



**本科毕业设计（论文）**

**基于可编程渲染管线的游戏效果研究**

**学 院 艺术与设计学院**

**专 业 数字媒体技术**

**年级班别 2015级1班**

**学 号 3115008365**

**学生姓名 黄云龙**

**指导教师 韦宇炜**

**2019年 05 月**

**基于可编程渲染管线的游戏效果研究**

**黄云龙**

**艺术与设计学院**

**摘 要**

随着玩家的品味逐渐上升，以及高端显卡价位逐渐平民化，渲染技术逐渐成熟，导致玩家对画面渲染质量的追求越来越高，传统注重游戏性而没有重视美术视觉效果的游戏已经逐渐吸引不了玩家。为了能够在游戏效果上捕捉玩家的眼球，吸引玩家注意，使得玩家可以沉浸于游戏氛围中，本设计将采用Unity2018版本新推出的可编程脚本渲染管线对常见的游戏效果进行研究和实现，探究真实又不失风格化的游戏效果实现的新流程。

本设计是在游戏《Doria》中进行实践尝试，采用可编程渲染管线及自定义着色器实现其简面风格化的美术特点，这是一款讲述一个小女孩追寻内心中呼唤着她的声音的剧情解谜向游戏。本设计就其中关键的两个游戏效果水和雾进行展开分析，在可编程渲染管线的新流程下进行实现，最终的效果不仅可以满足《Doria》的美术要求，又可以独立于游戏《Doria》之外，作为游戏效果插件运用于其他游戏或场景中。

**关键词：** Unity3D，可编程渲染管线，平静水面模拟，垂直雾模拟，屏幕后处理

**Abstract**

As the taste of the players gradually rises, and the price of high-end GPU is cheaper, the rendering technology is more mature, which leads to higher requirements for the quality of the rendering. The game that focus on gameplay without paying attention to the artistic visual effect has gradually become unable to attract players. In order to attract the attention of the player in the game effect, so that the player can be immersed in the game atmosphere, this design will use the scriptable render pipeline of Unity 2018 to study and realize the common game effect, and explore the new process of realizing real and stylized game effect.

This design is practiced in the game "Doria", using scriptable render pipelines and custom shaders to achieve its artistic features of simplicity and low-poly stylizing. The game "Doria" is a story about a little girl pursuing her inner voice. This design analyzes the two main game effects, water and fog, and realizes them under the new process of scriptable render pipeline. The final effect can not only meet the art requirements of "Doria", but also be used in other games or scenes as a plug-in for game effects.

**Keywords:** Unity3D，SRP，Clam Water Simulation，Vertical Fog Simulation，Post Process

**目 录**

[1. 绪论 1](#_Toc8540644)

[1.1 课题简介 1](#_Toc8540645)

[1.2 课题研究背景以及意义 1](#_Toc8540646)

[1.3 国内外研究现状 2](#_Toc8540647)

[1.4 课题核心研究内容 2](#_Toc8540648)

[2. 相关技术研究 4](#_Toc8540649)

[2.1 渲染管线 4](#_Toc8540650)

[2.2 Unity Shader 5](#_Toc8540651)

[2.3 Unity固定渲染管线 5](#_Toc8540652)

[2.4 可编程脚本渲染管线 7](#_Toc8540653)

[2.5 轻量级渲染管线 8](#_Toc8540654)

[2.6 Shader Graph 8](#_Toc8540655)

[2.7 Post Processing 8](#_Toc8540656)

[3. 平静水面模拟 10](#_Toc8540657)

[3.1 水效果介绍 10](#_Toc8540658)

[3.2 水的深度表现 10](#_Toc8540659)

[3.3 水波效果 14](#_Toc8540660)

[3.4 水下折射 18](#_Toc8540661)

[3.5 水面反射 21](#_Toc8540662)

[4. 基于高度的垂直雾模拟 31](#_Toc8540663)

[4.1 雾效果介绍 31](#_Toc8540664)

[4.2 后处理垂直雾 31](#_Toc8540665)

[4.3 雾的流动 35](#_Toc8540666)

[4.4 距离衰减 35](#_Toc8540667)

[结论 38](#_Toc8540668)

[参考文献 39](#_Toc8540669)

[致谢 40](#_Toc8540670)

1. 绪论

1.1 课题简介

本课题是基于可编程脚本渲染管线，对常见的游戏效果（例如游戏中的水效果，雾特效等）进行分析研究与重现，对比在固定渲染管线下实现同样的效果有什么不同及优势，同时将效果在游戏项目《Doria》中进行运用实践，并且针对游戏美术设计需求调整效果及做具体的优化。

1.2 课题研究背景以及意义

从1999年NVIDIA公司发布第一个具有顶点处理能够被大众所消费得起来的图形芯片Geforce 256，并提出GPU的概念，再到2001年发布行业第一个可编程的图形芯片Geforce 3，划开了可编程GPU的时代，使得GPU从以往仅可配置的固定功能管线过渡到了可配置又支持高度可编程的管线。直到如今，代替固定功能管线控制GPU的主要手段已经是可编程着色器了，尽管为了提高在某些情况下的渲染效率，GPU管线的一部分功能仍保持仅可配置的固定功能管线的特征，但朝着更具灵活性和可编程特征的方向发展已经成为了大趋势所在。

直到可编程脚本渲染管线推出之前，Unity长时间使用的是一套默认固定的渲染管线，虽然这个固定管线本身为了去支持所有的平台也经历了很多次的修改，但相比于最新的渲染技术的发展，这个旧管线始终都无法跟得上或者是保证其性能最优，还是有一些本质上的渲染问题是无法被修复或绕过的，从可配置性上来说，由于Unity并非一个开源引擎，渲染部分本身也是个黑盒，无法知道里面的代码怎么写的，开发者没办法改变渲染流程，提供的配置也不能满足现有特定的需求；从可发现性上，管线太过庞大复杂，bug也越来越多难以发现，且由于比较过时而且也比较复杂，添加最先进的效果也变得十分困难；从灵活性上，固定的渲染管线没办法跳过不必要的逻辑或者只能以比较绕的方式去做，无法根据具体的需求去定制一套渲染流程。所以Unity 在其2018版本的编辑器中正式引入了可编程脚本渲染管线（Scriptable Render Pipeline，简称SRP），用来解决仅有一个默认渲染管线所造成的从可配置性、可发现性、灵活性上无法修复或绕过的问题，且该管线代码开源，开发者可以根据具体的项目需求对现有的渲染管线进行简单的修改，或者直接按需编写个性化定制的管线。然而由于可编程脚本渲染管线部分功能目前仍处于试验阶段，且与旧的默认的固定渲染管线相比有巨大的改动，导致众多的可以制作不错的游戏效果的插件都无法在新管线下使用，同时由于新管线高度可定制性及灵活性，一些游戏效果的实现流程和方式与传统的实现方式相比也大的不同，固定管线所不可避免的渲染问题，在新管线下都可以被解决。毋庸置疑，随着可编程脚本渲染管线的发展越来越成熟，未来将会有越来越多的游戏，实时渲染电影，或任何对渲染或性能有硬需求的项目，都会在定制化的渲染管线的基础上进行开发。

本课题研究的意义在于面对未来定制化可编程渲染管线发展的趋势，跟着技术发展的脚步，以常见的游戏效果为例，进行分析与实现，探究可编程渲染管线下游戏效果实现的方式与固定渲染管线下的不同，以点带面，扩充其他游戏效果实现的新思路。

1.3 国内外研究现状

可编程渲染管线是在2018版本Unity作为预览版本引入的，直到2019年的4月份才在正式版本中发布非预览版本的轻量级可编程渲染管线，在此期间可编程渲染管线一直处于开发测试版，功能的更新迭代非常迅速。同时伴随着使用Unity制作的超写实科幻短片《Adam》三部曲、第一人称互动故事游戏《Book of the Dead》等令人惊艳的作品问世，将实时渲染推向了高潮，使得大家对游戏渲染有着空前的关注度。尽管同时期也涌现出了很多渲染效果相当不错的作品，然而却几乎没有国内团队的作品出现。不得不说，我国虽然有很多不错的商业游戏，收入也是全球排行版前列，然而却少有可以被玩家谈论画面渲染质量不错的作品。

游戏被称为第九艺术，而艺术是人们对于美的一种追求，反映人们的现实生活和精神世界的，所以一款游戏好的渲染质量不仅能够使得游戏作品更加出彩外，也能够为游戏赢得不错的口碑。虽然国内目前在渲染质量口碑好方面拥有的游戏较少，但仍时不时有不错的作品出现，比如拥有国内顶尖卡通渲染技术渲染的上海米哈游的手游作品《崩坏3》，网易端游《逆水寒》等，从中也可以看到渲染技术在国内受到的重视以及发展和进步。

1.4 课题核心研究内容

本课题研究的主要内容是如何在Unity新推出的可编程渲染管线的基础上实现游戏《Doria》中需要用到且需要营造的游戏效果，实现一个既有能够让玩家身临其境的场景氛围，又有一定故事性，能够带给玩家启发的游戏。因为可编程脚本渲染管线是新推出的较新的一个内容，同时也推翻了传统固定渲染管线的渲染方式，导致资源商店（Asset Store）几乎所有的特效和插件都在新管线下无法使用，所以本课题也会就游戏《Doria》中较为经典的两个美术效果需求水和雾进行深入研究。实现较为真实的水面效果，包括水的深度体现，水的流动，水底的折射，水面的反射，以及从水面上较为饱和的水汽所诞生的雾效，雾的流动，随着距离衰退等效果。分析实现的过程，思考与固定渲染管线下的实现有哪些不同，同时寻找同种效果在《Doria》中最适合的实现方式。

2. 相关技术研究

2.1 渲染管线

渲染管线(Render Pipeline)，也可以叫做渲染流水线，是将三维场景中的物体或者对象渲染成平面的二位的图像显示在屏幕上的一系列操作的统称，中间的过程包含坐标转换、裁剪剔除、屏幕后处理等一系列高级概念。这个任务一般是由CPU和GPU一起配合完成的。在《实时计算机图形学·第三版》一书中，将一个渲染管线，也即是从CPU读取数据内容开始，到渲染到二位图像画面中的流程划分为应用、几何、光栅化，共3个阶段。如图2.1所示。

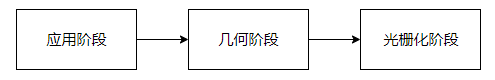


图2.1 渲染管线中的3个概念阶段

渲染管线的起点，是从应用阶段开始的，主要是由我们的应用进行来控制渲染的执行，实现的工作通常由CPU来负责。由于应用通常是由开发者开发，所以开发者掌握着这个阶段绝对的控制权。在这个阶段，开发者需要先布置好场景，调整相机位置，光源的方向等等， 然后设置好每个模型对应的材质，选择哪个着色器，纹理等信息。之后CPU会加载将渲染所需要的数据，以设定好的渲染状态，来调用Draw Call（绘制命令），指导GPU执行绘制操作，从而进入几何阶段。

​ 几何阶段，主要负责处理与顶点相关的信息，通常会在此阶段进行逐顶点的操作，将每个顶点进行坐标转换，对超出视野外的顶点进行裁剪，在把剩余的有效顶点投影到屏幕中去，同时为了减少光照效果所带来的复杂计算，有时也将光照计算放在几何阶段进行。最后输出每个顶点的屏幕空间坐标，以及对应的法线、切线、深度、颜色等有用的信息，然后传递到下一个阶段光栅化中去。

​ 光栅化阶段接收到从几何阶段传递过来的各个顶点数据后，对数据进行插值，然后对每个像素进行各种测试，包括模版测试，深度测试等，最终判断该像素是否应该被绘制在屏幕上，以及该以何种颜色来进行绘制，并渲染出最终显示在屏幕上的图像。

​ 后两个阶段主要都是在GPU中进行实现，因此开发者无法拥有完全的掌控权，但GPU仍对这两个阶段划分出来的若干个小的阶段分别提供了不同的可以用来配置等部分以及可以被开发者进行编程自定义的部分，如图2.2所示。

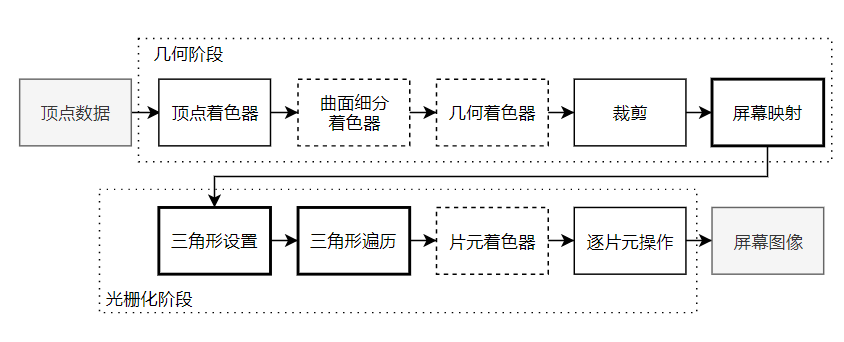


图2.2 GPU渲染管线细分阶段

其中，顶点着色器 、片元着色器阶段等主要阶段都是完全被开发者通过编程进行控制的，裁剪和逐片元操作阶段仅可配置不可被编程，这些阶段都可以由开发者通过编写Shader（着色器程序）进行配置或控制，其他阶段都是由GPU固定实现的，开发者没有任何控制权。

2.2 Unity Shader

渲染管线虽然复杂，但Unity作为一个优秀的引擎已经帮我们把复杂或者常用的功能都封装起来了，开发者很多时候可以在一个Unity Shader中定义一些输入都变量、编写顶点和片元着色器，进行一些必要的配置， 就可以作出大部分的渲染效果。Shader会在应用运行时编译出最终的代码在GPU上运行，也可以认为是渲染管线上的一些可被开发进行编程自定义的阶段，依靠Shader我们可以调整不同方案时，渲染管线中所被需要的渲染细节，例如为了性能考虑，光照计算可能会被放置在顶点着色器中进行，大多数情况下，顶点着色器主要是做着对顶点进行变换以及传递数据给片元着色器的工作，而片元着色器则是负责确定每个像素该不该渲染或以何种颜色进行渲染。

2.3 Unity固定渲染管线

一个三维对象或场景最终渲染到屏幕画面上让用户可以看到，大多数情况下并不是仅用一次绘制就可以的。比如阴影渲染的处理，常见的Shadow Map技术，是从灯光空间用相机渲染一张深度图，然后再回到主相机逐像素对到灯光位置的距离进行比较，判断是否该点处于阴影位置从而再一次渲染绘制出带阴影效果的图像。 所以在应用阶段，会根据渲染管线的设置，CPU可能会调用多次Draw Call 让GPU进行绘制渲染，绘制的内容和顺序由渲染管线决定。

Unity默认提供了两条内置的固定渲染管线，前向渲染（Forward）和或延迟渲染（Deferred），图2.3展示了两条渲染管线的渲染流程以及区别。

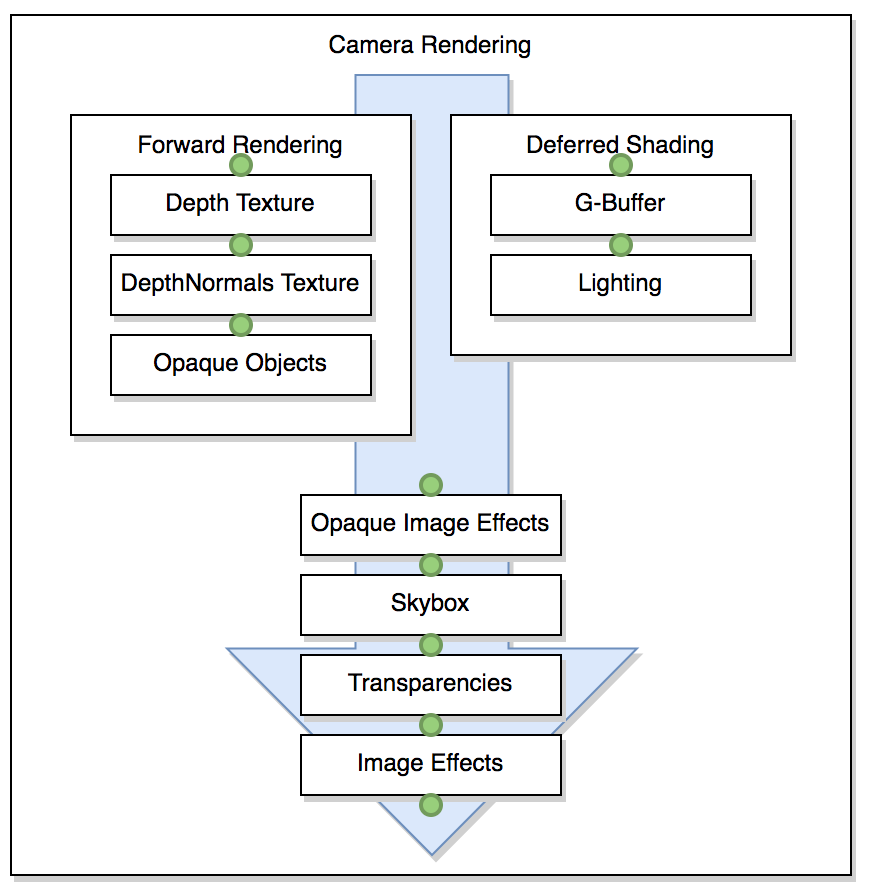


图2.3 Unity内置渲染管线

这两条内置的渲染管线是固定无法更改的，存在下列的缺点：

（1）只能够绕过无法修复的渲染问题

（2）可能需要应对所有不同的项目不同的开发需求，管线太过于复杂且庞大。

（3）由于固定管线太过复杂庞大，难以在原基础上添加最先进的效果。

于是Unity 2018版本引入了可编程脚本渲染管线以解决旧管线所存在的问题。

2.4 可编程脚本渲染管线

可编程脚本渲染管线（Scriptable Render Pipeline，简称SRP）是为了优化以往的固定渲染管线不能满足特定的需求，也难以对原来的关系进行扩展修改等问题，Unity在管线设计上做了调整，将渲染最核心的部分，放在性能比较好的C++部分中，让C++那部分的代码去使用多线程实现对性能要求比较高的内容。而在C#部分则是暴露出更多的API来让开发者更加方便的选择指定的API去调用特定功能的渲染指令如图2.4。这样做的好处即是只要保证C++部分的代码都可以严格通过各种白盒测试，就可以仅通过C#端的代码的调理来对实际项目进行匹配，同时也方便调试。

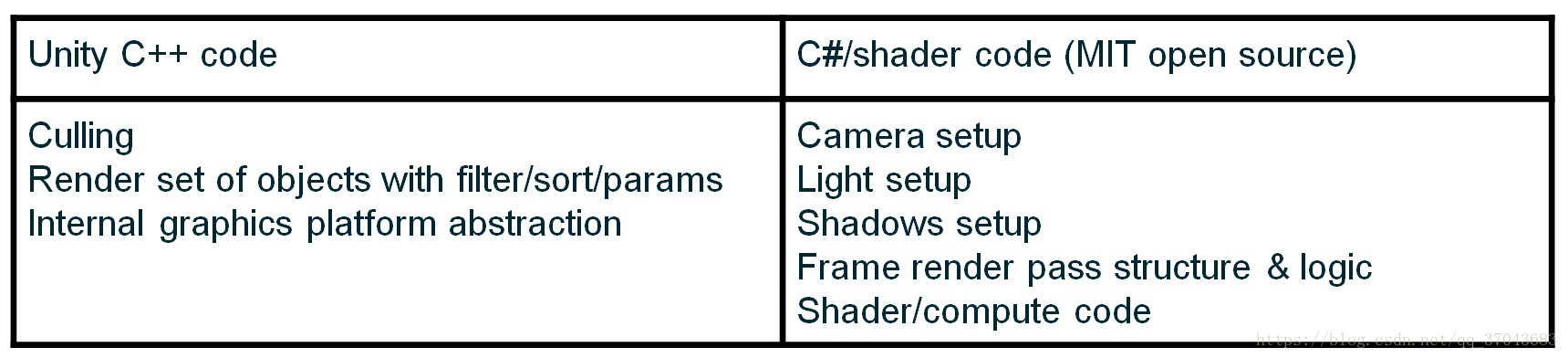


图2.4 可编程渲染管线C++端和C#端

对于开发者而言，主要是使用Unity提供的C#端API对现有管线进行修改，或者直接编写定制化的渲染管线。Unity将可编程渲染管线的使用层设计为渲染管线资源（Render Pipeline Asset）和渲染管线实例（Render Pipeline Instance）。如图2.5，渲染管线资源文件主要用来序列化存储该渲染该渲染管线上的一些用来设置的参数，当这些参数被改变时，无论是在编辑模式下，还是在运行模式下都会重新创建一个渲染管线实例，并摧毁原来的渲染管线实例。而固定渲染管线则自始自终都仅有一个渲染管线实例，这也是固定渲染管线的局限性的地方，无法实时浏览到调整管线参数后的结果。

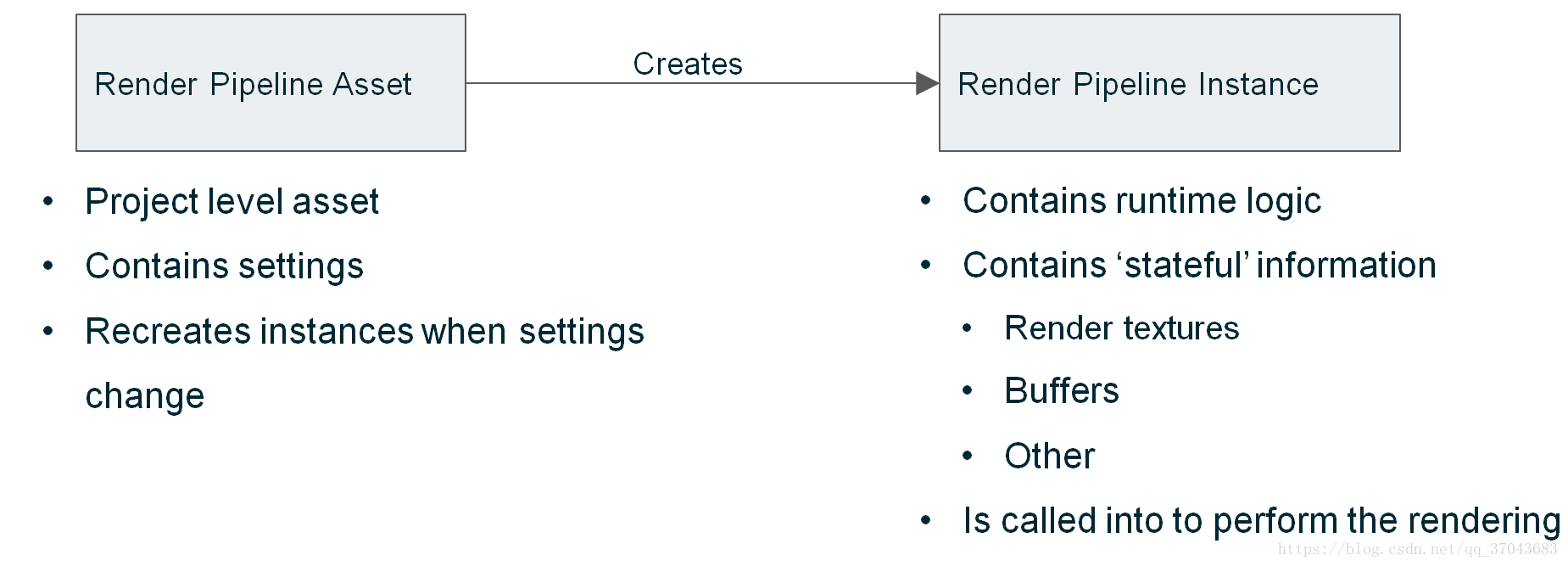


图2.5 可编程渲染管线使用层设计

2.5 轻量级渲染管线

轻量级渲染管线（Lightweight Rendering Pipeline，简称LWRP），是Unity所提供的两个可编程脚本渲染管线模板之一，目的为了帮助开发者可以在自定义渲染管线时避免在性能方面考虑的情况欠缺而导致出现更大的问题，另一个模板是高清渲染管线，两个模板各有侧重点。轻量级渲染管线的特点在于通过对光照和阴影方面采用一些折衷的手段，去除某些特定功能，提升条件较差的平台上的实时性能表现。所以其目标平台定位在移动平台，低端硬件，以及对实时光照需求较低或无光照的游戏及应用。相比于高清渲染管线而言，以轻量级渲染管线作为模板，其功能更简单、清晰，以此为基础进行管线的修改定制化也更加容易，且已足够满足非真实风格化游戏《Doria》的渲染需求。

2.6 Shader Graph

Shader Graph将原本需要手敲代码的工作改成了仅需要对可视化的节点进行连线即可生成着色器 可以通过可视化节点连接的方式来创建所需的着色器，但仅在可编程渲染管线中被支持。Shader Graph的每个节点相当于一个函数方法，输入和输出端口相当于函数的参数和返回值，节点与节点间的连线相当于是赋值，最终生成着色器代码插入到着色器模板的特定位置中形成一份完整的着色器程序。使用Shader Graph可为着色器的创作流程提供额外的便利。

2.7 Post Processing

Post Processing，后处理，是指在前面的渲染的渲染完成后，对该阶段的渲染结果进行再一次的渲染效果处理，比如景深、Bloom效果、动态模糊等。加入后处理效果，也即是在渲染管线绘制出画面的那个阶段后面，再加入一个后处理的流程，经过后处理后最终才绘制出显示在屏幕上被用户看到的画面。

3. 平静水面模拟

3.1 水效果介绍

在游戏中我们经常会看见各种各样的河流、湖泊、大海等水效果，这些水效果不仅可以使得游戏场景更美更真实，一个好的水效果往往也能直接体现游戏的质量。 《Doria》主角诞生于水，水是孕育的地方，起始的场景的设计也离不开水。在默认固定渲染管线下，可以找到许多现成效果不错的水插件，但截止至目前均不支持在可编程渲染管线下正常运行，所以我们尝试在可编程渲染管线下实现适合《Doria》起始场景的水效果。为了能够清晰看见实现流程，着色器部分采用Shader Graph可视化节点连接的方式来制作。

3.2 水的深度表现

当光线到达两个不同介质（水和空气）的边界时，一部分光线会被以入射角相同的角度反射出去，另一部分光线会穿过两个介质的边界，进入到另一个介质的内部中去，最后射入眼睛或相机。吸收和散射光线是让物体着色的关键。

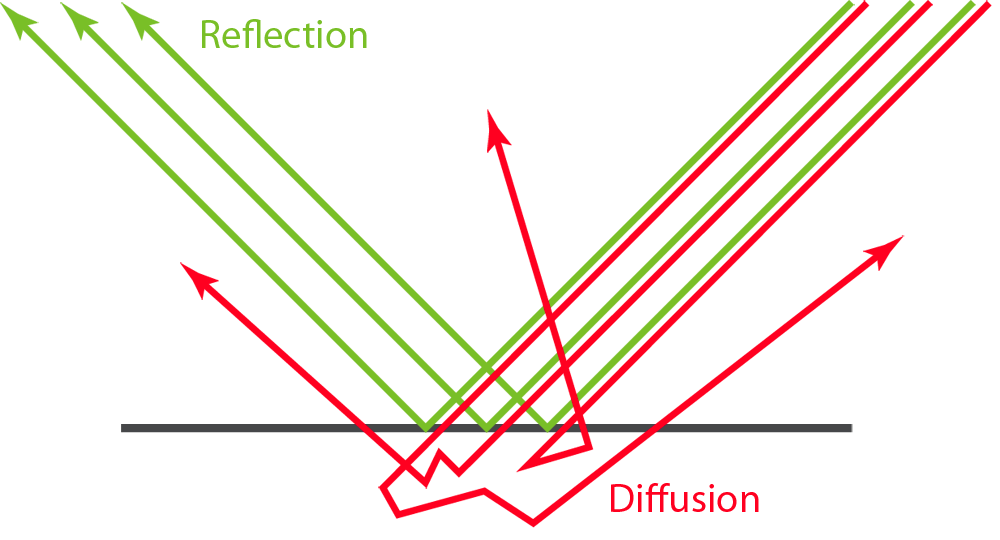


图3.1 光线到达表面的反射与散射

由于水的深度不同，对光线的吸收和散射程度也不同，所以不同深度的水的颜色也是不同的，这儿采用基于主相机视图空间下使用水的深度采样一张简单的颜色渐变贴图来模拟不同深度水的颜色不同的效果。这种方式的原理在于获取已经写入缓存的陆地的深度，也即是主相机的深度图的深度值，与水面的深度进行比较，计算深度差。深度差为0则为水陆交接处，水越深则深度差越大。如图3.2所示。

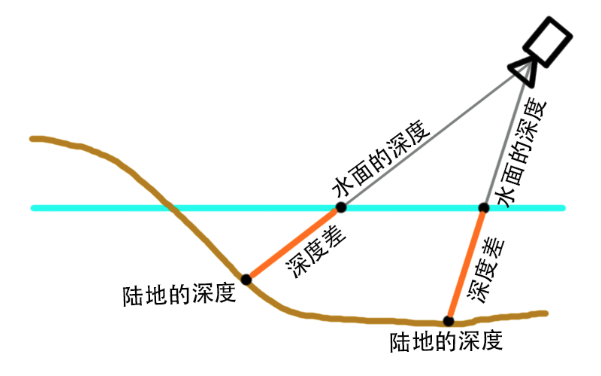


图3.2 深度差的计算

获取主相机深度图，需要先设置相机开启深度图渲染，之后便可在Shader中通过\_CameraDepthTexture 读取到有效的深度图。然后通过屏幕空间对深度图进行采样，获取到每个像素对应的原始深度值。原始深度值是非线性分布的从0到1的值，根据深度缓冲区的不同，精度将是24位或16位。因为我们要获取陆地到相机的深度距离，所以在使用原始深度值前，需要将其转换为线性关系的值。Unity提供了辅助函数LinearEyeDepth来帮助我们将原始深度值转换为视角空间下的线性深度值。

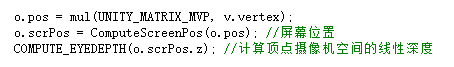


图3.3 顶点着色器中获取屏幕位置作为UV



图3.4 片元着色器对深度图进行采样还原

而在Shader Graph 中，Unity更是便利地提供了Scene Depth节点为我们省去了上述代码的编写。

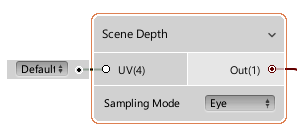


图3.5 Scene Depth节点

获得了陆地到相机的距离后，下一步是计算水面到相机的距离。将水面顶点的世界空间坐标减去相机的世界空间坐标，得到从相机指向水面顶点的向量。因为去掉透视的影响后，深度值是从相机方向出发进行计算的，所以还需将向量与相机方向进行点积运算，得到在相机方向上投影的长度，即为水面到相机的距离。

(3.1)

为了方便进行下一步效果的制作，引入一个新的变量 \_MaxDepth 表示水的最大深度，将水面与陆地深度差除以最大深度，使得在水面在最大深度以内的深度差分布在0到1之间。这样也方便通过0到1的值对水的颜色渐变贴图进行采样，为水的颜色进行上色。

(3.2)

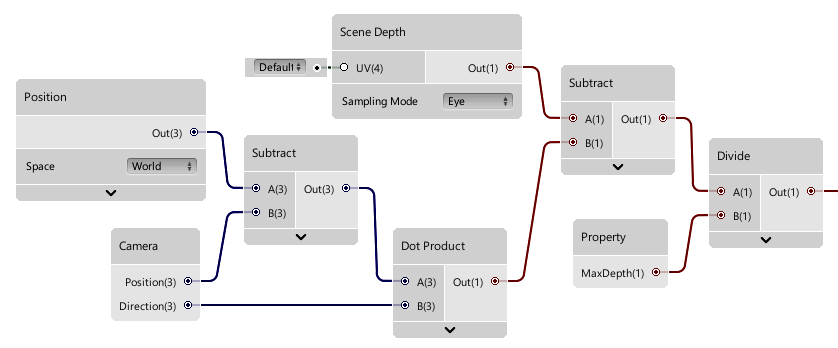


图3.6 Shader Graph计算深度差节点分布图

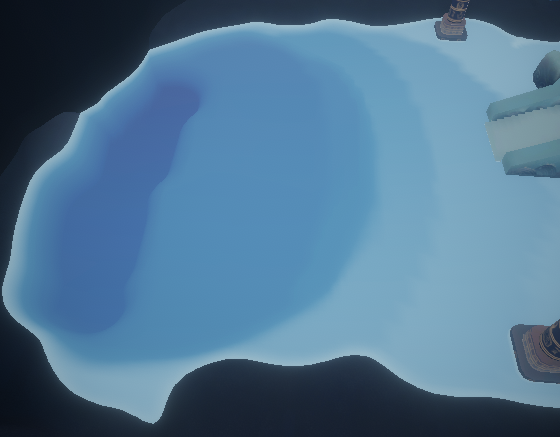


图3.7 带深度的水效果

值得注意的是，上述的方案得到的深度差并非水面到陆地正确的深度，因为在水面的同一点，其深度会随着主相机视角以及陆地斜率的变化而改变，这都不应该是会影响水面到陆地正确深度的因素，但对于进行水面深度的着色，遵循计算机图形学近似原则，看上去对的即是对的，这种方案反而更加接近生活中看到的水随深度颜色变化的感觉。 另一种可以正确获得水面到陆地深度的方案，需要额外创建一个从顶部垂直下看的正交相机，通过世界空间下的坐标换算成UV坐标采样该相机的深度图，与水面进行一次深度差计算即可，广泛运用于需要制作不受主相机视角影响的水面效果，比如冲上陆地的波浪、浮沫等，但由于《Doria》项目没有用到该效果，这里不做详细展开。

3.3 水波效果

水波是体现水的生命所在的精髓，也是体现水是流体的重要特征。一个合适的水波算法模拟水面的起伏是表现水效果所不可或缺的。现有的水系统大多数都是模拟大海的水体，对于需要形成浪头比较尖和浪槽比较宽的粗狂海洋，常用Gerstner波来模拟波形，其特点是顶点向波峰聚集，从而形成比较尖的浪头。然而采用Gerstner波不仅需要使用一堆公式逐顶点进行计算外，还需要进行多次叠加，这将会是一笔昂贵的开销。对于《Doria》在岩洞内相对平静的水体，采用正弦函数叠加来模拟波形制作水面的起伏足以满足我们的需要。

图3.8是2D平面中正弦函数的波形，当我们利用水面的x轴和z轴方向去计算3D空间中的一个平面波形将会得到图3.9的效果。

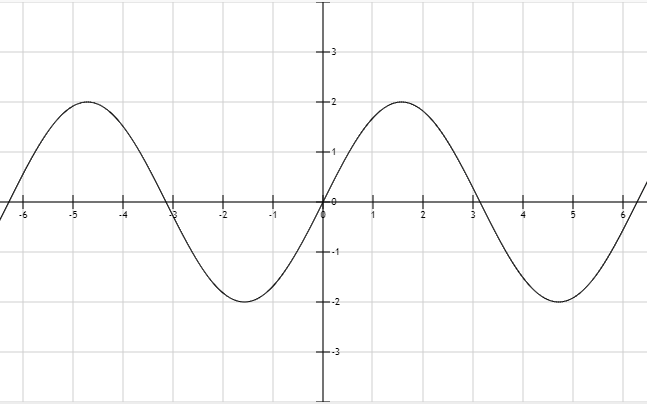


图3.8 2D平面中的正弦波

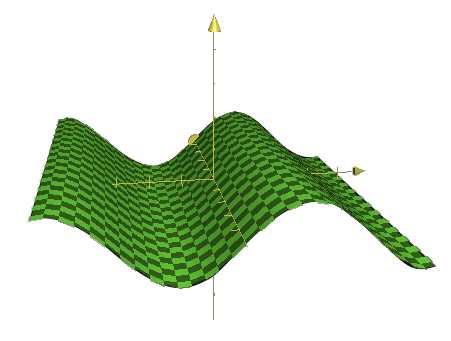


图3.9 3D空间中的正弦波

为了能够控制波形的效果，我们引入WaveRotate控制正弦波的方向，WaveSpeed控制正弦波的频率，WaveCount控制正弦波相位的倍数，WaveHeight控制正弦波的振幅。最终得到y轴上的增量∆y计算公式如下：

(3.3)

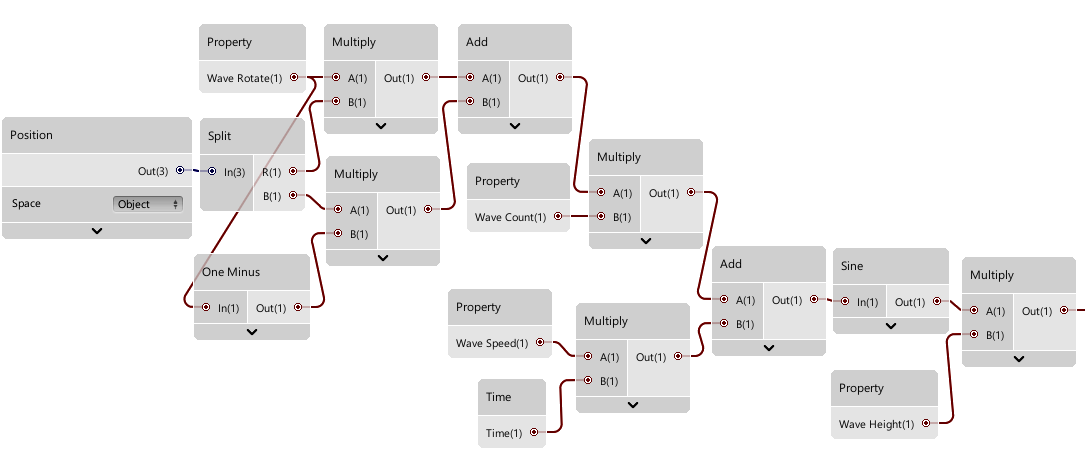


图3.10 Shader Graph计算单个正弦波节点分布图

将上述波形作为正弦波A，以同样的公式，改变其他正弦波的方向、频率、相位倍数及振幅，可以得到正弦波B和正弦波C，将三条波形进行叠加，可以得到图3.11的效果。

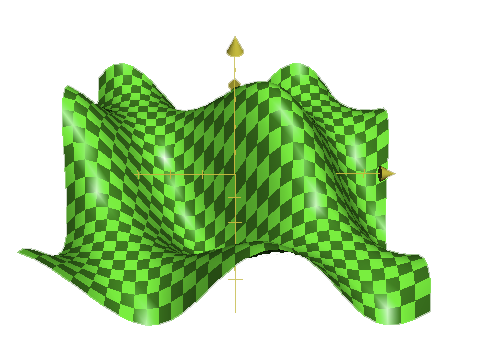


图3.11 最终的波形=正弦波A +正弦波B +正弦波C

最后将水面的顶点y坐标与最终波形的y轴坐标进行相加，得到具有起伏效果的水面如图3.12。为了降低计算的开销，将计算放在顶点函数中进行逐顶点计算，若此时水面的顶点数量偏低，则会导致出现低多边形风格水面的效果，然而这并不是我们想要的，可以通过增加水面的顶点数量来使得波形运动地更加平滑，但同时也会增加计算量。为了达到视觉上较好的波形效果，且同时降低顶点的计算量，我们可以制作多个不同顶点数的面片，以LOD技术控制近距离的水面使用顶点数较多面片，远距离看不出波形细节则可以使用较低顶点数的面片。

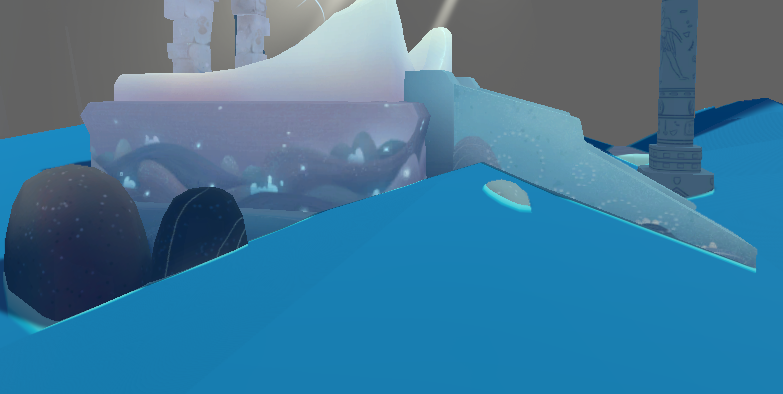
****

图3.12 起伏效果的水面

此时由于我们只通过顶点动画修改顶点的位置，实现水面起伏的效果，并没有修改法线的值，通过简单的光照渲染，也无法实现水面的高光和阴影的区分，最终只能看见一个纯色的平面，为了能过看到水面起伏带来的更加真实的效果，我们还需要通过对波形函数分别求x轴、z轴方向的偏导函数得到切线T（Tangent）和副切线B（Bitangent）的公式。需要注意的是，由于Unity里面的坐标系是左手坐标系，所以对z轴方向的偏导函数表示副切线，对x轴方向的偏导函数表示切线，通过切线与副切线的叉乘，我们可以得到波形的法线N（Normal）。公式如下：

（3.4）

（3.5）

（3.6）

为了优化水波的细节，在正弦函数叠加模拟出较大的波浪后，再加上法线贴图来模拟水面小波浪的细节。通过对法线贴图进行采样，得到的采样值再与正弦波浪的法线进行混合，得到最终的法线值。

最后根据《Doria》的美术风格及相对较静的水面调整参数，获得的效果如图：

****

图3.13 略微起伏流动的水面效果

3.4 水下折射

当光从空气中射入到水中时，传播的方向会发生变化，这种现象是光照射在水上的折射效果。正如光从水底射出到空气中被相机所捕捉一样，再加上水波的影响，故我们从水面上看到的水下景象必定是扭曲且摆动的。制作折射的效果的原理，在于我们需要先得到水面以下景象的图片，然后再渲染水面时扭曲水下景象，再着色到水面上。为此，我们需要改变水面在渲染队列中的位置，确保水面以下的景象是在水面绘制之前先行绘制的。如图3.14所示，我们将水面的绘制放置在了透明队列中，但实际上水面仍是Alpha通道为1的不透明面片。

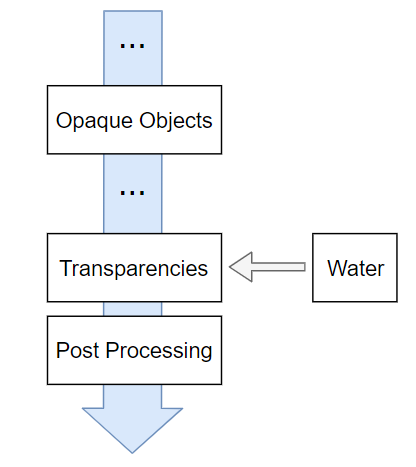


图3.14 水面渲染在渲染管线中的位置

水下的景象，事实上基本都是不透明物体，所以我们可以直接开启主相机的不透明贴图（Opaque Texture），之后可从Shader中通过\_CameraOpaqueTexture 获得水面还没有绘制前，相机所看到的所有不透明物体。Shader Graph也提供了Scene Color节点为我们节省了获取和采样不透明贴图的代码编写工作。



图3.15 Scene Color节点

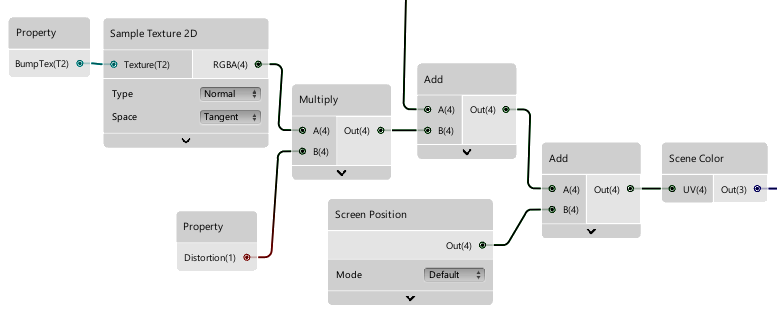
折射的扭曲效果主要受水面的起伏影响最大，而水面的起伏以及方向最直接的体现就在法线上。因此，我们可以直接将屏幕空间坐标加上法线值进行偏移，将偏移后的结果作为UV对不透明贴图进行采样，于是得到了扭曲后的效果。

图3.16 Shader Graph扭曲折射节点分布图



图3.17 水面上扭曲不合理的效果图

如图3.17可以看到水面以下的效果正确的扭曲，然而水面以上的部分也跟着扭曲了，这不符合常理。因此我们可以使用第3.2节计算出来的水面深度图，同样通过加上水面法线值进行采样扭曲处理，将采样出来的深度值进行判断，若小于或等于0则表示物体（或陆地）穿出水面，在水面之上，故该区域不进行扭曲，使用原屏幕空间坐标采样不透明贴图；而深度值大于0的则表示物体（或陆地）处于水面以下，进行正常折射扭曲，使用扭曲的屏幕空间坐标采样不透明贴图。

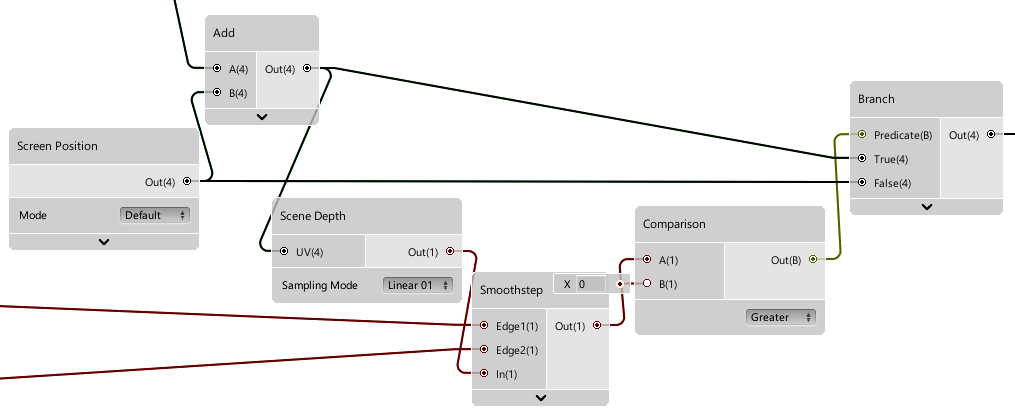


图3.18 Shader Graph修复水面上扭曲节点分布图



图3.19 修复水面上扭曲后的效果图

由于分支判断在着色器中的效率低下，应尽量避免分支判断，所以可以将原屏幕空间坐标和扭曲的屏幕空间坐标使用扭曲后的深度值进行插值运算，同样可以得出正确的折射扭曲效果，同时从水面到水底的扭曲程度，还可以根据深度有从不扭曲到完全扭曲的渐变效果，这正是我们想要的。

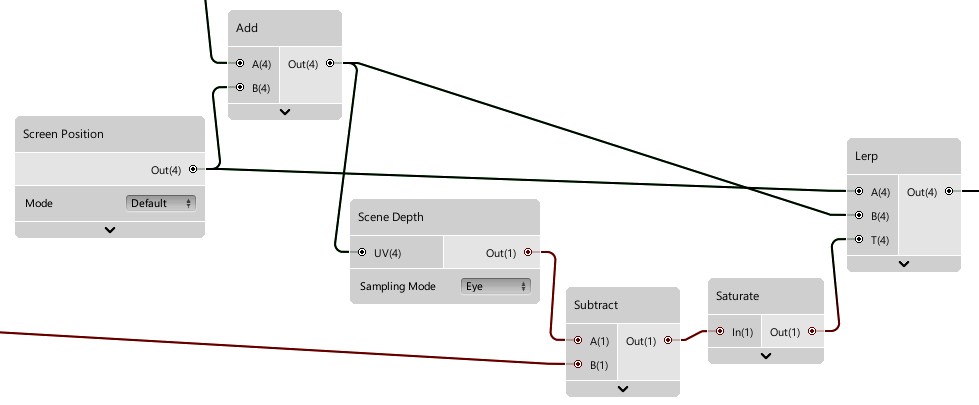


图3.20 Shader Graph采用插值的方式代替分支判断

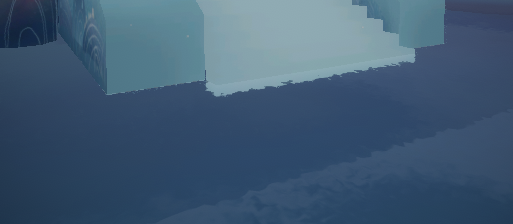


图3.21 水下折射扭曲效果图

3.5 水面反射

反射是光学的重要特征，同时也是渲染中不可或缺的一个效果，特别对于水面而言，反射是提升水效果的最大杀器。然而真正的反射计算是相当复杂的，在实时性要求较高的平台下，比如游戏，不可能进行太过复杂的计算，所以按照图形学的近似原则，仍是有性能友好的反射解决方案。最常见的是使用Cube Map的环境贴图反射方案，以及高级一些的有反射探头（Reflection Probe），还有平面反射方案以及更加复杂的屏幕空间反射（Screen Space Reflection）等。

针对水面，没有太大波浪的水面可以认为是一个光滑的平面，所以平面反射将会是反射效果最好的一个方案。其原理是用一个以指定平面进行反转的相机再一次渲染场景来模拟反射，因此需要将要反射的物体渲染多一次存储在一张贴图中（称为反射图），但控制好渲染层级，将不需要进行平面反射的内容给过滤掉，其性能是可以在接受范围内的。为了能够在渲染水面时对反射图进行采样着色，平面反射的渲染就需要控制在主相机进行渲染之前。这就涉及到我们需要在渲染管线上主相机渲染之前插入一个用于平面反射的相机进行提前渲染，如图3.22，图3.23所示。

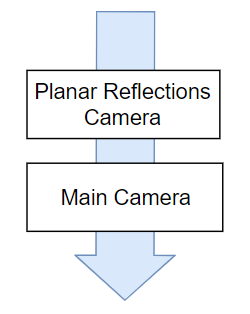


图3.22 平面反射相机在主相机之前渲染

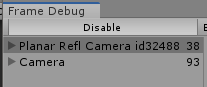


图3.23 Frame Debug中观察到的渲染位置

在可编程渲染管线的基础上，Unity已经为我们封装好了渲染管线管理类（RenderPipelineManager），我们需要在该类的静态事件beginCameraRendering上注册渲染平面反射图的函数 ExecutePlanarReflections()。同时，该事件有两个形参，一个是渲染上下文(ScriptableRenderContext)，另一个是当前渲染的相机（当场景中只有一个主相机时，则该相机即是主相机），如图3.24。

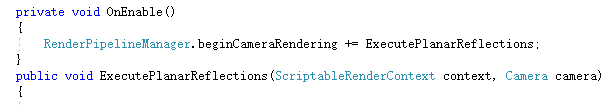


图3.24 注册渲染事件

在 ExecutePlanarReflections() 函数中的代码内容，考虑到画面渲染设置对性能的影响，且平面反射的效果并不需要看到场景或模型特别精细的一面，所以可以将LOD的设置减低一个档次。同时由于平面反射的相机渲染是镜像的，如果只改变了顶点，没有改变法向量和绕序方向，会导致裁剪结果会出现错误。所以需要反转背面剔除，可以将GL.invertCulling设置为true，这个设置将对所有渲染对象的剔除模式进行翻转。



图3.25 渲染前设置

UpdateReflectionCamera()函数用于实时更新用于反射的相机的位置和方向，是平面反射的核心部分，其内容是通过复制当前主相机上的一些设置，创建一个反射相机，通过进行相对于平面进行实时反射计算，更新镜像相机的位置和视角方向。

在数学上，平面可以用点法式来表示，也就是平面可以有平面上的任意一个点和平面的法向量来确定，若假设平面上的任意一点作为一个确定的点P0(x0, y0, z0)，以及平面的法向量N(nx, ny, nz)，则平面内的其他任意一点P(x, y, z)与点P0之间形成的向量PP0满足PP0 · N = 0，展开整理后可得nx\*x + ny\*y + nz\*z + (-nx\*x0 – ny\*y0 – nz\*z0)=0，其中(-nx\*x0 – ny\*y0 – nz\*z0) 为常数，可以用d来表示，则点法式方程可以表示为N·P + d = 0，那么d = - N·P，也即是要表示一个面，仅需要知道平面上的任意一确定的点以及平面的法向量即可。

因此，我们可通过所指定的平面，获取平面上的一点pos及平面的法线方向normal（target即是我们的平面），并根据上面的公式，计算出d值。从几何意义上理解，可以认为d是原点到平面上任意点(pos)在法线(normal)上的投影长度,也就是平面到原点的偏移。同时我们也可以通过ClipPlaneOffset控制这个偏移量。



图3.26 获得平面上一点P和法向量N



图3.27 获得用于表示平面的四个常数

如图3.27，我们获得了表示一个平面所需要的四个常数，也即是法向量和d值，保存在变量reflectionPlane中。根据反射的基本原理，想要求得一个点A相对于平面的反射的A’，如图3.28所示，设N为平面的法线，其为单位向量，A为空间中的一点，A’是A相对于平面的对称点，也即是反射虚像，B为A在平面上的投影点。根据反射的定律可知，|AB| = |A’B|，已知N，A，则A’ = A – 2 \* |AB| \* N。

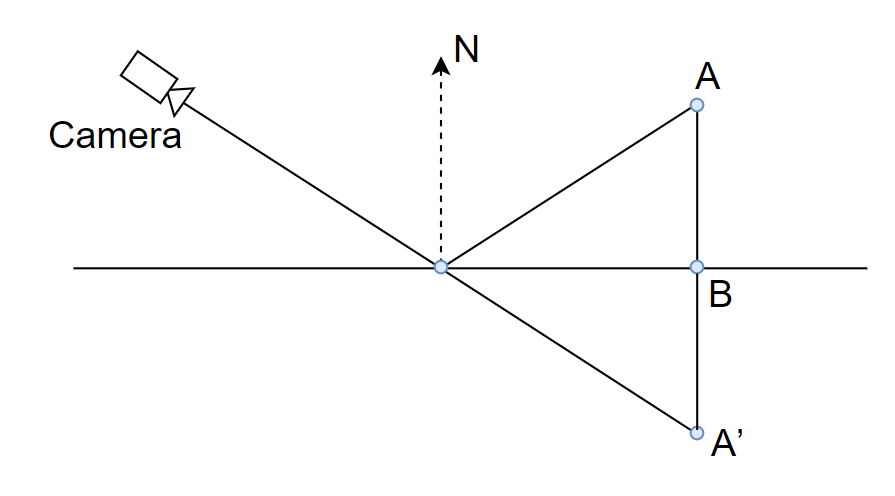


图3.28

所以只要知道空间中任意一点A到平面的距离|AB|，则可以知道平面反射后点A’的位置。如图3.29所示，设平面内任一点P，空间中任意一点A。

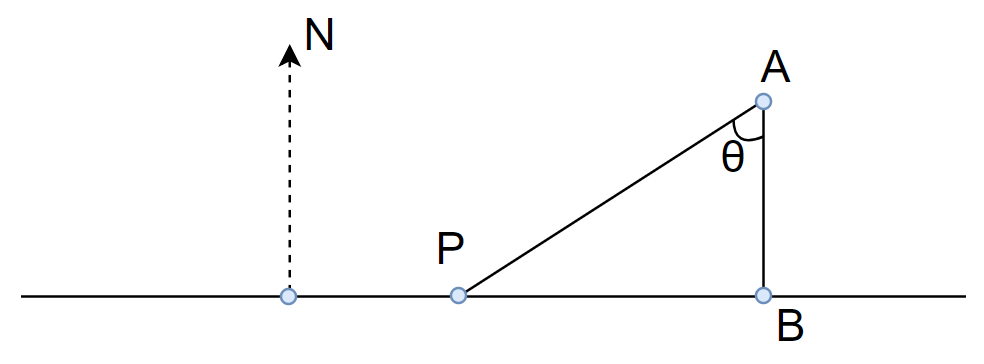


图3.29

则|AB| = |AP| cos θ，且 PA ·N = |PA||N| cos θ，合并可得|AB| = PA·N/|N|，由于N为单位向量，所以|AB| = Dot(PA, N) = Dot(A, N) – Dot(P, N) 。

结合点法式N·P + d = 0可知d = -Dot(N, P)，所以|AB| = Dot(A, N) – d。因此点A平面反射后的点A’ = A – 2(Dot(A, N) + d) \* N。

已知N(a, b, c)，d的值，A(x, y, z)平面反射后的点A’(x’, y’, z’)的计算公式如下：

x’ = x – 2 (x \* a + y \* b + z \* c + d) \* a = (1 - 2a \* a) x + (-2a \* b) y +(-2a \* c) z + (-2d\*a);

y’ = y – 2 (x \* a + y \* b + z \* c + d) \* b = (-2a \* b) x +(1 - 2b \* b) y +(-2b \* c) z + (-2d\*b);

z’ = z – 2 (x \* a + y \* b + z \* c + d) \* c = (-2a \* c) x +(-2b \* c) y +(1 – 2c \* c) z + (-2d\*c);

Unity中的矩阵和OpenGL矩阵风格一致，所以将上述计算公式转换为OpenGL风格的矩阵如图3.30：

图3.30 平面反射矩阵

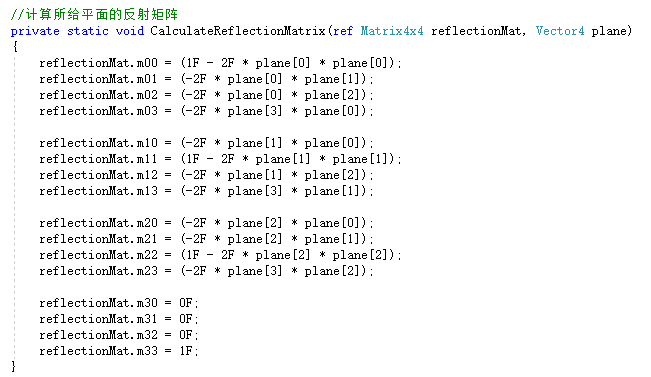


图3.31 代码中的实现

得到平面反射矩阵后，将反射相机从世界空间转为视角空间的矩阵追加上平面反射的运算，同时将反射相机相对于平面进行反转。如图3.32，图3.33所示。

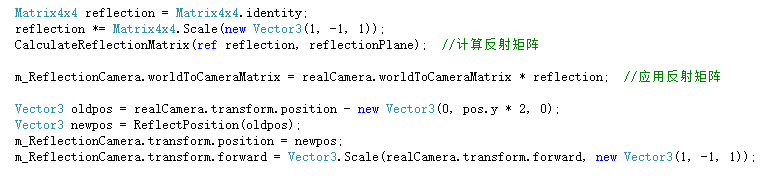


图3.32 反射相机设置部分代码

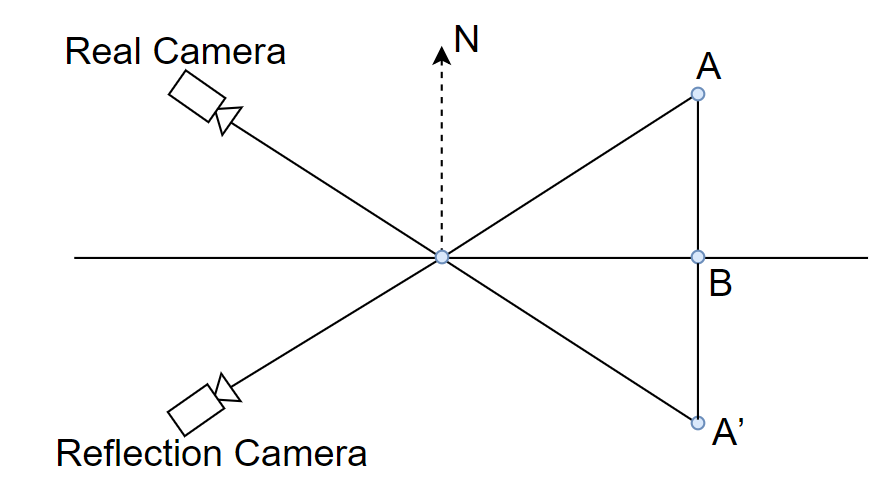


图3.33 反射相机示意图

此时我们已经能够通过反射相机得到了反射效果，然而却还得不到正确的反射结果，如图3.34所示。

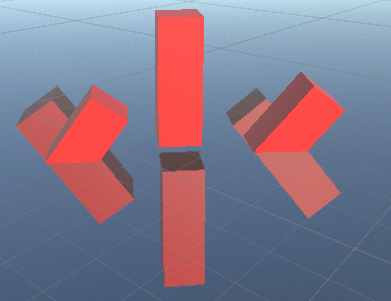


图3.34 错误的反射结果

我们仅需要反射平面以上的部分，而不需要反射平面以下的部分。所以可以在投影变换的时候对平面以下部分进行裁剪处理。

首先需要计算在相机空间下平面（即裁剪平面）的表示，求得其相机空间下的法向量N和d值，如图3.35。

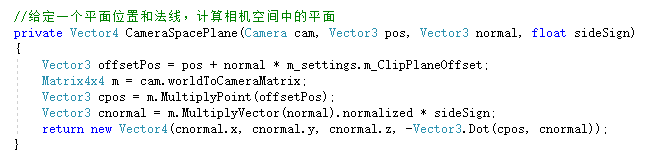


图3.35 计算裁剪平面部分代码

之后可通过CalculateObliqueMatrix() 函数给定这个裁剪平面，返回摄像机将这个裁剪平面设置为近平面的投影矩阵，将该矩阵赋予反射相机的投影矩阵即可，如图3.36，图3.37所示。

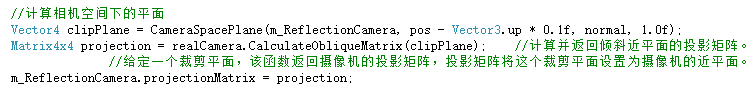


图3.36 设置裁剪平面部分代码



图3.37 正确的反射结果

为了减少不必要的渲染，可以控制好反射所需要渲染层级，如图3.38。



图3.38 设置不渲染的层级

为了能够将渲染的内容可以在着色器其中进行使用，我们将反射相机的targetTexture输出到指定的贴图\_PlanarReflectionTexture中，同时调用渲染管线的API执行单次相机的渲染即可，如图3.39。



图3.39 执行渲染

回到水面着色器中，以第3.4节的同样方法，对屏幕空间坐标进行扭曲采样\_PlanarReflectionTexture贴图，对水面进行着色。效果如图3.40。

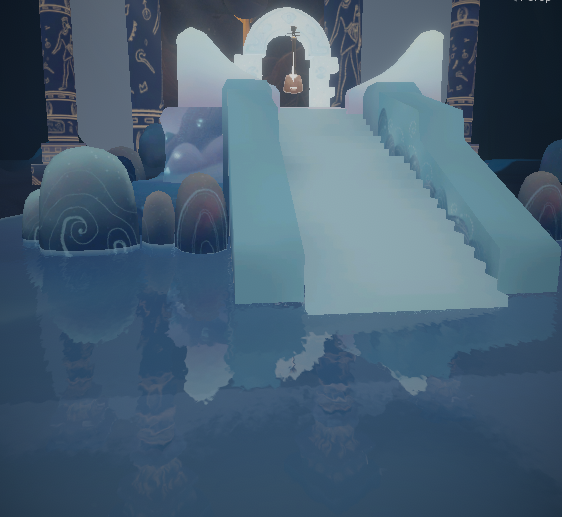


图3.40 平面反射效果图

上图的效果和我们印象中的水仍有那么一点不同，实际生活中，我们站在湖边低头看近距离脚下的水时，会发现是透明的，几乎没有反射，而看向远处的水时，反射效果特别强烈。这在物理上称为菲涅尔效应（Fresnel），简单的讲，就是视线与物体表面的夹角越小，反射效果越明显，当垂直于物体表面时，反射效果最弱。在水面上我们也要加入菲涅尔反射效果。菲涅尔的计算方程比较复杂，也难以求解，所以常用一条有理多项式公式来逼近结果，如式（3.7）。但在Shader Graph中，Unity同样提供了Fresnel Effect节点供我们直接使用。

（3.7）

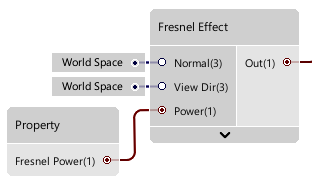


图3.41 Fresnel Effect节点



图3.42 加上菲涅尔后的反射效果

4. 基于高度的垂直雾模拟

4.1 雾效果介绍

雾是在水汽充足，温度较低的情况下，空气中的水蒸气凝结成的微细水滴悬浮于空气中形成的。雾特有的质感，能够带给人一种朦胧和特别的美感，如临仙境，是用于营造场景氛围的绝佳利器。

4.2 后处理垂直雾

在Unity中，实现雾效的基本方式，是使用Unity内置的全局雾效，其原理是根据物体的深度数据，生成基于距离的线性或指数雾效，然而缺点是要为所有的物体添加渲染雾效的代码，实现的效果相当有限，但也是最常见的雾特效。而高级一点的雾特效，其某一点的浓度应该不仅受与相机的距离的影响，同时由于环境因素及重力的存在，距离地面较近的区域雾浓度会相对较高，反之离地面高过一定阈值后，雾浓度又会急剧的下降，这种雾成为垂直雾。在《Doria》的起始场景中，有一处水环境，靠近水面的部分由于水汽较足，所以雾效相对于高处应该更浓，适合使用高级雾特效垂直雾了。

为了减少对每个物体单独绘制雾效带来的性能消耗，同时也为了使雾有种体积感，且可以置身于雾中的任何位置对雾进行观察，所以选择采用基于后处理实现的垂直雾特效，如图4.1。

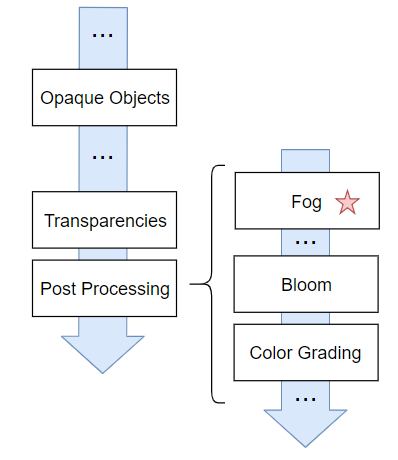


图4.1 雾在渲染管线中渲染位置

第2.7节介绍的Post Processing v2是unity推出的一个开源后处理插件的最新版本，其具有的重写栈和弹性框架，使得开发者可以更加容易的编写自定义效果。自定义后处理效果，分为两个类，一个是设置类，继承自PostProcessEffectSettings，同时通过[PostProcess]标识对渲染类，栈中位置和效果名称进行设置，在该类中定义在Inspector面板中出现的属性；另一个是渲染类，负责调用上下文执行渲染，渲染是放在GPU进行的，所以雾效的实现主要还是在着色器中进行，如图4.2，图4.3。

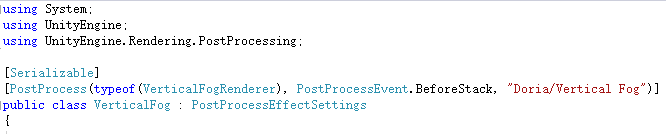


图4.2设置类

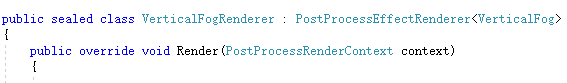


图4.3 渲染类

基于后处理实现垂直雾特效的关键在于，根据深度图计算出每个像素在世界空间下的位置，然后根据高度决定雾的浓度。直白的思路是在片元着色器中逐像素逆推出每一个像素的视角空间坐标，再通过与当前相机的视角\*投影矩阵的逆矩阵相乘转换为世界空间坐标。这种思路的缺点很明显，在片元着色器中进行矩阵乘法是个特别耗性能的一个操作。更好的解决方案则是借用顶点着色器将数据传递到片元着色器会进行插值的特点，由于屏幕后处理的本质是将一个刚好填充满整个屏幕的四边形面片使用特定的材质去渲染，也即是该四边形面片仅有4个顶点对应近裁剪平面的4个角，所以将矩阵运算放在顶点函数中，会极大减少计算的次数，达到更好的性能优化。这种方案重建世界坐标的思路是根据深度图计算某一点像素在世界空间中相对于相机的偏移量，再加上相机在世界空间下的位置，就可以得到该像素在世界空间中的坐标。公式如下：

（4.1）

在透视投影中，同一条视锥体射线下的点，最终都会被绘制在同一个位置，所以只需要计算4个顶点的视锥体射线，通过顶点着色器到片元着色器的插值，就可以得到每个像素的视锥体射线，再乘以线性化后的视角空间坐标下的深度值，就可以得到该像素点到相机的偏移，而相机的位置可以由Unity内置的变量\_WorldSpaceCameraPos访问到，这样我们就可以还原出场景的世界空间坐标了，如图4.4所示。计算的代码如图4.5，图4.6，其中向量o.cameraToFarPlane代表相机到远裁剪片面四个角的方向，同时也包含距离信息，乘以0至1的线性深度值即可获得偏移量。

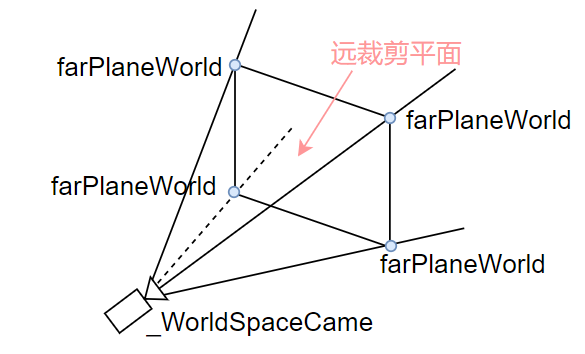


图4.4 示意图



图4.5 顶点着色器部分代码

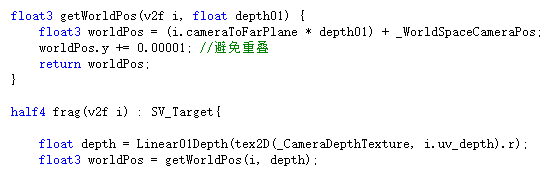


图4.6 片元着色器部分代码

还原出世界空间坐标后，根据高度值来进行雾效果的叠加绘制，效果如图4.7。



图4.7 雾效果图

4.3 雾的流动

我们获得了一个基于高度的均匀雾效，然而这样的雾不够自然，不够飘渺，可以通过一张噪声图来模拟雾效的不均匀，同时加上时间参数，使得雾效有一个流动的效果，代码如图4.8。

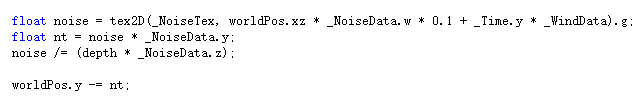


图4.8 雾流动核心代码

通过世界空间坐标xz加上时间的流动对噪声图进行采样获得噪声值，作为高度偏差值与原来雾的高度进行计算，使得雾的高度可以有不均匀的效果。同时也将噪声值作为计算雾效的混合系数使用，使得雾的浓度也能拥有不均匀的感觉。随着时间流动，获得的噪声值也跟着流动，使得雾也有了流动的效果，更加的飘渺好看，如图4.9。



图4.9 流动的雾效果

4.4 距离衰减

按照目前，如果仅仅只靠高度和噪声值决定雾的浓度，那仅适合俯视观察区域才能看到垂直雾最好的效果，倘若置身于雾中，则可能什么都看不清楚，如图4.10 ，为此应该使得雾随着与相机间的距离衰减，越靠近相机，应该越清晰，反之越远的地方雾效应该越浓。



图4.10 身处雾中看不清楚场景

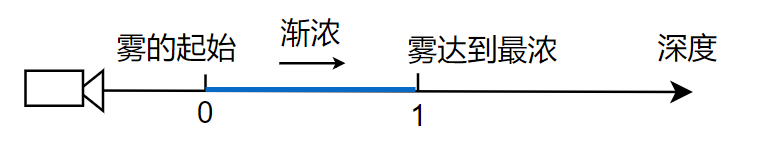


图4.11 示意图

若将雾与相机的距离比作一条轴，如图4.11，在雾效开始衰减前，雾仍是保持原有的浓度，直到随着与相机距离越来越近，雾的浓度逐渐消散到最终消失，对应轴上0到1的变化。所以实际上可以理解为将线性深度距离0到1与衰减距离产生联系，重新映射到从衰减距离开始的0到1的值，在代码中可以使用smootnstep()函数，使得衰减过程有过渡的效果。



图4.12 雾效衰减部分代码

将衰减值与混合系数相乘得到最终的混合系数即可。最终的雾效果如图4.13。



图4.13 拥有距离衰减的雾效

结论

可编程脚本渲染管线的出现，是现代化高品质游戏开发的划时代界线，使得游戏开发者对自己开发的游戏的渲染相关方面拥有更高的掌控空间，可以根据项目的需求对渲染管线进行个性化的定制开发。同时可编程脚本渲染管线也使得游戏效果开发的思路发生了本质性的改变，以往在固定渲染管线的基础上开发一个特定的游戏效果，需要考虑固定渲染管线是否支持这样的效果，遇到无法解决的问题也只能够规避，也即是实现的游戏效果需要依赖于渲染管线。而可编程渲染管线却将这个关系翻转了过来，使得渲染管线可以为游戏效果而服务，这将使得游戏开发者可以实现更多更好的游戏效果，同时也可以针对某些游戏效果进行渲染上及性能上的优化。然而可编程渲染管线的学习门槛也是相对比较高的，相关方面的文档和教程并不多，Unity也是2018年能才提出可编程渲染管线这个东西，直到2019年4月8日才正式发布了第一个非预览版的轻量级渲染管线，本人从2018年的12月份开始接触可编程渲染管线，由于其涉及的知识面比较广，对计算机图形学和引擎渲染底层的基础都有所要求，学习研究渲染管线之前也花了相当长一段时间补充了对应的前置知识，同时由于学习期间可编程渲染管线一直处于测试状态，更新迭代较为频繁，每一个新版本相对于前一个版本都有较为巨大的改变，也造成了学习过程中的一些困难。通过少量的文档了解了可编程渲染管线后，更多的深入学习是通过深入剖析开源项目《Boat Attack》进行的。

本毕业论文于2019年4月底基本完成，而毕业设计作品《Doria》仍处于开发与完善阶段。本论文仅精选了《Doria》中比较具有代表性的两个游戏效果水和雾进行描述，事实上，毕业设计作品的内容远远不止于此，程序设计、角色控制、镜头跟随、动画切换，再到与场景的交互，更多通过自定义着色器实现的游戏效果等，每个点都有值得展开与讨论的地方，但由于篇幅及论文主题的关系，这些点暂时被忽略，后续有机会将另起文章进行讨论。

参考文献

1. 冯乐乐著.UNITY SHADER入门精要[M].北京：人民邮电出版社.2016.
2. Unity Shader-反射效果(CubeMap，Reflection Probe，Planar Reflection，Screen Space Reflection) [EB/OL]. https://blog.csdn.net/puppet\_master/article/details/80808486, 2018-09-11.
3. Unity Manual-Scriptable Render Pipeline[EB/OL]. https://docs.unity3d.com/2018.3/Documentation/Manual/ScriptableRenderPipeline.html, 2018-12-12.
4. 冯开平，刘涛.基于3D Gerstner水波的实时模拟研究[J].工程图学学报.2009,(5)：53-57.
5. Unity Shader：Waveform波形-复合波[EB/OL]. https://blog.csdn.net/liu\_if\_else/article/details/78086749, 2017-09-25.
6. 大萌喵的着色器特效第三发---垂直雾(Vertical Fog)[EB/OL]. https://zhuanlan.zhihu.com/p/24160707, 2016-12-05.
7. BASIC THEORY OF PHYSICALLY-BASED RENDERING[EB/OL]. https://marmoset.co/posts/basic-theory-of-physically-based-rendering/, 2015-11-01.
8. 崔楠，柳有权，陈凯.关于流体模拟细节增强技术的研究[J].计算机与数字工程.2014,(4)：698-701.
9. 龚旭. 液体—气体两相流模型的研究[D].合肥工业大学,2017.
10. 饶志恒. 图形处理器图形管线的研究与实现[D].湖南大学,2011.
11. 张文辉,周小川,钟道欢.实时水面渲染[J].计算机与数字工程,2011,39(03):114-117+154.
12. 杨洋. 游戏引擎中非真实感渲染的研究与实现[D].电子科技大学,2008.
13. 周毓. 游戏中实时渲染的研究与应用[D].北京交通大学,2013.
14. 孙晓鹏,李翠芳.三维游戏中基于OGRE的动态水面模拟算法[J].计算机工程与设计,2011,32(12):4122-4124+4136.
15. RQ Chen.Real-time water rendering based on the graphics hardware acceleration[J].International Conference on Image Analysis & Si... ,?2009 :273-277.2009

致谢

历经数月的研究和论文的撰写，在此刻终于画上了句号。从选题的开始，再到最终的完成，期间遇到了不少了困难，也克服了不少的困难。在这个过程中，非常感谢我的指导老师韦宇炜老师的亲切关怀和悉心指导。自始至终，韦老师都以他严谨的治学精神和科学态度，精益求精的工作作风，深深地感染和激励着我，同时他渊博的学识，广阔的视野，深刻的思想也带给了我无限的启迪和收获，让我打开了很多新的思路。

感谢我的主管张伟先生在我实习期间对我的照顾和安排给我的关于渲染方面的研究工作，以及之后给予了我足够长的假期以完成我的毕设和论文的撰写。同时也感谢我的实习导师张阳先生，以丰富的经验和知识，循循善诱，在我不知方向的时候指出存在的问题和引导我思考解决问题的思路。

感谢我的父母，是他们的支持和鼓励使我走到现在；感谢我的团队成员们，是他们的陪伴以及遇到问题及时的帮助加快了我完成毕设的步伐；感谢我的舍友们，是他们关心，理解和包容，让疲惫的我回到宿舍时也有到了家一般的温暖和放松。

感谢所有帮助过我的老师，同学，朋友们，谢谢你们，请接受我真诚的感谢！