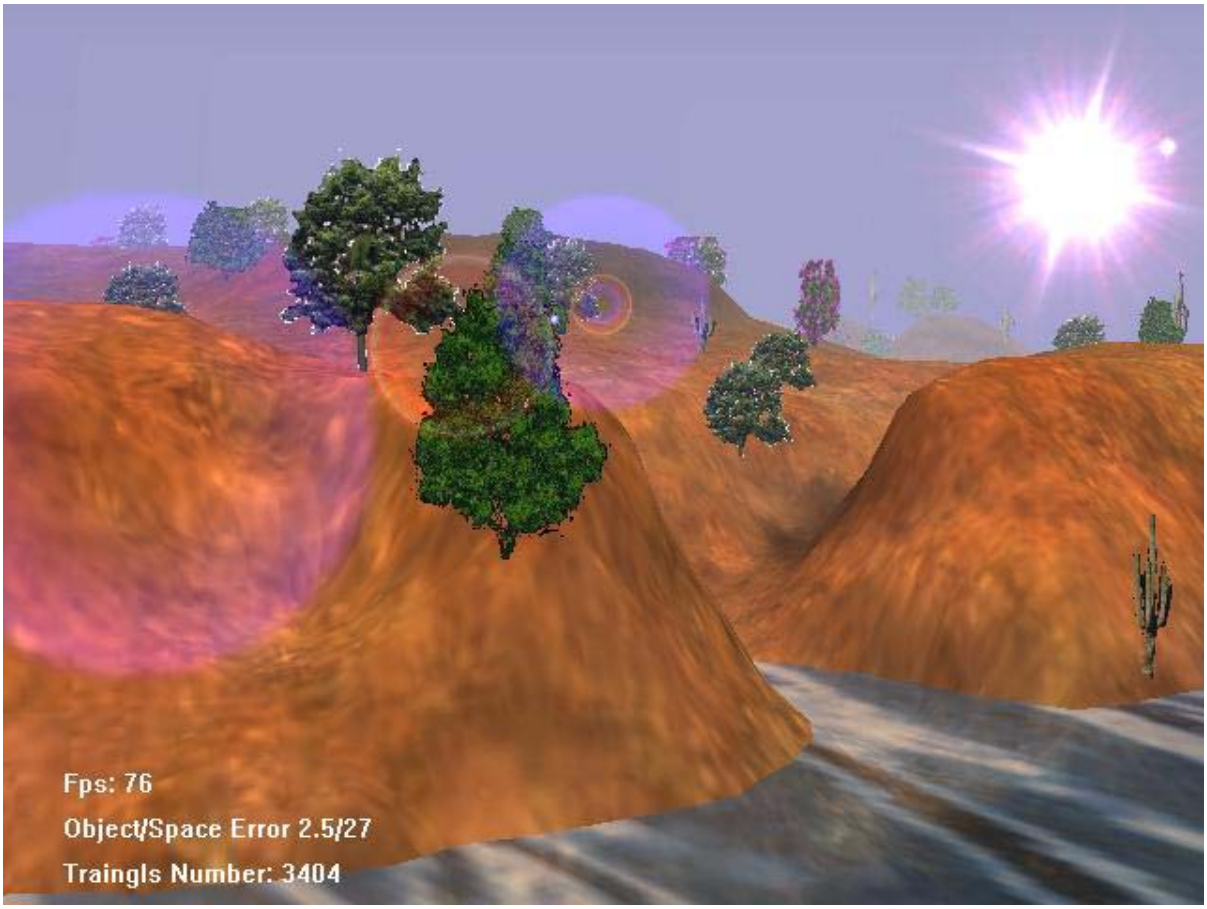
基于 LOD 的大规模真实感室外场景实时渲染技术的初步研究

2003　年　4 月- 9 月
最近更新 9 月 24 日
Demo 更新　9 月 5 日



关键字与术语：

LOD , (level of detail)、**ROAM** (Real-time optimize adaptive mesh).、**公告板、多遍纹理映射、运动模糊、 Lens Flare 、天空体、亮度图(**Light Map**)、四叉树 、 Perlin Noise、Diamond Fractal.、Voxel 、二叉剖分空间(BSP)、OpenGL、DirectX。Gama Control.**

Abstract:

The large scale terrain rendering is a hot issue in the Computer Graphics research field. It play an important role in GIS (Geographic Information System). Flight simulator and video game. The main two problem of the large scale terrain rendering are how to hold the terrain data and how to reduce the large amount triangles. For video game solution. I use a quad-tree based LOD algorithm to reduce the triangles count. I also introduce some other technology such as motion blur. to improve the realism of the scene without reducing too much FPS.

I implement an outdoor game's render engine in this paper. Not like other systems. The video game need to running on the PCs, It need a very high interactive FPS. Generally 30 FPS is the best in a video game. In this paper's LOD algorithm. The quad-tree is stored in a two dimension array instead of in a linkage structure. The triangles (the quad tree nodes) not in the view frustum is culled before send them to the render API.。.

The key technology in this paper is LOD algorithm. I use the view-dependent and terrain's roughness to determine the terrain detail. Before render the terrain. I refinement the terrain mesh. Only the vertices are needed will be rendered. So the mesh grid is irregular .The far from the viewer, the less detail. The stronger roughness of the terrain, the more detail. Like other quad-tree based LOD algorithm. I subdivide a terrain node until the desire detail. The subdividing depend on the node level, the terrain's roughness and the distance between the viewer and the node. The roughness factor is pre-calculated to improve speed.

This paper's major purpose is the LOD algorithm , So some problem such as Geomorphing, are still exit. Another important problem is storage layout, I only try a little to settle this problem, so the terrain data storage system is not so effective.

摘要：

大规模的地形渲染技术一直是图形学里的热点问题之一。它在 GIS、飞行模拟器、视频游戏里有重要的作用。大规模地形渲染的两个主要问题是地形数据存储问题和三角形数目问题。针对视频游戏，本文使用了一种基于四叉树的 LOD 算法来解决大规模地形渲染中的三角形数目问题。并且使用了其它一些技术，在保证渲染速度的前提条件下，有效的提高了场景的视觉真实程度。如运动模糊。

本文基本上实现了一个室外游戏的渲染引擎。不同于其它的应用系统，视频游戏一般要求在 PC 机上运行，而且要求比较高的交互性帧率。一般认为 30 FPS 是比较合适的速度，因此速度是第一前提。在本文的 LOD 算法中，四叉树信息被保存在一个二维数组中，而不是传统的链式结构。大部分不在视体内的三角形（四叉树节点）在送入渲染 API 之前就被切除掉。同时本文的算法只用了一遍四叉树遍历，从而大大提高了渲染速度。

本文的关键是 LOD 算法，我使用了视点相关以及和地形本身起伏程度相关的技术来决定地形应有的细节程度。在每次渲染前，我们都动态的更新地形网格，只渲染我们需要的节点。因此地形网格是不规则的。离观察者越远，细节越少，地面越粗糙；离观察者越近，细节越多，地面越细腻。和其它的基于四叉树的 LOD 算法一样，我们递归的分割每一个节点，直到到达需要的细节程度。这种分割依赖于节点的大小，节点内地形的起伏程度，以及观察者离节点的距离。地形的起伏程度事先被计算出来以提高速度。

本文主要的目的是实现 LOD 算法，所以其它的一些问题如几何形变依旧存在。另外的一个主要问题：数据存储布局也不是十分的有效，在这个问题上我只是做了一些简单的尝试。

目　　录

第一部分　简 介

引　　言

室外场景的实时渲染技术是游戏编程世界中的热点技术. 同时它在其它领域也有着同样重要的作用。如 GIS 系统，飞行模拟系统，VR 系统以及数字地球技术等都离不开室外场景的实时渲染技术。一个优秀的室外场景实时渲染技术在保证实时性以外还能创造出非常逼真的、有说服力的虚拟自然环境。如 Nova Logic 公司的著名的 3D 射击游戏 Delta Force 系列，它除了能模拟出各种如雪地、草地、沙漠等地形以外，还能模拟出各种树木，杂草，以及各种天气效果。

室外场景的实时渲染有许多技术上的难点,在以下的章节里我们将作详细的介绍以及针对一些主要问题的解决方案。

本文的主要内容分两个部分：一、大规模地形的渲染。二、如何提高场景的真实性。

3D 场景的渲染离不开 3D API。目前流行的 3D API 有两种，SGI 公司的 OpenGL 和微软公司的 Direct3D。两种 API 各自有自己的优点，均能很好的使用硬件加速功能。但是 OpenGL 是一种开放性的标准，有更好的移植性能，它能在运行 Linux 和 FreeBSD 的 PC 下甚至还可以使用硬件加速（nVidia 、ATI 公司专门为 Linux /FreeBSD 推出了驱动程序）。因此在本文里，我使用了 OpenGL。不过如果熟悉原理，其实也大同小异，D3D 到 8.0 以后做的非常的像 OpenGL。

第一章：大规模室外场景渲染技术简介

一．室外 Vs. 室内

下面我们把室内场景和室外场景做一个对比,来看看室外场景的实时渲染的主要难点。

目前最成功的商业室内游戏引擎有 Quake /DOOM 系列、Unreal 系列引擎。他们都基于 BSP 技术的。通过 BSP 技术，再加上 PVS , Portal 等技术可以大量减少场景的复杂程度，通过 Portal 技术甚至可以把一个室内场景和一个室外场景连接起来，关于室内引擎的渲染的进一步讨论已经超出了本文的范围。

我们知道，当我们站在一个房间里的时候，我们能欣赏到的景物不过是这个房间的摆设以及透过这个房间的门和窗能看到的景物而已。你只

能在墙壁的约束范围内走动。换句话说，我们处在一个有限的空间内。我们有足够的理由阻止人们穿过墙进入墙外的世界，也有足够的理由把视野约束在高墙以内。

图（1-1）

一个典型的室内场景，使用 id Software 的 Quake III 地图文件,用 www.GameToturials.net 的程序渲染

但是，这些约束在一个室外的场景中都是不可能的。在一个飞行模拟器中，理论上你可以驾驶飞机朝任何一个无限远的飞去。因为事实也是如此，如果你愿意的话，你可以驾驶飞机绕着地球飞行而不用担心有墙来阻止你的前进。换句话说，一个室外场景的理想大小是无限的大！除了场景的大小以外，同时视野也是无限的。如果你站在高处，你可以俯视任何比你的底的地方，也就是说你有几乎无限的视野。

图（1-2）

典型室外场景，图片来自北航王正盛的 Demo :**Nature Wing2.0** 截图

我们知道,无限的大的场景需要无限的场景数据。但是这是不可能的，

我们只能希望场景越大越好，室外场景的主要部分是地形的渲染，地形数据的多少决定了场景的大小。所以如何保存这些地形数据成了首要的问题。(但是在现在存储器成本迅速下降的今天，这个问题已经变的不是十分的突出)。其次是无限的视野问题，无限的视野就表示渲染无限的图元（**图元即是 3DAPI 支持的简单的几何图形，详见 OpenGL/Direct3D SDK**)，这也是不可能。图元的数量是以场景大小的平方的速度增长的。光考虑地形数据，一个 2048X2048 的地形，如果不进行减低细节程度和裁剪的话，它将要渲染 8M 的三角形，这样的三角形量在 PC 级别上目前还是远不能实现交互式帧率的。所以，如何减少要渲染地形时候的图元数目成了室外场景实时渲染的关键问题。

其他的情况还有如野外的地表衍生物：树木、杂草、地貌等。同时天气效果，如下雨、下雪，刮风和闪电等.这些东西在一个室内的环境下基本上是不需要考虑的。而且模拟这些效果都需要很高的代价，有些甚至根本就没有办法模拟。

二．Voxel Vs LOD

综上，我们知道，室外场景实时性渲染的关键是地形的渲染。我们需要一种技术来降低地形渲染的开销。

目前的地形渲染技术主要有两种 Voxel 和 LOD，下面我来做个简单的介绍。

Voxel 也就是 Volumetric Pixel。也就是所谓的“体素”，它是相对于像素来说的，如果说像素是一个二维的矩形的话，那么 Voxel 就是一个三维的立方体。它的原理是比较简单的。James Sharman 自称他在 1995 年时就想出了这种方法。前面的提到的 Delta Force 游戏就是使用了 Voxel 技术。关于 Voxel 的细节技术不是本文的重点，我不准备做深入的介绍。Voxel 有一个天生的优点就是渲染的时候它和场景的大小没有关系，而且绝对不会渲染多余的东西（自带裁剪功能)。它的复杂度只和我们需要的视野，以及分辨率有关。而且可以在不使用硬件加速的情况下达到比较理想的速度（Delta Force I 就没有使用硬件加速)，生成的图象也比较的细腻。它的缺点就是不够的灵活。

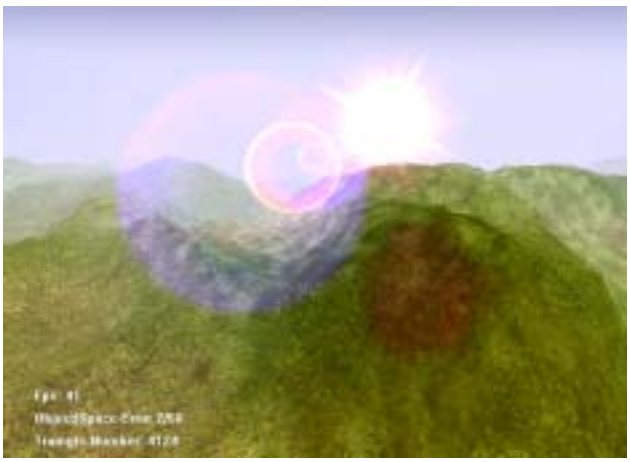
LOD 也就是层次细节（Level of Detail）的简称，不同于 Voxel 技术，它是一种使用多边形的，真正的 3D 渲染技术。它根据一定的规则来简化物体的细节，我们可以根据需要来选择不同细节程度的物体表达方式。如离观察者近的选择较高的细节程度、反之选择较低的细节程度。用在地形渲染中，有时我们也称它为多分辨率地形（Muti-resolution

terrain）渲染技术。

图（1-3）

基于 Voxel 的渲染场景，图片来自中国游戏开发者网络，

陈鹏《自己动手编 Voxel 3D 引擎》

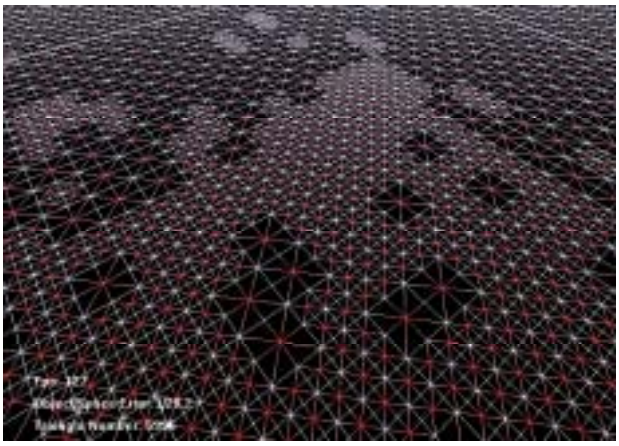


图（1-4）

基于 LOD 的渲染结果，图片来自本文的 Demo： **Sim-Nature.**

LOD 算法对场景的处理比较复杂，但是它让我们可以足够自由的去控制我们的场景渲染，更加方便的使用显卡的硬件加速功能。而且可以很容易的在场景中组合其他的物体。如树木，太阳以及粒子系统等，天空如它可以方便的让观察者以任意的角度去观察场景，我们只要让摄影机旋转一定的角度就可以了。但是这在 Voxel 中是比较困难的，例如 Voxel 在处理非水平的视线的时候就非常的麻烦。

LOD 技术是本文将要研究和实现的地形渲染技术。



图（1-5）

经过 LOD 处理的地形网格，有不同的细节。图片来自本文的 Demo： **Sim-Nature.**

三．动态地形 Vs 静态地形

地形的渲染通常分为动态和静态的两种。

静态的地形的细节可以是均匀的，也可以是不均匀的。但是细节通常在事先就计算好了，不均匀细节的静态地形也有许多的优点：如平原的地貌可以使用较低的细节，而起伏频繁的地方使用较高的细节等级。更为直观的一个例子是赛车一类的游戏。在这种游戏种，赛车可以到达的地方是有一定限制的。我们可以在离赛道近的地方建立起比较高的细节等级，而在较远的地方使用较少的细节。用这种方式也可以建立起不规则的地形（地形的形状不是一个矩形区域）。比如说，它可以沿着赛道的方向建立起一个地形模型，这样可以节省大量的空间。

动态的地形是视点相关的。它是本文将要研究和实现的方法。随着视点的移动，地形网格将被更新。相对于静态地形来说这是一种更为先进的算法。这种方式建立起来的场景更加符合人的视觉特性：即看到的细节是变化的。动态地形网格的建立和更新要耗费额外的时间，但是这种开销是值得的，我们所有要做的就是在精度和速度之间选择一个合适的平衡点。动态地形网格的建立是比较复杂的，它需要注意很多东西：如何决定细节，如何避免裂缝是两个主要的问题。同时它还应该把不可见的地形部分切除，几何形变（随着细节改变，地形表面的呼吸现象）也应该被考虑到。这些问题都在以后的章节被详细的讨论到。

四：其他

我们说过，在一个室外场景的渲染中。除地形以外其他的一些元素也是十分的重要的，这些元素可以提高场景的真实性。本文将实现其中的一部分元素，这些包括：树木、地面细节，太阳、天空以及运动模糊等

效果。他们将在本文第三部分集中讨论。

第二部分：基于 LOD 算法的地形简化

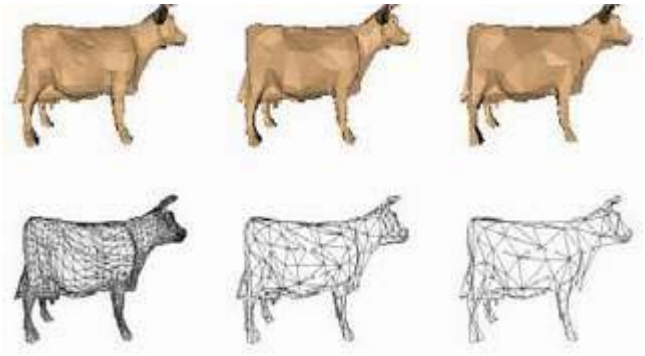
引　言

地形渲染是一个室外渲染引擎的核心部分。而实现一个大规模的地形渲染系统的关键是如何简化地形，抛弃不必要的渲染动作（如渲染看不见的三角形和不必要的细节）来加快渲染速度。动态 LOD 技术无疑是一个强有力的解决方案。

第二章：LOD 简介

当我们要生具有相当真实感的场景的时候，由于场景本身的复杂性，要实现实时性往往是不太可能的。我们必须从场景的本身的几何特性入手，通过适当的方法来简化场景的复杂性。层次细节（Levels of Details ）技术就是在这样的情况下提出来的。

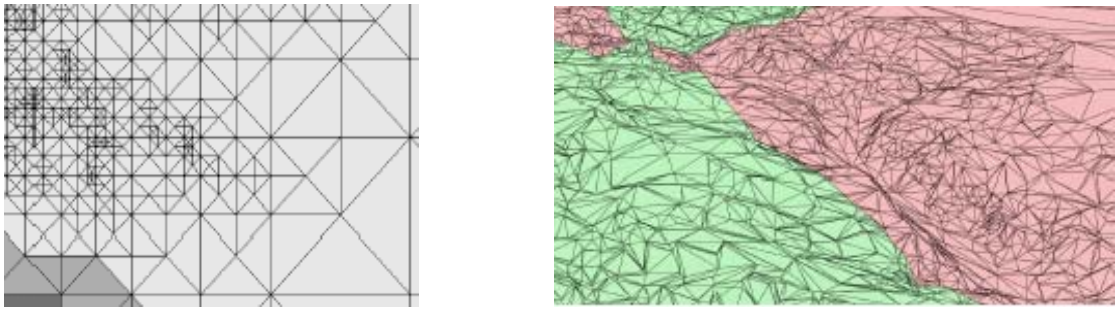
层次细节（LoD）技术是一种符合人的视觉特性的技术。我们知道，当场景中的物体离观察者很远的时候，它们经过观察、投影变换后在屏幕上往往只是几个像素甚至是一个象素。我们完全没有必要为这样的物体去绘制它的全部细节，我们可以适当的合并一些三角形而不损失画面的视觉效果。对于一般的应用，我们通常会为同一个物体建立几个不同细节层度的模型，如下图的牛的模型，最左边的有最高的细节层度，而最右边的则经过了相当的简化。这样的技术应用在地形渲染中，我们也称之为多分辨率地形（Multi-Resolution Terrain）。



（图 2.1）牛的层次细节模型，图片来自清华大学远程教育网

这些不同细节层度的模型可以是在程序运行前建立的。也可以是在运行时刻计算生成的。我们可以从一个全细节的模型出发，通过一系列简化操作生成低细节层度的模型，简化操作可以分成三种（见[参考文献 31]）：顶点删除，边压缩和面片收缩技术。通过这样处理后，我们可以在特定的场合下选择合适的模型，而不必每次都选用全细节的模型，这样可以大大的降低场景三角形数量。

地形作为一种特殊的几何物体，我们在运用 LOD 法则的时候有一些特殊的技巧。因为地形通常是一个规则的矩形网格。其简化模式可以有两种：规则的简化和非规则的简化，规则的简化通常是对这个矩形网格采用自顶向下(Up-to-Down)、分而治之的策略，典型的有四叉树和二叉树，它们从场景的最低细节层度开始，按需要不断的提高细节。非规则的简化通常是采用自底向上（Down-to-Up）的方法来处理的。它的实现则通常比较少。



(图 2.2)规则的简化（左边）和非规则的简化方式（右边）。图片来自[参考文献 2，12]

实现 LOD 算法时，除了如何对几何物体进行简化以外，还有一个很重要的问题就是如何决定是否对一个物体进行简化，或者说在某个时刻该如何决定使用哪个层次细节度的模型来表示物体。我们需要建立一个评价系统，由这个评价系统来决定要对物体简化到何种层度。这种评价系统通常是视点相关的，离视点远的物体通常只需要较少的细节，反之则需要比较多的细节。除此之外，物体本身的特性也必须考虑在内。比如说，一个平坦的表面只需要很少的三角形就能较好表现出来。而一个凹凸不平的表面是理所当然的需要更多的三角形去描绘的。

用LOD 算法渲染地形的时候，还有一个很重要的问题就是几何变形（Geomorphing）问题，由于对一些细节的丢弃，随着视点的移动，远处原来没有的细节很可能会突然出现，这种现象也称为“ 跳出 ”(“Pop”)。我们必须消除这种现象，或者至少要把它控制在可以接受的范围以内。

由上可知，LOD算法其实并不很复杂，本文认为其关键处可概括如下：

一．　**数据的存储布局。**

数据在内存中的布局必须要方便算法的实现，同时最好还要降低操作系统缺页中断的次数，也就是降低内外存之间的数据交换的次数。

二．　**如何在生成连续的LOD化的地形网格**

在地形LOD化过程中，要让两个由不同层度的细节的区域之间能平滑的过度。

三．　**节点评价系统。**

这个系统必须要使生成的网格能尽量的减少几何形变，尽量的使画面质量能接近全分辨率时候的地形。同时还要保证实时性。

第三章 相关研究

在过去的几年中,已经由相当的数量的实用的算法被开发出来。Bryan Turner 在他的论文**[参考文献 32]**中提到，LOD 地形法则可以由三篇优秀的论文来概括,它们为**[参考文献 12 ,4 和 7]**,在**[参考文献 12]**中,Hoppe 描述了一个 Progressive Mesh 的模型，它是使用自底向上的模式。**[参考文献 4]**作者是 Lindstrom，他则使用了一种基于四叉树的数据结构，他用四叉树递归的把一个地形分割成一个一个小块（tessellates）并建立一个近似的高度图。**[参考文献 7]**的作者是 Duchaineau，他描述了一个基于二元三角树结构的法则 ROAM（实时优化自适应网格)。这里每一个小片（Patch）都是一个单独的等腰直角三角形，从它的顶点到对面斜边的中点分割三角形为两个新的等腰直角三角形，分割是递归进行的可以被子三角形重复直到达到希望的细节等级。后两篇论文采用的都是规则的简化的模式，并采用分而治之的策略。而 Hoppe 采用的则是一种不规则的简化模式，它可以往任何一个三角形里增加细节，也可以删除任何一个顶点和边。

Hoppe 的法则使用比较少，很难在他以外的文章以外地方见到，Lindstrom 和 Duchaineau 的方法则不同，它们分别代表了当前的两大主流法则：基于四叉树的 LOD 地形分割和基于二叉树的 LOD 地形分割。

以上三篇文章是相当出色和精彩的。但是有一定的难度和复杂。本文更多的则是采用的是**[参考文献 2 和 13]**中的技术。两者都采用了四叉树的思想，这基本上同**[参考文献 4]**，但是更加的简单和快速，同时**[参考文献 2]**提供的顶点评价系统非常的快速。遗憾的是两者都没有建立完善

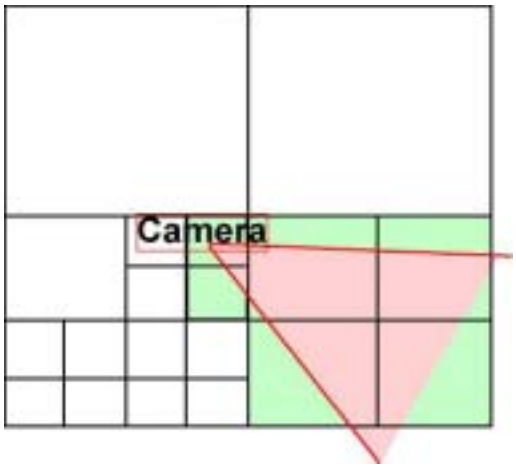
的内存数据布局来解决来地形数据的存储问题。

作为游戏开发领域的热点问题，自然有很多的 LOD 算法实现方案是源于游戏开发人员的，如上的**[参考文献 13]**就是出自 2000 年的 GDC。同时也已经有相当一部分游戏成功的采用了各种不同的 LOD 算法，如 Tread Marks ，Myth ,Soul Ride 等。**[参考文献 8]**的作者 Thatcher Ulrich 在他的文章里以 Soul Ride 游戏开发者的身份描述了应用于 Soul Ride 的基于四叉树的算法（详见 www. Gamasutra.com）。不同于学院派的算法，游戏开发者的算法通常更加的简洁和快速。

第四章 LOD 算法

一. 基本思想

在提出基本的算法之前，为了简单起见，本文必须对要渲染的地形做如下的规定：地形必须是一正方形区域。而且大小必须是 $(2^n+1)\times(2^n+1)$. 同时采样间隔必须均匀。



(图 4.1)一个地形的四叉树表示，左图中每一个正方形为四叉树的一个节点，粉红色的为观察者能看到的区域。

如图 4.1 所示，我们采用四叉树的概念来描述一个多分辨率地形，图中的每一个正方形为四叉树的一个节点，每个节点保存了一定区域的信息，包括：中心点的高度，从整个完整的地形出发，我们递归的把地形不断的分割（Sub-divide）成相等的四个区域，分割的深度越大，则得到的分辨率越高。即分割深度每提高一层，采样密度提高一倍。图 4.2 演示了分割的过程。

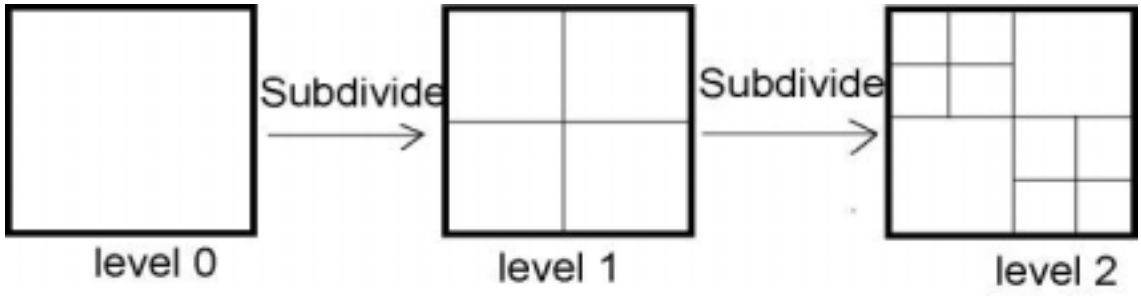
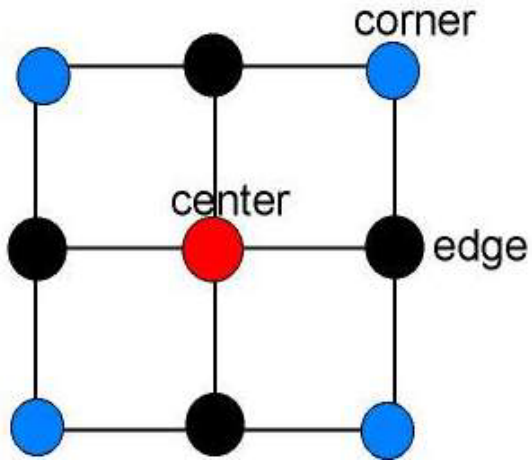


图 4.2，分割过程示意图，level 1 到 2 时，只对两个节点进行了分割

图 4.3 给出了我们在一个四叉树节点中要保存的信息。在本章的第四节中将看到，如何用这些信息渲染这个节点表示的地形区域。



(图 4.3) 一个节点中记录的信息，红色的为中心点，黑色的为边点，蓝色的角点。一共 9 个点。

采用四叉树的概念来表示多分辨率地形有许多优点，一个最直接有效的受益就是裁剪，如图 4.1 所示，其中红色的区域为观察者能看到的部分。我们很容易知道观察者能看到的只是绿色的节点，白色的节点则根本不需要考虑。因此，我们可以在节点递归分割的初期只花很少的代价就可直接把这些看不到的区域简单的丢弃掉。

有了地形的逻辑表示后，我们还要建立一个节点评价系统来判定一个节点何时需要被继续分割，何时被直接丢弃（当这个节点不能被观察者看到的时候，节点将被直接丢弃）。如果一个节点没有被丢弃，也不需要继续分割，那么这个节点将被送入渲染 API 进行图元渲染。

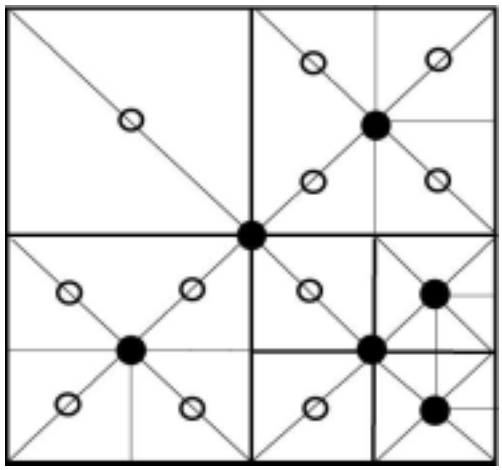
二．数据存储

地形数据通常存储在高度图里，在内存的结构即为一个二维数组。我们知道二叉树可以有顺序结构和链式结构，同理四叉树也可以采用类似的顺序结构。不同的是这里我们采用二维数组而不是一维数组。我们把全分辨率的地形数据存储在一个二维数组中，四叉树节点的信息（9个顶

点的信息）可以直接通过索引在数组中读取。同时还要建立一个和这个地形数据数组大小相同的标志数组，这个标志数组指示四叉树节点的状态。如果一个节点需要被继续分割，我们则把相应的位置标记为1，否则标记为0，如图4.4，标着问号的表示没有被访问到，（注意没有被访问到的地方的数值是不确定的）。

由图4.4的数组易知地形大小为什么要满足$(2^n+1)\times(2^n+1)$。（对于不满足大小要求的地形，我们必须把把它分割成满足要求的大小，然后进行拼接。出于算法的简洁性，本文只考虑大小为$(2^n+1)\times(2^n+1)$的地形）。

?	?	?	?	?	?	?	?	?
?	?	?	?	?	0	?	0	?
?	?	0	?	?	?	1	?	?
?	?	?	?	?	0	?	0	?
?	?	?	?	1	?	?	?	?
?	0	?	0	?	0	?	1	?
?	?	1	?	?	?	1	?	?
?	0	?	0	?	0	?	1	?
?	?	?	?	?	?	?	?	?

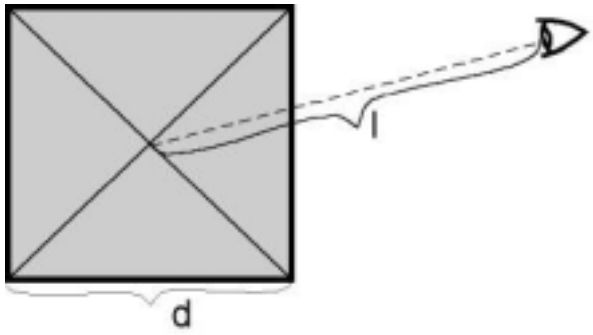


（图 4.4） 一个地形标记数组（9×9）示意图，右边为用这个标记数组渲染的地形，其中黑点表示当前节点需要继续分割，空心点表示不需要继续分割。

二维数组的存取是按行或者按列的。考虑到观察者在场景中移动时候的区域性，本文尝试使用了一种按区域存储的二维数组，即在物理上把地形数据分成等大小的块，块的大小不能太大，也不能太小。考虑到Intel 的 CPU 的内存页大小是 4K，块的大小应该为 64×64 比较合适，本文采用了 32×32 的块。这样的存储结构在一定层度上能提高存储效率，降低内存缺页的次数。关于如何使存储结构更加有效的方法在很多文章都有介绍，[参考文献 1]就给出了一种称为分簇的内存数据结构，能有效的降低内存缺页带来的性能影响。

三. 节点评价系统

首先我们要建立一个节点评价系统,决定何时该对一个节点进行继续分割。我们把这个评价系统分成两个部分，一是视点相关的，二是地形本身的粗糙层度相关的。裁剪器理论上也该是节点评价系统的一部分，但是考虑到它的特殊性，我们将裁剪器放在单独一章中讨论。



(图 4.5)视点距离因素示意图，l 为视点离节点中心的距离。d 为节点的尺寸

①我们希望离观察者近的地方细节越多，反之则越少。将距离因素应用于一个节点的时候，还必须考虑到节点的大小。因此，结合图 4.5 我们如下的公式：

$$\frac{l}{d} < C$$（C 为一个可以调节的因子）

其中 l 为节点的中心位置到视点的距离，d 为节点的大小，当它们满足这个公式的时候，节点需要继续分割。其中 C 为一个可以调节的因子，C 越大，地形细节越多。反之则细节越少。

②第二个需要考虑的因素是地形本身的粗糙程度，我们希望地形起伏比较崎岖的区域有较高的细节程度，而平坦的地方则不需要我们浪费过多的图元。

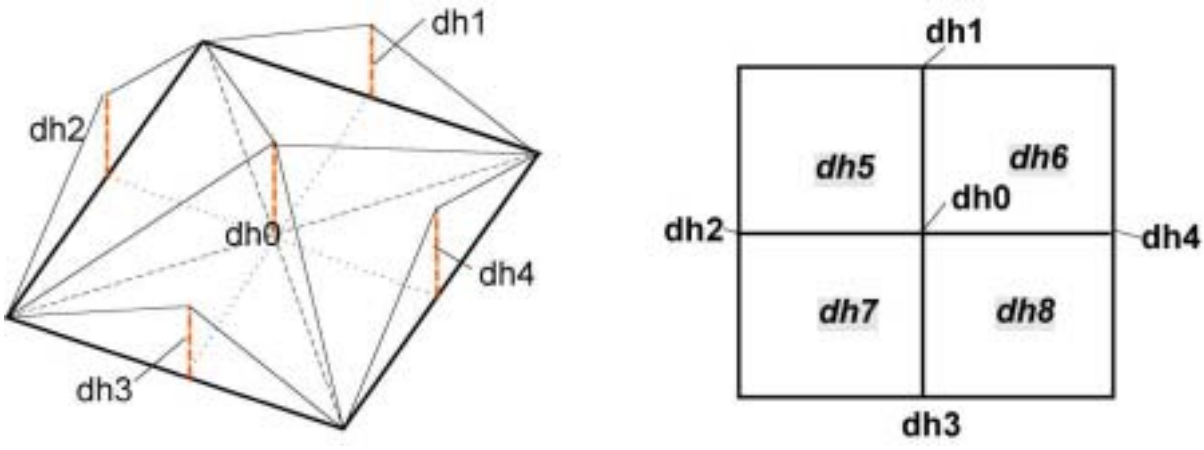


图 4.6，地形粗糙程度的度量，左图表示了一个节点内部 5 个粗糙度信息，右图则表示了它本身的粗糙度信息和它所有子节点的粗糙度信息

如图 4.6 所示，首先我们考虑一个节点包含的 9 个顶点，其中中心点

4 个边点在节点被分割和不被分割时候会引起一定的误差，这 5 个误差值为图 4.6 左图所示的 dh0 到 dh4。它们的数值越大表示这个节点表示的地形越粗糙。除此之外，我们还要考虑到这个节点的所有四个子节点的粗糙程度 dh5 - dh8，如图 4.6 右图所示（如果这个节点达到了最高分辨率表示的地形，则不需要考虑这一步）。我们取这九个值（dh0 - dh8）中的最大的一个除以节点的大小作为这个节点粗糙度的评价值，即 r = Max(dh0,…dh8)/d。由此可见，粗糙度的计算是一个递归的过程，考虑到计算的复杂性和值的不变性，我们需要事先把粗糙度评价值计算出来，使其在运行时刻不需要额外的计算。同样把它存储在一个二维数组中。

现在我们给出第二个评价公式即粗糙度评价公式：

$$\frac{1}{r} < C_2 \quad （C_2 为粗糙度调节因子）$$

C_2 为粗糙度调节因子，C_2 越大，细节程度越高。

综合以上两个公式，我们得到最终的节点评价公式：

$$f = \frac{l}{d \times r \times C \times C_2} < 1$$

公式中的字母含义同上，当满足 f<1 时候，节点需要继续分割。

至此，整个节点评价系统已经建立完成，但是我还必须提一下几何形变（Geomorphing）的概念。几何形变会造成“跳出”现象，即随着视点的改变，有些细节会突然消失和出现。解决这种现象的办法是比较困难的，目前在有些文章中，通过专门的方法甚至是插值的方式来消除或者降低几何形变（**[参考文献12]**等），但是实现这样的系统很困难，计算代价也很大。其次的方法就是把r值进行投影变换到屏幕空间，得到的值称为Projected pixel error（**[参考文献3]**等）。但是通过实验，我不认为这是一种很好的方法（详细原因可见**[参考文献8]**）。因此本文并没有采用这种pixel error的概念。其实本文并没有采用任何专门的消除几何形变的系统，但是我们可以通过适当的调节C和C2的值来降低几何形变，因为对于一个视频游戏来说，只要能把几何形变控制在一个可以接受的范围内就可以了。

四．网格的渲染

地形网格的渲染最终也是通过一个递归的过程来实现的。我们遍历整个四叉树，当我们到达四叉树的叶子的时候，即一个节点不再被分割的

时候，我们就可以把这个节点给绘制出来。

本文采用三角形扇（Triangle Fan）的方式来绘制节点，这是一种很自然的方式，因为一个节点包含了一个中心点和若干个围绕着中心点的点，这样的排列刚好形成一个三角形扇，如图 4.7a 所示。

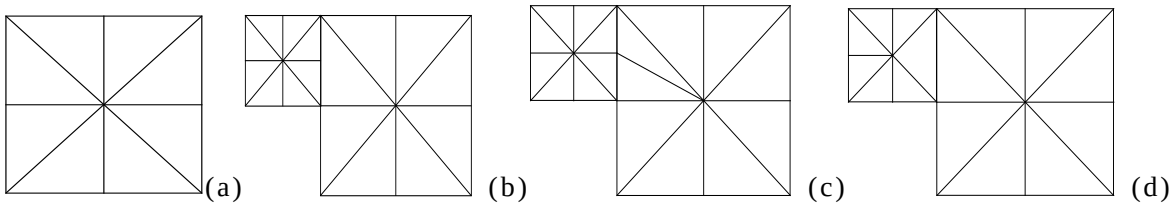


图 4.7

生成网格的时候,我们还有一个注意的地方就是两个不同分辨率的节点拼接处会产生 T 形裂缝，如图 4.7b 所示。我们必须消除这种裂缝，图 4.7c 演示了在拼接地方增加一条边的方法来消除裂缝，图 4.7d 则采用了去掉一条边的方法。相对来说，第一种方法更加的复杂，但是也更加的全面，因为拼接处的两个节点的分辨率可以相差任意大。第二种方法则更加简单，它要求拼接处的两个节点的层次差距最多不超过 1。本文采用第二种方法，对于如何满足这个要求在下面的**网格的生成**一小节中将作详细的介绍。

结合图 4.8 的例子，我们详细介绍一下如何生成满足要求的三角形扇。图中灰色区域为我们当前进行渲染的部分。首先，我们保证一个节点的四个角点（Corner vertics 见图 4.3）肯定被用到三角形扇中，对于剩下的四个边点（Edge vertics）我们则要检查和这个节点相邻的节点，因为边点要和其它的节点共享，如果相应的邻接节点没有被激活，我们就要跳过这个边点，如这个节点正上方邻接节点没有被分割，则我们要跳过标有 X 标记的那个边点。

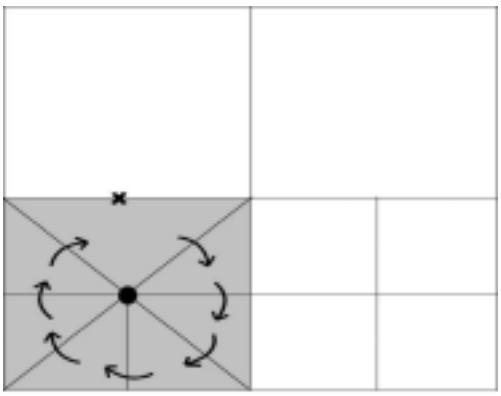


图 4.8 地形网格的渲染示意图其中灰色的节点为当前进行渲染的区域

五．网格的生成

在渲染网格之前，我们必须更新四叉树，生成符合规范的四叉树，在前一小节中，我们曾给出相邻的两个节点的层次最大不能相差 1，否则在拼接的地方会出现裂缝，图 4.9a 给出了一个符合规范的四叉树例子，图 4.9b 给出了一个非法的四叉树例子。

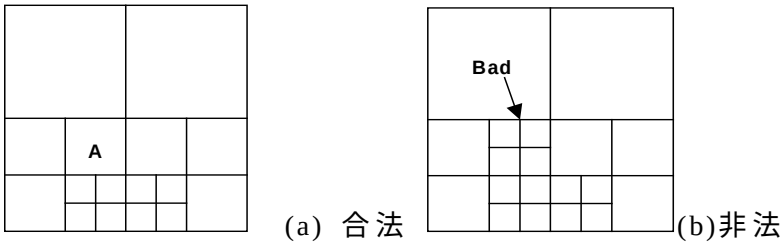


图 4.9 符合规范的和不符合规范的四叉树的例子

因此，我们必须要有一套规则来保证生成的四叉树的合法性

通常，我们采用的是两次遍历四叉树的方法，我们在第一次遍历的时候，生成地形网格，第二次渲染网格，同时消除节点之间的层次差异。

这里，我们采用一种更加有效的方法来生成四叉树----按广度优先的原则遍历四叉树。即一次生成所有属于同一个层次的节点，这样我们就只需要遍历四叉树一次。我们使用两个队列，一个队列保存着当前正在处理的层次的所有节点，另外一个队列则保存着处理当前层次节点后生成的所有的下一个层次的节点。当我们处理当前层次节点的时候，把分割生成的下一个层次节点都送入另一个队列种，当处理完所有当前层次队列中的节点以后，就可以进入下一个层次（简单交换两个队列就可以了）的节点处理。对那些不要继续分割的节点和已经到达最大分辨率的节点，我们就把它们送入渲染 API 进行渲染。不可见的节点则直接丢弃。

这样做有很多的优点。首先因为我们每检查一个节点的时候，和该节点层次相同的节点都已经生成。我们可以通过检查所有和这个节点相邻的节点，看它们是不是存在，如果它们都存在，则可以对这个节点进行继续分割，反之则不能对它进行分割。同时，我们还可以在第一遍遍历四叉树的时候有足够的信息让我们绘制三角型扇，根据本章第四节的方法，渲染一个节点的时候，我们只需要检查分辨率比该节点小的节点。而这些节点在此之前已经全部生成。

其次，我们还不必要每次都复位四叉树的状态（清空标记数组）。这对提高速度是很有帮助的。举个例子，假如一个地形的大小是 2048×2048。那么这个标记数组的大小是 4M。要清空一个 4M 大小的数组在时间上是一个不小的开销。在本文的程序中，测试一个 2048×2048 的地图，清空标记数组时的 FPS 值为 35。不清空时的 FPS 为 78。相差一倍多！

下面我给出生成网格的伪代码：

```
Function GenerateMesh
Begin
    Push the root node to the cur_Queue
    level = 0
    Loop Not reach the Full resolution)
    {
        For Each Quad-Tree Node in Cur_Queue
        {
                If(Node is not inside the view frustum)
                {
                    Simple Skip this Node
                }
                else if(level = Full Resolution level –1 )
                {
                    Draw The Node
                }
                else
                {
                    For Each Sub-Node in this Node
                        Check dependcy
                        If(SubNodeCanSubdivid() and SubNodeNeedActive())
                        {
                            Push this sub Node to Next_level_Queue
                            Set this sub Node flag to VS_ACTIVE
                        }
                        Else
                        {
                            Set this sub Node flag to VS_Disable
                        }End if
                    End for
                    If   No sub Node is active
                    {
                        Disable this Node
                        Set all four sub-node flag to VS_DISABLE
                        Draw the Node
```

```
                }
                else if Some Sub-Node is active
                {
                    Draw the Node
                }End if
            }End if
        } End for
        Swap (cur_Queue,Next_level_Queue);
        level    = next level
    }End Loop
End Function

Function NodeCanSubdivid as BOOL
Begin
    Check the four Neighbor Node.
    If All Neighbor Node is Active
        return TRUE
    Else
        return FALSE
    End If
End Function
```

六. 优化

上面介绍的算法在理论上是比较严谨的，但是稍微显的有些复杂，同时速度也不是十分的快。下面，我将对它进行一些优化和简化。

在上面的算法中，当一个节点的四个子节点中有一部分被分割，另一部分不被分割的时候，给我们渲染带来很大的麻烦，而且每处理一个节点的时候，我们都要检查四个子节点，比较麻烦。为此参照[参考文献 13]给出如下规则：当一个节点的四个子节点中任何一个需要继续分割的时候，四个子节点都进行分割。在本文的里，这个规则进一步成：**一个节点需要分割的时候，就把其四个子节点都生成并放入到下一层次的队列中去。**

按照这种简化的思想，图 4.4 中的标记数组对应的地形网格最终将如图 4.10 所示。对比图 4.4 右图，我们发现其实简化后的算法生成的地形细节更多，也就是说需要绘制更多的三角形。但是由于需要判断的条件少的多，因此在运速度上，反而是简化后的算法要更占有优势。

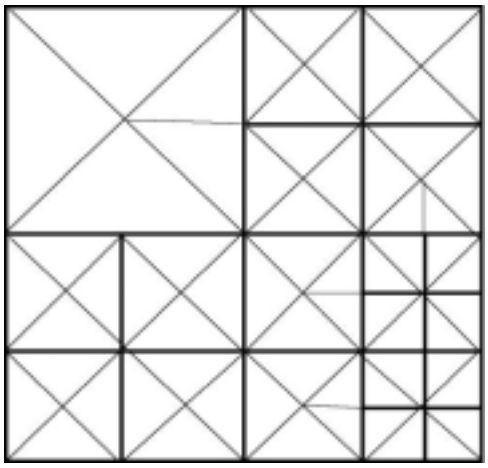


图 4.10(图 4.4)中的标记数组对应地形的简化生成算法

下面我给出优化后的伪代码,对比上面的算法,它显得更加的简洁了。

```
Function GenerateMesh
Begin
    Push the root node to the cur_Queue
     level = 0
    Loop while Not reach the Full resolution
    {
        For Each Quad-Tree Node in Cur_Queue
        {
                If(Node is not inside the view frustum)
                {
                    Simple Skip this Node
                 }
                else if(level = Full Resolution level –1 )
                {
                    Draw The Node
                }
                else if( NodeCanSubdivid() and NodeNeedActive())
                {
                        Sub Divide this Node
                        Set all four sub-node flag to VS_ACTIVE
                        Push the for sub-node to the next_level_Queue
                }
                else
```

```
            {
                    Disable The Node
                    Set all four sub-node flag to VS_DISABLE
                    Draw this Node
            }End if
        } End for
        Swap (cur_Queue,Next_level_Queue);
        level    = next level
    }End Loop
End Function
```

第五章：裁剪

通常，3D API 都会提供内置的裁剪系统，但是这些裁剪系统都是在经过视图、投影变换以后的。即裁剪是投影空间中进行的。因此我们需要建立一种方法，在顶点进行矩阵变换之前就进行裁剪，同时我们的裁剪系统还必须要有能力裁剪一个四叉树的节点。

在 3D 裁剪中，通常有两种裁剪是比较容易实现的：View Frustum Culling 和 Back face Culling。而基于阻挡关系的裁剪则比较难实现。本文需要在世界空间实现一个 View Frustum 裁剪器。

在 3D 到 2D 投影过程中，需要一个投影体，只有当物体处于这个投影体中的时候，我们才能看到这个物体，否则物体将被裁剪掉。因此这个投影体也通常被称为视见体（View Frustum）。在进行正交投影的时候，投影体为一个长方体，在进行透视投影的时候，投影体则为一个平头锥体。下面我们以平头锥体为例子来导出我们的裁剪系统。

如图 5.1，一个投影体由六个面组成，一个平面的方程可以表示为 $Ax+By+Cz+D=0$。我们规定朝投影体内部的方向为平面的正方向，判断一个顶点是否在投影体内部时，我们只要把顶点坐标代入到六个面的方程中，通过检查结果的符号就可以判断点是不是在投影体内部（所有的符号都为正）。下面我们推导世界空间中的投影体的六个面的方程。

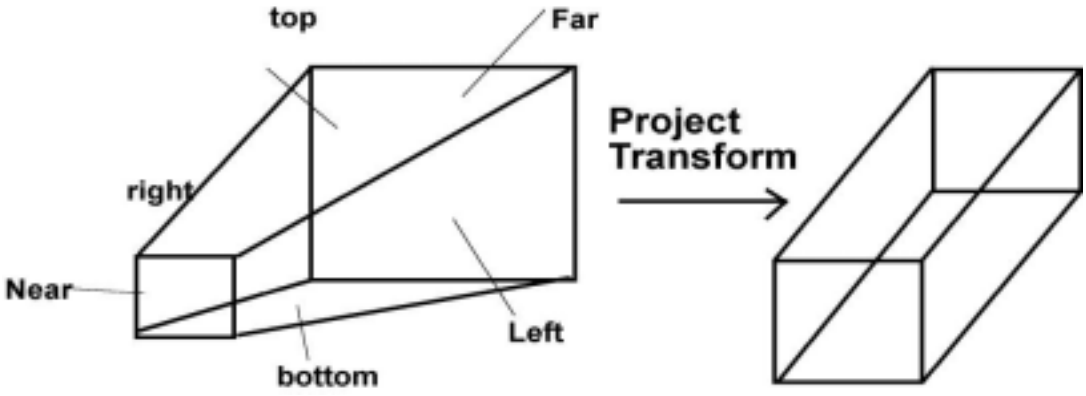


图 5.1 投影体，左图为世界空间中的投影体。右图为经过投影变换后的投影体。

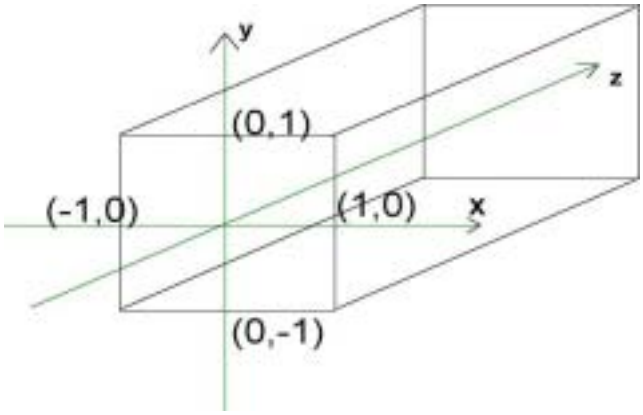


图 5.2 变换后的投影范体的尺寸。

世界空间的投影体在经过投影变换后，会成为一个范体，如图 5.1 右图所示。这个范体的尺寸见图 5.2。我们很容易得到这个范体的六个面的方程，它们是 $\begin{cases} near: 0x+0y+z+0=0 \\ far: 0x+0y-z+1=0 \\ left: x+0y+0z+1=0 \\ right: -x+0y+0z+1=0 \\ bottom: 0x+y+0z+1=0 \\ top: 0x-y+0z+1=0 \end{cases}$。我们假设这六个平面中某个平面上有一个点（x0,y0,z0,1），在进行投影变换之前的坐标为（$x0^{'}, y0^{'}, z0^{'}, 1$）。这个平面的方程为 $Ax+By+Cz+D=0$。投影变换前，在世界空间中的方程为 $A^{'}x+B^{'}y+C^{'}z+D^{'}=0$。则点必须满足

$(x0,y0,z0,1)\times\begin{pmatrix}A\\B\\C\\D\end{pmatrix}=0$（投影后）和 $(x0',y0',z0',1)\times\begin{pmatrix}A'\\B'\\C'\\D'\end{pmatrix}=0$（投影前）。如果变换矩阵为 T。则投影前后的点要满足 $(x0',y0',z0',1)\times T=(x0,y0,z0,1)$。结合这三个等式，我们立即就可以得到 $T\times\begin{pmatrix}A\\B\\C\\D\end{pmatrix}=\begin{pmatrix}A'\\B'\\C'\\D'\end{pmatrix}$。再根据投影空间中范体的六个面的方程，我们现在可以很容易的得到世界空间中的投影体的六个面的方程。

我们已经有了裁剪体的方程，当我们需要裁剪一个顶点的时候，这六个方程已经足够了。但是我们要判断一个区域的可见性时，我们进行一些额外的计算。如图 5.3 所示，一个物体和投影体的关系大致可以分为：包围、被包围、相交和相离四种情况。图中最大的浅蓝色的矩形包围了整个投影体。深绿色的小矩形则完全被投影体包围。浅绿色的矩形和投影体相交。这三种情况下物体都是可以被看到的。剩下红色的矩形则和投影体相离、只有它完全不可见。

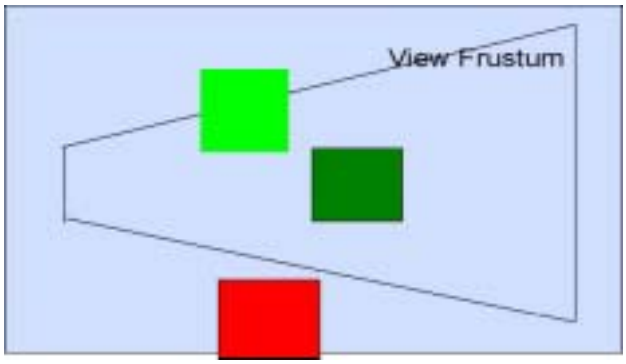


图 5.3 物体和投影体的关系。

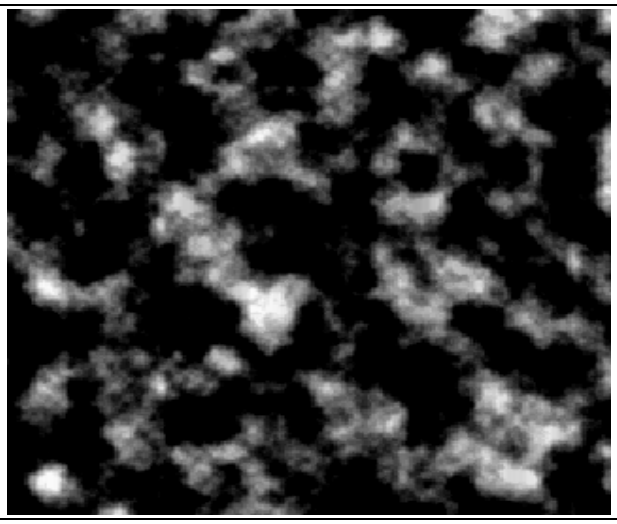
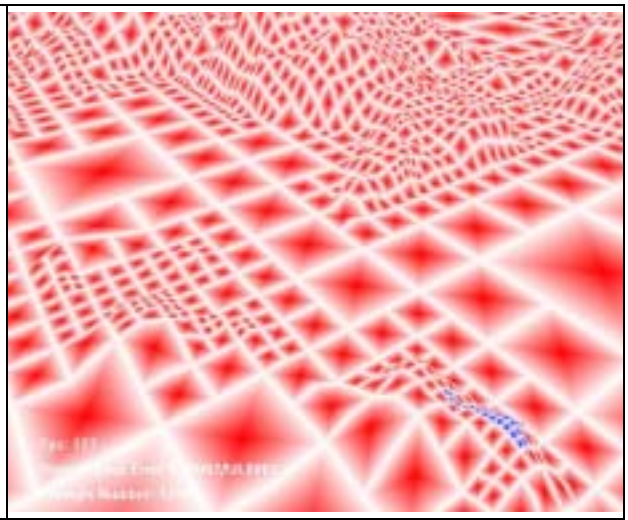
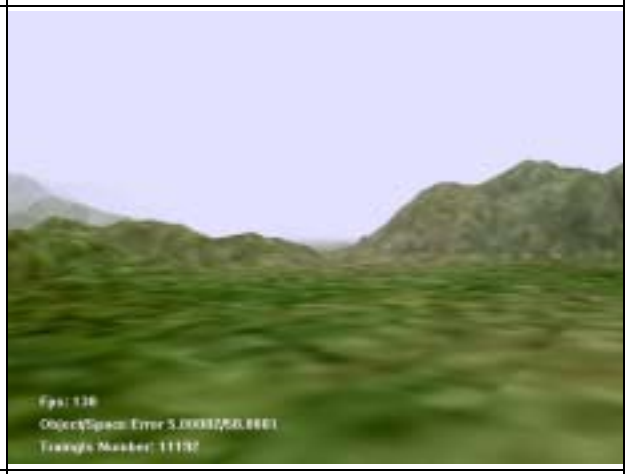
当处理节点的可见性的时候，由于节点的不规则性。我们还需要引入包围体的概念。所谓的包围体，就是用一个比较简单的几何体去度量另外一个比较复杂的几何体，让它刚好能包围另外一个几何体。比较合适

的包围体外形有矩形、正方形和球体。其中球体处理最为简单，但是近似度也最差。我们为每一个节点都建立一个包围体，只要测试这个包围体，我们就可以决定一个节点的可见性，由于包围体肯定大于这个节点，因此我们可以保证不会有任何可见的节点被裁剪在投影体之外。

第六章：性能测试

1. **内存消耗量：**本文的 LOD 算法的内存消耗量只和地形的大小有关，而且这种关系是线性的。对于地形中的每一个顶点，我们需要的信息如下：高度信息（1 字节），粗糙度信息（浮点型，4 字节）。四叉树信息（1 字节）。因此每个顶点需要 6 个字节来保存。对于一个 4097×4097 的地图，我们需要 96M 的存储空间，这在连 PC 机的 RAM 大小都几乎要以 GB 来计算的今天，应该不是很大的问题。同时考虑到本文采用的都是静态的数据结构，在内存消耗方面应该还存在着很多的优化余地。
2. **速度和图像质量：**本文的算法既没有在进行恒定的速度控制，也没有进行恒定三角形数量的控制。生成三角形的数量除了和 C,C2 的大小有关以外，还和地形本身的起伏程度有关。本文的演示程序在一般情况下，C=3,C2=30 左右就能达到比较好的效果。当 C=4.5 时候，就基本不存在几何形变的问题。而且 FPS 均在 120 以上（使用了细节纹理），速度完全达到要求。
3. **实例测试：**我选择的测试地图大小为 2049×2049 和 4097×4097。地图使用 PhotoShop 的分层云彩功能制作，并且用本文演示程序的地图工具进行修改加工。纹理为区域地貌纹理加亮度图调制混合得到。细节纹理被关闭(即只进行了一遍纹理映射)。测试平台为 Intel 赛扬 II 1.2G，128M RAM， nVidia GForce 2/MX400 32M DDR VRAM，。操作系统为 Windows 2000sp2。显卡驱动版本为 4345 官方发布版。注意驱动程序必须安装正确，Windows2000 提供的驱动并不能很好的支持 OpenGL。

表 6.1：2049×2049 地图的渲染结果。

	
测试用的高度图，大小为 2049×2049.	使用双色模式渲染。用来演示生成的三角形扇，三角形扇的中心红色，周围的点为白色。中心为蓝色的三角形扇表示这个地方达到了全分辨率。
	
C=25，C2=2.5. 三角形数量 6529。视野距离 600。FPS=167	C=50,C2=5.0.三角形数量 11192。视野距离 600。FPS=138

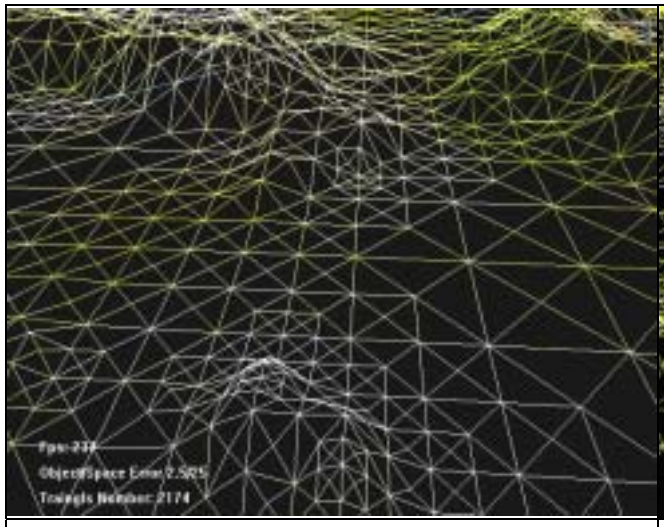
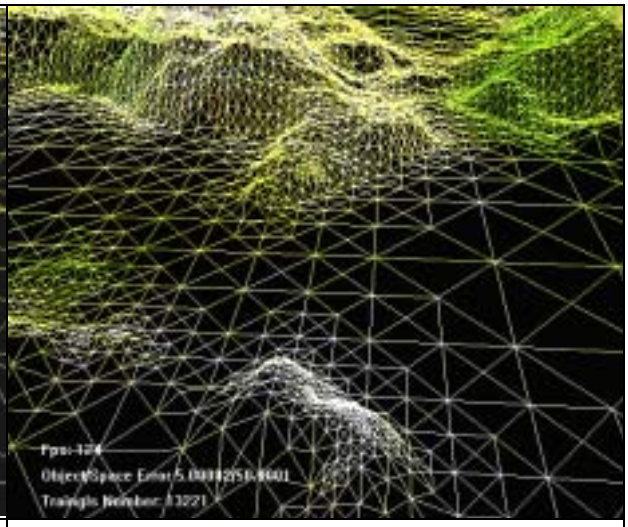
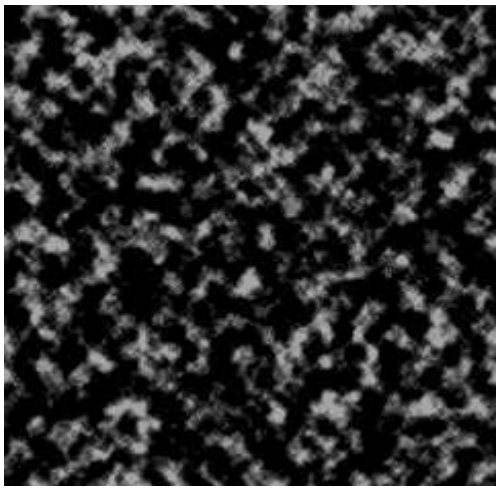
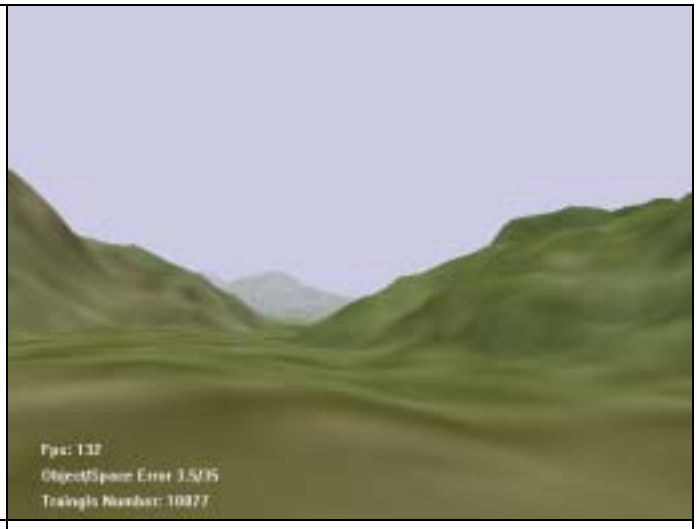
	
C=25，C2=2.5. 三角形数量 2174。视野距离 600。FPS=237。网格着色模式	C=50，C2=5.0. 三角形数量 13221。视野距离 600。FPS=124。网格着色模式

表 6.1：4097×4097 地图的渲染结果。

	
使用的高度图	C=35,C2=3.5.三角形数量 10877。视野距离 6000。FPS=137

本算法的速度基本上只和 C,C2 有关，图 6.1 为这两个因子和速度、三角形数量之间的关系。图中的测试数据中 C=35 恒定，C2 从 2.5 到 10.0。由于这两个由图可见，速度和三角形的数量和 C×C2 大致是先线性关系。

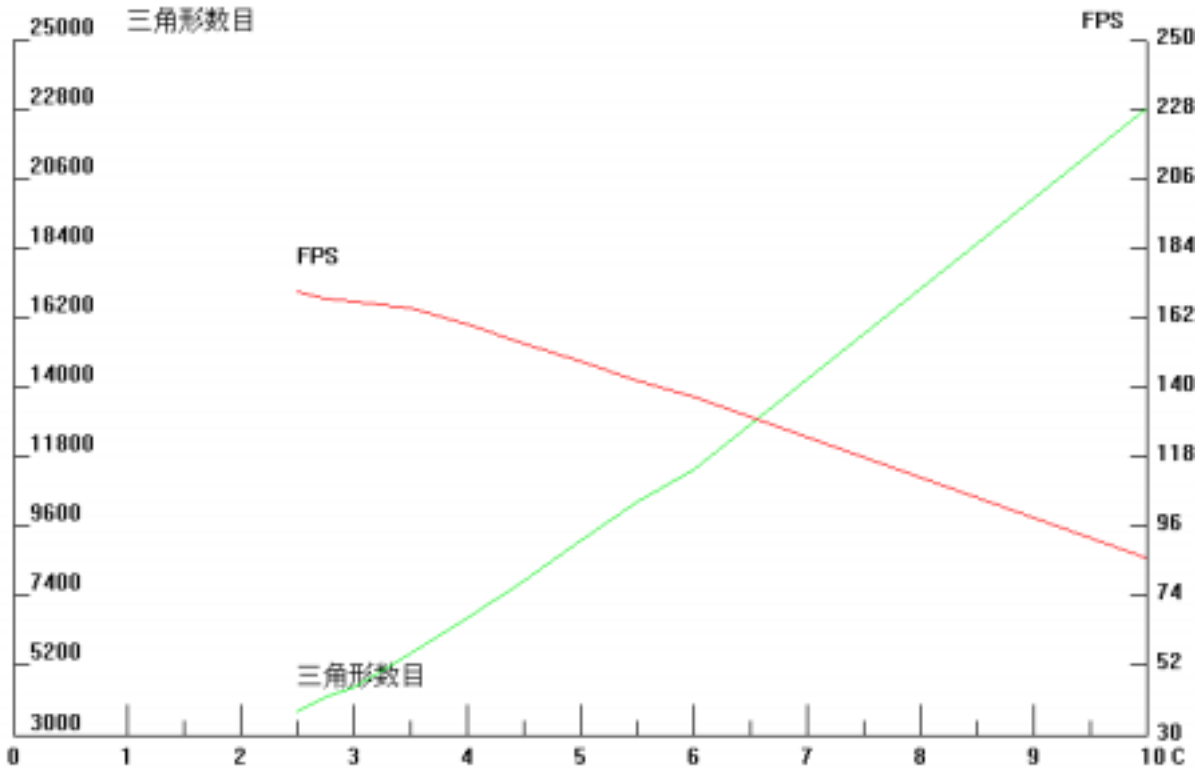


图 6.1 三角形数量，FPS 和 C×C2 之间的关系。红线为 FPS 值，绿线为三角形数量

在选择 4097×4097 的地图时，算法运行速度上没有任何的变化，因为需要渲染的地形区域仅限于视见体内部。当选择 8193×8193 的地图时候，由于内存缺页引起的丢帧就比较严重，如果在 Windows98 下就基本不能运行。

第三部分
真实感场景的生成技术

引　言

一个室外场景渲染引擎必须要具有真实感的渲染能力，这些能力包括让地形表面有更多的细节、渲染地表植被等，甚至各种天气效果。所有的这些都能让场景看起来更加的自然，更加的逼真。在下面的几章中，我们将要实现其中几种主要的项。

第七章：天空体和镜头眩光

1．天空体：如果一个室外场景中看不到天空是不可想象的。一个蓝天白云的天空能让视野变的更加的清朗。而布满了乌云的天空则能让场景看起来更加的压抑。通常天空体的实现有圆形的和盒状的。圆形的天空体有很多的优点，但是比较复杂（尤其是映射纹理的时候）。盒状的天空体则相对要显得的简单一些，它的主要缺点是离天空边缘近的时候天空会有非常明显的变形。关于天空体几何模型的生成很多文章都有介绍，如[参考文献 33]就是一篇很好的天空体制作教程。

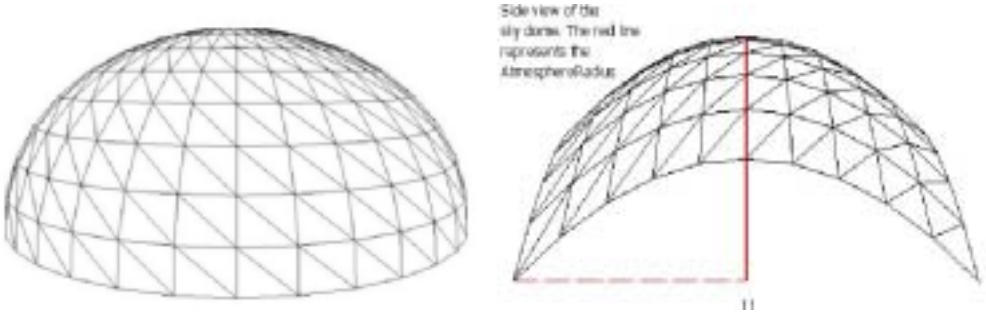


图 7.1 圆形天空体示意图。左边为半球状天空体，
右边弧顶状天空体。图片来自[参考文献 33]

天空体最难的是纹理的生成，当然你可以找一张天空的照片，但是这样天空是静态的。其次是用算法动态生成天空的纹理，生成这种纹理的算法通常是使用 Perlin Noise。关于生成天空的精彩文章在 http://freespace.virgin.net/hugo.elias 可以找到。这种方法生成的云彩真实性非常的好，但是致命的缺点生成大纹理的时候速度不够快，虽然能达到一定的实时性，但是应用到整个引擎中的时候，速度就非常的慢。最后的方法是用天空的视频文件做纹理，这种方法虽然速度还不是十分的

快，但是它比用 Perlin Noise 生成纹理的速度还是要快的多。（本文的演示程序附带了一个演示视频纹理的程序，它使用 DirectShow 来播放视频文件，将播放结果作为纹理映射到几何体上。）

2．镜头眩光：稍微有一点摄影经验的人都知道，当我们把摄影机的镜头对准发光物体的时候，会有光晕现象产生，这种光晕也称为镜头眩光。光晕的大小和镜头有关。当我们在场景中放置一个太阳的时候，如果实现了这种光晕现象会有很惊人的效果。

太阳的模拟非常的简单。只要在场景的特定位置放置一个圆形物体就可以了。镜头光晕则需要一些额外的处理。首先我们来看一个眩光体的位置的。如图 7.2 所示。

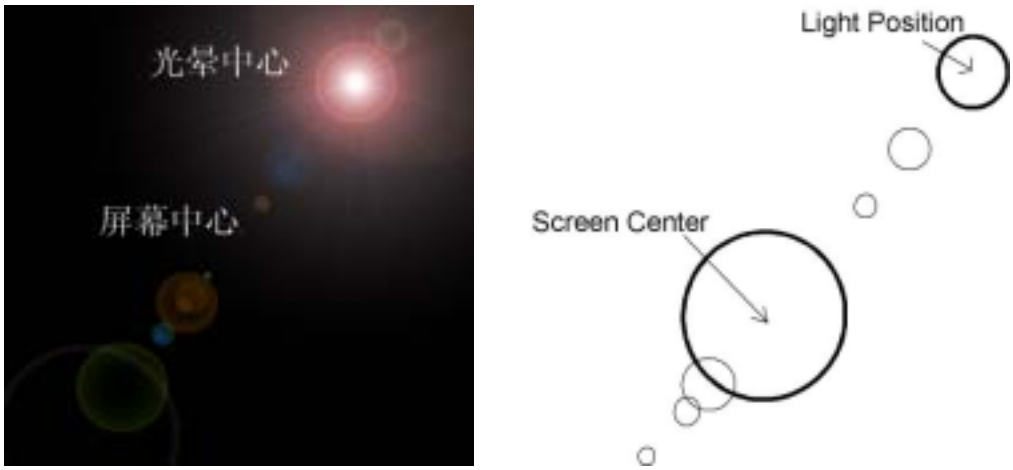


图 7.2 镜头眩光示意图，左图为用 Photoshop 生成的效果图，右图为其组成部分示意图。

镜头光晕由一系列的光环组成，所有的光环排列在一条直线上。这条直线由发光物体位置和屏幕中心确定。这些光环的形状则有多中方式，通常用一张放射线状的图用来作为发光物体的光芒，另外的则用不同厚度的环状图像。

我们组合这些光环的时候需要使用 Alpha Blending 功能把这些光环叠加起来自然的融合到场景中去。通常我们采用的 Alpha Blending 公式为

$$Color = (r_s + r_d, g_s + g_d, b_s + b_d)$$

所有的光环都是作为二维物体绘制到场景中去的。因此，发光物体在屏幕上的位置需要通过自己计算得到，在计算发光物体在屏幕上的位置的同时我们还可以得到这个位置的 Z-Buffer 的信息来判断是否能看到发光物体，当发光物体不可见的时候，光晕自然也不可见。下图为一个用上面算法生成的镜头光晕的例子。

图 7.3，镜头光晕示意图，左图为独立效果，右图为组合到场景中后的效果图。图片来自本文的 Demo

最后必须注意的是，眩光体必须在场景中所有的物体都绘制完毕后才绘制。否则会有不可想象的现象出现。

第八章：公告板技术

公告板（Billboard）是一种始终朝着观察者的一个物体。通常它是一个多边形。室外场景中通常有一些物体，比如说树木，柱子一类的物体，要么是细节非常的多以至无法用模型来表示，要么是从任何一个方向都一样（如柱子）。我们就简单的把这些物体用一个多边形加纹理映射来近似的表示。

用 Billboard 来表示树木非常的有效，著名的模拟驾驶游戏 Need for Speed 5 中的树木就是用这种技术实现的。我们所看到的树木其实只是一个矩形，如图 8.1 所示，矩形可以绕着一根轴旋转，使矩形始终对着观察者。因此无论你从哪一个角度去看，你都只能看到矩形的正面，而不会看到扁的一面。

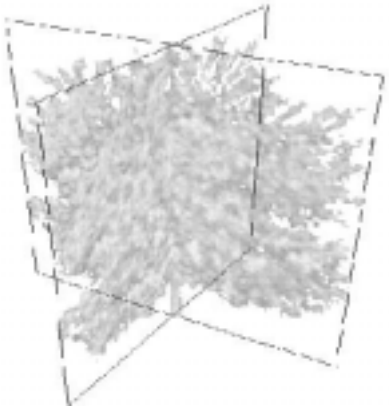


图 8.1Billboard 模拟树木的示意图

除此之外，Billboard 通常还用在粒子系统之中，因为 3D 场景中的粒子要始终正对着观察者。这种应用的典型例子就是 Quake III 和 Unreal 中绚丽的子弹光焰。

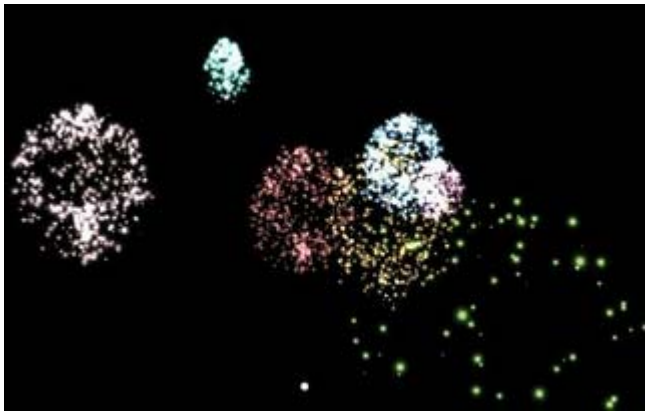


图 8.2 3D 场景中的用来模拟烟花的 Billboard。

Billboard 的实现关键在两个地方，其一是纹理贴图的模式，对于像绘制树木这一类形状不规则的物体，通常是采用透明贴图的模式，在 OpenGL/Direct3D 里我们可以采用 Alpha test 的功能。对于粒子系统则一般采用 Alpha Blending 的功能。第二个关键的地方就是如何让多边形的一个面始终朝着观察者，如图 8.2 所示。

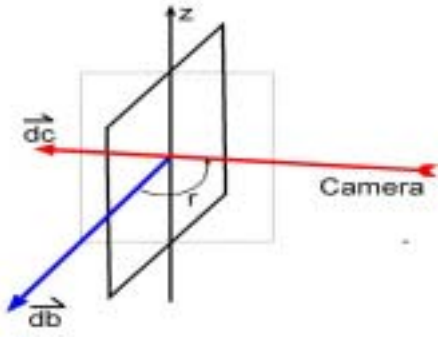


图 8.3 Billboard 旋转角度的计算，其中红的箭头为摄影机的朝向，蓝色的箭头为 Billboard 原来的朝向，z 为转动的中心轴

我们需要把多边形旋转一个角度 r 使其面向观察者。角度 r 的大小可以用如下的公式去计算：$\cos(\pi - r) = \frac{(\vec{dc}) \cdot (\vec{db})}{|dc| \cdot |db|}$ 。转动的中心轴则可以通过如下公式计算：$z = (\vec{dc}) \times (\vec{db})$ 。

第九章：地表的细节

地表细节涉及的方面非常的多，主要的有如下的方面：地面纹理，地面阴影（亮度图）。

1. 地面纹理 由于室外的场景比较大，通常需要一个很大的纹理来表达地面的细节信息。但是太大的地表纹理一来要占用很大的纹理内存，二来图形卡也不一定能支持。

解决这个问题的方法通常有两种，一是把大纹理分割成小的单位，动态加载需要的纹理（这种技术也称为 Texture Tiling，详见[参考文献 24]）。另一种方法是采用多遍纹理映射（Multi-Stage Texture mapping），它用一个较大一点纹理来表达地形的大概特征，然后再用一个小的纹理来和第一个纹理调制（Texture blending）在一起，第二个纹理通常采用循环贴图（即贴图的 u,v 坐标大于 1）。不同的细节纹理可以表达不同的地貌。

相比之下，第二种方法更加的简单，效果也更好，唯一需要注意的是不同细节纹理过渡的地方。要事先混合好过渡处的纹理，不然在不同的细节纹理过渡处的地貌变化会非常的生硬，十分的难看。

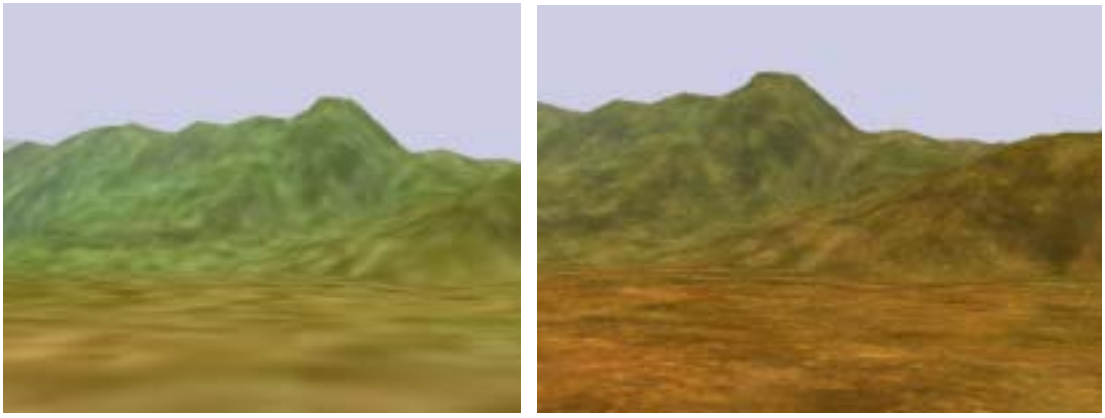


图 9.1 细节纹理示意图，左图为没有使用细节纹理，右图为使用了细节纹理。图片来自本文的 Demo

2 **亮度图** 在一个 3 维场景中增加光照和阴影效果可以更加提高场景的真实程度。

光照的计算需要为各个三角形指定法向量，动态计算法向量速度太慢，事先计算好法向量则需要大量的额外存储空间，而且在 LOD 算法中存储法向量也不是一件容易的事情。另一个问题是阴影问题，实时阴影的生成一直是图形学中的一个难点。何况在室外大场景下，实时动态生

成的阴影基本是不可能的。

解决以上两个问题的一个折中方法是使用亮度图，亮度图的通常使用方法是进行二次贴图模拟照明效果（如 Quake III）。但是二次贴图比较消耗时间。本文采用把亮度图和地面的纹理进行调制，得到最终的地面纹理图的方法，这种方法在运行时刻不需要额外的计算，也不需要额外的存储空间，同时也只进行了一次纹理贴图，大大的提高了速度。

计算亮度图的时候，我们需要给场景指定一个光源的位置，这个位置通常是镜头光晕的发光体（太阳）的位置。一个点的亮度由所有共享这顶点的三角形决定。其中任何一个三角形的亮度计算如下图：

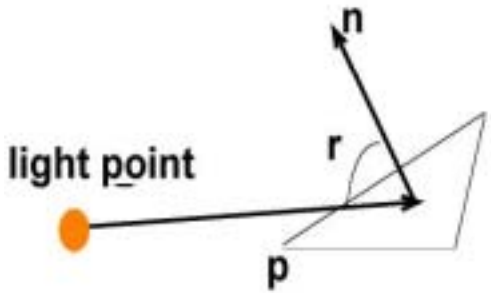


图 9.2。亮度图的计算,其中 r 为三角形法向量和光线方向的夹角

根据常识，只有朝着光源的面才会被照亮。假设全场景的环境光的强度为 Ie，则这个如图 9.2 中三角形的亮度 I 由下面的算法决定：

```
if cos (r) > 0
    I = cos (r) × (1 – Ie) + Ie;
Else
    I = Ie;
```

阴影的计算则如下图所示：

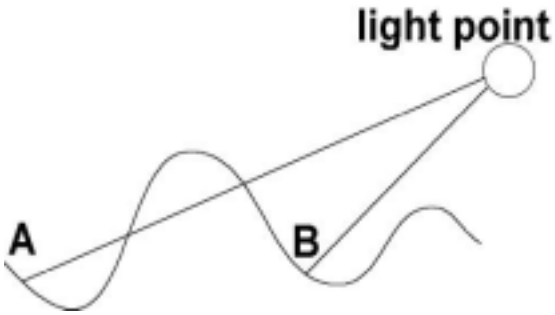


图 9.3，阴影的示意图，其中 A 点在阴影区内，B 直接被照亮

确定一个顶点是否在阴影区内的时候，我们需要逐个检查这个顶点和光源之间的顶点，验证是否由顶点挡住了光源的光线，如图 9.3，很明显光源的光线不能直接照到 A，所以 A 在阴影区内。如果一个点在阴影区内，则它的亮度就直接等于环境光的亮度 Ie。否则它的亮度由图 9.2 导出的算法决定。

最后我们还需要把计算得到的亮度图和纹理调制到一起，本文采用最简单的方式，就是把亮度值（0-1.0）乘纹理图上对应点的各个颜色的分量即可。

第十章：景深处理

通常使用计算机渲染得到的图像都非常的犀利，不管远的还是近的的物体看上去都是很清晰。但是事实并不是这样的，离观察者远的东西看上去会显得模糊一些，不是处于两只眼睛观察的焦点上的物体也同样会模糊一些。

实现第二现象的方法在[参考文献 24]的 4.2 和 9.2 节由详细介绍，而且实现这种效果也不是十分的必要，对渲染速度的影响也很大（需要对场景渲染两次）。这里就不作介绍。至于第一种现象，最简单有效的方法就是使用雾。关于雾的详细内容在几乎所有的 OpenGL,Direct3D 的书籍上都有介绍。

雾的计算方程通常有三种，本文推荐采用的公式为 $f = e^{-(density \times z)}$。使用了雾效果的另外一个好处就是可以把远处的（位于视见体外部的）三角形裁剪掉也不至于产生“跳出”现象，因为远处的景物已经完全成为雾的颜色。它能提供一种无限远视野的假象。下图是使用了雾效果和没有使用雾效果的图例。

图 10.1 左图为开启了雾效果，右图为关闭了雾效果。图片来自本文的 Demo

第十一章：运动模糊

运动模糊一种动态的效果,它是由于摄影机的底片的曝光需要一定时间，如果在曝光过程种场景发生改变，就会在底片上产生模糊效果，这种效果称为运动模糊。运动模糊现象在电影、摄影上非常的常见。

运动模糊在运动的场景中非常重要,你会发现如果缺少了运动模糊会带来严重的失真，看一下没有采用运动模糊的计算机动画，你会发现物体快速移动时，画面缺乏连贯性和真实感。在前面提到过的模拟驾驶游戏 Need for Speed 中，地面在赛车高速运动的时候就会变的模糊，给人一种扑面而来的感觉。

图 11.1 运动模糊示意图。图片来自[参考文献 19]

实现运动模糊的方法在[参考文献 19]里有详细的介绍,在这篇文章里作者采用了时间过采样的方式来实现运动模糊，这是一种精确的运动模糊的实现方式，通常这种方法要使用“累积缓存”（Accumulation Buffer）来实现，关于累积缓存的使用方法在 OpenGL,Direct3D 的书籍中均可以找到。一般情况下，我们只要累积 3 到 5 次就可以达到比较理想的运动效果了，但是这种有一个很大的缺点就是严重降低运行速度，当我们进行 n 次累积的时候，运行速度就是没有进行运动模糊时的 1/n。这样的代价在一个游戏程序里是不能被接受的。

作为一种近似的解决方案，我们可以把先前的渲染结果保存起来，然后按一定的比例削弱以后混合到当前的渲染结果中去。假设当前的渲染结果为 P1，先前的帧为 P2,最后得到的当前帧为 P。混合的公式如下：$P=(1-f)\times P1+f\times P2$。其中 f 为模糊因子，f 越大，运动模糊越明显。这种方法只需要渲染一次场景，对运行速度影响不是很大。下面给出这种方法在 OpenGL 下的实现步骤：

1．　首先建立一个足够大的纹理，

2. 渲染场景到帧缓冲区中。
3. 把保存先前帧内容的纹理设置为当前纹理。
4. 绘制一个和视口一样大的矩形，把这个矩形按上面给出的公式和步骤 3 中渲染结果混合在一起。
5. 把混合结果拷贝到纹理对象中（glCopySubTexImage）。
6. 重复步骤 2-5 绘制下一帧。

由于有些 OpenGL 的驱动不支持直接渲染到纹理的操作，因此用 OpenGL 实现起来比较麻烦而且速度受到的影响也比较大一些。在 Direct3D 中可以通过直接改变渲染目标来把场景渲染到纹理中，在速度上受到的影响要小一些，在实现上也要简单一些，但是基本原理还是相同的，这里不再赘述。下图是一个使用了这种方法的例子。

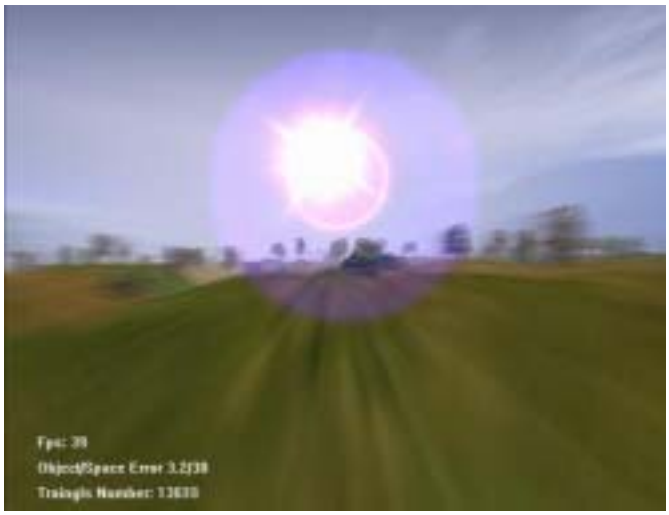
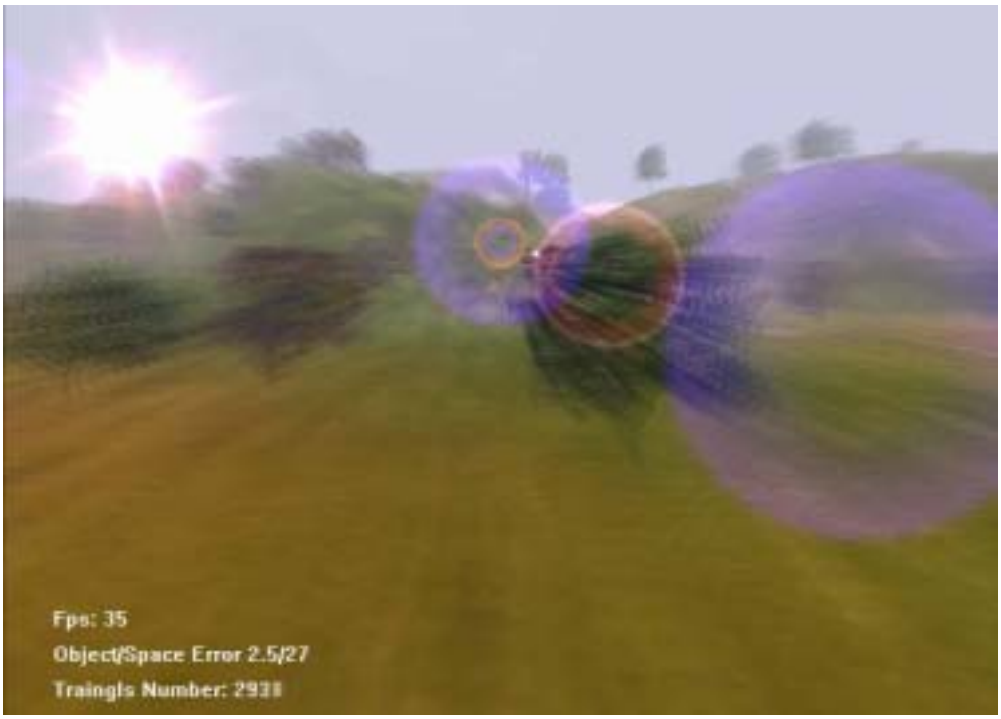


图 11.2 运动模糊。图片来自本文的 Demo

第四部分：结论和展望

本文已经基本上实现了一个室外 3D 游戏引擎的渲染核心部分。虽然提供的算法不是十分的精确，但是速度非常的快，用本文算法实现的 Demo 可以在当前流行的个人电脑上非常流畅的运行。加上本文第三部分的技术后，画面质量得到很大幅度的提升。下图为本文的演示程序的屏幕截图。所有的效果都已经开启。



但是急待解决的问题还很多，主要有以下几个：

1. **内存管理** 本文采用的内存数据结构虽然在一定的程度上考虑了内存缺页的问题，但是效果不是很好。由于地图是作为一个二维数组加载的，采用的是线性结构。数据的冗余比较大。可以考虑采用链式的数据结构来改善。
2. **几何形变。**
3. **无限重复地图** 一个地图再大也有一定的限度，当前许多的主流游戏采用的都是地图循环的方式，即沿着一个方向走到地图的边缘以后会返回到开始的地方，观察者永远也走不到地图的尽头。
4. **水面的模拟** 室外的场景中水的存在是很正常的，比如说湖面，小河等。水面应该倒影四周的场景，绘制倒影一般借助于模板缓冲区绘制两次场景，但是这对于室外大场景的开销非常大而且实现也很困难。我们需要一种更加有效的模拟水面的方法。

5. **各种天气效果** 本文只是简单的实现了天空体，真实的自然界中还有很多现象，比如下雨，闪电，下雪，刮风等。加上这些天气效果会使场景看上去更加的多变。

第五部分：附录

A 地形数据的生成

生成地形数据的方法主要有两种：

一是使用使称为 Perlin Noise 的随机函数来生成高低起伏不平地形。Perlin Noise 用来生成地形特别有效。

所谓的 Perlin Noise 其实就是把一些不同频率的伪随机数按照一定的规则把它叠加起来，不同频率的成分表示不同变化程度的地形细节。如：频率低的成分（幅度相对也大一些）可能表示山脉的起伏，而频率高的成分（幅度相对小一些）则可能表示地面的凹凸程度。关于这种方法的更加详细介绍请参见**[参考文献 14]**。

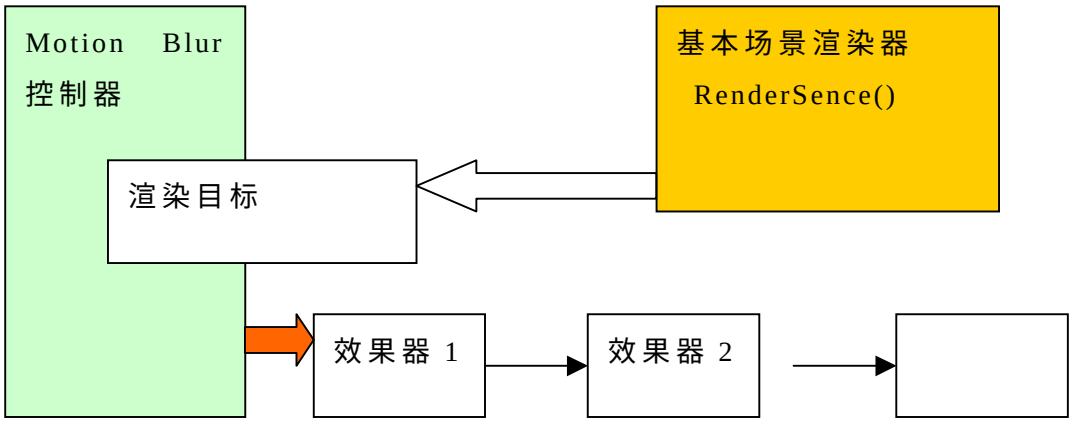
第二种方法是分形，通常也叫 Diamond－Square 或者是 Diamond Fractal 算法。这种方法则在用来生成几座山的时比较有效。关于这种算法可以在 www.fractal3d.com 上找到详细的介绍。

B: Demo 程序设计架构

引擎的基本结构为面向对象的模块化的组合。但是考虑到其他的因素，有许多非面向对象的成分在里面，

非面向对象的因素主要是用来构成渲染框架和初始化这些对象的。主要为 InitSence.cpp 中的文件，以及其他的 Win32 应用程序框架、配置文件参数的应用，以及大多数的全局参数。

本 Demo 的渲染流程基本如下：



Motion Blur 控制器 其实起到了一个渲染器的职责。它负责控制一个真正的场景渲染器 RenderSence 的渲染目标。它在需要的时候可以控制 RenderSence 把

场景渲染到一个纹理里，然后做进一步的后期处理。Motion Blur 控制器还是一个效果链的链首，所有的效果器都可以向 Motion Blur 控制器申请一个场景的渲染纹理。这样这些效果器就可以对渲染结果做进一步的处理。以下的效果器会用到场景的渲染纹理：Motion Blur、碎片效果、过度曝光效果等。

本文的所有效果器都采用一致的架构，详细内容见 CEffect 类,CEffect 类为所有效果器的基类。

本程序的场景对象有以下：

对象	用处
CTerrain	表示一个地形。管理其中的高度数据
CLOD	本文地形渲染的核心类。它是一个由 **CTerrain** 控制的类。该类主要用来渲染 **CTerrain** 的数据。
CSkyBox	表示一个天空体
CWaters	水平面。这个类管理着 **CWaterPlan** 类，把所有相同的纹理的 **CWaterPlan** 集合起来管理。
CBillBoardMgr	公告板集合，管理所有相同纹理的 **CBillBoard** 类。减少绘制树木等的时候纹理切换带来的开销
CLensFlar	不用说了。是镜头光晕的效果。

图形系统对象

对象	用处
CCamera.	摄影机类的基类接口
CViewerCamer	一个角度是固定的，类似于人的摄影机。它能抬头，转向。但是它的向上方向是不变的。
CFreeCamer	一个完全自由的摄影机类。
COpenGL	提供 OpenGL 的包装类。其中 包括字体的绘制，投影变换的设定等。它和摄影机是有点相关的。
CTextureMgr	自动管理所有的纹理对象。能创

	建一个纹理，并在程序结束以前销毁所有的纹理对象。
CFrustum	裁剪用的平头视见体，可以裁剪图元。即用于 View-Frustum 的裁剪。
DIBTEXDATA	纹理的数据对象。我们可以在加载了纹理数据后对其进行修改。该对象其实为一结构体。但是和纹理有关。特将其放在图形系统对象一表中。其实它为一个内存管理对象。
CMATH	处理所有的数学运算。包括向量的点积和旋转在内。而且还对快速三角函数的支持（查表法）
CGamaControl	控制图象画面的 gama 亮度。

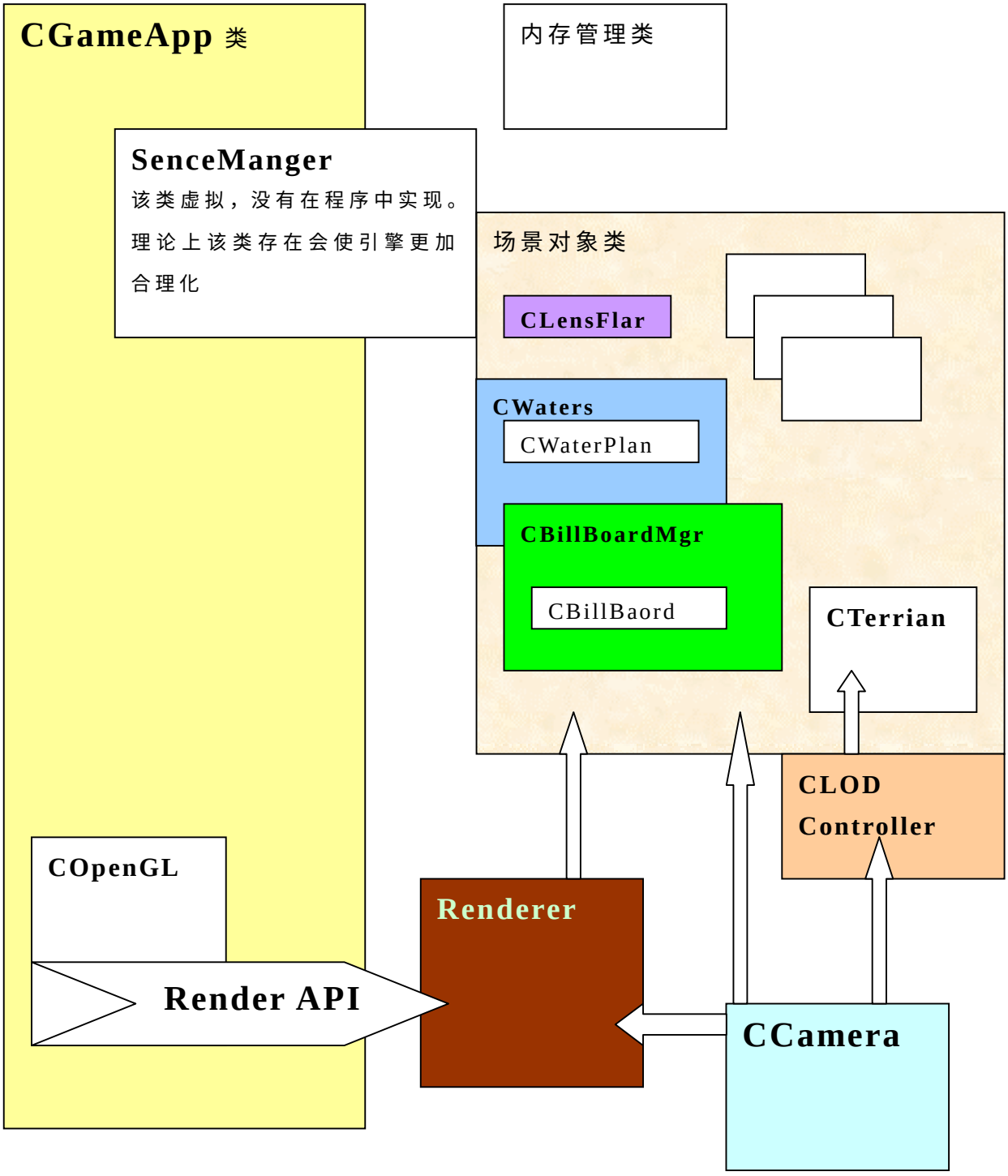
程序架构类

对象	用处
CConsole	就是程序刚刚开始运行时候出来的那个控制台。感谢 Quke 的创意
CConfigure	管理整个程序的配置，负责存取所有的配置信息
CGameApp	应用程序类。负责响应消息。即对整个 Win32 应用的行为负责。
CInput	处理所有对场景控制的输入。这些输入必需得到实时的响应。而不可以异步的响应。比如视角控制的输入等。其他的可以异步的响应由 CgameApp 处理。
CTimer	定时器。

内存管理类

主要为：CpageArray。由于该类，目前还是不是很完善，造成我们的内存管理还不是很有效。

下图为程序的基本架构层次图：



本文程序的地图操作函数，纹理操作函数，以及支持纹理操作的函数都被封装到三个 dll 里。这三个 dll 为 map.dll,image.dll 和 texture.dll。如果你下载了的 Demo 无法编译的话，请寻找这三个库的运行库和连接库以及它们对应的头文件。

Demo 程序的其它细节就不再仔细分析。

C 参考文献和资源

参考文献

[1]. XiaoHong Bao and Renato Pajarola *LOD-based Clustering Techniques for Optimizing Large-scale Terrain Storage and Visualization"* Department of Information & Computer Science University of California .Irvine.

[2]. Stefan Röttger , Wolfgang Heidrich and Philipp Slusalleck,Hans-Peter Seidel *Real-Time Generation of Continuous Levels of Detail for Height Fields* University Erlangen-Nürnberg

[3] P Lindstrom and V.Pascucci *Visualization of Large Terrains Made Easy* Lawrence Livermore National Laboratory August 7, 2001

[4] P.Lindstrom and V.Pascucci *Terrain Simplification Simplified: A General Framework for View-Dependent Out-of-Core* Lawrence Livermore National Laboratory May 8, 2002

[5] P.Lindstrom , David Koller , William Ribarsky, Larry F. Hodges, Nick Faust and Gregory A. Turner *Real-Time, Continuous Level of Detail Rendering of Height Fields* In SIGGRAPH 96 Conference Proceedings, pp. 109-118, Aug 1996.

[6] Reinhard Klein and Andreas Schilling *Efficient Multi-resolution Models for progressive Terrain Rendering*

[7] Mark Duchaineau, Murray Wolinski, David E. Sigeti, Mark C. Miller, Charles Aldrich and Mark B. Mineev-Weinstein. *ROAMing Terrain: Real-time, Optimally Adapting Meshes*. Proceedings of the Conference on Visualization '97, pp. 81-88, Oct 1997

[8] Thatcher Ulrich *Continuous LOD Terrain Meshing Using Adaptive Quad-trees* www. Gamasutra.com

[9] Luis Sempé, *Terrain Rendering using Heightmaps* March 2002, http://www.sphergames.com/

[10] Renato Pajarola, ETH Zürich *Large Scale Terrain Visualization Using The Restricted Quadtree Triangulation*

[11] 王宏武 董士海 《一个视点相关的动态多分辨率地形模型》 北京大学计算机图形研究所

[12] Hugues Hoppe *Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering* Microsoft Research

[13] Louis Castle, Jonathan Lanier and James McNeill *Real-time continuous level of detail (LOD) for PCs and Consoles* Technical Presentation GDC 2000

[14] Hugo *Perlin Noise*

[15] 中国游戏开发者网络 《BillBoard 技术详解》

[16] 中国游戏开发者网络 《BillBoard》

[17] 中国游戏开发者网络 《基于视锥包含的不可见物体消除算法》

[18] 中国游戏开发者网络 《OpenGL的渲染成纹理技术》(OpenGL Render-to-Texture)

[19] 中国游戏开发者网络 《运动模糊》(Motion Blur 原文见Hugo的主页)

[20] Michael Tanczos *The Art of Modeling Lens Flares* http://www.gamedev.net/

[21] Alan Gordie *Lens Flare Tutorial* http://www.gamedev.net/

[22]Jackie Neider ,Tom Davis and Mason Woo 《OpenGL编程权威指南》(OpenGL Programming Guide) OpenGL ARB Addsion-Weslly

[23]《OpenGL 超级宝典》人民邮电出版社

[24] Tom McReynolds *Programming with OpenGL: Advanced Rendering* SGI SIGGRAPH `97 Course

[25] *Microsoft DirectX SDK Document*

[26] *Mesa Lib 4.03 Source Code*

[27] *Microsoft MSDN –OpenGL section*

[28] *OpenGL Specification* (version 1.1 ,version 1.2,version 1.3 version 1.4)

[29]《计算机图形学算法基础》机械工业出版社

[30] 孙家广等 《计算机图形学》 清华大学出版社

[31] 清华大学远程教育 《计算机图形学》 第4章 第7节 《实时真实感图形学技术》

[32] Bryan Turner《Realtime Dynamic LOD Terrain Render With ROAM》

[33] Luis R. Sempé, *Sky Demo* http://www.spheregames.com/

资源

NeHe's OpenGL Tutorials Nehe.gamedev.net

FlipCode www.flipcode.com

GameTutorials: http://www.gametutorials.net/

nVidia: Developer.nvidia.com

OpenGL: www.opengl.org/

Mesa:　www.mesa3d.org

DirectX: www.microsoft.com/directX/

GameDev: www.gamedev.net

中国游戏开发者：Mays.soage.com

中国游戏资源网：www.gameres.com