硕士论文| LUND大学2016

体积的实时渲染云

里卡德奥拉

计算机科学系LTH

ISSN 1650-2884

内容

第1章
介绍

游戏开发者总是在推动他们的游戏在某些方面是最现实的。真实和令人印象深刻的图形一直是这样的一个方面。但他们只能在计算能力成为一个问题之前把图形推到这么远。这篇硕士论文将研究视觉和计算性能之间的平衡，特别是它将研究生成体积云和云景的不同方法。长期以来，使用立方体映射一直是在三维世界中渲染云的常见做法。这意味着，天空、云、甚至遥远的风景的图像被放置在观众周围的一个锁定的距离上，给人一种无限遥远的环境的印象。当观察者被放置在靠近地面的地方，并且不会走得太远，比如第一人称射击游戏和赛车游戏时，观察者周围就会有天空的错觉。但在日益流行的开放世界游戏和飞行模拟器中，天空盒会给人一种静态的感觉，而旅行的感觉也会消失。当使用天空盒时，也不可能渲染场景靠近云或在云的内部，这可能是可取的。为了解决这一点，必须将物理云放置在3D场景中。这可以使用二维纹理来完成，但结果可能看起来很平，不太现实；良好的结果需要体积云。

本硕士论文的目标是产生体积云，并探索不同的技术和方法，在给定的情况下，实时呈现它们。实时性意味着我们的目标是一个大约33毫秒的帧时间——有效地给我们提供了一个30帧/秒的渲染频率（FPS）[1].我们将首先实现一个模型，反映在真实云中发生的事情，然后做出权衡和算法选择来实现所需的帧时间。帧时间目标当然是任意的，算法需要为真实的应用程序重新调整。尽管结果会有所不同，但本文中讨论的算法仍然可以被使用。

在图1中。1和1.2，云的照片展示了我们希望完成的效果。图1。1也显示了我们所追求的云的形式

[1]Akenine-Moller等。[1]将实时定义为至少15帧，但为了确保图像不被认为太波动，我们的目标是30帧，而游戏开发者的目标通常是30-60帧。

当光线通过云层时，光线的照明效果会减弱。另一个图，图1。2，展示了太阳光如何散射通过云，这导致明亮的边缘和黑暗的核心，除了覆盖太阳的云是明亮的。

图1.1：一张照片，显示了具有圆形顶部和平坦底部的云。我们可以看到光在穿过云层时是如何衰减的。

图1。2：一张照片显示了太阳方向的光（在照片中心的云后面）有更多的前向散射。

本章的其余部分将解释底层理论，如渲染管道

以及人们是如何访问它的。在这章的最后，我们将讨论一些关于云的理论以及早期的工作和贡献。

.11渲染管道

渲染管道是我们获取应用程序中描述的信息并将其显示在我们的显示器上的过程，换句话说，是在2D显示器上显示3D空间的过程。通常，渲染管道如图1.3所示。在本章的解释中，我们将考虑具有可编程阶段的现代类型的渲染管道。

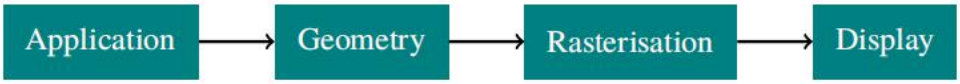


图1.3：渲染管道的一个非常基本的轮廓。应用程序是3D世界，显示是最终用户看到的。下面将更详细地解释所有阶段。来源：［14］。

1.1.1应用程序阶段

在应用程序阶段，将构建要渲染的场景。它可以由模型、照相机、灯等组成。通常，这些都被描述为在世界空间中的一个位置和某种方向。以一个立方体为例。我们可以描述为8个点（0、0、0）、（1、0、0）、（1、1、0）、（0、1、0）、（0、0、1）、（0、1、1）、（1、0、1）和（1、1、1），然后进入世界空间（2、2、1）（见图1.4)。正是这些点、位置和方向被输入到了几何图形阶段。

这意味着场景中的每个对象都有它自己的坐标系。因为目标是渲染到一个显示器，即。在显示器的坐标系（屏幕空间）中表示场景，需要进行一些转换。这些变换由应用阶段计算和提供，以矩阵的形式在几何阶段上传递，以及上述的点、位置和方向。这些点通常被称为原语，因为它们描述了原始的几何形状，如三角形、四边形或其他多边形。

.1.21几何阶段

几何阶段由几个子阶段组成，通常被分为图1.5［1,14]中所示的步骤。这一阶段涉及大量的转换，在后面第2.4节中介绍的射线探测器的构建中将非常重要。

通过几何阶段，一个定义为顶点（基元的角）的模型将从在模型空间中表示到屏幕空间坐标。模型、视图和投影变换矩阵都是从

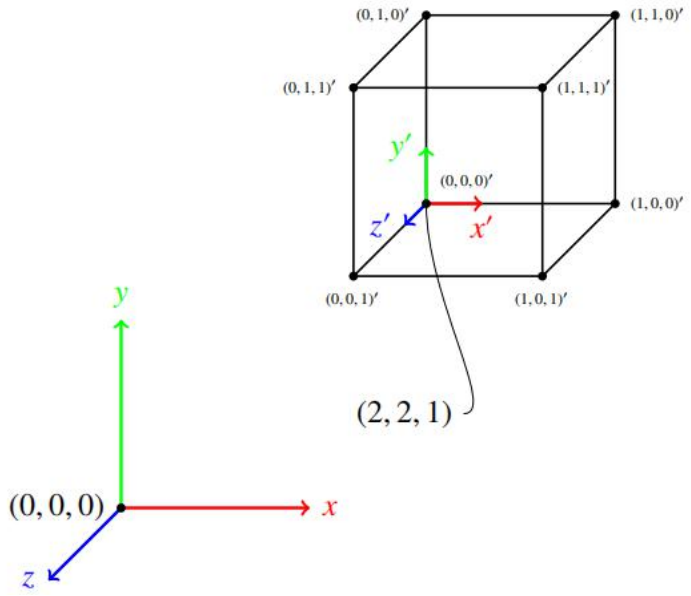


图1.4：该模型具有其局部坐标系（模型空间），然后通过定义立方体模型空间的原点的位置，将其放置在世界空间中。还可以为每个对象存储其他变换，如旋转或缩放。请注意，计算机图形学中的坐标系统并不总是右手边的系统。

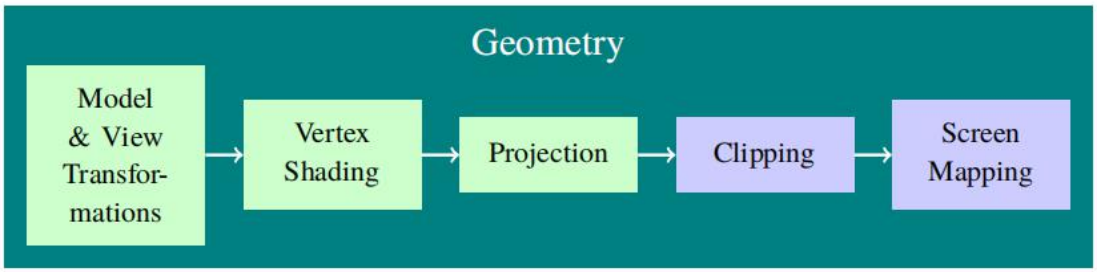


图1.5：几何图形阶段的子阶段。前三个子阶段是顶点着色器的一部分，并且是渲染管道中的第一个可编程阶段。来源：[14]。

应用程序阶段和通常应用在顶点着色器中，这是渲染管道中的第一个可编程阶段。“顶点着色器”传递每个顶点的顶点属性；顶点属性通常包含顶点位置、顶点颜色和纹理坐标。这允许我们在一个顶点的基础上修改最终结果，比如计算每个顶点的照明或将一个高度映射应用到一个平面上。这被称为顶点阴影。投影是顶点着色器的最后一部分，例如允许我们选择透视或正交投影。我们将使用透视投影作为相机如何捕捉场景；正交投影（也称为平行投影）保留了平行的线，可以用于艺术效果。

剪切和屏幕映射是几何图形阶段的最后两个子阶段。由于场景中的所有内容都不同时可见，所以所有的原语都在visi-之外

该场景由于无法继续通过管道而终止，部分可见的原语将被裁剪或切断，以便从管道中删除不可见的部分。屏幕映射会将可见的场景映射到用户的坐标系
屏幕

1.1.3变速阶段

栅化阶段是实际决定屏幕像素分配什么颜色的阶段。与几何阶段一样，光栅化阶段可以分为几个子阶段，如图1.6所示。

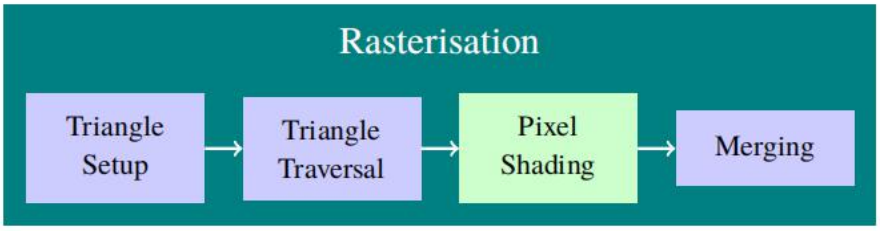


图1.6：光栅格化步骤的子阶段。像素着色步骤是渲染管道中的一个可编程阶段。
来源： [1]。

在三角形设置中，原语的顶点属性被处理并用于计算，以确定原语的面应该是什么样子的。*三角形遍历，有时被称为扫描转换，遍历三角形的面，以检查它们覆盖的屏幕像素。*

渲染管道的下一个可编程阶段是在栅格化阶段——像素着色器[2].本文中的大部分算法将在像素着色器中执行，这允许我们创建诸如每像素照明和纹理映射等效果。

栅格化的最后一步是合并。在像素着色器中计算出的颜色在这里与包含先验像素信息的颜色缓冲区合并。合并阶段还可以处理像素的可见性。这通常是使用z-缓冲区[1]来完成的。

.1.41显示阶段

渲染管道的最后一个阶段是显示阶段。这是将结果显示在屏幕上的结果。或者更确切地说，结果被发送到一个可能被发送到屏幕的框架缓冲区。我们将使用框架缓冲区来“捕获”结果，以便我们可以在向最终用户呈现这些结果之前添加额外的效果。

[2]实际上，前面有几何着色器和镶嵌着色器，但它们是可选的，不会在本文中使用。

.21图形处理单元

为了使渲染管道高效工作，包括所有阶段和转换，使用图形处理单元（GPU）加速硬件过程。GPU可以集成在主板或CPU中，或者以外部显卡的形式出现。gpu是专门通过并行工作来执行图形计算的，利用矢量化和SIMD单元[1,2]。

.31 OpenGL

有不同的api可以与GPU交互。其中一些更知名的是DirectX，
OpenGL和Vulkan。我们将使用OpenGL（开放图形库）来访问GPU。OpenGL是由Khronos集团开发的，使用一个扩展系统，允许最新的特性立即使用[1]。OpenGL还提供了它自己的着色器语言，即OpenGL着色语言（GLSL）。它用于编写着色器程序，这是渲染管道中的可编程阶段。GLSL是一种类似c的语言，它允许访问显式地用来处理图形的函数。着色程序使用所选API提供的驱动程序在GPU上编译。

.41云理论

云由小的液滴或晶体组成，主要由水组成，是在潮湿空气上升和达到低气压时膨胀时形成的。这是因为当空气膨胀时，温度下降，潮湿空气中的水蒸气凝结。云的形成和结构是非常动态的，主要是垂直运动的结果。我们将考虑积云，它们从对流中得到它们的垂直运动。这些垂直运动可以比作一种稠密的彩色液体被扔进水箱，产生一个循环涡旋[10]。对流在云的特征中起着很重要的作用。影响云团特征的另一个方面是液滴的大小分布。液滴的尺寸分布影响了Mie散射，准确地描述了不同角度接近[3]时光子在水滴中的散射。图1.7显示了由PhilipLaven的MiePlot软件[9]计算出的Mie散射相位函数。该软件允许通过输入不同的空气湿度和温度、平均液滴大小、液滴大小分布、光的波长等值来构建各种不同的情况。我们不会考虑不同的情况或波长相关的散射，而是将这种性能密集的散射与另一种计算上更便宜的散射类型进行比较。在这种情况下，选择了可见光谱中的任意波长，以及大气设置的常温和压力。如前所述，液滴的尺寸分布会影响散射，并选择作为布瑟尔等人所讨论的修正伽马分布。[3]和Mason [10]。.7图1中的图显示，大部分的光是向前散射的，但也有一些其他的峰，导致了一些有趣的现象。在140°左右的峰值被称为雾弓，在180°处的峰值被称为a

光荣关于这些现象的照片见图1.8。大的圆是雾弓，中间小的圆，围绕着摄影师的阴影，是
光荣

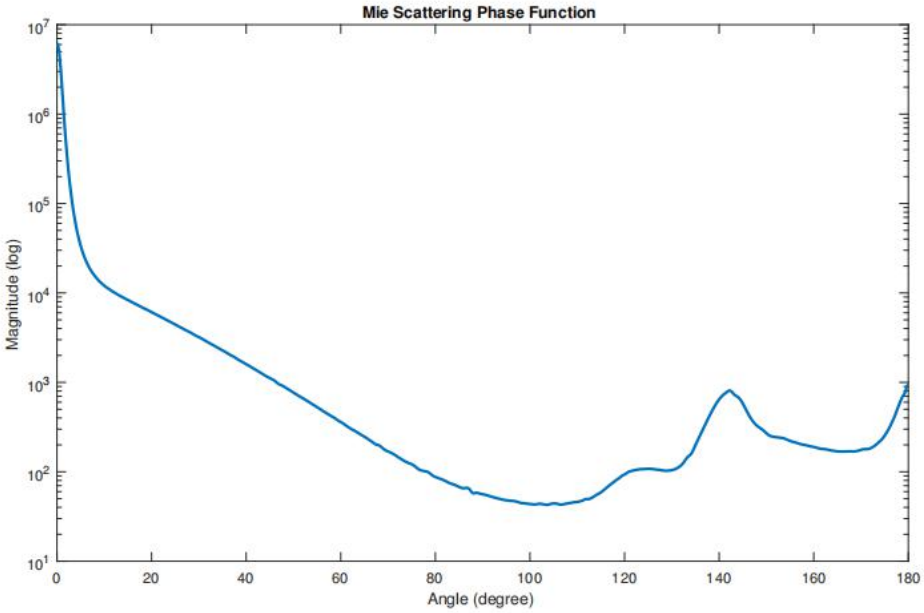


图1.7：显示光在不同方向散射时的强度的图。

图1.8：一张显示雾弓和荣耀的照片。来源： [7]

1.5相关工作

布瑟等人。[3,4]讨论了利用Mie理论计算云中的光散射。他们还诉诸于菲利普·拉文的MiePlot软件来获得真实的光线

散射功能，但对高阶光散射的方法不同，并利用云表面的集电极板来计算入射光。

Harris和Lastra [6]使用了一个类似于我们的实现，但却使用双向散射分布函数（BSDF）来处理他们的多次前向散射。对于相位函数，他们使用瑞利散射，并提到这可以代替一个更基于物理的函数。

施耐德和Vos [13]也提出了一个与我们非常相似的实现，但没有讨论一个更基于物理的相位函数的使用，比如Mie相位函数。

.61贡献

我们的实现比较了一个更简单的相位函数（亨耶-格林斯坦）和一个更基于物理的函数（Mie）的使用，以及实际意义，因为Mie相位函数实时计算。我们还提出了一种构建云的方法，通过从噪声纹理块中雕刻出云的形状。

第2章
实施

在本章中，我们将讨论不同的技术探索，以找到理想的性能，同时不损害视觉结果太多。第一部分在进入实现详细信息之前，给出了云生成和渲染的大纲。

.12.云渲染算法

为了真实地渲染云层，我们需要追踪到从太阳辐射出来的每个光子，跟踪它们在云层中分散的路径，并记录最终在相机中的几个光子。这当然是非常密集的计算，我们不想追踪所有的光子。这就是为什么我们的云渲染算法是建立在光线行进技术之上的，这意味着我们会将光线从摄像机反转到场景中。从相机投射的光线执行等距样本，以确定是否要渲染云。这些云是通过程序生成并存储在场景中的3D纹理中的。在云中的每个采样点上执行第二个射线行进。这条光线被投射向太阳，以进行照明计算。这意味着我们只考虑一阶散射。在第2.5节中，给出了高阶散射的近似值。图2。1显示了整个云渲染算法的基本设置。

2.2噪声功能

渲染程序云或任何程序云的典型方法是使用噪声函数，无论是2D云还是体积3D云。一个完全的白噪声缺乏结构，看起来也不太好看，因为它不像任何东西，所以在计算机图形学中，有不同的方法来修改白噪声和创造

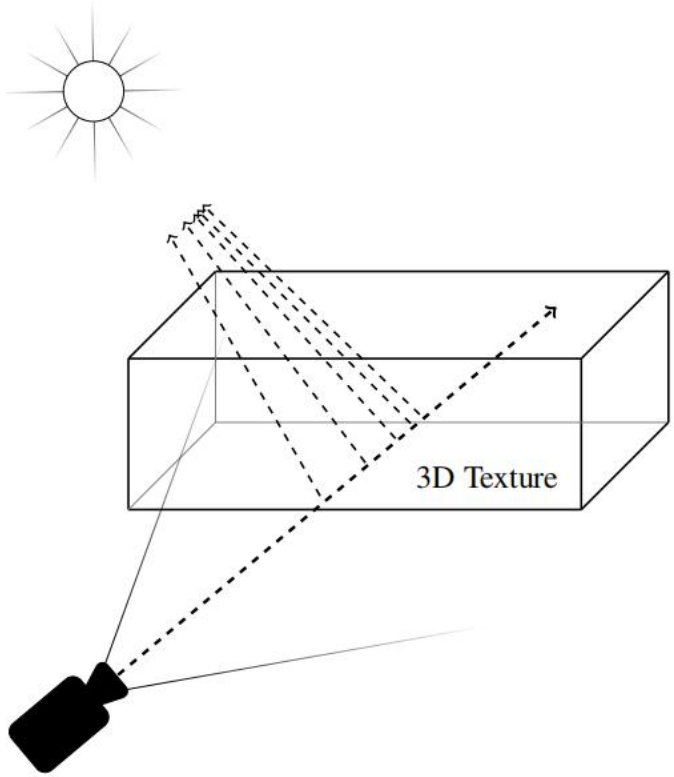


图2.1：从摄像机中为屏幕上的每个像素投射一条射线。光线在场景中行进，以预先设定的时间间隔进行采样。如果光线对云内的一个点进行采样，第二条射线就会投射到太阳上，以计算有多少光到达那个点。

一些具有更多结构而不引入人工制品的东西。这通常意味着我们想要保持所产生的纹理的梯度连续，即。纹理应该是C字符里的^2.我们还希望构造比纹理更大的云覆盖层，而没有可见的边缘，所以纹理需要连续地包裹在纹理的边缘上。

下面提出的噪声产生算法可能相当昂贵，因此所产生的纹理被预先计算并存储在硬盘驱动器上。

.2.12珀林噪声

1983年，肯·佩林发明了一种梯度噪声，现在通常被称为佩林噪声。Perlin已经改进了他的算法[12,11]，这有时被称为改进的Perlin噪声。我们使用的是这个改进后的版本。

Perlin噪声是基于格的，是通过为格的每个交点分配一个随机梯度向量来生成的（见图2.2）。当在特定坐标上读取噪声值时，确定该坐标所在的晶格单元，并通过使用点积和线性间插值该单元的角梯度来计算该值
polation.图2.3显示了由此产生的噪声纹理的外观。通过将梯度向量分配给一个重复模式中的晶格，可以获得一个包裹纹理。

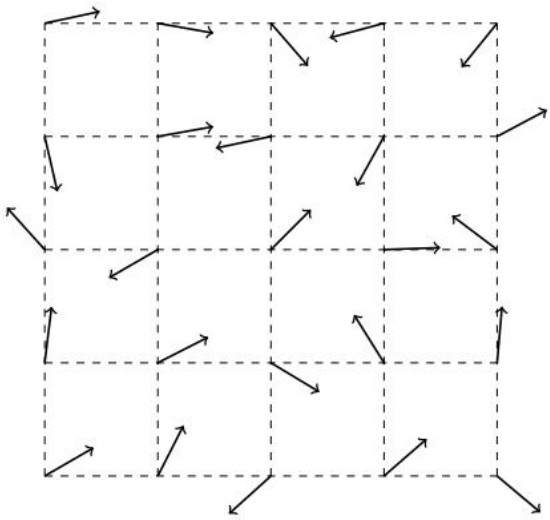


图2.2： Perlin噪声使用一个随机分配单位向量的格，表示格交点处纹理的梯度。

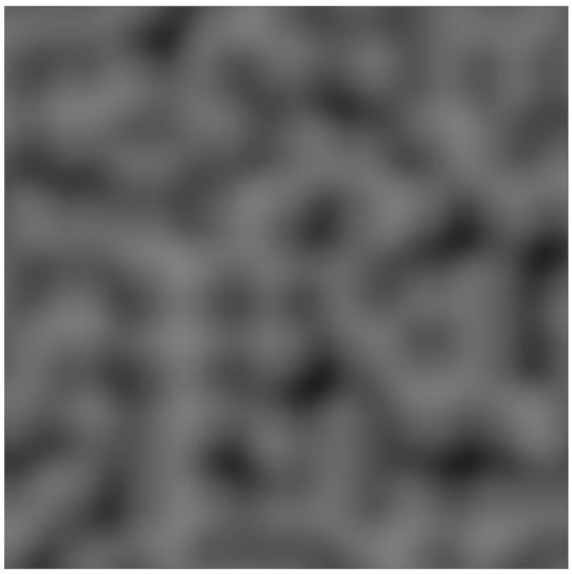


图2.3：这就是Perlin噪音的样子。纹理看起来是随机的，但它有一些结构，使它适合作为一个基础纹理，当模仿自然现象时，如木材和大理石表面或云。我们使用三维纹理，这张图像显示了一个佩林噪声三维纹理的横截面。

.2.22细胞噪声

蜂窝噪声，也通常被称为沃利噪声或沃罗诺伊噪声，是一种基于点的噪声
与基于晶格的Perlin噪声相反。它是由史蒂文·沃利在1996年的[16]公司首次推出的。其思想是在空间中随机选取特征点，然后为空间中的每一个点分配对应的范围到最近的特征点的值。在我们的单元格噪声的3D实现中，我们将空间划分为单元格并分配一个

每个单元格的特征点。这意味着与样本点最近的特征点必须位于同一个单元格或其中一个相邻的单元格中，如图2.4所示。就像Perlin噪声中的梯度向量一样，特征点是以重复的模式生成的，重复的一次迭代精确地拟合到纹理中，以确保纹理在边缘连续包裹。

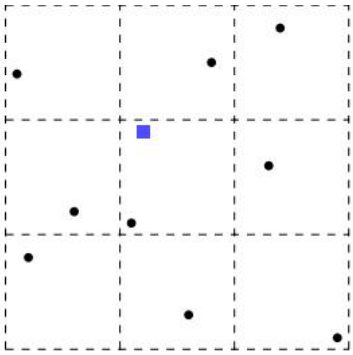


图2.4：这就是蜂窝噪声设置的样子。黑色的圆圈是随机选择的特征点。在这个设置中，每个单元格中恰好存在一个特征点。当噪声函数被采样时（在蓝色方块处），只需要检查邻近的单元是否最近的特征点。xx这意味着当使用3D纹理时，每个样本点都需要分析3 3 3个单元格块。

为了得到适用于云生成的结果，我们不采样最短的结果到一个特征点的距离，而是理论上的最大距离减去最短的距离采样范围。对于由单位立方体组成的单元，理论最大距离(空间对角线)是p3。因此，纹理的值(在这个函数中计算为该值也被归一化，以得到在[0,1]范围内的结果)

$$value = 1.0 - \frac{shortest\_distance}{\sqrt{3}}.$$

这就给出了一个在每个特征点周围都有球状特征的结果，这可以更好地模拟云的外观。我们的三维细胞噪声的横截面如图2.5所示。

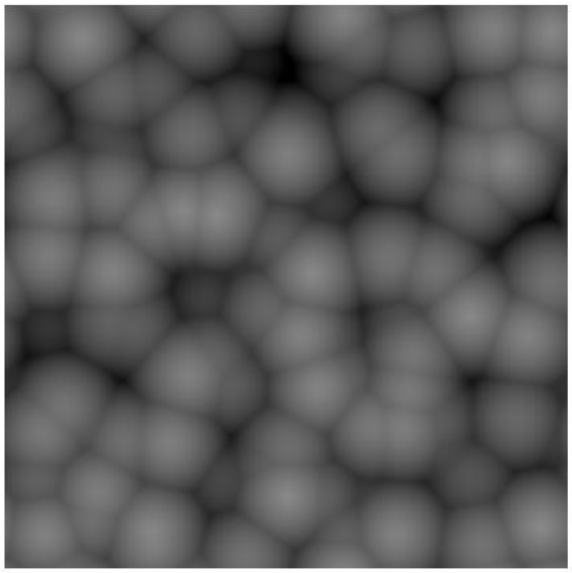
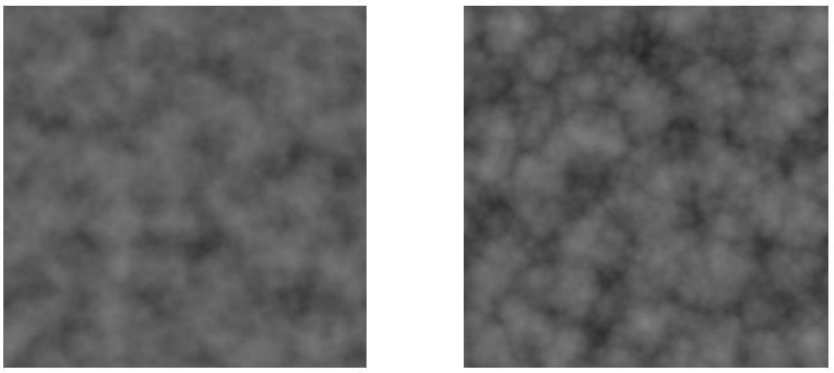


图2.5：这就是蜂窝噪声的样子。我们可以看到这些球状的特征。

.2.32分数布朗运动

上面描述的噪声纹理构成了我们的云纹理的基础。下一个
我们产生噪声的步骤是被称为分数布朗运动[17]，通常缩写为fBm，当应用于噪声纹理时，其结果看起来像烟雾或云。fBm噪声是通过将不同采样频率的噪声纹理分层在其上构造的。通过改变2的幂次频率，可以保持包裹效应。层数通常称为八度层数，图2.6显示了5
八度fBm版本的纹理。



(a)
(b)

图2.6： (a)这是具有5个八度频率的Perlin噪声。与图2.3相比，我们可以看到底层结构是相同的，但整体噪声纹理有更精细的细节。
(b)这是具有5个八度fBm的蜂窝噪声。这种质地有更明显的结构和更深的裂缝，类似于花椰菜的质地。

.32云形成

为了从噪声纹理中得到实际的云形状，我们进行了几个步骤来实现云的不同特征，这些特征是云的平坦底部和高耸的顶部是第1.4节中提到的对流的结果。

2.3.1高度分布

在我们的实现中，我们有一个高度分布函数，它控制着云的形状在三维纹理的底部更密集，而在顶部逐渐变薄。函数如下：

$$HD(h) = (1 - e^{-50 \cdot h}) \cdot e^{-4 \cdot h},$$

在那里，他的高度。这使得云有很高的可能性存在于纹理的底部附近，有效地给了云一个平坦的底部，以及保留了高耸的云的轻微机会。选择使用形式的$f(x) = (1-e^{-a \cdot x}) \cdot e^{-b \cdot x}$基于它提供了一种容易获得的调优结果的方法，通过改变a和b。通过追踪和误差发现了a = 50和b = 4，并分别决定了底部和顶部衰减的锐度。图2.7显示了应用于基于噪声的函数
三维纹理。

图2.7：高度分布函数均匀应用于整个三维纹理，对于纹理的每个点，纹理与函数的输出；纹理中点的高度坐标输入HD函数。这并不会破坏纹理包装的边缘。

.3.22高斯塔

由于高度分布函数应用均匀，因此不会出现明显的云塔。在我们的实现中，一个额外的高度功能允许在3D纹理中创建峰值或塔。我们选择将塔实现为多元正态分布峰。这为我们提供了一种简单的移动和修改峰值形状的方法。这种方法的唯一限制是，峰值可能不会被放置得太接近边缘，从而干扰纹理包装的连续性。

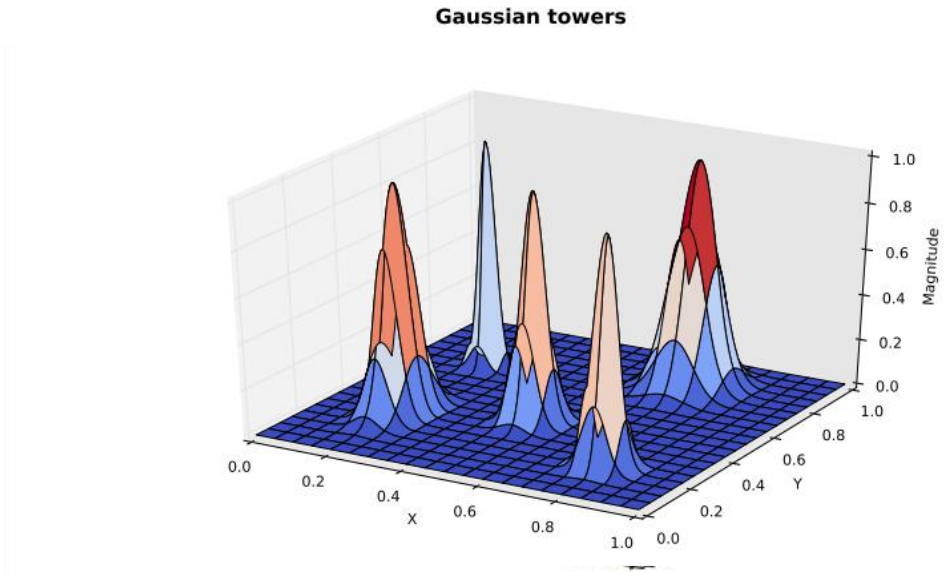


图2.8：这是高斯塔的一个示例。请注意，这个函数沿着边缘是零的，所以当
此函数作为一个偏移值添加到3D纹理，即tex-
结构包装保持完整。

.3.32 Finalisation

云形成的最后一步是使用一个阈值函数——一个简单的截止值。
这切断了3D纹理中“更薄”的部分，显示了更明显的云。图2。9显示了最终的三维纹理的横截面。

在我们的实现中使用的三维纹理在纹理的四个颜色通道中有不同频率的噪声。这意味着，通过一个纹理读取，我们可以从一个通道检索云的形状，以及更高频率的其他噪声函数。这允许我们应用更精细的细节，并通过混合基本云形状中每个通道的不同数量来轻松地调整最终结果。

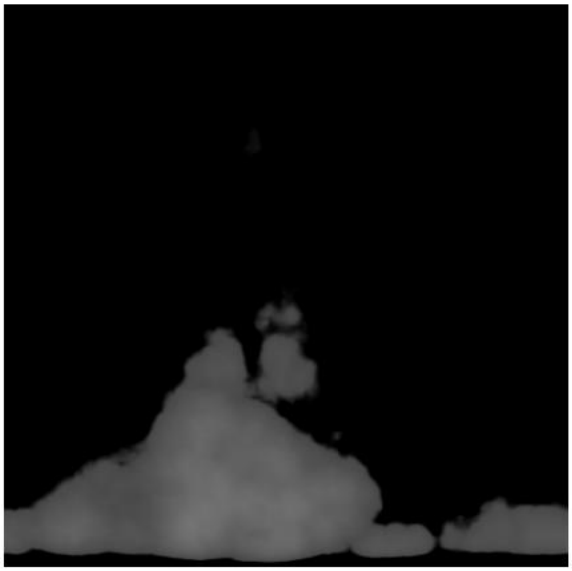


图2.9：这是应用阈值函数后的3D纹理的样子。

.42体积射线标记

到目前为止讨论的实现的细节是离线计算的（不是实时的），并保存到纹理中。射线行进是在GPU上实时完成的，因此实现必须考虑到性能命中。

.4.12.延迟着色和射线铸造

为了能够执行光线投射，将使用延迟的着色。延迟阴影意味着场景首先在没有云的情况下被渲染，然后，延迟到后期，云被渲染。除去云，整个场景通过渲染管道，并渲染为分配框架缓冲区的纹理。然后，通过渲染一个覆盖整个屏幕的原语来执行第二次渲染传递。全屏四分体。这允许我们在像素着色器中访问屏幕上的每个像素。要将光线投射到场景中，需要将屏幕坐标转换为世界空间坐标。这涉及到反转在第1节中讨论的所有转换。1.当光线穿过场景时，它在等距的点采样，以确定是否渲染云，如第2节所述。1.射线标记器第一次遇到云时，该样本点上的3D纹理值被存储为一个alpha值[1].后续的非零样本将被添加到这个值中。当alpha值达到1时，alpha值就会饱和，我们知道场景纹理中的任何颜色都不会显示出来。然后光线行进停止。设置了光线长度的上限，这样即使它没有遇到任何云，它最终也会停止。在光线探测器停止后，云的颜色（如下计算方法）使用采样的alpha值与预渲染的场景纹理混合。请参阅清单

[1]一个alpha值被用来描述像素的半透明性。

2.1，以更清楚地概述射线游行器。

```
1 vec4 cast_ ray（vec3起源，vec3 dir）{ 2
3 vec4值= vec4（0。0） ； 4
5浮动增量=1。0；
6浮动启动= gl_深度范围。靠近
7浮动端= 500。0；
8
9为（浮动t=开始；t<结束；t +=增量）{
10
计算新的样本点*/
12样本点=起源+ dir * t；13
14 /*停止已达到完全不透明度*/的光线
15如果(值。a    ==    1 .0)     {
16休息；
17                              }
18
19个值。一个+=云采样（采样点，增量）；
20个值。一个=钳夹(值。a ，    0 .0 ，    1 .0) ；
21
22 /*从阴影和散射的*/中计算云的颜色
23...
24个值。rgb += ...*阿尔法；25 }
26                  }
27返回值；28 }
```

清单2.1：这个列表显示了在GLSL中实现的射线游行器的高度压缩版本。cast_ray（）从原点向dir的方向投射一条射线。这里的步长被称为delta，并被设置为1个单位。射线探测器的上限设置为500个单位。在线23上放置第二个射线探测器以计算照明效果。

.4.22步长

射线游行器的性能影响在很大程度上取决于每个像素的样本数量（或步骤）。通过增加步数之间的距离，可以减少步数的数量。如果距离增加得太多，彩色带的形式就会开始出现。有一些方法可以减少条带效应[5]，但一个非常简单的方法是在行军的第一步添加一些小的随机长度。这扭曲了条带的结构，使它变得不那么可见。在图2中。我们已经构建了一些人工条带效应，来显示在射线行进开始时使用随机步长的结果是什么样子的。

自适应步长是我们在实现中测试的一种方法，它试图减少在射线游行器中所采取的步骤的数量。这意味着在经过一段距离后，我们允许步骤的长度增加。这意味着，当渲染遥远的云时，可以节省大量的计算能力。但是为了节省接近查看器的云的计算时间，接下来还将实现并讨论第二种方法。

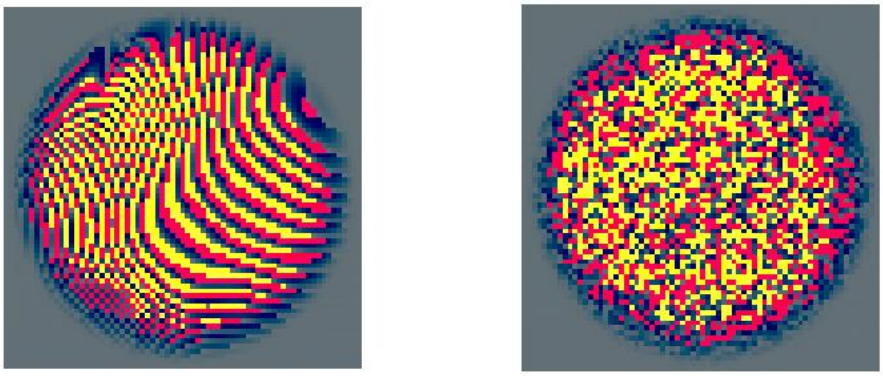


图2.10： (a)曲线是一个太长的结果
步长通常带不会这么严重；这是带的一个极端例子。(b)在第一步中应用一些随机步长后，条带问题就不那么明显了。该方法最适用于高频条带。如果在圆圈上只有几个条带，它们在施加噪声后仍然可见，尽管它们的边缘会不那么明确。

预处理云结构

我们想利用这样一个事实，即我们已经知道云在3D纹理中的确切位置。在程序开始时，在渲染开始之前，我们放置了一个云纹理的预处理步骤。在这一步中，构建了一个低分辨率的结构，使它完全包围云，并存储在另一个三维纹理中。图2。11显示截面
从这些三维纹理。

低分辨率结构被用作边界来决定应该使用小步长还是大步长。改变了射线探测器，以适应低分辨率的结构：

射线游行器从较大的步长开始。设置较大的步长，使其对应于低分辨率结构中的小于一个像素长度。

当射线探测器开始从低分辨率结构中采样非零位时，它会向后后退一步，更改为一个小的步长。最初的后退步骤是必要的，这样就不会遗漏“云位”。

射线游行器继续使用更小的步长，并对云纹理进行采样。每次小步长加成一个大步长时，对低分辨率结构进行采样，以确定它是否可以改变回大步长。

回到图2。我们可以看到，这个结构为实际的云增加了一个很大的边际。预处理步骤的实现是为云添加双填充，这样射线探测器就不能切割结构的角落，这将导致丑陋的人工制品。这是因为该结构的构造是为了使较大的步长是

图2.11： (a)这是通过的三维纹理
预处理阶段。(b)这就是低分辨率的结构。在这个图中，它被拉伸到与云纹理相同的大小，以证明它封装了原始纹理中的所有“云位”。

只是比一个结构单元的一侧稍微短一点。射线探测器将在正面接近时探测到云结构，但在观察者没有与云结构的网格模式相对齐的情况下，射线探测者可能会在细胞的角落走一步而没有检测到它。这就是为什么使用双填充，这意味着第二层的云结构包含了实际的云，即使最外层不包含任何要渲染的云。它只是为了确保我们在接近高分辨率的云纹理时，才试图对它进行采样。

2.5照明计算

在我们的实现中考虑了两种类型的光效应。第一个是光通过云时的衰减。这种效应可以用
比尔兰伯特定律

$$T(d) = e^{-m \cdot d},$$

其中T为透射光，m为材料因变量，d为光通过材料[15]的长度。

所考虑的第二个效应是光散射。.4.当来自太阳的光到达云中的水滴时，它根据第一节中讨论的Mie理论散射，计算所有这些光线如何实时散射无限时间是不可行的，所以散射被分成两部分。（还有其他方法来处理不同顺序的散射[3,4]，但这是我们决定在实现中这样做的方法。）其中一部分处理了适当的与角度相关的散射，如图1.7所示。这只适用于一次光散射。另一部分是云内多重散射的近似值。下面的函数是我们的近似值

云中的散射：

$$S(d) = 1 - e^{-c \cdot d},$$

其中S是散射光的强度，c是一个决定其速度的变量
散射效应在云中形成，d是光通过云的长度
云这个函数和比尔-兰伯特定律一起构成了我们的光函数
图2。12)。本文讨论了这种计算光的方法
游击游戏在签名图2015 [13]。

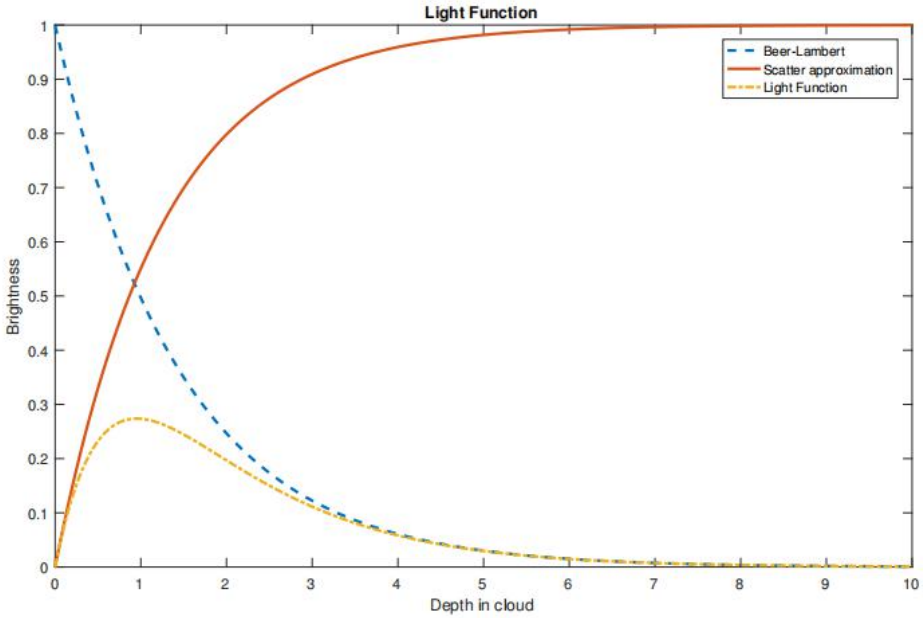


图2.12：图中显示了比尔-兰伯定律和我们的散点近似函数。将它们相乘，创造了我们的光功能。我们可以看到首先通过云的光线如何开始散射，但随着云减弱光线，亮度就会下降。

在光线行进的开始时，云被赋予深色的底色，根据它们对太阳的暴露程度，会增加更亮的颜色。如前所述，照明计算也是使用射线探测器来实现的。每次云射线探测器对云内的一个点进行采样时，用于光计算的射线探测器就会对云的纹理向太阳进行采样，以确定原始样本点在云中的嵌入程度有多深。这个深度被传递给光函数，该函数计算应该在暗基色中添加多少亮色。

2.6相功能

图1.7中的Mie散射相位函数可以用于单次光散射，方法是将PhilipLaven的MiePlot软件中的值保存到一个纹理中。这个纹理可以用作不同角度的查找表，这些角度是由相机看到的样本点和太阳之间的角度。

近似Mie散射相位函数的一种常用方法是使用更简单的亨耶-格林斯坦相位函数[4]。亨耶-格林斯坦相位函数是
表示为

$$\mathrm{HG}(\theta) = \frac{1 - g^2}{4\pi(1 + g^2 - 2g\cos\theta)^{3/2}}$$

其中HG是散射光的大小，对于一定的角度e，ge[一1,1]是决定散射浓度的参数；负g给出后向散射[8]。这个函数如图2所示。13.

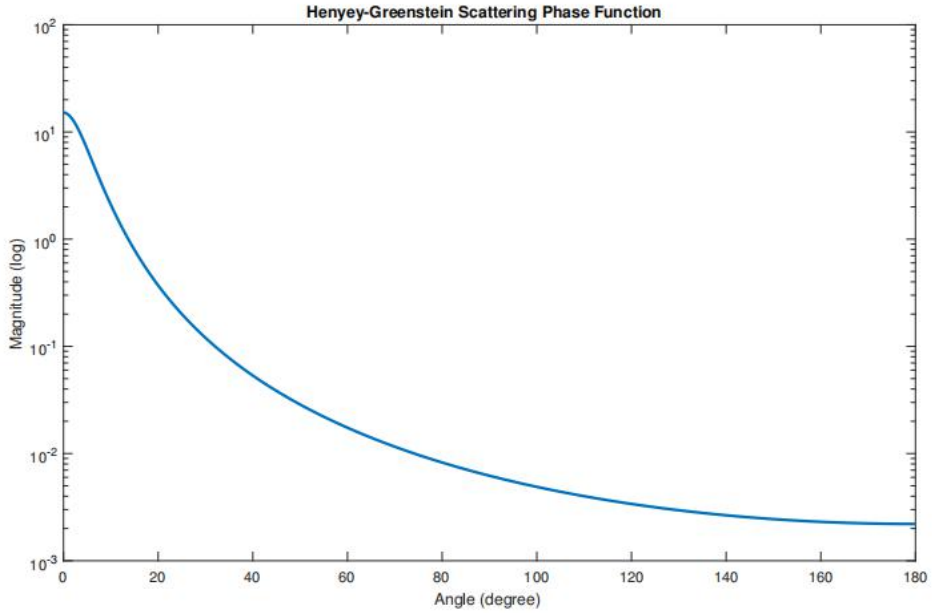


图2。13: 这是g = 0.9的亨尼-格林斯坦散射相位函数。这给出了一个高浓度的正向散射，但它缺少在140° 和180° 附近的峰值。

与更复杂的Mie相位函数可以直接在片段着色器中实现，而不是格林斯坦相位函数。这将保存在使用Mie相位函数时所涉及的一个纹理读取。在我们的实现中，我们尝试使用预先计算的Mie相位函数和片段着色器实现的亨耶-格林斯坦相位函数。

2.7高动态范围照明

通常，帧缓冲区中的颜色通道被存储为8位值，范围从0到
1.这意味着颜色通道只能假设256个不同的值。这适用于大多数应用程序，但云和太阳的性质为我们提供了屏幕上非常明亮的区域。这可能会导致屏幕的部分失去重要的颜色信息，因为颜色被四舍五入到最接近的256个状态。解决这个问题的一种方法是使用高动态范围（HDR）照明。不是使用8位值将每个颜色通道的纹理绑定到帧缓冲区，而是使用浮点值。在我们的实现中，我们为每个通道使用一个32位的浮点值。HDR照明还允许我们为框架缓冲区分配颜色在1以上的颜色值。在将最后的帧缓冲区呈现给以下对象之前
屏幕上，这些值需要转换回范围0到1。这是使用所谓的音调映射来完成的。这是所使用的音调映射（来自[1]）：

$$L_d(x,y) = \frac{L(x,y)}{1 + L(x,y)},$$

其中，L（x、y）是像素在（x、y）和L处的亮度d（x、y）是音调映射的值
在0到1的范围内。该音调映射压缩了高亮度值，同时保留了针对低亮度值[1]的更多信息。

HDR照明还能够实现一些额外的效果。光的盛开就是这样的一种效果。它允许明亮的区域渗透到邻近的区域，导致明亮区域周围一些更平滑的光过渡，并给明亮的区域发光的特征。这是通过提取高于某些阈值的明亮像素，并使用高斯模糊模糊这些像素来实现的。这些模糊的像素会被添加到色调映射之前的场景纹理中。

第三章

结果

.13硬件和软件

所有的结果都是在Windows 10机器上捕获的，其Intel Xeon E51620 v3运行在3.50 GHz和64 GB内存下。尽管CPU和RAM是高端的，但GPU是由Nvidia GeForce GTX 680卡提供的，在编写时可以被认为是中层卡。×结果以1280 720的分辨率被捕获。

OpenGL上下文由简单的直接媒体层（SDL）提供，而OpenGL扩展库牧马人（GLEW）用于提供OpenGL扩展。微软公司
Visual Studio 2013用于编译C-++代码。完整的实现可以在这里找到（光线探测器）：

- https://github .com/rikardolajos/clouds

以及此处（使用噪声生成纹理）：

- https://github .com/rikardolajos/noisegen.

为照相机构建了一个基准测试轨迹，以便在不同的实现之间进行公平的比较。在本章中，我们将比较不同算法沿着测试轨迹的帧时间，以更好地理解它们的性能命中。测试轨道的构造是为了通过场景中的不同情况(e。g.云下面，云上方，云内部，等等)这样我们就能看到哪种算法在什么条件下更可取。对于本章中所有显示帧时间的图，越少越好。

.23收集的数据

本节介绍了沿着测试轨迹收集的帧时间。图3。1和表3。1显示了使用预处理后的云结构的结果。在使用预处理的同时
云结构，我们可以看到帧时间的整体改善，除了射线探测器早期饱和的部分——当相机在云内。

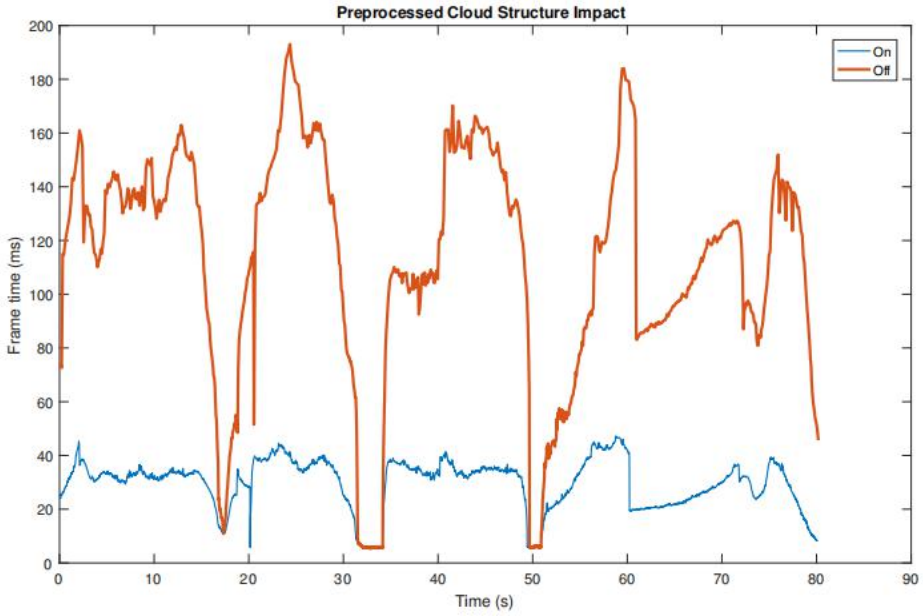


图3.1：显示使用预处理云结构与不使用它时的性能的图。图形重叠的部分是当相机穿过云的时候。

预处理云结构	平均帧时间	平均频率
向	23.728955 ms	42.142606 FPS
Off	56.155690 ms	17.807635 FPS

表3.1：预处理后的云结构平均性能快约32 ms。

在图3.2和图3中。3显示了使用自适应步长的结果。平均帧时间和帧渲染频率见表3.2。一般来说，自适应步长对帧时间没有明显的影响。它只影响相机放置在远离云的部分（参见图3.2中的大约60秒）。对于这些
测量大步长设置为20个单位，而自适应步长从1个单位开始，对于超过100个单位的样本，每个单位增加1个单位。

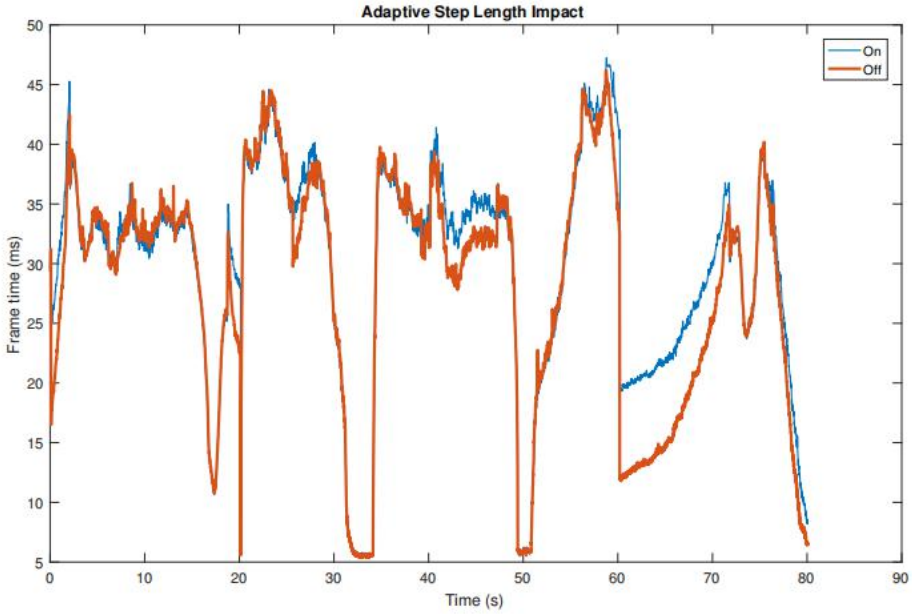


图3.2：该图显示了对于较小的步长，与静态步长相比的自适应步长。大约在60年代左右，照相机远离云层，图表显示了非常不同的结果。

自适应步长	平均帧时间	平均频率
向	21.761936 ms	45.951795 FPS
Off	23.728955 ms	42.142606 FPS

表3.2：当使用自适应步长时，平均节省的帧时间约为2 ms。

图3。3：此图显示了在使用自适应步长时云的放大。远离相机，在那里自适应步骤被允许变得很大，一些人工制品开始形成。

图3.4和表3。3表示整体照明计算对渲染的影响，i。e.运行光线探测仪的影响。我们可以看到，光的计算会影响帧时间，而不管相机的位置如何。

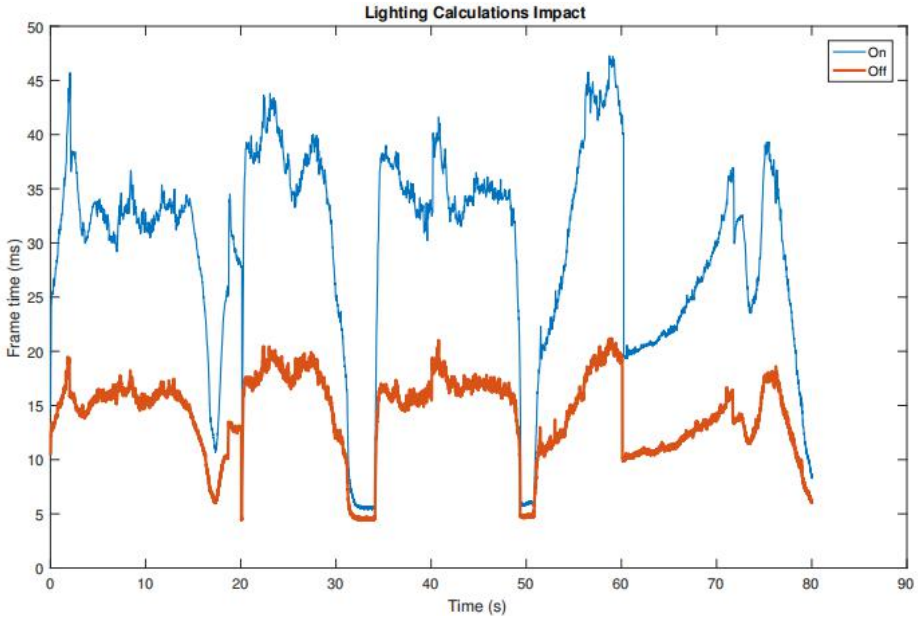


图3.4：绿色线表示未使用灯光计算渲染的云。蓝线表示执行光计算时的帧时间。

光的计算	平均帧时间	平均频率
向	23.764996 ms	42.078694 FPS
Off	.55380312 ms	79.657134 FPS

表3。3：用于照明计算的光线探测器在测试期间平均约需要11 ms。

从预先计算的纹理中读取相位函数与在片段着色器中实时计算相位函数的影响如图3.5和表3.4所示。我们可以看到，纹理读取比计算片段着色器中的相位函数需要更长的时间来执行。

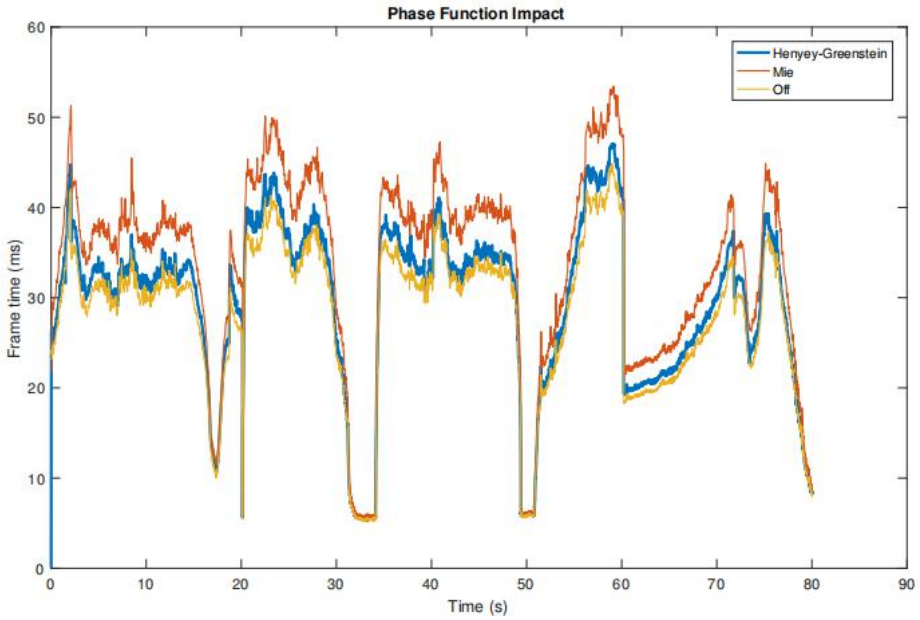


图3.5：Henyey-格林斯坦相位函数（在片段着色器上计算）与Mie相位函数（从纹理中读取）的比较。第三个图显示了不使用相位函数时的帧时间。

相函数	平均帧时间	平均频率
亨耶伊格林斯坦	23.643466 ms	42.294982 FPS
米	25.999220 ms	38.462692 FPS
Off	22.585036 ms	44.277105 FPS

表3.4：计算片段着色器（亨耶-格林斯坦）上的相位函数大约需要1 ms，而从纹理（Mie）中读取则需要3 ms多一点。

3.3渲染图像

本节介绍了最终实现的一些渲染图像（图3.6到3.9)。这些图像是利用预处理后的云结构、静态步长和亨耶-格林斯坦相位函数构建的。噪声纹理完全基于蜂窝噪声。图3中的图像。10和3。11显示了使用亨耶-格林斯坦与使用Mie相位函数之间的视觉差异。

图3.6：在这张图片中，相机被放置在地面上，仰望太阳。

图3.7：在此图像中，太阳已向左关闭。照相机被放置在云中和阴影之中，根据比尔-兰伯特定律，是清晰可见。

图3.8：放置照相机，以便我们可以看到云的平面底部。太阳就在照相机的后面。图像中心（高耸的云前面）的云较暗的轮廓是光函数“伪造的”散射效应的结果。

图3.9：这里的相机再次被放置在地上，仰望太阳。相位函数的影响可以在这里看作是覆盖太阳的云的部分几乎在发光。

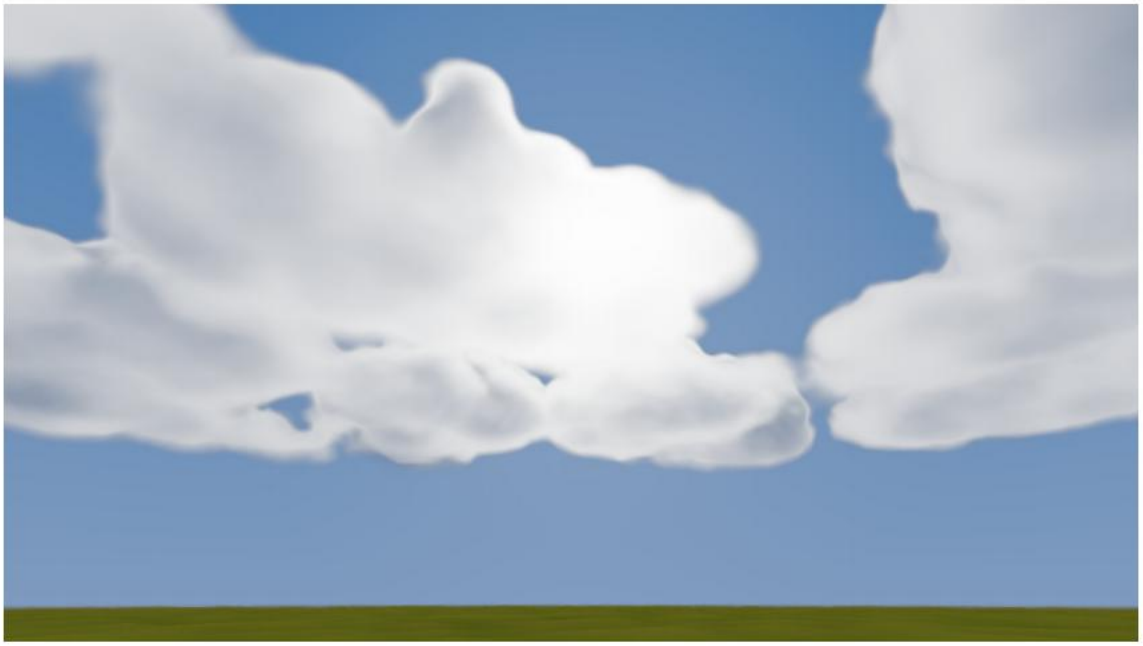


图3.10：这是一个显示亨尼-格林斯坦相位函数的图像。

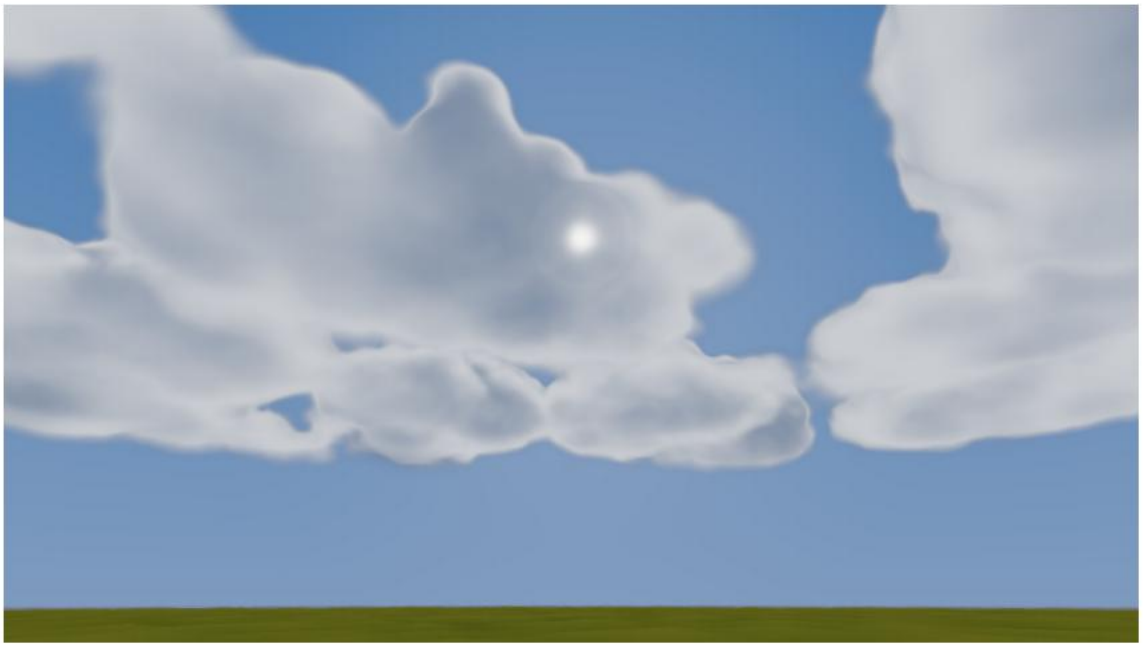


图3.11：这是一个显示Mie相位函数的图像。

第4章

讨论

本文的两个主要部分是云的形成和实时渲染云，而不丢失太多的视觉细节。云的形成是
很难客观地评估，而云渲染有第3节中的帧时间。2作为混凝土评价。

4.1云形成

在我们的实现中，我们测试了Perlin噪声和蜂窝噪声。我们测试使用普通Perlin噪声，普通细胞噪声和两者的混合作为我们的3D纹理的基础。最终结果中的差异并不是很明显，并且很难在结果章节中呈现出来。第3.3节中呈现的图像完全基于蜂窝噪声，因为蜂窝噪声为云提供了更圆的结构。这部分是因为云并不是完全随机形成的，就像基于随机分配的梯度向量的Perlin噪声一样，而是由于对流而有一个地下结构。

图3中的云彩。6到3.9缺乏一些更高分辨率的细节。我们的实现利用了更高频率的噪声，放置在3D纹理的不同颜色的通道中，这在图3.7的阴影部分中可见。但是整体表面并没有如图1所示的真实云的更精细的细节。1.

获得更精细细节的一种方法是对3D纹理使用更高的分辨率；我们对纹理使用128 128 128的分辨率。××但在提出决议时，需要考虑以下几点：

如果使用Perlin噪声，如果我们想要分辨率超过256 256 256，Perlin的噪声实现必须被改变，因为它本身不支持这一点，××

三维纹理的构造需要0（n^3），所以只需分辨率就需要加倍
建筑时间延长了8倍，

如果我们使用云结构的预处理，这也需要O（n^3）.

如果3D纹理的构建需要更长的时间，这可能不会造成太大的影响，因为3D纹理被保存并从磁盘读取，但低分辨率结构的预处理和构建在程序开始和每次云变化。这将导致最终产品中的加载时间大大增加。

.24云渲染

降低在射线游行器中所采取的步骤的数量对于降低帧时间至关重要。平均节省32 ms，表明预处理的云结构在测试跟踪用例中运行良好。这种方法在减少步骤数方面的一个很大的优点是，它不会影响渲染的结果，因为在云中总是使用较小的步骤。使用这种低分辨率纹理的缺点是，预处理需要时间，而且我们必须知道云将在哪里被渲染。如果我们想在片段着色器中生成云，如果没有重建，低分辨率的结构就无法工作。生成这样的云将允许动画可以很容易地实现，因为云将不会基于静态的3D纹理。另一种实现云的动画的方法是
使用4D纹理。这可能使我们有可能使用4D纹理，并可能节省大量的渲染时间。

自适应步长是减少所采取的步数的另一种方法。这种实现更像是一种平衡行为，因为在遥远的云中开始出现严重的条带和伪影之前，帧时间（约2 ms）不会出现显著的改进（见图3.3）。此外，所获得的性能还取决于相机相对于云团的放置位置。

轻的计算需要执行大量的时间。它所采用的11个ms在最终产品中可能很有价值，因为分配的33个ms不能全部花在渲染场景中的云上。在这种情况下，用于光计算的光线探测仪可以完全废弃，取而代之的是一些更“伪造”和计算成本更低的照明计算。

有趣的是，从纹理中读取的相位函数比动态计算的相位函数花费的渲染时间更长；Mie相位函数比亨耶-格林斯坦相位函数慢三倍多。这是内存带宽的结果。从纹理中读取会导致使用内存带宽，而且可能相当昂贵。仅使用这种与角度相关的散射的结果给出了一个可接受的视觉结果，而图3.9中显示的相位函数的效果可以与图1.2中的照片相比较。这两个图都显示了直接覆盖太阳的云部分是如何明亮的，而周围的云有明亮的轮廓，核心较暗。

对于高阶散射，当一阶散射也有贡献时，在光函数中使用近似的结果看起来可信（如图3.9），但在图3.8中，与太阳的角度约为180°，相位函数变得可以忽略不计。在这张图片中，云的边缘较暗，给了云一些明显的轮廓，但是核心——高阶光散射的近似位置——有一个严重饱和的白色外观，大部分细节都消失了。

第5章

结论

我们的实现表明，当使用射线游行器时，可以节省大量的计算时间，主要是通过假定与要渲染的场景有关的情况。通过假设将被光线传播的媒体的位置，可以进行最大规模的优化。

用于照明计算的第二个光线探测器对于现实主义的阴影是必要的，但是它确实对性能有很大的影响，如果时间允许，其他计算光线的方法也可以被测试。

基于噪声的云纹理可以很好地形成云，既是令人满意的视觉结果，还是非复杂的实现，允许部分要么在着色器上实现，要么预先计算并保存到纹理中。这有助于调整内存带宽使用和计算时间之间的最佳平衡。对于云的形成，我们只是触及了可能的方法的表面。我们通过从噪声纹理中“雕刻”出云来生成3D纹理，但是3D纹理可以通过使用流体求解器来生成，或者通过结合更小的云块（瓷砖）和使用Wang瓷砖来构建云。

相位函数的实现也提出了一个有趣的问题：用真实的相位函数读取纹理是否值得产生三倍的性能影响，还是我们对它在片段着色器中近似和计算感到满意？

|的|演示文稿2016-06-03

六聚乙二醇 实时的 渲染 的 体积的 云 射线 马丁 噪音 基础 3D 纹理）
学生 里卡德 奥拉约斯
车把 迈克尔 多格特绰号 （LTH）
检验员 弗莱维厄斯 格鲁伊安 （LTH）

Realtidsrendering av volymetriska moln

民粹主义 ¨ 击败奥拉霍斯

工艺工艺，拉克纳特i处理器-
我的基因组是真实的。男人是我的朋友
我的3D应用程序在巴林上丢失
我的名字。

我的意思，我是我的代表。在我的爱里，我的幻想，我的幻想。在你的生活中，放弃我的生活，免除我的生活。男人们打开世界的朋友，我的朋友，我的朋友。我的名字是这样的（2d文本处理器），我的名字是这样存在的。我的朋友们都是我的过路人。为了我的名字，我的名字是3d纹理器。

这是我的朋友
德萨3d技术师，我是我的朋友。3d的文本结构

这是我的朋友。请告诉你现在的生活，直到你现在的生活，直到你现在的生活，请告诉我现在的生活。我的名字是我的名字是我的名字。我们的名字是自然的，我的名字是，我的名字是我的名字。卡拉斯雷在游行。3d分析是我的语言，我是我的语言。我的工作是我的工作。