MYCUBES SPECIFICATION

Contents

Overview

The MYCUBES project is written using JavaFX 8 which is a fully integrated feature of the Java SE Runtime Environment (JRE 8).
The main two features used in the project are the Perspective Camera which applies perspective to the 3D shapes, and antialiasing which smooths out pixelated edges of shapes on screen.
The requirements for the project are:

- An N x N x N cube, where N is any value integer value greater than 1
- The cube should be able to twist in the same way a Rubiks cube does
- The cube must be rotatable by dragging mouse in the area outside of the cube
- Each of the 6 cube faces comprises of N x N square tiles which must each be able to launch a url/website via the users default browser if needed
- A menu bar is required that will house several buttons to aid in operating the cube
- A reset button is required to set the cube back to its original state
- An “Undo all” button, to undo all of the twists made, in a visual manner
- An “Undo” button to undo a single twist, in a visual manner
- A “Scramble” button to randomly scramble the cube in a visual manner
- A button to control the twist speed is required
- A button to let the cube spin is required
- Minimize, Maximize, and Close buttons should be implemented also
- The project window should remain square following resizing by dragging the mouse on its edges, except upon maximizing whereby the window should take up the entire screen.
- When viewed face on (as a square), the cube should take up 50% of its windows width and height, so as to give freedom to rotate the cube since the cube will take up more space when its displayed on an angle as opposed to face on
- The window should be able to have images for backgrounds, and also a border around it

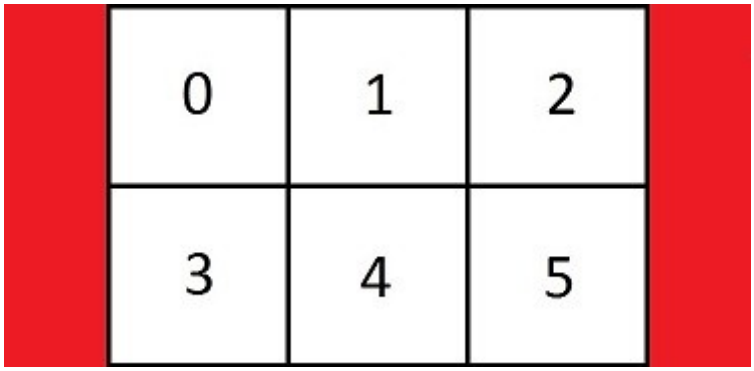
Media Content

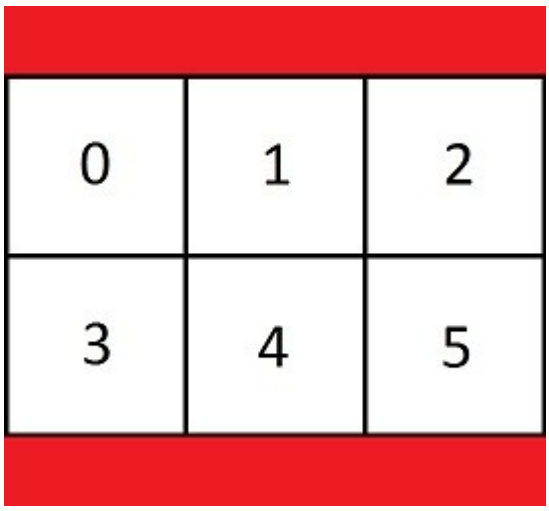
There are two types of content that can be applied to the cube, images (jpg/png) and movies (mp4).

Using more than one movie on the cube is possible, but due to performance issues, and the fact that the audio from multiple movies would clash, it is preferable to play only one movie at a time.

Forming the 3 x 2 rectangle from a single image / movie

Since a typical image or movie is rectangular, a maximally sized 3 x 2 rectangle can be formed inside it as shown:





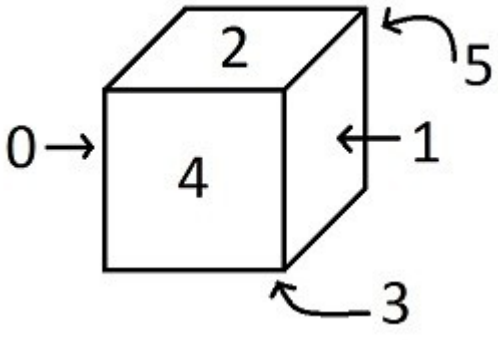
If the media is an image, it is cropped first, then displayed using an ImageView.
If the media is a movie, it is displayed using a MediaView object with a viewport in the shape of a 3 x 2 rectangle, since a movie cannot literally be cropped.

Forming the 3 x 2 rectangle from multiple images

6 individual images can be cropped square and made to form a 3 x 2 rectangle. Similarly, this is presented using a single ImageView object.

Now that the MediaView or ImageView has been constructed, it needs to be wrapped on to the cube in some suitable manner.

The image on the right shows the face numbering.



The default layout pairs the numbered squares in the MediaView / ImageView with the matching face on the cube.

Eg, the lower middle square (4) in the 3 x 2 MediaView / ImageView pairs with the front face of the cube, and in the same orientation.

When viewing movies on the cube, it may be desirable to maximise the number of edges whose two adjacent faces have content which naturally joins together at that edge. This maximal number of edges is exactly 4.

The images below illustrate how this is achieved:

Firstly split the MediaView / ImageView into 2 as shown.

Next the sides of each long rectangle are folded inwards to make two C shapes. These two C shapes join together to form a cube as can be seen below:

The CubeGroup Class

This class contains the cube, and is the central class in the program.

The main fields are:

private final Map<Integer, CubeletGroup> cubeletsMap

The cube is built from cubelets (See the CubeletGroup class), which have either 1 or 2 or 3 tiles, depending on if they are a middle cubelet, an edge cubelet, or a corner cubelet respectively.

The cubelets are positioned in a Cartesian fashion.
The X coordinate defines how far the cubelet is from the left going right.
The Y coordinate defines how far the cubelet is from the top going down.
The Z coordinate defines how far the cubelet is from the front going back.

In particular
The front top left cubelet has position (0,0,0)
The bottom back right cubelet has position (N-1, N-1,N-1),

Each cubelet has a fixed id number to make tracking the cubelets simpler.
The id number is defined by the cubelets original position:

$id = x + Ny + N^2z$

They are inserted into the map using this id as the key.

private final CubeletPositions cubeletPositions

The above map does not tell you where the cubelets actually are, it merely stores the cubelets themselves, allowing the CubeletPositions class to store the id at each (x,y,z) position.

For example: To retrieve the id for the top back left cubelet, call the method:

cubeletPositions.getID(0,0,N-1)

After each twist, the cubelets will be changing places, so this class is continually updated to keep track of where the cubelets are.

public LinkedList<Twist> twists

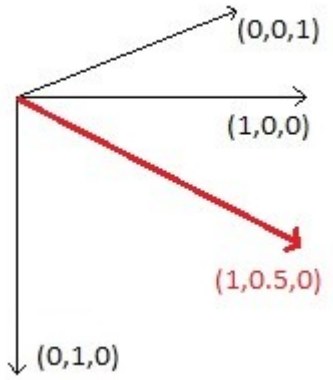
The Cubegroup has a list that stores all the twists that have been applied to the cube.

This allows for the undoing of twists, as twists are undone, the list gets smaller until it is empty.

public final RotBasis rotBasis

The cube uses an orthonormal vector basis to reflect its rotational position on screen.

The standard basis is represented by the matrix $B = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$



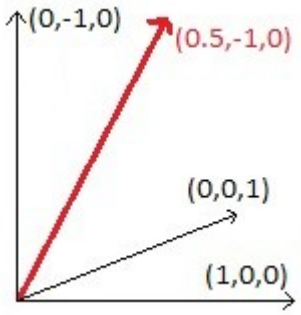
or by the system of 3 orthonormal vectors as shown on the right.

Consider the vector (1,0.5,0), multiplying this by the matrix above yields the same vector (1,0.5,0) as expected.

In mathematical notation B(1, 0.5, 0) = (1, 0.5, 0)

Suppose we want to rotate many vectors at once. Since the vectors “sit on top of” the vector basis, we need to just rotate the vector basis, and multiply each vector by this new basis to get the rotated vector.

Rotating the standard basis above around the z-axis by 90 degrees gives the new basis represented by $B = \begin{matrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{matrix}$ or by the system shown.



In particular multiplying (1,0.5,0) by this matrix yields (0.5,-1,0).

In mathematical notation B(1, 0.5, 0) = (0.5, -1, 0)

public TwistAnimTimer twistAnimTimer

This class extends the JavaFX AnimationTimer.

It loops around, and makes small changes to the cube every frame so that one sees a smooth animation.

Rotations: The rotation basis (as described above) gets modified here slightly during each animation frame, and subsequently you see the whole cube rotate, either manually or automatically.

Twists: Here the relevant cubelets that need to rotate are rotated around either the X or Y or Z axis, through up to 3 quarter turns.
The cubelets sit on the rotation basis, so when the cube is rotating and twisting simultaneously it is really small rotations on top of one large rotation.

The CubeletGroup Class

This class contains the tiles.

The main fields are:

private final int ID

This ID is given to the cubelet upon its instantiation, and it refers to the original (x, y, z) position of the cubelet.

Defined by $id = x + Ny + N^2z$

private final Rotate rx
private final Rotate ry
private final Rotate rz

These Rotates are only through the x or y or z axes, and through upto 3 quarter turns.

Since this class is a group, these rotations apply to all the children in the group together, so that the whole cubelet moves as one fixed piece.

These children are the tiles outlined below.

private int basisNumber = 0

To move a cube (say from one corner to an adjacent corner) can be done by updating its rotate angle, (from 0 up to 90 degrees).

But then to apply a second twist, it is awkward since there may not be a rotation that takes a cubelet from its original position to that of the endpoint of the 2^{nd} twist.

This has been solved by recognising that there are 24 unique vector bases.
To see this, consider the standard basis {(1,0,0), (0,1,0), (0,0,1)}.

Label the first vector as A, the second as B, last as C.
Each of A, B, and C start at the origin and have length 1.

There are 6 possibilities for A: (1,0,0), (-1,0,0), (0,1,0), (0,-1,0), (0,0,1), (0,0,-1).
Or in English: right, left, down, up, forwards(into screen), backwards(out of screen).

For each of these possibilities, vector B has 4 choices, leaving C with just 1 choice.
That gives 6 * 4 = 24 different vector bases.

Each cubelet begins with the basisNumber as 0.
Once a twist has occurred, we simply use a different basis number for that cubelet, and set its rotations back to 0.

The standard basis {(1,0,0), (0,1,0), (0,0,1)} is just one of them.

These bases are stored in the TurnUtils class, and are shown below:

```
public static final Vect[][] STANDARD_BASES = new Vect[][]
{
    new Vect[]{new Vect(1, 0, 0), new Vect(0, 1, 0), new Vect(0, 0, 1},
   new Vect[]{new Vect(1, 0, 0), new Vect(0, 0, 1), new Vect(0, -1, 0)},
   new Vect[]{new Vect(1, 0, 0), new Vect(0, -1, 0), new Vect(0, 0, -1)},
   new Vect[]{new Vect(1, 0, 0), new Vect(0, 0, -1), new Vect(0, 1, 0)},
   new Vect[]{new Vect(-1, 0, 0), new Vect(0, 1, 0), new Vect(0, 0, -1)},
   new Vect[]{new Vect(-1, 0, 0), new Vect(0, 0, 1), new Vect(0, 1, 0)},
   new Vect[]{new Vect(-1, 0, 0), new Vect(0, -1, 0), new Vect(0, 0, 1)},
   new Vect[]{new Vect(-1, 0, 0), new Vect(0, 0, -1), new Vect(0, -1, 0)},
   new Vect[]{new Vect(0, 1, 0), new Vect(1, 0, 0), new Vect(0, 0, -1)},
   new Vect[]{new Vect(0, 1, 0), new Vect(0, 0, -1), new Vect(-1, 0, 0)},
   new Vect[]{new Vect(0, 1, 0), new Vect(-1, 0, 0), new Vect(0, 0, 1)},
   new Vect[]{new Vect(0, 1, 0), new Vect(0, 0, 1), new Vect(1, 0, 0)},
   new Vect[]{ new Vect(0, -1, 0), new Vect(1, 0, 0), new Vect(0, 0, 1)},
   new Vect[]{new Vect(0, -1, 0), new Vect(0, 0, -1), new Vect(1, 0, 0)},
   new Vect[]{new Vect(0, -1, 0), new Vect(-1, 0, 0), new Vect(0, 0, -1)},
   new Vect[]{new Vect(0, -1, 0), new Vect(0, 0, 1), new Vect(-1, 0, 0)},
   new Vect[]{new Vect(0, 0, 1), new Vect(0, 1, 0), new Vect(-1, 0, 0)},
   new Vect[]{new Vect(0, 0, 1), new Vect(-1, 0, 0), new Vect(0, -1, 0)},
   new Vect[]{new Vect(0, 0, 1), new Vect(0, -1, 0), new Vect(1, 0, 0)},
   new Vect[]{new Vect(0, 0, 1), new Vect(1, 0, 0), new Vect(0, 1, 0)},
   new Vect[]{new Vect(0, 0, -1), new Vect(1, 0, 0), new Vect(0, -1, 0)},
   new Vect[]{new Vect(0, 0, -1), new Vect(0, 1, 0), new Vect(1, 0, 0)},
   new Vect[]{new Vect(0, 0, -1), new Vect(-1, 0, 0), new Vect(0, 1, 0)},
   new Vect[]{new Vect(0, 0, -1), new Vect(0, -1, 0), new Vect(-1, 0, 0)}
};
```

The map below is required to determine which base to use with a cubelet after a twist has occurred

```
public static final int[][] MAP = new int[][]
{
   //  1 2 -1
    new int[]{1, 2, 3, 21, 4, 16, 10, 6, 12},
   new int[]{2, 3, 0, 20, 7, 17, 11, 5, 15},
   new int[]{3, 0, 1, 23, 6, 18, 8, 4, 14},
   new int[]{0, 1, 2, 22, 5, 19, 9, 7, 13},
   new int[]{5, 6, 7, 16, 0, 21, 14, 2, 8},
   new int[]{6, 7, 4, 19, 3, 22, 15, 1, 11},
   new int[]{7, 4, 5, 18, 2, 23, 12, 0, 10},
   new int[]{4, 5, 6, 17, 1, 20, 13, 3, 9},
   new int[]{19, 12, 20, 9, 10, 11, 4, 14, 2},
   new int[]{16, 15, 23, 10, 11, 8, 7, 13, 3},
   new int[]{17, 14, 22, 11, 8, 9, 6, 12, 0},
   new int[]{18, 13, 21, 8, 9, 10, 5, 15, 1},
   new int[]{20, 8, 19, 13, 14, 15, 0, 10, 6},
   new int[]{21, 11, 18, 14, 15, 12, 3, 9, 7},
   new int[]{22, 10, 17, 15, 12, 13, 2, 8, 4},
   new int[]{23, 9, 16, 12, 13, 14, 1, 11, 5},
   new int[]{15, 23, 9, 0, 21, 4, 17, 18, 19},
   new int[]{14, 22, 10, 1, 20, 7, 18, 19, 16},
   new int[]{13, 21, 11, 2, 23, 6, 19, 16, 17},
   new int[]{12, 20, 8, 3, 22, 5, 16, 17, 18},
   new int[]{8, 19, 12, 7, 17, 1, 21, 22, 23},
   new int[]{11, 18, 13, 4, 16, 0, 22, 23, 20},
   new int[]{10, 17, 14, 5, 19, 3, 23, 20, 21},
   new int[]{9, 16, 15, 6, 18, 2, 20, 21, 22}
};
```

private final Tile tile0
private final Tile tile1
private final Tile tile2

There are 3 tiles, this is because some cubelets will require 3 tiles.
Cubelets that require 1 or 2 tiles will have the remaining tiles set as null.

The Tile class contains the single ImageView used in the tile.

Similarly to the vector bases, there are 24 possible orientations that an image can go in.
This is because there are 6 faces, and 4 orientations for each face, giving 6 x 4 = 24.

It takes at most 2 rotations and 1 translation to put each ImageView in place.

The main code for the procedure is shown below:

```
private Data getData(int orientation, int acQt, int x, int y, int z)
{
   switch (4 * orientation + acQt)
   {
      // ORIENTATION 0
     case 0: return new Data(Rotate.Z_AXIS, 0, Rotate.Y_AXIS, 1, -1, 0 + 0, -1, y + 0, -1, z + 1);
     case 1: return new Data(Rotate.Z_AXIS, 3, Rotate.Y_AXIS, 1, -1, 0 + 0, -1, y + 1, -1, z + 1);
     case 2: return new Data(Rotate.Z_AXIS, 2, Rotate.Y_AXIS, 1, -1, 0 + 0, -1, y + 1, -1, z + 0);
     case 3: return new Data(Rotate.Z_AXIS, 1, Rotate.Y_AXIS, 1, -1, 0 + 0, -1, y + 0, -1, z + 0);
     // ORIENTATION 1
     case 4: return new Data(Rotate.Z_AXIS, 0, Rotate.Y_AXIS, 3, 1, 0 + 0, -1, y + 0, -1, z + 0);
     case 5: return new Data(Rotate.Z_AXIS, 3, Rotate.Y_AXIS, 3, 1, 0 + 0, -1, y + 1, -1, z + 0);
     case 6: return new Data(Rotate.Z_AXIS, 2, Rotate.Y_AXIS, 3, 1, 0 + 0, -1, y + 1, -1, z + 1);
     case 7: return new Data(Rotate.Z_AXIS, 1, Rotate.Y_AXIS, 3, 1, 0 + 0, -1, y + 0, -1, z + 1);
     // ORIENTATION 2
     case 8: return new Data(Rotate.Z_AXIS, 0, Rotate.X_AXIS, 3, -1, x + 0, -1, 0 + 0, -1, z + 1);
     case 9: return new Data(Rotate.Z_AXIS, 3, Rotate.X_AXIS, 3, -1, x + 0, -1, 0 + 0, -1, z + 0);
     case 10: return new Data(Rotate.Z_AXIS, 2, Rotate.X_AXIS, 3, -1, x + 1, -1, 0 + 0, -1, z + 0);
     case 11: return new Data(Rotate.Z_AXIS, 1, Rotate.X_AXIS, 3, -1, x + 1, -1, 0 + 0, -1, z + 1);
     // ORIENTATION 3
     case 12: return new Data(Rotate.Z_AXIS, 0, Rotate.X_AXIS, 1, -1, x + 0, 1, 0 + 0, -1, z + 0);
     case 13: return new Data(Rotate.Z_AXIS, 3, Rotate.X_AXIS, 1, -1, x + 0, 1, 0 + 0, -1, z + 1);
     case 14: return new Data(Rotate.Z_AXIS, 2, Rotate.X_AXIS, 1, -1, x + 1, 1, 0 + 0, -1, z + 1);
     case 15: return new Data(Rotate.Z_AXIS, 1, Rotate.X_AXIS, 1, -1, x + 1, 1, 0 + 0, -1, z + 0);
     // ORIENTATION 4
     case 16: return new Data(Rotate.Z_AXIS, 0, Rotate.X_AXIS, 0, -1, x + 0, -1, y + 0, -1, 0 + 0);
     case 17: return new Data(Rotate.Z_AXIS, 3, Rotate.X_AXIS, 0, -1, x + 0, -1, y + 1, -1, 0 + 0);
     case 18: return new Data(Rotate.Z_AXIS, 2, Rotate.X_AXIS, 0, -1, x + 1, -1, y + 1, -1, 0 + 0);
     case 19: return new Data(Rotate.Z_AXIS, 1, Rotate.X_AXIS, 0, -1, x + 1, -1, y + 0, -1, 0 + 0);
     // ORIENTATION 5
     case 20: return new Data(Rotate.Z_AXIS, 0, Rotate.Y_AXIS, 2, -1, x + 1, -1, y + 0, 1, 0 + 0);
     case 21: return new Data(Rotate.Z_AXIS, 3, Rotate.Y_AXIS, 2, -1, x + 1, -1, y + 1, 1, 0 + 0);
     case 22: return new Data(Rotate.Z_AXIS, 2, Rotate.Y_AXIS, 2, -1, x + 0, -1, y + 1, 1, 0 + 0);
     case 23: return new Data(Rotate.Z_AXIS, 1, Rotate.Y_AXIS, 2, -1, x + 0, -1, y + 0, 1, 0 + 0);
     default: throw new IllegalStateException("INVALID VALUE");
   }
}
```

Initiating a twist

When a press inside the window is made, the precise face and tile that was pressed on (if any) has to be determined.

We can construct a mathematical object to represent the cube, to enable 2D mouse movements to extend into 3D movements on the cube

The coordinates (or vectors) of the 8 corners of the cube are: $(\pm W/2, \pm W/2, \pm W/2)$
where W is the width of the cube.

It is then a simple task to find the 4 vectors for the corners of each tile for each of the $6N^2$ tiles.
For a specific tile, they can be labelled c1, c2, c3, c4

For each set of 4 corner vectors, we multiply each of the 4 vectors c1, c2, c3, c4, by the Cube Basis (see The Cube Basis section), to give us the actual vectors given due to the cubes rotation.

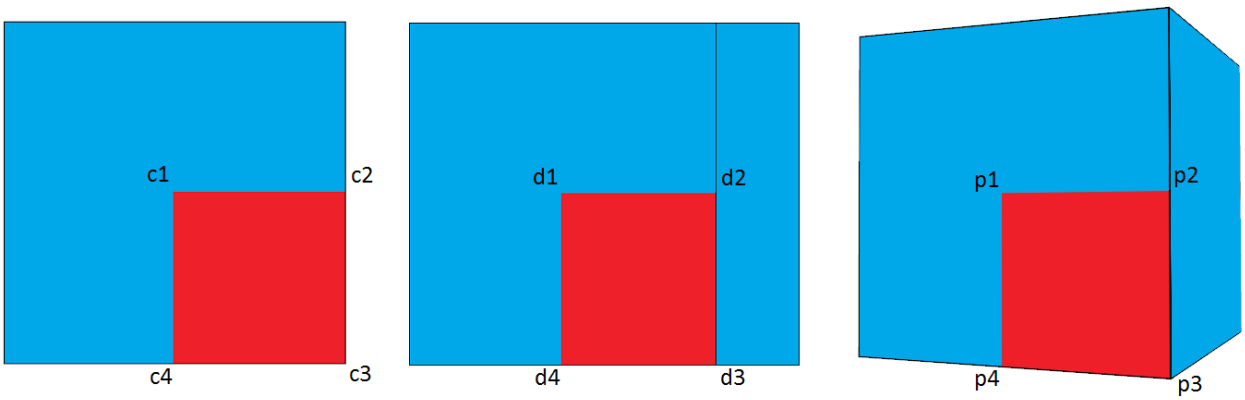
Hence d1 = Bc1, d2 = Bc2, d3 = Bc3, d4 = Bc4

Next, the perspective camera actually deforms an object so the end nearest to the viewer looks larger than the end furthest away. So the mathematical object must also be deformed:

A vector (x, y, z) transforms to ($\frac{d}{d+z+f}$ x, $\frac{d}{d+z+f}$ y, z) where d is the depth of

the scene, and f is the distance forward that the cube is.

Applying this to the vectors d1, d2, d3, d4 gives the “deformed” vectors p1, p2, p3, p4.



Finally, before we can use the vectors p1, p2, p3, p4, we have to determine if the face is pointing towards the user (out of screen), or away from the user (into the screen).

Taking the cross product $N = (p2 - p1) \times (p4 - p1)$ gives a normal vector to the tile’s plane.

If the z component of N is positive, then the tile is visible to the user, and a polygon can be formed from p1, p2, p3, p4, by ignoring their z components.

If the mouse press lies inside this polygon, then it is known which tile was pressed on.

Generating the twist

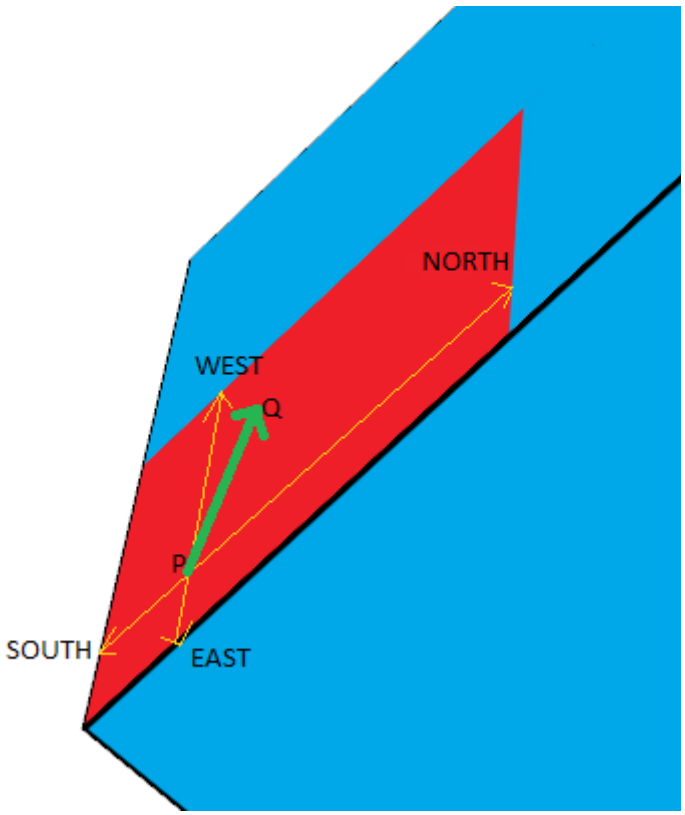
Once a twist has been initiated as shown in the last section, the program waits for the user to drag the mouse past a set radius from the original press point, or release the mouse to cancel the twist.

Given that the mouse has been dragged sufficiently far, there are 4 possible twists:

In the diagram P is the original press point and Q is a point far enough from P to trigger a twist.

The mouse drag line shown is in the North-West Quadrant.

In the diagram, the drag line is nearer to West than to North, so the twist must occur in the West direction.



Applying the twist

This is best illustrated with an example.

Consider the cube shown (with N = 3) in its initial state.

The red cubelet is in position (2,2,0), giving the cubelet the id = 2 + 3*2 + 9*0 = 8.
Note the basisNumber is 0 for this cubelet as all cubelets begin with basisNumber 0.

A Manual twist is always 90 degrees, if 180 degrees is required, two 90 degree twists are made.

Applying a twist (about y-axis, 90 degrees anticlockwise):

During the animation, the basisNumber is unchanged.
While the Rotate ry variable increases from 0-90.

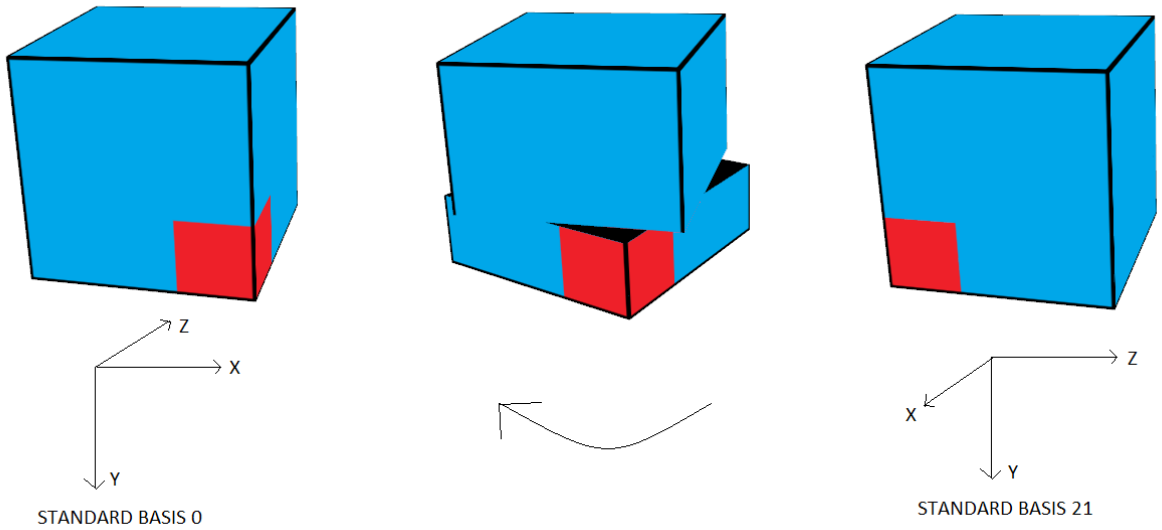
Finalising the twist:

All the cubelets involved will get new basisNumbers via the code:

basisNumber = TurnUtils.MAP[basisNumber][(qt - 1) + 3 * axis]; [qt = 1, axis = 1]

basisNumber = TurnUtils.MAP[0][(1 - 1) + 3 * 1]
= TurnUtils.MAP[0][0 + 3]
= TurnUtils.MAP[0][3]
= new int[]{1, 2, 3, 21, 4, 16, 10, 6, 12}[3]
= 21

Now, STANDARD_BASES[21] = {(0, 0, -1), (0, 1, 0), (1, 0, 0)}



It is evident that standard basis has updated to reflect the quarter turn about the y-axis.

Thus using standard basis 21, the red cubelet (id = 8) ends up in the bottom front left corner.

Reset button

The reset button does a complete rebuild of the cube.

If changing the N value, the cube has no other option that to completely reset, since any twists applied, will no longer be applicable.

For example changing from a 3 * 3 * 3 cube, to a 4 * 4 * 4 cube, it is impossible to preserve any twists that may have been previously applied.

Resetting the cube is equivalent to updating the cube but with the same value of N applied.

Undo all button

The cubegroup contains a list of all twists played (see The CubeGroup class).
Pressing this button takes each twist one at a time, from the end of the list, reverses its direction, and then plays it on the cube.

Once the list is empty the cube will be in its original state.
It is impossible to twist the cube while it is undoing, however a repress of the button will stop the process allowing for normal usage.

Undo button

This is identical to the Undo all button, however it just undoes the last twist applied assuming at least one twist has been applied.

Scramble button

The Scramble button generates twists at random. It does it in such a way that the axis of rotation differs from one twist to the next, to ensure the cube is highly scrambled, which may not be the case if same axis twists are played consecutively.

An N x N x N cube is essentially built from N slices along an axis. A twist generated by the computer allows for multiple slices to twist simultaneously.

This is not the case for manual twisting by the user since the user cannot physically start two or more twists occurring at the same time, there would be a delay between them. Hence user twists are registered as one slice for one twist.

Rotate button

The rotate button simple spins the cube.
This is achieved by rotating the CubeGroup rotBasis about an axis that also changes duing the animation process.

The rotate functionality is independent of all the other buttons, so it can be applied during a scramble or an ‘undo all’ for example.
Holding the mouse on the cube background as with manual rotation will override the automatic rotation until the mouse is released. This is mandatory to avoid manual rotation fighting against automatic rotation which makes the cube shaky which is undesirable.

Speed button

This button is really two buttons in one, use – to reduce twist speed and + to make it faster.

There are 140 speeds available, the fastest is 1 and the slowest is 554.
The speed value is defined as the number of frames of animation required to do 1 quarter turn.

So when the speed is on 1, there is no movement of slices, and the cube is always cube shaped.

Minimize / Maximize buttons

The minimize button minimizes the cube like any normal window.

To resize the cube, one can drag on its border. This always keeps the window square.

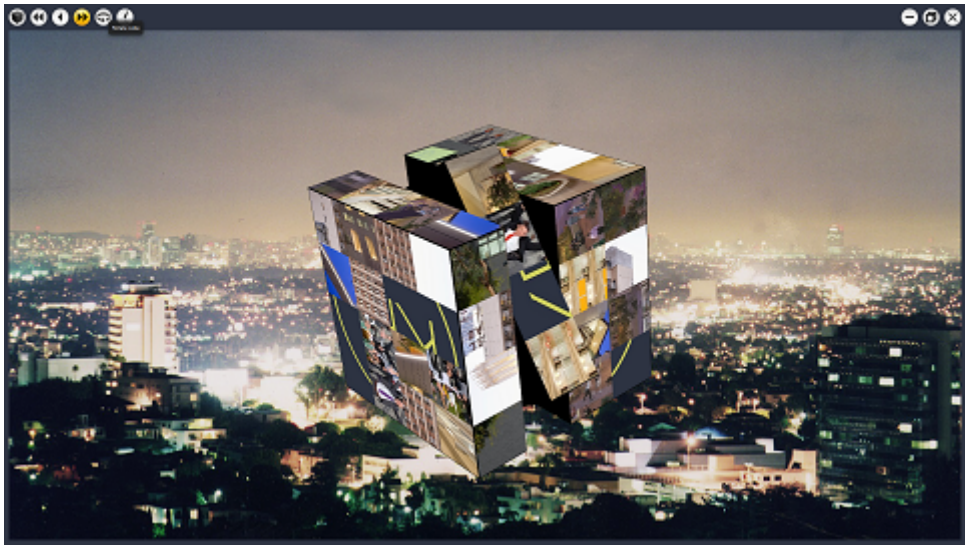
The exception is when it is maximized, the cube’s relative size is then based on the height of the screen (since this is typically smaller than the width).
Repressing the maximize button or double-clicking on the bar will cause the window to restore down again.

Screenshots

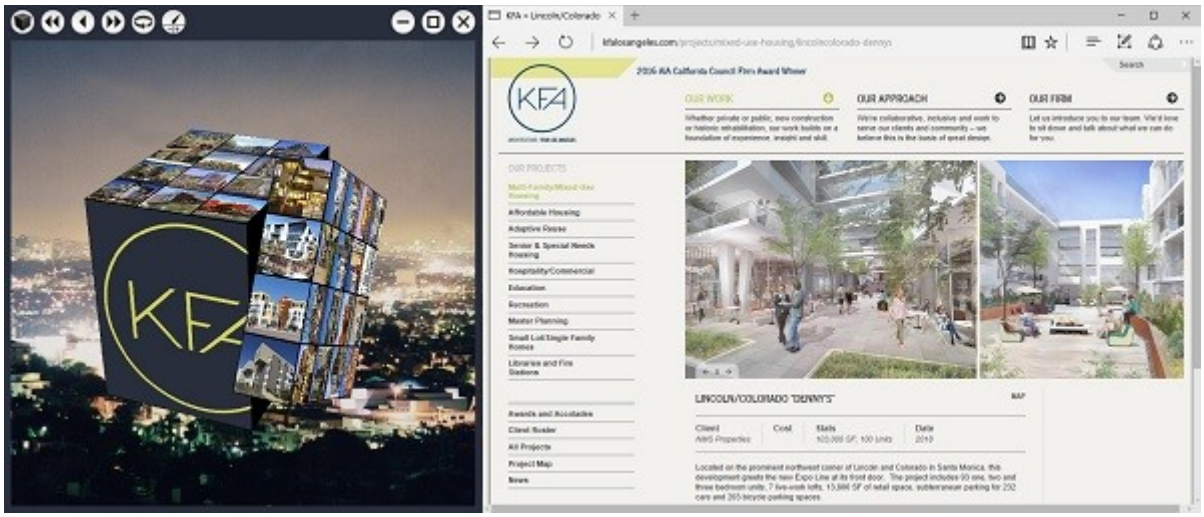
This screenshot shows manual twisting being applied.

Multiple slices can be twisted at the same time provided they they are twisting about the same axis obviously.

This screenshot shows a cube being scrambled at full speed



This screenshot is of the cube maximized, showing LA at night in the background.



This screenshot demonstrates the cube launching a website following a double click