Integrate Shaders into OpenGL

梁晨 3180102160

- 实验环境

  本课程的所有作业均在macOS Catalina 10.15.7上运用集成开发平台XCode 12.0.1完成。我在文件中附上了XCode的原始工程文件。C++相关源码在与最上层文件夹同名的文件夹中，而GLSL源码（即各个shader）在Debug文件夹中。编译生成的在macOS上可执行的UNIX可执行文件和建模过程中需要用到的.txt, .obj, 以及. tga（相关纹理）文件，也都在Debug文件夹中。渲染的结果实时、动态地显示在GLUT窗口中，我将渲染结果做了截图，统一放进output文件夹中。

- 建模过程和操作指南

  本实验的建模过程参考了MIT_6.837图形学课程的建模过程，线性代数库（包括vectors.h、matrix.h、matrix.cpp）也直接取自MIT_6.837。其他源码文件中，loader.h和loader.cpp是完全新写的，目的是载入相关Shader。其余文件都是在之前完成的MIT_6.837图形学课程的作业的基础上，略作修改得到的。

  在操作程序的过程中我们首先通过命令行载入参数，指明场景.txt文件和我们需要载入的Shader文件名：

  ```
  -input bunny.txt -vs minimal.vert -fs minimal.frag
  ```

  在场景.txt文件中，我们需要指明相机的中心和三个向量（即MVP矩阵的各相关参数）、背景色、光源分布情况、物体材质以及场景物体和对物体做的相关变换。为了简化建模过程，本作业中，我们只能通过.obj载入场景物体：

  ```
  Camera {
      center 0 0.8 5
      direction 0 0 -1
      up 0 1 0
      angle 30
  }

  Lights {
      numLights 1
      DirectionalLight {
          direction -0.5 -0.5 -1
          color 0.9 0.9 0.9
      }
  }

  Background {
  ```

```
    color 0.8 0.8 0.8
    ambientLight 0.1 0.1 0.1
}

Materials {
    numMaterials 1
   PhongMaterial { diffuseColor 1 1 1 }
}

Group {
    MaterialIndex 0
  Mesh {
           obj_file bunny_1k.obj
        }

    Transform {
        UniformScale  5
  Translate  0.03 -0.0666 0
        }
}
```

在实时渲染的窗口中，我们可以在按住鼠标左键的情况下进行拖动旋转相机，按住鼠标右键进行拖动沿direction向量推拉相机，按住鼠标滚轮（中键）进行拖动，在up向量和horizontal向量组成的平面中平移相机。

- Shader的载入

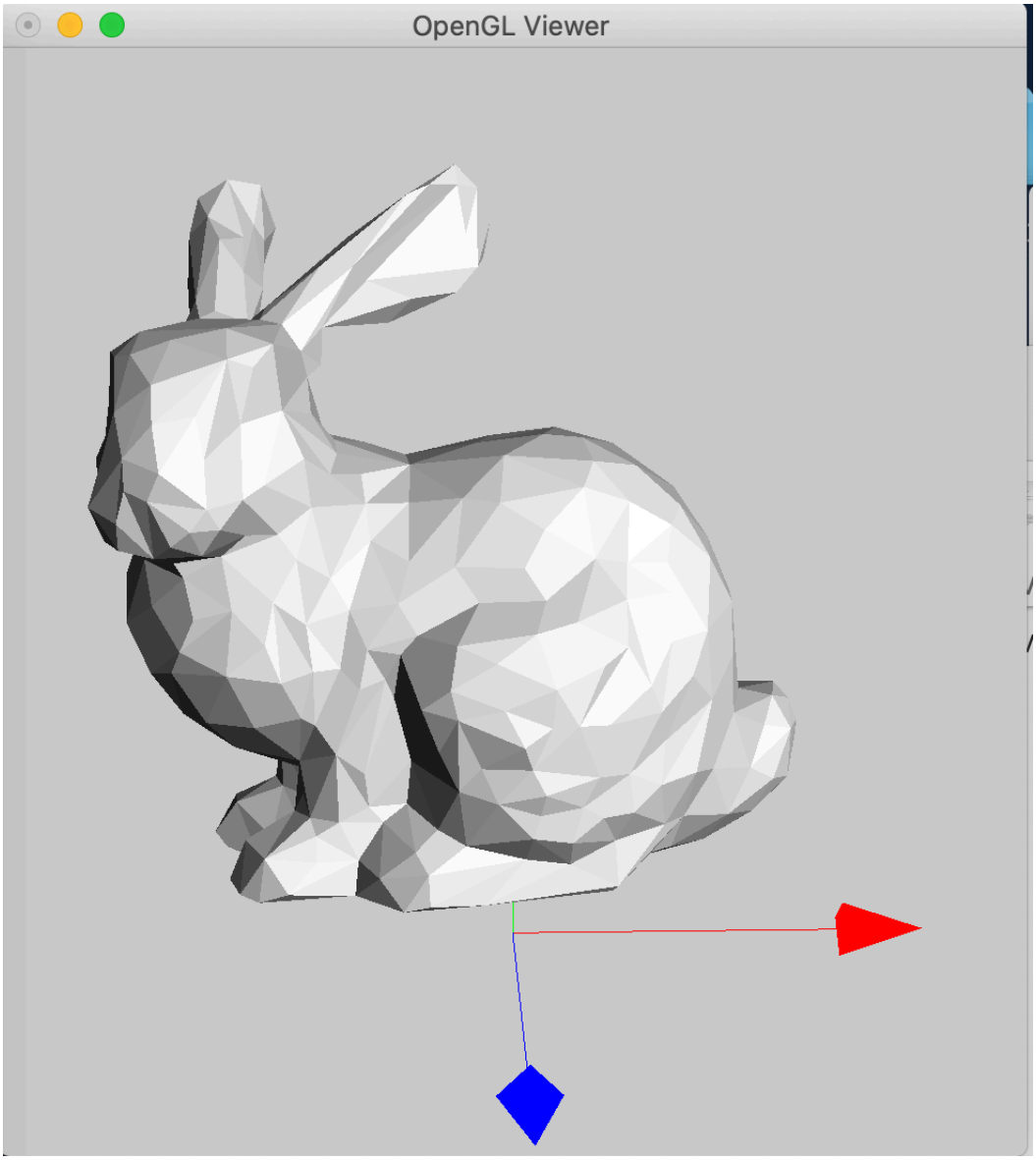
  本实验中Shader的载入在loader.cpp中完成，我们对vertex shader、fragment shader分别进行载入，过程相似，均为compile，attach。在两个shader都compile和attach之后，再对program对象去做link和use program。注意compile和link后要检查是否成功，如果不成功的话，需要打印错误信息，方便我们调试Shader，代码如下（由于对两个shader的处理相似，只附上载入fragment shader的代码）：

```
//compile and attach fragment shader
if(fragFilename){
    // read ascii character from .frag file
     unsigned char* fragmentShaderSource=readShaderFile(fragFilename);
    // create fragment shader object
     fragmentShaderObject=glCreateShader(GL_FRAGMENT_SHADER);
     GLint flength;
     glShaderSource(fragmentShaderObject,1,(const
char**)&fragmentShaderSource,&flength);
      if(fragmentShaderSource){
    free(fragmentShaderSource);
    fragmentShaderSource=NULL;
  }
  //compile fragment shader
  GLint fcompiled;
```

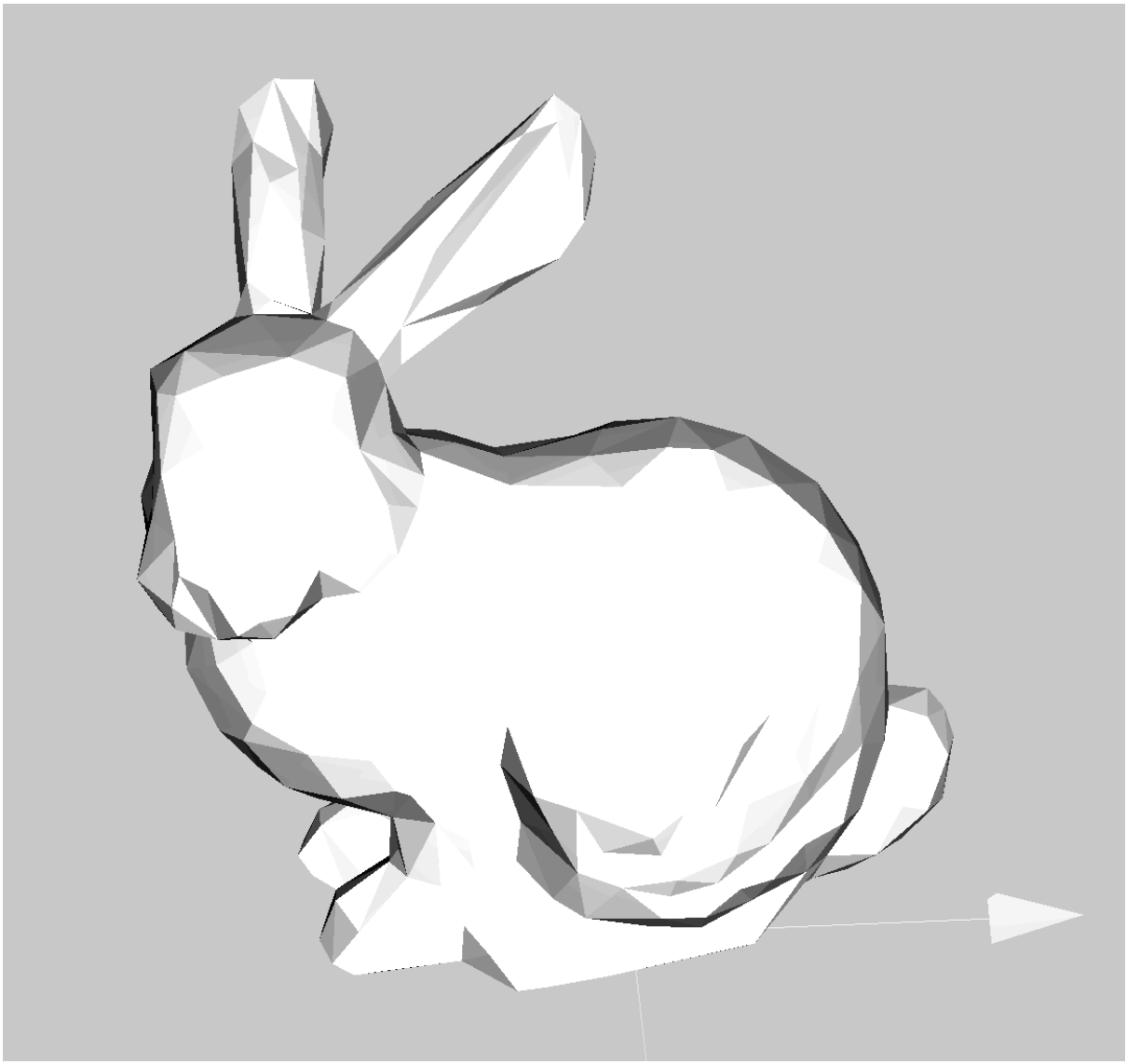
```
  glCompileShader(fragmentShaderObject);
    glGetShaderiv(fragmentShaderObject,GL_COMPILE_STATUS,&fcompiled);
  if(fcompiled){
    cout<<"Fragment shader compilation is successful!"<<endl;
  }
  else{
    //if compilation failed, print out log information
      cout<<"Fragment shader compilation failed."<<endl;
        int maxLength=0;
        char* errorLog=new char[maxLength];
            glGetShaderInfoLog(fragmentShaderObject, 1000, &maxLength,
errorLog);
        printf("%s\n",errorLog);
      assert(0);
    }
  //attach fragment shader to program object
      glAttachShader(programObject,fragmentShaderObject);
  }
  else cout<<"Use default fragment shader."<<endl;

  //link program object
  glLinkProgram(programObject);
  GLint linked;
    glGetProgramiv(programObject,GL_LINK_STATUS,&linked);
  if(linked){
    cout<<"Program linking is successful!"<<endl;
  }
  else{
    //if link failed, print out log information
      cout<<"Link failed."<<endl;
        int maxLength=0;
        char* errorLog=new char[maxLength];
        glGetProgramInfoLog(programObject, 1000, &maxLength, errorLog);
        printf("%s\n",errorLog);
    assert(0);
  }

  //Use program object
  glUseProgram(programObject);
  return programObject;
```

- 实验结果

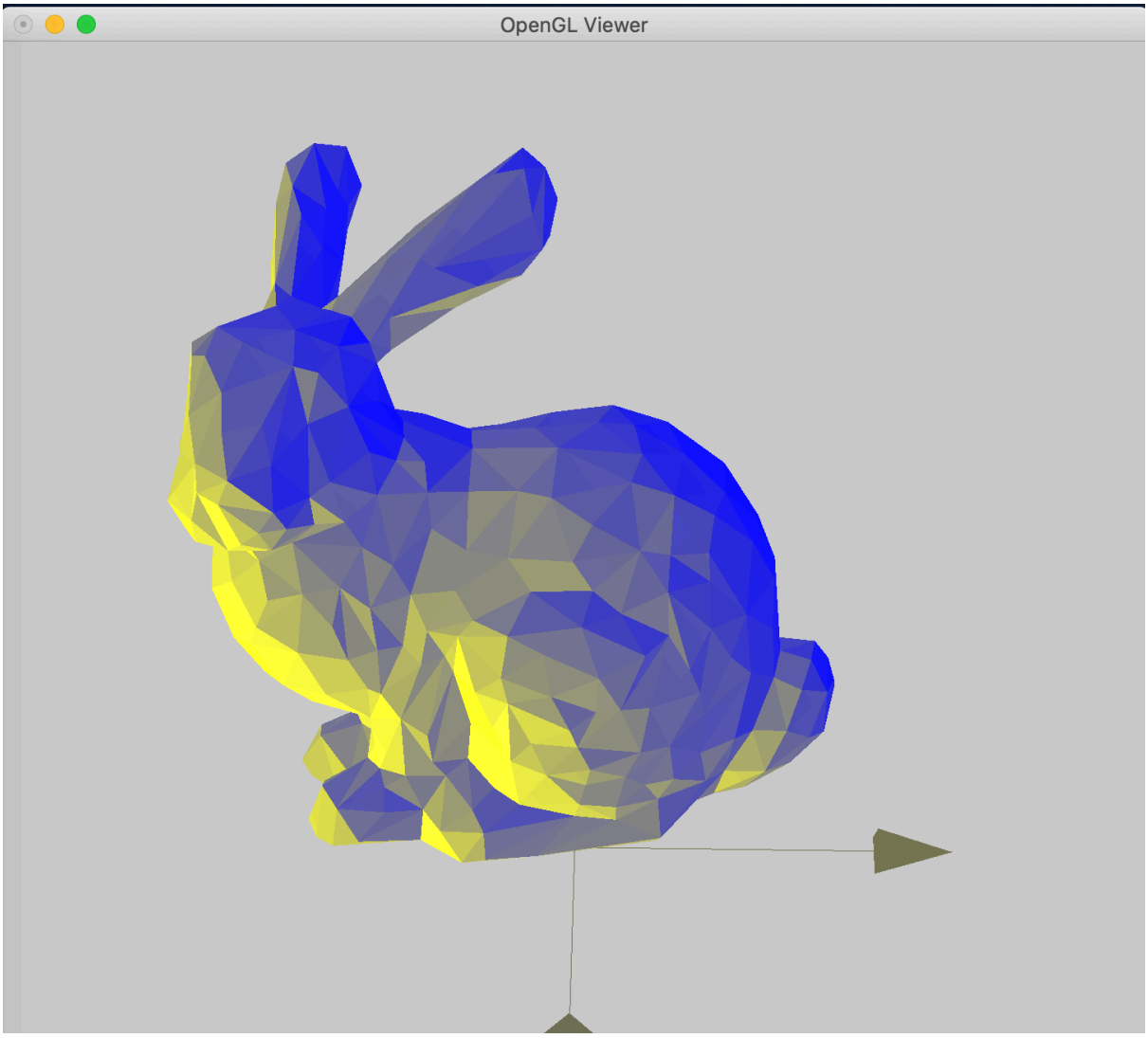
  不载入shader时，我们根据场景中指定的材质，进行Blinn-Phong渲染，结果如下：



当载入fragment shader载入ivory shader时，得到结果如下：



当载入goochVertexShader和goochFragmentShader时，得到的结果如下：



可以看到，我们成功实现了ivory shader和gooch shader的风格化绘制效果。