计算机图形学考试大作业

周萌 518030910125

程序使用说明

程序运行初始处于车削模式下

使用鼠标交互控制进行切削

当按下鼠标之后，刀具会出现，拖动鼠标可模拟车床车削工作

松开鼠标时不能进行车削

按下R，重置成原材料

按下E，切换原料材质

在车削模式下按下W，进入贝塞尔曲线编辑模式，此模式之下，只能进行贝塞尔曲线编辑。使用鼠标点出控制点，可看到最后的贝塞尔曲线，当有了一条贝塞尔曲线之后，在点击鼠标不会有任何响应

在贝塞尔曲线编辑模式下按下W，回到车削模式，此时，假如有贝塞尔曲线存在，则会约束切削，进刀不能超过曲线

当进刀过深会自动断刀

刀具不可在物料的两端切割

总体介绍

本次作业，我用的是固定管线，将它拆分成以下几个部分来解决：

- 核心部分
  - 车削体的绘制
  - 用鼠标来进行交互切削
- 其他部分
  - 材质、光照与纹理
  - 贝塞尔曲线控制
  - 飞溅的碎屑——粒子系统
  - 视角与背景

- 其他细节

接下来我将依次介绍各个部分

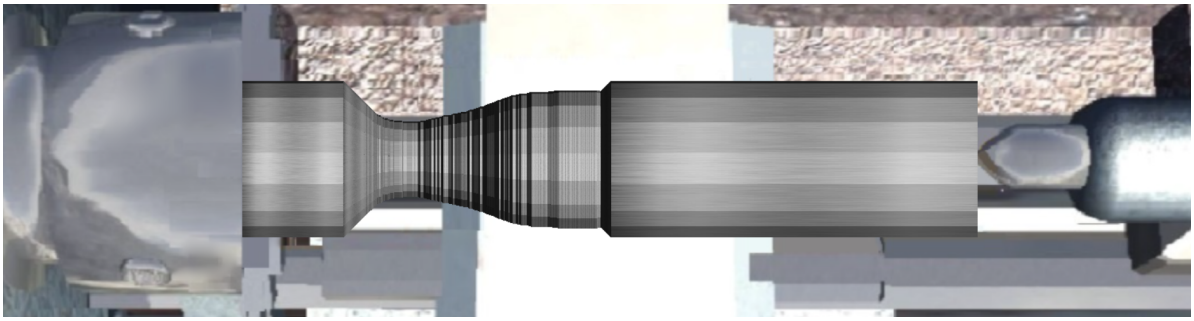
核心部分

一、车削体的绘制

在工业加工中，物体应该都是实体。但是在计算机绘制时，我们都只需要绘制其表面，这样一来，车削时物体的表面就会不停的变化，发生塌陷，那么如何表达这个车削体的表面就是个十分关键的问题。

我注意到，车削体本质上是一个旋转体，于是我考虑使用一条轮廓线，将其进行360度旋转，从而绘制整个面。

我使用了两种坐标系，柱坐标系和平面直角坐标系，使用柱坐标系搭出模型网格，从以下这张图可以比较容易看出原理。



外轮廓线

确认了采用旋转体的方法来绘制切削体之后，下一个困难就是基础外轮廓线的表达，事实上，整个外轮廓线十分的复杂，而所谓的曲线其实也是一段一段直线的拟合，所以我使用很多段线段来表示基础外轮廓线，把整个外轮廓线的点存在向量中：

```
vector<point> points;
```

其中point是我自己定义的平面点，储存x和y坐标：

```
struct point {
    float x;
    float y;
    point() :x(), y() {}
    point(float x, float y) :x(x), y(y) {}
};
```

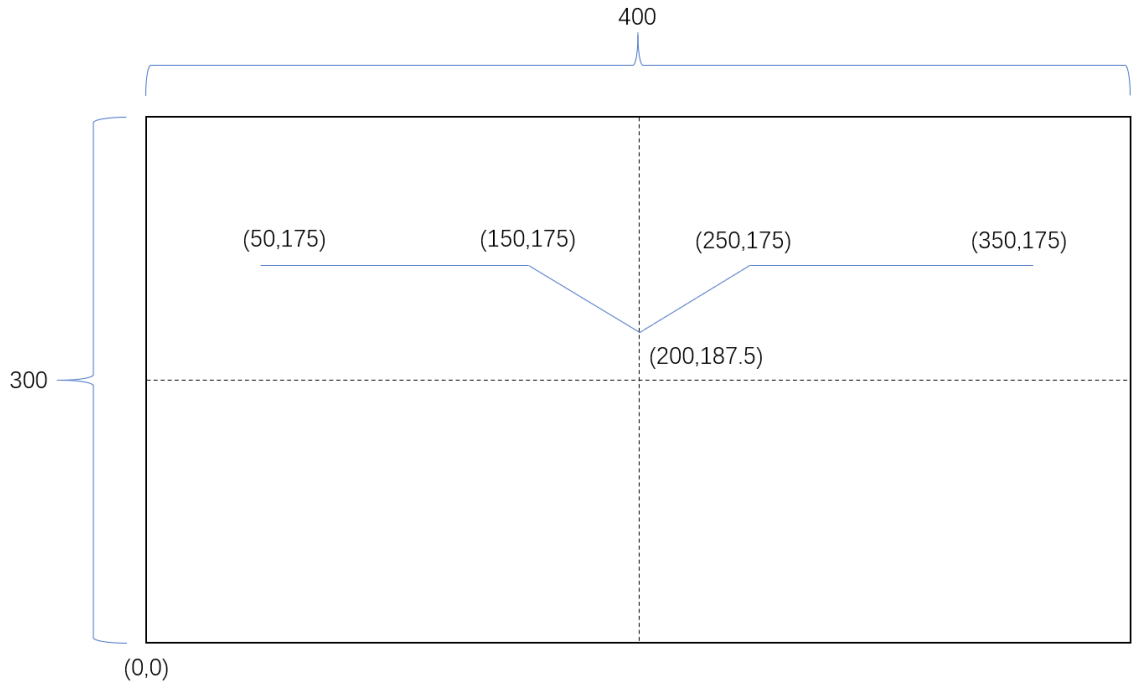
旋转体产生与绘制

我建立了一个新的类型为pointRect，来记录最终用来绘图的空间点位置：

```
struct pointRect {
    float x;
    float y;
    float z;
    pointRect() :x(), y(), z() {}
    pointRect(float x, float y, float z) :x(x), y(y), z(z) {}
};
```

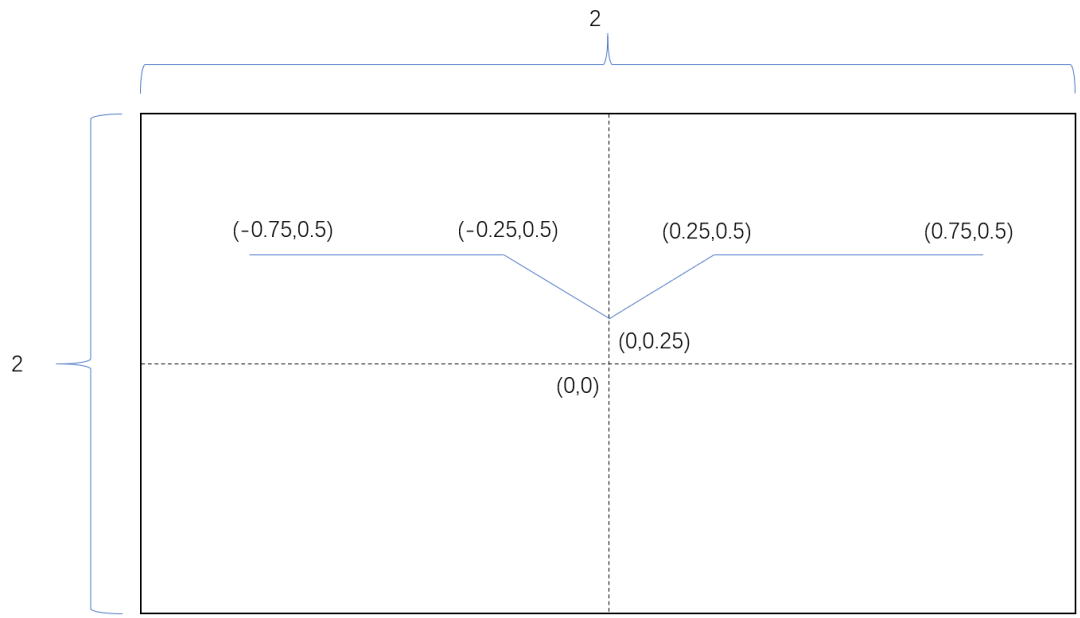
接下来我举个例子来演示旋转体的产生：

假设我们现在的基础轮廓线如下：



- 第一步：改变坐标映射

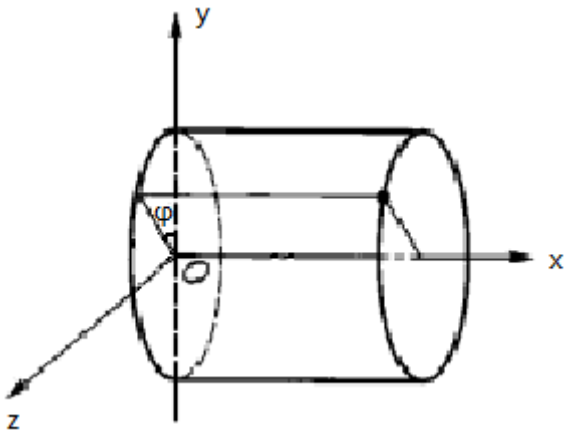
  调用changeScale函数，把坐标映射到（-1，1）之间。



- 第二步：把这条基础线进行旋转，计算出所有的外轮廓线

  利用柱坐标转直角坐标的公式：

$$
\begin{aligned}
x &= x \\
y &= ycos\varphi \\
z &= ysin\varphi
\end{aligned}
$$



  其中等式左边是直角坐标系下的最终坐标，右边是基础轮廓线上的点的坐标。

  让φ自己逐渐递增，就能形成一个旋转体，调整φ递增的步长，就能控制体表面的平整度。

  这一步是在generateMatrix()函数下产生的，最后的结果是下面这个二维数组：

```
vector<vector<pointRect>>
```

  接下来就是绘制了，我们相当于把整个切削体表面拆分成了一个又一个的方格，只需要用双重循环，一个一个方格去绘制即可。

二、用鼠标来进行交互切削

鼠标位置的转换

值得注意的是，当我们调用鼠标函数，获取了x，y时，并不能直接使用，它代表以下：

我们需要将其转换成如下：（对y做处理）

得到真正的鼠标位置。

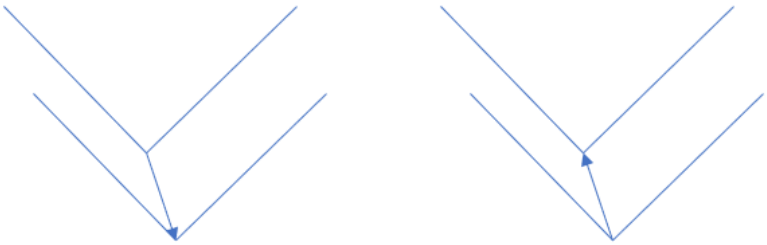
新的基础轮廓线生成

我使用的是45度角的刀头。每次切削的时候都只需要变化基础轮廓线，通过上面所叙述的旋转体计算方法就能够生成出整个切削体。

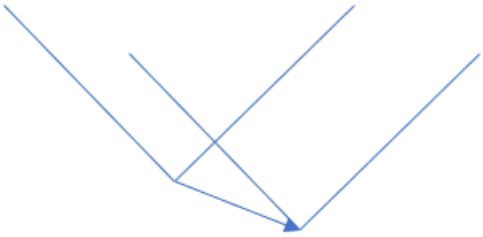
我依靠前一时刻鼠标位置和下一时刻鼠标位置来更新基础轮廓线。

总体分为以下两种情况：

- 当前一时刻x和下一时刻x的差绝对值不大于y的差的绝对值时，形成的移动覆盖面是一个尖头



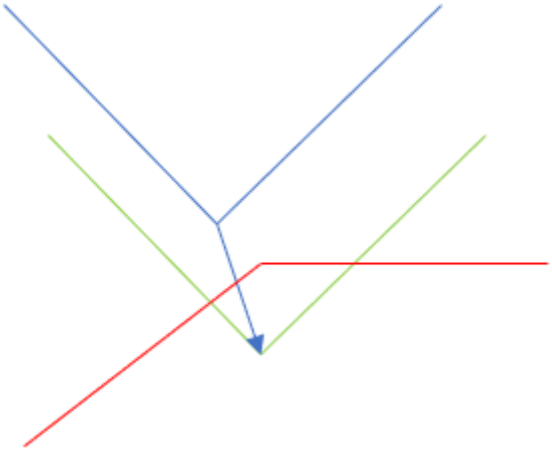
- 否则形成的是平头



然后对于这两种情况，我们分别去更新基础轮廓线。

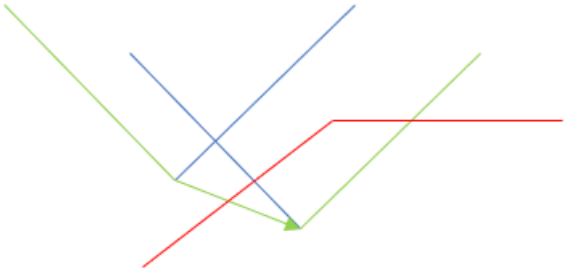
我直接举一个例子来讲解更新的过程：

对尖头来说：



首先应该计算整个刀片滑过的面和原来的基础轮廓线的交点，即绿色边界与红色的基础轮廓线的交点，并将其记录下来，算到新轮廓线里。之后，找到基础轮廓线里在绿色边界以上的点，将其塌陷掉（删除）。最后添加新刀尖那个点，就能形成新轮廓线。

对平头来说：



和尖头十分类似。首先应该计算整个刀片滑过的面和原来的基础轮廓线的交点，即绿色边界与红色的基础轮廓线的交点，并将其记录下来，算到新轮廓线里。之后，找到基础轮廓线里在绿色边界以上的点，将其塌陷掉（删除）。最后添加新刀尖那个点，就能形成新轮廓线。

它们的计算方式基本是一样的，我之所以将其分成两类，是因为它们在计算那个绿色轮廓线的时候十分的不同。

在每次按着鼠标左键拖动刀片切削的时候都会做这样的计算，因此就能实现切削的效果。

其他部分

材质，光照与纹理

材质

材质我是在画切削体的时候进行设置的。有一个全局变量name来记录使用的是哪种材质(metal or wood)。举个例子，设置木头材质：

```
    if (name == "wood") {
        //木纹
        GLfloat amb[] = { 0.292250, 0.292250, 0.292250, 1.000000 };
        GLfloat dif[] = { 0.9,0.9,0.9, 1 };
        GLfloat spec[] = { 0.01,0.01, 0.01,1 };

        // 环境光
        glMaterialfv(GL_FRONT, GL_AMBIENT, amb);
        // 漫反射光
        glMaterialfv(GL_FRONT, GL_DIFFUSE, dif);
        // 镜面反射光
        glMaterialfv(GL_FRONT, GL_SPECULAR, spec);
    }
```

光照

我使用的是一个平行白光，在init()函数里设置。

然后在绘制切削体的时候计算出每个面的法向量并进行设置，确保光照正确。

纹理

我用了网络上的一个texture.h类用来读取bmp图片，构造纹理，同样使用name来控制使用哪种纹理，在绘制切削体的时候每个小面片都贴上纹理。

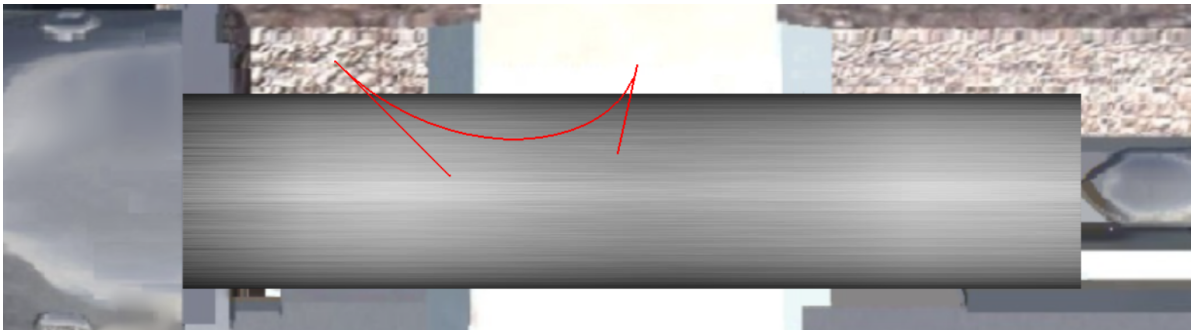
纹理和材质的切换

我在keyboard函数里指定，按下E/e键切换掉name的值，并重新调用ini()函数，绑定纹理。

贝塞尔曲线控制切削

贝塞尔曲线绘制

使用三次贝塞尔曲线，显示在平面上，控制切削，为了让用户看得更明显，我使用了红色。



具体过程如下：

- 按W进入贝塞尔曲线编辑模式，此时除了交互编辑贝塞尔曲线之外，所有交互都失效。
- 用鼠标在屏幕上点出四个点，分别在1和2，3和4之间连上直线，强调贝塞尔曲线端点的切线。
- 然后当屏幕上有4个点之后，绘制出贝塞尔曲线
  - 使用公式

$$B(t) = P_0(1-t)^3 + 3P_1t(1-t)^2 + 3P_2t^2(1-t) + P_3t^3, t \in [0,1]$$

    将t从0到1以0.05的步长递增，计算出曲线上的每个点的坐标，依次把线段连起来，就形成了贝塞尔曲线。
- 在这之后，无论鼠标怎么点都会失效（因为只允许一条贝塞尔曲线存在）
- 按Q键将清空所画的贝塞尔曲线，之后能重新绘制。
- 再次按W键将退出贝塞尔曲线绘制模式，进入切削模式，此时将只绘制曲线，控制点和切线都将消失。

贝塞尔曲线约束切削

在切削模式之下，如果有贝塞尔曲线存在，则会进行约束，用到了这个函数：

```
bool isUpBazier(point p, point p0, point p1, point p2,point p3)
```

它是用来判断p点是否在p0，p1，p2，p3做控制点形成的贝塞尔曲线的上或上方的。

知道了控制点就能知道贝塞尔曲线的，参数方程，我将p的xp坐标带入，求到了对应t（这里需要解一元三次方程的实根，我用了盛金公式求解，之后过滤掉0到1以外的根），然后代入求到了x = xp直线和贝塞尔曲线的交点(xp,y)。将y和yp比较，就能判断出p点是否在贝塞尔曲线上或上方了。

有了以上这个工具函数之后，我们只需要在切削的时候，记录刀尖前一刻位置和刀尖当前时刻位置，当当前时刻位置在贝塞尔曲线之下了，那么我们就不会更新基础轮廓线，而是会直接调用以下系统接口：

```
POINT cp;
GetCursorPos(&cp);
SetCursorPos(cp.x - (now.x - pre.x) ,cp.y + (now.y - pre.y));
```

强行把鼠标放到前一时刻位置上。

可能看起来这段代码有点奇怪，我解释一下：

GetCursorPos和SetCursorPos都是设置的全局鼠标位置，即相对于电脑屏幕而言，而我们能够获取的鼠标位置是在窗口里的。因此我用差值法，把鼠标归到前一时刻的位置上。即拿到当前时刻的全局鼠标位置，减去当前时刻的窗口鼠标位置和前一时刻的窗口鼠标位置的差值，即是前一时刻的全局鼠标位置。

为什么这里x和y不同？这是因为，这里的y是用的这个坐标系：

而全局的y是用的这个坐标系：

所以我们得给差值加个负号。

此外，在切削模式下，也可以按下Q/q来清空贝塞尔曲线，这样切削就不受到约束了。

飞溅碎屑——粒子系统

我这里用了一个非常非常简单的粒子系统来模拟碎屑飞溅。

具体来说，我写了一个Partcle类，来管理粒子系统。

每个粒子有以下属性：

```
// 粒子结构
struct Particle
{
    float x, y, z;             //粒子的位置
    unsigned int  r, g, b;  // 粒子的颜色
    float vx, vy, vz;         // 粒子的速度(x,y,z方向)
    float ax, ay, az;         // 粒子在x，y，z上的加速度
    float lifetime;          // 粒子生命值
    float size;              //粒子尺寸
    float dec;               //粒子消失的速度
};
```

在整个程序下有一些全局变量来管理粒子系统：

```
// 创建一个粒子系统
CParticle Snow;

//用来设置粒子的属性值
float x, y, z, vx, vy, vz, ax, ay, az, _size, lifetime, _dec;
int r, g, b;
float cx = 0, cy = 0, cz = 0;  //记录粒子发出的源点（刀尖）
```

每当开始切削时，整个粒子系统将被重新初始化到刀尖位置，并不断更新状态，我这里用了一种比较简单的方式：

粒子数目恒定（500），但寿命都很短，很快会消失，在一个切削过程中，假设某切削时刻所有粒子处于某一状态，在很短的时间里切削到下个位置时，所有的粒子位置并不会根据刀尖位置更新，因为如果直接更新其位置，粒子系统就会瞬移，显得很突兀，所有的粒子依旧会按照设定的速度和加速度运动，直到生命周期结束或者飞出屏幕，才会将生命结束的粒子重置到刀尖位置作为出生点，并重设初始速度和加速度。

由于粒子的生命周期我置得比较短，且切削的移动速度不快，移动量小，这样一来，整个粒子系统看起来就比较连贯不突兀。

粒子数目恒定的好处在于不用特别频繁地大量生成粒子和销毁粒子，能够提高交互的效率和流畅性。

另外，在按下E切换材质的时候，粒子系统也会被重新初始化。且根据name选择粒子系统的颜色。

视角与背景

由于车床加工有个实时交互精准度问题，另外还考虑到鼠标位置的获取和贝塞尔曲线的显示，我使用了正射投影，相当于把3D问题转换成了2D问题，只是在绘制车削体的时候做了3D变换。

然后为了让整个场景真实度更高，我加了一个车床的背景图（在最底层画了一个带纹理的矩形），并进行了调整和对位置。

其他细节

- 刀具只在按下鼠标，且不在车削体内部时才会显示，仅仅在从车削体外部进到内部时才进行切削。松开之后既不会显示刀具，也不会切削。这样既方便了操作，也增加了仿真性。
- 刀具不能从两端进行车削
- 当进刀到比较深的位置时，刀具会消失，切削中止，这是用来模拟断刀。
- 刀具的贴图是我在淘宝上找的一个真实45度车刀来贴的。
- 另外，由于刀具本身有45度角，因此根据我的算法，车出的轮廓切线不会有过于垂直的（必须在45度角以内）。

  示意图如下：

  左图那样的轮廓线是有可能的，但右图那样的轮廓线是不可能出现的。
- 由于我的基础轮廓线是一段一段的线段，因此切到后面，会出现很多很多细小的线段，这样一来轮廓线就会不太平滑，但真实切削原本就是如此，没有数控装置，手工动刀切出的表面本身就会不太平滑。