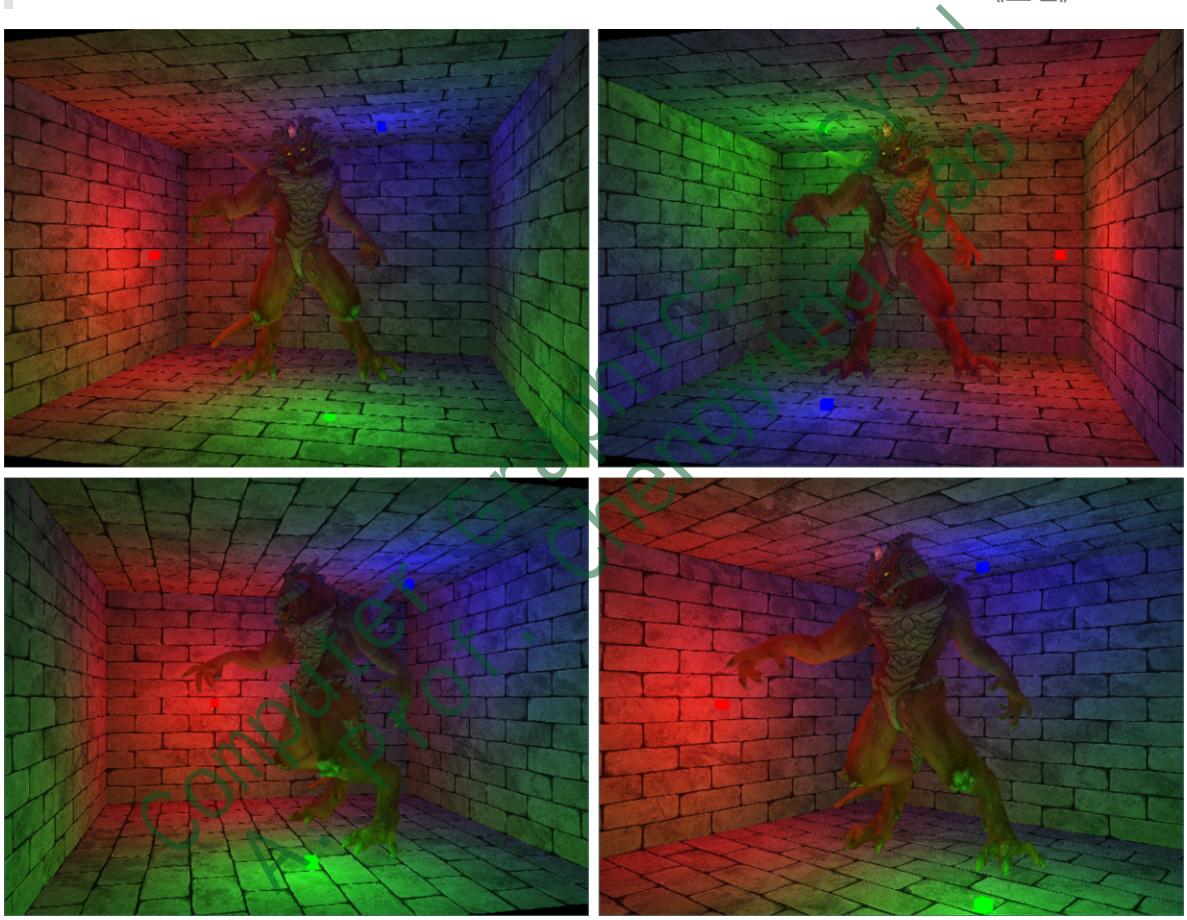
Assignment 3: Lighting & Texturing

Computer Graphics Teaching Stuff, Sun Yat-Sen University

Due Date: ***12 pm, 18 May 2021***

Submission: Send the report (In ***PDF*** *Format) to mailbox cg_hw_2021@163.com*

> 起初神创造天地，地是虚空混沌，渊面黑暗，神的灵运行在水面上。在神创造天地的第一日，**神说，要有光，就有了光**。神看光是好的，就把光暗分开了。神称光为“昼”，称暗为“夜”。
>
> ——《圣经》



本次作业你们将实现的效果图

1、作业概述

在之前的作业中，你们完成了物体模型从三维到二维的转换、以及屏幕空间的光栅化过程，可以在窗口上看到三维的物体了。在本次作业中，你们将在此基础上添加纹理和三维的光照效果，为这个虚拟的计算机世界施展光的魔法！本次作业你们将在给定的代码框架下，实现纹理最邻近采样、纹理双线性插值采样、Phong光照和Blinn-Phong光照。完成本次作业，你将对图形渲染中的着色有更加深入的掌握和理解。

2、代码框架

关于本次作业的框架代码部署和构建，请仔细阅读`CGAssignment3/readme.pdf`文档，基本上与`CGAssignment2`没有太大的差别。本次作业依赖的第三方库主要有四个，分别是：

- **GLM**：线性代数数学库，如果学过LearnOpenGL教程则你应该对这个数学库挺熟悉的
- **SDL2**：窗口界面库，主要用于创建窗口给并显示渲染的图片结果，本作业不需要你对这个库深入了解
- **TinyObjLoader**：模型数据加载库，用于加载obj模型，本作业不需要你对这个库深入了解
- **stb_image**：轻量级的图片加载库，用于加载纹理图片，本作也不需要你对这个库深入了解

这些第三方库同学们无需太过关注，我们的框架代码已经构建好了相应的功能模块。目录`CGAssignment3/src`存放我们的渲染器的所有代码文件：

- **main.cpp**：程序入口代码，负责执行主要的渲染循环逻辑；
- **TRFrameBuffer(.h/.cpp)**：帧缓冲类，存放渲染的结果（包括颜色缓冲和深度缓冲），你无需修改此文件；
- **TRShadingPipeline(.h/.cpp)**：渲染管线类，负责实现顶点着色器、光栅化、片元着色器的功能；
- **TRWindowsApp(.h/.cpp)**：窗口类，负责创建窗口、显示结果、计时、处理鼠标交互事件，无需修改；
- **TRDrawableMesh(.h/.cpp)**：可渲染对象类，负责加载obj网格模型、存储几何顶点数据，无需修改；
- **TRRenderer(.h/.cpp)**：渲染器类，负责存储渲染状态、渲染数据、调用绘制。
- **TRShadingState.h**：存放一些渲染的设置选项，无需修改。
- **TRTexture2D(.h/.cpp)**：纹理类，负责加载图片、根据uv坐标进行纹理采样。
- **TRUtils.h/.cpp**：存放一些工具函数，无需修改。

核心的渲染模块没有借助任何第三方库，完全是利用纯粹的代码构建了类似于OpenGL的渲染管线，同学们可以类比于OpenGL来阅读本代码框架（无任何硬件层面的加速和并行，因此通常叫软光栅化渲染器）。本次作业你需要修改和填补的代码的地方在`TRTexture2D.cpp`文件和`TRShadingPipeline.cpp`文件，请大家着重注意这两个文件的代码。

`main.cpp`已经实现了窗口的创建、渲染器的实例化、渲染数据的加载、渲染循环的构建，请你类比OpenGL的渲染流程，仔细体会其中的逻辑。对代码有任何的疑问，可在群里匿名讨论。渲染的核心逻辑在`TRRenderer.cpp`文件的`renderAllDrawableMeshes`函数（不需要你对这个函数做任何的修改），同学们可以结合第三节课讲的渲染管线流程进行理解：

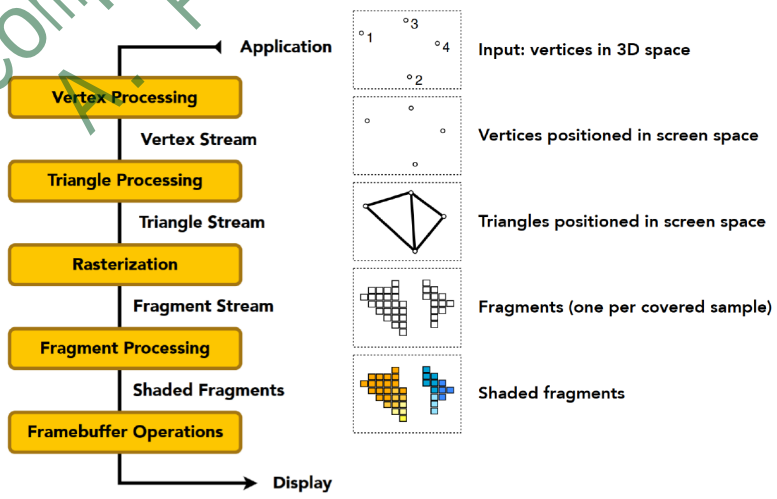


图1 基于光栅化的渲染管线

考虑到同学们的基础，同学们可能对渲染器的一些代码不是很理解，这个没关系，不理解的地方大家可以暂时放一下，随着学习的推进，后面可以回来再看。完成本次作业不需要你对整个框架代码做彻底的理解！本次作业的光源类型为点光源，其定义在`TRShadingState.h`文件：

Computer Graphics - SYSU
A. Prof. Chenglying Gao

```cpp
//Point lights
class TRPointLight
{
public:
    glm::vec3 lightPos;//Note: world space position of light source
    glm::vec3 attenuation;
    glm::vec3 lightColor;

    TRPointLight(glm::vec3 pos, glm::vec3 atten, glm::vec3 color)
        : lightPos(pos), attenuation(atten), lightColor(color) {}
};
```

点光源是处于世界中某一个位置的光源，它会朝着所有方向发光，但光线会随着距离逐渐衰减。想象作为投光物的灯泡和火把，它们都是点光源。

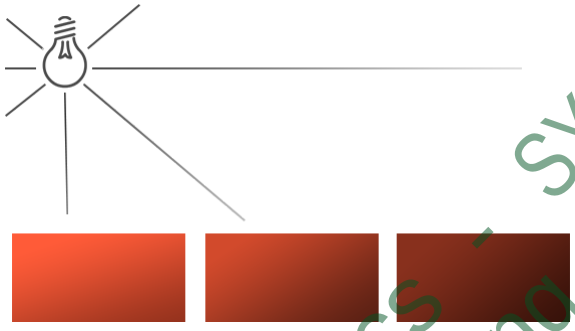


图2 点光源

随着光线传播距离的增长逐渐削减光的强度通常叫做衰减(Attenuation)。随距离减少光强度的一种方式是使用一个线性方程。这样的方程能够随着距离的增长线性地减少光的强度，从而让远处的物体更暗。然而，这样的线性方程通常会看起来比较假。在现实世界中，灯在近处通常会非常亮，但随着距离的增加光源的亮度一开始会下降非常快，但在远处时剩余的光强度就会下降的非常缓慢了。我们将使用如下的光源衰减公式：

$$F_{att} = \frac{1.0}{K_c + K_l \cdot d + K_q \cdot d^2}$$

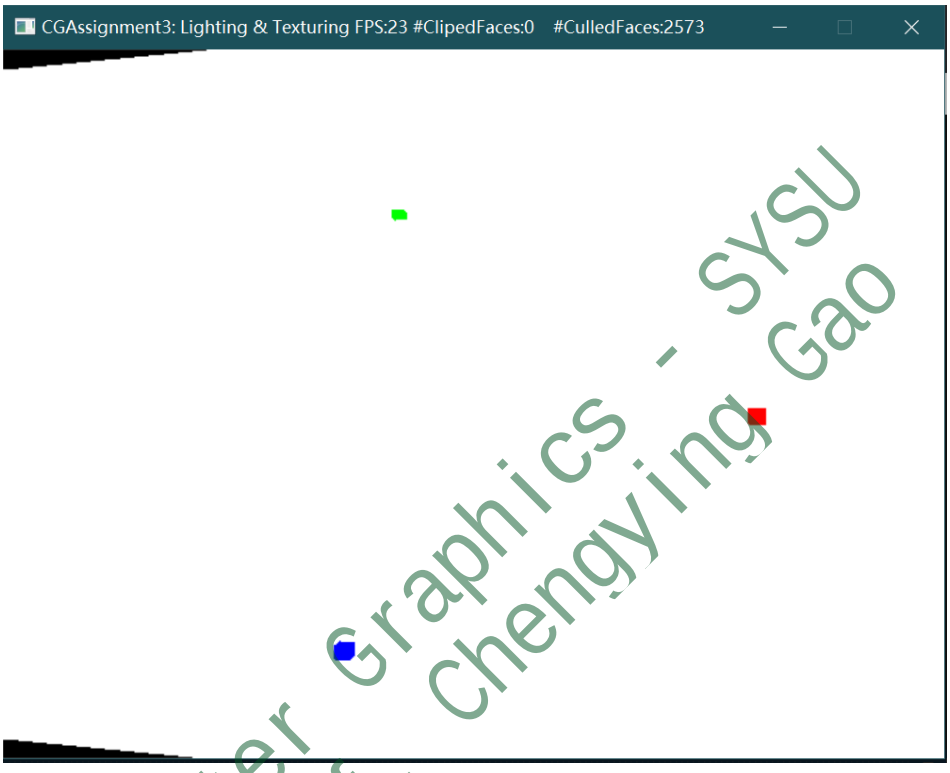
在这里d代表了片段到点光源的距离。在上式中，K_c是衰减常数项(constant)，通常取值为1.0，它的主要作用是保证分母永远不会比1小，否则的话在某些距离上它反而会增加强度，这肯定不是我们想要的效果。K_l是一次项(linear)，一次项会与距离值相乘，以线性的方式减少强度。K_q为二次项(quadratic)，二次项会与距离的平方相乘，让光源以二次递减的方式减少强度。二次项在距离比较小的时候影响会比一次项小很多，但当距离值比较大的时候它就会比一次项更大了。由于二次项的存在，光线会在大部分时候以线性的方式衰退，直到距离变得足够大，让二次项超过一次项，光的强度会以更快的速度下降。这样的结果就是，光在近距离时亮度很高，但随着距离变远亮度迅速降低，最后会以更慢的速度减少亮度。

`TRPointLight`的`attenuation`存储了光源的衰减系数，这是一个三维向量，它的x、y和z分量分别对应K_c、K_l和K_q。`TRPointLight`的`lightPos`是光源所在的位置，`lightColor`是光的颜色，在实现光照算法时你将会用到这些参数。本次作业的场景中有三个点光源，分别是红、绿、蓝三种颜色的光源，它们在`main.cpp`中被设置好了，如下所示，你无需改动。

Computer Graphics - SYSU
Prof. Chongying Gao

```
glm::vec3 redLightPos = glm::vec3(0.0f, -0.05f, 1.2f);
glm::vec3 greenLightPos = glm::vec3(0.87f, -0.05f, -0.87f);
glm::vec3 blueLightPos = glm::vec3(-0.83f, -0.05f, -0.83f);
int redLightIndex = renderer->addPointLight(redLightPos, glm::vec3(1.0, 0.7, 1.8), glm::vec3(1.9f, 0.0f, 0.0f));
int greenLightIndex = renderer->addPointLight(greenLightPos, glm::vec3(1.0, 0.7, 1.8), glm::vec3(0.0f, 1.9f, 0.0f));
int blueLightIndex = renderer->addPointLight(blueLightPos, glm::vec3(1.0, 0.7, 1.8), glm::vec3(0.0f, 0.0f, 1.9f));
```

成功构建并编译提供的代码框架，你应该得到如下的运行结果：



接下来，你将一步一步地为这个世界添加绚烂的色彩！

Computer Graphics - SYSU
Prof. Chengying Gao

3、作业描述

本次作业中，请你严格按照下面的顺序完成以下的任务：

Task 1、实现纹理的最邻近采样算法。

你要实现的函数在`TRTexture2D.cpp`文件中，如下所示：

```
glm::vec4 TRTexture2DSampler::textureSampling_nearest(const TRTexture2D &texture, glm::vec2 uv)
{
    unsigned char r = 255, g = 255, b = 255, a = 255;

    //Task1: Implement nearest sampling algorithm for texture sampling
    // Note: You should use texture.readPixel() to read the pixel, and for instance,
    //       use texture.readPixel(25,35,r,g,b,a) to read the pixel in (25, 35).
```

```
    //       But before that, you need to map uv from [0,1]*[0,1] to [0,width-1]*[0,height-1].
    {

    }

    constexpr float denom = 1.0f / 255.0f;
    return glm::vec4(r, g, b, a) * denom;
}
```

参数`texture`是要被采样的纹理，你可以调用它的`readPixel`方法去读取指定位置的像素（传入的位置应该是$[0, width-1] \times [0, height-1]$）。`uv`是二维纹理坐标，取值范围为$[0,1] \times [0,1]$。现在要你实现最邻近采样算法，对坐标进行四舍五入。此任务仅需填充几行代码。（提示：将纹理坐标`uv`从$[0,1] \times [0,1]$映射到$[0, width-1] \times [0, height-1]$，然后再进行坐标舍入）

贴出你的实现结果，简述你是怎么做的。

Task 2、在实现了Task 1的基础上，在片元着色器中实现Phong光照模型。参考链接

你要填充的函数在`TRShadingPipeline.cpp`文件中，如下所示：

```
void TRPhongShadingPipeline::fragmentShader(const VertexData &data, glm::vec4 &fragColor)
{
    fragColor = glm::vec4(0.0f);

    //Fetch the corresponding color
    glm::vec3 amb_color, dif_color, spe_color, glow_color;
    amb_color = dif_color = (m_diffuse_tex_id != -1) ? glm::vec3(texture2D(m_diffuse_tex_id, data.tex)) : m_kd;
    spe_color = (m_specular_tex_id != -1) ? glm::vec3(texture2D(m_specular_tex_id, data.tex)) : m_ks;
    glow_color = (m_glow_tex_id != -1) ? glm::vec3(texture2D(m_glow_tex_id, data.tex)) : m_ke;

    //No lighting
    if (!m_lighting_enable)
    {
        fragColor = glm::vec4(glow_color, 1.0f);
        return;
    }

    //Calculate the lighting
    glm::vec3 fragPos = glm::vec3(data.pos);
    glm::vec3 normal = glm::normalize(data.nor);
    glm::vec3 viewDir = glm::normalize(m_viewer_pos - fragPos);
    for (size_t i = 0; i < m_point_lights.size(); ++i)
    {
        const auto &light = m_point_lights[i];
        glm::vec3 lightDir = glm::normalize(light.lightPos - fragPos);

        glm::vec3 ambient, diffuse, specular;
        float attenuation = 1.0f;

        //Task2: Implement phong lighting algorithm
        // Note: The parameters you should use are described as follow:
```

Computer Graphics - SYSU
A. Prof. Chengying Gao

```
        //        amb_color: the ambient color of the fragment
        //        dif_color: the diffuse color of the fragment
        //        spe_color: the specular color of the fragment
        //          fragPos: the fragment position in world space
        //           normal: the fragment normal in world space
        //          viewDir: viewing direction in world space
        //             m_kd: diffuse coefficient
        //         lightDir: lighting direction in world space
        // light.lightColor: the ambient, diffuse and specular color of light source
        //      m_shininess: specular hightlight exponent coefficient
        //   light.lightPos: the position of the light source
        //light.attenuation: the attenuation coefficients of the light source (x,y,z) -> (constant,linear,quadratic)
        {

        }

        fragColor.x += (ambient.x + diffuse.x + specular.x) * attenuation;
        fragColor.y += (ambient.y + diffuse.y + specular.y) * attenuation;
        fragColor.z += (ambient.z + diffuse.z + specular.z) * attenuation;
    }

    fragColor = glm::vec4(fragColor.x + glow_color.x, fragColor.y + glow_color.y, fragColor.z + glow_color.z, 1.0f);

    //Tone mapping: HDR -> LDR
    //Refs: https://learnopengl.com/Advanced-Lighting/HDR
    {
        glm::vec3 hdrColor(fragColor);
        fragColor.x = 1.0f - glm::exp(-hdrColor.x * 2.0f);
        fragColor.y = 1.0f - glm::exp(-hdrColor.y * 2.0f);
        fragColor.z = 1.0f - glm::exp(-hdrColor.z * 2.0f);
    }
}
```

请在相应的位置实现光照计算的过程。分别贴出环境光效果、漫反射光效果、镜面高光的效果以及最终的整体效果，简述你是怎么做的。**请注意，在实现之前，请在`main.cpp`文件把下面的第三行删掉，把第七行代码反注释：**

```
//Simple texture
//Note: delete this for Task 2
renderer->setShaderPipeline(std::make_shared<TRTextureShadingPipeline>());

//Phong lighting
//Note: Uncomment this for Task 2
//renderer->setShaderPipeline(std::make_shared<TRPhongShadingPipeline>());
```

Task3、在实现了Task 3的基础上，对Phong的高光项进行改进，实现Blinn-Phong光照模型。[参考链接]

此任务仅需做很小的改动，贴出你的实现结果，并简述你是怎么做的。

Computer Graphics - SYSU
A. Prof. Chengying Gao

Task4、纹理的双线性插值纹理过滤采样（选做）。参考链接

在Task 1中，对于纹理采样，我们直接采用四舍五入的方法确定采样的像素值。这种方法在纹理被放大时能够明显看到像素块，看起来非常奇怪。一种更为平滑的方法是采用插值的对采样的区域进行加权混合。请你参考Task 1，实现更为平滑的双线性插值纹理过滤，你需要填充的函数在`TRTexture2D.cpp`文件中，如下所示：

```
glm::vec4 TRTexture2DSampler::textureSampling_bilinear(const TRTexture2D &texture, glm::vec2 uv)
{
    //Note: Delete this line when you try to implement Task 4.
    return textureSampling_nearest(texture, uv);

    //Task4: Implement bilinear sampling algorithm for texture sampling
    // Note: You should use texture.readPixel() to read the pixel, and for instance,
    //       use texture.readPixel(25,35,r,g,b,a) to read the pixel in (25, 35).
}
```

双线性插值实现并不复杂，我们鼓励有兴趣的同学尝试此任务。贴出你的实现结果，并简述你是怎么做的。

Task5、参考`TRPointLight`，实现聚光灯。（选做）参考链接

此任务需要你对框架代码做仔细的了解，不强求。欢迎有兴趣、有能力的同学实现此任务。若完成了此任务，贴出你的实现结果，并简述你是怎么做的。

Computer Graphics - SYSU
Prof. Chengying Gao

4、结语

在Assignment1到Assignment3中，我们要求你在提供的渲染器框架上实现相应的图形算法。相信如果你认真完成这几次作业的要求，应该会对目前基于光栅化的渲染管线有了相当深入的了解。我们的渲染器核心代码并不多，但涵盖了基础的三维渲染特性，我们推荐你仔细阅读代码，鼓励感兴趣的同学们自行尝试添加更多的特性，例如多线程并行加速、MSAA多重采样抗锯齿、纹理Swizzle布局优化、高质量Mipmap纹理、cubemap纹理采样、透明融合、基于MSAA的顺序无关透明、法线贴图、PBR光照算法甚至骨骼动画等等。这些不作为课堂作业强制要求，但是均属于基于光栅化渲染的相关技术，推荐有意向从事相关行业的同学做尝试！可参考这个项目。

注意事项:

- 将作文文档、源代码一起压缩打包，文件命名格式为：学号+姓名+HW3，例如19214044+张三+HW3.zip。
- **提交的文档请提交编译生成的pdf文件，请勿提交markdown、docx以及图片资源等源文件！**
- 提交代码只需提交源文件即可，请勿提交工程文件、中间文件和二进制文件（即删掉**build**目录下的所有文件！）。
- 禁止作业文档抄袭，我们鼓励同学之间相互讨论，但最后每个人应该独立完成。
- 可提交录屏视频作为效果展示（只接收mp4或者gif格式），请注意视频文件不要太大。