分布式海洋渲染

匡正非、路橙、张卡尔

August 4, 2018

Contents

1 项目介绍

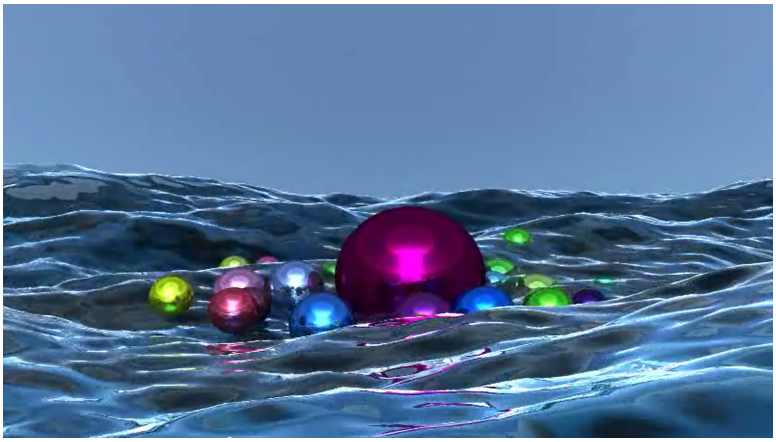
在计算机科学中，渲染一直是一项十分热门的前沿技术。在图像、电影、游戏等诸多领域内，都起着不可或缺的作用。自上世纪 70 年代左右提出起，渲染技术的发展为我们带来了质量越来越高的视觉产品，但同时，由于需要大量计算，它也对使用机器所能提供的计算能力提出了挑战。人们注意到，在渲染中涉及到的计算往往具有非常高的并行性，而只要采用了良好的并行策略便可以极大地提高渲染速度，因此为渲染设计良好的并行计算环境便成为了一门重要的研究领域。在这其中最为著名的当属 GPU，这种高度分散化、并行化的设备为高效率的渲染提供了可能，因此也获得了极快的发展。另一方面，如果提高单个硬件的并行度可以增加效率，那么利用多台机器并行工作也自然是一个有效的办法。在云技术出现之后，聪明的人类很快便将其也用在了渲染当中，这也就是云渲染，换句话说，便是利用分布式计算的方式来完成渲染的任务，从而进一步提高速度。

如果说计算能力只是工具的话，那么，要得到好的产品，渲染同时还需要一份像样的蓝图，这也成为了许多人专心研究的目标。为了让生成的图片或者视频更接近于真实世界（或者仅仅只是为了变得更加好看，符合人们的审美标准），学者们提出了愈来愈多的光照模型，这又涉及到了许多概念，例如 BRDF、光强、体积光等。而另一方面，如何搭建场景也成为一个重要的研究对象——显然大家不会仅仅为了生成一个好看的球和平面来耗费大量的计算成本，所以为了实现更加复杂、更加现实的模型，人们提出了很多解决方案，例如贴图技术、物理引擎、模拟算法以及各式各样的粒子渲染方法等等。我们此次试验所涉及的物理模型，便是在一篇 2004 年发布的论文《Ocean Simulation》中所提出，利用 FFT 算法模拟海面模型的一套方案——OceanFFT，这项成果已经成为了经典的图形学技术之一，甚至已经加入成为了 CUDA 的官方 Sample 程序之中，其影响力可见一斑。

对于 OceanFFT 而言，目前绝大多数已有的实现程序是通过 GPU 而完成的，但是我们的目标不仅于此。除了利用 GPU 以外，此次课程中我们所了解到的分布式计算，同样是一个用于解决并行计算问题的好手。事实上，无论是用于生成海洋的算法，还是将其渲染至图片的过程，我们都可以通过多机器同步运行来完成，这也正是我们这次实验的目标。

当然，仅仅完成一个海面，对于我们的计划来说仍然是不够的，因此为了体现海面的真实性，我们进一步在自己的计划之中加入了一个物理引擎，以表现出许多物体在海面上漂浮移动的效果。和许多往年同学们做过的项目类似，我们同样完成了一个基于分布式系统的实现，而且其功能完全可以支持之前的作业中提到过的各类需求。

在项目的最后，我们最终渲染出来了一篇长度为数秒的小视频作为本次项目的成果，视频和我们的代码地址都已经放在了本文的附录之中。下面是一张我们渲染结果的截图：



1.1 项目构成

本次项目一共包含三个模块：海面模拟模块、物理引擎模块以及渲染模块。海面模拟负责调用论文中的算法生成海面，物理引擎负责加入其它模型（球体），处理海面与模型以及模型之间的关系，渲染模块的功能则是将各个物体渲染成图片。

在接下来的内容中，第 2 章将介绍已有的一些工作（包括我们使用的论文的理论基础），第 3、4、5 章将分别介绍上述三个模块的实现方式，第 6 章着重介绍实现细节和测试结果，而第 7 章则是对项目的总体评价和未来的展望。

2 已有相关工作

2.1 Ocean Simulation

已知海洋表面满足 Navier-Stokes 流体动态方程，如下

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla p + \mathbf{F}$$

$$\nabla \cdot \mathbf{u}(x,t) = 0$$

OceanFFT 就是对 Navier-Stockes 方程进行简化后，使用 FFT 的方法加速求解。其核心方程为：

$$h(\mathbf{x},t) = \sum_{\mathbf{k}} \tilde{h}(\mathbf{k},t) exp(i\mathbf{k} \cdot \mathbf{x})$$

其中 $h(\mathbf{x},t)$ 为海洋平面在 $\mathbf{x}$ 坐标，t 时刻的海面高度。$\tilde{h}$ 为海面高度的频谱，$\mathbf{k}$ 为在频域 $\tilde{h}$ 中的坐标。

频谱 $\tilde{h}$ 在 $t=0$ 时的值可以通过 Phillips 光谱初始化得到：

$$\tilde{h}_0(\mathbf{k}) = \frac{1}{\sqrt{2}}(\xi_r + i\xi_i)\sqrt{P_h(\mathbf{k})}$$

$$P_h(\mathbf{k}) = A\frac{exp(-1/(kL)^2)}{k^4}|\hat{k} \cdot \hat{\omega}|$$

频谱 $\tilde{h}$ 在任意时刻的值可以通过 h 得到：

$$\tilde{h}(\mathbf{k},t) = \tilde{h}_0(\mathbf{k}) exp(i\omega(k)t) + \tilde{h}_0^*(\mathbf{k}) exp(-i\omega(k)t)$$

2.2 Path Tracing

Path Tracing 算法是光线追踪算法的一种最基础的实现，通过模拟相机发射的光线，逆向追踪，当碰撞到物体时，分别递归求出物体在光照下的颜色、物体镜面漫反射（蒙特卡洛模拟漫反射）的颜色、物体折射的颜色，将三者加权即可得到相机中单个像素点的颜色。这种算法可以较好地渲染出 3D 场景，且可以表达阴影、高光等基本的光学效果，且耗时相对较少，适合海洋场景渲染的基本要求。

在计算机图形学基础课程中，我们已经实现了基于 c++ 的单线程光线追踪算法，并实现相应的软阴影、超采样抗锯齿、景深等效果。由于当引入海洋网格时，每一条光线与复杂网格求交的计算量较大，尽管可以使用 kd-tree 进行加速，但时间消耗仍然较高。因此，海洋渲染的时间性能瓶颈并不在于 Path Tracing 算法本身，而在于引入复杂网格后求交的计算量。由于这一计算可以很好地并行，因此从理论上，分布式渲染可以极大地提高渲染的时间性能。

2.3 MapReduce

略。

3 Ocean FFT

3.1 $\tilde{h}$ 的初始化

根据 $\tilde{h}_0(\mathbf{k})$ 和 $\tilde{h}(\mathbf{k},t)$ 的公式可以用单机快速地生成海洋的初始频谱。由于 $tildeh_0$ 生成式中包含高斯随机数，因此每一次的生成的初始频谱都不相同。

$$\tilde{h}_0(\mathbf{k}) = \frac{1}{\sqrt{2}}(\xi_r + i\xi_i)\sqrt{P_h(\mathbf{k})}$$

$$P_h(\mathbf{k}) = A\frac{exp(-1/(kL)^2)}{k^4}|\hat{k} \cdot \hat{\omega}|$$

$$\tilde{h}(\mathbf{k},t) = \tilde{h}_0(\mathbf{k}) exp(i\omega(k)t) + \tilde{h}_0^*(\mathbf{k}) exp(-i\omega(k)t)$$

在实验中，我们的初始数据和 FFT 的规模为 256*256。

3.2 使用 MapReduce 计算 FFT

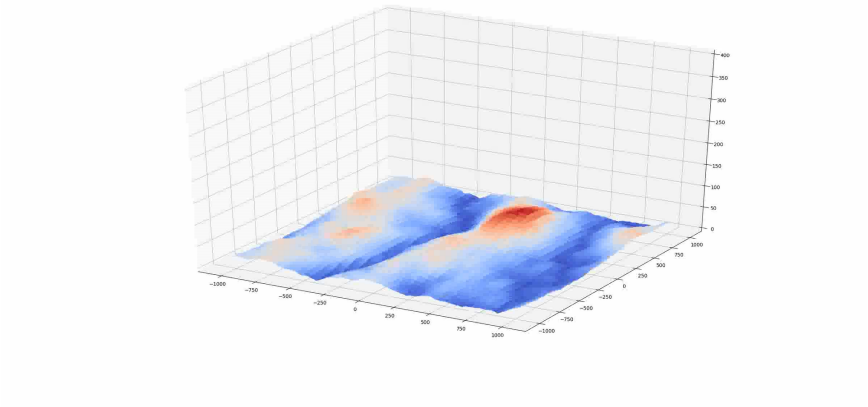
由于在二维 FFT 中，$\mathbf{k}\cdot\mathbf{x}$ 是各维之间独立的，因此我们可以对其进行拆分，将其表示为：

$$\sum_n(\sum_m \tilde{h}((n,m),t)\exp(\frac{2\pi m x_1 i}{N}))exp(\frac{2\pi n x_0 i}{N})$$

这实际上相当于先对二维数组的各行分别进行 FFT 运算，再对各列进行 FFT 运算。这样的计算方式实际上非常适合利用 MapReduce 完成：Mapper 的输入为一行的频谱数据，其功能为为对 1 行进行一维 FFT。以列数为 key 值，按元素进行 emit；Reducer 的输入为通过由一维 FFT 结果组成的一列数据，功能为进行第二次一维 FFT。将结果直接输出，再进行一次转置，即可得到在空间域上的海洋高度谱。

由于 FFT 的计算过程中，互不影响，因此可以将所有帧的初始数据全部输入，一次性得到全部帧的结果。整个过程速度较快，约 10 分钟即可得到 300 帧的海洋高度。

下图是利用 matplot 显示海面的结果：



4 物理引擎

我们此次使用的物理引擎在很大程度上参照了之前 NBody 项目的设计，为了进行简化，同样也只实现了球体的运动。在这样一个球体与海面共存的世界中，我们可以把可能发生的事件分为四种：

1. 球体受到来自海面的浮力
2. 球体之间互相碰撞
3. 球体受到重力
4. 海面受到来自球体的冲击力

对于第四种力，实际上在 Ocean Simulation 中的第五章给出过一个基于卷积运算的方案，但是由于这种方案并不适用于由 FFT 生成的海面，适用于一个平静的水面，所以我们无法将其采纳。对于 FFT 海面，我们最终没有得出一个较好的模拟受到小球冲击而产生水花的方案，因此在实现中忽略了这一部分。不过，当受到冲击力时，海面同时会对小球提供反作用力，这一点我们采用了阻尼的方式进行实现——当小球接触到水面时，每一时段速度会乘上一个阻尼常数。

除去冲击力和实现简单的重力以外，前两种力是相对而言比较复杂的。下面对其进行详细的说明：

4.1 浮力

计算浮力最为简单的方案是直接使用浮力公式：

$$F_{buoy} = \rho g V$$

所以，看似只要求出球与海洋的相交体积，就能够算出球所受到的浮力了。这在平静的水面上看来是可行、可靠的，但如果在一片波涛汹涌的海面上，事情没有这么简单。首先，如果海洋中的水是流动的，则一定会对小球产生额外的推力，而不仅仅是重力带来的压强；其次，按照浮力公式而言，浮力的方向往往都是垂直向上的，但是在这里，小球受到的压强未必垂直向上；再者而言，海面每时每刻都在发生着变化，因此与小球的相交体积也在不断变化。为了解决这些问题，我们采用了一系列方案（包括简化、近似等）来得出一个可以接受的解，具体包含以下几点：

1. 将海面按照各个采样点分成小格，分别计算每一格对小球的作用力然后求和
2. 对于每一格而言，用立方体近似在这一格中小球在海面以下的体积
3. 将浮力拆分成两部分：一部分是垂直向上的力，另一部分则是朝向球心方向的水平力
4. 对于第 3 步中的水平力，与水面受到的风向进行点积，以近似出额外受到的推力

对于一个小球 $B = (\mathbf{C}, r)$ 而言，我们将其受到的浮力表示为：

$$F_{buoy} = \sum_{i,j \in (0,N)} \rho g V_{B,ocean(i,j)}$$

其中 $ocean(i,j)$ 表示点 (i,j) 处小格的海面以下部分，将该式简化后可得：

$$F_{buoy} = \rho g \sum_{i,j \ in(0,N)} area \cdot dis(i,j,t)$$

其中 area 表示每个小格的面积：

$$dis(x) = \max(0, \min(ballBottom(i,j) - h((i,j),t), ballTop(i,j) - ballBottom(i,j)))$$

可以看出，只要算出小球在每个格子的中心处对应的高度，就可以近似求出小球受到的浮力了。但是，到了这一步，我们还没有使用之前提到的第 3、4 个优化，因此，需要将上式改成向量形式：

$$\mathbf{F}_{Up} = \rho \cdot g \cdot area \sum_{i,j \in (0,N)} dis(i,j,t) \cdot (0,0,1)$$

$$\mathbf{F}_{Horizonal} = \rho \cdot g \cdot area \sum_{i,j \in (0,N)} dis(i,j,t) \cdot (\mathbf{C} - \mathbf{ballBottom}(B,i,j))_{normalized}$$

4.2 小球碰撞

对于两个运动中的小球 $B_1, b_2(\mathbf{C}, r, \mathbf{v})$，其碰撞后的速度可以由碰撞公式得出（这里只显示 B_1 的情况）：

$$\hat{\mathbf{v}}_1 = \mathbf{v}_1 - \frac{2\mathbf{m}_1}{\mathbf{m}_1 + \mathbf{m}_2} \frac{< \mathbf{v}_1 - \mathbf{v}_2, \mathbf{x}_1 - \mathbf{x}_2 >}{||\mathbf{x}_1 - \mathbf{x}_2||^2} (\mathbf{x}_1 - \mathbf{x}_2)$$

对于一个时间片段，我们假定一个小球只会受到一次碰撞。因此对于每个球，我们只要找出所有的球中最早与其碰撞的那个球，然后利用上述公式，便能够计算出该球之后的运动轨迹了。除了碰撞以外，事实上，小球之间总是会不可避免地发生穿模的情况，因此对于穿模时的两个小球，对双方都要施加一个反弹力将其分开：

$$\mathbf{F}_{bounce} = c_b \cdot (||\mathbf{C}_1 - \mathbf{C}_2|| - r_1 - r_2)$$

c_b 为常数，在实验中取值为 30000。

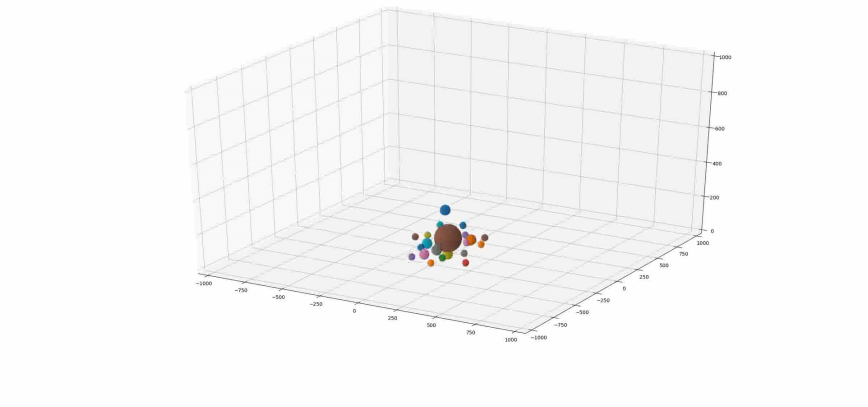
另外，在一些情况下（尤其是小球堆叠在一块的时候），一个小球可能会同时受到来自多处的碰撞，对于这种情况，我们采用的解决方案是：在确定了一个小球遇到的第一次碰撞之后，将接下来一个极小的碰撞窗口内所有的碰撞都看作是同时发生的，都对其进行处理。

4.3 MapReduce 实现框架

在实现当中，我们采用了逐帧迭代的方法，每次将该帧的海洋平面和小球信息作为输入，进行一次 mapreduce 之后，输出下一帧的小球信息。

对于 Mapper，首先会在 setup 时预加载所有输入小球的 AABB 盒，然后在 Map 中，每当输入一个小球或者一行海洋时（注意海洋的输入即为 OceanFFT 的输出），判断其和哪些 AABB 和相交，然后将该球/海洋 emit 到对方的集合当中（以对方的 id 为 key 值）。而在 reducer 中，每个小球会根据在 mapper 中获取到的与其相交的物体依次计算它们对其提供的作用力，然后根据这些力的产生时间和大小直接统计出下一轮的位置和速度。

下面的这张图是在计算完小球的位置之后利用 matplot 提前渲染的结果截图：（没有包含海面）



5 渲染引擎

渲染引擎采用 Path Tracing 作为渲染算法，并采用 mapReduce 进行加速。其中，海洋模型首先转换为 obj 格式以三角面片的形式输入到渲染引擎中，并采用 boudingBox + kd-tree 进行加速求交。此外，对网格的法向量采用经典的网格平滑算法进行了平滑处理，使得海洋的波浪较为平滑。为了较好地展现小球和海洋的光影，以及小球在水面下经过折射的效果，渲染引擎中还实现了超采样抗锯齿、软阴影等效果。

渲染引擎全部用 Java 语言实现，外部库只有 hadoop 包，其余所有算法全部使用 java SDK 自带包实现。

5.1 光线追踪算法

在 renderer/tracer 中实现了 Path Tracing 算法的单机版和分布式多机版算法。其中 RayTracer 是单机版实现，而 RayTracerRunner 是多机中每个机器中运行算法的类，RayTracerDriver、RayTracerMapper、RayTracerReducer 分别是 mapReduce 的启动、Mapper、Reducer 类。

MapReduce 过程中，输入文件是文本文件，每行的格式为 “i,j”，表示相机中行 i 列 j 的像素。因此，每个 map 函数的输入的 value 为 “i,j” 字符串，表示该 map 函数需要计算行 i 列 j 的像素点的颜色值。

每个 Mapper 在 setup 时读入场景文件（以文本形式表示），建立物体组成的场景和算法主要参数，当有网格模型时同时建立 kd-tree。在 map 函数中，从行 i 列 j 对应相机的位置发射出光线，执行光线追踪算法，得到该像素的颜色值。并且，为了实现软阴影，每个像素点还需要记录一个特定的 hash 值，表示这个像素在追踪时的碰撞序列。这样一来，若相邻两个像素点的这一数值不同，表示它们恰好在物体边缘，需要进行超采样以抗锯齿。Mapper 将数据发给同一个 Reducer，数据中包含了像素位置、颜色、hash 值。

每个 Reducer 在 setup 时与 Mapper 类似，也建立好对应的场景和数据结构，并汇总收到的所有颜色，得到初始图，并最终执行一次 resample，对所有需要超采样的点进行超采样。最终得到的图使用 ImageIO 等 java SDK 内部包进行输出。

由于采用 MapReduce 框架，光线追踪算法实现了像素级的并行，时间性能有很大的提升。但由于我们的场景并不是太复杂，在 mapReduce 的 setup 过程中 kd-tree 的建立也较为耗时，因此当机器数量较少时，并行加速比并不高。

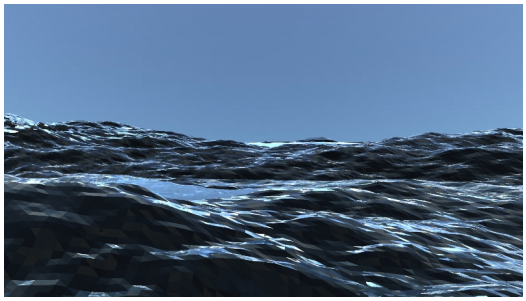


Figure 1: 平滑处理前的海洋

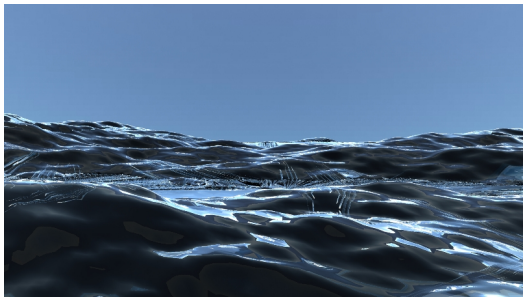


Figure 2: 平滑处理后的海洋

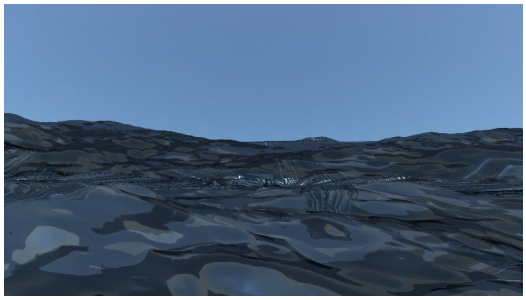


Figure 3: 增加 BRDF 前的海洋

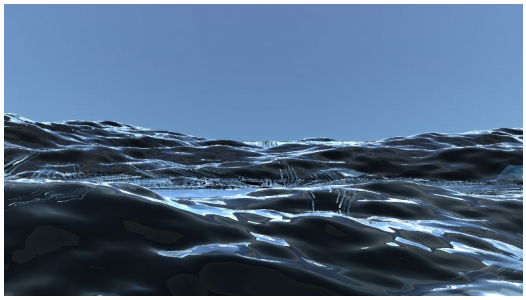


Figure 4: 增加 BRDF 后的海洋

5.2 网格求交加速与平滑处理

复杂模型常常以三角面片的形式近似表达，当模型纹理变化较大时，需要的面片数量往往很多。在我们的实验中，一帧的海洋模型有 17 万左右的面片，若每一根光线都要遍历面片进行求交，那么算法的复杂度是不可接受的。

在计算机图形学中，kd-tree 是网格求交的经典加速算法。在导入 obj 模型后进行建树，设面片数量为 n，初始化建树的时间消耗为 $n\log(n)$，而求交的时间消耗的数学期望仅仅为 $\log(n)$。因此，我们采用 kd-tree 算法进行求交加速后，可以极大地减小光线求交时的消耗。

由于 OceanFFT 只能得到海洋平面的采样点，若直接使用采样点构成的三角面片进行渲染，并不符合网格简化的基本原理，光滑程度也非常低。因此，有必要对网格进行平滑处理。

具体而言，对于每个三角形的顶点，计算包含它的所有三角面片的法向量，并将这些法向量进行加权平均，其中权重为该顶点在这一三角形中对应角的余弦值，计算结果作为该点的法向量 vn。对于三角面片中任何一个点 O，设该三角形为 ΔABC，并设 O 与边 AB、BC、AC 的距离分别为 l_{AB}、l_{BC}、l_{AC}，点 A、B、C 对应三角形的高分别为 h_A、h_B、h_C，那么点 O 在网格中的法向量计算式为：

$$vn_O = \frac{l_{BC}}{h_A}vn_A + \frac{l_{AC}}{h_B}vn_B + \frac{l_{AB}}{h_C}vn_C$$

经过这样的平滑处理后，海洋网格的平滑性有了显著提高，见图1、2。

5.3 海洋 BRDF 的实现

双向反射分布函数（Bidirectional Reflectance Distribution Function，BRDF）用来定义给定入射方向上的辐射照度对给定出射方向上的辐射率的影响，它描述了入射光线经过某个表面反射后如何在各个出射方向上分布。由于海洋在不同的起伏位置往往会有不同的光学表现，折射、反射等光学性质并不能用同一个常量来表现，因此，我们还实现了 [1] 中提出的海洋表面的 BRDF 函数，用来刻画光线在不同海洋位置的反射、折射比例。

假设海洋表面没有漫反射，全部光效应只包含反射光和折射光，则可以用 R 和 T 来表示反射光和折射光能量的比例关系，且有 $R+T=1$。根据海洋表面的光学物理是性质，R 可以通过入射角和反射角得到：

$$R(\hat{\mathbf{n}}_i, \hat{\mathbf{n}}_r) = \frac{1}{2}\{\frac{\sin^2(\theta_t-\theta_i)}{\sin^2(\theta_t+\theta_i)} + \frac{\tan^2(\theta_t-\theta_i)}{\tan^2(\theta_t+\theta_i)}\}$$

测试项目	OceanFFT	物理引擎	渲染
规模	T=300, 采样点 $=256 \times 256$	T=300, 小球数量 =30	1 帧,450×800, 光线追踪深度 =5, 海洋面片数量 =171494
单机时间	2min35s	5min40s	15min
集群时间	1min8s	28min	13min

Table 1: 实验性能测试结果

6 代码实现、测试

6.1 实验环境

实验代码需要 java SDK 1.8，hadoop 3.0.3。我们分别在单机、双机集群上进行了性能测试。

6.2 测试结果

我们分别测试了 oceanFFT、物理引擎、图片渲染在单机和多机下的时间性能，见表1。

6.3 结论分析

对于 OceanFFT 的快速傅里叶变换的计算而言，由于 mapReduce 计算的并行度高，可以很大程度地简化并加速二维 FFT 的计算。并且，由于 mapReduce 的计算可以将所有帧的所有海洋采样点同时计算，一轮 mapReduce 即可得到最终结果，因此 OceanFFT 的二维傅里叶变换十分适合使用 mapReduce 进行加速。

对于物理引擎而言，由于每一帧计算都需要执行一轮 mapReduce，而 mapReducce 的额外建立时间较长，因此在引入 mapReduce 后，执行多帧的时间反而增加了很多。由于 mapReduce 在物理引擎上只用来计算不同小球之间的碰撞、小球受到的浮力大小，这一加速难以弥补额外的 mapReduce 任务建立开销，因此对于我们这样的简单物理引擎而言，mapReduce 反而会起到减速的效果。

对于渲染任务而言，每一帧的性能主要取决于图像分辨率、光线追踪深度、三角面片数量。在实验中，当这三个参数变大时，mapReduce 的加速比也会更高。但由于 resample 在 Reducer 中单线程进行，而复杂场景（有很多球时）往往绝大多数像素点都需要超采样（光线追踪深度为 1），因此 Reducer 的单机计算也会变成性能瓶颈，故我们的 mapReduce 算法只减少了 Mapper 的时间消耗，并不能做到完全线性地加速。

7 未来展望

我们项目的未来可能尝试的改进有：

1. 当前天空只是使用一个大球体模拟了地球，但没有环境光。未来可以采用体积光，加入云朵、大气效果
2. 改进水面的 BRDF，得到更接近自然的水面效果
3. 采用其它的分布式框架（如 spark）来加速
4. 加入墙体等障碍物，实现与海面的碰撞
5. 渲染图中加入其它物体元素，如加入船只
6. （难）实现水面和物体互动效果 0 如球进入水后溅出水花、有水面波纹等
7. （难）实现水底不平整时的水面效果

8 附录

8.1 小组分工

匡正非：OceanFFT 的 FFT 部分、物理引擎、部分 Demo、性能测试

路橙：框架搭建、渲染引擎

张卡尔：框架搭建、OceanFFT 的初始化和 BRDF 部分、前后端合并

8.2 项目地址

我们小组的项目代码见https://github.com/LuChengTHU/DistributedSystem-OceanRenderring

References

[1] J Tessendorf. Simulating ocean water. *Simulating Nature Realistic & Interactive Techniques Siggraph*, 2001.