Lab 2- Transformations
Computer Graphics CS4052

Edmond O’ Flynn
12304742

“Keyboard Control”

I modified main.cpp to include a callback function that gets invoked on keypress. The binding occurs in main via glut. For transformations, I leveraged Anton’s maths classes for matrix arithmetic.

```
void onKeyDown(unsigned char key, int x, int y) {
    std::cout << key << std::endl;
    switch (key) { ... }
}
```

Callback function for invoking functions on keypress

```
int main(int argc, char** argv) {

    // Set up the window
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(800, 600);
    glutCreateWindow("Lab 2 - Transformations");

    // Tell glut where the display function is
    glutDisplayFunc(display);
    glutKeyboardFunc(onKeyDown);

    // A call to glewInit() must be done after glut is initialized!
    GLenum res = glewInit();
    // Check for any errors
    if (res != GLEW_OK) {
        fprintf(stderr, "Error: '%s'\n", glewGetErrorString(res));
        return 1;
    }
    // Set up your objects and shaders
    init();
    // Begin infinite event loop
    glutMainLoop();

    // delete allocated objects in memory to prevent leaks
    cleanUp();
    return 0;
}
```

glutKeyboardFunc() takes a delegate onKeyDown for its callback

“Keypress to show: Rotation around x-, y-, and z- axes”

For this lab, I made the transform into an object in order to more easily have multiple objects and transformations in the scene for the same buffer. My functions for rotation take an incrementation parameter to allow for rotation in both directions.

```
void rotateX(GLfloat angle) {
    rotate_x += angle;
    transformation.m[5] = cos(rotate_x);
    transformation.m[6] = sin(rotate_x) * -1;
    transformation.m[9] = sin(rotate_x);
    transformation.m[10] = cos(rotate_x);
}

void rotateY(GLfloat angle) {
    rotate_y += angle;
    transformation.m[0] = cos(rotate_y);
    transformation.m[2] = sin(rotate_y);
    transformation.m[8] = sin(rotate_y) * -1;
    transformation.m[10] = cos(rotate_y);
}

void rotateZ(GLfloat angle) {
    rotate_z += angle;
    transformation.m[0] = cos(rotate_z);
    transformation.m[1] = sin(rotate_z) * -1;
    transformation.m[4] = sin(rotate_z);
    transformation.m[5] = cos(rotate_z);
}
```

Implementations for x, y, and z rotation for buffer objects

Regular triangle

Combined x, y & z transformations

Individual x, y, & z rotations

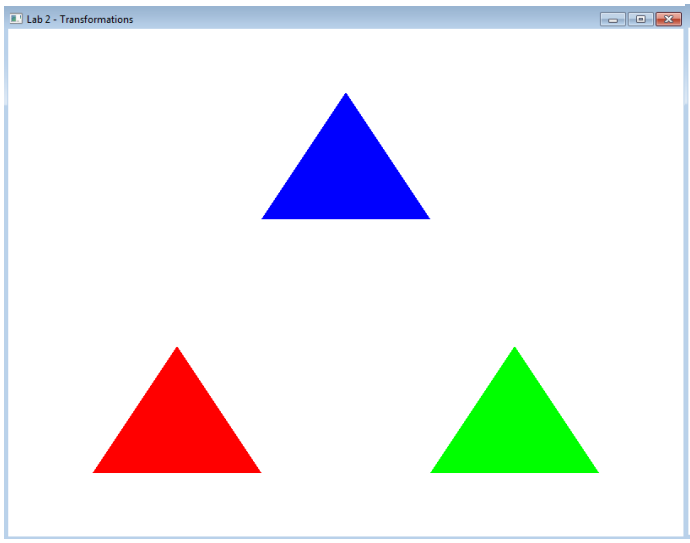
“Keypress to show: Translation in the x-, y- and z- directions”

On keypress, the red triangle has an operation performed onto its transformation matrix which was then fed into the vertex shader and multiplied with its position vector.

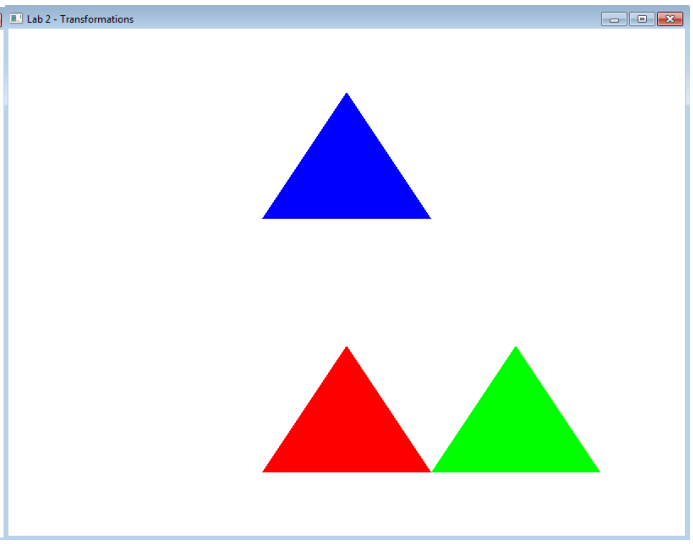
```
void translateToPos(GLfloat x, GLfloat y, GLfloat z) {
    this->x = x;
    this->y = y;
    this->z = z;

    transformation.m[3] = this->x;
    transformation.m[7] = this->y;
    transformation.m[11] = this->z;
}
```

```
void translate(GLfloat x, GLfloat y, GLfloat z) {
    this->x += x;
    this->y += y;
    this->z += z;

    transformation.m[3] += x;
    transformation.m[7] += y;
    transformation.m[11] += z;
}
```

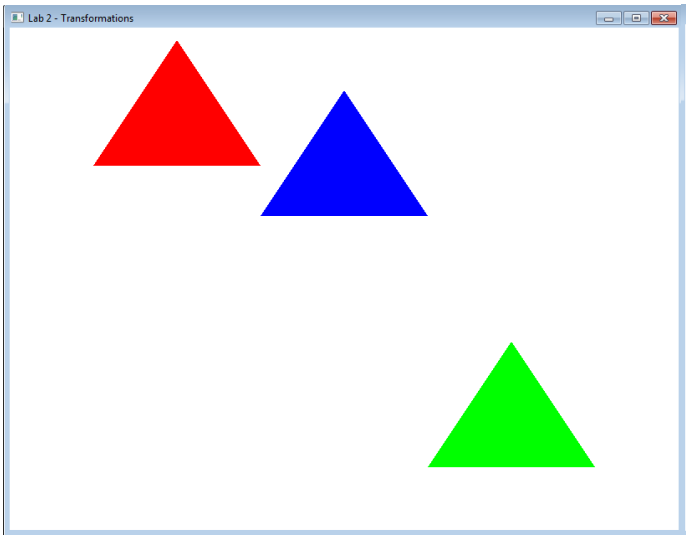
Functions for translating to a specific position and to transform by incrementing a parameter



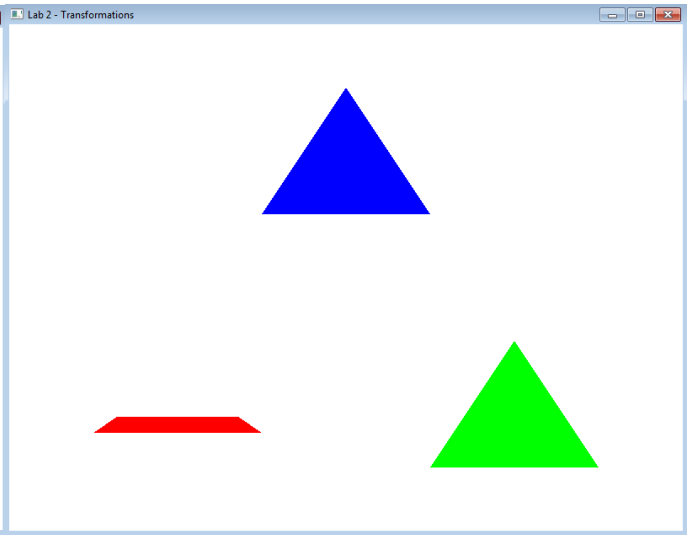
Untransformed triangles



Red triangle transformed in x direction



Transformed in the y direction



Transformed in the z direction (with rotation)

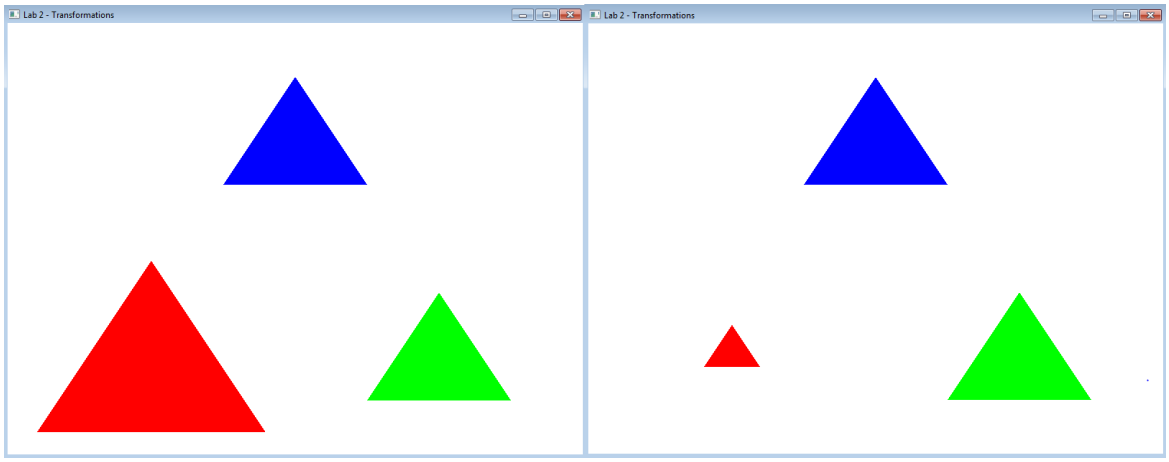
“Keypress to show: Uniforms and non-uniform scaling”

```
void uniformScale(GLfloat scale) {
    transformation.m[0] += scale;
    transformation.m[5] += scale;
    transformation.m[10] += scale;
}
```

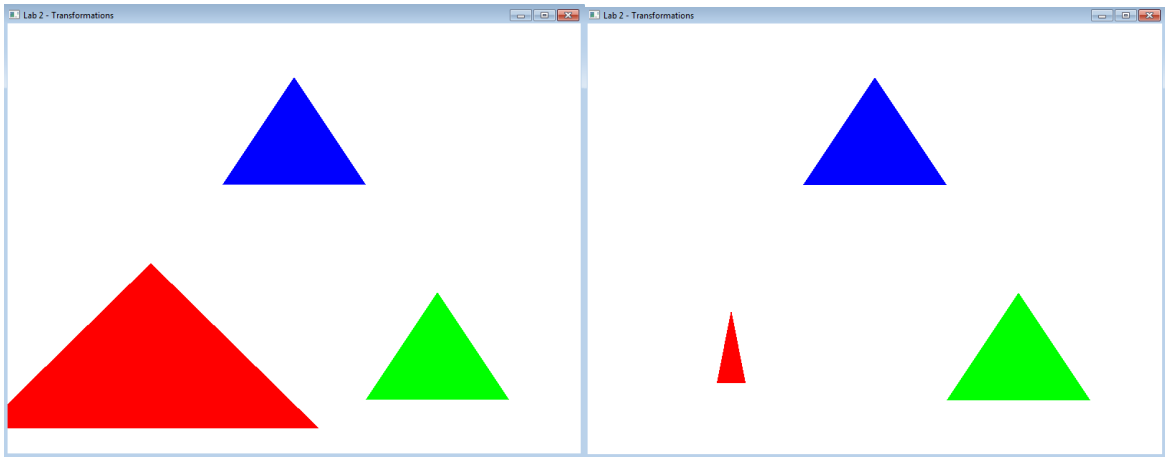
Uniform scaling with parameter

```
void nonUniformScale(GLfloat scaleX, GLfloat scaleY, GLfloat scaleZ) {
    transformation.m[0] += scaleX;
    transformation.m[5] += scaleY;
    transformation.m[10] += scaleZ;
}
```

Non-uniform scaling with multiple scale parameters



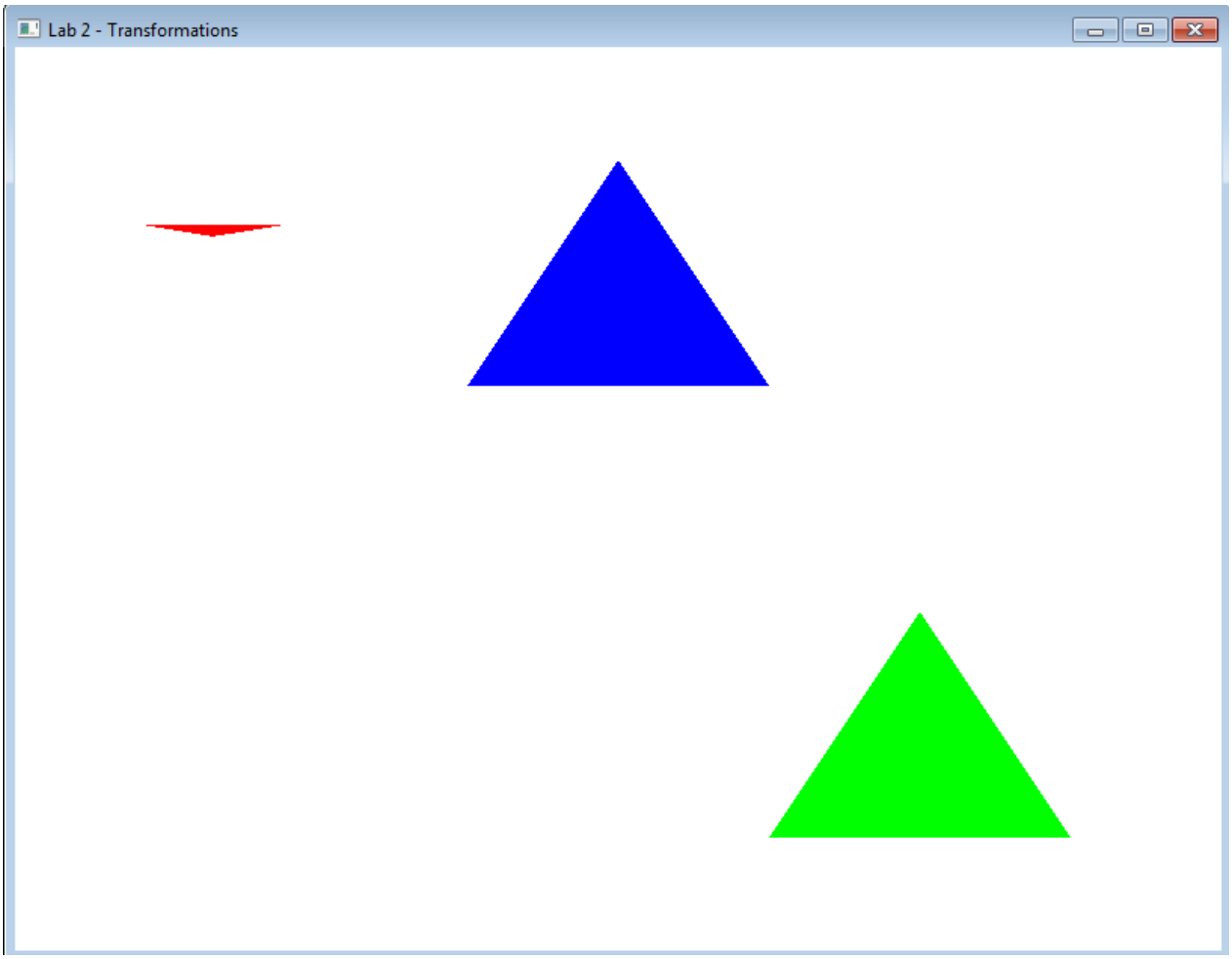
Uniform scaling up and down



Non-uniform scaling up and down

“Keypress to show: Combined transformations”

As I was feeding in a transformation matrix into my vertex, all of my matrix calculations were done in a function that was directly fed into the vertex shader.



Red triangle has a translation, scale and rotation applied in one transformation

```
void combinedTransformations() {
    rotate_x += 0.01f;
    scaleVar += 0.01f;
    posVar += 0.01f;

    mat4 translationTransformation = identity_mat4();
    translationTransformation.m[3] += posVar;
    translationTransformation.m[7] += posVar;
    translationTransformation.m[11] += posVar;

    mat4 scaledTransformation = identity_mat4();
    scaledTransformation.m[0] -= scaleVar;
    scaledTransformation.m[5] -= scaleVar;
    scaledTransformation.m[10] -= scaleVar;

    mat4 rotationTransformation = identity_mat4();
    rotationTransformation.m[5] = cos(rotate_x);
    rotationTransformation.m[6] = sin(rotate_x) * -1;
    rotationTransformation.m[9] = sin(rotate_x);
    rotationTransformation.m[10] = cos(rotate_x);

    transformation = translationTransformation * scaledTransformation * rotationTransformation;
}
```

Operation performed with multiple transforms in one

“Multiple triangles in the scene using the same buffer but creating a new transformation matrix for each one”

As I had to also modify the vertex shader to feed in another variable in order to pass a transformation in, I had to create a new variable, register it with an ID, find it, and then pass data into it.

```
static const char* pVS = "
#version 330

in vec3 vPosition;
in vec4 vColor;
uniform mat4 transformation;
out vec4 color;

void main()
{
    gl_Position = transformation * vec4(vPosition / 2, 1.0);
    color = vColor;
}";
```

New vertex shader with additional uniform mat4 transformation

```
void linkCurrentBuffertoShader(GLuint shaderProgramID) {
    GLuint numVertices = 9;
    // find the location of the variables that we will be using in the shader program
    GLuint positionID = glGetAttribLocation(shaderProgramID, "vPosition");
    GLuint colorID = glGetAttribLocation(shaderProgramID, "vColor");
    transformationID = glGetUniformLocation(shaderProgramID, "transformation");

    // Tell it where to find the position data in the currently active buffer (at index positionID)
    glEnableVertexAttribArray(positionID);
    glVertexAttribPointer(positionID, 3, GL_FLOAT, GL_FALSE, 0, 0);

    // Similarly, for the color data.
    glEnableVertexAttribArray(colorID);
    glVertexAttribPointer(colorID, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(numVertices * 3 * sizeof(GLfloat)));
}
```

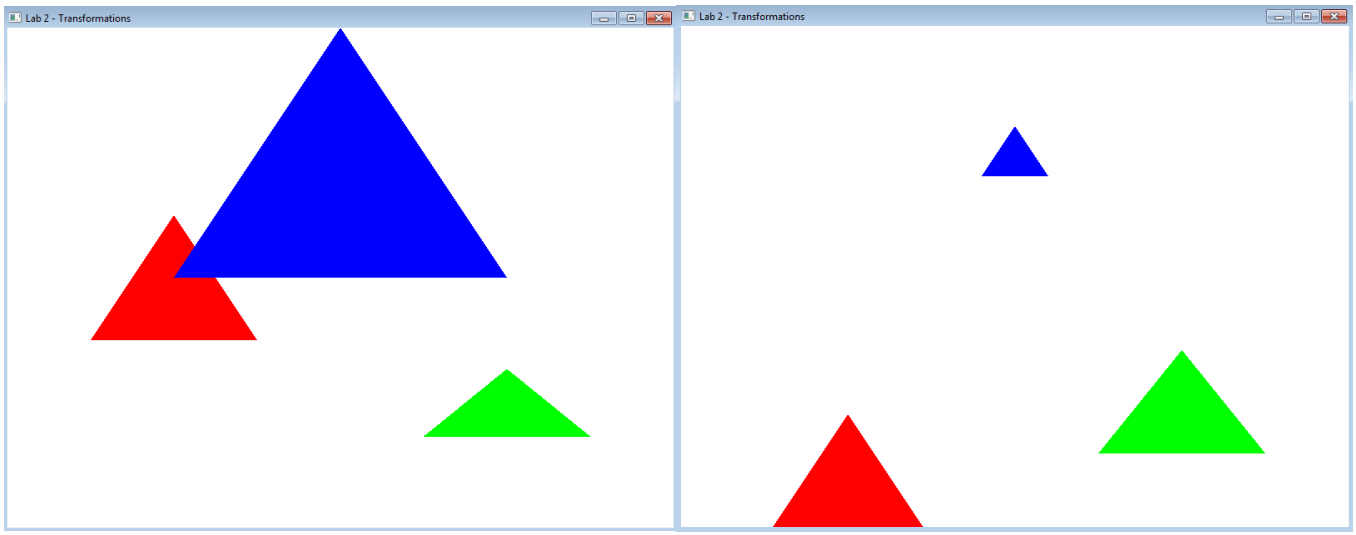
Assigning transformationID a value via glGetUniformLocation()

```
void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    glClearColor(1, 1, 1, 1);
    // NB: Make the call to draw the geometry in the currently activated verte
    //glUniformMatrix4fv(transformationID, 1, GL_TRUE, &transformation.m[0]);
    //glDrawArrays(GL_TRIANGLES, 0, 9);

    glUniformMatrix4fv(transformationID, 1, GL_TRUE, &t1->getMatrix().m[0]);
    glDrawArrays(GL_TRIANGLES, 0, 3);

    glUniformMatrix4fv(transformationID, 1, GL_TRUE, &t2->getMatrix().m[0]);
    glDrawArrays(GL_TRIANGLES, 3, 3);

    glUniformMatrix4fv(transformationID, 1, GL_TRUE, &t3->getMatrix().m[0]);
    glDrawArrays(GL_TRIANGLES, 6, 3);

    glutSwapBuffers();
    glutPostRedisplay();
}
```

Passing in a transformation matrix via class object to the vertex uniform variable

```
void init()
{
    // Create 3 vertices that make up a triangle that fits on the viewport
    GLfloat vertices[] = {
        -0.5f, -0.5f, 0.0f,
        0.5f, -0.5f, 0.0f,
        0.0f, 0.5f, 0.0f,

        -0.5f, -0.5f, 0.0f,
        0.5f, -0.5f, 0.0f,
        0.0f, 0.5f, 0.0f,

        -0.5f, -0.5f, 0.0f,
        0.5f, -0.5f, 0.0f,
        0.0f, 0.5f, 0.0f
    };
    // Create a color array that identfies the colors of each vertex (forma
    GLfloat colors[] = {
        //1st triangle
        1.0f, 0.0f, 0.0f, 1.0f,
        1.0f, 0.0f, 0.0f, 1.0f,
        1.0f, 0.0f, 0.0f, 1.0f,

        //2nd triangle
        0.0f, 1.0f, 0.0f, 1.0f,
        0.0f, 1.0f, 0.0f, 1.0f,
        0.0f, 1.0f, 0.0f, 1.0f,

        //3rd triangle
        0.0f, 0.0f, 1.0f, 1.0f,
        0.0f, 0.0f, 1.0f, 1.0f,
        0.0f, 0.0f, 1.0f, 1.0f,
    };
    // Set up the shaders
    GLuint shaderProgramID = CompileShaders();
    // Put the vertices and colors into a vertex buffer object
    generateObjectBuffer(vertices, colors);
    // Link the current buffer to the shader
    linkCurrentBuffertoShader(shaderProgramID);

    t1->reset();
    t2->reset();
    t3->reset();

    t1->translateToPos(-0.5f, -0.5f, 0.0f);
    t2->translateToPos(0.5f, -0.5f, 0.0f);
    t3->translateToPos(0.0f, 0.5f, 0.0f);
}
```

init function feeding in 3 triangles into one buffer with setting up of positional data



Each triangle concurrently having its own transformation applied in its matrix in the same buffer