Project 1

实验人：223071130424 曹云聪
实验时间：2024.4.17

1 曲线的绘制

1.1 要求：

- 程序必须能够生成和显示分段三次B´ezier和B样条曲线。此外，它必须正确计算局部坐标系（包括法向量N，切向量T，和次法线向量B）并显示它们。应该将曲线渲染为白色，并将N、B和T向量分别渲染为红色、绿色和蓝色。
- 需要在curve.cpp中填写evalB´ezier和evalBspline函数，显示将由初始代码处理。

1.2 理论知识

- 第一个任务是实现分段三次B‘ezier和B样条曲线。给定一个控制点数组，应该生成一组位于样条曲线上的点（然后可以通过将它们与线段连接来绘制样条曲线）。
- 如果唯一的目标是绘制样条曲线，那么我们根据公式计算点的坐标就足够了。但是对于动画和表面建模等其他应用程序，需要生成更多的信息（比如切向量和法向量）。

1.2.1 二维曲线

（1）为了确定在某些 t 处的适当变换，我们引入了一些简记符号。首先，我们将位置 V 定义为 V = q(t)。我们将单位切向量 T 定义为T = q'(t) / ||q'(t)||。
（2）对于平面运动，我们可以引入一个新的与 T 正交的向量，我们称之为 B ， 这被称为次法线，我们将随意选择它指向正的 z 方向。给定 B ， 我们可以计算一个备选法线为：N = B × T。

1.2.2 三维曲线

我们将沿着 q (t) 沿着曲线生成离散点。换句话说，我们可以沿着第一步 t1 = 0， 第二步 t2 = 0.1等等的 q (t) 前进。在第 i 步中，我们可以计算以下值（就像二维情况一样）：

$$\mathbf{V}_i = \mathbf{q}(t_i)$$
$$\mathbf{T}_i = \mathbf{q}'(t_i).\mathrm{normalized}()$$

为了选择 Ni 和 Bi ， 我们使用以下递归更新方程：

$$\mathbf{N}_i = (\mathbf{B}_{i-1} \times \mathbf{T}_i).\text{normalized}()$$
$$\mathbf{B}_i = (\mathbf{T}_i \times \mathbf{N}_i).\text{normalized}()$$

（1）三次贝塞尔曲线

$$P(t) = G_{\text{BEZ}} \cdot M_{\text{BEZ}} \cdot T = [P1, P2, P3, P4] \cdot \begin{bmatrix} 1 & -3 & 3 & -1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ t \\ t^2 \\ t^3 \end{bmatrix}$$

$$P(t)' = [P1, P2, P3, P4] \cdot \begin{bmatrix} 1 & -3 & 3 & -1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 2t \\ 3t^2 \end{bmatrix}$$

（2）三次 B 样条曲线

$$P(t) = G_{Bj} \cdot M_B \cdot T_j = [P_{j-3}, P_{j-2}, P_{j-1}, P_j] \frac{1}{6} \begin{bmatrix} 1 & -3 & 3 & -1 \\ 4 & 0 & -6 & 3 \\ 1 & 3 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ t-j \\ (t-j)^2 \\ (t-j)^3 \end{bmatrix}$$

$$P(t)' = [P_{j-3}, P_{j-2}, P_{j-1}, P_j] \frac{1}{6} \begin{bmatrix} 1 & -3 & 3 & -1 \\ 4 & 0 & -6 & 3 \\ 1 & 3 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 2(t-j) \\ 3(t-j)^2 \end{bmatrix}$$

贝塞尔曲线与 B 样条曲线的关系：将 B 样条曲线中令 s = t j 可以得到类似贝塞尔曲线的形式，我们只需要对B样条曲线中每一段的控制点 G_{bj} 乘上 M_B M^{-1}_{BEZ} 然后按贝塞尔曲线生成即可。

$$P(s) = G_{Bj} \cdot M_B \cdot \begin{bmatrix} 1 \\ s \\ s^2 \\ s^3 \end{bmatrix} = \left(G_{Bj} \cdot M_B \cdot M_{BEZ}^{-1}\right) \cdot M_{BEZ} \cdot \begin{bmatrix} 1 \\ s \\ s^2 \\ s^3 \end{bmatrix}$$

1.3 实现思路

1.3.1 三次贝塞尔曲线

（1）初始化结果曲线数据结构：定义一个Curve类型的变量result，它将用来存储计算得到的曲线点（包括顶点位置、切线、法线和次法线）。
（2）初始化法向量和次法向量：初始化法向量N_prev为(0, 0, 1)，代表垂直于XY平面的向量。次法向量B_prev将在后续计算中初始化。
（3）遍历每一段曲线：通过循环遍历给定控制点P，每次处理一段曲线（3个控制点加上起始点）。
（4）在每一段曲线上细分：对每一段曲线，根据steps的值进行细分，即计算出该曲线段上steps+1个点。
（5）计算贝塞尔曲线上的点：对于每个参数t（从0到1），使用矩阵乘法计算出曲线上的点V，以及该点的切线T。然后，T被归一化。
（6）计算法线和次法线：如果是第一段的第一个点，将使用T和初始的N_prev通过叉乘计算第一个次法向量B_prev。接着，使用B_prev和当前的T计算当前点的法向量N，然后计算当前点的次法向量B。
（7）存储计算结果：将计算得到的顶点位置V、切线T、法线N和次法线B存储到result中。
（8）更新N和B用于下一个点的计算：更新N_prev和B_prev为当前点的N和B，为计算下一个点做准备。
（9）输出信息：最后，代码输出控制点信息和steps的值，以及表示完成曲线计算的信息。
（10）返回结果：函数返回计算得到的曲线result，其中包含了曲线上每个点的顶点位置、切线、法线和次法线，这些信息可以用于后续的渲染或其他处理。

1.3.2 三次 B 样条曲线

（1）定义基矩阵和逆矩阵：定义了B样条曲线的基矩阵MB，贝塞尔曲线的基矩阵MBEZ，以及贝塞尔基矩阵的逆矩阵MBEZ_inv。这些矩阵用于将B样条的控制点转换为贝塞尔控制点。
（2）转换控制点：遍历B样条曲线的控制点，每次处理一组4个控制点。由于B样条到贝塞尔的转换过程需要处理每组4个控制点，此过程将重复直到所有控制点都被处理。
（3）对于每组控制点，创建一个Matrix4f Gb，其中每一列代表一个控制点，扩展为齐次坐标（实际应用中，我们需要考虑如何正确填充这个矩阵，可能需要调整代码以匹配库的使用方式）。
应用转换矩阵：对每组控制点使用转换矩阵（MB * MBEZ_inv * Gb）来计算对应的贝塞尔控制点。这一步实际上将B样条控制点转换为贝塞尔控制点，使得可以利用贝塞尔曲线算法来渲染曲线。
（4）存储转换后的控制点：从转换后的矩阵Gb_bezier中提取每列，这些列代表转换后的贝塞尔控制点。然后将这些控制点添加到bezierControlPoints向量中，以便后续使用。
（5）生成贝塞尔曲线：最后，调用evalBezier函数，并传入转换后的贝塞尔控制点和步骤数steps。evalBezier将根据这些控制点和指定的细分步骤数计算曲线上的点，并返回最终的曲线数据。

1.4 具体代码：

1.4.1 三次贝塞尔曲线

```
Curve evalBezier(const vector<Vector3f>& P, unsigned steps) {
    // 检查是否有足够的控制点
    if (P.size() < 4 || P.size() % 3 != 1) {
        cerr << "evalBezier must be called with 3n+1 control points." << endl;
        exit(0);
    }

    Curve result;  // 存储曲线点的结果数据结构

    // 初始法向量N，垂直于xy平面
    Vector3f N_prev(0, 0, 1);
    Vector3f B_prev;  // 初始次法向量B0

        // 计算初始次法向量B0
        float t = static_cast<float>(0)/steps;
        Matrix4f MBEZ = Matrix4f(1, -3, 3, -1,
                                 0, 3, -6, 3,
                                 0, 0, 3, -3,
                                 0, 0, 0, 1);
    Vector4f TV = Vector4f(1, t, t*t, t*t*t);
    Vector4f TN = Vector4f(0, 1, 2*t, 3*t*t);
    Matrix4f Gb;
        for (size_t j = 0; j < 4; ++j) {
        Vector4f P_new(P[j], 1.0f);
        Gb.setCol(j, P_new);
    }
    Vector3f T = (Gb * MBEZ * TN).xyz();
    T.normalize();
        B_prev = Vector3f::cross(T, Vector3f::cross(N_prev, T)).normalized();

    // 遍历每段曲线
    for (size_t i = 0; i <= P.size() - 4; i += 3) {
        // 通过步数细分每段曲线
        for (unsigned step = 0; step <= steps; step++) {
                        if(i != 0)continue;
            float t = static_cast<float>(step) / steps;
            TV = Vector4f(1, t, t*t, t*t*t);
            TN = Vector4f(0, 1, 2*t, 3*t*t);
```

```
            for (size_t j = i; j < i + 4; ++j) {
                Vector4f P_new(P[j], 1.0f);
                Gb.setCol(j - i, P_new);
            }

            Vector3f V = (Gb * MBEZ * TV).xyz();
            Vector3f T = (Gb * MBEZ * TN).xyz();
            T.normalize();

            Vector3f N = Vector3f::cross(B_prev, T).normalized();
            Vector3f B = Vector3f::cross(T, N).normalized();

                        CurvePoint point;
                        point.V = V;
                        point.T = T;
                        point.N = N;
                        point.B = B;

            result.push_back(point);  // 添加计算好的点到结果中

            N_prev = N;
            B_prev = B;
        }
    }
    return result;
}
```

1.4.2 三次 B 样条曲线

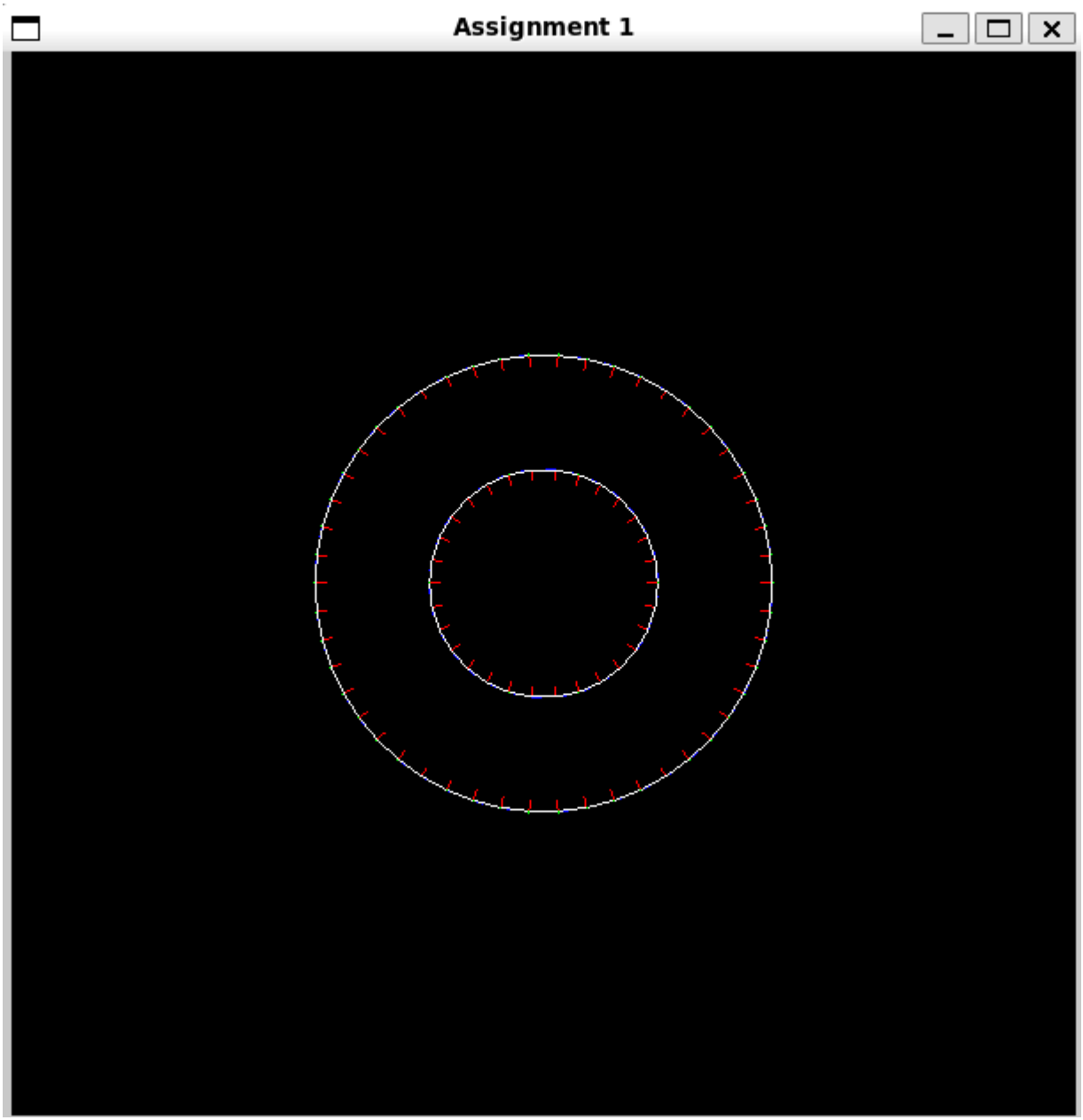
```
Curve evalBspline(const vector< Vector3f >& P, unsigned steps)
{
        // Check
        if (P.size() < 4)
        {
                cerr << "evalBspline must be called with 4 or more control points." << endl;
                exit(0);
        }

    Matrix4f MB; // B样条基矩阵
    Matrix4f MBEZ; // 贝塞尔基矩阵
    Matrix4f MBEZ_inv; // 贝塞尔基矩阵的逆

        // B样条基矩阵
    MB = Matrix4f(      1, -3, 3, -1,
                                4, 0, -6, 3,
                                1, 3, 3, -3,
                                0, 0, 0, 1);
    MB /= 6.0;

        // 贝塞尔基矩阵
    MBEZ = Matrix4f(1, -3, 3, -1,
                                0, 3, -6, 3,
                                0, 0, 3, -3,
                                0, 0, 0, 1);

        MBEZ_inv = MBEZ.inverse();

    Curve result;

        // 初始法向量N，垂直于xy平面
    Vector3f N_prev(0, 0, 1);
    Vector3f B_prev;  // 初始次法向量B0

        // 计算初始次法向量B0
        float t = static_cast<float>(0)/steps;
    Vector4f TV = Vector4f(1, t, t*t, t*t*t);
    Vector4f TN = Vector4f(0, 1, 2*t, 3*t*t);
    Matrix4f Gb;
        for (size_t j = 0; j < 4; ++j) {
        Vector4f P_new(P[j], 1.0f);
```

```
    Gb.setCol(j, P_new);
}
Vector3f T = (Gb * MBEZ * TN).xyz();
T.normalize();
    B_prev = Vector3f::cross(T, Vector3f::cross(N_prev, T)).normalized();

for (size_t i = 0; i < P.size() - 3; i++) {
    // 把B样条的控制点放入一个4x4的矩阵，最后一行填 1
    for (size_t j = i; j < i + 4; ++j) {
                    Vector4f P_new(P[j], 1.0f);
        Gb.setCol(j-i, P_new);
    }

    // 应用转换矩阵
    Matrix4f Gb_bezier = Gb * MB * MBEZ_inv;

            // 提取转换后的贝塞尔控制点
    vector<Vector3f> bezierControlPoints;

    // 把转换后的控制点加入结果数组
    for (size_t j = 0; j < 4; ++j) {
            Vector4f column = Gb_bezier.getCol(j);
            bezierControlPoints.push_back(column.xyz()); // 提取转换后的x, y, z坐标
            }

            for (unsigned step = 0; step <= steps; step++) {
        float t = static_cast<float>(step) / steps;
        TV = Vector4f(1, t, t*t, t*t*t);
        TN = Vector4f(0, 1, 2*t, 3*t*t);

        for (size_t j = 0; j < 4; ++j) {
            Vector4f P_new(bezierControlPoints[j], 1.0f);
            Gb.setCol(j, P_new);
        }

        Vector3f V = (Gb * MBEZ * TV).xyz();
        Vector3f T = (Gb * MBEZ * TN).xyz();
        T.normalize();

        Vector3f N = Vector3f::cross(B_prev, T).normalized();
        Vector3f B = Vector3f::cross(T, N).normalized();

                    CurvePoint point;
```

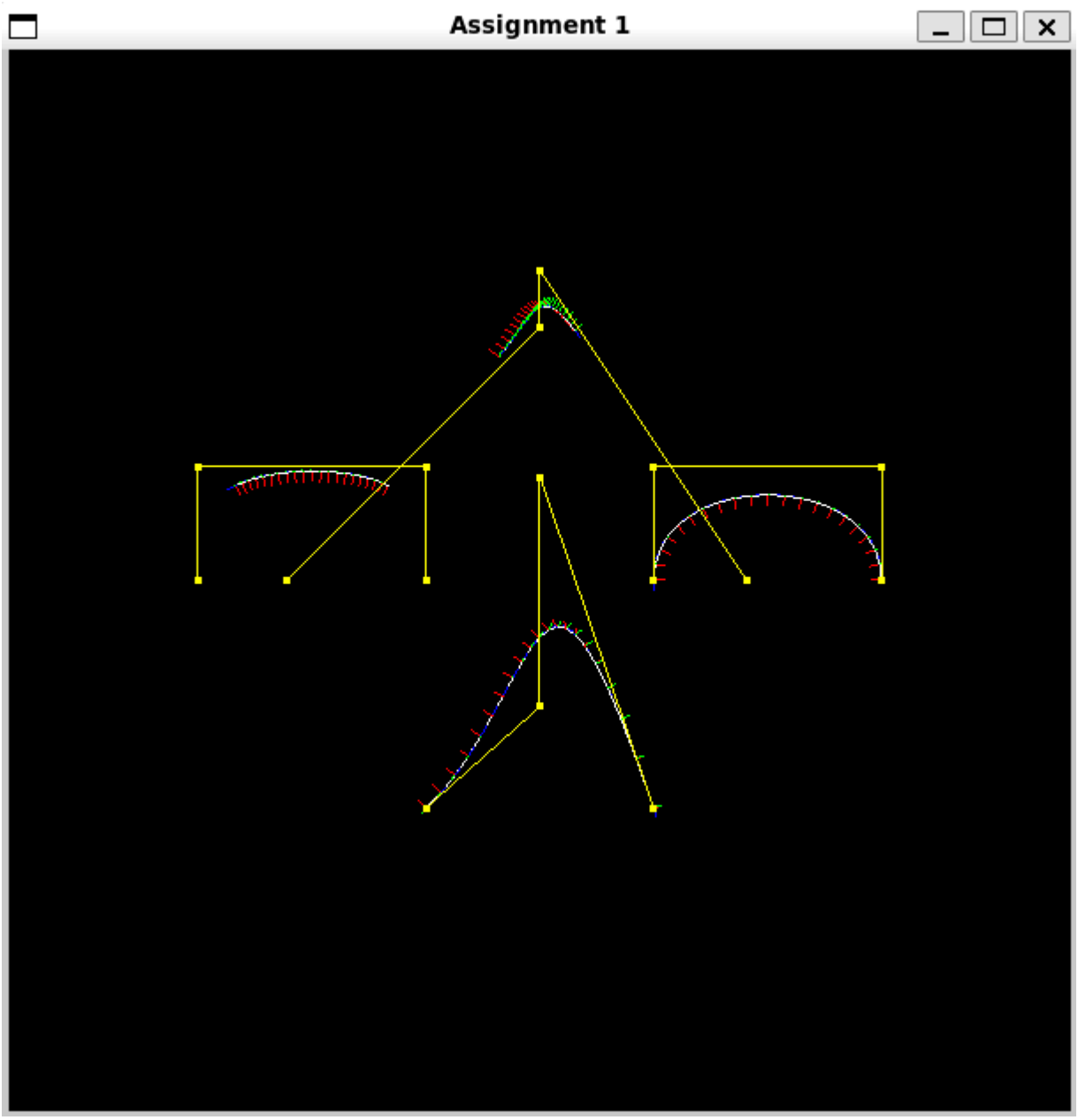
```
                        point.V = V;
                        point.T = T;
                        point.N = N;
                        point.B = B;

            result.push_back(point);  // 添加计算好的点到结果中

            N_prev = N;
            B_prev = B;
        }
    }
    return result;
}
```

1.5 运行截图

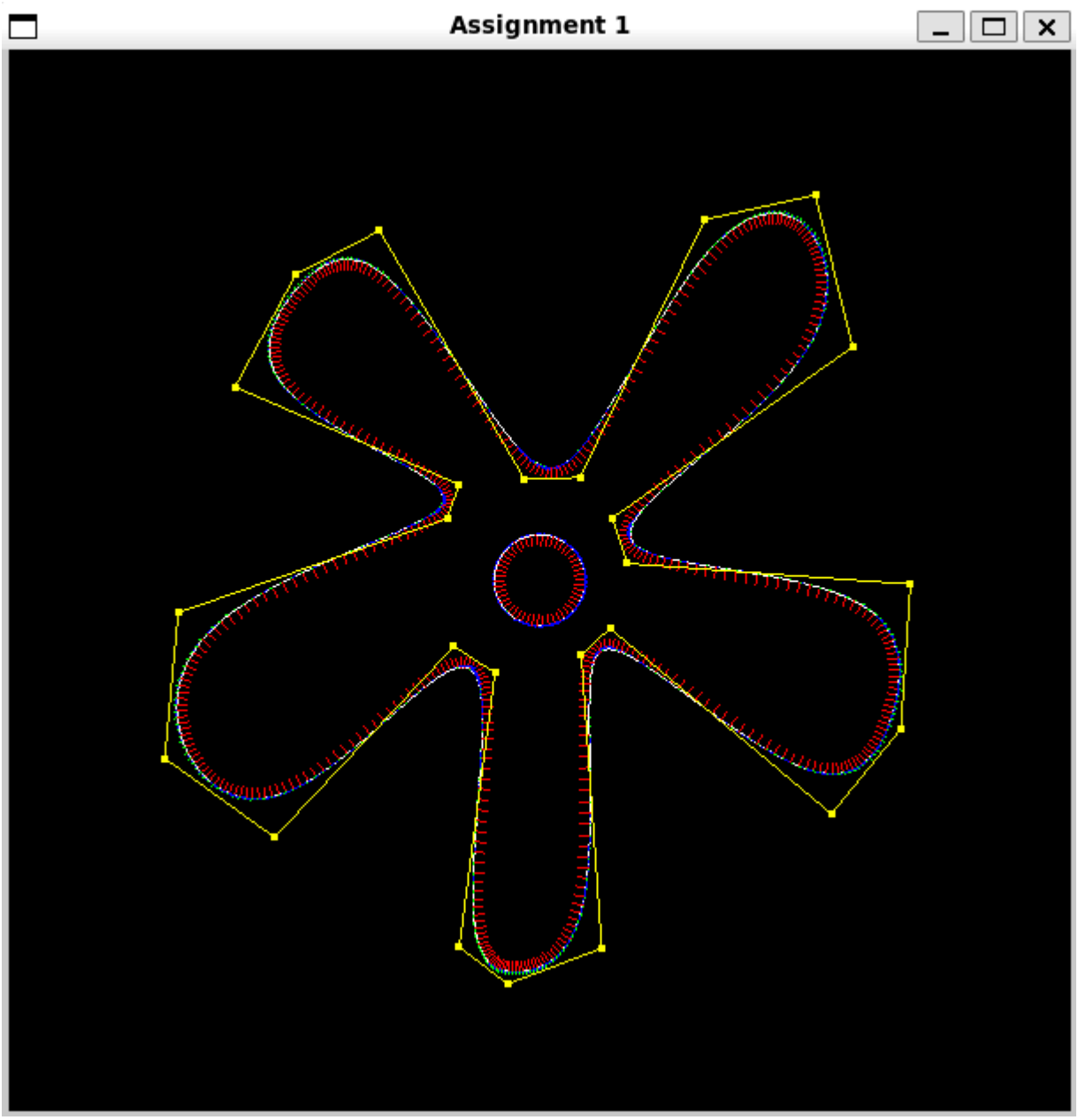
1.5.1 circles



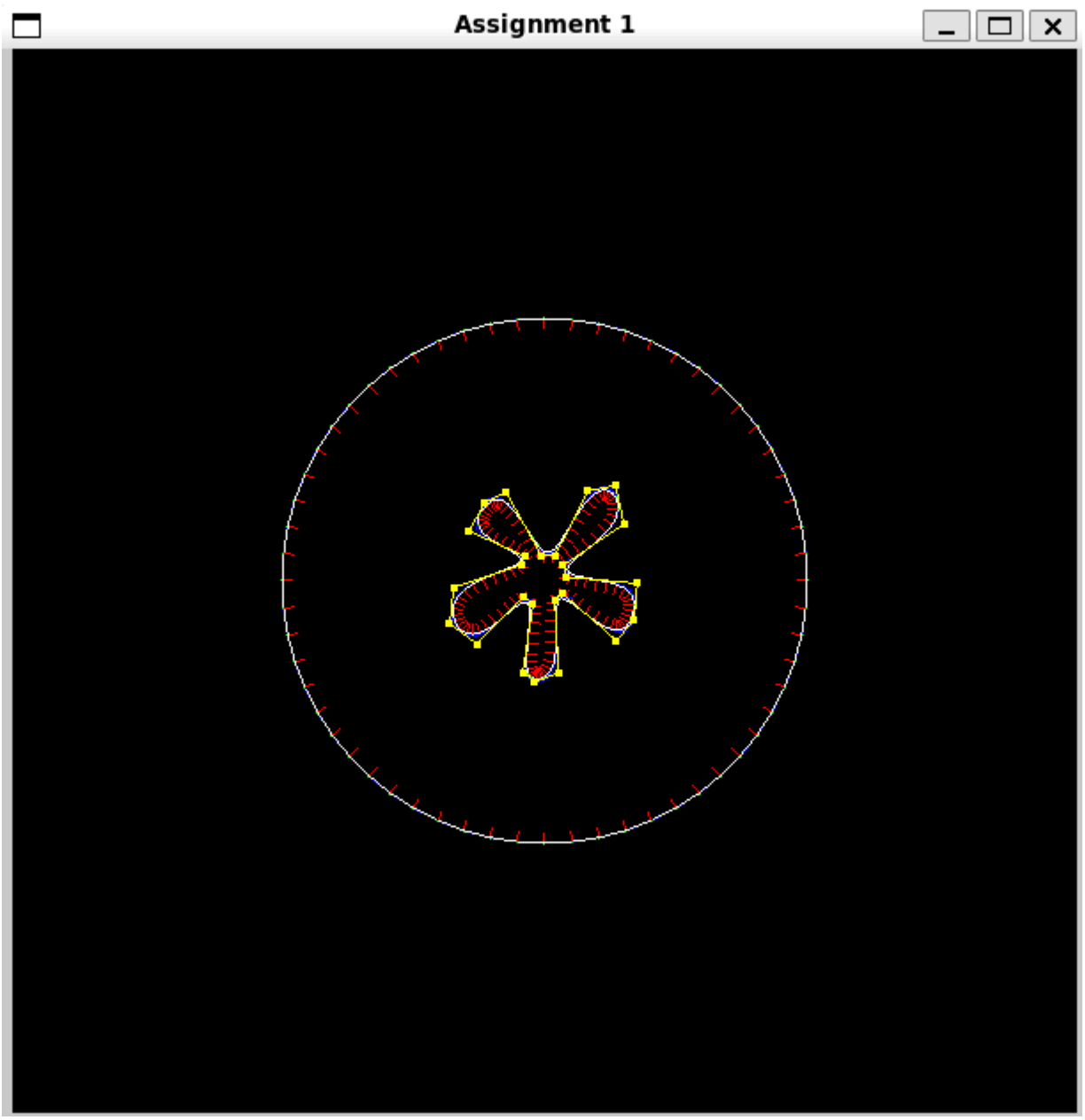
1.5.2 core



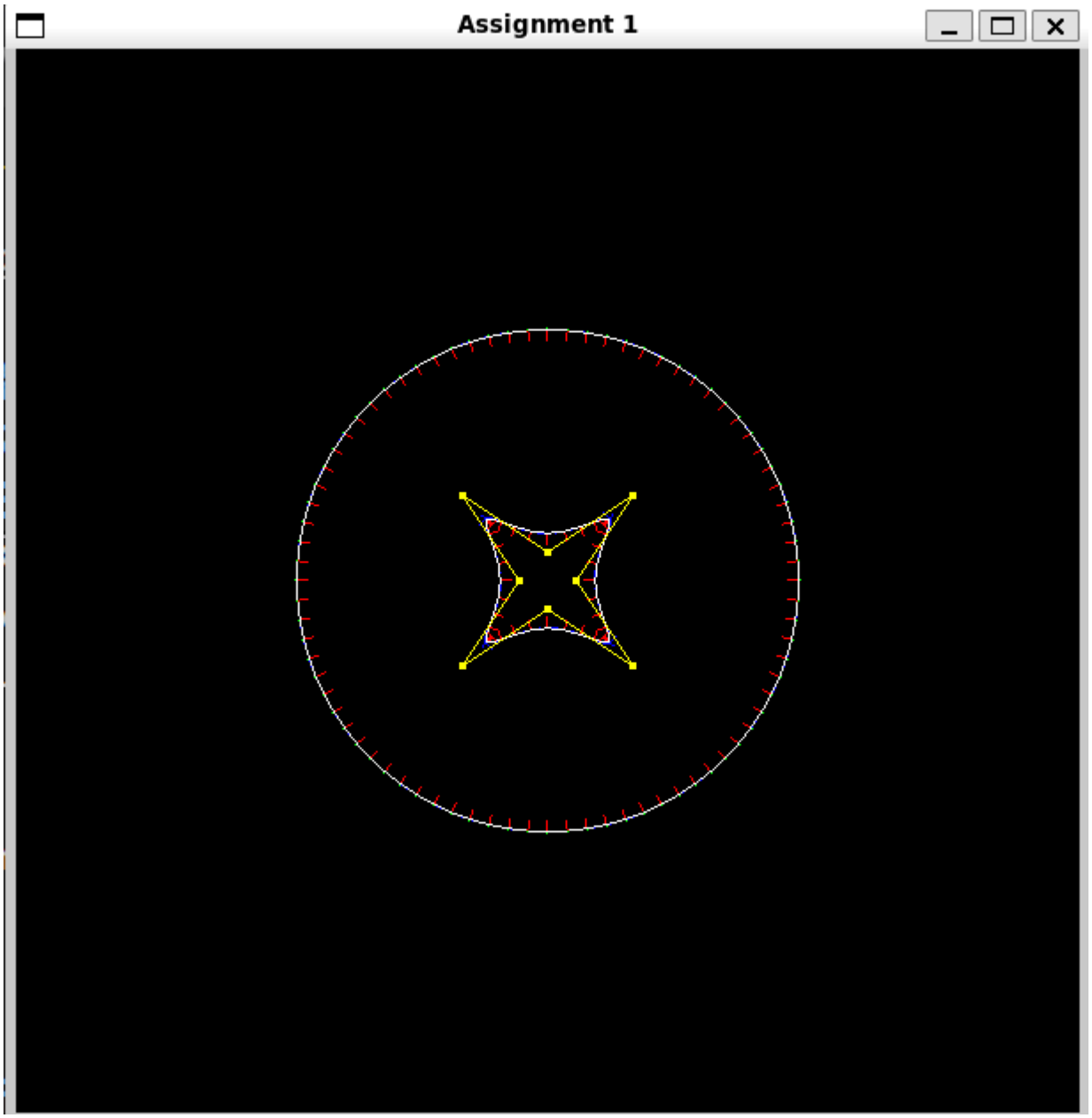
1.5.3 flircle



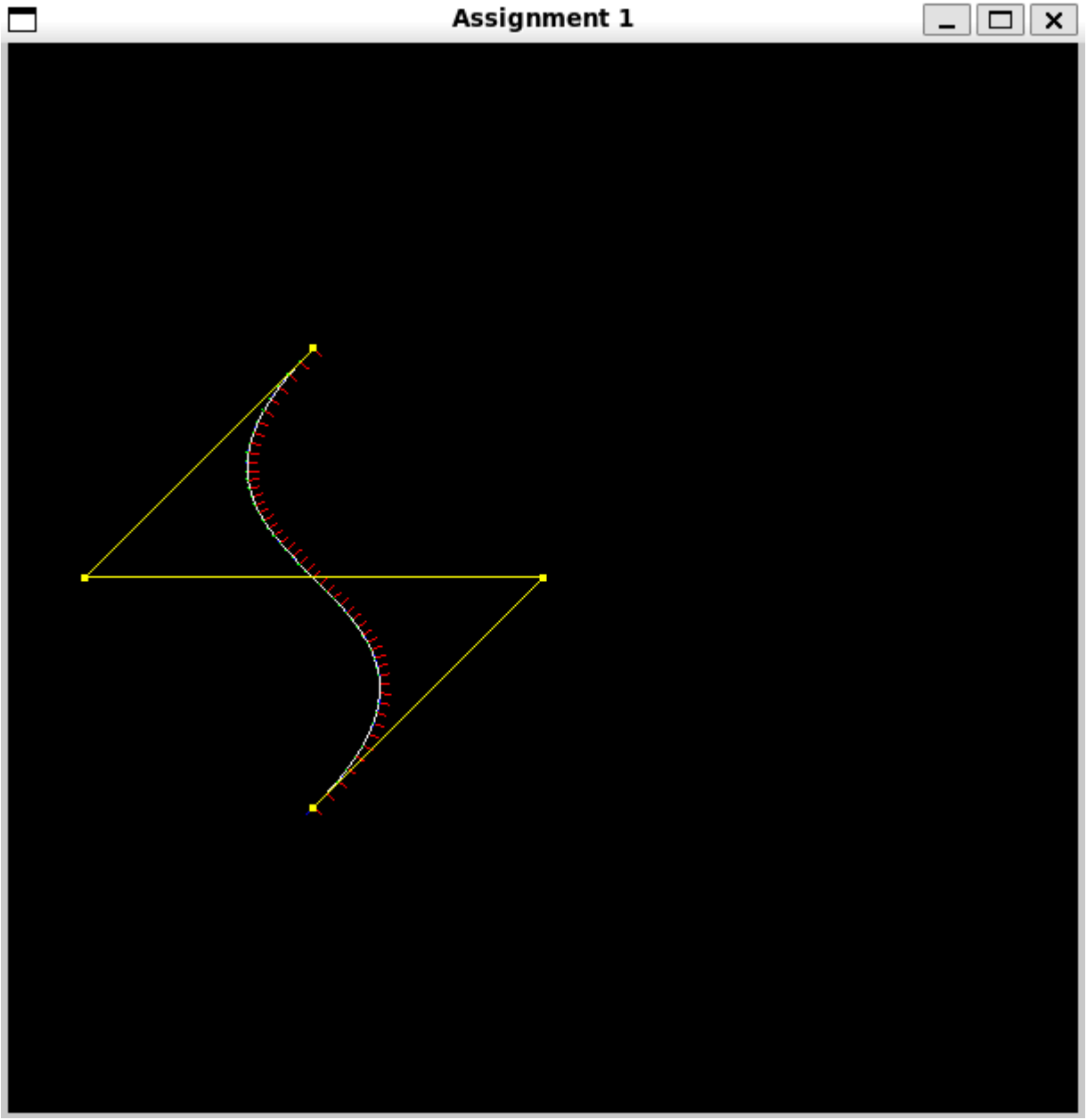
1.5.4 florus



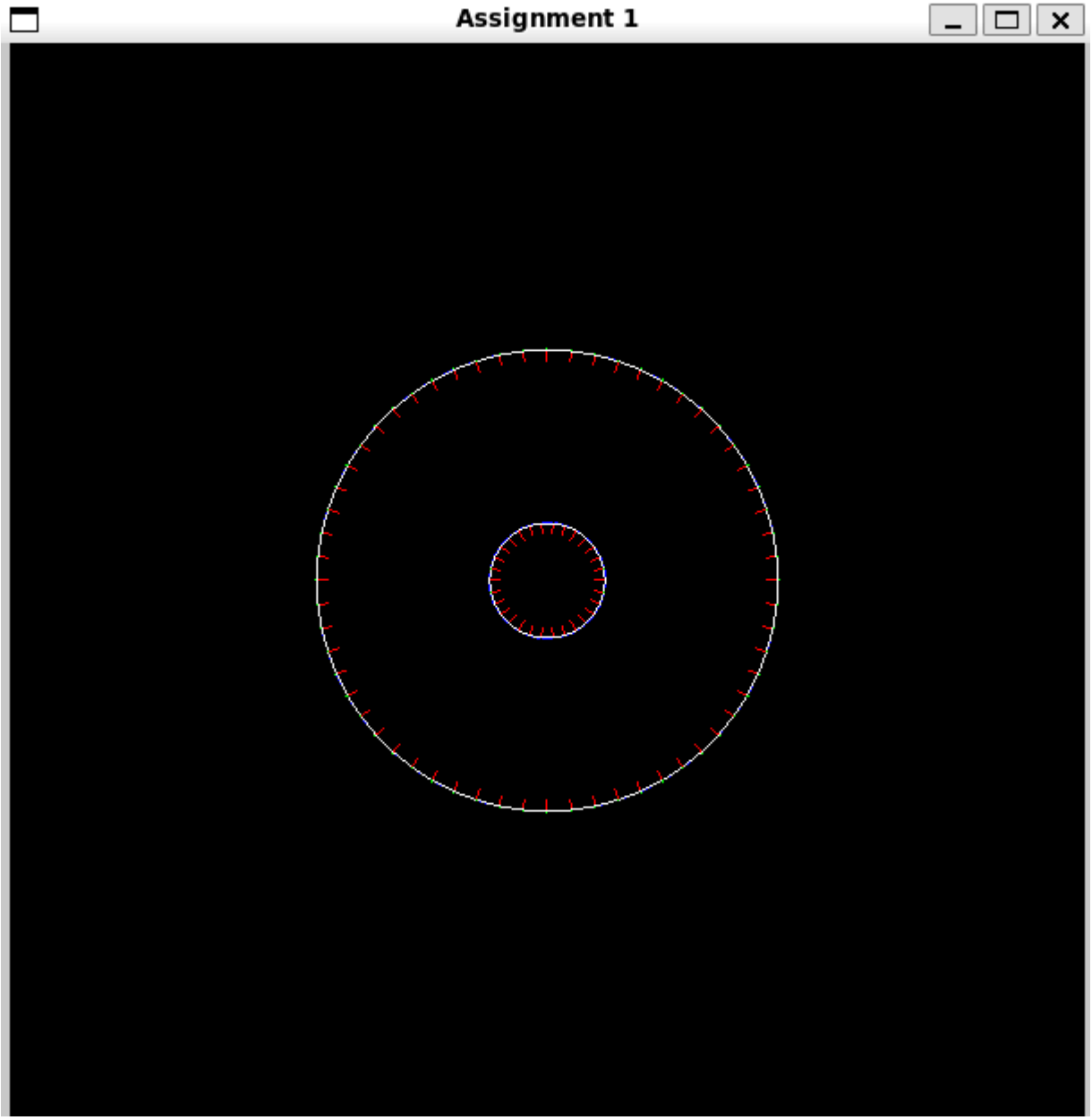
1.5.5 gentorus



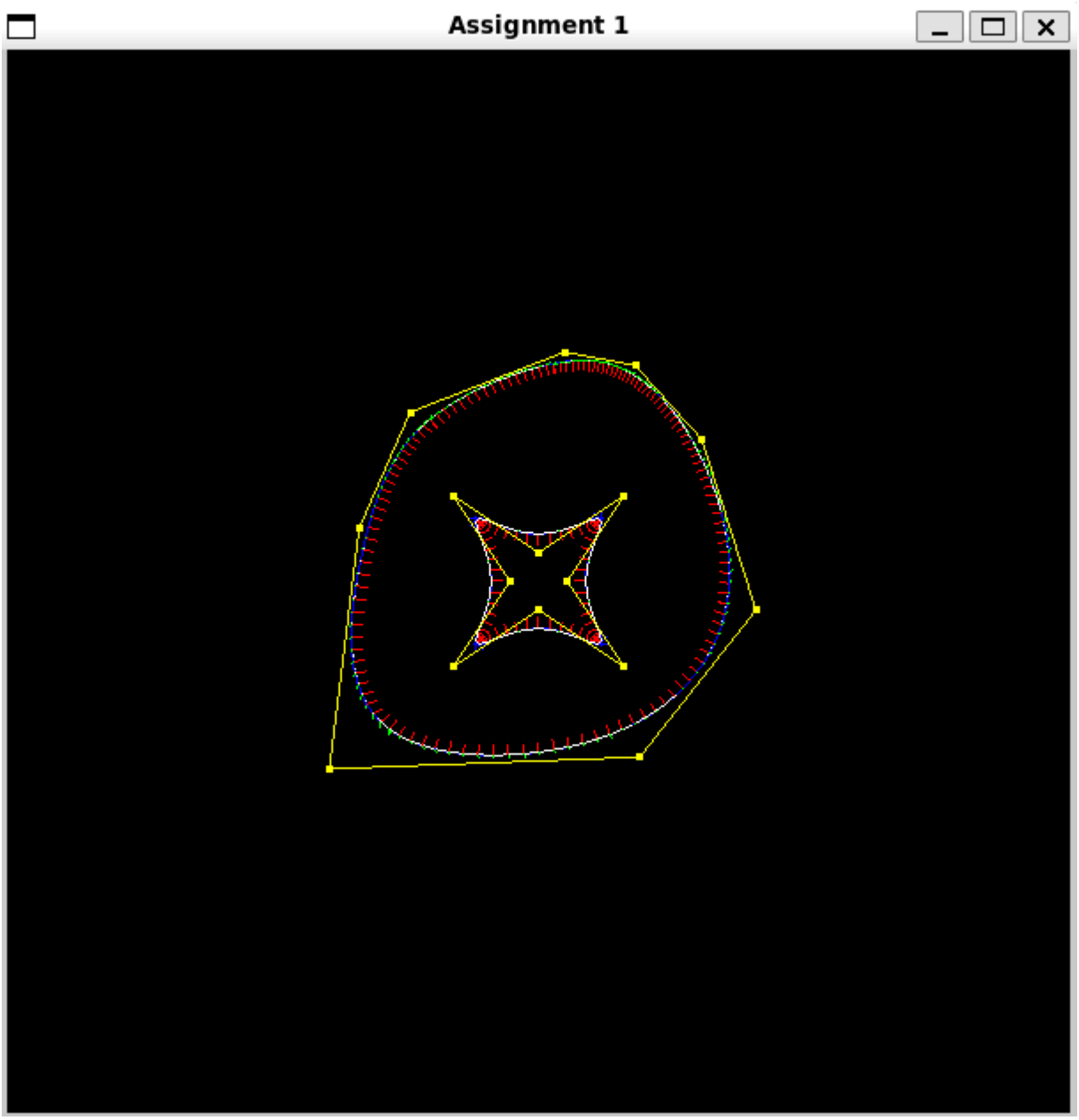
1.5.6 norm



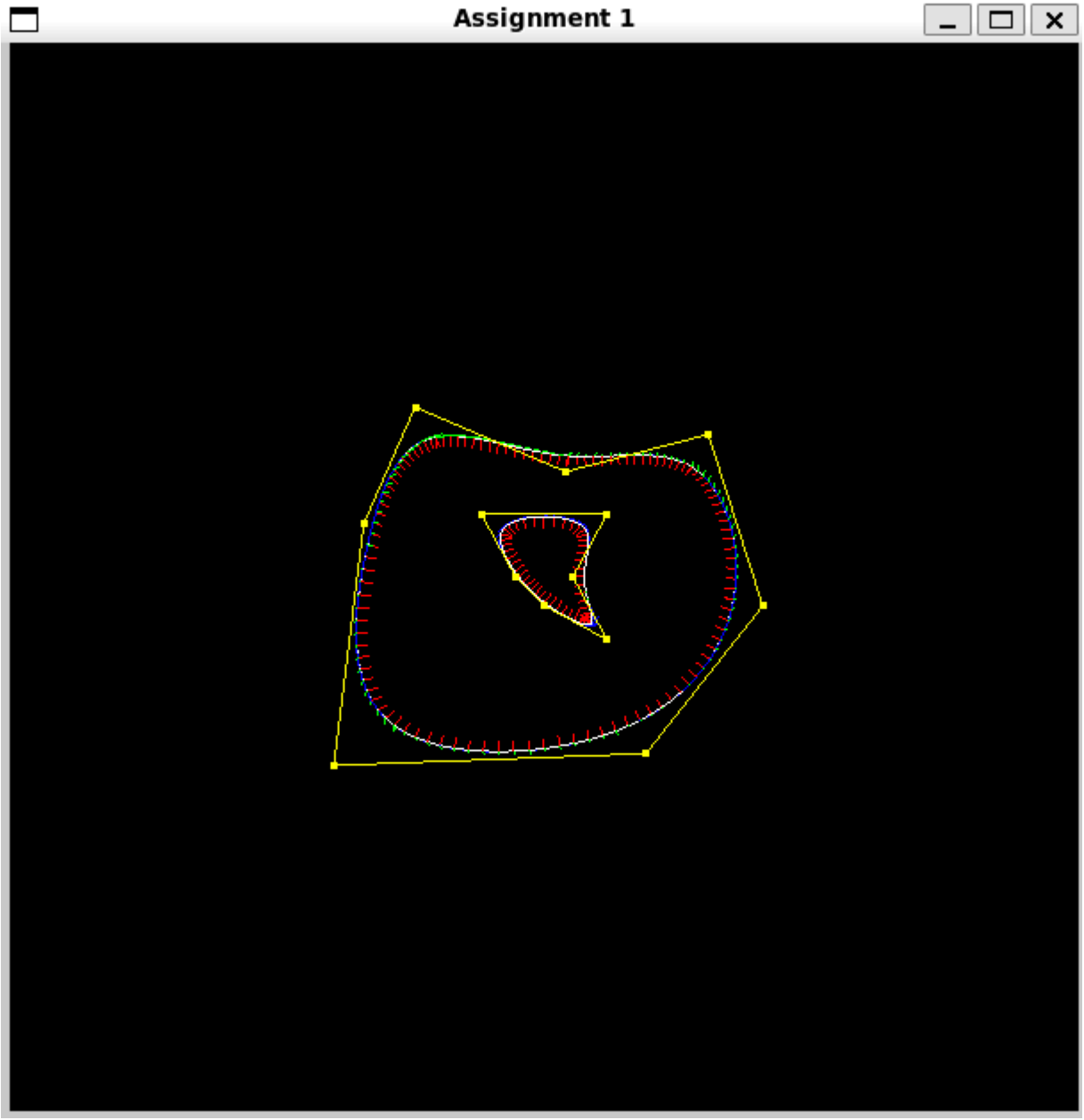
1.5.7 tor



1.5.8 weird



1.5.9 weirder



1.5.10 wineglass

2 曲面的绘制

2.1 要求

- 程序必须能够生成和显示旋转曲面（围绕y轴的xy平面上的曲线）和广义圆柱面。它必须正确地计算曲面法线。要在程序中演示这一点，应该有两种显示模式。一种模式应该显示以线框模式绘制的曲面，并将顶点法线从曲面向外绘制。另一个应该使用平滑的阴影来显示表面。
- 只需要在 surf.cpp 中填写 makeSurfRev 和 makegenCyl 函数。对于广义圆柱体，不需要精确地与 sample_solution 一致，因为它会任意选择初始坐标系（B0）。

2.2 理论知识

2.2.1 旋转曲面

将旋转曲面定义为围绕正 y 轴逆时针在 xy 平面上扫曲线的乘积，其中旋转的具体方向很重要。

法线：

从曲线的计算中，我们已经得到了法向量 N 。结果表明，如果我们用齐次变换矩阵 M 变换一个顶点，它的法线应该通过 M 的左上角 3×3 子矩阵的逆转置进行变换。在这个任务中，我们将假设，对于二维曲线，法线（红线）将总是指向遍历方向（蓝线）的左边。在此假设下，在将曲线法线应用于旋转曲面时，我们需要反转曲线法线的方向得到曲面的法线（法线反向），使得曲面法向量向外。从上至下的曲线的法线指向旋转轴，但是曲面的法线则远离旋转轴。

面：

除了定义法线我们还需要定义面。任务是生成三角形，连接轮廓曲线重复的曲线。基本的策略是在相邻的重复之间来回曲折来构建三角形。在OpenGL中，需要按照特定的顺序指定顶点和绘制法向量。它们必须按逆时针的顺序形成一个三角形（假设看的是三角形的前面）。

齐次坐标与二维变换的矩阵表示：

$$M = R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$P' = M \cdot P$$

$$N' = normalize((M^{-1})^T N)$$

2.2.2 广义圆柱体

广义圆柱体是通过重复轮廓曲线并用三角形连接轮廓曲线的每个副本而形成的。它和旋转曲面不同之处在于，将不是沿着 y 轴扫过二维轮廓曲线，而是沿着三维扫描曲线扫过它。

$$\mathbf{M} = \begin{bmatrix} \mathbf{N} & \mathbf{B} & \mathbf{T} & \mathbf{V} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

R 对应 N、B、T 三列；t 对应 V 列。

$$P' = M \cdot P$$

$$N' = normalize((M^{-1})^T N)$$

2.2.3 曲线的闭合问题

插值解决闭合问题：

我们将扫掠曲线（Sweep）的法向量（N）和次法向量（B）绕切向量（T）旋转θ=α/surf_size度，其中α为曲线起始法向量与结束法向量的夹角。、

罗德里格旋转公式：
将N带入v，T带入k，那么B=k×v，公式可以简化为如下：

$$N' = cos\theta \mathrm{N} + sin\theta \mathrm{T} \times \mathrm{N} = cos\theta \mathrm{N} + sin\theta \mathrm{B}$$

2.3 实现思路

2.3.1 建立曲面

（1）网格生成过程

- 外部循环: 这个循环遍历每一个“步骤”，或者说是沿着闭合轮廓的每一个段，构造一个围绕轴的闭环结构。
- 内部循环: 对于每个段，这个循环遍历轮廓上的每个顶点（除了最后一个点，因为最后一个点将会和下一段的起点重合）。

（2）顶点索引计算

- 当前顶点 (current): 这是当前段当前点的索引。计算公式为 s * profileSize + i，其中 s 是当前段的索引，i 是当前点在轮廓中的位置。
- 下一段的对应顶点 (next): 这是位于下一个段上与当前点对应的顶点的索引。计算公式为 ((s + 1) % steps) * profileSize + i。使用模运算确保当处理最后一个段时，索引会回到第一个段，形成闭环。

（3）三角形面片的创建

每对四边形生成两个三角形: 对于网格的每个四边形，通过以下顶点索引来定义两个三角形：

- 第一个三角形: 使用顶点 current, current + 1 和 next。
- 第二个三角形: 使用顶点 next, current + 1 和 next + 1。

这里顶点的选择确保了三角形的顶点是以逆时针顺序排列的。

2.3.2 旋转曲面

（1）输入验证

首先检查输入的轮廓曲线是否在 xy 平面上。因为算法假设所有旋转都是绕 z 轴进行的。如果轮廓不在 xy 平面上，函数将显示错误消息并退出。

（2）角度步长计算

计算每一步旋转的角度，将圆的总度数（2π）除以步数（steps），定义了每次迭代旋转的精度。

（3）空间预留

为了提高效率，在开始转换之前预留足够的空间来存储顶点和法线向量。也为面片预留空间，假定每个四边形可以分解为两个三角形，所以预留的数量是 (profile.size() - 1) * steps * 2。

（4）顶点转换和存储

对于每个步骤，计算一个旋转矩阵，该矩阵根据当前步骤的旋转角度来定义。旋转矩阵用于旋转轮廓上的每个点和对应的法线。使用旋转矩阵变换每个顶点的位置，将顶点位置转换为齐次坐标（添加一个额

外的 1 作为第四维），然后应用旋转矩阵。法线也被相应地转换，使用旋转矩阵的逆转置矩阵。法线在变换后被标准化，被反向以保证外向性。

（5）构建面片

使用 buildFaces 函数，利用存储的顶点来构建网格。

（6）返回生成的表面

最终返回构建好的表面结构。这个表面包含了所有的顶点、法线以及由顶点组成的面片。

2.3.3 广义圆柱体

（1）前提条件验证

首先检查提供的轮廓曲线是否在 xy 平面上。因为算法需要一个平面轮廓以简化变换计算过程。如果轮廓不是平面的，函数将报错并退出。

（2）扫掠路径处理

创建扫掠路径的副本，这样原始数据不会在变换过程中被修改，保持数据的完整性。对扫掠路径的法线进行插值处理，以确保在路径上的法线连续且平滑。

（3）变换矩阵的计算与应用

对每一个扫掠路径点，计算一个变换矩阵。设置矩阵的列为 N, B, T 和位置向量 V，其中位置向量 V 包含齐次坐标的 1。计算矩阵的逆转置，这是为了正确变换法线向量。对轮廓曲线的每个点，使用上述变换矩阵来计算新的位置和法线。

（4）存储变换后的顶点和法线

变换后的顶点和法线被添加到 surface 结构中，为下一步的网格生成做准备。

（5）生成网格

使用 buildFaces 函数，基于变换后的顶点数据构建面片。

（6）返回生成的表面

最终构建的表面包含了所有变换后的顶点和法线，以及构成表面的三角形面片。

2.3.4 曲线的闭合问题

（1） isClosedCurve

这个函数的目的是判断一条曲线是否闭合。闭合曲线的首尾点位置相同或非常接近。

- 输入参数：curve，代表曲线，每个点含有位置向量V。
- 实现思路：函数比较曲线的第一个点和最后一个点的位置。使用 (curve.front().V - curve.back().V).absSquared() 来计算两点间距离的平方，判断这个值是否小于 1e-6。如果小于这个阈值，则认为曲线闭合。

（2）interpolateNormals

这个函数用于在确认曲线闭合的情况下，对曲线上的法向量进行插值处理，使得法向量在曲线首尾闭合处平滑过渡。

- 输入参数：curve，代表曲线，每个点含有法向量N和次法向量B。
- 实现步骤：
  （1）检查闭合：如果曲线不闭合，直接返回。
  （2）检查法向量一致性：比较曲线起始和结束点的法向量是否接近相同。如果非常接近，则不需要插值。
  （3）计算夹角：通过点积计算起始和结束法向量之间的夹角alpha。
  （4）分配旋转角度：将总夹角平均分配到曲线上的每一点。
  （5）更新法向量：对每个点的法向量进行旋转。

（3）interpolateBinormals

这个函数与 interpolateNormals 非常相似，只不过它处理的是次法向量B的插值。

- 实现步骤：
  （1）检查闭合：与 interpolateNormals 相同。
  （2）检查次法向量一致性：比较曲线起始和结束点的次法向量。
  （3）计算夹角：与处理法向量相似，通过点积计算。
  （4）分配旋转角度：平均分配夹角到每个点。
  （5）更新次法向量：对每个点的次法向量进行旋转。

总结

这些函数的共同目标是确保在曲线闭合的情况下，其几何特性（法线和次法线）在曲线的起点和终点之间能平滑过渡，从而在视觉和计算上都达到一致性。

2.4 具体代码

2.4.1 建立曲面

```
void buildFaces(Surface& surface, unsigned steps, unsigned profileSize) {
    // 遍历每一步
    for (unsigned s = 0; s < steps; ++s) {
        // 遍历profile中的每个点，除了最后一个，因为它将在下一个循环中作为起始点
        for (unsigned i = 0; i < profileSize - 1; ++i) {
            // 计算四边形顶点的索引
            unsigned int current = s * profileSize + i;
            unsigned int next = (s + 1) % steps * profileSize + i;

            // 为每个四边形创建两个三角形，确保顶点是逆时针顺序
            surface.VF.push_back(Tup3u(current, current + 1, next));
            surface.VF.push_back(Tup3u(next, current + 1, next + 1));
        }
    }
}
```

2.4.2 旋转曲面

```
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);
    }

    const float angleStep = 2.0f * M_PI / steps;

    // Reserve space for efficiency
    surface.VV.reserve(profile.size() * steps);
    surface.VN.reserve(profile.size() * steps);
    surface.VF.reserve((profile.size() - 1) * steps * 2); // two triangles per quad

    // Iterate over each step
    for (unsigned s = 0; s < steps; ++s) {
        // Compute rotation matrix for this step
        float angle = s * angleStep;
        Matrix4f rotationMatrix(
            cos(angle), 0, sin(angle), 0,
            0, 1, 0, 0,
            -sin(angle), 0, cos(angle), 0,
            0, 0, 0, 1
        );

        // Transform profile curve vertices and normals using the rotation matrix
        for (unsigned i = 0; i < profile.size(); ++i) {
            Vector4f pos(profile[i].V[0], profile[i].V[1], profile[i].V[2], 1);
            Vector4f normal(profile[i].N[0], profile[i].N[1], profile[i].N[2], 0);

            pos = rotationMatrix * pos;
            normal = rotationMatrix.inverse().transposed() * normal;  // Remove normalization te
            normal = -normal.normalized();  // Negate and normalize the vector to reverse the di


            // Add the vertex position and normal to the surface
            surface.VV.push_back(pos.xyz());
```

```
            surface.VN.push_back(normal.xyz());
        }
    }

    // 使用公共函数构建面片
    buildFaces(surface, steps, profile.size());
    return surface;
}
```

2.4.3 广义圆柱体

```
Surface makeGenCyl(const Curve &profile, const Curve &sweep) {
    Surface surface;

    // 创建 sweep 副本以进行修改
    Curve sweepCopy = sweep;

    // 检查副本是否闭合，并对扫描曲线副本应用法线插值
    interpolateNormals(sweepCopy);
    interpolateBinormals(sweepCopy);
    sweepCopy.pop_back();

    if (!checkFlat(profile)) {
        cerr << "genCyl profile curve must be flat on xy plane." << endl;
        exit(0);
    }

    // 遍历sweep副本曲线，计算变换矩阵，并应用它们到profile曲线的每个点上
    Matrix4f M_inv_transpose;
    for (size_t i = 0; i < sweepCopy.size(); ++i) {
        const CurvePoint& sweepPt = sweepCopy[i];

        // 创建变换矩阵M，将轮廓曲线变换到正确的位置
        Matrix4f M = Matrix4f::identity();
        M.setCol(0, Vector4f(sweepPt.N, 0));
        M.setCol(1, Vector4f(sweepPt.B, 0));
        M.setCol(2, Vector4f(sweepPt.T, 0));
        M.setCol(3, Vector4f(sweepPt.V, 1));
        M_inv_transpose = M.inverse().transposed();

        for (size_t j = 0; j < profile.size(); ++j) {
            const CurvePoint& profilePt = profile[j];
            Vector4f P(profilePt.V, 1);
            Vector4f N(profilePt.N, 0);

            // 变换轮廓点和法向量
            Vector4f Pprime = M * P;
            Vector4f Nprime = M_inv_transpose * N;
            Nprime = -Nprime.normalized();

            // 将变换后的点Pprime和法向量Nprime添加到surface数据结构
            surface.VV.push_back(Pprime.xyz());
```

```
            surface.VN.push_back(Nprime.xyz());
        }
    }

    // 使用公共函数构建面片
    buildFaces(surface, sweepCopy.size(), profile.size());
    return surface;
}
```

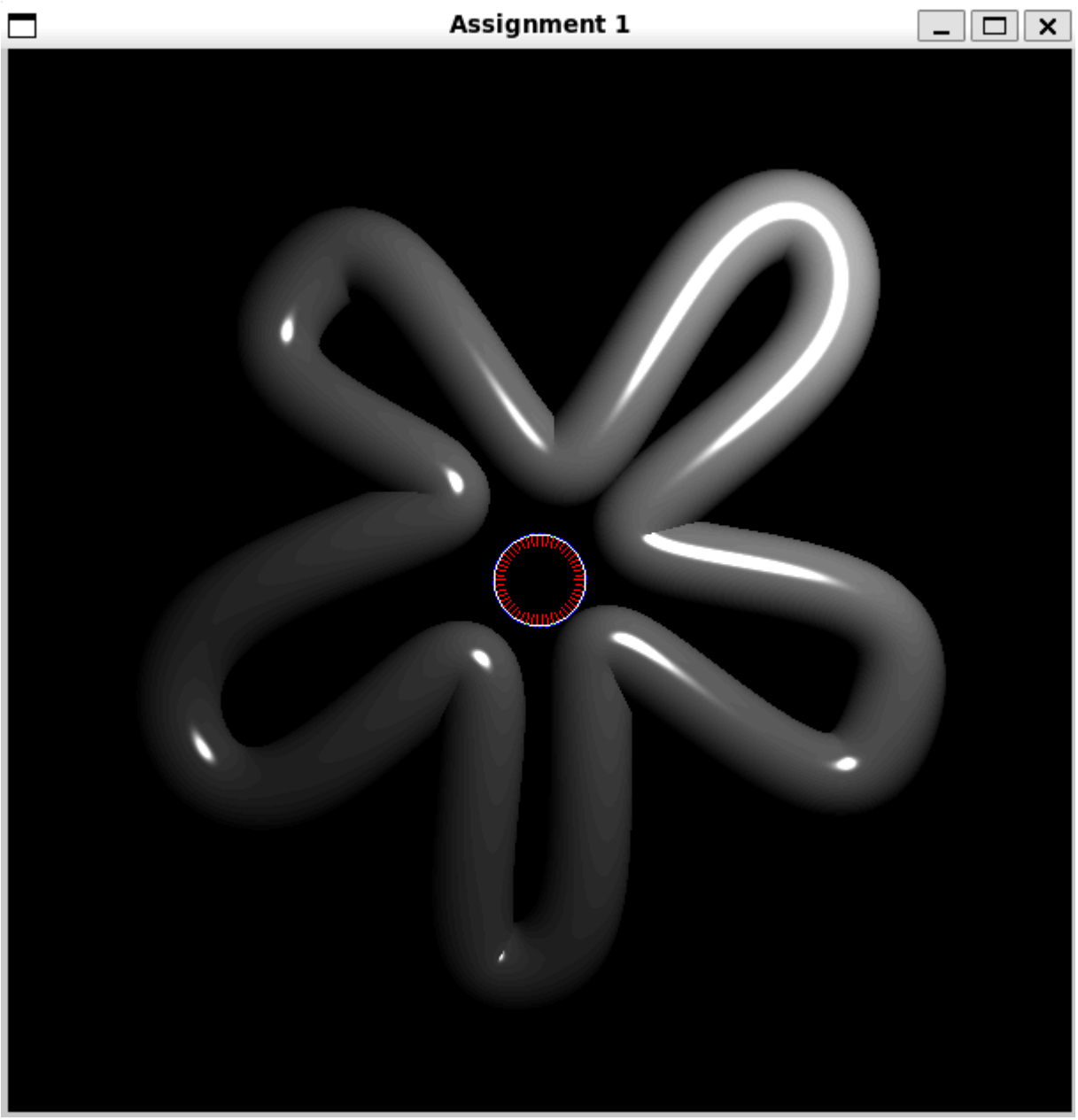
2.4.4 曲面的闭合问题

```
bool isClosedCurve(const Curve& curve) {
    // 判断曲线是否闭合：比较曲线首尾点的位置
    return (curve.front().V - curve.back().V).absSquared() < 1e-6;
}



void interpolateNormals(Curve& curve) {
    if (!isClosedCurve(curve)) return;

    // 获取曲线的起始和结束法向量
    Vector3f startNormal = curve.front().N;
    Vector3f endNormal = curve.back().N;

    if (startNormal[0] - endNormal[0] < 1e-5 && startNormal[1] - endNormal[1] < 1e-5 && startNor

    // 计算起始和结束法向量之间的夹角 alpha
    float dot = Vector3f::dot(startNormal, endNormal);
    dot = std::max(-1.0f, std::min(1.0f, dot)); // 限制点积的范围为[-1, 1]以防止计算错误
    float alpha = std::acos(dot);

    // 计算每一段的旋转角度 theta
    float theta = alpha / curve.size();  // 使用 curve.size() 代替 surf_size

    for (size_t i = 0; i < curve.size(); ++i) {
        // 获取当前点的法向量和次法向量
        Vector3f N = curve[i].N.normalized();
        Vector3f B = curve[i].B.normalized();

        // 计算当前点的法线旋转角度
        float currentTheta = -theta * i; // i 从 0 到 curve.size()-1，平均分配角度

        // 使用简单的二维旋转公式更新法线
        Vector3f rotatedNormal = (std::cos(currentTheta) * N + std::sin(currentTheta) * B).norma

        // 标准化并更新当前点的法线
        curve[i].N = rotatedNormal;
    }
}



void interpolateBinormals(Curve& curve) {
```

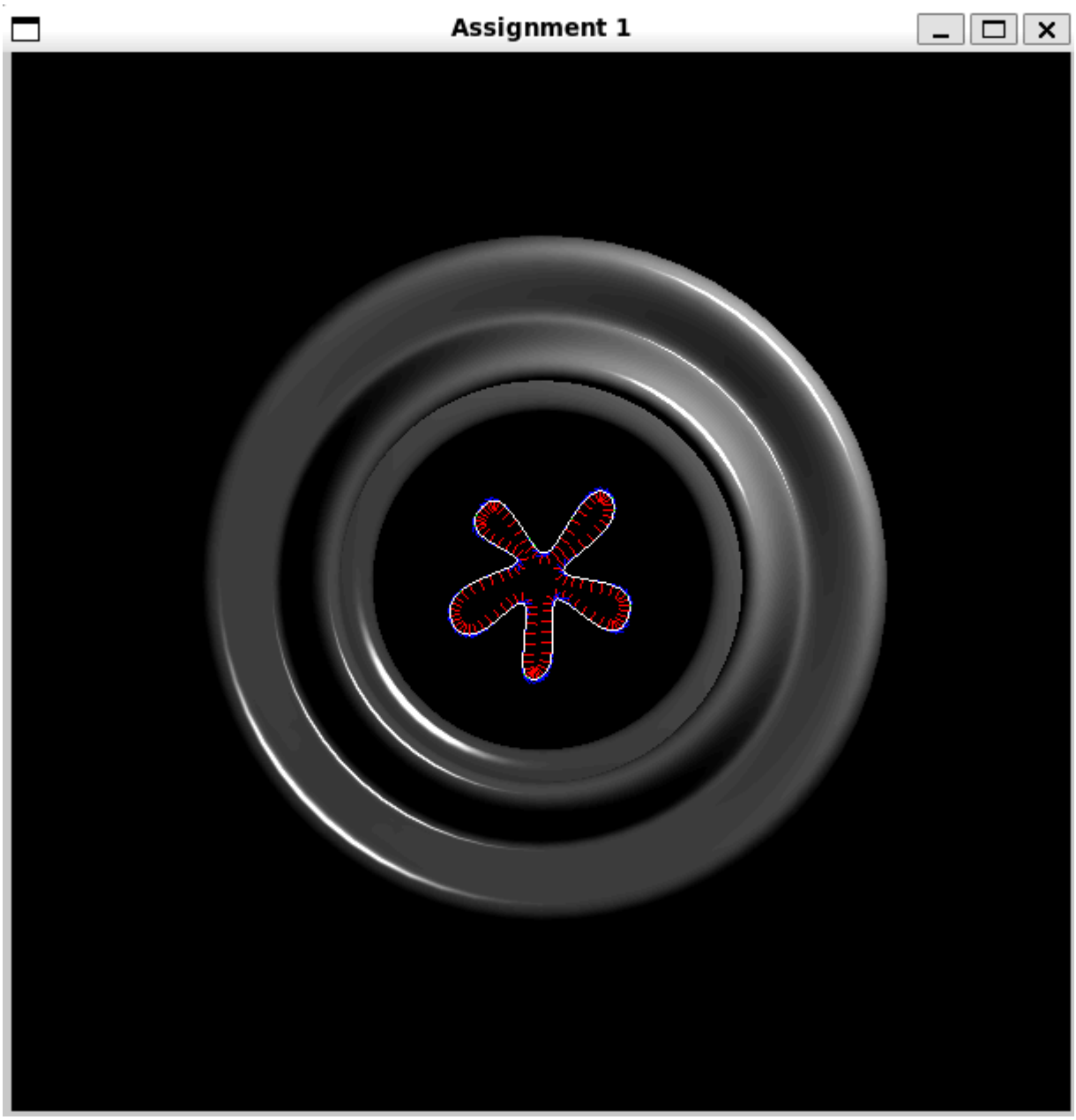
```
    if (!isClosedCurve(curve)) return;

    // 获取曲线的起始和结束次法向量
    Vector3f startBinormal = curve.front().B;
    Vector3f endBinormal = curve.back().B;

    if (startBinormal[0] - endBinormal[0] < 1e-5 && startBinormal[1] - endBinormal[1] < 1e-5 &&

    // 计算起始和结束次法向量之间的夹角 alpha
    float dot = Vector3f::dot(startBinormal, endBinormal);
    dot = std::max(-1.0f, std::min(1.0f, dot)); // 限制点积的范围为[-1, 1]以防止计算错误
    float alpha = std::acos(dot);

    // 计算每一段的旋转角度 theta
    float theta = alpha / curve.size();  // 使用 curve.size() 代替 surf_size

    for (size_t i = 0; i < curve.size(); ++i) {
        // 获取当前点的法向量和次法向量
        Vector3f N = curve[i].N.normalized();
        Vector3f B = curve[i].B.normalized();

        // 计算当前点的法线旋转角度
        float currentTheta = theta * i; // i 从 0 到 curve.size()-1，平均分配角度

        // 使用公式更新法线
        Vector3f rotatedBinormal = (std::cos(currentTheta) * B + std::sin(currentTheta) * N).nor

        // 标准化并更新当前点的法线
        curve[i].B = rotatedBinormal;
    }
}
```

2.5 运行截图

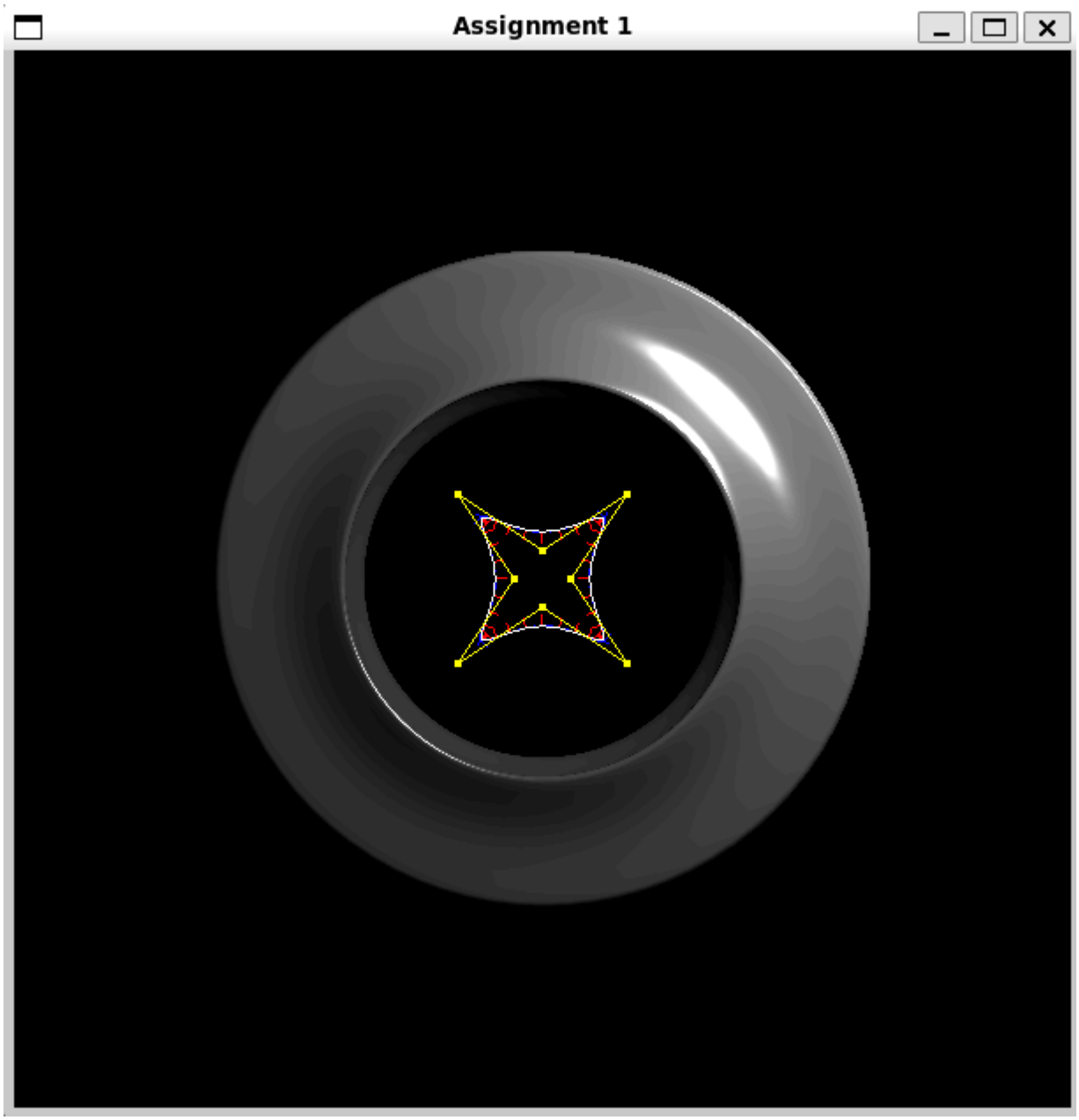
1.5.1 flircle



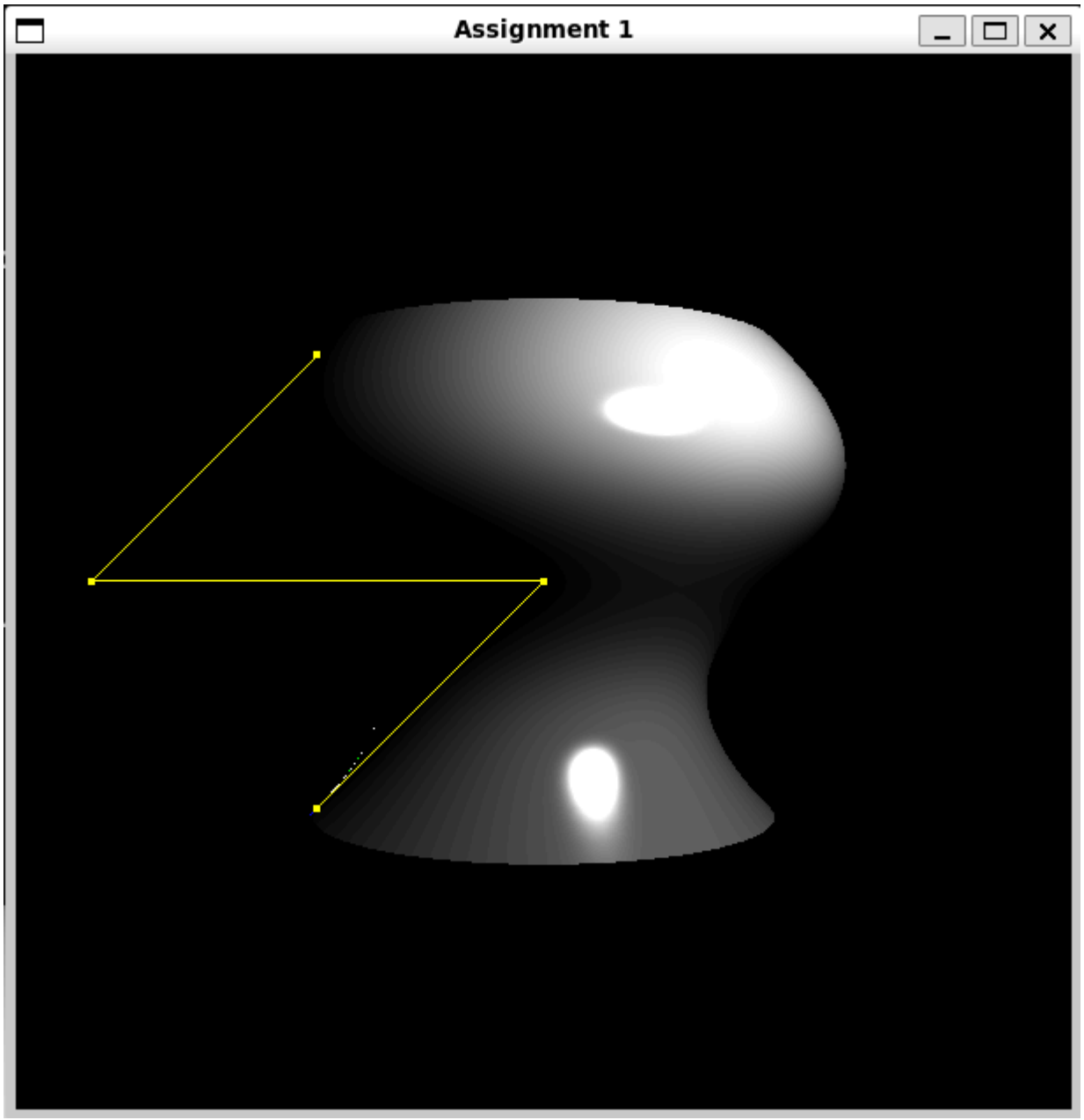
1.5.2 florus



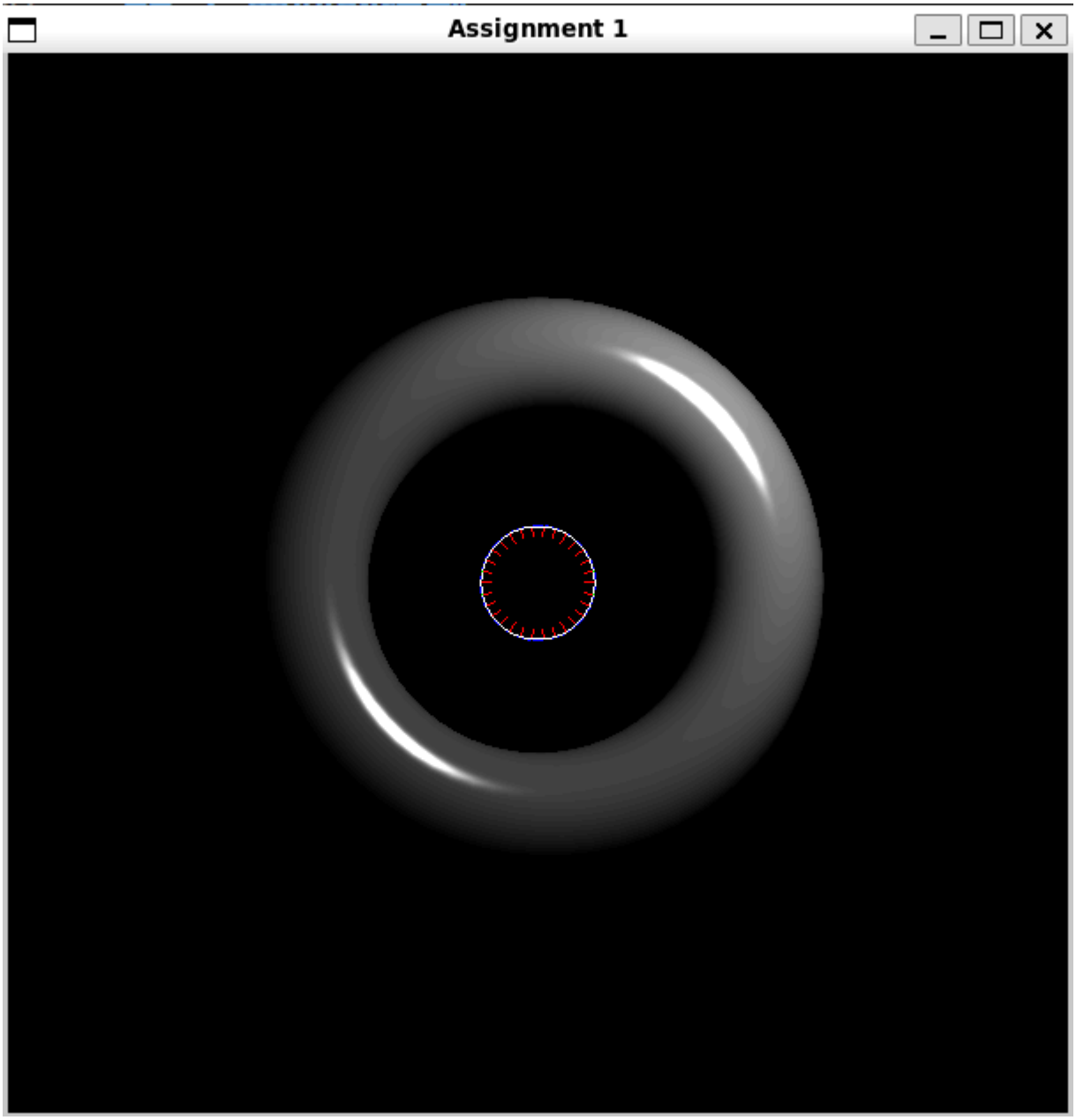
1.5.3 gentorus



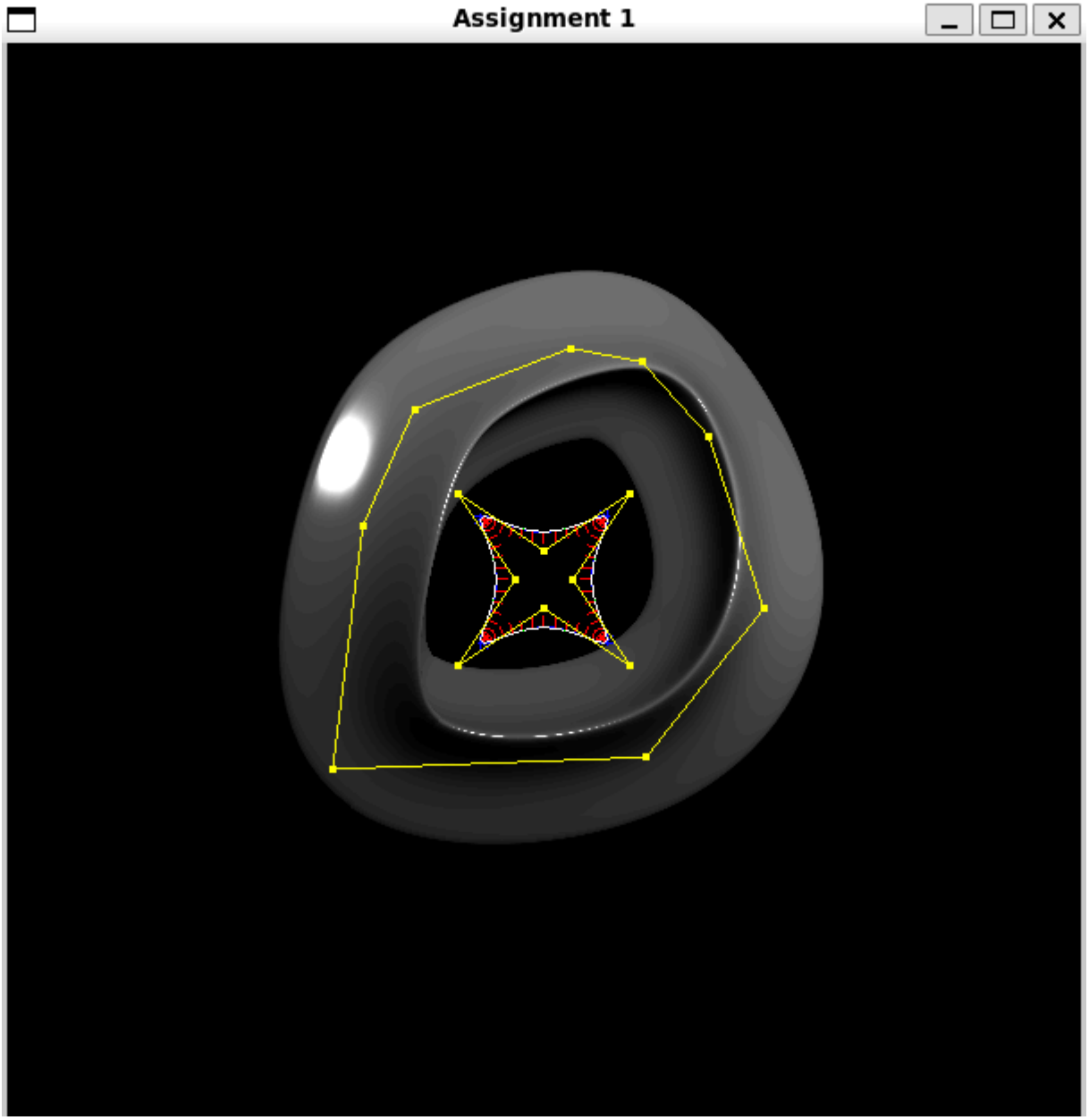
1.5.4 norm



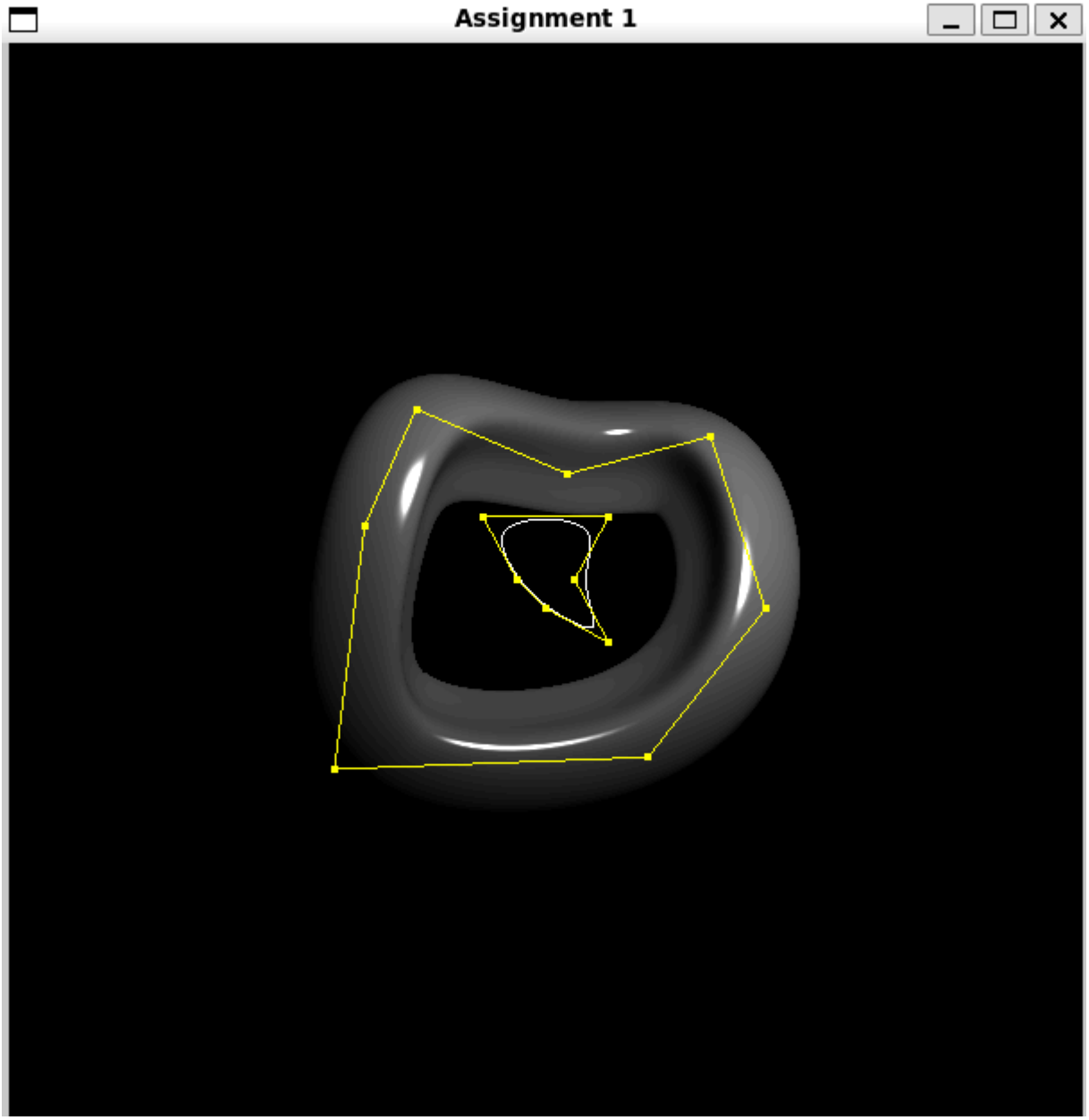
1.5.5 tor



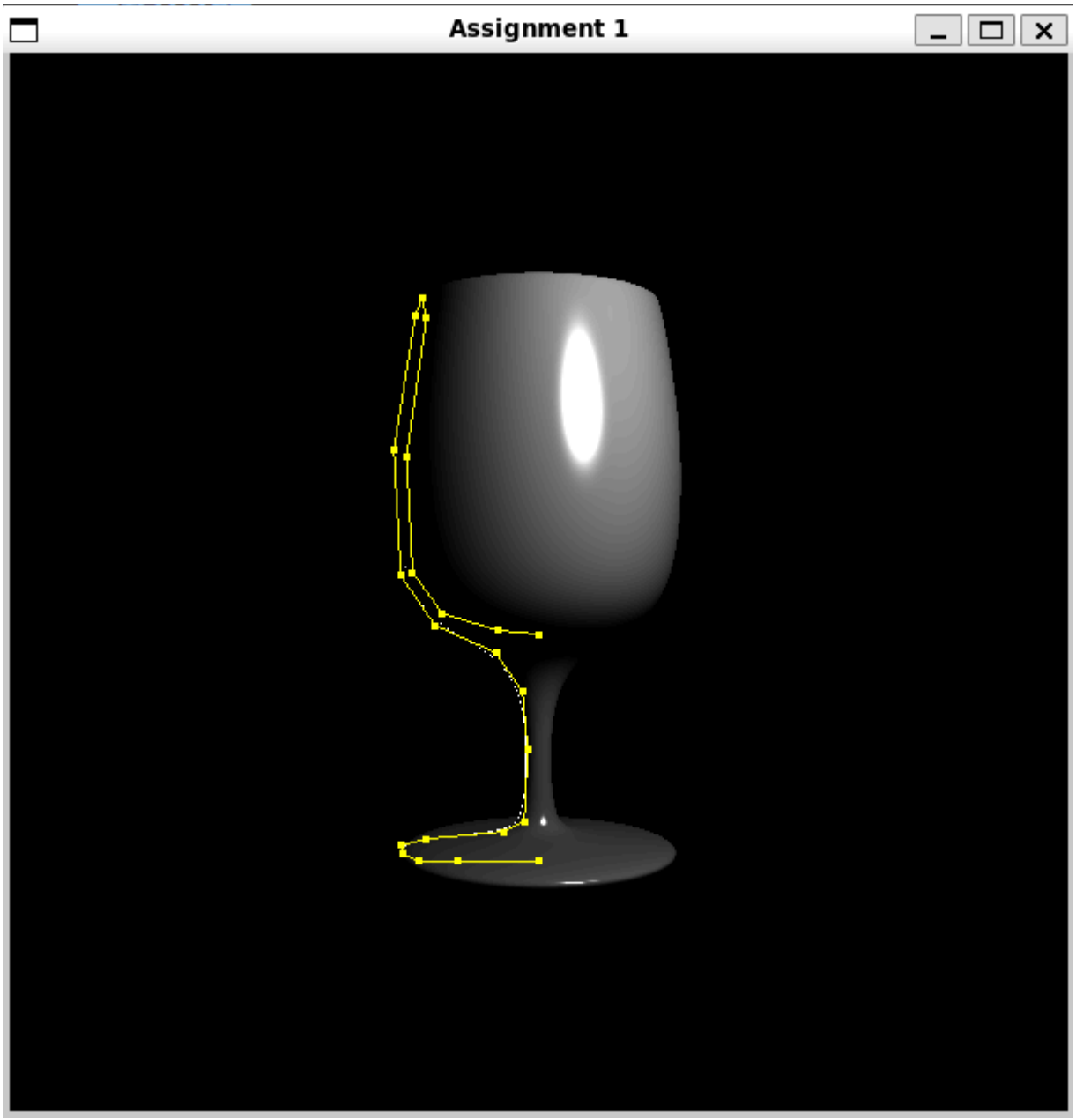
1.5.6 weird



1.5.7 weirder



1.5.8 wineglass



3 总结

在本次实验中主要遇到了两个困难：

（1）在绘制广义圆柱体时，每个sweep的控制点处都有断层，刚开始不知道是什么问题，改了很久广义圆柱体的代码，后来在与助教讨论后发现问题出在B样条曲线的绘制，每次调用贝塞尔曲线函数都重新初始化了B0，应该只需要初始化一次。

（2）在解决曲线的闭合问题时，刚开始发现插值后没有什么变化，于是输出结果后发现旋转方向反了，改成反向就能让首尾点法线相同了，但是次法线还是不一样，后来将次法线也旋转即可。