浙江大学

本科实验报告

课程名称：	基于 GPU 的绘制
实验名称：	Edge Detection
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：Edge Detection 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

在实验过程中熟悉片段着色器的使用

b) 实验内容

使用 GLSL 语言编写着色器，对屏幕上的图像应用边缘检测效果。

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

本次实验使用了顶点着色器和片段着色器，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。

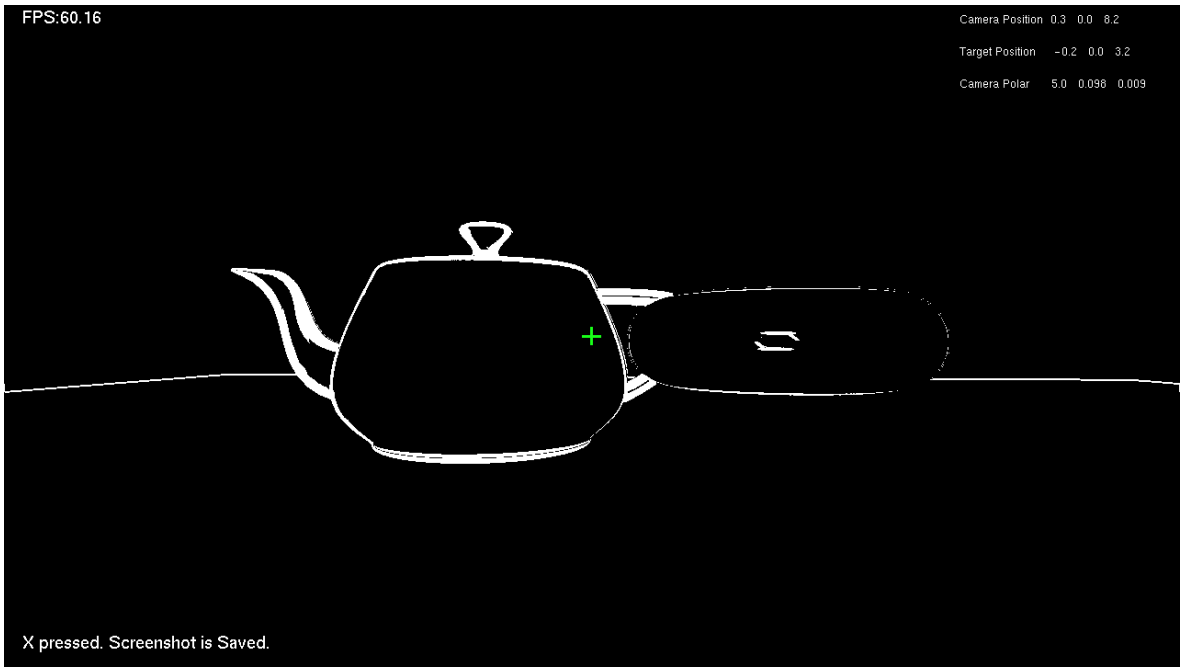

Figure 1 **实验效果**

b) OpenGL：准备 FBO 并绑定纹理

完成本次实验的场景渲染，需要进行两次绘制，分别为 Pass 1 和 Pass 2。在绘制场景之前，我们需要做一些准备工作，目的是建立 Pass 1 与 Pass 2 之间的数据沟通连接。为了使 Pass 1 得到的图像数据能够在 Pass 2 的处理过程中顺利读取，我们使用了帧缓存对象 Frame Buffer Object。首先，我们向 OpenGL 申请一个帧缓存标识符并将它绑定为帧缓存类型：

```
glGenFramebuffers(1, &fboHandle);
glBindFramebuffer(GL_FRAMEBUFFER, fboHandle);
```

Figure 2 **创建帧缓存对象** 514:5@system.cpp

接着我们创建一个与窗口尺寸相同的纹理对象，并为它指定纹理参数：

```
glGenTextures(1, &renderTex);
glBindTexture(GL_TEXTURE_2D, renderTex);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 1280, 720, 0, GL_RGBA, GL_UNSIGNED_BYTE, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, 0);
```

Figure 3 **创建纹理对象** 518:5@system.cpp

将纹理对象绑定到帧缓存的 0 号颜色缓存：

```
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, renderTex, 0);
```

Figure 4 **将纹理对象绑定到帧缓存** 526:5@system.cpp

创建一个与窗口尺寸相同的深度缓存：

```
GLuint depthBuf;
glGenRenderbuffers(1, &depthBuf);
glBindRenderbuffer(GL_RENDERBUFFER, depthBuf);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, 1280, 720);
```

Figure 5 **创建深度缓存** 529:5@system.cpp

将深度缓存绑定到帧缓存：

```
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, depthBuf);
```

Figure 6 **将深度缓存保存到帧缓存** 535:5@system.cpp

最后，设定绘制的图像输出到 0 号颜色缓存：

```
GLenum drawBuffers[] = {GL_COLOR_ATTACHMENT0};
glDrawBuffers(1, drawBuffers);
```

Figure 7 **设定绘制时的目标** 538:5@system.cpp

c) Pass 1：绘制场景到帧缓存

在片段着色器 basic.frag 中，定义了两条绘制路径：pass1()和 pass2()，在第一次绘制时，我们使用 pass1()。pass1()的作用是按普通的方法绘制出使用了 Phong 光照模型的场景。在第一次绘制之前，我们启用刚才创建的帧缓存，将它指定为当前渲染目标：

```
glBindFramebuffer(GL_FRAMEBUFFER, fboHandle);
```

Figure 8 **指定当前渲染目标** 58:5@system.cpp

然后，我们指定使用第一条渲染路径并绘制场景：

```
glUniformSubroutinesuiv(GL_FRAGMENT_SHADER, 1, &pass1Index);
glEnable(GL_DEPTH_TEST);
// Draw something here
updateMVPZero();
updateMVPOne();
teapot->render();
updateMVPTwo();
plane->render();
updateMVPThree();
torus->render();

glFlush();
```

Figure 9 **使用第一条渲染路径并绘制场景** 72:5@system.cpp

第一次绘制的结果如 Figure 10 所示。

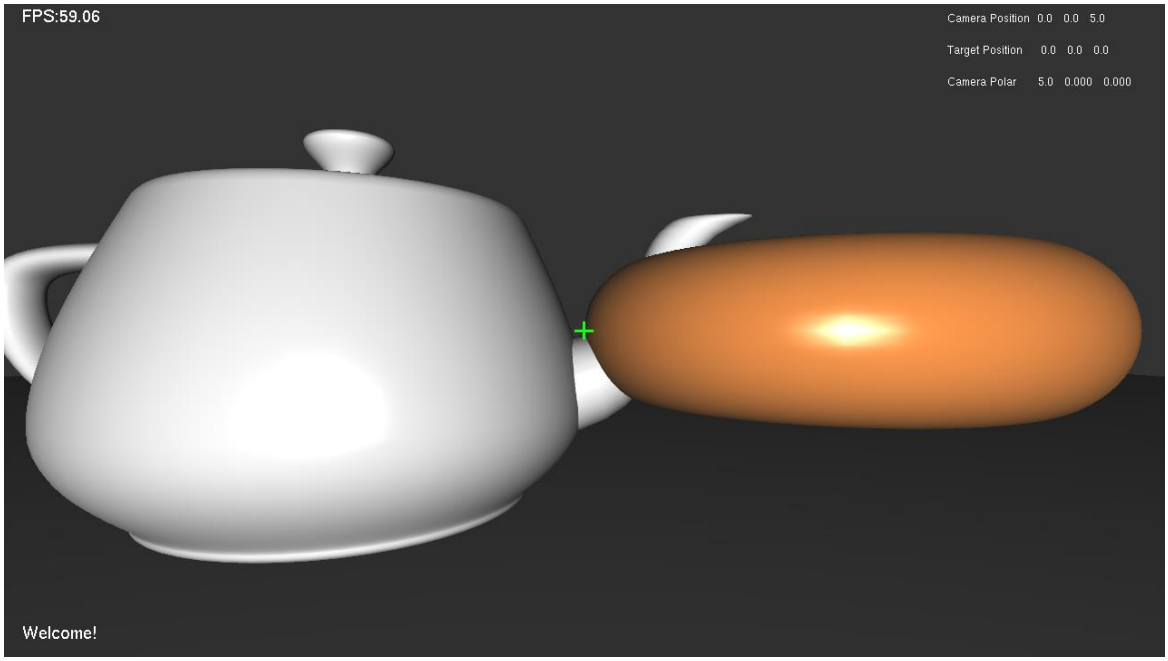

Figure 10 **第一次绘制结果**

d) Pass 2：对图像应用边缘检测

在第二次绘制中，我们所需要做的是从 FBO 绑定的纹理中读出图像数据，应用边缘检测，最后输出到屏幕上。

1. 读出图像数据

第一次绘制的结果已经保存在定义好的纹理对象中，在开始第二次绘制之前我们需要启用它。

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, renderTex);
```

Figure 11 **启用纹理对象** 93:5@system.cpp

于是我们在片段着色器中可以直接使用 0 号纹理位置读取数据，它里面保存的就是第一次绘制时得到的结果。

```
layout(binding = 0) uniform sampler2D RenderTex;
```

Figure 12 **读取纹理输入** 7:1@basic.frag

2. 应用边缘检测

在这一步中，我们获取每一个片段的 8 邻域内的像素颜色，转化为灰度值后使用 Sobel 算子计算这一片段上的亮度梯度变化，与阈值进行比较，若大于阈值则设置为白色，若小于预置则设置为黑色。

```
// Pass #2
subroutine (RenderPassType)
vec4 pass2() {
    float dx = 1.0 / float(Width);
    float dy = 1.0 / float(Height);
    float s00 = luma(texture(RenderTex, TexCoord + vec2(-dx, dy)).rgb);
    float s10 = luma(texture(RenderTex, TexCoord + vec2(-dx, 0.0)).rgb);
    float s20 = luma(texture(RenderTex, TexCoord + vec2(-dx, -dy)).rgb);
    float s01 = luma(texture(RenderTex, TexCoord + vec2(0.0, dy)).rgb);
    float s21 = luma(texture(RenderTex, TexCoord + vec2(0.0, -dy)).rgb);
    float s02 = luma(texture(RenderTex, TexCoord + vec2(dx, dy)).rgb);
    float s12 = luma(texture(RenderTex, TexCoord + vec2(dx, 0.0)).rgb);
    float s22 = luma(texture(RenderTex, TexCoord + vec2(dx, dy)).rgb);
    float sx = s00 + 2 * s10 + s20 - (s02 + 2 * s12 + s22);
    float sy = s00 + 2 * s01 + s02 - (s20 + 2 * s21 + s22);
    float dist = sx * sx + sy * sy;
    if (dist > EdgeThreshold) {
        return vec4(1.0);
    }
    else {
        return vec4(0.0, 0.0, 0.0, 1.0);
    }
}
```

Figure 13 **使用 Sobel 卷积** 50:1@edge.frag

3. 输出到窗口

为了将边缘检测的结果输出到屏幕上，我们按视口大小绘制两个三角形，使其拼合起来正好填满视口。绘制这两个图形的意义在于允许我们在片段着色器中绘制像素时能够得到当前像素在屏幕上的位置，所以这两个三角形拼合起来的矩形应该具有左下(0, 0)，左上(0, 1)，右上(1, 1)和右下(1, 0)的纹理坐标。结合上一步中的输出结果，就能在屏幕上显示出边缘检测效果了。

```
GLfloat verts[] = {
        -1.0f, -1.0f, 0.0f, 1.0f, -1.0f, 0.0f, 1.0f, 1.0f, 0.0f,
        -1.0f, -1.0f, 0.0f, 1.0f, 1.0f, 0.0f, -1.0f, 1.0f, 0.0f
};
GLfloat tc[] = {
        0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 1.0f,
        0.0f, 0.0f, 1.0f, 1.0f, 0.0f, 1.0f
};
```

Figure 14 **两个三角形的顶点坐标及其纹理坐标** 554:5@system.cpp

第二次绘制的结果如 Figure 15 所示。

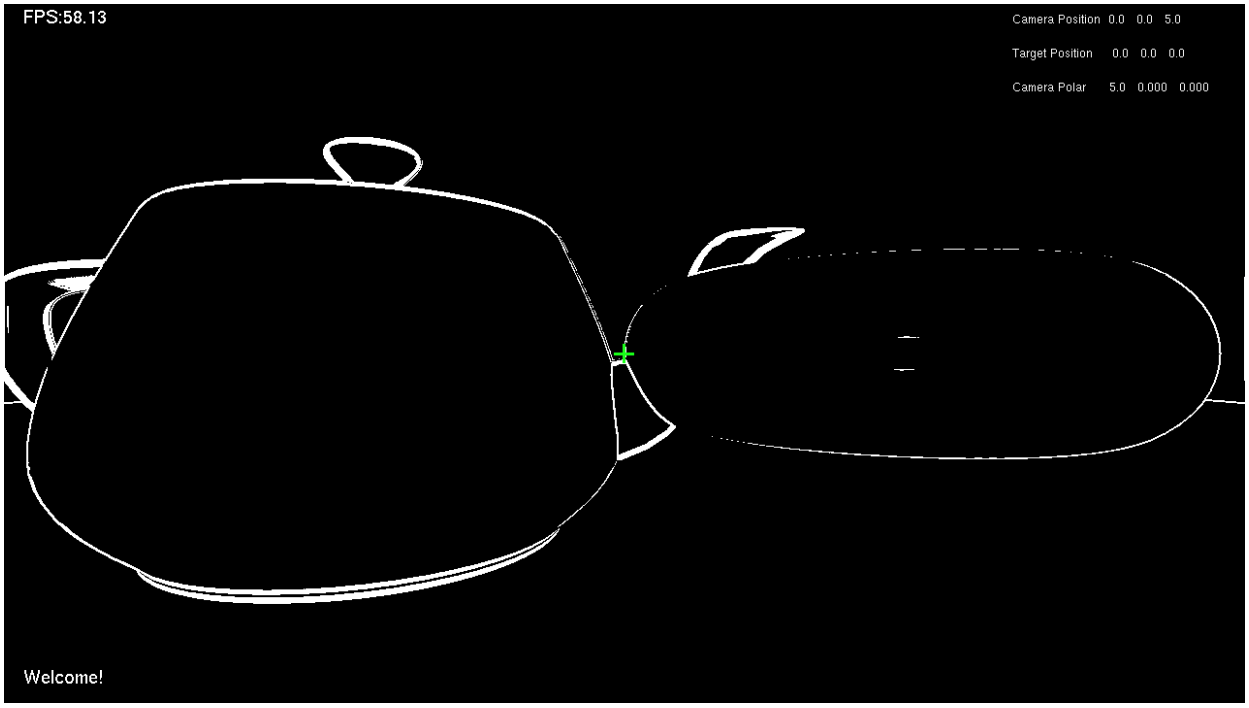

Figure 15 **第二次绘制的结果**

e) 阈值变化对效果的影响

在场景中按小键盘的“+”和“-”键可以调整边缘检测的阈值 edgeThreshold（范围为 0.03 至 0.08）。从结果 Figure 17 和 Figure 18 中我们可以看到阈值设置得越高，边缘越细（甚至消失）；阈值越低，边缘范围越大。

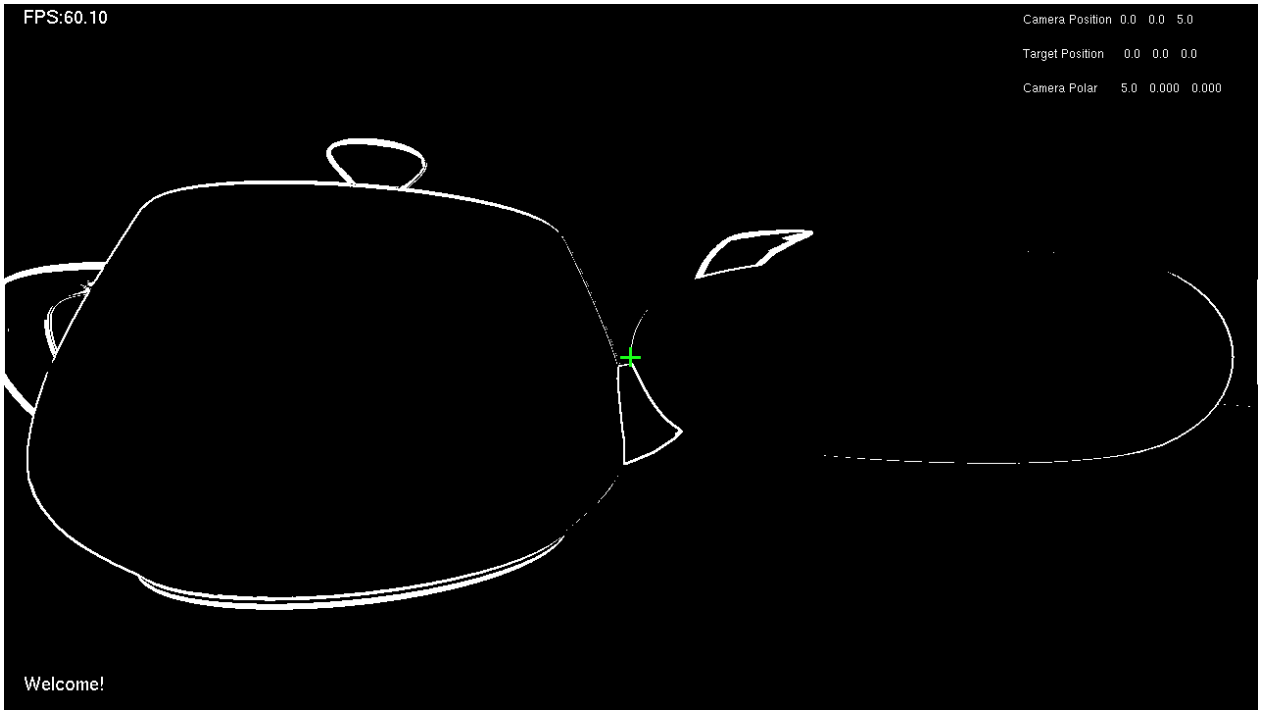

Figure 16 edgeThreshold = 0.08

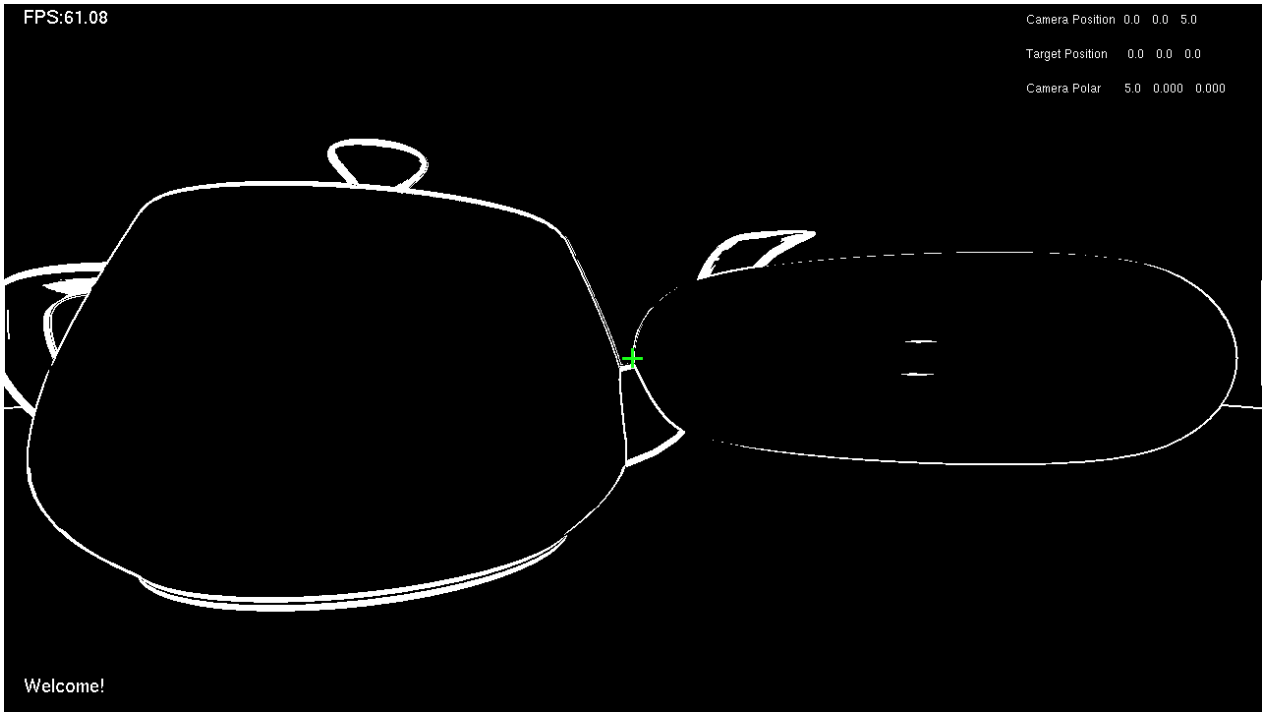

Figure 17 edgeThreshold = 0.04

五、 实验总结与心得

a) 可能的改进

i. 可以参考 Canny 边缘检测算法，分三步进行边缘检测：降噪（利用高斯核）、非极大值抑制（Sobel 算子计算梯度变化）、连接边缘（双阈值算法）。但是这需要更加复杂的操作和更多的计算。

ii. 现在利用 Sobel 算子计算时（Figure 13）有不少重复的操作，存在优化空间。

b) 实验体会

经过本次实验，我对片段着色器和边缘检测算法有了更深入的理解，已经可以熟练修改 OpenGL 程序和着色器代码。