

Lab3-Report

王想 2100013146

1 Task 1: Phong Illumination

Q1 顶点着色器从顶点数据中直接接受输入并计算顶点相关数据，片段着色器从顶点着色器中接受顶点相关数据并处理得到最终的颜色值。二者传输数据的流程如下：在顶点着色器中用 `out` 关键字声明了三个输出, `v_Position`, `v_Normal`, `v_TexCoord`, 在片段着色器中用 `in` 关键字声明了三个名字和类型均相同的输入变量，并且设置了一一对应的 `location` 标识符, 这时 OpenGL 就会把两个变量链接到一起, 从而它们之间就可以传输数据了。

Q2 `if(diffuseFactor.a < .2)discard;` 语句的作用是忽略掉透明度小于 0.2 的点的颜色值, 如果用 `if(diffuseFactor.a == 0.)discard;` 意味着只忽略完全透明的点, 这时候会导致对许多透明度很高、本来不应该渲染的点进行了渲染, 会得到不真实的结果。

Implement 根据 Phone Shadding 模型, $I = K_a I_a + f_{att} I_l (k_d \cos \theta + k_s (\cos \phi)^{n_s})$ 由于 main 函数里 total 变量已经加上了环境光照, 并且后续对每一个光源计算时乘上了相应的衰减系数, 所以在 shade 函数里面只需要计算并返回 $I_l (k_d \cos \theta + k_s (\cos \phi)^{n_s})$ 这一部分的值, 将参数对应代入到公式即可完成。这里 Phone 与 Blinn-Phone 光照模型的区别在于 $\cos \phi$ 的计算, 在 Phone 光照模型中, $\cos \phi = \max(\dot(\text{viewDir}, \text{reflect}(-\text{lightDir}, \text{normal})), 0.0)$; 在 Blinn-Phone 光照模型中, $\cos \phi = \max(\dot(\text{normal}, \text{normalize}(\text{lightDir} + \text{viewDir})), 0.0)$; 最终效果如图

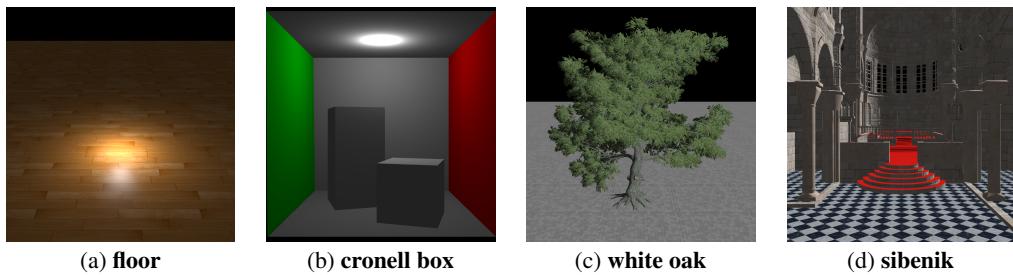


图 1: Phone

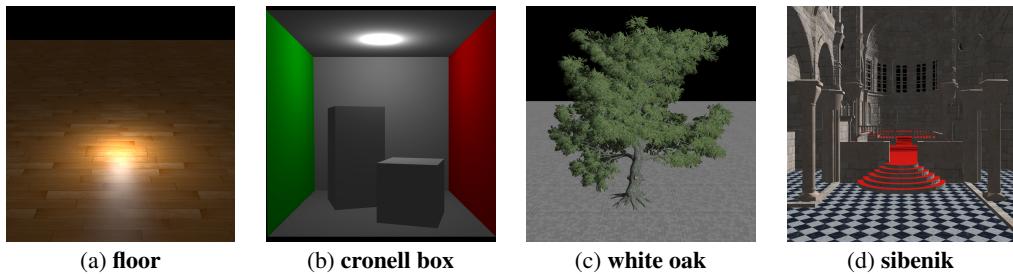


图 2: Blinn Phone

Bump Mapping 凹凸映射的实现参考了 `tutori` 中提供的参考资料, 对资料中提供的代码进行了相应的参数调整迁移到本 Lab 后得到效果如图, 一定程度的 bump mapping 确实能够使渲染效果变得更加真实, 可以看到当 `bump=25` 时效果最真实, 并非 bump 比例越高越好。

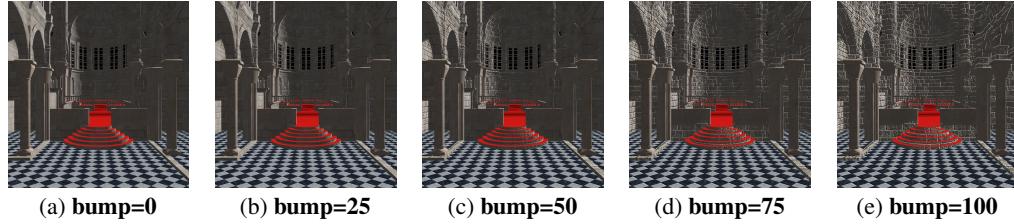


图 3: Bump Mapping

2 Task 2: Environment Mapping

Implement 此函数的实现参考了 `tutori` 中提供的参考资料, 对资料中提供的代码进行了相应的参数调整迁移到本 Lab 后得到效果如图

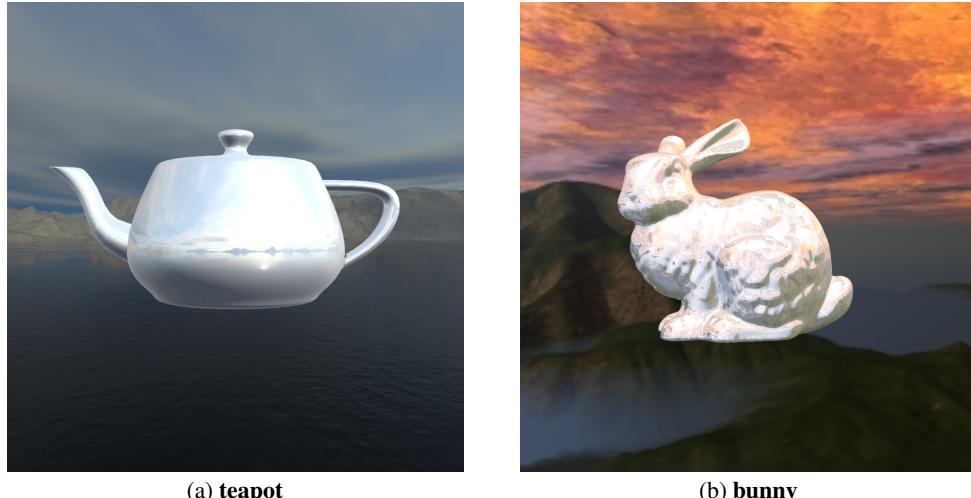


图 4: Environment Mapping

3 Task 3: Non-Photorealistic Rendering

Q1 源代码中是先利用 `glEnable(GL_CULL_FACE)` 指令开启面剔除功能, 之后用 `glCullFace(GL_FRONT)` 指令剔除掉模型的正面, 此时单独对模型的背面进行渲染。再用 `glCullFace(GL_BACK)` 指令剔除模型的背面, 并且用 `glEnable(GL_DEPTH_TEST)` 开启深度测试, 只渲染最靠前的像素点, 此时是单独对模型的正面进行渲染, 在渲染完成后关闭深度测试与面剔除功能。

Q2 因为如果只是简单将每个顶点在世界坐标中沿着法向移动一些距离来实现轮廓线的渲染, 会导致程序无法泛化到不同的显示屏上, 不同的显示屏的长宽都不一样, 如果只是简单移动固定距离就会导致相对位移量因屏幕改变而改变, 进而在某些屏幕上可能呈现出模型外侧轮廓线“悬空”的现象。

Implement 根据公式 $I = rate * k_{cool} + (1 - rate * k_{warm})$, 其中 $rate = (1 + dot(lightDir, normal)) / 2$; 代入公式计算即可实现函数。最终效果如图

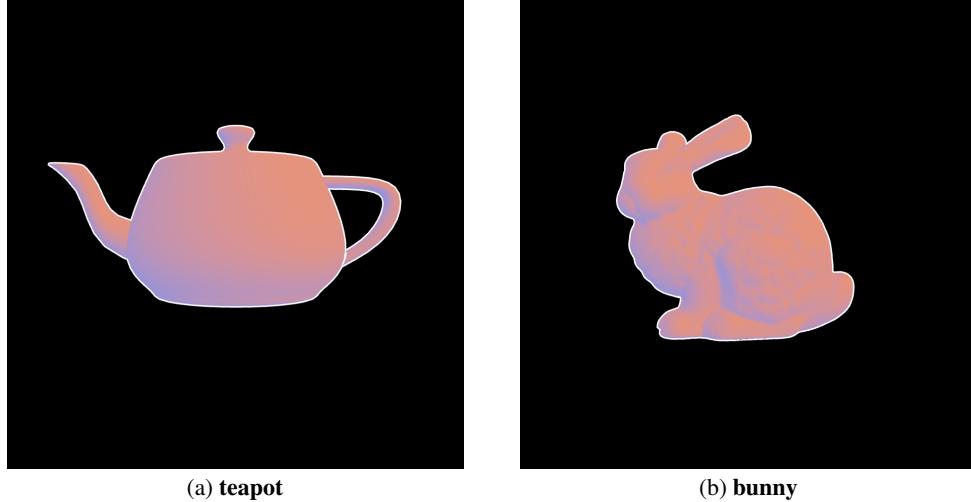


图 5: Non-Photorealistic Rendering

但这种连续性的变化或许不一定符合二维风格, 我们可以设置几个区间, 当 rate 位于某个区间中就将 rate 分别设置成该区间中的某个定值, 呈现出有序的层次感, 本次设置全部取区间的均值, 将 $[0, 1]$ 区间划分成 num 个区间的效果如下:

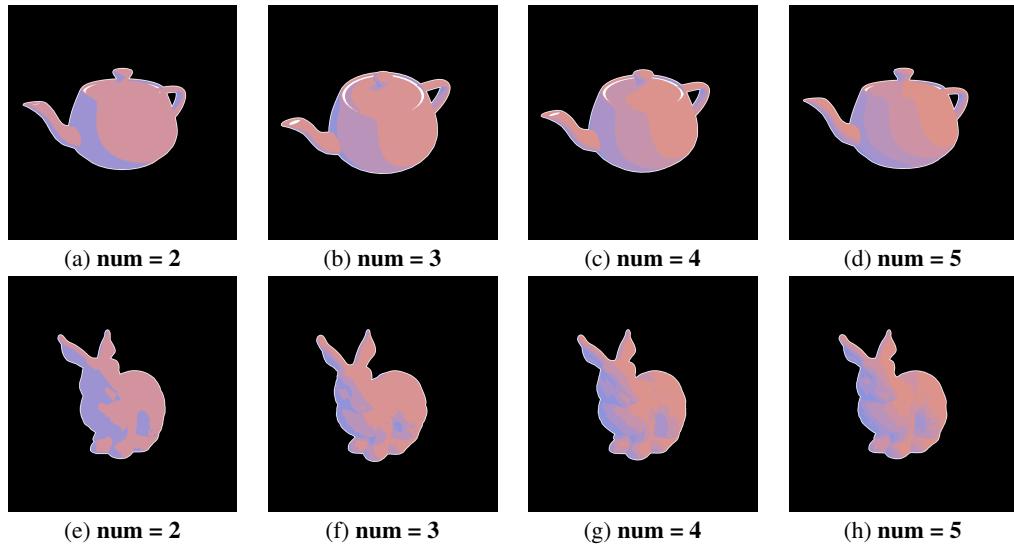


图 6: New Style

4 Task 4: Shadow Mapping

Q1 有向光源应该用正交投影矩阵, 点光源应该用透视投影矩阵。因为正交投影矩阵并不会将场景用透视图进行变形, 所有视线/光线都是平行的, 这些正是有向光源的特点; 而透视投影矩阵会将所有顶点根据透视关系进行变形, 投影到一个点上, 这正是点光源的特点。

Q2 利用齐次坐标 (homogeneous coordinate) 和投影矩阵相乘将空间位置坐标转化成透视图上的坐标, 再将坐标进行一次透视线除法 (xyz 分量分别除以 w 分量), 之后归一化到 $[0, 1]$ 范围上, 此时就可以直接取坐标的 z 分量与深度贴图的深度进行比较。公式结果的正确性可以作图并利用三角形相似进行推导。

Implement 此函数的实现参照了 tutori 中提供的参考资料, shade 函数中采取的是计算更快的 Blinn-Phone 光照模型, 进一步对资料中提供的代码进行了相应的参数调整迁移到本 Lab 后得到效果如图

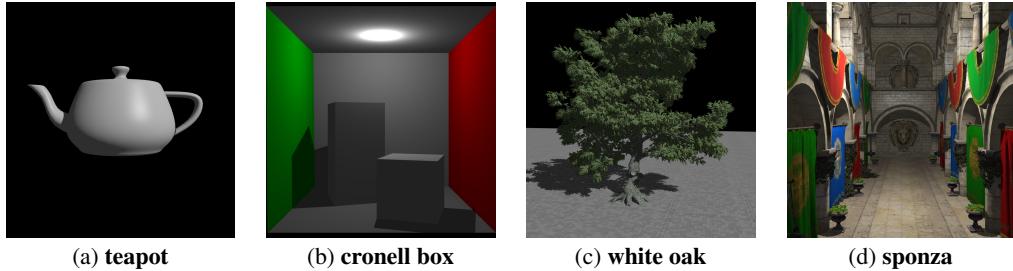


图 7: Shaddow Mapping

5 Task 5: Whitted-Style Ray Tracing

Q1 在 cronell box 场景中, 光线追踪可以实现类似镜子一样的效果, 但光栅化实现不了, 这是因为光栅化渲染无法考虑到全局的影响, 现实生活中的光会经过多次反射折射, 但光栅化渲染只会考虑一次, 导致无法实现类似现实中的镜面的正确渲染效果。光线追踪的渲染结果更加真实, 光栅化渲染显得不那么真实。

Implement 光线求交算法参照第一个链接资料中的算法 (Fast, minimum storage ray/triangle intersection) 实现, 其中判断平行的阈值采用 task.h 中定义的 EPS2. RayTrace 函数中的着色部分参照老师所讲的算法, 处于阴影中的部分就只加上环境光的颜色, 其他部分还要加上漫反射和高光效果, 为了加快计算, 这里采用 Blinn-Phone 模型。具体实现时, 先加上环境光照, 再遍历所有光源, 如果发现该点在此光源的阴影下就 continue, 不加上此光源贡献的光照颜色。判断是否在阴影中则是通过从光源到物体射出一跳光线, 如果中途碰到其他物体且该物体透明度大于或等于 0.2, 此时判断在阴影中; 否则, 这条光线继续向物体前进, 直到击中物体, 若仍未碰到其他不透明物体, 表明物体不再阴影中, 此时加上漫反射和高光, 然后继续遍历光源。

在附加题中, 我采用的是 BVH 结构进行加速, 在建树时采用 Surface Area Heuristic 算法进一步提升算法性能, 在光线与包围盒求交时采取 Slabs 算法, 光线与三角形求交则调用前面实现的函数 IntersectTriangle。在实现 BVH 树的过程中附加了一个辅助数据结构 Triangle, 其中只有两个 int 变量和一个 vec3 变量, 用来记录此三角形对应的 model 编号和 mesh 编号, vec3 变量记录三角形的中心用于 SAH 建树。BVH 树的节点数据结构 BVHNode 中含两个 BVHNode 指针, 分别指向左右子树, 节点还包括两个向量, 分别记录包围盒的左下和右上坐标。IntersectRay 函数主体框架仍采用本实验给出的 TrivialRayIntersector 的函数, 但是在求交时使用的是 BVH 树的 hitBVH 函数进行加速。最终结果如下。

可以看到, 随着采样率的提升, 锯齿效果逐渐减少, 到采样率为 3 就基本没有锯齿效果了, 然而, 采样率提高后渲染所需要的时间大幅度增加, 采样率为 3 时需要接近 24 分钟才可以渲染出来。随着反射次数的增加, 渲染效果越来越真实, 尤其是树这一组较为明显, 最初树叶间隙偏黑, 这显然不太符合现实, 但随着反射次数增加, 树叶渲染得越来越真实, 这是因为现实中光也是经历了数次反射, 反射次数的增加也会造成渲染时间相应增加, 但增长不快, 因为大多数光反射几次之后就不再击中模型了。

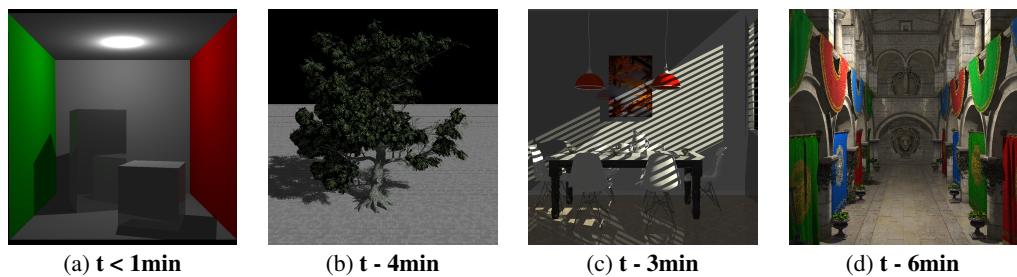
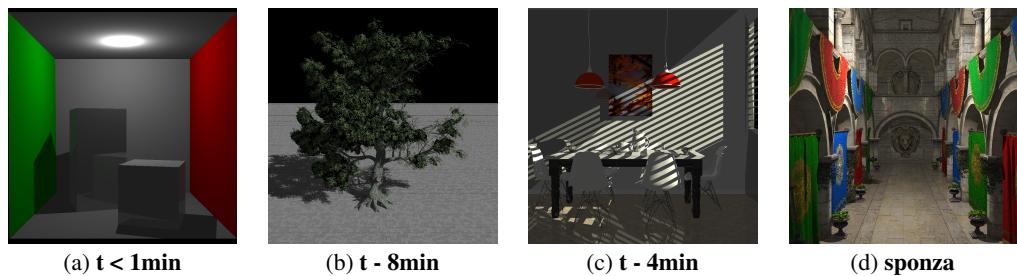
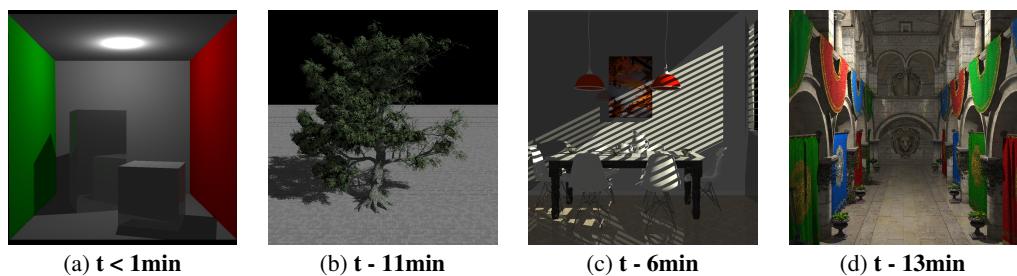
图 8: $\text{max depth} = 3$ 图 9: $t - 10\text{min}$ 图 10: $\text{max depth} = 7$ 

图 11: subsampling