大连理工大学程序设计训练实验报告

EasyCube-基于 Unity 3D 的智能魔方教学空间

EasyCube - Intelligent Cube Teaching Space Based on Unity 3D

学 院（系）：电子信息与电气工程学部

专　　业：计算机科学与技术

团 队 成 员：龙博、薛昕磊、杨悦航

刘啸尘、李英平

联 系 方 式：15940992407

指 导 教 师：齐恒

完 成 日 期：2018 年 7 月 21 日

大连理工大学

Dalian University of Technology

目　　录

引　言

魔方最初由匈牙利布达佩斯建筑学院的 Rubik 教授发明，最开始的目的是为了培养学生的对立体几何的空间理解力。可是令人想不到的是，自从魔方在 1974 年在匈牙利问世之后，魔方就发展迅速，被世人所喜爱。在三阶魔方出来之后，许多魔方爱好者制做出了一系列的衍生产品，与此同时，魔方的作用也不再是最初只是为了培养人的空间想象力了，而是发展到了研究魔方的复原，追求魔方复原的速度，向复合型手、眼、脑协调运用的竞技运动发展。魔方不但是一种益智玩具，一种教学道具，更是一种锻炼脑力的工具，玩魔方并且玩好魔方已经是一种充满智慧的时尚的休闲活动。

然而，慑于魔方表面上的复杂性，很多人一直都未能学会复原魔方；即使成功学会复原魔方，由于魔方高级解法在市面上的教程十分难以理解，绝大多数魔方玩家也都未能领略到魔方真正的魅力。

我们小组成员都十分钟爱魔方，有感于市面上教材的粗糙给魔方初学者带来的困难，决定开发一个智能魔方教学软件，让更多的人快速掌握复原魔方的方法，能够在 30 秒内完成复原魔方，真正享受到魔方运动带来的乐趣。

我们设计的魔方复原系统，目的是利用设计的复合型的巧手结构，使得复原系统具有类似人手一样的功能，结合研究的魔方复原来实现高速复原魔方。实现上述功能，需要面临较多的技术，包括魔方的状态识别、魔方的复原算法的研究、系统复合型巧手的合理设计、快速原型设计方法等。

我们仔细研究了魔方复原算法（CFOP 算法、角先法、TM 算法）、任意两个魔方之间的状态转换算法（传递法、同构法）等经典算法，结合我们实验的具体情况对算法进行了筛选和一定改进。在我们的实验中内核求解魔方的时候使用了类人脑思维的搜索算法——启发式搜索，启发式搜索是一个状态空间搜索策略。在这个搜索策略中，一个具有深度限制的深度优先搜索算法会不断重复地运行，并且同时放宽对于搜索深度的限制，直到找到目标状态。项目最终可以实现的功能有：3D 魔方模拟、魔方求解、魔方教学、进阶训练。此外还可以接上机械手，实现硬件与控制系统之间的信息交换，在硬件基础上最终实现更加完整的功能，利用机械手实现魔方的快速还原，从而体现了物联网思想概念的实际应用。系统通过良好的交互界面，讲解魔方的高级解法，宣传魔方真正的魅力。整个项目是基于启发式搜索，辅以良好的 UI 讲解界面：通过 3D 建模方法，将魔方的状态以及魔方的各种转动以较为美观的形式表现在屏幕上；并提供多种求解种类：整体求解与单步求解；为了使其具有更好的普适性，提供了初级、高级与进阶魔方求解教学教程。

1　开发环境及技术栈概述

1.1　开发环境

Unity 3D 2017.1.3 版本
Visual Studio 2017

1.2　运行环境

工程编译结果为 Win PE 可执行文件，在 Windows 操作系统上可直接启动入口程序 esay_cub.exe 即可运行，无需附加依赖。

1.3　技术栈概述

本作品前端 UI 基于 Unity 3D 实现，后端算法内核使用 C++开发。

1.3.1 前端技术栈：Unity、C#

Unity 3D 是一个基于 C++的功能全面的跨平台专业游戏引擎。Unity 3D 提供三维场景可视化编辑、物理特效引擎、实时音视频混流、光影和粒子效果、跨平台打包部署等属性。Unity 3D 支持当前市场常见 3D 建模软件的图形格式，可快速导入模型，并可导入动画及贴图等。Unity 底层支持 OpenGL 和 Direct 11 物理引擎，其粒子效果细腻逼真。作为当前游戏引擎的两大巨头之一，Unity 性能卓越，开发敏捷，可移植性优良，是一个优秀的三维引擎。

三维模型的动画效果实现使用脚本语言 C#编写。C#基于.NET Framework，是一种优雅而稳定的面向对象编程语言。使用 C#和 Unity 3D 对接，可以优雅的完成场景动画的帧渲染以及 Unity 场景与外部的通信。同时，现实世界的机械手的传感器也是通过网络与 Unity 通信，数据处理和实时渲染均由 C#服务脚本控制。

而作为 Microsoft 开发生态的重要组成部分的 C#和.NET 的使用使得 Visual Studio 成为了工程构建的绝佳工具。此外，Visual Studio 提供 Unity 工程支持，故而，我们选用集成开发环境 Visual Studio 开发和构建应用。

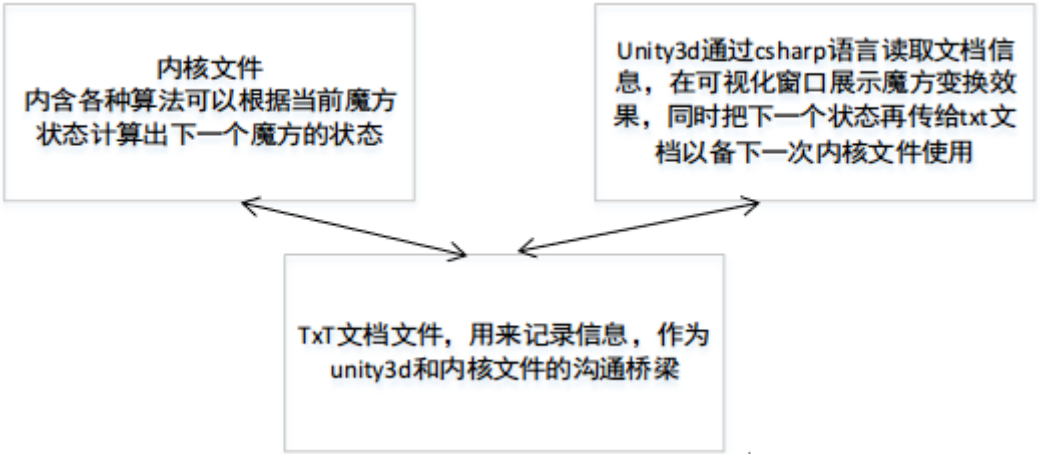
1.3.2 后端技术栈：C++、Qt

C++是一种高效的面向对象编程语言，其在标准 C 的基础上拓展了诸多属性很好的契合了内核驱动程序的开发。C++运行速度快，作为实现算法内核的工具，C++可以使程序运行速度更加有保证，程序运行更加流畅。Qt 是基于 C++的跨平台 GUI 开发框架，Qt 采用面向对象的方式封装模块，提供了丰富的 API 和模板，性能优良。

2 算法原理及技术实现

2.1 Unity 3D 部分

在经过内核的状态复杂计算之后，传回的参数可以被可视化地传达出来，这是我们这个工程最大的优点，而这部分的工作就是通过 Unity 3D 来实现的。内核接受当前的魔方状态矩阵，经过内部一系列函数计算之后，将返回的状态写入一个 txt 文档中，然后 Unity 3D 通过 C#语言读取这个变换后的状态，通过 UI 显现出来，然后将变换后的状态再导入 txt 文档，这样内核部分和 Unity 3D 可视化部分就通过一个 txt 文档巧妙地联系在一起了，他们的关系直观地可以用如下关系图来描述：



2.1.1 主菜单的实现

主菜单由输入颜色、更改魔方颜色、开始教程三个按钮型对象组成，在本实验中我们先搭好一个 Button 按钮框架让它跟其他三个模块函数利用 Unity 3D 自带的 event 事件监听器对接。这个只要编写 On_click()函数，在函数里实现相应的功能即可。输入颜色、改变魔方颜色跳转到魔方状态输入部分。

然后主菜单部分还有一个十分重要的任务--初始化魔方操作，在进入程序的时候，我们要初始化魔方的状态，由之前的约定，魔方的状态存储在一个二维数组里，而且由于内核处理的方便，我们将魔方所有的状态以 0-5 的数字来显示，所以对应魔方的每一个块，我们分配一个存储单元，用来存放它当前的状态然后我们定义一个 flag 标志，用来记录这个魔方当前的状态，这样在回到主菜单的时候，魔方就可以从数组里读取相应的状态这样就完成了我们的初始化工作

在主菜单函数里，我们还设置了一个路径函数，因为要实现与内核的链接，所以还有一个 path 函数用于查找 txt 文件和 cpp 生成的.exe 文件的绝对路径。其中用到了一个十分关键的技术——Process 类的运用

```
private static System.Diagnostics.Process p;
    public void Onclick_Begin()
    {
        change_tr(7);
        string exeName = Path.exe_path + "\\EasyCube.exe";
        Process g = new Process();
        g.StartInfo.WorkingDirectory = Path.exe_path;
        g.StartInfo.UseShellExecute = false;
        g.StartInfo.RedirectStandardOutput = false;
        g.StartInfo.FileName = exeName;
        g.StartInfo.CreateNoWindow = false;
        g.StartInfo.Arguments = "-R";
        g.Start();
        g.WaitForExit();
        comds   =   System.IO.File.ReadAllLines((Path.exe_path   +
"\\solve_out.txt"));
        //这里可以写对 comds 的后续操作
    }
```

这是 Process 类运用的大框架，它的作用是给 Unity 3D 创建一个子进程，这十分有利于我们在主程序内访问外部的 txt 文件，与它进行信息的交换，上一行中的 readallLines 方法做的就是将 txt 中内核文件处理好的状态读入，使得 Unity 3D 能进行下一步操作。

process.startinfo 是调用 process 里的一个方法。

首先设置它下面的 workingDirectory 方法：当 UseShellExecute 属性为 false 时，将获取或设置要启动的进程的工作目录。当 UseShellExecute 为 true 时，获取或设置包含要启动的进程的目录

RedirectStandardOutput 获取或设置一个值，该值指示是否将应用程序的输出写入 Process.StandardOutput 流中。当 Process 将文本写入其标准流中时，通常将在控制台上显示该文本。通过重定向 StandardOutput 流，可以操作或取消进程的输出。例如，

您可以筛选文本、用不同方式将其格式化，也可以将输出同时写入控制台和指定的日志文件中。有了这些方法，Unity 3D 和内核之间才能形成良好的互动，多线程方法是起到了至关重要的作用

最后，主菜单的 UI 的显示与消失是基于 setactive 方法实现的，原理是通过控制 UI 对象的状态，如果这个值为 false，UI 就不会出现。所以基于此，我们可以通过函数参数的传递，来控制 UI 的变化

2.1.2 魔方状态输入界面

为了最大化地利用 Unity 3D 这一可视化的优点，我们设计了魔方输入模块。魔方输入可以通过两种不同的方式，一种是利用之前写入文本的随机生成魔方状态，另一种是自主输入魔方状态。

1. 随机生成魔方状态

在主菜单中点击“输入颜色”，Unity 3D 通过调用 read 函数，将之前写好的 txt 随机状态文档读入，作为其初始状态，具有自动、随机的特点。

2. 自主输入魔方状态

这个是我们利用 Unity 3D 最大的优点之一，用户可以通过可视化输入，在一个友好的人机交互界面里 360 度全方位地设置魔方的状态，这是不同与其他解魔方程序的最大亮点之一。通过可视化输入，用户可以在学魔方的过程中，遇到不会的状态原封不动地输入进程序，Unity 3D 能实时记录下当时的状态然后反馈给内核函数，内核函数通过计算返回下一步的进行状态再通过可视化窗口反馈给用户，全程突出了可视化这一优点。具体过程可以在下一版块的过程演示截图可以看到。

自主输入魔方状态主要基于更改 cube 对象上的颜色材质，每一个材质贴图事先都设置成按钮类型，这样点击魔方方块上的颜色的时候可以做出相应事件反应，我们要做的就是当接受到用户点击颜色更改的信息的时候，将 button 上相应的 Image 信息更改，这样就完成了魔方状态（颜色）的输入

但是这样做会遇到一个问题，那就是如何处理不恰当的输入。这个问题的解决也依赖于内核的处理。在写程序的时候我们有一个约定，当输入送入内核进行计算发现该状态无解的时候，内核向文本框输出‘@’字符，有了这个“协议”之后，我们就可以编写程序获取文本信息，当检测到只有一个单字符@的时候，中止教程返回初始化。下面是我们具体的实现代码

```
if (comds[0][0] == '@')
    Te[19].SetActive(true);
else
{
    for (int i = 0; i < 7; i++)
    {
        Trans_cmd(comds[i]);
        cmds[i] = (int[])cmd.ToArray(typeof(int));
        cnum[i] = num;
    }
    Pos.Step_pos();
}
```

comds 是获取文本信息的二维数组，通过判断里面信息的内容处理异常输入，如果输入正确，通过 pos.step_pos()方法返回参数，告诉程序正常运行，否则调用 19 号文本信息，处理异常输入，提示用户重新输入。

2.1.3 教学界面交互的实现

接下来到了 Unity 3D 模块的最后一个也是最关键的一个部分了，就是教程的交互板块点击开始教程后，进入教程交互板块，在这里，我们采用的是对初学者十分友好的七步还原算法。我们对每一步都做了详细的介绍以及旋转动画，还设置了单步暂停和继续的功能。让用户有更好的体验，在 7 步算法的每一步内都可以随意暂停，单步操作。

我们利用 Unity 3Dy 的帧操作，在每一帧来到的时候采用定时功能，设置一个旋转角变量让其以一定的速度递增，当这个值大于我们的旋转阈值的时候停止，利用这个技术，让我们在不设置动画的情况下也能达到动画的效果。这样在 Unity 3D 上呈现的动画就是缓缓旋转的效果而不是突变效果。另外，我们在左上角和右下角设置了两个 text 文本，一个用来文字解说公式以及魔方的状态，这个是根据内核文件提供的公式做的文字解说，一个是将公式直接打印在了屏幕下方。

由于程序较大，变量较多，而且由于鼠标点击事件十分频繁，所以导致事件的监听不是那么容易，不同部分之间相互影响。对于这种情况，我们解决的办法就是定义全局变量 flag，通过不同函数对它状态的修改，去影响其他状态的改变，这个对监控那些互相关联的类函数十分有利，比如在本实验中暂停/继续按钮的设置：

```
public void Onclick_pause()
  {
      flag1 = flag1 ^ 1;
      obj[flag1].SetActive(true);
      obj[flag1 ^ 1].SetActive(false);
  }    //暂停按钮的设置
  public void Onclick_go()
  {
      if(T>=90&&flag==0)
      {
          flag = 1;
          T = 0;
          next++;
          if (next >= comd.Length) flag = 0;
      }
    }    //继续按钮的设置
```

可以看到，这两个函数是通过 flag1 这个全局变量来互相影响的，而继续按钮和其他主函数的程序又是通过 flag 影响的。这样使程序相关联的部分操控起来更加紧凑。

最后是七步还原法的设置，我们新建了 7 个 button 类型的按钮，赋予了相应的 click 函数，当点击第 i 步的按钮时，Unity 3D 会从 txt 文档找相应的 string，然后转换为旋转操作，举个例子如下是第一步的 On_click 方法

```
public void Onclick_first()
   {
       change_tr(0);
       for (int i = 0; i < 27; i++)
       {
           Show_rotate.cubs[i].transform.position =
Pos.pos[0][i];
           Show_rotate.cubs[i].transform.localEulerAngles =
Pos.rot[0][i];
           Show_rotate.real_num[i] = Pos.real_num[0][i];
       }
       Show_rotate.cmd_num = cmds[0].Length;
```

```
        Show_rotate.comd = cmds[0];
        Show_rotate.Fast_rotate();
        Init.pos = Show_rotate.step_pos;
        Init.rot = Show_rotate.step_rot;
        Show_rotate.Slow_rotate();
    }
```

综上，这是整个程序 unity 部分的逻辑，利用 Unity 3D 只是为了达到可视化教学魔方的目的，其算法内涵都在内核中，我们在 u3d 里只是利用了一些关键的线程技术，以及相应的文件操作还有旋转动画，使得内核里的公式计算结果能被可视化地传达出来，这就是 Unity 3D 版块最根本的目的。

2.2 魔方求解的数学建模

在本次的程序设计团队项目中，我们所选择的魔方复原问题需要进行合理的数学建模。经过对于各种各样文献资料的查阅、以及我们五个人的探讨，我们最终决定使用以下的符号与术语来描述魔方及魔方的变换。

2.2.1 魔方的符号描述

魔方总共有六个面，如果对魔方进行坐标系描述，那么本文用 UFR 坐标系来描述魔方，假设魔方处于如下图左所示的位置，那么此刻红色的面可以用字母 F 表示，表示前平面；橙色面可以用字母 B 表示，表示后平面；绿色面可以用字母 R 表示，表示右平面；蓝色面可以用字母 L 表示，表示左平面；黄色面可以用字母 U 表示，表示上平面；白色面可以用字母 D 表示，表示下平面；如下图右所示：

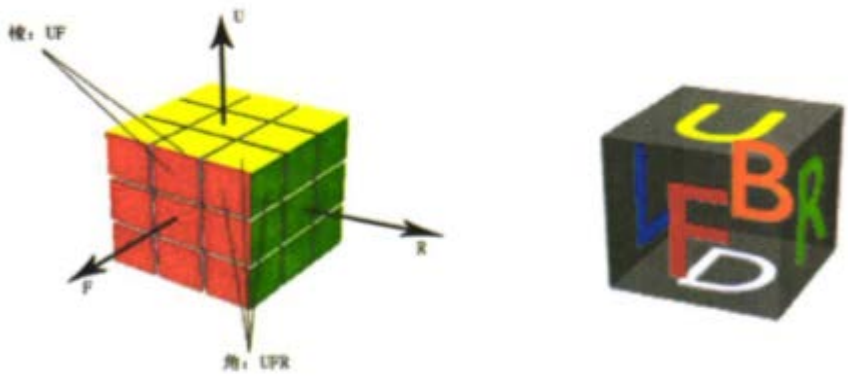


图 2-1 魔方的空间坐标系描述图

接着，魔方的一些转动操作的要用符号表示。根据对魔方面的符号表示，有如下比较简单的约定：

◆顺时针用字母表示，如 U 表示上层顺时针旋转 90 度，或者 U1。

◆逆时针用右上角加一撇表示，如 U’表示上层逆时针旋转 90 度。

◆90 度用加 1 来表示或者直接用大写字母表示，如 U1 表示上层顺时针旋转 90 度，U 也可以表示上层顺时针旋转 90 度。

◆180 度用加 2 来表示，如 U2 表示上层顺时针旋转 180 度，U’2 表示上层逆时针旋转 180 度。

◆270 度用加 3 来表示，U3 表示上层顺时针旋转 270 度，U’1 也表示上层顺时针旋转 270 度。

◆对于转动序列是重复串的情况用大括号和数字 2 表示，如 U3F3U3F3 可以用(U3F3)2 来表示。

◆E 表示中间层按照 U 方向转动 90 度，M 表示中间层按照 R 方向转动 90 度，S 表示中间层按照 F 方向转动 90 度。

同理规定 D、F、B、L、R 的转动表示。

最后，魔方的角块和棱块要用符号表示。

在实际的程序中，我们按照规定的符号输出魔方旋转的指令，以下图为例，第一步的魔方旋转指令为：RDFDB’：

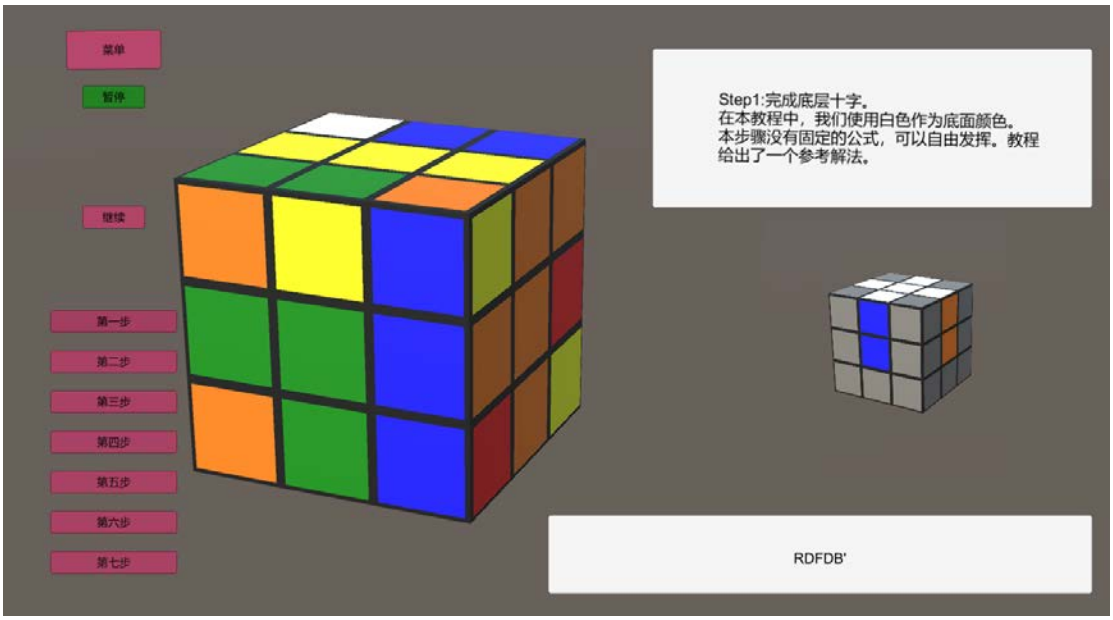


图 2-2 魔方指令示意图

魔方的中心块是没有方向性的，所以可以位于右平面和前平面之间的棱块小立方体可以用字母组合 FR 表示，位于右平面、前平面、和上平面之间的顶角上的那个角块小立方体可以用字母组合为 UFR 表示，以此类推。

十二个棱块写成矩阵形式可以表示为：

$$A=\begin{bmatrix} UF & UR & UB & UL \\ DF & DR & DB & DL \\ FR & FL & BR & BL \end{bmatrix}$$

八个角块写成矩阵形式可以表示为：

$$B=\begin{bmatrix} UFR & URB & UBL & ULF \\ DRF & DFL & DLB & DBR \end{bmatrix}$$

2.2.2 魔方的状态描述

为了更加方便观察每一个小面的状态，以下给出对魔方的数字表示法。容易知道，每一个魔方状态都可以得到相应确定的唯一的魔方六个面的展开图。下面是魔方的原始状态或者说是复原状态的平面展开图，如图 b 所示。也就是说 U 面的其他小面分别用数字 l，2，3，4，5，6，7，8 来表示，其他面以此类推。最后用数字 1 到 48 的数字排序来完成表示一个魔方的任务。

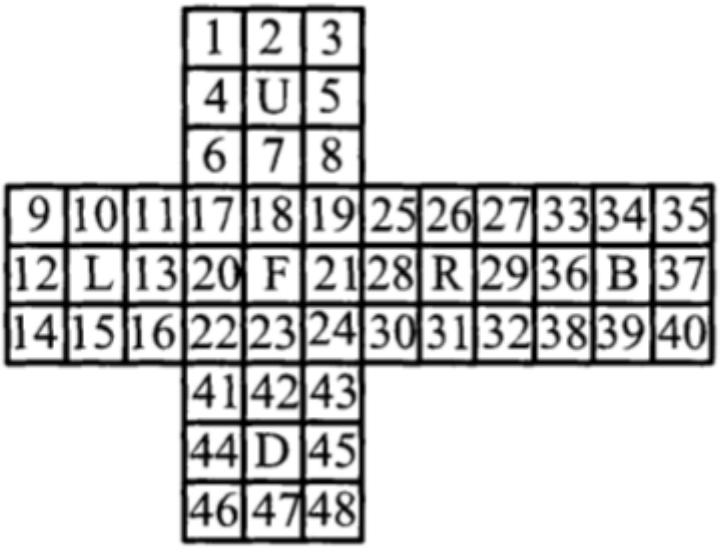


图 2-3 魔方状态描述示意图

2.2.3 魔方置换群的基本概念

在这次的程序设计项目中，我们经过对于魔方数学模型的详细研究，结合已存在的研究结果、和我们刚刚学习完毕的“离散数学 2”，最终采用了魔方置换群作为解决问题的主要数学模型，为了方便下文中对于解魔方原理及步骤的分析，以下首先给出相关数学概念：

定义 2.2.1 代数运算：设 A 是一个非空集合，任意一个 A*A 到 A 的映射成为定义在 A 上的一个代数运算。

定义 2.2.2 群：整数和运算 "+" 一起，形成了一个数学对象，它属于一个广泛的类，这类对象具有相似的结构性质。为了适当地理解这些结构，而不用个别地处理所有具体情况，发展出了下列抽象定义来涵盖上述和很多其他例子。

群是一个集合 G，连同一个运算 "·"，它结合任何两个元素 a 和 b 而形成另一个元素，记为 a · b。符号 "·" 是对具体给出的运算，比如上面加法的一般的占位符。要具备成为群的资格，这个集合和运算 (G, ·)必须满足叫做群公理的四个要求：

.	封闭性。	对于所有 G 中 a, b，运算 a · b 的结果也在 G 中。	
.	结合性。	对于所有 G 中的 a, b 和 c，等式 (a · b) · c = a · (b · c) 成立。	
.	单位元。	存在 G 中的一个元素 e，使得对于所有 G 中的元素 a，等式 e · a = a · e = a 成立。	
.	反元素。	对于每个 G 中的 a，存在 G 中的一个元素 b 使得 a · b = b · a = e，这里的 e 是单位元。	

进行群运算的次序是重要的。换句话说，把元素 a 与元素 b 结合，所得到的结果不一定与把元素 b 与元素 a 结合相同；等式

$$a \cdot b = b \cdot a$$

不一定恒成立。这个等式在整数于加法下的群中总是成立，因为对于任何两个整数都有 a+b=b+a（加法的交换律）。但是在下面的对称群中不总是成立。使等式 a·b=b·a 总是成立的群叫做阿贝尔群。

定义 2.2.3 对称群：设 X 是一个集合（可以是无限集），X 上的一个双射：a:X→X（即是置换）。集合 X 上的所有置换构成的族记为 S(x)，S(x)关于映射的复合运算构成了一个群，当 X 是有限集时，设 X 中的元素个数为 n，则称群 S(x)为 n 次对称群。

定义 2.2.4 置换群：n 对称群的任意一个子群，都叫做一个 n 元置换群，简称置换群。

置换群是最早研究的一类群，是十分重要的群，每个有限的抽象群都与一个置换群同构，也就是说，所有的有限群都可以用它来表示，下面我们将把置换群的概念引入魔方。

2.2.4 魔方置换群的定义

定义 2.2.5 魔方的转动合成运算:对魔方进行若干次转动的操作称为魔方的转动合成运算,转动合成运算也称为转动序列。规定转动合成运算是从左到右的。若用 RU 表示先转动魔方 R(右)面，再转动魔方的 U(上)面，用 C 表示魔方某一状态，即各块和小面的方位，用 M(c)表示魔方在状态 C 下经转动合成运算 M 后所生成的新状态。考虑到转动合成运算是从左到右的，有：

$$M1M2(c)=M2(M(c)$$

容易知道，对于魔方来说任何一个转动序列都有逆转动序列，一个魔方进行一个转动序列，再进行这个转动序列的逆序列，那么该魔方必然回到原来的状态。

定义 2.2.6：魔方群：如果把魔方的所有的状态放到一个集合 G 里面，显然 G 是一个非空有限集合,在 G 上定义了以魔方的转动合成作为代数运算,那么就会构成一个群,可以称为魔方群。

魔方的转动合成可以看作对于原状态集合的一个置换，由此便可以使用置换群的相关数学理论。下面将继续引入一些关于异常的概念。

2.2.5 魔方的异常状态定理

魔方的错误状态指的是魔方被错误的组装，而导致的无法复原的情况。也就是说找不到转动序列，使得魔方复原。当然只要不是魔方的正确状态，那么就是魔方的错误状态。魔方的一些错误状态如下图所示。

图 2-4 魔方的异常状态示意图

魔方异常状态定理：对正确状态的三阶魔方无论进行怎么样的转动，找不到转动序列 P 使得魔方变化为以下四种情况：

(1)只有两个棱块对换位置，其他块都是复原位置

(2)只有两个角块对换位置，其他块都是复原位置

(3)只有单个棱块翻转，其他块都是复原位置

(4)只有单个角块扭转，其他块都是复原位置

依照异常状态定理，我们编写了异常检测模块，当初始化输入不正确时会进行报错提示：

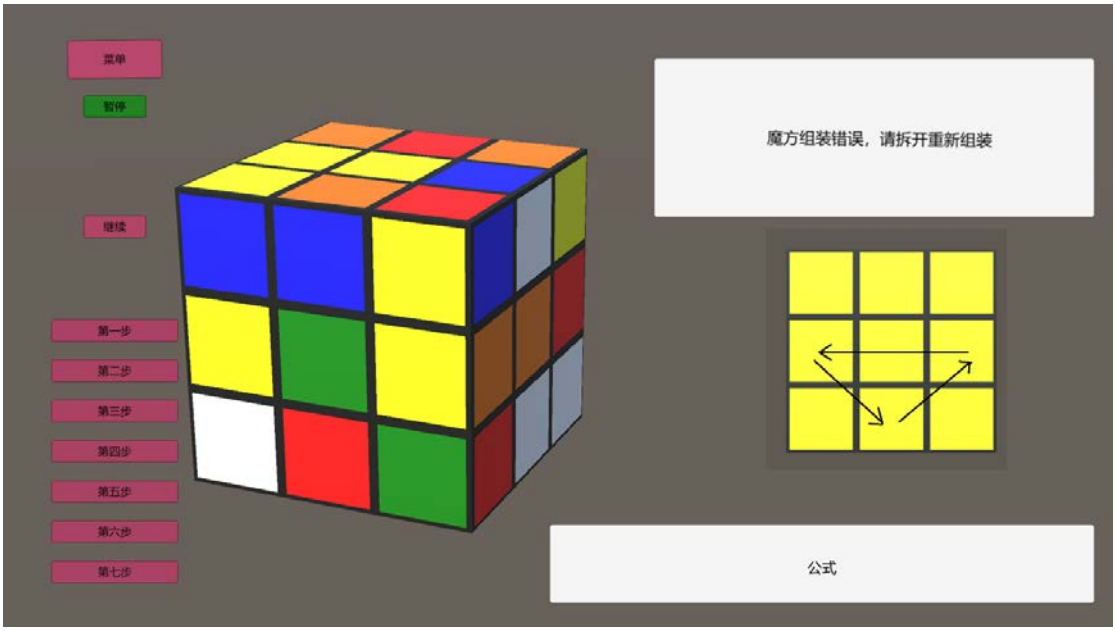


图 2-5 魔方的异常状态的报错提醒

2.3 魔方还原算法及其 C++实现

在魔方的复原方法中，出于对教学软件使用者友好的考虑，我们最终选择了较为经典易学的“七步还原法”，七步还原法的步骤在网上有非常详细的资料，在此不再赘述。

2.3.1 算法内核的实现解析

魔方复原程序部分核心代码如下：

```
int Cube::Judge_Up_Edgepermu() const
//判断棱块排列的状态：
//0：排列正确 1：十字交换（LR 对换， FB 对换） 2：相邻交换 1（LF 对换， RB 对换）
//3：相邻交换 2（LB 对换， FR 对换）
//5-8 为顺时针三棱换的情况 5：L 正确 6：F 正确 7：R 正确 8：B 正确
//9-12 为逆时针三棱换的情况 9：L 正确 10：F 正确 11：R 正确 12：B 正确
    {
    if ( block[11] == LEFT_COLOR && block[38] == BACK_COLOR && block[29] == RIGHT_COLOR && block[20] == FRONT_COLOR )
        return 0;
    if ( block[11] == RIGHT_COLOR && block[38] == FRONT_COLOR && block[29] == LEFT_COLOR && block[20] == BACK_COLOR )
        return 1;
    if ( block[11] == FRONT_COLOR && block[38] == RIGHT_COLOR && block[29] == BACK_COLOR && block[20] == LEFT_COLOR )
        return 2;
    if ( block[11] == BACK_COLOR && block[38] == LEFT_COLOR && block[29] == FRONT_COLOR && block[20] == RIGHT_COLOR )
        return 3;
    if ( block[11] == LEFT_COLOR )
        {
        if ( block[38] == RIGHT_COLOR && block[29] == FRONT_COLOR && block[20] == BACK_COLOR )
            return 5;
        if ( block[20] == RIGHT_COLOR && block[38] == FRONT_COLOR && block[29] == BACK_COLOR )
```

```
            return 9;
        }
    if ( block[20] == FRONT_COLOR )
        {
        if ( block[11] == BACK_COLOR && block[38] == RIGHT_COLOR &&
block[29] == LEFT_COLOR )
            return 6;
        if ( block[11] == RIGHT_COLOR && block[38] == LEFT_COLOR &&
block[29] == BACK_COLOR )
            return 10;
        }
    if ( block[29] == RIGHT_COLOR )
        {
        if ( block[11] == BACK_COLOR && block[38] == FRONT_COLOR &&
block[20] == LEFT_COLOR )
            return 7;
        if ( block[11] == FRONT_COLOR  && block[38] == LEFT_COLOR &&
block[20] == BACK_COLOR )
            return 11;
        }
    if ( block[38] == BACK_COLOR )
        {
        if ( block[11] == RIGHT_COLOR && block[29] == FRONT_COLOR &&
block[20] == LEFT_COLOR )
            return 8;
        if ( block[11] == FRONT_COLOR  && block[29] == LEFT_COLOR &&
block[20] == RIGHT_COLOR  )
            return 12;
        }
    qDebug() << "Judge_Up_Edgepermu,ERROR!";
    return -1;
}
```

此函数判断魔方棱块的排列状态，将状态传递给 SolveUpEdgepermu 函数，由它给出下一步的复原操作。

```
QString Cube::SolveUpEdgepermu() const{
    int situ = Judge_Up_Edgepermu();
    const QString s1 = " R2 U R U R' U' R' U' R' U R'";
    const QString s2 = " R U' R U R U R U' R' U' R2";
    if (situ == 0) return "";
    if (situ == 1) return s2 + " U" + s2 + " U'";
    if (situ == 2) return s2 + " U2" + s1 + " U2";
    if (situ == 3) return s2 + " U'" + s2 + " U";
    if (situ == 5) return " U" + s1 + " U'";
    if (situ == 6) return " U2" + s1 + " U2";
    if (situ == 7) return " U'" + s1 + " U";
    if (situ == 8) return s1;
    if (situ == 9) return " U" + s2 + " U'";
    if (situ == 10) return " U2" + s2 + " U2";
    if (situ == 11) return " U'" + s2 + " U";
    if (situ == 12) return s2;
    qDebug() << "SolveUpEdgepermu ERROR!";
    return 0;
}
```

此函数接收 Judge_Up_Edgepermu 函数返回的状态，根据复原算法给出下一步的复原操作。

2.3.2 核心算法与数据结构描述

魔方看似简单，但由于其灵活的变换，一个三阶魔方所包含的正确状态总数就有43252003274489856000 种。即使找到了合适的复原魔方步骤，合理的内部算法优化对于程序来讲也是必不可少的。幸运的是，我们小组中的薛昕磊同学有着丰富的 ACM 参赛经验，对于各种搜索算法以及优化数据结构有着很深的理解，以及熟练的掌握。最终我们主要采用了（迭代加深）启发式搜索来 solve 魔方，并采用红黑树这种数据结构来保证程序在最坏情况下，搜索时间也能满足要求。

迭代加深搜索是一种不断拓展搜索范围的搜索方式。可以将其形象成一棵树，这棵树有无数层（深度无限深），每一层有无限个节点（无限广）因此，无论是 DFS 还是 BFS，都不能拓展一支，拓展一层。而其答案可能就在第二层第二支上……为了求解这种问题，有了迭代加深搜索。

顾名思义，迭代加深就是逐渐增加搜索的范围。

最初我们只搜索 maxd 的范围，当 maxd 内没有我们需要的答案时，我们将 maxd+1，而后继续搜索。

```
for(maxd=1;;maxd++)
  if(dfs(...))
    break;
```

迭代加深算法，又称启发式搜索。用公式表示为

$$f(n)=g(n)+h(n)$$

其中：

f(n)是初始点经由节点 n 到目标点的估价函数

g(n)是在状态空间中从初始节点到 n 节点的实际代价

h(n)是从 n 到目标结点的最佳路径的估计代价

具体是什么意思呢，就是我们要优先选择一条我们估计最好的分支进行拓展，对于我们估计不是很好的分支进行剪枝。由于最好只是我们的估计，因此，如果估计的不合理，可能会丢失最优解或者并没有有效提升效率。

在程序中，我们根据魔方当前的状态作为指导搜索的路标，使得对于魔方下一步的解法的破解搜索向最有希望的方向前进，降低了复杂性。通过删除某些状态及其延伸，启发式算法可以消除组合爆炸，并得到令人能接受的解。

红黑树是在普通二叉树上，对没个节点添加一个颜色属性形成的，同时整个红黑二叉树需要同时满足一下五条性质：

性质一：节点是红色或者是黑色
性质二：根节点是黑色
性质三：每个叶节点（NIL 或空节点）是黑色
性质四：每个红色节点的两个子节点都是黑色的
性质五：从任一节点到其没个叶节点的所有路径都包含相同数目的黑色节点

这五条性质约束了红黑树，可以通过数学证明来证明，满足这五条性质的二叉树可以将查找删除维持在对数时间内。当我们进行插入或者删除操作时所作的一切操作都是为了调整树使之符合这五条性质。

在程序中，我们使用红黑树来存储我们想要搜索的目标解法，配合迭代加深启发式搜索，可以使魔方最坏状态下的破解时间维持在对数时间之内。

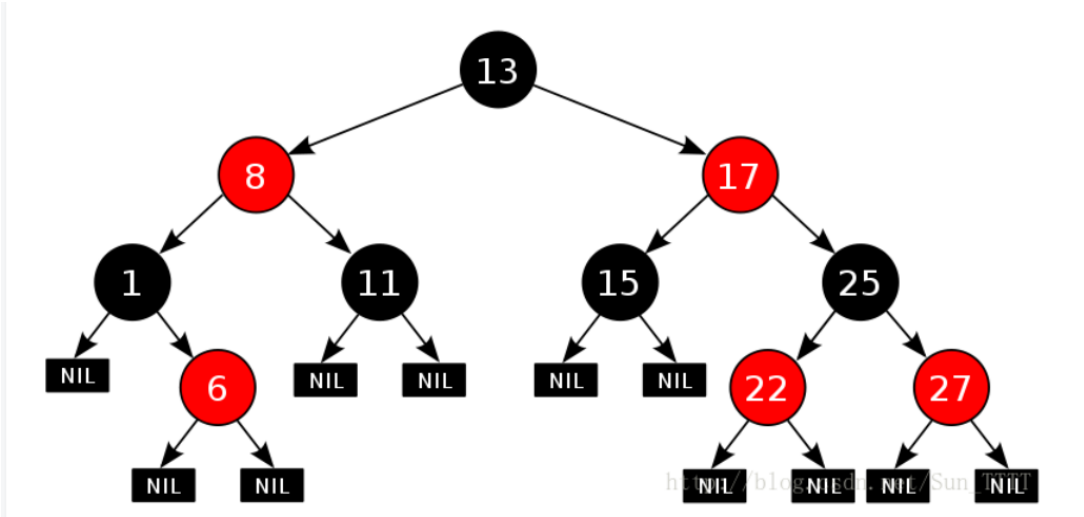


图 2-6 红黑树示意图

3 实验过程

Easycube 简单魔方程序设计包含后端算法设计和前端 UI 设计两个部分。前端 UI 设计由龙博、杨悦航两位同学完成，后端算法设计主要由薛昕磊、刘啸尘、李英平三位同学完成。程序设计、编写、调试以及对接整个过程从 7 月 11 日开始，到 7 月 20 日完成，共计 10 天。

3.1 Unity 3D 开发过程记录

UI 界面部分主要依赖于 Unity 实现 3D 效果，应用光影、画布等模块，实现了场景模型构建，鼠标跟踪调用等功能需要。

负责前端 UI 设计的杨悦航和龙博同学都曾使用 Unity 设计过程序，因而我们确定了基于 Unity 实现魔方可视化。在编写程序前，我们首先从图书馆借阅了 Unity 开发的有关书籍，进一步学习前端 UI 的设计方法，用时 1 天，我们主要阅读的书目如下：

1. 《Unity 3D 脚本编程——使用 C#语言开发跨平台游戏》陈嘉栋（这本书介绍了 3D 游戏开发的各个过程，讲解清楚且浅显易懂）
2. 《Unity 游戏设计与实现 南梦宫一线程序员的开发实例》加藤政树（这本书内容比较深奥，知识代替浏览了思路）
3. 《Unity 官方案例精讲》Unity Technologies （这本书内容主要是一些 Unity 开发案例，其中一些案例给了我们启发）

由于时间比较紧张，我们只把上述书籍中与魔方可视化设计有关的部分进行了详细的阅读，其余部分只是进行了大体翻阅。

在具备了理论基础后，我们进行了前端 UI 整体框架的设计，用时 1 天。在这一阶段，我们确定了与后端算法对接的方案，即将魔方展开图不同位置的颜色输出到文件进行信息传递。同时，我们将前端 UI 的设计进行分工，龙博同学负责开发程序主界面、魔方展示功能，杨悦航同学负责开发魔方的修改颜色功能。

我们用 4 天时间分别完成了各部分的代码，并用 2 天时间进行代码整合和调试。在此过程中，我们将设想的程序逻辑和功能进行了一些调整，发现并修复了一些 bug。在后端算法程序设计完成后，经过我们五人的讨论，决定在程序中加入执行魔方教程的“暂停”与“继续”功能，提升用户的使用体验。

在最终设计的 UI 中，主页面包括了输入颜色、更改魔方颜色、开始教程三个选项。点击“输入颜色”按钮会将魔方打乱，点击“更改魔方颜色”按钮进入用户自定义魔方

颜色状态模式。“开始教程”是魔方教学功能。最终我们实现的 UI 效果将在后面部分展示。

3.2 算法内核开发过程记录

后端算法需要完成的主要功能是求解魔方。我们知道，即使是求解魔方最简单的“层先法”，想要尽可能减少使用公式的次数，也有需要判定很多种情况，以确定在哪一个方向上使用公式。因而，后端算法设计的难点不单单在于代码的编写，更在于数据结构的合理使用、多种情况的判断和决策，以及设计高效的算法找到复原魔方最简单的方案。

这部分程序设计由薛昕磊、李英平、刘啸尘三位同学完成，由于三位同学对 C++语言比较熟悉，后端算法采用 C++语言进行实现。

我们在编写程序前，首先对魔方状态存储的数据结构进行了设计。一种比较显然的存储方法是，我们将魔方按一种确定的展开方式得到展开图，然后对 6 个面共计 54 个颜色块依次进行标号用来存储，这样，魔方的颜色状态就可以存储在一个一维数组中了。

但是，这种方法有着不小的弊端，我们即使做一次最简单的转动，展开图中各个块的颜色也会发生不小的变化，且变化的规律并不显然，这给我们魔方状态的转移造成了很大的困难。魔方的转动方式有上、下、左、右、前、后六个方位，考虑到顺时针、逆时针，每一次操作有 12 种转动方法，就对应 12 种展开图的变换，程序设计会比较复杂。

经过我们用时 1 天的讨论，并没有想到其他更加简便的做法，最终还是决定采用展开图标号的方式进行存储。在程序编写中，魔方转动时计算颜色变化比较容易出现 bug。为了解决这一问题，我们购买了一块魔方，按照展开图标号的顺序用记号笔对它进行了标号，并将所有 12 种转移方式转移前后的标号变化画在纸上，将程序需要的赋值数据提前写好，从而保证在程序编写时头脑清晰。

在敲定程序使用的数据结构后，李英平同学与负责 UI 开发的同学讨论了接口定义以及程序的对接方案，用时 1 天。最终确定的方案是，前端将当前的魔方状态输出到一个文档中，后端程序从文档中读取当前魔方状态，并计算出下一个魔方状态，输出到另一个文档中，前端程序根据后端算法的输出显示这一步的转动。

在上述准备工作完成后，我们开始后端程序的开发，用时 5 天。我们每天根据上一天代码的完成情况以及当前的需求分配当天每个人需要完成的代码模块，并且采用迭代式开发的方法，每实现一个功能都进行充分的测试，降低代码的调试难度。

最后，我们与前端完成对接并进行功能测试。通过启发式搜索、迭代加深启发式搜索、以常数优化为目的的剪枝、基于红黑树的算法，我们高效地实现了求解魔方、输出当前操作的功能，程序完成了预期的功能。

4 实验结果

4.1 作品功能简述

该作品旨在让更多人能够快速掌握复原魔方的方法，能够在 30 秒内完成复原魔方。通过良好的交互界面，讲解魔方的高级解法，宣传魔方真正的魅力。

该软件具有良好的讲解界面：通过 3D 建模方法，将魔方的状态以及魔方的各种转动以较为美观的形式表现在屏幕上；并提供多种求解种类：整体求解与单步求解；为了使其具有更好的普适性，提供了初级、高级与进阶魔方求解教学教程。

4.2 软件使用方法

1.启动程序 esay_cube 目录下 esay_cub.exe，选择界面分辨率。（推荐 1600×900，此分辨率下显示效果最佳。）

2.点击主菜单中更改魔方颜色功能修改魔方初始状态，在此功能里，先点击屏幕中间色块，再点击右侧颜色即可将该方块改为对应的颜色。（为方便测试演示，在主界面里点击输入颜色按钮会从文件里加载一套初始状态方案。）

3.点击开始教程按钮，开始魔方教学。

4.依次点击左侧第一步至第七步的按钮。执行教学，右侧的界面会出现逐步的操作动画以及操作公式及文字解析。

5.点击菜单按钮返回主菜单重新配置魔方，或退出应用。

4.4 应用运行效果展示

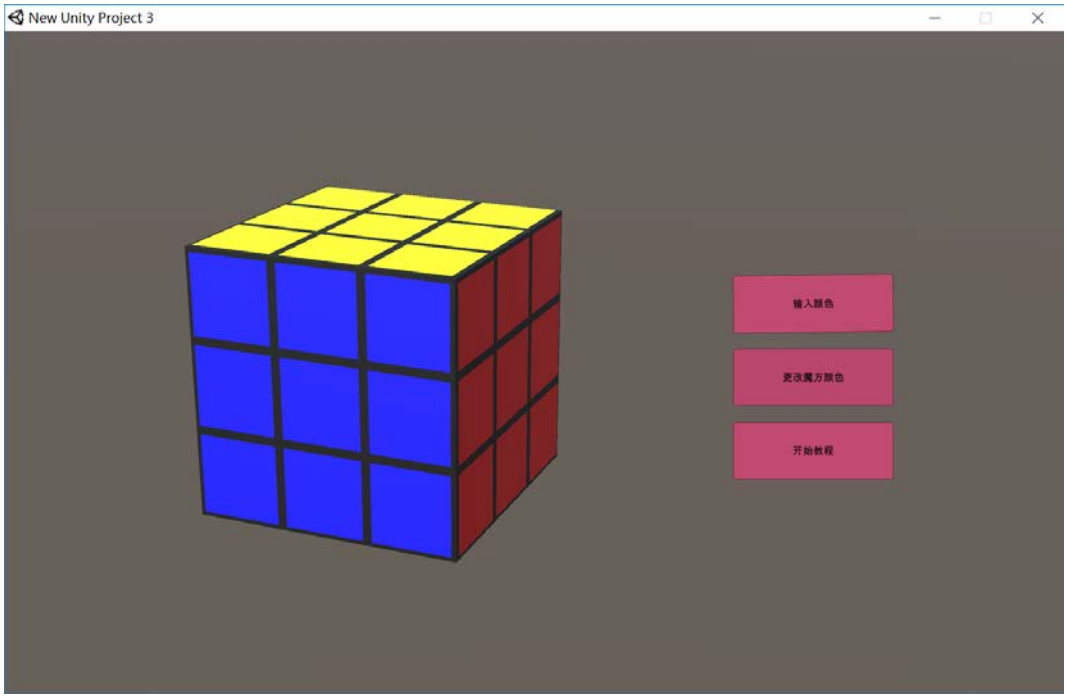


图 4-1 Easycube 魔方教学主菜单

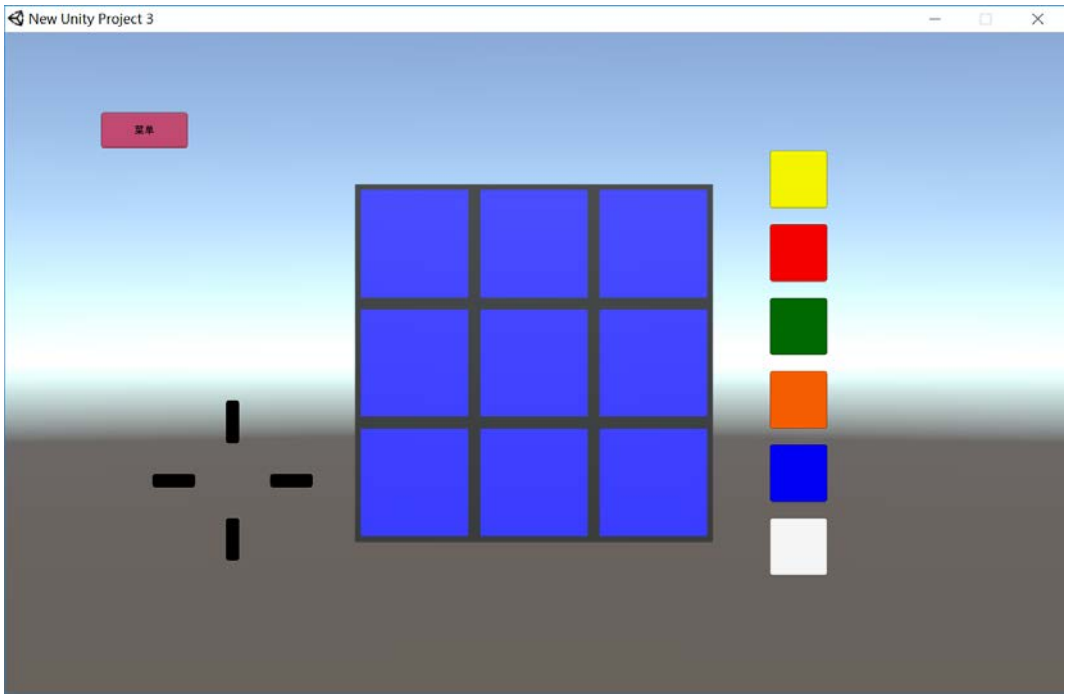


图 4-2 改变魔方颜色功能界面图

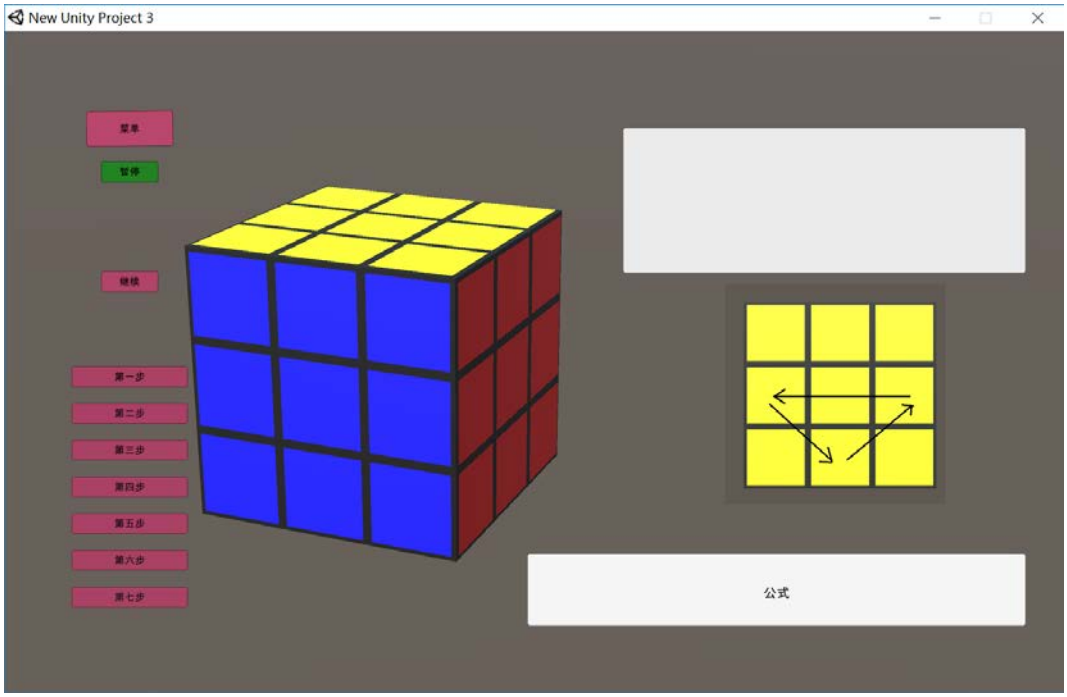


图 4-3 魔方教学界面图

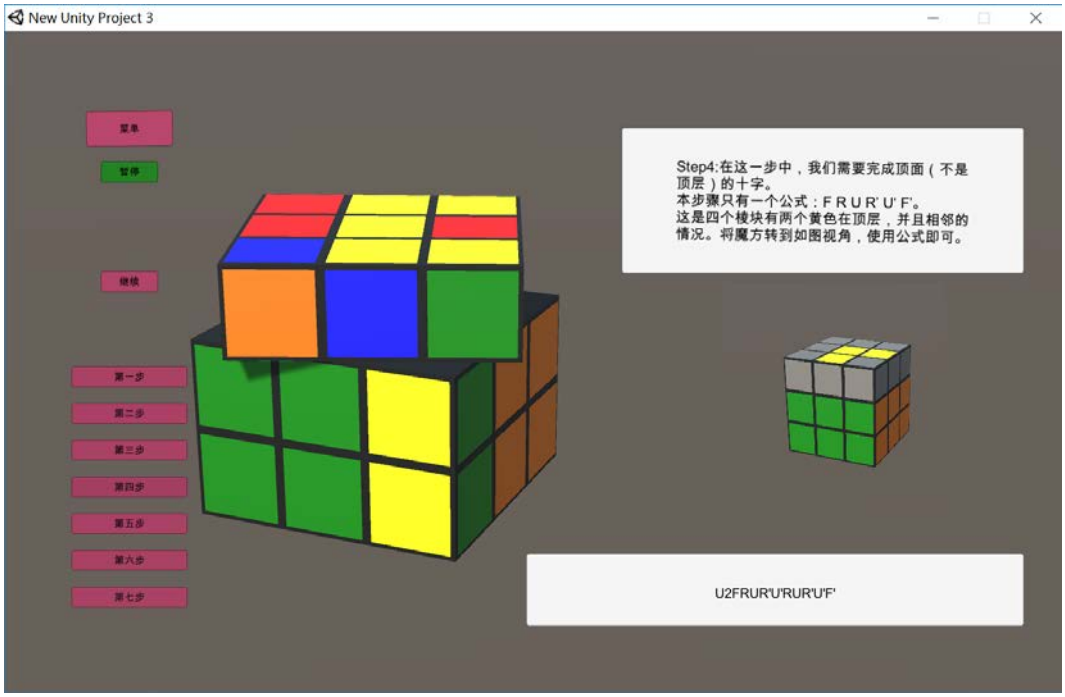


图 4-4 魔方旋转并给出公式解说效果示意图

5 实验总结

我们小组经过几天的奋斗，获得了完整的 exe 文件，其中 UI 界面部分主要依赖于 Unity 实现 3D 效果，应用了光影、画布等模块，实现了 game object model，鼠标跟踪调用，main camera model 的 Unity 模块。后端算法则主要运用了启发式搜索、迭代加深启发式搜索，以常数优化为目的的剪枝，基于红黑树的算法。并以 Cross，First two layers，Orientation of last layer，Permutation of last layer 四个步骤完成了嵌入智能图文教程部分。最终我们的产品具有良好的交互性，较为完整的页面，具体易懂翔实的教程，结合界面展示，可以适合各个水平的魔方爱好者。我们的项目有不少创新点，比如市面上的魔方初级教程多为文字和视频模式，少数为 Flash 动画模式，但都没能做到满足用户的个性化需求。而我们的项目通过设计算法，对于用户的输入生成不同的教程，可以使得用户能够轻松复原魔方；市面上的魔方高级教程晦涩难懂，让刚刚学会复原的新手望而却步。本项目将魔方高级解法化整为零，以更加智能的手段将高级解法展现给用户，使得用户能够真正理解高级解法的奥妙所在。

通过这个项目我们收获了很多，也认识到自己仍然有许多不足。最终这个项目有了一个较为完整的产品，可以翔实地为各个层次地魔方玩家提供辅助性的讲解教程，有了一个较为完整的产出。这其中，我们更详细地了解了各种软件，如 Qt，Unity 等。也进一步了解了各个软件的优劣。同时，我们也多次尝试了不同的算法进行优化，将理论结合实际，收到了一定的效果。同时我们也明白了良好交互性是设计产品的必须元素，而锐意创新则是产品的生命力所在，针对用户需求才能提供更具价值的产品。

我们也有一定不足，比如没能进行更好的优化，也许可以找到更适合的算法，界面也缺乏一定的美观性。这些缺憾也将促使我们在以后的时间内继续完善产品。