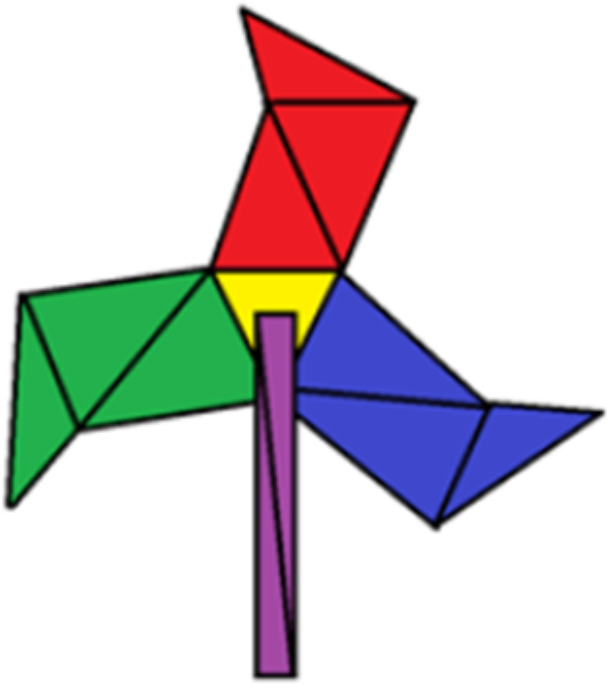
Lab1:Spinning-Pinwheel

需求

- 构建且渲染一个风车模型（如下图所示）。其中，三个叶片（包括中间的黄色三角形）在一个2D平面上，手柄在另外一个离相机更近的2D平面上。三个叶片、中心的三角形、手柄分别用不同的颜色显示。
- 实现风车的旋转动画。要求风车的三个叶片以及中间的黄色三角形（在其所在平面上）一起绕着中心一点不停旋转，且将此动画渲染出来。
- 实现通过键盘对动画的交互控制，包括切换旋转方向、增大旋转速度以及减小旋转速度。
- 设计按钮和菜单两个控件，用于动画的交互控制：点击按钮可以切换旋转方向；选择三个菜单项分别可以切换旋转方向、增大旋转速度以及减小旋转速度。



实验环境

```
CPU Intel(R) Core(TM) i5-8250U CPU @1.60GHZ 8核
IDE VS2022
包管理工具 Nuget
依赖库 nupengl.core 0.1.0.1
```

虽然nupengl已经停止维护了，但是历史版本还是可以用的

实验步骤

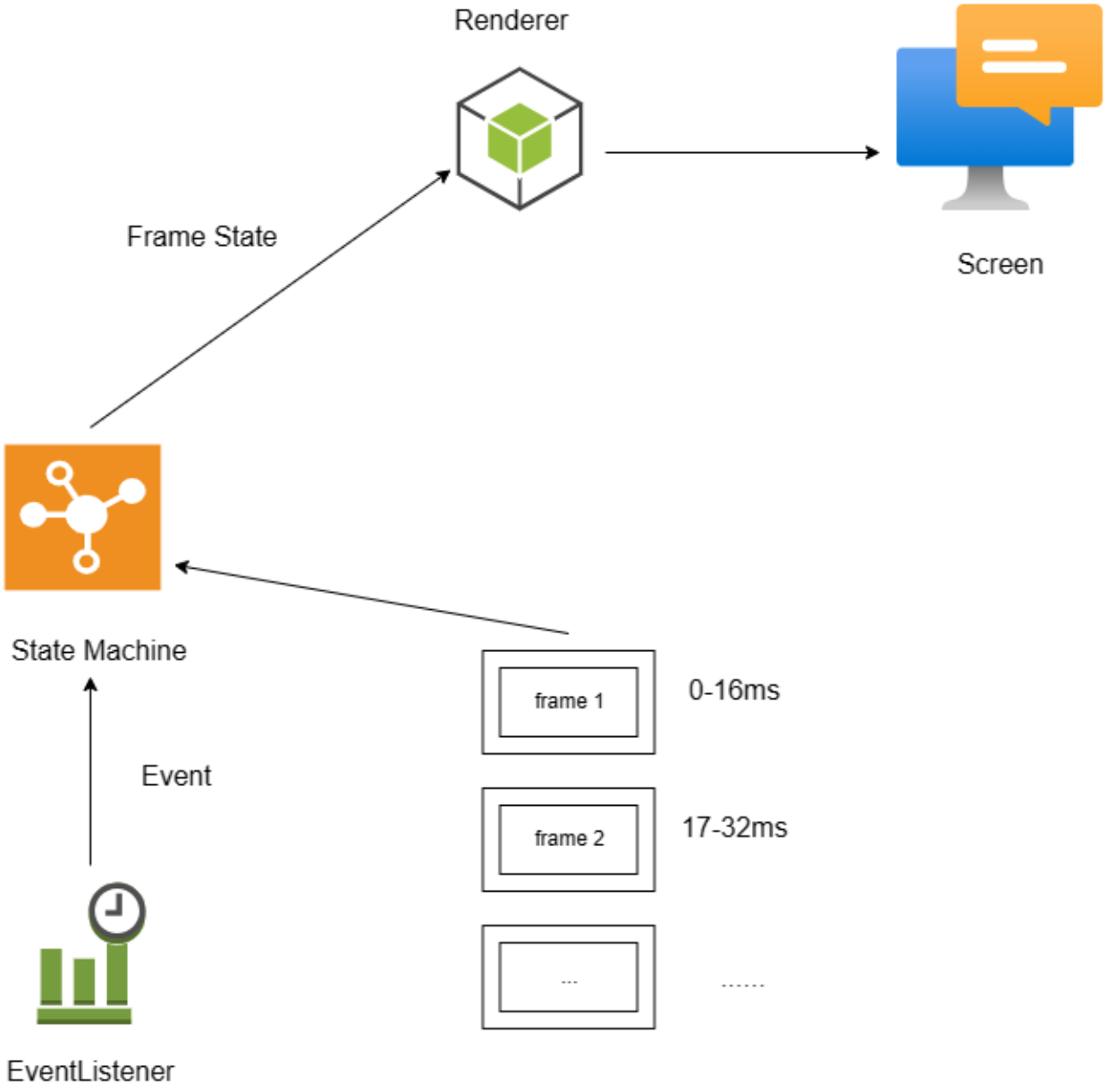
概念与流程

主要是“帧”这个概念。

在计算机图形学中，一帧是一个时间槽内渲染到显示屏上的图形界面。一般来说一秒有60帧左右（0.016s=16ms)。动画（Animation）看上去是连续的，但实际上是离散的，只是渲染时间太短了，在人眼中看起来像是连续的。

所以我们要做的就是根据需求，把业务逻辑细化到每一帧。也就是说我们要知道渲染到显示屏上的每一帧都是怎么画出来的（参数、位置、状态更新等）。然后才能下手写代码。

这里我用draw.io画了个流程图。



每过16ms就需要渲染当前帧，首先先经过一个有限状态机（Final State Machine)，确定在当前System Tick下该帧应该处于的状态。比如，当一个游戏的某个关卡结束时，状态就应该是游戏结束Game-Over。

同时，系统中会设置各种各样的EventListener。用于监听各种各样的事件。比如来自键盘或鼠标的各类事件。不过上图中没有画出的是，这样的listener一般是异步的。一个callback function并不会在事件未触发时阻塞主进程，而是在系统调用结束后负责“善后”。例如，在游戏中一个玩家因为有急事，所以选择按下ESC退出本局游戏，那么对应的监听器将发现这一举动，并做相应的逻辑处理（可能会保存游戏现场）。此外，最重要的是紧急更新状态。因为一开始游戏正处于running状态，但这一帧渲染前玩家恰好按下了ESC，那么状态机的输出应该被修改为Game-Over。

最后，渲染器（Renderer)拿到状态机给出的当前帧的状态，就顺着上面的例子说吧，Game-Over状态。因为状态机只有有限个状态输出，因此我们可以从驱动表中查出某个输出状态对应的渲染逻辑。比如，Game-Over时我们要在屏幕上打出一行“Good Game!”，那么就要在渲染时把这行字加上去。直到某一帧渲染前玩家做了别的操作（比如又重新开始了游戏），状态机的输出得以改变，这一行字才不会被继续渲染。

业务逻辑分析

从局部到整体：风车的组件

无论当前帧风车的状态是怎样的（叶片的位置，角度等），总归在应用窗口中都是有”风车“这个东西的。因此我们需要想办法渲染出风车的实体，之后才能考虑根据其他诸多因素来渲染某一帧风车的姿态。

为了渲染出整个风车，我们采用蚂蚁搬大象的方式，每一段代码渲染风车的一部分。

因为我们并不采用纹理贴图的方式来渲染风车（Texture+FrameBuffer+Animation实现动画），所以只能手动地在每一帧中都把风车给画出来。这需要使用OpenGL提供地图像绘制工具函数。包括glVertex系列与glBegin(Glenum mode)提供的画笔与策略类。

glBegin/glEnd

如果想绘制图形，那么绘制动作一定发生在被glBegin与glEnd包裹在中间的代码段。

形如：

```
glBegin(mode);
// 你的绘制点
//...
glEnd();
```

glBegin与glEnd一定是成对出现的。

GLenum

OpenGL指定了一大堆连接绘制点并填充对应包围区域的策略。这个策略类叫做GLenum（顾名思义是个枚举类）。当我们要连接绘制点时，需要告诉OpenGL他应该以什么策略连接这些点。这个策略通过glBegin被传入画笔。

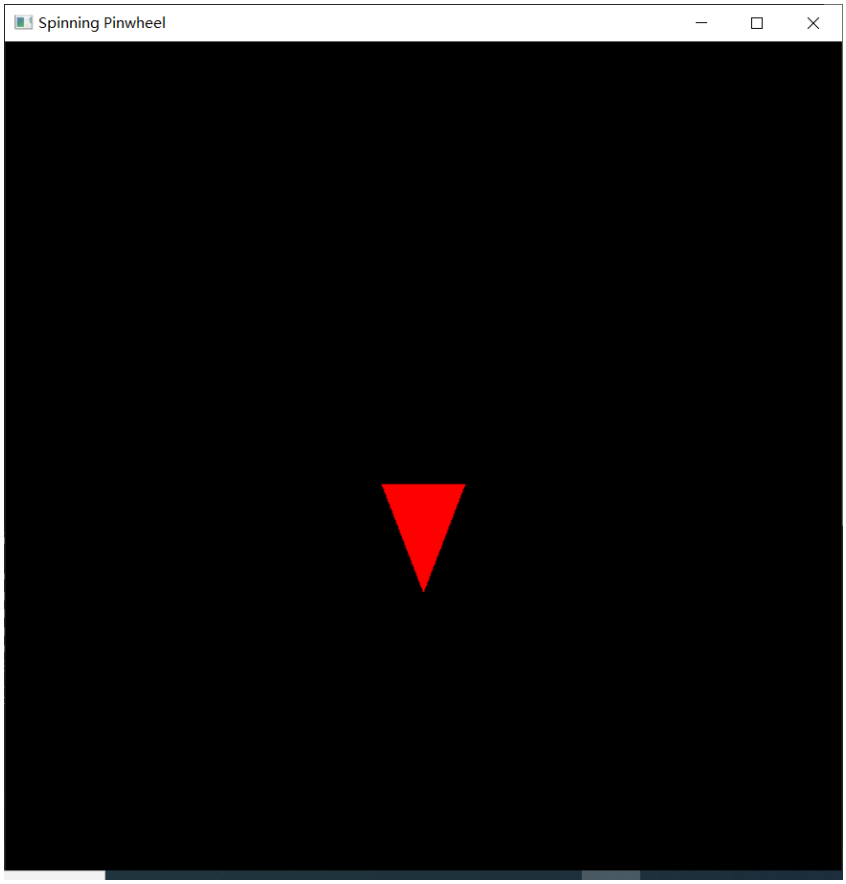
```
GLenum mode = xxx //你的绘制策略
glBegin(mode);
//你的绘制点
//...
```

举个例子，假设我想要在屏幕上绘制一个红色的三角形。那么我就要去API文档中找到绘制三角形的策略，这个策略的名字叫做`GL_TRIANGLES`。所以我们传入这个策略，并给出绘制点。

```
void drawWindmill() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    glRotatef(0, 0, 0, 1);
    glScalef(1, 1, 1);

    //red wind
    glColor3f(1, 0, 0);
    GLenum mode = GL_TRIANGLES;
    glBegin(mode);

    glVertex3f(-0.1f, -0.2 * sqrt(3) / 6, 0);
    glVertex3d(0.1f, -0.2 * sqrt(3) / 6, 0);
    glVertex3d(0.0f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);

    glEnd();

    glPopMatrix();
    glFlush();
    glutSwapBuffers();
}
```

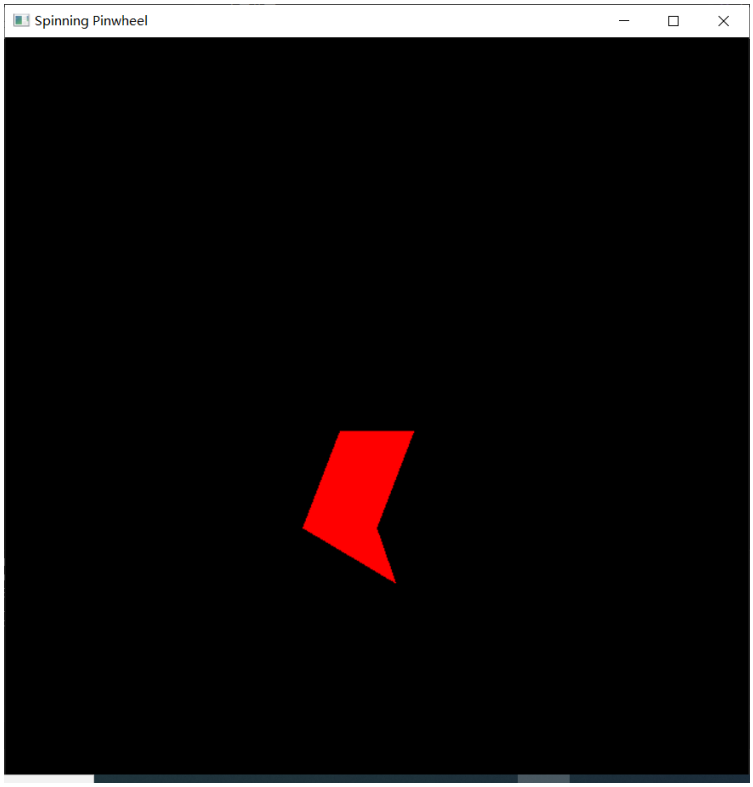
代码中有些其他API还没提到，只关注glBegin和glEnd中间的部分。如何调用这个绘制函数、具体位置参数什么的也不是这里的重点。

demo运行结果如下：



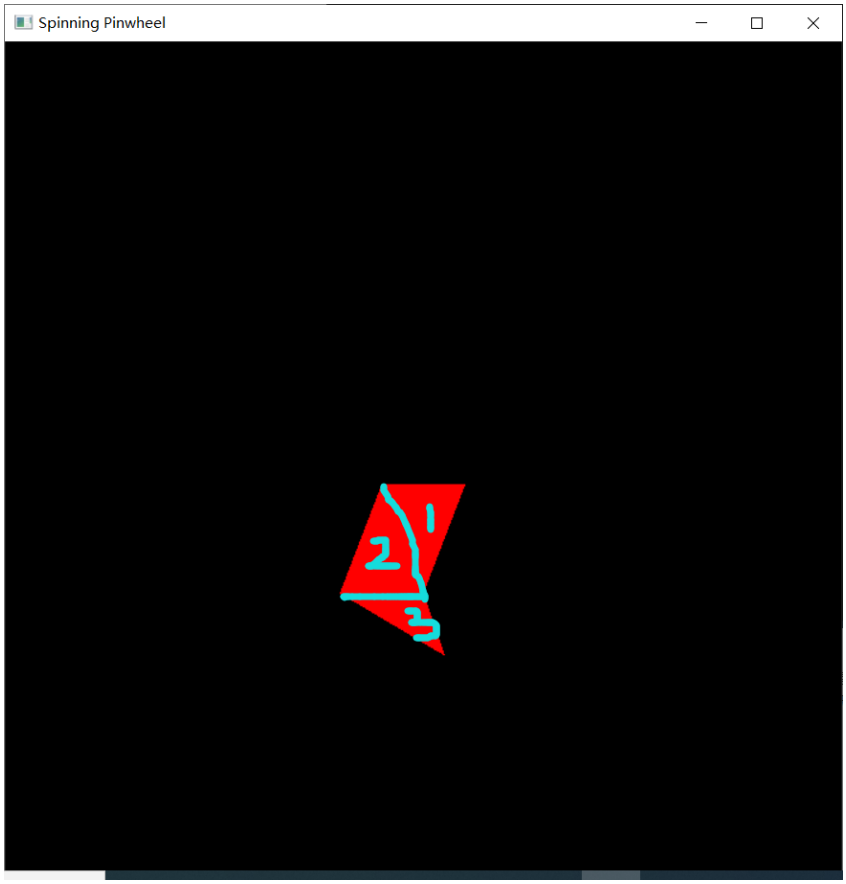
To Counter that…

其实很多时候我们想要绘制的图形并不是一个规则的，或者说直接能叫上名字的图形。比如说下面这个图形：



你觉得他叫什么比较好？一个平行四边形拼一个三角形？很怪异的名字…

但是很多图形都可以拆解成其他图形的组合，就像我刚才说的，上述图形就是一个平行四边形加上一个三角形得到的。众所周知，平行四边形本质上也是两个三角形拼起来的。所以我们可以这样分割一下。



比如像这样分割成三个三角形。这样我们通过GL_TRIANGLES画3个三角形，就能组合出这个图形了。

代码如下:

```
void drawWindmill() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    glRotatef(0, 0, 0, 1);
    glScalef(1, 1, 1);

    //red wind
    glColor3f(1, 0, 0);
    GLenum mode = GL_TRIANGLES;
    glBegin(mode);

    //绘制一号区域三角形
    glVertex3f(-0.1f, -0.2 * sqrt(3) / 6, 0);
    glVertex3d(0.1f, -0.2 * sqrt(3) / 6, 0);
    glVertex3d(0.0f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);

    //绘制二号区域三角形
    glVertex3d(-0.1f, -0.2 * sqrt(3) / 6, 0);
```

```
    glVertex3d(0.0f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);
    glVertex3d(-0.2f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);

    //绘制三号区域三角形
    glVertex3d(0.0f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);
    glVertex3d(-0.2f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);
    glVertex3d(0.1 * 27 / 52, -0.2 * 147 / 113 / 146 * 230 - 0.2 * sqrt(3) / 6, 0);

    glEnd();

    glPopMatrix();
    glFlush();
    glutSwapBuffers();
}
```

当然，GLenum也提供了自动纠错的机制，比如你声明了自己的策略是GL_TRIANGLES，但实际上传入了四个绘制点，那么第四个点将被自动省略。

这样我们可以将风车的某个叶片渲染到屏幕上了。

从静态到动态：叶片的相对位置

直到现在为止，我们仍在渲染静态的图片。等到静态的图片渲染结束后，才会加入参数和一些逻辑控制，来渲染动态的图片。所以叶片的相对位置的含义，其实是一张静态风车图中，三个叶片之间的相对位置。

观察文档中提供的效果图，三个叶片均匀地将平面划分开，因此我们可以认为这三个叶片两两之间的夹角为120°，这是我们在渲染静态风车时最重要的一环。

那么怎么实现这个过程？一种比较”数学“的方式是手工计算三个叶片的坐标，然后将推出的公式放到代码中，以此渲染。但这就很不”计算机“，为什么不将三个叶片放到同一个初始位置，然后分别旋转他们直到满足两两之间夹角为120°呢？显然后者的工作量也比较小。

因此我们只要将上一步渲染出的叶片换个颜色，然后对他施加一个旋转操作即可。我们知道在OpenGL中任何变换在底层都是通过矩阵实现的。所以对于一个叶片上的任意一点与相机原点形成的向量，我们都用一个旋转矩阵与它做矩阵乘法，得到变换后的向量，也就是这个叶片旋转后该点的对应点。

假设效果图中蓝色叶片是初始位置，那么红色叶片是蓝色叶片逆时针旋转120°，绿色叶片是蓝色叶片逆时针旋转240°。也就是我们要对红色叶片施加一次旋转操作，但对绿色叶片施加两次旋转操作，这应该怎么实现呢？

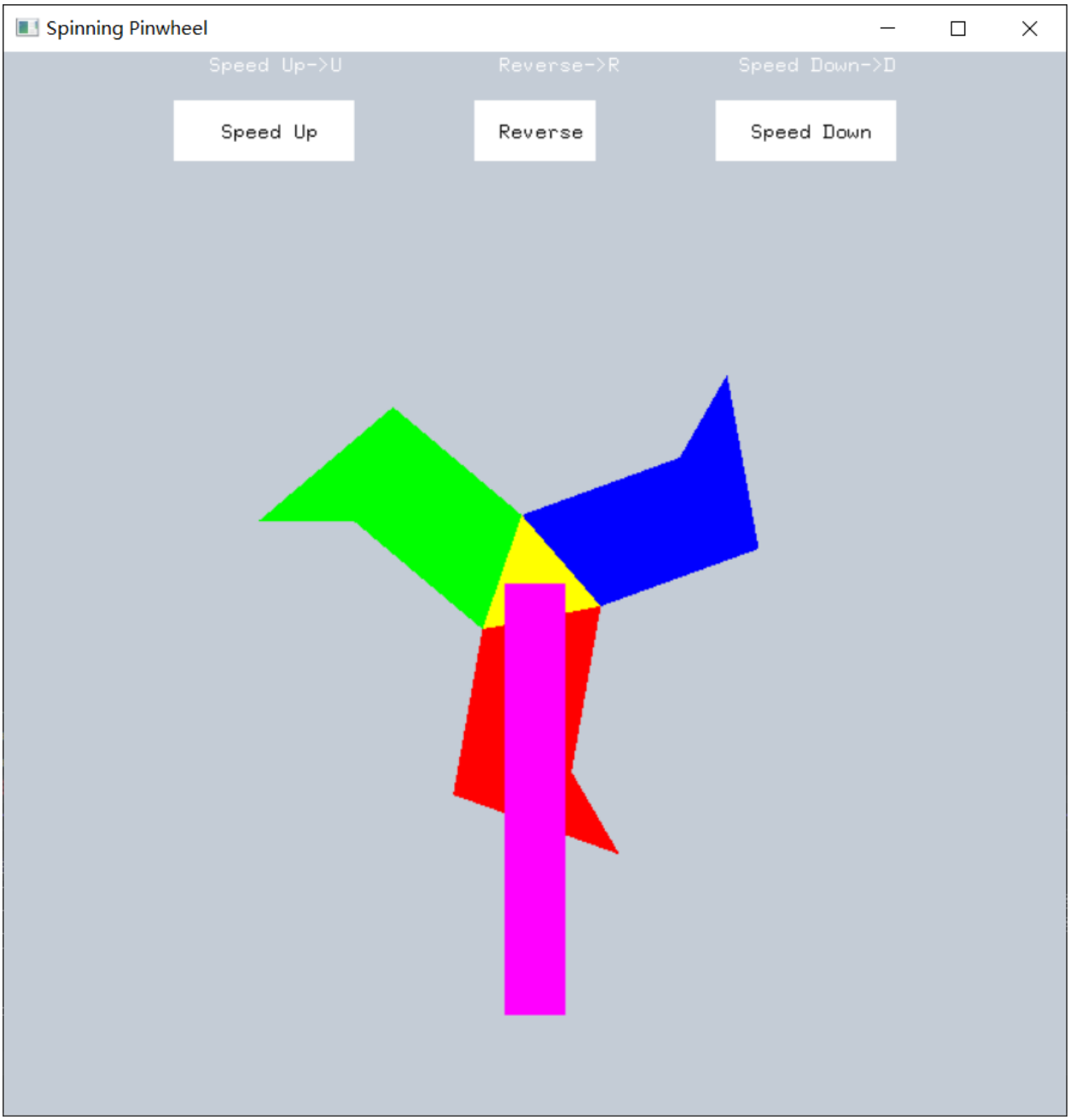
在OpenGL中，一个矩阵变换在不受到矩阵堆栈的影响时，将对之后所有的变化产生影响。比如一开始有一个Rotate操作，后续有两次变换，那么Rotate操作将对后续两次变换都产生影响。

因此我们可以这样来实现叶片的相对位置。

```
//红扇叶
glColor3f(1, 0, 0);
glBegin(GL_TRIANGLES);
glVertex3f(-0.1f, -0.2 * sqrt(3) / 6, 0);
glVertex3d(0.1f, -0.2 * sqrt(3) / 6, 0);
glVertex3d(0.0f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);

glVertex3d(-0.1f, -0.2 * sqrt(3) / 6, 0);
glVertex3d(0.0f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);
glVertex3d(-0.2f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);

glVertex3d(0.0f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);
glVertex3d(-0.2f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);
glVertex3d(0.1 * 27 / 52, -0.2 * 147 / 113 / 146 * 230 - 0.2 * sqrt(3) / 6, 0);
glEnd();

//蓝扇叶
glRotatef(120, 0, 0, 1);
glColor3f(0, 0, 1);
glBegin(GL_TRIANGLES);
glVertex3f(-0.1f, -0.2 * sqrt(3) / 6, 0);
glVertex3d(0.1f, -0.2 * sqrt(3) / 6, 0);
glVertex3d(0.0f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);

glVertex3d(-0.1f, -0.2 * sqrt(3) / 6, 0);
glVertex3d(0.0f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);
glVertex3d(-0.2f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);

glVertex3d(0.0f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);
glVertex3d(-0.2f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);
glVertex3d(0.1 * 27 / 52, -0.2 * 147 / 113 / 146 * 230 - 0.2 * sqrt(3) / 6, 0);
glEnd();

//绿扇叶
glRotatef(120, 0, 0, 1);
glColor3f(0, 1, 0);
glBegin(GL_TRIANGLES);
glVertex3f(-0.1f, -0.2 * sqrt(3) / 6, 0);
glVertex3d(0.1f, -0.2 * sqrt(3) / 6, 0);
glVertex3d(0.0f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);

glVertex3d(-0.1f, -0.2 * sqrt(3) / 6, 0);
glVertex3d(0.0f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);
glVertex3d(-0.2f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);

glVertex3d(0.0f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);
glVertex3d(-0.2f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);
glVertex3d(0.1 * 27 / 52, -0.2 * 147 / 113 / 146 * 230 - 0.2 * sqrt(3) / 6, 0);
glEnd();
```

可以看到在红扇叶（作为初始位置）前，是没有Rotate的，也就是红扇叶没有得到120°的旋转。而在蓝扇叶和绿扇叶的绘制点代码前都有Rotate操作。但由于绿扇叶在蓝扇叶的代码段之后，所以它得到了两次Rotate操作。与此相对的是蓝扇叶之前只有一次Rotate，所以它只会旋转一次。

效果如下：



风车把和中间那个黄色三角形就没什么技术含量了，把参数卡好就能画出来，不再赘述。

风车的转动：矩阵管理工具glPush/PopMatrix与守护线程IdleFunc

现在静态的风车已经渲染完了，该让它转动起来了。怎么做？

假设当前System Tick = t，且红扇叶与初始位置的夹角为α。假设经过ΔT时间后，红扇叶转动到与t时间点时的夹角为β的位置。也就是说红扇叶最终在(t+ΔT)时与初始位置的夹角为(α+β)。所以我们只需要对红扇叶施加一个逆时针(α+β)的旋转操作就可以了，对蓝扇叶和绿扇叶来说也是这样的。所以在渲染时加上这个操作就可以了。

所以最终的目标，就落到了计算β上。（别忘了α也是由β累加出来的旋转角！）

然而，实验要求中并没有说这个风车的旋转速度是多少，也就是旋转速度s完全是自己设定的。这样就好说了，我们预设一个旋转速度s，每一个时隙让旋转角加上这个速度就可以了。

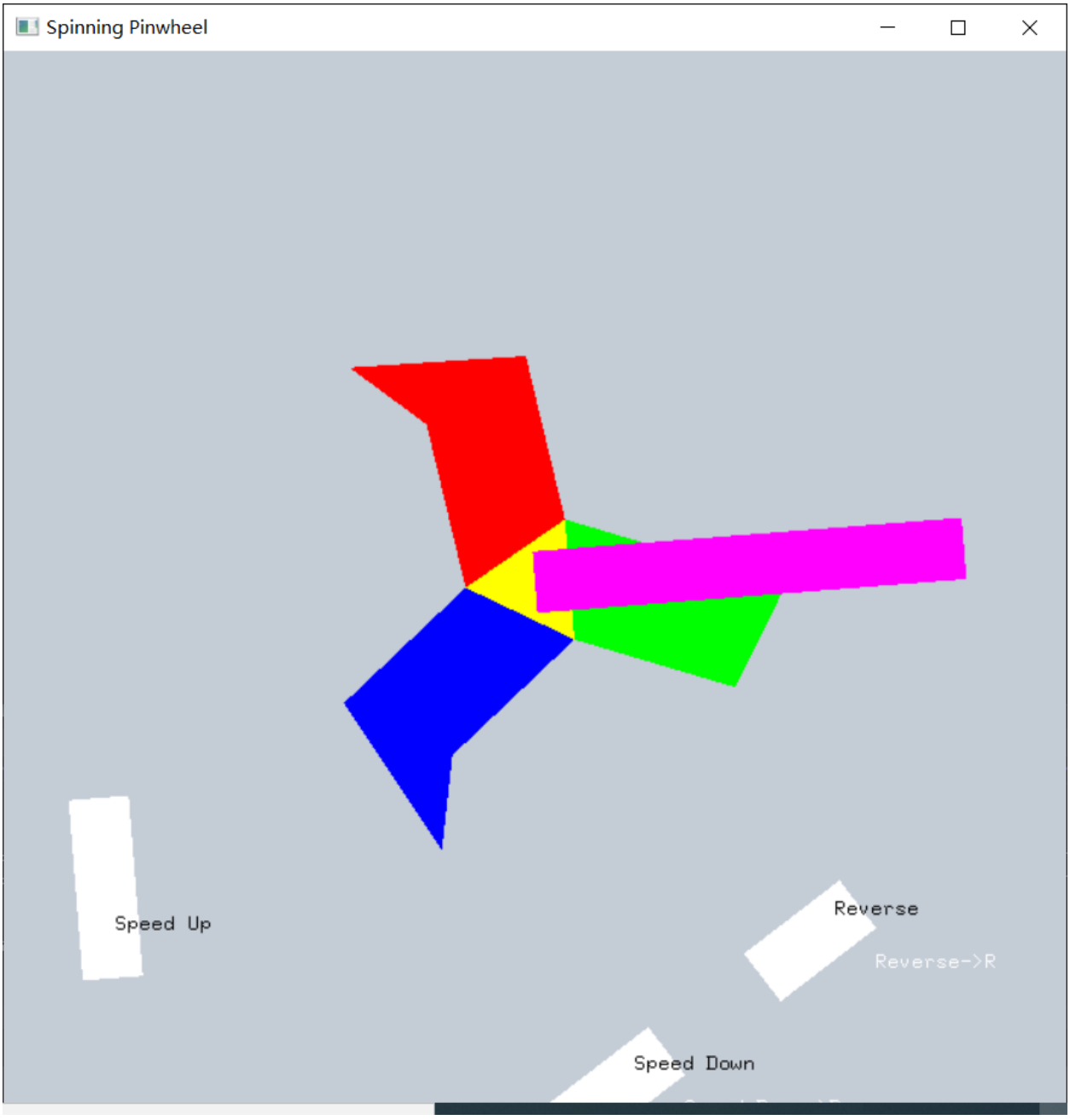
```
double angle = 0;
double speed = 0.5;

void rotate()
{
    angle += speed;
    if (angle > 360) angle = 0;
    glutPostRedisplay();
}
```

angle即下一个要渲染的帧对应的(α+β)的旋转角。

在渲染时，加上逆时针旋转angle角度的代码即可。

```
glRotatef(angle, 0, 0, 1);
```

但需要注意的一点是，风车扇叶和风车中心是旋转的，但风车的柄不是，如果我们在一开头就加入了这个Rotate操作，后面的风车柄就也跟着转了，如下图。



这个问题怎么解决呢？这涉及OpenGL中矩阵堆栈的概念。

首先矩阵分为很多种，平移、旋转、投影、裁剪等等...将一个矩阵作用于一个向量后，就可以得到另外一个向量。而这样的操作是可以链式调用的。

例如：

```
Translate Matrix A;
Rotate Matrix B;
```

```
vector x;
vector y = ABx;
```

最终，y表示向量x先经过B的旋转操作，再经过A的平移操作得到的向量。

Trans A
Rot B

用栈的数据结构来存储，栈底的矩阵将更靠近被作用于的向量x。因此B先作用于x，随后才是A。

现在问题来了，如果我还有另外一个向量p，我只想让B作用于它，不想平移在此操作之后得到的向量，该怎么办？

```
Translate Matrix A;
Rotate Matrix B;
vector x;
vector y = ABx;

Rotate Matrix B;
vector p;
vector q = Bp;
```

上面是最笨的方式。难道要重新定义一个一模一样的旋转矩阵B吗？难道不能复用B吗。

我们可以这么想，既然在AB对x进行操作之后，我们不想要A了。那么只需要把A扔了就可以了，保留B。在数据结构中，我们可以简单的Pop矩阵堆栈。这样A就被扔了，随后我们再把B作用于Y即可。

用代码表示即：

```
Matrix Stack mst;
Translate Matrix A;
Rotate Matrix B;
vector x,p,y,q;

mst.push(B).push(A);
y = mst.matrixChain(x);//AB作用于x
mst.pop();//把A扔了
q = mst.matrixChain(p);//B作用于p
```

在OpenGL中,glPushMatrix将把当前矩阵复制一份存到堆栈中去（注意是复制一份，也就是说这个矩阵依然是生效的）。在push之后的任何变换都不会影响之前的矩阵。

而glPopMatrix将弹出栈顶的矩阵并将其更新为当前矩阵。他之后的任何变换都不会影响之前的矩阵。

上述代码用OpenGL来写即：

```
//假设之前有一些矩阵操作,但栈是空的
glRotatef(a,x,y,z); //B
glpushMatrix(); //stack : B
glTranslatef(x,y,z); //A

//绘制x
// 当前矩阵：A，栈中B
//ABx

glPopMatrix();
//栈：null
//当前矩阵：B

//绘制p
//Bp
```

回到风车的题目中，我们的需求是旋转angle的操作只对三个扇叶作用，而对风车柄无效，那么应该将Rotate(angle)和三个扇叶的绘制放到push/pop对中，而风车柄在其外。代码如下：

```
glPushMatrix();
glRotatef(angle, 0, 0, 1);
//扇叶的绘制
//...
glPopMatrix(); //把Rotate操作拿出来，不要作用于后面的变换
//风车柄的绘制
//...
```

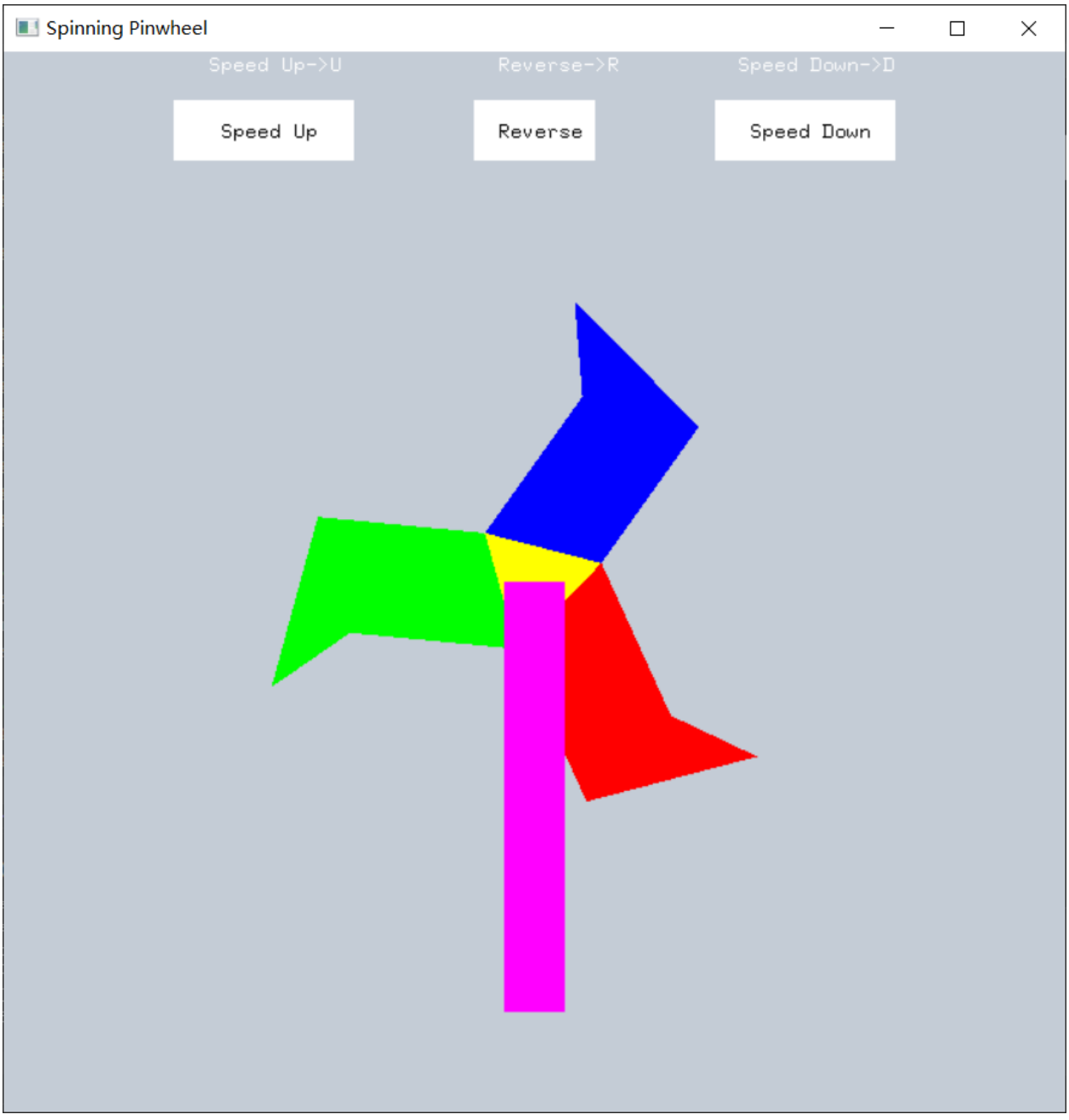
现在风车柄就不会跟随扇叶旋转了。

还有一个小问题，每一帧扇叶的位置都和上一帧不同（假设风车确实在旋转），因此每一帧我们都要将这个区别于上一帧的旋转操作渲染上去，也就是说我们想要一个比较trivial的循环，这个循环只是负责渲染旋转的过程，这应该怎么实现呢？

OpenGl提供了一个工具函数，glIdleFunc。当没有窗口事件到达时，这个idle function会被不断调用，直到有窗口事件发生。它没有参数，当前的窗口和菜单在执行idle func之前不会改变。

因此我们只需要将旋转函数作为这个idle function，在没有任何中断事件时，就会不停地更新本帧的旋转角了。

```
//这里是主线程
//... 一些前置代码
glIdleFunc(&rotate);//守护线程的行为是更新旋转
//... 一些其他代码
```



现在风车的行为终于正常了。

键鼠交互控制：EventListener与回调函数

需求中要求我们实现键鼠控制风车旋转方向反置，以及风车旋转速度提升或下降的功能。这涉及到程序与键鼠的交互控制。在实现这一功能前，我们先看看如何来编写这段逻辑。

首先是方向反置。这个比较容易，我们可以在之前设定的旋转速度s前加上一个前缀direction。当direction=1时，direction×s = s。也就是原先的逆时针方向。当direction=-1时，direction×s = -s，变成了顺时针方向。

```
int direction = 1;
//反置方向
void reverse()
{
    direction = -direction;
}
//旋转时速度乘以方向前缀
void rotate()
{
    angle += direction*speed;
    if (angle > 360) angle = 0;
    glutPostRedisplay();
}
```

随后是旋转速度提升或下降。由于s是一个变量，因此我们可以对它进行任意增量。也就是说交互事件到来时，我们给s加上或减去一个增量即可。

```
double Acceleration = 0.25;
//速度提升
void speedUp()
{
    speed+=Acceleration;
}
//速度下降
void speedDown()
{

    if (speed >= Acceleration) {
        speed -= Acceleration;
    }
    else {
        //注意s是一个非负值
        speed = 0;
    }
```

```
}
```

现在逻辑上的问题解决了，但怎么让交互事件到来时程序调用这几段逻辑呢？OpenGL提供了两个函数，用于监听事件的到来。

glutKeyboardFunc

用于设置处理键盘输入的回调函数。这个回调函数需要有三个参数：

1. 一个表示按键的ASCII码；
2. 一个表示按键时鼠标的x坐标；
3. 一个表示按键时鼠标的y坐标。

例如

```
void processNormalKeys(unsigned char key, int x, int y) {
    if (key == 27) {
        exit(0);
    }
}
```

可以使用gluKeyboardFunc来注册这个回调函数，例如：

```
gluKeyboardFunc(&processNormalKeys);
```

glutMouseFunc

用于设置处理鼠标点击事件的回调函数。

这个回调函数有四个参数：

1. 一个表示被按下或松开的鼠标键（可以是GLUT_LEFT_BUTTON，GLUT_MIDDLE_BUTTON或者GLUT_RIGHT_BUTTON）；
2. 一个表示鼠标的状态（按下或者松开，可以是GLUT_DOWN或者GLUT_UP）；
3. 鼠标点击的x坐标；
4. 鼠标点击的y坐标。

例如

```
void processMouseClick(int button, int state, int x, int y) {
    // 处理鼠标点击事件的代码
}
```

可以使用glutMouseFunc来注册这个回调函数

```
glutMouseFunc(&processMouseClick);
```

因此我们只需要通过驱动表将键鼠事件对应的逻辑置于一个回调函数中，然后将回调函数传给监听器就可以了。

```
//以监听键盘为例
void keyBoard(unsigned char key, int x, int y)
{
    switch (key)
    {
    case 'r':reverse(); break;
    case 'u':speedUp(); break;
    case 'd':speedDown(); break;
    case 27:exit(0);
    }
}
//主线程中监听
//... 一些前置代码
glutKeyboardUpFunc(&keyBoard);//监听键盘事件
//... 其他代码
```

鼠标与按钮的交互逻辑上跟这个是一样的，不再赘述。

至此我们完成了Lab1。

完整代码如下：

```
#include <iostream>
#define GLEW_STATIC
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <string>
#include <cmath>
#include <Windows.h>
#include <Gl/glut.h>
#define MAX_CHAR 128
```

```
#pragma comment( linker, "/subsystem:\"windows\" /entry:\"mainCRTStartup\"" )
using namespace std;

double angle = 0;
int direction = 1;
double speed = 0.5;
double Acceleration = 0.25;


void reverse()
{
    direction = -direction;
}

void speedUp()
{
    speed+=Acceleration;
}

void speedDown()
{

    if (speed >= Acceleration) {
        speed -= Acceleration;
    }
    else {
        speed = 0;
    }

}

void rotate()
{
    angle += direction*speed;
    if (angle > 360) angle = 0;
    glutPostRedisplay();
}

void printCharater(const char* str,float labelX,float labelY)
{
    glRasterPos2f(labelX, labelY);
    for (int i = 0; i < strlen(str); i++) glutBitmapCharacter(GLUT_BITMAP_8_BY_13, *(str + i));
}

struct button
{
    char* str = new char[100];
    bool cleck;
    void showButton(int r, int g, int b, const char* b_label,float labelX,float labelY,
        float top, float bottom, float left, float right) {

        glPushMatrix();
```

```
        if (cleck)
        {
            glScalef(0.98, 0.98, 1.0);
        }
        glColor3f(r, g, b);
        printCharater(b_label,labelX,labelY);
        glColor3f(1, 1, 1);
        glBegin(GL_QUADS);
        glVertex3f(left, top, 0);
        glVertex3f(right, top, 0);
        glVertex3f(right, bottom, 0);
        glVertex3f(left, bottom, 0);

        glEnd();


        glPopMatrix();
    }


};
button *reverseBtn,*speedUpBtn,*speedDownBtn;


void mouseFunc(GLint btn, GLint sta, int x, int y)
{
    int size = 700;
    int rxl = size/2-size*0.1, rxr=size/2+size*0.1, yb=size*0.05, yt=size*0.1,
        suxl=size*0.2,suxr=size*0.35,
        sdxl=size*0.65,sdxr=size*0.8;


    if (btn == GLUT_LEFT_BUTTON)
        switch (sta)
        {
        case GLUT_DOWN:
        {
            if (y >= yb && y <= yt) {
                if (x >= rxl && x <= rxr) {
                    reverse();
                    reverseBtn->cleck = true;
                }
                else if (x >= suxl && x <= suxr) {
                    speedUp();
                    speedUpBtn->cleck = true;
                }
                else if (x >= sdxl && x <= sdxr) {
                    speedDown();
                    speedDownBtn->cleck = true;
                }
            }
            break;
        }
```

```
        case GLUT_UP:
        {
            reverseBtn->cleck = false;
            speedUpBtn->cleck = false;
            speedDownBtn->cleck = false;
        }
        }
    glutPostRedisplay();
}

void keyBoard(unsigned char key, int x, int y)
{
    switch (key)
    {
    case 'r':reverse(); break;
    case 'u':speedUp(); break;
    case 'd':speedDown(); break;
    case 27:exit(0);
    }
}

void drawPinwheel()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glPushMatrix();
    glRotatef(angle, 0, 0, 1);
    glScalef(1, 1, 1);
    //中轴
    glColor3f(1, 1, 0);
    glBegin(GL_TRIANGLES);
    glVertex3f(-0.1f, -0.2 * sqrt(3) / 6, 0);
    glVertex3d(0.1f, -0.2 * sqrt(3) / 6, 0);
    glVertex3d(0.0f, 0.1 * sqrt(3) - 0.2 * sqrt(3) / 6, 0);
    glEnd();

    //红扇叶
    glColor3f(1, 0, 0);
    glBegin(GL_TRIANGLES);
    glVertex3f(-0.1f, -0.2 * sqrt(3) / 6, 0);
    glVertex3d(0.1f, -0.2 * sqrt(3) / 6, 0);
    glVertex3d(0.0f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);

    glVertex3d(-0.1f, -0.2 * sqrt(3) / 6, 0);
    glVertex3d(0.0f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);
    glVertex3d(-0.2f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);

    glVertex3d(0.0f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);
    glVertex3d(-0.2f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);
    glVertex3d(0.1 * 27 / 52, -0.2 * 147 / 113 / 146 * 230 - 0.2 * sqrt(3) / 6, 0);
    glEnd();

    //蓝扇叶
```

```
    glRotatef(120, 0, 0, 1);
    glColor3f(0, 0, 1);
    glBegin(GL_TRIANGLES);
    glVertex3f(-0.1f, -0.2 * sqrt(3) / 6, 0);
    glVertex3d(0.1f, -0.2 * sqrt(3) / 6, 0);
    glVertex3d(0.0f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);

    glVertex3d(-0.1f, -0.2 * sqrt(3) / 6, 0);
    glVertex3d(0.0f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);
    glVertex3d(-0.2f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);

    glVertex3d(0.0f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);
    glVertex3d(-0.2f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);
    glVertex3d(0.1 * 27 / 52, -0.2 * 147 / 113 / 146 * 230 - 0.2 * sqrt(3) / 6, 0);
    glEnd();

    //绿扇叶
    glRotatef(120, 0, 0, 1);
    glColor3f(0, 1, 0);
    glBegin(GL_TRIANGLES);
    glVertex3f(-0.1f, -0.2 * sqrt(3) / 6, 0);
    glVertex3d(0.1f, -0.2 * sqrt(3) / 6, 0);
    glVertex3d(0.0f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);

    glVertex3d(-0.1f, -0.2 * sqrt(3) / 6, 0);
    glVertex3d(0.0f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);
    glVertex3d(-0.2f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);

    glVertex3d(0.0f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);
    glVertex3d(-0.2f, -0.2 * 147 / 113 - 0.2 * sqrt(3) / 6, 0);
    glVertex3d(0.1 * 27 / 52, -0.2 * 147 / 113 / 146 * 230 - 0.2 * sqrt(3) / 6, 0);
    glEnd();

    glPopMatrix();

    //风车把
    glScalef(1, 1, 1);
    glColor3f(1, 0, 1);
    glBegin(GL_POLYGON);
    glVertex3f(0.05, 0, 0.1);
    glVertex3f(-0.05, 0, 0.1);
    glVertex3f(-0.05, -0.7, 0.1);
    glVertex3f(0.05, -0.7, 0.1);
    glEnd();

    //按钮
    speedUpBtn->showButton(0,0,0,"Speed Up", -0.52,0.74,0.8, 0.7, -0.6, -0.3);
    reverseBtn->showButton(0, 0, 0, "Reverse",-0.06,0.74, 0.8, 0.7, -0.1, 0.1);
    speedDownBtn->showButton(0, 0, 0, "Speed Down", 0.36,0.74,0.8, 0.7, 0.3, 0.6);

    //按键提示
    printCharater("Speed Up->U", -0.54, 0.85);
    printCharater("Reverse->R", -0.06, 0.85);
```

```
    printCharater("Speed Down->D", 0.34, 0.85);


    glFlush();
    glutSwapBuffers();
}

void Init()
{
    glEnable(GL_DEPTH_TEST);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glClearColor(0.77, 0.80, 0.84, 0.19);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(20, 1, 1, 50);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0, 0, 5, 0, 0, 0, 0, 1, 0);
    glShadeModel(GL_SMOOTH);
    reverseBtn = new button;
    reverseBtn->cleck = false;
    speedUpBtn = new button;
    speedUpBtn->cleck = false;
    speedDownBtn = new button;
    speedDownBtn->cleck = false;
}

int main(int argc, char* argv[])
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(700, 700);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Spinning Pinwheel");
    glEnable(GL_DEPTH_TEST | GL_LINE_SMOOTH | GL_POLYGON_SMOOTH | GL_POLYGON_STIPPLE);

    Init();
    glutDisplayFunc(&drawPinwheel);
    glutIdleFunc(&rotate);
    glutKeyboardUpFunc(&keyBoard);
    glutMouseFunc(&mouseFunc);
    glutMainLoop();

    return 0;
}
```