1 总览

在这次编程任务中，我们会进一步模拟现代图形技术。我们在代码中添加了 Object Loader(用于加载三维模型), Vertex Shader 与 Fragment Shader，并且支持了纹理映射。

而在本次实验中，你需要完成的任务是:

1. 修改函数 `rasterize_triangle(const Triangle& t)` in rasterizer.cpp: 在此处实现与作业 2 类似的插值算法，实现法向量、颜色、纹理颜色的插值。
2. 修改函数 `get_projection_matrix()` in main.cpp: 将你自己在之前的实验中实现的投影矩阵填到此处,此时你可以运行 `./Rasterizer output.png normal` 来观察法向量实现结果。
3. 修改函数 `phong_fragment_shader()` in main.cpp: 实现 Blinn-Phong 模型计算 Fragment Color.
4. 修改函数 `texture_fragment_shader()` in main.cpp: **在实现 Blinn-Phong 的基础上**，将纹理颜色视为公式中的 kd，实现 Texture Shading Fragment Shader.
5. 修改函数 `bump_fragment_shader()` in main.cpp: **在实现 Blinn-Phong 的基础上**，仔细阅读该函数中的注释，实现 Bump mapping.
6. 修改函数 `displacement_fragment_shader()` in main.cpp: **在实现 Bump mapping 的基础上**，实现 displacement mapping.

2 开始编写

2.1 编译与使用

在课程提供的虚拟机上，下载本次实验的基础代码之后，请在 SoftwareRasterizer 目录下按照如下方式构建程序:

```
$ mkdir build
$ cd ./build
$ cmake ..
$ make
```

这将会生成命名为 `Rasterizer` 的可执行文件。使用该可执行文件时，你传入的第二个参数将会是生成的图片文件名，而第三个参数可以是如下内容：

- texture: 使用代码中的 **texture** shader.
  使用举例: `./Rasterizer output.png texture`
- normal: 使用代码中的 **normal** shader.
  使用举例: `./Rasterizer output.png normal`
- phong: 使用代码中的 **blinn-phong** shader.
  使用举例: `./Rasterizer output.png phong`
- bump: 使用代码中的 **bump** shader.
  使用举例: `./Rasterizer output.png bump`
- displacement: 使用代码中的 **displacement** shader.
  使用举例: `./Rasterizer output.png displacement`

  当你修改代码之后，你需要重新 make 才能看到新的结果。

2.2 框架代码说明

相比上次实验，我们对框架进行了如下修改：

1. 我们引入了一个第三方.obj 文件加载库来读取更加复杂的模型文件，这部分库文件在 `OBJ_Loader.h file`. 你无需详细理解它的工作原理，只需知道这个库将会传递给我们一个被命名被 TriangleList 的 Vector，其中每个三角形都有对应的点法向量与纹理坐标。此外，与模型相关的纹理也将被一同加载。**注意：如果你想尝试加载其他模型，你目前只能手动修改模型路径。**

2. 我们引入了一个新的 Texture 类以从图片生成纹理，并且提供了查找纹理颜色的接口：Vector3f getColor(float u, float v)

3. 我们创建了 Shader.hpp 头文件并定义了 fragment_shader_payload，其中包括了 Fragment Shader 可能用到的参数。目前 main.cpp 中有三个 Fragment Shader，其中 fragment_shader 是按照法向量上色的样例 Shader，其余两个将由你来实现。

4. 主渲染流水线开始于 `rasterizer::draw(std::vector<Triangle> &TriangleList)`. 我们再次进行一系列变换，这些变换一般由 Vertex Shader 完成。在此之后，我们调用函数 `rasterize_triangle`.

5. `rasterize_triangle` 函数与你在之前实验中实现的内容相似。不同之处在于被设定的数值将不再是常数，而是按照 Barycentric Coordinates 对法向量、颜色、纹理颜色与底纹颜色 (Shading Colors) 进行插值。回忆我们上次为了计算 z value 而提供的 `[alpha, beta, gamma]`，这次你将需要将其应用在其他参数的插值上。你需要做的是计算插值后的颜色，并将 Fragment Shader 计算得到的颜色写入 framebuffer，这要求你首先使用插值得到的结果设置 fragment shader payload，并调用 fragment shader 得到计算结果。

2.3 运行与结果

在你按照上述说明将上次作业的代码复制到对应位置，并作出相应修改之后（**请务必认真阅读说明**），你就可以运行默认的 normal shader 并观察到如下结果：

实现 Blinn-Phong 反射模型之后的结果应该是：

实现纹理之后的结果应该是：

实现 Bump Mapping 后，你将看到可视化的凹凸向量:

实现 Displacement Mapping 后，你将看到如下结果:

3 评分与提交 （供助教参考）

评分:

- [5 分] 提交格式正确，包括所有需要的文件。代码可以正常编译、执行。
- [10 分] 参数插值: 正确插值颜色、法向量、纹理坐标、位置 (Shading Position) 并将它们传递给 fragment_shader_payload.
- [20 分]Blinn-phong 反射模型: 正确实现 phong_fragment_shader 对应的反射模型。
- [5 分] Texture mapping: 将 phong_fragment_shader 的代码拷贝到 texture_fragment_shader，在此基础上正确实现 Texture Mapping.
- [10 分] Bump mapping 与 Displacement mapping: 正确实现 Bump mapping 与 Displacement mapping.

- [Bonus 3 分] 尝试更多模型：找到其他可用的.obj 文件，提交渲染结果并把模型保存在 /models 目录下。这些模型也应该包含 **Vertex Normal** 信息。
- [Bonus 5 分] 双线性纹理插值：使用双线性插值进行纹理采样，在 Texture 类中实现一个新方法 Vector3f getColorBilinear(float u, float v) 并通过 fragment shader 调用它。为了使双线性插值的效果更加明显，你应该考虑选择更小的纹理图。请同时提交纹理插值与双线性纹理插值的结果，并进行比较。