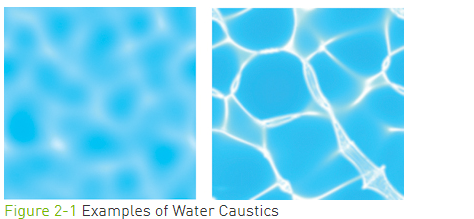
# 第二章 渲染水的焦散

## 介绍

水与光相互作用的方式有些催眠：微妙的反射和折射，光在海底弯曲形成舞动焦散的方式，以及海面无限变化的外观。 请参见图 2-1。 这些现象及其复杂性吸引了物理学领域以及近年来计算机图形学领域的许多研究人员。 模拟和渲染逼真的水，就像模拟火一样，是一项令人着迷的任务。 在实时渲染下取得好的效果并不容易，因此必须经常采取创造性的方法。



焦散是由从曲面反射或折射的光线产生的，因此仅聚焦在接收表面的某些区域。 本章解释了一种以美学导向实时渲染水下焦散的方法。 我们纯粹以美学为导向的方法会将现实主义排除在外。 结果会是一个看起来不错的场景，但可能无法正确模拟物理特性。 正如我们在本章中所展示的，我们方法的结果看起来非常逼真，并且该方法可以在大多数图形硬件上轻松实现。 这种简化的方法已被证明在许多与分形相关的学科中非常成功，例如山和云渲染或树建模。

本章的目的是展示一种用于渲染实时焦散的新技术，从物理基础到实现细节描述该方法。 因为该技术是程序性的，所以它可以优雅地使用高级着色语言的实现。

## 计算焦散

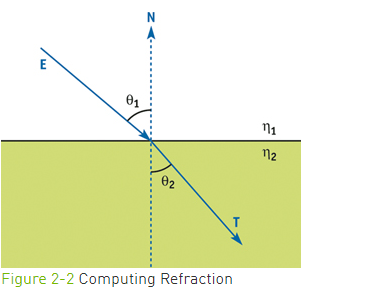
准确计算水下焦散是一个复杂的过程：涉及数以百万计的单个光子，并发生许多相互作用。 为了正确模拟它，我们必须首先从光源（例如，对于海洋场景，太阳）拍摄光子。 这些光子中的一小部分最终与海洋表面发生碰撞，海洋表面要么反射要么折射它们。 让我们暂时忘记反射，看看透射的光子是如何根据斯涅尔定律（Snell's Law）折射的，该定律指出：



或者写做：



在前面的方程中，和是相应材料的折射率，和是入射角和折射角，如图 2-2 所示。 折射率 IOR 可以简单地写为入射光线和折射光线角度的正弦比。



斯涅尔定律不容易用这个公式进行编码，因为它只施加了一个限制，使得折射光线的计算变得非常重要。 假设入射、透射和表面法线是共面的，可以使用各种编码器友好的公式，例如 Foley 等人的公式. 1996 年：



这里T是透射光线，N是表面法线，E是入射光线，和是折射率。

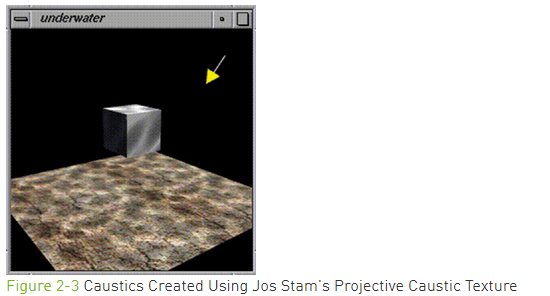
一旦弯曲，光子就会在水中前进，它们的强度会随着它们的深入而减弱。 最终，这些光子中的一些将撞击海底，照亮它。 由于海洋表面的波纹，从不同路径进入水中的光子最终会照亮海底的同一区域。 每当发生这种情况时，我们都会看到由焦散中的光集中产生的亮点，类似于镜头聚焦光的方式。

从模拟的角度来看，焦散通常通过正向或反向光线追踪来计算。 在正向光线追踪中，光子从光源发出并跟照亮整个场景，在地面的离散区域累积它们的贡献。 这种方法的问题在于，许多光子甚至不会与海洋表面发生碰撞，而在实际与海洋表面发生碰撞的光子中，实际上很少会形成焦散部分。 因此，它是一种野蛮的算法，即使由于空间细分而有一些加速。

后向光线追踪的工作方向相反。它从海底开始，按时间倒序向后追踪光线，试图计算给定点的所有入射光的总和。理想情况下，这将通过求解来自被照亮点上方的所有光的半球积分来实现。尽管如此，出于实际原因，积分的结果是通过蒙特卡洛采样解决的。因此，一束候选光线以采样点为中心在半球上的所有方向发送。那些撞到其他物体（如鲸鱼、船或石头）的光线将被丢弃。反观海面的射线，肯定是从外面来的，是不错的候选。因此，它们必须被折射，使用斯涅尔定律的倒数。这些剩余的光线必须在空气中传播，以检验每条假设的光线是否实际上是从光源发出的，或者仅仅是一个错误的假设。同样，只有那些实际上最终击中光源的光线才对焦散有贡献，其余的光线只是作为错误假设而被丢弃。

因此，这两种方法都非常昂贵：实际上只有一小部分计算时间有助于最终结果。 在商用焦散处理器中，有用光线与总光线的比率通常在 1% 到 5% 之间。

Jos Stam 首次探索了实时焦散（Stam 1996）。 Stam 的方法涉及使用波动理论计算动画焦散纹理，因此它可以用来照亮底层。 该纹理与对象的基础纹理相加混合，呈现出漂亮、令人信服的外观，如图 2-3 所示。

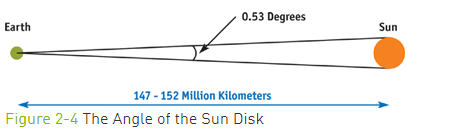


Lasse Staff Jensen 和 Robert Golias 在他们出色的 Gamasutra 论文中探索了另一种有趣的方法（Jensen 和 Golias 2001）。 该论文不仅涵盖焦散，还涵盖了完整的水动画和渲染框架。 该平台基于用于波函数建模的快速傅里叶变换 (FFT)。 最重要的是，他们的方法处理反射、折射和焦散，试图为每个模型建立物理上准确的模型。 因此，Jensen 和 Golias 的焦散方法试图模拟实际过程也就不足为奇了：光线从太阳追踪到波浪网格中的每个顶点。 这些光线使用斯涅尔定律折射，因此产生了新的光线。

## 我们的方法

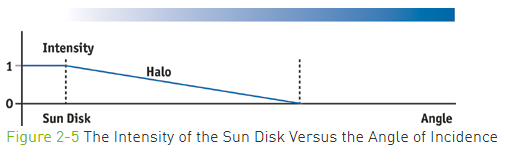
我们用来模拟水下焦散的算法只是上一节中解释的反向蒙特卡洛光线追踪思想的简化。 我们对焦散的良好候选者做出了一些激进的假设，并且我们只计算到达光线的一个子集。 因此，该方法具有非常低的计算成本，并且它产生的东西虽然在物理上“不正确”，但非常类似于真实焦散的外观和行为。 整体效果看起来非常有说服力，焦散线提供的卓越图像质量使其值得使用。

首先，我们假设我们在中午计算赤道上的焦散。 这意味着太阳就在我们的正上方。 为了我们的算法，我们需要计算太阳圆盘所覆盖的天空的角度。 太阳距离地球的距离在 147 到 1.52 亿公里之间，具体取决于一年中的不同时间，它的直径为 142 万公里，相当于太阳盘的角度为 0.53 度，如图 2-4 所示。



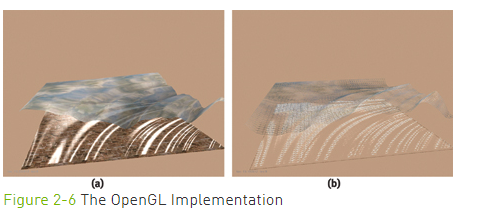
我们做出的第二个假设是，海底被上方垂直发出的光线照亮。 水的透明度在每延米 77% 到 80% 之间，因此每米 20% 到 23% 的入射光被介质吸收，用于加热水。 从逻辑上讲，这意味着当光线从进入水中到到达海底的那一刻传播最短距离时，很容易形成焦散。 因此，焦散对于垂直光线而言将是最大的，并且对于侧向进入水中的光线而言将不那么可见。 这是一个激进的假设，但它是算法成功的关键。

然后我们的算法流程如下。我们从海底开始，就在我们画好地平面之后。然后，第二步，使用additive blended Pass在其上渲染焦散。为此，我们创建了一个与波浪网格具有相同粒度的网格，并给每个顶点将顶点焦散值着色：0 表示无光照； 1 表示一束非常聚焦的光击中海底。为了构建这种照明，使用了反向光线追踪：对于网格的每个顶点，我们将其垂直投影，直到到达位于其正上方的波点。然后，我们使用有限差分计算波在该点的法线。使用矢量和法线，和斯涅尔定律（水的折射率为 1.33），我们可以创建从波传播到空气中的第二条射线。这些射线是为海底带来照明的潜在光线。为了测试它们，我们计算了每一条光线与垂直线之间的角度。因为太阳圆盘距离很远，我们可以简单地用这个角度来衡量照度：越接近垂直方向，从那个方向射入海洋的光就越多，如图 2-5 所示。



## 用OpenGL实现

该算法的初始实现只是简单的 OpenGL 代码，唯一的区别是使用多Pass纹理。 第一个Pass将海底渲染为规则的纹理四边形。 然后，使用我们的焦散生成器按顶点着色相同的细分网格地板，如图 2-6b 所示。 对于细分网格中的每个顶点，我们垂直射出一条射线，将其与海面碰撞，使用斯涅尔定律生成弯曲射线，并使用该射线执行射线四边形测试，我们用它来索引纹理贴图， 如图 2-6a 所示。 最后，该操作与平面环境映射过程没有太大区别。 第三，也是最后一个Pass使用我们的波形生成器渲染海浪。 这些三角形将采样平面环境贴图，因此可以在海浪上获得漂亮的天空反射。 其他效果，例如菲涅耳方程，可以在此基础上实现。



该技术的伪代码如下：

1. 绘制海底。
2. 对于细分网格中的每个顶点：
3. 发送垂直射线。
4. 将射线与海洋的网格碰撞。
5. 逆向使用斯涅尔定律计算折射光线。
6. 使用折射光线计算“太阳”贴图的纹理坐标。
7. 将纹理坐标应用于细分网格中的顶点。
8. 渲染海洋表面。

## 使用高级着色语言实现

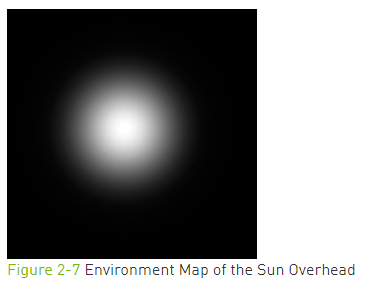
在GPU上完全实现这项技术可以提供更好的视觉质量，并提高其性能。一种高级着色语言，如微软的HLSL或NVIDIA的Cg，允许我们使用类似C的语法将所有这些计算快速移动到GPU。事实上，在基本的OpenGL实现中，之前在CPU上执行的相同波函数只需使用一个包含文件复制到像素和顶点着色器中，并稍作修改，以适应这些高级语言中基于向量的结构。

在每个像素中而不是每个顶点中计算焦散可以提高整体视觉质量，并将效果与几何复杂性解耦。有两种方法将该技术移植到GPU的像素着色器中，都使用波函数的偏导数来生成法线，这与依赖有限差分的原始方法不同。

第一种方法利用了程序纹理可以在屏幕空间中渲染的优点 ，从而在只有一小部分像素可见时节省了渲染时间。不幸的是，这也意味着当大量像素可见时，每一个像素都要做大量的工作，即使添加的细节可能不明显。有时，这样大量的工作会使帧速率降低到不希望的程度，因此需要另一种方法来克服这一限制。

第二种方法在纹理空间中渲染到固定分辨率的渲染贴图。虽然在纹理空间中进行渲染的优点是在每一帧都保持恒定的工作负载，但只渲染可见像素的好处已经失去。此外，很难确定哪个渲染目标分辨率最适合当前场景，如果纹理与像素的比率降得太低，依赖纹理过滤可能会引入双线性过滤交叉模式。

我们可以通过观察使用斯涅尔定律折射入射光线，并简单地根据从水面到地板表面沿波法线的距离进行环境映射之间的视觉差异来进一步优化算法。见图2-7。因此，垂直光线进入纹理的中心，这是明亮的，而角度光线产生逐渐衰减的光源。此外，在更大的深度处，环境贴图的相对大小以及光源的相对大小都会减小，从而产生更清晰的焦散，焦散也会随距离衰减。这种方法可以很容易地将一个标量分量编码成一个非均匀的地面。



因此，该效果计算了波函数的法线，以追踪射线的路径到平面上的截点——在本例中为海底。 为了生成法线，可以很容易地找到波函数在 x 和 y 中的偏导数。

让我们将原始方法中使用的波函数写为：



其中 i 表示用于生成波的octaves； c 1、c 2 和 c 3 是常数，分别给出波的频率、幅度和速度； x 和 y 的取值范围为 0 到 1（含0和1），表示单位正方形上的一个点。

应用链和乘积规则产生两个函数，即偏导数，写为：





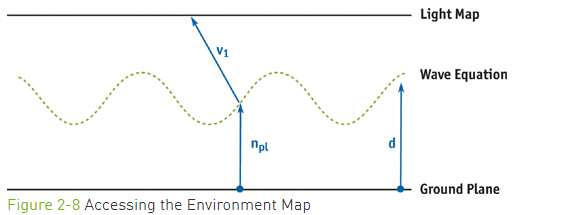
偏导数实际上是梯度向量在评估点的分量。 因为该函数实际上表示高度或 z，所以关于 z 的偏导数只是 1。法线是梯度向量的叉积，所以：



渲染焦散线所需的最后一个方程是线平面截距。 我们可以使用以下方法计算从一条线上的一点到平面上的交点的距离：



其中是水平面到原点的距离，是地平面的法线，是描述折射光方向的向量（有效地沿波面法线方向），是折射光线和海底的交点。 请参见图 2-8。



显然，按像素完成的新计算非常复杂，但考虑到高级着色语言的灵活性和当前硬件设备中可用的像素处理能力，它们实现起来很简单，而且渲染速度很快。 示例 2-1 中的示例代码显示了 Cg 中的实现。

示例 2-1。 波函数、波函数梯度和线平面截距方程的代码示例:

// Caustics

// Copyright (c) NVIDIA Corporation. All rights reserved.

//

// NOTE:

// This shader is based on the original work by Daniel Sanchez-Crespo

// of the Universitat Pompeu Fabra, Barcelona, Spain.

#define VTXSIZE 0.01f   // Amplitude

#define WAVESIZE 10.0f  // Frequency

#define FACTOR 1.0f

#define SPEED 2.0f

#define OCTAVES 5

// Example of the same wave function used in the vertex engine

float wave(float x,float y, float timer)

{

    float z = 0.0f;

    float octaves = OCTAVES;

    float factor = FACTOR;

    float d = sqrt(x \* x + y \* y);

    do{

        z -= factor \* cos(timer \* SPEED + (1/factor) \* x \* y \* WAVESIZE);

        factor = factor/2;

        octaves--;

    } while (octaves > 0);

  return 2 \* VTXSIZE \* d \* z;

}

// This is a derivative of the above wave function.

// It returns the d(wave)/dx and d(wave)/dy partial derivatives.

float2 gradwave(float x,float y,float timer)

{

    float dZx = 0.0f;

    float dZy = 0.0f;

    float octaves = OCTAVES;

    float factor = FACTOR;

    float d = sqrt(x \* x + y \* y);

    do {

        dZx += d \* sin(timer \* SPEED + (1/factor) \* x \* y \* WAVESIZE) \*

            y \* WAVESIZE - factor \*cos(timer \* SPEED + (1/factor) \* x \* y \* WAVESIZE) \* x/d;

        dZy += d \* sin(timer \* SPEED + (1/factor) \* x \* y \* WAVESIZE) \*x \* WAVESIZE - factor \*

            cos(timer \* SPEED + (1/factor) \* x \* y \* WAVESIZE) \* y/d;

        factor = factor/2;

        octaves--;

    } while (octaves > 0);

    return float2(2 \* VTXSIZE \* dZx, 2 \* VTXSIZE \* dZy);

}

float3 line\_plane\_intercept(float3 lineP,float3 lineN,float3 planeN,float  planeD)

{

    // Unoptimized

    // float distance = (planeD - dot(planeN, lineP)) /

    //                    dot(lineN, planeN);

    // Optimized (assumes planeN always points up)

    float distance = (planeD - lineP.z) / lineN.z;

    return lineP + lineN \* distance;

}

一旦我们计算了交点点，我们就可以使用它来读取焦散光照贴图，然后我们可以将焦散贡献添加到我们的地面纹理中，如示例 2-2 所示。

示例 2-2。 最终渲染通道的代码示例，显示了相关的纹理读取操作：

float4 main(VS\_OUTPUT vert,uniform sampler2D LightMap  : register(s0), uniform sampler2D GroundMap : register(s1), uniform float Timer) : COLOR {

            // Generate a normal (line direction) from the gradient

            // of the wave function and intercept with the water plane.

            // We use screen-space z to attenuate the effect to avoid aliasing.

            float2 dxdy = gradwave(vert.Position.x, vert.Position.y, Timer);

            float3 intercept = line\_plane\_intercept( vert.Position.xyz,float3(dxdy, saturate(vert.Position.w)),

                                float3(0, 0, 1), -0.8);

            // OUTPUT

            float4 colour;

            colour.rgb = (float3)tex2D(LightMap, intercept.xy \* 0.8);

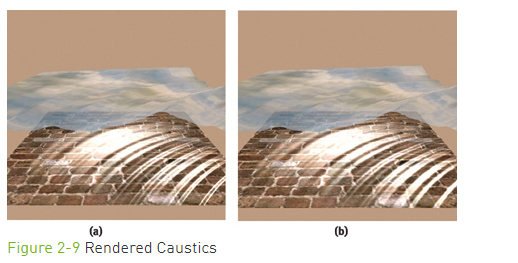
            colour.rgb += (float3)tex2D(GroundMap, vert.uv);

            colour.a = 1;

            return colour;

}

图 2-9 中的结果显示了通过按像素而不是按顶点进行计算所实现的质量改进。 可以看出，足够大的分辨率纹理具有与屏幕空间渲染一样好的质量，而在显示大量像素时不会影响性能。 此外，当使用渲染目标纹理时，我们可以利用自动生成 Mipmap 和各向异性过滤来减少焦散的混叠，而这在渲染到屏幕空间时无法做到。



## 结论

本章中描述的效果展示了如何升级和增强经典算法以利用基于着色器的技术。 随着着色器处理能力的提高，完整的蒙特卡罗方法最终将完全在图形硬件上运行，因此计算物理上正确的焦散将成为现实。

对于一个完整的、美观的水下渲染模型，另一个值得研究的效果是黄昏射线。 这种效应，也称为神射线，看起来像可见的光线，是由水下粒子的反射引起的。