Virtual Rubik's cube

Wen-bin Wu
Virtual Rubik’s Cube  by Wen-bin wu
May, 2K

wxw2405@rit.edu
Rochester Institute of Technology
A thesis submitted to the faculty of the college of Image and Arts and Science
in candidacy for the degree of master of Fine Arts

Virtual Rubik's Cube by Wen-bin Wu
May, 2000
I, Wen-bin Wu, hereby granted permission to the Wallace Memorial Library of RIT to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 1</td>
<td>INTRODUCTION TO VIRTUAL RUBIK'S CUBE</td>
<td>01</td>
</tr>
<tr>
<td>Chapter 2</td>
<td>GEOMETRICAL TRANSFORMATION</td>
<td>03</td>
</tr>
<tr>
<td>Chapter 3</td>
<td>OBJECT HIERARCHY AND MODELING</td>
<td>06</td>
</tr>
<tr>
<td>Chapter 4</td>
<td>DATA FORMAT AND ORGANIZATION</td>
<td>09</td>
</tr>
<tr>
<td>Chapter 5</td>
<td>OBJECT ORIENTATION</td>
<td>13</td>
</tr>
<tr>
<td>Chapter 6</td>
<td>THREE DIMENSION ENGINES</td>
<td>19</td>
</tr>
<tr>
<td>Chapter 7</td>
<td>PARENT SCRIPTS REVIEW</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>NOTE</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>BIBLIOGRAPHY</td>
<td>34</td>
</tr>
</tbody>
</table>

**APPENDICES**

<table>
<thead>
<tr>
<th>Appendix</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appendix A</td>
<td>PARENT SCRIPT - PanelScript</td>
<td>35</td>
</tr>
<tr>
<td>Appendix B</td>
<td>PARENT SCRIPT - CubeScript</td>
<td>39</td>
</tr>
<tr>
<td>Appendix C</td>
<td>PARENT SCRIPT - EightCubeScript</td>
<td>42</td>
</tr>
<tr>
<td>Appendix D</td>
<td>FRAME SCRIPT - Frame4Script</td>
<td>45</td>
</tr>
<tr>
<td>Appendix E</td>
<td>MOVIE SCRIPT - PrepareMovie</td>
<td>47</td>
</tr>
</tbody>
</table>
This thesis is to present three-dimensional information to the viewer with a two-dimensional application. Human beings deal with a three-Dimensional environment on a daily basis. It is human nature for people to process information from a three-dimensional aspect.

For a human being, presenting information can be achieved by using language verbally or orally. Originally, the verbal language is used to reproduce the vocal presentation of verbal language. The invention of letters and giving fixed pronunciation to each letter becomes the bases of reproduction of sound graphically. Later, by combining letters as words and combining words as sentences, verbal information presentation is accomplished. What people write is exactly the same as what people speak.

Using language, however, puts restrictions on transmitting information because groups agree to compose on language in certain ways. This means, in order to transcend ideas freely, the message givers and receivers must stand on the same ground. They share knowledge and common sense at a certain level. Based on the shared knowledge, the receiver is able to understand the conceptual idea with the clues provided by the information givers. Without these same bases it is difficult to transmit new information, this is the fatal decade of language.

Strictly speaking, verbal language is considered image information.

"The capacity for language is a capacity not only to response to symbols but to create them - for language is symbolic behavior." -- Walter Goldschmidt, Man's way

The "symbolic behavior" reveals the true property of graphic information. Notice that not only language can describe the messages and the images themselves can also present the concepts too. Pictures are used to compensate for inefficiencies of verbal presentation.

Certainly, one single picture sometimes is not suitable for explaining all situations. Picture presentations are based on two-dimensional information. For describing a three-dimensional object, approaching the object from various angles is necessary. Here, multiple pictures are brought in to solve the problem. Depending on the human being's three-dimensional instinct, people exam the different pictures, finding the relationships of these pictures from assorted aspects. Consequently, people are able to visualize the whole object and the information that the object itself possessed.
The object chosen for this thesis study is the Rubik's cube, which is inherently three-dimensional and compounds complex combinations of each individual square. The challenge of this study is to take the object, the Rubik's cube, and translate properties into a two-dimensional platform. By giving the surreal control feeling with interaction devices, the user is able to observe the object from every aspect continually without any obstacles. The two-dimensional platform also needs to help the user to apply his/her tactile sense; eventually build up the acknowledge of the relative connections between each squares. In doing so the object needs to take on the appearance of these properties within the interface; meaning each section must rotate on every axis independently.

To implement the thesis project, Macromedia Director 7.0 is chosen to be recreate the Rubik's cube on screen. The programming language LINGO is powerful enough to create three dimensions effect to accomplish the task.
2-1 GEOMETRY ORIENTATION

Arrange three axes to be right angles to each other and label the width and height axes \( x \) and \( y \), the third axis, for depth, is named \( z \) axis. The orientation of coordinate system is set to a right-handed system. So the object behind the display screen has a positive \( z \) value.

2-2 REPRESENTATION OF POINTS

A point is represented in three dimensions by its coordinates. These three values are specified as the elements of a 1-row, 3-column matrix:

\[
\begin{bmatrix} x, y, z \end{bmatrix}
\]

Furthermore, the homogeneous coordinate system is introduced in this project for simplifying the calculation purpose. In homogeneous coordinate system, an \( n \)-dimensional space is represented by \( n+1 \) dimensions\(^1\); three-dimensional data which give the position of a point is represented by four coordinates. The values are specified as the elements of a 1-row, 4-column matrix:

\[
\begin{bmatrix} x, y, z, 1 \end{bmatrix}
\]

2-3 TRANSFORMATION AND MATRICES

Consider the results of the multiplication of a matrix \( [ x, y, z, 1 ] \) containing the coordinates of a point and a general 4x4 matrix \( [ T ] \):

\[
\begin{bmatrix} x, y, z, 1 \end{bmatrix} [ T ] = \begin{bmatrix} x', y', z', 1 \end{bmatrix}
\]

The results is a 1x4 matrix which specifys the new coordinates of this point after transformation\(^2\). The matrix \( [ T ] \) is called transformation matrix. Based on experience, there are fixed formulas to generalize transformation matrices for different transformation\(^3\).
Rotation is the one kind of transformation performed in this study. The list below is the transformation matrices for rotation about a coordinate axis:

1. A rotation about z-axis

\[
[T] = \begin{bmatrix}
\cos\theta, & \sin\theta, & 0, & 0 \\
-s\sin\theta, & \cos\theta, & 0, & 0 \\
0, & 0, & 1, & 0 \\
0, & 0, & 0, & 1
\end{bmatrix}
\]

2. A rotation about x-axis

\[
[T] = \begin{bmatrix}
1, & 0, & 0, & 0 \\
0, & \cos\theta, & \sin\theta, & 0 \\
0, & -\sin\theta, & \cos\theta, & 0 \\
0, & 0, & 0, & 1
\end{bmatrix}
\]

3. A rotation about y-axis

\[
[T] = \begin{bmatrix}
\cos\theta, & 0, & -\sin\theta, & 0 \\
0, & 1, & 0, & 0 \\
\sin\theta, & 0, & \cos\theta, & 0 \\
0, & 0, & 0, & 1
\end{bmatrix}
\]

The problem was to derive a transformation matrix for a rotation of an angle \( \theta \) about an arbitrary line. The solution was to build this transformation matrix out of those which already exist.

First rotate about x-axis until the axis of rotation is in the xz plane, second rotate about y-axis until the z-axis corresponds to the axis of rotation, then rotate about z-axis (the axis of rotation) instead of fixing the object and rotating the axes. Finally reverse the rotation about y-axis and reverse the rotation about x-axis. Assuming the angle of rotation about x-axis is \( \theta_1 \) and the angle of rotation about x-axis is \( \theta_2 \):

4. A rotation about an arbitrary axis

\[
[R_x] = \begin{bmatrix}
1, & 0, & 0, & 0 \\
0, & \cos\theta_1, & \sin\theta_1, & 0 \\
0, & -\sin\theta_1, & \cos\theta_1, & 0 \\
0, & 0, & 0, & 1
\end{bmatrix}
\]

\[
[R_y] = \begin{bmatrix}
\cos\theta_2, & 0, & -\sin\theta_2, & 0 \\
0, & 1, & 0, & 0 \\
\sin\theta_2, & 0, & \cos\theta_2, & 0 \\
0, & 0, & 0, & 1
\end{bmatrix}
\]

\[
[R_z] = \begin{bmatrix}
\cos\theta, & \sin\theta, & 0, & 0 \\
-s\sin\theta, & \cos\theta, & 0, & 0 \\
0, & 0, & 1, & 0 \\
0, & 0, & 0, & 1
\end{bmatrix}
\]

04
\[
[ R_y ]^{-1} = \begin{bmatrix}
\cos \theta_2, & 0, & \sin \theta_2, & 0 \\
0, & 1, & 0, & 0 \\
-\sin \theta_2, & 0, & \cos \theta_2, & 0 \\
0, & 0, & 0, & 1
\end{bmatrix}
\]

\[
[ R_x ]^{-1} = \begin{bmatrix}
1, & 0, & 0, & 0 \\
0, & \cos \theta_1, & -\sin \theta_1, & 0 \\
0, & \sin \theta_1, & \cos \theta_1, & 0 \\
0, & 0, & 0, & 1
\end{bmatrix}
\]

\[
[ T ] = [ R_x ][ R_y ][ R_z ][ R_y ]^{-1}[ R_x ]^{-1}
\]

Translation is another kind of transformation performed in this study. Unlike rotation, only one matrix is need to represent the transformation matrix. Assuming \( dx, dy, dz \) are the translation factors in \( x, y \) and \( z \) direction:

5. Translate along coordinate axes

\[
[ T ] = \begin{bmatrix}
1, & 0, & 0, & 0 \\
0, & 1, & 0, & 0 \\
0, & 0, & 1, & 0 \\
dx, & dy, & dz, & 1
\end{bmatrix}
\]

2-4 PROJECTION

Since the viewing surface is only two-dimensional, the three-dimensional object must be projected onto the two-dimensional screen. The simplest way of doing this is parallel projection. To perform parallel projection means just to discard the \( z \) coordinate. So, the screen, or viewing surface, is parallel to the \( xy \) plane, and the lines of projection are parallel to the \( z \) axis. And this method is used in this project.
3-1 DIRECTION AND VECTORS

Objects in three-dimensional space possess a special property - direction. In general, the direction indicates whether the objects' surface is facing toward the viewer or another. It means this property can determine the visibility of the particular surface.

In math, the direction can be represented by the surfaces' normal vector. In order to calculate the normal vector, here we define the vertices of a surface in a counterclockwise fashion and by using a 1-row, 3-column matrix to store these vertices.

```
matrix = [ vertex, vertex, vertex]
```

Now we can use this vertices matrix to represent the surface.

3-2 CONSTRUCTION OF CUBE

Each cube can be represented by six unit squares; each unit square can be represented by four vertices. Following the description, a unit cube can be illustrated by a set of twenty-four vertices.
1. Each unit square can be represented by four vertices. These four vertices are specified as the elements of a 1-row, 4-column matrix:

![Square](image)

Square = [ v1, v2, v3, v4 ]

2. Each unit cube can be represented by six squares. These six squares are specified as the elements of a 1-row, 6-column matrix:

![Squares of cube](image)

Squares of cube = [ s1, s2, s3, s4, s5, s6 ]

3. Combining the results from 1 and 2:

s1 = [ v11, v12, v13, v14]
s2 = [ v21, v22, v23, v24]
s3 = [ v31, v32, v33, v34]
s4 = [ v41, v42, v43, v44]
s5 = [ v51, v52, v53, v54]
s6 = [ v61, v62, v63, v64]

Squares of cube = [ [ v11, v12, v13, v14],
                   [ v21, v22, v23, v24],
                   [ v31, v32, v33, v34],
                   [ v41, v42, v43, v44],
                   [ v51, v52, v53, v54],
                   [ v61, v62, v63, v64] ]

In fact, if each vertex corresponds to a certain point in the space, only eight points are used to illustrate the unit cube because some vertices have the same coordinates, it means they share the same points.

4. The unit cube can be represented by eight points. These points are organized by a 1-row, 8-column matrix:

Points of cube = [ p1, p2, p3, p4, p5, p6, p7, p8]
Then substitute the vertices of the cube with these eight points.

Squares of cube = [ [p6, p5, p8, p7], [p7, p8, p4, p3],
                    [p5, p1, p4, p8], [p2, p6, p7, p3],
                    [p2, p1, p5, p6], [p1, p2, p3, p4] ]

3-3 CONSTRUCTION OF RUBIK'S CUBE

Twenty seven unit cubes build up a rubik's cube. In theory, it uses up to 576 vertices to illustrate a rubik's cube. Again, after eliminating the repetition, we need 64 points (64 coordinates) to create one object. But in this project, we don't actually use 64 coordinates to compose the rubik's cube. Instead, eight coordinates are used as the primitive positions to initiate a single unit then this unit is duplicated twenty six times to create the fundamental components for rubik's cube. By performing translation, these cubes are placed into the proper position to form the rubik's cube.

In this project, each square of the unit cube is assigned to a different colour. Using the method mentioned above, resulting a rubik's cube at its original state. All nine squares of one side of the rubik's cube are in the same colour.
Since each section can rotate on every axis independently, updating the changing relationships between each square plays the major role in this study. In this chapter, the concepts of tracing critical information and methods of using a matrix to organize the data will be explained.

4-1 UNIT SQUARE AND CUBE

From chapter two, the matrices storing the eight points (coordinates) and the six sides to form the unit cube are:

Points of cube = [ p1, p2, p3, p4, p5, p6, p7, p8 ]
Squares of cube = [ s1, s2, s3, s4, s5, s6 ]
Squares of cube = [ [ p6, p5, p8, p7 ], [ p7, p8, p4, p3 ], [ p5, p1, p4, p8 ], [ p2, p6, p7, p3 ], [ p2, p1, p5, p6 ], [ p1, p2, p3, p4 ] ]

The relationship between the square and points must be fixed. This means that s1 is always composed by p6, p5, p8, p7 in sequence, s2 is always composed by p7, p8, p4, p3 in sequence and so on. Transformations only effect the coordinates each point corresponds to. This is important for drawing the right colours for all sides of cube after transformation.

4-2 IDENTIFICATION

A rubik’s cube is composed of twenty seven subdivided unit cubes. In order to distinguish them, each
unit cube is assigned to a unique identifier. The following explains how to generate the three-letters identifier for each unit cubes.

1. Based on the three axes (x, y, z), which is right angle to each other, a rubik's cube can be divided into three panels in one direction along one axis and each panel consists of nine unit cubes. So there are a total of three different directions to do so. One is along the x-axis, the second is along the y-axis and the third is along the z-axis. Then all panels are given an identifier according to their coordinates. In one direction, the panel has the lowest position is labeled A. The second lowest is labeled B and the highest is labeled C.

2. Dividing the cube in the first and second direction (x and y), we can create nine columns. We use two letters to represent the columns' identifier. The first letter inherent the identifier from x direction and the second letter inherent the identifier from y direction. The picture below shows the division of cube and identifiers.

3. Add the third division and using the strategy discussed above. The three-letters identifier and the division are show below:
4-3 INDEX SYSTEM

Now we are going to look at a rubik's cube from another aspect. Here a rubik's cube is no longer an object consisted of twenty seven unit cubes but a cubical space divided into twenty seven sub-units and each sub-unit is placed with one unit cube which posses a unique identifier.

In order to identify each sub-unit of the cubical space, we label them systematically with unique identification call index. Index is a three digits number. The way to generate an index is similar to label the unit cubes. Base on the three axes (x, y, z) which are right angles to each other, the first digit represents the position along x-axis, the second digit represents the position along the y-axis and the third digit represents the position along z-axis. Along the same axis, number for highest position is 3 and the lowest is 1.

Then we use matrices to manage these two identifications.

1. Use a 1-row, 2-columns matrix. The second element is the index of a unit space. The first element is the identifier of the unit cube which occupies the unit space. Totally, there are twenty seven matrices created.

\[
[\text{identifier}, \text{index}] = [\ BBB, \ 222 ]
\]
2. Organize these matrices with a 1-row, 27-column matrix and we call it `IdIndexList`.

```
IdIndexList = [
    [ AAA, 111 ], [ BAA, 211 ], [ CAA, 311 ],
    [ AAB, 112 ], [ BAB, 212 ], [ CAB, 312 ],
    [ AAC, 113 ], [ BAC, 213 ], [ CAC, 313 ],
    [ ABA, 121 ], [ BBA, 221 ], [ CBA, 321 ],
    [ ABB, 122 ], [ BBB, 222 ], [ CBB, 322 ],
    [ ABC, 123 ], [ BBC, 223 ], [ CBC, 323 ],
    [ ACA, 131 ], [ BCA, 231 ], [ CCA, 331 ],
    [ ACB, 132 ], [ BCB, 232 ], [ CCB, 332 ],
    [ ACC, 133 ], [ BCC, 233 ], [ CCC, 333 ]
]  
```
The object orientation involves two different kinds of transformation. One is overall rotation. The other is partially rotation. By combing these two transformations, the viewers are able to examine the cube in every aspect and rearrange the combination of squares in any desired way.

5-1 OVERALL ROTATION

Overall rotation means rotate the rubik’s cube or twenty seven unit cubes at the same time. We place the cube on the origin of the coordinate system, the center of the cube is the origin, and using three basic rotation - rotating about x-axis, y-axis and z-axis. With serial rotations about different axis, all angles for the user to view the cube are covered.

![Overall rotation about x-axis by 0 degrees](image)

5-2 PARTIAL ROTATION

When rotating one panel of the rubik’s cube (which is consisted of nine connecting unit cubes), we call it partial rotation. Partial rotation is non-restricted rotation - rotating about any arbitrate axis. Two essential conditions must be concerned. This arbitrary axis must be right angle to the panel which is going to rotate and it must through the center of the cube.
5-3 RELATIONSHIP

The relationship of unit cubes indicate the positions of their offset. For any one cube, which cube share the front side with it, which cube is on top of it or which cube is underneath it is called relationship. Therefore, relationship of unit cubes changes after performing partial rotation but not overall rotation.

Adapting the previous concept that a rubik's cube is a cubical space divided into twenty seven sub-units and each sub-unit is placed with one unit cube. Overall rotation also means rotating this cubical space. No matter how many times the transformation (overall rotation or partial rotation) is performed, the relationship of indices is never changed.

Unit space whose index is 333 is always connected with another seven units whose Indices are 233, 232, 332, 322, 323 and 223.
Consequently, each time the overall rotation is performed, the cubical space rotates as well. So under this circumstance, the IdIndexList is unchanged. But after performing partial rotation, updating IndexList becomes necessary.

Notice: [BBC, 223] is still [BBC, 223]

Notice: [BBC, 223] become [BBC, 232]

5-4 PANEL SELECTION

Finding out which nine unit cubes compose the panel to perform partial rotating is another task. Each randomly chosen unit cube belongs to three different panels. The common factor amount these cubes exist in indices of the unit of space where these cubes are placed.

If any unit cube is selected, we automatically acquire the cube's identifier. Retrieving through the IdIndexList, the index of the space where this cube located is accessible too.

1. The cube is selected, its identifier is BCC and its index is 233:
2. Below are the three panels that this cube is included and their indices:

3. Here are the unit cubes of each panel and their corresponding indices.

4. As shown above. The third digit of the indices of the first panel are the same number 3, the first digit of the indices of the second panel are 2 and the second digit of the indices of the second panel are 3.

Conclusively, the method to group the cubes for each panel is to choose the cubes located in the unit spaces which posses the same number at the corresponding digit of their indices as the index desired.

5-5 RELATIVE POSITION LIST

In order to update the IdIndexList sufficiently, we create a special matrix to handle this task. It is called RelativePositionList. We try to transform the index system into a matrix showing the relationship between the cubes occupying the spaces and their indices. Therefore, the matrix can reveal the corresponding relationships between the fixed indices system and cubes' identifiers, respectfully without using index numbers.
1. Divide index system into three groups according to the first digit number of the indices. Then divide these groups individually according to the second digit number of the indices. The following are the nine subgroups after division.

<table>
<thead>
<tr>
<th>111</th>
<th>112</th>
<th>113</th>
<th>211</th>
<th>212</th>
<th>213</th>
<th>311</th>
<th>312</th>
<th>313</th>
</tr>
</thead>
<tbody>
<tr>
<td>121</td>
<td>122</td>
<td>123</td>
<td>221</td>
<td>222</td>
<td>223</td>
<td>321</td>
<td>322</td>
<td>323</td>
</tr>
<tr>
<td>131</td>
<td>132</td>
<td>133</td>
<td>231</td>
<td>232</td>
<td>233</td>
<td>331</td>
<td>332</td>
<td>333</td>
</tr>
</tbody>
</table>

2. Using 1-row and 3-columns matrix to represent these nine subgroups and these three big groups.

\[
\begin{bmatrix}
\end{bmatrix}
\]

3. Again, put these three groups (matrices) in a 3-rows, 1-columns matrix:

\[
\begin{bmatrix}
\end{bmatrix}
\]

Notice that the first digit of index represent the column number in the outer matrix. The second digit of index represent the column number of the inner matrix which is one element of the outer matrix. Finally the third digit represent the column number of the core matrix.

4. Substitute the indices with their corresponding cube identifiers. The result is a matrix called relativePositionList.

\[
\text{relativePositionList} = \\
\begin{bmatrix}
[ [[AAA, AAB, AAC], [ABA, ABB, ABC], [ACA, ACB, ACC]] ] \\
[ [[BAA, BAB, BAC], [BBA, BBB, BBC], [BCA, BCB, BCC]] ] \\
[ [[CAA, CAB, CAC], [CBA, CBB, CBC], [CCA, CCB, CCC]] ]
\end{bmatrix}
\]

Example: Find the cube corresponding to the index 233.

a) From first digit, the matrix chosen is:

\[
[ [BAA BAB BAC], [BBA BBB BBC], [BCA BCB BCC]]
\]

b) From second digit, the matrix chosen is:

[BCA BCB BCC]

c) From third digit, the cube locate in this unit is:

BCC
5-6 MATRIX ROTATION

For updating IdIndexList and relativePositionList, the cubes that involve partial rotation are organized by a 3-row, 3-column matrix. By using the method mentioned in 4-4, first choose the panel whose first digit of index is 2. The result is the element of the second column of the outer matrix.

\[
\begin{bmatrix}
[BAA \ BAB \ BAC], [BBA \ BBB \ BBC], [BCA \ BCB \ BCC]
\end{bmatrix}
\]

Using a 3-row, 3-column matrix to represent these nine cubes. Sequentially store their identifiers into a matrix. The result and their correspond indices are:

\[
\begin{bmatrix}
BAA & BAB & BAC \\
BBA & BBB & BBC \\
BCA & BCB & BCC
\end{bmatrix}
\begin{bmatrix}
211 & 212 & 213 \\
221 & 222 & 223 \\
231 & 232 & 233
\end{bmatrix}
\]

Once the partial rotation is done. These nine cubes rotate 90 degree counterclockwise. The new matrix becomes:

\[
\begin{bmatrix}
BAA & BAB & BAC \\
BBA & BBB & BBC \\
BCA & BCB & BCC
\end{bmatrix}
\rightarrow
\begin{bmatrix}
BAC & BBC & BCC \\
BAB & BBB & BCB \\
BAA & BBA & BCA
\end{bmatrix}
\begin{bmatrix}
211 & 212 & 213 \\
221 & 222 & 223 \\
231 & 232 & 233
\end{bmatrix}
\]

Comparing the old and new matrix, element at row 1, column 1 in old matrix is now at row 3 column 1 in new matrix. It means that the cube occupies the space whose index is 211 is now switched from BAA to BAC. According to this change, renew the IdIndexList and relativePositionList.
Lingo is the primary programming language used in this study and in this chapter we are going to apply the concepts through the coding process gradually. The theoretical idea mentioned before may be implemented differently according to computing format or to achieve better performance in different computers.

6-1 GLOBAL VARIABLES

1. global stageCenter
   The position for displaying the object in the window (stage). It is not necessary to be on the center of the stage. It is represented as:

   point(x,y)

2. global modelMatrix
   If the object is to perform different transformation at one time, we simply multiply all transformation matrices in sequence to create the final transformation matrix, called modelMatrix. After multiplying this matrix with the old coordinates, the result is new coordinates four points.

3. global IDIndexList
   A property list to store the object's identifier and its corresponding index.

   IDIndexList = [ ID1: Index1, ID2: Index2, ... ]

4. global IDInstList
   A property list to store the object's identifier and its corresponding instance.

   IDInstList = [ ID1: Instance1, ID2: Instance2, ... ]
5. global relativePosiList
A list contains three elements which are lists containing three elements whose elements are also three lists containing three elements. This variable is to represent the indices and their corresponding cube identifiers.

\[
\text{relativePosiList} = \\
\left[ \\
\left[ \left[ \text{ID}, \text{ID}, \text{ID} \right], \left[ \text{ID}, \text{ID}, \text{ID} \right], \left[ \text{ID}, \text{ID}, \text{ID} \right] \right], \\
\left[ \left[ \text{ID}, \text{ID}, \text{ID} \right], \left[ \text{ID}, \text{ID}, \text{ID} \right], \left[ \text{ID}, \text{ID}, \text{ID} \right] \right], \\
\left[ \left[ \text{ID}, \text{ID}, \text{ID} \right], \left[ \text{ID}, \text{ID}, \text{ID} \right], \left[ \text{ID}, \text{ID}, \text{ID} \right] \right] \\
\right]
\]

6. global xIDList
A list of identifiers of cubes whose corresponding indices have the same number at the first digit. These cubes are to perform partial rotation together. The list is a 1-row, 3-column matrix and it elements are also 1-row, 3-column matrices.

\[
\text{xIDList} = \left[ \left[ \text{BCC}, \text{BCB}, \text{BCA} \right], \left[ \text{BBC}, \text{BBB}, \text{BBA} \right], \left[ \text{BAC}, \text{BAB}, \text{BAA} \right] \right]
\]

\[
\text{corresponding indices} = \\
\left[ \left[ \text{233}, \text{232}, \text{231} \right], \left[ \text{223}, \text{222}, \text{221} \right], \left[ \text{213}, \text{212}, \text{211} \right] \right]
\]

7. global yIDList
A list of identifiers of cubes whose corresponding indices have the same number at the second digit.

\[
\text{yIDList} = \left[ \left[ \text{ACC}, \text{ACB}, \text{ACA} \right], \left[ \text{BCC}, \text{BCB}, \text{BCA} \right], \left[ \text{CCC}, \text{CCB}, \text{CCA} \right] \right]
\]

\[
\text{corresponding indices} = \\
\left[ \left[ \text{133}, \text{132}, \text{131} \right], \left[ \text{233}, \text{232}, \text{231} \right], \left[ \text{333}, \text{332}, \text{331} \right] \right]
\]

8. global zIDList
A list of identifiers of cubes whose corresponding indices have the same number at the third digit.

\[
\text{zIDList} = \left[ \left[ \text{ACC}, \text{BCC}, \text{CCC} \right], \left[ \text{ABC}, \text{BC}, \text{CBC} \right], \left[ \text{AAC}, \text{BAC}, \text{CAC} \right] \right]
\]

\[
\text{corresponding indices} = \\
\left[ \left[ \text{133}, \text{233}, \text{333} \right], \left[ \text{123}, \text{223}, \text{323} \right], \left[ \text{113}, \text{213}, \text{313} \right] \right]
\]

The local coordinates can be defined by a couple of rules. An axis along the normal vector of a partial rotating panel is called local axis. If the cubes that compose this panel have the same first digit number of their corresponding indices, the axis is called local x-axis. Similarly, if the second digit number is the same, the axis is called local y-axis and if the third digit number is the same, the axis is called local z-axis.
To define the positive direction of these local axes depending on the digit number. For local x-axis, there are three panels rotating about this line. Along this axis, the direction form the panel which has the smallest digit number (1) to the panel which has the highest digit number (3) is pointing toward the positive direction of x-axis. Same rules can be applied to other two axes.

When preparing for partial rotation, a point \( p \) fall on the rotation axis is required for calculation. The basic setting is that no matter which arbitrary axis to rotate about, always rotate 90 degree counterclockwise. Combining the methods to create this point and setting the transformation matrix, a problem is revoked which his some panels rotate clockwise. Consequently, it affected the method to update the IDIndexList and relativePosList.

Notice that eventually all panels will rotate about the coordinate z-axis by 90 degree counterclockwise. For panel facing toward -x, it actually perform clockwise rotation according to local coordinate system.

In order to correct this exception, determining the rotating direction of each panel becomes crucial. As you know, the index system is fixed and synchronizing to the local coordinate system. Two different matrix updating methods can be chosen under different circumstances. The following three variables are used to represent this important information - clockwise or counterclockwise

9. global localxRotaDir
   Locally x-axis rotating direction. According to right-hand rules, this global value is 0 when the panel rotates clockwise and the value is 1 when the panel rotates counterclockwise.

10. global locallyRotaDir
    Locally y-axis rotating direction. According to right-hand rules, this global value is 0 when the panel rotates clockwise and the value is 1 when the panel rotates counterclockwise.
11. global localzRotaDir
Locally z-axis rotating direction. According to right-hand rules, this global value is 0 when the panel rotates clockwise and the value is 1 when the panel rotates counterclockwise.

6-2 THREE DIMENSION ENGINES

The following handlers transform the coordinate in three-dimensional space and plot the points onto a two-dimensional plan.

1. on multiMatrix V, M
This handler manipulates a new matrix by multiplying two matrices, V and M. The requirement is that the number of columns of matrix V must be equal to the number of rows of matrix M.

2. on loadIdentity
The return value is a template matrix (identity Matrix) for creating a transformation matrix. It is a list contain four sub-lists which have four elements. It is important to call this function and set the global variable modelMatrix to identify Matrix every time before starting a new transformation.

   identity Matrix = [ [ 1.0, 0.0, 0.0, 0.0 ], [ 0.0, 1.0, 0.0, 0.0 ], [ 0.0, 0.0, 1.0, 0.0 ], [ 0.0, 0.0, 0.0, 1.0 ] ]

3. on setTransMatrix Tx, Ty, Tz
The translation matrix is:

   [ T ] = [ [ 1, 0, 0, 0], [ 0, 1, 0, 0], [ 0, 0, 1, 0], [ dx, dy, dz, 1]. ]

This handler multiples modelMatrix with a translation matrix to manipulate the new modelMatrix after translation. Doing so, it takes sixty four multiples and forty eight adds to implement the task.

   \[
   \begin{bmatrix}
   1.0 & 0.0 & 0.0 & 0.0 \\
   0.0 & 1.0 & 0.0 & 0.0 \\
   0.0 & 0.0 & 1.0 & 0.0 \\
   0.0 & 0.0 & 0.0 & 1.0 \\
   \end{bmatrix}
   \begin{bmatrix}
   1 & 0 & 0 & 0 \\
   0 & 1 & 0 & 0 \\
   0 & 0 & 1 & 0 \\
   dx & dy & dz & 0 \\
   \end{bmatrix}
   \]

Notice that some multiplying or adding with 0 can be eliminated. For efficiency consideration, we actually use a fixed structure to manipulate the multiplication. The three arguments for the handler is the translating amount along three coordinate axes.

4. on setXrotationMatrix s, c
Manipulating the modelMatrix after rotating counterclockwise about coordinate x-axis by \( \phi \) degrees. The arguments s is the sine value of the rotation angle \( \phi \) and c is the cosine value of the rotation angle \( \phi \).
5. on setYrotationMatrix s, c
Manipulating the modelMatrix after rotating counterclockwise about coordinate y-axis by $\theta$ degrees. The arguments $s$ is the sine value of the rotation angle $\theta$ and $c$ is the cosine value of the rotation angle $\theta$.

6. on setZrotationMatrix s, c
Manipulating the modelMatrix after rotating counterclockwise about coordinate z-axis by $\theta$ degrees. The arguments $s$ is the sine value of the rotation angle $\theta$ and $c$ is the cosine value of the rotation angle $\theta$.

7. on setArbitraryRotateMatrix p, sinV, cosV
Manipulating the modelMatrix after counterclockwise rotating about an arbitrary axis by $\theta$ degrees. The transformation is built out of those rotations about the coordinate axes. First we shall perform a translation to move the center of the object onto the line (axis). Then make rotations about x and y axes to align the z-axis with the line. The rotation about the line then becomes a rotation about the z-axis. Finally we apply the inverse transformations for the rotations about the x and y axes and for the translation to restore the object to their original orientation.

This handler is called when performing the partial rotation. According to 4-2, we build the rubik's cube and place its center on the orig. The arbitrary axis we choose is always a line through the center of the cube. Then the translation can be skipped under that situation. However the rest of the steps are still needed.

a) Argument $p$ is a point falling on this arbitrary axis. The axis is a line through this point $p$ and the orig.

b) Arguments $\sin V$ and $\cos V$ are the sine value and cosine value of the rotating angle $\theta$.

8. on getNormal vector p1, p2, p3
This handler takes three points and manipulates the normal Vector of the surface which these three points fall on. The return value is a point represented with a homogeneous matrix.

\[
\text{normal Vector} = [ x, y, z, 1 ]
\]
9. on rotateMatrix M
   This handler takes a 3-row, 3-column matrix and manipulates a new matrix after rotating this matrix counterclockwise 90 degrees (see 4-5). This function is called for assassinating updating relativePosList and IDIndexList after performing a partial rotation. The elements of the argument matrix are the identifiers of the cubes involved in the rotation.

   \[ M = \begin{bmatrix}
   \text{BAA, BAB, BAC}, & \text{BBA, BBB, BBC}, & \text{BCA, BCB, BCC}
   \end{bmatrix} \]

   return Matrix = \[ \begin{bmatrix}
   \text{BAC, BBC, BCC}, & \text{BAB, BBB, BCB}, & \text{BAA, BBA,BCA}
   \end{bmatrix} \]

10. on backRotateMatrix M
    This handler takes a 3-row, 3-column matrix and manipulate a new matrix after rotating this matrix clockwise 90 degrees. This function is called for assassinating updating relativePosList and IDIndexList after performing a partial rotation. The elements of the argument matrix are the identifiers of the cubes involved in the rotation.

    \[ M = \begin{bmatrix}
   \text{BAA, BAB, BAC}, & \text{BBA, BBB, BBC}, & \text{BCA, BCB, BCC}
   \end{bmatrix} \]

    return Matrix = \[ \begin{bmatrix}
   \text{BCA, BBA, BAA}, & \text{BCB, BBB, BAB}, & \text{BCC, BBC, BAC}
   \end{bmatrix} \]

11. updateIDIndexList oldlist, newList
    This handler is to update the global variable IDIndexList. The argument oldlist is the same argument used in handlers rotateMatrix() and backRotateMatrix(). The newList is the return value from these two handlers. Cubes in the newList shall correspond to the index of the cube located at the same position in the oldlist.

    | before     | after     |
    |------------|-----------|
    | BAA:211    | BAC:211   |
    | BAB:212    | BBC:212   |
    | BAC:213    | BCC:213   |
    | BBA:221    | BAB:221   |
    | BBB:222    | BBB:222   |
    | BBC:223    | BCB:223   |
    | BCA:231    | BAA:231   |
    | BCB:232    | BBA:232   |
    | BCC:233    | BCA:233   |

    To update IDIndexList, first duplicate the original IDIndexList as the new IDIndexList, take one cube's identifier (BAA) in oldlist, retrieve its index according to the original IDIndexList (211). Find the corresponding cube identifier in the newList (BAC). Assign the index (211) as the value of the property (BAC) in new IDIndexList.

12. setRelativePosList
    Call this function to initialize or renew the relativePosList when needed. Simply go through IDIndexList, get the identifier and its index. Depending to the index find the correct position in relativePosList, replace the element in the list with this new identifier.
13. `getPointOnAxis instList`
   In order to perform partial rotation, a point $p$ falls on the rotating axis (local axes) except the origin is needed. The center point of the panel to rotate is one of the kind points. First we need the nine cubes involving the rotation. Choose two cubes located on opposite position. Average the coordinates of the centers of these two cubes, the result is treated as the point $p$. If this point happens to be the orig, then use the normal Vector of this panel to represent the point $p$. The argument `instList` is a matrix contain the identifiers of the cubes participate the rotation. Later use `IDInstList` to retrieve the objects' instances.
Borrowing the terms from C/C++, a parent script is a class. Property is data member and handlers in this cast member are member functions. Therefore we can treat a parent script as a data type. In this study, we create three essential data types (three parent scripts) to compose a rubik's cube. They are Panelscript, cubeScript and EightCubesScript. The tactic is simple, in EightCubesScript, there will be a list to manage child cubeScript objects. In cubeScript, there is a list to store child Panelscript objects.

7-1 PANELSCRIPT PROPERTIES

The most basic object is PanelScript object and a unique sprite and a cast member are assigned to it. All three-dimensional illusions are manipulated by the sprite property quad and locZ.

1. property pBase
   The instance of the object which use this object to form a unit cube.
   <offspring "Panelscript" 3 7c65438>

2. property pID
   The identification of this object
   "AAC1"

3. property pSprite
   Number of the sprite assigned to this object

4. property pMember
   The cast member assigned to this object
   "front"

5. property pLocz
   The location Z property of the sprite assigned to the object
The sprite property quad is represented by a list containing four two-dimensional points \((x, y)\) indicating four location in the window. The coordinates of these four points relate to the four vertices to form the square in a three-dimensional space.

\[
sprite(pSprite).quad = \\
[\text{point}(x_1, y_1), \ \text{point}(x_2, y_2), \ \text{point}(x_3, y_3), \ \text{point}(x_4, y_4)]
\]

6. property \(p\text{VertexList}\)
   A list contains the four original coordinates to form this object

7. property \(p\text{CurrentVist}\)
   A list contains the four coordinates to form this object

\[
p\text{CurrentVist} = p\text{VertexList} = \\
[ [x_1, y_1, z_1, 1], \ [x_2, y_2, z_2, 1], \ [x_3, y_3, z_3, 1], \ [x_4, y_4, z_4, 1] ]
\]

Plot these four vertices in \(p\text{CurrentVList}\) onto screen and store the location in a list for assigning to quad.

8. property \(p\text{PosList}\)
   A list contains four positions on screen

\[
p\text{PosList} = [\ \text{point}(x_1, y_1), \ \text{point}(x_2, y_2), \ \text{point}(x_3, y_3), \ \text{point}(x_4, y_4) ]
\]

9. property \(p\text{PolyList}\)
   The sequence to transform the vertices in \(p\text{CurrentVist}\) onto screen and to store the result in \(p\text{PosList}\).

\[
p\text{PolyList} = [1, 2, 3, 4]
\]

10. property \(p\text{Center}\)
    The center of this object. Average either two vertices which locate on diagonal corners.

\[
( [x_1, y_1, z_1, 1] + [x_2, y_2, z_2, 1] ) / 2 = [X_c, Y_c, Z_c, 1]
\]
7. property pOffset
A location on screen where the origin of the three-dimensional coordinate is located.

7-2 PANELSCRIPT HANDLERS

1. on new me
Initializing constructor. Initiating all properties and assigning values to them by calling another handler to help initialize this object. After initialization, call another handler to draw this object.

-- Function call
   a) add sprite(pSprite).spritInstanceList, me
      In order to receive mouse events, add the instance into the spritInstanceList 11.

   b) add the actorList, me
      In order to receive stepFrame message, add this instance into the actorList 12.

   c) initDraw()
      Plot this object onto the screen.

2. on getLocZ me
To manipulate the value for sprite locZ property we simply go through the four points and take the lowest z coordinate value for locZ. Be aware that the lowest cast shall have the highest sprite number in Director. Using a negative value of this coordinate and adding a constant to this value makes sure that the value is always positive.

What if two different sprites have the same locZ, which sprite's cast member shall be drawn on top? Due to the natural property of a cube. At one time the viewer is able to see at least one side of the cube and at most three sides of the cube. These sides never overlap each other. So, it is possible for two squares to happen to have the same locZ value, but because they are not overlapping who will be drawn on top is not an issue.
For the same reason that at the most only three sides can be seen. It is a chance to improve the performance of the computer by not drawing all the hidden sides.

3. on isVisible me
   This function uses the z coordinate value of the normal vector to determine whether this square is visible. If it is not then it will not be drawn on screen. In three-dimensional space, the normal vector is shown as below:

Notice that when the z value is smaller than zero, the square is visible. When the z value is equal to zero, we can only see a line, so in this situation the square is invisible.

4. on setIDList me
   Two tasks are implemented in this handler. The first is setting up the global variables xIDList, yIDList and zIDList. The second is setting the global variables, localxRotaDir, localyRotaDir and localzRotaDir.

If the user selects the cube composed by this square to be one of the nine cubes to rotate (the user uses the mouse to click on this square) we can retrieve the corresponding index of the cube, then update these three matrices - xIDList, yIDList and zIDList.

Because of the methods to pick the point and set the transformation matrix for partial rotation, we have to conclude that if the digit of the panel is smaller than 2, it will always perform a clockwise rotation. Then apply this rule for assigning the value for localxRotaDir, localyRotaDir and localzRotaDir.
7-3 CUBESCRIPT PROPERTIES

The properties in the cubescript are basically the same as the properties in the panelScript. However, some of them are defined slightly different.

1. property pSprite
   The pSprite still refers to the sprite number but it is the first sprite for the panel objects which composing this cube object.

2. property pMember
   The pMember here is not a single cast member but a list of cast members which will be assigned to the panels.
   
   \[
   \text{pMember} = \text{[ "front", "top", "right", "left", "bottom", "back" ]}
   \]

3. property pLocz
   The pLocz represent the sprite locz property. It is used for all panels of this cube.

Because a cube has eight corners and six sides, there are some properties that are affected. Properties pVertexList, property pCurrentVist, property pPosList and pPolyList change the elements in their list, but the definition remaining the same.

\[
\begin{align*}
\text{pCurrentVist} &= \text{pVertexList} = \\
&= \text{[ [x1, y1, z1, 1], [x2, y2, z2, 1], [x3, y3, z3, 1], [x4, y4, z4, 1],}
&\quad [x5, y5, z5, 1], [x6, y6, z6, 1], [x7, y7, z7, 1], [x8, y8, z8, 1] ]
\end{align*}
\]

\[
\begin{align*}
\text{pPosList} &= \text{[ point(x1,y1), point(x2,Y2), point(x3, y3), point(x4, y4),}
&\quad point(x5,y5), point(x6,Y6), point(x7, y7), point(x8, y8) ]}
\end{align*}
\]

\[
\begin{align*}
\text{pPolyList} &= \text{[ [6,5,8,7 ], [ 7,8,4,3 ], [ 5,1,4,8 ], [ 2,6,7,3 ], [ 2,1,5,6 ], [ 1,2,3,4 ] ]}
\end{align*}
\]

4. property pChildList
   A property list which stores the instances and the identification of the panelScript objects which composing this cube.

\[
\begin{align*}
\text{pChildList} &= \text{[ "AAC1" :<offspring "Panelscript" 3 7c613cc>,}
&\quad "AAC2" :<offspring "Panelscript" 3 7c613b8>,
&\quad "AAC3" :<offspring "Panelscript" 3 7c613a4>,
&\quad "AAC4" :<offspring "Panelscript" 3 7c61390>,
&\quad "AAC5" :<offspring "Panelscript" 3 7c6137c>,
&\quad "AAC6" :<offspring "Panelscript" 3 7c61368> ]}
\end{align*}
\]

7-4 CUBESCRIPT HANDLERS

1. on new me
   The initializing constructor. Initiating all properties and assigning values to them or calling another handler to help initializing the object.
-- Function call
  on initChildObj()
-- To initiate the panelScript objects of this cube and add them to pchildList.

2. on initChildObject me
   Initiate the child objects, panelScript objects, and add them to pChildList. There are six elements in the pChildList.

3. on getLocz me
   To manipulate the value for the sprite locZ property. The method is to go through all six points in pCurrentVist and take the lowest z coordinate value for locz. Be aware that the lowest cast shall have the highest sprite number in Director. The result needs to be adjusted. Take the negative value of the coordinate and add a constant to make the locz value positive.

4. on Abstransform me
   Performing transformation by multiplying the transformation matrix with pVertexList, which is never changed after the cube is initiated. So, the change of this transformation is not incremental if using this handler.

7-5 EIGHTCUBESRIPT PROPERTIES

1. property pSprite
   The very first sprite used for this EightCubeScript object.

2. property pMembre
   A list of cast members.
   Later this list is assigned to the cubeScript objects and all the unit cubes of this rubik's cube object are identical. Therefore, only six different cast members are used.

3. property pVertexList
   A list contains eight coordinates to form this object (see chapter two).
   The cubes used to construct the rubik's cube share this set of coordinates.

4. property pChildList
   A property list contain twenty seven cubes' identification and instances of this object.

   pChildList = [ "AAA" : <offspring "cubeScript" 3 7c61368>,
                 "AAB" : <offspring "cubeScript" 3 7c613b8>,
                 "AAC" : <offspring "cubeScript" 3 7c613a4>,
                 *     *     *     *     * ,
                 *     *     *     *     * ,
                 *     *     *     *     * ,
               "CCC" : <offspring "cubeScript" 3 7be9530> ]
5. property pChildIDList
   A list contains twenty seven identifiers to assign to the unit cubes.

   \[ p\text{ChildIDList} = [ "\text{AAA","BAA","CAAD","AAB","BAB","CAB",}
   \quad "\text{AAC","BAC","CAC","ABA","BBA","CBA",}
   \quad "\text{ABB","BBB","CBB","ABC","BBC","CBC",}
   \quad "\text{ACA","BCA","CCA","ACB","BCB","CCB",}
   \quad "\text{ACC","BCC","CCC"} ] \]

After all cubes are initiated, they need to be re-allocated in order to sculpt the rubik's cube. Performing translation on each cube can implement the task. The move amounts along three coordinate axes, for one cube is represented by a three elements list.

The translation amount is the length of one side of the square constructing the unit cube. If the cube move toward the negative direction then the value is negative.

6. property pChildshiftList
   A list contains twenty seven lists which represent the translation amounts for the unit cubes.

   \[ p\text{ChildshiftList} = [ [ dx1, dy1, dz1 ], [ dx2, dy2, dz2 ], ..., [ dx27, dy27, dz27 ] ] \]

7-6 EIGHTCUBESCRIP'T HANDLERS

1. on initChildObject me
   Initiate the child objects, cubelScript objects, and add them to pChildList. There are twenty seven elements in the pChildList.

2. on initDraw me
   Except calling the member functions of its child object to plot the cube on screen, this function also calls child objects' member function to perform translation and re-allocate their positions.
01. David F. Rogers, J. Alen Adams, Mathematical Elements for Computer Graphic, p.6
02. David F. Rogers, J. Alen Adams, Mathematical Elements for Computer Graphic, p.62
03. David F. Rogers, J. Alen Adams, Mathematical Elements for Computer Graphic, p.102, 107, 113
04. Steven Harrington, Computer Graphic: a programming approach, pp.256 - 261
05. Steven Harrington, Computer Graphic: a programming approach, pp. 261 - 262
06. Mark Walmsley, Graphics programming in C++, pp.176 - 180
07. David F. Rogers, J. Alen Adams, Mathematical Elements for Computer Graphic, p.504
08. Steven Harrington, Computer Graphic: a programming approach, p.255
09. Gary Rosenzweig, Special Edition Using Macromedia® Director® 7, pp.372 - 305
10. Bruce A. Epstein, Director in a nutshell, p.58, 70, 143, 148
11. Bruce A. Epstein, Director in a nutshell, p.347
12. Bruce A. Epstein, Director in a nutshell, p.109


Epstein, Bruce A. Director in a nutshell. California: O'Relly and Associates, Inc. 1999


Tamahori, Che. How To Cook 3D in Director, http://www.sfx.co.nz/tamahori/thought/shock_3d_howto.html

APPENDIX

A

PARENTSCRIPT - EightCubeScript

Property pBase
-- instance of base object which includes this object as a group
Property pID
-- identification of this object
property pSprite
-- sprite number //if there is derived class object, base sprite number
property pMember
-- sprite cast member //if there is derived class object, a list
property pLocz
-- sprite locz
property pVertexList
-- coordinate in Modeling view // [ x,y,z,1 ]/list[4x8]
property pCurrentVist
-- coordinate for current modeling view // [ x,y,z,1 ]/list[4x8]
property pOffset
-- org position of projection view on screen //point(x,y)
property pPosList
-- coordinates on screen // [ point(x,y) ]/list[1x4]
property pPolyList
-- the sequence for each vertex to form the panel object// [ 1,2,3,4 ]/list[4x6]
-- this property is shared by all child objects
property pCenter
-- the center coordinate of this object
property pChildList
-- derived class objects included in this object // [ ID:instance ]/list[6]
property pChildIDlist
-- child objects' ID //list[8]
property pChildshiftList
-- child objects initial translation amount list // [ Tx, Ty, Tz ]/list[3x8]
property pChildIndexList
-- child objects initial index in IDIndexList

global modelMatrix

global IDIndexList

-- CONSTRUCTOR: Create instance & initializing
on new me, base, ID, spriteNumber, castMember, vertexList, origOffset, polyList, childIDlist, childshiftList, childIndexList

  -- initializing
  pBase = base
  pID = ID
  pSprite = spriteNumber
  pMember = castMember
  pLocz = sprite(spriteNumber).locz -- default sprite locZ
  pVertexList = duplicate(vertexList)
  pCurrentVist = []
  pOffset = origOffset
  pPolyList = polyList
  pCenter = []

  -- update only when function called from outside
pChildList = [:]
pChildIDlist = childIDlist
pChildshiftList = childshiftList
pChildIndexList = childIndexList

-- initiate the instance of this object's child object
initChildObject(me)

-- in order to receive stepFrame message, add this object to the actorList
add the actorList, me

-- plot object on screen
initDraw(me)
return me
end new

-- MEMBER FUNCTION: initiate the instance of this object's child object
-- PROPERTY: pChildIDlist, pSprite, pMember, pVertexList, pOffset, pPolyList, pChildList,
pChildIndexList
-- GLOBAL: IDIndexList
on initChildObject me
    repeat with c = 1 to count(pChildIDlist)
        cubeID = pChildIDlist[c]
        cubeSprite = pSprite + (c - 1)*6
        cubeMember = pMember
        cubePolyList = pPolyList
        cubeVertex = pVertexList
        -- base, ID, spriteNumber, castMember, vertexList, origOffset, polyList
        cube = new(script "cubeScript", me, cubeID, cubeSprite, cubeMember, cubeVertex, pOffset, cubePolyList)
        -- add child ID:instance in pChildList
        addProp pChildList, cubeID, cube
        -- add global IDIndexList
        cubeIndex = pChildIndexList[c]
        addProp IDIndexList, cubeID, cubeIndex
    end repeat
end

-- MEMBER FUNCTION: plot object on screen
-- PROPERTY: pChildIDlist, pChildshiftList, pChildList
-- GLOBAL: modelMatrix
-- FUNCTION CALL: loadIdentity(), setTranMatrix(Tx, Ty, Tz)
-- baseTransform(), baseZorthProject() //From child objects' child objects
on initDraw me
    repeat with c = 1 to count(pChildIDlist)
        -- get translation amount
        xShift = pChildshiftList[c][1]
        yShift = pChildshiftList[c][2]
zShift = pChildshiftList[c][3]

-- build transformation matrix
modelMatrix = loadIdentity()
setTranMatrix (xShift, yShift, zShift)

-- update child object
pChildList[c].transform()
pChildList[c].zOrthProject()

-- update child object's child objects
repeat with child in pChildList[c].pChildList
  child.baseTransform()
  child.baseZorthProject()
  child.pLocz = child.getLocz()
end repeat
pChildList[c].childDraw()
pChildList[c].pVertexList = duplicate(pChildList[c].pCurrentVist)
end repeat

end initDraw

-- MEMBER FUNCTION: step through child objects and set locz
-- PROPERTY: pChildList
-- FUNCTION CALL: getLocz(me) // From child objects
on childGetLocz me
  repeat with child in pChildList
    child.pLocz = child.getLocz()
  end repeat
end childSetLocz

-- MEMBER FUNCTION: Step through child objects and Plot this object on screen
-- PROPERTY: pChildList
-- FUNCTION CALL: childDraw(me) // From child objects
on childDraw me
  repeat with child in pChildList
    child.childDraw()
  end repeat
end childDraw

-- MEMBER FUNCTION: Step through child objects and transform the pCurrentVist coordinate
-- PROPERTY: pChildList
-- FUNCTION CALL: transform(me) // From child objects
on childTransform me
  repeat with child in pChildList
    child.transform()
  end repeat
end childTransform

-- MEMBER FUNCTION: Step through child objects and transform the pVertexList coordinate
-- PROPERTY: pChildList
-- FUNCTION CALL: AbsTransform(me) // From child objects
on childAbsTransform me
    repeat with child in pChildList
        child.AbsTransform()
    end repeat
end childAbsTransform

-- MEMBER FUNCTION: Step through child objects and parallel projection on to 2D screen
-- PROPERTY: pChildList
-- FUNCTION CALL: zOrthProject(me) // From child objects
on childZorthProject me
    repeat with child in pChildList
        child.zOrthProject()
    end repeat
end childZorthProject
APPENDIX

PARENTSCRIPT - CubeScript

Property pBase  -- instance of base object which includes this object as a group
Property pID    -- identification of this object
property pSprite -- sprite number //if there is derived class object, base sprite number
property pMember -- sprite cast member //if there is derived class object, a list
property pLocz  -- sprite locz
property pVertexList -- coordinate in Modeling view //[[x,y,z,1]]/list[4x8]
property pCurrentVist -- coordinate for current modeling view //[[x,y,z,1]]/list[4x8]
property pOffset -- orig position of projection view on screen //point(x,y)
property pPosList -- coordinates on screen //[[point(x,y)]]/list[1x4]
property pPolyList -- the sequence for each vertex to form the panel object/[[1,2,3,4]]/list[4x6]
property pCenter -- the center coordinate of this object
property pChildList -- derived class objects included in this object //[[ID:instance]]/list[6]

global modelMatrix
global IDIndexList

-- CONSTRUCTOR: Create instance & initializing
on new me, base, ID, spriteNumber, castMember, vertexList, origOffset, polyList
   -- initializing
   pBase = base
   pID = ID
   pSprite = spriteNumber
   pMember = castMember
   pLocz = sprite(spriteNumber).locz           -- default sprite locZ
   pVertexList = duplicate(vertexList)
   pCurrentVist = duplicate(vertexList)   -- current vertexList to initialized vertexList
   pOffset = origOffset
   p = point(origOffset.locH, origOffset.locV)
   pPosList = [p,p,p,p,p,p]                   -- assign the orig as initialized position for each point on screen
   pPolyList = polyList
   pCenter = []                              -- update only when function called from outside
   pChildList = []

   -- initiate the instance of this object's child object
   initChildObject(me)
in order to receive stepFrame message, add this object to the actorList
add the actorList, me
return me

-- MEMBER FUNCTION: initiate the instance of this object's child object
-- PROPERTY: pSprite, pMember, pVertexList, pOffset, pPolyList, pChildList
-- FUNCTION CALL: getChildVertexList(me, polyList)
on initChildObject me
repeat with p = 1 to 6
    panID = pID&p
    panSprite = (pSprite + p - 1)
    panMember = pMember[p]
    panPolyList = pPolyList[p]
    panVertex = getChildVertexList(me, panPolyList)
end repeat
end

-- MEMBER FUNCTION: Determine the child object's pVertexList
-- ARGUMENT: polyList, child object's pPolyList property
-- PROPERTY: pVertexList
-- RETURN: tempVlist
on getChildVertexList me, PolyList
tempVlist = []
repeat with v = 1 to 4
    tempV = pVertexList[PolyList[v]]
    add tempVlist, tempV
end repeat
return tempVlist
end getChildVertexList

-- MEMBER FUNCTION: Determine the locz of this object
-- PROPERTY: pCurrentVist
-- RETURN: locz value
on getLocz me
tempz = pCurrentVist[1][3]
repeat with i = 2 to 8
    temp = pCurrentVist[i][3]
    if temp < tempz then
        tempz = pCurrentVist[i][3]
    end if
end repeat
-- first adjust tempz insure the value is always greater than 0
-- negative adjustment is due to the viewer is on -z axis, lower z coordinate has higher sprite locz
tempz = -tempz + 3000
return tempz
end getLocz

-- MEMBER FUNCTION: Plot this object on screen
-- PROPERTY: pChildList
-- FUNCTION CALL: draw(me), from child objects
on childDraw me
    repeat with child in pChildList
        child.draw()
    end repeat
end childDraw

-- MEMBER FUNCTION: get the center modeling coordinate of this object
-- PROPERTY: pChildList
-- RETURN: list[4]
on getCenter me
    -- thake child objects' center (front & back) then average them
    tempCenter = (pChildList[1].getCenter() + pChildList[6].getCenter()) / 2
    return tempCenter
end

-- MEMBER FUNCTION: transform the pCurrentVist coordinate
-- PROPERTY: pCurrentVist
-- FUNCTION CALL: multiMatrix(V,M)
-- GLOBAL: modelMatrix
on transform me
    pCurrentVist = multiMatrix(pCurrentVist, modelMatrix)
end transform

-- MEMBER FUNCTION: transform the pVertexList coordinate
-- PROPERTY: pVertexList
-- FUNCTION CALL: multiMatrix(V,M)
-- GLOBAL: modelMatrix
on AbsTransform me
    pCurrentVist = multiMatrix(pVertexList, modelMatrix)
end AbsTransform

-- MEMBER FUNCTION: parallel projection on to 2D screen
-- PROPERTY: pCurrentVist, pPosList, Offset
on zOrthProject me
    repeat with q = 1 to 8
        x = integer(pCurrentVist[q][1]) / 40 + Offset.locH
        y = integer(pCurrentVist[q][2]) / 40 + Offset.locV
        pPosList[q] = point(x, y)
    end repeat
end zOrthProject
Property pBase -- instance of base object which includes this object as a group
Property pID -- identification of this object
property pSprite -- sprite number //if there is derived class object, base sprite number
property pMember -- sprite cast member //if there is derived class object, a list
property pLocz -- sprite locz
property pVertexList -- coordinate in Modeling view // [x,y,z,1] / list[4x8]
property pCurrentVist -- coordinate for current modeling view // [x,y,z,1] / list[4x8]
property pOffset -- orig position of projection view on screen // point(x,y)
property pPosList -- coordinates on screen // [point(x,y)] / list[1x4]
property pPolyList -- the sequence for each vertex to form the panel object // [1,2,3,4] / list[4x6]
property pCenter -- the center coordinate of this object
property pChildList -- derived class objects included in this object // [ID:instance] / list[6]

global modelMatrix
global IDIndexList

-- CONSTRUCTOR: Create instance & initializing
on new me, base, ID, spriteNumber, castMember, vertexList, origOffset, polyList
-- initializing
  pBase = base
  pID = ID
  pSprite = spriteNumber
  pMember = castMember
  pLocz = sprite(spriteNumber).locz -- default sprite locZ
  pVertexList = duplicate(vertexList)
  pCurrentVist = duplicate(vertexList) -- current vertexList to initialized vertexList
  pOffset = origOffset
  p = point(origOffset.locH, origOffset.locV)
  pPosList = [p,p,p,p,p,p] -- assign the orig as initialized position for each point on screen
  pPolyList = polyList
  pCenter = [] -- update only when function called from outside
  pChildList = [:]

-- initiate the instance of this object's child object
initChildObject(me)
-- in order to receive stepFrame message, add this object to the actorList
add the actorList, me
return me
end new

-- MEMBER FUNCTION: initiate the instance of this object's child object
-- PROPERTY: pSprite, pMember, pVertexList, pOffset, pPolyList, pChildList
-- FUNCTION CALL: getChildVertexList(me, polyList)
on initChildObject me
    repeat with p = 1 to 6
        panID = pID&p
        panSprite = (pSprite + p - 1)
        panMember = pMember[p]
        panPolyList = pPolyList[p]
        panVertex = getChildVertexList(me, panPolyList)
    end repeat
end

-- MEMBER FUNCTION: Determine the child object's pVertexList
-- ARGUMENT: polyList, child object's pPolyList property
-- PROPERTY: pVertexList
-- RETURN: tempVlist
on getChildVertexList me, PolyList
    tempVlist = []
    repeat with v = 1 to 4
        tempV = pVertexList[PolyList[v]]
        add tempVlist, tempV
    end repeat
    return tempVlist
end getChildVertexList

-- MEMBER FUNCTION: Determine the locz of this object
-- PROPERTY: pCurrentVist
-- RETURN: locz value
on getLocz me
    tempz = pCurrentVist[1][3]
    repeat with i = 2 to 8
        temp = pCurrentVist[i][3]
        if temp < tempz then
            tempz = pCurrentVist[i][3]
        end if
    end repeat
end

-- first adjust tempz insure the value is always greater than 0
-- negative adjustment is due to the viewer is on -z axis, lower z coordinate has higher sprite locz
tempz = -tempz + 3000
return tempz
end getLocz

-- MEMBER FUNCTION: Plot this object on screen
-- PROPERTY: pChildList
-- FUNCTION CALL: draw(me), from child objects
on childDraw me
    repeat with child in pChildList
        child.draw()
    end repeat
end childDraw

-- MEMBER FUNCTION: get the center modeling coordinate of this object
-- PROPERTY: pChildList
-- RETURN: list[4]
on getCenter me
    -- thake child objects' center (front & back) then average them
    tempCenter = (pChildList[1].getCenter() + pChildList[6].getCenter())/2
    return tempCenter
end

-- MEMBER FUNCTION: transform the pCurrentVist coordinate
-- PROPERTY: pCurrentVist
-- FUNCTION CALL: multiMatrix(V,M)
-- GLOBAL: modelMatrix
on transform me
    pCurrentVist = multiMatrix(pCurrentVist, modelMatrix)
end transform

-- MEMBER FUNCTION: transform the pVertexList coordinate
-- PROPERTY: pVertexList
-- FUNCTION CALL: multiMatrix(V,M)
-- GLOBAL: modelMatrix
on AbsTransform me
    pCurrentVist = multiMatrix(pVertexList, modelMatrix)
end AbsTransform

-- MEMBER FUNCTION: parallel projection on to 2D screen
-- PROPERTY: pCurrentVist, pPosList, pOffset
on zOrthProject me
    repeat with q = 1 to 8
        x = integer(pCurrentVist[q][1])/40 + pOffset.locH
        y = integer(pCurrentVist[q][2])/40 + pOffset.locV
        pPosList[q] = point(x, y)
    end repeat
end zOrthProject
on exitFrame

global rubikObj
global mode
global stageCenter
global modelMatrix
global IDIndexList
global IDInstList
global relativePosList

--* build 3x3x3 eCube *--
-- base, ID, spriteNumber, castMember, vertexList, origOffset, polyList, childIDList, childshiftList, childIndexList
multiCubeSprite = 26
multiCubeMember = ["front", "top", "right", "left", "bottom", "back"]
multiCubeVertex = [[1000,-1000,-1000,1],[-1000,-1000,-1000,1],[-1000,1000,-1000,1],[1000,1000,-1000,1],[1000,1000,1000,1],[-1000,1000,1000,1],
                   [1000,-1000,1000,1],[-1000,-1000,1000,1],[1000,-1000,0,1],[1000,1000,0,1],[-1000,1000,0,1],[1000,0,1000,1],
                   [-1000,0,1000,1],[1000,0,0,1],[-1000,0,0,1],[1000,0,0,0],[-1000,0,0,0],[1000,1000,1000,0],[-1000,1000,1000,0],
                   [1000,-1000,0,1],[-1000,-1000,0,1],[1000,-1000,1000,1],[1000,-1000,1000,1]]
multiCubePolyList = [[6,5,8,7],[7,8,4,3],[5,1,4,8],[2,6,7,3],[2,1,5,6],[1,2,3,4]]
multiCubechildIDlist = ["AAA","BAA","CAA","AAB","BAB","CAB","AAC","BAC","CAC","ABA","BBA","CBA","ABB","BBB","CBB","BBC","ABC","CBC","ACA","BCA","CCA","ACB","BCB","CCB","ACC","BCC","CCC"]
multiCubechildshiftList = [[-2000,-2000,-2000],[0,-2000,0],[2000,-2000,0],
                          [-2000,0,0],[0,0,0],[2000,0,0],
                          [-2000,0,2000],[0,0,2000],[2000,0,2000],
                          [-2000,2000,0],[0,2000,0],[2000,2000,0],
multiCubechildIndexList = ["111","211","311","112","212","312","113","213","313","121","221","321","122","222","322","123","223","323","131","231","331","132","232","332","133","233","333"]
rubikObj = new(script "EightCubesScript", "none", "multiCube", multiCubeSprite,
multiCubeMember, multiCubeVertex, stageCenter,
updateStage

-- initialize IDInstList
IDInstList = rubikObj.pChildList

-- initialize and update relativePosList
relativePosList = [ [ "n", "n", "n" ], [ "n", "n", "n" ], [ "n", "n", "n" ] ,
[ [ "n", "n", "n" ], [ "n", "n", "n" ], [ "n", "n", "n" ] ,
[ [ "n", "n", "n" ], [ "n", "n", "n" ], [ "n", "n", "n" ] ] ]

setRelativePosList()
end exitFrame
on prepareMovie

cursor 260
clearglobals
the actorList = []
global rubikObj
  -- the rubik object
global mode
  -- for testing what kind of object is created
global stageCenter
  -- center point of stage
global modelMatrix
  -- matrix for modeling view transformation
global IDIndexList
  -- objects' pID & index table, for tracking the physical relative position of objects
      [ pID: Index ]
global IDInstList
  -- objects' pID & instance table, act as a lookup list
global relativePosList
  -- relative objects' pID list
global xIDList
  -- objects' pID list which have the same relative x position
global yIDList
  -- objects' pID list which have the same relative y position
global zIDList
  -- objects' pID list which have the same relative z position
global localxRotaDir
  -- local x-axis rotation direction, 1 for clockwise, 0 for counterclockwise
global localyRotaDir
  -- local y-axis rotation direction, 1 for clockwise, 0 for counterclockwise
global localzRotaDir
  -- local z-axis rotation direction, 1 for clockwise, 0 for counterclockwise
global rotateAngle
  -- rotation angle
global pressOn
  -- the clickOn boject's instance, if none, the value is 0

-- initializing
rubikObj = 0
mode = #none
centerX = 185
  -- (the stageRight - the stageLeft)/2
centerY = 160
  -- (the stageBottom - the stagetop)/2
stageCenter = point(centerX, centerY)
modelMatrix = []
IDIndexList = []
IDInstList = []
relativePosList = []
xIDList = []
yIDList = []
zIDList = []
rotateAngle = 10
pressOn = 0

-- initializing the visibility of sprites
repeat with f = 9 to 14
    sprite(f).visible = true
end repeat
repeat with f = 15 to 17
    sprite(f).visible = true
end repeat
    sprite(18).visible = false
end prepareMovie

-- FUNCTION: Create identity list[4x4]
-- RETURN: Identity matrix
on loadIdentity
    return [ [1.0,0.0,0.0,0.0],[0.0,1.0,0.0,0.0],[0.0,0.0,1.0,0.0],[0.0,0.0,0.0,1.0] ]
end loadIdentity

-- FUNCTION: Translation
-- ARGUMENT: Tx, x-axis translation amount
--          Ty, y-axis translation amount
--          Tz, z-axis translation amount
-- GLOBAL: modelMatrix
on setTranMatrix Tx, Ty, Tz
    global modelMatrix
    modelMatrix[4][1] = modelMatrix[4][1] + Tx
end translation

-- FUNCTION: set matrix of Rotate x-axis
-- ARGUMENT: s, sine value of the rotation angle
--          c, cosine value of the rotation angle
-- GLOBAL: modelMatrix
on setXrotateMatrix s, c
    global modelMatrix
    repeat with i = 1 to 4
        temp = modelMatrix[i][2]*c - modelMatrix[i][3]*s
        modelMatrix[i][3] = modelMatrix[i][2]*s + modelMatrix[i][3]*c
        modelMatrix[i][2] = temp
    end repeat
end translation

-- FUNCTION: set matrix of Rotate y-axis
-- ARGUMENT: s, sine value of the rotation angle
--          c, cosine value of the rotation angle
-- GLOBAL: modelMatrix
on setYrotateMatrix s, c
    global modelMatrix
    repeat with i = 1 to 4
        temp = modelMatrix[i][1]*c - modelMatrix[i][3]*s
        modelMatrix[i][3] = modelMatrix[i][1]*s + modelMatrix[i][3]*c
        modelMatrix[i][1] = temp
    end repeat
end translation
modelMatrix[i][3] = modelMatrix[i][1]*s + modelMatrix[i][3]*c
modelMatrix[i][1] = temp
end repeat
end translation

-- FUNCTION: set matrix of Rotate z-axis
-- ARGUEMENT: s, sine value of the rotation angle
-- c, cosine value of the rotation angle
-- GLOBAL: modelMatrix
on setZrotateMatrix s, c
global modelMatrix
repeat with i = 1 to 4
    temp = modelMatrix[i][1]*c - modelMatrix[i][2]*s
    modelMatrix[i][2] = modelMatrix[i][1]*s + modelMatrix[i][2]*c
    modelMatrix[i][1] = temp
end repeat
end translation

-- FUNCTION: set matrix of local rotation
-- ARGUEMENT: p, a point fall on the rotation arbitrary axis
-- sinV, sine value of rotation angle for z-axis rotation
-- cosV, cosine value of rotation angle for z-axis rotation
-- FUNCTION CALL: setXrotateMatrix()
-- setYrotateMatrix()
-- setZrotateMatrix()
-- setYrotateMatrix()
-- setXrotateMatrix()
-- GLOBAL: modelMatrix, rotateAngle
on setArbitraryRotateMatrix p, sinV, cosV
-- calculate each rotation angle value
V = sqrt( power(p[2],2) + power(p[3],2) )
L = sqrt( power(p[1],2) + power(p[2],2) + power(p[3],2) )

-- x-axis rotation
if v = 0 then
    sinI = 0
    cosI = 1
else
    sinI = p[2]/V
    cosI = p[3]/V
end if

-- y-axis rotation
if L = 0 then
    sinJ = 1
    cosJ = 0
else
    sinJ = p[1]/L
    cosJ = V/L
end if
-- build transformation matrix
setXrotateMatrix(sinI, cosI)
setYrotateMatrix(sinJ, cosJ)
setZrotateMatrix(sinV, cosV)
setYrotateMatrix(-sinJ, cosJ)
setXrotateMatrix(-sinI, cosl)
end setAbritraryRotateMatrix

-- FUNCTION: matrix multiplication
-- ARGUEMENT: V, M, the two matrices
-- RETURN: tempV, new matrix after multiplication
on multiMatrix V, M
  -- // multiple matrix // return matrix
  -- C(i,k) = \sum(j) A(i,j)*B(j,k)  max_i = count(V)  max_j = count(M)  max_k = count(M[1])
  max_j = count(M)
  if count(V[1]) = max_j then -- test requirement
    max_i = count(V)
    max_k = count(M[1])
    tempV = []
    repeat with i = 1 to max_I
      temp = []
      repeat with k = 1 to max_k -- which position in this list member
        sum = 0
        repeat with j = 1 to max_j -- sum the value
          sum = sum + V[i][j]*M[j][k]
        end repeat
        add temp, sum
      end repeat
      add tempV, temp
    end repeat
    return tempV
  else
    put "Error from multiMatrix!  V.colums != M.rows"
  end if
end multiMatrix

-- FUNCTION: get normal vector
-- ARGUEMENT: p1, p2, p3 vertex to calculate the normal vector
-- RETURN: normalV, normal vector
on getNormalVector p1, p2, p3
  normalV = [ (y32*z12 - y12*z32), (x12*z32 - x32*z12), (x32*y12 - x12*y32), 1 ]
  return normalV
end getNormalVector

-- FUNCTION: set relativePosList
-- GLOBAL: IDIndexList, relativePosList
on setRelativePosList
  global IDIndexList
  global relativePosList
  LCV = count(IDIndexList)
  temp = []
  repeat with i = 1 to LCV
    xIndex = value(char 1 of IDIndexList[i])
    yIndex = value(char 2 of IDIndexList[i])
    zIndex = value(char 3 of IDIndexList[i])
    ID = getPropAt(IDIndexList, i)
    relativePosList[xIndex][yIndex][zIndex] = ID
  end repeat
end setRelativePosList

-- FUNCTION: update IDIndexlist
-- ARGUMENT: oldList // objects' ID list according to old relative position, objects in this list is to ----
-- change relative position
-- GLOBAL: IDIndexList
on updateIDIndexList oldList, newList
  global IDIndexList
  newIDIndexList = duplicate(IDIndexList)
  repeat with i = 1 to count(oldList)
    repeat with j = 1 to count(oldList[i])
      oldprop = oldList[i][j]
      oldIndex = getaProp(IDIndexList, oldprop)
      newProp = newList[i][j]
      newIndex = getaProp(IDIndexList, newProp)
      newpropPos = getPos(IDIndexList, newIndex)
      newIDIndexList[newpropPos] = oldIndex
    end repeat
  end repeat
  -- put "FROM updateIDIndexList newIDIndexList"
  -- put newIDIndexList
  IDIndexList = newIDIndexList
End

-- FUNCTION: get a point which fall on the abritrary axis for local rotation
-- ARGUMENT: instList // list of all objects perform local rotation
-- FUNCTION CALL: getCenter(), getNormalVector(V1, V2, V3)
-- RETURN: pOnAxis
on getPointOnAxis instList
  global IDInstList
  lastIndex = count(instList) -- cause instList is 2x2 or 3x3, skip the second count
  inst1 = getaProp(IDInstList, instList[1][1])
  inst2 = getaProp(IDInstList, instList[lastIndex][lastIndex])
  tempPoint = (inst1.getCenter() + inst2.getCenter() )/2
  checkX = integer(tempPoint[1])
  checky = integer(tempPoint[2])
checkz = integer(tempPoint[3])

-- check if pOnAxis is the origin
if checkX = 0 and checky= 0 and checkz = 0 then
    V1 = inst1.getCenter()
    V2 = [0.0000, 0.0000, 0.0000, 1.0000]
    inst3 = getaProp(IDInstList, instList[1][3])
    V3 = inst3.getCenter()
    tempPoint = getNormalVector(V1, V2, V3)
end if
return tempPoint
end getPointOnAxis

-- FUNCTION: accept list and return the new list after counterclockwise 90 degree rotation
-- ARGUMENT: m // matrix to be rotated
-- RETURN: tempM, the new matrix
on rotateMatrix M
    maxC = count(M) -- max columns
    maxR = count(M[1]) -- max rows
    p = count(M[1]) + 1
    tempM = []
    repeat with i = maxC down to 1
        temp = []
        repeat with j = 1 to maxR
            addAt temp, 1, M[j][p-i]
        end repeat
        add tempM, temp
    end repeat
    return tempM
end rotateMatrix

-- FUNCTION: accept list and return the new list after clockwise 90 degree rotation
-- ARGUMENT: m // matrix to be rotated
-- RETURN: tempM, the new matrix
on backRotateMatrix M
    maxC = count(M) -- max columns
    maxR = count(M[1]) -- max rows
    p = count(M[1]) + 1
    tempM = []
    repeat with i = 1 to maxC
        temp = []
        repeat with j = maxR down to 1
            addAt temp, 1, M[j][p-i]
        end repeat
        add tempM, temp
    end repeat
    return tempM
end backRotateMatrix
-- FUNCTION: when stop move, clear global variables, delete all instances created by new()
on stopMovie
actors = count(the actorList)
    if actors then
        repeat with a = 1 to actors
            deleteAt the actorList, 1
        end repeat
    end if
end clearglobals
end