Project2: Rubik's Cube

Chunxu Xu

Contents

1 Required Tasks

The goal of this project is to build a Rubik's cube. Figure 1 gives a snapshot of the sample program.

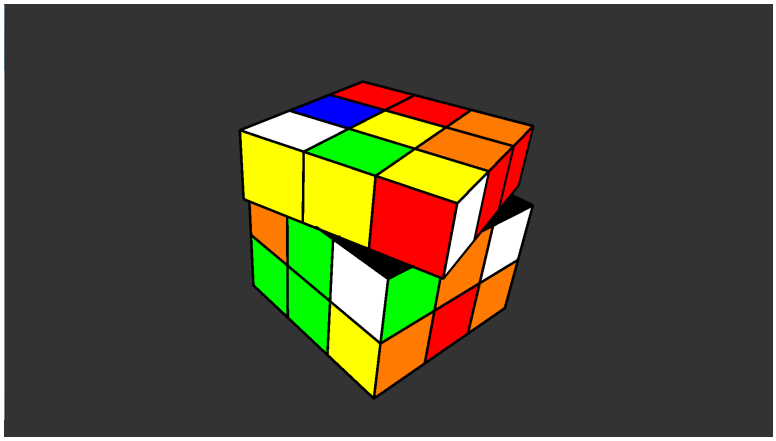


Figure 1: Overview of a Rubik's cube

This project mainly relates to the following knowledge about OpenGL:

- Model-View transformation
- Picking
- Interaction Design

This project concerns more about the data structure than rendering techniques. We prepared a framework for you with some kernel functions being left out[1]. You are free (and encouraged) to choose to start from scratch if you think that makes things easier. However, **some basic requirements are listed as follows**:

- The animation of rotation on Rubik's cube;
- A natural interaction method with mouse and keyboard, which including operations both on the whole cube and each rotating operation.

2 Detail Description

This section will give a detailed introduction to the framework.

[1] RubikCube_win.zip for Windows, and RubikCube_linux.tar for Linux. For the windows version, both solution and project file of MVS 2010 are also included.

2.1 Geometry Class

Vector3D.h gives a simple implementation of 3D vectors with some general vector operations such as addition, subtraction, dot/cross product and so on. Feel free to add or modify any part of this file as long as the interfaces and functions of the existing methods are kept the same.

2.2 Display

Before we go into the structure of the Rubik's cube, let's see the display function (*void myDisplay()*) first. It basically works as follows: For each rotating operation for the Rubik's cube, we rotate a small angle (*rotateSpeed*) each time, and after *rotateIteration* times the rotation is done. Three methods that the *RubikCube* class provides are used in this rendering part: *RotateStep()* rotates the small angle and accumulate it to the structure of Rubik's cube; *RotateFinish()* needs to be called when each rotation is done (explained in the next subsection); *Render()* is called twice to render the face and the border lines separately.

2.3 Structure of Rubik's Cube

RubikCube.h/RubikCube.cpp build the structure of a 3-order Rubik's cube. The majority of the methods for you to implement in this project is in *RubikCube.cpp*. There are mainly three classes:

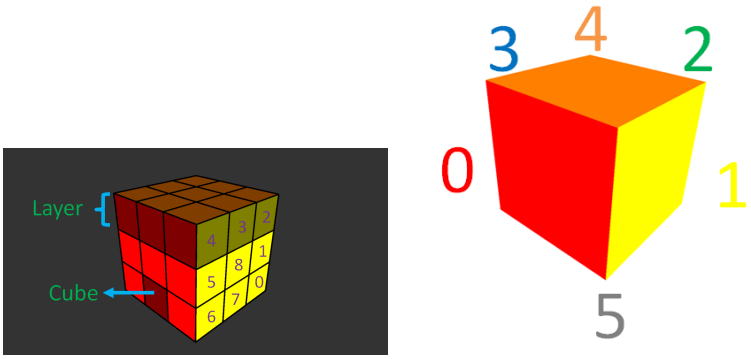
```
//the small sub-cube of the Rubik's cube
struct Cube;

//a structure containing a group of Cubes that can rotate together
struct Layer;

//Structure of the Rubik's cube
class RubikCube;
```

The structure of *Cube*, *Layer* and *RubikCube* are illustrated in Figure 2. The numbers are indices in *cubes* of *Layer*((a)) and in *faceColor* of *Cube*((b)).

Thus we can see that the *RubikCube* class consists of 27 *Cube* objects and 9 *Layer* objects. Since there is connection between *Cube* and *Layer* structure, we organize the *Cube*s as $3 \times 3 \times 3$ (*magicCube*), and the *Layer*s as 3×3 (*layers*). In details, the *magicCube* is organized in a relative position in the whole Rubik's cube, for example, *magicCube[0][0][0]* represents the left-bottom-far sub-cube, while *magicCube[2][2][2]* represents the right-top-near sub-cube. The *layer* has two dimensions: the first dimension represents which axis the layer follows when rotates, i.e. the layers *layer[0]* all rotate around x axis, *layer[1]* all rotate around y axis, and *layer[2]* all rotate around z axis; the second dimension gives the accurate layer (there are three



(a) Rubik's cube (b) Sub-cube

Figure 2: Illustration of *Rubik's Cube*

layers rotating around the same axis). Each *Cube* has its own position, colors of each face, and a matrix for the animation when it rotates. Each *Layer* contains all 9 *Cube* objects and the only axis that it can rotate around. The 9 *Cube*s are organized as the 8 ones around the center takes the position from 0th to 7th and the centered one is at 8th. See the code for details.

The most important parts in the *RubikCube* class are the methods it provides. We leave out three methods for you to implement.

- *void RotateStep(int x, int y, int dir)*
- *void RotateFinish(int x, int y, int dir)*
- *bool QueryLayer(const std::vector<TriInt>& select, int& layerX, int& layerY, int& dir)*

The first two ones are used in the rendering function in *main.cpp* and their functions are as we've mentioned above.

1. The first function should append a rotation matrix representing to rotate the angle of $90/rotateSpeed$ degree to each *matrix* of rotated sub-cubes. Note that all the Model-View operations are cascaded in a right multiplication way, but here obviously, we need a left multiplication.

2. The second function needs your more attention. There are two things to be achieved. First, we need to reset each *matrix* of sub-cube to be identity again. This can be easily done. Second, we need to change the appearance of the Rubik's cube to the state after rotation. You may think that we can just exchange the sub-cubes in the rotated layer. However, a direct assignment among the sub-cubes will expose the inner sides of them. Thus we need to set each face of the rotated sub-cubes manually.

3. The third function does a query on the layers. Given a list of selected sub-cubes (indicated by *select*, vector of indices of *magicCube*), the function should return the layer that will rotate (in *layerX* and *layerY*) and the

direction that the rotation will follow (in *dir*, either 1 or -1). Note that *select* is sorted by the order that the sub-cubes passed by the cursor. This function will decide how your program will react to the user's operation, so implement it carefully.

Besides these three methods, there are other methods in *RubikCube* used in rendering or assist you when you implement the above three methods. Please refer to the code to find more details.

2.4 Interaction

Intersection will play a significant role in the Rubik's cube project. The framework designs the operations on the Rubik's cube as follows:

- Left Mouse + Drag: Rotate the Rubik's cube as a whole.
- Right Mouse + Move: Mouse gesture to rotate a layer.
- 'z' key: Go back to last step.

You don't have to care about the third interaction which is completely done.

2.4.1 Select the sub-cubes passed by the cursor

The framework uses the picking provided by OpenGL to decide which sub-cubes are selected. For this part, we only need you to implement a function selecting the nearest sub-cube from one picking, i.e. the *GLuint findOutTheNearestCube(GLint hitsNum)* function in *main.cpp*. The framework collects all the names of the *Cube*s passed through by the cursor during the right mouse is pressed, and the parameter *hitsNum* is the number of hit objects (select buffer *selectBuffer* is a global variable). For more details, see the comment of this function.

2.4.2 Trackball

Figure 3 [2] is an illustration of trackball model we use to interact with the Rubik's cube.

Assume that we pass the screen coordinate (x, y) to the trackball, and it will try to calculate the 3D point (x, y, z) generated by mapping (x, y) onto the surface of the trackball. When the mouse moves to a new position, a new point on the surface (x', y', z') will be created, then the rotation we want to make is from the previous one to the current one. So, a cross product will be used to calculate the rotation axis, and the rotation angle can also be calculated easily (by cross/dot product).

[2] from Google image search.

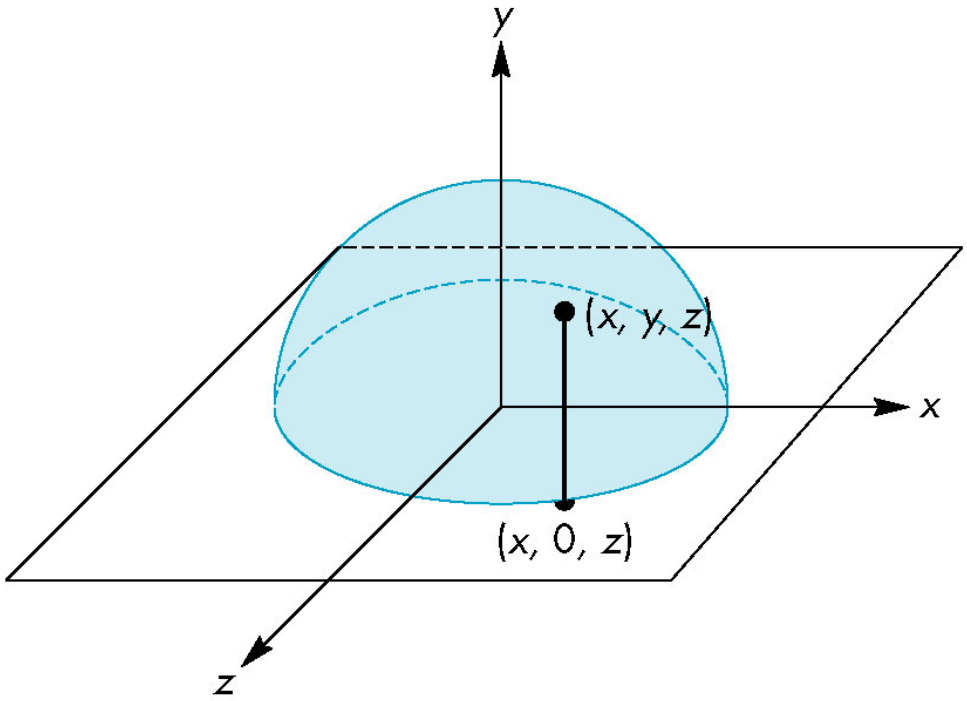


Figure 3: Trackball

Here comes another problem: If the object is not centered on the origin of the coordinate system in OpenGL, the rotation will be wrong — after all, we want the object (here it's Rubik's cube) to rotate around the center of itself. But to make the Rubik's cube visible, a shift is necessary along the z-axis. So, for the purpose of rotating object correctly, we must shift the object BACK to orgin, rotate it, then shift it to its original position. And to get the shifting vector, you need to get the Model-View matrix, get the 12th to 14th element (counted from 0) as x, y and z of the shift vector (because it's affine transformation under homogeneous coordinate, the right-most column represents the shift-operation).

For the framework, the following methods are left blank for you to implement:

- *void TrackBall::mapToSphere(const Vector2D& screen_v2d, Vector3D& v3d)*
- *void TrackBall::mouseMove(Vector2D v2d)*
- *void TrackBall::rotate(const Vector3D& axis, double angle)*
- *void TrackBall::translate(const Vector3D& shift)*

Note that the trackball is readjusted to fill the whole screen (so the track ball may be an ellipse), see the first few lines in function *mapToSphere()*.

3 Bonus

Any extra features can be considered for extra credits. The difficulty will determine how many extra grades you'll get. Please describe the features you implement in the final report in details.

Bellow are some suggested options:

- Pure color is relatively easy compared to use images as textures. If you try to use textures, note that the order of the vertices also need to be changed to get a right position after rotation. Try to use same picture on sub-cubes on the same face and different pictures on different faces. Even more, try to use a picture as the texture for a whole face.
- Try high order Rubik's cube. Note that the framework may be not suitable for this.