# COMPUTER GRAPHICS – CS452

**Mobeen Tariq** – 17100047

**Abdullah Siddique** – 17100181

PROJECT REPORT

Project Title: Rubik’s Cube

Platform: OpenGL

Brief Description: The main objective of this project was to simulate the complete working a Rubik’s Cube through OpenGL, which may seem easy at first but becomes increasing difficult as you go into the depths of it’s logic.

Logical Implementations:

Our code can be logically categorized into 2 different sections that work in tandem to simulate the complete workings of a 3-Dimensional Rubik’s Cube.

1. The use of OpenGL transformations and a 3-D array to simulate to physical animation of the rotations of a Rubik’s Cube.

Our Rubik’s Cube comprises of 27 individual cubes made through GL\_QUADS. Each cube is initially assigned a separate unique number that identifies the each cube, these are stored inside a 3-D array, indices of which store the position of the cube. So when we want to rotate a specific set of cubes, we check the positions in the array to see if the rotation is possible for the cubes and then we rotate them.

1. The use of Texture coordinates and vectors to assign colors to cube.

The colors for our Rubik’s cube are stored in 6 vectors, each corresponding to one face of the cube. Each vector holds the texture coordinates of the image that is to be texture mapped onto a single cube face. Specifically indexed vector elements are passed to the drawcube1() functions which initializes the cube with those specific colors. We number these according to diagram below where the numbers represent the index which store the y and x of the texture picture to be mapped respectively.

|  |  |  |
| --- | --- | --- |
| 0,1 | 2,3 | 4,5 |
| 6,7 | 8,9 | 10,11 |
| 12,13 | 14,15 | 16,17 |

Working Together

When a certain rotation takes place, we update the values of the array which correspond to the position of the cubes and also update the vectors to keep track of the colors of the cube. Basically out cube only rotates 90 degrees and then resets, but since we are updating the vectors with the rotations, when the cube resets and reinitializes, it is reinitialized with the updated colors. For this to work, we created out own specific texture so that we only have to load on texture image. This is how we get a fully functional animation of a Rubik’s Cube. A working example would be when the top is rotated clockwise the top three colors from left face will replace the top three colors from front face and so on (specific elements of vector swapped). Since these are fixed patterns for each rotation we use conditions to swap the colors. The colors on top face in this rotation will also be swapped but not with colors from other faces but from colors from the same face. These swaps take place only when the rotation is 90 degrees.

These two aspects combined enable us to properly show a Rubik Cube rotation as when the cubes are rotated even though they reset at 90 degrees the color have been swapped which gives the image that the cubes have been rotated.

Difficulties Faced:

Many difficulties were faced during the simulation.

* When starting the project we tried simple rotation methods to rotate certain parts of the 9x9 cube. The single rotations worked perfectly but when multiple rotations were applied the cube started to deform. This was because OpenGL rotated the cubes according to the axis of the cube and not the origin so when we had already applied one rotation to a certain cube its axis changes and the second rotation was applied according to the new axis which started deforming the cube if applied on many different parts at the same time. We also faced the Gimbal Lock which is a problem faced in rotations that use Euler Angles and has no solution.
* We realized that through a lot of research that this problem could only be fixed by Quaternions which use a specific vector to do rotations around instead of using axis of the cube. The quaternion took time to understand as it was a tricky and new concept not done in the class. The Quaternion class was implemented and a rotation was defined for a single small cube. The Quaternion requires that it should be left multiplied with the matrix but OpenGL had no function that could do this. After much trial and error we used a get function to get the Model View matrix and left multiplied but it still did not give the proper result. We realized this was because the camera that was defined in the beginning was not a quaternion itself. So we started working on defining the camera using Quaternions but this proved much difficult than we expected as we were not able to order our many quaternion multiplications properly so we decided to work on other strategies to solve our problem.
* We then tried to manipulate the textures to give the illusion of rotation but we could not fit all the patterns in the texture and the colors would randomly change after a series of rotations. We finally came to our working solution through applying both texture coordinate manipulations and rotation transforms to get a fully functioning Rubik’s Cube.

Final Product:

**Controls**:

‘**p**’: Top clockwise rotation

‘**P**’ (Shift + ‘p’): Top anti-clockwise rotation

‘**o**’: Bottom clockwise rotation

‘**O**’ (Shift + ‘o’): Bottom anti-clockwise rotation

‘**g**’: Front clockwise rotation

‘**G**’ (Shift + ‘g’): Front anti-clockwise rotation

‘**h**’: Back clockwise rotation

‘**H**’ (Shift + ‘h’): Back anti-clockwise rotation

‘**b**’: Right clockwise rotation

‘**B**’ (Shift + ‘b’): Right anti- clockwise rotation

‘**n**’: Left clockwise rotation

‘**N**’ (Shift + ‘n’): Left anti-clockwise rotation

‘**r**’: Master Reset

A simulation of a 9x9 Rubik cube that starts off solved. Colors are represented using texture mapping of images with different colors. Using the controls above the cube can be rotated to give different color combinations. A very easy and good looking pattern can be made by rotating top, bottom, left, right, front and back by 180 degrees in that specific order.