# Project 1 实验报告

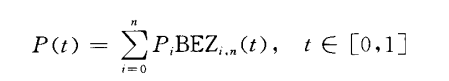
詹佳豪 22307140116

## 背景知识

### Bezier曲线

Bezier曲线可以用于在计算机上显示复杂曲线。具体来说，其将复杂连续曲线的绘制转换为对几个离散的点的控制，这些点称为控制顶点，生成的Bezier曲线会尽可能去拟合控制顶点所形成的控制多边形。如果需要调整曲线形状，只需要调整控制顶点位置即可。我们接下来将依次介绍Bezier曲线的定义以及离散生成算法。

* + 1. Bezier曲线定义：

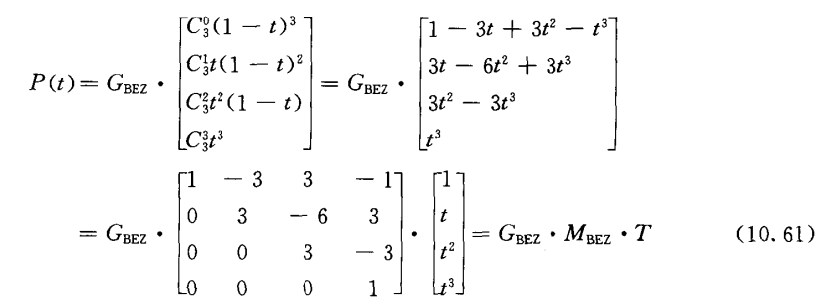


Pi作为控制顶点，BEZ为Beinstein多项式，对于控制顶点以BEZ进行加权即得到了每个t取值时对于的点的位置。

对于Bezier曲线有许多良好的性质：拟局部性，局部性指改变一个控制顶点的位置只会对曲线的局部产生影响，而Bezier曲线并不具备这么强的性质，但它展现出来越远的点影响越小的性质；端点切矢量，起点终点的切线与控制多边形的两边重合；仿射不变性，在经过左边变换时，只需要变换控制点即可......

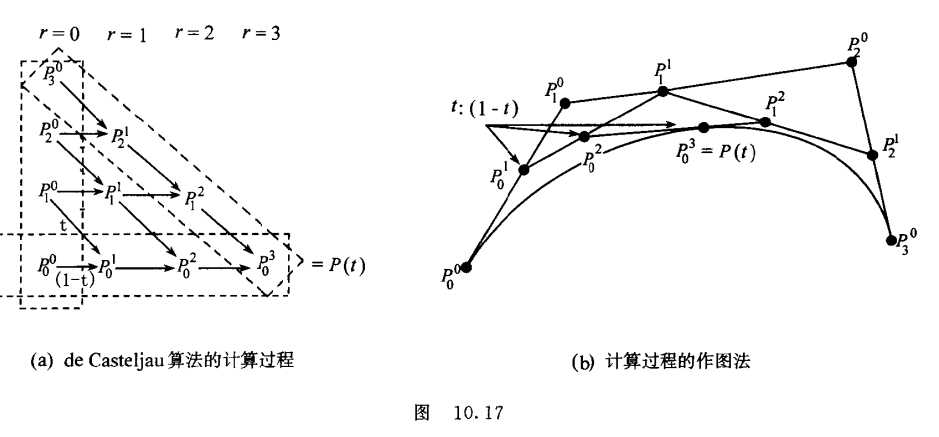
* + 1. 三次Bezier曲线的生成：

对于三次Bezier曲线，也就是四个控制顶点的曲线，我们可以通过矩阵运算来求出Bezier曲线表达式，并且通过取较小的step来模拟绘制曲线。



* + 1. 离散生成

但对于控制点较多的Bezier曲线来说，上述方法计算量过大，不适用，我们采用离散方法来进行生成。此处可以参考de Casteljau算法，其核心在于基于已有的控制顶点，按比例（t：1-t）去拟合出新的控制顶点，这样不断减少控制顶点的数量，最后得到的控制顶点也就刚好在Bezier曲线上。



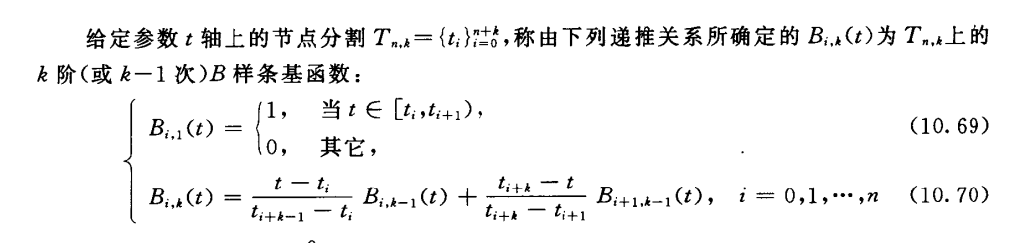
另外，在面对控制顶点较多的Bezier曲线时，我们还可以采用分段生成最后拼接的方法，即拆分为四个一组的组别，之后每个组利用三次Bezier曲线生成的方法去做，最后将所有组别的结果进行拼接，这里需要注意，组别间的起始点、终点要求重合，且拼接过程中要求衔接点光滑。

### B样条曲线

Bezier曲线的缺点在于，如果我要修改曲线的局部形状，在调整控制顶点的过程中，不可避免地会导致整个曲线发生变化，而我们希望曲线有较好的局部性，所以我们引入了B样条曲线。

* + 1. B样条曲线定义

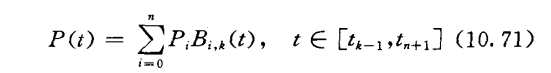
我们首先需要引入B样条基函数的概念：



而B样条曲线则是以B样条基函数为权重对于控制顶点进行加权和。具体分析B样条基函数，可以发现其阶数表征了该基函数被多少的控制点所影响，这也就是B样条曲线后续可以具有局部性的重要原因。

我们可以通过对于Bezier曲线与B样条曲线来更深入地理解它们。可以注意到P（t）的定义中，最后加权和的每个项t的定义是有限制的，而恰好是[tk-1,tn+1]，t为节点向量，而对于Bezier曲线范围是0-1；此外，Bezier曲线的加权为Beinstein函数，该函数是与控制顶点的个数有关，但是B样条曲线的加权与k有关，这是个自己可以控制的量。

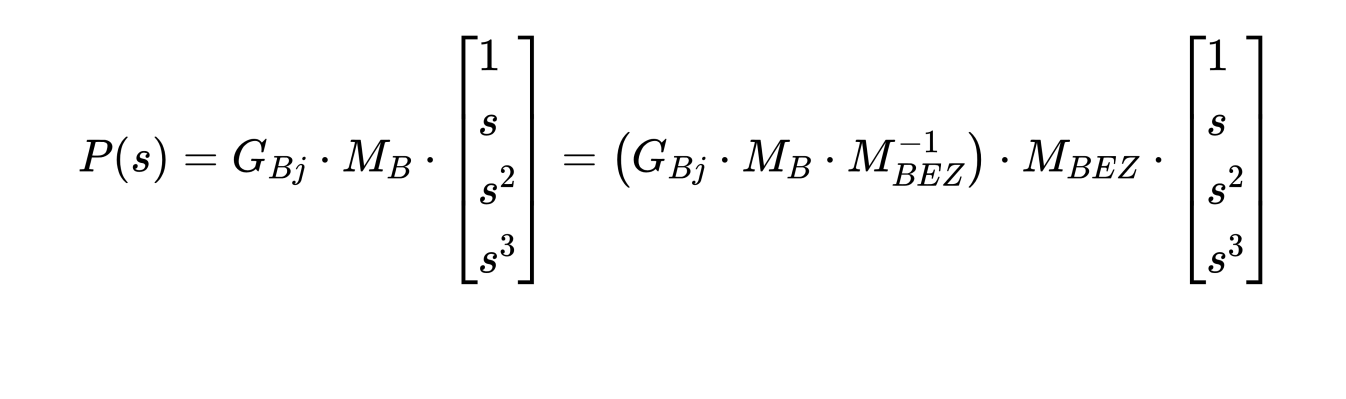
我们具体展开定义域的问题。对于Bi，k涉及到k+1个节点，所以我们的节点向量有t0到tn+k个。而关于它的定义区间，确定阶数后，每个Bi，k都会涉及到k+1个节点，因此在数轴上会有重合，也就是说有的区间会涉及多个基函数。而“区间要合法，区间里必须要有足够的基函数与顶点配对”，由此对于一系列顶点，实际上B样条只在中间的部分有定义，这样就保证了曲线在不同区间是一点点过渡过去的，拼接的效果非常好。



* + 1. B样条曲线的生成

针对B样条曲线的生成主要有两种方法，其一是通过deBoor-Cox算法进行离散生成，其做法类似之前Bezier曲线的离散生成，也就是用已知控制点不断去拟合出唯一的控制点。

而还有已知方法是将B样条曲线转换为Bezier曲线。在任一个[tj,tj+1]上，B样条曲线是k-1次的多项式曲线，它和k-1次的Bezier曲线之间存在一定的联系。通过公式推导，可以发现通过进行变换就可以转换为Bezier曲线：



### 曲面的生成

曲面的生成分为两种，一种是绕y轴旋转曲线形成的曲面，一种是广义圆柱体。

第一种的做法很简单，即将曲线绕y轴旋转，形成绕y轴的一圈点，对于这些点定义局部的法向量（朝外），并且，依次定义三角片，即可绘制出面。

第二种曲面的绘制涉及坐标平移，包括第一种方法中的坐标旋转，我们都可以使用齐次坐标系来进行处理，通过齐次坐标系将profile上一圈的点移动到扫过的轨迹上即可。

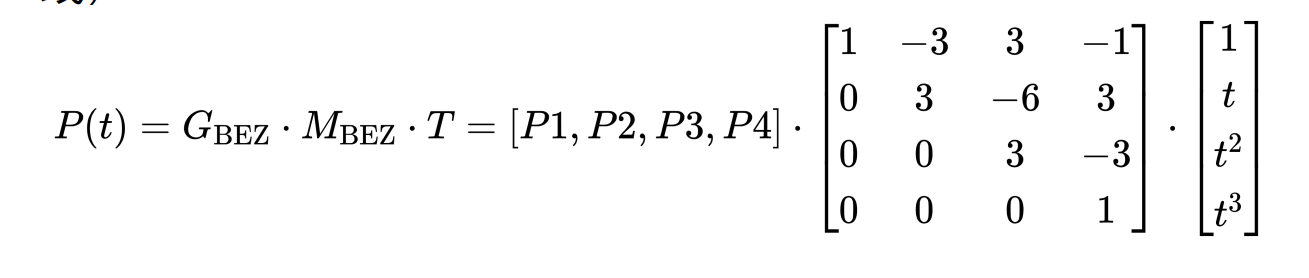
## 代码实现

* 1. Bezier曲线

针对Bezier曲线，我们这里采用的做法是针对输入的一堆点，分组为4个一组（首尾要求重合），分别进行Bezier曲线的绘制，然后进行拼接。所以一开始我们需要判断输入的控制点能否实现分组的要求，需要是3n+1个点，如若不是，报错。

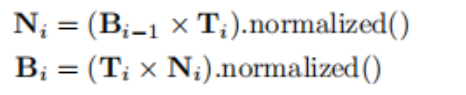
而之后通过pieces计算分组的组数，并且初始化一共计算出的描点个数pieces\*(steps+1)-pieces+1。

针对分组数进行循环，对于每一个组，由于我们需要对于四个点加权计算，而每个点实际上有三个坐标，我们可以采用Px、Py、Pz分别表示四个点的x、y、z坐标，获取这些坐标后，即可计算每一个分组内的描点个数。



对每一分组内的描点个数进行for循环。Step每次递增delta，之后用得到的Px、Py、Pz分别按照权重加和，即得到每个描点对应坐标的值，然后作为V存在Curve（本质为点的列表）中，而对t变量求导则得到T，即曲线局部坐标系的一个坐标轴。

对于NB的计算，我们希望可以规避二阶导为0、局部坐标系变化不连续的问题，所以引入如下公式计算：



至于第一个点的B初始化为（0，0，1.0）即可。但这个问题其实是值得深思的，后面会进行分析。

Curve evalBezier(const vector< Vector3f >& P, unsigned steps){

    // Check

    if (P.size() < 4 || P.size() % 3 != 1)

    {

        cerr << "evalBezier must be called with 3n+1 control points." << endl;

        exit(0);

    }

    int pieces = (P.size()-1) / 3;

    Curve R(pieces\*(steps+1)-pieces+1);

    // construct the matrix

    Matrix4f M(1, -3, 3, -1,

                0, 3, -6, 3,

                0, 0, 3, -3,

                0, 0, 0, 1);

    // std::cout<<"the number of pieces:"<<pieces<<endl;

    // std::cout<<"the number of steps:"<<steps<<endl;

    cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;

    for(int j=0;j<pieces;j+=1){

        //cout<<"the j round "<<j<<" is done"<<endl;

        Vector4f Px(P[0+j].x(), P[1+j].x(), P[2+j].x(), P[3+j].x());

        Vector4f Py(P[0+j].y(), P[1+j].y(), P[2+j].y(), P[3+j].y());

        Vector4f Pz(P[0+j].z(), P[1+j].z(), P[2+j].z(), P[3+j].z());

        float delta = 1.0f / steps;

        float t = 0;

        for(unsigned int i = 0; i < steps+1; i++){

            //cout<<"the i round "<<i<<" is done"<<endl;

            // cout<<"i+j\*(steps) "<<i+j\*(steps)<<endl;

            Vector4f pos(1, t, t\*t, t\*t\*t);

            Vector4f weight = M \* pos;

            R[i+j\*(steps)].V = Vector3f(Vector4f::dot(Px,weight), Vector4f::dot(Py,weight), Vector4f::dot(Pz,weight));

            //cout<<"the "<<i<<" V is "<<R[i+j\*(steps)].V.x()<<" "<<R[i+j\*(steps)].V.y()<<" "<<R[i+j\*(steps)].V.z()<<endl;

            Vector4f tangent(0, 1, 2\*t, 3\*t\*t);

            Vector4f tangent\_weight = M \* tangent;

            R[i+j\*(steps)].T = Vector3f(Vector4f::dot(Px,tangent\_weight), Vector4f::dot(Py,tangent\_weight),Vector4f::dot(Pz,tangent\_weight)).normalized();

            //cout<<"the T is "<<R[i+j\*(steps)].T.x()<<" "<<R[i+j\*(steps)].T.y()<<" "<<R[i+j\*(steps)].T.z()<<endl;

            if(i == 0){

            // If it is the first step, we should initialize the N

                Vector3f tmp(0,0,1.0);

                R[i+j\*(steps)].N = Vector3f::cross(tmp, R[i+j\*(steps)].T).normalized();

                R[i+j\*(steps)].B = Vector3f::cross(R[i+j\*(steps)].T,R[i+j\*(steps)].N).normalized();

                //cout<<"here?"<<endl;

            }

            else{

            // iteratively

                R[i+j\*(steps)].N = Vector3f::cross(R[j\*(steps)+i-1].B,R[i+j\*(steps)].T).normalized();

                R[i+j\*(steps)].B = Vector3f::cross(R[j\*(steps)+i].T,R[i+j\*(steps)].N).normalized();

                //cout<<"the N is "<<R[i+j\*(steps)].N<<endl;

            }

            //cout<<"the "<<i<<" N is "<<R[i+j\*(steps)].N.x()<<" "<<R[i+j\*(steps)].N.y()<<" "<<R[i+j\*(steps)].N.z()<<endl;

            //cout<<"the B is "<<R[i+j\*(steps)].B<<endl;

            t += delta;

        }

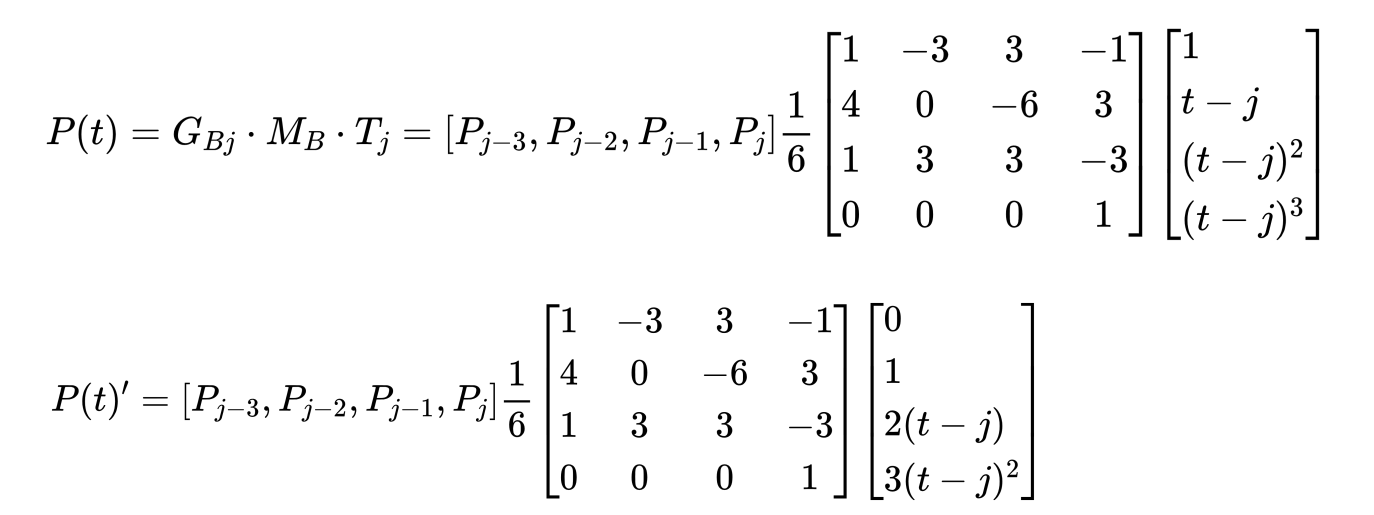
    }

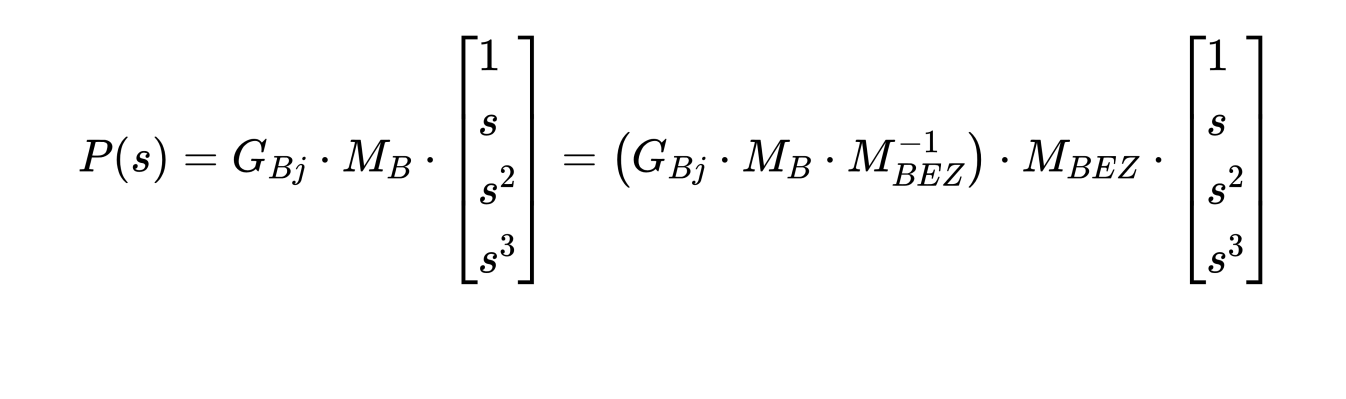
    return R;

}

* 1. B样条曲线

B样条曲线的生成依赖于通过坐标变化，接着就能套用已经实现的Bezier曲线。数学基础如下：





因此我们首先计算出坐标变换矩阵，然后作用在Gb上，得到新点。但此时点的数量值得深思，每一次坐标变换将j-3、j-2、j-1、j这四个点变换为新的四个点，但j是步长为1来增加的，因此其中是有点被重复计算的，基于这个假设，我们最后实际上会得到更多的点，几乎是原来的四倍。

而此时如果直接调用Bezier曲线的生成会导致错误，因为这些点之间的相互依赖关系并不满足控制顶点的要求。所以每生成的四个点我都会调用一次Bezier曲线，得到Curve点集。

但如果就这样来实现项目，会导致曲线法线方向不连贯的问题。具体来说，每四个点都输入Bezier曲线的生成，但我们可以回忆下我们前面初始化第一个次法向的方向为（0，0，1），这正常来说没什么问题，但每个一小段都以（0，0，1）初始化，会导致之后生成的面会对不齐，一拐一拐的。所以我这里抛弃了直接调用Besier的方法，而是重新写了个功能于Bezier曲线几乎一样的函数，这个函数唯一的不同就是会多接受一个初始化的次法向方向，这样就保证了段与段之间尽可能的连续。

Curve evalBspline(const vector< Vector3f >& P, unsigned steps)

{

    //cout<<"the number of P:"<<P.size()<<endl;

    // Check

    if (P.size() < 4)

    {

        cerr << "evalBspline must be called with 4 or more control points." << endl;

        exit(0);

    }

    // TODO:

    // It is suggested that you implement this function by changing

    // basis from B-spline to Bezier.  That way, you can just call

    // your evalBezier function.

    Matrix4f M(1, -3, 3, -1,

                4, 0, -6, 3,

                1, 3, 3, -3,

                0, 0, 0, 1);

    M/=6;

    Matrix4f Mbezier(1, -3, 3, -1,

                    0, 3, -6, 3,

                    0, 0, 3, -3,

                    0, 0, 0, 1);

    Matrix4f Mbezier\_inv = Mbezier.inverse();

    Matrix4f GM = M \* Mbezier\_inv;

    Curve R;

    Vector3f B(0,0,1);

    for(unsigned int i=3;i<P.size();i++){

        vector<Vector3f> Pnew;

        Vector4f Px(P[i-3].x(), P[i-2].x(), P[i-1].x(), P[i].x());

        Vector4f Py(P[i-3].y(), P[i-2].y(), P[i-1].y(), P[i].y());

        Vector4f Pz(P[i-3].z(), P[i-2].z(), P[i-1].z(), P[i].z());

        Pnew.push\_back(Vector3f(Vector4f::dot(Px,GM.getCol(0)), Vector4f::dot(Py,GM.getCol(0)), Vector4f::dot(Pz,GM.getCol(0))));

        Pnew.push\_back(Vector3f(Vector4f::dot(Px,GM.getCol(1)), Vector4f::dot(Py,GM.getCol(1)), Vector4f::dot(Pz,GM.getCol(1))));

        Pnew.push\_back(Vector3f(Vector4f::dot(Px,GM.getCol(2)), Vector4f::dot(Py,GM.getCol(2)), Vector4f::dot(Pz,GM.getCol(2))));

        Pnew.push\_back(Vector3f(Vector4f::dot(Px,GM.getCol(3)), Vector4f::dot(Py,GM.getCol(3)), Vector4f::dot(Pz,GM.getCol(3))));

        // cout<<"R[R.size()-1].B is "<<R[R.size()-1].B.x()<<" "<<R[R.size()-1].B.y()<<" "<<R[R.size()-1].B.z()<<endl;

        Curve R\_=process\_for\_B(Pnew,steps,B);

        for(unsigned int j=0;j<R\_.size();j++){

            R.push\_back(R\_[j]);

        }

        B = R[R.size()-1].B;

        //cout<<"The round "<< i<<" is done!!!!!"<<endl;

    }

    cout<<"R's size is :"<<R.size()<<endl;

    if(approx(R[R.size()-1].T,R[0].T)&&!approx(R[R.size()-1].N,R[0].N)){

        cout<<"we should smooth the curve"<<endl;

        cout<<"the last T is "<<R[R.size()-1].T.x()<<" "<<R[R.size()-1].T.y()<<" "<<R[R.size()-1].T.z()<<endl;

        cout<<"the first T is "<<R[0].T.x()<<" "<<R[0].T.y()<<" "<<R[0].T.z()<<endl;

        cout<<"the last N is "<<R[R.size()-1].N.x()<<" "<<R[R.size()-1].N.y()<<" "<<R[R.size()-1].N.z()<<endl;

        cout<<"the first N is "<<R[0].N.x()<<" "<<R[0].N.y()<<" "<<R[0].N.z()<<endl;

        //we need to smooth the curve

        float dotProduct = Vector3f::dot(R[R.size()-1].N,R[0].N);

        float v1Length = R[R.size()-1].N.abs();

        float v2Length = R[0].N.abs();

        float alpha = acos(dotProduct / (v1Length \* v2Length));

        double angle = alpha/R.size();

        cout<<"the alpha is "<<alpha<<endl;

        for(unsigned int i=0;i<R.size();i++){

            R[i].N = (cos(-angle\*i)\*R[i].N + sin(-angle\*i)\*R[i].B).normalized();

            R[i].B = Vector3f::cross(R[i].T,R[i].N).normalized();

        }

    }

    cout<<"after smooth:"<<endl;

    cout<<"the last N is "<<R[R.size()-1].N.x()<<" "<<R[R.size()-1].N.y()<<" "<<R[R.size()-1].N.z()<<endl;

    cout<<"the first N is "<<R[0].N.x()<<" "<<R[0].N.y()<<" "<<R[0].N.z()<<endl;

    return R;

}

* 1. 旋转曲面

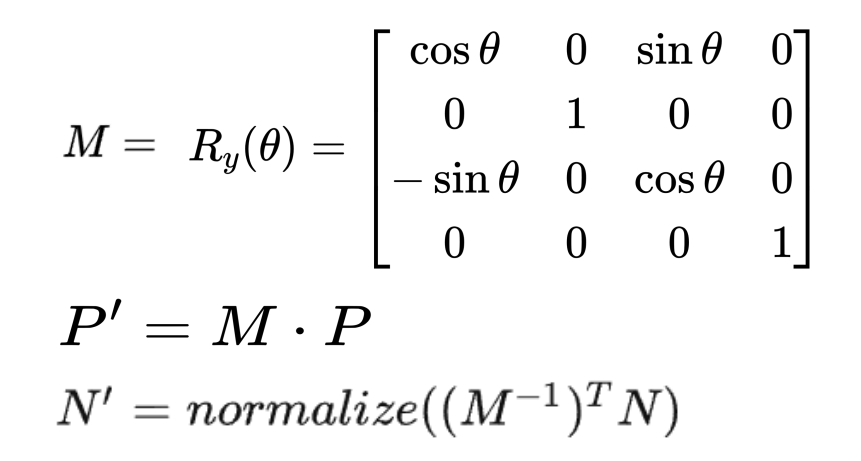
旋转曲面的实现较为容易，基于已经生成的曲线，围绕y轴，只要每次角度上旋转一共小step，这样就可以形成围绕在y轴周围的一圈点，基于这些点，定义好他们的局部法向方向，以及定义好逆时针的面就可以实现曲面的绘制。

具体来说，曲线上每个点的旋转可以利用旋转矩阵来实现，注意这个旋转矩阵是在齐次坐标系下来实现的。齐次坐标系是一个非常有趣的东西，我们在线性代数中知道矩阵的坐标变换可以使用Ax+b的方式实现，但这样在计算机上不好操作，因为有一个加法运算，我们希望都是乘法运算就好了。基于这个目的，我们引入了齐次坐标系，齐次坐标系在三维空间呈现出4\*4的状态，其中左上角是一个旋转矩阵，而最右边的前三个构成了平移坐标，最后一共位置取1，可以经过验证发现这个变换施加在原本的坐标（最后一位补1为新的点坐标，补0为向量坐标）可以完成仿射变换。另外值得一提的是，绕y轴旋转时-sin的位置换了一下，实际上这是按照右手坐标系布置的结果，因此是自然的。

我们进入旋转的循环中时，只需要构造出对应的变换矩阵M，左乘在原本的坐标上即可。我这里的实现时不断累加step，然后乘在最开始的点上，也可以保持step，乘在前一个点上。

剩下的工作就是得到局部法向方向，对于N，只需要直接将M的逆转置左乘原本的N即可。但有一点需要注意，原本的法向是指向y轴的，我们需要让局部法向向量指向曲面外部，所以取反即可，如果没有取反会发现整个曲面是黑的，没有任何反光，我想是由于opengl的默认设置原因。

而对于三角面的构造，只需要注意保持逆时针的顺序即可。另外，需要计算好点的序号，避免出现多算面或者漏算面的情况。



Surface makeSurfRev(const Curve &profile, unsigned steps)

{

    Surface surface;

    //surface = quad();

    //

    if (!checkFlat(profile))

    {

        cerr << "surfRev profile curve must be flat on xy plane." << endl;

        exit(0);

    }

    float delta = 2.0f \* c\_pi / steps;

    int num = profile.size()\*(steps);

    int total = num;

    for(unsigned int i=0;i<=steps ;i++){

        //cout<<"the i round "<<i<<" is done"<<endl;

        float theta = i \* delta;

        Matrix4f M(cos(theta), 0, sin(theta), 0,

                    0, 1, 0, 0,

                    -sin(theta), 0, cos(theta), 0,

                    0, 0, 0, 1);

        Matrix4f M\_inverse\_transpose = M.inverse();

        M\_inverse\_transpose.transpose();

        for(int j=0;j<(int)profile.size();j++){

            Vector3f pos = profile[j].V;

            Vector3f normal = profile[j].N;

            Vector4f pos4(pos[0], pos[1], pos[2], 1);

            Vector4f normal4(normal[0], normal[1], normal[2], 0);

            Vector4f pos\_new = M \* pos4;

            Vector4f normal\_new = (M\_inverse\_transpose \* normal4);

            surface.VV.push\_back(Vector3f(pos\_new[0], pos\_new[1], pos\_new[2]));

            surface.VN.push\_back(Vector3f(-normal\_new[0], -normal\_new[1], -normal\_new[2]).normalized());

        }

        if(i==steps){

            break;

        }

        for(int j=0;j<(int)profile.size()-1;j++){

            Tup3u face1;

            Tup3u face2;

            face1[0] = (i \* profile.size() + j)%total;

            face1[1] = (i \* profile.size() + j + 1)%total;

            face1[2] = ((i + 1) \* profile.size() + j )%total;

            face2[0] = (i \* profile.size() + j + 1)%total;

            face2[1] = ((i + 1) \* profile.size() + j + 1)%total;

            face2[2] = ((i + 1) \* profile.size() + j)%total;

            surface.VF.push\_back(face1);

            surface.VF.push\_back(face2);

            // cout<<"the face1 is "<<face1[0]<<" "<<face1[1]<<" "<<face1[2]<<endl;

            //cout<<"the face2 is "<<face2[0]<<" "<<face2[1]<<" "<<face2[2]<<endl;

        }

        //cout<<"the size of VV is "<<surface.VV.size()<<endl;

    }

    // TODO: Here you should build the surface.  See surf.h for details.

    //cerr << "\t>>> makeSurfRev called (but not implemented).\n\t>>> Returning empty surface." << endl;

    return surface;

}

* 1. 广义圆柱体

此时的曲面不再是通过绕y轴旋转得到的，而是任意轴，此时也不难，只需要进行同上的坐标变换即可。

首先遍历sweep曲线上的所有点，对于每一个点，建立变换矩阵，使得坐标系变成以sweep上该点的T方向为y轴。

准备好变换矩阵后，遍历profile曲线上的所有点，依次变换到sweep周围，这样就形成了围绕在sweep周围的一圈点，继续循环，就会出现广义圆柱体的一堆点。

接下来可以模仿上题进行局部法向向量的定义以及三角面的定义。

Surface makeGenCyl(const Curve &profile, const Curve &sweep )

{

    Surface surface;

    //surface = quad();

    if (!checkFlat(profile))

    {

        cerr << "genCyl profile curve must be flat on xy plane." << endl;

        exit(0);

    }

    // TODO: Here you should build the surface.  See surf.h for details.

    int num=sweep.size();

    int num\_profile = profile.size();

    int total = num \* num\_profile;

    for(int i = 0;i<num;i++){

        Matrix4f M(sweep[i].N[0], sweep[i].B[0], sweep[i].T[0], sweep[i].V[0],

                sweep[i].N[1], sweep[i].B[1], sweep[i].T[1], sweep[i].V[1],

                sweep[i].N[2], sweep[i].B[2], sweep[i].T[2], sweep[i].V[2],

                0, 0, 0, 1);

        Matrix4f M\_inverse\_transpose = M.inverse();

        M\_inverse\_transpose.transpose();

        for(int j = 0;j<num\_profile;j++){

            Vector3f pos = profile[j].V;

            Vector3f normal = profile[j].N;

            Vector4f pos4(pos[0], pos[1], pos[2], 1);

            Vector4f normal4(normal[0], normal[1], normal[2], 0);

            Vector4f pos\_new = M \* pos4;

            Vector4f normal\_new = (M\_inverse\_transpose \* normal4);

            surface.VV.push\_back(Vector3f(pos\_new[0], pos\_new[1], pos\_new[2]));

            surface.VN.push\_back(Vector3f(-normal\_new[0], -normal\_new[1], -normal\_new[2]).normalized());

        }

        for(int j=0;j<num\_profile;j++){

            Tup3u face1;

            Tup3u face2;

            face1[0] = (i \* profile.size() + j)%total;

            face1[1] = (i \* profile.size() + j + 1)%total;

            face1[2] = ((i + 1) \* profile.size() + j )%total;

            face2[0] = (i \* profile.size() + j + 1)%total;

            face2[1] = ((i + 1) \* profile.size() + j + 1)%total;

            face2[2] = ((i + 1) \* profile.size() + j)%total;

            surface.VF.push\_back(face1);

            surface.VF.push\_back(face2);

            // cout<<"the face1 is "<<face1[0]<<" "<<face1[1]<<" "<<face1[2]<<endl;

            // cout<<"the face2 is "<<face2[0]<<" "<<face2[1]<<" "<<face2[2]<<endl;

        }

    }

    //cerr << "\t>>> makeGenCyl called (but not implemented).\n\t>>> Returning empty surface." <<endl;

    return surface;

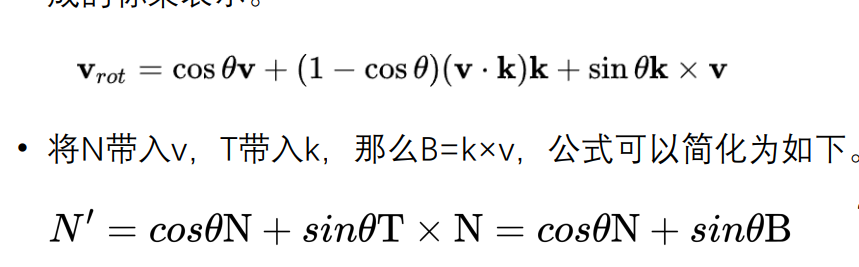
}

* 1. 拓展：插值

对于那些首尾相连、无比崎岖的广义圆柱体，我们会发现首尾交界点上对不齐，这是由于我尾部经过一系列坐标系的转动后没法和初始位置对齐。

为了解决这个问题，我们首先检测其实位置与最后位置的法向量是否相差较大，这里由于浮点运算误差很大，所以采用approx的方法。

检测出问题后，进一步计算二者法向量的差异角度，这里可以使用向量内积除以模长乘积取arccos的方法，得到角度变换，而后将这个角度差均摊到整个一圈的所有点上，每个点的法向量都基于下述公式进行一个微调，最后就能保证首尾连接正常。



cout<<"R's size is :"<<R.size()<<endl;

    if(approx(R[R.size()-1].T,R[0].T)&&!approx(R[R.size()-1].N,R[0].N)){

        cout<<"we should smooth the curve"<<endl;

        cout<<"the last T is "<<R[R.size()-1].T.x()<<" "<<R[R.size()-1].T.y()<<" "<<R[R.size()-1].T.z()<<endl;

        cout<<"the first T is "<<R[0].T.x()<<" "<<R[0].T.y()<<" "<<R[0].T.z()<<endl;

        cout<<"the last N is "<<R[R.size()-1].N.x()<<" "<<R[R.size()-1].N.y()<<" "<<R[R.size()-1].N.z()<<endl;

        cout<<"the first N is "<<R[0].N.x()<<" "<<R[0].N.y()<<" "<<R[0].N.z()<<endl;

        //we need to smooth the curve

        float dotProduct = Vector3f::dot(R[R.size()-1].N,R[0].N);

        float v1Length = R[R.size()-1].N.abs();

        float v2Length = R[0].N.abs();

        float alpha = acos(dotProduct / (v1Length \* v2Length));

        double angle = alpha/R.size();

        cout<<"the alpha is "<<alpha<<endl;

        for(unsigned int i=0;i<R.size();i++){

            R[i].N = (cos(-angle\*i)\*R[i].N + sin(-angle\*i)\*R[i].B).normalized();

            R[i].B = Vector3f::cross(R[i].T,R[i].N).normalized();

        }

    }

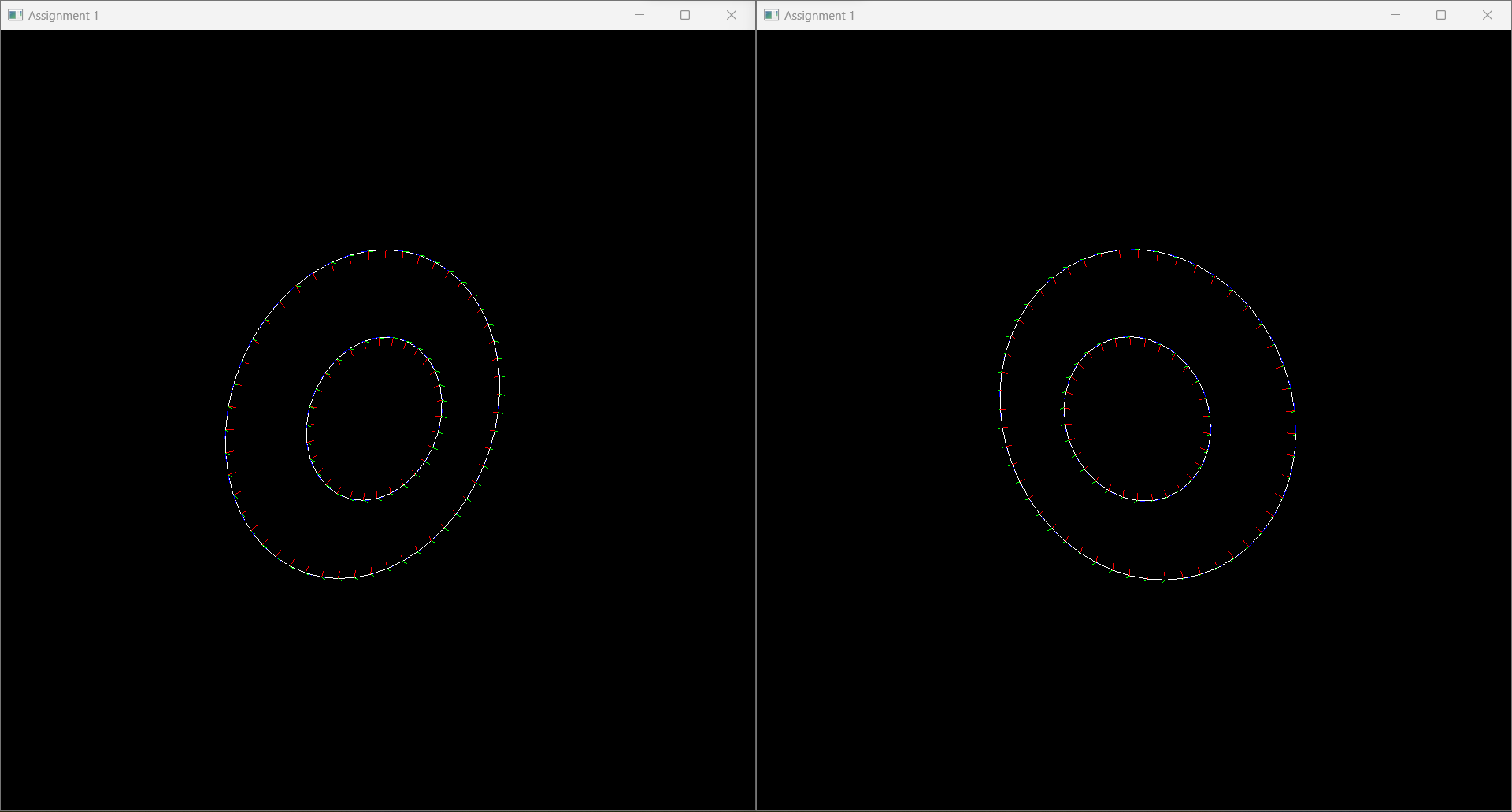
    cout<<"after smooth:"<<endl;

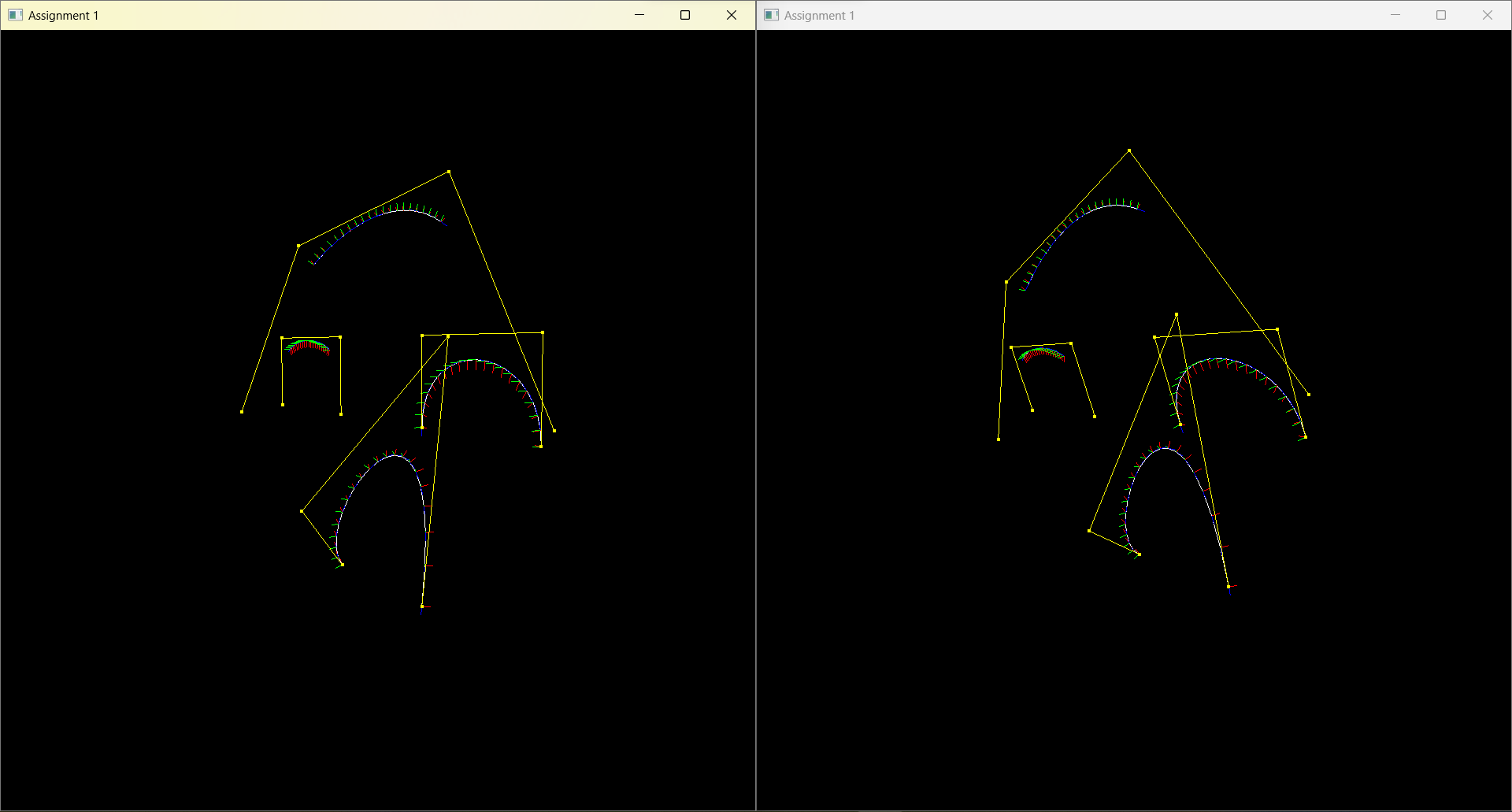
    cout<<"the last N is "<<R[R.size()-1].N.x()<<" "<<R[R.size()-1].N.y()<<" "<<R[R.size()-1].N.z()<<endl;

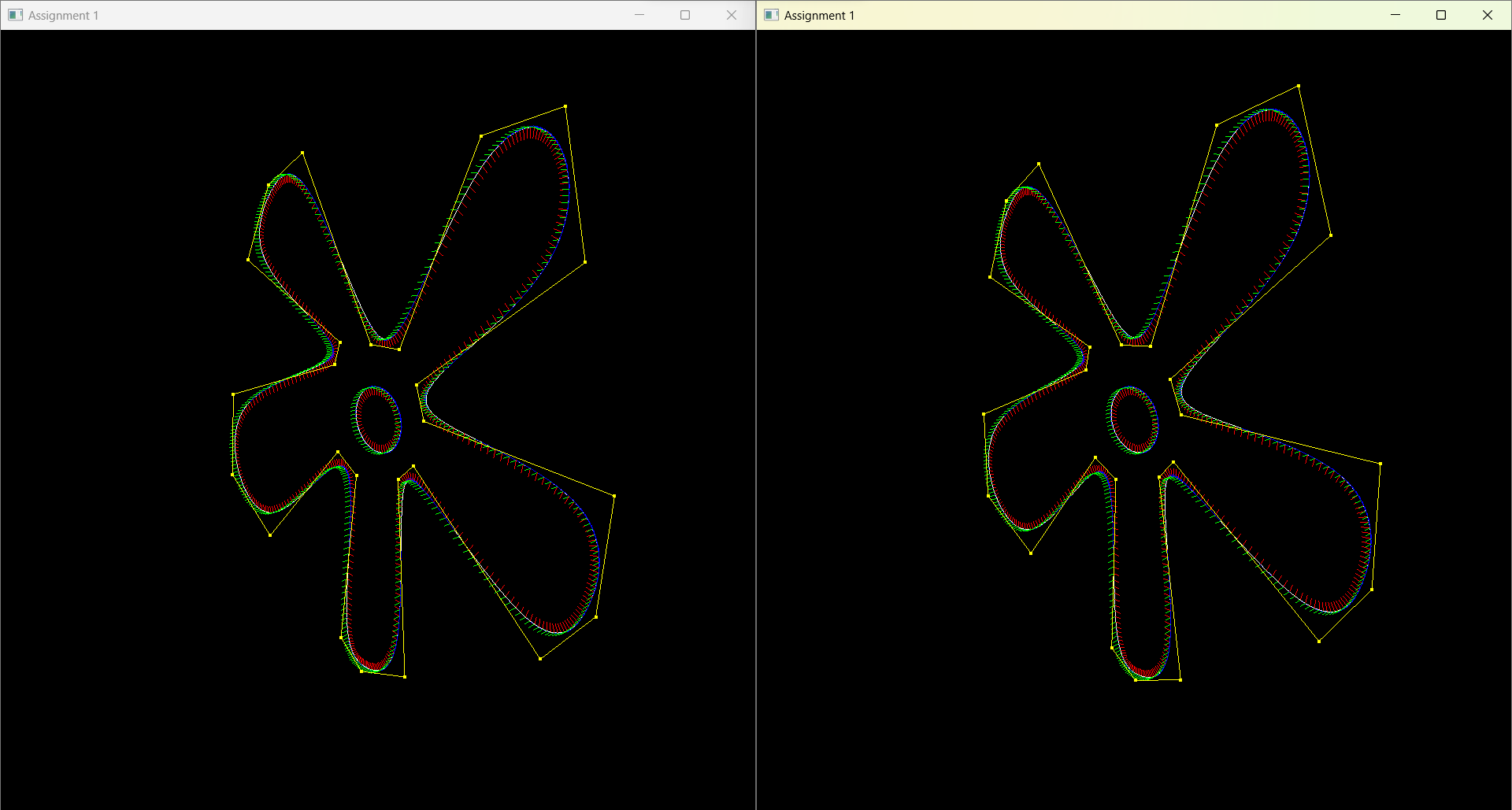
    cout<<"the first N is "<<R[0].N.x()<<" "<<R[0].N.y()<<" "<<R[0].N.z()<<endl;

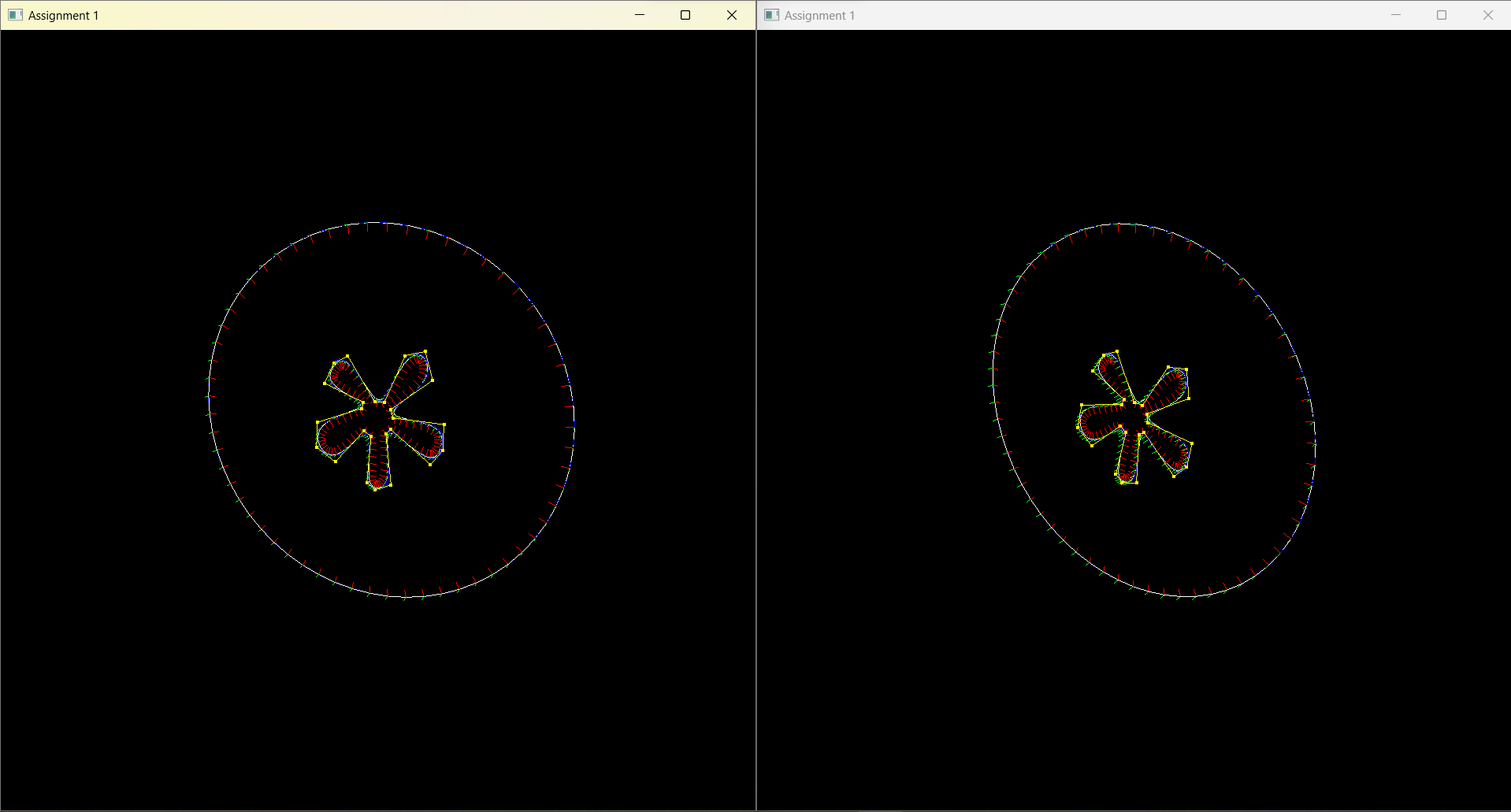
## 实验结果（右图为我的实现）

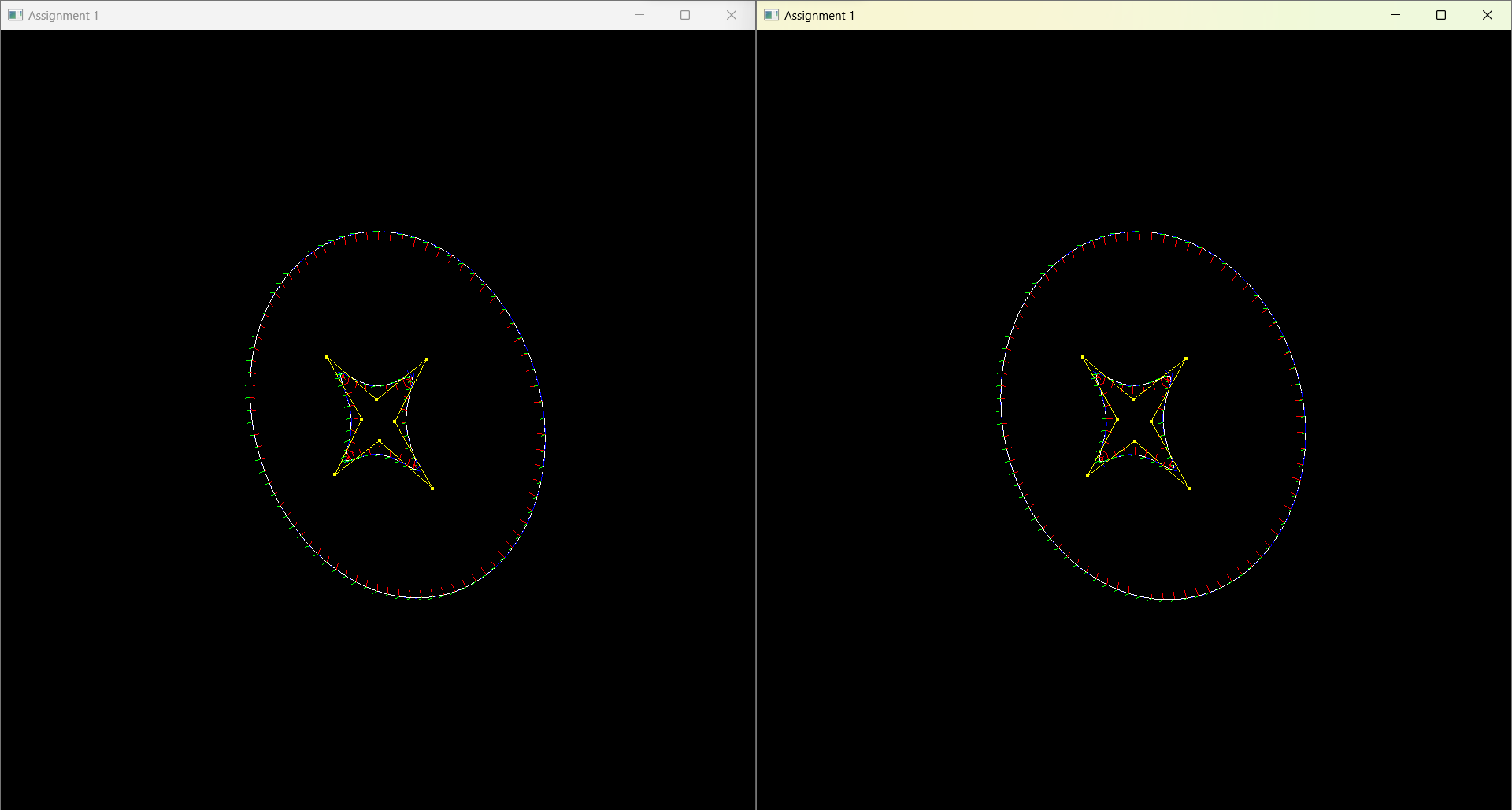
* 1. 实验一：曲线生成

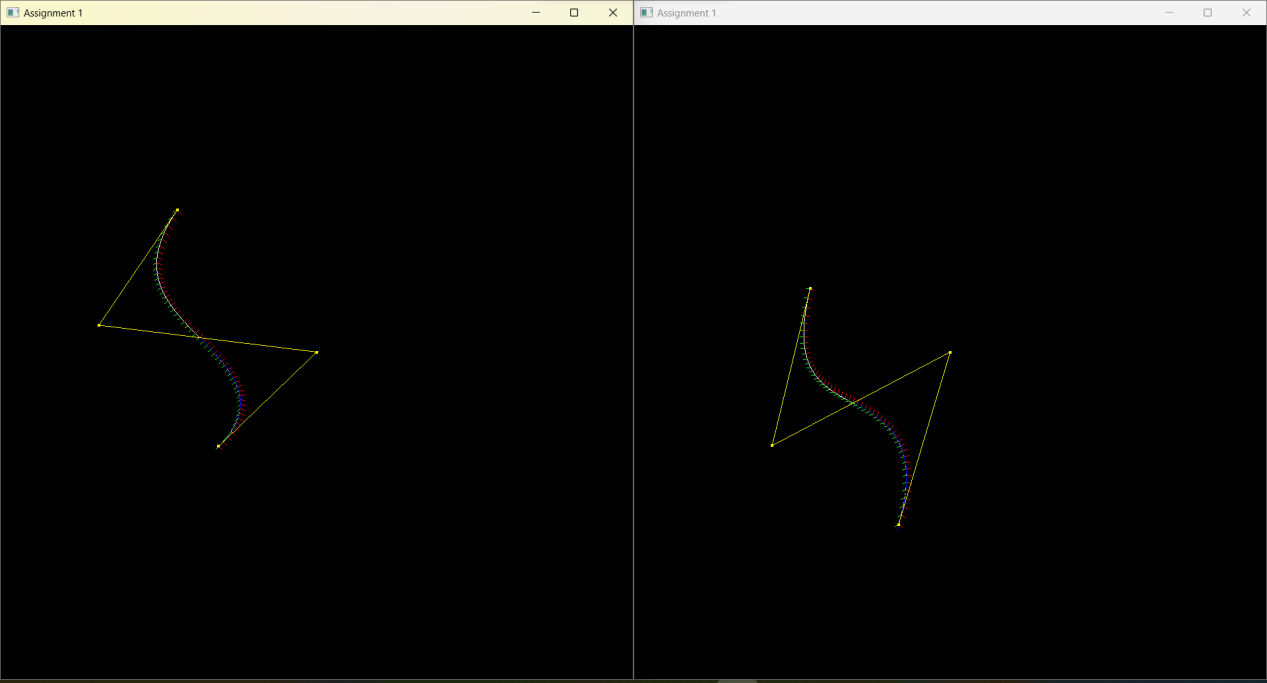


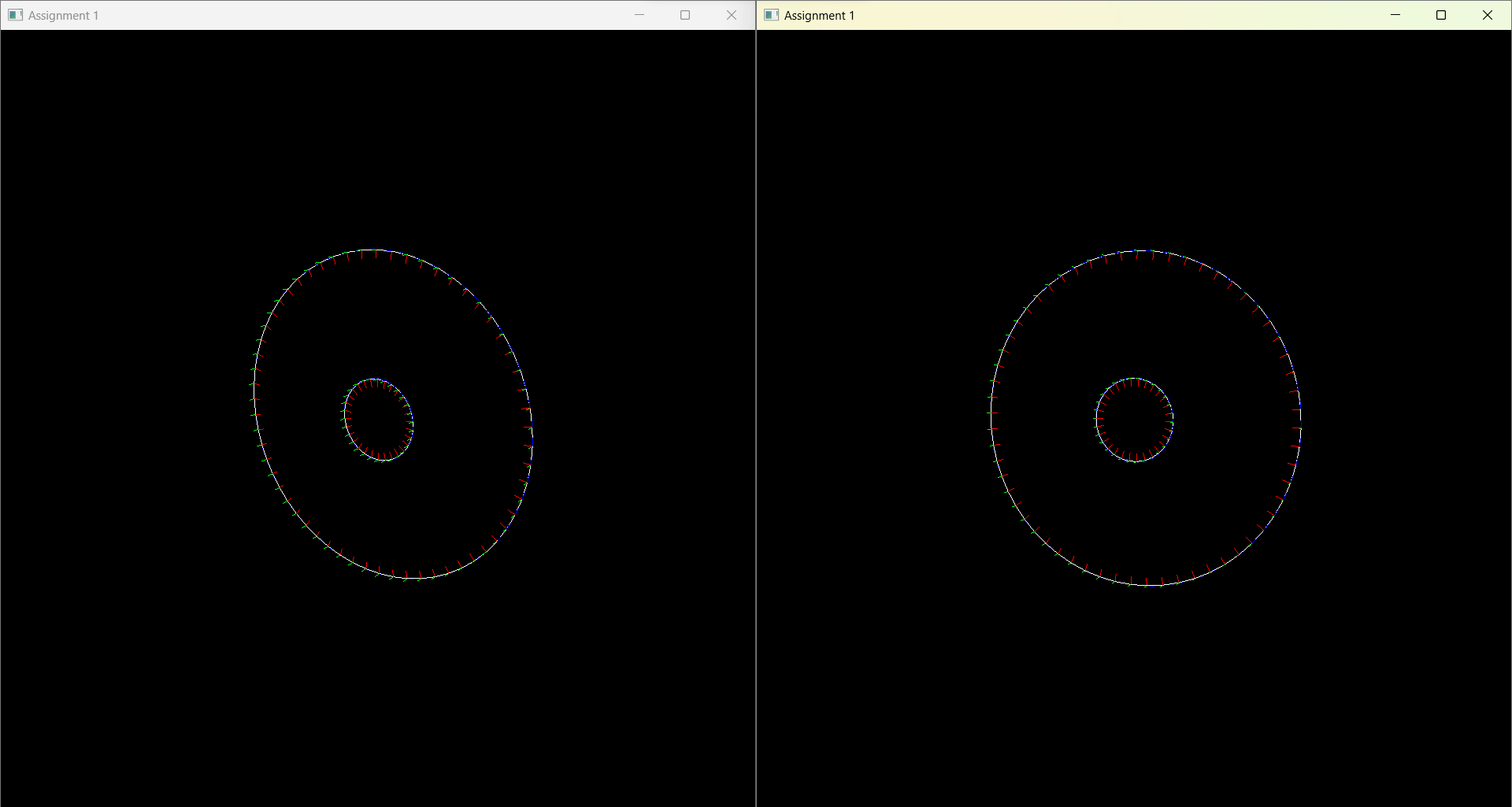


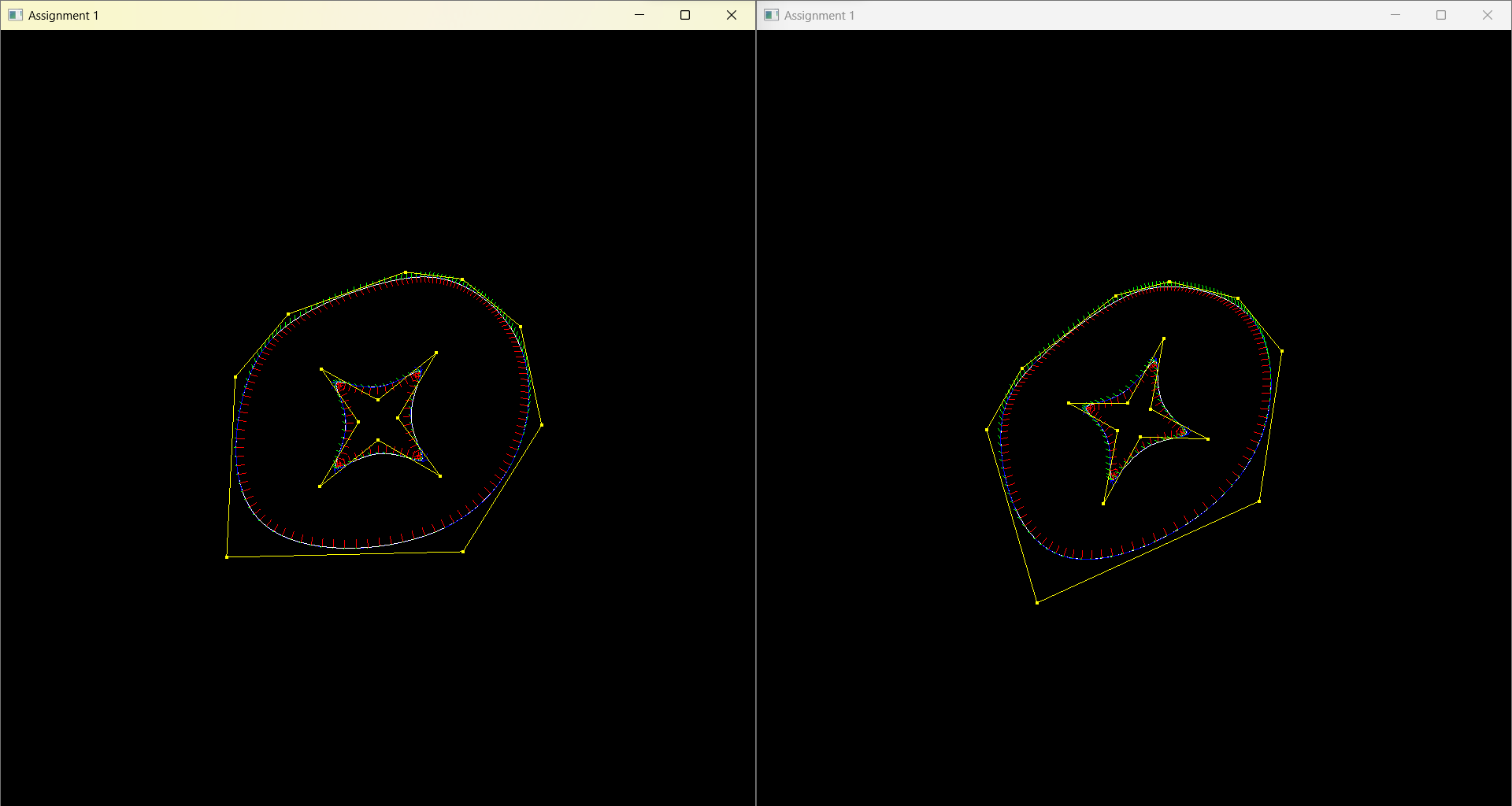


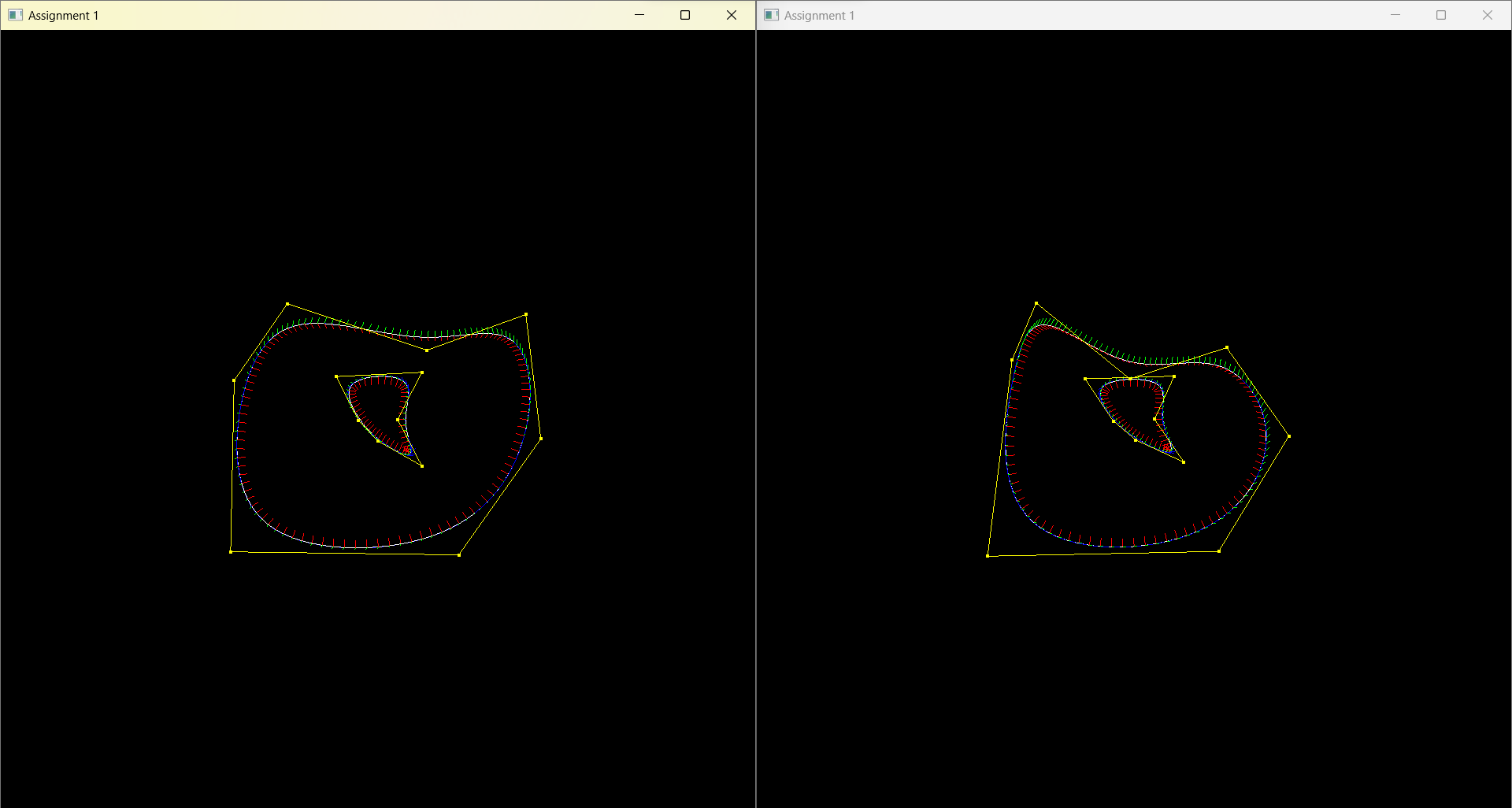


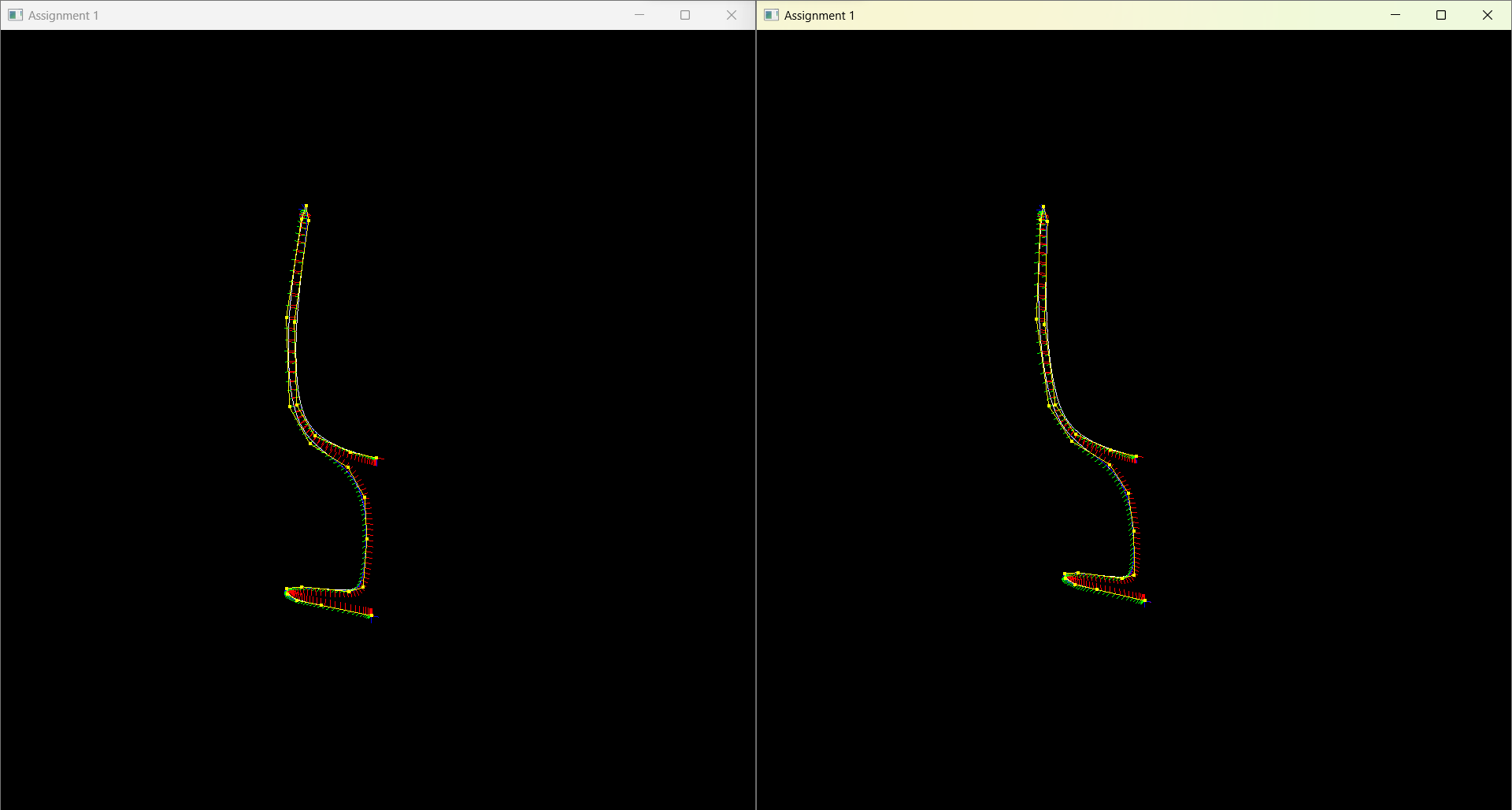












* 1. 实验二：曲面生成

