计算机图形学 PJ1

姓名: 陈锐林, 学号:21307130148

2024 年 4 月 16 日

一、曲线绘制准备工作：

1. 对于贝塞尔曲线的理解：

其实看完 PJ 说明一个头两个大，不知道这玩意是干啥的；在看了教材以及一篇文章 (point here) 之后就悟了。贝塞尔曲线就是通过几个点拉扯控制点的生成；一阶由两个点控制，就是 $(1-t)P_0 + tP_1$，二阶是先在 P_0 和 P_1 和 P_1 和 P_2 之间各做一次如上操作，最后根据这两段生成的点再做一次。

于是类似的，三阶方程就是四个一组，方程是 $P_0(1-t)^3 + 3P_1t(1-t)^2 + 3P_2t^2(1-t) + P_3t^3$；这也就是后面矩阵形式的由来。除此之外，B 样条曲线就是类似地，在贝塞尔曲线的基础上做了优化，更改了迭代的表达式。

2. 对绘图中用到的方法和程序：

对于绘曲线，我们需要用到 V, T, N, B，其中 $V_i = q(t_i), T_i = q'(t_i)$（某点的值和导数）；而其余两个更新方法是 $N_i = B_{i-1} \times T_i, B_i = T_i \times N_i$。

在学习了 evalcircle 函数后，就知道我们只要根据计算公式返回一个 Curve 给绘制函数即可，Curve 的每个成员都存储了各自的 V, T, N, B；只要懂得上面的表示和意义就能很好做出了。具体的见后面代码分析。

二、完成函数 evalBezier()：

1. 完成思路：

注意到输入是点集 P，和 $steps$。根据前面的分析，三阶贝塞尔曲线 4 个点为 1 组，又因为要维护曲线的平滑，于是前一组的最后一个点是后一组的第一个点。总共是 $3k+1$ 个点 $(k \geq 1)$。所以可用 $|P|/3$ 得到要生成的组数；而参数 $steps$ 标识了每组要生成多少个点，相当于把 $t \in [0,1]$ 作了细分，就得到 t 的值。

2. 具体代码：

Curve 变量 Bezier 存储生成的点，i 标识现在求解第 i 组的点，j 用来求取 t(将其强制转化为 double 型计算步长)；$i \times steps + j$ 就是要填入的 Bezier 位置，其余的就是代入计算四个变量；需要注意的是 B 的初始是 $(0,0,1) \times T$（如课件建议的）。

```
int ngroups = P.size() / 3;
Curve Bezier(ngroups * steps);

for (int i = 0; i < ngroups;i++)
{

    for (int j = 0; j < steps; j++)
    {
        double t = double(j) / steps;
        /*group 0 use points : 0,1,2,3;  group 1 use : 3,4,5,6;... group i use: 3 * i, 3 * i + 1, 3 * i + 2, 3 * i + 3*/
        /*update V, need normalization*/
        Bezier[steps * i + j].V = (1 - t) * (1 - t) * (1 - t) * P[3 * i] + 3 * t * (1 - t) * (1 - t) * P[3 * i + 1] +
            3 * t * t * (1 - t) * P[3 * i + 2] + t * t * t * P[3 * i + 3];
        /*update T, need normalization*/
        Bezier[steps * i + j].T = (-3 * (1 - t) * (1 - t) * P[3 * i] + 3 * (1 - 3 * t) * (1 - t) * P[3 * i + 1] +
            3 * t * (2 - 3 * t) * P[3 * i + 2] + 3 * t * t * P[3 * i + 3]).normalized();
        /*update N, need special judgement, need normalization*/
        if(!(i == 0 && j == 0)){
            Bezier[steps * i + j].N = Vector3f::cross(Bezier[steps * i + j - 1].B, Bezier[steps * i + j].T).normalized();
        }
        else Bezier[steps * i + j].N = Vector3f::cross(Vector3f(0, 0, 1), Bezier[steps * i + j].T).normalized();
        /*update B, need normalization*/
        Bezier[steps * i + j].B = Vector3f::cross(Bezier[steps * i + j].T, Bezier[steps * i + j].N).normalized();
    }
}
```

三、完成函数 evalBspline()：

1. 完成思路：

注意到输入是很相似的，只需要搞定和 Bezier 的不同点就可以。同样是 4 个点为一组生成，但是 B 样条曲线中前一组的后 3 个点都会被复用，所以最终会生成的组数是 $|P|-3$。t 的处理一样。

2. 具体代码：

Curve 变量 Bspline 存储生成的点，i 仍控制第几组，其余就是类似的代入计算（填入位置的是 $i \times steps + j$）。但是需要注意的是 B 样条曲线有些时候需要闭合，即在给出的 P 中，前三个点和后三个点重合了，这时候需要在返回 Bspline 之前做一个特判，代码如下：

```
int ngroups = P.size() - 3;
Curve Bspline(ngroups * steps);

for (int i = 0; i < ngroups; i++)
{
    for (int j = 0; j < steps; j++)
    {
        double t = double(j) / steps;
        /*group 0 use points : 0,1,2,3;  group 1 use : 1,2,3,4;... group i use: i,i + 1,i + 2,i + 3*/
        /*update V, need normalization*/
        Bspline[steps * i + j].V = (1 - t) * (1 - t) * (1 - t) / 6.0f * P[i] + (4 - 6 * t * t + 3 * t * t * t) / 6.0f * P[i + 1]
            + (1 + 3 * t + 3 * t * t - 3 * t * t * t) / 6.0f * P[i + 2] + (t * t * t) / 6.0f * P[i + 3];
        /*update T, need normalization*/
        Bspline[steps * i + j].T = (-(1 - t) * (1 - t) / 2.0f * P[i] + (-4 * t + 3 * t * t) / 2.0f * P[i + 1] +
            (1 + 2 * t - 3 * t * t) / 2.0f * P[i + 2] + (t * t) / 2.0f * P[i + 3]).normalized();
        /*update N, need special judgement, need normalization*/
        if(!(i == 0 && j == 0)){
            Bspline[steps * i + j].N = Vector3f::cross(Bspline[steps * i + j - 1].B, Bspline[steps * i + j].T).normalized();
        }
        else Bspline[steps * i + j].N = Vector3f::cross(Vector3f(0, 0, 1), Bspline[steps * i + j].T).normalized();
        /*update B, need normalization*/
        Bspline[steps * i + j].B = Vector3f::cross(Bspline[steps * i + j].T, Bspline[steps * i + j].N).normalized();
    }
}
/*the tail may be connected to head, so need special judgement*/
if (P[ngroups] == P[0] && P[ngroups + 1] == P[1] && P[ngroups+ 2] == P[2])
    Bspline.push_back(Bspline[0]);
```

** 输出图例都放在最后了 **

四、曲面绘制准备工作：

1. 实现目标：

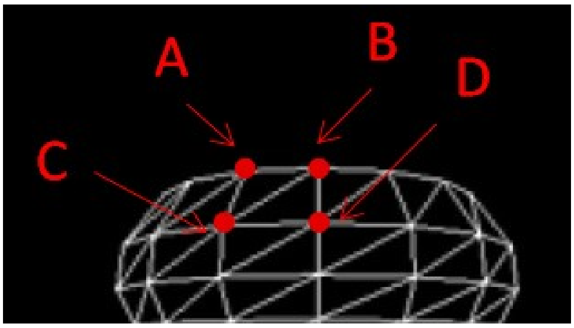
通过阅读 pj 的说明，能注意到曲面绘制和曲线绘制是有相似点的。旋转曲面绘制中，同样是根据输入的点坐标，完成对后续点的生成；但是曲线绘制时是 4 个点生成 1 组点（$steps$ 个），这是以轮廓 $profile$ 上每一个点都转一周。而 ppt 里已经给出了旋转的公式，计算 M, P', N'。

对于广义圆柱体的绘制，同样要利用 $profile$ 生成一些点，但这些点不再由 $steps$ 控制，而是根据输入的扫描曲线 $sweep$ 的坐标进行变换。

回到这两个函数的输入和输出，我们要返回的是 Surface 类型用于存储生成的点；含 VV, VN, VF 的三个 vector，对应：点、法线、相邻 4 点连成的三角形。注意到广义圆柱体涉及到取矩阵逆、转置、子矩阵的操作；而这些函数在 vecmath 库里都有对应的。

2. 三角形网格划分函数：

对于这两个任务，我们都要把采样生成的点连接（才算是面），所以下面给出函数用于划分三角形网格。



InsertTriangle 函数完成这个任务，就是把 A, B, C, D 这 4 个点分组相连；若 A 点索引是 x，则 B, C, D 分别是 $x+1, x+sz2, x+sz2+1$（$sz2$ 是 $profile$ 的大小）；再利用 Surf.h 中的 $Tup3u$ 类型就可以完成插入。当然要注意 A 不能是 $profile$ 最后一个点，需要特判。具体代码如下：

```
void InsertTriangle(Surface &surface, const int &sz2)
{
    /*two input, formed SUrface and size of profiles (need special input)*/
    int i = 0;
    int sz1 = surface.VV.size();
    /*use to size to control*/
    while (i < sz1 - sz2)
    {
        if ((i + 1) % sz2 != 0)
        {   /*according to the define in surf.h*/
            surface.VF.push_back(Tup3u(i, i + 1, i + sz2));
            surface.VF.push_back(Tup3u(i + sz2, i + 1, i + sz2 + 1));
        }
        i++;
    }
}
```

五、完成函数 makeSurfRev：

1. 实现过程：

如上说明的，代码逻辑就是三部分。利用 $profile$ 的数据，根据步长使用旋转公式生成点；加入原 $profile$ 的点，完成闭合；调用函数 InsertTriangle 完成三角网格绘制。

其中第一步同样使用 j 完成步长 t 的生成，每次循环对 $profile$ 上的每个点旋转生成 1 次；共进行 $steps$ 次。

2. 具体代码：

这里与 InsertTriangle 相对应，$sz2$ 声明为 $profile$ 的大小。在第一个 for 循环中，和 curve.cpp 一样声明了圆周率 c_pi；并使用已有的接口生成步长 t 对应的正余弦值。

```
int sz2 = profile.size();
for(int i = 0; i < steps; i++)
{
    /*use vecmath api to calculate t / cost / sint, used to rotate */
    double t = 2.0f * c_pi * double(i) / steps;
    double ccos = cos(t);
    double ssin = sin(t);
    /*for every point in profile, advance once per circle */
    for (int j = 0; j < sz2; j++)
    {
        surface.VV.push_back(Vector3f(ccos * profile[j].V[0] + ssin * profile[j].V[2],
            profile[j].V[1], -ssin * profile[j].V[0] + ccos * profile[j].V[2]));
        surface.VN.push_back(Vector3f(-ccos * profile[j].N[0] - ssin * profile[j].N[2],
            -profile[j].N[1], ssin * profile[j].N[0] - ccos * profile[j].N[2]));
    }
}
for(int k = 0; k < sz2; k++)
{
    surface.VV.push_back(surface.VV[k]);
    surface.VN.push_back(surface.VN[k]);
}
InsertTriangle(surface, sz2);
```

六、完成函数 makeGenCyl：

1. 实现过程：

逻辑没有很大的改变，只是 $steps$ 的循环由 $sweep$ 代替。这里就没有步长 t 的概念，而是根据每一个 $sweep$ 的点生成新的点。所以这里也不必加入原有的点，而是只保留新生成的就可以。

对每一个点 a, $\boldsymbol{M}_a = \begin{bmatrix} \boldsymbol{N}_a & \boldsymbol{B}_a & \boldsymbol{T}_a & \boldsymbol{V}_a \\ 0 & 0 & 0 & 1 \end{bmatrix}$,之后再生成 $\boldsymbol{N}' = normalize((\boldsymbol{M}^{-1})^T \boldsymbol{N})$。

2. 具体代码：

第一层循环中，我们利用公式和定义得到前面的矩阵 Ma，之后取其一小部分得到子矩阵 $subMa$。这里最后省略了 InsertTriangle。

```
int sz2 = profile.size();
int sz3 = sweep.size();
for(int i = 0; i < sz3; i++)
{
    Matrix4f Ma(Vector4f(sweep[i].N, 0), Vector4f(sweep[i].B, 0),
        Vector4f(sweep[i].T, 0), Vector4f(sweep[i].V, 1));
    Matrix3f subMa = Ma.getSubmatrix3x3(0, 0);
    for (int j = 0; j < sz2; j++)
    {
        surface.VV.push_back((Ma * Vector4f(profile[j].V, 1)).xyz());
        surface.VN.push_back(-1 * subMa * profile[j].N);
    }
}
```

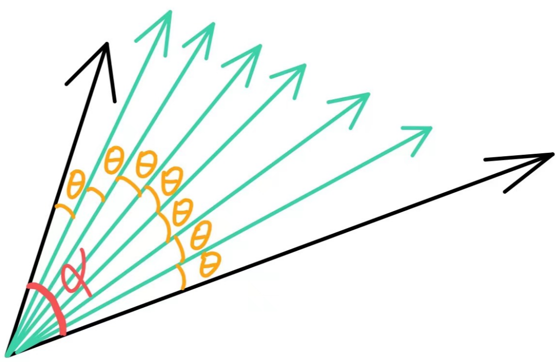
七、拓展部分，解决曲面闭合问题：

1. 思路：

解决两个问题，一是什么时候发生了这种情况；二是怎么弥补。

首先输入图形本来应该是闭合的，如果本来不是闭合曲线就无须处理（闭合，即：切向量相同）；其次首尾的法向量应该是不同的。在实际样例中我们也能看到，α 要比较大才会对输出有影响，所以这里加一个限制为 $\alpha > 0.2$

解决方法就是 PPT 上介绍的旋转插值，可以理解为要把下图这一撮向量都移到同一条线上；其中 α 就是始末向量法向量的夹角，θ 是均分后的结果。从右往左数，分别要旋转 $0, \theta, 2\theta, \cdots$。



具体地，对输入 $sweep$ 的每一个点都做旋转变换：$\begin{cases} N' = \cos\beta N + \sin\beta B \\ B' = \cos\beta B - \sin\beta N \end{cases}$

但注意到，根据 $\cos\alpha = \dfrac{\boldsymbol{v}_1 \cdot \boldsymbol{v}_2}{|\boldsymbol{v}_1|\,|\boldsymbol{v}_2|}$ 算出的 α 是 $(0, \pi)$ 的，并且给出的公式是顺时针旋转。所以我们要确定是从 $\boldsymbol{v}_1$ 旋转到 $\boldsymbol{v}_2$，还是从 $\boldsymbol{v}_2$ 旋转到 $\boldsymbol{v}_1$。仅就本实验来说，是从始端转到末端。

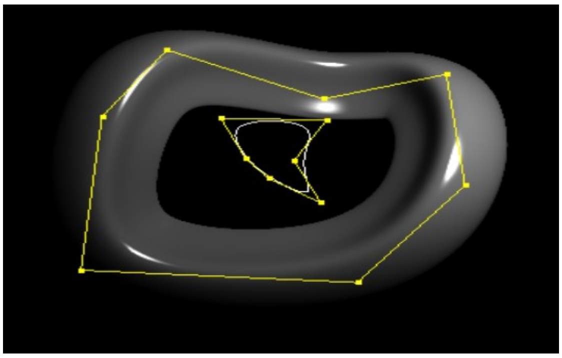
2. 具体代码:

在实际样例中，$sweep[0]$ 和 $sweep[sz3-1]$ 是相同的，我们求的 α 也是从对下标为 $0, sz3-2$ 的求；所以在旋转的角度上稍微有点改变 $sweep[sz3-1]$ 是转最大的角，$sweep[0]$ 转次大... 但总体上来说无伤大雅，且最后效果很好。

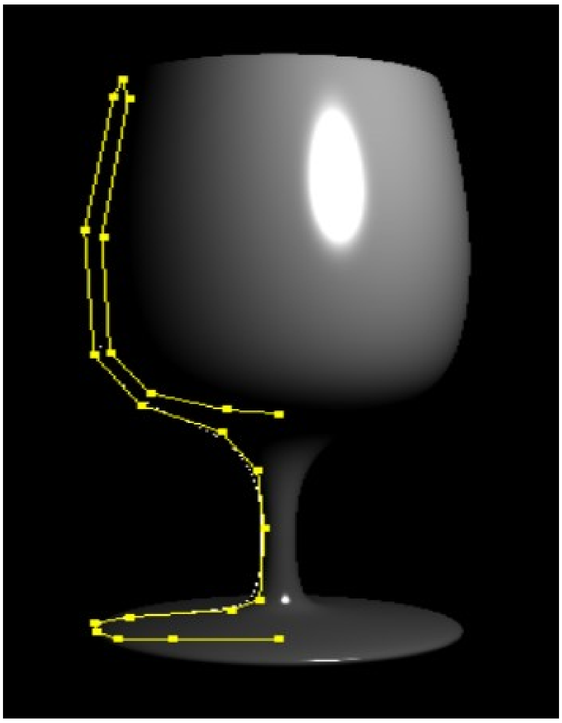
这里主要展示计算角度、判断是否插值、更新 N, B 的代码，后面的是一样的。

```
Vector3f startNormalized = sweep[0].N.normalized();
Vector3f endNormalized = sweep[sz3-2].N.normalized();
double dotProduct = Vector3f::dot(startNormalized,endNormalized);
double alpha = std::acos(dotProduct);
printf("alpha is %lf\n",alpha);
double angle = std::acos(dotProduct) / (sz3-1);
bool flag = true;
if (sweep[0].V != sweep[sz3 - 1].V) || alpha < 0.2)
    flag = false;
printf("flag is %d\n",flag);
for (int i = 0; i < sz3; i++)
{
    int p = i == sz3 - 1 ? sz3 - 1 : sz3 - 2 - i;
    double coss = cos(p*angle), sinn = sin(p*angle);
    Vector3f _N = flag ? coss * sweep[i].N + sinn * sweep[i].B: sweep[i].N;
    Vector3f _B = flag ? coss * sweep[i].B - sinn * sweep[i].N: sweep[i].B;
```

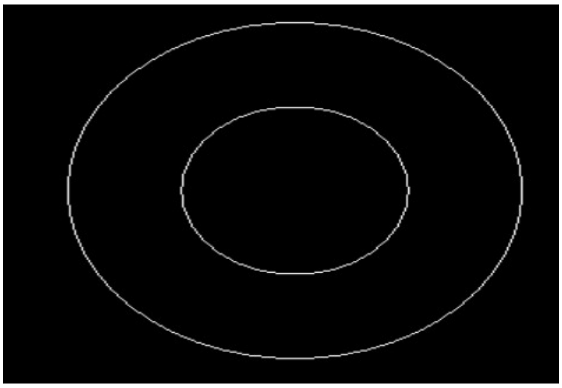
八、样例生成:



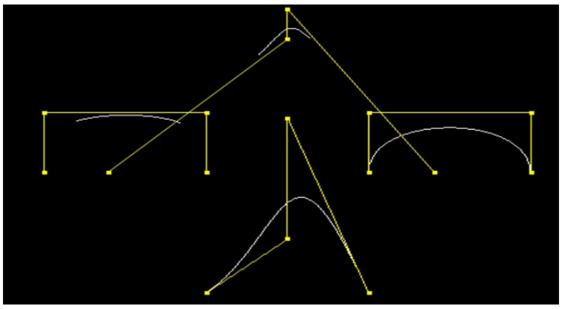
weirder



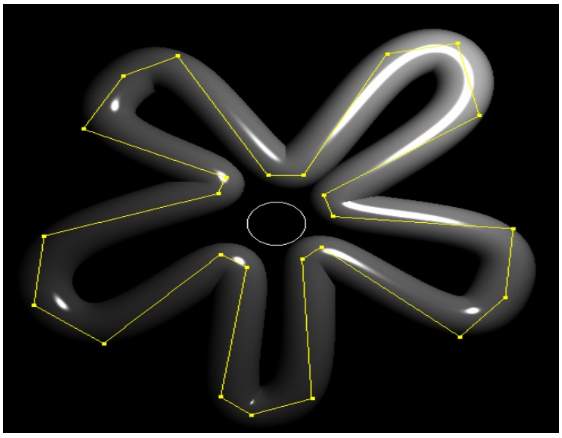
wineglass



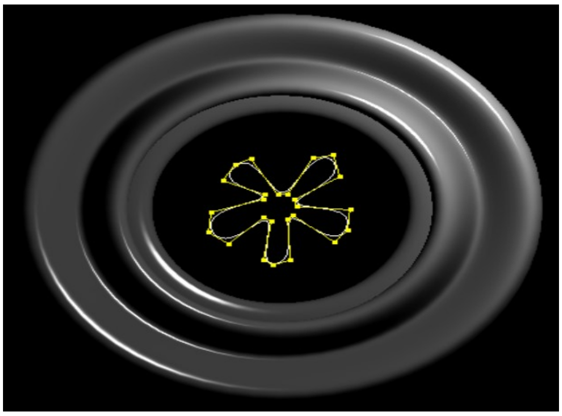
circles



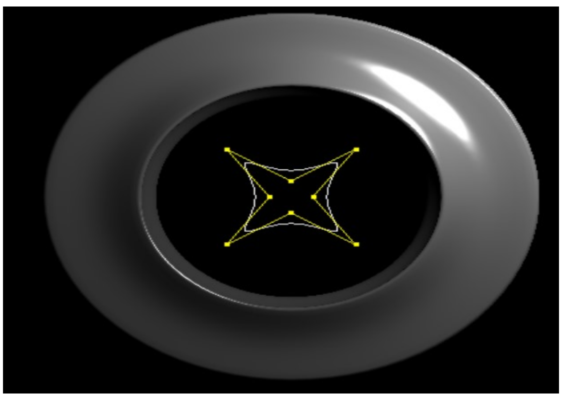
core



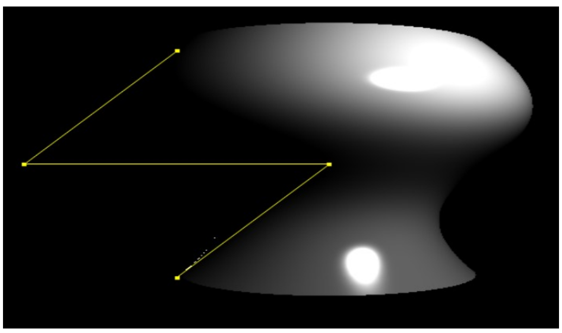
flircle



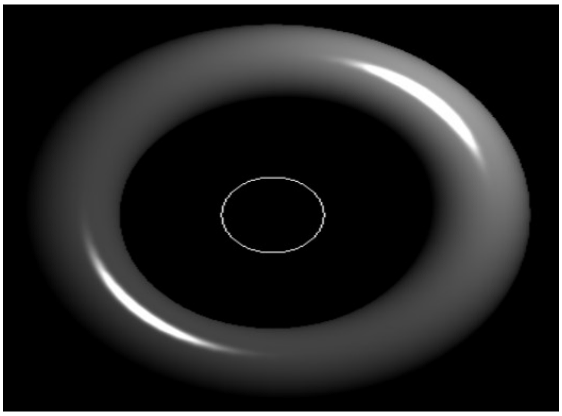
florus



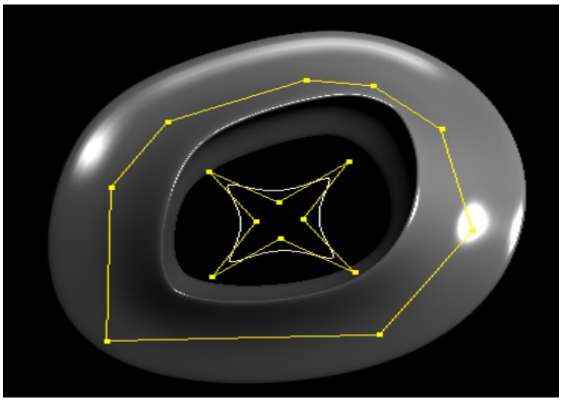
gentorus



norm



tor



weird