作业 2-Bezier 曲线绘制与旋转扫描建模

1. OpenGL 环境

本次作业采用 Freeglut+Glew 可编程管线，使用 CLion+CMake(MinGW)进行编译。

2. 程序使用说明

程序包含三个工作模式，分别为曲线绘制模式、曲线调整模式、模型显示模式，不同模式下可以完成不同的操作。

2.1. Bezier 曲线绘制与调整

程序启动后，默认进入曲线绘制模式，只可在屏幕右侧区域（即紫色分界线右侧）进行绘制，左侧曲线仅作为参考。点击鼠标左键添加一个控制点，点击鼠标右键结束当前曲线，开始下一条曲线的绘制，并默认将上一条曲线的终点作为第一个控制点；点击鼠标中键结束绘制，进入曲线调整模式，可以调整控制点的位置：按下鼠标左键选中控制点，然后将该控制点移动到指定位置，松开鼠标左键结束移动。绘制和调整曲线时，可以根据辅助线完成两条曲线的平滑连接。

2.2. 旋转扫描模型创建与观察

在曲线调整模式下，按下 C 键生成扫描模型，进入模型显示模式。多边形默认为线形模式，按 P 键切换为填充模式。使用鼠标左键拖动可以调整模型的空间角度，滑动鼠标中键可以调整模型与视点的距离。默认模型为灰蓝色表面，不包含纹理，按下 T 键可以导入 bmp 格式的纹理图并加载到当前模型上，支持导入多个纹理，按 N 键进行切换。sources/bmp 文件夹下存有 4 张效果较好的纹理图。

2.3. 模型的储存和读入

在模型显示模式下，点击 D 键可将模型导出到指定的 obj 文件；在曲线绘制模式下，点击 L 键可以导入指定的 obj 模型，并自动进入模型显示模式。obj 文件与标准格式一致，可以通过其他 3D 模型编辑器打开。为简化实现，事先规定了 obj 文件中的数据写入顺序，因此仅支持导入从本程序中导出的模型，导入其他 obj 文件可能会出现错误。sources/obj 文件夹下存有 5 个从本程序导出的模型。

3. Bezier 曲线绘制

3.1. 单条 Bezier 曲线的绘制

主程序中 Bezier 曲线样本点的位置参考 GitHub 上的样例程序，按照一般差值方法计算，计算公式如下：

$$C(t) = \sum_{i=0}^{n} P_i \, B_{i,n}(t)$$

$$B_{i,n}(t) = \frac{n!}{i!\,(n-i)!} t^i (1-t)^{n-i}$$

其中P为控制顶点，$B(t)$为调和函数。每次增加新的控制顶点后，重新计算所有样本点的位置，在不太要求性能的情况下，这种方式可以绘制任意阶次的 Bezier 曲线，灵活性更强。事先规定曲线的取样率，根据控制顶点线段的总长度确定取样数目，适当调整取样率能够以较少的样本点呈现平滑的曲线效果。下图为本程序绘制的一条 4 次 Bezier 曲线：

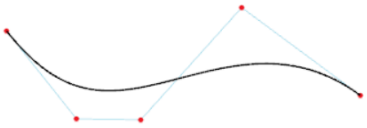

图 1 Bezier 曲线示例

3.2. 多条 Bezier 曲线的连接

开始下一条 Bezier 曲线的绘制时，将上一条 Bezier 曲线的终点（即最后一个控制顶点）作为第一个控制顶点；同时，根据上一条曲线的最后两个控制顶点绘制一条辅助线，当下一条曲线的第二个控制顶点在该辅助线上时，根据 Bezier 曲线的性质，此时前后两条曲线的切线斜率在连接点处相同，达到 G1 连续，由此实现平滑连接的效果。管理 Bezier 曲线的绘制的代码位于 bezier.h/bezier.cpp 中。

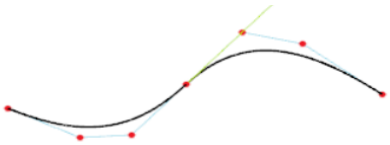

图 2 两条 Bezier 曲线的连接方式示意图

3.3. 鼠标交互确定控制顶点位置

当屏幕宽高比例改变时，若不进行一定的调整，将对交互造成困难。为此，动态记录屏幕的宽尺寸，结合屏幕坐标与 OpenGL 空间笛卡尔坐标的关系，得到控制点坐标的计算公式如下：

$$\begin{cases} P.x = \left(2 * \dfrac{S.x}{w} - 1\right) * \max\left(\dfrac{w}{h}, 1\right); \\ P.y = \left(1 - 2 * \dfrac{S.y}{h}\right) * \max\left(\dfrac{h}{w}, 1\right); \end{cases}$$

P为控制顶点在笛卡尔坐标系中的位置，S为鼠标在屏幕上点击的位置。与此同时，shader 中计算顶点坐标的公式为：

$$\begin{cases} V.x = P.x * \min\left(\dfrac{h}{w}, 1\right); \\ V.y = P.y * \min\left(\dfrac{w}{h}, 1\right); \end{cases}$$

V 为顶点在 shader 中的显示坐标。

由此可以在任意屏幕尺寸下绘制控制顶点，整体图像的尺寸由屏幕宽与高中较短的一方决定，且当屏幕宽高比例发生改变时，模型的显示比例保持不变。

4. 旋转扫描曲面生成算法

4.1. 扫描线：从 Bezier 曲线取样

绘制 Bezier 曲线时，为了能够在屏幕上呈现平滑的曲线效果，所设置的取样率较高，而对于三维模型的显示，无需在各处都使用如此高的取样率。因此，在生成扫描线时，并不采用 Bezier 曲线上所有的样本点，而是根据一定的二次取样率从样本点中间隔取点，生成模型之后再交由 shader 根据视点距离细分。下图（左）为模型的取样效果。

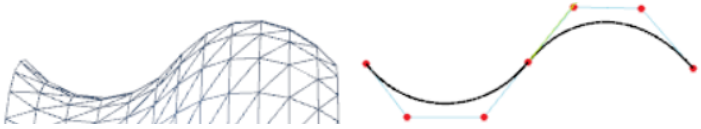

图 3 曲面与曲线的取样率对比

4.2. 扫描面：选择合适的角度间隔

扫描线的旋转角度间隔按照以下方法计算：根据二次取样率计算出矩形面片的预期边长，采用 5 点取样法计算描线上各点的平均半径（即该点到旋转轴 y 轴的距离），前者除以后者得到扫描线的角度间隔。然后将扫描线依次旋转，计算所有样本点的位置、法矢量以及矩形面片的顶点索引、二维纹理坐标，并绑定 OpenGL 缓存。以上内容均在 CPU 中完成。

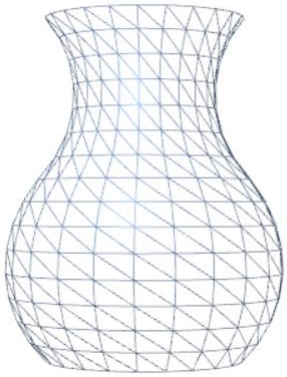

图 4 未加曲面细分的模型示意图

法矢量按照标准的计算方式进行，即对该顶点所在的所有面片的法矢量求平均值。由于旋转扫描面每一个面片的索引顺序已知，对法矢量的计算有一定的简化，无需遍历所有面片。

5. 曲面细分算法

参考 GitHub 上的样例代码在 shader 中完成细分，并根据顶点法矢量对面片细分时的坐标计算进行优化，代码位于 model.tesc 与 model.tese 中。

5.1. 根据视点距离确定细分参数

OpenGL 曲面细分基于 patch 完成，每一个 patch 的顶点数目可以自定义，因此有三角细分和矩形细分等多种模式。根据实际情况，此处采用矩形细分模式。在计算每条边的细分粒度时，跟据该边两端点到视点的平均距离，大小设置不同的粒度。在本程序中，细分粒度与视点距离的关系为：

表 1 视点距离与细分粒度的关系

视点距离	0.0 ~ 3.0	3.0 ~ 6.0	6.0 ~ ∞
细分粒度	3	2	1

下图为各细分级别的示意图：

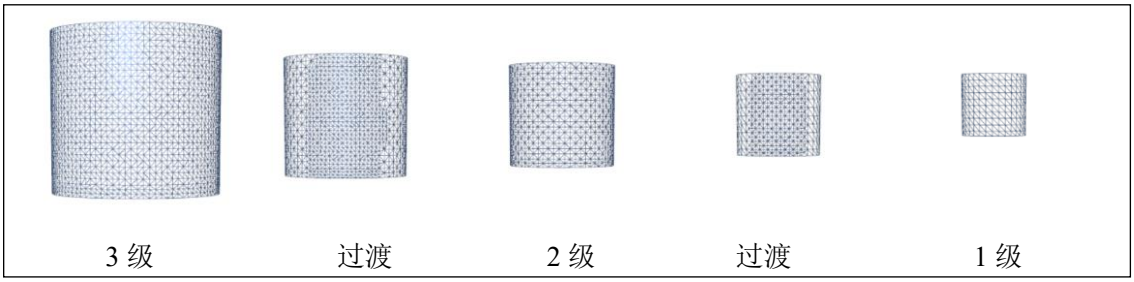

图 5 细分级别示意图

5.2. 计算细分顶点法矢量与纹理坐标

细分顶点法矢量与纹理坐标按照普通插值算法计算即可。

5.3. 计算细分顶点坐标

由于曲面的空间属性，仅按照普通插值算法做平面内的细分显然不够充分，无法呈现更

加精细的曲面效果，达不到细分的目的，需要结合顶点的法矢量在空间中进行计算。在本次作业中，利用矩形细分以及旋转扫描带来的便利性，采取双二次 Bezier 曲面插值计算细分顶点坐标，可以获得良好的细分效果。

5.3.1. 基本方法

双二次 Bezier 曲面需要 9 个控制顶点，根据面片中已知的 4 个顶点的坐标以及法矢量可以计算出剩余 5 个控制顶点。由于旋转曲面扫描特殊的对称性，这部分的计算得到了很大程度的简化。

9 个控制顶点的位置如下图所示，除 4 个面片顶点（编号 1,3,7,9）外，边框顶点（编号 2,4,6,8）与相邻两个面片顶点的连线分别与其法矢量垂直，中心顶点（编号 5）与 4 个面片顶点的连线与其法矢量均垂直。值得注意的是，这些假设并不适用于一般曲面，旋转曲面特殊的对称性确保了以上条件可以满足。

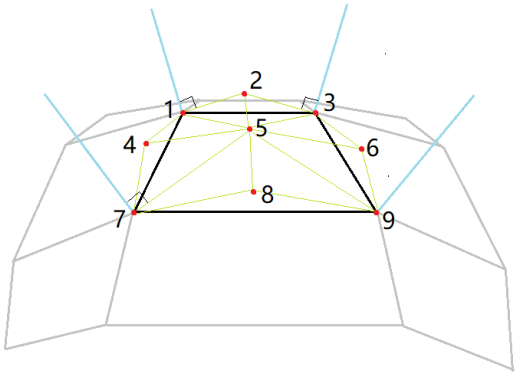

图 6 双二次 Bezier 曲面控制顶点位置示意图

5.3.2. 中心控制顶点坐标的计算

根据对称性，考虑中心顶点P_5与 3 个面片顶点（$P_1P_3P_7$）连线与法矢量的垂直关系即可，得到以下方程组：

$$\begin{cases} N_1 \cdot (P_5 - P_1) = \mathbf{0} \\ N_3 \cdot (P_5 - P_3) = \mathbf{0} \\ N_7 \cdot (P_5 - P_7) = \mathbf{0} \end{cases}$$

其中P_i与N_i分别为顶点（编号 i）的坐标与法矢量，上式可以用矩阵形式表现为：

$$\begin{bmatrix} N_1^T \\ N_3^T \\ N_7^T \end{bmatrix} \cdot P_5 = \begin{bmatrix} N_1^T \cdot P_1 \\ N_3^T \cdot P_3 \\ N_7^T \cdot P_7 \end{bmatrix}$$

那么可以得到中心控制顶点P_5的坐标为：

$$P_5 = \begin{bmatrix} N_1^T \\ N_3^T \\ N_7^T \end{bmatrix}^{-1} \cdot \begin{bmatrix} N_1^T \cdot P_1 \\ N_3^T \cdot P_3 \\ N_7^T \cdot P_7 \end{bmatrix}$$

5.3.3. 边框控制顶点坐标的计算

根据对称性，相邻两个面片顶点的法矢量位于同一平面 M 内，以控制顶点P_2为例，考虑P_2与相邻面片顶点（P_1P_3）连线与法矢量的垂直关系，以及P_2也在平面 M 内，可以得到如下方程组：

$$\begin{cases} N_1 \cdot (P_2 - P_1) = \mathbf{0} \\ N_3 \cdot (P_2 - P_3) = \mathbf{0} \\ (P_2 \cdot (P_1 - P_3) + P_1 \times P_3) \cdot \dfrac{N_1 + N_3}{2} = \mathbf{0} \end{cases}$$

同样可以得到控制顶点P_2的坐标为：

$$P_2 = \begin{bmatrix} N_1^T \\ N_3^T \\ (P_1 - P_3) \times \dfrac{N_1 + N_3}{2} \end{bmatrix}^{-1} \cdot \begin{bmatrix} N_1^T \cdot P_1 \\ N_3^T \cdot P_3 \\ P_3 \times P_1 \cdot \dfrac{N_1 + N_3}{2} \end{bmatrix}$$

在确定 9 个控制顶点之后，即可根据双二次 Bezier 曲面的计算方法求出细分顶点的坐标。此处采用分割递推 Casteljau 算法，分别根据两个方向的参数 u，v 进行插值，得到细分顶点的坐标。

5.3.4. 特殊情况处理

对于 5.2.2.中计算中心控制顶点，若涉及到的 3 个法矢量中有两个法矢量相等（只可能有$N_1 = N_7$的情况），则先求出边框P_1，P_7的控制顶点，再平移至中心即可；对于 5.2.3.中计算边框顶点，若涉及到的两个法矢量相等（$N_1 = N_3$），则直接返回线段P_1，P_3的中点即可。以上两种特殊情况均因为矩阵非满秩而无法计算。

另外，若面片位于曲率方向发生改变的地方，则可能会出现法矢量在面片平面内的投影方向不一致（不是均向内或均向外）的情况，此时根据以上方法计算出的结果将会严重失真，因此直接退化为普通插值算法。

5.3.5. 连续性问题

由于计算边框控制顶点仅用到两个面片顶点的数据，因此对于相邻两个矩形面片，公共边上的控制顶点一定重合，达到 G0 连续；由于多重相互的垂直关系以及旋转曲面的对称性，可以证明下图中三段深绿色实线部分上的 3 个顶点均分别在同一直线上，达到 G1 连续；水平方向上的相邻面片之间，由于宽度一致，同时也达到了 C1 连续。

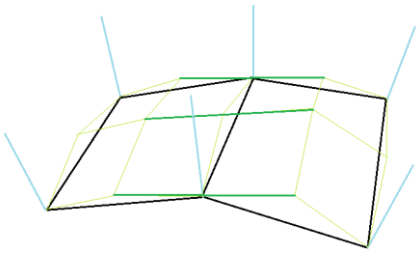

图 7 相邻 Bezier 曲面连续性示意图

下图为两种细分方式的效果对比图。可以观察到双二次 Bezier 曲面细分算法得到的曲面平滑性更好，具有更加优良的视觉效果。

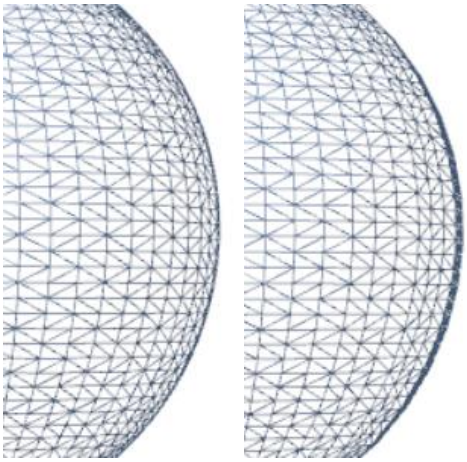

图 8 双二次 Bezier 曲面细化（左）与普通插值细化（右）

6. 模型储存与导入

这一部分的实现较为简单，2.3.小节中详细介绍了操作方法，此处不再赘述。在 obj 文件夹中预存了导出的 5 个模型，可以使用第三方 3D 模型编辑器打开，如下图所示：

图 9 使用第三方编辑器打开导出的模型

不同之处在于第三方 3D 模型未实现曲面细分，模型的曲面轮廓相对比较粗糙。

7. 模型显示

7.1. 光照明模型

光照明模型与作业 1 中相同，采用 Phong 模型实现，包含一个位于指定位置的点光源，此处不再赘述。代码位于 model.frag 中。下图为增加了光照明模型的效果图：

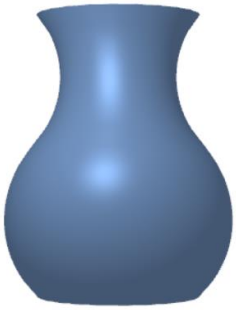

图 10 光照明效果示意图

7.2. 纹理贴图

顶点的纹理坐标在生成扫描面时计算，按照展开图的形式旋转映射至模型表面。为支持纹理的拼接，需要将扫描面的始边与终边重合，并对始边和终边顶点的纹理坐标分开计算，分别对应于纹理图的始边与终边。如下图所示，当纹理图本身可衔接时，模型表面不会出现贴图割裂的情况。

图 11 纹理拼接示意图

7.3. 坐标变换

根据作业要求，本程序不使用任何第三方数学库，关于坐标变换的计算均通过自定义方法实现，代码位于 view.h/view.cpp 中。

7.3.1. 透视投影

透视投影矩阵的推导较复杂，这里只给出结论：在投影中心位于原点且观察平面在近裁剪平面时，透视投影矩阵的形式为：

$$M_{pers,norm} = \begin{bmatrix} \frac{1}{aspect \cdot \tan\left(\frac{fovy}{2}\right)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\left(\frac{fovy}{2}\right)} & 0 & 0 \\ 0 & 0 & \frac{zNear + zFar}{zNear - zFar} & \frac{2 \cdot zNear \cdot zFar}{zNear - zFar} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

其中各项参数的含义如下：

fovy：摄像机视角，如：45°，90°；

aspect：裁剪平面的宽高比，当视口为整个屏幕时可以设置为屏幕宽高比；

zNear：摄像机与近裁剪平面的距离；

zFar：摄像机与远裁剪平面的距离。

本程序中采用 30° 视角，在几何着色器（model.geom）中乘以以上矩阵，得到每一个顶点的观察坐标，可以得到“近大远小”的立体视觉效果，随着与视点距离的增大，逐渐接近平行投影，如下图所示：

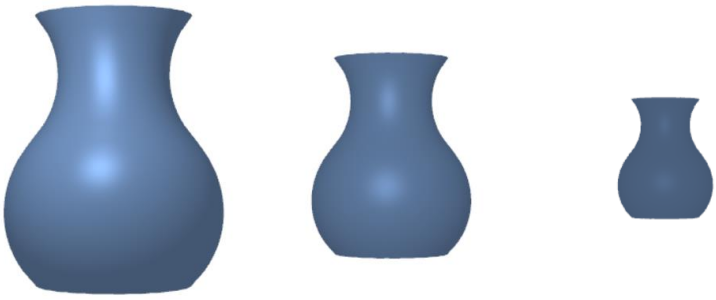

图 12 透视投影示意图

7.3.2. 混合变换

混合变换包括缩放、旋转、平移，参考上课内容以及网上资料，实现变换矩阵计算算法。

缩放矩阵：

$$M_{scale} = \begin{bmatrix} s_1 & 0 & 0 & 0 \\ 0 & s_2 & 0 & 0 \\ 0 & 0 & s_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

旋转矩阵（为简化计算，分别绕坐标轴旋转）：

绕 x 轴旋转:

$$M_{rot_x} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\,\alpha & -sin\,\alpha & 0 \\ 0 & sin\,\alpha & cos\,\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

绕 y 轴旋转：

$$M_{rot\_y} = \begin{bmatrix} cos\,\alpha & 0 & -sin\,\alpha & 0 \\ 0 & 1 & 0 & 0 \\ sin\,\alpha & 0 & cos\,\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

绕 z 轴旋转：

$$M_{rot\_z} = \begin{bmatrix} cos\,\alpha & -sin\,\alpha & 0 & 0 \\ sin\,\alpha & cos\,\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

合并旋转矩阵：

$$M_{rot} = M_{rot\_x} \cdot M_{rot\_y} \cdot M_{rot\_z}$$

平移矩阵：

$$M_{trans} = \begin{bmatrix} 1 & 0 & 0 & x' \\ 0 & 1 & 0 & y' \\ 0 & 0 & 1 & z' \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

混合变换矩阵：

$$M = M_{trans} \cdot M_{rot} \cdot M_{scale}$$

需要注意的是，缩放、平移不改变顶点的法矢量，因此在顶点着色器（model.frag）中，对顶点坐标乘以混合变换矩阵M，而对顶点法矢量仅乘以旋转矩阵M_{rot}即可。混合变换不改变纹理坐标。

7.3.3. 鼠标交互的模型视角与位置改变

基于以上坐标变换，参考 Windows 3D 查看器的视角变换模式实现鼠标交互，2.3.小节中详细介绍了操作方法。为保持模型姿态的稳定性，不支持绕 z 轴旋转。在多边形显示为线形模式的情况下，通过鼠标中键调整模型与视点的距离可以查看面片的多级细分效果。

8. 总结

本次作业较为复杂，需要对 GLSL 管线流程、细分算法及其代码实现、齐次坐标系和坐标变换矩阵的关系等基本知识有清晰的认识，否则寸步难行。算法实现也存在许多不足，如细分算法不具有普适性，在两条曲线的连接处可能出现面片细分不均匀的情况；仍然局限于基础光照模型，且对不同纹理表现效果不同等等。由于时间和精力的限制，以上问题暂时得不到较好的解决，后续可以进一步探究。