Modeling, Controlling, and Solving a Rubik's Cube with VPython

Robert Kaucic

10 November 2015

Abstract

A Rubik's Cube is a fantastic toy and an equally fantastic geometric puzzle. Programming one graphically presents the challenge of developing a clear model, intelligent controls, and an effective solution method. For my project I model a Rubik's Cube in VPython and allow users to manipulate it through a series of intuitive keyboard inputs. A key feature of my Rubik's Cube model is the intuitive controls; almost every model of a Rubik's Cube readily available online has sloppy or confusing controls, making it harder to understand how to input commands than to actually solve the cube itself. Algorithmic solutions to the Rubik's Cube are quite difficult, so my attempt focuses on accuracy rather than efficiency.

1 Introduction and Backstory

In 1974, Hungarian professor of architecture Erno Rubik constructed the first Rubik's Cube in an effort to construct a model that would help him better understand three dimensional geometries. To better keep track of how the parts of the cube moved, he colored each face. Once he scrambled the cube, however, it took him an entire month to successfully re-solve the it. While the cube's value in understanding three dimensional geometry was limited, Rubik quickly realized the entertainment value of the cube as a toy and a puzzle. Four decades later, the Rubik's Cube is one of the most successful toys of all time. World Championships hold contests to solve the cube the

fastest, and countless hours have been dedicated to finding short, easy, and quick solutions to the scrambled cube [1].

Since the is a game borne from a geometric model, it is a prime candidate for being translated into an RTICA, especially for a novice graphical programmer thanks to its relative simplicity. The key components of my project are the model of the cube itself, the controls for the cube, and an algorithmic solver. Thanks to the nature of VPython, the modeling of the cube is relatively easy. The controls for the cube, however, are surprisingly complex. Finally, algorithmically solving the Rubik's Cube—especially when I don't actually know how to solve a real cube—is a fairly complicated and tedious process with many intricacies that must be handled perfectly to achieve the desired result. The bulk of the learning for the project will take place in the implementation of the solver.

2 Modeling the Rubik's Cube in VPython

All code for the project was written using the standard VPython code library.[1] No additional libraries were utilized.

2.1 Translating the Physical Cube into Code

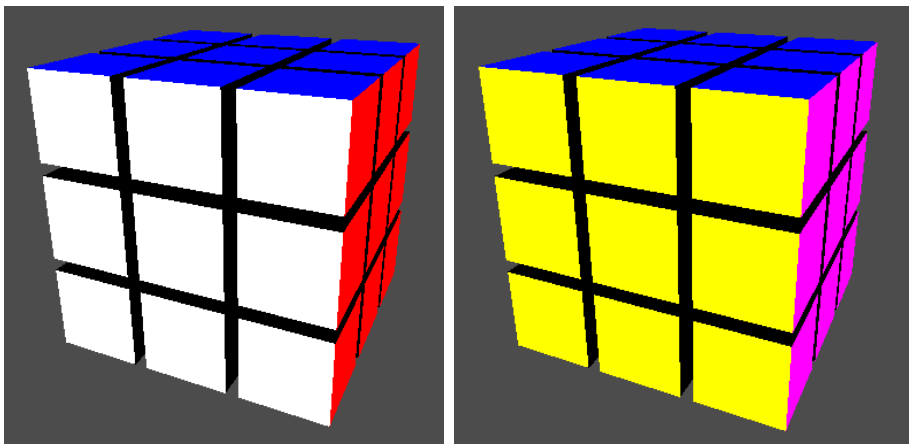
A physical Rubik's Cube consists of 26 small *cubelets* which each have one to three faces visible, depending on their location in the cube. A cubelet is a cube in which each face can be a different color. Corner cubelets have three faces visible, edge cubelets have two, and cubelets in the center of one of the faces of the Rubik's Cube have only a single face visible. Each cubelet is connected to a rotator mechanism housed inside the Rubik's Cube itself, and each cubelet maintains its position relative to other cubelets via its connection to the rotator mechanism.

This structure, while ingenious for physical Rubik's Cubes, is unnecessarily complex for a computer-simulated Rubik's Cube. Instead, I created a three-by-three-by-three collection of cubelets that looks just like a Rubik's Cube. The cubelets are instantiated with geometric centers starting at (-1, -1, -1) and going to (1, 1, 1), using only integer values for coordinates. The

[1]The standard VPython code library provides the basic features needed to create and manipulate three dimensional scenes. Full documentation is available at http://vpython.org/contents/docs/index.html

colors on each cubelet's face are hardcoded into the initialization of the cube, as shown in *Fig 1* below.

Fig 1: A visual of the Rubik's Cube model. The blue face in each picture is oriented in the positive-y direction. The green face (not shown) is oriented in the negative-y direction. In the left picture, the white face is oriented in the positive-z direction and the red face in the positive-x direction. In the right picture, the yellow face is oriented in the negative-z direction and the magenta face in the negative-x direction.



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.

2.2 Building Cubelets with VPython

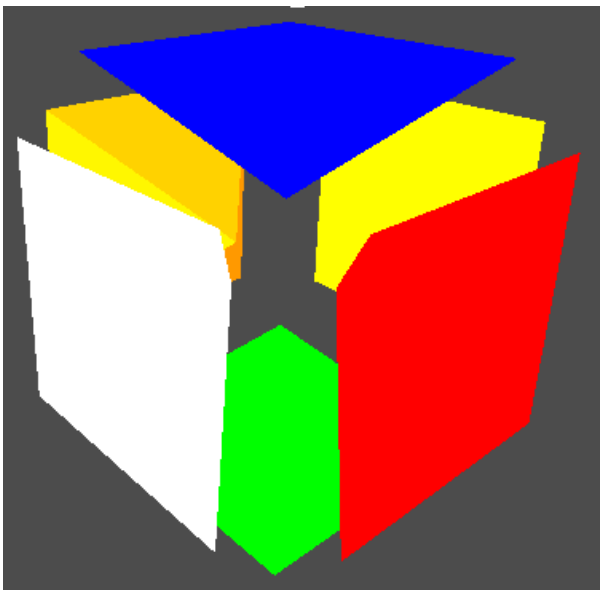
As mentioned before, a cubelet is a cube where each face can be a different color. VPython has a number of 3D objects as part of its standard code library, and unfortunately cubelet is not one of them. The object that seems most immediately useful is the *box* object. A box takes a position, a length, a width, a height, and a color as parameters, and creates a visible rectangular prism of the given color geometrically centered 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 problem 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 does not work. Instead, I devised a scheme for creating cubelets using the *pyramid* object that is part of the VPython standard code library. The pyramid object 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 by width. The geometric center of the base is located at the given position and the vertex of the pyramid is located height units away from the geometric center in the positive x direction by default. The entire pyramid is the specified color.

By putting six of these pyramids together such that their vertices all lie at the same point (which is the center of the cubelet), but they are oriented in six different directions (the positive and negative x, y, and z directions), I constructed a cubelet where the base of each pyramid is one of the cubelet's faces *(Fig 2)*. 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.

Fig 2: A visualization of the orientations of the cubelet faces. Note that in this image, vertices of the pyramids do not all lie at a single point. This was done intentionally for visual clarity, and does not accurately represent an actual cubelet.



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 $3 \times 3 \times 3$ list (analogous to a three-dimensional array in Java) of cubelets where each cubelet's center position was set to be the cubelet's three-part index (i.e. [x][y][z] corresponds to the position (x, y, z)), meaning that the cubelet center positions ranged from (-1, -1, -1) to (1, 1, 1), as stated in the "Translating the Physical Cube into Code" subsection.

2.3 Controlling the Cube

The controls are a series of keyboard inputs I designed after some basic testing for speed and intuitiveness. They are not perfect, but they are relatively simple and effective. 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,` and `c` keys are used for selection and rotation. The aforementioned keys q through c are arranged in a 3×3 grid on QWERTY keyboards. The locations of these keys in the grid correspond to the locations of the cubelets that they *select*; e.g. `(q)` will select the top-left cubelet from the user's point of view. Selecting a cubelet highlights it by lowering its opacity so that it is visually differentiated from the other cubelets.[2] 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

[2]It can be argued that "highlight" is a poor word for visual differentiation by lowered opacity, since highlight generally means increasing visibility, and decreasing a cubelet's opacity actually decreases its visibility by making it translucent. But a decrease in opacity makes it clear which cube has been selected and is quite easy to do. So in a sense it highlights the fact that the cube has been selected, if not the cube itself.

the user is looking at). Pressing `e` works just like pressing `q` except that the rotation is clockwise.

2.4 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, looked at a handful of resources including the official Rubik's Cube website's beginner solving technique [2] as well as some algorithms, such as Thistlethwaite's algorithm [3] designed specifically for computers, and talked with my helpful roommate Orlando [4] 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 must 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.[3]

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. One very simple way to do this would be to store the rotations used to scramble the cube, and perform those rotations in reverse order. Hopefully I am capable of writing a more legitimate algorithm than that.

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 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. For a single rotation in a Rubik's Cube, this path is a quarter-circle. The axis of rotation is the coordinate axis (remembering that the way we built the Rubik's Cube places its geometric 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

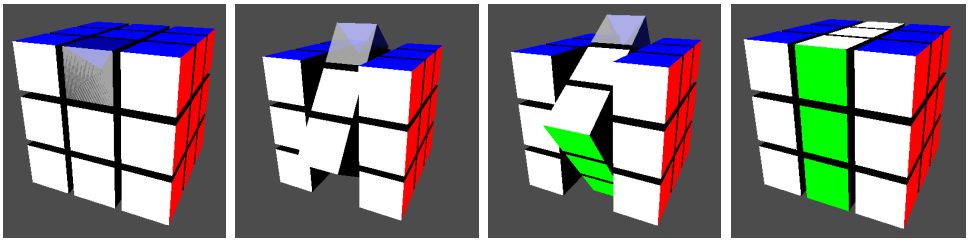
[3] I understand this is somewhat vague. I fully intend to have a stronger and clearer grasp of the mechanics behind applying algorithms to a Rubik's Cube by the time of my final presentation.

`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 calculates and sets the new orientation and position of the rotating object.

The `object.rotate(...)` method is convenient, but it is not actually applicable to cubelets. Cubelets are a collection of six pyramids, as well as a bit of other data, so VPython's `object.rotate(...)` method doesn't actually know how to rotate them. To handle this, I defined a `.rotate(...)` method in the cubelet class that class `.rotate(...)` on each of the pyramids in the cubelet. However, `object.rotate(...)` still does not animate the rotation of the cubelet, which is important for being able to keep track of where each cubelet on the screen has rotated to. In order to animate rotations, a complete 90°rotation of a row or column is actually split into 30 3°rotations, which are performed once every 30th of a second. The degree of each smaller rotation and the rate at which the rotations are performed can be changed, but for simplicity's sake I standardized them throughout my program.

Finally, it is worth mentioning why I did not use frames to perform the rotations. Simply put, each cubelet is not actually attached to the cubelets adjacent to it. Depending on whether you move a row or a column, two adjacent cubelets may move together or they may not. This functionality is not possible using frames, unless the frames are deleted and re-calculated before each rotation. As it happens, that is considerably more work than calculating the list of cubelets that would be in the frame and rotating them individually using the `cubelet.rotate(...)` method I wrote.

Fig 3: An example of a column of the Rubik's Cube being rotated up, around the axis `(1,0,0)`.



3.1 Mathematical Methods

There are a handful of relatively rudimentary calculations behind rotations. Principally, you must determine the center of rotation, the axis of rotation, the radius of rotation, and the angle of rotation. For a Rubik's Cube, these calculations are further simplified by the fact that the axis of rotation will always be one of the coordinate axes, and the angle of rotation will always be 90°.

3.1.1 Defining the Axis of Rotation

Since the axis of rotation essentially confines the three-dimensional rotation to a two-dimensional plane, all that matters for the rotation is the direction of the axis; the magnitude of the axis has no effect. If we rotate 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 determines the direction of rotation. 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).

3.1.2 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 be the origin, save for the cubelet's coordinate value for the non-zero component of the axis of rotation. So for a cubelet centered at position (0, 1, 1) (which means that the cubelet's `position` attribute would be (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.

3.1.3 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 rotation):

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

We substitute our values in:

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

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.

References

[1] "The History of the Rubik's Cube." *Rubik's.* Rubik's Brand Ltd, *n.d.* Web. 30 October 2015.

[2] Ferenc, Dénes. "How to solve the Rubik's Cube." *Ruwix.* Ruwix.com. 16 December 2012. Web. 3 November 2015.

[3] Scherphuis, Jaap. "Thistlethwaite's 52-move algorithm." *Jaap's Puzzle Page. n.p.* Web. 5 November 2015.

[4] Orlando Melchor-Alonso, personal communications, fall 2015.