Homework9-Bezier Curve

Basic:

1. 用户能通过左键点击添加Bezier曲线的控制点，右键点击则对当前添加的最后一个控制点进行消除。

   ① 设置鼠标捕获

   首先写出回调函数

   需要注意的是通过glfwGetCursorPos得到的y坐标和OpenGL绘制的y坐标是相反的（在屏幕上的表示关于中心对称）。用**glfwGetCursorPos**获得的y坐标**越靠近底部越大**，而**OpenGL**的y坐标是**越靠近底部越小**，所以要做一下变换：posY = (HEIGHT -posY)

```
void mouse_callback_bezier(GLFWwindow* window, int button, int action, int
mods) {

    if (button == GLFW_MOUSE_BUTTON_LEFT && action == GLFW_RELEASE) {
        double posX, posY;
        glfwGetCursorPos(window, &posX, &posY);
        cout << posY << endl;
        cout << "x: " << posX << " y: " << (HEIGHT -posY) << endl;  // y坐
标需要变换
        bezier_points.push_back(glm::vec3(posX, HEIGHT - posY, 0.0));
    }

    else if (button == GLFW_MOUSE_BUTTON_RIGHT && action == GLFW_RELEASE) {
        double posX, posY;
        glfwGetCursorPos(window, &posX, &posY);
        cout << "x: " << posX << " y: " << (HEIGHT - posY) << endl;

        for (vector<glm::vec3>::iterator it = bezier_points.begin(); it !=
bezier_points.end(); )
        {
            // 点在点周边的范围就可以消除，不用刚好点在点上
            // 遍历vector，删除点
            if (abs((*it).x - posX) <= 50 && abs((*it).y - (HEIGHT - posY))
<= 50) {
                it = bezier_points.erase(it);
            }
            else
            {
                ++it;
            }
        }
}
```

② 绘制控制点、连线

在上面的回调函数中，我们把每一个点击生成的控制点压入bezier_points里面，bezier_points里面存的坐标的形式是glm::vec3。我们需要生成一个float数组，然后与VBO绑定，之后渲染出每个点。

```
// point
auto vec1 = convertVec3ToFloat(bezier_points);
float* data1 = vec1.data();
size_t data_len1 = vec1.size() * sizeof(float);
int draw_size1 = vec1.size() / 3;


glBindVertexArray(VAO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, data_len1, data1, GL_STATIC_DRAW);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat),
(GLvoid*)0);
glEnableVertexAttribArray(0);
glPointSize(20);
glDrawArrays(GL_POINTS, 0, draw_size1);
```

其中convertVec3ToFloat就是用来把vector<glm::vec3>转成vector<float>

```
vector<float> convertVec3ToFloat(vector<glm::vec3> points) {
    vector<float> result;
    for (int i = 0; i < points.size(); i++) {
        result.push_back(points[i].x);
        result.push_back(points[i].y);
        result.push_back(points[i].z);
    }
    return result;
}
```

我们还需要把控制点连线。为了方便我直接使用glDrawArrays然后图元选择GL_LINES

```
if (draw_size1 > 1) { // 注意要draw_size1>1即有两个点才开始画边连线
    // line
    auto vec2 = convertVec3ToFloatLine(bezier_points);
    float* data2 = vec2.data();
    size_t data_len2 = vec2.size() * sizeof(float);
    int draw_size2 = vec2.size() / 3;
    glBufferData(GL_ARRAY_BUFFER, data_len2, data2, GL_STATIC_DRAW);
    glDrawArrays(GL_LINES, 0, draw_size2);
}
```

注意不能一开始就画线，不然会出现AssertionFail，vector subscript out of range。所以我加上一个判断条件，大于等于两个点才开始画直线。

其中convertVec3ToFloatLine也是用来把vector<glm::vec3>转成vector<float>，但里面储存的点的顺序是画直线所需要的顺序。(point1, point2, point2, point3会画出1->2, 2->3两条直线)

```
// 用于画线的
vector<float> convertVec3ToFloatLine(vector<glm::vec3> points) {
    vector<float> result;
    for (int i = 0; i < points.size()-1; i++) {
        result.push_back(points[i].x);
        result.push_back(points[i].y);
        result.push_back(points[i].z);
        result.push_back(points[i+1].x);
        result.push_back(points[i+1].y);
        result.push_back(points[i+1].z);
    }
    return result;
}
```

2. 工具根据鼠标绘制的控制点实时更新Bezier曲线。

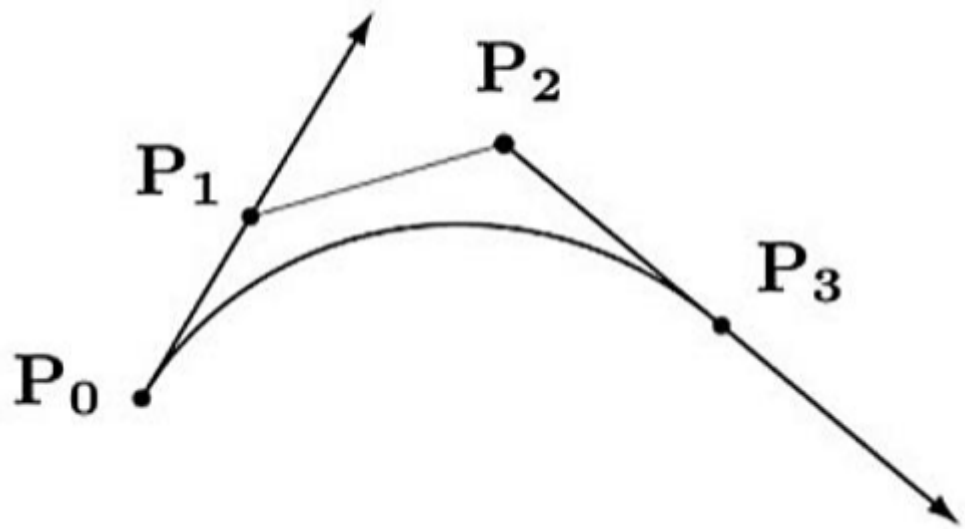
   这一部分其实就只需要在t ∈ [0, 1] 里面取出足够数量的点然后带入Bezier曲线的表达式中计算出坐标即可。

   参数方程如下

   $Q(t) = \sum_{i=0}^{n} P_i B_{i,n}(t), t \in [0, 1]$

   其中

   $B_{i,n}(t) = \frac{n!}{i!(n-i)!} t^i (1-t)^{n-i}, i = 0, 1..n$



① 写计算阶乘的函数

```
// 计算阶乘
int factorial(int number) {
    int result = 1;
    if (number == 0 || number == 1) { return 1; }
    else {
        for (int i = 1; i <= number; i++) {
            result *= i;
        }
    }

    return result;
}
```

② 写计算伯恩斯坦基函数的函数

```
// 计算伯恩斯坦基函数
float B(int i, int n, float t) {
    float result = 0.0;
    result = (float)factorial(n) / ((float)factorial(i) * factorial(n - i))
* pow(t, i) * pow((1 - t), n - i);

    return result;
}
```

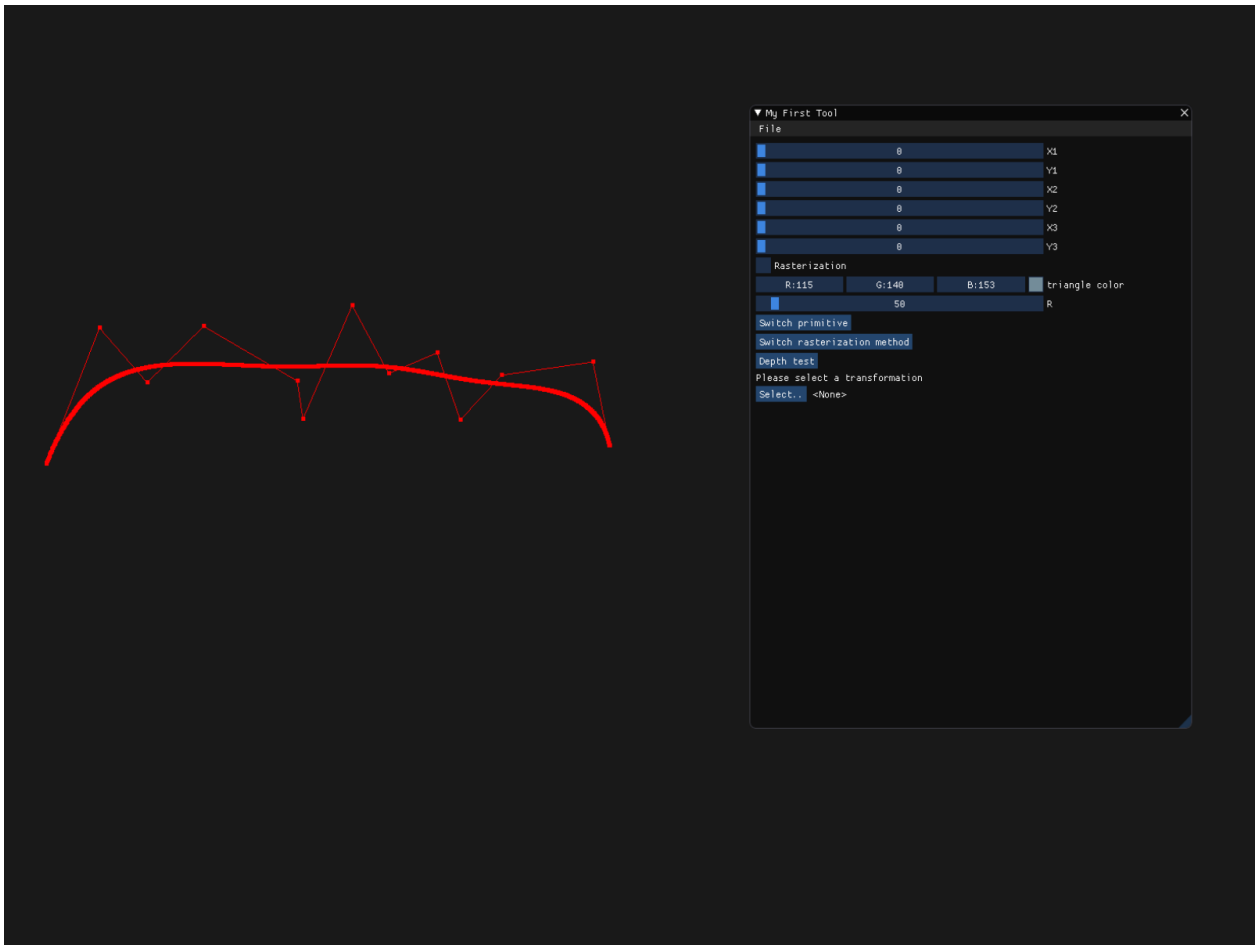
③ 写出利用Bezier函数计算出曲线点的函数

```
// 计算bezier函数
vector<glm::vec3> calculateBezier(vector<glm::vec3> bezier_points) {
    // 计算出points的个数
    int n = bezier_points.size();

    //
    vector<glm::vec3> result;

    // 循环计算出点
    for (float t = 0; t <= 1; t += 0.001) {
        glm::vec3 one_point(0.0f, 0.0f, 0.0f);

        for (int i = 0; i < n; i++) {
            one_point += bezier_points[i] * B(i, n-1, t);
        }

        result.push_back(one_point);
    }

    return result;
}
```

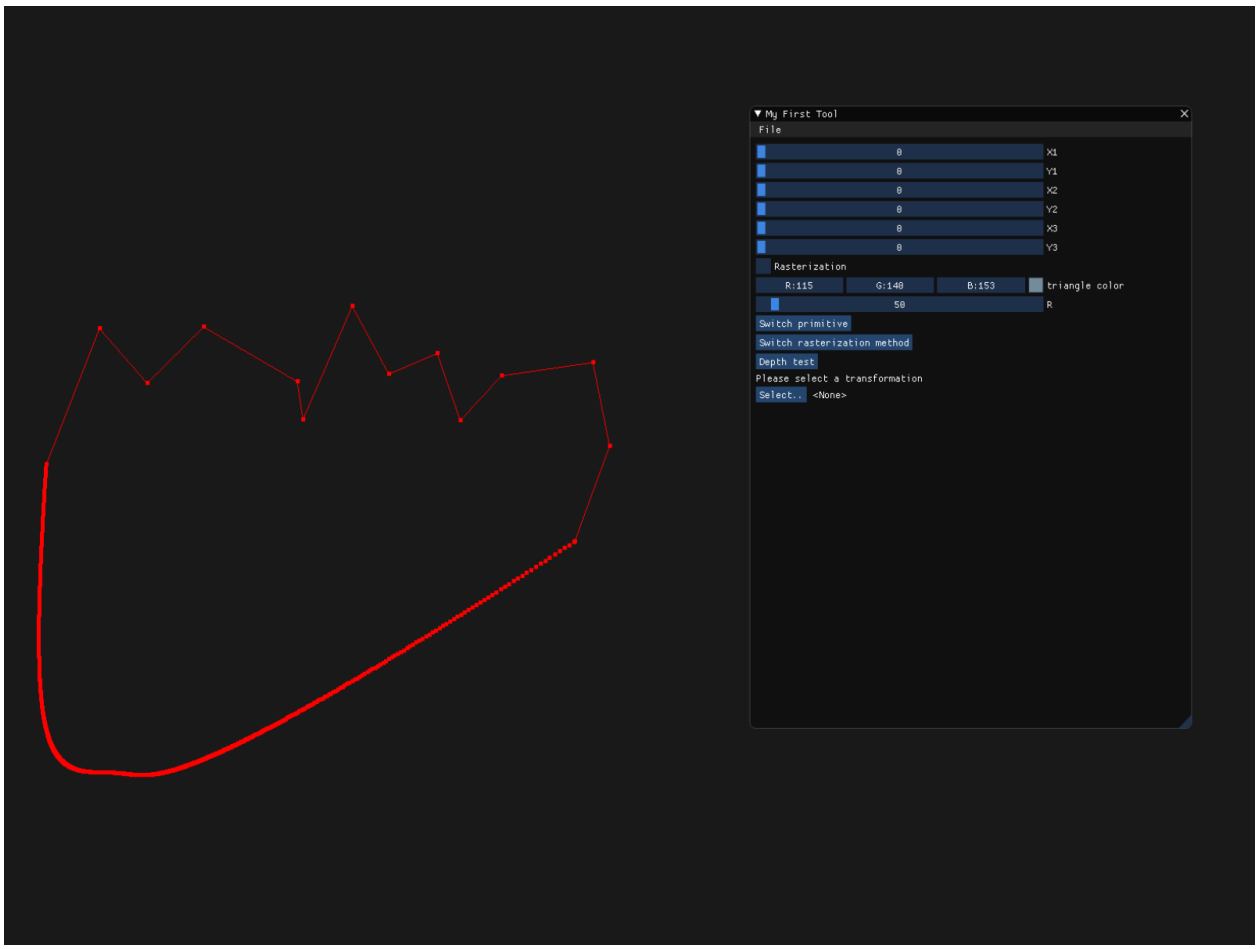
④ 在循环中渲染

```
// bezier曲线的点

vector<glm::vec3> points = calculateBezier(bezier_points);
auto vec_bezier = convertVec3ToFloat(points);
float* data_bezier = vec_bezier.data();
size_t data_len_bezier = vec_bezier.size() * sizeof(float);
int draw_size_bezier = vec_bezier.size() / 3;
glBufferData(GL_ARRAY_BUFFER, data_len_bezier, data_bezier,
GL_STATIC_DRAW);
glPointSize(5);
glDrawArrays(GL_POINTS, 0, draw_size_bezier)
```

效果



但是直接像上面那样写会有问题。上面的图有13个控制点，显示是正常的。但是当我再加一个点的时候，整个曲线突然大幅度变化。



这是由于溢出导致的。我们上面直接算阶乘点的数量一多就很容易溢出。所以我们需要对阶乘计算那部分处理一下，把能约简的先约简。同时把int换为long long int。

$$B_{i,n}(t) = \frac{n!}{i!(n-i)!}t^i(1-t)^{n-i}, i = 0, 1..n$$

基函数里面的 $\frac{n!}{i!(n-i)!}$ 中分子$n!$是固定的（当n一定时），而$n!$里面一定有一部分是可以和分母里面$i!$或$(n-i)!$化简的。为了化简得比较多，我们先和$i!$和$(n-i)!$里面的较大者进行约分。分子前面一部分会被约掉，只剩下后面一部分。

```
float result = 0.0;
long long int numerator = 1; // 分子
long long int denominator = 1;

long long int max = (i >= (n - i)) ? i : n - i;
long long int min = (i < (n - i)) ? i : n - i;
int start = max + 1; // 从max+1开始
for (start; start <= n; start++) {
    numerator *= start;
}
```

之后$i!$和$(n-i)!$里面的较小者可能还是可以和分子约分的，这个时候计算它们的最大公约数，然后化简。

```
denominator = factorial(min);
long long int gcd = get_greatest_common_divisor(numerator, denominator);

// 再进一步化简
// 计算最大公约数
numerator /= gcd;
denominator /= gcd;

result = (float)numerator / denominator;

return result;
```

其中get_greatest_common_divisor如下

```
long long int get_greatest_common_divisor(long long int numerator, long
long int denominator)
{
    long long int temp;
    long long int a = numerator;
    long long int b = denominator;
    if (a < b)
    {
        temp = a;
        a = b;
        b = temp;
    }
    while (b != 0)
    {
        temp = a % b;
        a = b;
        b = temp;
    }
    return a;
}
```

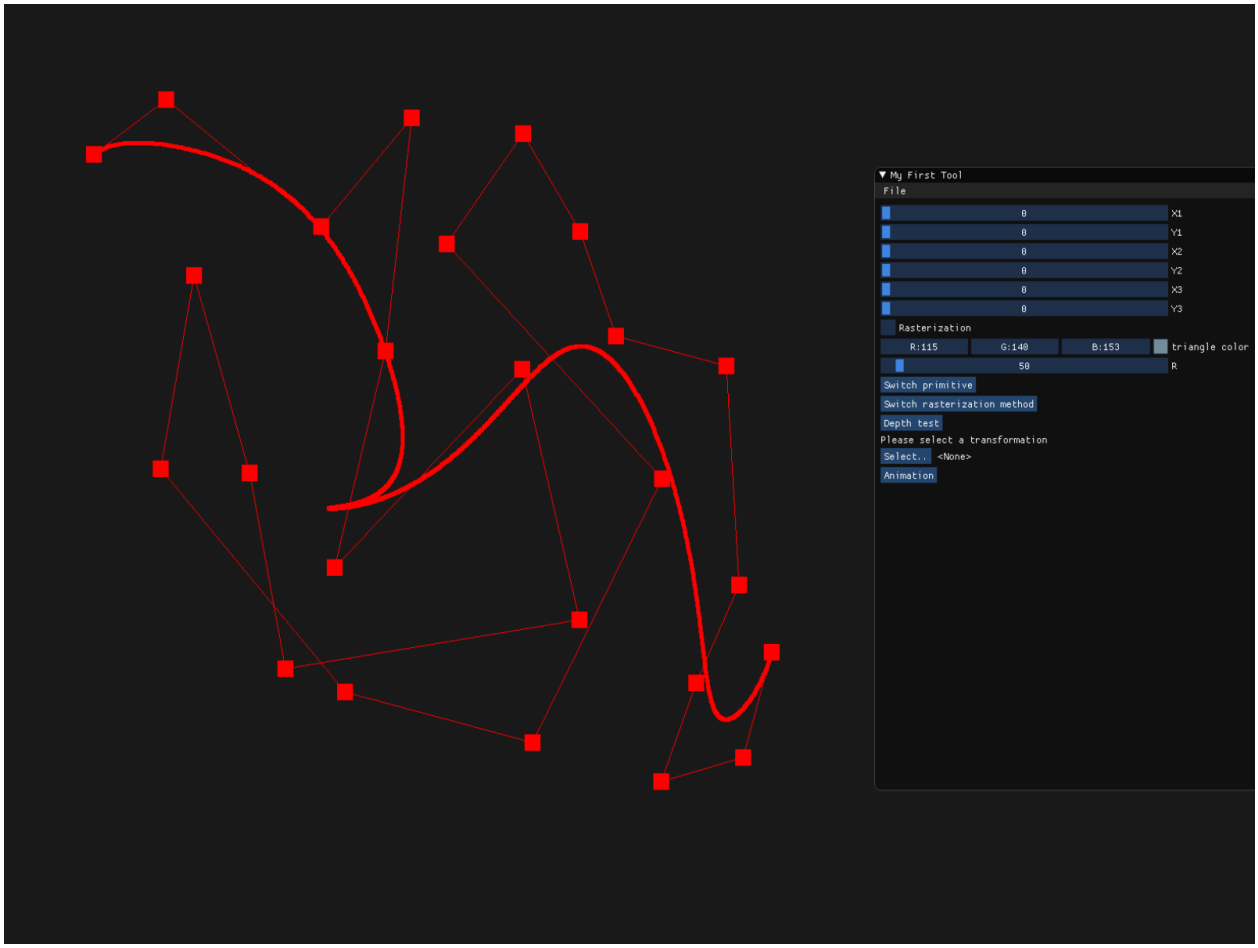
完整的处理过程

```
float factor(int n, int i) {
    float result = 0.0;
    long long int numerator = 1; // 分子
    long long int denominator = 1;

    long long int max = (i >= (n - i)) ? i : n - i;
    long long int min = (i < (n - i)) ? i : n - i;
    int start = max + 1;
    for (start; start <= n; start++) {
        numerator *= start;
    }

    denominator = factorial(min);
```

```
    long long int gcd = get_greatest_common_divisor(numerator,
denominator);

    // 再进一步化简
    // 计算最大公约数
    numerator /= gcd;
    denominator /= gcd;

    result = (float)numerator / denominator;

    return result;
}
```

处理完之后，就可以较高阶的bezier函数了。

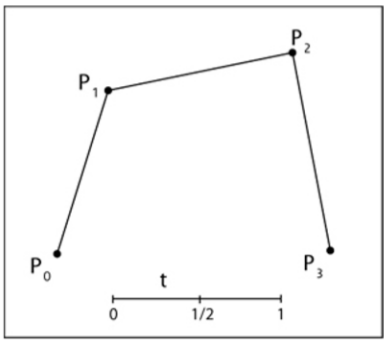
下图有25个控制点。

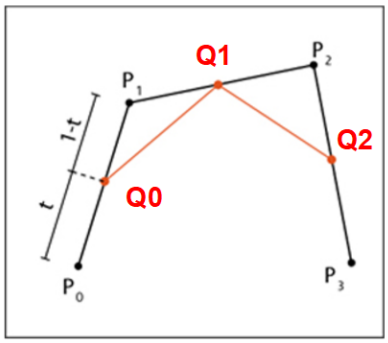


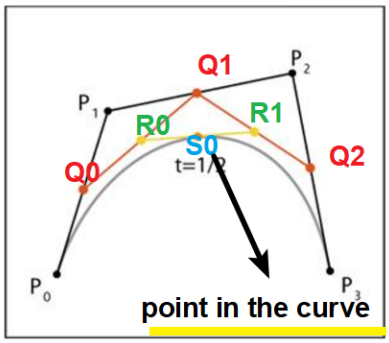
Bonus:

1. 可以动态地呈现Bezier曲线的生成过程。

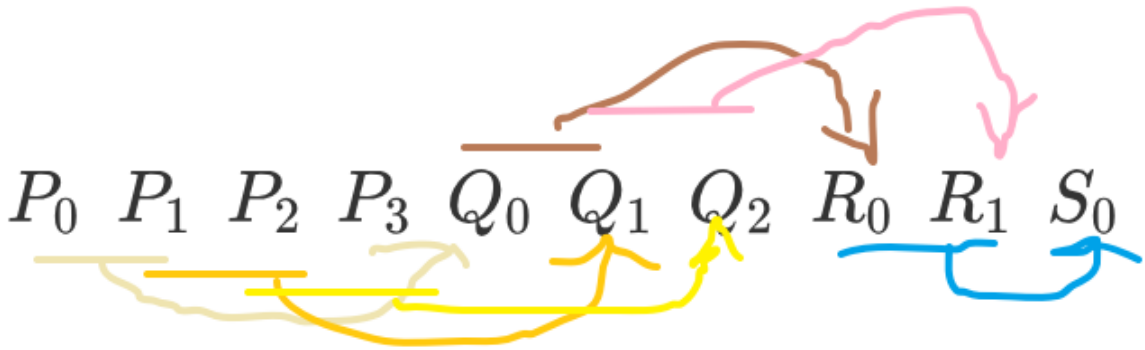
   这里需要理解Bezier曲线是怎么插值来的。







假设有4个控制点，t = 1/2，我们先在P_0P_1, P_1P_2, P_2P_3分别插值出Q_0, Q_1, Q_2（使用$Q = (1-t)P + tP'$）。然后把新生成的点再连线，再计算出插值点，一直做下去直到新生成的插值点只有一个，而这个点其实也就是落在曲线上的点。



写出函数如下

```
// 计算动画显示的点和线的vector
void generate_points(float delta_t, vector<glm::vec3> bezier_points,
vector<float>&animation_points, vector<float>&animation_lines) {
    int size = bezier_points.size();
    int idx = 0;

    while (size != 1) {
        for (int i = 0; i < size - 1; i++) {
            //cout << "delta_t" << delta_t << endl;
            glm::vec3 new_point;
            new_point = (1-delta_t) * bezier_points[idx] + delta_t *
bezier_points[idx + 1];
            animation_lines.push_back(bezier_points[idx].x);
            animation_lines.push_back(bezier_points[idx].y);
            animation_lines.push_back(bezier_points[idx].z);
            animation_lines.push_back(bezier_points[idx+1].x);
            animation_lines.push_back(bezier_points[idx+1].y);
            animation_lines.push_back(bezier_points[idx+1].z);

            bezier_points.push_back(new_point);

            animation_points.push_back(new_point.x);
            animation_points.push_back(new_point.y);
            animation_points.push_back(new_point.z);
```

```
            idx++;
        }

        idx++;
        size -= 1;

}
```

从上面的函数我们可以得到任意$t \in [0,1]$所对应的插值点以及它们的连线。要动态显示则需要将t与时间结合起来，这个时候就想到了glfwGetTime()。

首先设置一个animation_t0，然后在每一个while循环里面glfwGetTime()得到现在的时间，然后把delta_t = glfwGetTime()-animation_t0作为t代入上面的函数计算出要渲染的点与直线。若delta_t大于1了，则曲线已经绘制完成，不再渲染。我们也可以对glfwGetTime()-animation_t0乘上一个系数使动画更加易于观察。

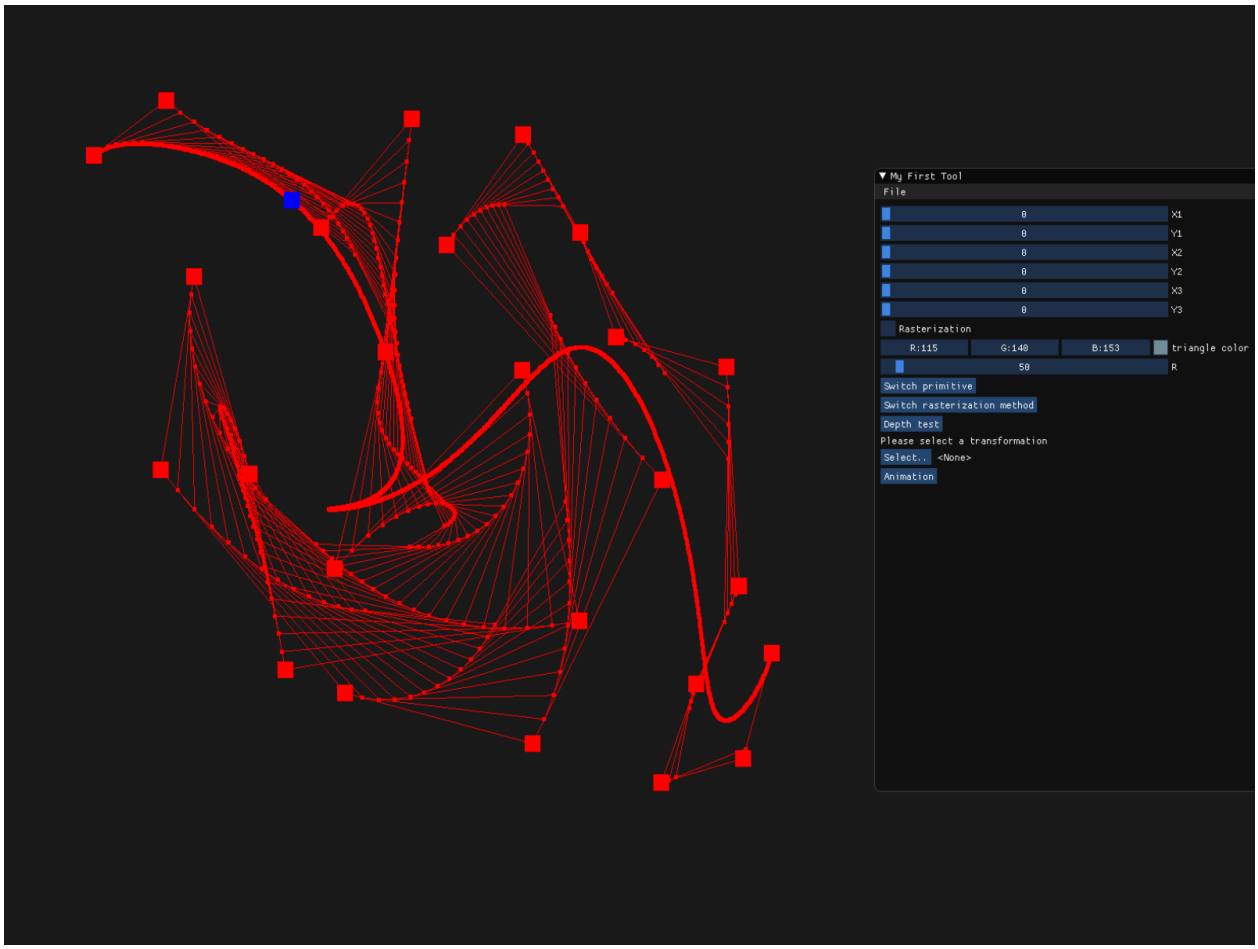
然后设置读取键盘操作。当我们绘制好控制顶点之后，按下Enter键，animation_t0就会被设置为当前的时间（通过glfwGetTime()），然后就可以使delta_t处于[0,1]之间了，这时候就会产生动画。

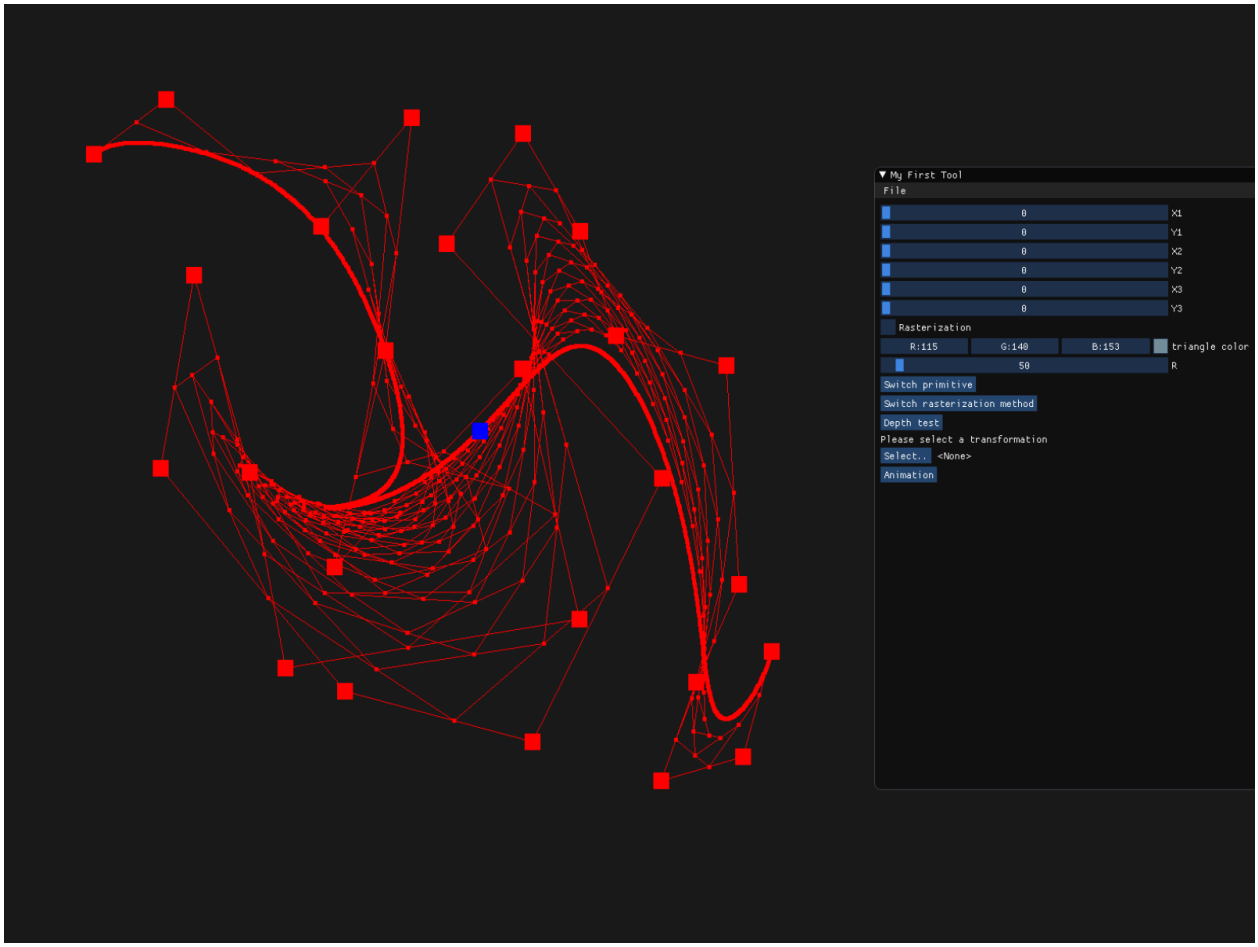
```
void processInputBezier(GLFWwindow* window, float& animation_t0) {
    if (glfwGetKey(window, GLFW_KEY_ENTER) == GLFW_PRESS) {
        animation_t0 = glfwGetTime();
    }
}
```

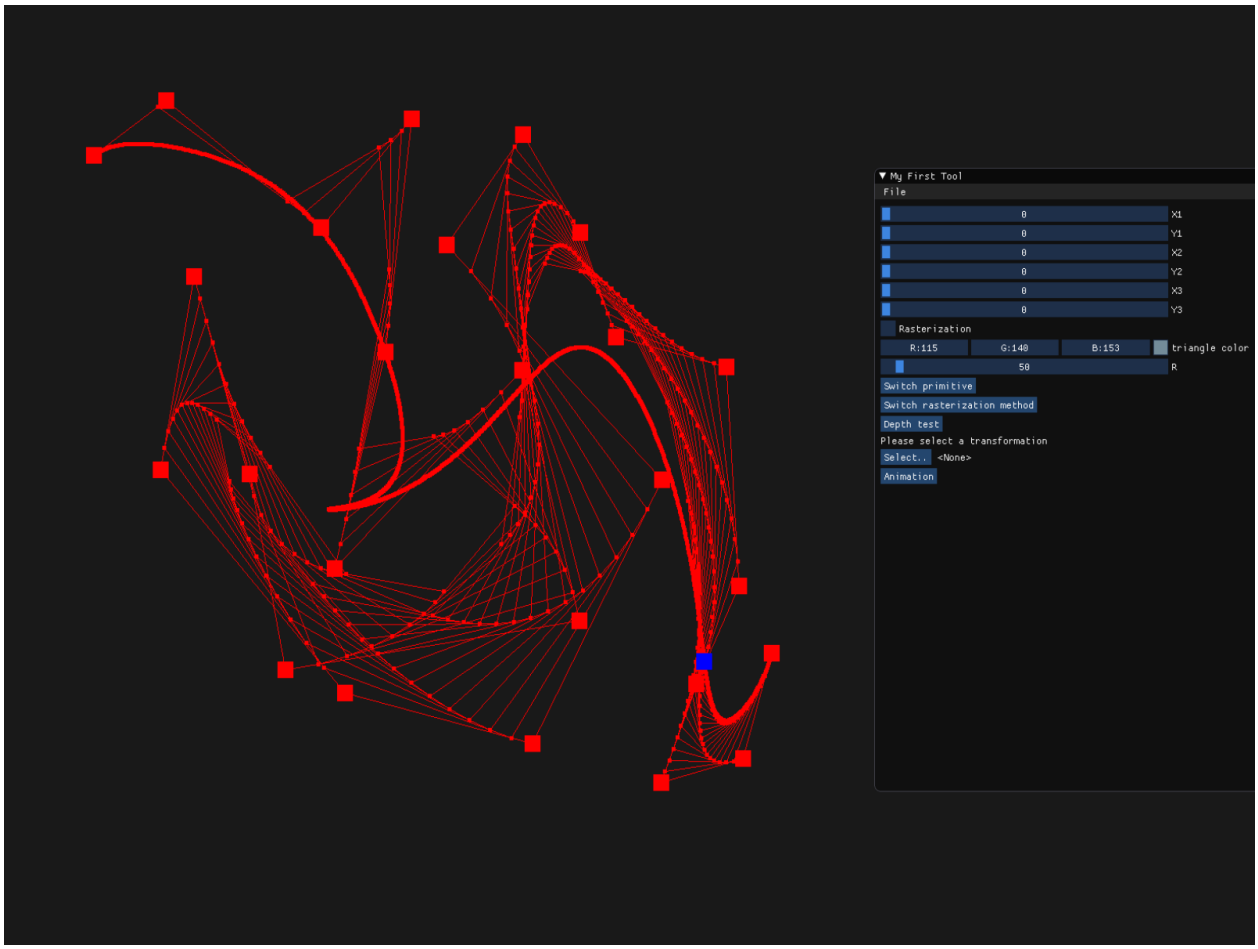
```
float delta_t = (glfwGetTime() - animation_t0) * 0.1;
if (delta_t<= 1 && delta_t >= 0) {
    vector<float> animation_points, animation_lines;
    generate_points(delta_t, bezier_points, animation_points,
animation_lines);

    // 画线
    float* data_animation_lines = animation_lines.data();
    size_t data_len_animation_lines = animation_lines.size() *
sizeof(float);
    int draw_size_animation_lines = animation_lines.size() / 3;
    glBufferData(GL_ARRAY_BUFFER, data_len_animation_lines,
data_animation_lines, GL_STATIC_DRAW);
    glDrawArrays(GL_LINES, 0, draw_size_animation_lines);

    // 画点
    float* data_animation_points = animation_points.data();
    size_t data_len_animation_points = animation_points.size() *
sizeof(float);
    int draw_size_animation_points = animation_points.size() / 3;
    glBufferData(GL_ARRAY_BUFFER, data_len_animation_points,
data_animation_points, GL_STATIC_DRAW);
    glDrawArrays(GL_POINTS, 0, draw_size_animation_points - 1);
```

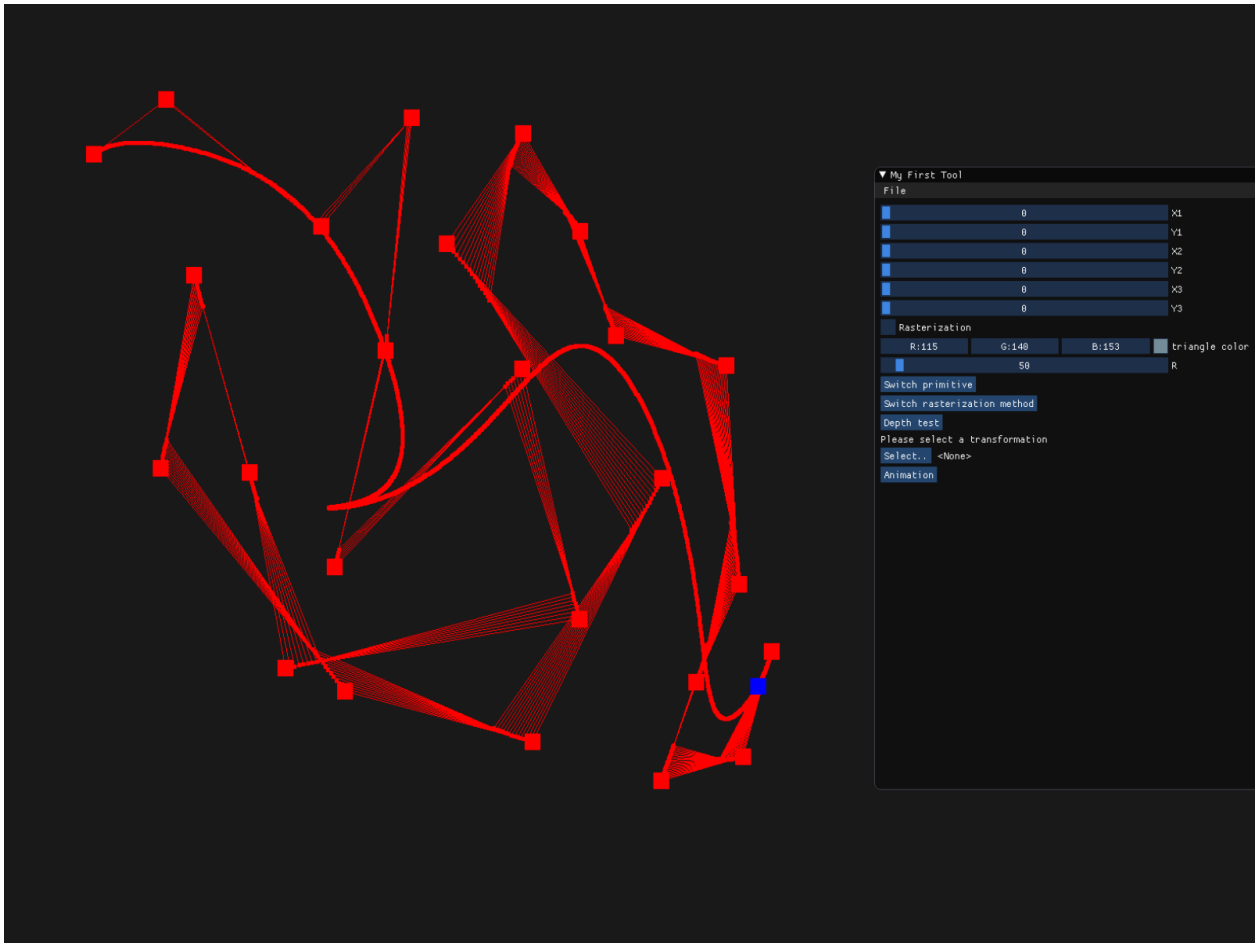
```
    glPointSize(20);
    glUniform1i(glGetUniformLocation(bezierShader.Program, "animation"),
1); // 渲染成蓝色
    // 用蓝色绘制落在bezier曲线上的点
    glDrawArrays(GL_POINTS, draw_size_animation_points - 1, 1);

}
```

效果




My First Tool
File
0
X1
0
Y1
0
X2
0
Y2
0
X3
0
Y3
Rasterization
R:115
G:140
B:153
triangle color
50
R
Switch primitive
Switch rasterization method
Depth test
Please select a transformation
Select..
<None>
Animation


My First Tool
File
0
X1
0
Y1
0
X2
0
Y2
0
X3
0
Y3
Rasterization
R:115
G:140
B:153
triangle color
50
R
Switch primitive
Switch rasterization method
Depth test
Please select a transformation
Select..
<None>
Animation


My First Tool
File
0
X1
0
Y1
0
X2
0
Y2
0
X3
0
Y3
Rasterization
R:115
G:140
B:153
triangle color
50
R
Switch primitive
Switch rasterization method
Depth test
Please select a transformation
Select..
<None>
Animation