目录

[一、问题定义 Problem Definition 2](#_Toc514088083)

[二、使用说明 Instruction 2](#_Toc514088084)

[（一）使用手册 2](#_Toc514088085)

[（二）常见问题 6](#_Toc514088086)

[三、分析 Analysis 7](#_Toc514088087)

[（一）魔方可视化的算法分析 7](#_Toc514088088)

[1、完成从3维坐标点到2维坐标点的转化 7](#_Toc514088089)

[2、判断一个点是否被平面遮挡 1](#_Toc514088090)0

[3、计算某点旋转后的坐标 1](#_Toc514088091)2

[4、自定义数据类型 1](#_Toc514088092)3

[5、计时器实现动画 1](#_Toc514088093)4

[6、刻画魔方 1](#_Toc514088094)5

[(二) 三阶魔方复原算法的分析： 1](#_Toc514088095)7

[模拟人脑的复原过程。 1](#_Toc514088096)7

[四、解决方案 Solution 2](#_Toc514088097)2

[五、参考文献 Reference 2](#_Toc514088098)3

[六、附录 2](#_Toc514088099)4

# **一、问题定义 Problem Definition**

小组作业的核心问题包括两个部分：第一是给出复原打乱魔方的步骤，第二是把复原步骤用动画的形式播放出来。

在第一部分中，首先要给用户一个输入打乱魔方样式的接口。这个接口不应该是打乱魔方的平面展开图。如果这样的做的话，用户需要通过空间想象对平面展开图进行赋值，而这个过程非常容易出错。这个接口应该提供一个3D视角，使得用户能够即时看到已赋值面的情况，以确认自己的赋值过程没有出错。完成赋值之后，程序应根据用户的输入给出有限步复原步骤。

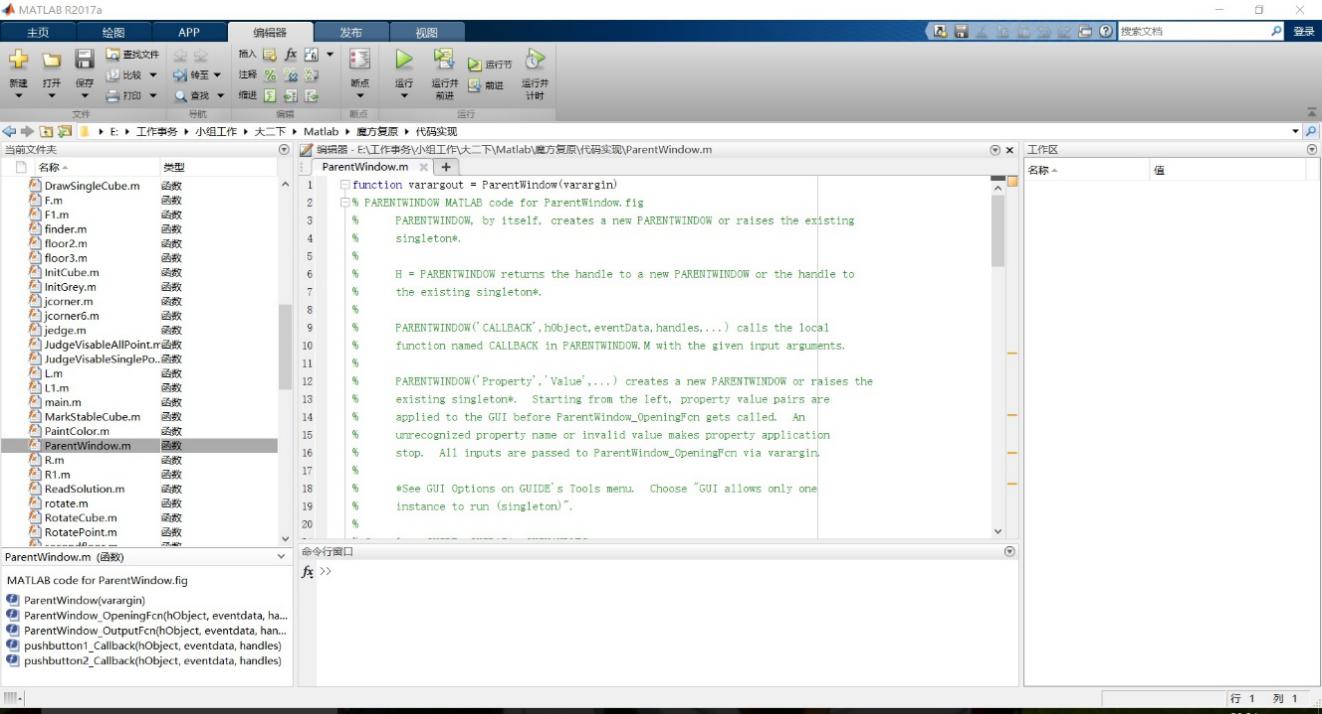
在第二部分中，程序已计算得出有限步复原步骤。这时程序应提供一个新的界面显示用户输入的魔方立体图，然后提示用户点击按钮或其他控件类型来依次序播放复原步骤。在播放过程中，魔方每一次旋转的过程也应被播放。否则魔方在打乱状态下，只通过观察某次旋转前和旋转后的图像得出魔方到底作了怎样的旋转，是非常困难的。魔方的立体图随着播放的进行而改变。程序应当允许用户在某次恢复步骤结束之后，调整观察魔方的视角，以在现实中核实魔方的变化是否属实。当复原步骤被执行完毕，显示出来的魔方应当是完全复原（此处的复原指魔方的六个面颜色都相同）。

# **二、使用说明 Instruction**

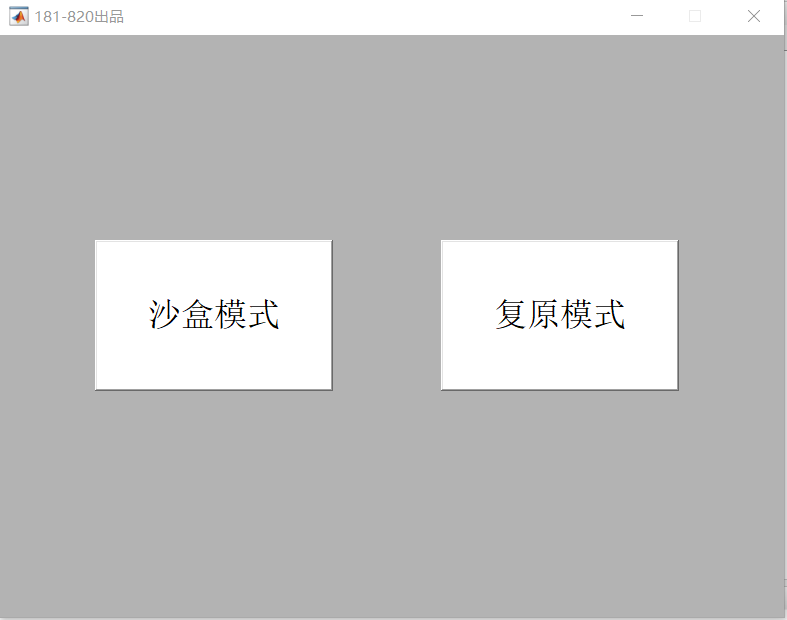
## （一）使用手册

以下对代码的全部使用过程都基于Matlab 2017a。如果使用2017b的话，程序的用户交互界面会出现变形，但不会影响基本功能。如果使用2014版本，那么程序的动画效果会完全丧失，并且会弹出大量警告和错误。由于程序中所有的动画效果都是由CPU运算处理（Matlab把数据传进GPU的时间是直接使用CPU计算的3-4倍），因此动画的效果与CPU配置有关。程序在win10 64位系统 内存8G CPU i5的电脑测试通过。如果在运行过程中仍有问题，请看常见问题。如果仍无法解决，请联系我们。

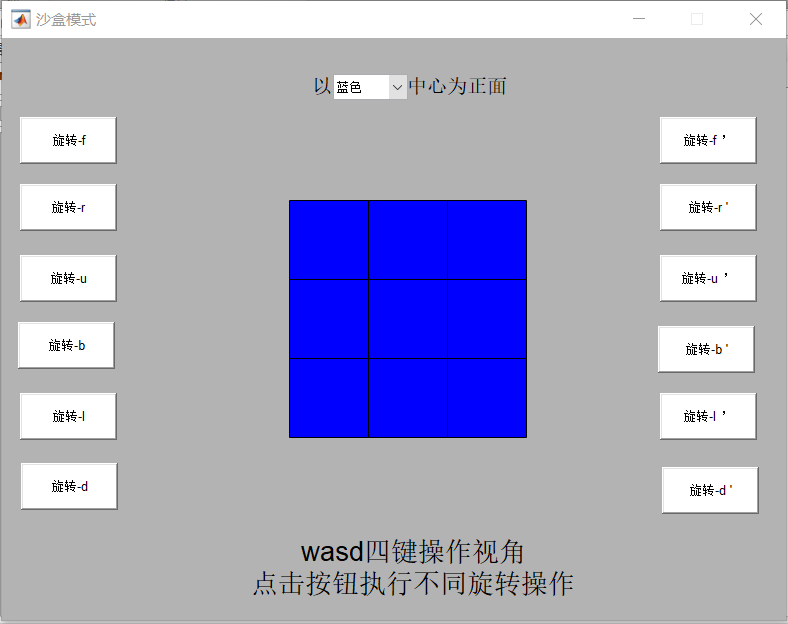
1.运行Matlab打开程序所在文件夹，文件夹应包括97个项目，其中2个mat类型，95个m文件。打开ParentWindow.m



2.点击运行，弹出窗口

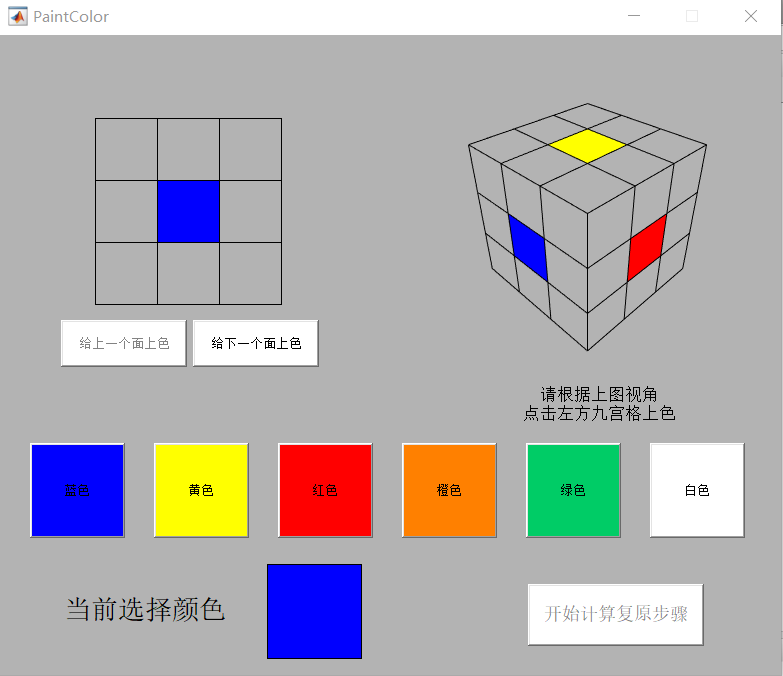


3.如果点击“沙盒模式”，稍等片刻，可弹出新界面

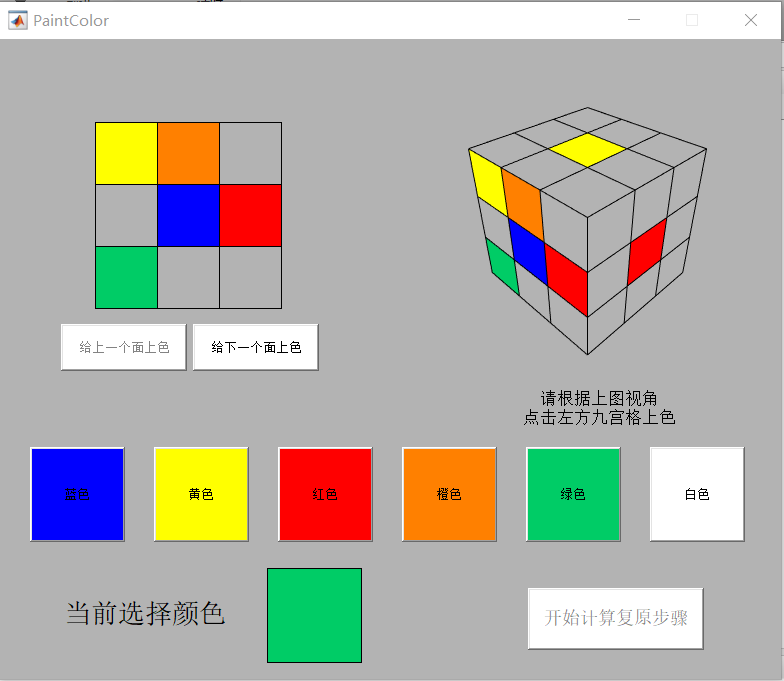


4.此模式下，用户相当于在玩一个虚拟的魔方，12个按钮分别对应对魔方的12种旋转方式，与国际通用的对应方式一致。按钮执行的旋转与选择的颜色中心有关。用户敲击键盘上的wasd可以用不同角度观察魔方即时的情况。

5.如果运行ParentWindow.m后点击复原模式，稍等片刻后会弹出新窗口

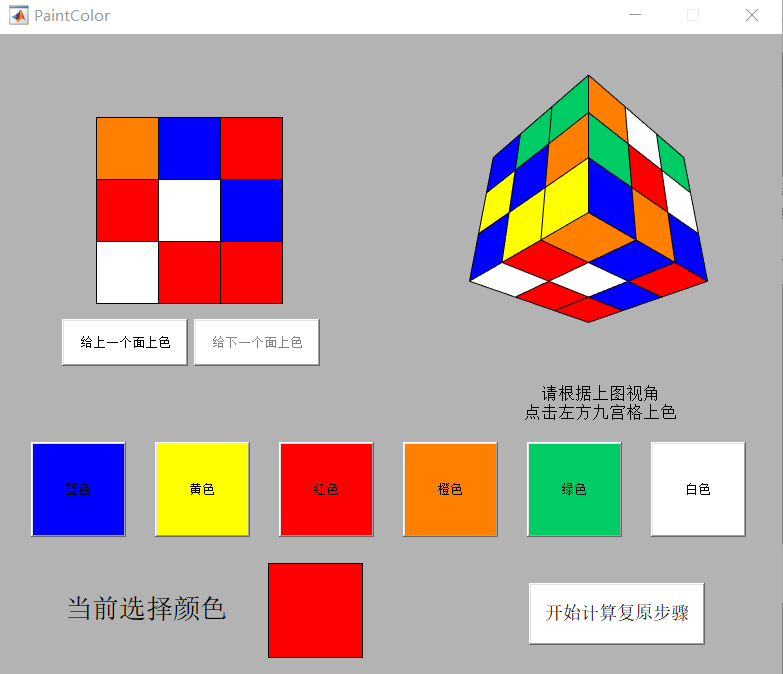


6.点击九宫格，右侧的魔方会即时显示目前的上色情况。请保持与右侧魔方一致的视角来个九宫格上色。也正因要规定上色的视角，因此在此界面下用户不被允许自由旋转魔方。

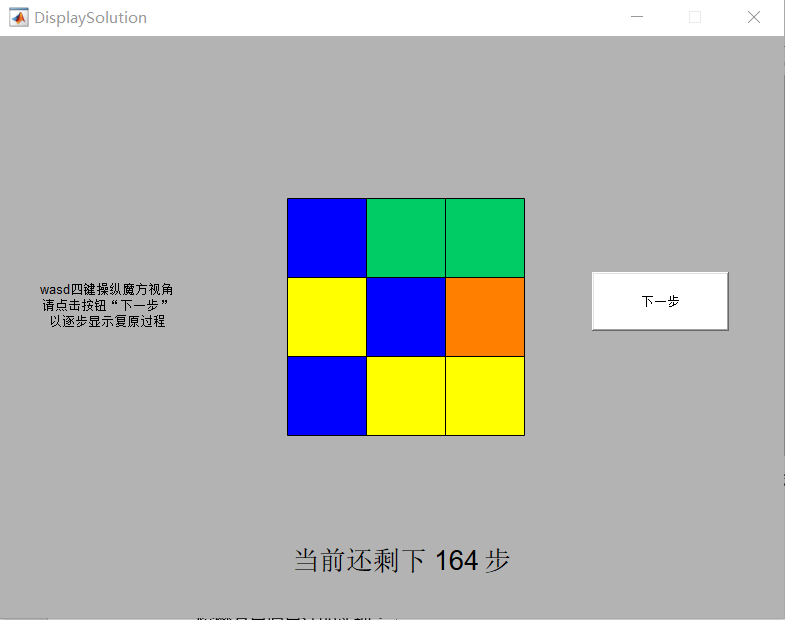


7.点击下方不同颜色的按钮，可以控制当前要涂的颜色

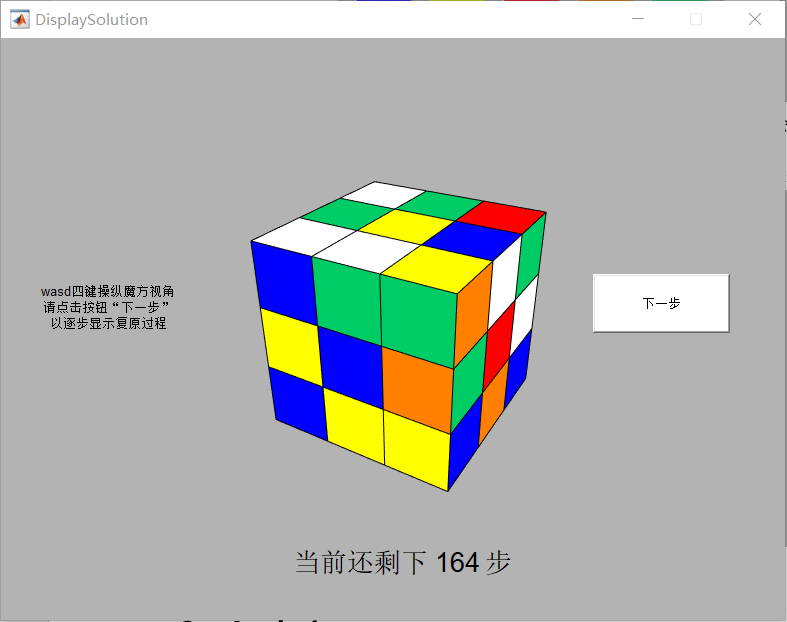
8.当魔方的每一个面都上色完毕，“开始计算复原步骤”按钮会变为可用。



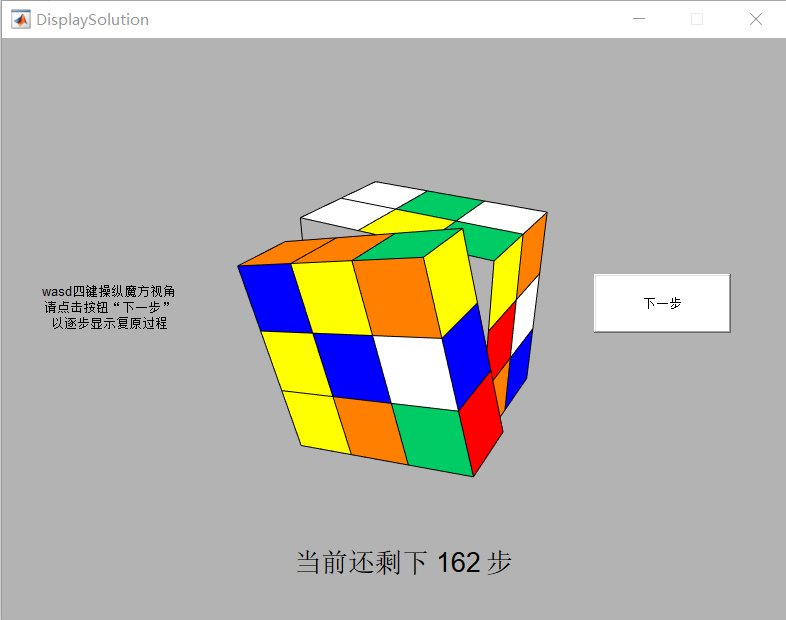
9.点击按钮“开始计算复原步骤”，该窗口会关闭，稍等片刻会有新窗口弹出



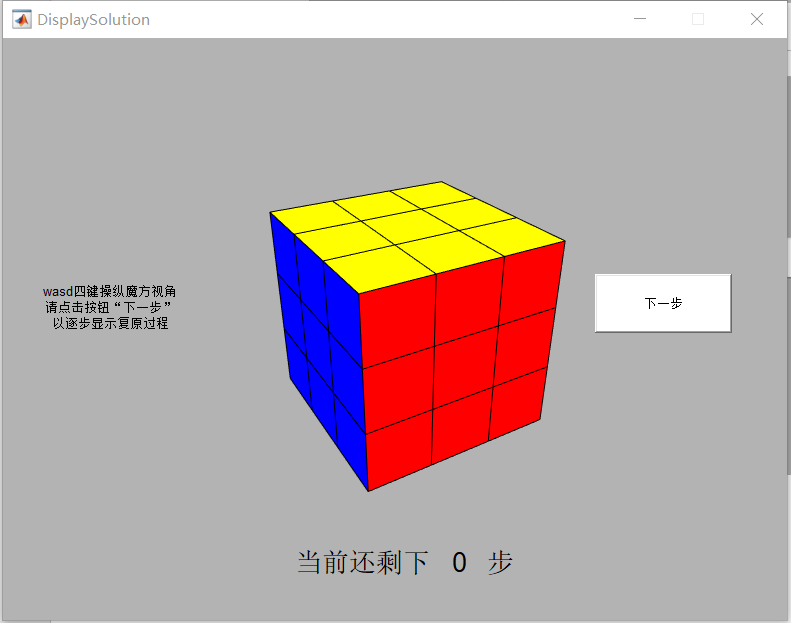
10.在此窗口下，用户可以点击键盘wasd调整自己观察的视角。



11.不断点击按钮“下一步”，则魔方会开始逐步复原



12.当剩余步数为0，魔方复原成功



## （二）常见问题

**Q1：**动画无法播放或者魔方部分方块消失

**A1：**Matlab版本会影响动画的播放效果，由于在绘图过程中需要设置句柄中成员变量的属性，因此在2014版本中运行会失败。尚未测试2017与2014版本以外的运行情况。

**Q2：**旋转动画能正常播放，但会出现明显卡顿

**A2：**在测试中，旋转动画播放时一般能做到0.11~0.13秒一帧，视觉效果应该是基本流畅的。由于所有动画都由CPU计算处理，如果动画出现了明显的卡顿，可能是CPU性能差别，也有可能是CPU被其他程序大量占用所致。

**Q3：**在复原过程中，某些旋转步骤毫无意义，例如先转U，之后又转了U’

**A3：**这个问题确实普遍存在于我们给出的魔方复原方法。其原因是我们的复原方法是模仿人在恢复魔方。为了减少运算工作量，我们每完成一个小目标，就会在把魔方“整体旋转一次”。详细原因请看本篇报告3.Analysis. 三阶魔方复原算法的实现.2

**Q4：**复原步骤很多步

**A4：**经过测试，一个完全打乱的魔方经本程序计算，一般需要170~230步彻底完成复原。原因既有如上一个问题所说的浪费步骤，但也有来自算法本身的原因。我们采用了魔方初级恢复算法而没有采用魔方速拧以及盲拧的编号方法。我们按照我们在现实中掌握的初级魔方复原公式编写本套程序。这也是我们的智力贡献之一，我们把人的步骤一般化，并用程序模仿出了人在复原魔方中的行为，包括搜索目标方块，调整视角，使用公式等。但我们承认我们的算法还有不少值得完善之处。

**Q5：**程序卡死，似乎是进入了死循环

**A5：**如果这种情况发生，那应该就是进入了死循环，可以直接强制退出Matlab,也可以在控制台中执行任意指令，然后中止本次执行跳出死循环。产生死循环的可能原因是输入的魔方是不可复原的，也就是说，假设输入的魔方在现实中存在，任何人也做不到把它完全复原，除非把它拆掉重装。我们的算法中只检查了每种颜色出现的次数，但这并不能保证用户所有的输入错误都能被排除。没能做到这点，确实是我们程序的一个遗憾。

**Q6：**魔方步骤读完了，但是依旧没有复原

**A6：**我们测试了20个完全打乱的魔方，以及若干种只拧了一步的魔方，尽管复原的步骤数要远大于我们打乱的步骤数，但最后魔方都被复原了。从我们原理上的设计来说，只要输入的魔方确实是能被复原的，我们的程序就能用有限步旋转把它复原。或许我们有考虑漏的地方，如果这种情况确实发生了，请保留文件夹中的MagicBox.mat，并联系我们。

**Q7：**我的魔方颜色和程序中的不同

**A7：**市面上确实存在各种配色不同的魔方。对于三阶魔方而言，它的六个中心相对位置是不变的。我们所使用的魔方样式，是市面上普遍流行的样式。如果您对此感到不满，也请联系我们，我们可以专门为您制作适合您的程序。

**Q8:**在我的电脑上运行程序，颜色效果与截图不同。

**A8：**在电脑屏幕上的颜色依赖于显示器的设置Matlab的版本。如果使用2014版本的话，可以发现蓝色非常深，以至于将近紫色了。这并不是程序的问题，只是Matlab不同版本对颜色的读取不同而已。如果调整显示器和Matlab版本后仍然色差明显，请联系我们。

**Q9：**为何魔方的顶面一直是黄色，底面一直是白色？能否通过调整视角使得黄色和白色作为正面呢？

**A9：**这个问题的答案是否定的。我们在一开始讨论多视角观察魔方的时候就已经考虑到了这个问题。如果想要实现多视角观察魔方，无非就两种思路，一种是观察者的位置在动，另一种是魔方整体在动。我们发觉很难正确描述魔方整体运动的方向矢量，其理解是存在歧义的。当魔方的旋转指令只来源点击按钮和屏幕，而不来自手的触觉，我们发现使用者很难做到把自己想要执行的旋转反映到点击的按钮和键盘上。这种思路是如此的费力而又不讨好，以至于我们决定只让使用者调整自己的视角。在这种思路下，如果不固定一个顶面和底面，我们按下“w”（上升镜头），我们还得确定此时你是正立着看魔方，还是倒立着看魔方，这两种结果完全相反。如果我们始终默认人是站立着看魔方，就不得不固定一个顶面和一个底面。

如果以上的问答无法让您感到满意，**请拨打电话13435417729李子天**

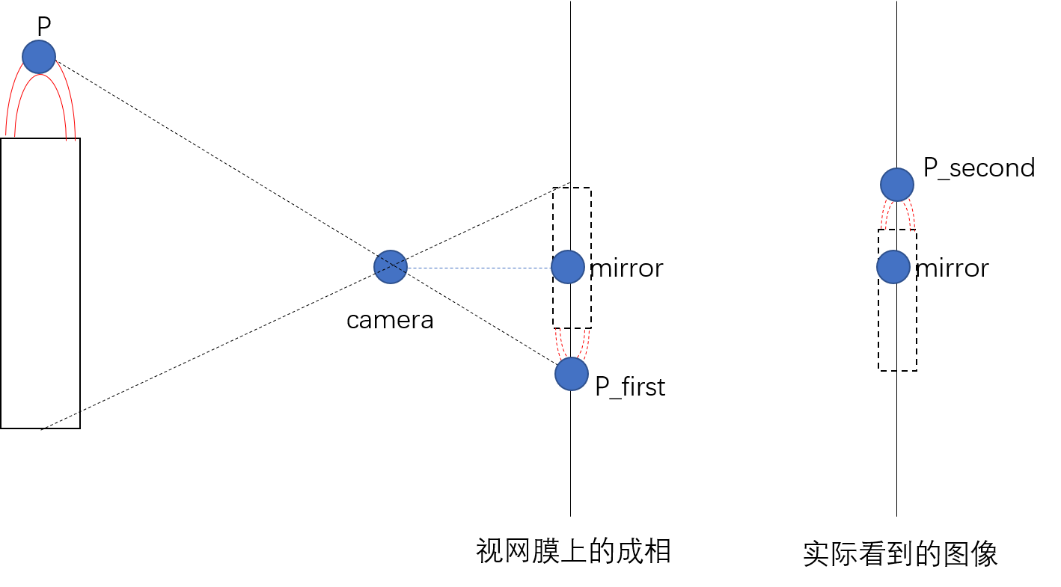
# **三、分析 Analysis**

## （一）魔方可视化的算法分析

### 1、完成从3维坐标点到2维坐标点的转化

如果要完成旋转动画，首先要考虑一个基本的问题。已知人的观察坐标和待观察物体的坐标。人看到的图像是怎样的？我们建立了以下的模型来解释这个问题。

#### 基本模型



这个模型来源于初中物理课上的“蜡烛成相”实验。我们把它放到三维坐标系下，并总结出了坐标转换的一般方法。

输入参数：空间中一点P，观察者位置camera，观察者位置到接收屏距离

1、连接P点与Camera,与接收屏交点为P\_first

2、用camera点和距离计算mirror空间坐标

3、以mirror点为旋转中心，P\_first为旋转点，计算P\_second空间坐标

4、以mirror点为二维平面原点，计算P\_second的相对坐标

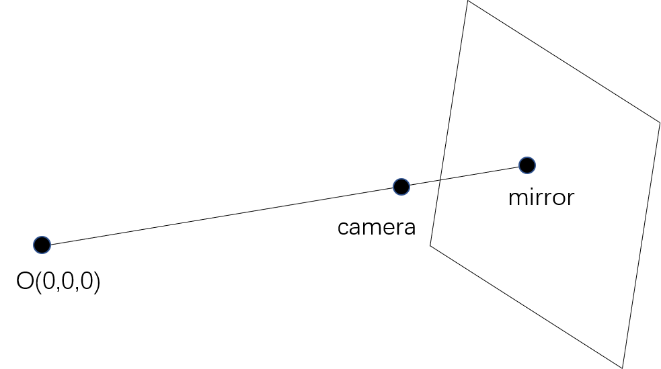
5、输出P\_second的相对坐标

#### 数学推导

1、从camera点求mirror点坐标

由空间坐标轴原点，camera点，mirror原点三点共线，我们得出数学关系

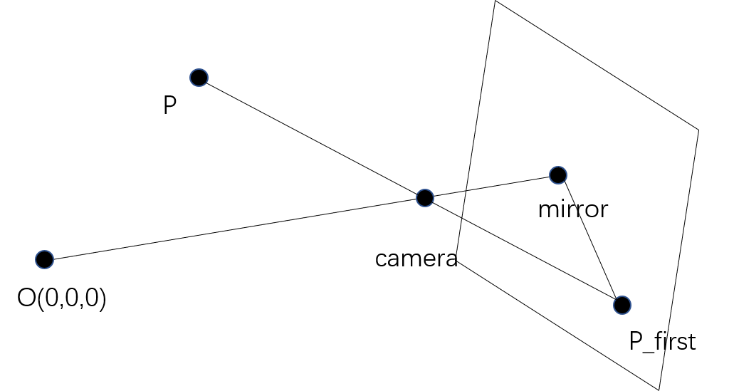
其中已知，可以通过camera的三维坐标求得，于是此方程可以解出O点坐标



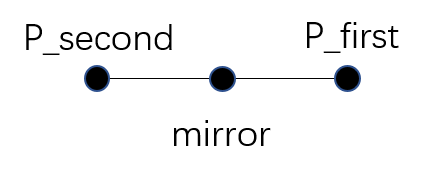
2、计算P\_first点坐标

根据P、camear、P\_first三点共线；以及与法向量垂直：

本质上这是四元一次方程组，可通过这组方程解出P\_first坐标和，可用于判断解是否合理。



3、计算P\_second点坐标

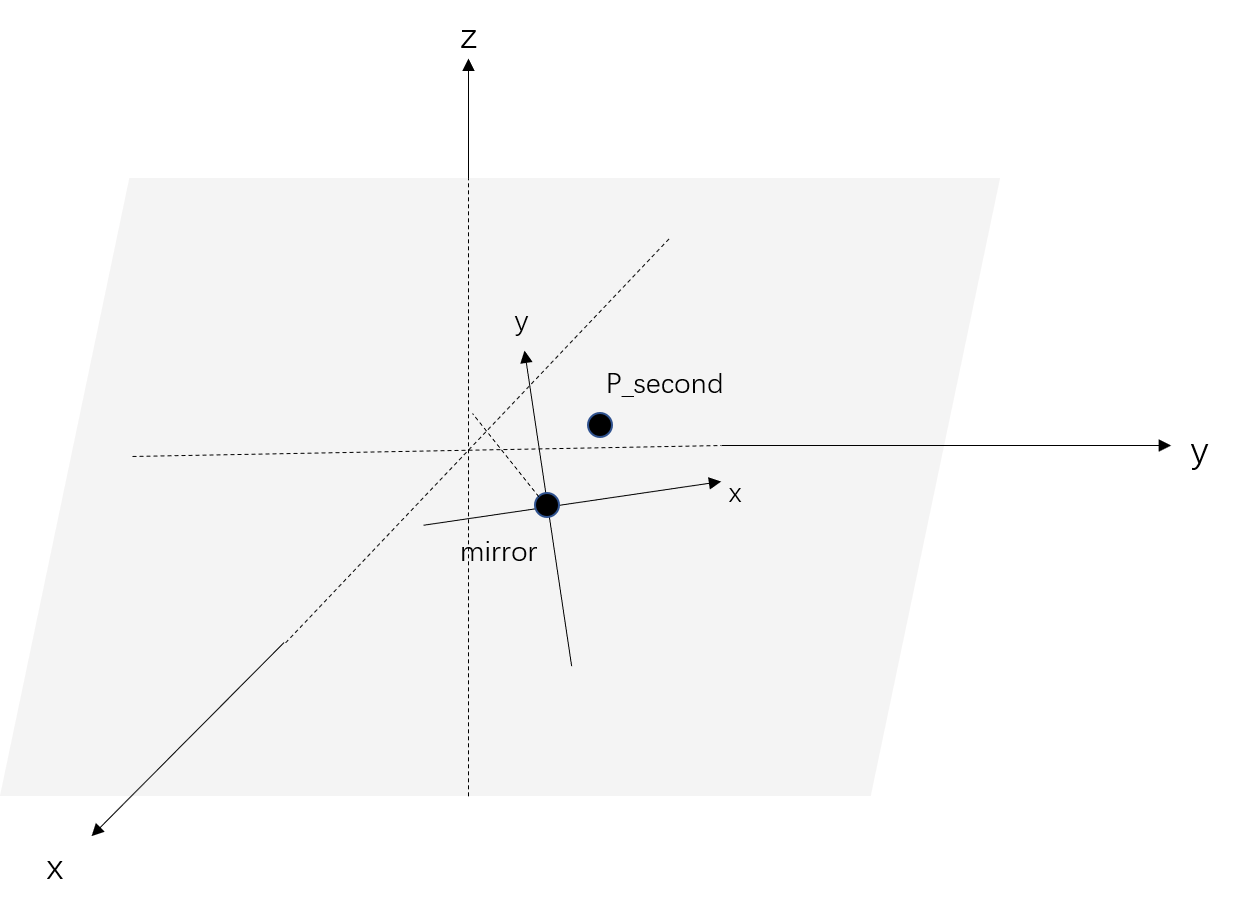


P\_second由P\_first以mirror为旋转中心旋转得到。

因此计算得到P\_second坐标

4、计算P\_second相对于mirror坐标

为方便整体运算，在本程序中我们没有使用传统右手坐标系的摆放方法

灰色的面为接收屏。我们在接收屏上以mirror点为原点建立一个坐标系，计算所有落在接收屏上点的相对mirror坐标，并把这个坐标在matlab中用plot输出到屏幕上。对于以mirror点为原点的二维坐标系， 过mirror点作与空间坐标系Z轴的垂线，则二维坐标系的x轴应与这条垂线垂直。取y轴与刚取好的x轴垂直。则我们以刚取好的两条坐标轴上的单位向量为基向量，可以解出P\_second的相对坐标。具体运算过程略去，最后可以得出结果：

x\_P也可以写成

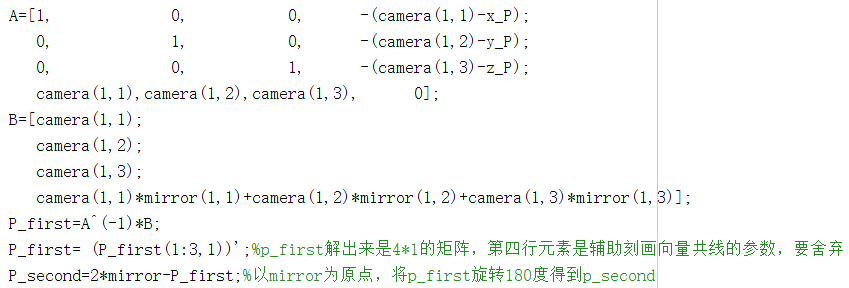
这两种写法等价

当mirror.x不为0时使用第一种写法算；为0时使用第二种写法算

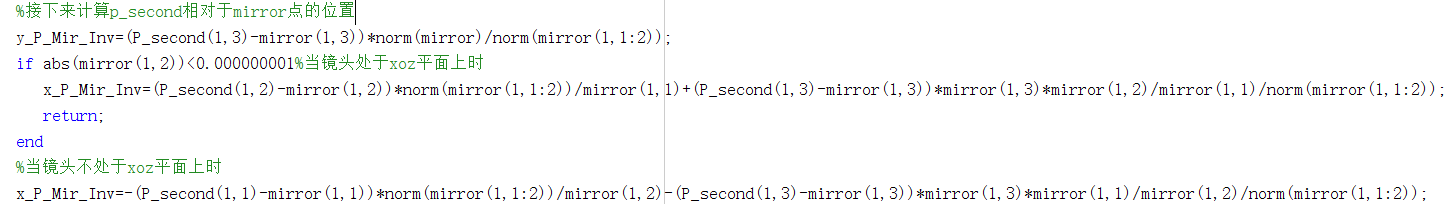
#### 代码实现——完成以上坐标转换的函数保存在Camera.m

计算mirror

计算P\_first



计算相对坐标

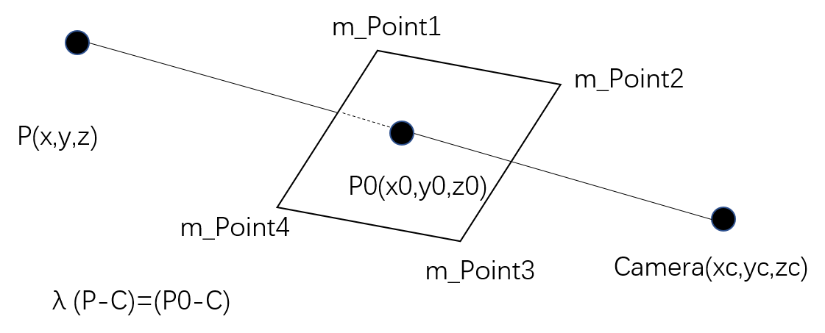


### 2、判断一个点是否被平面遮挡

#### 基本模型

在用plot,fill作图时，需要首先判断某些点是否真的需要被画到屏幕上。对于一个正常的立体图，有一些点会被平面遮挡。

那么如果已知目前自己视角的位置、小正方形四个端点的空间坐标、待观察点，我们需要判断待观察点到底能否被看到，进而确定是否把这个点画到屏幕上



#### 数学推导

我们默认m\_Point1,m\_Point2,m\_Point3,m\_Point4（m\_PointX是在程序中的写法，以下简称P1,P2,P3,P4）四点确实是共面的，不再检查其输入合理性。

我们使用两种方法判断P0点是否在小正方形内。分别是以及判断参数（用于刻画和共线）以及求P0相对坐标。

使用参数时，我们列出方程：，如果最后解得，说明P点与Camera在平面同一侧。这个时候当然能看到P点。如果最后解得，说明P点与Camera点在平面两侧，需要进一步求P0相对坐标完成判断。如果说明这个平面本身都无法被看到，也就不应该用来判断是否遮挡。在本程序中，的情况不会出现。

如果经过第一种方法发现P与Camera在平面两侧，我们需要进一步求P0的相对坐标。我们的思路是连接Camera与P点，联立平面方程求解P0点的坐标。再以和为基向量求P0在正方形内的相对坐标。如果这个相对坐标在内，我们就可以说P0在小正方形内。

实际在操作过程中，我们并没有先求，再求P0相对坐标，这些都可以通过可以通过线性方程组一次性全部求出。但在判断过程中，我们先判断了的范围，再判断了P0的相对坐标，最后返回是否能够观察到P点的结论。

方程如下：

，其中a和b是要求的P0相对坐标

把向量形式写为线性方程组的形式：

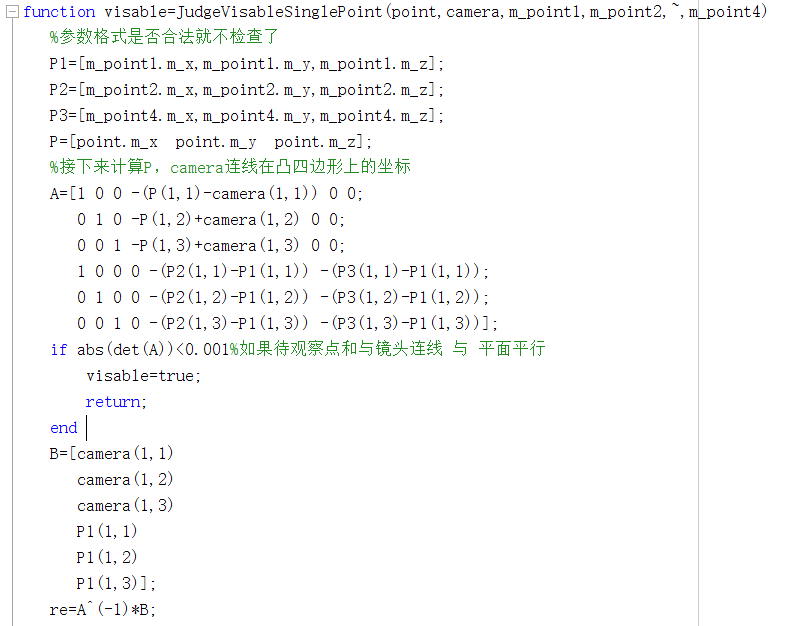
整理得：

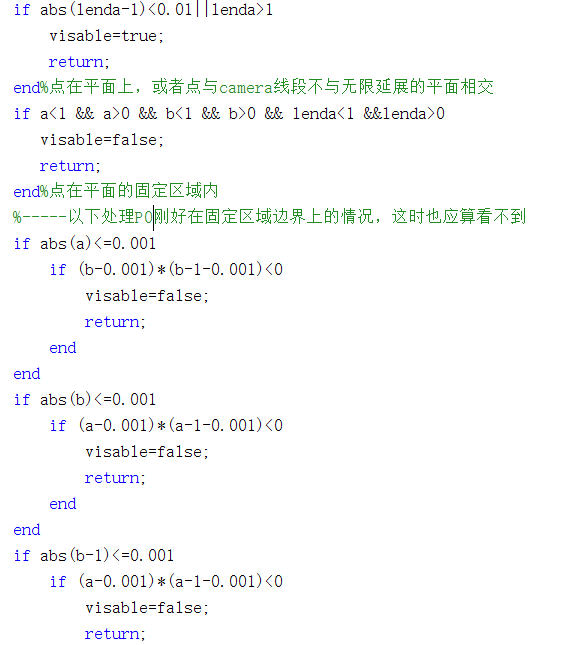
如果这个六元一次线性方程组有无穷组解（系数矩阵行列式为0），说明此时P点与Camera连线与平面平行，此时程序直接退出，并返回可以看到的信息。

在有解的情况下，解该方程即可求出和P0的相对坐标a,b。

由于计算机内浮点数表达不精确的情况，例如原本应解出=0的情况下可能解出。所以在判断条件时本程序尽量避免使用相等，而是允许误差在-0.000001到0.000001内。

#### 代码实现——本部分代码保存在JudgeVisableSinglePoint.m

求解线性方程组 判断点是否可见

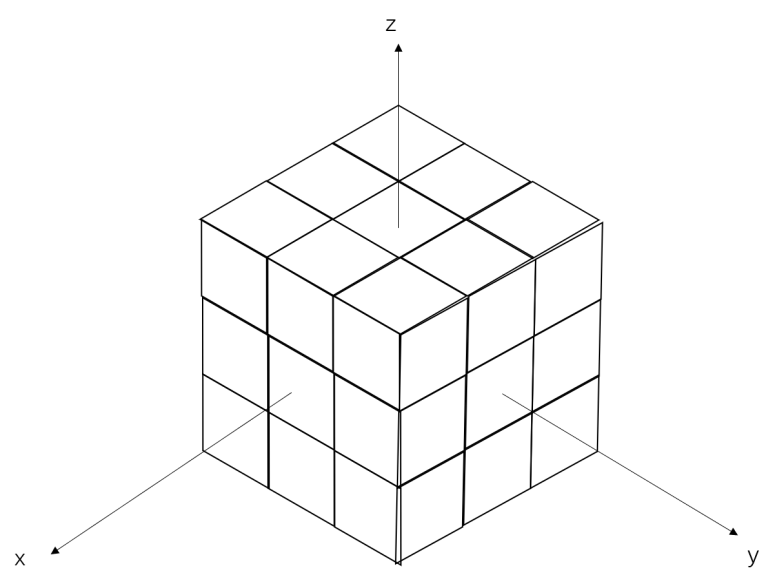


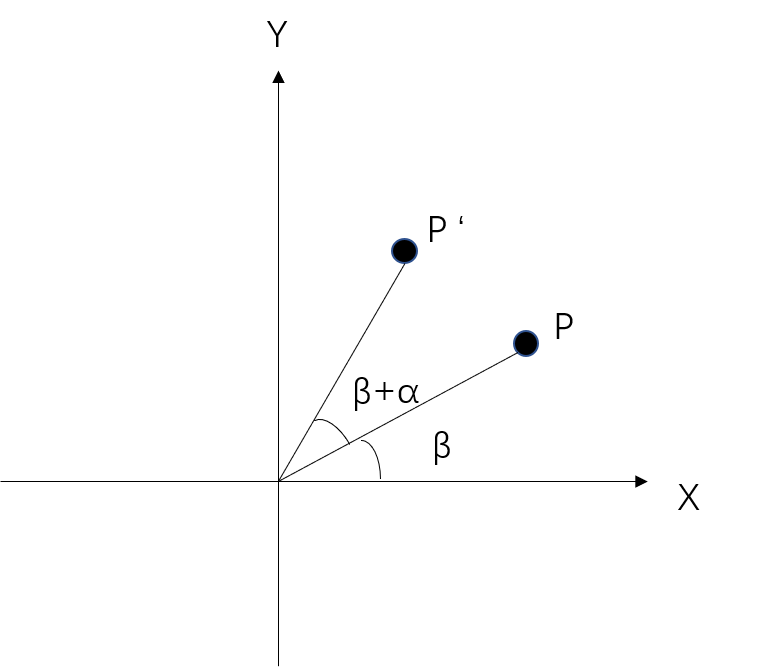
当我们能够判断出某点在当前视角下是否可见，我们就能对应的判断包含这个点的平面和直线是否可见，进而决定是否要把这些平面和直线画出来。

### 3、计算某点旋转后的坐标

#### 基本模型

在逐帧画出旋转的过程中，需要不断的更新每一个小方块8个顶点在旋转一定角度后的坐标。在本程序中，只有三种旋转：绕空间中x轴旋转，绕空间中y轴旋转, 绕空间中z轴旋转。这三种旋转本质上没有区别。只要我们解决了任何一种，另外两种原理类似。





#### 数学推导

于是我们首先研究在二维平面上点围绕原点旋转之后坐标的变化（P->P’）。我们的思路是把直角坐标转化为极坐标，先算旋转前点的极坐标，再算旋转后点的极坐标，最后把旋转后极坐标转化为直角坐标。

已知P点坐标为（x,y），求P点逆时针旋转α度后P’点的坐标。

1、易得P极坐标为，其中

2、则P’极坐标为，

3、把P’极坐标转化为直角坐标

=

类似的，可得：

把结果转化为矩阵运算：

如果是顺时针转动，取就好

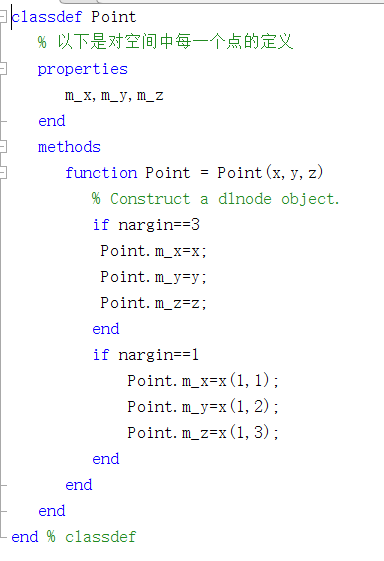
以上是在二维平面时的情况。在本程序中，需要把三维问题先简化为二维问题。例如使空间中某个点绕x轴旋转,则可以取该点(y,z)作为以上讨论的(x,y)。

#### 代码实现——本部分代码保存在RotatePoint.m中

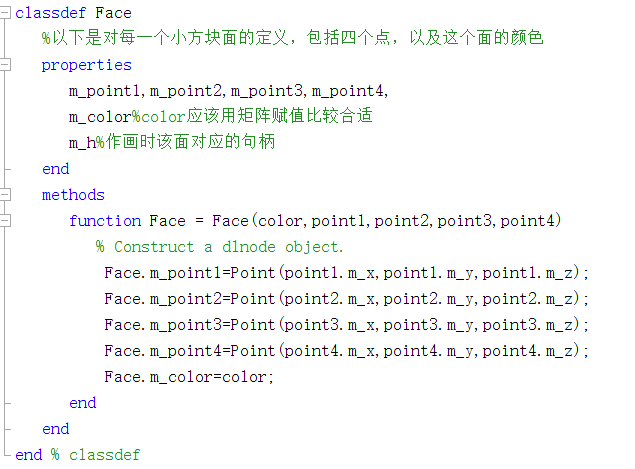
我们用输入参数center作为区别三种旋转（绕x轴，绕y轴，绕z轴）的参数。Center是一个1\*3的矩阵。程序内部已经保证输入的center只有一个非零元素，此非零元素的位置决定RotatePoint执行什么运算

### 4、自定义数据类型

三阶魔方一共包含64个顶点，36个小面，27个方块。虽然我们可以分别使用矩阵涵盖所有信息，然后在使用时再提取子矩阵的信息。但由于这个当矩阵过大时，很有可能在使用的过程中出错，并且维护和调试的时候难度很大，我们针对本程序的需要新定义了四种数据类型，分别是Point，Face，Cube，CubeWithoutColor。



Point.m中保留了Point类的完整代码定义。 Point类描述的是空间中的一个点。它只有三个成员变量，分别对应空间中的一点坐标。它的构造函数允许有两种初始化方式，一是直接使用三个数字赋值，二是使用一个1\*3的矩阵完成赋值。使用该类定义点后，调用点的某个坐标不再使用矩阵选取元素的方式。P.m\_x会比P（1,1）的可读性强很多。

 Face.m中保留了Face类的完整代码定义。Face类描述的是魔方中某一个小方块的某一个面的性质。它的成员包括这个面得四个点（均为Point类型），一个描述这个面颜色的矩阵m\_color，还有一个成员变量保存该面被显示在屏幕上时的句柄。它的构造函数需要5个变量，分别对应它的四个Point类型的点和颜色矩阵。它的成员变量m\_h只有在作图时才会被赋值和使用。



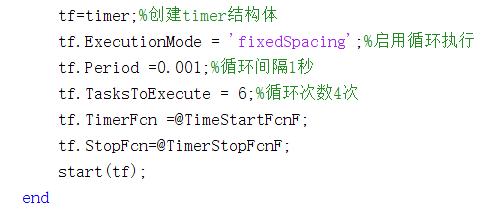
Cube.m中保留了Cube类的完整代码定义。Cube类描述的是魔方的小方块。这个类包含8个Point类型的成员变量，6个描述颜色的成员变量，6个Face类型的成员变量。使用这个类定义，可以把三阶魔方用3\*3\*3个Cube表示。涉及旋转、透视功能时，程序可以方便的读取某一个小正方体的某一面颜色，代码可读性增强很多。由于魔方的每一个小块都是正方体，因此其构造函数只要输入整个正方体的“中心”，就可以完成对8个顶点的赋值



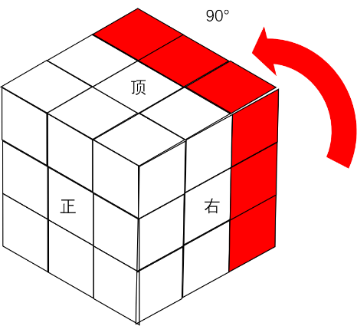
CubeWithoutColor.m中保留了CubeWithoutColor类的完整定义。这一个类只会在函数要判断点是否可见时使用。由于在判断点是否可见时不仅仅只是要判断一个面是否遮挡，还需要判断一个长方体的六个面是否遮挡住了某个点。我们使用这个类来存放本次判断下可能遮挡住视角的长方体。它的8个Point类型成员变量对应一个长方体的8个顶点，在判断一点是否可见时，我们从这8个点里按照一定顺序给JudgeVisableSinglePoint提供输入参数，当一个点不会被任何平面遮挡时，我们才返回这个点可以被看到的信息。

### 5、计时器实现动画

为了能够逐帧地画出旋转过程，本程序需要使用Matlab自带的结构体timer计时器来完成逐帧绘制。

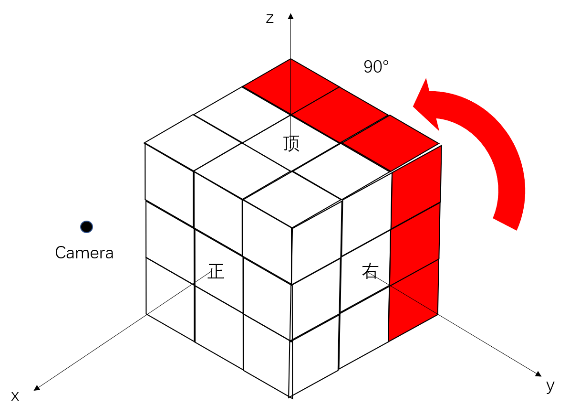
 在Matlab中，timer结构体可以模仿多线程的处理过程。通过设置timer结构体的参数设置，我们能够做到按照固定频率执行一个函数（图片中为TimeStartFcnF函数）。本程序把timer发送消息的规律定为：前一个TimeStartFcnF执行完毕后，等待0.001秒，再执行一次TimeStartFcnF，执行6次为止。当执行够6次以后，timer结构体销毁，销毁前执行TimeStartFcnF。由于执行一次TimeStratFcnF的时间为0.11-0.13秒，而人眼无法察觉0.001秒的差别，因此这种执行方式能够使每一帧都能出现在屏幕前一次，效果也最为流畅。

我们在TimeStartFcnX函数里里完成了图像的绘制，由于视觉残留，人眼会感觉魔方真的旋转起来了。本程序一共有12个结构类似的TimeStartFcnX函数和12个TimeStopFcnX函数,分别对应12种对魔方的操作。我们又准备了12个结构类似的Animation\_X函数，用来初始化12种即时器消息，以下我们以“B”旋转为例介绍Animation\_X函数的功能



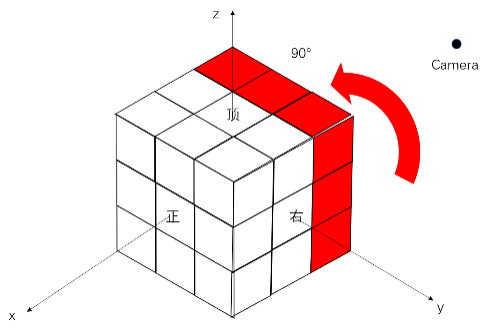
B旋转示意图

在Animation\_B函数内，首先，我们判断用户执行本次旋转时观察视角的位置。



如果Camera在正面（本程序中判断条件为Camera.x>-1）,我们在绘制魔方的顺序是：先画红色部分，再画白色部分。并且每次绘制结束，都应该把整个魔方擦掉，然后再开始制图。

在绘制开始前，我们使用MarkStableCube函数（MarkStableCube.m）记录下白色部分魔方每一个能看到的面的颜色和所在位置。等待计时器发出消息，收到消息之后对红色部分所有方块执行RotateCube函数（RotateCube.m）,再在RotateCube内部调用RotatePoint函数（RotatePoint.m）对一个魔方小方块的所有点进行旋转。所有方块的旋转都进行完毕之后，绘制红色部分方块在屏幕中的图案。之后根据MarkStableCube函数标记下的没有任何移动的颜色区域，绘制白色部分方块在屏幕中的图案，这样就自然地实现遮挡效果。本次计时器消息执行完毕，等待下一次计时器消息。下一次计时器消息执行前先用cla指令把所有图案擦除。



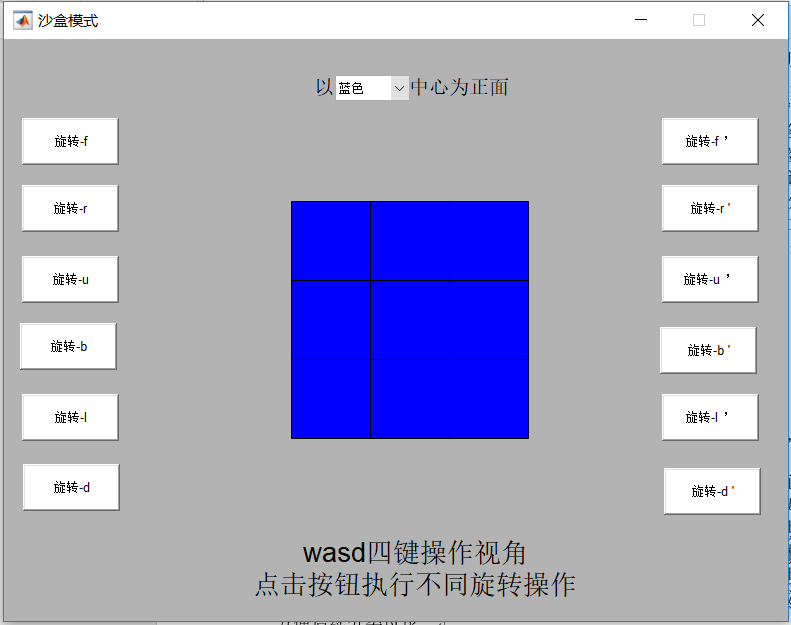
如果Camera在背面（本程序中判断条件为Camera.x<=-1），我们绘制魔方的顺序是：先画白色部分，再画红色部分。不过这种情况下我们只需擦掉上一次的红色部分，就能再画下一次的红色部分。

这种情况下，我们不再需要调用MarkStableCube函数，而只需重画一次白色部分所有方块的图案，保留句柄，**把句柄的HandleVisibility设为‘Callback’**。之后我们使用的cla指令就只会擦除红色部分的方块图案。这一步绘制结束之后，等待计时器发出消息，收到消息之后对红色部分所有方块执行RotateCube函数（RotateCube.m）,再在RotateCube内部调用RotatePoint函数（RotatePoint.m）对一个魔方小方块的所有点进行旋转。所有方块的旋转都进行完毕之后，绘制红色部分方块在屏幕中的图案。等待下一次计时器消息。下一次收到时间消息后先使用cla清除所有红色区域方块的图案。

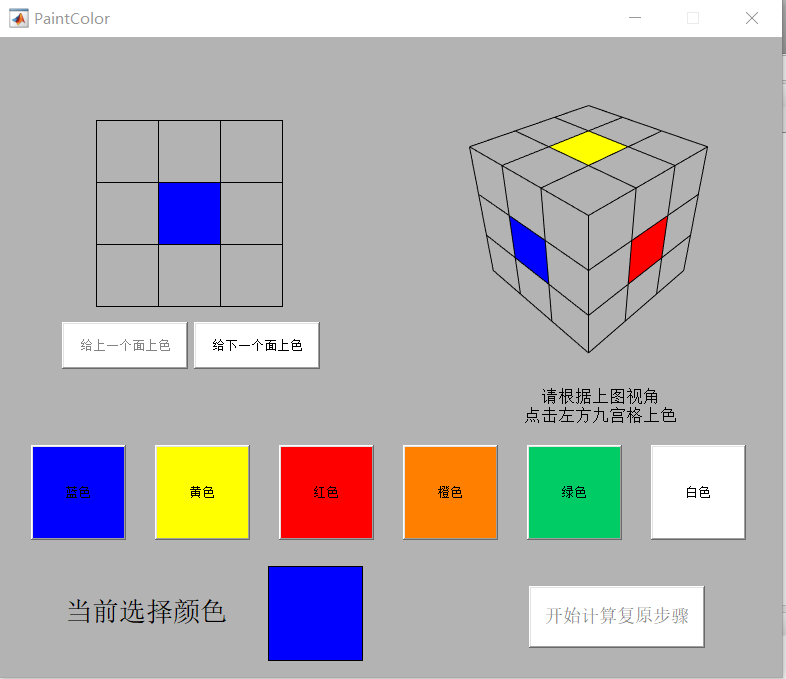
### 6、刻画魔方

在以上五点想法的基础上，我们提出了刻画魔方的办法（与复原部分的刻画方式有较大不同，原因是复原魔方的计算过程不需要可视化，也就不需要保留坐标、颜色等信息）。我们把一个魔方的所有信息都储存在一个Cube类型的3\*3\*3结构体数组，这个结构体数组的每一个元素都是Cube类型。在本程序中，这个结构体数组的变量名是integralcube。integralcube作为全局变量可以被程序中任意函数读取和修改。

当我们需要针对某一个，或者某一些小方块进行操作时，比如说判断后方楞块的颜色，我们可以用访问integralcube(1,2,1).m\_colorX。本程序的功能分为两个大部分，第一个大部分是沙盒模式，用户相当于在玩一个虚拟的魔方。第二大部分是复原模式，给用户提供输入打乱魔方样式的接口，输出魔方复原的步骤和动画。以下分别简要介绍在这两种模式下integralcube变量的使用。

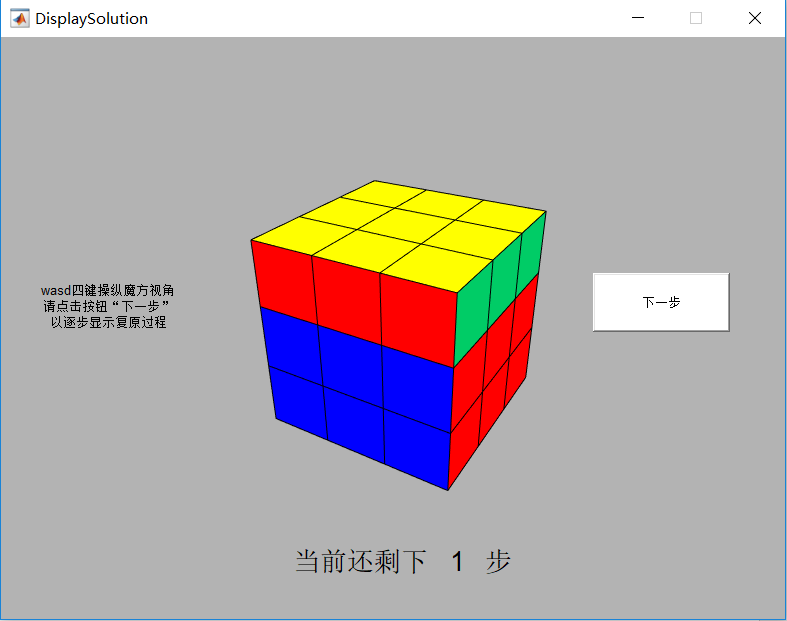
 在沙盒模式下（对应fig文件为Canvas.fig，但请不要双击直接运行该文件），我们在Creat消息的响应函数中调用InitCube函数（InitCube.m）对integralcube进行赋值。在InitCube赋值过后，魔方是已经复原好的。用户可以在此基础上任意打乱，也可以打乱之后尝试复原。每次旋转后integralcube的更新在TimeStopFcnX中执行。当一个旋转正在进行中，一个on\_off的变量会被设为真，只有当这个变量值为假时，用户点击按钮才会有效果，以此防止用户点击过快使程序出错。

沙盒模式



在复原模式（接口对应fig文件为PaintColor.fig，但请不要直接双击运行，可能会产生不可预知错误）下。在PaintColor.fig的Creat消息响应函数下，我们调用InitGrey函数完成对integralcube的赋值。InitGrey只会给魔方的6个面的中心进行赋值。当用户点击九宫格时，九宫格的单击消息响应函数会对对应面的颜色进行赋值，并调用DrawCube把赋值的最新情况画出来

复原模式（输入接口）



在复原模式的输出界面内,integralcube的变化与在沙盒模式下基本一致。只不过旋转的命令不来自用户的输入，而来自用户计算得出的指令。之后根据指令使用不同的Animation\_X函数完成旋转动画的赋值

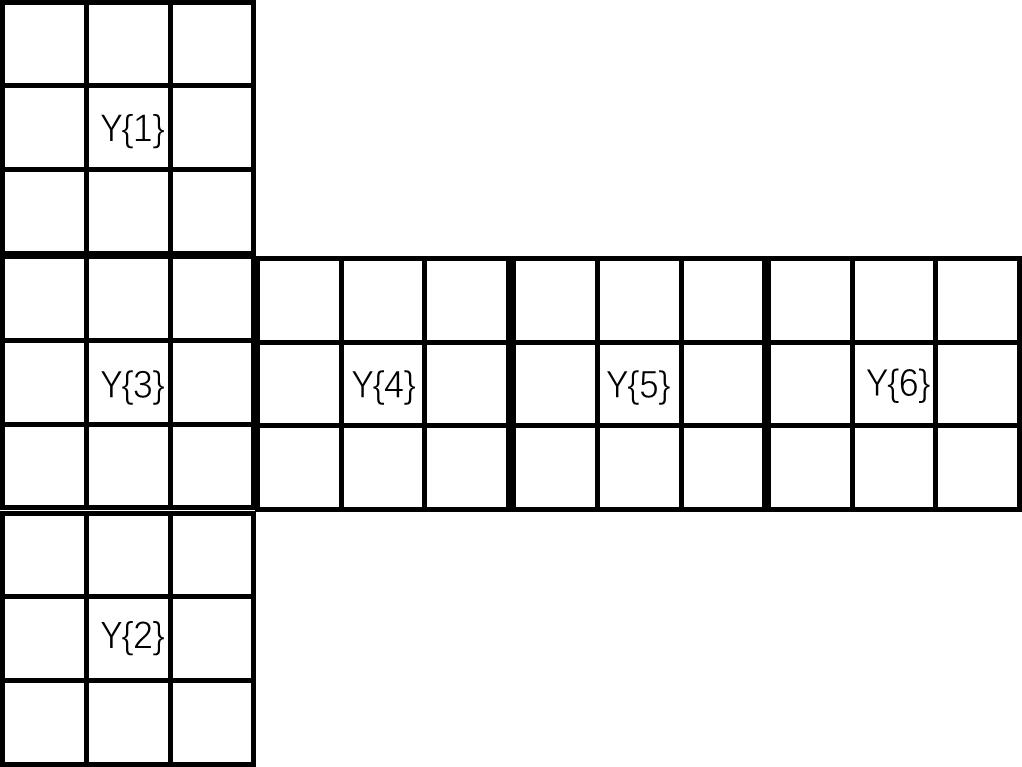
复原模式（输出界面）

## (二) 三阶魔方复原算法的分析：

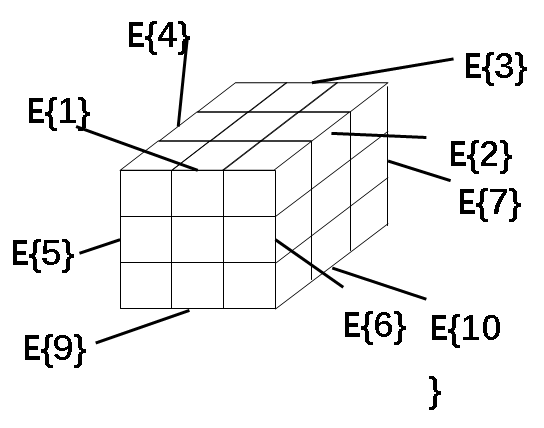
### 模拟人脑的复原过程。

使用层进法的复原，基本分为以下几步：拼出底层十字（Edge cube的复原），拼出底层对应的角块（Corner cube），拼出第二层的Edge Corner，拼出顶层的十字（cross），使顶层的角块（Corner）位置正确，使角块（Corner）朝向正确。[[1]](#footnote-1)

在复原过程中首先把魔方视为6个3\*3的矩阵，六个颜色赋值为1到6，对于每个面的旋转执行操作就是对矩阵的行和列进行操作。据此我们可以定义出12个操作函数，即U，U1，R，R1之类的函数。复原过程中，为了确定每个角块和棱块的位置，我们使用E(1,2,12)和C（1,3,8）这样的矩阵来完成对每个角块和棱块颜色的确定。对于12个棱块，按从上到下顺时针方向赋值1到12，对于8个角块使用C来确定位置和颜色。通过自己编写finder函数来查找所需要颜色的角块或棱块所处的位置，方便后续分类讨论。



以下为对12个edge和8个corner的 另一种定义方式 方便finder函数查找对应的位置

E=zeros(1,2,12);

E(1,:,1)=[Y{1}(3,2),Y{3}(1,2)]; E(1,:,2)=[Y{1}(2,3),Y{4}(1,2)];

E(1,:,3)=[Y{1}(1,2),Y{5}(1,2)]; E(1,:,4)=[Y{1}(2,1),Y{6}(1,2)];

上四E

E(1,:,5)=[Y{6}(2,3),Y{3}(2,1)]; E(1,:,6)=[Y{3}(2,3),Y{4}(2,1)];

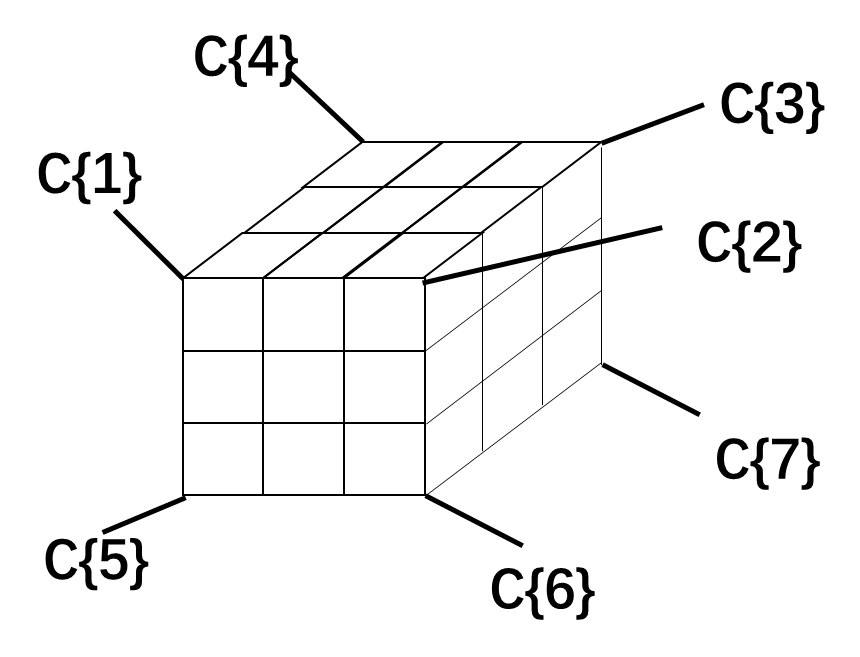
E(1,:,7)=[Y{4}(2,3),Y{5}(2,1)]; E(1,:,8)=[Y{5}(2,3),Y{6}(2,1)];

中四

E(1,:,9)=[Y{3}(3,2),Y{2}(1,2)]; E(1,:,10)=[Y{4}(3,2),Y{2}(2,3)];

E(1,:,11)=[Y{5}(3,2),Y{2}(3,2)];E(1,:,12)=[Y{6}(3,2),Y{2}(2,1)];

上四C below

C=zeros(1,3,8);

C(1,:,1)=[Y{6}(1,3),Y{1}(3,1),Y{3}(1,1)];

C(1,:,2)=[Y{3}(1,3),Y{1}(3,3),Y{4}(1,1)];

C(1,:,3)=[Y{4}(1,3),Y{1}(1,3),Y{5}(1,1)];

C(1,:,4)=[Y{5}(1,3),Y{1}(1,1),Y{6}(1,1)];

下四C below

C(1,:,5)=[Y{6}(3,3),Y{3}(3,1),Y{2}(1,1)];

C(1,:,6)=[Y{3}(3,3),Y{4}(3,1),Y{2}(1,3)];

C(1,:,7)=[Y{4}(3,3),Y{5}(3,1),Y{2}(3,3)];

C(1,:,8)=[Y{5}(3,3),Y{6}(3,1),Y{2}(3,1)];

其次是对操作的定义，因为魔方有六个面，共计12种旋转方式【顺时针和逆时针】，所以共计有12个操作文件：U，U1，R，R1，F，1，B，B1，L，L1，D，D1

function []=F( )

%前面顺时针

global Y

global S

A=[];

Y{3}=fliplr((Y{3})');

A=Y{1}(3,:);

Y{1}(3,:)=flipud(Y{6}(:,3));

Y{6}(:,3)=Y{2}(1,:);

Y{2}(1,:)=flipud(Y{4}(:,1));

Y{4}(:,1)=A;

S=[S,'F,'];

% flipud(a);% 行变换

% fliplr(a);% 列变换

end

而在模拟复原的过程中，通过分类讨论的思想，实现对所有位置的复原。比如，拼第一层的十字时，应该复原到九号位的棱块目前可能的位置有12种，其对应的朝向又用两种，故仅有24种情况需要讨论。接下来的底层角块复原时，一个角块所处的位置只有八种情况，朝向有三种，仅需要讨论3\*8=24种情况即可。随着程序的递进，所需的讨论越来越少，执行的循环越来越多，后续的执行思维难度降低，编程要求提高。

global Y

global S

做一个底层十字的第一个棱块

t=finder(Y{3}(2,2),Y{2}(2,2));

%查找位置

if t==1

F;F;

end

给出对应的操作

假如棱块的朝向不正确的话，我们通过后面的if语句进行判断，继而进行输出。

if Y{3}(2,2)~=Y{3}(3,2)

%朝向不正确

F;F;L;R1;U1;R;U1;F;R;R;L;L;

end

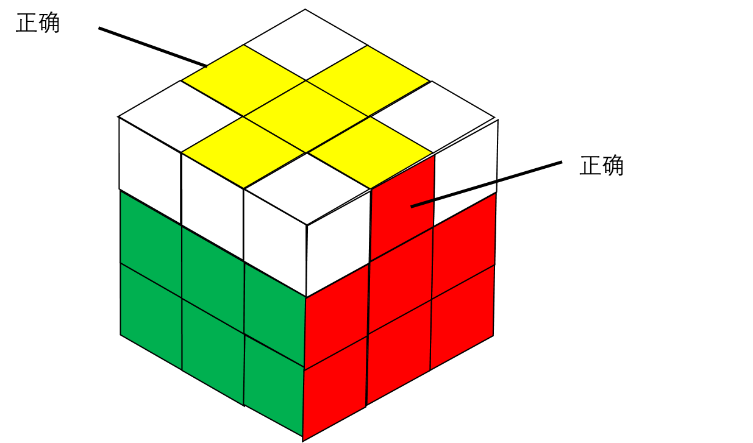
但是在这种模拟过程中，会出现一些无用的步骤，比如说一些棱块本身是可以通过更简单的操作来进行复原，可是在固定的solution之下，是很有可能输出一些重复的步骤的，比如U,U1。

以上

在复原到第三层的时候，我们要注意的是此时的分类讨论只需要分三大类共计八小类分析。

%以上，我们就完成了最顶层的十字复原，下一步是使得棱块的位置也正确

%以下三种是对位复原



R;U;R1;U;R;U;U;R1;

if (Y{3}(2,2)==Y{4}(1,2))&&(Y{5}(2,2)==Y{6}(1,2))&&(Y{4}(2,2)~=Y{5}(1,2))&&(Y{6}(2,2)~=Y{3}(1,2))

U;

R;

U;

R1;

U;

R;

U;

U;

R1;

end

if (Y{3}(2,2)==Y{5}(1,2))&&(Y{5}(2,2)==Y{3}(1,2))&&(Y{4}(2,2)~=Y{6}(1,2))&&(Y{6}(2,2)~=Y{4}(1,2))

U;

U;

R;

U;

R1;

U;

R;

U;

U;

R1;

end

%以上为对位复原

%以下为邻位复原

if (Y{3}(2,2)==Y{3}(1,2))&&(Y{4}(2,2)==Y{4}(1,2))&&(Y{5}(2,2)~=Y{5}(1,2))&&(Y{6}(2,2)~=Y{6}(1,2))

B;

U;

B1;

U;

B;

U;

U;

B1;

end

%邻位复原情况2

if (Y{3}(2,2)==Y{4}(1,2))&&(Y{4}(2,2)==Y{5}(1,2))&&(Y{5}(2,2)~=Y{6}(1,2))&&(Y{6}(2,2)~=Y{3}(1,2))

U;

B;

U;

B1;

U;

B;

U;

U;

B1;

end

%邻位复原情况三

if (Y{3}(2,2)==Y{5}(1,2))&&(Y{4}(2,2)==Y{6}(1,2))&&(Y{5}(2,2)~=Y{3}(1,2))&&(Y{6}(2,2)~=Y{4}(1,2))

U;

U;

B;

U;

B1;

U;

B;

U;

U;

B1;

end

%邻位复原情况四

if (Y{3}(2,2)==Y{6}(1,2))&&(Y{4}(2,2)==Y{3}(1,2))&&(Y{5}(2,2)~=Y{4}(1,2))&&(Y{6}(2,2)~=Y{5}(1,2))

U1;

B;

U;

B1;

U;

B;

U;

U;

B1;

end

%完全混乱的复原不在此处列举

在做第三层的循环是，比较关键的是终止的条件，如果终止条件不能合理设置的话，可能会出现不合理的输出。比如一次的旋转可以解决的问题给出的solution会执行很多步。

1. 视角的转换问题：在实现算法的过程中，Cross.m 文件是讨论了一个棱块的24个可能性的文件，但是如果固定视角的话，复原一个底层十字需要讨论96次。

我们的解决办法是：通过使用自己编写rotate函数来执行一个转动整个魔方的过程，即将整个面的颜色进行调换，实现模拟人的调整视角的操作。

function [ ] = rotate( )

%把整个魔方向左转90度 执行对每个矩阵的代换

% flipud(a);% 行变换

% fliplr(a);% 列变换

global Y

global E

global C

temp=[];

Y{1}=fliplr((Y{1})');

Y{2}=flipud((Y{2})');

temp=Y{3};

Y{3}=Y{4};

Y{4}=Y{5};

Y{5}=Y{6};

Y{6}=temp;

end

这样整个魔方的复原就是可以通过重复调用Cross 函数来完成。

1. 出语句的调整。因为在可视化的过程中，输出的语句要固定视角，所以要对输出的语句转换为固定视角下的执行语句。因此通过编写change函数来完成不同视角的操作转换为同一视角下的操作。

在这个里面一定要注意每一次rotate之后，其实视角是在变化的。所以记录步骤的global S里的S要进行change，而且旋转了三次视角就要change函数执行三次。

S=[];

%下面完成底层十字

%1

Cross;

sol=[sol,S];

S=[];

%2

rotate;

%第一次

Cross;

change;

sol=[sol,S];

S=[];

%3

rotate;

%第二次

Cross;

change;

change;

sol=[sol,S];

S=[];

%4

rotate;

%第三次

Cross;

change;

change;

change;

sol=[sol,S];

S=[];

1. 输出结果可以保证准确率，即任意一个通过正规操作而打乱的三阶魔方，都可以复原。

# **四、解决方案 Solution**

首先，solution的给出思路是模拟人脑的解决方式来完成复原，但是人脑模拟出来的算法过于繁琐，通常都要150步到200步。基于群论知识的Thistlethwaite算法可以在45步以内完成魔方复原，但是限于数学水平，不能完全理解和使用该算法，故不得不放弃这种思路。如果数学水平较高的编程人员可以使用该算法来完成，输出的语句会大大减少。

第二，因为编程过程中是使用分类讨论的方法给出solution的，但是这样的话可能会产生一些无用的操作，比如R,R1这样的步骤，因此我们通过使用编写的adjustsolution函数来完成对输出的solution中不合理的地方的调整。

例如：

temp=[];

t=length(s);

i=0;

while i<=(t-5)

temp=[s(1+i),s(2+i),s(3+i),s(4+i),s(5+i)];

if temp=='R,R1,'

for m=1:5

s(1+i)=[];

end

end

i=i+1;

t=length(s);

end

第三，模拟人脑的操作固然可行，根据MATLAB社区里的成果，假如数学知识足够的话，我们认为可以通过数学的方法来判断选取哪一步来进行操作，而不是通过分类讨论的思想来实现整个代码。如果使用数学方法，那么可以把solution这部分的的代码量从我们小组的1000行缩短到300行。因此我们使用的算法思想受限于知识水平，不够精巧。

第四，我们不能判断一个魔方是否真的可以完全复原。因为如果使用者在填充颜色的过程中填充的魔方是不可复原的，我们的算法就会进入死循环。

# **五、参考文献 Reference**

《七步玩转三阶魔方还原公式及非公式步骤图解（新手适用超详细）》<http://www.360doc.com/content/13/0528/18/3383772_288826902.shtml> 2018年4月27日访问

# **六、附录**

分工说明：

李子天（16332092）：空间坐标到平面坐标的转化、方块遮挡效果的实现、计时器实现动画

吕笳赈（16332124）：魔方复原算法solution.m及其内部调用的复原函数

马梓航（16332125）：十二组TimeStopX代码的编写、PaintColor.m界面的编写、rotate.m、 change.m、adjustsolution.m的编写、canvas.m界面的编写

宋誉卓（16332144）：十二组TimeStartX代码的编写、PaintColor.m界面的编写、rotate.m、 change.m、adjustsolution.m的编写、ParentWindow.m界面的编写

（由于文件量过大，难以具体列举每个人所负责的文件名称）

1. 《七步玩转三阶魔方还原公式及非公式步骤图解（新手适用超详细）》<http://www.360doc.com/content/13/0528/18/3383772_288826902.shtml> 2018年4月27日访问 [↑](#footnote-ref-1)