

实验四

实验目的

- 使用de Casteljau算法来绘制Bézier 曲线
- 提高部分：实现对Bézier 曲线的反走样

实验过程

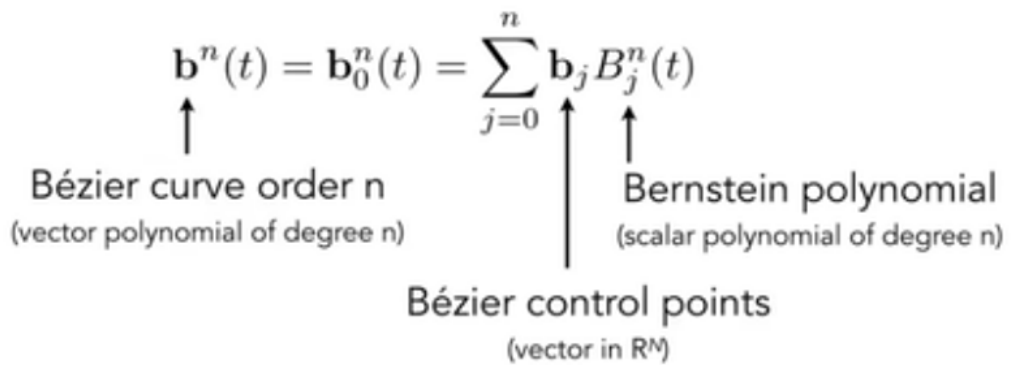
1. 使用de Casteljau算法来绘制Bézier 曲线

代码框架给出了naive_bezier的实现方法，实际上是三次Bézier 曲线的公式法绘图，公式如下：

Bézier Curve – General Algebraic Formula

Bernstein form of a Bézier curve of order n:

$$\mathbf{b}^n(t) = \mathbf{b}_0^n(t) = \sum_{j=0}^n \mathbf{b}_j B_j^n(t)$$




Bézier curve order n
(vector polynomial of degree n)

Bézier control points
(vector in \mathbb{R}^M)

Bernstein polynomial
(scalar polynomial of degree n)

Bernstein polynomials:


$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}$$

使用de Casteljau算法来绘制Bézier 曲线，我们可以得到任意次数的Bézier 曲线。算法说明如下：

- 考虑一个 p_0, p_1, \dots, p_n 为控制点序列的 Bézier 曲线。首先,将相邻的点连接起来以形成线段
- 用 $t : (1-t)$ 的比例细分每个线段,并找到该分割点
- 得到的分割点作为新的控制点序列,新序列的长度会减少一
- 如果序列只包含一个点,则返回该点并终止。否则,使用新的控制点序列并转到步骤 1
- 使用 $[0,1]$ 中的多个不同的 t 来执行上述算法,就能得到相应的 Bézier 曲线

我们需要修改的有两个函数bezier和recursive_bezier。

```

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.
    std::vector<cv::Point2f> point_set;
    for(float t=0; t<=1; t+=0.001)
    {
        auto point = recursive_bezier(control_points, t);
        window.at<cv::Vec3b>(point.y, point.x)[4] = 255;
    }
}

```

bezier函数包含两个操作：递归计算bezier曲线上的点和绘图。

```

cv::Point2f recursive_bezier(const std::vector<cv::Point2f> &control_points,
float t)
{
    // TODO: Implement de Casteljau's algorithm
    std::vector<cv::Point2f> cp;
    for(int i=0; i<=control_points.size()-1; i++)
    {
        cp.push_back(control_points[i]);
    }
    int l = cp.size();
    while(l!=1)
    {
        for(int i=0; i<l-1; i++)
        {
            cp[i] = cp[i] + t * (cp[i+1] - cp[i]);
        }
        l--;
    }
    return cp[0];
}

```

recursive_bezier函数就是de Casteljau算法的实现过程，采用迭代的思想来避免函数递归。

2. 代码执行

```

mkdir build
cd build
cmake ..
make -j8
./BezierCurve

```

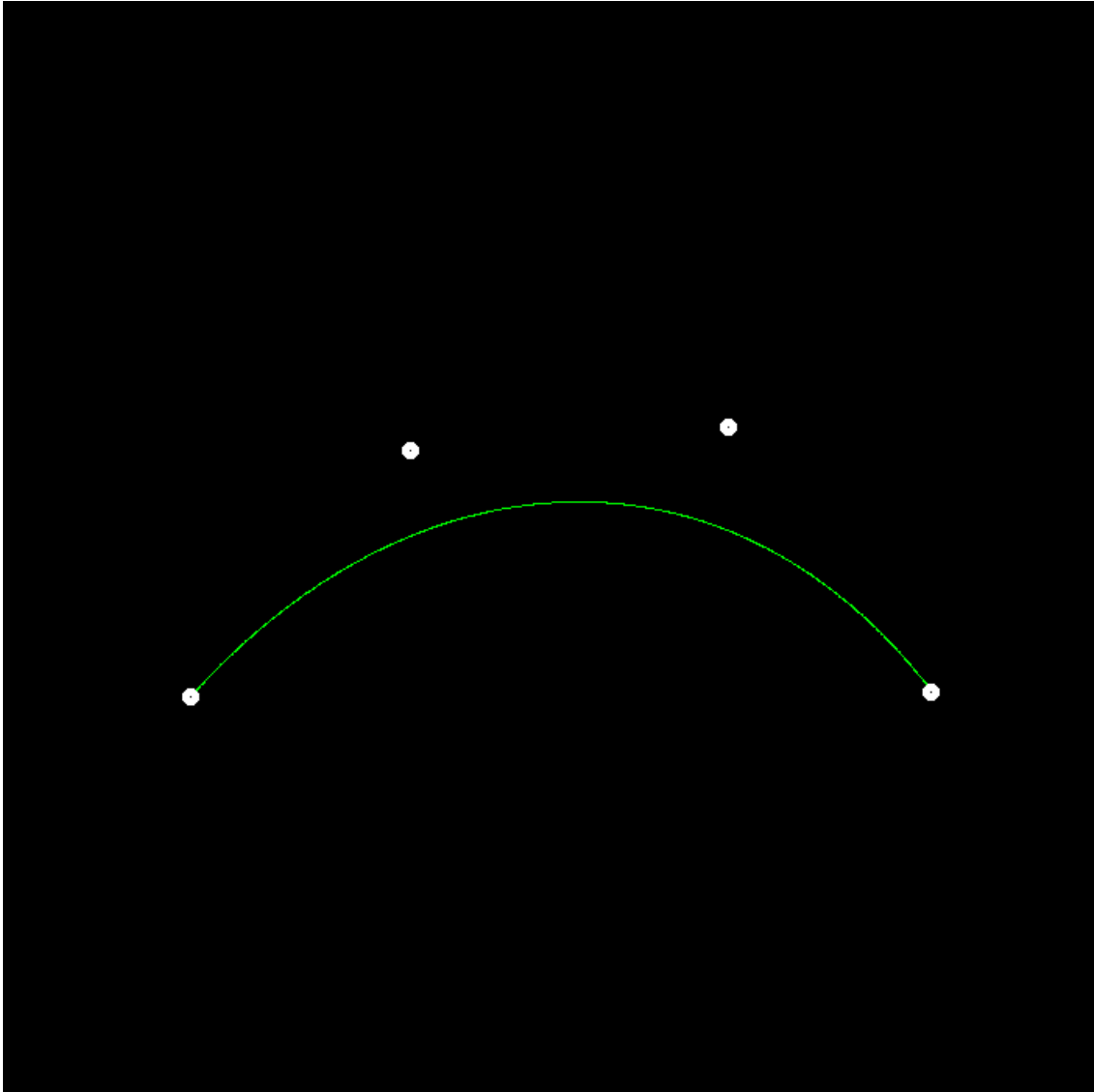
编译成功：

```

cs18@games101vm:~/Mnt/ngifs/GAMES101/4/code/build$ make -j8
Scanning dependencies of target BezierCurve
[ 50%] Building CXX object CMakeFiles/BezierCurve.dir/main.cpp.o
[100%] Linking CXX executable BezierCurve
[100%] Built target BezierCurve

```

结果：



曲线为黄色说明de Casteljau算法和naive bezier的实现重叠，结果正确。

3. 提高：实现对Bézier 曲线的反走样

所谓反走样即降低图像分辨率的过程，通过抗锯齿等一系列手段，让图像看起来更清晰。指导书给出了实现的思路：对于一个曲线上的点,不只把它对应于一个像素,你需根据到像素中心的距离来考虑与它相邻的像素的颜色。

```
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.
    std::vector<cv::Point2f> point_set;
    for(float t=0; t<=1; t+=0.001)
    {
        auto point = recursive_bezier(control_points, t);
        // 找到与曲线上的点相邻的四个像素
        cv::Point2f point00 = {std::floor(point.x), std::floor(point.y)};
        cv::Point2f point01 = {std::floor(point.x), std::ceil(point.y)};
        cv::Point2f point10 = {std::ceil(point.x), std::ceil(point.y)};
        cv::Point2f point11 = {std::ceil(point.x), std::floor(point.y)};

        // 找出四个像素到曲线点的距离的最小值， 并根据与曲线点的距离赋值颜色
    }
}
```

```

float dmin = p2p(point, point00);
window.at<cv::Vec3b>(point00.y, point00.x)[4] = std::max(255*(1 -
dmin), (float>window.at<cv::Vec3b>(point00.y, point00.x)[4]);
float d2 = p2p(point, point01);
if(d2 < dmin)
{
    window.at<cv::Vec3b>(point01.y, point01.x)[4] = std::max(255*(1 -
d2), (float>window.at<cv::Vec3b>(point01.y, point01.x)[4]);
    dmin = d2;
}
float d3 = p2p(point, point10);
if(d3 < dmin)
{
    window.at<cv::Vec3b>(point10.y, point10.x)[4] = std::max(255*(1 -
d3), (float>window.at<cv::Vec3b>(point10.y, point10.x)[4]);
    dmin = d3;
}
float d4 = p2p(point, point11);
if(d4 < dmin)
{
    window.at<cv::Vec3b>(point11.y, point11.x)[4] = std::max(255*(1 -
d4), (float>window.at<cv::Vec3b>(point11.y, point11.x)[4]);
}
}
}

```

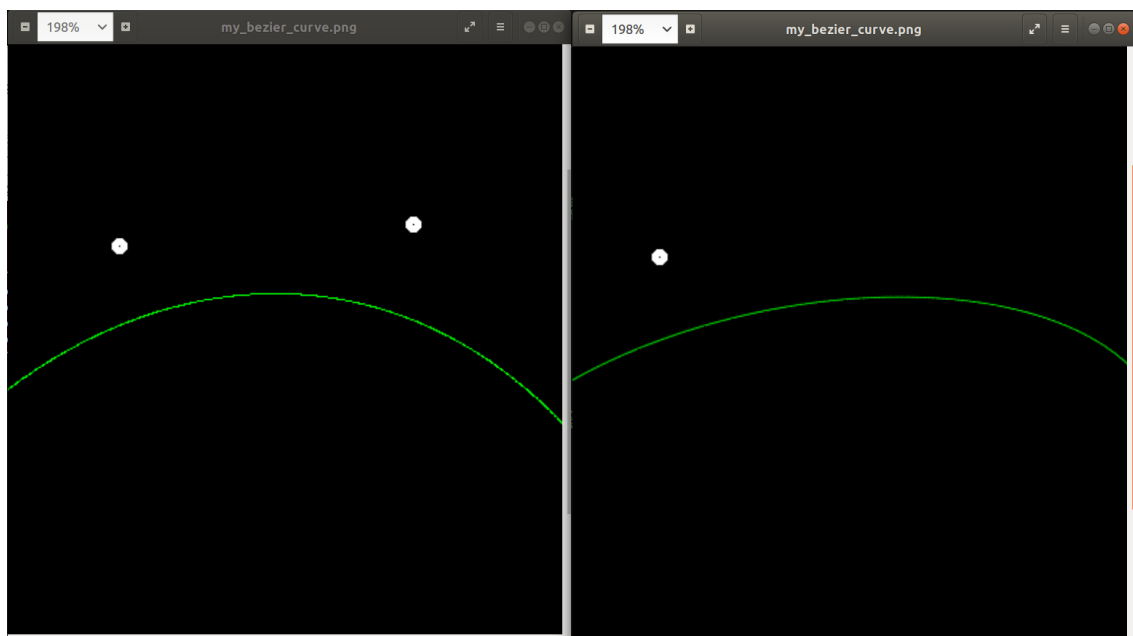
我们可以通过floor()和ceil()这两个取整函数来找到与曲线点最近的四个像素点，分别计算四个像素点到曲线点的距离，找到距离最小的像素点，根据它离曲线点的距离赋给它不同的绿色（距离越远绿色越弱）。距离的计算我封装了一个函数，

```

float p2p(cv::Point2f point1, cv::Point2f point2)
{
    return std::sqrt(std::pow(point1.x - point2.x, 2) + std::pow(point1.y -
point2.y, 2));
}

```

反采样前后的对比：



采样后的图像锯齿程度更小。

实验总结

手动实现de Casteljau算法的过程中，由于函数传递参数control_points是一个静态值，不可修改，导致在用递归实现时一直行不通，最后选择了迭代的思想，创建了一个control_points的副本，在一个函数内循环解决。在提高部分由于没有理解反采样和超采样的区别，一开始选择了MSAA的算法解决曲线锯齿的问题，但是行不通，最后仔细理解指导书给出的思路，结合网上论坛的思路才实现。