CM20219 OpenGL Coursework

I used the GLint and GLfloat data types throughout most of the coursework because on different systems int and float can be represented as different sizes. Using the GL typedefs ensures that my code will work on all other systems, regardless of their primitive values.

Requirement 1 (Draw a simple cube)

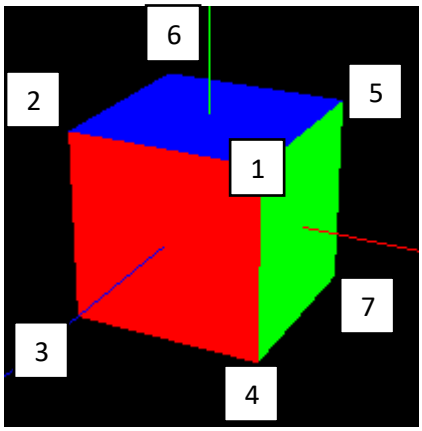
Drawing cube in OpenGL was simple. All that was required was to create an array of the eight vertices of the cube, and then draw six faces from those vertices. To help with the creation of the faces I used a second array called cubeFaces[6][4] which held the index of the vertices needed to create each face. For example, the front face has vertices {1,2,3,4} and the back face has vertices {5,6,7,8}.

```
void drawQuads(void) {
	glBegin(GL_QUADS);
	for (int i = 0; i < 6; i++) {
		glColor3fv(colours[i]);
		for (int j = 0; j < 4; j++) {
			glVertex3fv(&cubeVertex[cubeFaces[i][j] * 3]);
		}
	}
	glEnd();
}
```

The drawQuads() function above draws all six faces at once. The code between glBegin() and glEnd() is used to define a shape to be drawn, for example a quadrilateral. Using GL_QUADS treats each group of four vertices as an independent quadrilateral, in this case it means that for every four vertices defined, a face of the cube will be drawn.

The second loop, using variable j, defines each vertex. It takes the address of the correct cube vertex found in the array of cubeFaces[6][4].

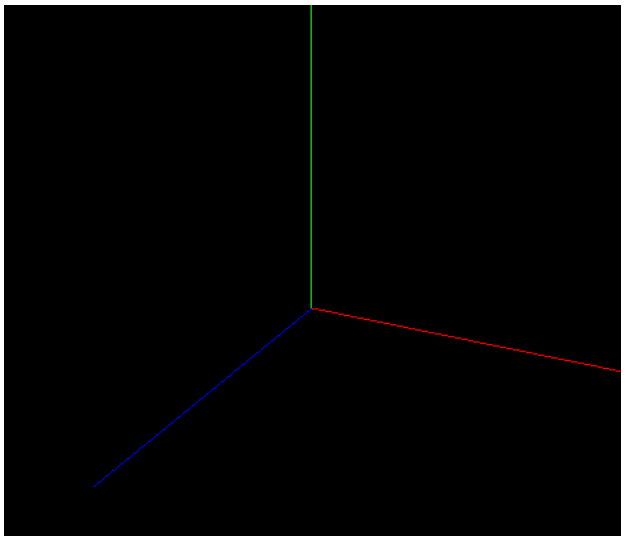
Each face I have drawn is a different colour, this helped with troubleshooting the cube. I had a few issues when first creating the cube because I had written the wrong vertices in my cubeFaces[6][4] array, which meant that the faces were drawn from incorrect vertices. I managed the six colours in an array; red, yellow, green, cyan, blue and magenta.



The screenshot (left) shows the cube with axis drawn for requirement 2. I have edited the screenshot to show the numbering of the vertices, I found that numbering them anti-clockwise from the top right of the cube was the best way to go. Vertex 8 is behind the cube out of view.

You can see from the picture that the cube has square faces that are orthogonal from each corresponding axis.

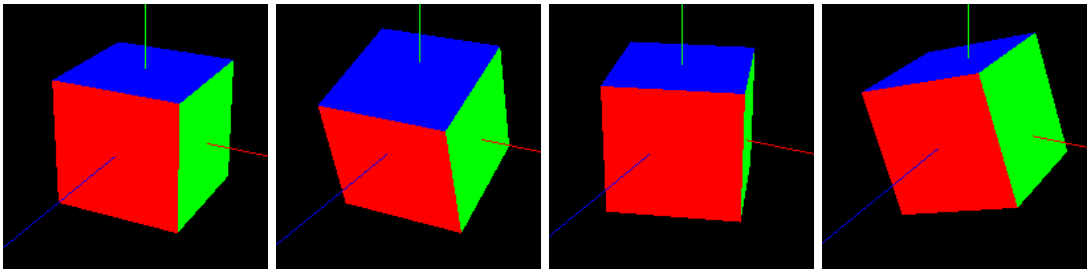
Requirement 2 (Draw coordinate system axes)



```
void drawAxis(void) {
	glBegin(GL_LINES);

	glColor3fv(colours[0]);
	glVertex3f(0.0f, 0.0f, 0.0f);
	glVertex3f(5.0f, 0.0f, 0.0f);

	glColor3fv(colours[2]);
	glVertex3f(0.0f, 0.0f, 0.0f);
	glVertex3f(0.0f, 5.0f, 0.0f);

	glColor3fv(colours[4]);
	glVertex3f(0.0f, 0.0f, 0.0f);
	glVertex3f(0.0f, 0.0f, 5.0f);

	glEnd();
}
```

Requirement 2 asked to draw the coordinate system axes. Seeing as no arrows or tick marks were necessary, all that was needed was for three line to be drawn from the origin (0,0,0) along each of the x, y and z axes. The code was simple, using what had been learned from creating the cube in requirement 1, I was able to draw each line as a different colour and made each line five units long. GL_LINES is used to draw a line between each pair of vertices.

Requirement 3 (Rotate the cube)



[1] Original cube [2] Rotated on X axis [3] Rotated on Y axis [4] Rotated on Z axis

The images above show that the cube can be rotated on fixed axes. In my program the x, y and z keys on the keyboard are used to rotate the cube anticlockwise by pi/30 radians with each key press.

To enable the keys to work with rotation, they first need to be defined in the keyboard function as cases in a switch statement to change the renderMode variable. I then created a function called rotate() that takes arguments that define which axis and which direction to rotate the cube in. The plus character ensures an anticlockwise rotation, whereas the subtract character ensures a clockwise rotation. This was done by a simple if statement that changes the sign of the angle variable theta, which is used in the rotation matrices Rx, Ry and Rz.

Rotation matrices are standard 3x3 matrices that include variable angles at which to rotate an object in three dimensions. By increasing the theta value, the angle of rotation is increased. The matrices to rotate the cube in relation to each axes are different. I chose a theta of pi/30 because I thought that when holding down a rotation key, the cube span at a good rate. A smaller theta, such as pi, would have a full rotation in two key presses, which is not good when trying to determine which axis the cube has rotated on.

```
case 'x':
	rotate('x', '+');
	renderMode = prevRenderMode;
	break;
case 'y':
	rotate('y', '+');
	renderMode = prevRenderMode;
	break;
case 'z':
	rotate('z', '+');
	renderMode = prevRenderMode;
	break;
```

```
GLfloat theta;
if (pos == '-') {
	theta = -M_PI / 30;
}
else {
	theta = M_PI / 30;
}
```

When I first ran the code for rotating the cube, I ran into a logical error. When I clicked one of the rotation keys, the cube completely disappeared from view. I realised after checking through the code that the renderMode would not automatically go back to showing the faces of the cube. I therefore decided to create a new variable called prevRenderMode which saved the renderMode previously used, whether it was faces, edges, or vertices. After making this change, the rotation worked perfectly.

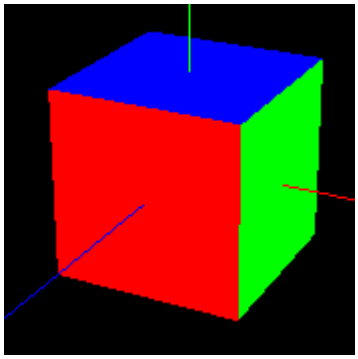
Another hurdle that I had to tackle was multiplying the rotation matrix with the eight vertices of the cube. To solve this, I created a function names multiplyMatrices(). This function took the argument of a 3x3 rotation matrix with which to multiply with each vertex.

In the code below, you can see that the first loop runs eight times, with n being the index of the first element of each vertex. Further loops allow for matrix multiplication; this is done by multiplying each row of the 3x3 matrix by each element of the vertex, and then adding up the total after every row has been used for one element of the vertex. The vertices of the cube are then updated, which in turn will be shown on the display.

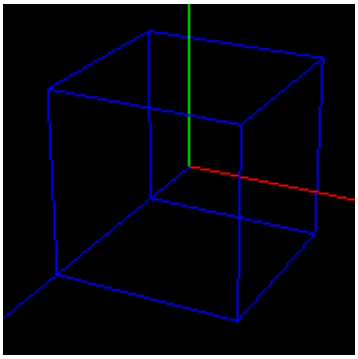
```
void multiplyMatrices(const GLfloat m[3][3]) {
	for (int n = 0; n < 24; n = n + 3) {
		GLfloat temp[3] = { 0.0f, 0.0f, 0.0f };
		for (int i = 0; i < 3; i++) {
			for (int j = 0; j < 3; j++) {
				temp[i] = temp[i] + m[i][j] * cubeVertex[n+j];
			}
		}
		cubeVertex[n] = temp[0];
		cubeVertex[n + 1] = temp[1];
		cubeVertex[n + 2] = temp[2];
	}
}
```

Of course, the axes and camera remain fixed when rotating along each axis because the only things that is being changed in the functions above are the vertices of the cube.

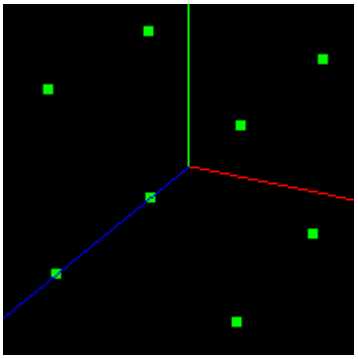
Requirement 4 (Different render modes)



[1] Faces only



[2] Edges only



[3] Vertices only

As before in requirement 3, new cases had to be entered into the keyboard function. This time I added 'v' for vertices, 'e' for edges, and 'f' for faces. *The code for drawing the faces of the cube is shown in requirement 1.*

Vertices

The vertices of the cube were the simplest to draw, as they only required me to iterate through the array cubeVertex[24] once, with a new vertex every three indices along the array.

```
void drawVertices(void) {
	glBegin(GL_POINTS);
	glColor3f(0.0f, 1.0f, 0.0f);
	for (int i = 0; i < 24; i = i + 3) {
		glVertex3f(cubeVertex[i], cubeVertex[i + 1], cubeVertex[i + 2]);
	}
	glEnd();
	glPointSize(5);
}
```

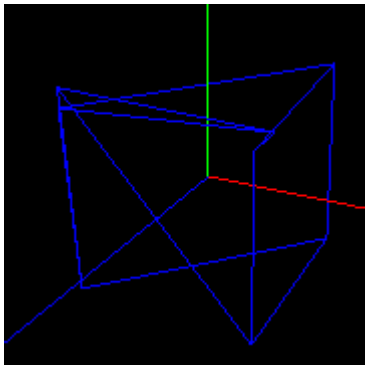
Each vertex is shown in green, and with a point size of 5. Changing the point size will change the visible size of the vertex when displayed. GL_POINTS treats each vertex as a single point.

Edges

To draw the edges of the cube, I first created an array cubeEdges[12][2]. This array works in the same way as cubeFaces[6][4], as it provides the indices of the array cubeVertex[24] needed to create the lines between the correct vertices of the cube. Thus, cubeEdges[12][2] is an array of pairs of vertex indices. The functions for drawEdges() and drawLine() are self-explanatory.

```
void drawEdges(void) {
	for (int i = 0; i < 12; i++) {
		drawLine(colours[4], &cubeVertex[cubeEdges[i][0]*3],
		&cubeVertex[cubeEdges[i][1]*3]);
	}
}
```

```
void drawLine(const GLfloat colour[3], GLfloat p[3], GLfloat q[3]) {
	glBegin(GL_LINES);
	glColor3fv(colour);
	glVertex3fv(p);
	glVertex3fv(q);
	glEnd();
}
```



During the development of my drawEdges() function, I came across an error (shown left). As you can see, the edges are drawn, but only ten out of twelve are visible, and they go to and from incorrect vertices. I assume that in this case, the other two edges are in fact present, but are hiding behind other edges. This error was caused by an array index confusion, I had forgotten to multiply the integer value found in cubeEdges[12][2] by 3, which would have found the correct index in the cubeVertex[24] array.