中国科学技术大学

本科毕业论文

基于 GPGPU 和 IFFT 的实时水面仿真

作者姓名：	谭玺扬
学　　号：	PB18010439
专　　业：	信息与计算科学
导师姓名：	陈仁杰 教授
完成时间：	2023 年 5 月 17 日

摘　要

由 Mastin et al.[1] 提出, 经 Tessendorf[2] 推广后的应用**快速 Fourier 变换** (Fast Fourier Transformation, 后文简称 FFT)[3] 进行水面模拟的算法至今仍是电影工业模拟海洋表面的标准解决方案[4]. 这种算法所渲模拟水面真实感出色, 细节丰富, 曾用于电影泰坦尼克号和未来水世界中的海面场景渲染, 但计算量较大, 起初仅能用于离线渲染[5]. 在本文中, 我们借助 Flügge[5] 的想法, 利用 OpenGL API(后文简称 OpenGL) 提供的计算着色器 (Computing Shader) 特性加速 FFT 的计算, 从而实时 实现了基于上述算法的水面渲染. 本文的创新点在于: 利用 OpenGL 的纹理 (Texture) 特性传输数据时, 通过将不同的纹理绑定于一个图像单元 (Image Unit) 中, 分层存取数据, 从而节约了程序执行所需要的内存, 提升了渲染效率.

关键词：计算机图形学、仿真、渲染、并行计算

ABSTRACT

Created by Mastin et al.[1], and generalized by Tessendorf[2], applying the **Fast Fourier Transform** (hereinafter referred to as FFT)[3] algorithm for water surface simulation is still the standard solution in the film industry for simulating the ocean surface[4]. The simulated water surface rendered by this algorithm has excellent realism and rich details, and has been used to render the sea scene in the famous movie *Titanic* and *Water World*. However, the computational cost of this algorithm is relatively large, and it could only be used for offline rendering at first[5]. In this paper, we use the idea of Flügge[5] to use the Computing Shader feature provided by OpenGL API (hereinafter referred to as OpenGL) to accelerate FFT, and realize the fore mentioned water simulation algorithm in real-time. The innovation of this paper is: when using the Texture feature of OpenGL to transmit data, we bind different Textures to a single Texture Unit, and use hierarchical access to data, which saves the memory required for program execution and improves rendering efficiency.

Key Words: Computer Graphics, Simulation, Rendering, Parallel Computing

目　录

符 号 说 明

FFT　快速 Fourier 变换 (Fast Fourier Transformation)

IFFT　逆向快速 Fourier 变换 (Inverse Fast Fourier Transformation)

$\exp\{A\}$　自然常数 e 的 A 次幂[①]

$\|\boldsymbol{x}\|$　$\boldsymbol{x}$ 向量的 Euclid 范数

$\hat{\boldsymbol{x}}$　与向量 $\boldsymbol{x}$ 同向的单位向量

$\sum_{i=1}^{n} a_i$　对 a_i 从 1 到 n 求和

[①]本文中涉及比较复杂的指数 A, 这种情况下记号 $\exp\{A\}$ 相比记号 e^A 更清晰明了, 因此本文均采用前者

第一章　背景简介

第一节　GPGPU 与 OpenGL

一、GPGPU

如今, 随着硬件的不断发展, 计算机中的**图形处理单元 (Graphics Processing Unit, 后文简称 GPU)** 的计算能力得到了显著提升. 最初的 GPU 仅充当视觉处理器 (Visual Processing Unit) 的任务, 图形处理器使显卡减少对中央处理器 (Central Processing Unit, 后文简称 CPU) 的依赖, 并承担部分本来是由 CPU 所承担的工作[6]. **图形处理单元上的通用单元 (General-Purpose Graphics Processing Unit, 后文简称 GPGPU)** 是利用处理图形任务的图形处理器来计算本来本由 CPU 处理的通用计算任务, 这些通用计算任务通常与图形处理没有任何关系[7]. 由于现代图形处理器有强大的并行处理 (Parallel Processing) 能力和可编程流水线 (Programmable Pipeline), 令图形处理器也可以处理非图形数据. 在特定任务下, GPGPU 在运算速度上大大超越了传统的 CPU 应用程序[7]. 下图展示了截至 2015 年 CPU 和 GPU 的浮点运算能力的发展曲线.

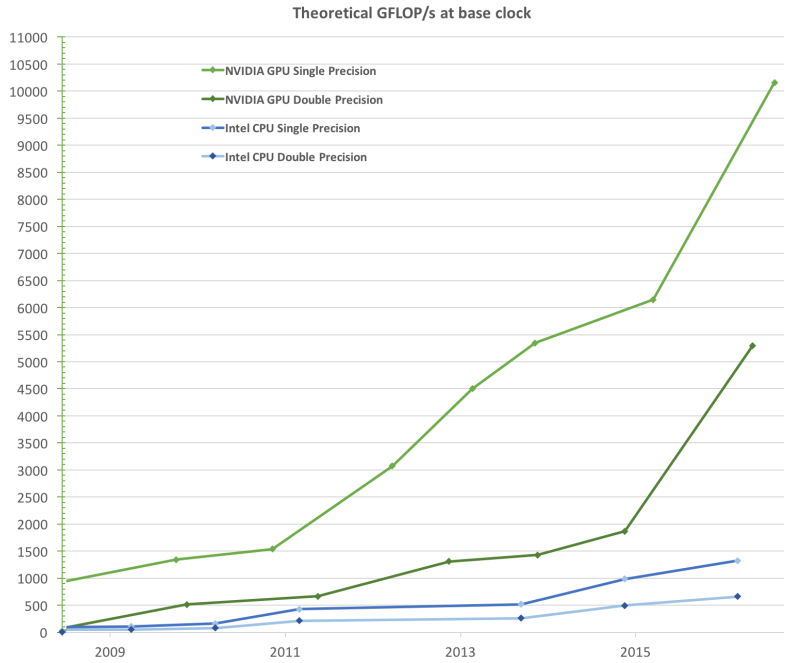
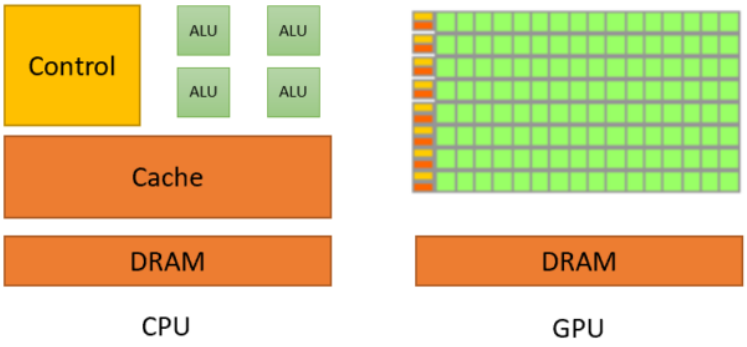


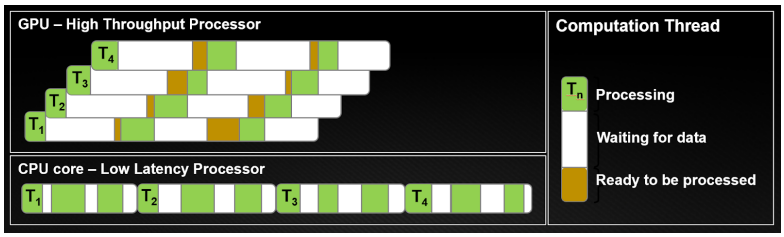
图 1.1　CPU 与 GPU 运算能力对比[8]

GPU 针对高度并行的计算所设计, 又叫做大吞吐量处理器 (Throughput Proces-sors). 许多的科学和 AI 工作负载在其计算过程中固有大规模并行性, 并

在 CPU 上运行速度可能非常慢. GPU 将大部分晶体管用于数据处理, 而 CPU 还需要为大型缓存 (Big Caches), 控制单元 (Control Units) 等预留区域. CPU 处理器的工作原理是最小化每个线程 (Thread) 内的延迟 (latency), 而 GPU 则通过计算来避免指令和内存的延迟. 下图 (a) 展示了 CPU 与 GPU 在内部结构上的区别, 下图 (b) 则展示了两者计算线程的差异. 通过以上对比我们可以看到, GPU 在处理高



(a) GPUs devote more transistors to compute data processing



(b) Low latency or high throughput

图 1.2 CPU vs. GPU:[9]

度并行的任务时相对 CPU 有得天独厚的优势. 这正是我们可以利用这一特性对许多复杂算法进行显著加速的根本原因.

二、OpenGL 与着色器

（1）OpenGL

OpenGL(Open Graphics Library, 译名: 开放式图形库) 是用于渲染 2D, 3D 矢量图形的跨语言, 跨平台的应用程序编程接口 (API). 这个接口由近 350 个不同的函数调用组成, 用来从简单的图形比特绘制复杂的三维景象. OpenGL 常用于 CAD[①], 虚拟现实 (Virtual Reality), 科学可视化程序和电子游戏开发[10].

（2）图形渲染管线与着色器

图形渲染管线 (Graphics Rendering Pipeline) 是利用电脑的图形系统, 把三维模型渲染在二维显示器上的步骤. 简单地说, 在计算机即将显示电子游戏或者

[①]计算机辅助设计 (Computer Aided Design)

三维动画内的三维模型时, 绘图流水线就是把该模型转换为屏幕画面的过程[11].

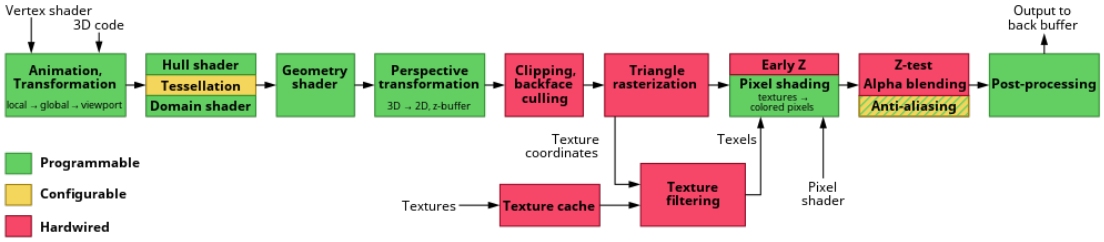


图 1.3　图形渲染管线示意图[11]

着色器 (Shader) 是一种运行在 GPU 上的程序, 这些程序为图形渲染的某个特定部分运行. 从基本意义上来说, 着色器只是一种把输入转化为输出的程序. 着色器也是一种非常独立的程序, 因为它们之间不能互相联通; 它们仅有的沟通只有通过输入以及输出[12].

着色器在图形硬件上计算渲染具有很高的自由度, 可以使用着色器语言对渲染管线进行编程, 调节最终图像的像素, 顶点, 纹理, 色相, 饱和度, 亮度, 对比度等等[13]. 在图形渲染管线中, 负责渲染流程的最基本着色器一般有:

1. **顶点着色器 (Vertex Shader):** 输入单个顶点, 进行坐标转换和顶点基本性质处理;
2. **片元着色器 (Fragment Shader):** 负责计算一个像素的最终颜色, 实现光照, 阴影等效果;
3. **几何着色器 (Geometry Shader)** 将图元映射为屏幕上的像素, 生成供片元着色器使用的片段 (Fragment).

（3）计算着色器

在 4.3 版本及以后的 OpenGL 中增加了**计算着色器 (Compute Shader)** 特性. 计算着色器不充当图形渲染管线的必要成员, 而是利用 GPU 进行一些普适的计算. 虽然它可以完成渲染任务, 但它的一般功能与绘制三角形和像素并无关联[14]. 与其他着色器不同, 计算着色器无法由用户定义输入, 并且没有任何输出. 内置输入仅用来执行特定计算着色器使用的 “空格 (Invocation)” 的位置[14].

（4）GLSL

OpenGL 着色器语言 GLSL(OpenGL Shading Language) 也称为 GLslang, 是一个以 C 语言为基础的高级着色语言. 它提供了开发者对图形渲染管线更多的控制, 而无需使用汇编语言或硬件规范语言[15].

GLSL 在 GPU 中以着色器的形式运行, 控制图形渲染管线的特定步骤, 以达成用户需要的效果. 值得一提的是, GLSL 对向量和矩阵运算的支持是无与伦比

的, 我们可以通过内置的变量来方便的进行各种矩阵以及向量运算, 大大提升了编程效率.

本文中, 我们将利用 OpenGL 的计算着色器实现二维 IFFT 算法.

第二节　快速 Fourier 变换与水面仿真

一、快速 Fourier 变换

快速 Fourier 变换 (Fast Fourier Transformation, FFT) 算法 (Cooley et al.[3]) 在信号处理领域有着广泛的应用, 它利用分治法 (Divide and Conquer) 的原理和离散 Fourier 变换 (Discrete Fourier Transformation, DFT) 中指数因子所具有的对称和周期性质, 使得 DFT 算法可以在 $\mathcal{O}(N \log N)$ 的时间复杂度内完成, 大大提升了运算效率. DFT 提供了信号在时间域 (Time Domain) 到频域 (Frequency Domain) 的转换, 用来计算信号之间的离散卷积 (Discrete Convolution) 和相关性 (Correlation)[5]. IEEE 杂志将 FFT 算法列为 20 世纪影响力最大的十大算法之一, 并将其描述为 “当今处理数字信号和离散数据最为普适的算法[16]”. 需要特别指出的是, Fourier 变换与逆 Fourier 变换 (Inverse Fast Fourier Transformation) 在运算上没有本质区别, 所以对正向 FFT 的算法复杂度讨论同样适用于 IFFT.

二维快速 Fourier 变换 (2D-FFT) 具有良好的并行性[5]. 在本文的水面渲染中, 每一帧需要对特定纹理作 3 次独立的 2D-IFFT, 我们将在后续章节中详细介绍利用 OpenGL 的计算着色器并行计算 2D-IFFT 的细节.

二、水体仿真

真实感的水体渲染和模拟一直以来都是计算机图形学领域的众多核心难点之一. 在水体渲染中, 最核心的部分是波浪的仿真技术, 即如何模拟出逼真的水面波浪的流动变化[4]. 近 50 年内, 出色的水体模拟技术层出不穷. 在这篇文章中, 我们主要应用的是由 Mastin et al.[1] 提出, 经 Tessendorf[2] 推广后的应用 FFT 进行水面模拟的算法. 这种算法也常被各类文献称为基于频谱 (Spectrum-Based) 的方法, 其核心是通过将 2 维 IFFT 作用于海浪频谱 (通常通过测量统计数据获得) 来构造出水体的表面高度场 (Height Field). 其中最为常用的频谱也是 Tessendorf[2] 文章中使用的 Phillips[17] 频谱[4], 也是本文中所采用的频谱.

值得一提的是, 应用 FFT 方法不仅可以实现本文中提及的实时水面模拟, 还

可以用来实现处理后效果 (Post-Processing Effects)[①]和延迟着色 (Deferred Shading)[②]. 图1.4中的 (a) 和 (b) 均展示了由 GPGPU 算法实现的图片后期处理效果, 后文的图6.1, 6.2和6.3展示了本文的 GPGPU 水面渲染结果.

(a) Screenshot of a rendered Scene with post processing bloom, depth of field blur and motion blur effect

(b) Screenshot of a rendered Scene with sun light scattering and lens flare effect

图 1.4 Post-Processing Effects by GPGPU [5]

[①]对已渲染完成的图片施加特殊的效果

[②]一种在屏幕空间内的着色技术, 在几何处理环节完成后加入光线和阴影

第二章　Cooley-Tuckey FFT 算法

在本章节中, 我们详细介绍 Cooley et al.[3] 的 FFT(快速 Fourier 变换) 算法. 假设存在一个信号 $\mathbf{s}$ 的 N 点采样:

$$\mathbf{s} = (\mathbf{s}[0], \mathbf{s}[1], \cdots, \mathbf{s}[N-1]) \tag{2.1}$$

式2.1的 DFT(离散 Fourier 变换) 定义为:

$$\mathbf{S} = (\mathbf{S}[0], \mathbf{S}[1], \cdots, \mathbf{S}[N-1]), \tag{2.2}$$

其中

$$\mathbf{S}[k] := \sum_{n=0}^{N-1} \mathbf{s}[n] \exp\left\{-\frac{2\pi \mathrm{i} kn}{N}\right\}, \quad n = 0, 1, \cdots, N-1. \tag{2.3}$$

如果采用直接计算的方式, 为计算 $\mathbf{S}$ 的每个元素, 我们需要做 N 次乘法与 $N-1$ 次加法, 从而整体的时间复杂度为 $\mathcal{O}(N^2)$. Cooley-Tuckey FFT 算法将这个时间复杂度缩小到 $\mathcal{O}(N \log N)$, 当 N 的值比较大时, 此算法将会使得运算效率大大提升. FFT 算法的一个潜在缺陷是: 为了达到最好的运算效率, 原信号采样数 N 最好是 2 的幂次. 如果 N 不满足此条件, 一种解决方式是在信号中添加 0, 直到达到最近的 2 的幂次. 这种策略也被称为 “Zero Padding”[18]. 在这篇论文里, 方便起见, 我们仅考虑 $N = 2^n$, $n \in \mathbb{N}$ 的情形.

第一节　四点 FFT

作为开始, 我们先探究 $N = 4$ 这样相对简单的情形. 我们注意到:

$$\exp\left\{\frac{\pi \mathrm{i}}{2}\right\} = -\mathrm{i} \tag{2.4}$$

从而当 $N = 4$ 时, DFT 过程 (式2.3) 可以被写为:

$$\mathbf{S}[k] = \sum_{n=0}^{3} (-\mathrm{i})^{kn} \mathbf{s}[n], \tag{2.5}$$

即:

$$\mathbf{S}[k] = \mathbf{s}[0] + (-\mathrm{i})^k \mathbf{s}[1] + (-1)^k \mathbf{s}[2] + \mathrm{i}^k \mathbf{s}[3]. \tag{2.6}$$

我们用矩阵形式写出 $\mathbf{S}^{\mathsf{T}}$:

$$\mathbf{S}^{\mathsf{T}} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -\mathrm{i} & -1 & \mathrm{i} \\ 1 & -1 & 1 & -1 \\ 1 & \mathrm{i} & -1 & -\mathrm{i} \end{pmatrix} \mathbf{s}^{\mathsf{T}} = \begin{pmatrix} \mathbf{s}[0] + \mathbf{s}[1] + \mathbf{s}[2] + \mathbf{s}[3] \\ \mathbf{s}[0] - \mathrm{i}\mathbf{s}[1] - \mathbf{s}[2] + \mathrm{i}\mathbf{s}[3] \\ \mathbf{s}[0] - \mathbf{s}[1] + \mathbf{s}[2] - \mathbf{s}[3] \\ \mathbf{s}[0] + \mathrm{i}\mathbf{s}[1] - \mathbf{s}[2] - \mathrm{i}\mathbf{s}[3] \end{pmatrix}, \tag{2.7}$$

其需要 $\mathcal{O}(N^2)$ 的时间复杂度计算 $\mathbf{S}$. 对于 $N = 4$ 来说, 我们需要做 16 次复数乘法运算, 16 次复数加法运算.

现在我们重新排列 $\mathbf{S}[k]$ 如下:

$$\begin{pmatrix} \mathbf{s}[0] + \mathbf{s}[2] + \mathbf{s}[1] + \mathbf{s}[3] \\ \mathbf{s}[0] - \mathbf{s}[2] - \mathrm{i}\mathbf{s}[1] + \mathrm{i}\mathbf{s}[3] \\ \mathbf{s}[0] + \mathbf{s}[2] - \mathbf{s}[1] - \mathbf{s}[3] \\ \mathbf{s}[0] - \mathbf{s}[2] + \mathrm{i}\mathbf{s}[1] - \mathrm{i}\mathbf{s}[3] \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -\mathrm{i} & -1 & \mathrm{i} \\ 1 & -1 & 1 & -1 \\ 1 & \mathrm{i} & -1 & -\mathrm{i} \end{pmatrix} \begin{pmatrix} \mathbf{s}[0] \\ \mathbf{s}[2] \\ \mathbf{s}[1] \\ \mathbf{s}[3] \end{pmatrix}. \tag{2.8}$$

观察上式, 注意到

$$\begin{pmatrix} \mathbf{s}[0] + \mathbf{s}[2] + \mathbf{s}[1] + \mathbf{s}[3] \\ \mathbf{s}[0] - \mathbf{s}[2] - \mathrm{i}\mathbf{s}[1] + \mathrm{i}\mathbf{s}[3] \\ \mathbf{s}[0] + \mathbf{s}[2] - \mathbf{s}[1] - \mathbf{s}[3] \\ \mathbf{s}[0] - \mathbf{s}[2] + \mathrm{i}\mathbf{s}[1] - \mathrm{i}\mathbf{s}[3] \end{pmatrix} = \begin{pmatrix} (\mathbf{s}[0] + \mathbf{s}[2]) + (\mathbf{s}[1] + \mathbf{s}[3]) \\ (\mathbf{s}[0] - \mathbf{s}[2]) - \mathrm{i}(\mathbf{s}[1] - \mathbf{s}[3]) \\ (\mathbf{s}[0] + \mathbf{s}[2]) - (\mathbf{s}[1] + \mathbf{s}[3]) \\ (\mathbf{s}[0] - \mathbf{s}[2]) + \mathrm{i}(\mathbf{s}[1] - \mathbf{s}[3]) \end{pmatrix} \tag{2.9}$$

若我们提前计算子表达式 $\mathbf{s}[0] + \mathbf{s}[2]$, $\mathbf{s}[0] - \mathbf{s}[2]$, $\mathbf{s}[1] + \mathbf{s}[3]$, $\mathbf{s}[1] - \mathbf{s}[3]$, 则显然我们可以减少复数加法运算的次数为 8 次.

蝶式结 (Butterfly Diagram) 是 FFT 中的组成单位. 其将原本的较大点数的离散 Fourier 运算, 拆成较小点数的离散 Fourier 运算组合, 反之亦然 (将原本点数较小的离散 Fourier 运算，组合成较大点数的离散 Fourier 运算组合)[19]. 下图2.1展示了两点间的蝶式结, 图2.2和图2.3分别展示了以 $\mathbf{s}[0], \mathbf{s}[2]$ 以及以 $\mathbf{s}[1], \mathbf{s}[3]$ 作为输入的蝶式结. 我们将图2.1, 图2.2, 和图2.3中展示的运算过程成为一次**蝶式运算 (Butterfly Computing)**, 其中数据乘以权重再相加的过程 (比如图2.1中得到 $A = a + \alpha b$ 的过程) 称为一次**上碟式运算 (Top Butterfly Span)**; 反之, 数据乘以权重再相减的过程称为一次**下蝶式运算 (Bottom Butterfly Span)**; 所有数据完成一次蝶式运算的过程称为一个**阶段 (Stage)**.

利用分治法 (Devide and Conquer) 的思想, 上一步蝶式运算 (第一阶段) 输出结果可用于下一步蝶式运算 (第二阶段) 的输入, 请参考图2.4.

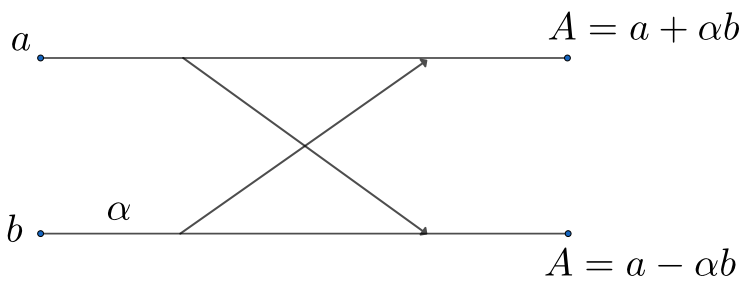


图 2.1　蝶式运算示意图: 输入两个复数 *a*, *b* 以及一个固定的复数 *α*, 输出 *A* 和 *B*

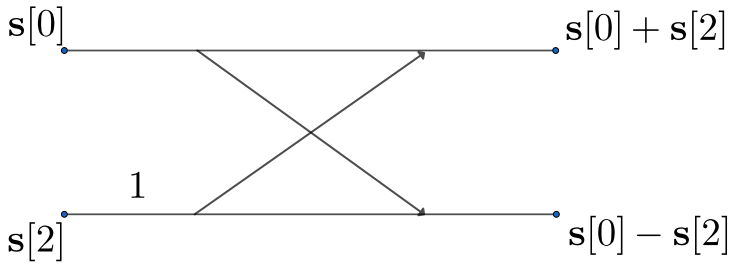


图 2.2　输入 **s**[0] 与 **s**[2] 的蝶式结

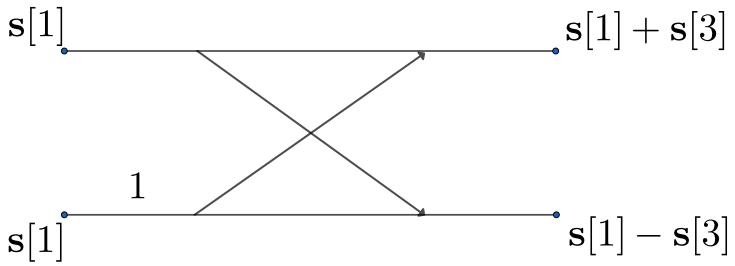


图 2.3　输入 **s**[1] 与 **s**[3] 的蝶式结

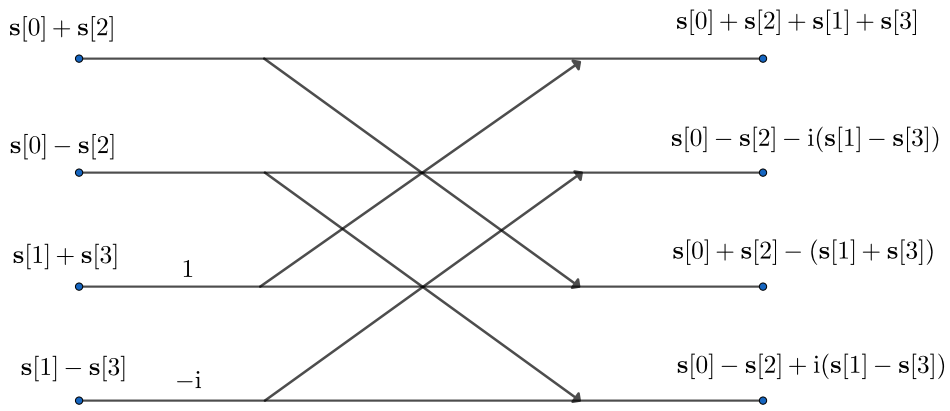


图 2.4　四点 FFT 算法的第二阶段

我们将图2.2, 图2.3和图2.4结合起来, 得到:

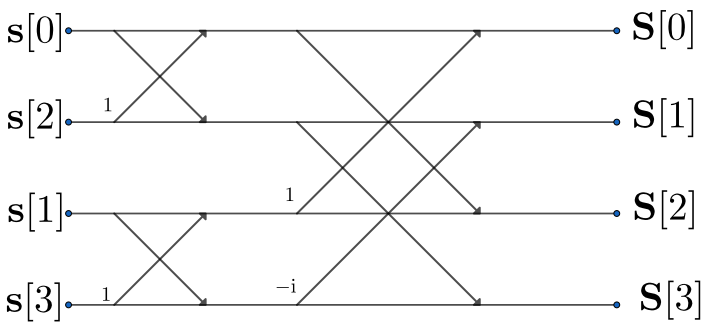


图 2.5　四点 FFT 蝶式运算全过程

第二节　从四点 FFT 推广到 N 点 FFT

在上一节的讨论中, 我们将 $N=4$ 情形的 FFT 分为两个蝶式运算阶段: 第一个阶段引入常量 $\alpha=1$(回顾图2.1), 用其计算 $\mathbf{s}[0]\pm\mathbf{s}[2]$ 以及 $\mathbf{s}[1]\pm\mathbf{s}[3]$, 第二个阶段引入常量 $\alpha=-\mathrm{i}$, 用其得到最终的 $\mathbf{S}[0]\sim\mathbf{S}[3]$. 对于一般的 N 点 FFT, 全过程分为 $\log_2 N$ 个阶段, 每一个阶段接收 N 个复数作为输入 (它们是上一阶段的输出, 或者为原始输入 $\mathbf{s}$), 并执行 $N/2$ 次蝶式运算, 得到的 N 个输出结果将作为下一个阶段的输入, 若为最后一个阶段则输出 $\mathbf{S}$. 由于每一个阶段中包括了 $N/2$ 次蝶式

运算, 每次蝶式运算包含两次复数加法, 两次复数乘法, 以及一次符号转换 (数据乘以 -1), 每一个阶段将在 $\mathcal{O}(N)$ 时间复杂度内完成. 一共分为 $\log_2 N$ 个阶段, 则总共的时间复杂度是 $\mathcal{O}(N \log N)$.

正如我们在第 一 节中所见, FFT 算法可以用蝶式结的方式很好的诠释. 对于 N 点 FFT, 我们在第 一 节中式2.4的简化方式变得不再适用. 因此, 参考 Lyons[20] 的记号, 我们引入**旋转因子 (Twiddle Factor)**:

$$\mathcal{W}_N^k := \exp\left\{-\frac{2\pi k\mathrm{i}}{N}\right\}. \tag{2.10}$$

注意到 $N = 4$ 时, 旋转因子与式2.4完全一样. 我们进而可以利用旋转因子将式2.8的右侧写为:

$$\begin{pmatrix} \mathcal{W}_4^0 & \mathcal{W}_4^0 & \mathcal{W}_4^0 & \mathcal{W}_4^0 \\ \mathcal{W}_4^0 & \mathcal{W}_4^2 & \mathcal{W}_4^1 & \mathcal{W}_4^3 \\ \mathcal{W}_4^0 & \mathcal{W}_4^0 & \mathcal{W}_4^2 & \mathcal{W}_4^2 \\ \mathcal{W}_4^0 & \mathcal{W}_4^2 & \mathcal{W}_4^3 & \mathcal{W}_4^1 \end{pmatrix} \begin{pmatrix} \mathbf{s}[0] \\ \mathbf{s}[2] \\ \mathbf{s}[1] \\ \mathbf{s}[3] \end{pmatrix}. \tag{2.11}$$

注意到 $\mathcal{W}_N^k$ 的周期性:

$$\begin{aligned} \exp\left\{\mathrm{i}\frac{2\pi k}{N}\right\} &= \exp\left\{\mathrm{i}\frac{2\pi N + k}{N}\right\}, \\ \Longrightarrow\ \mathcal{W}_N^k &= \mathcal{W}_N^{k+m\cdot N}, \quad \forall m \in \mathbb{Z}, \end{aligned} \tag{2.12}$$

此矩阵表示可以进一步简化为:

$$\begin{pmatrix} \mathcal{W}_4^0 & \mathcal{W}_4^0 & \mathcal{W}_4^0 & \mathcal{W}_4^0 \\ \mathcal{W}_4^0 & \mathcal{W}_4^2 & \mathcal{W}_4^1 & \mathcal{W}_4^3 \\ \mathcal{W}_4^0 & \mathcal{W}_4^4 & \mathcal{W}_4^2 & \mathcal{W}_4^6 \\ \mathcal{W}_4^0 & \mathcal{W}_4^6 & \mathcal{W}_4^3 & \mathcal{W}_4^9 \end{pmatrix} \begin{pmatrix} \mathbf{s}[0] \\ \mathbf{s}[2] \\ \mathbf{s}[1] \\ \mathbf{s}[3] \end{pmatrix}. \tag{2.13}$$

除周期性外, 旋转因子还具有以下性质:

$$-\mathcal{W}_N^k = \mathcal{W}_N^{k+\frac{N}{2}}, \tag{2.14}$$

这使得我们在进行蝶式运算时, 统一用旋转因子作为系数, 而不用额外进行符号反转. 以上关于旋转因子性质的证明是容易的, 将定义带入直接验证即可. 关于旋转因子更多的知识, 读者也可以参考 Lyons[20] 或 Osgood[18].

引入旋转因子后, 我们可以将图2.5改为:

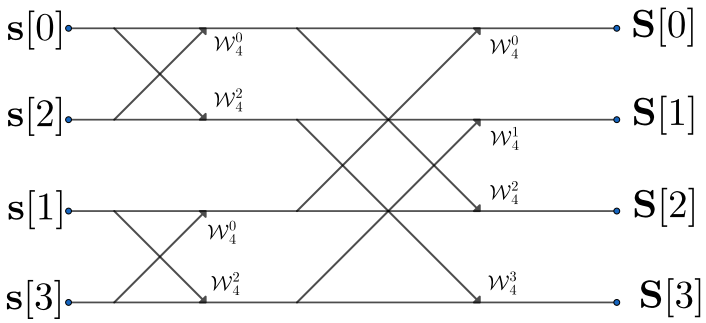


图 2.6　四点 FFT 蝶式结流程图

对于四点 FFT 运算, 我们将原信号的顺序改为 **s**[0], **s**[2], **s**[1], **s**[3]. 当对于长度为 $N(N = 2^n)$ 的原信号 **s**, 对原信号的调整遵循**位反转 (Bit-Reversed)** 规律, 这里不作详细介绍, 有兴趣的读者请参考附录中算法A.1, 或者阅读 Osgood[18] p. 289~291. 另外, 旋转因子的幂次 k 满足:

$$k = n \cdot \frac{N}{2^{\text{stage}}}, \tag{2.15}$$

这里 n 为当前结点在蝶式结流程图竖直方向上的位置, stage 为当前所处的阶段.

在本章的最后, 我们给出 $N = 16$ 时的完整 FFT 蝶式结流程图供读者参考.

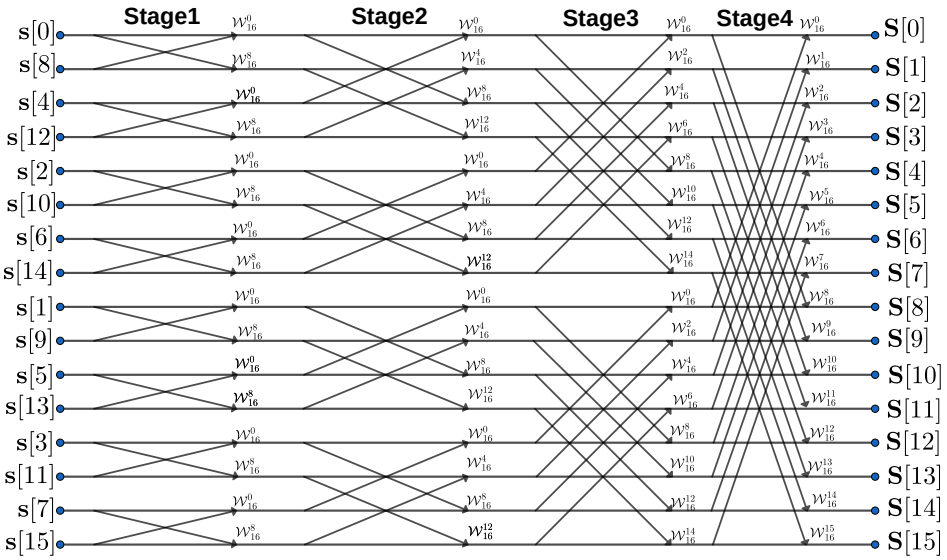


图 2.7　$N = 16$ 时的 FFT 蝶式结流程图

第三章　基于 DFT 的水面仿真模型

本章将详细介绍 Tessendorf[2]用 FFT 算法模拟水面高度场的算法. 在这篇论文中, 我们考虑一个均匀划分的 $N \times N$ 正方形三角网格作为水面的初始形态 (与上一章类似的, $N = 2^n$). Tessendorf[2]的算法将计算出给定 t 时刻网格各格点的高度, 即给出随时间变化的高度场 (Height Field). 作为约定, 我们规定水平面 (即初始网格所在平面) 为 xz 平面, 水平面上的一点 $\boldsymbol{p}$ 可由坐标 (x, z) 唯一表示, 即 $\boldsymbol{p} = (x, z)$.

根据 Tessendorf[2], 水平面上任意一点 $\boldsymbol{p}$ 在时刻 t 的高度可以表示为:

$$h(\boldsymbol{p}, t) = \sum_{\boldsymbol{k}} \tilde{h}(\boldsymbol{k}, t) \exp\{\mathrm{i}\boldsymbol{k} \cdot \boldsymbol{p}\}. \tag{3.1}$$

其中:

1. **波动向量 (Wave Vector)**$\boldsymbol{k}$ 是一个二维水平向量, 指向水波前进的方向. 它由两个分量组成: $\boldsymbol{k} = (k_x, k_z)$, 并定义为:

$$\begin{cases} k_x = \dfrac{2\pi(n - N/2)}{L} \\ k_z = \dfrac{2\pi(m - N/2)}{L} \end{cases}, \quad 0 \leqslant n, m \leqslant N - 1. \tag{3.2}$$

   其中 L 是正方形湖面的边长.

2. **Fourier 振幅分量 (Height Amplitude Fourier Components)**$\tilde{h}(\boldsymbol{k}, t)$ 由正态随机数和空间频谱 (Spacial Spectrum) 共同产生. 具体而言, $\tilde{h}(\boldsymbol{k}, t)$ 由 Phillips 频谱 $P_h(\boldsymbol{k})$ 结合独立的服从正态分布的扰动产生[2], 其中

$$P_h(\boldsymbol{k}) = A \frac{\exp\left\{\frac{-1}{(\|\boldsymbol{k}\| E)^2}\right\}}{\|\boldsymbol{k}\|^4} |\hat{\boldsymbol{k}} \cdot \hat{\boldsymbol{w}}|^2, \tag{3.3}$$

   这里:

   - $$E = V^2/g, \tag{3.4}$$

     V 是风速, 可由用户指定; g 是重力加速度, 我们取 $g = 9.8\ \mathrm{m/s}^2$;
   - $\|\cdot\|$ 表示向量的 Euclid 范数;
   - $\hat{\boldsymbol{w}}$ 是风向 $\boldsymbol{w}$ 同向的单位向量, $\hat{\boldsymbol{k}}$ 同理;
   - A 是可以用户指定的放大因子;

- $|\hat{\boldsymbol{k}} \cdot \hat{\boldsymbol{w}}|^2$ 项使得与风向垂直的水波无法造成影响, 另外, 对于尺度极小的的水波 ($l \ll L$), 通过额外乘以一个 $\exp\{-\|\boldsymbol{k}\|^2 l^2\}$ 项可以去除他们的影响.

我们引入 **Fourier 振幅 (The Fourier Amplitudes)**$\tilde{h}_0(\boldsymbol{k})$, 其定义为:

$$\tilde{h}_0(\boldsymbol{k}) = \frac{1}{\sqrt{2}}(\xi_r + \mathrm{i}\xi_i)\sqrt{P_h(\boldsymbol{k})}, \tag{3.5}$$

其中 ξ_r 和 ξ_i 是服从标准分布 (均值为 0, 方差为 1 的正态分布) 的两个独立随机变量.

最后, 我们定义**色散关系 (Dispersion Relation)** 如下:

$$w^2(\boldsymbol{k}) = g\|\boldsymbol{k}\|. \tag{3.6}$$

根据 Tessendorf[2], 色散关系描述了水波频率和高度的关系, g 仍为重力加速度. 结合色散关系, 我们可以给出 Fourier 振幅分量 $\tilde{h}$ 的最终表达式:

$$\tilde{h}(\boldsymbol{k}, t) = \tilde{h}_0(\boldsymbol{k}) \exp\{\mathrm{i}w(\boldsymbol{k})t\} + \tilde{h}_0^*(-\boldsymbol{k}) \exp\{-\mathrm{i}w(\boldsymbol{k})t\}. \tag{3.7}$$

其中 $w(\boldsymbol{k}) = \sqrt{g\|\boldsymbol{k}\|}$, $\tilde{h}_0^*$ 表示复数 $\tilde{h}_0$ 的复共轭.

最后需要注意的是: 式3.1实际上是对二维信号 $\tilde{h}(\boldsymbol{k}, t)$ 的二维 DFT. 这也是我们可以利用 FFT 算法的原因.

接下来, 在式3.1的基础上, 结合式3.2我们给出二维 DFT 的计算式:

$$h(x, z, t) = \frac{1}{N^2} \sum_{n=0}^{N-1} \sum_{m=0}^{N-1} \tilde{h}(n, m, t) \exp\left\{\mathrm{i}\frac{2\pi nx - \pi Nx}{N}\right\} \exp\left\{\mathrm{i}\frac{2\pi mz - \pi Nz}{N}\right\}. \tag{3.8}$$

注意到 $\exp\{\mathrm{i}\pi\} = -1$, 我们可以继续简化:

$$h(x, z, t) = \frac{1}{N^2} \sum_{n=0}^{N-1} \sum_{m=0}^{N-1} \tilde{h}(n, m, t)(-1)^n \exp\left\{\mathrm{i}\frac{2\pi nx}{N}\right\} (-1)^m \exp\left\{\mathrm{i}\frac{2\pi mz}{N}\right\}. \tag{3.9}$$

我们将求和顺序交换:

$$h(x, z, t) = \frac{1}{N^2}(-1)^n \sum_{n=0}^{N-1} \left[(-1)^m \sum_{m=0}^{N-1} \tilde{h}(n, m, t) \exp\left\{\mathrm{i}\frac{2\pi zm}{N}\right\}\right] \exp\left\{\mathrm{i}\frac{2\pi xn}{N}\right\}. \tag{3.10}$$

式3.10也展示了一个二维 DFT 计算可以被转化为两个一维 DFT 计算的复合.

第四章　光照模型

本章节中, 我们介绍 Blinn-Phong 反射模型. 我们将利用此光照模型进行最终的水面渲染. Blinn-Phong 反射模型 (Blinn-Phong Reflection Model) 是 Blinn[21] 在 Phong[22] 基础上改进发展出的一种光照模型. 它提供了一种基于光线反射的对场景的近似, 虽然其在物理上并非是正确的, 但仍具有相当不错的模拟效果.

Phong 反射模型 (Phong Reflection Model)[22] 认为, 场景的光学结构由三个分量组成:

- **泛光 (Ambient):** 即使在黑暗的情况下, 世界上也通常有一些光亮 (比如来自月亮或很远处的光), 物体并不会是完全黑色的. 因此, 我们用这个常量来模拟这个特性;
- **漫反射 (Diffuse):** 光源对物体的方向性影响 (Directional Impact). 它是 Phong 光照模型中最显著的分量, 物体的某一部分越是正对着光源, 它就越亮;
- **镜面 (Specular):** 模拟有光泽的物体表面上出现的亮斑. 可以理解的是: 在水面的渲染中, 我们并不需要镜面光照的计算.

我们在水面渲染中仅考虑前两项. 另外, 为了使结果更逼真, 我们还需要考虑 Fresnel 效应[①].

第一节　泛光项

我们可以非常容易地添加环境光照, 只需要光的颜色乘以一个很小的常量环境因子, 再乘以物体的颜色, 就可以得到环境光照项:

$$I_{\text{env}} = K_a I_a \tag{4.1}$$

其中 K_a 代表物体表面对环境光的反射率, I_a 代表入射环境光的亮度, I_{env} 存储结果, 即人眼所能看到的从物体表面反射的环境光的亮度, 参考图4.1.

第二节　漫反射光照

这里, 我们简单地假定所有光线均满足反射定律, 即入射光线和出射光线, 法线均处于同一平面内; 出射角等于入射角.

[①] 当视线垂直于物体表面时, 光的反射较弱, 而当视线非垂直时, 夹角越小, 反射越明显

图 4.1　泛光项效果

Phong 模型认为, 只有当入射光线与平面垂直的时候才能完整的接受所有光的能量, 而入射角度越倾斜能量损失越大. 具体来说, 我们应该将光强乘上一个 $\cos\theta = \boldsymbol{I} \cdot \boldsymbol{n}$, 其中 $\boldsymbol{I}$ 是入射光方向, $\boldsymbol{n}$ 是平面法线方向. 参考图4.2. 除了入射角度

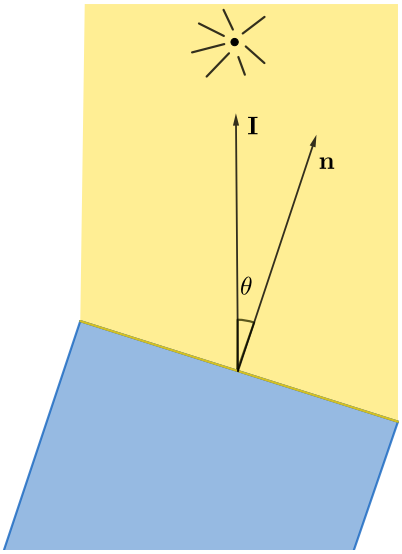


图 4.2　每个单位光照与 $\cos\theta = \boldsymbol{I} \cdot \boldsymbol{n}$ 成正比

以外, 光源与照射点之间的距离也应当被考虑. 我们可以简单地认为: 距离光源 r 的点处的光强为 I/r^2. 参考图4.3.

结合之前所讲: 我们可以得到:

$$L = I_{\text{env}} + k_d(I/r^2)\max\{0, \boldsymbol{n} \cdot \boldsymbol{I}\}. \tag{4.2}$$

其中 k_d 是漫反射系数, I 为入射光强, $\boldsymbol{n}$ 和 $\boldsymbol{I}$ 分别表示法向和入射方向. max 函数

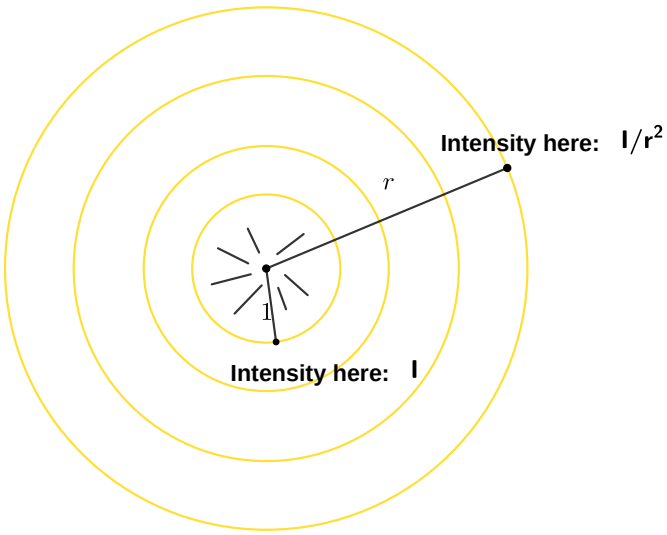
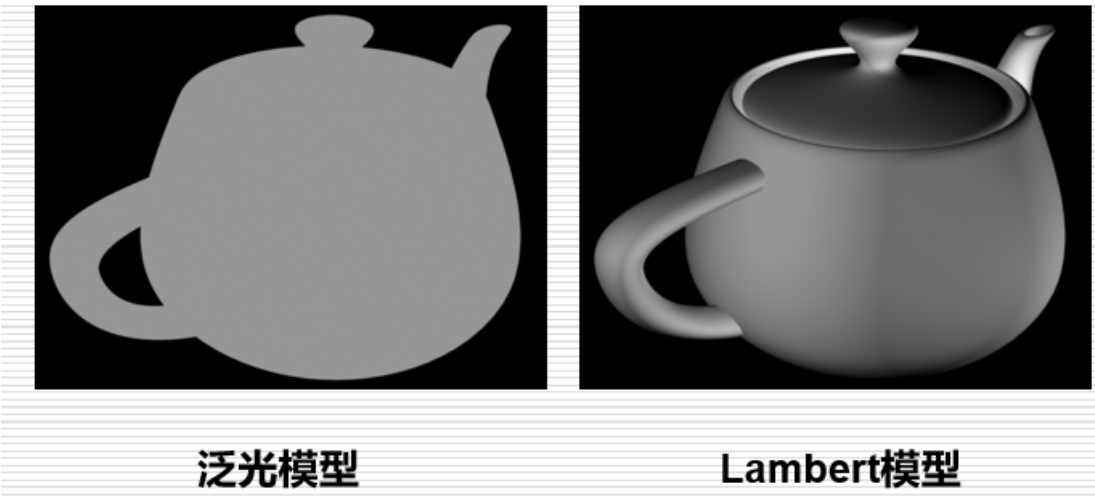


图 4.3　光强与距离平方的倒数成正比

是为了剔除夹角大于 90° 的光. 我们需要注意漫反射光线强度与出射方向是无关的, 因此无论人眼在哪里观察, 接收到的强度都是一样的. 在泛光项后加入漫反射项形成的光照模型成为 Lambert 模型:



第三节　Fresnel 效应

Fresnel 效应是我们生活中常见的一种现象. 具体来说, Fresnel 效应是指光线入射到介质交界面后的反射强度取决于入射光线与交界面的夹角. 如果入射光线与交界面法线接近, 那么将会产生相对较弱的反射; 反之, 如果我们以几乎平行于交界面的光线入射, 那么几乎所有的光线都会反射, 而不是折射.

为计算具体有多少光线反射, 多少光线折射, 我们有:

$$R_s = \left| \frac{n_1 \cos\theta_i - n_2 \cos\theta_t}{n_1 \cos\theta_i + n_2 \cos\theta_t} \right|^2, \tag{4.3}$$

以及

$$R_p = \left| \frac{n_1 \cos\theta_t - n_2 \cos\theta_i}{n_1 \cos\theta_t + n_2 \cos\theta_i} \right|^2. \tag{4.4}$$

其中:

- n_1 和 n_2 分别是两种介质的折射率;
- θ_i 是入射角;
- θ_t 是出射角.

则折射光线的比例

$$T = \frac{1}{2}\left[(1 - R_s) + (1 - R_p)\right]. \tag{4.5}$$

一种更为直接的近似被称为 Schlick 近似, 它能提供比之前提及的计算 Fresnel 效应的方法计算量更少的方式[23]:

$$T = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2 + \left[1 - \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2\right](1 - \cos\theta)^5. \tag{4.6}$$

在本项目的水面渲染中, 我们将采取式4.6的算法.

第五章　GPGPU-FFT 水面渲染的程序实现

在本章中, 我们将详细展示用 C++ 编程语言和 OpenGL 实现基于 GPGPU 的 FFT 水面渲染算法, 读者可以在https://github.com/supertan0204/IFFT-Lake-Simulation中查看完整的 Microsoft Visual Studio 解决方案[①]. 本章将给出流程中较为关键的代码并辅以详细解释.

第一节　程序整体框架

作为本章的开始, 我们先给出程序的整体框架结构, 如图5.1:

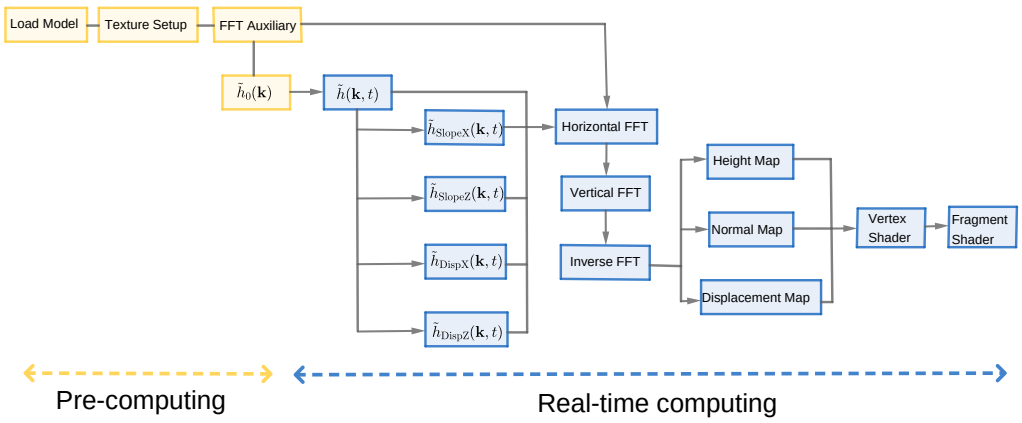


图 5.1　程序流程

流程图中黄色方框内部分指在实时渲染过程之前需要完成的计算的数据, 它们在渲染流程开始后并不会有任何改变; 而蓝色方框内的部分则表示在渲染流程中需要一直跟踪计算的数据. 在实时渲染每一帧的过程中, 我们需要完成 5 次独立的 FFT 过程, 将结果以纹理 (Texture) 的形式保存, 作为下一阶段的输入. 最后, 我们根据水面高度场以及第 四 章中的光照模型进行最终的渲染 [图5.1中顶点着色器 (Vertex Shader) 以及片元着色器 (Fragment Shader)]. 这

[①]Visual Studio(简称 VS) 是美国微软 (Microsoft) 公司的开发工具包系列产品. VS 是一个基本完整的开发工具集, 它包括了整个软件生命周期中所需要的大部分工具, 如 UML 工具, 代码管控工具, 集成开发环境 (IDE) 等等; 解决方案是 VS 的一个容器, 用于包含一个或多个相关项目, 以及生成信息, Visual Studio 窗口设置和不与特定项目关联的任何杂项文件.

里以算法的形式给出渲染循环内执行 2D-IFFT 的执行过程:

算法 5.1 2D-IFFT Algorithm

Data: Precomputed $\tilde{h}_0(\boldsymbol{k})$ and $\tilde{h}_0(-\boldsymbol{k})$
while *Rendering Loop* **do**
  Fourier components $\tilde{h}(\boldsymbol{k}, t)$ shaderpass;
  pingpong←0;
  for $i = 0 \to i < \log_2 N$ **do**
    Horizontal butterfly shaderpass for stage i;
    pingpong←pingpong++ mod 2;
  end
  for $i = 0 \to i < \log_2 N$ **do**
    Vertical butterfly shaderpass for stage i;
    pingpong←pingpong++ mod 2;
  end
  Inversion and permutation shaderpass;
end

第二节 $\tilde{h}(\boldsymbol{k}, t)$ 预计算

首先, 我们利用第 三 章中式3.5来计算 $\tilde{h}_0$. 为了获得 ξ_i 与 ξ_r, 注意到, 他们是满足标准分布的两个随机变量, 我们将利用一个噪声纹理随机读取数据. 在https://aeroson.github.io/rgba-noise-image-generator/中可以通过相关设置产生以下噪声纹理:

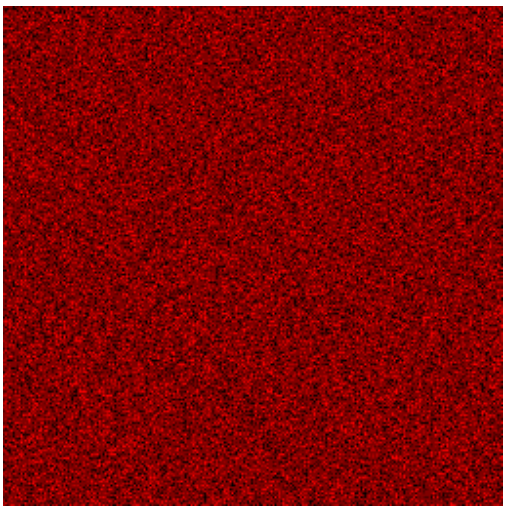


图 5.2 噪声纹理

图5.2的分辨率为 256×256, 每一像素的 G, B 通道[①]均设为 0, R 通道的色彩值服从从 0 到 1 的均匀分布.

[①]JPG 图片格式中, 每一像素的色彩值由 3 条颜色通道的色彩值组成, 分别为R(红色), G(绿色), B(蓝色)通道. 每条通道的色彩值取 0~1 间的一个浮点数.

我们在图5.2中随机采样两点, 其色彩值的 R 通道值应相互独立, 且服从从 0 到 1 的均匀分布. 为了获得相互独立的服从标准正态分布的两个随机变量, 我们采用 Box-Muller 转换法, 他是说, 若我们有两个随机变量 U_1, U_2 服从 $[0,1]$ 上均匀分布, 则若随机变量 X, Y 满足:

$$\begin{aligned} X &= \cos(2\pi U_1)\sqrt{-2\ln U_2} \\ Y &= \sin(2\pi U_1)\sqrt{-2\ln U_2} \end{aligned}, \tag{5.1}$$

则 X, Y 服从标准正态分布. 这个结论的证明请参考附录中定理A.1.

利用此结论, 我们利用采样图5.2中的两点来获得 ξ_i, ξ_r. 在 GLSL 中, 我们用如下代码来完成 Box-Muller 转换:

```
// Box-Muller
vec4 gaussRND(){
  ivec2 p1 = ivec2(gl_GlobalInvocationID.x - 1, gl_GlobalInvocationID.y);
  ivec2 p2 = ivec2(gl_GlobalInvocationID.x + 1, gl_GlobalInvocationID.y);
  ivec2 p3 = ivec2(gl_GlobalInvocationID.x, gl_GlobalInvocationID.y - 1);
  ivec2 p4 = ivec2(gl_GlobalInvocationID.x, gl_GlobalInvocationID.y + 1);
  // clamp函数限制采样值在0.001到1.0之间
  float noise00 = clamp(imageLoad(noise, p1).r, 0.001, 1.0);
  float noise01 = clamp(imageLoad(noise, p2).r, 0.001, 1.0);
  float noise02 = clamp(imageLoad(noise, p3).r, 0.001, 1.0);
  float noise03 = clamp(imageLoad(noise, p4).r, 0.001, 1.0);

  float u0 = 2.0 * M_PI * noise00; // M_PI = 3.14159
  float v0 = sqrt(-2.0 * log(noise01));
  float u1 = 2.0 * M_PI * noise02;
  float v1 = sqrt(-2.0 * log(noise03));

  vec4 rnd = vec4(v0 * cos(u0), v0 * sin(u0), v1 * cos(u1), v1 * sin(u1));

  return rnd;
}
```

这里:

- `gl_GlobalInvocationID`是 GLSL 的内置全局变量, 它是一个`ivec3`型变量, 表示当前执行单元在全局工作组中的三维索引. 上述代码中, 我们在噪声中由`gl_GlobalInvocationID`指定的位置周围采样四个点的 R 通道值. 由于噪声纹理中的 R 通道值服从均匀分布, 我们由此得到了四个服从从 $[0,1]$ 均匀分布的随机变量`noise00`, `noise01`, `noise02`, `noise03`[注意到我们需要计算 $\tilde{h}_0$ 及其复共轭 (式3.7), 因此一共需要两组 (ξ_i, ξ_r)]. 随后利用式5.1得到两组服从标准正态分布的 (ξ_i, ξ_r).
- `imageLoad`是 GLSL 的内置函数, 它可以在指定纹理的指定位置采样. 这里我们在`noise`纹理中四个不同指定位置 (`p1`,`p2`,`p3`,`p4`) 进行了采样.

随后, 我们按照式3.3计算 $\sqrt{P_h(\pm\boldsymbol{k})}/\sqrt{2}$:

```
float mag = length(k);
float magSq = mag * mag; // k的平方
// sqrt(Ph(k)) / sqrt(2)
float h0k = clamp(sqrt((A / (magSq * magSq))
* pow(dot(normalize(k), normalize(windDirection)), 2.0)
* exp(-(1.0 / (magSq * E * E)))
* exp(-magSq * pow(L / 2000.0, 2.0))) / sqrt(2.0), -4000, 4000);

// sqrt(Ph(-k)) / sqrt(2)
float h0minusk = clamp(sqrt((A / (magSq * magSq))
* pow(dot(normalize(-k), normalize(windDirection)), 8.0)
* exp(-(1.0 / (magSq * E * E)))
* exp(-magSq * pow(L / 2000.0, 2.0))) / sqrt(2.0), -4000, 4000);
```

最后, 我们将`gaussRND`函数返回的结果`gauss_random`前两个分量 (`.xy`) 与`h0k`相乘, 后两个分量 (`.zw`) 与`h0minusk`相乘, 并将结果存储到一个新的纹理`tilde0`的 R, G, B, A 通道中:

```
vec4 gauss_random = gaussRND();
imageStore (tilde0, ivec2(gl_GlobalInvocationID.xy), vec4(gauss_random.xy
    * h0k, gauss_random.zw * h0minusk));
```

若将`tilde0`渲染出来, 将得到如下纹理:

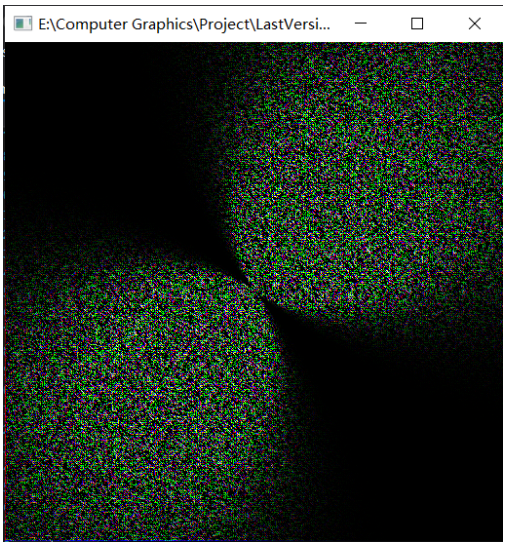


图 5.3　储存 $\tilde{h}_0$ 和 $\tilde{h}_0^*$

的纹理, 这里 $N = 256$, $L = 1300$, $\boldsymbol{w} = (1.0, 1.0)$, $V = 40$.

最后, 我们用式3.5产生 $\tilde{h}(\boldsymbol{k}, t)$ 的纹理:

```
float magnitude = length(k);
float w = sqrt(g * magnitude); // g是重力加速度
vec2 tilde_h0k_values = imageLoad(tex0, ivec3(ivec2(gl_GlobalInvocationID)
    , 1)).rg;
complex fourier_cmp = complex(tilde_h0k_values.x, tilde_h0k_values.y);
```

```
    vec2 tilde_h0minusk_values = imageLoad(tex0, ivec3(ivec2(
        gl_GlobalInvocationID.xy), 1)).ba;
    complex fourier_cmp_conj = conj(complex(tilde_h0minusk_values.x,
        tilde_h0minusk_values.y));

    float cos_w_t = cos(w * time);
    float sin_w_t = sin(w * time);

    // Euler公式
    complex exp_iwt = complex(cos_w_t, sin_w_t);
    complex exp_iwt_inv = complex(cos_w_t, -sin_w_t);

    complex h_k_t = add(mul(fourier_cmp, exp_iwt), mul(fourier_cmp_conj,
        exp_iwt_inv));

    imageStore(tilde_hkt, ivec2(gl_GlobalInvocationID.xy), vec4(h_k_t.real,
        h_k_t.im, 0, 1));
```

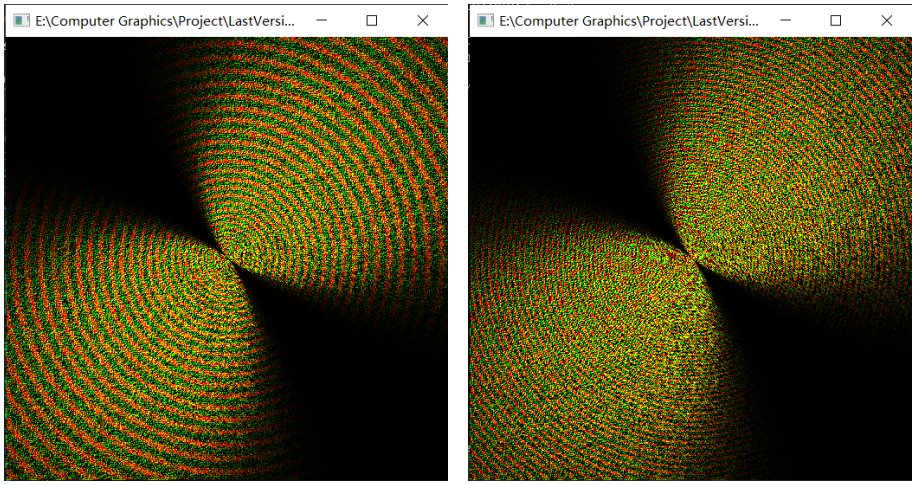
这里:

- 我们预先在计算着色器中定义了复数类 (`complex`) 并且定义其加法 (`add`函数) 和乘法 (`mul`函数), 在这里, 我们利用 Euler 公式:

$$\exp\{i\theta\} = \cos\theta + i\sin\theta$$

  来存储复数的实部和虚部;
- $\tilde{h}_0(\boldsymbol{k}, t)$ 是一个复数, 我们将其实部存储在`tilde_hkt`纹理的 R 通道中, 虚部存储在 G 通道中, B 和 A 通道分别指定为 0 和 1.

另外, 我们需要注意`tilde_hkt`纹理在渲染流程中是随时间不断变化的.



(a) $t = 50$ s (b) $t = 200$ s

图 5.4 $\tilde{h}(\boldsymbol{k}, t)$ 纹理 (随时间 t 变化), 其余参数与图5.3中相同

第三节　利用 GPGPU 执行 IFFT

我们已经成功获得随时间 t 不断变化的 $\tilde{h}(\boldsymbol{k},t)$ 纹理, 我们只需对其应用式3.10, 即对其应用二维 IFFT. 在此之前我们需要 FFT 辅助 (FFT Auxiliay) 预计算流程. 首先, 我们将计算 IFFT 必要的信息存储在一个**蝶式纹理 (Butterfly Texture)** 中 (本节第 一 小节); 另外, 我们还需要另外的两个**弹跳纹理 (Ping-pong Texture)** 来存储原纹理在 FFT 流程中不同阶段 (Stage) 的变化 (本节第 二 小节). 如果读者对这部分理论有所遗忘, 请回过头阅读第 二 章中的内容.

一、蝶式纹理

碟式纹理用以存储蝶式结与蝶式运算的相关信息, 它的 R, G, B, A 通道均需要被使用以存储信息. 值得注意的一点是, 蝶式纹理的大小为 $N \times \log_2(N)$, 这是因为其水平的维度表示了 IFFT 过程的总阶段数.

对于蝶式纹理的任意像素 (x, y)[总共有 $N \times \log_2(N)$ 像素], 其 x 坐标存储了当前 IFFT 过程所处的阶段 $[0 \leqslant x \leqslant \log_2(N),\ x \in \mathbb{N}]$; 像素 (x, y) 的 R 和 G 通道用来存储旋转因子的实部和虚部 ($\mathcal{W}_N^k = \mathrm{R} + \mathrm{G} \cdot \mathrm{i}$), B 和 A 通道用来存储当前蝶式运算中应输入的序列序数 (即位反转序列此点处的值).

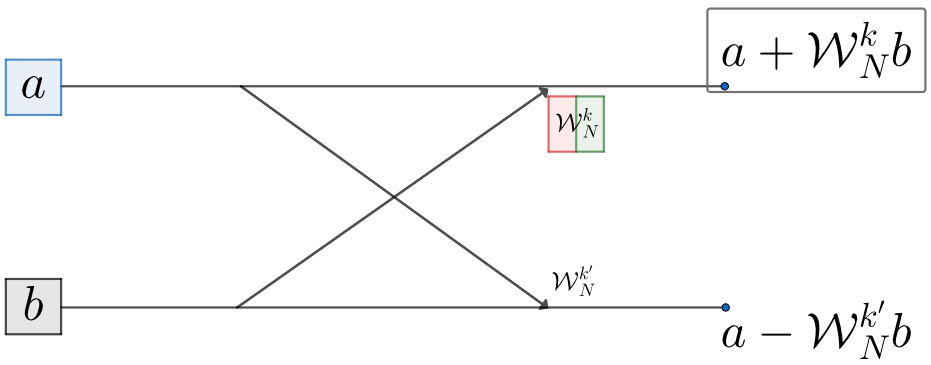


图 5.5　上蝶式运算以获得 $a + \mathcal{W}_N^k b$, 这里 B 通道用来存储上输入, A 通道用来存储下输入

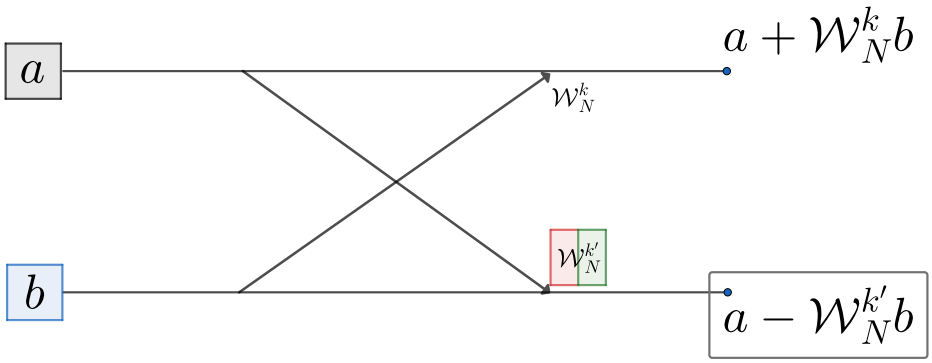


图 5.6　下蝶式运算以获得 $a - \mathcal{W}_N^k b$, 这里 B 通道用来存储下输入, A 通道用来存储上输入

为计算 $\mathcal{W}_N^k$, 首先我们注意到, 对给定的 N, $\mathcal{W}_N$ 应是一个固定值. 利用式2.15, 我们通过

$$k = \frac{y \cdot N}{2^{x+1}} \mod N \tag{5.2}$$

获得 k 的正确值.

为了确定像素 (x, y) 执行上蝶式运算还是下蝶式运算, 我们判断不等式

$$y \mod 2^{x+1} < 2^x \tag{5.3}$$

是否为真. 若为真, 则当前像素执行上蝶式运算, 否则执行下蝶式运算. 在第一个阶段, 我们将使用位反转序列 (`bit_reversed`), 随后的阶段中, 我们直接用上一阶段的输出作为输入:

```
vec2 p = gl_GlobalInvocationID.xy;
int butterflywing;
int butterflyspan = int(pow(2, p.x));
if(mod(p.y, pow(2, p.x + 1)) < pow(2, p.x)){
  butterflywing = 1; // 执行上蝶式运算
}
else{
  butterflywing = 0; // 执行下蝶式运算
}
// 第一阶段使用位反转序列
if(p.x == 0){
// 上蝶式运算
    if(butterflywing == 1){
      imageStore(tex2, ivec3(ivec2(p), 0),
      vec4(twiddle.real, twiddle.im,
      bit_reversed.j[int(p.y)],
      bit_reversed.j[int(p.y + 1)]));
    }
    else{
```

```
20        // 下蝶式运算
          imageStore(tex2, ivec3(ivec2(p), 0),
22        vec4(twiddle.real, twiddle.im,
          bit_reversed.j[int(p.y - 1)],
24        bit_reversed.j[int(p.y)]));
      }
26  }
  else // 后续阶段使用上一阶段输出
28  {
    //top butterfly wing
30    if(butterflywing == 1){
      imageStore(tex2, ivec3(ivec2(p), 0),
32    vec4(twiddle.real, twiddle.im,
      p.y, p.y + butterflyspan));
34    }
      else{
36      //bot butterfly wing
        imageStore(tex2, ivec3(ivec2(p), 0),
38      vec4(twiddle.real,twiddle.im,
        p.y - butterflyspan, p.y));
40    }
  }
```

若将碟式纹理渲染出, 可得到图5.7

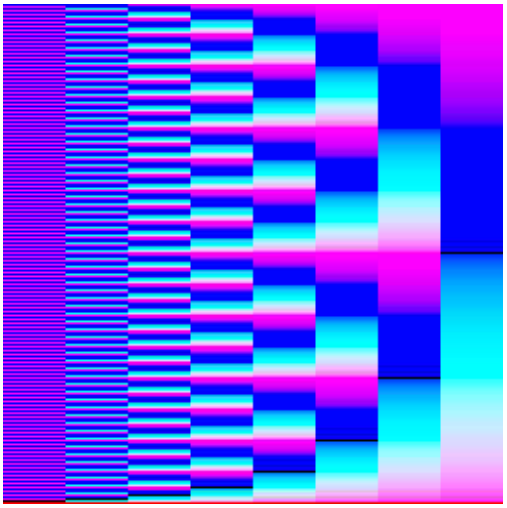


图 5.7　蝶式纹理 (256 × 256)

二、弹跳纹理

这一节中, 我们介绍如何利用两个纹理pingpong0和pingpong1来记录 IFFT 不同阶段的数据. 算法执行过程中, 数据将会在pingpong0和pingpong1之间 “弹跳”(比如: 第一阶段在pingpong0内执行完毕后, 传入pingpong1, 再在pingpong1中执行第二阶段, 完成后传回pingpong0, 以此类推), 故将这两个纹理命名为 “弹跳纹理”. 在执行 “弹跳” 操作之前, 我们先在 CPU 中定义一个int型变量pingpong, 它用来存储当前

我们正在从哪个弹跳纹理中存取数据, 它的值仅能取 0 或 1. 当我们进入渲染循环 (Rendering Loop) 后, 我们首先将pingpong变量的值设为pingpong=0, 这意味着我们将从pingpong0开始计算. 接下来, 我们按照阶段顺序依次计算 IFFT, 并且每一次计算完成后, 我们切换pingpong的值. 在完成弹跳计算后, 根据式3.10, 我们将结果乘上 $(-1)^m$ 和 $(-1)^n$, 并且在前面除上 $1/N^2$. 注意, 我们的初始pingpong0纹理即是 $\tilde{h}(\boldsymbol{k},t)$ 纹理 (图5.4所示纹理).

与pingpong一样的是, 我们定义direction来指定水平或是竖直的计算. 它的值只能取 0 或 1. 下面的代码展示了计算着色器中计算水平 FFT 的过程, 计算竖直方向 FFT 的代码几乎一样, 这里不再展示.

```
void horizontalButterflies(){
  complex H;
  ivec2 x = ivec2(gl_GlobalInvocationID.xy);

  if(pingpong == 0){
    vec4 data = imageLoad(butterflyTexture, ivec2(stage, x.x), 0).rgba;
    vec2 p_ = imageLoad(butterflyTexture, ivec2(data.z, x.y), 1).rg;
    vec2 q_ = imageLoad(butterflyTexture, ivec2(data.w, x.y), 1).rg;
    vec2 w_ = vec2(data.x, data.y);

    complex p = complex(p_.x, p_.y);
    complex q = complex(q_.x, q_.y);
    complex w = complex(w_.x, w_.y);

    H = add(p, mul(w, q));

    imageStore(pingpong1, x, vec4(H.real, H.im, 0, 1));
  }
  else if(pingpong == 1){
    ..... // 与pingpong == 0的情形几乎完全一样
}
```

根据算法5.1, 我们在渲染循环中传入相关数值, 适当分配计算着色器空间:

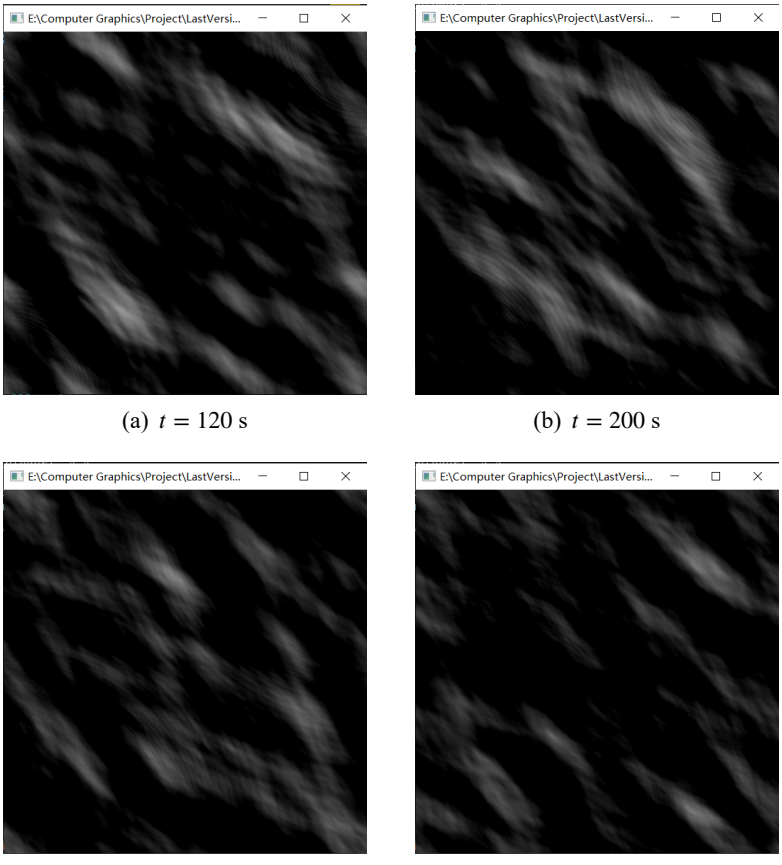
```
int pingpong = 0;
int direction = 0; // 指定水平计算
for (int i = 0; i < log2(FourierGridSize); ++i) { // FourierGridSize为2的整数次幂
  glUseProgram(PingPongComputeProgram);
  int stageLocation = glGetUniformLocation(PingPongComputeProgram, "stage"); // 获得当前stage地址
  int pingpongLocation = glGetUniformLocation(PingPongComputeProgram, "pingpong"); // 获得pingpong地址
  int directionLocation = glGetUniformLocation(PingPongComputeProgram, "direction"); // 获得当前direction地址
  glUniform1i(stageLocation, i); // 传入stage
  glUniform1i(pingpongLocation, pingpong); // 传入pingpong
  glUniform1i(directionLocation, direction); // 传入direction
  glDispatchCompute((GLuint)FourierGridSize, (GLuint)FourierGridSize, 1); //分配OpenGL 计算着色器计算空间
  // 切换pingpong值
  pingpong++;
  pingpong = pingpong % 2;
```

```
15
    }
17  direction = 1; // 指定竖直计算
    ..... // 与水平计算的过程完全一样
```

最后, 我们将获得的振幅乘以系数 $(-1)^m$ 和 $(-1)^n$, 并且在最终结果前除以 N^2:

```
    ivec2 p = ivec2(gl_GlobalInvocationID.xy);
2
    float perms[] = {-1.0, 1.0};
4   int index = int(mod(int(p.x + p.y), 2));
    float perm = perms[index];
6
    if (pingpong == 0){
8       float h = imageLoad(pingpong0, p).r;
        float value = (perm * h) / (float(N) * float(N));
10      imageStore(tilde_disp, p, vec4(vec3(value), 1));
    }
12  else if (pingpong == 1){
        ..... // 与pingpong == 0几乎完全一样
14  }
```

经过上述流程后, 我们可以得到最终的随时间变化的水面高度场:



(a) $t = 120$ s (b) $t = 200$ s

(c) $t = 280$ s (d) $t = 360$ s

图 5.8 高度场 $h(\boldsymbol{k}, t)$

三、波浪修正

我们已经成功获得了水面的高度场 (即 y 方向上的位移), 现在需要利用波浪修正来使得渲染的水面看上去更加真实. Tessendorf[2] 以及 Flügge[5] 指出, 自然的水面会出现一些尖锐的波峰效果, 而若我们只采用图5.8得到的高度场来渲染, 得到的水面波峰会显得比较圆, 与真实场景不相符. 为了得到尖锐的波峰, 我们还需要计算 x 和 z 方向的波浪的位移 (这里需要提醒的是, 我们始终以 xz 平面作为水平面).

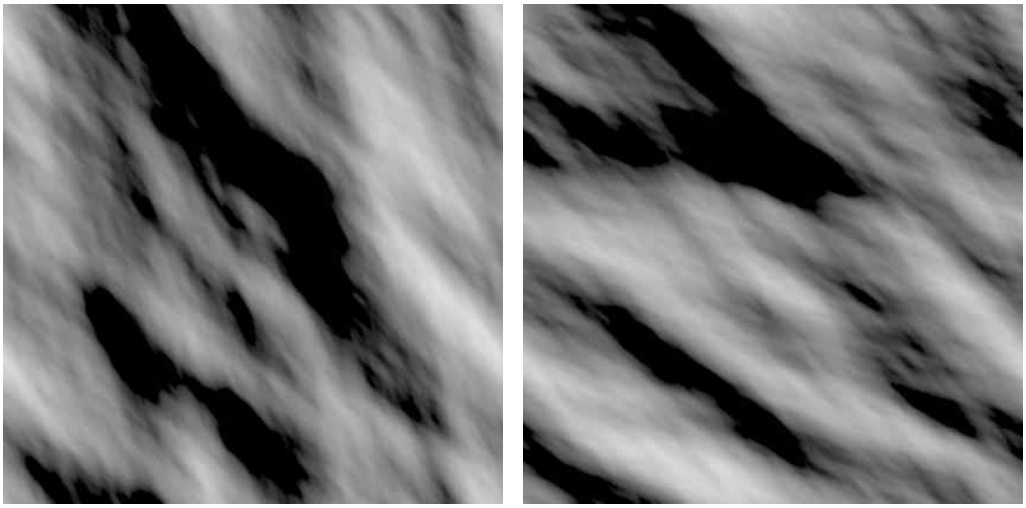
为了得到水平面上的位移, 我们还需要额外进行两次二维 IFFT 操作. 对于水平面的位移场, 我们用如下表达式来描述[2]:

$$\boldsymbol{D}(\boldsymbol{p},t)=\sum_{\boldsymbol{k}}-\mathrm{i}\hat{\boldsymbol{k}}\tilde{h}(\boldsymbol{k},t)\exp\{\mathrm{i}\boldsymbol{k}\cdot\boldsymbol{p}\}. \tag{5.4}$$

对比产生高度场 $h(\boldsymbol{k},t)$ 的过程, 我们唯一需要调整的就是对 Fourier 振幅分量 $\tilde{h}(\boldsymbol{k},t)$ 乘上

$$-\mathrm{i}\hat{\boldsymbol{k}}. \tag{5.5}$$

这里回顾: $\hat{\boldsymbol{k}}$ 是 $\boldsymbol{k}$ 方向的单位向量. 我们得到渲染后的 x 与 z 方向的位移纹理:



(a) x 方向　　(b) z 方向

图 5.9　水平方向的位移纹理

四、多层纹理序列

在实现上述渲染的流程中, 一个实际的问题需要被考虑: 因为 GPU 和 OpenGL 的硬件限制, 我们只能最多使用 8 个图像单元 (Image Unit), 这个值由 OpenGL 的内置变量`GL_MAX_IMAGE_UNITS`规定. 然而, 在我们的渲染流程中,

一共需要多达 12 个纹理, 它们分别是:

1. (1 个) 存储噪声的纹理 (图5.2);
2. (1 个) 存储 $\tilde{h}_0$ 及其共轭的纹理 (图5.3);
3. (1 个) 存储 $\tilde{h}(\boldsymbol{k},t)$ 的纹理 (图5.4);
4. (1 个) 蝶式纹理 (图5.7);
5. (2 个) 执行式3.10的弹跳纹理`pingpong0`以及`pingpong1`;
6. (1 个) 存储高度场 $h(\boldsymbol{k},t)$ 的纹理 (图5.8);
7. (2 个) 存储水平方向位移的纹理 (图5.9);
8. (4 个) 执行式5.4的纹理, 各需要两个弹跳纹理.

在原本的设计中, 一个图片单元只用来存储一个纹理, 造成了存储空间的浪费. 我们的改进措施是: 将多个纹理绑定在一个由 OpenGL 内部定义的`image2DArray`对象上, 构造出一个**多层纹理序列**. OpenGL 规定, 一个`image2DArray`对象只占用一个图片单元, 这就使得我们可以大大缩小所使用的图片单元数量. 在读取纹理数据时, 我们只需用一个`ivec3`型向量, 其中最后一个分量用来指定所需读取的纹理在纹理单元内的层数, 前两个分量即是 2D 纹理的像素坐标, 即可按原来的方法读取相应数据, 比如:

```
layout (binding = 2, rgba32f) uniform image2DArray tex2;

..... // 运算过程

imageStore(tex2, ivec3(position, 2), vec4(real, im, 0, 1))
```

这里`tex2`纹理序列 (`image2DArray`) 由 3 层 2D 纹理 (`image2D`) 组成, 按顺序依次是: 蝴蝶纹理, `pingpong0`纹理, `pingpong1`纹理. 在调用`imageStore`存储数据时的代码表示在`tex2`的第二层的`position`(`ivec2`型) 处存入数据`vec4(real,im,0,1)`.

第四节 水面渲染

我们采用第四章中提及的 Phong 光照模型以及 Fresnel 效应, 并根据图5.8和图5.8的纹理进行最终的水面渲染. 本节给出其程序实现.

一、顶点着色器

在着色操作之前, 我们先从外部导入一个具有 $N \times N$ 结点的三角网格 (使用 Assimp 库[①]导入). 这篇文章的程序中, 此网格由 Blender[②]生成, 且 N 值被设置为 256.

对于网格的每一个顶点, 其法向量在初始时刻均被定义为 $(0, 1, 0)$. 在程序运行的过程中, 我们根据之前计算的高度场修改每个顶点的法向量, 最后再将其化为单位向量:

```
// tex3包含tilde_disp_x(第0层), tilde_disp_z(第1层), tilde_slope_x(第2层),
    tilde_slope_z(第3层)
vec4 textureSampledSlopeX = imageLoad(tex3, ivec3(N * aTexCoord, 2));
vec4 textureSampledSlopeZ = imageLoad(tex3, ivec3(N * aTexCoord, 3));

Normal = normalize(vec3(textureSampledSlopeX.x, 1.0f, textureSampledSlopeZ
    .x)); // 单位化法向量
```

这里`aTexCoord`是 OpenGL 内置的纹理坐标, 它在 $[0, 1]$ 范围内取值, 这里乘上 N 使得其能在 $[0, N]$ 范围取值.

每个顶点的世界坐标 (World Coordinates)[③], 只需要在初始位置的基础上, y 方向加上高度场对应值, x, z 方向加上水平位移场对应值:

```
vec4 textureSampledDispX = imageLoad(tex3, ivec3(N * aTexCoord, 0));
vec4 textureSampledDispZ = imageLoad(tex3, ivec3(N * aTexCoord, 1));

Position = vec3(model * vec4(aPos.x + textureSampledDispX.x, aPos.y +
    textureSampledHeightMap.x, aPos.z + textureSampledDispZ.x, 1.0));
```

这里`model`是模型矩阵, 它在渲染流程前预先指定, 用于将物体从以物体中心为原点的局部坐标系投影到世界坐标系中; `aPos`是`vec3`型的局部坐标.

未加入着色效果的湖面网格顶点渲染结果如图5.10.

二、片元着色器

我们利用第 四 章中的 Phong 光照模型. 对于 Fresnel 效应, 我们采取 schlick 近似的方法减小计算量 (式4.6). 在这里, 我们将空气的折射率设为 1, 水的折射率

[①]一个非常流行的模型导入库, 它是 Open Asset Import Library(开放的资产导入库) 的缩写. Assimp 能够导入很多种不同的模型文件格式 (并也能够导出部分的格式), 它会将所有的模型数据加载至 Assimp 的通用数据结构中

[②]Blender 是一款免费开源的跨平台 3D 创作套件, 用户可以利用它创建 3D 可视化效果, 例如静态图像, 3D 动画,VFX(视觉特效) 快照和视频编辑

[③]世界坐标系是系统的绝对坐标系, 在没有建立用户坐标系之前画面上所有点的坐标都以该坐标系的原点来确定各自的位置

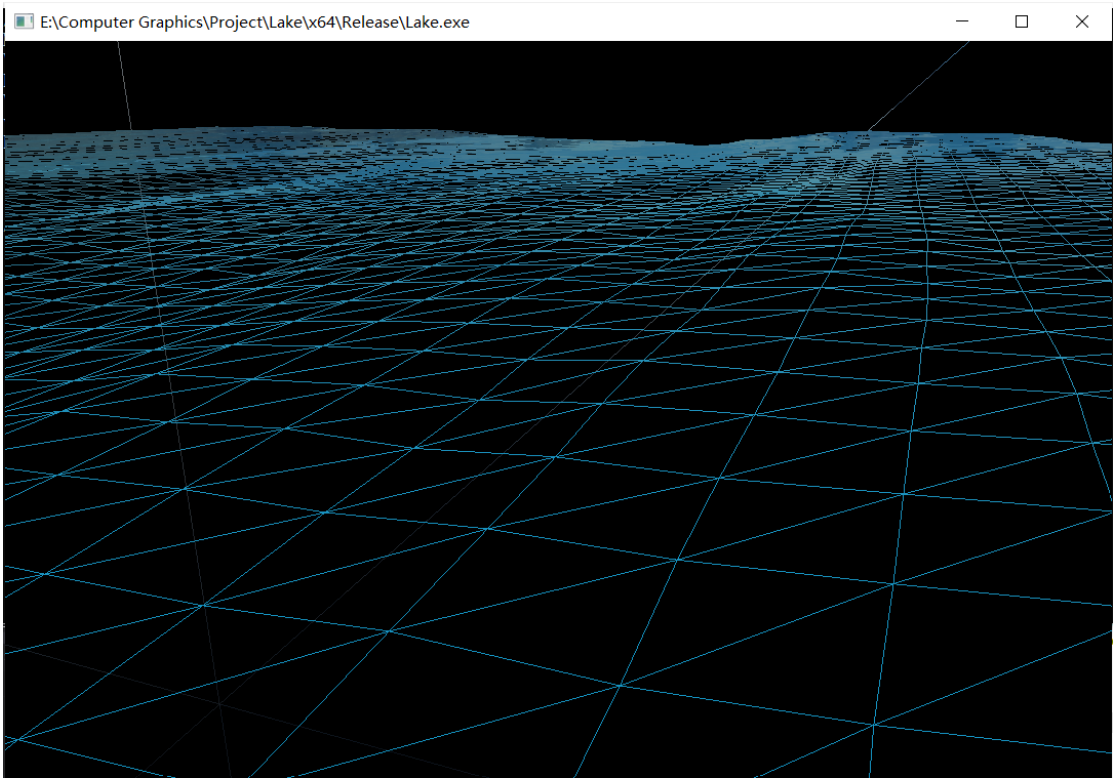


图 5.10　湖面网格渲染结果

设为 1.33, 按照式4.6有:

$$R(\theta) = 0.02 + 0.98 \times (1 - \|\mathbf{n} \cdot \mathbf{n}_i\|)^5 \tag{5.6}$$

代码:

```
float fresnel = 0.02 + 0.98 * pow(1.0 - abs(dot(norm, Inject)), 5.0);
FragColor = vec4(light_result * mix(material.ambient, Reflect, fresnel),
    1.0);
```

最终的`FragColor`为渲染管线输出的颜色, 它是由光照效果以及材料属性共同决定的. `mix`是 GLSL 的内置函数, 专门用来处理结果的混合.

三、天空盒

天空盒 (Skybox) 是一个包含了整个场景的立方体, 它包含周围环境的六个图像, 让用户 “认为” 他处在一个比实际大得多的环境中. 天空盒由 OpenGL 的**立方体贴图 (Cube Map)** 特性实现. 立方体贴图就是一个包含了 6 个 2D 纹理的纹理, 每个 2D 纹理都组成了立方体的一个面. 它的一个极为有用的特性是, 它可以通过一个方向向量来进行采样[24].

有关立方体贴图和天空盒效果的详细实现过程不是本文的重点, 有兴趣的读

者可以参考 JoeyDeVries[24]. 这里给出代码:

```
// 顶点着色器
#version 440 core
layout (location = 0) in vec3 aPos;

out vec3 TexCoords;

uniform mat4 projection;
uniform mat4 view;

void main()
{
  TexCoords = aPos;
  vec4 pos = projection * view * vec4(aPos, 1.0);
  gl_Position = pos.xyzw;
}
```

```
// 片元着色器
#version 440 core
out vec4 FragColor;

in vec3 TexCoords;

uniform samplerCube skybox;

void main()
{
  FragColor = texture(skybox, TexCoords);
}
```

第六章　结果展示与总结

第一节　结果展示

图6.1, 图6.2和图6.3展示了三个不同背景 (天空盒) 下的湖面渲染效果:

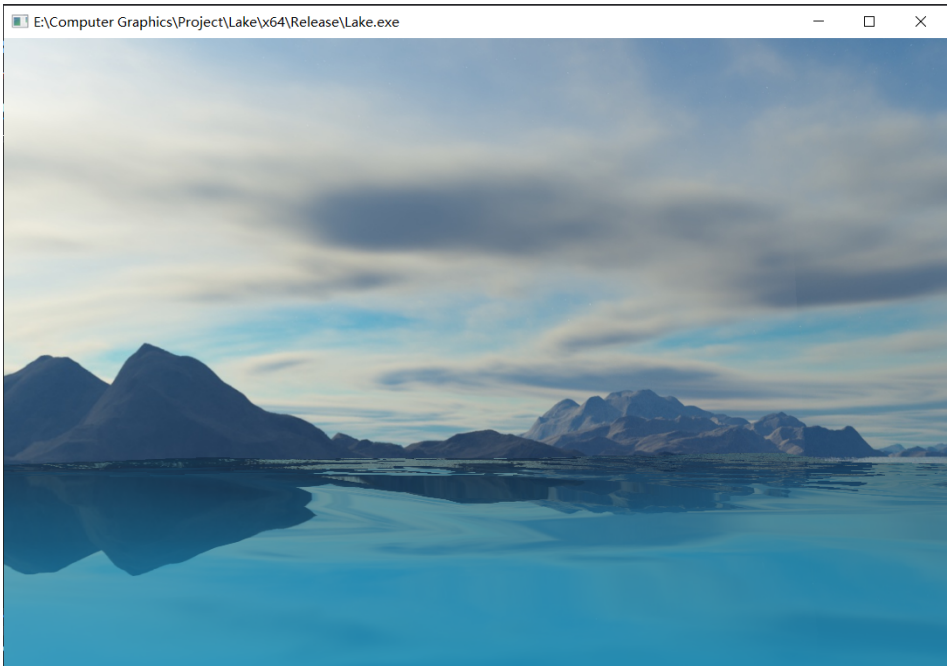


图 6.1　渲染结果 (视角 1)

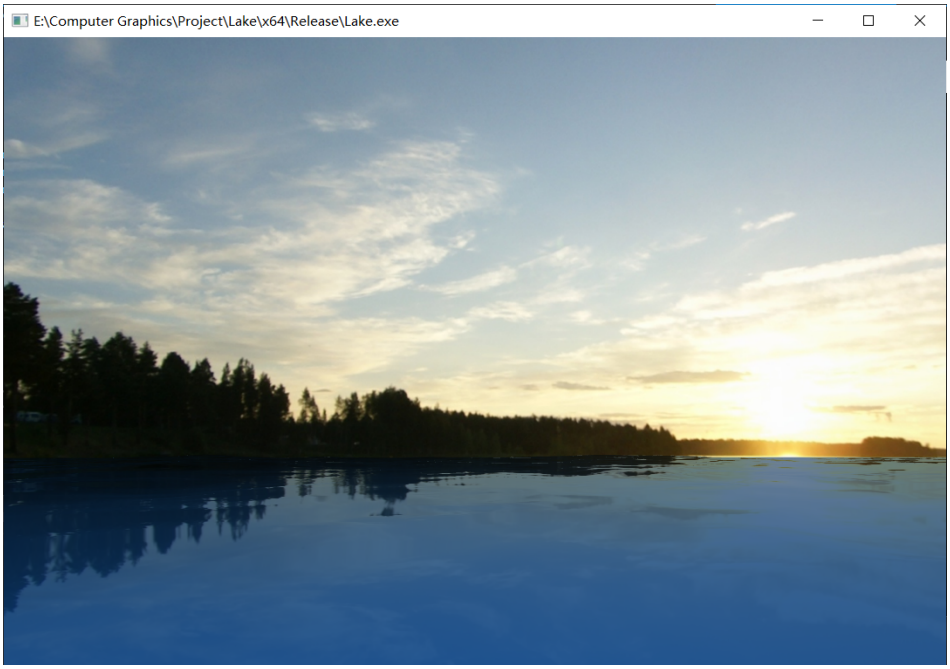


图 6.2　渲染结果 (视角 2, 效果最佳)

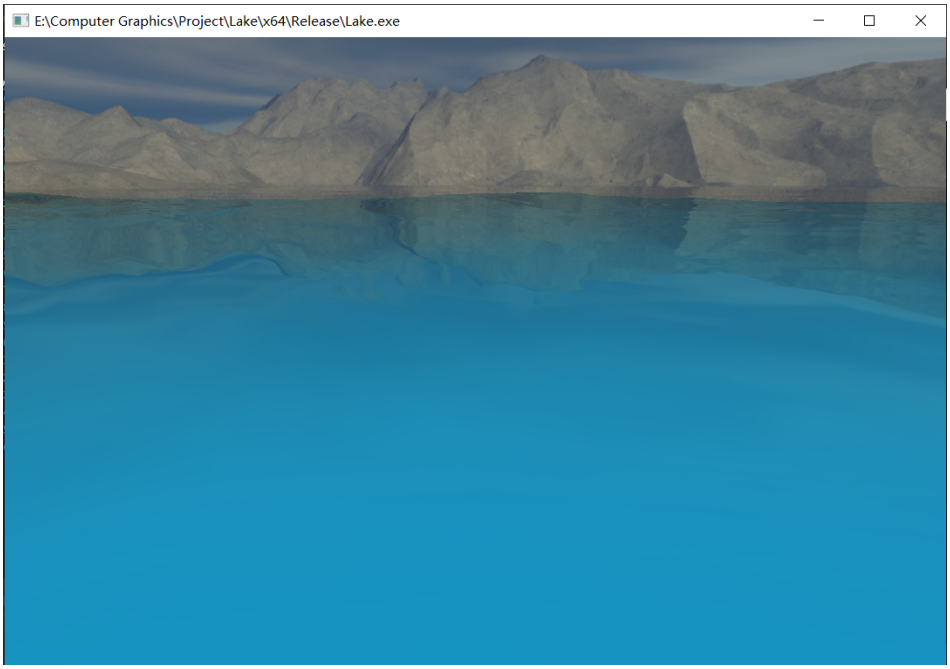


图 6.3　渲染结果 (视角 3)

此程序使用核显卡 **Intel(R) HD Graphics620** 渲染, 并设置窗口分辨率为 1080×720, Fourier 网格大小为 $N = 256$, 运行帧率为 20~30 帧每秒.

第二节　总结与不足

整体来看, 程序运行较为稳定流畅, 产生的水面波动效果远处看还不错, 基本上实现了实时运算的 FFT 以及正确的水面效果, 但仍有一些缺陷没有解决:

1. 窗口下半部分的水面渲染效果并不好, 看上去像是一整块颜色, 没有波动感. 这可能是由于使用的 Fourier 网格个数不够导致的, 但 Flügge[5] 文章同样使用 256×256 的网格, 却能得到及其逼真的效果, 所以笔者倾向于产生此问题的原因是使用的光照模型 (Phong 反射模型) 比较粗略, 并且没有采用水面材质等信息;
2. 波纹的反射处仍有走样 (Aliasing) 现象, 可以通过添加 MSAA 技术解决, 但会增加更多计算成本.
3. 运行的速度仍不够好, 这可能是由于实验使用的电脑配置不够高的原因.

参 考 文 献

[1] Mastin GA, Watterberg PA, Mareda JF. Fourier synthesis of ocean scenes. *IEEE Computer Graphics and Applications*, 1987, 7(3): 16-23. DOI: 10.1109/MCG.1987.276961.

[2] Tessendorf J. Simulating ocean water. *SIG-GRAPH'99 Course Note*, 2001.

[3] Cooley JW, Tukey JW. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 1965, 19(90): 297-301.

[4] 真实感水体渲染技术总结. https://zhuanlan.zhihu.com/p/95917609.

[5] Flügge FJ. Realtime gpgpu fft ocean water simulation. 2017. http://tubdok.tub.tuhh.de/handle/11420/1439. DOI: 10.15480/882.1436.

[6] Wikipedia. https://en.wikipedia.org/wiki/Graphics_processing_unit.

[7] Wikipedia. https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units.

[8] Understanding gpu architecture: Performance: Gpu vs. cpu. https://cvw.cac.cornell.edu/gpuarch/performance.

[9] Gupta P. Cuda refresher: Reviewing the origins of gpu computing. https://developer.nvidia.com/blog/cuda-refresher-reviewing-the-origins-of-gpu-computing/.

[10] WikiPedia. https://zh.wikipedia.org/wiki/OpenGL.

[11] Graphics rendering pipeline. https://en.wikipedia.org/wiki/Graphics_pipeline.

[12] JoeyDevries. 着色器. https://learnopengl-cn.github.io/01%20Getting%20started/05%20Shaders/.

[13] Wikipedia. https://zh.wikipedia.org/wiki/%E7%9D%80%E8%89%B2%E5%99%A8.

[14] Compute shader. https://www.khronos.org/opengl/wiki/Compute_Shader.

[15] Wikipedia. https://zh.wikipedia.org/zh-cn/GLSL.

[16] Dongarra J, Sullivan F. Guest editors introduction to the top 10 algorithms. *Computing in Science & Engineering*, 2000, 2(1): 22-23. DOI: 10.1109/MCISE.2000.814652.

[17] Phillips OM. The equilibrium range in the spectrum of wind-generated waves. *Journal of Fluid Mechanics*, 1958, 4: 426 - 434.

[18] Osgood B. Fourier transform and its applications. https://see.stanford.edu/materials/lsoftaee261/book-fall-07.pdf.

[19] Wikipedia. https://en.wikipedia.org/wiki/Butterfly_diagram.

[20] Lyons R. Computing fft twiddle factors. https://www.dsprelated.com/showcode/232.php.

[21] Blinn JF. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, 1977, 11(2): 192–198. https://doi.org/10.1145/965141.563893.

[22] Phong BT. Illumination for computer generated pictures. *Commun. ACM*, 1975, 18(6): 311–317. https://doi.org/10.1145/360825.360839.

[23] Fresnel and beer's law. https://graphicscompendium.com/raytracing/11-fresnel-beer.

[24] JoeyDeVries. 立方体贴图. https://learnopengl-cn.github.io/04%20Advanced%20OpenGL/06%20Cubemaps/.

附录

第一节　Box-Muller 变换

定理 A.1 (Box-Muller 变换)　选取两个服从 $[0,1]$ 上均匀分布的随机变量 U_1, U_2, 若随机变量 X, Y 满足:

$$\begin{aligned} X &= \cos(2\pi U_1)\sqrt{-2\ln U_2} \\ Y &= \sin(2\pi U_1)\sqrt{-2\ln U_2} \end{aligned}, \tag{A.1}$$

则 X, Y 服从标准正态分布.

证明　假定 X, Y 满足标准正态分布 (均值为 0, 方差为 1 的正态分布), 且相互独立. 令 $p(X), p(Y)$ 分别为其密度函数, 则:

$$p(X) = \frac{1}{\sqrt{2\pi}}\exp\left\{-\frac{X^2}{2}\right\},\ p(Y) = \frac{1}{\sqrt{2\pi}}\exp\left\{-\frac{Y^2}{2}\right\}.$$

由于 X, Y 相互独立, 因此他们的联合密度函数为:

$$p(X,Y) = \frac{1}{2\pi}\exp\left\{-\frac{X^2+Y^2}{2}\right\}.$$

将 X, Y 作坐标变换, 使得:

$$X = R\cos\theta,\ Y = R\sin\theta,$$

则:

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\frac{1}{2\pi}\exp\left\{-\frac{X^2+Y^2}{2}\right\}\mathrm{d}X\mathrm{d}Y = \int_0^{2\pi}\int_0^{\infty}\frac{1}{2\pi}\exp\left\{-\frac{R^2}{2}\right\}R\mathrm{d}\theta\mathrm{d}R = 1.$$

由此可得 R 与 θ 的分布函数:

$$\begin{aligned} P_R(R\leqslant r) &= \int_0^{2\pi}\int_0^{r}\frac{1}{2\pi}\exp\left\{-\frac{R^2}{2}\right\}R\mathrm{d}\theta\mathrm{d}R = 1-\exp\left\{-\frac{r^2}{2}\right\}, \\ P_\theta(\theta\leqslant\varphi) &= \int_0^{\varphi}\int_0^{\infty}\frac{1}{2\pi}\exp\left\{-\frac{R^2}{2}\right\}R\mathrm{d}\theta\mathrm{d}R = \frac{\varphi}{2\pi}. \end{aligned}$$

显然 θ 服从 $[0,2\pi]$ 上的均匀分布. 令

$$F_R(r) = 1-\exp\left\{-\frac{r^2}{2}\right\},$$

则其反函数

$$R = F_R^{-1}(z) = \sqrt{-2\ln(1-z)},$$

当 z 服从 $[0,1]$ 上的均匀分布时, R 的分布函数为 $F_R(r)$. 因此可以选取两个服从 $[0,1]$ 上均匀分布的随机变量 U_1, U_2 使得

$$\theta = 2\pi U_1,\ 1-z = U_2 \text{ 即} R = \sqrt{-2\ln U_2},$$

将其带入

$$X = R\cos\theta,\ Y = R\sin\theta$$

即可得到最初的两个关于 X, Y 的表达式, 它们服从标准正态分布. ∎

第二节　位反转序列

下面给出获得位反转序列的伪代码, 通过 Gold Rader Bit-Reversal 算法实现:

算法 A.1　Gold Rader Bit-Reversal Algorithm

Data: Input g[n]

Result: The Bit-reversed array

```
for i = 0 ... n - 2 do
    k = n / 2
    if i < j then
        swap g[i] and g[j]
    end
    while k ⩽ j do
        j ← j - k
        k ← k / 2
    end
    j ← j + k
end
```

C++ 代码 (CPU 中运行):

```cpp
int get_computation_layers(int num) {
  int nLayers = 0;
  int len = num;
  if (len == 2) return 1;
  while (true) {
    len = len / 2;
    nLayers++;
    if (len == 2) return nLayers + 1;
    if (len < 1) return -1;
  }
}
auto reverse(int N) {
  std::vector<int> bit_reversed(N, 0);
  int index = 0;
  int r = get_computation_layers(N);
  for (int i = 0; i < N; i++) {
    index = 0;
    for (int m = r - 1; m >= 0; m--) {
      index += (1 && (i & (1 << m))) << (r - m - 1);
    }
    bit_reversed[i] = index;
```

```
22      }
        return bit_reversed;
24  }
```

致　谢

匆匆五年, 科大在我的人生中留下不可磨灭的烙印. 临别之际, 许多话想说.

首先, 感谢本科期间的各位老师, 他们治学严谨, 教学认真, 在他们的课堂上我收获良多; 同时, 他们对学生尽心尽责, 也让我在迷茫之际有所指引. 其中, 我要特别向数学科学学院的张举勇教授和陈仁杰教授致谢. 我在大二时认识张教授, 他给了我初尝科研的机会, 通过完成他的作业, 我走进了计算机图形学世界的大门, 对这门充满魅力的学科打开了一个全新的视角; 在此推动下, 我在大三选修了陈教授的图形学课程, 在他的引领下在这领域不断深入, 并找到了自己喜欢并且愿意继续从事的方向. 另外, 陈教授在我毕业论文写作期间对我悉心指导, 扫清了我学习路上诸多障碍, 并给我许多鼓励, 在此再一次向他表示由衷的感谢. 另外, 我要特别感谢我院的刘利刚教授. 虽然我没有参与过他的课程, 但他仍然乐意地为我的留学申请提供慷慨无私的帮助, 让我追寻理想的步伐多了一份强有力的背书. 最后, 我还要向我的两位班主任老师: 李书敏副教授和许金兴副教授致以谢意, 他们尽职尽责, 关爱学生, 若没有他们的帮助, 我的本科生涯将平添许多困难.

我还要感谢这五年来陪伴在我身边朝夕相伴的同学们. 他们有的是我在排球队并肩战斗的队友, 有的是无私分享经验的学长学姐, 有的是拥有共同话题的难觅知音. 在我迷茫之际他们总能在身边, 我们互相帮助和鼓励, 让这一段艰苦的求学旅程不那么枯燥, 反而充满乐趣. 很幸运能结识他们, 成为一辈子的朋友.

最后, 最应当感谢的是父母无微不至的关心与爱护, 感谢他们对我各种选择都无条件支持, 不论这一路上我经历多少失败或是成功, 他们始终是我最坚实的后盾, 目送着我跌落又爬起. 没有他们, 就没有今天的我.

科大生活走到了尾声, 这五年来, 对她的自豪, 喜悦, 抑或是怀疑, 愤恨, 这些情绪都交融如织, 无可分割, 化为一股复杂的力量, 成为了我人生的一部分. 告别这一段旅程, 我的人生终将迎来崭新的篇章. 无论过去和未来怎样, 在这离别的关头, 我想我都愿心怀感激, 因为我始终坚信这一切的一切, 都是最好的安排. 也许科大教会我最深刻的一门课, 就是学会接纳, 学会放下吧: 不必强求, 更不必懈怠, 未来自有神采, 未来如期而至.

长风破浪会有时, 直挂云帆济沧海.

2023 年 5 月