计算机图形学

Bézier 曲线

Assignment 6

姓名：陈家豪

班级：2018 级计科 1 班

学号：18308013

目录

1 作业任务与要求

(1) Task 1：实现 de Casteljau 算法，并用它来绘制 Béizer 曲线；

(2) Task 2：在 Task 1 的基础上，调整一下代码以支持更多的控制点；

(3) 谈谈你对 Béizer 曲线的理解；

2 实验过程与结果

2.1 Task 1：de Casteljau 算法的实现

de Casteljau 算法本质上就是通过递归的形式获取 Béizer 曲线。其实现过程如下：

(1) 对一个控制点序列 $P_1, P_2, \cdots, P_n$，将相邻点连接起来形成线段；

(2) 用 $t:(1-t)$ 的比例划分每条线段，每条线段获得一个分割点；

(3) 每条线段的分割点组成新的控制点序列，数量相比原控制点序列减 1；

(4) 对新的控制点序列递归执行 (1) 至 (3)，直到新的序列只包含一个点，则该点就是最终输出；

(5) 不断调整 (2) 中 t 的值（$t \in [0,1]$），获得不同的点。当点足够密集时，其组成的曲线就是 Béizer 曲线。

基于上述思想，完成代码如下所示：

```
1  cv::Point2f de_Casteljau(const std::vector<cv::Point2f> &control_points, float t)
2  {
3      // TODO: Implement de Casteljau's algorithm
4      // return cv::Point2f();
5      if (control_points.size()==0)       // 如果没有点，返回初始值
6      return cv::Point2f();
7      else if (control_points.size()==1)// 如果只有一个点，返回该点
8      return control_points[0];
9      else if (control_points.size()==2)// 如果只有两个点，进行插值返回
10     return control_points[0]+t*(control_points[1]-control_points[0]);
11
12     std::vector<cv::Point2f> new_control_points;// 新的点vector
13     for (int i=0;i<control_points.size()-1;i++) // 依次对相邻的点插值，新点存入vector
14     new_control_points.push_back(control_points[i]+t*(control_points[i+1]-control_points[i]));
15     return de_Casteljau(new_control_points,t);  // 递归
16 }
```

可见，最重要的是第 15 行的递归。每次调用该函数，最终返回该控制点序列和相应 t 下生成的点坐标。

bezier 函数代码如下所示：

```
void bezier(const std::vector<cv::Point2f> &control_points, cv::Mat &window)
{
    // TODO: Iterate through all t = 0 to t = 1 with small steps, and call de Casteljau's
    // recursive Bezier algorithm.
    float interval=0.0001;
    for (float t=0;t<=1;t+=interval)
    {
        cv::Point2f newPoint=de_Casteljau(control_points,t);// 该t下对应的点
        window.at<cv::Vec3b>(newPoint.y,newPoint.x)[1]=255; // 对该点所在坐标的对应像素填充绿色
    }
}
```

我们设置 *interval* 为 0.0001，得到原始控制点序列下不同 t 值对应的点坐标，然后将该坐标对应的像素设成绿色，即可得到绿色的 bezier 曲线。

最后，我们在 main 函数通过调用 bezier 函数而不是 naive_bezier 函数，使用 de Casteljau 算法完成 Béizer 曲线。最终结果如图1和2所示，可见成功实现使用 de Casteljau 算法完成 Béizer 曲线。

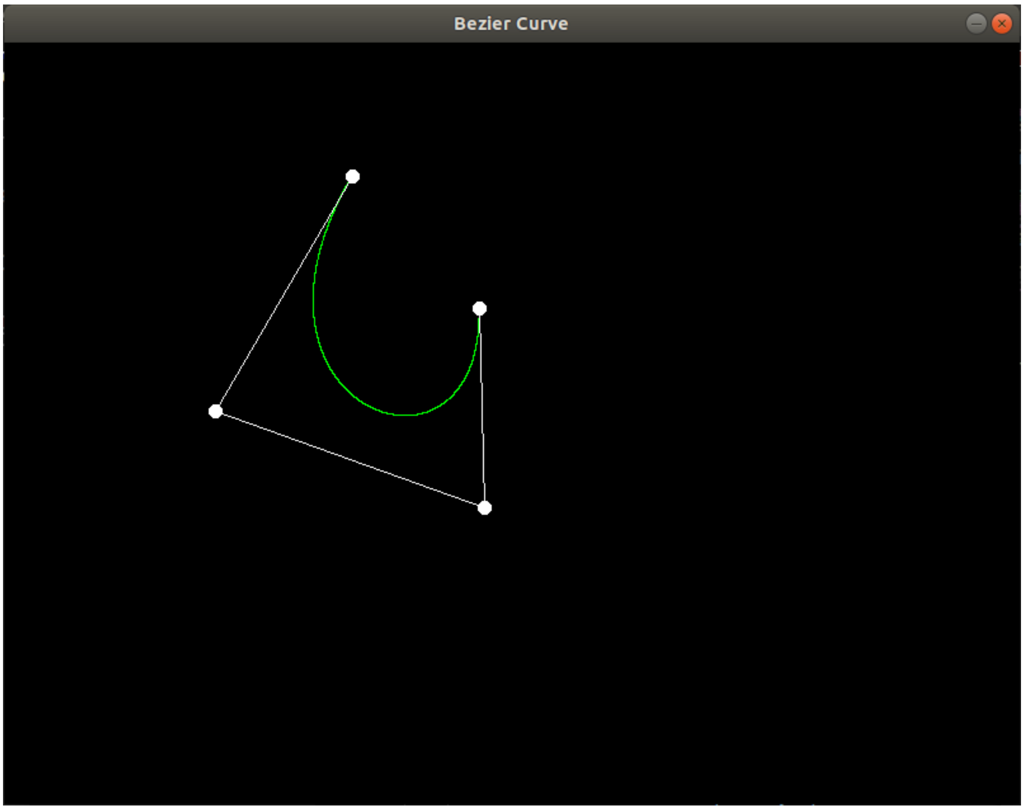

图 1: 4 个控制点的 Béizer 曲线（1）

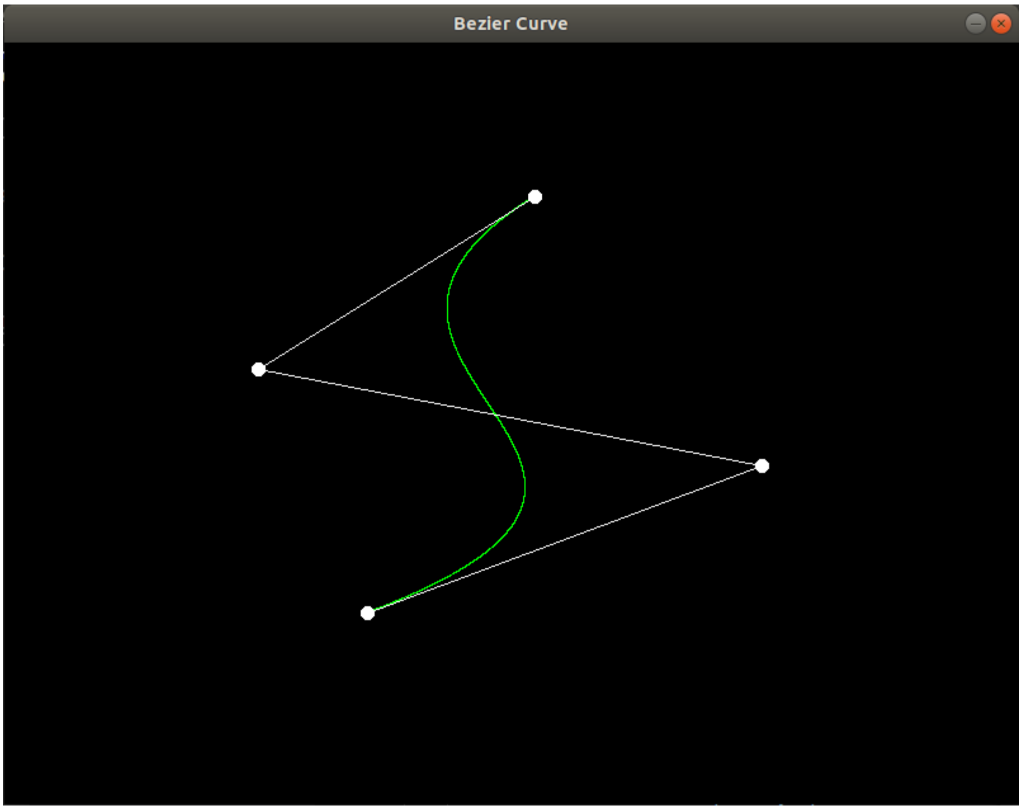

图 2: 4 个控制点的 Béizer 曲线（2）

2.2 Task 2：调整代码支持更多顶点

为方便后续修改，我首先设置标识符 *point_num* 用于标识控制点数量。为了支持更多的顶点，我们总共需要修改 mouse_handler 函数和 main 函数。

mouse_handler 函数用于获得鼠标点击得到的位置坐标。修改如下：

```
void mouse_handler(int event, int x, int y, int flags, void *userdata)
{
    if (event == cv::EVENT_LBUTTONDOWN && control_points.size() < point_num)
    {
        std::cout << "Left button of the mouse is clicked - position (" << x << ", "
        << y << ")" << '\n';
        control_points.emplace_back(x, y);
    }
}
```

通过设置 size 小于 point_num，使得我们可以获得更多的控制点坐标。main 函数中，当其检测到 *control_points* 数量达到规定值时开始显示曲线。代码修改如下：

```
int main()
{
    ......
    while (key != 27)
    {
        ......
        if (control_points.size() == point_num)
```

```
        {
            //naive_bezier(control_points, window);
            bezier(control_points, window);

            cv::imshow("Bezier Curve", window);
            cv::imwrite("my_bezier_curve.png", window);
            key = cv::waitKey(0);

            return 0;
        }
        ......
    }
return 0;
}
```

设置 *point_num*，编译运行代码，实验结果如图3和4所示，可见正确运行。

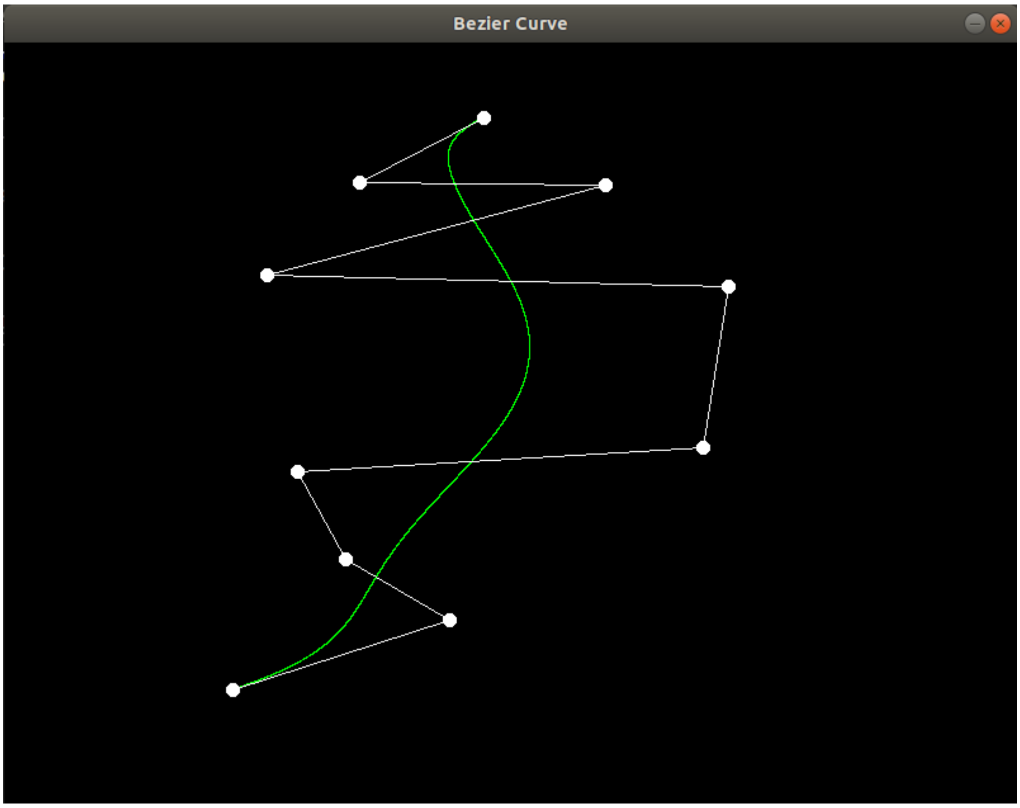

图 3: 10 个控制点的 Béizer 曲线（1）

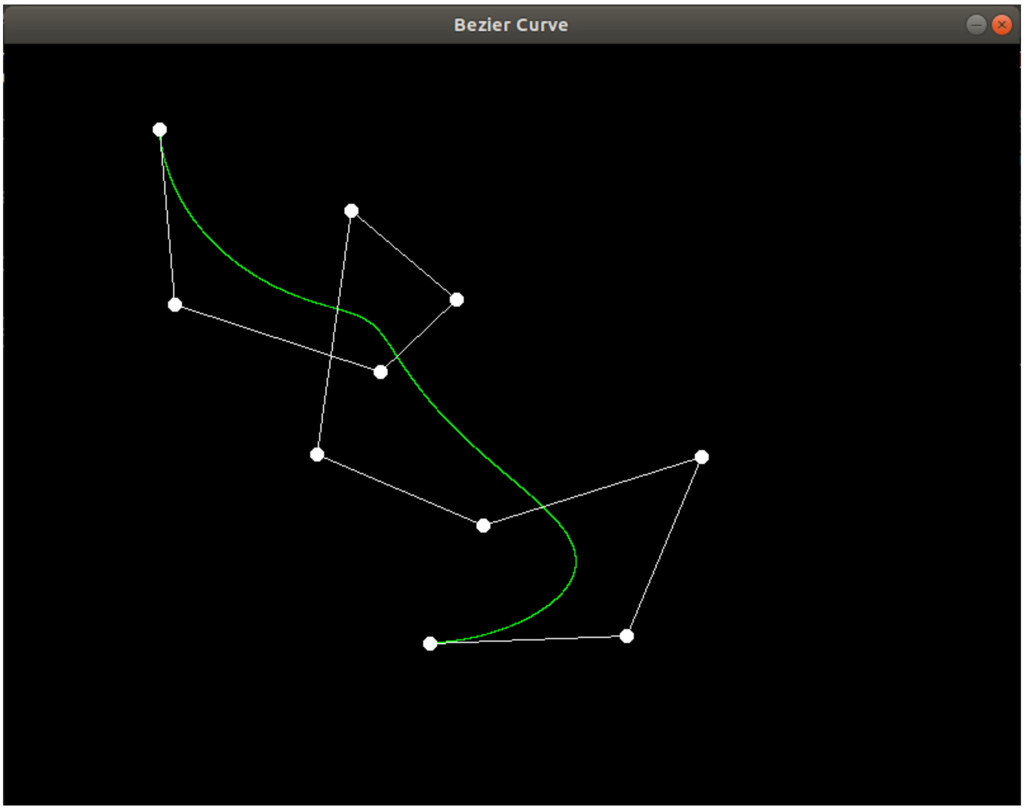

图 4: 10 个控制点的 Béizer 曲线（2）

2.3 Task 3：谈谈对 Béizer 曲线的理解

Béizer 曲线相当于一种逼近曲线，其灵活性强，通过调整控制点来得到相应的曲线。本质上而言，Béizer 曲线是由调和函数根据控制点插值生成，每个控制点对应一个 Bernstein 基函数，$n+1$ 个控制点可以生成 n 次多项式。在 $[0,1]$ 范围中，组成该 Béizer 曲线的基函数和恒为 1。根据其公式，我们可以得知 Béizer 曲线的以下特性：

(1) Béizer 曲线一定通过特征多边形的起点和终点；

(2) 对称性：只要保持特征多边形顶点位置不变，即使顺序颠倒、所得的新 Béizer 曲线形状不变，但参数变化方向相反；

(3) 凸包性：Béizer 曲线一定落在其控制多边形的凸包中；

(4) 仿射不变性：Béizer 曲线的形状不随坐标变换而变换，只与各控制顶点的相对位置有关；

虽然对 Bernstein 基函数的计算十分复杂，但我们可以通过一种递归的方式：de Casteljau 算法完成 Béizer 曲线的绘制，大大减少工作量。

不过，Béizer 曲线也存在一些缺点，例如一旦一个控制点发生了移动，就会导致整条曲线发生变化，即过于灵活、“牵一发而动全身”；另一方面，控制点的数量决定曲线次数，这就导致了为了获得一条曲线、其函数次数往往过高。因此，学者们也提出了很多种改进方法，如分段 Béizer 曲线等。

3 实验感想

本次作业是图形学的最后一次作业，难度不是很大，代码量不多，比较顺利地做完了。在理解 de Casteljau 算法的递归思想中我花了比较多的时间，但一旦理解了其思想与做法，代码的完成也就轻而易举了。

在此感谢老师和助教提供的帮助，希望我能顺利完成期末 project。