VPython Rubik's Cube Narrative

Robert Kaucic

14 December 2015

Abstract

The goal of this project was to create an intuitively-controllable Rubik's Cube in VPython, as well as implement an algorithmic solver for the Rubik's Cube. The model was achieved by constructing a 3D object suited to the Rubik's Cube using objects native to the Python visual library. The controls were implemented through a series of keyboard presses roughly corresponding to the visual display of the Rubik's Cube. The solver was not achieved algorithmically, but rather through reversing the moves made to scramble the Rubik's Cube.

1 Introduction, Backstory, and Goals

Hungarian professor of architecture Erno Rubik created the first Rubik's Cube in 1974 while attempting to construct a tangible visualization of 3D geometries. Rather than a geometrical model, however, the Rubik's Cube became one of the world's most popular toys, spawning dozens of other twisty-puzzles and even a biennial world championship for solving Rubik's-esque puzzles. [1]

The primary goal of this project was to learn more about graphical programming. The secondary goal was to create a fun RTICA that was easier to use than other versions available online. By using VPython I ended up not learning much about the OpenGL graphical pipeline, but instead was able to create a higher-quality RTICA and became much more adept at using the more basic elements of graphical programming.

All code for this project was written using only the standard VPython code library. [1]

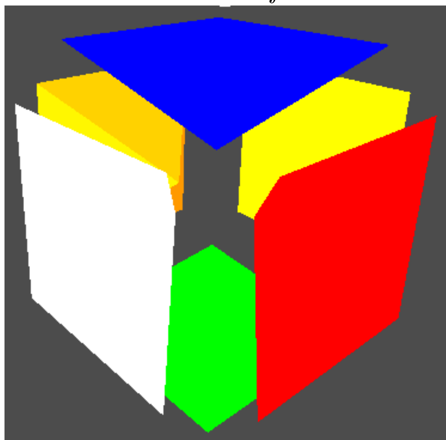
[1] The standard VPython code library provides the basic features needed to cre-

2 Modeling the Rubik's Cube in VPython

2.1 Building the Cubelet Class

In order to build a Rubik's Cube, one must have *cubelets*. A cubelet is a cube with up to six different-colored faces. On a standard Rubik's Cube, only one, two, or three faces are colored on any given cubelet. It just so happens that VPython's pre-built Pyramid class works quite well as a cubelet face, since six pyramids can be arranged as in *Fig 1*, barring the empty space between them, to make a cube.

Fig 1: A visualization of how six pyramids can be oriented to look like a cube. For the purposes of clarity, there is open space between the pyramids. This would not be the case for an actual cubelet.



2.2 Initializing the Rubik's Cube

When the Rubik's Cube is initialized, a triple-loop is used to iterate through the indices of the cubelets, and each cubelet is created with a list specifying which faces should be colored. In the cubelet's initialization method, the pyramids are created and colored (they are defaulted to black if no color is specified) and are rotated into the correct orientations depending on what color they are. This method does not allow for easy creation of cubelets at later states in the program, since the orientation of each of the colored faces

ate and manipulate three dimensional scenes. Full documentation is available at http://vpython.org/contents/docs/index.html

on the cubelets is based on the initial desired orientation of the Rubik's Cube. In order to place the cubelets in 3D space, a position value is calculated based on the values of the three loop variables in the Rubik's Cube's initialization method.

2.3 Properly Storing the Rubik's Cube Data

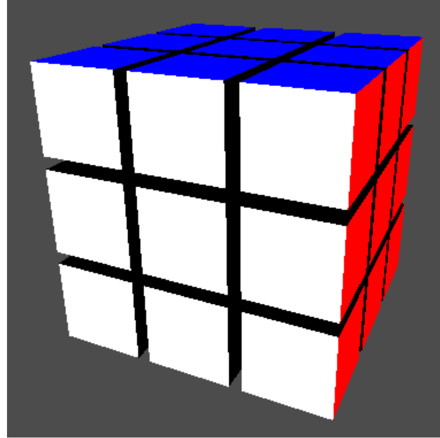
In order to properly store data about the Rubik's Cube (in particular; the cubelets), I used two triple-nested arrays. The first one holds references to each of the cubelets, specified upon creation, so that a given cubelet can always be found regardless of the state of the program. The second list updates after any move on the cube is performed, be it rotation of a side or of the camera, so that the cubelets in the list correspond visually to the cubelets on the screen. This is a time-saver for methods that involve locating a certain cubelet or generating a list of cubelets adjacent to a specific cubelet.

3 Controlling the Rubik's Cube

3.1 Building Intuitive Controls

The most important aspect of my controls are their intuitiveness. In light of the confusing, inconsistent, or outright nonsensical controls for many Rubik's Cubes currently available on the web, I made implementation of controls that do what they *look* like they should do a priority. Such design forces certain constraints on what exactly can be manipulated, however. When the user is allowed to freely rotate the camera around the cube, or when controls are implemented through the mouse, the user inevitably ends up with a laundry list of input sequences that they must memorize before they can intuitively use the controls, and frequently the controls work in unpredictable ways anyway. As such, I constrained the user's view of the cube to an off-centered view (*Fig 2*) that displays three sides at a time while making it clear which face the user is directly interacting with.

Fig 2: The standard view of the Rubik's cube, showing three faces, with one face being the obvious focus. The camera angle cannot be changed by the user, though the cube can be viewed from any side.

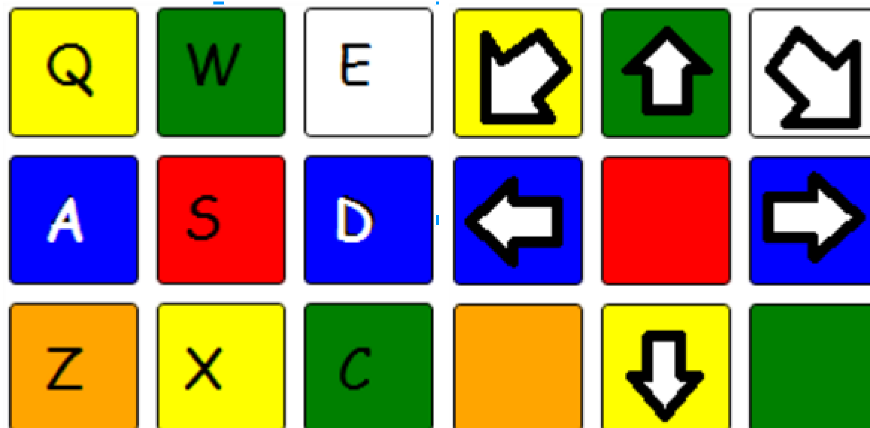


Ultimately, free rotation of the Rubik's Cube, as well as any mouse controls, were scrapped in favor of a simpler, more functional, system.

3.2 Taking and Processing User Input

The mechanism of accepting and processing input lies solely in VPython's built-in `scene.waitfor('keydown')` function. This well-named function halts action going on in the scene until the user has pressed some key down. In order to then process this input, the lines of code following `scene.waitfor('keydown')` determine which key exactly was pressed down, and executes the corresponding action. In some cases (e.g. the selection of a cubelet), this action is simply assigning a value to a variable. In other cases, a method is executed. If the type of input corresponds to a key press *(Fig 3a)* the input method enters a second stage of input processing. In this second stage, the mapping of keys to actions changes, so that keys that used to correspond to cubelet selection now correspond to performing a rotation *(Fig 3b)*. In both stages, a key that is not mapped to simply does nothing, returning to the beginning of the first stage of input accepting and processing.

Figs 3a (left) and 3b (right): In the first stage of input (3a), the shown keys select a cubelet. In the second stage of input (3b), the keys from before define a rotation axis and perform a move on the Rubik's Cube.



3.3 Performing a Move

A *move* is when a certain list of cubelets (either a column or a row of the Rubik's Cube) is rotated 90° around the center cubelet of that row or column, thus altering the state of the Rubik's Cube. The simpler definition is that a move is when you rotate a row or column of the Rubik's Cube. A move uses the input cubelet location (first stage input) and the axis (second stage input) to calculate the center of rotation, the list of cubelets to be rotated, and the direction of rotation. The implementation itself is quite simple: a method tells each cubelet in the list to rotate 3° around the center of rotation, repeated a total of thirty times. The cubelets in turn rotate by calling VPython's `object.rotate(...)` method on each of their pyramids. This keeps the cubelets in sync during their rotation. A challenge with this, however, is that a 90° rotation in 30 steps doesn't always result in clean position values. For example, a value that was supposed to be zero might end up being an almost-zero value because of rounding. To counter this, I used a normalization method that hard-set position values to the nearest logical value for a cubelet to be in. This kept the program from building up rounding errors on the cubelet positions over time.

3.4 Camera Rotations

A clever trick that simplified the controls a lot was implementing camera rotations via a move on the Rubik's Cube; when the user presses a key corresponding to a camera rotation, the standard method used to make a move is called with the entire Rubik's Cube as the list of cubelets to be rotated. This makes it look like the camera is rotating around the Rubik's Cube when in reality the entire cube is rotating. This has a handful of bonuses: it was very easy to implement, it simplifies the controls even more, and allows me to use absolute cubelet positions for calculation purposes, since it means the face that the user is interacting with is always in the same position.

4 The Solver and Utility Methods

4.1 The Planned Solver

The idea was to use a clear-cut, sequential method with predefined checkpoints and algorithms to establish a step-wise solution to the Rubik's Cube [2]. This is a beginner's method, so it is slow, but it also has the least overhead (in terms of how many algorithms and states one needs to have memorized) of any solution process [3]. As it turns out, the difficulties in writing even the relatively small amount of overhead, as well as in processing and evaluating the state of the Rubik's Cube and managing that data as the program acted on the cube, were too much for me. I was unable to develop a functioning solution method.

4.2 The Replacement Solution

In lieu of a proper solution to the Rubik's Cube, I accomplished moving the cube from an unsolved state to a solved one by reversing whatever moves were performed to bring the cube from its initial solved state to whatever unsolved state it was in when the solver was called. The program keeps track of the moves made in a list, and the solver method simply iterates backwards through this list, performing the moves in reverse order, and with a reversed axis (thus making the move backwards, instead of forwards again). The solver even returns the camera to its original position, since camera rotations are coded as moves. Additionally, the solver uses the number of moves that have

been made to calculate how fast it should undo the moves, so that a solution that takes a large number of moves occurs just as fast as one that takes only a few moves.

4.3 The Undo Method

This utility method is helpful in that it saves a bit of time. If a user accidentally makes a move, they can simply undo it with the press of a button. The method reads the last element of the list of moves made and performs the rotation with a negated axis before removing the element from the list. Due to minor differences in the data used in each method, the undo method isn't actually called in the solve method, but that could easily be changed with a minor tweak to each method.

4.4 Scramble

The scramble method uses lists of possible axes and cubelet positions to generate the data needed to make a move. It makes between 30 and 50 moves (decided randomly) and sequentially adds those moves to the list of moves made. Multiple scrambles can be done in sequence, and user-input moves can be made between any of the scrambles, and the solve method will still function appropriately.

4.5 Orbit

The orbit function is somewhat special. It is the only function that actually changes the position of the camera, rather than faking a camera rotation by moving the entire Rubik's Cube. This, coupled with the fact that the makeshift UI that goes with the Rubik's Cube was built on fixed positions, makes the orbit method screw up the UI. It's not really a problem since canceling the orbit function fixes everything, but it looks rather silly. The UI could be reworked to have a fixed location on the screen, not in 3D space, but I didn't have the resources to do so when I coded the UI. As noted in my documented code, the orbit function modifies the camera position using a function that Dr. Francis provided.

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