《Matlab 基础及其应用》大作业

实验报告

实验名称：魔方的识别、建模与解算

学号及姓名： 18373686 孙保成

17375205 张宗晔

18182671 韩雨飞

18373528 杨凌华

18373191 于源龙

2020 年 12 月 29 日

目录

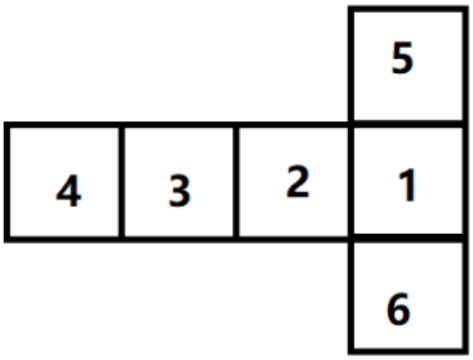
一、实验介绍

1.实验目的

通过 Matlab 对三阶魔方进行识别、建模与求解。

2.实验问题分析

首先要对魔方进行拍照传入，传入的魔方各面编号应如下图所示。



GUI 界面提供图像的目录，可以选择不同编号的图片路径。将上述图片按照顺序传入图像切割模块，完成图像切割之后输出顺序不变的图像。对切割后的图像进行颜色识别，给出 3*3*6 的颜色编号信息和每个颜色编号的 RGB 信息。将识别完成的 3*3*6 的颜色信息进行展示，并提示用户对识别结果进行修改和确认。接着对魔方进行求解，对魔方进行三维建模并将求解过程通过动画进行展现。

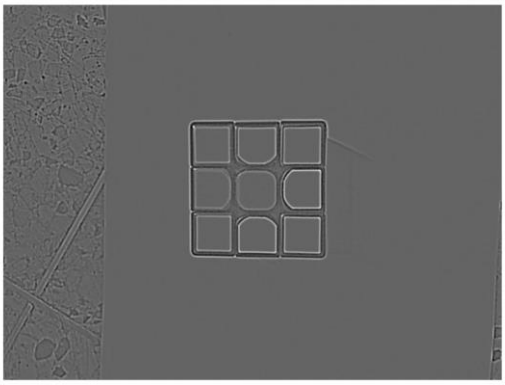
二、实验实现过程

1.裁剪魔方正面

1.1 问题分析

剪裁魔方正面分为五部分，分别是阴影去除，形态学变换，边界提取和投影变换。原图如下。

1.2 阴影去除



可以看到测试图片中阴影非常的深，所以我们使用了同态变换进行了阴影去除。

同态滤波是把频率过滤和灰度变换结合起来的一种图像处理方法，它依靠图像的照度/ 反射率模型作为频域处理的基础，利用压缩亮度范围和增强对比度来改善图像的质量。使用这种方法可以使图像处理符合人眼对于亮度响应的非线性特性，避免了直接对图像进行傅立叶变换处理的失真。同态滤波的基本原理是：将像元灰度值看作是照度和反射率两个组份的产物。由于照度相对变化很小，可以看作是图像的低频成份，而反射率则是高频成份。通过分别处理照度和反射率对像元灰度值的影响，达到揭示阴影区细节特征的目的。

同态滤波处理的基本流程如下：

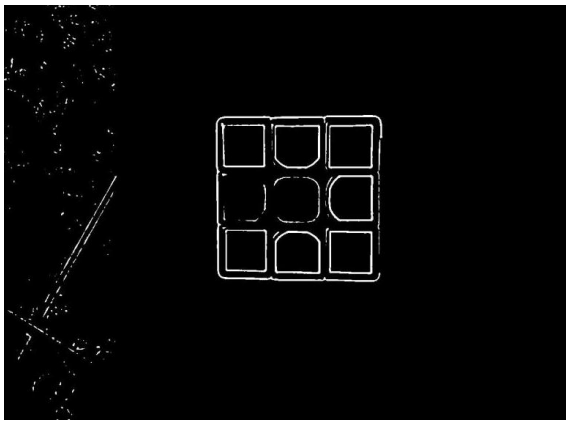
S(x,y)---->Log---->DFT---->频域滤波---->IDFT---->Exp---->T(x,y)

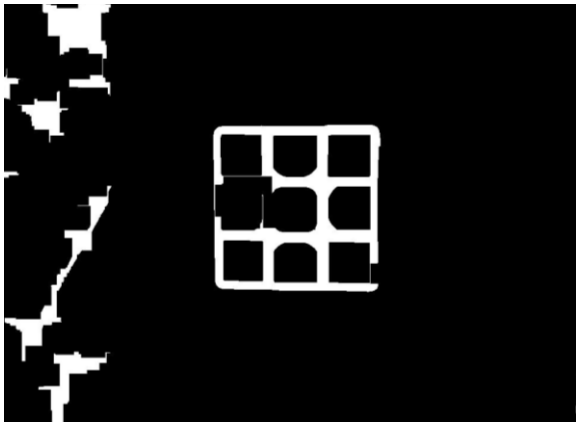
其中 S(x,y)表示原始图像；T(x,y)表示处理后的图像；Log 代表对数运算；DFT 代表傅立叶变换（实际操作中运用快速傅立叶变换 FFT)；IDFT 代表傅立叶逆变换（实际操作中运用快速傅立叶逆变换 IFFT)；Exp 代表指数运算。

```
img = rgb2gray(pic);
filter1 = fspecial('gaussian', [36 36], 6);
filter2 = fspecial('gaussian', [36 36], 12);
high_pass = filter1 - filter2;
log_img = log(1.0 + double(img));
high_log_part = imfilter(log_img, high_pass, 'symmetric', 'conv');
high_part = exp(high_log_part) - 1.0;
```

这里我们没有使用 FFT 和 IFFT 而是使用了简单的高斯滤波替代。

1.3 形态学变换





将同态滤波结果直接二值化会发现魔方部分轮廓不明显，且图片中有较大噪音。

我们使用形态学变换进行处理。其基本元素为：

腐蚀：对白色部分进行减少。

膨胀：对白色部分进行膨胀。

他们可以组合成：

开操作：先腐蚀、后膨胀处理，消除小亮点。

闭操作：先膨胀、后腐蚀处理，消除小黑点。

我们依次进行

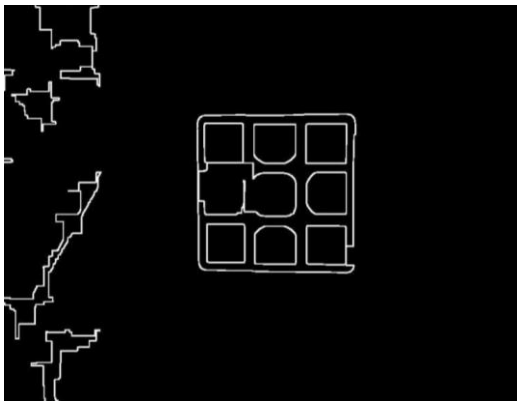
（1）膨胀，使得轮廓变清晰

（2）去除小连通块，去掉部分噪音

（3）腐蚀，使得轮廓大小复原，甚至更小一些

（4）再次去除小连通块，去除薄弱连接的小块

1.4 边界提取



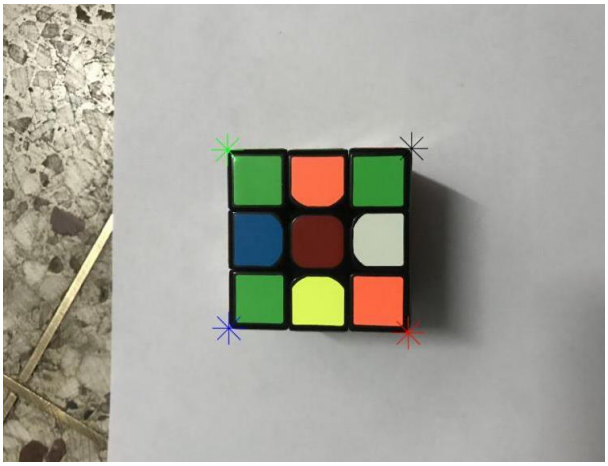
Canny 边缘检测算子是澳洲计算机科学家约翰·坎尼于 1986 年开发出来的一个多级边缘检测算法。Canny 算法包含许多可以调整的参数，它们将影响到算法的计算时间与实效。

高斯滤波器的大小：第一步所用的平滑滤波器将会直接影响 Canny 算法的结果。较小的滤波器产生的模糊效果也较少，这样就可以检测较小、变化明显的细线。较大的滤波器产生的模糊效果也较多，将较大的一块图像区域涂成一个特定点的颜色值。这样带来的结果就是对于检测较大、平滑的边缘更加有用，例如彩虹的边缘。

阈值：使用两个阈值比使用一个阈值更加灵活，但是它还是有阈值存在的共性问题。设置的阈值过高，可能会漏掉重要信息；阈值过低，将会把枝节信息看得很重要。很难给出一个适用于所有图像的通用阈值。目前还没有一个经过验证的实现方法。

使用 canny 变换提取轮廓以便进一步的识别，之后进行了膨胀变换，以连接不慎被分离的轮廓。

1.5 角点提取



先使用 regionprops 提取各个连通块属性。

再选择其中比例大小合适的块，选择其中满足条件的最大的块。

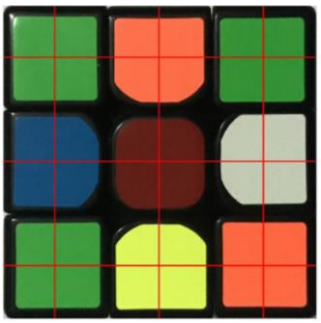
```
if sz(1) > 1.5 * sz(2) || sz(2) > 1.5 * sz(1)
    continue
end

if area < total_area * 0.01 || area > total_area * 0.5
    continue
end
```

最后使用四条直线从左上、右上、左下、右下切获得的区域，得到四个交点。相比于只用上下左右四个方向的直线，这种做法更加严谨。

```
[a,idx_1]=max(hull(:, 1) + hull(:, 2));
[a,idx_2]=max(hull(:, 2) - hull(:, 1));
[a,idx_3]=min(hull(:, 1) + hull(:, 2));
[a,idx_4]=min(hull(:, 2) - hull(:, 1));
p1 = hull(idx_1, :);
p2 = hull(idx_2, :);
p3 = hull(idx_3, :);
p4 = hull(idx_4, :);
```

1.6 投影变换



最后使用投影变换将切得的区域转变为规则的正方形。

```
pos = [0 0;1000 0;0 1000; 1000 1000];
tform = fitgeotrans([p3;p4;p2;p1], pos, 'projective');
J = imwarp(pic, tform, 'OutputView', imref2d([1000, 1000]));
```

这里与很多组同学不同我们使用了 fitgeotrans 函数。查阅 matlab 文档可知这个函数的作用是“Fit geometric transformation to control point pairs”对控制点对组进行几何变换拟合。所以只用认真查标准库就不用自己实现 transform 矩阵了。

1.7 方法的不足

目前该方法需要用户拍摄魔方时只拍到其一个面，而没有测试过另一个面有不慎拍到时结果是否稳定。虽然魔方是立体结构有近大远小特性，想只拍到一个面并不难，但是这种设定对于用户而言并不友好。

2.颜色识别

2.1 问题分析

颜色处理部分需要解决的问题是：如何从已经裁剪好的 RGB 图片中获取魔方的颜色信息，组织为下一个模块可以识别的数据结构。

魔方色块的定位：由于之前图片裁剪模块已经得到规则的图像，直接对图像取等分点，获取每一块中心像素的颜色即可。为了减小随机噪声带来的影响，可考虑取周围 3*3 的颜色平均值。

RGB 色彩空间与颜色相关的有三个维度：红绿蓝。而 HSV 色彩空间，除了黑、白以外，只有一个维度 Hue 决定颜色，其他维度决定了色彩的饱和度和明度，故 HSV 空间常用于颜色识别与分类。

首先考虑固定阈值算法，但阈值算法有如下缺点：

（1）对魔方的颜色进行提取后，发现魔方的部分颜色的 Hue 值相近，只有颜色深浅的区别，不易设置阈值。如下图 1 展示的深红色和橙色，其 Hue 值均在 0 左右，而区别仅在 Value 维度。

（2）不同种类的魔方颜色会有细微差异，一旦更换魔方，本系统的阈值需要重新调整。

（3）相同魔方在不同光照条件下得到的图片也会产生不一致的颜色效果，同样需要重新调整阈值

图 1

考虑到阈值算法有诸多缺点，更换其他思路。有如下观察：
（1）可能不同环境下，光照条件会有不同。但能基本保证每次采集 6 个面的图片的时候，光照条件基本一致。
（2）在固定光照条件下，魔方六个面中心块的颜色即为该光照条件下魔方六种颜色确定的标准。

故聚类算法或许能够更好地解决该问题，尝试对魔方提取到的所有颜色在 HSV 三维空间进行聚类，得到如图 2 结果，发现每种颜色较好地聚集在一个区域，最终采用聚类算法。

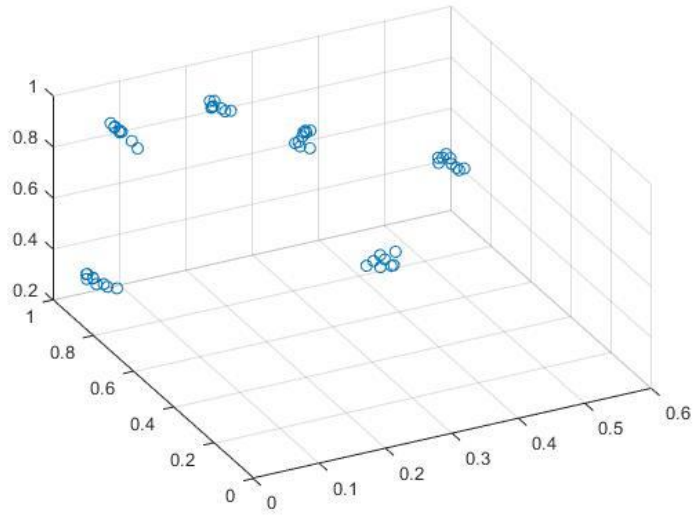


图 2

2.2 具体实现过程

首先使用 rgb2hsv 将 RGB 图片变为 HSV 色彩空间图片，然后使用 size 函数获取图片的大小。由于传入的图片已经进行了裁剪，可以保证得到如图 1 所示的正面图片，只需对图片的大小进行三等分即可。由于图片的质量较好，仅采集了中心点一个点的像素 HSV 值，如果图片噪点较多则可考虑取中心点周围的颜色的平均值。获取的像素值放入一个 3*3*6 的矩阵中。

```
for i = 1 : rubik_scale
    for j = 1 : rubik_scale
        % 提取中心点色彩放入矩阵
        hue_mat(i,j,side) = hue_channel(center_ws(i,j),center_hs(i,j));
        sat_mat(i,j,side) = sat_channel(center_ws(i,j),center_hs(i,j));
        val_mat(i,j,side) = val_channel(center_ws(i,j),center_hs(i,j));
        if (plot_hsv)
            text(center_ws(i,j)-30,center_hs(i,j)-30,...
                ['H:',num2str(hue_mat(i,j))],'FontSize', 16);
            text(center_ws(i,j)-30,center_hs(i,j)+30,...
                ['S:',num2str(sat_mat(i,j))],'FontSize', 16);
            text(center_ws(i,j)-30,center_hs(i,j)+60,...
                ['V:',num2str(val_mat(i,j))],'FontSize', 16);
        end
    end
end
```

完成每个色块颜色获取后，在调试阶段输出了三维散点图，为使用 scatter3 函数，使用 reshape 函数将原有的 3*3*6 矩阵压平为只有一行的向量。在确定了聚类算法的可行性后，使用 Matlab 提供的 K-Means 聚类算法函数 kmeans 进行聚类。在测试的过程中发现聚类不太稳定，有时候会得到错误的结果。

在阅读了 K-Means 算法的原理后，发现起始中心点如果不指定则会随机选择若干个点（取决于希望分类的数量）作为起始点，进行迭代，只要达到接受条件就会被接受。故可能会出现如下情况：选择了相对较偏离中心的点，然后迭代过程中将不属于该聚类的点也纳入范围。为解决这个问题也很简单，只需要指定中心点即可。中心点的选择，由于魔方的特殊性质，中心块的颜色一定不同，选择了魔方每个面的中心块作为中心点。此时得到的聚类结果稳定且正确，达到了预定目的。

```
% 聚类观察
if (plot_scatter)
    figure(7);
    scatter3(reshape(hue_mat, 1, rubik_scale * rubik_scale * 6), ...
        reshape(sat_mat, 1, rubik_scale * rubik_scale * 6), ...
        reshape(val_mat, 1, rubik_scale * rubik_scale * 6));
end

% K-means聚类
scatter_mat = [reshape(hue_mat, 1, rubik_scale * rubik_scale * 6);...
    reshape(sat_mat, 1, rubik_scale * rubik_scale * 6);...
    reshape(val_mat, 1, rubik_scale * rubik_scale * 6)]';

% 提取中心方块颜色，并作为K-Means的起始点
C_pri = [reshape(hue_mat(2,2,:), 1, 6);...
         reshape(sat_mat(2,2,:), 1, 6);...
         reshape(val_mat(2,2,:), 1, 6)]';


[idx, C] = kmeans(scatter_mat, 6, 'Start', C_pri);
```

3. 识别结果确认

3.1 具体操作流程

在识别确认环节要将上一步颜色识别完成的 3*3*6 的 cell 格式的颜色信息通过魔方的展开图进行展示，在此我们选择的是以按钮的形式进行展示。每一个色块对应着一个按钮，每个按钮对应着其相应的颜色并且标记着其所对应的颜色编号，这样可以让用户更加直观的对识别结果进行判断以及修改。点击识别错误的色块，会弹出一个由六个颜色的按钮组成的窗口，按钮上同样标记着颜色和颜色编号。点击要修改成的颜色，该窗口自动关闭，魔方展开图中选中的错误色块

的颜色和编号都变换为修改后的，同时存储颜色信息的 cell 数组中该位置的数据也会更新为修改后的。重复上述步骤直至用户对识别结果完成修改。

修改结束点击确认时，系统会自动对用户修改结果进行检查。检查 cell 数组中每种颜色编号是否出现了 9 次，若不满足，则弹出窗口提示用户“结果仍有误”，让用户继续进行修改；若满足条件，则提示用户“结果正确”，并关闭所有窗口，将数据传递给接下来的步骤。

3.2 具体实现过程

通过函数ReviseColor()对6个面的每个色块都生成按钮，并且每个面的所有按钮都会生成一个回调函数。

下面是以第一个面为例的实现代码，callbackFcn_1为第一个面的回调函数，并且把每个色块的所对应的行号和列号作为参数进行传递。

```
for i = 0:2
    for j = 1:3
        RC.face1(3*i+j) = uicontrol('parent',RC.window,...
                          'style','pushbutton',...
                          'string',R_{3-i,j,1},...
                          'position',[x+size*(j-1),y+size*i,size,size],...
                          'visible','on',...
                          'backgroundcolor',RGB(R_{3-i,j,1},:),...
                          'callback',@(~,~)callbackFcn_1(3-i,j));
    end
end
```

在每个面的按钮的回调函数中，还会实现生成一个由六个颜色的按钮组成的窗口，这六个按钮也对应着一个回调函数callbackFcn_7来对原来的颜色进行修改，要修改成为的颜色编号、要修改色块的行号列号面号都作为参数传递，下面是以第一个面为例的代码实现。

```
for i = 1:3
       RC.color(i) = uicontrol('parent',RC.cb,...
                 'style','pushbutton',...
                  'string',i,...
                  'Fontsize',10,...
                 'position',[x+size*i,y,size,size],...
                 'visible','on',...
                 'backgroundcolor',RGB(i,:),...
                 'callback',@(~,~,~,~)callbackFcn_7(i,index1,index2,1));
            x = x+delta;
end
```

下面是颜色修改函数 callbackFcn_7 对颜色进行修改的部分代码。

```
function callbackFcn_7(index,index1,index2,face)
global RC;
global R_;
global RGB;

    button_state = index;
    switch face
        case 1
            set(RC.face1(3*(3-index1)+index2),'backgroundcolor',RGB(button_state,:));
            set(RC.face1(3*(3-index1)+index2),'string',button_state);
            R_{index1,index2,1} = button_state;
        case 2
```

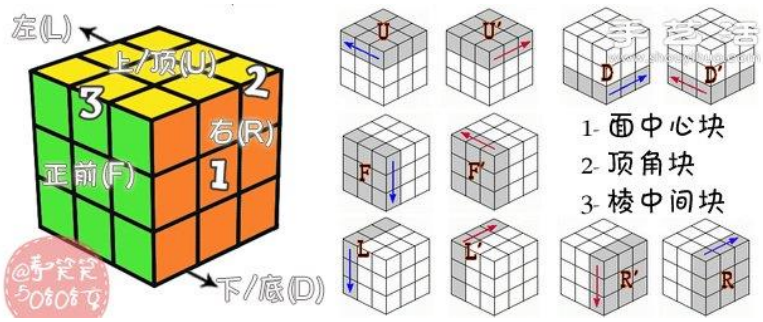
确认修改结束后点击的确认按钮也对应一个检查修改结果的回调函数，对每个颜色编码的出现次数进行计数判断是否出现 9 次。下面是具体实现代码。

```
        for i=1:3
            for j=1:3
                for k=1:6
                     switch(R_{i,j,k})
                         case 1
                             num1 = num1 + 1;
                         case 2
                             num2 = num2 + 1;
                         case 3
                             num3 = num3 + 1;
                         case 4
                             num4 = num4 + 1;
                         case 5
                             num5 = num5 + 1;
                         case 6
                             num6 = num6 + 1;
                         otherwise
                     end
                end
            end
        end
        if num1==9&&num2==9&&num3==9&&num4==9&&num5==9&&num6==9
           flag=1;
```

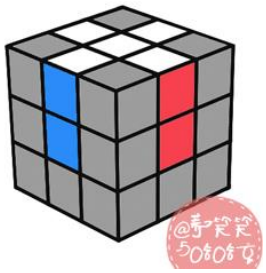
4. 魔方求解算法

4.1 逐层解决方案

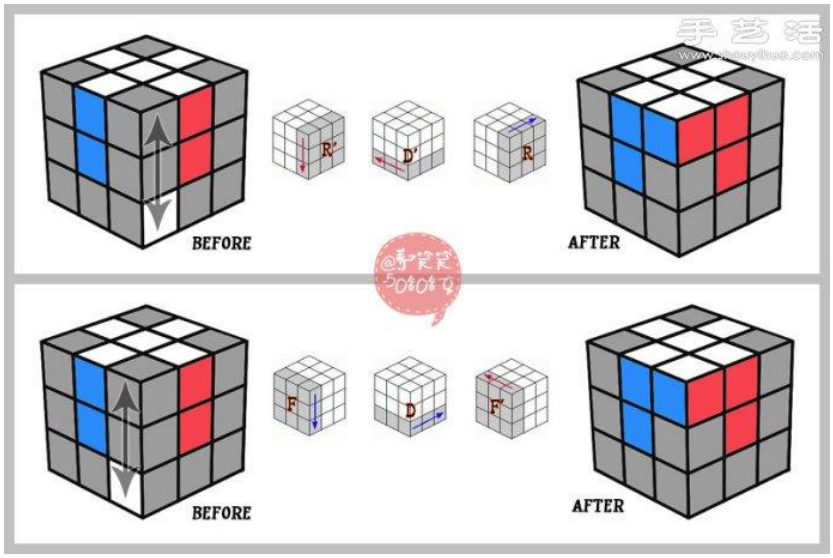
首先定义魔方的 6 个面名称，以及定义魔方的 12 种操作。



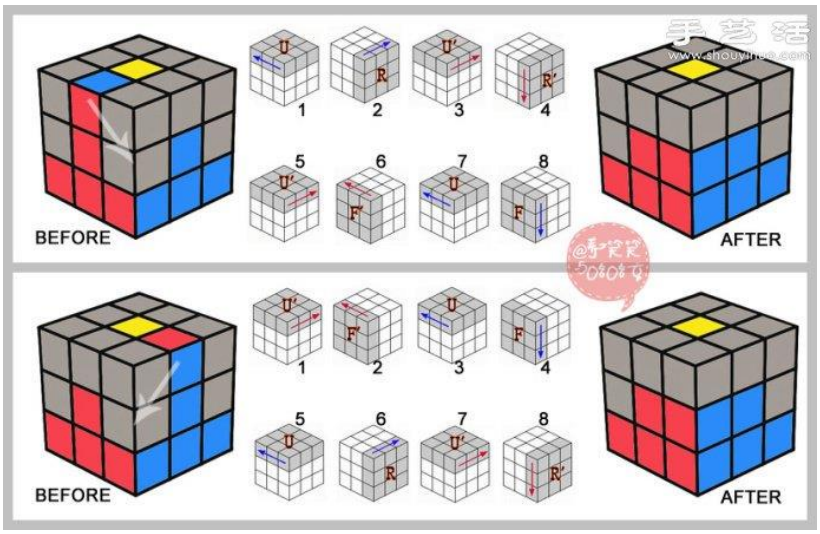
第一步 首面十字



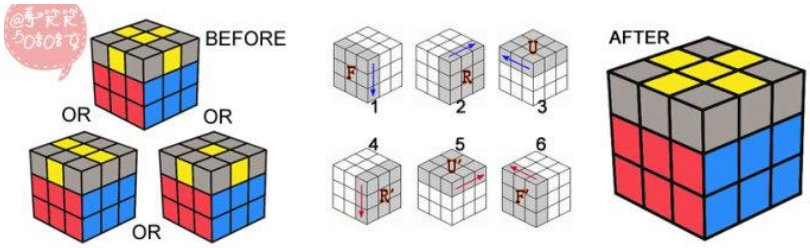
第二步首面顶角归位&完成第一层



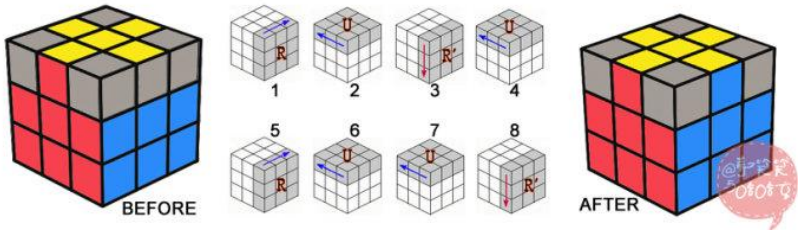
第三步完成第二层



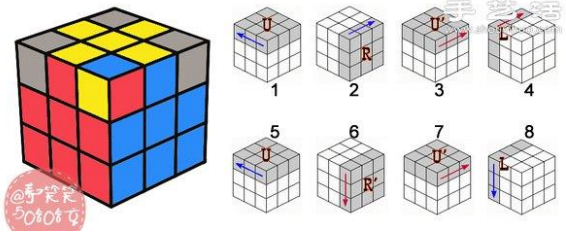
第四步顶面十字



第五步顶层棱中间块归位

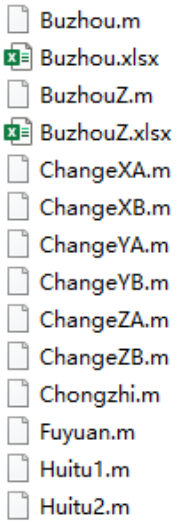


第六步顶层顶角半归位



第七步顶层顶角归位&魔方完成

最后的找不到贴图了。可以很轻易的看出对于三阶魔方，如果采用逐块还原法，只需要预先设计好完成步骤，然后对魔方的各个状态进行遍历就好了。算法具体如下所示。



（完整代码可能并没有上传，因为没用上）

4.2 T45 方法

我们组采用的方法 t45 方法。首先来介绍一下魔方的上帝之数是怎么来的：对于一个 3×3×3 的魔方总共有这些种情况。

$$\begin{aligned} & 8! \times 3^8 \times 12! \times 2^{12} \times \frac{1}{3} \times \frac{1}{2} \times \frac{1}{2} \\ & = 40320 \times 6561 \times 479001600 \times 4096 \div 12 \\ & = 43252003274489856000 \end{aligned}$$

最后得到总数约为 4.3*10^19 种。

然后通过计算机遍历这 4.3 千亿亿种情况，得出所有情况复原的最少步骤的上限是 20，所以三阶魔方的上帝之数就是 20。

3x3x3 的 Thistlethwaite45 (T45)：此算法的解不超过 45 个，平均值为 31，是一种非常常用的魔方解法。普通解法是通过逐块（piece or block or layer）还原来减少下一步剩余块的状态数，最后所有块还原。而 Thistlethwaite45 方法（TM）则与此有本质的不同。TM 的思想是逐步降解魔方所处的群到更小的子群，最后到单位子群，也即还原状态。所以在还原的每一步实体魔方看起来还是乱的，但实际上魔方的状态数是随着所处的群的减小而规则的减小的。

加个形象点的解释帮助理解。如果魔方通过<U，D，L，R，F，B>六个基本动作打乱，那么它的混乱状态可以达到最大，有 10^20 次方种。但假如我只用<U2,D2,L2,R2,F2,B2>来打乱魔方，显然魔方没有前一种情况乱，只有 60 万种。极端一点的，我只用 R 转动打乱魔方，那么魔方就只有四种混乱状态。

上面这个逐步降解到子群的过程，就是把魔方由最大打乱状态一步一步的变到更小的打乱状态，最后达到复原状态。

矩阵 prunes 中储存着 t45 算法的 4 个群类。

Moves 中储存着这四个群类的操作。

```
P = load('Prunes');
P1 = P.P1;
P2 = P.P2;
P3 = P.P3;
P4 = P.P4;
```

G0=<U,D,L,R,F,B>　　4.33*10^19
Phase1：　减少 2048（2^11）
对好所有楞块方向。

```
%PHASE 1: CURE EDGES
moves = {'L'  ,'R'  ,'F'  ,'B'  ,'U'  ,'D';...
         'L2' ,'R2' ,'F2' ,'B2' ,'U2' ,'D2';...
         'L''','R''','F''','B''','U''','D'''};

n = State2Ind(E(2,:))+1;
N = P1(n);

while N>0
    for i=1:18
        E2 = TwistEdges(E,moves{i});
        n = State2Ind(E2(2,:))+1;
        M = P1(n);
        if M<N
            N = M;
            E = E2;
            C = TwistCorners(C,moves{i});
            sol = [sol moves(i)];
            break
        end
    end
end
```

G1=<U,D,L,R,F2,B2>　2.11*10^16
Phase2：　减少 1082565（12!/(8!4!)*3^7）
将中间层楞块放到中间层，并调整好角块朝向

Solve45.m | rubsolve.m | TwistCorners.m | TwistEdges.m | State2Ind.m

```
%PHASE 2: MOVE LR-SLICE EDGES TO UD-SLICE + ORIENT CORNERS
moves = {'L'  ,'R'  ,'F'  ,'B'  ,'U2';...
         'L2' ,'R2' ,'F2' ,'B2' ,'D2';...
         'L''','R''','F''','B''','00'};

Clist = P.ClistP2;
Elist = P.ElistP2;

F = E(1,:);
F = double(F>=9);
Cind = State2Ind(C(2,:));
Eind = State2Ind(F);

n = Clist==Cind;
m = Elist==Eind;

N = P2(n,m);

while N>0
    for i=1:14
        C2 = TwistCorners(C,moves{i});
        E2 = TwistEdges(E,moves{i});
        F2 = E2(1,:);
        F2 = double(F2>=9);
        Cind = State2Ind(C2(2,:));
        Eind = State2Ind(F2(1,:));
        n = Clist==Cind;
        m = Elist==Eind;
        M = P2(n,m);
        if M<N
            N = M;
            C = C2;
            E = E2;
            sol = [sol moves(i)];
            break
        end
    end
end
```

G2=<U,D,L2,R2,F2,B2>　　1.95*10^10

Phase3：　减少 29400(8!/(4!4!)^2*2*3)

调整 F/B 面为 RO 色，L/R 面为 BG 色，并调整角块相对位置。

Solve45.m | rubsolve.m | TwistCorners.m | TwistEdges.m | State2Ind.m

```
78
79      %PHASE 3: FIX EDGES IN THEIR SLICE W/ EVEN PERMUTATION + FIX CORNERS IN
80      %         ORBIT W/ EVEN PERMUTATION
81 -    moves = {'L' ,'L''','L2',...
82               'R' ,'R''','R2',...
83               'F2','B2','U2','D2'};
84
85 -    Clist = P.ClistP3;
86 -    Elist = P.ElistP3;
87
88 -    F = ceil(E(1,1:8)/4)-1;
89
90 -    Cind = State2Ind(C(1,:));
91 -    Eind = State2Ind(F,2);
92
93 -    n = find(Clist==Cind);
94 -    m = find(Elist==Eind);
95 -    N = P3(n,m);
96
97 -    while N>0
98 -        for i=1:10
99 -            C2 = TwistCorners(C,moves{i});
100 -           E2 = TwistEdges(E,moves{i});
101 -           F = ceil(E2(1,1:8)/4)-1;
102 -           Cind = State2Ind(C2(1,:));
103 -           Eind = State2Ind(F,2);
104 -           n = Clist==Cind;
105 -           m = Elist==Eind;
106 -           M = P3(n,m);
107 -           if M<N
108 -               N = M;
109 -               C = C2;
110 -               E = E2;
111 -               sol = [sol moves(i)];
112 -               break
113 -           end
114 -       end
115 -   end
```

G3=<U2,D2,L2,R2,F2,B2>　　6.63*10^5

Phase4：　减少 663552(4!^5/12)

还原所有角块和楞块

```
%PHASE 4: SOLVE THE CUBE
moves = {'L2','R2','F2','B2','U2','D2'};

Clist = P.ClistP4;
Elist = P.ElistP4;

Cind = State2Ind(C(1,:));
Eind = State2Ind(E(1,:));

n = Clist==Cind;
m = Elist==Eind;
N = P4(n,m);

while N>0
    for i=1:6
        C2 = TwistCorners(C,moves{i});
        E2 = TwistEdges(E,moves{i});
        Cind = State2Ind(C2(1,:));
        Eind = State2Ind(E2(1,:));
        n = Clist==Cind;
        m = Elist==Eind;
        M = P4(n,m);
        if M<N
            N = M;
            C = C2;
            E = E2;
            sol = [sol moves(i)];
            break
        end
    end
end

sol = rubopt(sol);
```

G4

TwistCorners:确定角块

Solve45.m | rubsolve.m | TwistCorners.m | TwistEdges.m | State2Ind.m

```
function C = TwistCorners(C,move)

if iscell(move)
    for i=1:numel(move)
        C = TwistCorners(C,move{i});
    end
    return
end

if numel(move)==2 && move(2)=='0'
    return
end
if move(1)~='r' && numel(move)==2
    if move(2)=='1'
        C = TwistCorners(C,move(1));
    elseif move(2)=='2'
        C = TwistCorners(C,{move(1),move(1)});
    elseif move(2)=='''' || move(2)=='3'
        C = TwistCorners(C,{move(1),move(1),move(1)});
    end
    return
elseif move(1)=='r'
    switch move(2)
        case 'x'
```

TwistEdges:确定中心块

Solve45.m | rubsolve.m | TwistCorners.m | TwistEdges.m | State2Ind.m

```
function E = TwistEdges(E,move)

if iscell(move)
    for i=1:numel(move)
        E = TwistEdges(E,move{i});
    end
    return
end
if move(1)~='r' && numel(move)==2
    E = TwistEdges(E,{move(1),move(1)});
    if move(2)=='''' || move(2)=='3'
        E = TwistEdges(E,move(1));
    end
    return
elseif move(1)=='r'
    switch move(2)
        case 'x'
            move = 'p';
        case 'y'
            move = 'q';
        case 'z'
            move = 'r';
    end
end
```

Srare2Ind: 将排列或方向转换为唯一索引。

Solve45.m | rubsolve.m | TwistCorners.m | TwistEdges.m | State2Ind.m

```
function X = State2Ind(Y, varargin)
%
% Convert permutation or orientation to unique index
%

n = numel(Y);

if nargin == 1
    if n==8
        v = 3;
    else
        v = 2;
    end
else
    v = varargin{1};
end

if max(Y)==numel(Y)
    %%PERMUTATION
    X = 0;
    for i = 1:n-1
        X = X*(n+1-i);
        for j=i+1:n
            if Y(j) > Y(i)
```

5. 三维建模与动画

5.1 三维模型的建立

将魔方分解成 3×3×3 = 27 个小方块，对每个小方框建模，小方块总共 6 个面，每个面 4 个端点，每个端点对应一个空间坐标（x，y，z），故分别对 X、Y、Z 三个轴建立一个 4×6 的矩阵，矩阵每一个列代表一个面的四个端点在各个坐标轴上的位置，这里是相对于坐标原点(0, 0)，小方块的边长设置为 1。

```
x=[
      -0.5 +0.5 +0.5 -0.5 -0.5 -0.5;
      +0.5 +0.5 -0.5 -0.5 +0.5 +0.5;
      +0.5 +0.5 -0.5 -0.5 +0.5 +0.5;
      -0.5 +0.5 +0.5 -0.5 -0.5 -0.5;
  ];
y=[
      -0.5 -0.5 +0.5 -0.5 -0.5 -0.5;
      -0.5 +0.5 +0.5 +0.5 -0.5 -0.5;
      -0.5 +0.5 +0.5 +0.5 +0.5 +0.5;
      -0.5 -0.5 +0.5 -0.5 +0.5 +0.5;
  ];

z=[
      -0.5 -0.5 -0.5 -0.5 -0.5 +0.5;
      -0.5 -0.5 -0.5 -0.5 -0.5 +0.5;
      +0.5 +0.5 +0.5 +0.5 -0.5 +0.5;
      +0.5 +0.5 +0.5 +0.5 -0.5 +0.5;
  ];
```

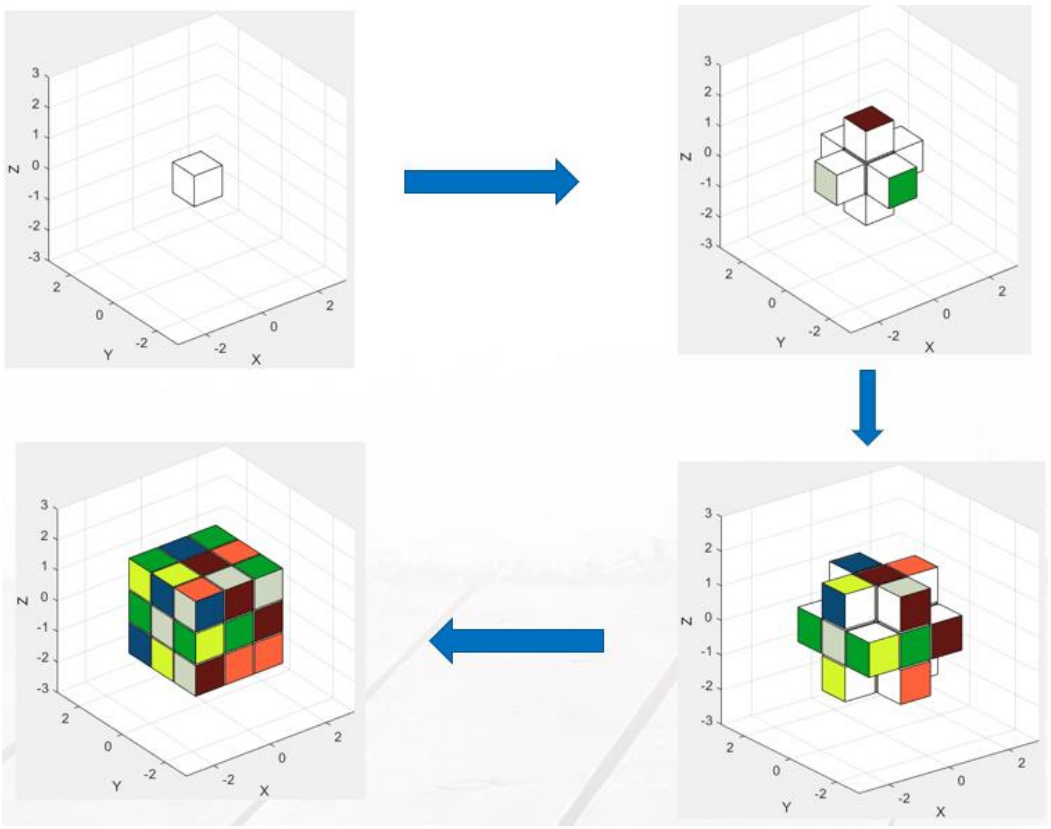
上述创建的单个小方块，是魔方最中心的那个方块，而要得到其他外围的 26 个方块，需要将该基准方块通过在 X、Y、Z 三轴上进行一些偏移以及偏移的组合，就能够得到，此处定义 X1、X2、Y1、Y2、Z1、Z2 如下：

```
x1=x+1.05;  x2=x-1.05;
y1=y+1.05;  y2=y-1.05;
z1=z+1.05;  z2=z-1.05;
```

其中对每两个相邻小方块之间的间隙宽度，我们设定为 0.05，那么 X1、X2 分别就代表在 X 轴上正向和负向偏移一个方块位置，Y 轴、Z 轴同理，再例如通过将（X1，Y1，Z1）组合，得到的就是魔方某个角落处的小方块。

之后通过函数：fill3(X1,Y1,Z1,C1,X2,Y2,Z2,C2,...)来对各个小方块的六个面进行着色。

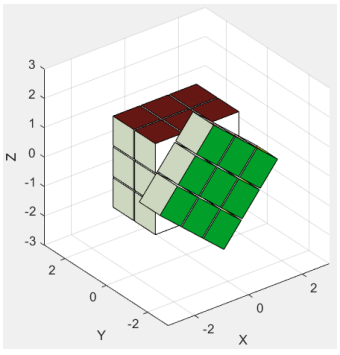
下面是魔方各个小方块的分解与组合示意图：

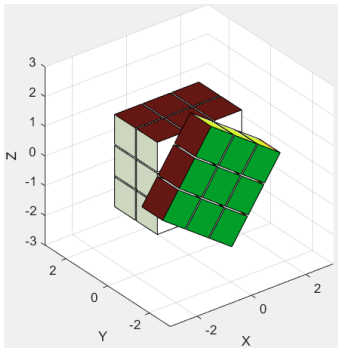


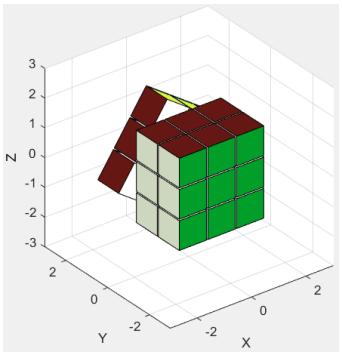
5.2 旋转动画

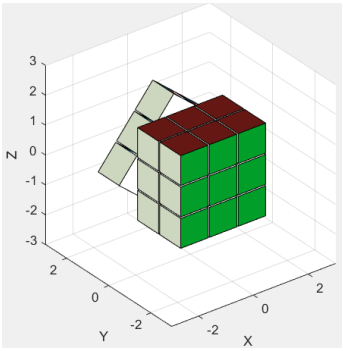
5.2.1 具体实现原理

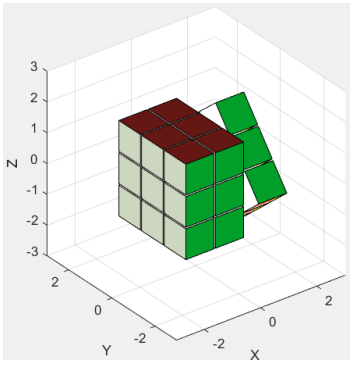
将魔方的旋转分为如下 12 种类型，分别覆盖 6 个面，每个面有逆时针和顺时针两种旋转方向：

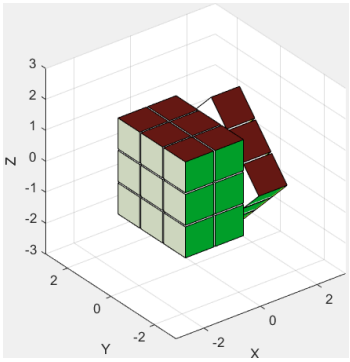
(1) R : 

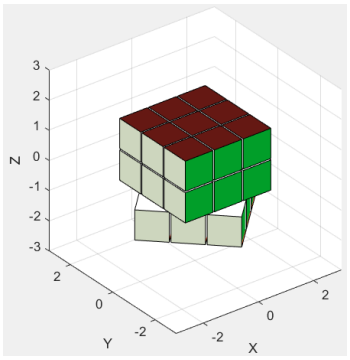
(2) R- : 

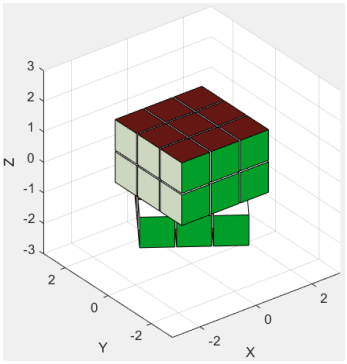
(3) L : 

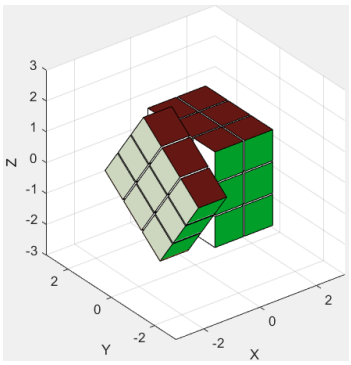
(4) L- : 

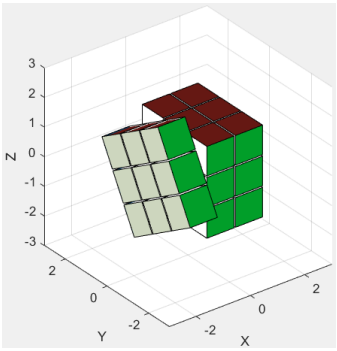
(5) B : 

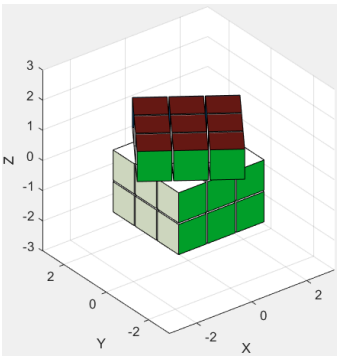
(6) B- : 

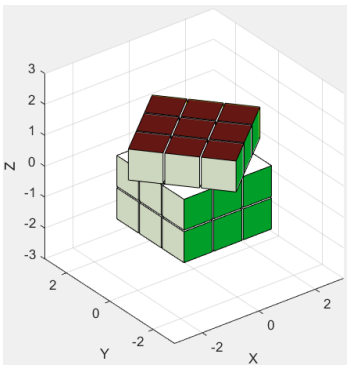
(7) D : 

(8) D- : 

(9) F : 

(10) F- : 

(11) U : 

(12) U- : 

以 R 操作为例阐述旋转动画的具体实现原理：

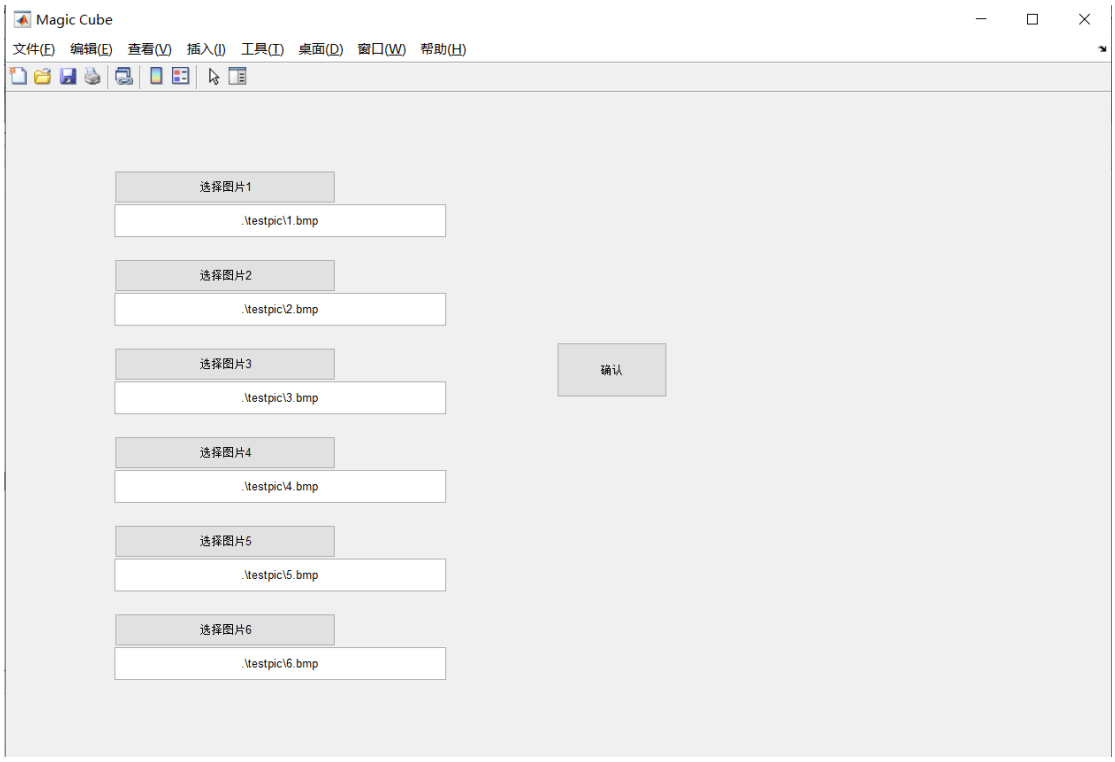
```
function R()
    global R_Cube;
    s=R_Cube;
    vector=[0 1 0];step=5;Point0=[0 0 0];
    for ii=1:18
        for jj=1:3
            for kk=1:3
                rotate(R_Cube{3,jj,kk},vector,step,Point0);
            end
        end
        pause(0.01);
    end
    R_Cube{3,1,1}=s{3,1,3};R_Cube{3,1,2}=s{3,2,3};R_Cube{3,1,3}=s{3,3,3};
    R_Cube{3,2,1}=s{3,1,2};R_Cube{3,2,2}=s{3,2,2};R_Cube{3,2,3}=s{3,3,2};
    R_Cube{3,3,1}=s{3,1,1};R_Cube{3,3,2}=s{3,2,1};R_Cube{3,3,3}=s{3,3,1};
end
```

5.2.2 具体操作流程

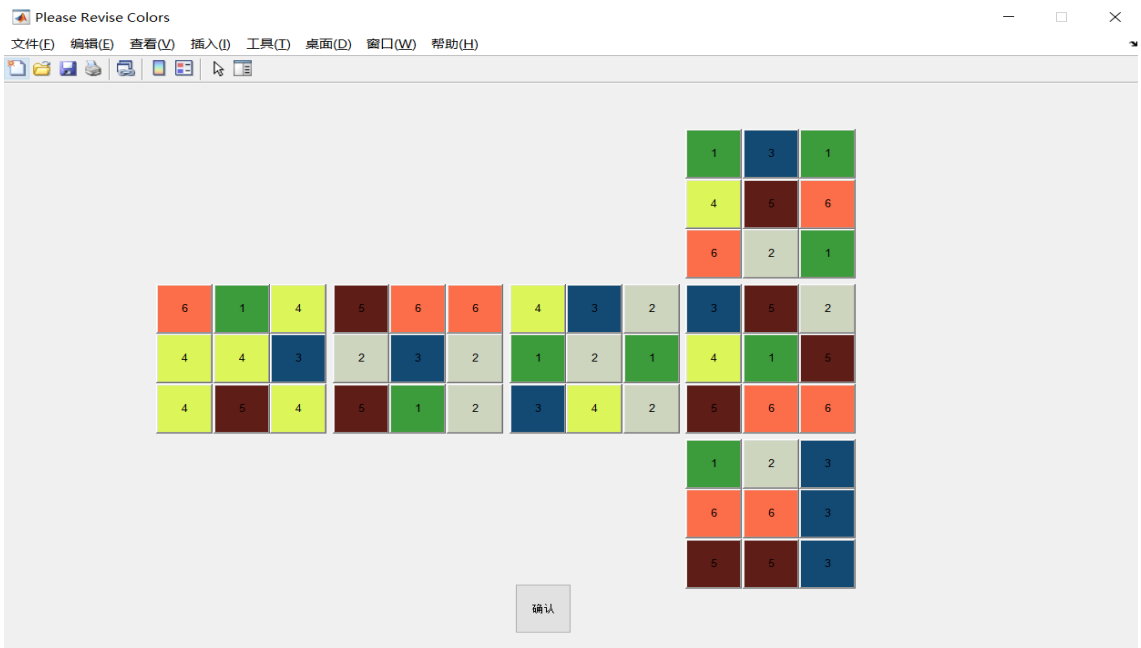
首先建立当前魔方各个面的状态的副本 S 用于暂存，之后以向量 vector 为方向，5° 为步长，以原点 Point0 为向量的起始点，将右面的 9 个小方块，同步分成 18 步来进行旋转 （ 18 * 5° = 90° ） ，每旋转 5° 就 pause(0.01)延迟 0.01 秒，从而产生渐变的效果，最后，将旋转后魔方右面新的状态以原状态 s 为起始，更新到新的状态。

三、实验结果

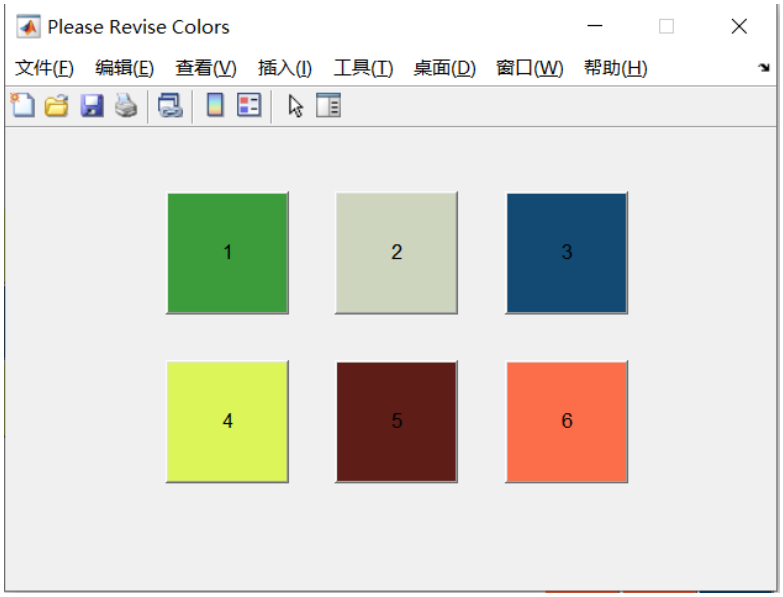
运行程序，会弹出一个选择图片的 GUI 界面：



点击确认，经过图片切割和颜色识别之后，弹出识别结果确认的 GUI 界面：

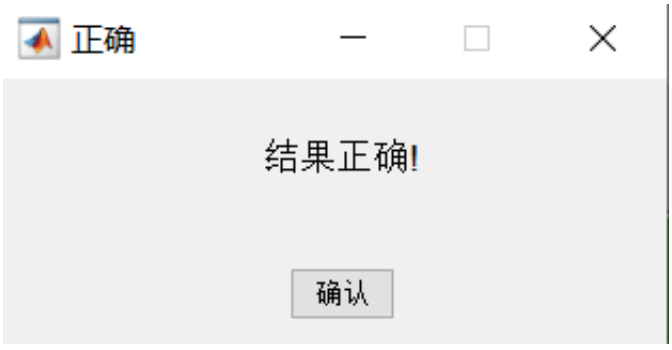


点击错误色块，弹出颜色选择的 GUI 界面：

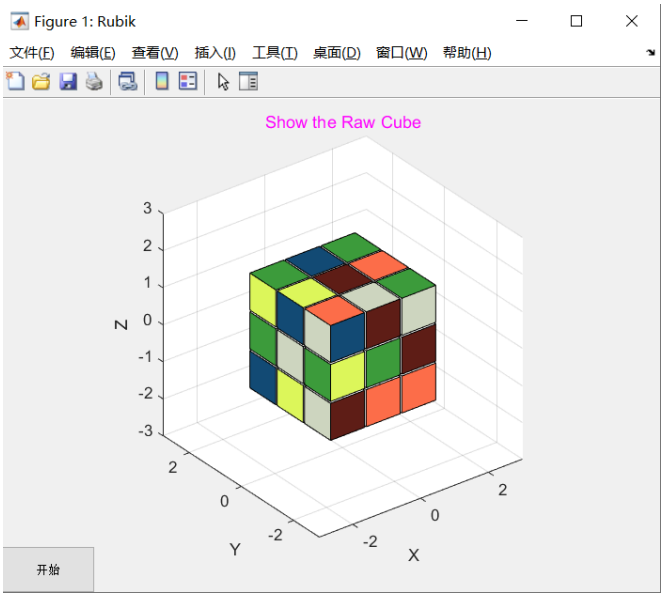


点击要修改成的颜色进行修改，原展开图进行更新。在修改结束后点击确认，若仍有错会弹出报错界面：

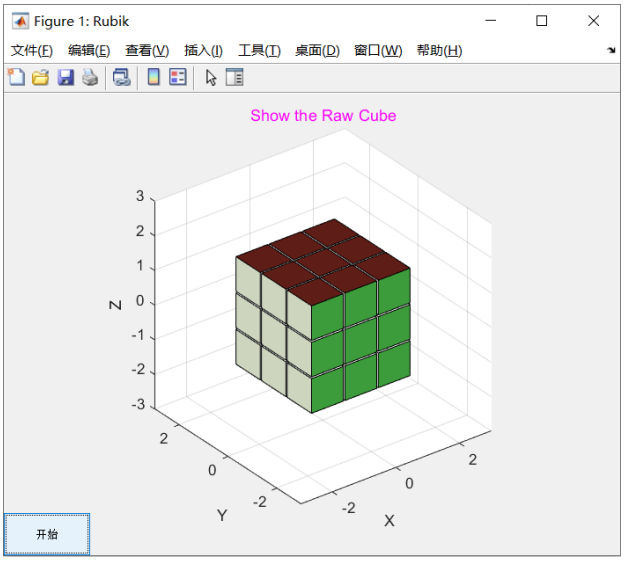
若修改结果正确会弹出提示窗口：



确认完成后会弹出一个魔方三维展示界面：



点击开始，魔方旋转进行还原，最终得到六面完全还原的三阶魔方：



四、小组分工

组员	负责任务	工作量占比
孙保成	裁剪魔方正面部分	20%
张宗晔	1. 图片转换与数据整理 2. 颜色聚类测试与实现	20%
韩雨飞	1、识别结果确认部分 2、实验报告整理汇总	20%
杨凌华	1、魔方的三维建模 2、魔方旋转动画的实现 3、魔方图片选择前端界面搭建 4、适配各个组员写好的程序模块之间的接口，并形成完整可运行的程序	20%
于源龙	魔方求解算法部分	20%