Homework 9 - 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
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

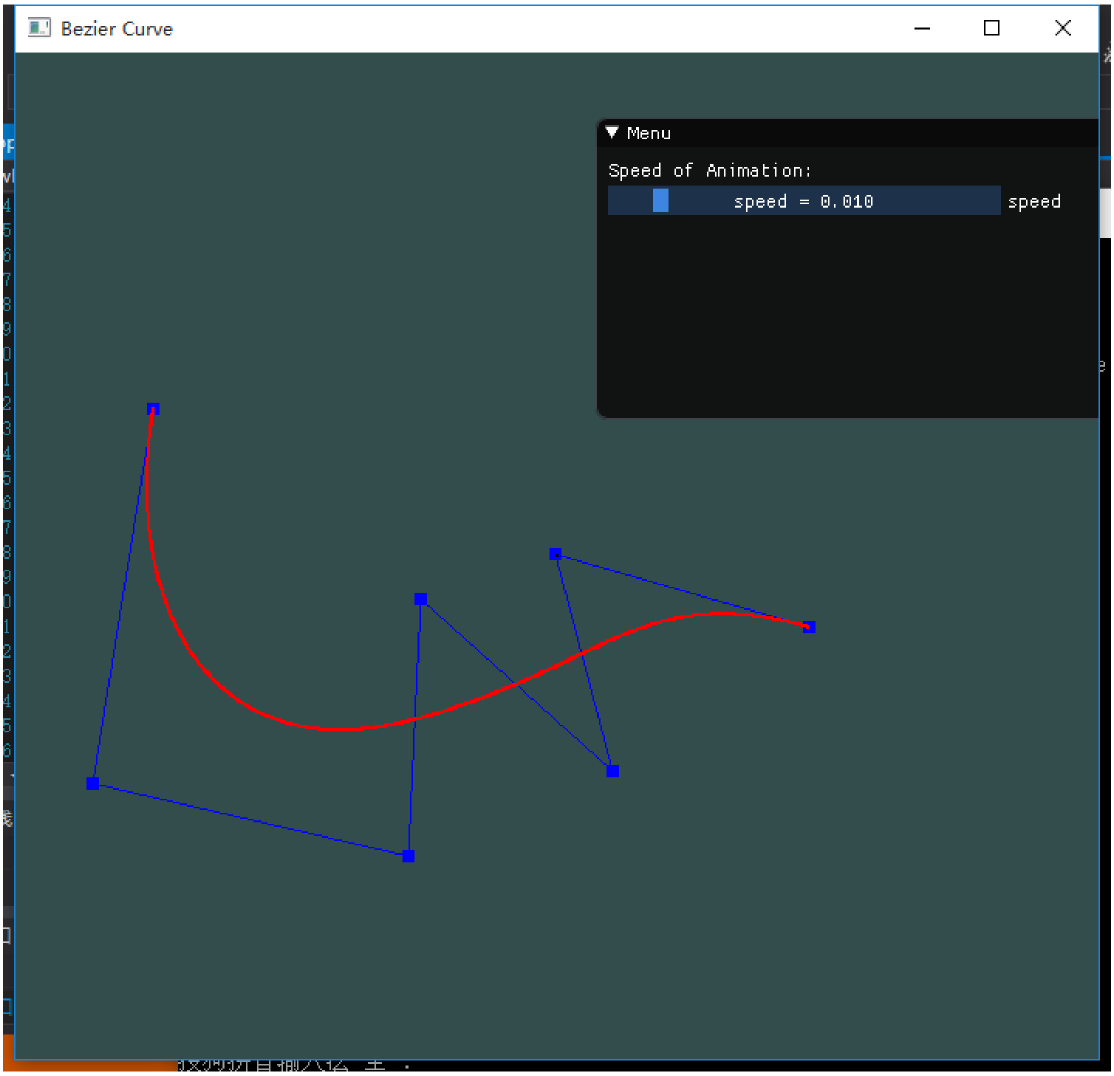
Bonus:

1. 可以动态地呈现 Bezier 曲线的生成过程。

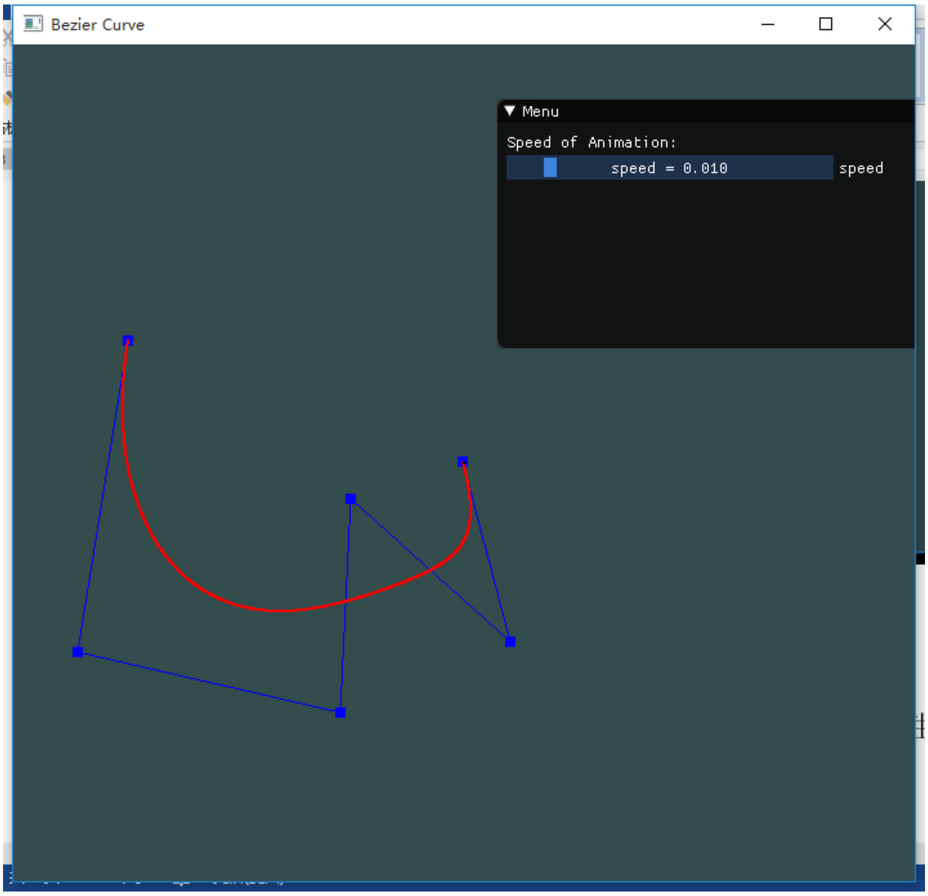
实现效果：

（展示结果见[演示视频.mp4]）

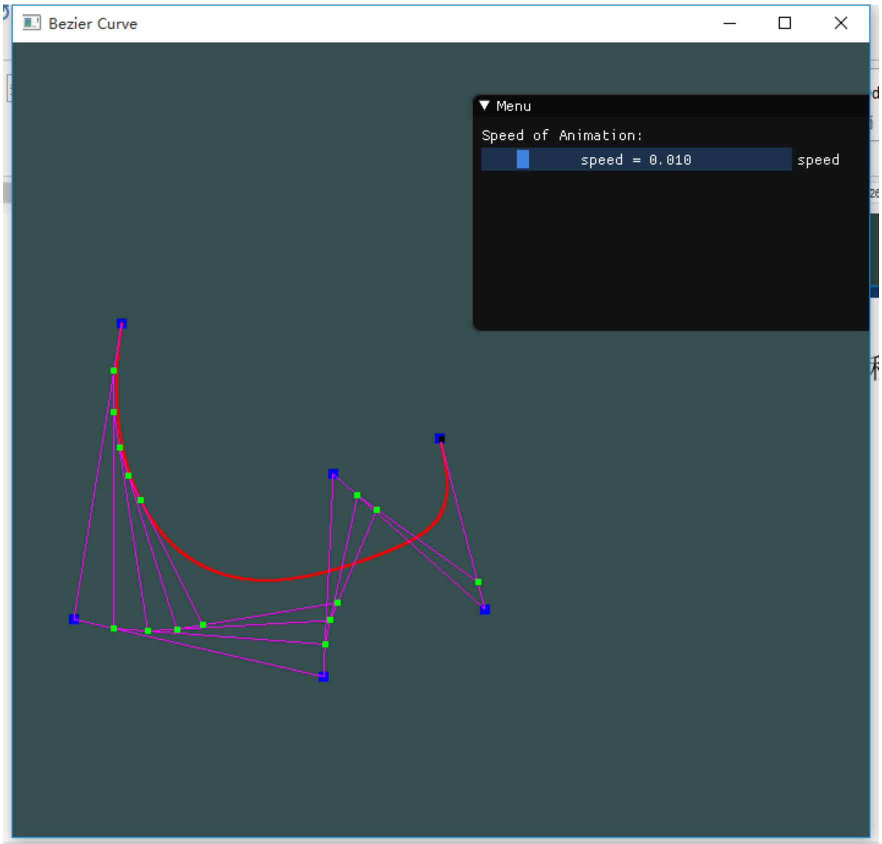
鼠标左键点击添加 7 个控制点：



鼠标右键点击减少一个控制点：



按下回车键（Enter），进行动态地呈现 Bezier 曲线的生成过程。



实现过程：

使用一个全局变量 control_points 来记录控制点，用全局变量 is_running 来标记是否进行动态生成演示：

```cpp
//一些全局变量
bool is_running = false;
vector<glm::vec2> control_points;   //控制点
```

设置鼠标输入回调函数 mouseCallback，捕获鼠标事件。鼠标左键点击，捕获当前点击位置，将坐标转化后，记录在 control_points 中；鼠标右键点击，去掉最新增加的一个控制点。此外，为了防止动态生成演示过程中增加点或减少点，动态演示过程中阻塞鼠标点击事件。

```cpp
void mouseCallback(GLFWwindow* window, int button, int action, int mods) {
    //如果正在运行bonus的动画，阻塞鼠标事件
    if (is_running) return;

    if (button == GLFW_MOUSE_BUTTON_RIGHT && action == GLFW_PRESS) {
        control_points.pop_back();
    }
    else if (button == GLFW_MOUSE_BUTTON_LEFT && action == GLFW_PRESS) {
        //获取光标位置
        double x, y;
        glfwGetCursorPos(window, &x, &y);
        //坐标归一化
        control_points.push_back(glm::vec2(((float)x / (float)WIN_WIDTH * 2.0f) - 1, -(((float)y / (float)WIN_HEIGHT * 2.0f) - 1)))
    }
}
```

处理键盘输入，当输入回车键（Enter）时，设置 is_running 为 True，表示开始动态演示。

```cpp
//处理键盘输入
void processInput(GLFWwindow* window) {
    //Esc输入
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS) {
        glfwSetWindowShouldClose(window, true);
    }
    //输入回车，设置参数播放逼近过程动画
    if (!is_running && glfwGetKey(window, GLFW_KEY_ENTER) == GLFW_PRESS) {
        is_running = true;
    }
}
```

Bezier 曲线确定：

曲线的参数方程如下：

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

其中，$B_{i,n}(t)$的多项式如下：

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

可以看出，$B_{i,n}(t)$实际上有两部分组成，第一部分为$\binom{n}{i}$，即二项式系数组合，第二部分为$t^i(1-t)^{n-i}$。在此，不妨先计算第一部分：

```
//计算二项式系数
int biCoe(int n, int m) {
    if (m << 1 > n) m = n - m;
    if (m == 0) {
        return 1;
    }
    int coe = n;
    for (int i = 1; i < m; i++) {
        coe *= (n - i);
        coe /= (i + 1);
    }
    return coe;
}
```

对于第二部分$t^i(1-t)^{n-i}$，也可以看成两个部分，第一部分（α_1）为 t，第二部分（α_2）为$(1-t)^n$，则对于 i 从 0 递增到 n，t^i可通过$\alpha_1 = \alpha_1 * t$ 实现，$(1-t)^{n-i}$可通过$\alpha_2 = \frac{\alpha_2}{1-t}$实现。

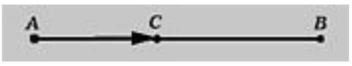
```
//计算Q(t) = P * B(t)
glm::vec2 bezier(double t) {
    int order = control_points.size() - 1;

    double alpha_1 = 1;
    double alpha_2 = glm::pow(1 - t, order);

    double x = 0;
    double y = 0;
    for (int i = 0; i <= order; i++) {
        x += biCoe(order, i) * alpha_1 * alpha_2 * control_points[i].x;
        y += biCoe(order, i) * alpha_1 * alpha_2 * control_points[i].y;
        alpha_1 *= t;
        alpha_2 /= (1 - t);
    }
    return glm::vec2(x, y);
}
```

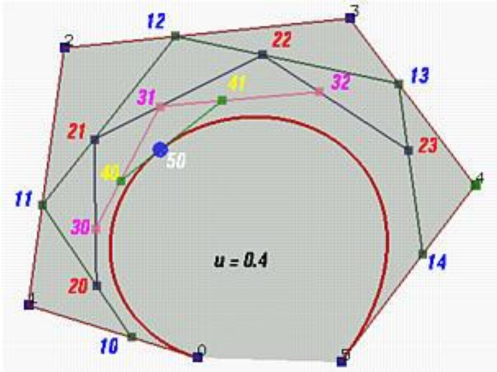
Q = P* B

对于 Bonus 部分，可使用 De Casteljau 算法实现[参考博客]：



De Casteljau 算法的基本观点是选择在 AB 中的一个点 C，C 将 AB 分为 u:1-u(A 到 C 的距离与 AB 之间的距离之比是 u)，从 A 到 B 的向量是 B-A。因为 u 是在 0 和 1 之间的比率，点 C 位于 u(B-A)。将 A 的位置加以考虑，点 C 为 A+u(B-A)=(1-u)A+uB。因此，对于给定的 u，(1-u)A+uB 是在 A 和 B 之间的点 C，将 AB 分为 u:1-u 的两段。

De Casteljau 算法的想法如下：假设我们想要找到 C(u)，u 在[0,1]中。由第一个多段线 00-01-02-03...-0n 开始，利用上面的法则找到在线段上的点 1i，1i 在 0i 到 0(i+1)的连线上并且将这段线分为 u:1-u 的两部分。依次地，我们可以得到 n 个点 10,11,12,...,1(n-1)，他们定义了一个新的多段线（polyline），一共有 n-1 段



在上图中，u 是 0.4,10 是在 00 和 01 的线段（leg），11 是在 01 和 02 的线段,...,并且 14 是在 04 到 05 的线段，所有的新点都由蓝色的表示。

新点由 1i 进行标记，再次利用上面的规则我们可以得到第二个多段线，具有 n-1 个点（20,21,...,2(n-2)）和 n-2 条边。从这个多段线开始，进行第三次，得到新的多段线，由 n-2 个点 30,31,...,3(n-3)和 n-3 条边组成。重复这个过程 n 次得到一个点 n0，Casteljau 已经证明在曲线上的点 C(u)对应 u

以上图举例，20 是在 10 和 11 上将这段线分为 u:1-u 的点，类似地选取 21 在 11 和 12 上，22 在 12 和 13 上，23 在 13 和 14 上.第三个多段线是 20,21,22,23.第三个多段线有四个点和三条边。继续做得到 30,31,32 组成的多段线，这是第四个多段线。继续得到有 40 和 41 组成的线段，再做一次得到 50，为点 C(0.4)。

在本次实现过程中，使用循环来实现上述的递归过程：

```
//使用循环实现递归过程
vector<float> renderTempPoints;
vector<glm::vec2> tempPoints(control_points);
while (tempPoints.size() > 1) {
    vector<glm::vec2> next;    //记录下一轮递归用到的点
    for (int i = 1; i < tempPoints.size(); ++i) {
        float tempLine[]{
            tempPoints[i - 1].x, tempPoints[i - 1].y, 0.0f, 1.0f, 0.0f, 1.0f,
            tempPoints[i].x, tempPoints[i].y, 0.0f, 1.0f, 0.0f, 1.0f
        };
        //画出辅助直线
        glBufferData(GL_ARRAY_BUFFER, 12 * sizeof(float), tempLine, GL_STATIC_DRAW);
        glDrawArrays(GL_LINES, 0, 2);

        glm::vec2 nextPoint;
        nextPoint.x = tempPoints[i].x * run_time + tempPoints[i - 1].x * (1 - run_time);
        nextPoint.y = tempPoints[i].y * run_time + tempPoints[i - 1].y * (1 - run_time);
        next.push_back(nextPoint);
        vector<float> temp{ nextPoint.x, nextPoint.y, 0.0f, 0.0f, 1.0f, 0.0f };
        renderTempPoints.insert(renderTempPoints.end(), temp.begin(), temp.end());
    }
    tempPoints = next;
}
glPointSize(4.5f);
glBufferData(GL_ARRAY_BUFFER, renderTempPoints.size() * sizeof(float), renderTempPoints.data(), GL_STATIC_DRAW);
glDrawArrays(GL_POINTS, 0, renderTempPoints.size());
run_time += speed;
if (run_time >= 1.0f) {
    run_time = 0.0f;
    is_running = false;
}
```