Modeling a Rubik's Cube in 3D

Robert Kaucic

Math 198, Fall 2015

1 Abstract

Rubik's Cubes are a classic example of a three dimensional puzzle thoroughly based in mathematics. In the trigonometry and geometry of a relatively simple collection of cubes, as well as in the combinations of rotations representing possible future states of the cube, one finds remarkably intriguing mathematical and graphical problems. For my project I modeled a Rubik's Cube in VPython and allowed users to manipulate it through a series of keyboard and mouse inputs. A key feature of my Rubik's Cube model is intuitive controls. Almost every model of a Rubik's Cube that you can find online has sloppy or confusing controls, making it harder to understand how to input commands than to actually solve the cube itself. Using combined keyboard and mouse input allows for the user to intuitively manipulate the Rubik's Cube.

2 Project Overview

All code for the project was written using only the standard VPython code library.

2.1 Modeling the Rubik's Cube

A physical Rubik's Cube consists of 26 small *cubelets* which each have one to three faces visible, depending on their location in the cube. Corner cubelets have three faces visible, edge cubelets have two, and cubelets in the center of one of the faces of the Rubik's Cube have only a single face visible. Each cubelet is connected to the cubelets adjacent to it, and the center-of-face cubelets are connected to a six-pronged rotator mechanism housed inside the Rubik's Cube itself. This structure, while ingenious for physical Rubik's Cubes, is unnecessarily complex for a computer-simulated Rubik's Cube. Instead, I created a 3x3x3 collection of cubelets ranging in positions from (-1, -1, -1) to (1, 1, 1), incrementing only in integers, that looks just like a real Rubik's Cube. The advantage to this was not needing to design the tricky rotator mechanism, but the drawback was abandoning the traditional method of connecting cubelets to one another.

VPython has a number of 3D objects as part of its standard code library. The one that seems most immediately useful is the *box* object. A box takes a position, a length, a width, a height, and a color as parameters, and creates a visible rectangular prism of the given color at the given position with the given dimensions. By passing length, width, and height of equal value into the constructor, the box appears to be a cube. The primary issue with using the box object, however, is that it can only have one color. Cubelets need to have up two three colors, depending on how many faces are visible, so the box object simply does not work. Instead, I used the *pyramid* class that is part of the VPython standard code library. The pyramid class takes a position, length, width, height, and color, and constructs a rectangular pyramid, where the rectangular base lies in the yz-plane and has dimensions length times width. The geometric center of the base is located at the given position, the vertex of the pyramid is located height units away from the geometric center in the positive x direction. The entire pyramid is the specified color. By putting six of these pyramids together such that their vertices all lie at the same point, but they are oriented in six different directions (the positive and negative x, y, and z directions), you can construct a cubelet where the base of each pyramid is one of the cubelet's faces. Since each of the pyramids, which now correspond to each of the cubelet's faces, can be colored differently, this structure can be used to create something that looks exactly like a cube but has different-colored faces.

I then defined the cubelet class to be a collection of six pyramids with square bases and heights equal to one half their side length, all with vertex at a single given position, oriented in six different directions and colored (red, green, blue, magenta, white, or yellow) based on their orientation during the initialization of the Rubik's Cube. The Rubik's Cube itself was then a simple 3x3x3 list of cubelets where each cubelet's position was set to be the cubelet's three-part index (i.e. [x][y][z] corresponds to the position (x, y, z)).

2.2 Controls

The controls are a series of keyboard inputs. The arrow keys move the camera position in 90°increments, to allow the user to view any side of the cube. The q, w, e, a, s, d, z, x, c keys are used for selection and rotation. The aforementioned keys q through c are arranged in a 3x3 grid on QWERTY keyboards. The locations of these keys in the grid correspond to the locations of the cubelets that they *select*; e.g. pressing q will select the top-left cubelet from the user's point of view. Selecting a cubelet "highlights" it (by lowering its opacity so it's visually differentiated from the other cubelets) and enables the second part of keyboard input needed to perform a rotation. For the second rotation, only the q, w, e, a, d, and x keys are enabled. These correspond to the direction that the selected cubelet (and the cubelets in its row, column, or face, depending on the rotation) will be rotated. w, a, d, and x are the most intuitive; w rotates the column of the selected cubelet up, x rotates the column of the selected cubelet down, and a and d rotate the row of the selected cubelet left and right, respectively. The q and e keys are for face rotations. Pressing q, regardless of the selected cubelet, will rotate the entire *front face* of the Rubik's Cube counterclockwise (front face being the nine cubelets that compose the side of the Rubik's Cube that the user is looking at). Pressing e works just like pressing q except that the rotation is clockwise.

2.3 Solving a Rubik's Cube

Solving a Rubik's Cube is considered a relatively daunting task for humans, since there are numerous complicated algorithms that one must remember in order to manipulate each cubelet into the desired position without detrimentally changing the position of other cubelets. Solving a Rubik's Cube is considerably harder for a computer; not because a computer has difficulty remembering complicated algorithms, but because a computer has trouble analyzing the state of a Rubik's Cube at each moment and determining the next algorithm to use in order to get some cubelet into the right position. Currently I have yet to implement the Rubik's Cube solver. I have, however, done some research and talked with my helpful roommate Orlando, who knows how to solve a Rubik's Cube, in order to learn how to implement a rudimentary Rubik's Cube solver. The algorithm is conceptually as follows:

1. Establish a list of states of the Rubik's Cube such that the nth state much be achieved before the *n+1*th state.

2. Identify the *important positions* of cubelets on the Rubik's Cube that contribute to the state (i.e. figure out what cubelets need to be in what positions in order to achieve the state).

3. For each important position, scan the Rubik's Cube to determine where the cubelet that needs to be in the important position under consideration currently is.

4. Once the current position of the desired cubelet is identified, check to see if any of the pre-written algorithms (which have certain preconditions for what beginning and ending positions they can accept as parameters) is capable of moving the desired cubelet from its current position to the important position under consideration.

5. If such an algorithm exists, perform it. If such an algorithm does not exist, continue.

6. Continue this process until each important position has been checked.

7. At this point either all of the important positions will have the correct cubelets in them or they will not. If the not all of the important positions have the correct cubelets in them, repeat steps 3 through 6 until all important positions have the appropriate cubelets. This may seem like it will not work, but according to my research and my roommate, if a cubelet is not in a position that works for the available algorithm(s) during the

first iteration of these steps, it will be in at least one of the subsequent iterations, due to the nature of the Rubik's Cube.[1]

Depending on how much time I have, the solver may have features such as a step-by-step solution, to allow the user to step through each move and follow the algorithms, or it may have a feature that reads the current state of the cube that the user is trying to solve and performs the next algorithm needed to place the correct cubelet in the next unsolved important position. At a minimum I intend to implement a feature that scrambles the Rubik's Cube and then plays out the solution without user input.

3 Performing Rotations

To actually perform a rotation of a set of cubelets in three-dimensional space, you must know the position of each cubelet, the axis of rotation, and the origin of rotation. When a rotation command is executed, it actually runs VPython's object.rotate(...) command on each of the faces of each of the cubelets. One of the key features of a rotation in three-dimensional space is the path that the object travels along, which is, for a Rubik's Cube, a quarter-circle. The axis of rotation is the coordinate axis (remembering that the way we built the Rubi's Cube obviously places its "center" at the origin) for which every cubelet being rotated has the same coordinate value for that axis (e.g. if you are rotating the set of cubelets with x-coordinate = 1, the axis of rotation is the x-axis). The origin of rotation specifies the center point of the circular path that would be taken if the cubelet were to be rotated a full 360°. Finally, the position of the cubelet is used to calculate the radius of the circular path that would be taken if the cubelet were to be rotated a full 360°. Using this information, VPython's object.rotate(...) method runs a series of *differential steps* to calculate frame-to-frame changes in the position and angular orientation of the object being rotated.

3.1 Math

There are a handful of relatively rudimentary calculations behind rotations for a Rubik's Cube.

Defining the axis of rotation:
Since the axis of rotation essentially exists to confine the three-dimensional rotation to a two-dimensional plane, all we are worried about is direction and orientation; magnitude is irrelevant. If we're rotating clockwise around the x-axis, for instance, it makes sense to set the axis of rotation to (1, 0, 0). The sign of the non-zero component effectively determines the direction of rotation.

[1]I understand this is somewhat vague. I fully intend to have a stronger and clearer grasp of the mechanics behind applying algorithms to a Rubik's Cube by the time of my final presentation.

Imagine that you are sitting at the origin and looking out along the axis of rotation. VPython's object.rotate(...) method will always rotate clockwise around the axis of rotation from that point of view. So to make a counterclockwise rotation about the axis (1, 0, 0) we simply set the axis of rotation to (-1, 0, 0).

Defining the origin of rotation:
Once again, due to the constraints of a Rubik's Cube, the origin of rotation of any given cubelet around some axis of rotation will simply be the origin, save for the cubelet's coordinate value for the non-zero component of the axis of rotation. So for a cubelet at position (0, 1, 1) rotating around the axis (0, 1, 0), the origin of rotation would simply be (0, 1, 0), since the cubelet's coordinate value for the non-zero component of the axis of rotation (the y-component, for the example axis) is 1.

Defining the radius of rotation:
Calculating radius of rotation, like the origin and axis of rotation, is simplified by the fact that rotations for a Rubik's Cube occur along coordinate axes. Step one is to determine the non-zero component of the axis of rotation. Step two is to perform a two-dimensional distance calculation between the cubelet's position and the origin of rotation using only the variables for which the axis of rotation's component for that variable equals zero.

For example: given a cubelet at (1, 1, 1) and an axis of rotation defined by (0, 1, 0), we know (from above) that the origin of rotation must be (0, 1, 0). We exclude the y component from our distance calculation, since the cubelet will be rotating in the y = 1 plane exclusively. So using the distance equation, and the notation that *cubelet.x* refers to the x-coordinate value of cubelet's position (with similar notation for the origin of rotatio):

$$d = \sqrt{(}(cubelet.x - origin.x)^2 + (cubelet.z - origin.z)^2) \tag{1}$$

We plug our values in:

$$d = \sqrt{(}(1-0)^2 + (1-0)^2) = \sqrt{(}2) \tag{2}$$

And get a very boring result. Since the actual numbers used in the construction of the Rubik's Cube are so simple, much of the math becomes rudimentary. You may notice that due to the way that the origin of rotation is set, it would be completely valid to use three-dimensional distance, since the third dimension will always cancel out. That would work just fine, but I believe illustrating the math in two dimensions (since that is all that is necessary) makes it somewhat easier to visualize.

4 References

www.vpython.org [No Author Given], accessed Oct 21, 2015
Orlando Melchor-Alonso: advice on solving a Rubik's Cube, Oct 21, 2015