School of Computing
FACULTY OF ENGINEERING

UNIVERSITY OF LEEDS

Computer Graphics

Coursework 2

Jonathan Alderson

Rubik's Cube

2019

Table of Contents

Rubik's Cube

Initialisation

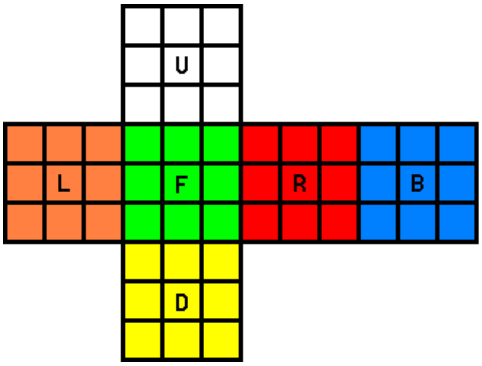
The first stage of this project was to create the logic to implement the Rubik's cube itself.

The cube has an object oriented structure, there is a class for both the cube and an individual cubie. Each cube has 27 cubies, 9 on each layer.

The cubes constructor initialises all of the cubies. It gives each of them a relative position to the cube. Where each of the x, y, z coordinates are either -1, 0, 1. This helps keep things simple, having their relative coordinates makes calculations and other things much easier. Even though their actual positions in the scene will be somewhere else.

In the constructor for each of the cubies, we must initialise all of the faces with which colour they are. In the cube class we store a static list of the colours for each face. An integer represents a colour or material for the faces.

0	White	Top
1	Green	Front
2	Red	Right
3	Blue	Back
4	Orange	Left
5	Yellow	Down
6	Black	Edges



For each cubie we store the Up, Down, Left, Right, Forward and Back faces. With an integer for each one which eventually gets turned into the corresponding material.

```
faceData.push_back({0, 6, 4, 6, 3, 6});  // top layer
faceData.push_back({0, 6, 6, 6, 3, 6});
faceData.push_back({0, 6, 6, 2, 3, 6});
faceData.push_back({0, 6, 4, 6, 6, 6});
faceData.push_back({0, 6, 6, 6, 6, 6});
faceData.push_back({0, 6, 6, 2, 6, 6});
faceData.push_back({0, 1, 4, 6, 6, 6});
faceData.push_back({0, 1, 6, 6, 6, 6});
faceData.push_back({0, 1, 6, 2, 6, 6});

// This goes on for the other two layers as well
```

These lines in the constructor set up the initial colours for a solved cube.

Logic Flow

The basic logic for the cube is.

If there is currently a move in progress

    Do one more step of the animation

    Perform a check to see if it the animation has been finished

    If animation has finished

        Update the colours and the positions of all cubies

If we are in the middle of a solve

    Wait until the current move has finished

    Do one more move of the solve

    Perform a check to see if the solve has finished

If we are in the middle of a scramble

    Wait until the current move has finished

    Do one more move of the scramble

    Perform a check to see if the scramble has finished

If we are not scrambling or solving

    Open to take user input

This logical flow helps keep consistency with the cube. Previously, there were consistency errors when solves and scrambles were happening at the same time, or user moves would be inputted mid scramble. It was slightly harder than it should have been to get this water tight. But now, the cube will never lose consistency with itself.

Rotations

To achieve the rotations. We handle whole cube rotations and face rotations separately. We store the angle of rotation for each of the faces and the whole cube. When a rotation starts, we set the appropriate cubeRotation or faceRotation to be either -90 or 90, depending on the direction of the inputted move. By doing a step of the animation we gradually rotate the face, until the remaining rotation value is taken back to 0. At this stage, one of the faces or axis' will be rotated 90 or -90 degrees. Once this has finished we update each of the cubies positions and colours so that it looks like a rotation has been completed, then the rotation is reset so everything is back at 0 like in the code below. This helps keep consistency, as when we want the 'front' face of the cube, we just look for cubies with z = +1, ect. So we don't need to keep track of anything when compound rotations start, this makes it easier to understand what is going on.

```
1.  cubeRotations = {0, 0, 0};
2.  remainingCubeRotations = {0, 0, 0};
3.
4.  faceRotations = {0, 0, 0, 0, 0, 0};
5.  remainingFaceRotations = {0, 0, 0, 0, 0, 0};
```

```
1.  // Whole Cube X
2.   if(rot[0] == 'X'){this->remainingCubeRotations[0] = -thisRotateAmount;}
3.
4.  // this continues for both Y and Z
5.  // ...
6.
7.   // Front Face
8.   else if(rot[0] == 'f'){this->remainingFaceRotations[0] = -thisRotateAmount;
9.                          cubiesToRotate = this->getCubies(-2, -2, 1);}
10. // this continues for b, l, r, u, d
11. // ...
```

Above code runs from the keyboard input. The index into remainingCubeRotations changes based on the axis or face.

cubesToRotate makes sure that we only update the colours and positions of the cubies on the face that are doing a face rotation. In the getCubies function, an argument of -2 is equivalent to *, so line 9 will get us all cubies with a z of positive 1, with any x and any y position. Getting all the cubies on the front face.

The draw function takes care of drawing the rotations, once we have what angle each face should be at. So this will be further discussed in the rendering parts of the report.

Scrambling

To achieve the scrambling effect. We recycle the same functions used for the users keyboard input, we simply create a list of 20 random moves. Then input the moves one at a time, waiting in-between.

The difficulties when doing this were that, since we do not have a threaded application, inputting the moves has to be managed so that they do not all get inputted on the same frame. To do this, we use a Boolean to check if a scramble is in progress at all. This overrides keyboard input and iterates a scramble move as soon as the previous is done. When the scramble runs out of moves, we reset the scramble moves and give control back to the user.

Another difficulty was with the helper cubes. Initially the scramble chose a random move at each step. This led to each of the helper cubes choosing their own moves, making the cubes be inconsistent with each other. Figure 2 shows consistent cubes after the fix.

To fix this, instead of the individual cubes creating their own moves. We create the random moves on the button press, and pass this list to each cube, so they always have the same moves to input.

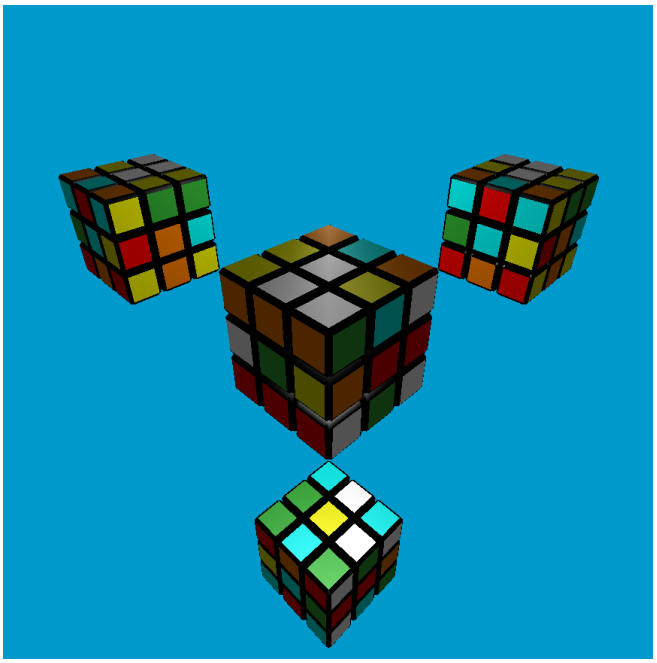


Figure 2: Consistent helper cubes

```
1.  void RubiksWidget::scrambleCubeButton()
2.  {
3.    // Try to scramble the cube, unless somethings already happening
4.    // We have to make a global list of random scramble moves.
5.    // If we let each cube pick their own, they will just pick
6.    // randomly and mess up the cubes relative to eachother
7.    // so we have to make a list here and pass it to each cube
8.    srand(time(NULL));
9.    vector<string> allMoves = {"u", "d", "l", "r", "f", "b"};
10.   vector<string> randomMoves;
11.   for(int i = 0; i < cubes[0].movesInAScramble; i++)
12.   {
13.     randomMoves.push_back(allMoves[rand() % 6]);
14.   }
15.   for(unsigned int j = 0; j < cubes.size(); j++)
16.   {
17.     cubes[j].startScramble(randomMoves);
18.   }
19. }
```

Solving

For the scope of this coursework, it would be a unfeasible to include a full solver. We could achieve a real solver in 3 different ways.

- Using a God's Number approach Link
- Using beginners method approach Link
- Using a CFOP approach Link

Each of these methods of solving are interesting in themselves. Using either the beginners method of the CFOP method would be the most satisfying to watch. As we would be able to see the final solution coming together a lot more. As it gets built up layer by layer. These would be tedious to implement, as ~123 algorithms would have to be hardcoded (If we were implementing full OLL and full PLL). This was not worth my time, especially in a graphics related module.

The God's Number approach would be implemented by contacting an API with a representation of the current cube state. Again, a lot of effort.

To get around this I used a simple cheating approach. We simply use a stack to store all the moves, and pop them off until the stack is empty. Then the cube must be solved.

All the solve moves can be calculated easily for each cube, so when the Qt button is clicked, we just set the cube state to be in the in the middle of a solve.

The solving has the same management system as with the scrambling. So there can be no user input mid solve. So at every opportunity, we can just pop a move off the stack and input it like a user would.

```
1.  void Cube::solveCube()
2.  { // Pop the stack of moves, and do one more
3.    string nextMove;
4.    string prime = "'";
5.    if(!allMoves.empty()) // if there's any more moves to do
6.    {
7.      nextMove = allMoves.top();
8.      allMoves.pop();
9.      // Invert the move, add the prime ‘ on the move
10.     if(nextMove.length() == 2){ nextMove = nextMove[0];}
11.     else                      { nextMove = nextMove + prime;}
12.     this->rotate(nextMove);
13.     allMoves.pop(); // Remove what we just did, since it doesn't matter
14.                     // otherwise, we just get stuck in a loop
15.   }  else
16.   {
17.     // we have finished solving, and can resume keyboard input as normal
18.     this->solvingCube = 0;
19.   }
20. }
```

40% - 50% Band

Reasonable Complexity Through Instancing

The instancing used in this project is with the cubies. In my scene, there are 4 cubes, each of which has 27 cubies. Each cubie is made up of 6 faces, 8 cylinders and 8 spheres. This makes the scene quite complex with:

648 GL_POLYGONs

864 gluCylinders

864 gluSpheres

As well as the complexity from instancing. The project also makes use of confetti, the confetti is stored in a vector, they are also in their own class and are responsible for drawing themselves. Below is the code to setup all of the instances of my instanced objects. All of these are stored in vectors, so when we want to update a group of items, we can iterate through the vector and call an update or draw function on all of them, giving clean simple main code.

```
1.  // Instancing for the confetti  (In the widget constructor)
2.  for(unsigned int i = 0; i < 500; i ++)
3.  {
4.    confetti.push_back(Confetti(-5, 5, 15, 150, -5 , 5, (float)0.01));
5.  }
6.
7.
8.  // Instancing for the cube  (In the widget constructor)
9.  cubes.push_back(Cube()); // Calls the constructor
10. cubes.push_back(Cube()); // This is for the 3 cubes to help you see
11. cubes.push_back(Cube()); // the other
12. cubes.push_back(Cube()); // sides
13.
14.
15. // Instancing for the cubies  (In the cube constructor)
16. for (unsigned int y = 0; y < yList.size(); y++)
17.   {
18.     for (unsigned int z = 0; z < zList.size(); z++)
19.     {
20.       for (unsigned int x = 0; x < xList.size(); x++)
21.       {
22.         Cube thisOne(xList[x], yList[y], zList[z], 1, faceData[counter][0], faceDa
    ta[counter][1],
23.           faceData[counter][2], faceData[counter][3],
24.           faceData[counter][4], faceData[counter][5], counter);
25.           cubies.push_back(thisOne);
26.           counter += 1;
27.         }
28.       }
29.     }
```

Light and Material Properties

When constructing the Rubik's cube plastic materials seemed like the best choice. Learn OpenGL resources has a convenient list of premade materials. For the default colour scheme I chose the plastic materials, creating an orange material as there was not one available. For the alternative colour scheme I chose relevant metal materials, with the plastic materials being my preferred choice.

To achieve beautiful reflections. Two light sources were used, one to light the scene, and a secondary rotating light source, which gets the highlights.

Figure 3's first picture shows the highlights light above the white face, giving specular reflections to the top.

The second picture showing the highlights light on the right hand side face, giving specular reflections on the rounded corners

I found that the y height of this rotating light mattered quite a lot, and by varying it, allowed me to achieve nice specular reflections in 3 different places. My plan for varying the y height was to have y = 8 for the first rotation, then y = 4, finally y = -4. This worked well initially except for the sudden jump between the heights every rotation. This was fixed by adding one of my favourite neat little lines in the whole project:

```
1.  lightY = 8.0 - min(max(0.0, r - 330.0) * 0.1, 4.0)
2.               - min(max(0.0, r - 690.0) * 0.1, 8.0)
3.               + min(max(0.0, r - 960.0) * 0.1, 12.0);
```

This function smoothly changes the y value between all the values I need to achieve my specular reflections in the different places.

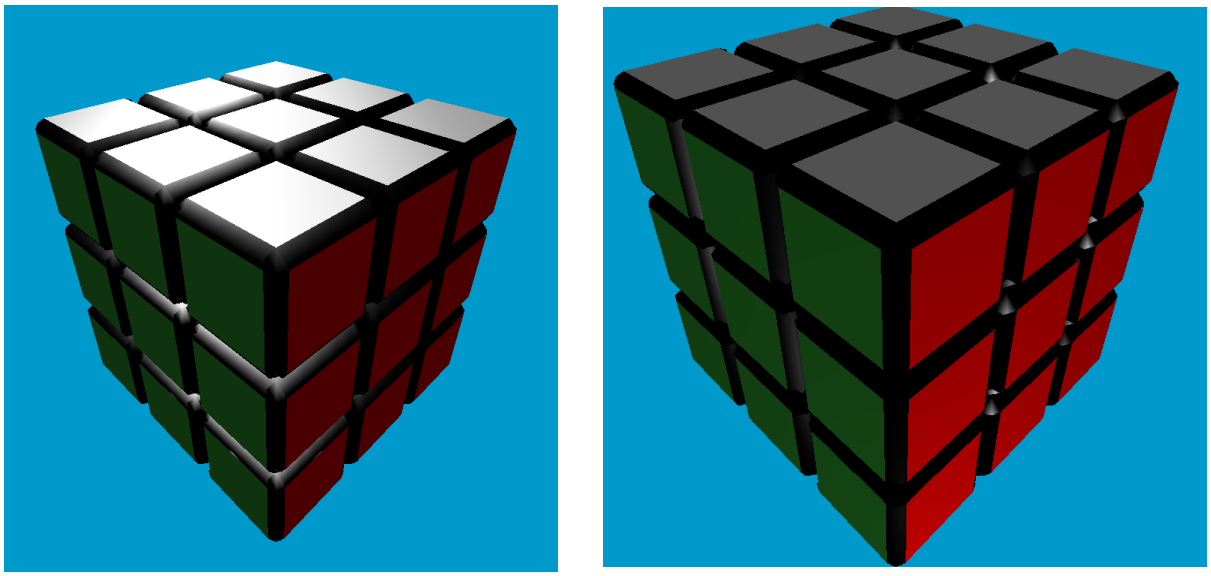


Figure 3: Specular Reflections

Design Explanation

*Use of Glut objects will be explained in the 60% - 70% band as that is a convex object constructed from polygons.

The design of this project is to have a fully solvable 3D Rubik's cube. Where the user can control each face, including prime moves which allow the move to be reversed.

It is very fun to give the cube a scramble and try to get the whole thing solved. If you need help try looking at the beginners method. Since the key bindings are the same as Rubik's cube notation, it's very easy to follow the algorithms, so you should be able to get it solved. Once completed, turn on the confetti to reward yourself.

The controls for the Rubik's cube are:

(Number Pad) (Each of these can be inverted by holding Shift)

- 8 Clockwise X Axis rotation
- 5 Anticlockwise X Axis rotation
- 4 Clockwise Y Axis rotation
- 6 Anticlockwise Y Axis rotation
- 9 Clockwise Z Axis rotation
- 7 Anticlockwise Z Axis rotation
-
- F Clockwise Front face rotation
- B Clockwise Back face rotation
- L Clockwise Left face rotation
- R Clockwise Right face rotation
- U Clockwise Up face rotation
- D Clockwise Down face rotation

50% - 60% Band

User Interaction

This project contains an awful lot of user interaction. As well as controlling the cube itself. We have many manipulatable parameters on the right hand side.

Camera Distance

Moves the camera further or closer.

Rotation Speed

Changes how long it takes a single move to be carried out, this applies to any user moves, or scrambles or solves.

Cube Spacing

Changes how large the gaps are between cubies.

Edge Thickness

Since the corners are rounded. This can change the cube from looking perfectly square, to made of balls. This slider changes the sizes of the rounded edges made from spheres and cylinders.

Stretch Factor

When a rotation is happening, the stretch factor will exaggerate the movement by use of a sin function. This controls the amplification.

Path Radius

For use only when 'Move in a Circular Path' is ticked, varies the cubes spinning path.

Marc

This is actually a dropdown to control the texture of the confetti, Money is my favourite.

Confetti

Enables or disables confetti

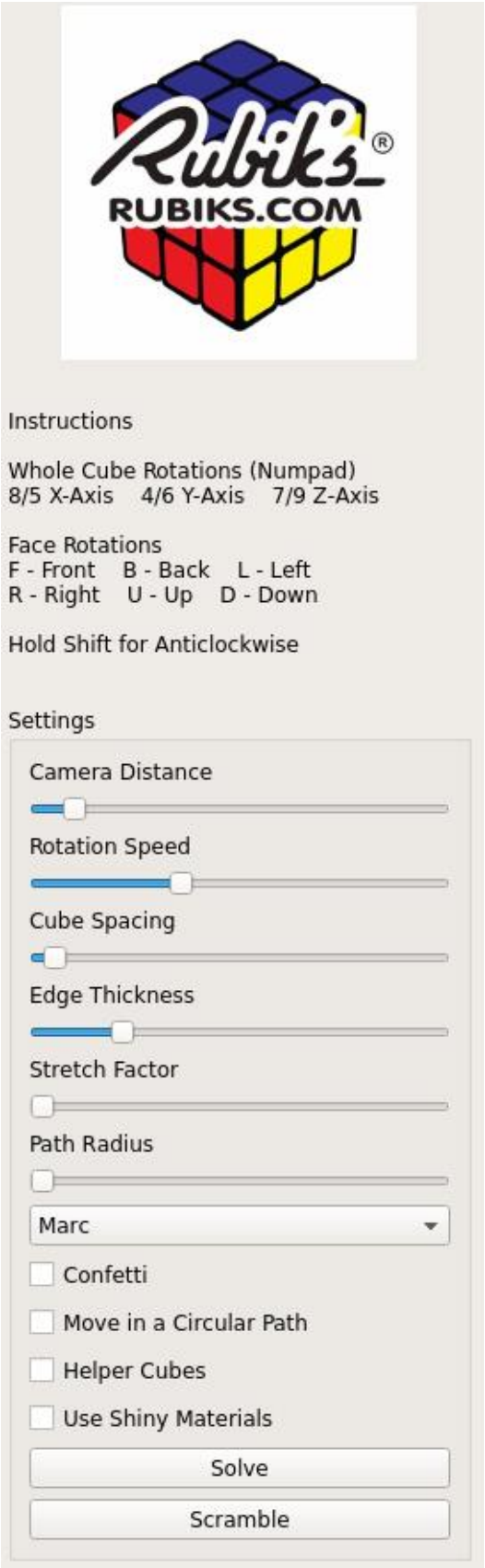


Figure 4: User Interface

Move in a Circular Path

Enables or disables the whole scene rotating

Helper Cubes

Enables or disables three extra cubes, these cubes show you the faces that you cannot see.

Use Shiny Materials

This changes, for all cubes in the scene, the colour scheme and material choice, making the materials the metal ones instead of the plastic ones.

Solve

This button solves the cube. This is nice in conjunction with Move in a Circular Path, as we can see the cube spinning as it solves it.

Scramble

This button scrambles the cube. 20 random moves are inputted. This can be done at any stage, so can stack them on top of each other.

The user interaction is setup with Qt layouts, a grid layout for the whole window and a box layout for the settings window. I also added the Rubik's cube logo in the top right. All of the inputs are connected to slots, and have their own functions to update or change the relevant values. All of these are very similar.

```
1.  // Rotation Speed Slider
2.  rotationSpeedSlider = new QSlider(Qt::Horizontal);
3.  rotationSpeedSlider->setSliderPosition(35);
4.  connect(rotationSpeedSlider, SIGNAL(valueChanged(int)), cubeWidget, SLOT(updateRota
    teSpeed(int)));
5.  settingsLayout->addWidget(rotationSpeedSlider);
```

For each of the UI elements, there is some code that roughly looks like this. Where we setup all the values. The initial slider positions are set to match the values in the initialiser functions at other points in the code.

```
public slots:
    void updateScene();
    void updateCameraDistance(int);  // Moves the camera further or closer
    void updateRotateSpeed(int);     // Makes each move go faster
    void updateCubeSpacing(int);     // Makes the cubies further apart
    void updateEdgeThickness(int);   // Makes the rounded edges thicker
    void updateStretchFactor(int i);  // Makes the cube more animated
    void updateRadius(int i);
    void toggleConfetti(bool toggle);
    void updateTexture(int index);
    void toggleRotateAboutCircle(bool toggle);
    void toggleHelperCubes(bool toggle); // will show or hide the helper cubes
    void toggleShinyMaterials(bool toggle); // will use different materials
    void solveCubeButton();          // Solves the cube
    void scrambleCubeButton();       // Performs 20 random moves on the cube
```

Each of these Qt UI elements are connected to a slot in the Rubik's widget. All of these functions are very similar to each other, and are different variations of changing simple variables.

```
void RubiksWidget::updateRotateSpeed(int i)
{
  // Have a mapping from 0 - 100
  // to the speeds we actually want
  // to make the slider appear more smooth
  vector<int> speeds = {1, 2, 3, 5, 6, 9, 10, 15, 18, 30};

  // This makes sure that the speed does not change
  // on this frame. But after the cube has completed
  // A full rotation, if we do it isntantly. We get
  // errors where the rotations overshoot, because
  // the speeds are not multiples of eachother
  for(unsigned int j = 0; j < cubes.size(); j++)
  {
    cubes[j].updatedRotateSpeed = speeds[i/10];
  }
}
```

This is one of the more complicated slot functions. Even this one is quite simple. This includes a mapping, going from the sliders $0 - 100$ to the different speeds available on the cube. We then need to iterate through every cube in the scene and update their speeds.

60% - 70%

Animation

This application has a few different animations. We have full cube rotations. Face rotations. As well as the animations on the confetti.

The cube animations were partially explained before. Where each whole cube axis is stored, and each face rotation is also stored. The animiation function gradually updates these until the animation has been finished, then the cubie positons and colours are updated.

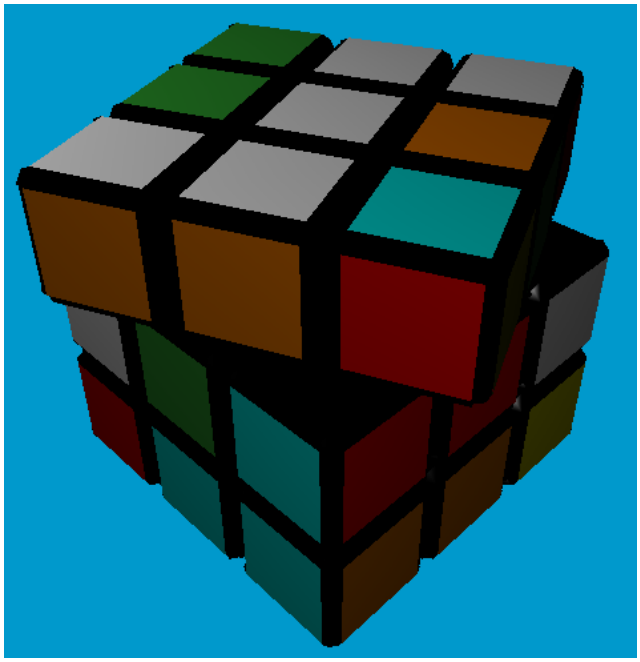


Figure 5: Standard cube rotation

```
1.  glPushMatrix();
2.    // Whole cube rotations
3.    glRotatef(this->cubeRotations[0], 1.0, 0.0, 0.0);
4.    glRotatef(this->cubeRotations[1], 0.0, 1.0, 0.0);
5.    glRotatef(this->cubeRotations[2], 0.0, 0.0, 1.0);
6.
7.  // Below the cubie rotations are done
```

The whole cube rotations are very simple. Before we do any of the face rotations, we just do the x, y and z ones, one at a time. This is done inside the cube class, so we can still rotate around the whole scene freely inside the drawGL function.

The animation function for each face is more complicated. It is contained in the cube draw function.

The draw function for each cubie is just done around the origin. So before we draw every face we have to translate by the x, y, z coordinates of the cubie. Then draw it.

```
1.  // Draw each cubie
2.   for(unsigned int i = 0; i < cubies.size(); i++)
3.   {
4.     glPushMatrix();
5.     // Put it in it's place by looking at the cubies x, y, z coords
6.     glTranslatef(space * cubies[i].x, space * cubies[i].y, space * cubies[i].z);
7.
8.     // Now we have the position sorted
9.     // draw it, making sure to pass the edge thickness
10.    cubies[i].draw(this->edgeThickness);
11.    glPopMatrix();
12.  }
```

For cubies that are currently under a rotation, there are more complicated things that need doing.

We loop through the cubies which are on the currently on a face being rotated.

```
1.  exaggerateMove = stretchFactor * sin(abs(faceRotations[j])* 3.14159/90) + space;
2.
3.  glRotatef(faceRotations[j], 0.0, 0.0, 1.0);
4.  glTranslatef(exaggerateMove * cubies[i].x,
5.                exaggerateMove * cubies[i].y,
6.                space * cubies[i].z);
```

We calculate the exaggerated move based on the stretch factor. We use the sin function to get a smooth exaggeration around the cube. The exaggerateMove only effects the translation, as translate further away as the sin function increases. The amount we rotate by is just whatever rotation amount is left on the face.

Once we have calculated the correct position and rotation for each cubie, we can simply draw it in place.

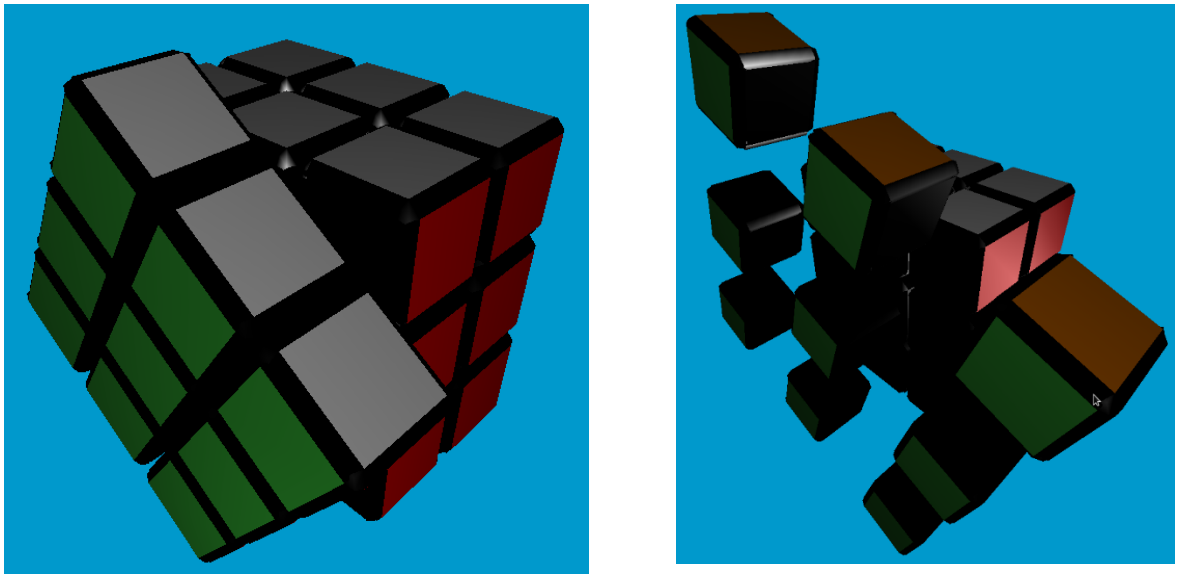


Figure 6: Non-exaggerated vs very-exaggerated move

Confetti

Animating the confetti was fun. Since they are all instances of a class, we can give them all random starting positions, rotation starts, speeds, ect. This makes the confetti appear more natural when they all have different falling speeds, there is much more variation.

For the colours, they are all initialised at random RGB values. Every frame the RGB values smoothly change as well. This effect is very fun, if you track one piece as it goes down, this smooth colour change can be seen.

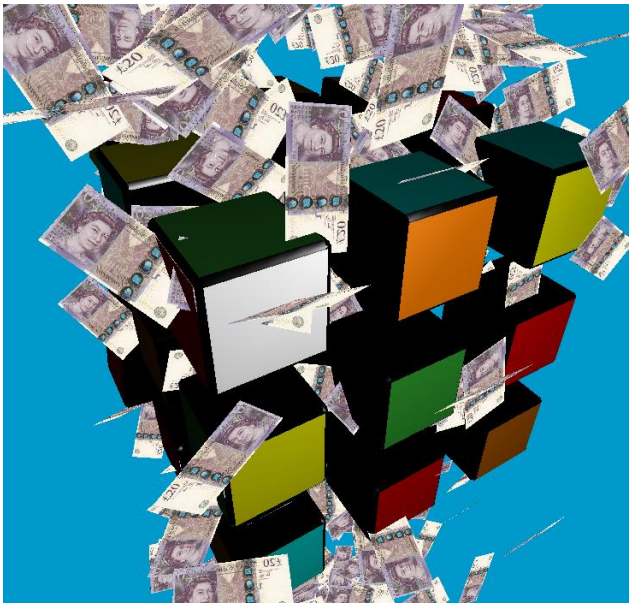


Figure 7: Money Confetti

```
1.  frame += 1; // Frame controls the position in the sin function
2.  chaos = chaos + randFloat(0.0, speed / 5.0); // Varies the speed
3.  y = y - chaos;
4.  zRot = 8 * frame; // Have a constant spin
5.  xRot = (sin((float)frame / (speed*1000.0)) * 90) + 90; // makes the nice flopping
6.  draw();
```

We have a frame counter which increments the sin function. Instead of the speed being linear, the speed is controlled by a separate variable which itself increases by random amounts. This makes not only the speed of each piece of confetti different, but the rates of change of each one different. This helps add more realism. The xRotation is based on a sin function, so they achieve the nice flopping motion that confetti has. Doing this at the same time as the zRotations achieves a nice effect.

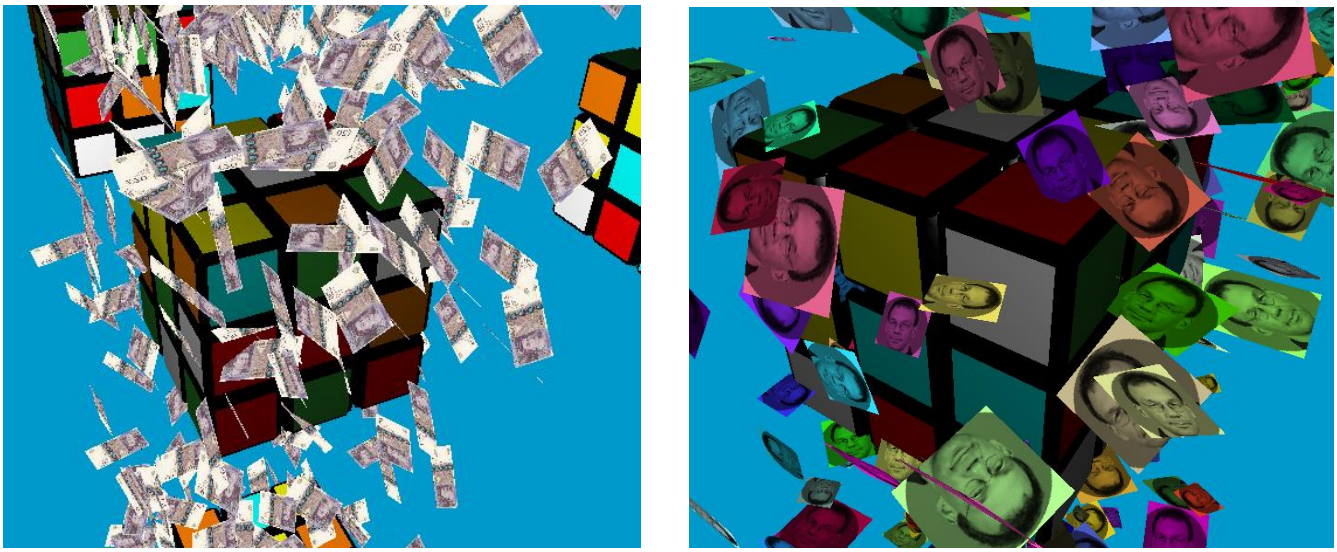


Figure 8: Money Confetti and Marc RGB Confetti

Object Constructed from Polygons

For my convex object constructed from polygons I chose a rounded cube. This fit the aesthetic very well. Since there 108 cubies, the smoothness of the spheres and cylinders had to be somewhat reduced for performance reasons. Increasing the poly count for the rounded corners and edges improves the quality of the reflections.

I enjoyed modelling these, instead of using static values for the sizes, I enjoyed using variables as the style of the cubes can be changed. Figure 9 shows some different values of the edge thickness.

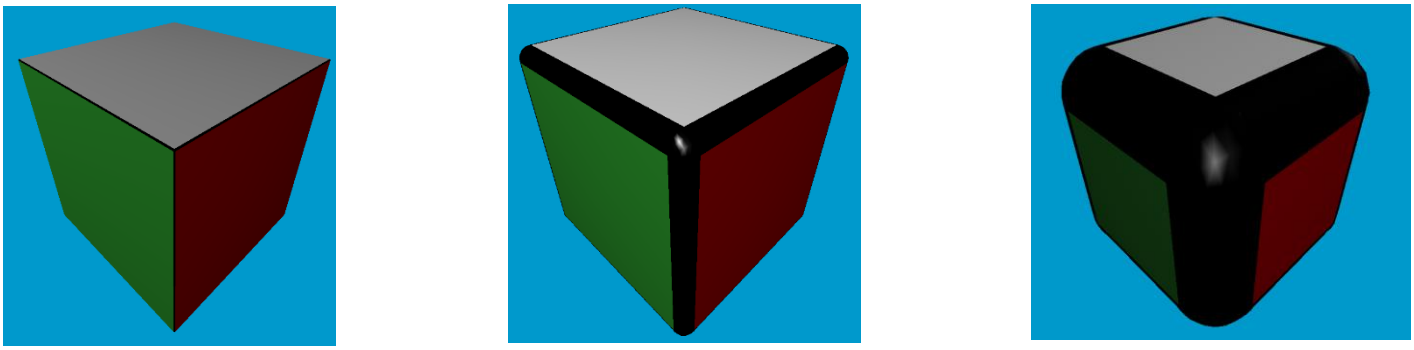


Figure 9: Differing edge thicknesses

```
glPushMatrix();
for(int i = 0; i < edgeCoords.size(); i++)
{
  // get into the right positon
  glTranslatef(size * edgeCoords[i][0],
               size * edgeCoords[i][1],
               size * edgeCoords[i][2]);
  // do rounded edge
  this->roundedEdge(gap);
  if(i < 4)
  { // Downward facing edges
    glRotatef(90, 1.0, 0.0, 0.0);
    this->roundedCorner(gap, size * 2); // rounded corner at the bottom
    glRotatef(-90, 1.0, 0.0, 0.0);
  }
  // Do the rotaiton, so the edge is oriented correctly
  glRotatef(-90 * (i + 1), 0.0, 1.0, 0.0);
  this->roundedCorner(gap, size * 2);
  glRotatef(90 * (i + 1), 0.0, 1.0, 0.0);
 }
  glPopMatrix();
```

To model these rounded cubes. We translate around to every corner and place a rounded sphere. We then perform a rotation to align the edge so that the direction is correct. As the edges get larger, the flat faces get smaller.

```
materialStruct* mat = &materials[front + materialsOffset];
glMaterialfv(GL_FRONT, GL_AMBIENT,    mat->ambient);
glMaterialfv(GL_FRONT, GL_DIFFUSE,    mat->diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR,   mat->specular);
glMaterialf(GL_FRONT, GL_SHININESS,   mat->shininess);

glNormal3fv(normals[2]);
glBegin(GL_POLYGON);
glVertex3f(-size, -size, size + gap);
glVertex3f( size, -size, size + gap);
glVertex3f( size,  size, size + gap);
glVertex3f(-size,  size, size + gap);
glEnd();
```

The flat faces have code to that of above. We find the material which this face has, assign it, then create a flat face with this material. The size variable is directly connected to the size of the edges. This means there is never an overlap, or any gaps, as we change the edge thickness.

Textures

The textures used are for the confetti. The textures are quite simple, using the Image.h and the Image.cpp provided. We simply need 3 functions in the main widget to update the currently used texture. When the dropdown is clicked these get updated.

Figure 10: Different confetti textures

70% - 100%

Hierarchical Modelling

The hierarchical modelling used has been described in the first section on the programming of the Rubik's cube. I am very happy with how this has turned out. The cube looks very accurate, it's enjoyable to play with all the settings and get different results. As far as all the hierarchical rotations on the cube, they all work correctly, it is nice to see the whole cube rotating while it is solving or scrambling.

Final Things

Here are some interesting settings you can get.

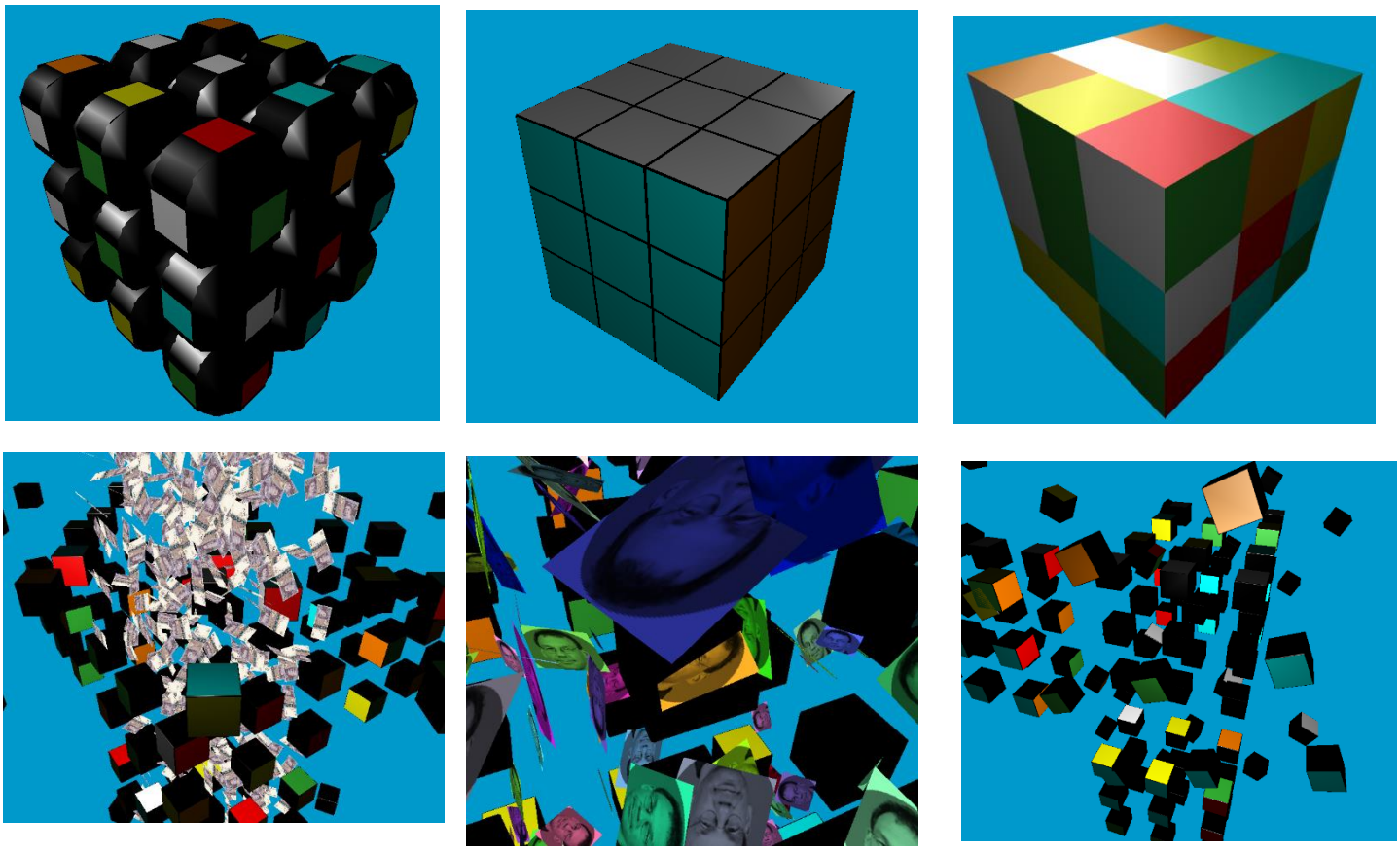


Figure 11: Interesting variations