>

#define PI 3.141592654


float** multiplyMatrices(float **m1, float **m2, int r1, int c1, int c2)
{
   // assume compatible matrices
   float **res = (float**)malloc(sizeof(float*)*r1);
   for(int i=0; i<r1; i++) {
       *(res+i) = (float*)malloc(sizeof(float)*c2);
       for(int j=0; j<c2; j++) {
           res[i][j] = 0;
           for(int k=0; k<c1; k++)   {
               res[i][j] += m1[i][k] * m2[k][j];
           }
       }
   }
   return res;
}
```

```
void displayMatrix(float **matrix, int r, int c)  {
   printf("\n");
   for(int i=0;i<r;i++)    {
       for(int j=0; j<c; j++)  {
           printf("%f ", matrix[i][j]);
       }
       printf("\n");
   }
}


void plotDivisionLines()    {
   glBegin(GL_LINES);
   glVertex3d(-320, 0, 0);
   glVertex3d(320, 0, 0);
   glVertex3d(0, -240, 0);
   glVertex3d(0, 240, 0);
   glVertex3d(-320, 240, -100);
   glVertex3d(320, -240, 100);
   glEnd();
   glFlush();
}


void plotPoint(int x, int y, int x_offset, int y_offset)    {
   glBegin(GL_POINTS);
   glVertex2d(x + x_offset, y + y_offset);
   glEnd();
}


void plotTriangle(float *xs, float *ys, float *zs)   {
   glBegin(GL_TRIANGLES);
   for(int i=0; i<3; i++)  {
       glVertex3f(xs[i], ys[i], zs[i]);
   }
   glEnd();
```

```
}


float** makeTriangleMatrix(float *xs, float *ys, float *zs)  {
   float **res = (float**)malloc(sizeof(float*)*4);
   for(int i=0; i<4; i++) {
       *(res+i) = (float*)malloc(sizeof(float)*3);
   }
   for(int i=0; i<3; i++) {
       res[0][i] = xs[i];
       res[1][i] = ys[i];
       res[2][i] = zs[i];
       res[3][i] = 1;
   }
   return res;
}


float** makeTranslationMatrix(float tx, float ty, float tz) {
   float **res = (float**)malloc(sizeof(float*)*4);
   for(int i=0; i<4; i++) {
       *(res+i) = (float*)malloc(sizeof(float)*4);
       for(int j=0; j<4; j++) {
           if(i==j){
               res[i][j] = 1;
           }
           else{
               res[i][j] = 0;
           }
       }
   }
   res[0][3] = tx;
   res[1][3] = ty;
   res[2][3] = tz;
   return res;
}
```

```c
float** makeScalingMatrix(float sx, float sy, float sz)   {
   float **res = (float**)malloc(sizeof(float*)*4);
   for(int i=0; i<4; i++) {
       *(res+i) = (float*)malloc(sizeof(float*)*4);
       for(int j=0; j<4; j++) {
           if(i==j){
               res[i][j] = 1;
           }
           else{
               res[i][j] = 0;
           }
       }
   }
   res[0][0] = sx;
   res[1][1] = sy;
   res[2][2] = sz;
   return res;
}


float** makeXRotationMatrix(int theta) {
   float **res = (float**)malloc(sizeof(float*)*4);
   for(int i=0; i<4; i++) {
       *(res+i) = (float*)malloc(sizeof(float*)*4);
       for(int j=0; j<4; j++) {
           if(i==j){
               res[i][j] = 1;
           }
           else{
               res[i][j] = 0;
           }
       }
   }
   res[2][2] = cos(theta*PI/180);
   res[0][0] = res[2][2];
   res[2][0] = -sin(theta*PI/180);
   res[0][2] = -res[2][0];
   return res;
```

```
}


float** makeYRotationMatrix(int theta) {
   float **res = (float**)malloc(sizeof(float*)*4);
   for(int i=0; i<4; i++) {
       *(res+i) = (float*)malloc(sizeof(float*)*4);
       for(int j=0; j<4; j++) {
           if(i==j){
               res[i][j] = 1;
           }
           else{
               res[i][j] = 0;
           }
       }
   }
   res[1][1] = cos(theta*PI/180);
   res[2][2] = res[1][1];
   res[1][2] = -sin(theta*PI/180);
   res[2][1] = -res[1][2];
   return res;
}


float** makeZRotationMatrix(int theta) {
   float **res = (float**)malloc(sizeof(float*)*4);
   for(int i=0; i<4; i++) {
       *(res+i) = (float*)malloc(sizeof(float*)*4);
       for(int j=0; j<4; j++) {
           if(i==j){
               res[i][j] = 1;
           }
           else{
               res[i][j] = 0;
           }
       }
   }
   res[0][0] = cos(theta*PI/180);
```

```
    res[1][1] = res[0][0];
    res[0][1] = -sin(theta*PI/180);
    res[1][0] = -res[0][1];
    return res;
}

void display_rotation_translation_scaling() {

    static int theta_vals[3] = {0, 0, 0};
    static float scale_vals[3] = {0, 0, 0};
    static float tr_vals[3] = {0, 0, 0};

    int theta_deltas[3] = {10, 5, 4};
    float scale_deltas[3] = {0.005, 0.01, 0.02};
    float tr_deltas[3] = {0.02, 0.04, 0.01};

    for(int i=0; i<3; i++)  {
        // rotation
        if(theta_vals[i]>360){
            theta_vals[i] -= 360;
        }
        else{
            theta_vals[i] += theta_deltas[i];
        }
        // scaling
        if(scale_vals[i]>1.2){
            scale_vals[i] = 0.8;
        }
        else{
            scale_vals[i] += scale_deltas[i];
        }
        // translation
        if(tr_vals[i]>2){
            tr_vals[i] = -2;
        }
        else{
            tr_vals[i] += tr_deltas[i];
        }
```

```
    }

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0f, 0.0f, -7.0f);

    glColor3f(1.0f, 1.0f, 1.0f);
    plotDivisionLines();

    // front, right, back, left
    float triangle_xs[4][3] = {
        {0.0f, -1.0f, 1.0f},
        {0.0f, 1.0f, 1.0f},
        {0.0f, 1.0f, -1.0f},
        {0.0f, -1.0f, -1.0f}
    };
    float triangle_ys[4][3] = {
        {1.0f, -1.0f, -1.0f},
        {1.0f, -1.0f, -1.0f},
        {1.0f, -1.0f, -1.0f},
        {1.0f, -1.0f, -1.0f}
    };
    float triangle_zs[4][3] = {
        {0.0f, 1.0f, 1.0f},
        {0.0f, 1.0f, -1.0f},
        {0.0f, -1.0f, -1.0f},
        {0.0f, -1.0f, 1.0f},
    };

    float **trans_triangle;
    glBegin(GL_TRIANGLES);
    for(int i=0; i<4; i++)  {
        if(i==0){
            glColor3f(1.0f, 0.0f, 0.0f);     // Red
        }
        else if(i==1){
            glColor3f(0.0f, 1.0f, 0.0f);     // Green
```

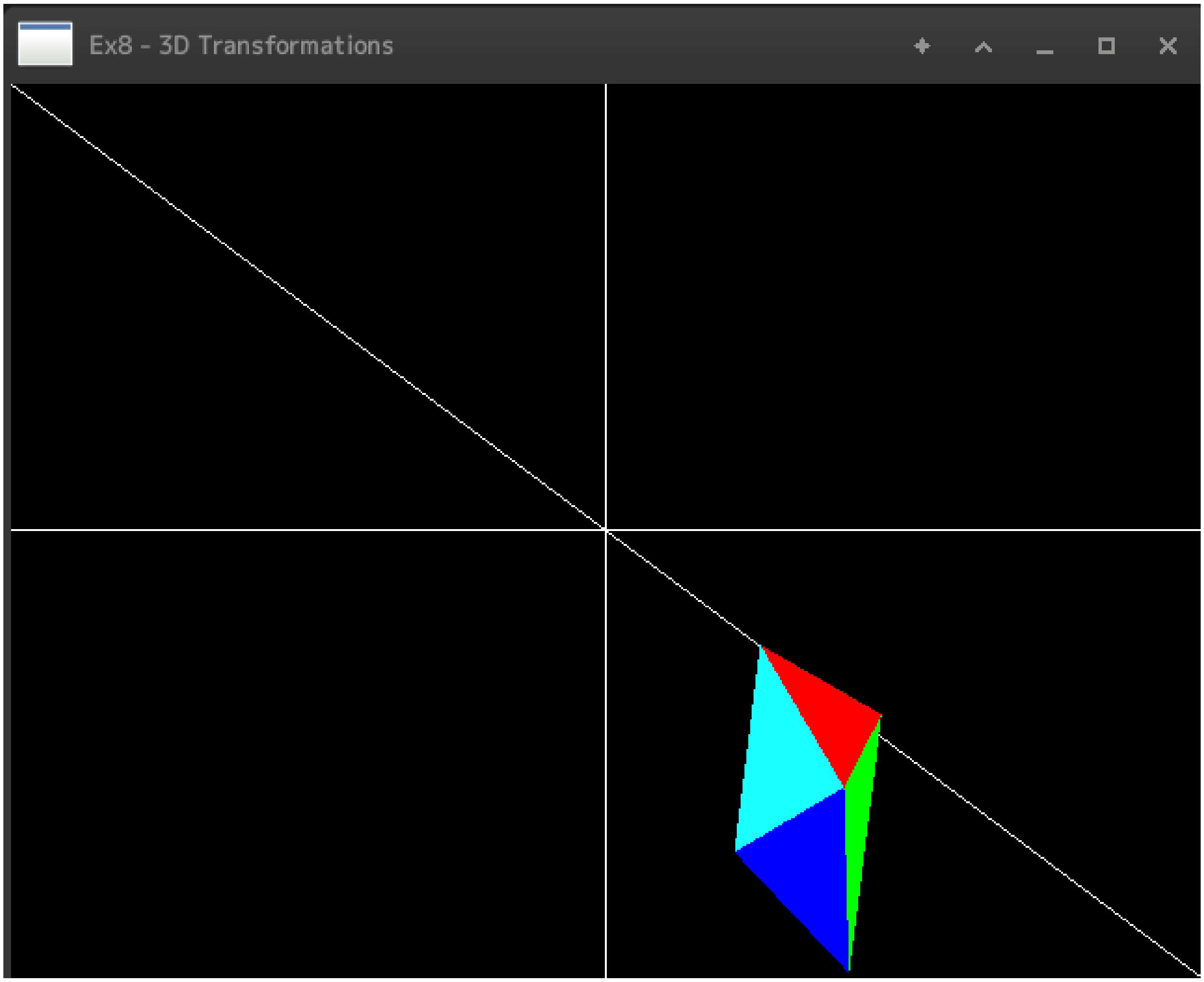
```
        }
        else if(i==2){
            glColor3f(0.0f, 0.0f, 1.0f);     // Blue
        }
        else if(i==3){
            glColor3f(0.1f, 1.0f, 1.0f);     // White
        }

        // rotate
        trans_triangle = multiplyMatrices(
            makeZRotationMatrix(theta_vals[2]),
            multiplyMatrices(
                makeYRotationMatrix(theta_vals[1]),
                multiplyMatrices(
                    makeXRotationMatrix(theta_vals[0]),
                    makeTriangleMatrix(triangle_xs[i], triangle_ys[i],
triangle_zs[i]),
                    4, 4, 3
                ), 4, 4, 3
            ), 4, 4, 3
        );

        // scale
        // displayMatrix(makeScalingMatrix(scale_vals[0], scale_vals[1],
scale_vals[2]), 4, 4);
        trans_triangle = multiplyMatrices(
            makeScalingMatrix(scale_vals[0], scale_vals[1],
scale_vals[2]),
            trans_triangle,
            4, 4, 3
        );

        // translate
        trans_triangle = multiplyMatrices(
            makeTranslationMatrix(tr_vals[0], tr_vals[1], tr_vals[2]),
            trans_triangle,
            4, 4, 3
        );
```

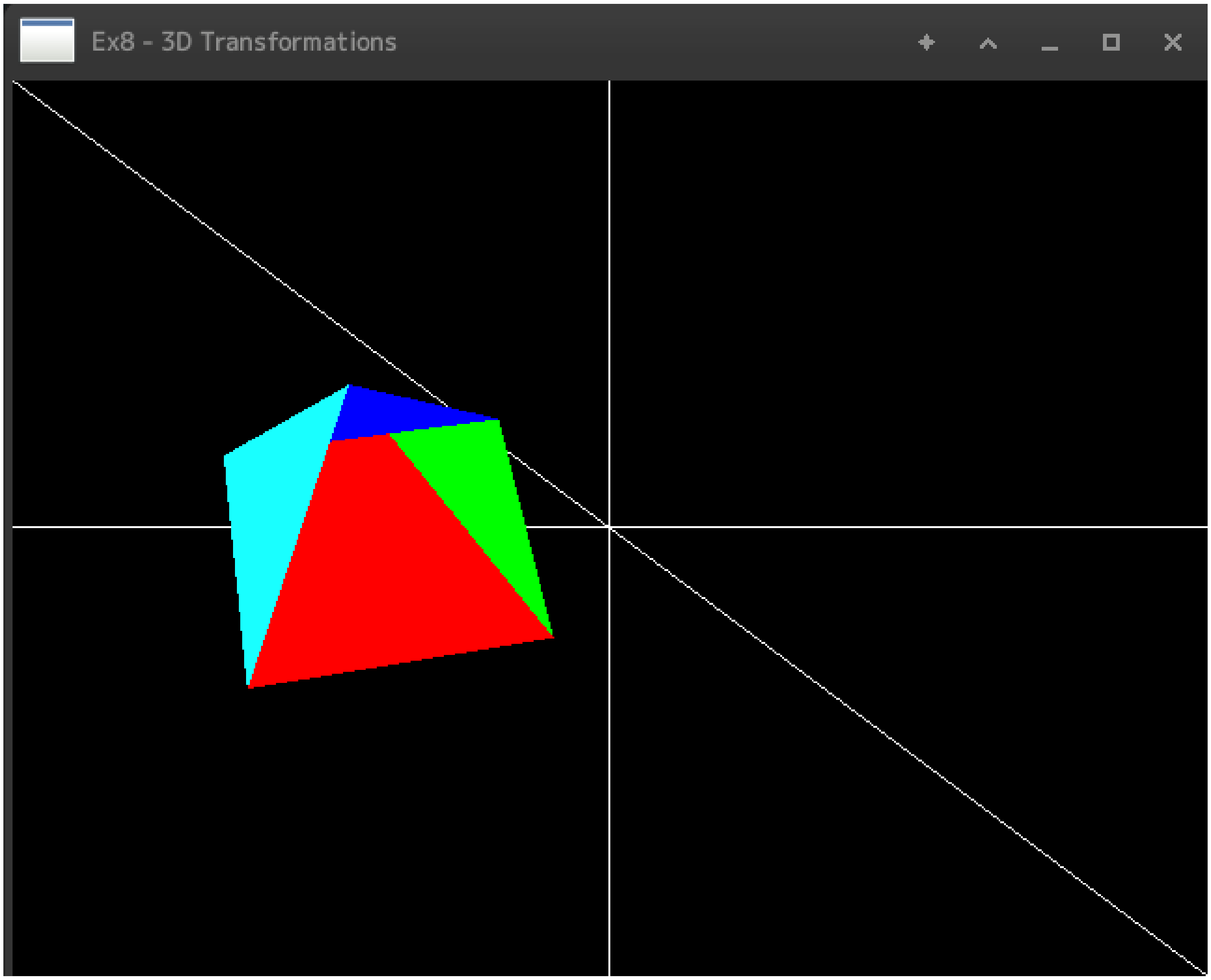
```
        for(int i=0; i<3; i++)  {
            glVertex3f(trans_triangle[0][i], trans_triangle[1][i],
trans_triangle[2][i]);
        }
    }

    glEnd();
    glutSwapBuffers();
    glFlush();
}


void reshape(GLsizei width, GLsizei height) {
  // Compute aspect ratio of the new window
  if (height == 0) height = 1;
  GLfloat aspect = (GLfloat)width / (GLfloat)height;
  // Set the viewport to cover the new window
  glViewport(0, 0, width, height);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45.0f, aspect, 0.1f, 100.0f);
}

void Timer(int value){
   glutPostRedisplay();
   glutTimerFunc(value, Timer, value);
}


void init() {
  glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
  glClearDepth(1.0f);
  glEnable(GL_DEPTH_TEST);
  glDepthFunc(GL_LEQUAL);
  glShadeModel(GL_SMOOTH);
  glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
}
```

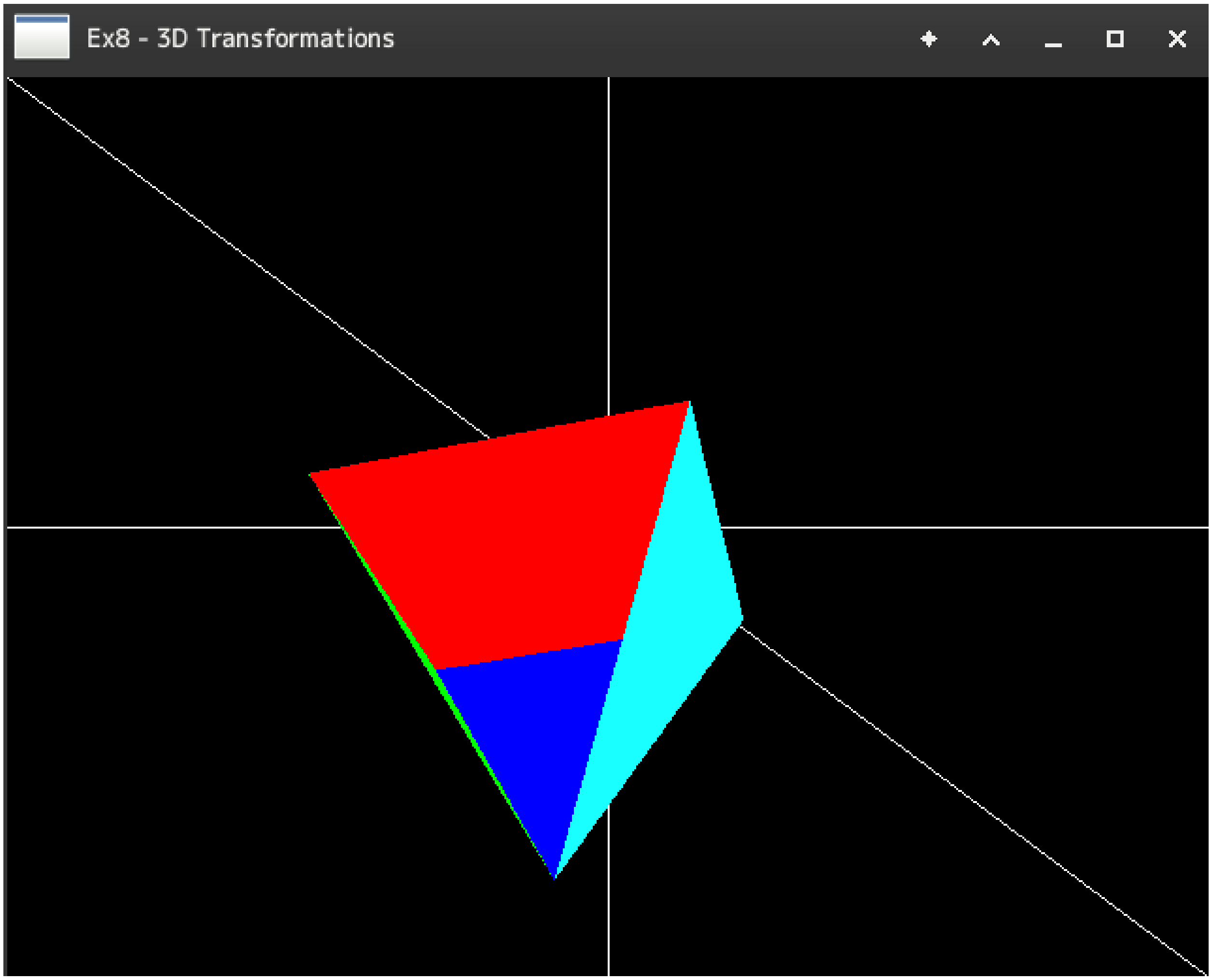
```
int main(int argc, char** argv) {
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_DOUBLE);
   glutInitWindowSize(640, 480);
   glutInitWindowPosition(50, 50);
   glutCreateWindow("Ex8 - 3D Transformations");
   glutDisplayFunc(display_rotation_translation_scaling);
   glutReshapeFunc(reshape);
   init();
   Timer(200);
   glutMainLoop();
   return 0;
}
```

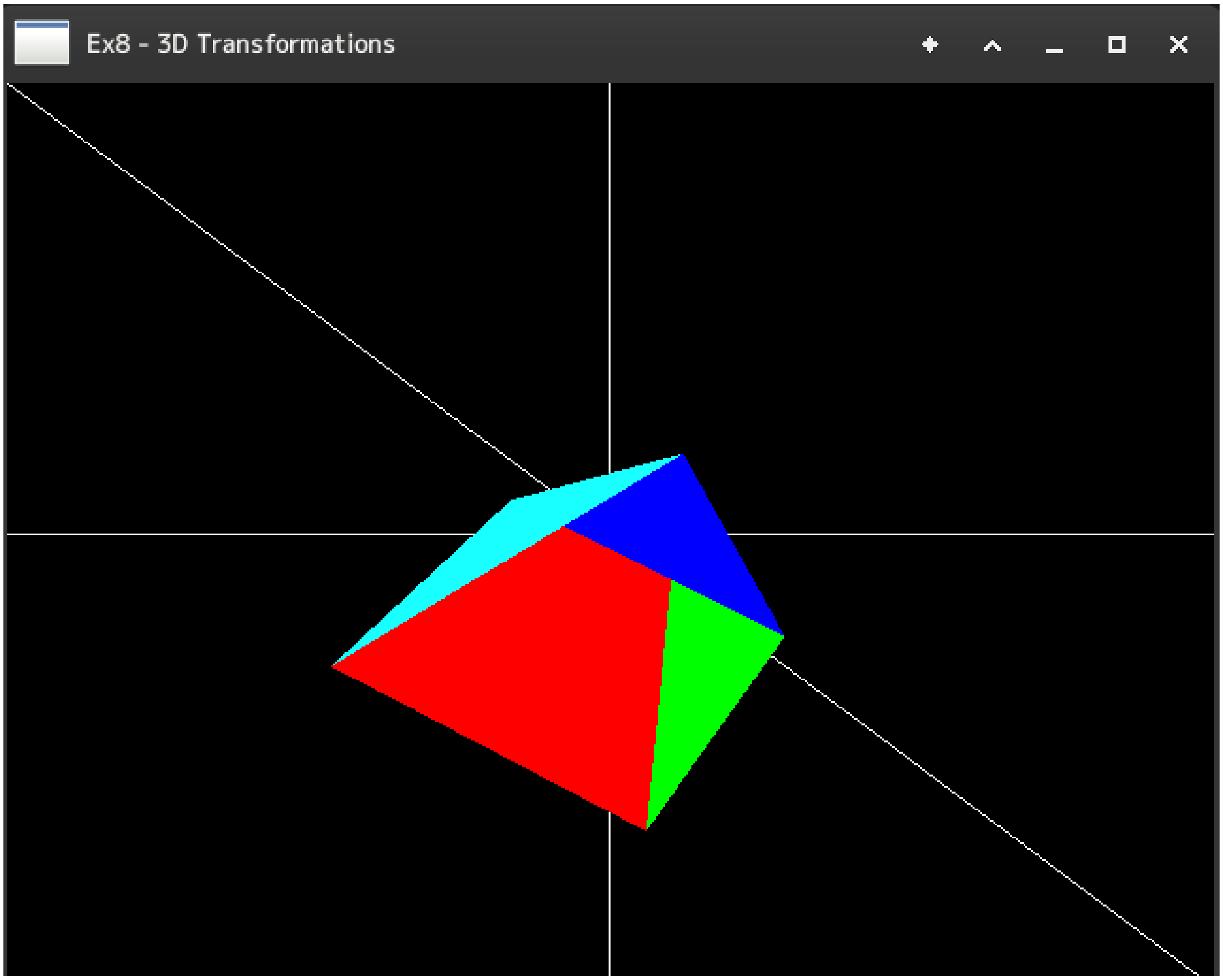
Sample Output

- **All cases 3D Translation, Rotation and Scaling (roates, translates and scales over time in 3D space)**




Ex8 - 3D Transformations


Ex8 - 3D Transformations



Learning Outcomes

Through this implementation of 2D transformation algorithms using the OpenGL framework and C++ programming language, the following concepts were learnt:

1. The working of various 2D transformations — translation, rotation, scaling.
2. The use and application of homogeneous coordinates when rendering transformations using matrices.
3. General understanding of the OpenGL framework and its APIs.