Homework 8 - Bezier Curve

黄树凯

16340085

作业要求

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
2. 工具根据鼠标绘制的控制点实时更新Bezier曲线。

   Hint: 大家可查询捕捉mouse移动和点击的函数方法;

Bonus:

1. 可以动态地呈现Bezier曲线的生成过程。

算法原理

贝塞尔曲线本质上是由调和函数(Harmonic functions)根据控制点(control points)插值生成的。其参数方程如下：

$$\mathbf{B}(t) = \sum_{i=0}^{n} \mathbf{P}_i \mathbf{b}_{i,n}(t), \quad t \in [0,1]$$

上式为 n次多项式，具有 n+1项。其中， $Pi(i$=0,1...$n)$ 表示特征多边形的 n+1个顶点向量； $Bi,n(t)$为伯恩斯坦（Bernstein）基函数，其多项式表示为：

$$\mathbf{b}_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}, \quad i = 0, \dots n$$

线性贝塞尔曲线

给定点**P**0、**P**1，线性贝塞尔曲线只是一条两点之间的直线。这条线由下式给出：

$$\mathbf{B}(t) = \mathbf{P}_0 + (\mathbf{P}_1 - \mathbf{P}_0)t = (1-t)\mathbf{P}_0 + t\mathbf{P}_1 \ , t \in [0,1]$$

且其等同于线性插值。

二次方贝塞尔曲线

二次方贝塞尔曲线的路径由给定点**P**0、**P**1、**P**2的函数**B**（t）追踪：

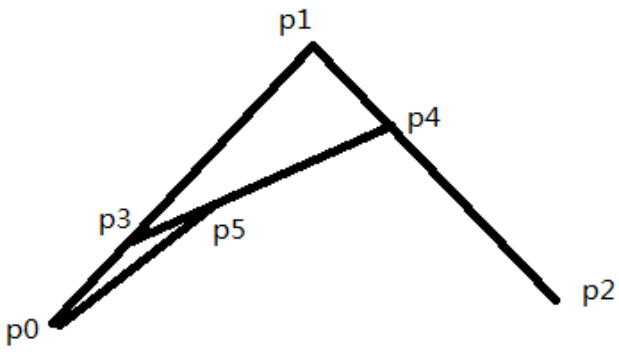
$$\mathbf{B}(t) = (1-t)^2\mathbf{P}_0 + 2t(1-t)\mathbf{P}_1 + t^2\mathbf{P}_2 \ , t \in [0,1]$$

n 次方贝塞尔曲线

$$\mathbf{B}(t) = \sum_{i=0}^{n} \mathbf{P}_i \mathbf{b}_{i,n}(t), \quad t \in [0,1]$$

$$\mathbf{b}_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}, \quad i = 0, \ldots n$$

举例二次方贝塞尔曲线可以由一下过程实现



简单来说，我们就是要求当t时p5的位置，前面我们可以知道：

p3=(1-t)*p0+t*p1

p4=(1-t)*p1+t*p2

p5=(1-t)*p3+t*p4

大家代入可得我们上面给出的二次方贝塞尔曲线.

p2=$(1-t)^2P_0+2t(1-t)P_1+t^2P_2, t\in[0,1]$

实现过程

鼠标绘制控制点

添加一个捕获鼠标当前坐标的函数和一个鼠标点击事件响应函数，需要注意的问题是坐标的转换，鼠标点击得到的坐标是以窗口左上角为（0,0），x轴正方向向右，y轴正方向向下的坐标系。而openGL显示的坐标系是以窗口中心为原点，划分为四个象限。因此 x = mouseX - SCR_WIDTH / 2，y = SCR_HEIGHT / 2 - mouseY

```
// 设置窗口大小
const unsigned int SCR_WIDTH = 1280;
const unsigned int SCR_HEIGHT = 720;
// 记录鼠标点击位置
float mouseX = 0;
float mouseY = 0;

void mouse_callback(GLFWwindow* window, double xpos, double ypos) {
    mouseX = xpos;
    mouseY = ypos;
}
void mouse_button_callback(GLFWwindow* window, int button, int action, int mods) {
    if (action == GLFW_PRESS) {
        curvePoints.clear();
        step = 0;
```

```
            finish = false;
            switch (button) {
            // 点击鼠标左键时添加控制点
            case GLFW_MOUSE_BUTTON_LEFT: {
                Point p(mouseX - SCR_WIDTH / 2.0f, SCR_HEIGHT / 2.0f - mouseY);
                ctrlPoints.push_back(p);
                break;
            }
            // 点击鼠标右键时删除最后一个添加的控制点
            case GLFW_MOUSE_BUTTON_RIGHT: {
                if (!ctrlPoints.empty()) {
                    ctrlPoints.pop_back();
                }
                break;
            }
            default:
                break;
            }
        }
    }
```

Bezier 曲线绘制

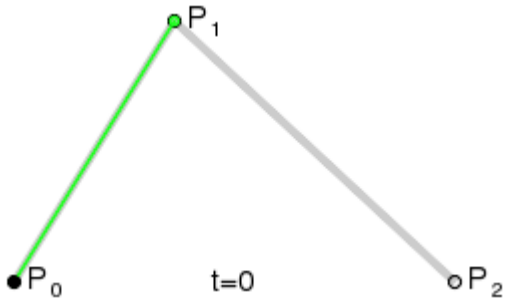
先由一次线性贝塞尔曲线进行分析，一次贝塞尔曲线的方程如下：

$$\mathbf{B}(t) = \mathbf{P}_0 + (\mathbf{P}_1 - \mathbf{P}_0)t = (1-t)\mathbf{P}_0 + t\mathbf{P}_1 \ , t \in [0,1]$$

给定两个控制点，以及一个变化的参数t，求出当参数 t 变化时的过程点，这一系列的点就组成了贝塞尔曲线。

再对二次线性贝塞尔曲线进行分析，二次贝塞尔曲线的方程如下：

$$\mathbf{B}(t) = (1-t)^2\mathbf{P}_0 + 2t(1-t)\mathbf{P}_1 + t^2\mathbf{P}_2 \ , t \in [0,1]$$



结合图片进行理解，二次方线性贝塞尔曲线有三个控制点，同理按照一次方的做法，对于一个给定的参数 t，对相连的控制点按照一次线性贝塞尔曲线的方法求出过程点，再将求得的过程点相连视为一组一次线性贝塞尔曲线，再进行一次贝塞尔曲线计算，得到的点就是二次贝塞尔曲线上的点。

那么 n 次方的贝塞尔曲线的计算方法也可以以此类推，对于一个给定的参数 t；第一次是用控制点，每两个相连的控制点作一次贝塞尔曲线计算得到一组位于控制点连线上的点。再用得到的新的一组点作为控制点求下一组内层的点。直到剩下最后两个点作最后一次计算得到 n 次方贝塞尔曲线上的点。根据变化的参数 t 求出的一系列点就可以构成 n 次方贝塞尔曲线。

代码实现过程如下：

```
struct Point {
    float x;
    float y;
    Point(float _x, float _y) {
        x = _x;
        y = _y;
    }
};

static void Bezier(const vector<Point> &ctrlPoints, vector<Point> &curvePoints) {
    float t = 0;
    int n = ctrlPoints.size() - 1;

    for (int i = 0; i <= PRECISION; i++) {
        t = (float)(i) / (float)(PRECISION);
        Point p = getCurvePoints(t, ctrlPoints, n);
        curvePoints.push_back(p);
    }
}

// 计算贝塞尔曲线上的点
static Point getCurvePoints(float t, const vector<Point> &ctrlPoints, int n) {
    vector<Point> temp;
    Point p(0, 0);
    for (int i = 0; i < n; i++) {
        p.x = ctrlPoints[i].x * (1 - t) + ctrlPoints[i + 1].x * t;
        p.y = ctrlPoints[i].y * (1 - t) + ctrlPoints[i + 1].y * t;
        temp.push_back(p);
    }
    while (temp.size() != 1) {
        vector<Point> next;
        for (int i = 0; i < temp.size() - 1; i++) {
            p.x = temp[i].x * (1 - t) + temp[i + 1].x * t;
            p.y = temp[i].y * (1 - t) + temp[i + 1].y * t;
            next.push_back(p);
        }
        temp = next;
    }
    return temp[0];
}
```

Bonus 动画过程

为了显示曲线绘制的动画过程，要对传入的参数 t 进行控制，将原本在一次绘制中完成的对贝塞尔曲线的计算分到多次绘制中完成，从而实现动画过程。另一个要点是每次绘制过程计算中间点的位置，并进行正确的连线。

对上面的贝塞尔曲线实现代码作一些修改，将参数 t 由外部传入，每次绘制前进行迭代增量。每一层迭代计算时将过程点保存。

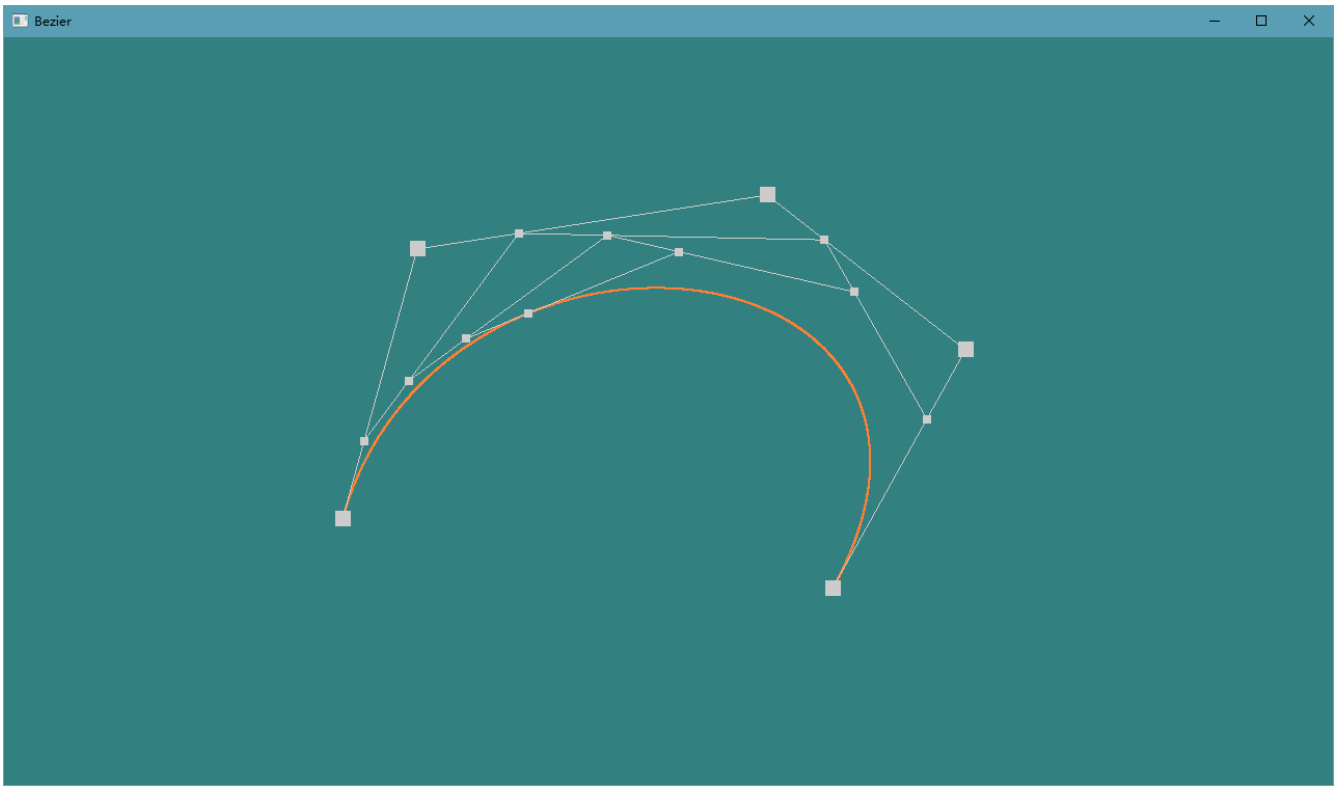
```
// 渲染循环
while (!glfwWindowShouldClose(window)) {
    processInput(window);
    glfwPollEvents();
    ...
    // 绘制过程动画
    if (step < 2000) {
        step += 5;
    }
    BezierCurve::getActionPoints(step, ctrlPoints, actionPoints);
    ...
}
```

```
// 计算过程点（同计算曲线点相同，仅是为了展示动画过程额外写一个函数）
static void getActionPoints(int step, const vector<Point> &ctrlPoints, vector<Point>
&actionPoints) {
    float t = (float)step / (float)PRECISION;
    int n = ctrlPoints.size() - 1;
    vector<Point> temp;
    Point p(0, 0);
    for (int i = 0; i < n; i++) {
        p.x = ctrlPoints[i].x * (1 - t) + ctrlPoints[i + 1].x * t;
        p.y = ctrlPoints[i].y * (1 - t) + ctrlPoints[i + 1].y * t;
        actionPoints.push_back(p);
        temp.push_back(p);
    }
    while (temp.size() != 1) {
        vector<Point> next;
        for (int i = 0; i < temp.size() - 1; i++) {
            p.x = temp[i].x * (1 - t) + temp[i + 1].x * t;
            p.y = temp[i].y * (1 - t) + temp[i + 1].y * t;
            actionPoints.push_back(p);
            next.push_back(p);
        }
        temp = next;
    }
    actionPoints.push_back(temp[0]);
}
```

将同一层的点进行连线，由于每一层迭代的控制点减一，由以下代码可以实现：

```
int current = 0;
int count = ctrlPoints.size() - 1;
while (count > 1) {
    glDrawArrays(GL_LINE_STRIP, current, count);
    current += count;
    count--;
}
```

实验结果



实验结果演示见附件动图