浙江大学

本科实验报告

课程名称：	基于 GPU 的绘制
实验名称：	Subdivision
姓　　名：	******
学　　院：	计算机学院
系：	数字媒体与网络技术
专　　业：	数字媒体技术
学　　号：	315010****
指导教师：	唐敏

2017 年 12 月 21 日

浙江大学实验报告

实验名称：Subdivision 实验类型：操作实验

同组学生：无 实验地点：外经贸楼

一、 实验目的与内容

a) 实验目的

在实验过程中熟练细分着色器和几何着色器的使用。

b) 实验内容

使用 GLSL 语言编写着色器，将输入的正三角形图元细分并且能调整细分层次，使其逼近八分之一球面。

二、 实验环境与配置

a) 实验环境

i. 开发环境

Windows 7 SP1 64-bit + CLion 2017.3 + MinGW 5.0 + CMake 3.9.1

ii. 硬件

NVIDIA Quadro K620 (OpenGL 4.5)

iii. 第三方库

GLEW 7.0 + GLUT 3.7 + GLM 0.9.8.5

b) 实验配置

本实验项目使用 C++编写，在项目中依次包含 GLEW, GLUT 和 GLM 的头文件以及其他必要的系统头文件，使用 CMake 生成可执行文件。

三、 实验方法与步骤

a) 编写窗口、事件、键鼠控制等基础模块

复用曾经编写的工程模块。

b) 编写 Shader 类封装着色器相关操作

复用曾经编写的工程模块。

c) 编写着色器以实现功能

本次实验使用了五种着色器，其细节将在下一节中介绍。

四、 实验结果与分析

a) Overall：实验效果

本次实验的最终效果如 Figure 1 所示。

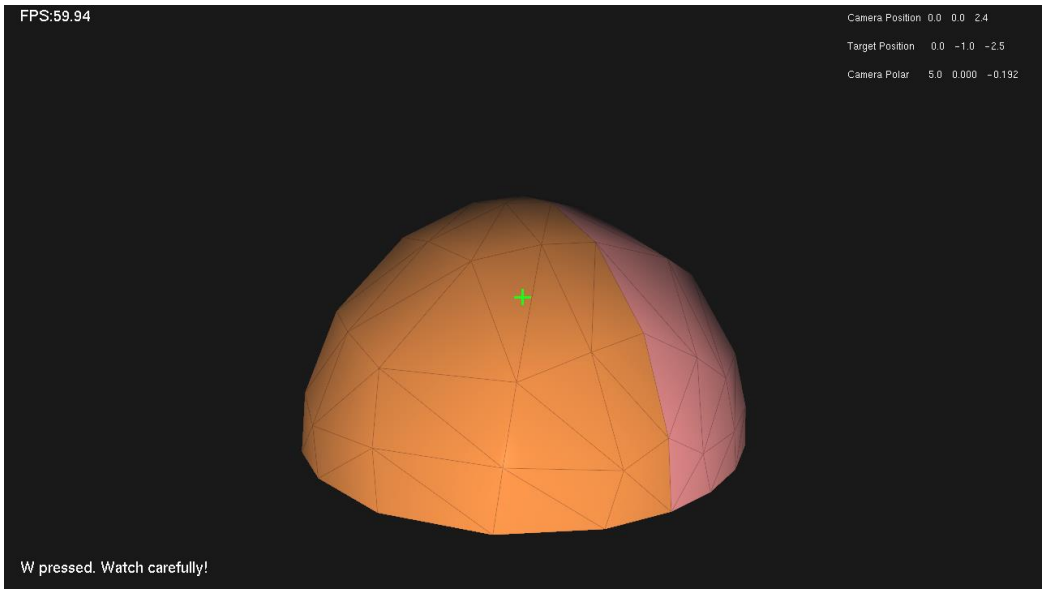

Figure 1 **实验效果**

b) OpenGL：输入正三角形模型

根据实验要求，输入的模型为一个正三角形。在实验中为了观察对比两种求值算法的差异，总共两个三角形被输入（顶点分别为(0, 1, 0), (0, 0, 1), (1, 0, 0)的正三角形和与它关于 xOy 平面对称的三角形，不包含两个用于观察未求值前的细分结果的三角形）。

```
float v0[] = {0.0f, 1.0f, 0.0f,
              0.0f, 0.0f, 1.0f,
              1.0f, 0.0f, 0.0f};
float v1[] = {0.0f, 1.0f, 0.0f,
              0.0f, 0.0f, 1.0f,
              -1.0f, 0.0f, 0.0f};
```

Figure 2 **输入顶点信息** 433:5@system.cpp

输入的三角形的顶点信息被存储在 Vertex Array 中以使用 Array Buffer 加速数据传递。

```
glGenVertexArrays(2, vaoHandle);
glGenBuffers(2, vboHandle);

glBindVertexArray(vaoHandle[0]);
glBindBuffer(GL_ARRAY_BUFFER, vboHandle[0]);
glBufferData(GL_ARRAY_BUFFER, 9 * sizeof(float), v0, GL_STATIC_DRAW);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(0);

glBindVertexArray(vaoHandle[1]);
glBindBuffer(GL_ARRAY_BUFFER, vboHandle[1]);
glBufferData(GL_ARRAY_BUFFER, 9 * sizeof(float), v1, GL_STATIC_DRAW);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(0);

glBindVertexArray(0);
```

Figure 3 **使用** Array Buffer 440:5@system.cpp

在绘制时，只需调出已经准备好的顶点数组，即可快速绘制。

```
glBindVertexArray(vaoHandle[0]);
glDrawArrays(GL_PATCHES, 0, 4);
```

Figure 4 **绘制三角形** 19:5@draw.cpp

c) Tessellation Control Shader：根据深度控制细分等级

TCS 的功能主要是控制自定义图元在各个方向上的细分等级，细分等级分为内部细分控制和边缘细分控制，分别通过对 gl_TessLevelInner []和 gl_TessLevelOuter[]传值来控制细分等级。具体的细节不在此展开，我们简单地用变量 tessLevel 来控制所有方向上的细分等级。

```
gl_TessLevelOuter[0] = tessLevel;
gl_TessLevelOuter[1] = tessLevel;
gl_TessLevelOuter[2] = tessLevel;

gl_TessLevelInner[0] = tessLevel;
gl_TessLevelInner[1] = tessLevel;
```

Figure 5 **控制细分等级** 19:5@basic.tcs

在 TCS 的编写过程中，我加入了根据模型距离摄像机的深度来控制三角形细分等级的功能。每个顶点的坐标在 TCS 处理过程中，除了被不加修改地直接传到下一段 pipeline 之外，会被转换到观察坐标系中，然后取得 z 值，这个 z 值即该点与摄像机的距离。将它与预先定义的最大距离、最小距离做比较，得到它处于最小距离 MinDepth 和最大距离 MaxDepth 之间的位置（利用 clamp()函数将范围限制在 0.0 至 1.0）。该值被用来与最大细分等级 MaxTessLevel 和最小细分等级 MinTessLevel 共同计算该位置对应的细分等级（利用 mix 函数），该结果即我们将要使用的 tessLevel。

```
vec4 p = ModelViewMatrix * gl_in[gl_InvocationID].gl_Position;
float depth = clamp((abs(p.z) - MinDepth) / (MaxDepth - MinDepth), 0.0, 1.0);
float tessLevel = mix(MaxTessLevel, MinTessLevel, depth);
```

Figure 6 **根据深度计算细分等级** 13:5@basic.tcs

在 OpenGL 侧允许调整 MaxDepth 的值（通过小键盘上的“+”和“-”），由于在摄像机靠近物体时（距离小于 MaxDepth 即可）始终使用最大细分等级来细分，我们调整 MaxDepth 的值即可调整显示的模型细分状态。

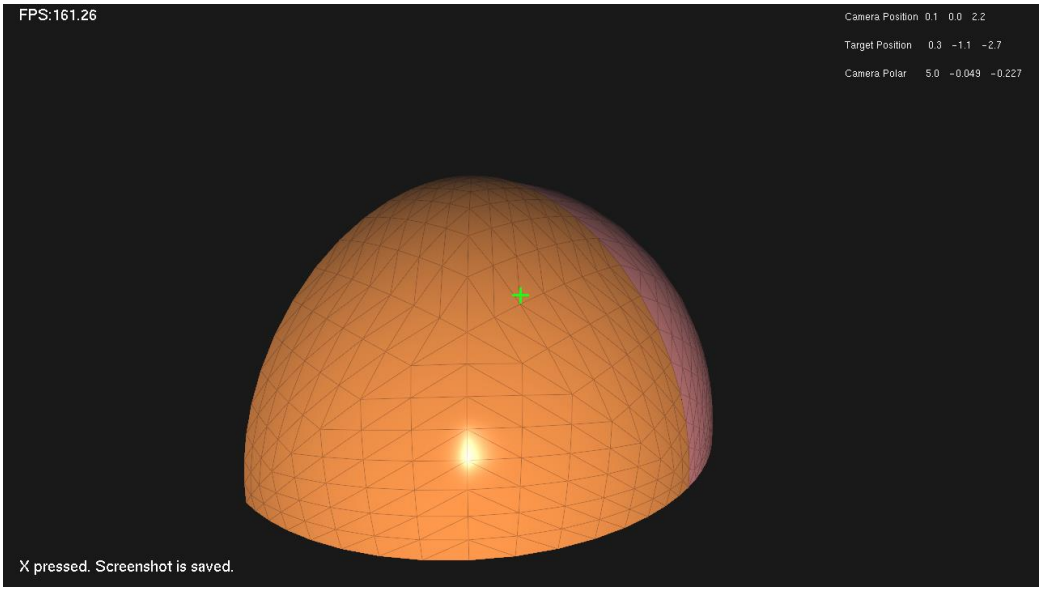

Figure 7 **增大最大细分等级**

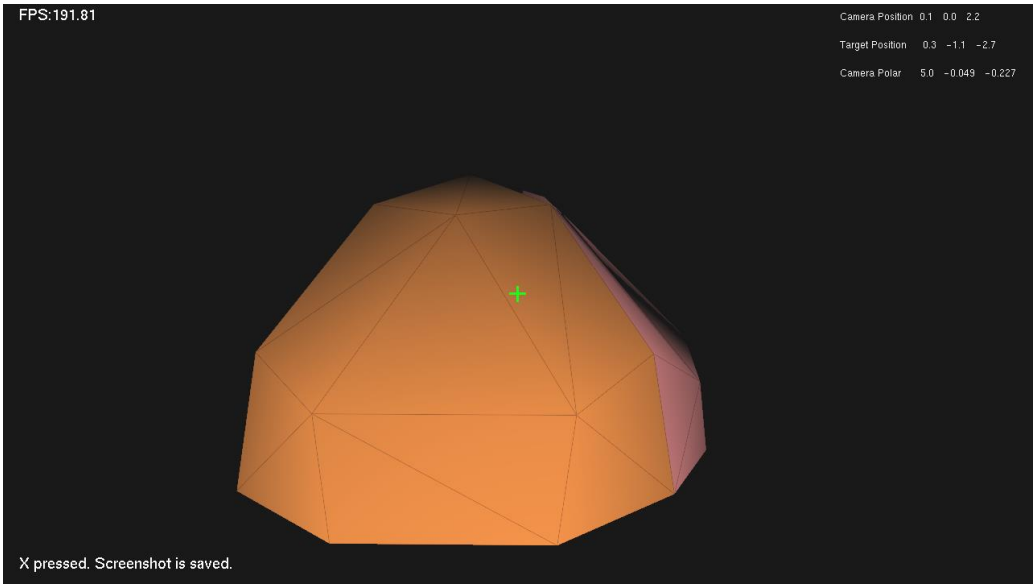

Figure 8 **减小最大细分等级**

TCS 还起到了定义图元的顶点数目的作用，它需要与 OpenGL 侧的语句共同作用，如 Figure 9 和 Figure 10 所示。两者中的较小者决定了每个 patch 的顶点数目。

```
layout(vertices = 3) out;
```

Figure 9 **在** TCS **中定义** patch **中的顶点数目** 3:1@basic.tcs

```
glPatchParameteri(GL_PATCH_VERTICES, 3);
```

Figure 10 **在** OpenGL **侧定义** patch **中的顶点数目** 464:5@system.cpp

d) Tessellation Evaluation Shader：细分产生顶点并计算位置

除了 TCS 中的控制参数，TES 也根据自己的细分条件来细分图元为自定义的 patch，并且将每个 patch 中的每个顶点传入 main 函数中去计算：

```
layout(triangles, equal_spacing, ccw) in;
```

Figure 11 **定义细分属性** 3:1@basic.tes

参数 triangles 表示以三角形化的图元来进行细分，在稍后对当前顶点求值时，可以访问到同一 patch 上的其他顶点；参数 equal_spacing 表示在细分过程中应该以等间距分割 u 和 v（u 和 v 是表示当前顶点在所在 patch 上的位置，一般从左到右为 u 从 0 到 1，从下到上为 v 从 0 到 1）；参数 ccw 表示 patch 的顶点以 counter-clock wise 即逆时针方向定义。

求值过程中，首先通过 gl_TessCoord 的 x 和 y 获取 u 和 v，然后通过 gl_in[]数组来获取 patch 中的三个顶点坐标，如 Figure 12 所示：

```
u = gl_TessCoord.x;
v = gl_TessCoord.y;

p0 = gl_in[0].gl_Position;
p1 = gl_in[1].gl_Position;
p2 = gl_in[2].gl_Position;
```

Figure 12 **获取** uv **和顶点坐标** 62:5@basic.tes

随后，我们利用这些值来计算插值顶点的坐标。在 basic.tes 中，我使用了三种计算方法，分别位于用 subroutine(EvaluationPassType)装饰的三个函数中：pass1(), pass2(), pass3()。

pass1()：根据三角形 patch 中的三个顶点和当前顶点的 u、v 坐标，可以将(p2 - p1)和(p0 - p1)作为基向量，通过以下计算式计算坐标：

$$p = p1 + u * (p2 - p1) + v * (p0 - p1)$$

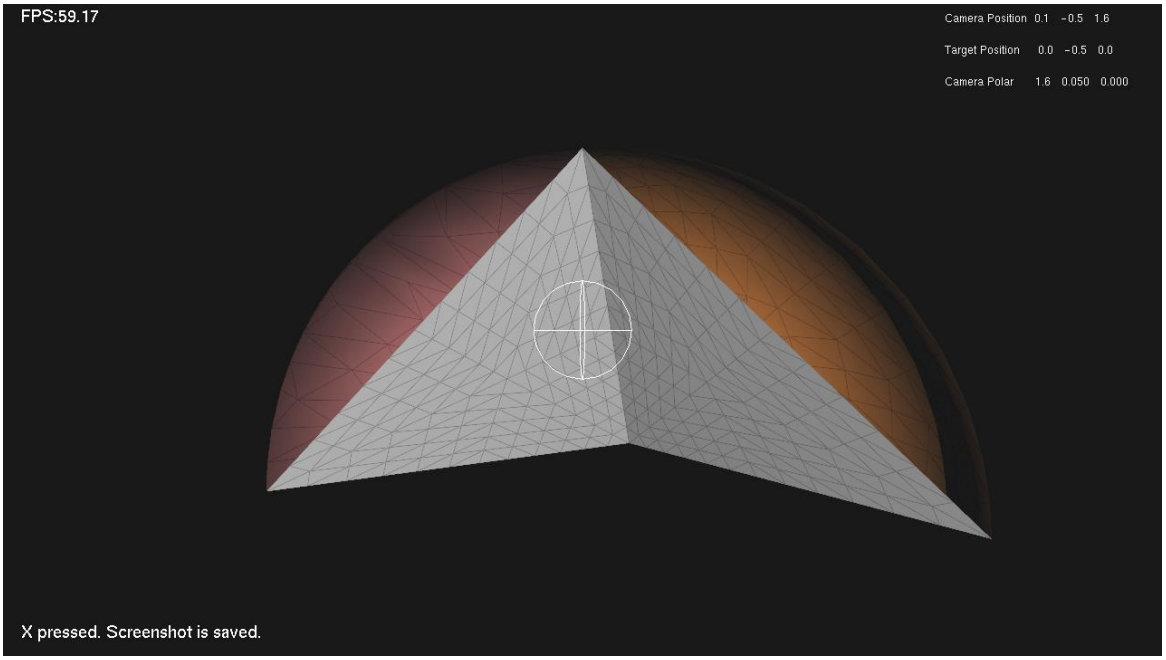

Figure 13 pass1()**计算结果**

pass2()：调用 pass1()，并在结果的基础上将这个点到球心的距离归一化。由于我们创建的正三角形三个顶点正好是以原点为球心，以 1 为半径的的球面上的一点，若对刚才的结果使用 normalize()函数，即可快速得到所求的球面上的点，那么我们就完成了从三角形中的点到球面点的插值。

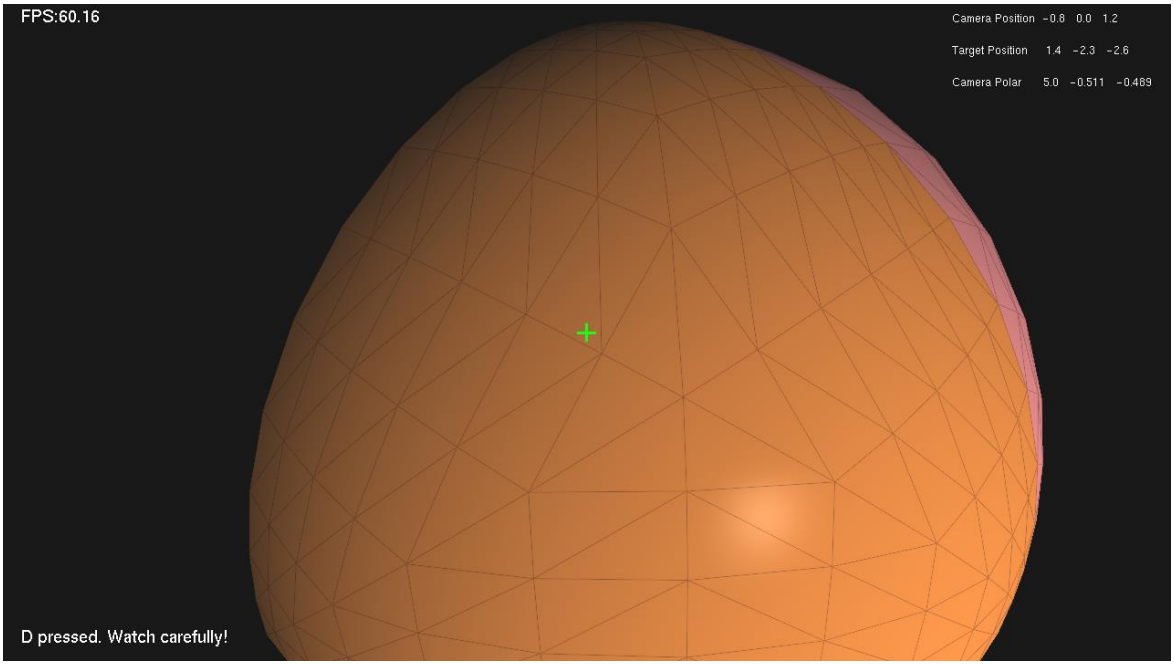

Figure 14 pass2()**计算结果**

进入场景后看到一个四分之一球面，左半边为橘色，右半边为粉色。橘色代表使用 pass2()计算得到的结果。从计算结果中可以看到该方法准确，均匀，几乎没有误差。

pass3()：pass3 是主要使用 u 和 v 来计算球面顶点的坐标的过程，关键在于将点的坐标换算为相对球心的两个旋转角 A 和 T。略去计算过程，在单位球的第一卦限内，当前顶点的参数

u、参数 v 与 xOz 平面上的旋转角 A、和垂直于 xOz 平面的旋转角 T 之间有如下关系：

$$\mathrm{T} = \arctan\left(\frac{v}{1-v}\right)$$

$$\mathrm{A} = \arcsin\left(\frac{u}{1-\sin(T)}\right)$$

目标位置 P 的坐标可以由以下计算式得到：

$$P_x = \cos(T)\sin(A)$$

$$P_y = \sin(\mathrm{T})$$

$$P_z = \cos(T)\cos(A)$$

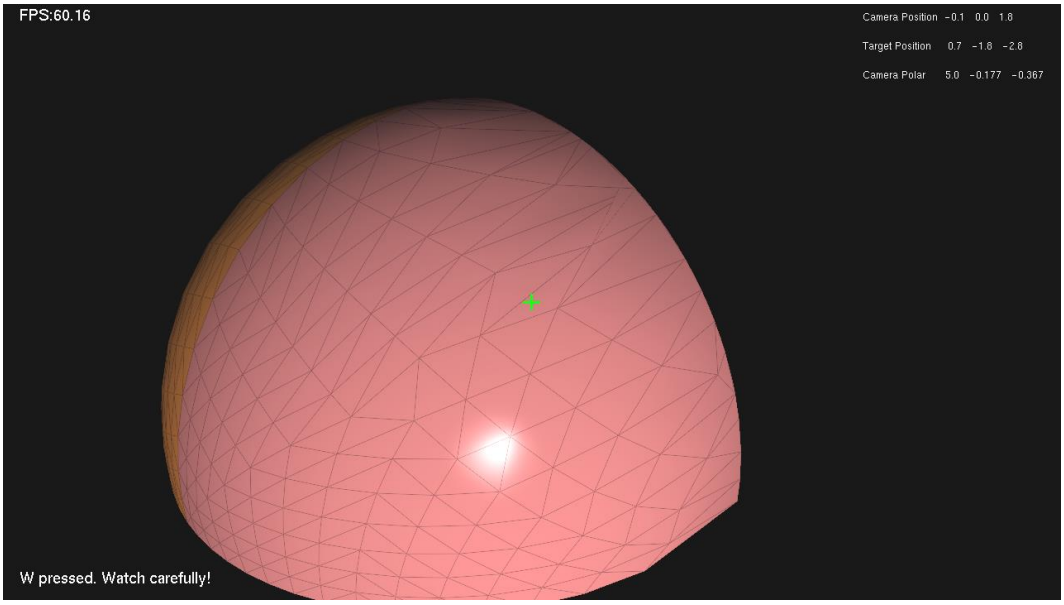

Figure 15 pass3()**计算结果**

右半边是使用 pass3()计算得到的八分之一球面。可以发现在该球面的 u＞0.5 部分出现了严重的偏差和扭曲。仔细观察边缘，甚至出现了表面不光滑、三角形重叠的现象。

e) Geometry Shader：计算多边形线框

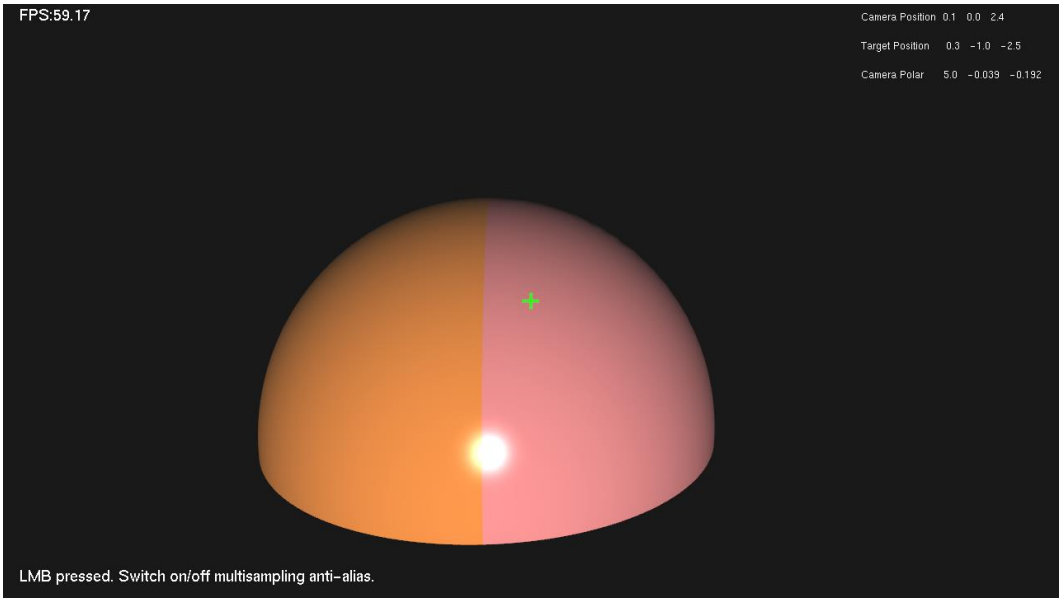

Figure 16 **按 O 键开启/关闭线框显示**

为了更好地观察细分结果，我们绘制模型三角形面片的边。我们使用几何着色器并利用 OpenGL 的插值功能以在片段着色器中获得该片段到它所在的三角形面片的三边的距离。我们只需在几何着色器中对每一个输入的三角形面片的三个顶点，计算它到对边的距离并保存在向量 EdgeDistance 中。

```
vec3 p0 = vec3(ViewportMatrix * (gl_in[0].gl_Position / gl_in[0].gl_Position.w));
vec3 p1 = vec3(ViewportMatrix * (gl_in[1].gl_Position / gl_in[1].gl_Position.w));
vec3 p2 = vec3(ViewportMatrix * (gl_in[2].gl_Position / gl_in[2].gl_Position.w));

float a = length(p1 - p2);
float b = length(p2 - p0);
float c = length(p1 - p0);
float alpha = acos((b * b + c * c - a * a) / (2.0 * b * c));
float beta = acos((a * a + c * c - b * b) / (2.0 * a * c));
float ha = abs(c * sin(beta));
float hb = abs(c * sin(alpha));
float hc = abs(b * sin(alpha));

EdgeDistance = vec3(ha, 0, 0);
```

Figure 17 **计算三角形顶点到对边的距离** 17:5@basic.geom

EdgeDistance 变量应加关键字 noperspective，指定 OpenGL 使用普通的线性插值。当这一切完成之后，OpenGL 会将模型光栅化，利用三个顶点的 EdgeDistance 插值得到当前片段的 EdgeDistance。于是我们在片段着色器中就可以根据片段到面片的三边的距离（分别存储在 EdgeDistance 的三个坐标分量中）与意图绘制的线条宽度 LineWidth 比较，然后决定是否将线条颜色加入混合。

```
vec4 color;
if (Normal.z > 0) {
    color = vec4(phongModel(vec3(Position), Normal), 1.0);
}
else {
    color = vec4(phongModel(vec3(Position), -Normal), 1.0);
}
FragColor = DrawWireFrame ? mix(color, LineColor, edgeMix()) : color;
```

Figure 18 **绘制线框** 59:5@basic.frag

f) Fragment Shader：使用 Phong 光照模型和双面光照

片段着色器不仅用于对多边形线框着色，我还用它来实现 Phong 光照模型，在模型表面产生明暗和高光的效果。与此同时双面光照也被应用在模型上，只需增加一次片段的面法线方向的判断。但是直到现在我们从未处理过顶点的法线方向，于是我们回到 TES，在计算顶点位置时即计算顶点法线方向，并且一路传递过来，最后进入到片段着色器中参与计算。在实验过程中，细分等级越高，模型表面的高光越明亮：对比 Figure 1, Figure 7, Figure 8 可知。

五、 实验总结与心得

a) 实验中遇到的困难

i. 利用 pass3()计算得到的八分之一球面，在 u 较大的区域出现了三角形面丢失的情况。经判断为变量精度不足导致了除 0 错误等问题。但是我试图使用 precision highp float;语句来设置浮点数精度，但是失败了。最后，我增加了一次条件判断，若有可能导致除 0 错

误，将除数手动设置为一个接近 0 的小数，如 0.0001。使用这个方法，勉强解决了问题。

```
float t;
float diff = 1 - v;
if (diff <= 0) {
    diff = 0.0001;
}
t = atan(v, diff);
```

Figure 19 **增加判断** 41:5@basic.tes

b) 改进

i. pass3()算法产生的顶点出现重复、交叉的问题，怀疑是算法本身有瑕疵，应该予以改进。

ii. 对模型应用 Phong 光照模型时，在正反侧的交界处会出现不自然的漫反射颜色，如 Figure 13 所示，怀疑是法线问题，应该予以改进。

c) 实验体会

经过本次实验，我对细分着色器和几何着色器有了基本的了解，学会了如何用它们来创建新的顶点和添加顶点属性。实验中综合使用 5 种着色器，收获很大。