作业报告

院系：数据科学与计算机学院　　专业：计算机类　　科目：计算机图形学
姓名：郑康泽　　学号：17341213

一. 作业题目

在作业模板中的 scene_0()函数使用了 OpenGL 函数（如，glBegin(GL_LINE_STRIP)，glTranslatef()及 glRotatef()等）绘制一个三角形及一个四边形的场景。同学们需要在不使用以上函数的情况下重现同样场景（在模板中的 scene_1()函数汇总实现）。

二. 作业要求

1. 实现平移 Translation 及旋转 Rotation 变换

实现平移及旋转函数，计算点经过平移、旋转等变换后，于屏幕中所处的位置。

- 不能使用 glTranslate()， glRotate()， glMultMatrix()等函数。
- 可选：通过矩阵乘法及四元数两种方式实现旋转。

2. 实现直线绘制算法

使用 OpenGL 实现你的直线光栅化算法。

- 输入：两个二维平面上的点（由 1 中计算得到）。
- 输出：在屏幕上输出一条直线。
- 只能使用整数运算，并且只能使用 GL_POINTS 作为基本元素。
- 可选：使用抗锯齿算法或机制对绘制直线进行平滑。

3. 其他注意事项

- 在报告中写下主要算法，说明多个变换之间的顺序关系对结果的影响，并附上运行结果截图。
- scene_1()函数中已经提供一个基本模板，用于对每个坐标对应的像素点着色。
- scene_0()及 scene_1()的绘制可通过键盘的"0", "1"按键进行切换。
- 在窗口 resize 过程中，需要保持 scene_1()绘制结果与 scene_0() 一致。

三. 主要算法及代码

1. 通过矩阵乘法实现平移 Translation 及旋转 Rotation 变换

1) 算法描述如下：

在 OpenGL 中点的坐标是 4 维齐次坐标：$\mathbf{P} = [x, y, z, w]^T$

a) 假设将点平移(d_x, d_y, d_z)个坐标，其变换矩阵可以直接得到，如下：

$$\begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

b) 假设绕轴(a, b, c)旋转 θ 度。首先我们需要通过以 x 为轴旋转 α 度，将旋转轴置于 xOy 平面，然后以 y 为轴旋转 β 度，将旋转轴与 z 轴重合，，然后以 z 轴旋转 θ 度，最后再将坐标系恢复回来，即以 y 为轴旋转-β 度，以 x 为轴旋转-α 度。

最终的变换矩阵为：

$$R_V(\theta) = R_x(-\alpha)R_y(-\beta)R_z(\theta)R_y(\beta)R_x(\alpha) =$$

$$\begin{bmatrix} a^2(1-\cos\theta)+\cos\theta & ab(1-\cos\theta)-c\cdot\sin\theta & ac(1-\cos\theta)+b\cdot\sin\theta \\ ab(1-\cos\theta)+c\cdot\sin\theta & b^2(1-\cos\theta)+\cos\theta & bc(1-\cos\theta)-a\cdot\sin\theta \\ ac(1-\cos\theta)-b\cdot\sin\theta & bc(1-\cos\theta)+a\cdot\sin\theta & c^2(1-\cos\theta)+\cos\theta \end{bmatrix}$$

扩展到齐次坐标下，矩阵如下表示：

$$\begin{bmatrix} a^2(1-\cos\theta)+\cos\theta & ab(1-\cos\theta)-c\cdot\sin\theta & ac(1-\cos\theta)+b\cdot\sin\theta & 0 \\ ab(1-\cos\theta)+c\cdot\sin\theta & b^2(1-\cos\theta)+\cos\theta & bc(1-\cos\theta)-a\cdot\sin\theta & 0 \\ ac(1-\cos\theta)-b\cdot\sin\theta & bc(1-\cos\theta)+a\cdot\sin\theta & c^2(1-\cos\theta)+\cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2) 代码展示：

因为点在变换后，坐标中的 w 依旧是 1，所以为了简单表示矩阵，我将最后一行去掉，即所有变化矩阵都是 3x4。

a) 矩阵乘法函数：

```
/*
函数：matrix_mul
函数描述：将3x4的矩阵乘上points中每个4维的点，并更新每个点的xyz坐标
参数描述：points ——— 包含n个4维的点
         matrix ——— 一个3x4的变换矩阵
*/
void matrix_mul(vector<vector<float>>& points, float matrix[3][4])
{
    unsigned long long l = (points.size());
    for (unsigned long long i=0; i<l; ++i)
    {
        float xyz[3] = {0.0};
        for (unsigned long long j=0; j<3; ++j)
        {
            for (unsigned long long k=0; k<4; ++k)
                xyz[j] += matrix[j][k] * points[i][k];
        }
        for (unsigned long long j=0; j<3; ++j)
            points[i][j] = xyz[j];
    }
}
```

b) 平移变换函数：

```
/*
##################################################################
##  函数：translate
##  函数描述：定义平移变换矩阵，并调用矩阵乘法函数更新点坐标
##  ##############################################################
##  参数描述：
##  points ——— 包含n个4维的点
##  mx ——— x方向平移的单位
##  my ——— y方向平移的单位
##  mz ——— z方向平移的单位
##################################################################
*/
void translate(vector<vector<float>>& points, float mx, float my, float mz)
{
    float matrix[3][4] = {{1, 0, 0, mx}, {0, 1, 0, my}, {0, 0, 1, mz}};
    matrix_mul(points, matrix);
}
```

c) 旋转变换函数：

```
/*
################################################################
##  函数：rotate
##  函数描述：定义旋转变换矩阵，并调用矩阵乘法函数更新点坐标
##  ############################################################
##  参数描述：
##  points ——— 包含n个4维的点
##  angle ——— 旋转角度
##  (a, b, c) ——— 旋转轴的向量
################################################################
*/
void rotate(vector<vector<float>>& points, float angle, float a, float b, float c)
{
    // 单位向量！！！
    float length = float(sqrt(a*a+b*b+c*c));
    a /= length;
    b /= length;
    c /= length;
    float change = float(acos(-1)) / 180.0f;
    float sin_val = float(sin(angle * change));
    float cos_val = float(cos(angle * change));
    float matrix[3][4] = {{a*a*(1-cos_val)+cos_val, a*b*(1-cos_val)-c*sin_val, a*c*(1-cos_val)+b*sin_val, 0},
                          {a*b*(1-cos_val)+c*sin_val, b*b*(1-cos_val)+cos_val, b*c*(1-cos_val)-a*sin_val, 0},
                          {a*c*(1-cos_val)-b*sin_val, b*c*(1-cos_val)+a*sin_val, c*c*(1-cos_val)+cos_val, 0}};
    matrix_mul(points, matrix);
}
```

2. 通过四元数实现旋转 Rotation 变换

1）算法描述：

令$\boldsymbol{Q}=[\boldsymbol{n}\sin\theta,\cos\theta]$，可以证明$\boldsymbol{QvQ}^*$得到的是$\boldsymbol{v}$绕$\boldsymbol{n}$旋转 2 θ之后的向量（这里就不证明了）。

四元数的乘法如下：

$$\boldsymbol{Q}_a\boldsymbol{Q}_b=[\boldsymbol{v}_a,w_a][\boldsymbol{v}_b,w_b]=[\boldsymbol{v}_a\times\boldsymbol{v}_b+w_a\boldsymbol{v}_b+w_b\boldsymbol{v}_a,w_aw_b-\boldsymbol{v}_a\cdot\boldsymbol{v}_b]$$

注意点表示为四元数时，w应该为 0。

2）代码展示：

```
/*
################################################################
##  函数：rotate_2
##  函数描述：定义旋转变换矩阵，并调用矩阵乘法函数更新点坐标
##  ############################################################
##  参数描述：
##  points ——— 包含n个4维的点
##  angle ——— 旋转角度
##  (a, b, c) ——— 旋转轴的向量
################################################################
*/
void rotate_2(vector<vector<float>>& points, float angle, float a, float b, float c)
{
    // 单位向量
    float length = float(sqrt(a*a+b*b+c*c));
    a /= length;
    b /= length;
    c /= length;
```

```
    float change = float(acos(-1)) / 180.0f;
    float sin_val = float(sin(angle / 2.0f * change));
    float cos_val = float(cos(angle / 2.0f * change));
    // 定义Q和Q的逆
    vector<float> Q = {a*sin_val, b*sin_val, c*sin_val, cos_val};
    vector<float> inv_Q = {-a*sin_val, -b*sin_val, -c*sin_val, cos_val};
    unsigned long long l = points.size();
    for (unsigned long long i=0; i<l; ++i)
    {
        // 将点表示为四元数
        vector<float> point(points[i]);
        point[3] = 0;
        // Qv
        vector<float> res = {0.0f, 0.0f, 0.0f, 0.0f};
        res[0] = Q[1]*point[2]-Q[2]*point[1]+Q[3]*point[0]+Q[0]*point[3];
        res[1] = Q[2]*point[0]-Q[0]*point[2]+Q[3]*point[1]+Q[1]*point[3];
        res[2] = Q[0]*point[1]-Q[1]*point[0]+Q[3]*point[2]+Q[2]*point[3];
        res[3] = Q[3]*point[3] - (Q[0]*point[0] + Q[1]*point[1] + Q[2]*point[2]);
        // Qvinv(Q)
        points[i][0] = res[1]*inv_Q[2]-res[2]*inv_Q[1]+res[3]*inv_Q[0]+res[0]*inv_Q[3];
        points[i][1] = res[2]*inv_Q[0]-res[0]*inv_Q[2]+res[3]*inv_Q[1]+res[1]*inv_Q[3];
        points[i][2] = res[0]*inv_Q[1]-res[1]*inv_Q[0]+res[3]*inv_Q[2]+res[2]*inv_Q[3];
    }
}
```

3. 实现直线光栅化算法
   1. 算法描述：

      课件上给出了斜率在[0, 1]之间的递推公式，即

$$p_0 = 2\Delta y - \Delta x$$

$$p_{i+1} = \begin{cases} p_i + 2\Delta y, & p_i < 0 \\ p_i + 2\Delta y - 2\Delta x, & p_i \geq 0 \end{cases},$$

      如果斜率在[-1, 0]之间，经过计算，递推公式为：

$$p_0 = 2\Delta y + \Delta x$$

$$p_{i+1} = \begin{cases} p_i + 2\Delta y + 2\Delta x, & p_i < 0 \\ p_i + 2\Delta y, & p_i \geq 0 \end{cases}。$$

      如果斜率的绝对值大于 1，那么可以通过将 x 轴与 y 轴交换而得到以上的两种情况中的一个。

      如果直线与 x 轴平行，或者与 y 轴平行，直接画线即可。

   2. 代码展示：

      a) 判断斜率的函数，如果是平行坐标轴的线就直接画，否则调用光栅化函数画。（由于没有保存未加抗锯齿的版本，所以这里展示的是加了抗锯齿之后的）

```
/*
################################################################
##  函数：draw_line
##  函数描述：使用光栅化算法画一条直线
##  ############################################################
##  参数描述：
##  (x1, y1) ——— 第一个点的坐标
##  (x2, y2) ——— 第二个点的坐标
##  color ——— 画线的颜色
################################################################
*/
void drawline(int x1, int y1, int x2, int y2, vector<float> color)
{
    // 平行于y轴
    if (x1 == x2)
    {
        if (y1 > y2)
            myswap(y1, y2);
        for (int i=y1; i<=y2; ++i)
            glVertex2i(x1, i);
    }
```

```
    // 平行于x轴
    else if (y1 == y2)
    {
        if (x1 > x2)
            myswap(x1, x2);
        for (int i=x1; i<=x2; ++i)
            glVertex2i(i, y1);
    }
    else
    {
        // 坐标x2相当于在分辨率x2的图上，方便之后抗锯齿的超采样
        int n_x1 = x1*2;
        int n_y1 = y1*2;
        int n_x2 = x2*2;
        int n_y2 = y2*2;
        // 记录在分辨率x2的图上采用光栅化算法后的坐标
        vector<int> line_x;
        vector<int> line_y;

        int dx = n_x2 - n_x1;
        int dy = n_y2 - n_y1;
        //double k = double(dy) / double(dx);
        // 斜率绝对值在0到1之间
```

```
        // 斜率绝对值在0到1之间
        if (abs(dy) <= abs(dx))
        {
            // 使x1 < x2,统一从x1增加到x2
            if (dx < 0)
            {
                myswap(n_x1, n_x2);
                myswap(n_y1, n_y2);
            }
            line_x.push_back(n_x1);
            line_y.push_back(n_y1);
            // 斜率在0到1之间
            if (dy * dx > 0)
                rasterize(n_x1, n_y1, n_x2, n_y2, 0, line_x, line_y);
            // 斜率在-1到0之间
            else
                rasterize(n_x1, n_y1, n_x2, n_y2, 1, line_x, line_y);
            line_x.push_back(n_x2);
            line_y.push_back(n_y2);
            // 调用抗锯齿，以x方向自增
            anti_alias(line_x, line_y, color, 0);
        }
```

```
        // 斜率绝对值大于1
        else
        {
            // 交换坐标轴
            myswap(n_x1, n_y1);
            myswap(n_x2, n_y2);
            // 使y1 < y2,统一从y1增加到y2
            if (dy < 0)
            {
                myswap(n_x1, n_x2);
                myswap(n_y1, n_y2);
            }
            line_x.push_back(n_y1);
            line_y.push_back(n_x1);
            // 斜率大于1
            if (dy * dx > 0)
                rasterize(n_x1, n_y1, n_x2, n_y2, 2, line_x, line_y);
            // 斜率小于-1
            else
                rasterize(n_x1, n_y1, n_x2, n_y2, 3, line_x, line_y);
            line_x.push_back(n_y2);
            line_y.push_back(n_x2);
            // 调用抗锯齿，以y方向自增
            anti_alias(line_x, line_y, color, 1);
        }
    }
}
```

b）光栅化函数：

```
/*
################################################################
##  函数：rasterize
##  函数描述：使用光栅化算法算出最靠近线的整数坐标
##  ############################################################
##  参数描述：
##  (x1, y1) ——— 第一个点的坐标
##  (x2, y2) ——— 第二个点的坐标
##  mode ——— 0代表斜率在0到1之间，1代表斜率在-1到0之间，2代表斜率大于1，3代表斜率小于-1
##  line_x ——— 记录整数坐标的x坐标
##  line_y ——— 记录整数坐标的y坐标
################################################################
*/
void rasterize(int x1, int y1, int x2, int y2, int mode, vector<int>& line_x,
               vector<int>& line_y)
{
    int dx = x2 - x1;
    int dy = y2 - y1;
    int y = y1;
    int p;
    // 斜率在0到1之间
    if (mode == 0 || mode == 2)
        p = 2 * dy - dx;
    // 斜率在-1到0之间
    else
        p = 2 * dy + dx;
```

```
for (int i=x1+1; i<x2; ++i)
{
    // 取lower点
    if (p < 0)
    {
        if (mode == 0 || mode == 2)
        {
            p += 2 * dy;
            if (mode == 0)
            {
                //glVertex2i(i, y);
                line_x.push_back(i);
                line_y.push_back(y);
            }
            // x轴与y轴交换回来
            else
            {
                //glVertex2i(y, i);
                line_x.push_back(y);
                line_y.push_back(i);
            }
        }
```

```
        else
        {
            p += 2 * dy + 2 * dx;
            y = y - 1;
            if (mode == 1)
            {
                //glVertex2i(i, y);
                line_x.push_back(i);
                line_y.push_back(y);
            }
            // x轴与y轴交换回来
            else
            {
                //glVertex2i(y, i);
                line_x.push_back(y);
                line_y.push_back(i);
            }
        }
    }
```

```
    else
    {
        if (mode == 0 || mode == 2)
        {
            p += 2 * dy - 2 * dx;
            y = y + 1;
            if (mode == 0)
            {
                //glVertex2i(i, y);
                line_x.push_back(i);
                line_y.push_back(y);
            }
            // x轴与y轴交换回来
            else
            {
                //glVertex2i(y, i);
                line_x.push_back(y);
                line_y.push_back(i);
            }
        }
```

```
            else
            {
                p += 2 * dy;
                if (mode == 1)
                {
                    //lVertex2i(i, y);
                    line_x.push_back(i);
                    line_y.push_back(y);
                }
                // x轴与y轴交换回来
                else
                {
                    //glVertex2i(y ,i);
                    line_x.push_back(y);
                    line_y.push_back(i);
                }
            }
        }
    }
}
```

4. 使用抗锯齿算法

a) 算法描述：

我用的超采样的方法，在原来分辨率 2 倍下重新采样，将 2x2 的像素块合并成一个像素。由上面光栅化后得到的点的坐标的列表，遍历每个点，判断该点于下个点是否在同个的 2x2 像素块，如果在，则合并后的像素块的颜色较深，否则较浅。

b) 代码展示：

```
/*
################################################################
##  函数：anti_alias
##  函数描述：利用重采样的方法抗锯齿
##  ############################################################
##  参数描述：
##  line_x ——— 光栅化后直线上的点的x轴坐标
##  line_y ——— 光栅化后直线上的点的y轴坐标
##  color ——— 画线的颜色
##  mode ——— mode为0表示以x轴自增，为1表示以y轴自增
################################################################
*/
void anti_alias(vector<int> line_x, vector<int> line_y, vector<float> color, int mode)
{
    // 调出一种较color更浅的颜色
    vector<float> n_color(color);
    unsigned long long max_pos = 0;
    for (unsigned long long i=1; i<3; ++i)
    {
        if (color[i] > color[max_pos])
        {
            max_pos = i;
            n_color[max_pos] *= 1.1f;
        }
        else
            n_color[i] *= 1.1f;
    }
```

```
    // 抗锯齿
    unsigned long long l = line_x.size();
    if (mode == 0)
    {
        for (unsigned long long i=0; i<l;)
        {
            // 当前的点位于奇数x轴坐标，不会与下个点在同个像素块
            if ((line_x[i] % 2 == 1) || (i == l-1))
            {
                glColor3f(n_color[0], n_color[1], n_color[2]);
                glVertex2i(line_x[i]/2, line_y[i]/2);
                ++i;
            }
            else
            {
                // 下个点与当前点在同个像素块
                if ((line_y[i]==line_y[i+1]) || ((line_y[i]-line_y[i+1]==1)&&(line_y[i]%2==1))
                        || ((line_y[i+1]-line_y[i]==1)&&(line_y[i]%2==0)))
                {
                    glColor3f(color[0], color[1], color[2]);
                    glVertex2i(line_x[i]/2, line_y[i]/2);
                    i += 2;
                }
                // 因为y轴坐标，下个点与当前点不在同个像素块
                else
                {
                    glColor3f(n_color[0], n_color[1], n_color[2]);
                    glVertex2i(line_x[i]/2, line_y[i]/2);
                    ++i;
                }
            }
        }
    }
```

```
    else
    {
        for (unsigned long long i=0; i<l;)
        {
            if ((line_y[i] % 2 == 1) || (i == l-1))
            {
                glColor3f(n_color[0], n_color[1], n_color[2]);
                glVertex2i(line_x[i]/2, line_y[i]/2);
                ++i;
            }
            else
            {
                if ((line_x[i]==line_x[i+1]) || ((line_x[i]-line_x[i+1]==1)&&(line_x[i]%2==1))
                        || ((line_x[i+1]-line_x[i]==1)&&(line_x[i]%2==0)))
                {
                    glColor3f(n_color[0], n_color[1], n_color[2]);
                    glVertex2i(line_x[i]/2, line_y[i]/2);
                    i += 2;
                }
                else
                {
                    glColor3f(n_color[0], n_color[1], n_color[2]);
                    glVertex2i(line_x[i]/2, line_y[i]/2);
                    ++i;
                }
            }
        }
    }
}
```

5. scene_1 函数的代码展示

```
void MyGLWidget::scene_1()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0, width(), 0.0, height(), -1000.0, 1000.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    // 定义点
    vector<vector<float>> triangle = {{10.0, 10.0, 0.0, 1.0},
                                      {50.0, 50.0, 0.0, 1.0},
                                      {80.0, 10.0, 0.0, 1.0}};
    // 定义颜色
    vector<float> color1 = {0.839f, 0.153f, 0.157f};
    // 平移
    translate(triangle, -50.0f, -30.0f, 0.0f);
    // 旋转
    //rotate(triangle, 45.0f, 1.0f, 0.0f, 1.0f);
    rotate_2(triangle, 45.0f, 1.0f, 0.0f, 1.0f);
    // 平移
    translate(triangle, -20.0f, -10.0f, 0.0f);
    translate(triangle, 50.0f, 50.0f, 0.0f);
    // 为了使点更密集，看的窗口为(width, height)，则将点坐标乘上相应倍数，等同于增加分辨率
    for (unsigned i=0; i<3; ++i)
    {
        triangle[i][0] *= width()/100.0f;
        triangle[i][1] *= height()/100.0f;
    }
```

```
    // 画线
    glBegin(GL_POINTS);
    //glColor3f(0.839f, 0.153f, 0.157f);
    drawline(int(round(triangle[0][0])), int(round(triangle[0][1])),
             int(round(triangle[1][0])), int(round(triangle[1][1])), color1);
    drawline(int(round(triangle[0][0])), int(round(triangle[0][1])),
             int(round(triangle[2][0])), int(round(triangle[2][1])), color1);
    drawline(int(round(triangle[1][0])), int(round(triangle[1][1])),
             int(round(triangle[2][0])), int(round(triangle[2][1])), color1);
    glEnd();

    vector<vector<float>> quad = {{-20.0, -20.0, 0.0, 1.0},
                                  {20.0, -20.0, 0.0, 1.0},
                                  {20.0, 20.0, 0.0, 1.0},
                                  {-20.0, 20.0, 0.0, 1.0}};
    vector<float> color2 = {0.122f, 0.467f, 0.706f};
    //rotate(quad, 30.0f, 1.0f, 1.0f, 1.0f);
    rotate_2(quad, 30.0f, 1.0f, 1.0f, 1.0f);
    translate(quad, 20.0f, 20.0f, 0.0f);
    translate(quad, 50.0f, 50.0f, 0.0f);
    for (unsigned i=0; i<4; ++i)
    {
        quad[i][0] *= width()/100.0f;
        quad[i][1] *= height()/100.0f;
    }
```

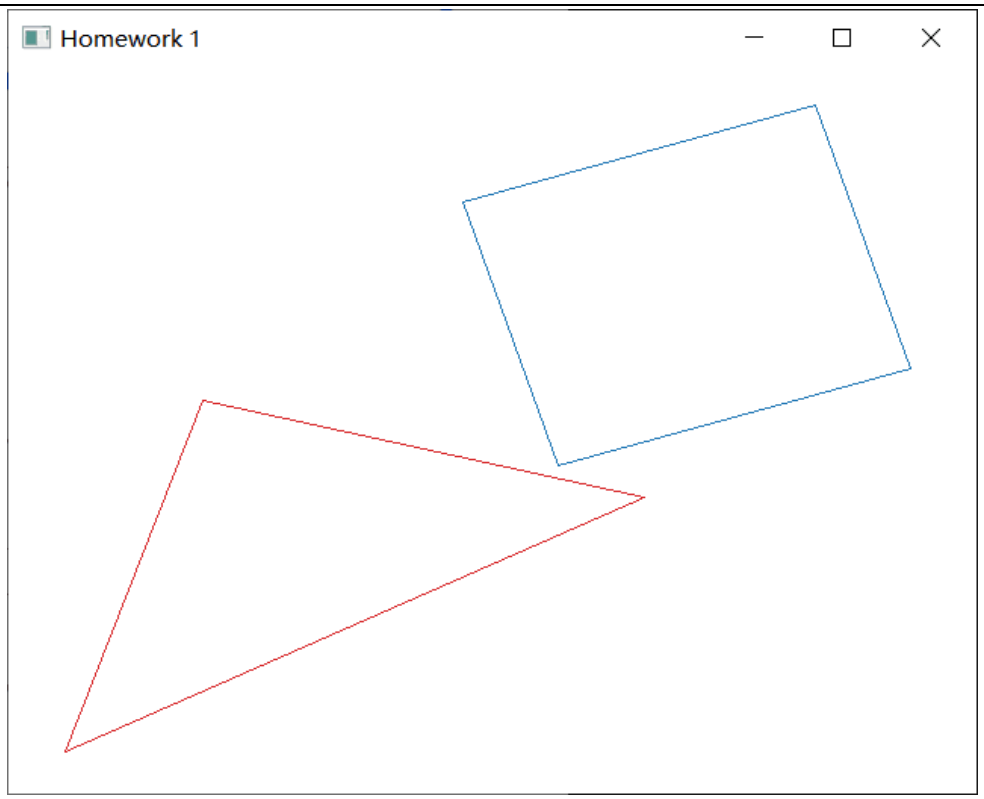
```
    glBegin(GL_POINTS);
    //glColor3f(0.122f, 0.467f, 0.706f);
    drawline(int(round(quad[0][0])), int(round(quad[0][1])),
             int(round(quad[1][0])), int(round(quad[1][1])), color2);
    drawline(int(round(quad[1][0])), int(round(quad[1][1])),
             int(round(quad[2][0])), int(round(quad[2][1])), color2);
    drawline(int(round(quad[2][0])), int(round(quad[2][1])),
             int(round(quad[3][0])), int(round(quad[3][1])), color2);
    drawline(int(round(quad[3][0])), int(round(quad[3][1])),
             int(round(quad[0][0])), int(round(quad[0][1])), color2);
    glEnd();
}
```

四. 运行结果截图和回答问题

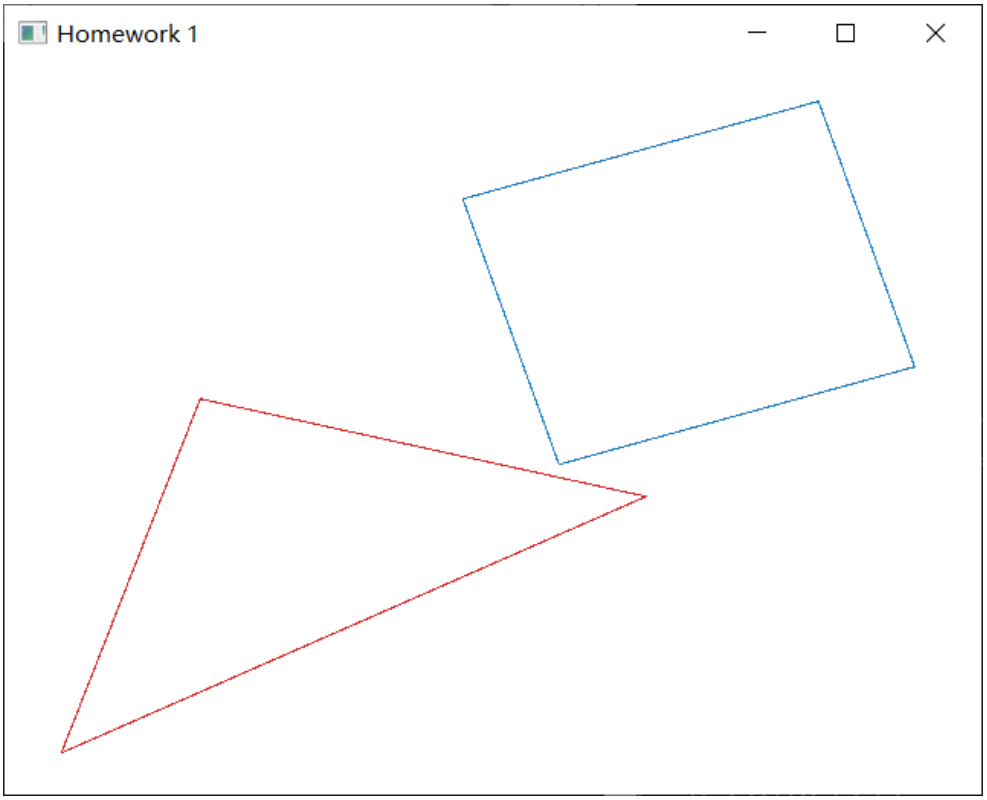
1. 抗锯齿与没有抗锯齿的比较

中山大學 SUN YAT-SEN UNIVERSITY

作业报告



无抗锯齿



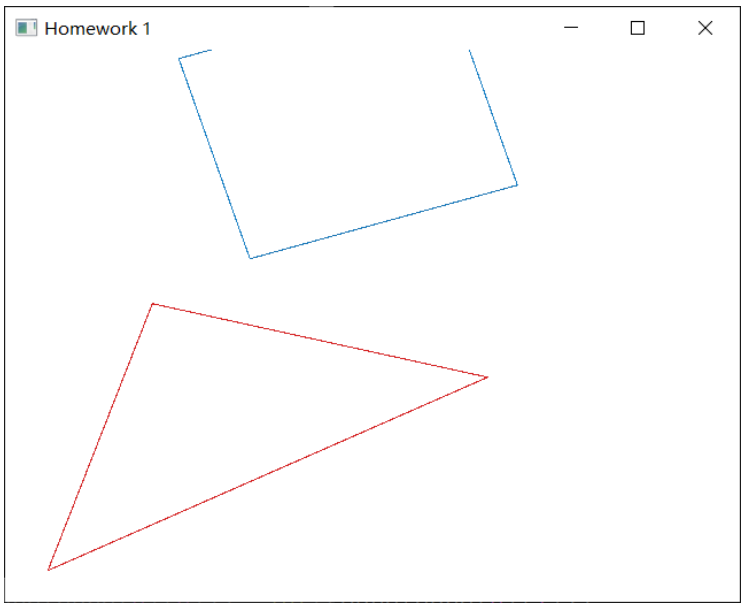
抗锯齿

感觉对于斜率绝对值较大的线条，抗锯齿的效果就比较明显，对于斜率绝对值较小的线条，效果就不怎么样了。

2. 问题：多个变换之间的顺序关系对结果的影响。

回答：就拿画方形来说，在 scene_0 中可以看到，它是先平移了(50, 50, 0)，再平移了(20, 20, 0)，最后绕着(1, 1, 1)为轴旋转 30 度，但是我们知道，在 OpenGL 中，这些变换都是倒着进行的，也就是说，先定义的动作慢做，最后定义的动作最先做。即在 scene_1 中，你要先绕着(1, 1, 1)为旋转 30 度，再平移(20, 20, 0)，最后平移(50, 50, 50)。

在 scene_1 中，如果你写的变换顺序与在 scene_0 中一样的话，你会得到下面的图（由于没有改三角形的变换顺序，所以三角形的位置是对的）：



而正确的图应该长这样：