

**《Web前端开发》创新实践训练项目**

**期中项目报告**

**基于Three.js实现简单的PBR/NPR渲染**

**姓 名：**

**李钦俊、吴炫煜**

**学 号：**

**2018213258、2018213286**

**指导教师： 高立/兰名荥**

**北京邮电大学数字媒体与设计艺术学院**

**2020年11月2日**

**基于Three.js实现简单的PBR/NPR渲染**

**期中项目摘要**

学习基于物理的渲染的相关知识，以金属质地工作流的PBR方案为基础，选择Cook-Torrance BRDF，在期中在网页实现了直接光照显示的渲染效果，并可调整参数在线查看效果。

学习非真实感绘制的相关知识，期中在网页实现卡通渲染的Cel Shading形式，并可调整参数在线查看效果。

该项目前端基于HTML、CSS、JS技术，结合Three.js这一JavaScript编写的WebGL第三方库，结合GLSL语言编写着色器，结合WebStrom进行开发实现。

**关键词：**Web前端开发，Three.js，实时渲染，PBR，Cook-Torrance BRDF，NPR，Cel Shading，着色器，GLSL

# 一 引言

## 项目简介

在该项目中，组员学习渲染相关知识与图形学基本理论，利用three.js作为前端的实现工具，预计最终实现基于物理的渲染(Physically Based Rendering，PBR)中的基于图像的光照(Image based lighting, IBL)，以及探索非真实感绘制(Non-photorealistic rendering，NPR)中卡通渲染（Toon Shading），在网页上进行参数调整与效果展示。

在该项目的实现过程中，组员掌握对应的渲染相关知识，积累实践经验，并与前端知识进行结合展示。

该项目也可作为图形学与渲染相关的可视化教学工具。

在期中的进展中，PBR部分学习了PBR理论，实现了金属质地工作流的PBR方案中的直接光照形式，NPR部分实现轮廓线效果与的Cel Shading形式。

## 背景需求

### 真实感渲染

图形学重要的分支之一是真实感渲染，真实感渲染当中基于物理的渲染(Physically Based Rendering，PBR)是非常重要的。

PBR指的是由与基于现实世界的物理原理更相符的基本的理论所构成的一些渲染技术的集合。

其目的是为了使用一种更符合物理学规律的方式来模拟光线，这种渲染方式与原来的经验光照算法Phong或者Blinn-Phong算法相比总体上看起来要更真实些。

除了看起来更好些以外，由于它与物理性质非常接近，因此美术设计者可以直接以物理参数为依据来编写表面材质，而不必依靠修改与调整来让光照效果看上去正常。

使用基于物理参数的方法来编写材质还有个好处就是不论光照条件如何，这些材质看上去都会是正确的，而在非PBR的渲染管线当中有些东西就不会那么真实了。

PBR渲染管线所需要的每一个表面参数都可以用纹理来定义或者建模。通过使用纹理，我们可以逐片段的来控制每个表面上特定的点对于光线是如何响应的。

这是PBR渲染管线最大的优势之一，因为不论环境或者光照的设置如何改变这些表面的性质是不会改变的，这使得美术设计者们可以更便捷的获取物理可信的结果。

在PBR渲染管线中编写的表面可以非常方便的在不同的PBR渲染引擎间共享使用，不论处于何种环境中它们看上去都会是正确的，因此看上去也会更自然。

这也是我们要学习掌握它的原因。

### 非真实感渲染

卡通渲染也是图形学中一个热门的话题，属于图形学中非真实感渲染（NPR）的范畴，在NPR领域中，其也最多地被应用到实际游戏中，近年来流行的《守望先锋》，《英雄联盟》，《DOTA2》，《崩坏3》等游戏中都或多或少地出现过卡通渲染的身影。

卡通渲染最关键的特征包括不同于真实感渲染的艺术化光影效果和描边。

在正常的渲染着色过程中，由于使用了真实的光照模型，在模型上的颜色是随着光照的角度渐变的，而卡通渲染为了模拟动画绘制时候的风格，根据每个像素的法线和光照的方向的关系，来使得这片区域的像素投影到其中一个明暗区域上面，达到多段离散的明暗区域的效果。

我们选择实现的根据亮度分为若干颜色梯度的效果，是卡通渲染的一种简单形式，称为 Cel Shading。

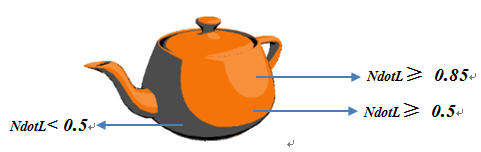


图1-2 卡通渲染示例

## 市场同类产品分析

### 同类产品特点

基本上研究渲染效果会选择在unity或者unreal这样的商业软件中，它们不只是引擎，也提供了一个团队如何开发协作的流程和各种设施，是一套完整的工具链，完善且庞大，也可以选择使用图形API进行搭建。

### 同类产品不足

在商业引擎中不太方便效果展示，使用底层的图形API写也比较繁杂，需要环境准备。

## 项目可能的创新点

### 理念创新

Three.js这种（JavaScript 3D library），没什么工具链，只是抽象了一层API，比底层API比也是简化了开发。

Three.js 相对UE4,U3D来说会轻量很多，所以加载更容易做到更快。

在我们这样的简单项目中，使用网页形式展现效果可以说很方便直观了。

### 内容创新

将前端技术与图形学的知识进行结合，做到方便展示。

# 二 项目总体架构设计

## 2.1 项目总体架构

学习相关理论知识→使用Three.js编写着色器与其他代码→网页测试

### 2.1.1 项目原型设计

界面简洁为主，该项目主要展示的是制作者对图形渲染的技术的掌握程度。

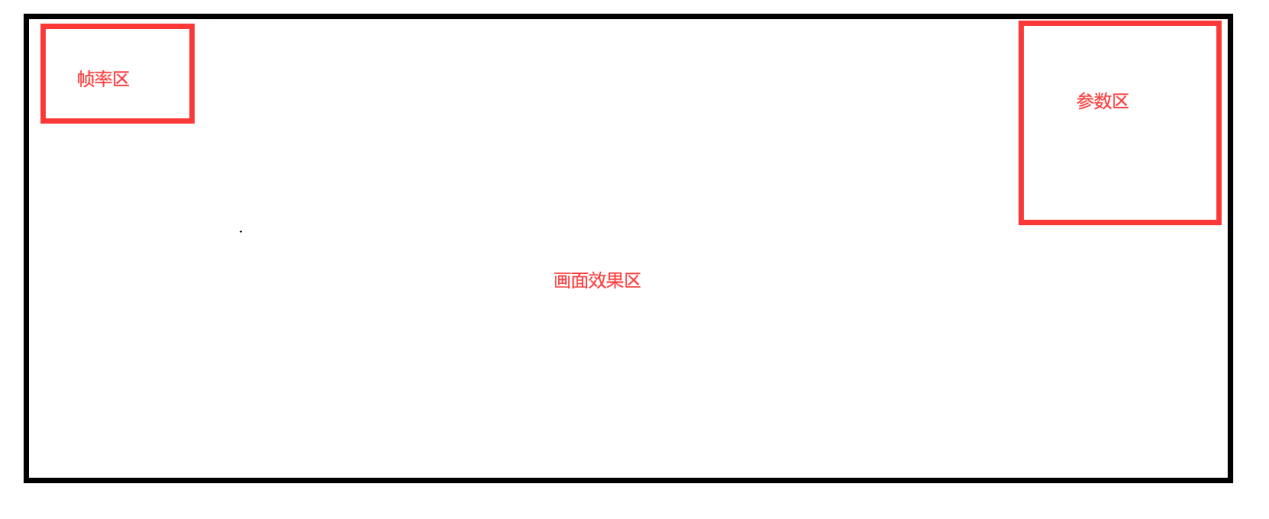


图 2‑1 原型设计（样例）

以Three.js的example的界面设计风格为主，中央为展示区，上方为参数区

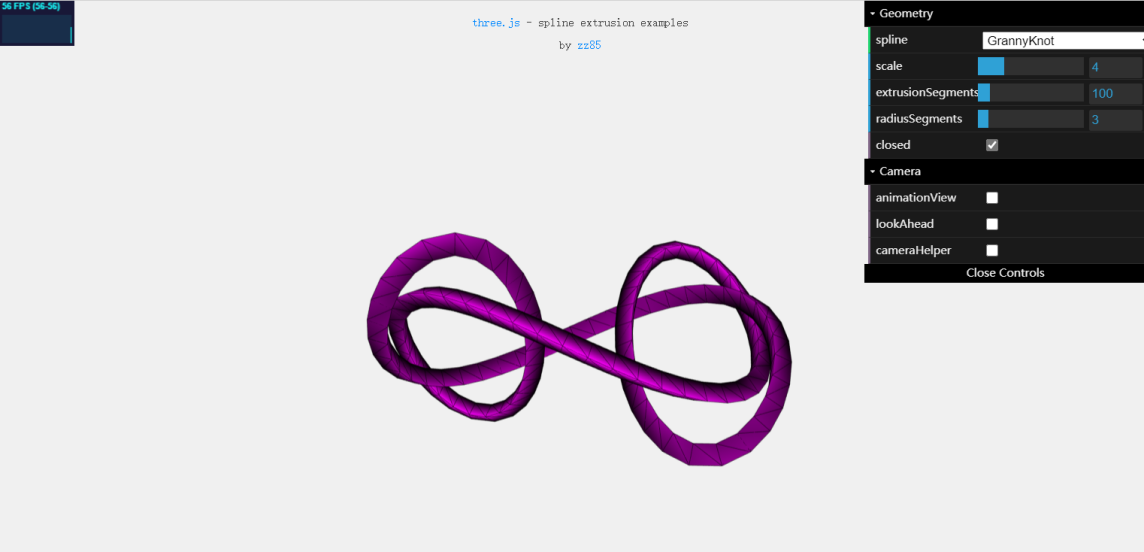


图2-2 页面结构示例

### 2.1.2 项目前端技术框架

该项目前端基于HTML、CSS、JS技术，结合Three.js这一JavaScript编写的WebGL第三方库，结合GLSL语言编写着色器，结合WebStrom进行开发实现。

## 2.2 总方案设计

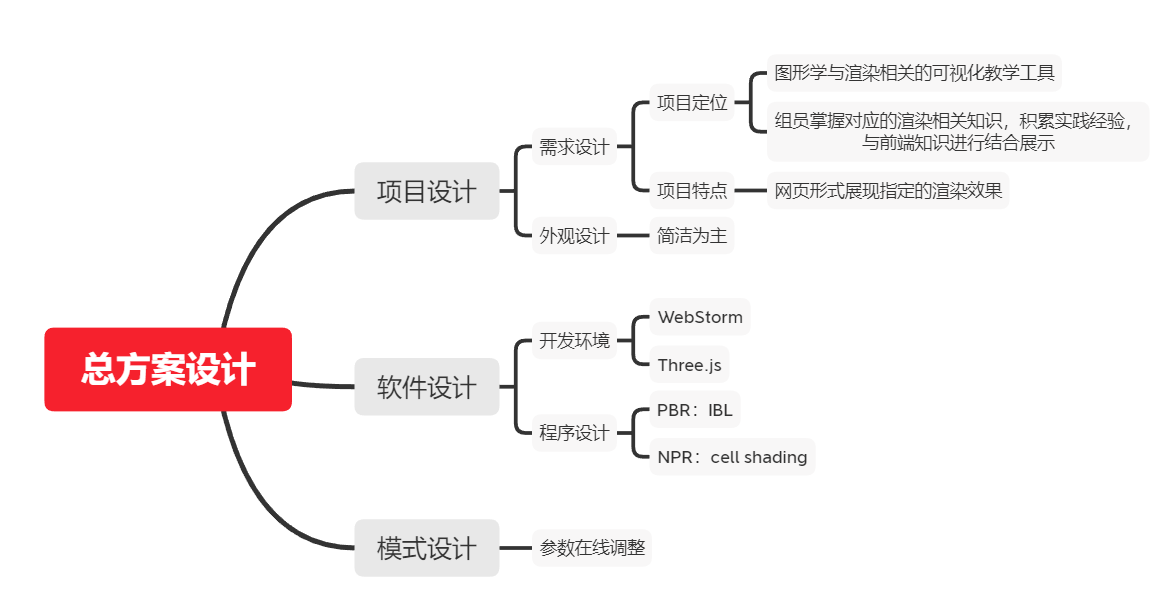


图 2‑3 总方案设计架构图

# 三 真实感渲染部分

## 3.1 原理

### 3.1.1 引入

在背景需求部分对PBR进行介绍后，下面介绍其原理。

判断一种PBR光照模型是否是基于物理的，必须满足以下三个条件：

1.基于微平面(Microfacet)的表面模型。

2.能量守恒。

3.应用基于物理的BRDF。

在这次的实现与研究之中，选择的PBR方案是最先由迪士尼(Disney)提出探讨并被Epic Games首先应用于实时渲染的基于金属质地工作流(Metallic Workflow)的方案，其广泛应用于各种流行的引擎之中并且有着很好的视觉效果。

### 3.1.2 微表面模型

所有的PBR技术都基于微平面理论/微表面模型。

这项理论认为，达到微观尺度之后任何平面，都可以用被称为微平面(Microfacets)的细小镜面来进行描绘，也就是没有任何平面是完全光滑的。

产生的效果就是：一个平面越是粗糙，这个平面上的微平面的排列就越混乱。

这些微小镜面无序取向排列的影响在当我们特指镜面反射时会很明显：



图3-1 微表面示例

·对于一个粗糙的表面，入射光线更趋向于向完全不同的方向发散(Scatter)开来，进而产生出分布范围更广泛的镜面反射。

·对于一个光滑的平面，光线大体上会更趋向于向同一个方向反射，造成更小更锐利的反射。

然而由于这些微平面已经微小到无法逐像素的继续对其进行区分，因此我们会假设一个粗糙度(Roughness)参数，然后用统计学的方法来估算微平面的粗糙程度。

我们可以基于平面的粗糙度来计算出某个向量的方向与微平面平均取向方向一致的概率。

这个向量便是位于光线向量l和视线向量v之间的中间向量(Halfway Vector)。



微平面的取向方向与中间向量的方向越是一致，镜面反射的效果就越是强烈锐利。

于是我们设定一个介于0到1之间的粗糙度参数，这样我们就能概略的估算微平面的取向情况了。

### 3.1.3 能量守恒

微平面近似的过程中使用了这样一种形式的能量守恒(Energy Conservation)：除了发光面，其他面出射光线的能量永远不能超过入射光线的能量。

为了遵守能量守恒定律，我们需要对漫反射光和镜面反射光之间做出明确的区分。

我们知道当一束光线碰撞到一个表面的时候，它就会分离成一个折射部分和一个反射部分。

·反射部分即为直接反射开来而不会进入平面的那部分光线，也就是我们所说的镜面光照。

·折射部分即为余下的会进入表面并被吸收的那部分光线，也就是我们所说的漫反射光照。

我们也知道因为当光线接触到一个表面的时候，折射光是不会立即就被吸收的。通过物理学常识，光线可以被认为是一束没有耗尽就不停向前运动的能量，而光束是通过碰撞的方式来消耗能量。

每一种材料都是由无数微小的粒子所组成，这些粒子都能如下图所示一样与光线发生碰撞。粒子在每次的碰撞中都可以吸收光线所携带的一部分或者是全部的能量而后转变成为热量。光线会在内部直至能量完全耗尽或者再次离开这个表面

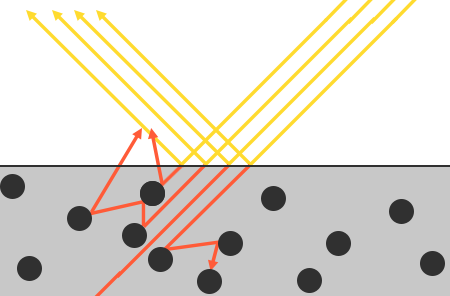


图3-2 次表面散射示例

光线脱离物体表面后将会一起帮助构成该表面的漫反射颜色，加入反射光，不过这个过程在这次的期中项目中进行了简化，我们假设进入物体的光会被完全吸收，也就是不发生次表面散射。

同时对于金属(Metallic)的表面，在讲反射与折射的时候还有一个细节需要注意。金属表面对光的反应与非金属材料（电介质(Dielectrics)材料）表面相比是不同的。

它们遵从的反射与折射原理是相同的，但是对于金属来说所有的折射光都会被直接吸收而不会散开，只留下反射光。金属与电介质之间存在明显的区别，所以它们两者在PBR渲染管线中被区别处理，之后也会提到。

基于能量守恒的思想，也就是比如我们首先计算了镜面反射部分（百分比表示反射比例），然后折射光部分就可以直接由镜面反射部分计算得出，比如以下伪代码：

float kS = calculateSpecularComponent(...); // 反射部分—镜面

float kD = 1.0 - ks; // 折射部分—漫反射

按照这种方法折射与反射所占的份额都不会超过1.0，如此就能保证它们的能量总和永远不会超过入射光线的能量。这是传统经验光照模型没有考虑的问题。

### 3.1.4 反射方程相关概念

在这里我们要提下渲染方程(Render Equation)，它是一个精妙的方程式，是如今我们所拥有的用来模拟光的视觉效果最好的模型。

基于物理的渲染所遵循的是被称为反射方程(The Reflectance Equation)的方程，其是渲染方程的特化版本。

为了引出这个方程以及和代码结合，我们需要拆解一下方程里的相关概念。

#### 3.1.3.1 辐射度量学相关概念

要正确的引出这个方程式，需要稍微涉及一些辐射度量学(Radiometry)的内容。

辐射度量学是一种用来度量电磁场辐射（包括可见光）的方法。

首先要知道的其中的**辐射通量(Radiant Flux)**

辐射通量Φ表示的是一个光源所输出的能量，以瓦特为单位。

我们知道一束光是由多种不同波长的能量所集合而成的，而每种波长则与一种特定的可见的颜色相关。

因此一个光源所放射出来的能量可以被视作这个光源包含的所有各种波长的能量的一个函数。

下面的图片，展示了日光中不同波长的光所具有的能量

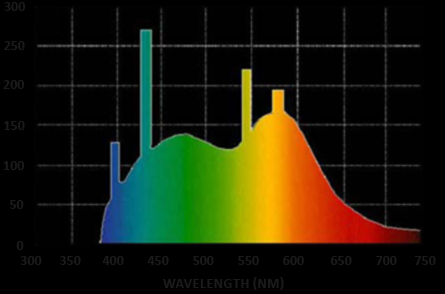


图3-3 光的能量在不同波长的分布

辐射通量将会计算这个由不同波长构成的函数的总面积。在实现上我们通常不直接使用波长的强度而是使用RGB颜色来作为辐射通量表示的简化。

接下来是**辐射强度(Radiant Intensity)，**其定义为在单位时间内，往每个立体角（Solid Angle）上辐射出的能量

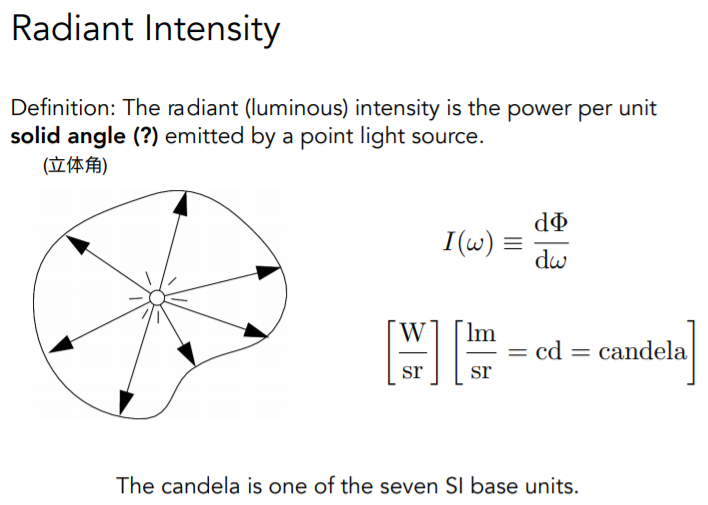


图3-4 立体角

接下来是**入射辐照度(Irradiance)**，定义为在单位时间内，每个单位面积上接受到的光照的能量即power per unit area**。**

· 单位面积要和光线垂直，如果不垂直的话要把面积投影到垂直的方向上（图中没有写cos，暂时就一个dA）

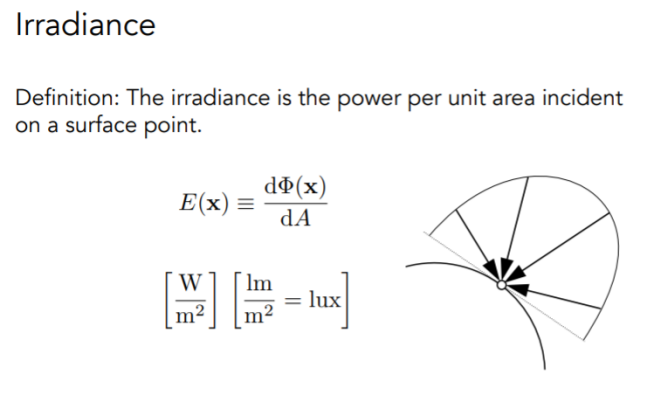


图3-5 Irradiance

投影方法也就是使用Lambert’s余弦定理，cosθ就直接对应于光线的方向向量和平面法向量的点积：

float cosTheta = dot(lightDir, N);

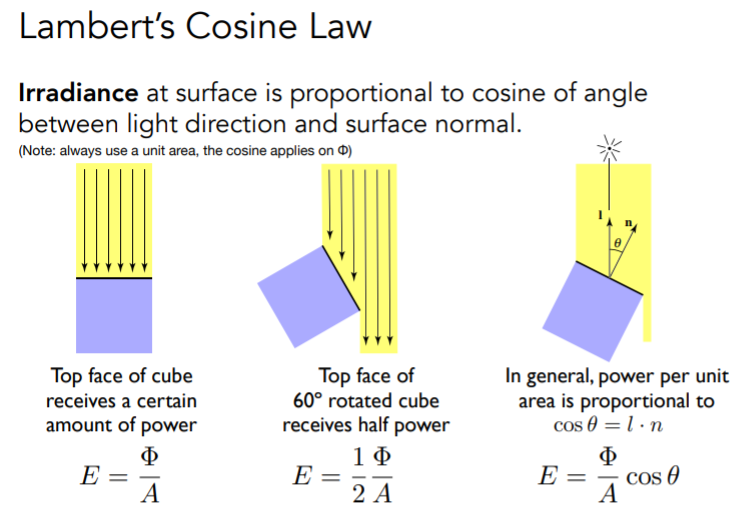


图3-6 Lambert’s Cosine Law

接下来是**出射辐射度(Radiance)**，它描述环境中光的分布的基本场量，主要用来准确描述光线的一些属性，Radiance也就是我们的反射方程要算的东西

Radiance定义：The radiance (luminance) is the power emitted, reflected, transmitted or received by a surface, per unit solid angle, per projected unit area.

某个单位面往某个单位立体角方向上的发出的能量

下图表示一个拥有辐射强度Φ的光源在单位面积A，单位立体角ω上的辐射出的总能量

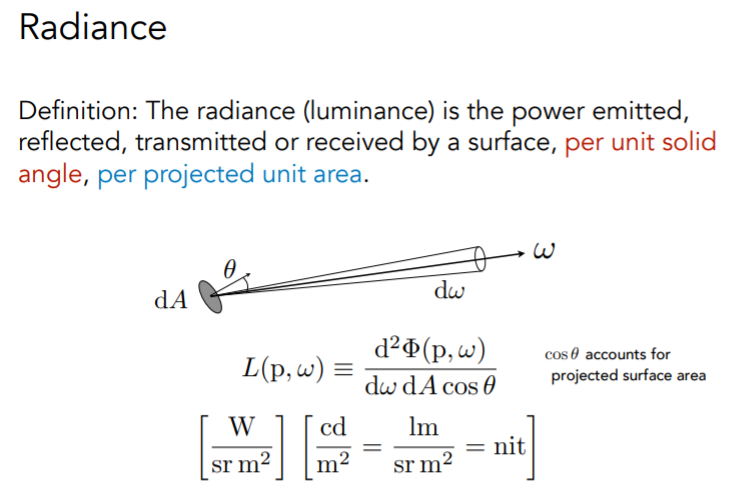


图3-7 Radiance

·如果我们把立体角ω和面积A看作是无穷小的，那么我们就能用Radiance来表示单束光线穿过空间中的一个点的通量，我们实际上把立体角ω转变为方向向量ω，然后把面A转换为点p，这样可以直接在我们的着色器中使用。

#### 3.1.3.2 双向反射分布函数(BRDF)

我们需要有一个函数来描述这种性质：从某个方向进来的能量，并且反射到某个方向去后它的能量是多少。

我们把反射过程看作是这样：从某一个方向来的光线，到达物体表面后，被物体表面所吸收，然后再由物体表面发射到其他方向去。

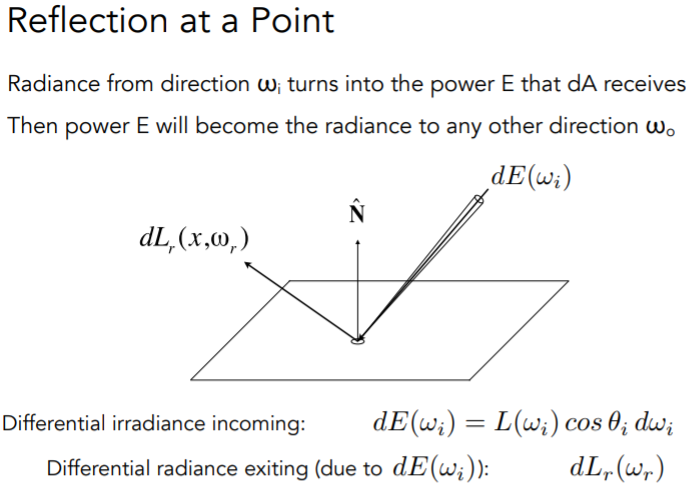


图3-8 点反射

·dE(wi)表示：从wi方向射来的Radiance在到达反射点时，被吸收转化，然后得到的Irradiance

·d L ( x , wr ) )则表示反射点向wr方向反射出去的Radiance

·我们知道反射点要吸收来自某一个方向的Radiance把它变成Irradiance，之后要反射到四面八方的立体角Radiance上。但是我们不知道反射到wr的方向上的Radiance会有多少能量

·所以我们就为此定义一个函数，计算出：考虑一个微小的面积dA，从某一个微小的立体角dw上接受得到到的Irradiance，会如何被分配到各个立体角上。其实求得是一个比例：对于任何一个立体角上发出去的 Radiance）/（单位面积dA接收到的Irradiance）

这就是BRDF的定义，它会告诉我们如何把表面上收集到的能量以某一种比例放射到其他的方向上去。

忽略推导部分，其实BRDF就是描述了物体和光线之间的相互作用。

正是由于BRDF的这种概念，决定了物体不同的材质到底是怎么一回事，也就是BRDF定义了材质。

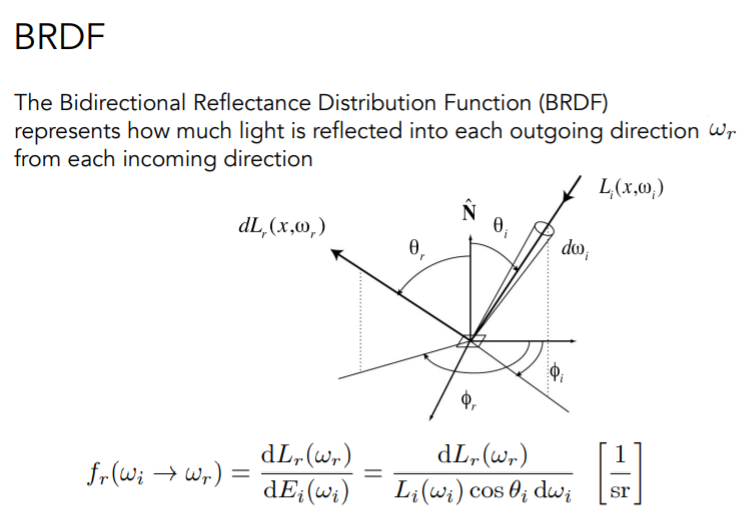


图3-9 BRDF

#### 3.1.3.3 反射方程

从上面的分析可以得到，dLr (wr)是从wi角度入射的能量最后反射到wr方向上的能量。

而在反射的过程，我们并不是只对单一的，来自某一个方向的能量进行反射，而是要将来自四面八方的能量都收集起来，然后反射到某一个角度上。

所以要得到最终的反射效果，应该是：

通过BRDF计算出：每个方向接受的能量 它反射到wr方向上的能量是多少，然后把这些能量全部累加起来，得到的就是wr方向上反射的能量，可以用积分表示

（下面的整个表达式就是反射方程，H的平方表示的积分面积为整个半球面）

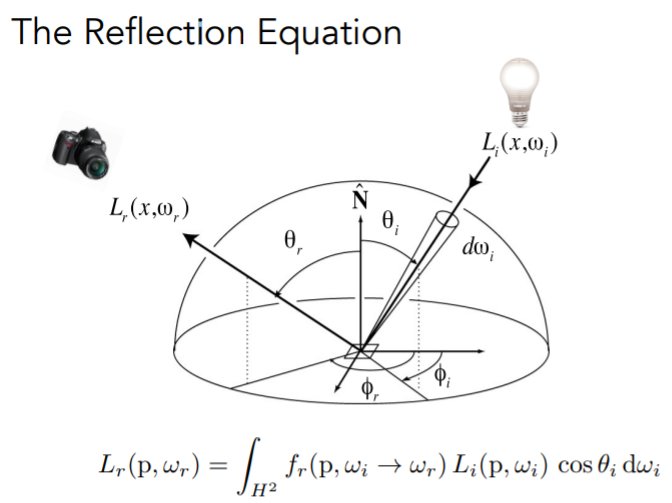


图3-10反射方程

把角度换成向量点积也就是这样：



基于反射方程是围绕所有入射辐照度(Irradiance)的总和来计算的，所以我们需要计算的就不只是是单一的一个方向上的入射光，而是一个以点p为球心的半球领域Ω内所有方向上的入射光。

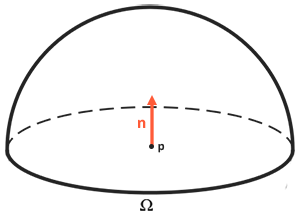


图3-11 半球

积分运算的计算结果要么是解析解要么就是数值解。

由于渲染方程和反射率方程都没有解析解，我们将会用离散的方法来求得这个积分的数值解。这个问题就转化为，在半球领域Ω中按一定的步长将反射方程分散求解，然后再按照步长大小将所得到的结果平均化。这种方法被称为黎曼和(Riemann sum) ，我们可以用下面的伪代码粗略的演示一下：

int steps = 100;

float sum = 0.0f;

vec3 P = ...;//点的位置

vec3 Wo = ...;//出射方向

vec3 N = ...;//法向量

float dW = 1.0f / steps;

for(int i = 0; i < steps; ++i)

{

vec3 Wi = getNextIncomingLightDir(i);

sum += Fr(p, Wi, Wo) \* L(p, Wi) \* dot(N, Wi) \* dW;

}

入射辐照度(Irradiance)可以由光源处获得，此外还可以利用一个环境贴图来测算所有入射方向上的入射辐照度(Irradiance)，不过期中项目目前先不考虑除了光源以外的光。

### 3.1.5 Cook-Torrance反射方程

BRDF接受光方向ωi，观察方向ωo，平面法线n以及一个用来表示微平面粗糙程度的参数a作为函数的输入参数。

其可以近似的求出每束光线对一个给定了材质属性的平面上最终反射出来的光线所作出的贡献程度。

现在已经有好几种BRDF都能近似的得出物体表面对于光的反应，但是几乎所有实时渲染管线使用的都是一种被称为Cook-Torrance BRDF模型。

Cook-Torrance BRDF兼有漫反射和镜面反射两个部分：



·kd是入射光线中被折射部分的能量所占的比率，而ks是被反射部分的比率

BRDF的左侧表示的是漫反射部分，它被称为Lambertian漫反射，用如下的公式来表示：

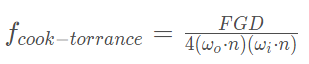


c表示表面颜色，除以π是为了对漫反射光进行标准化，因为前面含有BRDF的积分方程是受π影响的。

目前存在着许多不同类型的模型来实现BRDF的漫反射部分，大多看上去都相当真实，但是相应的运算开销也非常的昂贵。

不过按照EpicGames给出的结论，Lambertian漫反射模型已经足够应对大多数实时渲染。

BRDF的镜面反射部分要稍微复杂一些，它的形式如下所示：



Cook-Torrance BRDF的镜面反射部分包含三个函数，此外分母部分还有一个标准化因子 。

字母D，F与G分别代表着一种类型的函数，各个函数分别用来近似的计算出表面反射特性的一个特定部分。

三个函数分别为

法线分布函数(Normal **D**istribution Function)：估算在受到表面粗糙度的影响下，取向方向与中间向量一致的微平面的数量。这是用来估算微平面的主要函数。

几何函数(**G**eometry Function)：描述了微平面自成阴影的属性。当一个平面相对比较粗糙的时候，平面表面上的微平面有可能挡住其他的微平面从而减少表面所反射的光线。

菲涅尔方程(**F**resnel Rquation)：菲涅尔方程描述的是在不同的表面角下，表面所反射的光线所占的比率。

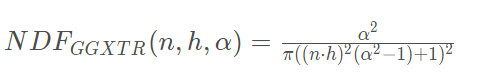
以上的每一种函数是用来估算相应的物理参数的，而且用来实现相应物理机制的每种函数都有不止一种形式。

我们将会采用Epic Games在Unreal Engine 4中所使用的函数，逐个介绍

·法线分布函数D：

法线分布函数D，从统计学上近似的表示了与半程向量h取向一致的微平面的比率。

举例来说，假设给定向量h，如果我们的微平面中有35%与向量h取向一致，则法线分布函数或者说NDF将会返回0.35。



在这里h表示用来与平面上微平面做比较用的中间向量，而a表示表面粗糙度。

·几何函数G：

几何函数G从统计学上近似的求得了微平面间相互遮蔽的比率，这种相互遮蔽会损耗光线的能量。

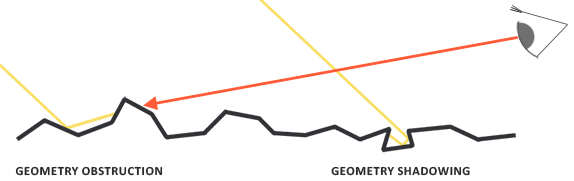


图3-12 几何函数

几何函数采用一个材料的粗糙度参数作为输入参数，粗糙度较高的表面其微平面间相互遮蔽的概率就越高。

我们将要使用的几何函数是GGX与Schlick-Beckmann近似的结合体，因此又称为Schlick-GGX：



K来自粗糙度参数α，k根据几何函数是针对直接光照还是针对IBL光照，进行了重映射(Remapping) (这里只展示了直接光的重映射):



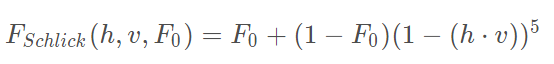
为了有效的估算几何部分，需要考虑两个方向观察方向（几何遮蔽(Geometry Obstruction)），光线方向（几何阴影(Geometry Shadowing)），我们使用史密斯法(Schlick-GGX作为G\_sub)



·菲涅尔方程F：

菲涅尔方程描述的是，被反射的光线对比光线被折射的部分所占的比率，这个比率会随着我们观察的角度不同而不同，利用这个反射比率和能量守恒原则，我们可以直接得出光线被折射的部分以及光线剩余的能量。

菲涅尔方程是一个相当复杂的方程式，不过可以用Fresnel-Schlick近似法求得近似解：



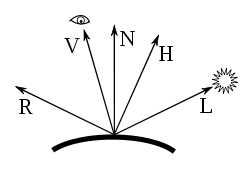


图3-13 向量

F0表示平面的基础反射率，是利用折射指数(Indices of Refraction)计算得出。

但是Fresnel-Schlick近似仅仅对电介质或者说非金属表面有定义。对于导体(Conductor)表面，使用它们的折射指数计算基础折射率并不能得出正确的结果，这样我们就需要使用一种不同的菲涅尔方程来对导体表面进行计算。

由于这样很不方便，所以我们预计算平面对于法向入射（F0）的反应。

然后基于相应观察角的Fresnel-Schlick近似对这个值进行插值，用这种方法来进行进一步的估算。这样我们就能对金属和非金属材质使用同一个公式了。

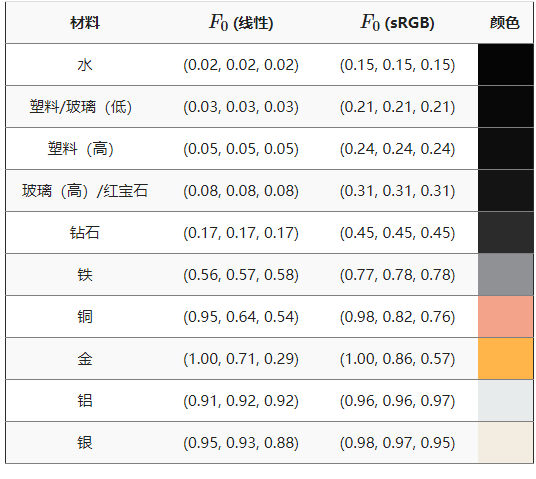


图3-14 F0

对于导体或者金属表面而言基础反射率一般是带有色彩的，这也是为什么F0要用RGB三原色来表示的原因。

也就是我们需要额外使用一个被称为金属度（Metalness）的参数来参与编写表面材质，伪代码如下。

vec3 F0 = vec3(0.04);

F0 = mix(F0, surfaceColor.rgb, metalness);

随着Cook-Torrance BRDF中所有元素都介绍完毕，我们现在可以将基于物理的BRDF纳入到最终的反射方程当中去了，它现在可以认为就是我们一般意义上理解的基于物理的渲染也就是PBR。



PBR渲染管线所需要的每一个表面参数都可以用纹理来定义或者参数指定，这也是程序里要实现的参数。

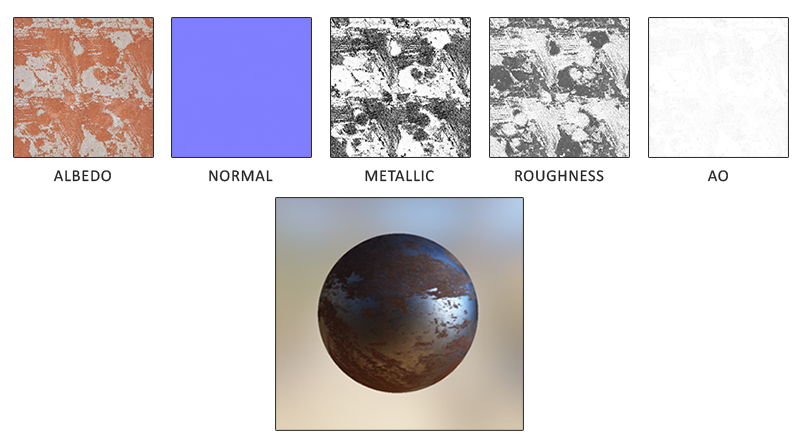


图3-15 PBR材质

·反射率：反射率(Albedo)纹理为每一个金属的纹素(Texel)（纹理像素）指定表面颜色或者基础反射率。

这和传统的diffuse漫反射纹理相当类似，不同的是所有光照信息都是由一个纹理中提取的。

diffuse漫反射纹理的图像当中常常包含一些细小的阴影或者深色的裂纹，而反射率纹理中是不会有这些东西的。它应该只包含表面的颜色（或者折射吸收系数）。

·法线：法线贴图使我们可以逐片段的指定独特的法线，来为表面制造出起伏不平的假象。

·金属度：金属(Metallic)贴图逐个纹素的指定该纹素是不是金属质地的。根据PBR设置的不同，可以将金属度编写为灰度值又可以编写为1或0这样的二元值。

·粗糙度：粗糙度(Roughness)贴图可以以纹素为单位指定某个表面有多粗糙。采样得来的粗糙度数值会影响一个表面的微平面统计学上的取向度。

一个比较粗糙的表面会得到更宽阔更模糊的镜面反射（高光），而一个比较光滑的表面则会得到集中而清晰的镜面反射。

·AO：环境光遮蔽(Ambient Occlusion)贴图或者说AO贴图为表面和周围潜在的几何图形指定了一个额外的阴影因子。AO贴图则会把那些光线较难逃逸出来的暗色边缘指定出来。

## 3.2 实现

### 3.2.1 HTML网页框架搭建/样式编写

将网页样式定位左上文字，中间效果，右上参数的形式。

在外部编写css和js文件



图3-16 HTML

CSS中使用标签选择器指定窗体外边距，背景色，字体颜色，字体，字体大小，行高。

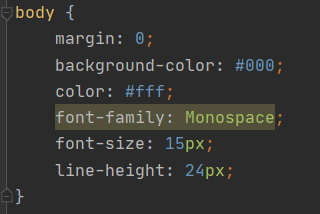


图3-17 CSS body

使用id选择器指定文字的位置，顶部边缘，宽度，内边距，盒子样式，左对齐，禁止点击，设置堆叠顺序。

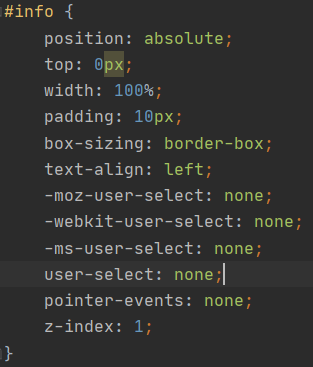


图3-18 CSS info

在js中使用three.js建立相机，鼠标控制，GUI显示，导入shader。

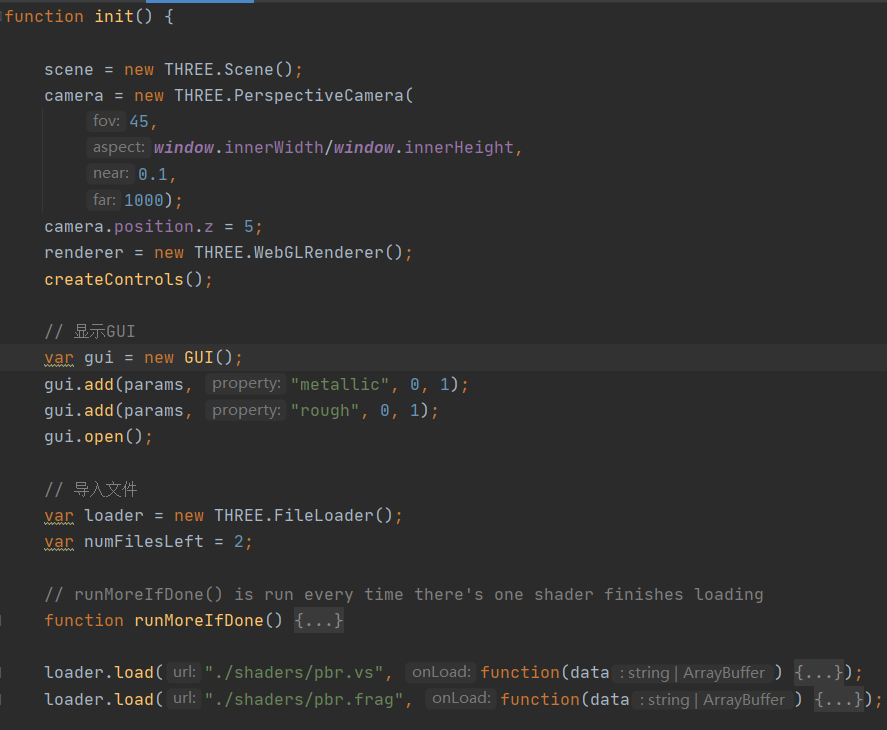


图3-19 js-1

添加灯光，球体网格，shader材质，uniform属性，建立好基础框架。



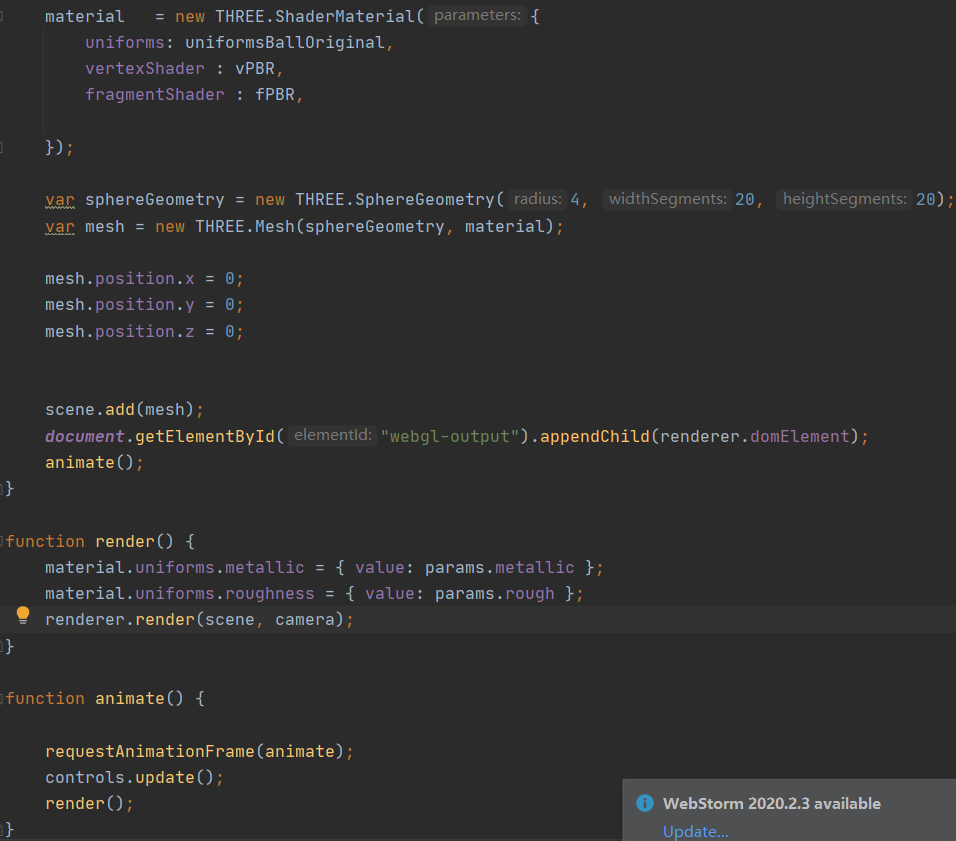
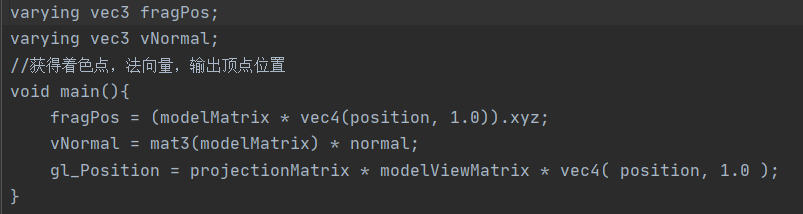


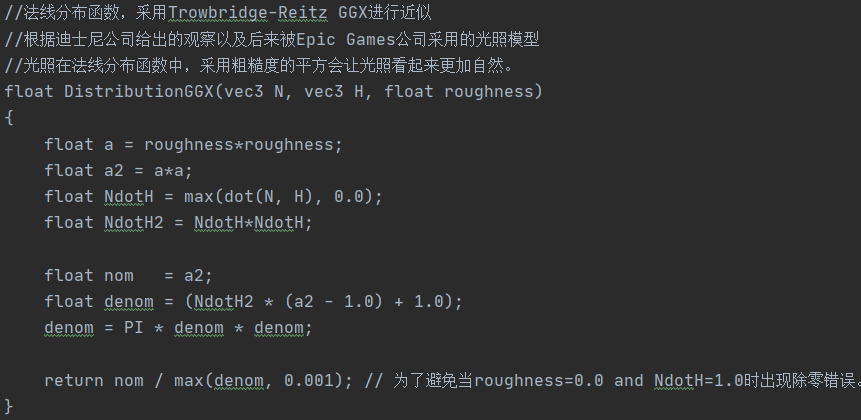
图3-20 CSS info

### 3.2.2 直接光照明的PBR

顶点着色器读取几何信息



片段着色器把之前提到的近似方法进行实现





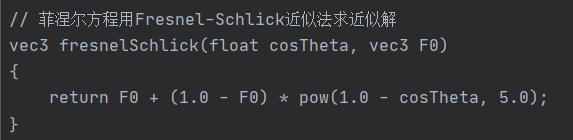
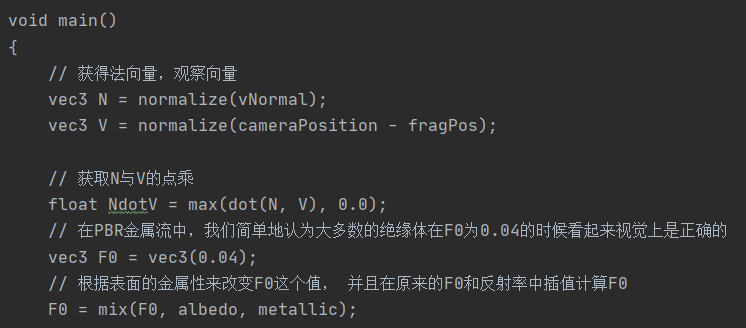


图3-21 近似函数

在实现上，我们采用的是点光源，所以对于一个片段来说，每个光源的入射方向只有一个，化积分为累加，将反射方程进行简化实现。



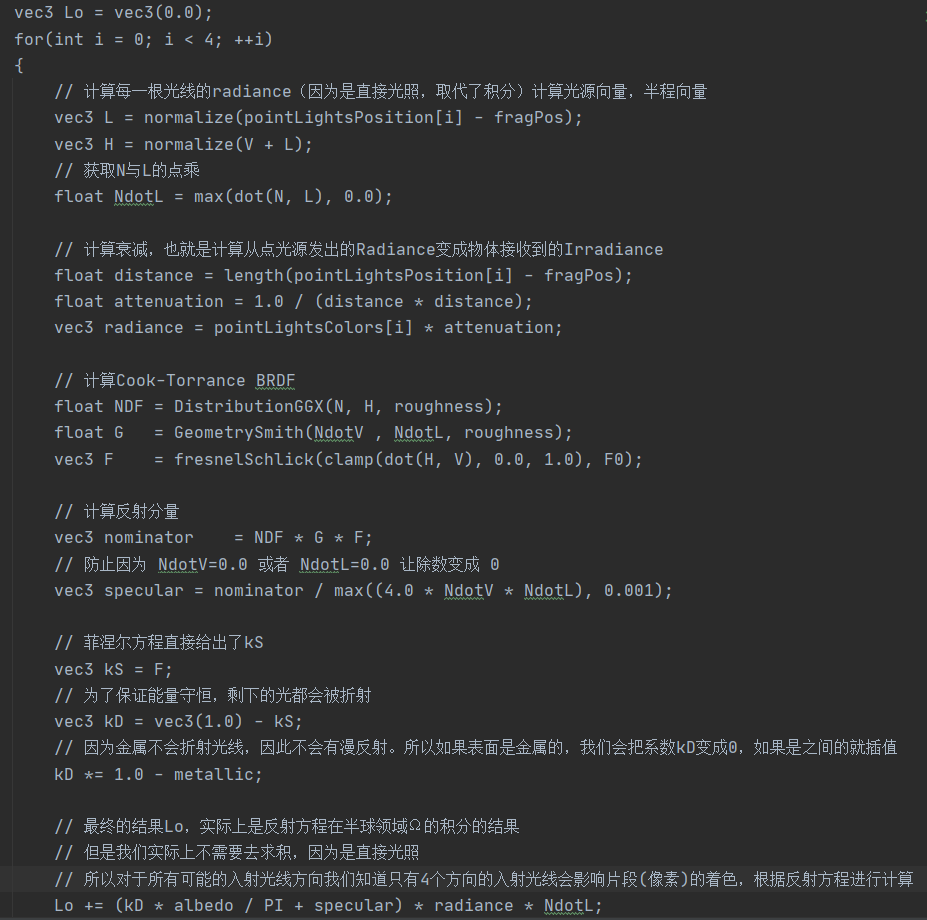


图3-22 反射方程实现

计算得到颜色值。

### 3.2.3 线性空间和HDR渲染

我们假设的所有计算都在线性的颜色空间中进行的，因此我们需要在着色器最后做伽马矫正，同时因为是基于物理的光照计算，所以颜色值可能会超过1，所以需要进行色调映射从HDR变为LDR进行显示。

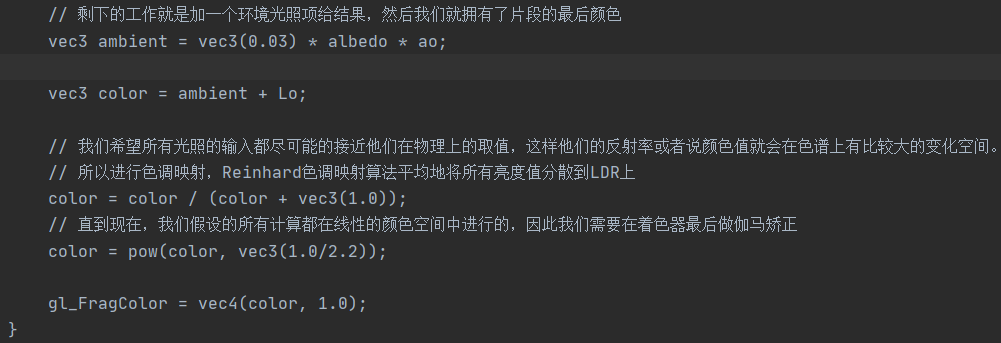


图3-23 色调映射与伽马校正实现

## 3.3 结果展示

对参数做不同调整得到效果。

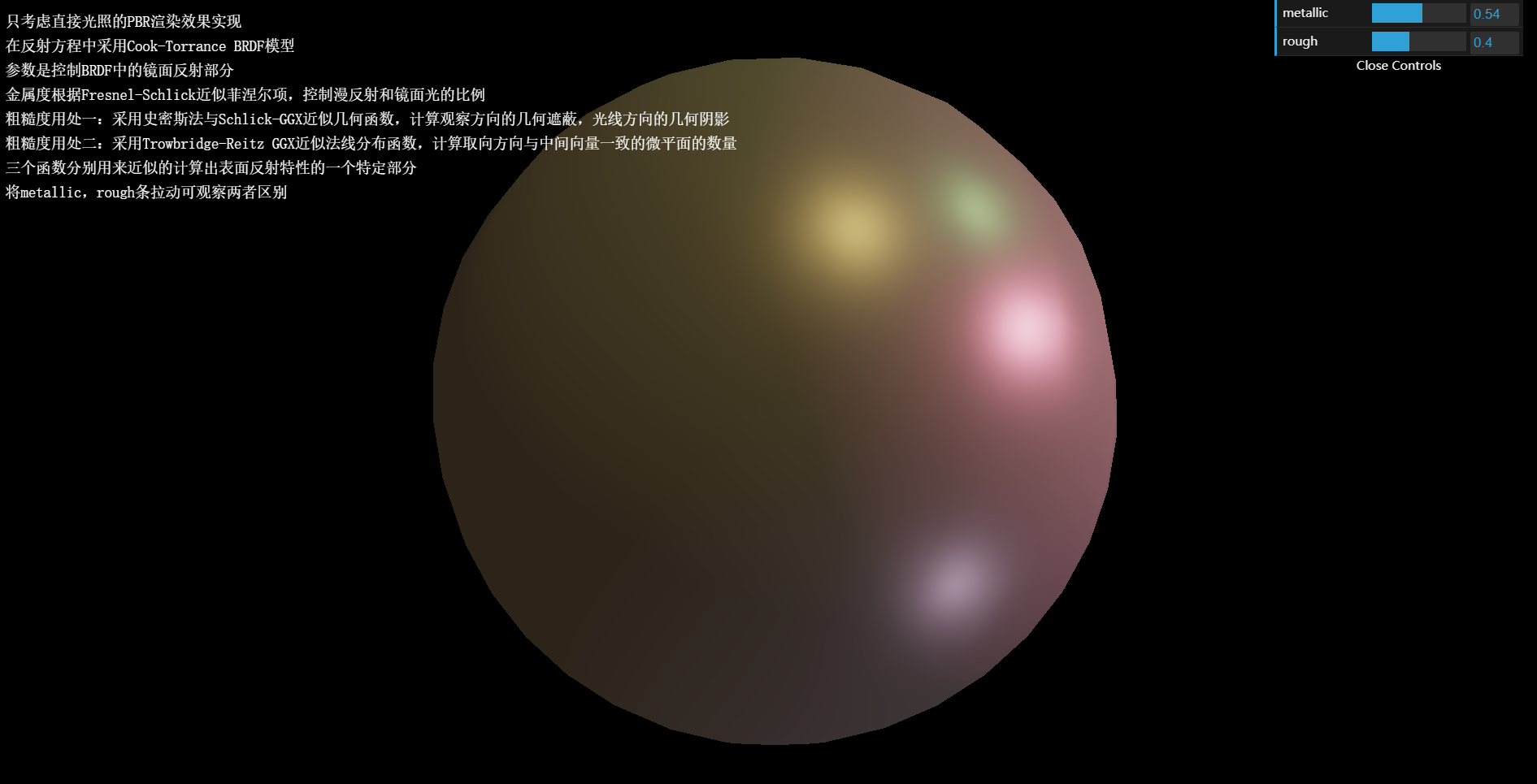


图3-24 效果一

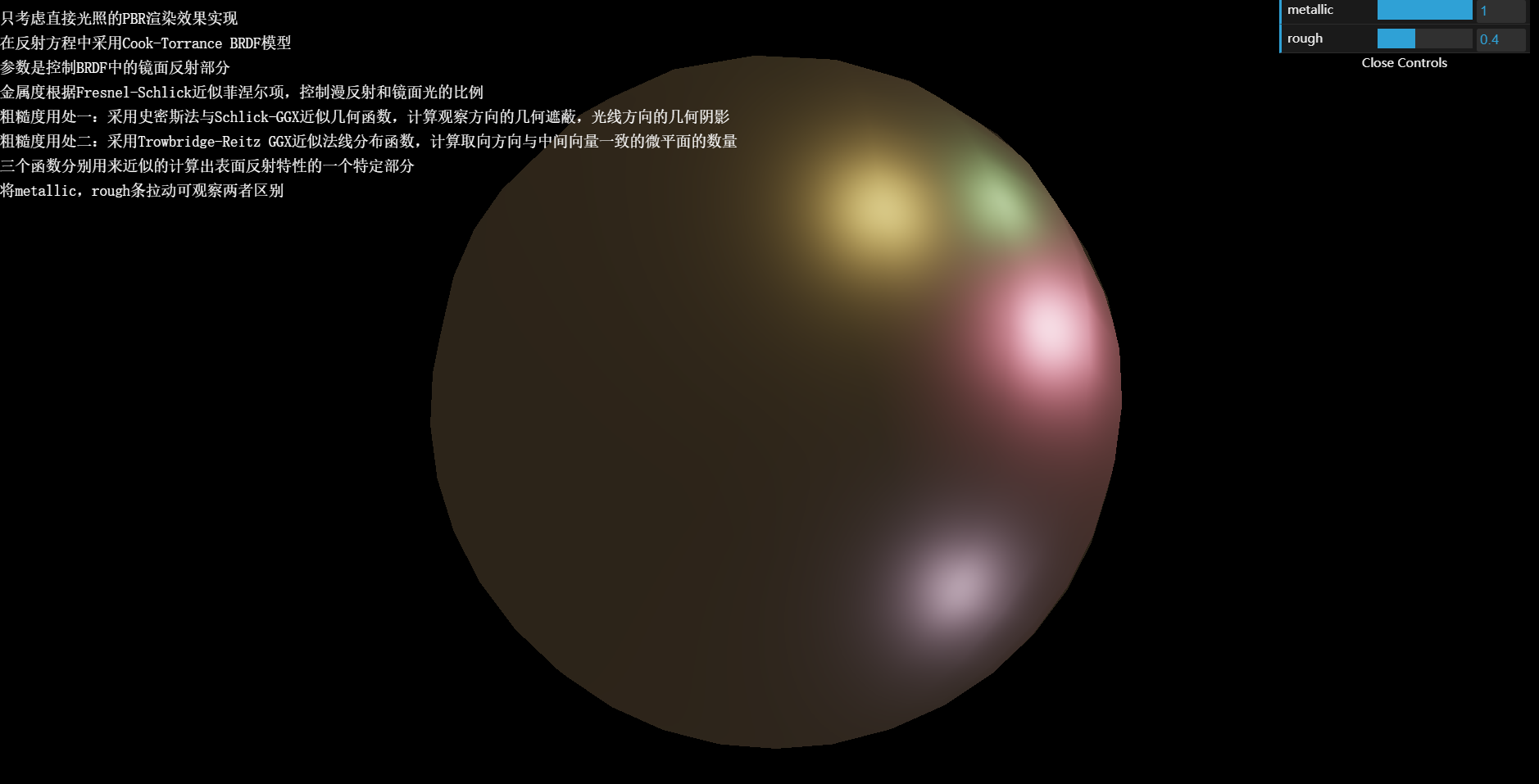


图3-25 效果二 高金属度

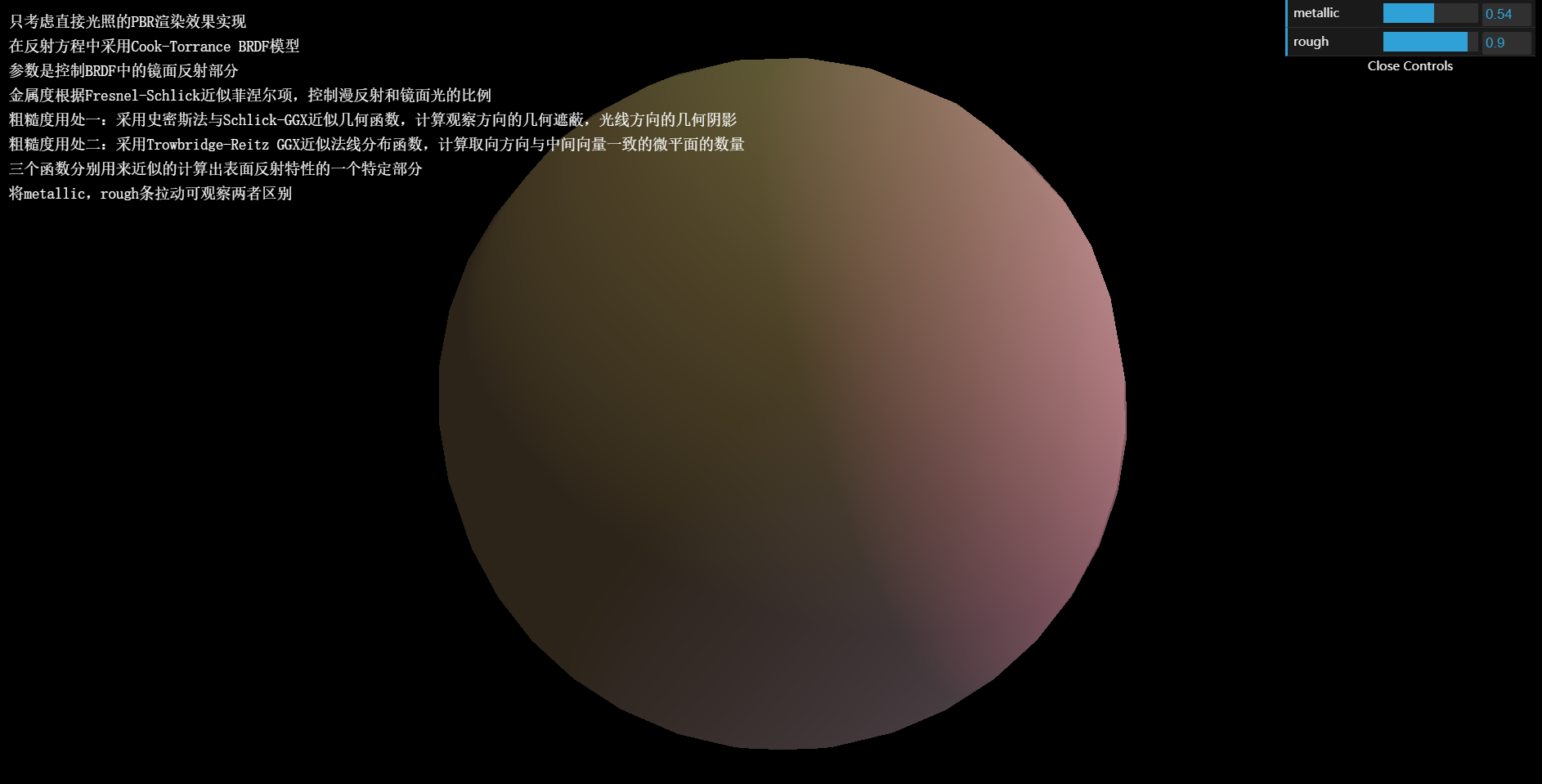


图3-26 效果三 高粗糙度

## 3.4 遇到的问题

### 3.4.1 在threejs中编写shader的问题

此前的都是用VS编写shader，在opengl3.3中自定义渲染管线，在three.js中，其将shader的使用与材质进行结合，同时不需要手动调整缓冲区，为了适应相关变化进行了区别。

Webgl将顶点向片段着色器传输的相关物体数据都绑定好了存进了varying类型的变量直接使用，自定义参数可以通过uniform变量传递。

在建立物体并渲染的过程中，采用先建立网格，后添加材质的方式

### 3.4.2 本地文件读取的问题

因为将shader文件存在单独的文件夹，所以读取文件会报跨域读取的错误。

Cross origin

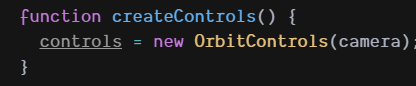
requests are only supported for protocol schemes: http, data, chrome,

chrome-extension, https.

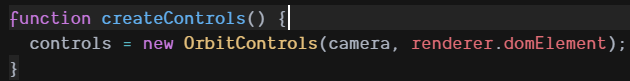
在HTML中引入方式是file协议，本地跨域仅支持ajax跨域只支持这些协议框架。解决方法是使用http-server启动本地服务器。

### 3.4.3 GUI栏鼠标移动问题

一开始设置控制器后，鼠标移动参数栏，球也会变，很困惑。



后来发现没有加到追加生成的canvas元素里，修改后解决。



### 3.4.4 文字栏CSS设置问题

最初设置CSS时，没有把鼠标相应行为去掉，导致可以拉拽文字。

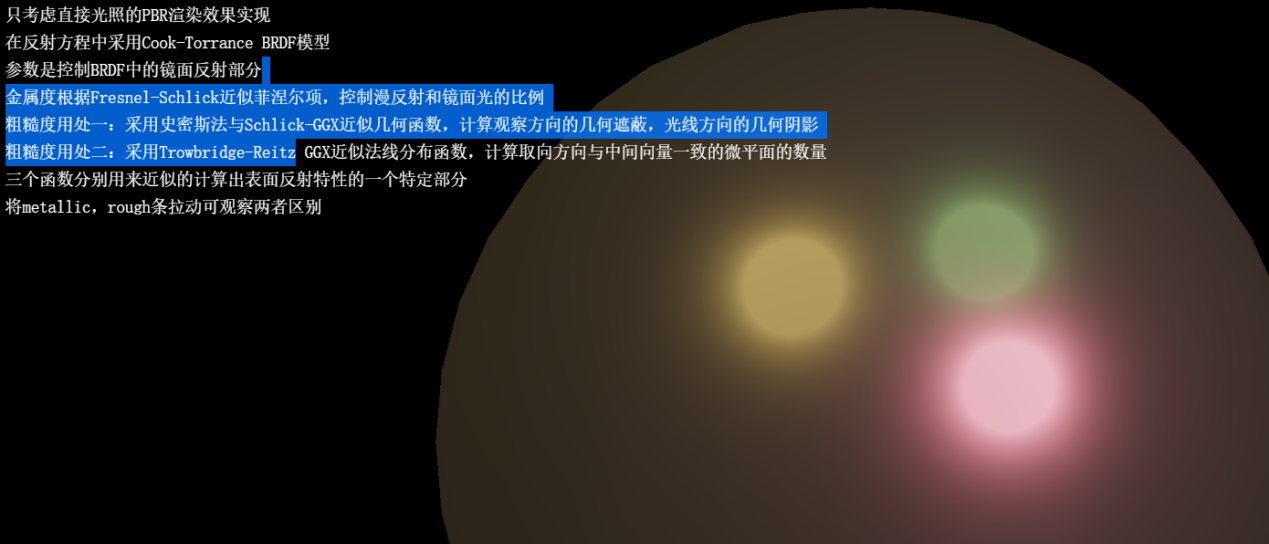
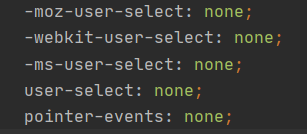


图3-27 文字错误

加上以下选项后文字就不可点击了。



# 四 卡通渲染部分

## 4.1 原理

### 4.1.1 Cel Shading

卡通渲染过程从一个普通的[3D模型](https://zh.wikipedia.org/wiki/%E4%B8%89%E7%BB%B4%E8%AE%A1%E7%AE%97%E6%9C%BA%E5%9B%BE%E5%BD%A2" \l ".E5.BB.BA.E6.A8.A1" \o "三维计算机图形)开始。与常规渲染不同的是，卡通渲染的光照效果是经过去真实感处理的。常规光源（明暗间有平滑过渡）的取值被逐一像素计算并投射到一小片独立的明暗区域上，以产生卡通式的单调色彩。

生成“黑线勾边”效果和轮廓线条可以用多种方法。常用方法之一是先渲染出比物体略大的黑色的边线，之后关闭背面探测（[Backface culling](https://en.wikipedia.org/wiki/Hidden_surface_determination" \o "en:Hidden surface determination)，用于剔除物件背面不需渲染部分的功能），将朝后的面被用黑色绘制出来。为了扩大轮廓，这些背面会多次以[线框模型](https://zh.wikipedia.org/wiki/%E7%BA%BF%E6%A1%86%E6%A8%A1%E5%9E%8B" \o "线框模型)绘制，且每次都有微小的改变。另一种方法是，背面先以固实方式渲染，并且其[顶点](https://zh.wikipedia.org/wiki/%E9%A0%82%E9%BB%9E_(%E9%9B%BB%E8%85%A6%E5%9C%96%E5%AD%B8)" \o "顶点 (电脑图学))均由[顶点着色器](https://zh.wikipedia.org/wiki/%E9%A0%82%E9%BB%9E%E8%91%97%E8%89%B2%E5%99%A8" \o "顶点着色器)沿着其顶点法线进行转译。在绘制边线后，重启背面探测来绘制物体的明暗和附属材质。最后，由于场景中的背面比正面所处位置更深远，图像会由[深度缓冲](https://zh.wikipedia.org/wiki/%E6%B7%B1%E5%BA%A6%E7%BC%93%E5%86%B2" \o "深度缓冲)组合。最终我们看到的就是，物件拥有黑色的勾边以及内部的轮廓线条。普遍的观点认为，这种用在动画和游戏上使用的“勾边”效果就是卡通渲染，但是实际上，卡通渲染指的是这种渲染方式，而无关是否拥有勾边效果。

下图是使用卡通渲染处理后的[犹他茶壶](https://zh.wikipedia.org/wiki/%E7%8A%B9%E4%BB%96%E8%8C%B6%E5%A3%B6" \o "犹他茶壶)：

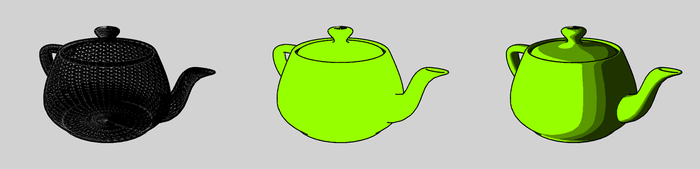
[](https://zh.wikipedia.org/wiki/File:Celshading_teapot_large.png)

图4-1

1. 用粗线条绘制背侧
2. 用基础材质绘制物件
3. 渲染

第2、3步可以通过复数材质处理结合为一步（参考[材质贴图](https://zh.wikipedia.org/wiki/%E6%9D%90%E8%B4%A8%E8%B4%B4%E5%9B%BE" \o "材质贴图)）。

还有一种勾边方法使用了平面图像处理技术。首先，场景被卡通渲染至屏幕大小的颜色纹理：

[](https://zh.wikipedia.org/wiki/File:Cel_shading_no_outlines.png)

图4-2

然后，场景的深度和空间表面法线信息被渲染至屏幕大小的颜色纹理：

[](https://zh.wikipedia.org/wiki/File:Cel_shading_depth.png)

图4-3

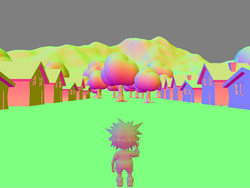
[](https://zh.wikipedia.org/wiki/File:Cel_shading_normals.png)

图4-4

之后对法线和深度纹理使用[查找边缘](https://zh.wikipedia.org/wiki/%E8%BE%B9%E7%BC%98%E6%A3%80%E6%B5%8B" \o "边缘检测)滤镜（例如[索贝尔算子](https://zh.wikipedia.org/wiki/%E7%B4%A2%E8%B2%9D%E7%88%BE%E7%AE%97%E5%AD%90" \o "索贝尔算子)），可以生成边缘纹理。位于边缘的像素纹理为黑色，其他的为白色：

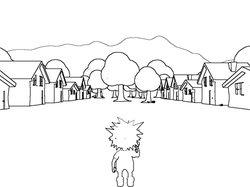
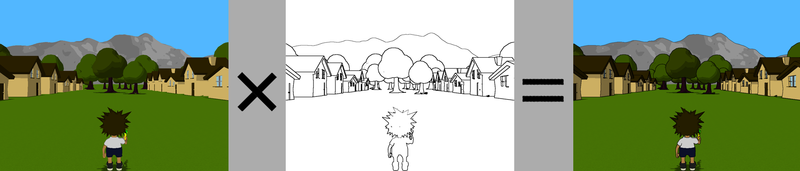
[](https://zh.wikipedia.org/wiki/File:Cel_shading_edge_detection.png)

图4-5

最后，边缘纹理和颜色纹理结合起来，生成最终渲染图：

[](https://zh.wikipedia.org/wiki/File:Cel_shading_composite_final_image.png)图4-6

## 4.2 实现

### 4.2.1 轮廓线

在NPR卡通渲染中，轮廓线的作用为增强对比使得物体与背景隔离开来，使得渲染效果更加强烈。

目前的实现为通过检测平行于屏幕方向的法向量，将其亮度值修改为零从而完成描边效果。

float dir = length(vNormal \* vec3(0.0, 0.0, 1.0));

if (dir < 0.5) {

dir = 0.0;

gl\_FragColor = vec4(dir, dir, dir, 1.0);

}

vNormal 是顶点着色器中传递过来的法向量；vec3(0.0, 0.0, 1.0)是垂直于屏幕的方向，也就是视图坐标系下的视角方向。vNormal \* vec3(0.0, 0.0, 1.0) 是将法向量和视角方向进行点乘，得到法向量在视角方向上的投影。length() 得到该点乘结果的模长，如果它较小，代表法向量在视角方向上的投影较小，也就是法向量较接近于平行屏幕的方向。

### 4.2.2 着色

目前的NPR卡通渲染使用了Cel Shading的形式来进行渲染，而Cel Shading的渲染方法为将颜色丰富的渐变色彩变成单调的平面色彩，也就是降低色阶数量，达到一种卡通的光照效果。

完成Cel Shading需要以下步骤

1.指定一个颜色作为物体的基础颜色；

2.通过光照计算得出每个片元对应的亮度；

3.将亮度由连续的映射到离散的若干个亮度值；

4.将亮度值和基础颜色结合得到片元颜色。

首先完成步骤1

sm = new THREE.ShaderMaterial(

{

uniforms:

{

light:{type:'v3',value:directionalLight.position},

color: { type: 'v3',

value: new THREE.Color('#48D1CC')}

},

vertexShader: document.getElementById( 'fish-vertexShader' ).textContent,

fragmentShader: document.getElementById( 'fish-fragmentShader' ).textContent,

side: THREE.FrontSide,

blending: THREE.AdditiveBlending,

} );

通过以上代码可以完成着色器材质的设置，其中代码value: new THREE.Color('#48D1CC') 为设置物体的基础颜色。

此后再创建形状var geometry = new THREE.TorusGeometry(3, 1, 12, 18);

再将形状与材质赋值给网格便完成了物体的创建

var cube = new THREE.Mesh( geometry, sm );

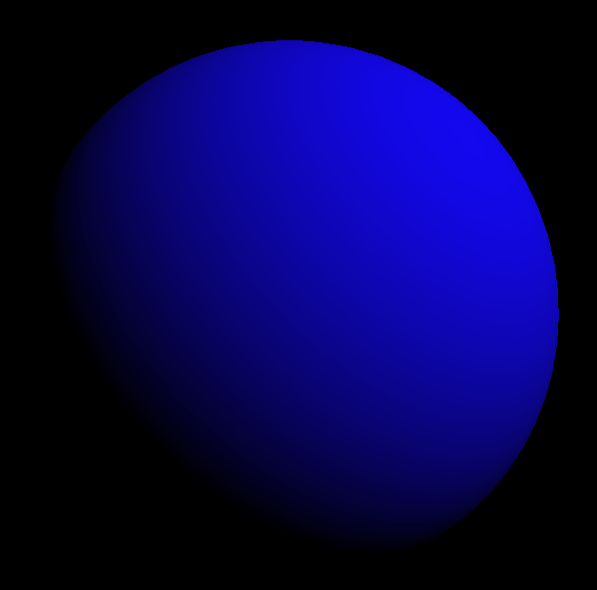


图4-7球体

此时未进行渲染的效果如上图

1. 通过光照计算得出每个片元对应的亮度；

首先在顶点着色器中，我们将从 Three.js 中获得的颜色赋值给 vColor 变量，让它传递给片元着色器。

然后，使用法向矩阵乘以法向量，并将其归一化后，传递给 vNormal。

vNormal = normalize(normalMatrix \* normal);

在片元着色器中将归一化后的光线与nNormal进行点乘。得到每个片元的亮度值  diffuse

float diffuse = dot(normalize(light), vNormal);

3.将亮度由连续的映射到离散的若干个亮度值；

根据亮度值diffuse的范围，赋给diffuse一个确切的值以完成NPR渲染所需的从渐变颜色变为单调的平面颜色的需求。

如此处理会造成颜色的突变，如需改变可以使用线性插值。

if (diffuse > 0.9) {

diffuse = 1.0;

}

else if (diffuse > 0.7) {

diffuse = 0.8;

}

else if (diffuse > 0.5) {

diffuse = 0.6;

}

else if (diffuse > 0.3) {

diffuse = 0.4;

}

else if (diffuse > 0.1) {

diffuse = 0.2;

}

else {

diffuse = 0.1;

}

4.将亮度值和基础颜色结合得到片元颜色。

gl\_FragColor = vec4( color\* diffuse, 1.0);

将物体的基础颜色color与diffuse相乘，作为颜色输出便完成了Cel Shading形式的NPR卡通渲染。

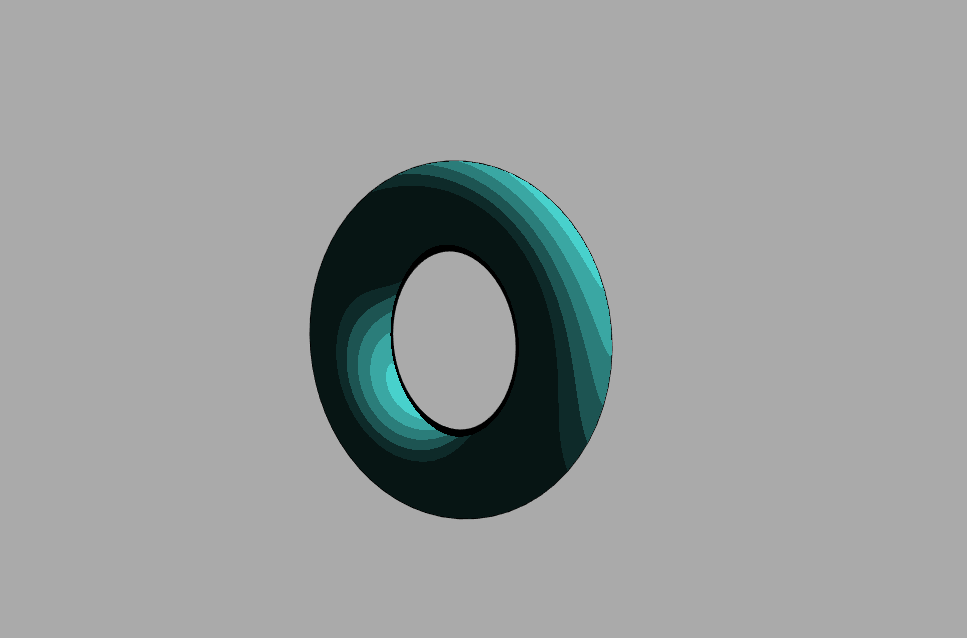


图4-8初步渲染的圆环

### 4.2.3 着色改进

此时渲染效果并不好会产生很明显的颜色分层显得不自然。

于是便对diffuse做了平滑的处理

diffuse =0.6+0.3\* smoothstep(0.6,0.96,diffuse);

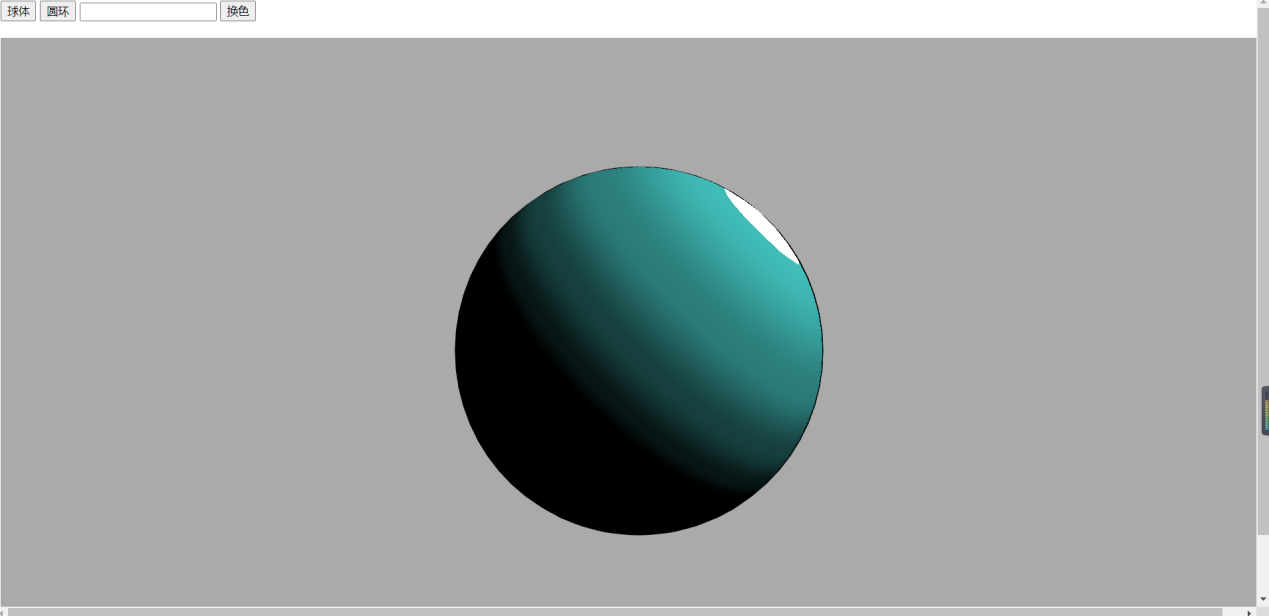
再改变亮度值最高的部分将其设置为白色作为高光

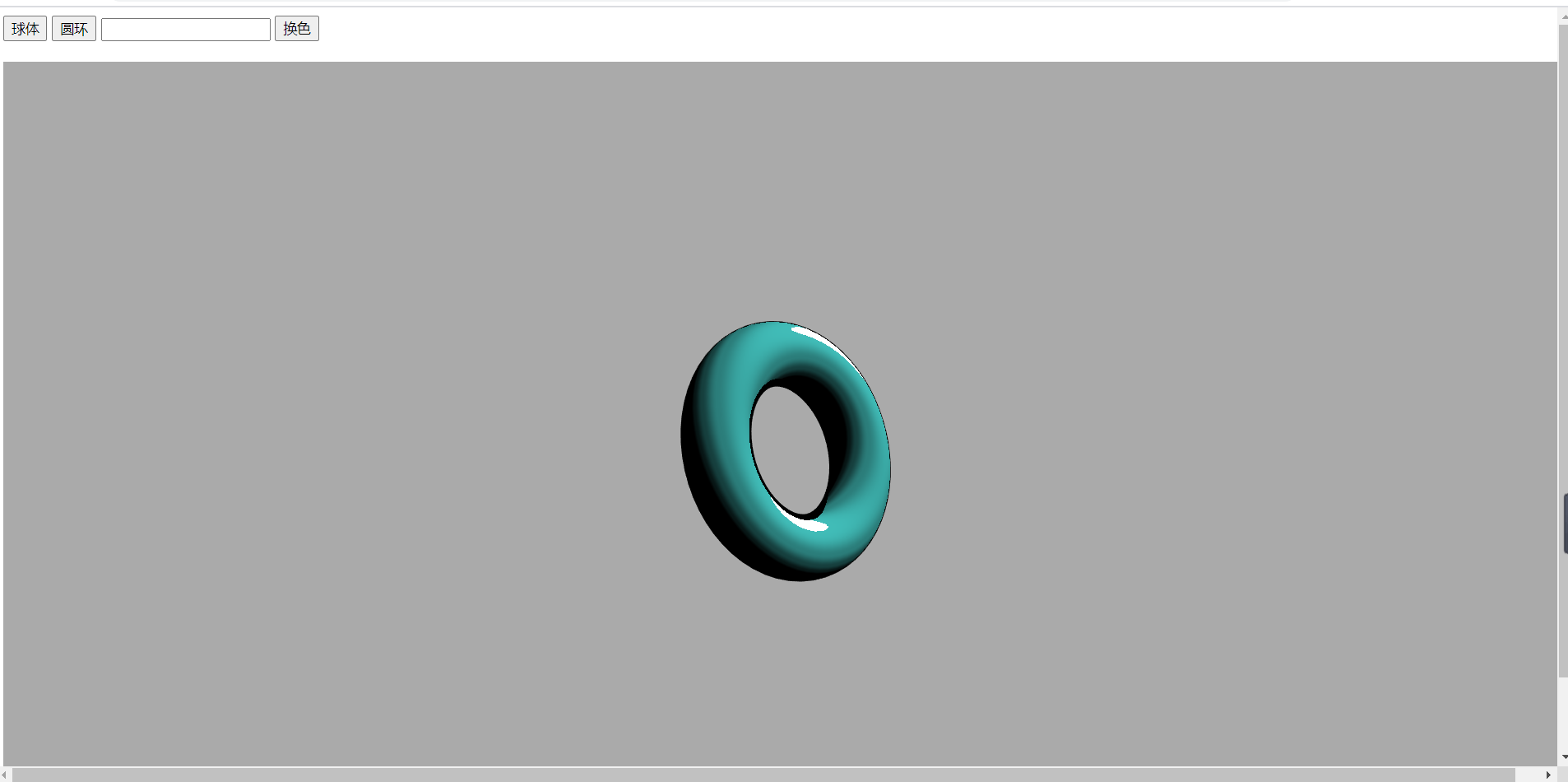
if (diffuse > 0.96) {

gl\_FragColor = vec4(1.0, 1.0, 1.0, 1.0);

}

## 3.3 结果展示

图4-9球体展示

图4-10圆环展示

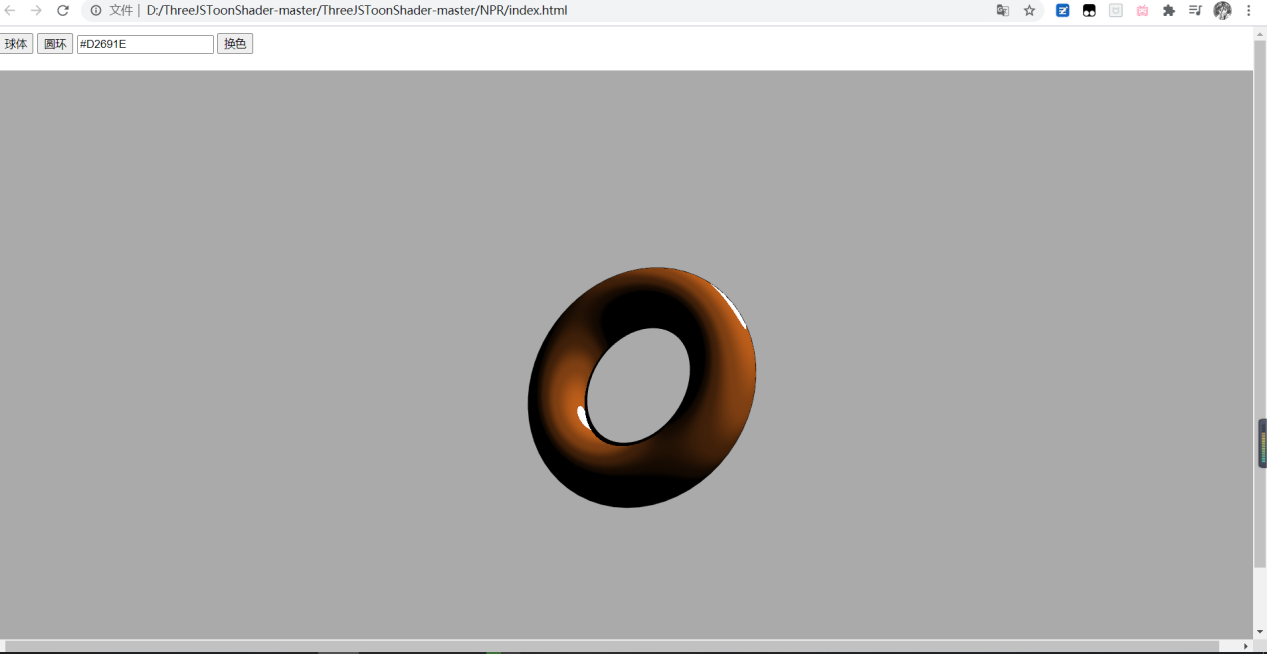


图4-11换色展示

# 五 结论/展望/心得/人员分工

李钦俊——PBR部分：

在本次期中项目中，学习了图形渲染中PBR相关理论知识，并将知识应用于代码实现，与HTML，CSS，JS技术进行结合，实现了网页端的直接光照形式下的PBR结果。跨学科交叉应用很有意义。

在后面的学习中会尝试基于图像的光照，将反射方程的积分形式还原，尝试结合paper添加多次反射的结果，并尝试添加本地替换图片等功能。

吴炫熠——NPR部分：

在本次期中项目中，学习了图形渲染中NPR相关理论知识，并将知识应用于代码实现，实现了网页端的NPR渲染结果，与切换模型及颜色的功能。

后面的学习中会尝试优化界面设计，改良该渲染并学习其他卡通渲染的方法。

# 六 参考文献

[1]林红旭 Leo BIM轻量化引擎开发工程师.如何在threejs中使用shader[EB/OL].https://zhuanlan.zhihu.com/p/145890220,2020-6-4.

[2]毛星云.【基于物理的渲染（PBR）白皮书】（一） 开篇：PBR核心知识体系总结与概览[EB/OL].https://zhuanlan.zhihu.com/p/53086060

[3]毛星云.【基于物理的渲染（PBR）白皮书】（二） PBR核心理论与渲染光学原理总结[EB/OL].https://zhuanlan.zhihu.com/p/56967462

[4] dxyzhbb.Three.js - THREE.Color颜色对象详解 （方法介绍）[EB/OL].https://blog.csdn.net/dxyzhbb/article/details/94437226,

[5] AndreaH.MIME type 问题的解决[EB/OL].https://blog.csdn.net/weboof/article/details/79851414,

[6] SongpingWang.http-server—nodejs 开启本地服务器[EB/OL].https://wangsp.blog.csdn.net/article/details/84100316