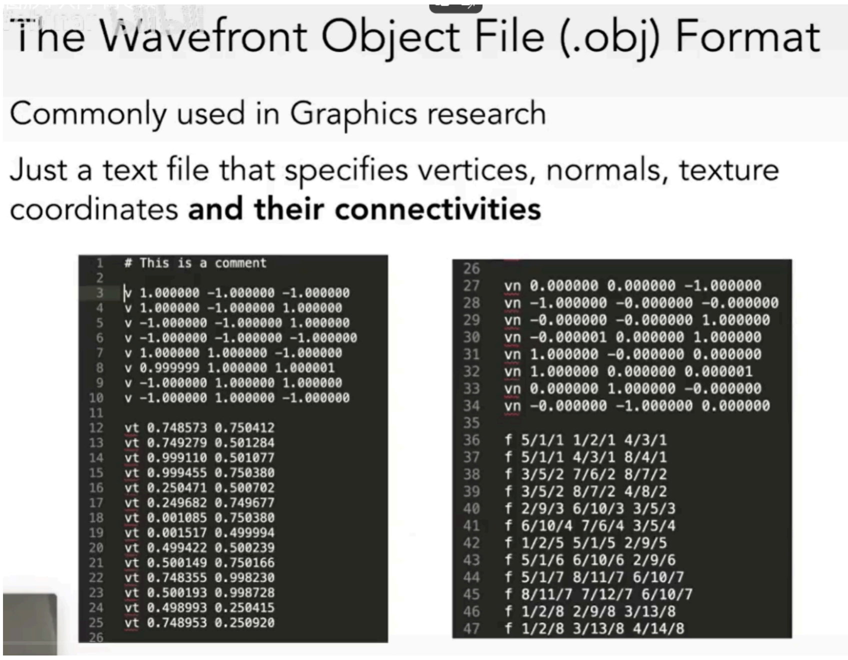
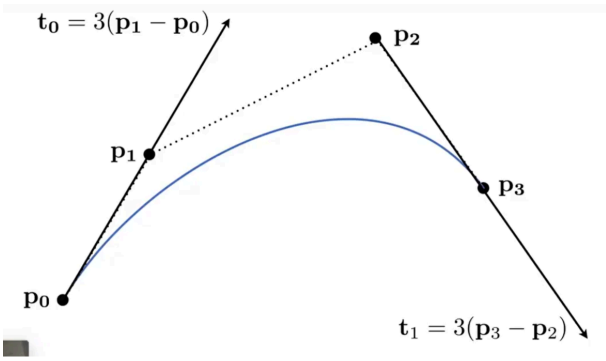


`.obj` 文件的格式，看上去还挺文本化的。

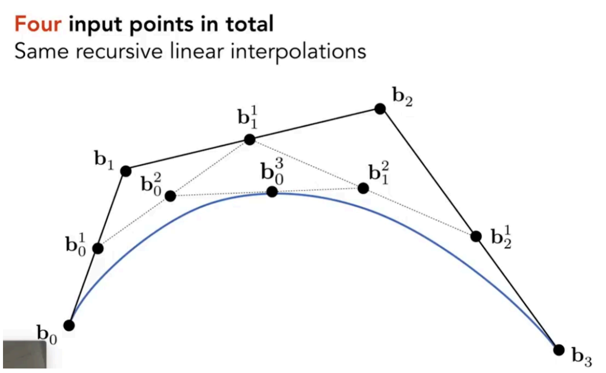
Curve

贝塞尔曲线：用控制点定义唯一的一条曲线

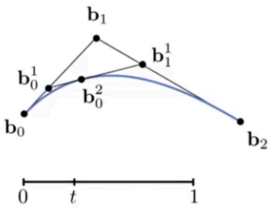


算法过程是将四点三线转化为三点二线，再转化为两点一线，最后转化为一点零线，这个最终的点就是曲线上的某一点。每次选取的线段上的点都是一个固定比例的插值。

这个比例就是时间，通过在这个插值比例上不断采样，就能画出最终的曲线。



Example: quadratic Bézier curve from three points



$$\mathbf{b}_0^1(t) = (1-t)\mathbf{b}_0 + t\mathbf{b}_1$$
$$\mathbf{b}_1^1(t) = (1-t)\mathbf{b}_1 + t\mathbf{b}_2$$

$$\mathbf{b}_0^2(t) = (1-t)\mathbf{b}_0^1 + t\mathbf{b}_1^1$$

$$\mathbf{b}_0^2(t) = (1-t)^2\mathbf{b}_0 + 2t(1-t)\mathbf{b}_1 + t^2\mathbf{b}_2$$

代数上看就是定义了一个关于时间 t 的多次函数，只不过函数的系数和输出值是点的坐标罢了。

具体的定义如下，嗯，看来二项式系数相关的组合数学还是得学的。

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)

Bernstein polynomial
(scalar polynomial of degree n)

Bézier control points
(vector in R^N)

Bernstein polynomials:

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

Bezier 曲线的好性质是指对于仿射变换 F，F(Curve(points)) 等于 Curve(F(points))，这就大大减少了仿射变换的运算量。(对非仿射变换可不一定，比如对投影变换就不成立)

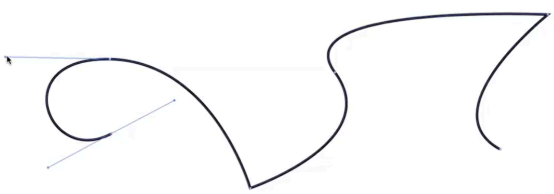
凸包……计算几何，启动!

而 Bezier 曲线一定在控制点的凸包内。

常用的是四个控制点的三次 bezier 曲线，然后拼起来。

Instead, chain many low-order Bézier curve

Piecewise cubic Bézier the most common technique



[redacted]dely used (fonts, paths, Illustrator, Keynote, …)

接下来提到了一些参数连续性和几何连续性的东西，可以看

https://zhuanlan.zhihu.com/p/682706735

接下来是另一种定义曲线的方法。

splines 样条：一条可控的曲线

B-splines (Basis Splines)，是贝塞尔曲线的超集。

老师说太难了不讲了……

还有一个 NURBS (非均匀有理 B-样条，Non-Uniform Rational Basis Splines)

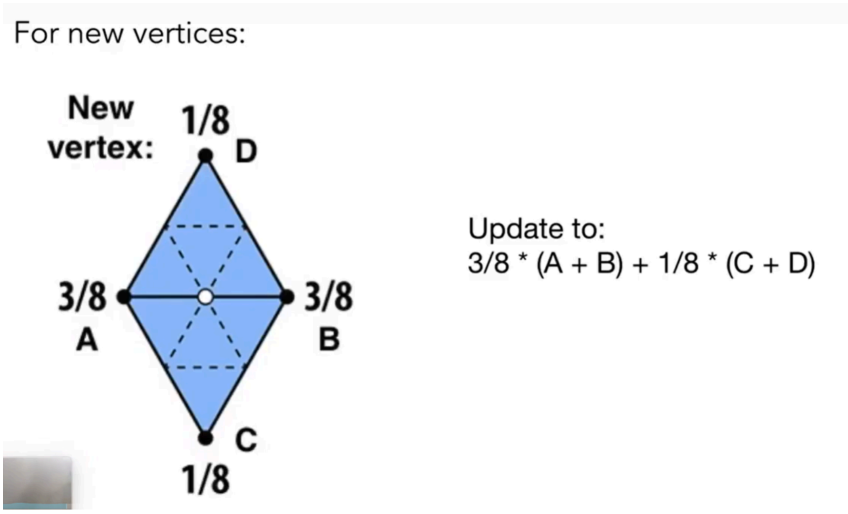
课上提到的课是

https://www.bilibili.com/video/av66548502

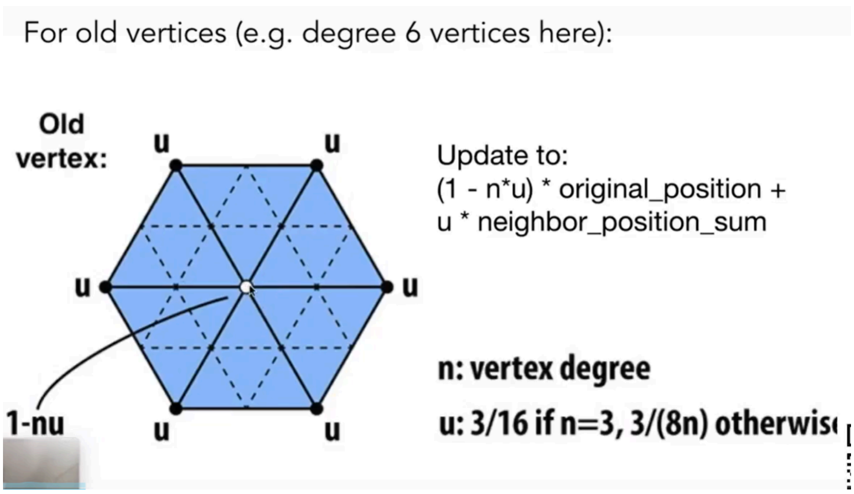
三角形增加

Loop（没想到吧，这个 Loop 是个姓） Subdivision：将一个三角形分成四个，新增的顶点和原来的顶点按照不同的规则改变位置

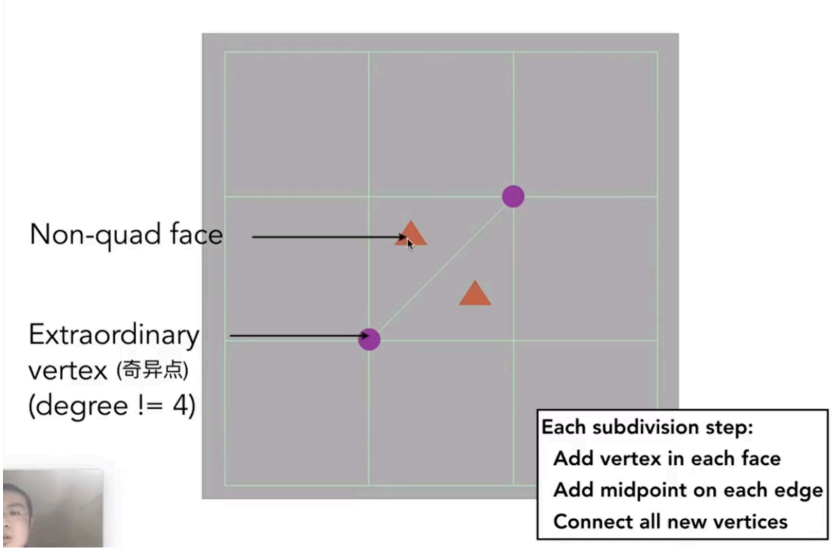
新增的点总会是某两个三角形共享边上的点。



旧的点就取这个旧点自己和其相邻旧点的加权平均。

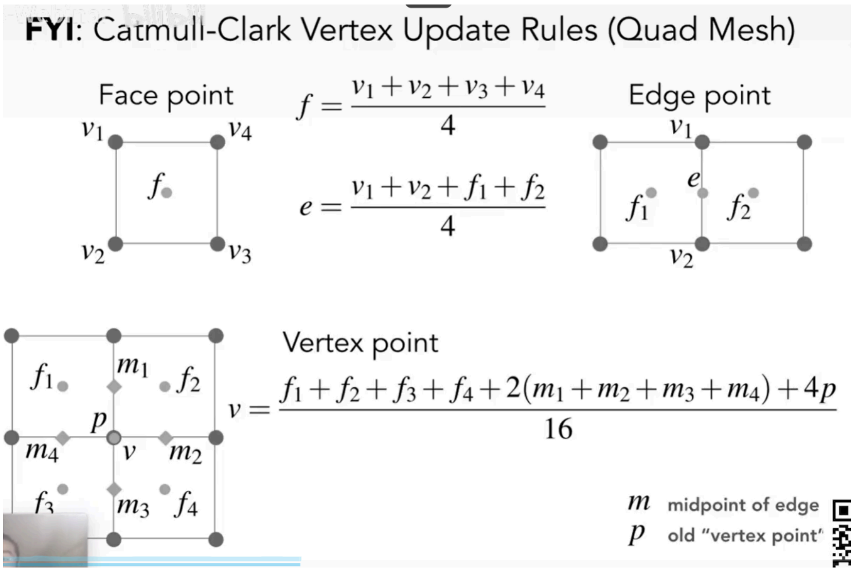


Catmull-Clack Subdivision：对于更一般的模型，比如有四边形面的模型。



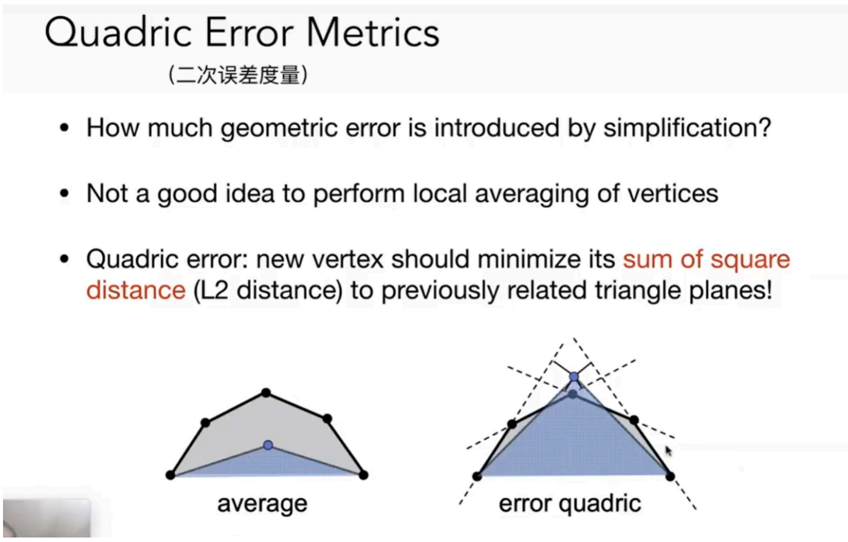
在第一次迭代后所有非四边形面就消失了，伴随着引入同样数量的奇异点。而之后的迭代中显然不会再引入奇异点了。

看一眼公式。



三角形减少

`edge collapsing`: 边塌缩



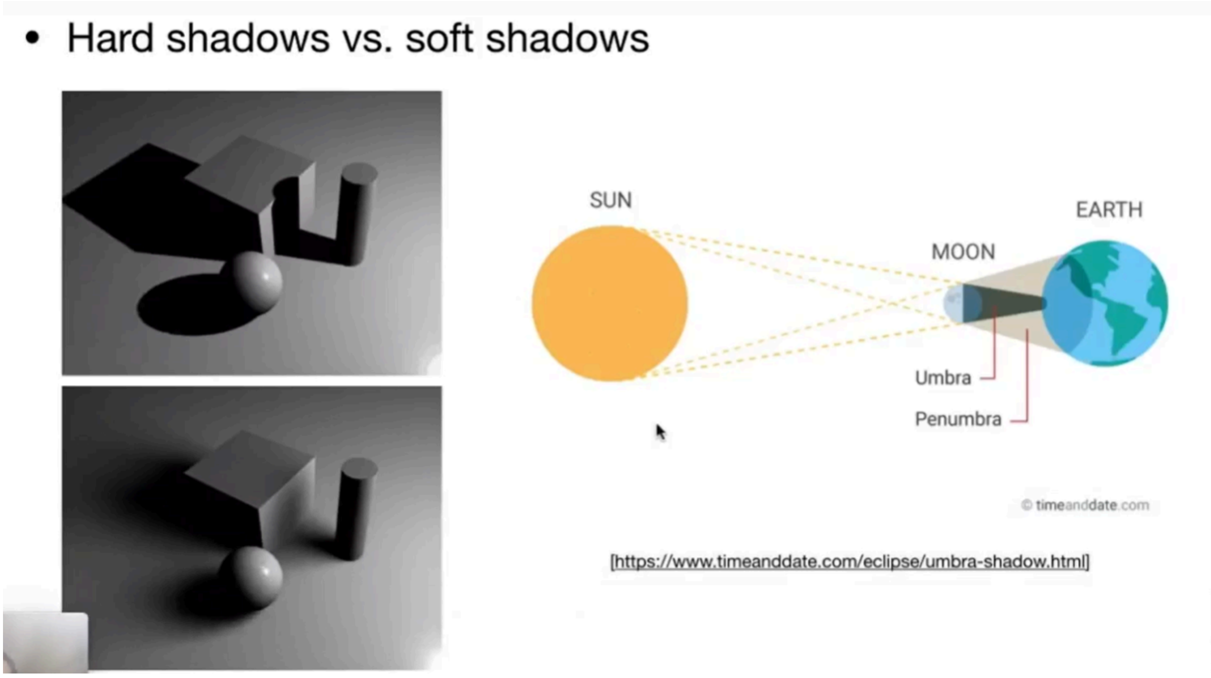
光栅化下的物体间遮挡阴影阴影

`Shadow Mapping`

- Key idea:
  - the points NOT in shadow must be seen both **by the light** and **by the camera**

虽然有很多问题，且基本用来做硬阴影。

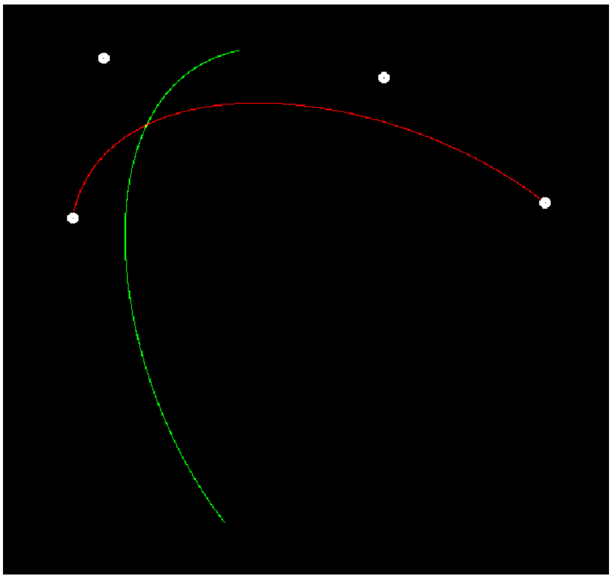
软阴影的产生是因为光源有一定的大小，所以点光源产生不了软阴影。



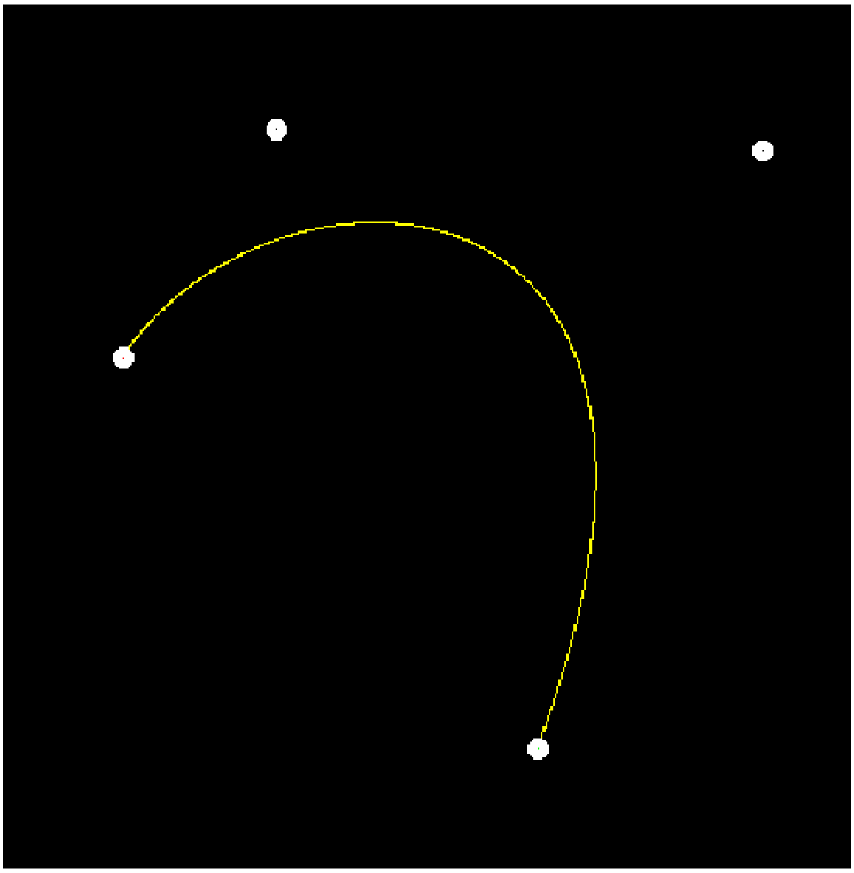
接下来做作业 4。

笑死了还整了个 naive_bezier

首先随便写了一下结果成这样了



后来核对了下 naive_bezier 发现 window.draw 的时候 point.y 和 point.x 写反了……



```
cv::Point2f recursive_bezier(const std::vector<cv::Point2f> &control_points, float t)
{
    int n = control_points.size();
    if(n == 1) return control_points[0];

    std::vector<cv::Point2f> next_level_points;
    // Calculate the linear interpolation between each pair of points
    for (size_t i = 0; i < n - 1; ++i) {// 这里是直接丢弃最后一个点，原本是想在数组本身修改的，但由于原本是用作 const，所以就新开数组了
        cv::Point2f p1 = control_points[i];
        cv::Point2f p2 = control_points[i + 1];
        next_level_points.emplace_back(p1 * t + p2 * (1 - t));
    }
    return recursive_bezier(next_level_points, t);
}

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.
    for (float t = 0.0f; t <= 1.0f; t += 0.001f)
    {
        cv::Point2f point = recursive_bezier(control_points, t);
        window.at<cv::Vec3b>(point.y, point.x)[1] = 255;//没想到通道 1 就是 green，我一次就猜对了哈哈哈
    }
}
```