摘 要

随着中国数字娱乐产业的发展，中国网络游戏也保持着强劲的增长势头，网络游戏的兴盛，是数字娱乐文化发展的一个必然环节，因此 3D 游戏引擎技术成为计算机图形学研究的热点问题。游戏及虚拟现实等模拟系统都具有感知性、沉浸性、真实性等重要特征，随着计算机硬件水平的不断提高，游戏场景的真实性成为衡量系统性能的一个重要标准。

电子科技大学数字媒体研究所研究的 “网络游戏公共技术平台关键技术研究”和 “支持数字媒体内容创作的集成环境”两个国家 863 项目，就是对数字媒体技术和游戏引擎技术的一种尝试。结合以上项目，本文研究了在游戏引擎中真实感特效方面的一些关键问题：高动态范围、景深模拟及运动模糊，并在已有的引擎基础上开发了三个特效插件。本文对课题的研究主要集中在以下三个方面。

高动态范围技术，是逐渐开始流行的实时绘制技术中的一个热门研究领域，其技术出发点就是让计算机能够显示更接近于现实照片的画面质量。本文从光照模型着手，在 GPU 上实现了对 Phong 模型的扩展；接着探讨了高动态范围技术，详细介绍了高动态范围的绘制技术和实现流程，在我们的游戏引擎中以插件形式实现了基于 GPU 的高动态范围光照系统，增强了视觉真实感。

景深是人眼视觉系统中成像的重要特征。本文从光照效果的成像机制，对透镜成像模型与针孔成像模型进行研究，在此基础上提出一种在计算机三维场景成像中实时地模拟出景深效果的算法。算法利用了 MRT（Multiple Render Targets）技术及 GPU 的可编程性，对清晰图像和模糊图像进行融合，实现了实时景深模拟，增强了用户的沉浸感。

运动模糊是虚拟环境和 3D 用户界面真实感的表现之一。本文继续研究光照效果的成像机制，在计算机上模拟真实相机对运动物体的拍摄产生的运动模糊效果。为了提高执行效率，我们将计算转移到了 GPU，实现了对运动模糊的实时模拟，并以一个匀速直线运动物体为例，展示了运动模糊系统的模拟效果。

在本文的最后，我们将运动模糊与高动态范围光照系统和景深模拟结合，综合展示了三个特效插件的集成运行效果。

关键词： 可编程图形硬件，高动态范围，景深模拟，运动模糊

ABSTRACT

With the development of digital entertainment industry in China, China's online games have also maintained a strong momentum of growth. The rise of online games is an ievitable link of the evelopment of digital entertainment. So, in recent years, 3D game engine technology becomes a hot issue in computer graphics. The main features of both 3D game and virtual reality are multi-sensory, immersion, and realistic. With the development of computer hardware, the realistic of game becomes an important criterion of game performance.

The 863 projects "Massive Multiplayer Online Game development platform" and "Digital Media Creative Integration Environment", developed by Digital Media Institute of UESTC, are examples of trying to research on digital media technique and game engine technique. With the projects metioned above, this dissertation do research on the realistic effect in game engine such as: high dynamic range, depth of field, and moiton blur. This dissertation focuses on the study in following aspects.

First, high dynamic range. This dissertation starts with illumination model, and implement an extention of Phong model, and discusses High Dynamic Range, which is a hot research field. Furhermor, the dizzertation introduces the rendering techniques and implementation flow of High Dynamic Range. We realize the high dynamic range lighting system on GPU as a plugin of our game engine, and enhance the virtual effect of scene.

Second, depth of field, which is an important characteristics of human visual system. Research on the imaging mechanism of lighting effect, this dissertation proposes an efficient and real time algorithm for the simulation of camera imaging in the three-dimensional scene based on both ideal and real camera models. The algorithm takes extensive use of the Multiple Render Targets(MRT) Technology and Graphics Processing Unit(GPU) for programmable performance. We output the image of the three-dimensional scene and store as a texture, and also output each pixel's depth and blurriness information and also store as texture. At last, we filter the image,

calculate the size of Circle of Confusion(CoC) for each pixel by the blurriness, and use Circle of Confusion(CoC) to blend between the original(clear) image and blurred image.

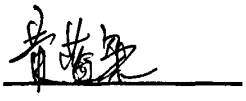
Third, motion blur, which is one of the realistic representation of virtual enviroment. This dissertation continues do research on the imaging mechanism of lighting effect, simulates the blur effect of motion object shoted by the real camera. In order to improve efficiency, we transfer the calculation from CPU to GPU.

At last, we combine the simulation of high dynamic range lighting, depth of field, and motion blur together.

Keywords: GPU, High Dynamic Range, Depth of Field, Motion Blur

独 创 性 声 明

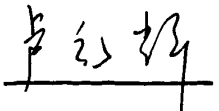
本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得电子科技大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

签名：  日期：2008 年 2 月 25 日

关于论文使用授权的说明

本学位论文作者完全了解电子科技大学有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权电子科技大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后应遵守此规定）

签名：  导师签名： 

日期： 2008 年 2 月 25 日

第一章 绪论

1.1 课题背景

本文是基于高动态范围的计算机特效的模拟研究，是以国家 863 项目“网络游戏公共技术平台关键技术研究”（No.2005AA114080）和国家 863 项目“支持数字媒体内容创作的集成环境”（No.2006AA01Z335）为依托研究开发的。前者是大型实时网络三维游戏引擎设计，整合网络、图形、物理、AI 和游戏逻辑、音效等功能模块，并开发以场景编辑器为主的数字内容创作工具，最终实现了多通道大视景的大型室外场景实时模拟。后者是研究构建数字媒体产品开发的创作工具集和通用引擎；开发支持大规模网络协作式内容创作的关键技术，高效的对象造型技术，支持多种媒体融合的建模技术与创作，以及支持各种计算机特效的集成创作工具。

1.2 课题研究意义

真实感图形绘制技术是计算机图形学的一个重要领域，随着计算机游戏产业和虚拟现实技术的发展，在游戏软件、飞行仿真和战场模拟等具有很强实时交互仿真系统中，对产生逼真的图像提出了更高的要求。图形的真实感往往需要实现若干特效，包括景深处理、运动模糊的模拟等，这反映了计算机生成的图形与客观世界的相似程度，在虚拟环境中直接影响着场景的“沉浸感”。在近三十年的时间里，随着硬件技术和计算机图形学的高速发展，真实感图形绘制技术取得了举世瞩目的成就，并已广泛应用于计算机仿真、计算机动画、虚拟现实、科学可视化等众多领域。

高动态范围技术（HDR，“High-Dynamic Range”），是一种逐渐开始流行的显示技术，其技术出发点就是让计算机能够显示更接近于现实照片的画面质量。游戏成为了目前在民用领域最多应用 HDR 技术最多的。从游戏的发展史以及未来的发展状况来看，随着科学技术的发展，游戏越来越趋向于真实化，玩家不但要求游戏中的每一种物体都有真实的外观，真实的物理特性，同时还要有真实的光照和阴影效果。在过去的十几年中，基于图像的绘制[1]技术得到了更进一步的成

熟和普遍的应用。在此扩展了的一个更具挑战性的问题，即基于图像的光照[2]研究，并逐渐受到人们的关注。在过去的基于图像的绘制技术中，有一个共同的假设，即场景本身是静态的，换句话说就是场景的光照条件固定不变，绘制过程只是通过改变视点位置或视线方向。接着，基于图像的光照研究突破了场景静态的限制，用户不仅可以改变视点，还可以改变场景，然后产生更丰富的光照效果。这方面的研究有[3, 4, 5]，其中最引人关注的是高动态范围图像（High Dynamci Range Image）[6, 7, 8, 13]在基于图像的光照中的应用。因此，HDR 技术在游戏中的应用必将成为评价一款游戏是否优秀的标准之一。

景深（DOF，depth of field）是人眼视觉系统中成像的重要特征。人眼对现实世界成像时，自动调节焦距以适应不同的取景距离，眼睛注视的物体便处于聚焦平面（focus plane）上，因此清晰成像于视网膜；而处于聚焦平面之外的物体，成像便模糊不清。透镜（瞳孔）的焦距、直径、以及物距共同决定了物体成像时的模糊程度。然而，目前的虚拟现实系统中，几乎都未引入景深效果，整个场景成像后都是清晰的，这样，整个场景则显得不够真实、自然，并且缺少景深所带的深度暗示[9]。景深大量存在于摄影和具有大深度的照片中。加入景深效果，有助于立体照片的合成以及缓解虚拟现实系统中常有的眼睛疲劳，增强场景的真实感、沉浸感。

运动模糊（Motion Blur）是虚拟环境和 3D 用户界面真实感的表现之一。快速物体的运动模糊在游戏、训练模拟等计算机图形学的实时应用中有着重要的地位，它是一种表述运动物体真实感以及增强用户沉浸感的一种方法。运动模糊描述了非无限小的相机快门速度带来的效果。当物体运动速度比快门速度快、快门还未关闭时，物体在胶片上的成像会发生移动，因此最终相片上物体显得半透明模糊。减缓物体速度或者提高相机快门速度都可以减少模糊，但是可能会产生易碎的、静止的图片。运动模糊大量存在于摄影和运动照片中。例如，摄影师会有意的将运动模糊作为一种技巧来表达运动的物体。运动相片的照相机，由于它的快门速度相对慢，快速运动的物体在电影每一张胶片上都成像模糊。由于运动模糊是胶片摄影的内在属性，因此拥有运动模糊的图像更具有真实感。

随着数字娱乐的快速发展，计算机图形处理器（GPU）以大大超过摩尔定律的速度高速发展，极大地提高了计算机图形处理的速度和图形质量，并促进了计算机图形相关应用领域的发展。自从 1999 年由 NVIDIA 首次引入 GPU 的可编程这一概念开始，GPU 的架构和功能都发生了巨大的改变，特别是其 3D 渲染流水线，经历了固定功能到强大的可编程功能管线的革命性演变。而 GPU 从其一开始

就出于特殊的应用（渲染），因此具有高度的并行性。并且 GPU 的运算能力每 6 个月就提升一倍，18 个月就经历一次架构的改变，其变化速度甚至超过了 CPU 的演变。虚拟现实、电影特效和计算机游戏的强大需求推动着工业界不断提升硬件的性能，设计出新的硬件体系架构，发挥硬件加速的优势。在保证一定真实感的前提下提高计算效率，得到更加真实和精致的渲染效果是我们追求的目标。因此，为了将高动态范围技术、景深模拟技术、运动模糊模拟技术在实时应用中使用，我们都会利用计算机图形处理器（GPU）来进行计算。

计算机游戏技术、现代计算机动画技术、虚拟现实系统都离不开对场景真实感的模拟，对真实感渲染的深入研究综合体现了当代计算机图形学的发展水平，具有重要的理论和实践意义。

1.3 本文工作

本文对游戏引擎中的光照特效技术进行了系统深入地研究。论文的主要工作包括下面这些（主要在 4、5、6 章进行介绍）：

- 首先从光照模型、光照计算和场景亮度范围的角度实现了具有真实感的高动态范围光照系统，并对这种方法进行了实验和分析；
- 然后从光照效果真实感和透镜成像原理的角度进行研究，提出了一种利用 GPU 加速的实时景深模拟算法；
- 最后同样从光照效果真实感和快门时间的角度进行研究，对运动物体成像模糊进行了实时的模拟。

最后基于所研究的方法，分别实现了实时高动态范围光照插件、实时景深模拟插件、实时运动模糊模拟插件，并对其分别进行了详细分析，三个插件可以结合使用，并最终整合到游戏引擎中。

概括起来，本文主要的创新点和研究成果如下：

讨论了各种主要的光照模型，根据我们具体的需求，对 Phong 模型进行了改进，并在 GPU 上进行了实时实现；实现了高动态范围光照模型，用实验说明高动态范围能产生很好的效果和性能；实现了基于 GPU 的实时景深模拟，提出了一种实时模拟算法；实现了基于 GPU 的运动模糊模拟，将运动模糊的模拟转移到了 GPU，释放了 CPU 时间，满足了实时性的要求；最终将高动态范围光照、景深模拟、运动模糊三者结合应用，对目前越来越追求高清画质和具有真实感的游戏玩家来说，是一个非常有意义的补充。

1.4 论文的章节安排

全文共分七章，对游戏引擎中的特效技术进行详细的分析与研究，本文作者的工作主要集中在第四、五、六章，章节安排如下：

第一章是全文的绪论。主要介绍本文的课题背景和研究意义，指出论文作者的主要工作、创新点和章节安排。

第二章是背景知识介绍。介绍了标准三维编程接口（OpenGL 和 Direct3D）和基础图形引擎（OGRE）；介绍了常用的光照模型；可编程硬件的发展过程；并对本文的重点工作内容（高动态范围、景深、运动模糊）的相关知识进行了详细的介绍。

第三章，介绍了项目中整个游戏引擎的架构，并指出了高动态范围光照插件、景深模拟插件及运动模糊模拟插件在此系统中的位置。

第四章，详细介绍了高动态范围的实时光照系统的实现。介绍了实现高动态范围光照的算法、系统的设计目标、数据结构和实现流程，并逐步详细介绍了高动态范围在 GPU 上的实现，最后对实现结果进行了分析。

第五章，详细介绍了实时景深模拟系统的实现。在这一章，我们先从单个景深模拟模块着手，介绍了实时景深模拟的算法、系统的设计目标、数据结构和在 GPU 上的实现流程，最后我们给出了实验结果和渲染截图。

第六章，详细介绍了实时运动模糊模拟系统的实现。这一章结构与第五章类似，我们先从单个运动模糊模拟模块着手，介绍了实时模拟运动模糊的算法和在 GPU 上的实现流程，然后再将运动模糊和景深模拟、高动态范围结合，实现了基于高动态范围光照的实时景深和运动模糊模拟，最后我们给出了实验结果和渲染截图。

第六章，对全文做出系统全面的总结，并对今后需要进一步深入研究的方向做了展望。

第二章 背景知识介绍

2.1 三维编程接口及基础图形引擎介绍

3D API 是架设在 3D 图形应用程序和 3D 图形加速卡之间用于沟通的桥梁。对于三维图形应用开发者来说，可供选择的三维(3D)图形编程接口(API)太丰富了，多达 50 多种。如此丰富的选择同时也为开发者出了一道难题，即如何选择适合自己应用的 API。对于具有丰富软件开发经验的编程者来说，以往选择 API 的经验可以帮助他们迅速做出正确的选择。开发的应用要面向什么平台，是否支持跨平台的工作，是否支持客户机/服务器模式，是否支持面向对象的开发等等，这些都是选择 API 的一般准则。总的说来，著名的大公司和在图形应用方面获得广泛认可的公司所提供的三维 API 引擎是大多数应用开发首先考虑的。如 SGI 公司的 OpenGL，Apple 公司的 Quick-Draw 3D (QD3D), 以及 Microsoft 公司的 Direct3D 等。对于我们来说，重要的是在开发工作开始之前必须详细了解这些 API 的功能和优缺点，以便作出恰当的抉择。

（一）OpenGL

OpenGL（“Open Graphics Library”）是图形硬件的软件接口。OpenGL 包括大约 250 个不同的函数，程序员可以使用这些函数设定要绘制的物体和操作，来制作交互的三维应用程序。OpenGL 是专业图形处理，科学计算等高端应用领域的标准图形库。它的主要竞争对手是微软的 Direct3D。OpenGL 曾长期处于技术上的领先地位,但近年来 Direct3D 也迎头赶上。目前这两种图形 API 在性能上可说是旗鼓相当。不过 OpenGL 支持众多的操作系统，而 Direct3D 只在 Windows 平台可用。因此 OpenGL 仍然广受瞩目。如魔兽 3、CS、Doom、Quake 等游戏采用了 OpenGL 进行渲染。

OpenGL 版本比较：

1、OpenGL 的版本区别（在 opengl 官方文档中有详细说明），针对 OpenGL 不同版本的升级是主要是扩展指令集。

(1) OpenGL1.1

1995 年，SGI 推出了更为完善的 OpenGL 1.1 版本。OpenGL 1.1 的性能比 1.0 版本提高甚多。其中包括改进打印机支持，在增强元文件中包含 OpenGL 的调用，

顶点数组的新特性，提高顶点位置、法线、颜色、色彩指数、纹理坐标、多边形边缘标识的传输速度，引入了新的纹理特性等等。

(2) OpenGL1.3

2001 年 8 月，ARB 发布 OpenGL 1.3 规范，它增加了立方纹理贴图、纹理环境、多重采样、纹理框架压缩等扩展指令，但是改进程度非常有限。

(3) OpenGL1.4

2002 年 7 月，ARB 正式发布 OpenGL 1.4，它也只加入了深度纹理 / 阴影纹理、顶点设计框架、自动纹理贴图等简单的功能。

(4) OpenGL1.5

2003 年的 7 月，ARB 公布 OpenGL 1.5 规范。OpenGL 1.5 内包含 ARB 制定的“正式扩展规格绘制语言”(OpenGL Shading Language v1.0)，该语言用于着色对象、顶点着色、片断着色等扩展功能，同时也将作为下一代 OpenGL 2.0 版本的内核。OpenGL 1.5 的变化还增加了顶点缓冲对象（可提高透视性能）、非乘方纹理(可提高纹理内存的使用效率)以及阴影功能、隐蔽查询功能等等。其主要内容包括：

- 顶点 Buffer Object：进行顶点配列方式可以提高透视性能；
- Shadow 功能：增加用来比较 Shadow 映射的函数；
- 隐蔽查询(QUERY)：为提高 Curling 性能采用非同步隐蔽测试；
- 非乘方纹理(Texture)：提高 mipmap 等纹理内存的使用效率；
- OpenGL Shading Language v.1.0：用于着色(shader)对象、顶点着色以及片断着色技术(fragment shader)的扩展功能。

(5) OpenGL2.0

OpenGL 1.0 推出后的相当长的一段时间里，OpenGL 唯一做的只是增加了一些扩展指令集，这些扩展指令是一些绘图功能，像是 ClearCoat、Multisample、视频及绘图的整合工具(某些是通过 OpenML 的努力而开发出来的，它本身属于 OpenGL ARB 扩展指令之一。

OpenGL 2.0 将在 OpenGL 1.3 基础上进行修改扩充、但它将有下面五个方面的重大改进：①复杂的核心被彻底精简；②完全的硬件可编程能力；③改进的内存管理机制、支持高级像素处理；④扩展至数字媒体领域，使之跨越高端图形和多媒体范畴；⑤支持嵌入式图形应用。

为了在获得强大功能的同时保持理想的兼容性，OpenGL 2.0 经历以下两个发展阶段：第一个阶段注重兼容能力和平滑过渡，为此，OpenGL 2.0 核心将在精简

后的 OpenGL 1.3 功能模块的基础上加上可完全兼容的新功能共同组成，这种做法在满足兼容性的同时，还可将原有 OpenGL 中数量众多、且相互纠缠不清的扩展指令进行彻底精简。第一阶段的任务只是为了过渡，而第二阶段才是 OpenGL 2.0 的真正成熟期。此时，ARB 将合成出一个“纯 OpenGL 2.0”内核，纯内核将包含更多新增加的“精简型 API 函数”，这些函数具有完全的可编程特性、结构简单高效、功能强大且应用灵活。除了完成这项任务外，ARB 组织还指导开发商抛弃繁琐的 OpenGL 1.X、转用更具弹性的“纯 OpenGL 2.0”。

2、OpenGL 扩展（OpenGL Extensions）

OpenGL 和 Direct3D 比较起来，最大的一个长处就是其扩展机制。硬件厂商开发出一个新功能，可以针对新功能开发 OpenGL 扩展，软件开发人员通过这个扩展就可以使用新的硬件功能。所以虽然显卡的发展速度比 OpenGL 版本更新速度快得多，但程序员仍然可以通过 OpenGL 使用最新的硬件功能。而 Direct3D 则没有扩展机制，硬件的新功能要等到微软发布新版 DirectX 后才可能支持。

OpenGL 扩展也不是没有缺点，正因为各个硬件厂商都可以开发自己的扩展，所以扩展的数目比较大，而且有点混乱，有些扩展实现的相同的功能，可因为是不同厂商开发的，接口却不一样，所以程序中为了实现这个功能，往往要为不同的显卡写不同的程序。这个问题在 OpenGL 2.0 出来后可能会得到解决，OpenGL 2.0 的一个目标就是统一扩展，减少扩展数目。

（二）DirectX

DirectX 是一组低级“应用程序编程接口 (API)”，可为 Windows 程序提供高性能的硬件加速多媒体支持。Windows 支持 DirectX 8.0，它能增强计算机的多媒体功能。使用 DirectX 可访问显卡与声卡的功能，从而使程序可提供逼真的三维 (3D) 图形与令人如醉如痴的音乐与声音效果。DirectX 使程序能够轻松确定计算机的硬件性能，然后设置与之匹配的程序参数。该程序使得多媒体软件程序能够在基于 Windows 的具有 DirectX 兼容硬件与驱动程序的计算机上运行，同时可确保多媒体程序能够充分利用高性能硬件。

DirectX 包含一组 API，通过它能访问高性能硬件的高级功能，如三维图形加速芯片和声卡。这些 API 控制低级功能（其中包括二维 (2D) 图形加速）、支持输入设备（如游戏杆、键盘和鼠标）并控制着混音及声音输出。构成 DirectX 的下列组件支持低级功能：Microsoft DirectDraw（2D 绘图）、Microsoft Direct3D（3D 绘图）、Microsoft DirectSound（声音相关）、Microsoft DirectMusic（MIDI 相关）、Microsoft DirectInput（输入相关）、Microsoft DirectShow（动画播放）、Microsoft

DirectPlay（网络相关）。

DirectDraw是DirectX家族中的元老，它为高速的2D渲染提供了良好的支持，由于其具备直接显存访问和位快传送的能力，使得2D图形的绘制速度相对GDI有了一个质的飞跃，其渲染速度甚至有上百倍的差距。如“红色警戒”和“Diablo”就是用DirectDraw进行开发的。

DirectDraw在DirectX3.0时就已经接近极致，但是随着PC图形技术的飞速发展，人们逐渐不再满足于2D的图形效果，而通过2D技术实现伪3D模拟又非常损失效率，这种需求直接导致了Direct3D的诞生，早期的Direct3D技术不甚完善，相对于2D技术还有一定的差距，直到图形加速卡支持硬件3D特效后，Direct3D才逐渐步入正轨，慢慢显示出它的性能优势来。下面让我们来回顾一下历史，看看Direct3D的发展过程：

1、DirectX 5.0：D3D日益强大

微软似乎没有发布4.0版本的DirectX，DirectX3.0发布后没多久发布了DirectX5.0。尽管5.0与3.0时间间隔不长，但它的意义可不简单。DirectX5.0的D3D效果可以与当时的OpenGL平分秋色。首次引入了雾化的支持，让3D游戏更有空间真实感，更能让玩家体验到真实的三维三维游戏环境；除此以外在游戏系统的兼容性方面作了很大改善。

2、DirectX 6：D3D权威出现

在DirectX5.0发布不久第二代3D加速卡问世了，这一代3D加速卡借助DirectX6.0的技术争得不可开交。主要代表显卡是Nvidia的Riva TNT，并连的VooDoo2，VooDoo3。到了这个时代，市场格局已经很清晰，是NVidia与3DFX的斗争。DirectX6中的Direct3D添加了如下新特性：

- 几何形体的灵活顶点格式定义；
- 几何形体的顶点缓冲存储；
- 支持多纹理渲染；
- 自动纹理管理；
- 可选深度缓冲（使用Z Buffer或W Buffer）；
- 通过凹凸环境贴图（BUMPENVMAP）为反光面和水波特效提供逐像素的渲染和贴图能力。

3、DirectX 7：D3D权威确立。DirectX7中的Direct3D添加了以下新特性：

- 硬件坐标转换和灯光（T&L）支持（DirectX7最大的特色）；
- 立方体表面的环境贴图；

- 几何渲染；
- 改进的纹理渲染；
- 自动纹理坐标生成、纹理转换、投影纹理和任意面裁剪；
- D3DX 实用库；
- 支持 Intel MMX 架构、Intel 单指令多数据流（SIMD）、SSE®和 AMD® 3DNow®架构。

4、DirectX 8：D3D 的疯狂

DirectX 的版本到了 8.0 的时候，虽然它依然保持着向前的兼容性，但是它的结构发生了巨大的变化，3D 图形处理技术逐渐统一在 Vertex Shader 和 Pixel Shader。Vertex Shader 被用来描述和修饰 3D 物体的几何形状，同时也用来控制光亮和阴影；Pixel Shader 则用来操纵物体表面的色彩和外观。Direct3D 和 DirectDraw 合二为一，DirectX 家族诞生了一个新的成员-----DirectGraphics。同时也增添了很多令人激动的特性：

- 完全集成 DirectDraw 与 Direct3D
- 简化程序初始化过程并提高数据分配和管理的性能，这将减少内存消耗。同时，集成后的图形应用编程接口（API）允许并行的顶点输入流以达到更加灵活的渲染；
- 可编程顶点处理语言；

  允许用户编写定制的着色器，如变形和渐变动画，矩阵调色板蒙皮，用户定义的光照模型，一般环境映射，可编程几何体或者任何其他开发者定义的算法。
- 可编程像素处理语言；

  允许用户编写定制的硬件着色器，例如通用纹理组合公式，逐像素光照（凹凸贴图），适用于实现照片（真实）级镜面效果的逐像素环境贴图或者任何其他开发者定义的算法。
- 支持多重采样渲染；

  允许全场景反走样和多重采样效果，例如运动模糊及景深（镜头的聚焦效果）。
- 点精灵；

  允许高性能的粒子系统渲染，例如火花、爆炸、雨、雪等等。
- 3-D 空间纹理；

  允许范围衰减，实现逐像素级光照及空间大气效果，甚至是更复杂的

几何图形应用。

- 支持高维图元；

  对来自主要的3-D创作工具的3-D内容，增强其外观并简化内容映射。

- 高级技术；

  包含了用于输出Direct3D蒙皮网格的三维内容创建工具插件，使用了Direct3D 多种不同技术，多分辨率层次细节（LOD）几何，还有高维表面数据。

- 索引顶点混合；

  扩展了几何混合的支持，允许通过指定使用一个矩阵索引把矩阵应用于顶点混合处理。

- 扩充了Direct3DX实用库；

  包含了大量的新函数。Direct3DX实用库是一个位于Direct3D之上的辅助层，用于简化 3-D 图形开发者的常规工作。它包括了一个蒙皮库，支持对网格的操作，还有组装顶点与像素着色器的功能。

5、DirectX 9：让人耳目一新

DirectX每一次升级都会有重大的内核结构改变，它会给我们带来巨大的视觉冲击。DirectX7核心的T&L引擎到DirectX9干脆被抛弃了。DirectX 9具有多项全新功能特征：

- Vertex Shader 2.0和Pixel Shader 2.0

  在DirectX9中，Vertex Shader和Pixel Shader的版本升级到了2.0，它们都支持64或甚至128位浮点色彩精度。浮点色彩在动态和精度上的增加给图像质量带来质的飞跃，这样在DirectX9中用户可以轻易实现电影级别般逼真的效果。

  Vertex Shader 2.0引入了流程控制，增加了条件跳转、循环和子程序。Vertex程序现在最多可以由1024条指令组成（之前只能用128条指令），增加的指令带来更加复杂和强大的表现，新的操作如正弦、余弦及其他强大的函数运算大大简化了代码的编写，并且能够表现更加复杂的效果。

  强大的可编程 Pixel Shader 是实现具有电影质量级别效果的真正精华所在。Pixel Shader 2.0可以支持高级程序语言和汇编语言，开发人员还可以将其汇编代码嵌入较高级的程序语言中。前一版的Pixel Shader语言被限制为只能使用最多6个材质和28条指令，而2.0版则将这一上限提升至最大16材质和160条指令，也新增了很多强化的运算和操作。

此外各种 Shader 工具的大力协助，如 nVidia 的 Cg 语言和 ATI 的 RenderMonkey，使得实时图形渲染的质量飞速提高，nVidia 和 ATI 官方的 Demo 就足以说明这一点。DirectX9 中也改进了部分接口定义，Vertex Shader 和 Pixel Shader 分别单独提供了一个接口，而且提供了常见图形的绘制接口，如线段的绘制等，这进一步减轻了开发人员的负担。

- 浮点型色彩和 32 位帧缓冲格式

目前大多数色彩表示法（如 RGB）用 8 位整数表示红、绿、蓝色，也许对于显示来说这已经足够了，但在运算中似乎还远不够。由于整数没有小数部分，因此当它们经过 Pixel Shader 极其复杂的数学运算后，就会产生较大的误差，这可能导致色彩明显失真。而 DirectX9 支持数种浮点色彩格式，使得其精度有了很大提高。同时 DirectX9 支持精确到每像素 32 位的帧缓冲格式，能够表现出 4 倍于目前亮度等级的数量，这使得图像看上去更加清晰和自然。

除此以外，DirectX9 还提供位移贴图（displacement maps）以及改进的设备模拟等特性的支持，并且毫无意外地，DirectX9 SDK 中也进一步扩展了 Direct3DX 实用库，提高了开发效率。

6、DirectX10：微软一统 3D 规格的王牌

DirectX10 将完全放弃 GPU 当中的固定渲染模式，并且支持 GPU 行为的完全自由化，即 GPU 不在明确划分像素着色和顶点着色单元，并且支持多种任务，如 2D/3D/视频加速等等任务的自由分配。DirectX10 将加入 Shader4.0 技术，并首度实现 RayTracing 光线追踪，将位移贴图 Displacement 作为标准之一。

（三）OGRE

上面我们介绍了目前最流行的两个基础三维编程接口：OpenGL 和 Direct3D，它们都属于底层的应用编程接口，下面我们将介绍一个上层的图形引擎 OGRE。

OGRE（Object-oriented Graphics Rendering Engine）是一款授权模式为 LGPL 的图形引擎。OGRE（面向对象的图形渲染引擎）是用 C++开发的面向对象且使用灵活的 3D 引擎。它的目的是让开发者能更方便和直接地开发基于 3D 硬件设备的应用程序或游戏。引擎中的类库对更底层的系统库（如：Direct3D 和 OpenGL）的全部使用细节进行了抽象，并提供了基于现实世界对象的接口和其它类，使开发者独立于渲染的 API 上。

由于 OGRE 开放式的架构设计和灵活的授权模式，吸引了世界各地众多的开放人员和用户。并通过 wiki 等形式积累了大量的学习资源。因此是目前最为成功的开源图形引擎。值得强调的是 OGRE 并不是完整的游戏引擎，其设计目标就是

专注于架构一个强大的图形渲染引擎。但是，另一方面，它预留了接口，可以很方便地与其他的引擎结合。比如物理引擎、网络引擎等。目前已经有比较成功的商业游戏采用了 OGRE 作为图形引擎，比如 Deck13 工作室的冒险游戏 Ankh。

下面我们对 OGRE 具体特性作一个分析介绍：

1、效率特性

- 简单、易用的面向对象接口设计使用户能更容易地渲染3D 场景，并使用户的实现产品独立于渲染API（如Direct3D/OpenGL/Glide 等等）；
- 可扩展的程序框架（framework）使用户能更快的编写出更好的程序；
- 为了节省用户的宝贵时间，OGRE 会自动处理常见的需求，如渲染状态管理，hierarchical culling，半透物体排序等等；
- 清晰、整洁的设计加上全面的文档支持。

2、平台和 3D API 支持

- 支持Direct3D 和OpenGL；
- 支持Windows 平台，用Visual C++ 6（或Visual C++.Net）和STLport 来编译；
- 支持Linux 平台，用gcc 3+（或gcc 2.9x）和STLport 来编译；
- 材质/Shader 支持；
- 支持从PNG、JPEG 或TGA 这几种文件中加载纹理；自动产生MipMap；自动调整纹理大小以满足硬件需求；
- 支持可程序控制的纹理坐标生成（如环境帖图）和转换（平移、扭曲、旋转）；
- 材质可以拥有足够多的纹理层，每层纹理支持各种渲染特效，支持动画纹理；
- 自动应用多通道渲染和多纹理，从而大幅度提高渲染质量；
- 支持透明物体和其它场景级别的渲染特效；
- 通过脚本语言可以不用重新编译就设置和更改高级的材质属性。

3、网格 Meshes

- 高效的网格数据格式；
- 提供插件支持从Milkshape3D 导出OGRE 本身的.mesh 和.skeleton 文件格式；
- 支持骨骼动画（可渲染多个动画的组合）；
- 支持用贝赛尔样条实现的曲面

4、场景特性

- 拥有高效率和高度可配置性的资源管理器，并且支持多种场景类型。使用系统默认的场景组织方法，或通过亲自编写插件使用自己的场景组织方法；
- 通过绑定体（如绑定盒）实现的场景体系视锥拣选；
- 提供的BspSceneManager 插件是快速的室内渲染器，它支持加载Quake3关卡和shader 脚本分析；
- 优秀的场景组织体系；场景结点支持物体的附属（attach），并带动附属物体一起运动，实现了类似于关节的运动继承体系。

5、特效

- 粒子系统包括可以通过编写插件来扩展的粒子发射器（emitter）和粒子特效影响器（affector）。通过脚本语言可以不用重新编译就设置和更改粒子属性。支持并自动管理粒子池，从而提升粒子系统的性能；
- 支持天空盒、天空面和天空圆顶，使用非常简单；
- 支持公告板，以实现特效；
- 自动管理透明物体（系统自动帮用户设置渲染顺序和深度缓冲）。

6、其它特性

- 资源管理和文档加载（ZIP、PK3）；
- 支持高效的插件体系结构，它允许用户不重新编译就扩展引擎的功能；
- 运用“Controllers”用户可以方便地改变一个数值。例如动态改变一个带防护罩的飞船的颜色值；
- 调试用的内存管理器负责检查内存溢出。

上面我们对对 OGRE 具体特性作了分析介绍，接着我们对 OGRE 的核心对象作一个分析介绍：

下面的这张 UML 图表展现了 OGRE 中的核心对象以及它们之间是如何互相联系的，这张图表并没有描述所有的类，它只把几个关键的类彼此间是如何关联的作了一个大概的描述。

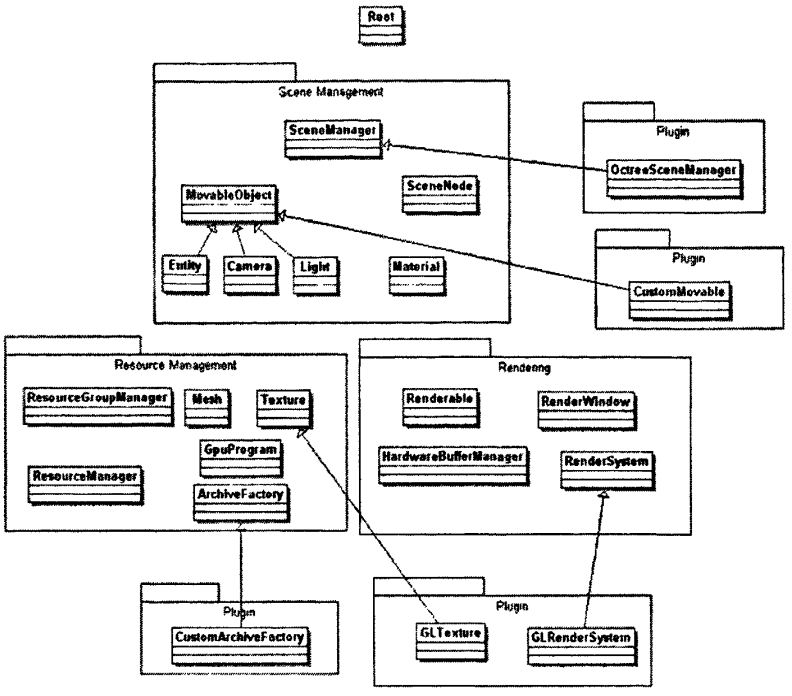


图 2-1 OGRE 子系统及核心类图

在这张图表的最顶部是根（Root）对象。这是用户进入 OGRE 系统的“入口”，根（Root）对象用来创建 OGRE 系统中的所有的基础元素，比如：场景管理器（Scene Managers），绘制系统（Rendering Systems），绘制窗口（Render Windows）和插件加载器（Loading Plugins）。根（Root）对象是 OGRE 系统中一切的开始，它会提供给用户实现具体功能的核心对象，根（Root）对象更像是 OGRE 系统中的一个组织者。OGRE 的其余的类大致分为下列三种角色：

1、场景管理

场景管理描述了场景中有哪些内容，以及它们是如何组织在一起的。场景管理提供了与现实情况贴近的极其自然的接口供用户调用，也就是说，用户在调用这些接口时不会使用到“设置这几个对象的绘制状态，然后画三个多边形”这样的语言，而是“我想要把这几个事物放在这几个地方，它们的材质是这样的，我可以从这个场景中看到它们”这种贴近现实情况的语言。

2、资源管理

不管地形、纹理还是字体等一切对象，绘制它们都需要不同的资源。如何加载、重用、卸载这些资源是非常重要的，因此，有专门的一批类来完成这些事情。

3、渲染

最后，屏幕上的所有一切是如何被渲染出来的，这涉及到渲染管线、指定的渲染系统、被传送到管道的所有的缓存、渲染状态等 API 对象等底层对象。场景管理器子系统中的类将这所有的一切组织起来，提供了更加简单的高级别的接口以供调用。

在图表的边缘上还有一些插件(Plugins)。OGRE 在设计时就考虑了其扩展性，而使用插件则是一种最为常见的扩展方式。许多 OGRE 中的类能被继承及不断扩展，用户可以自定义一个改变了场景的组织方式，添加了新的绘制系统应用（绘制系统应用指的是 Direct3D、OpenGL）的场景管理器（SceneManager）的插件，或者提供加载资源的其它方式（比如从网络加载或从数据库加载）的插件。虽然插件只完成 OGRE 系统中某个方面的工作，但插件这种方式可以覆盖到系统的方方面面。从这个意义上来说，OGRE 决不仅仅只是一个针对某类问题而提供的封闭型的解决方案，它在许多方面都可以方便地进行扩展，以满足用户的需要。

2.2 图形处理器介绍

20 世纪六、七十年代，受硬件条件的限制，图形显示器只是计算机输出的一种工具。限于硬件发展水平，人们只是纯粹从软件实现的角度来考虑图形用户界面的规范问题。图形用户界面国际标准 GKS(GKS3D)，PHIGS 就是其中的典型代表。

20 世纪 80 年代初期，出现 GE(Geometry Engine)为标志的图形处理器。GE 芯片的出现使得计算机图形学的发展进入图形处理器引导其发展的年代。GE 的核心是四位向量的浮点运算。它可由一个寄存器定制码定制出不同功能，分别用于图形渲染流水线中，实现矩阵，裁剪，投影等运算。12 个这样的 GE 单元可以完整地实现三维图形流水线的功能。芯片设计者 James Clark 以此为核心技术建立的 SGI 公司，基于 SGI 图形处理器功能的图形界面 GL 及其后的 OpenGL，成为图形用户界面事实上的工业标准。

20 世纪 80 年代和 90 年代，GE 及其图形处理器功能不断增强和完善，使得图形处理功能逐渐从 CPU 向 GPU 转移。现代图形处理的流水线主要功能分为顺序处理的两个部分：第一部分对图元实施几何变化以及对图元属性进行处理(含部分光照计算)；第二部分则是扫描转换进行光栅化以后完成一系列的图形绘制处理，包含各种光照效果和合成、纹理映射、遮挡处理、反混淆处理等。

20 世纪 90 年代，NVIDIA 进入个人电脑 3D 市场，并于 1999 年推出具有标

志意义的图形处理器——GeForce 256，第一次在图形芯片上实现了 3D 几何变换和光照计算。此后 GPU 进入高速发展时期，平均每隔 6 个月就出现性能翻番的新的 GPU。

从 SGI 的 GE 到 NVIDIA 的 GeForce，GPU 经历了 20 年，芯片的线宽从 3μm 缩小到 90nm(2007 年的 GeForce 8800 集成了 6.81 亿晶体管)，集成电路的逻辑设计能力提高几千倍，但处理器数据通道接口带宽仅提高十几倍。同时对图形处理器计算能力的需求不断增长，出现了可编程的图形处理器，以 NVIDIA 和 ATI 为代表的 GPU 技术正是适应这种趋势。

而到目前为止，GPU 已经过了七代的发展，每一代都拥有比前一代更强的性能和更完善的可编程架构。

第一代 GPU(到 1998 为止)包括 NVIDIA 的 TNT2，ATI 的 Rage 和 3dfx 的 Voodoo3。这些 GPU 拥有硬件三角形处理引擎，能处理具有 1 或 2 个纹理的像素，能够大大提高 CPU 处理 3D 图形的速度。但这一代图形硬件没有硬件 T&L 引擎，更多只是起到 3D 加速的作用，而且没有被冠以“GPU”的名字。

第二代 GPU(1999-2000)包括 NVIDIA 的 Geforce256 和 Geforce2，ATI 的 Radeon7500，S3 的 Savage3D。它们将 T&L 功能从 CPU 分离出来，实现了高速的顶点变换。相应的图形 API 即 OpenGL 和 DirectX7 都开始支持硬件顶点变换功能。这一代 GPU 的可配置性得到了加强，但不具备真正的可编程能力。

第三代 GPU(2001)包括 NVIDIA 的 Geforce3 和 Geforce4 Ti，微软的 Xbox，及 ATI 的 Radeon8500。这一代 GPU 首次引入了可编程性，即顶点级操作的可操作性，允许应用程序调用一组自定义指令序列来处理顶点数据，并可以将图形硬件的流水线作为流处理器来解释。也正是这个时候，基于 GPU 的通用计算开始出现。但是片段操作阶段仍然不具备可编程架构，只提供了更多的配置选项。开发人员可以利用 DirectX8 以及 OpenGL 扩展(ARB-vertex-program，NV-texture-shader 和 NV-register-combiner)来开发简单的顶点及片段着色程序。

第四代 GPU(2003)包括 NVIDIA 的 GeforceFX(具有 CineFX 架构)，ATI 的 Radeon9700。相比上一代 GPU，它们的象素级和顶点级操作的可编程性得到了大大的扩展，可以包含上千条指令，访问纹理的方式更为灵活，可以用做索引查找。最重要的是具备了对浮点格式的纹理的支持，不在限制在[0，1]范围内，从而可以做任意数组，这对于通用计算而言是一个重要突破。DirectX9 和各种 OpenGL 扩展(ARB-vertex-program、ARB-fragment-program、NV-vertex-program2、NV-fragment-program)可以帮助开发人员利用这种特性来完成原本只能在 GPU 上

进行的复杂顶点像素操作；Cg 语言等其他高级语言在这一代 GPU 开始得到应用。

第五代 GPU(2004)主要以 NVIDIA GeForce6800 为代表。NVIDIA GeForce 6800 集成了 2 亿 2 千 2 百万晶体管，具有超标量的 16 条管线架构。功能相对以前更加丰富、灵活。顶点程序可以直接访问纹理，支持动态分支；象素着色器开始支持分支操作，包括循环和子函数调用，TMU 支持 64 位浮点纹理的过滤和混合，ROP(象素输出单元)支持 MRT(多目标渲染)等。

第六代 GPU(2006)主要以 NVIDIA GeForce 7800 为代表。GPU 内建的 CineFX 4.0 引擎，做了许多架构上的改良，提高许多常见可视化运算作业的速度，支持更复杂的着色效果，且仍能维持最高的影像质量。新的图形硬件具备着新的特征，概括起来有如下几方面（以 Nvidia GeForce 7800 为例）：

1) 在顶点级和像素级提供了灵活的可编程特性，其 CineFX 4.0 引擎同时支持 DirectX 9.0c、OpenGL2.0 及 Shader Model3.0，提供 8 个定点着色器单元和 24 个象素着色器单元；

2) 在顶点级和像素级运算上都支持 IEEE32 位浮点运算，还提供 64 为的纹理过滤和混合功能及 128 位高精度色彩，能更精确地分析各种材质的表现；

3) 支持多遍绘制的操作，这样避免了多次的 CPU 与 GPU 之间的数据交换；

4) 支持绘制到纹理的功能(Render-to-Texture/pbuffer)，从而避免将计算结果拷贝到纹理这一比较费时的过程；

5) 支持依赖纹理功能，以方便数据的索引访问，可以将纹理作为内存来使用。

NVIDIA CineFX 4.0 引擎将许多突破性绘图技术融入顶点着色器、像素着色器、以及材质引擎的核心。加快了三角模型元素设定、像素着色器的关键数学元素、以及材质处理等方面的作业，最新引擎让 3D 绘图研发者能达到更上一层楼效能与视觉质量。

第七代 GPU(2007)主要以 NVIDIA GeForce 8800 为代表。在 2007 年初 NVIDIA 发布的 GeForce 8800 正在引领下一代 GPU 的疾速风暴，G80 核心拥有空前规模的 6.81 亿晶体管，是 G71 的 2.5 倍，而且依然采用 90nm 工艺制造，再加上高频率的 12 颗显存，使 8800GTX 拥有超强的性能。

NVIDIA 的工程师为 GeForce 8800 GPU 体系架构定下了许多前瞻性的设计目标，而排在最前面的四个设计目标则是：

- 比当代 GPU 提供显著的性能提升。
- 显著的画面品质提升。
- 提供强大的 GPU 物理和浮点计算能力。

- 与微软协力定义下一代 DirectX（即 DirectX 10）标准，为 GPU 流水线加入新的增强特性，例如 geometry shader（几何着色器）、stream out（渲染流输出）。

GeForce 8800 GTX 采用了统一着色器架构，内部有 128 个标量 32 位浮点精度流处理器，每个流处理器每个周期可以执行一条乘加指令。

这 128 个流处理器会被 GigaThread 线程处理器根据工作负荷，自动分配执行顶点着色器、几何着色器、像素着色器指令，线程调度是硬件执行完全自动化的，加上采用的是标量架构，不管是 DX7、DX8、DX9、DX10 还是 OpenGL，G80 的统一着色器都会达到 100%的运作效率。

2.2.1 图形流水线——从固定管线到可编程管线

所谓流水线就是把一个大的任务按照可能的次序切成若干份连续的小任务，实现分工处理。我们这里说图形流水线其实就是指 3D 图形渲染的步骤或者过程，首先是建模，对顶点或者多边形做位移变换以及打光，把三角形的坐标转换成屏幕对应的坐标即 Setup 引擎所作光栅化转换。然后就是对这些对应屏幕像素的点做各种计算和贴图操作，接着把渲染好的像素输出到色彩缓存里，透过 RAMDAC 或者是 TMDS 把像素转换成适当的电压信号发送到显示器，最后就是大家在屏幕或者其他显示设备上看到的栩栩如生的画面。

简单的说，3D 加速卡的主要功能就是协助 CPU，负责将内存中的矢量图像数据（顶点集合）进行变换、光照计算、裁剪等操作，最后经过光栅化将图像呈现给人眼。这个过程就叫做渲染，以 D3D 为例，它把整个渲染分为 9 个步骤，9 个步骤的组合，就叫做流水线，或者叫管线。

D3D 的渲染管线（Rendering Pipeline）：局部坐标变换 -> 世界坐标变换 -> 观察坐标变换->背面消除->光照->裁剪->投影->视口计算->光栅化。无论是固定管线还是可编程管线都要经过这 9 个步骤。

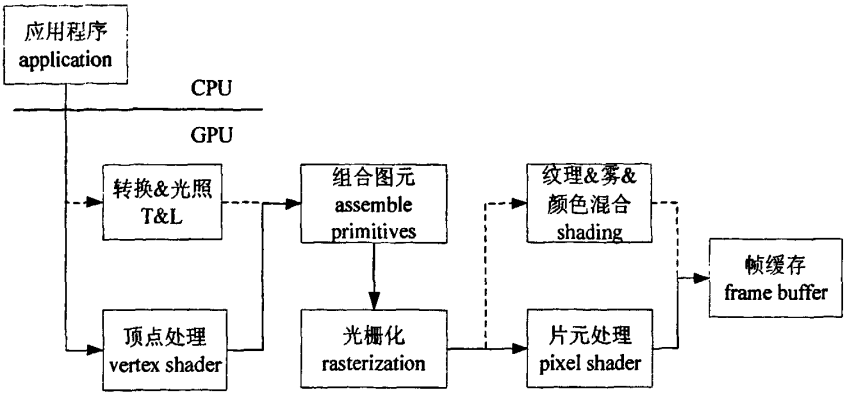


图 2-2 固定管线与可编程管线图

基于光栅化（Rasterization）图形管线如图 2-2 所示，图中虚线部分数据流向代表固定图形管线。应用程序通过点、线、多边形等几何图元构建出物理模型或可视化的数据结构，这些模型最后表示成具有对象空间坐标、法向量、颜色、纹理坐标等属性的顶点集（Vertex）。对顶点进行坐标变换、光照计算后，图元装配和光栅化操作对几何图元进行纹理和颜色的插值，生成和窗口屏幕像素相对应的片元集（Fragment）。片元集具有窗口坐标、颜色、纹理坐标、雾化坐标等属性，每个片元经过纹理应用颜色混合以及雾化等操作后计算出片元最终颜色和深度。通过模板、深度、透明混合等测试的片元最终写入帧缓存中显示出来。

大致在 1998 年后期，以 Nvidia TNT2、ATI Rage 和 3DFX Voodoo3 为代表的图形芯片支持多纹理操作，在光栅化过程中完成多幅纹理的融合操作。1999 年后期，以 Nvidia GeForce256、GeForce 2 和 ATI Radeon 7500 为代表的图形芯片可以处理顶点的矩阵变换和进行光照计算。这一时期，图形特性的固化和硬件加速是图形处理器设计的方向，这一阶段的图形处理器特点是“固定图形管线（Fix Function Pipeline）”。

但很快就发现，照这样的方向发展下去，不断增加的图形新特性将会使得图形库和图形处理器设计不堪重负。同时，有限的特性终究束缚了图形应用开发者的手脚，不能满足人们对更真实的场景和更绚丽的动画效果的需求。2001 年底和 2002 年初，以 Nvidia 和 ATI 为代表的主流图形处理器厂商将硬件图形管线的流水线作为流处理器来解释，顶点级出现可编程性，片元级出现有限的可编程能力。这一重要的变革使图形芯片成为真正意义的图形处理器，这一时期的处理器有 Nvidia GeForce3、GeForce4 和 ATI Radeon 8500。2003 年 Nvidia GeForce FX 系列

和 ATI Radeon 9700/9800 最终通用化了顶点和片元级可编程能力。固定的图形管线中顶点的光照和坐标变换由顶点程序(vertex shader)取代，片元的颜色融合及纹理操作由片元程序(pixel shader)取代，固定图形管线演变为可编程图形管线，如图 2-2 上部实线所示。人们可以在顶点和片元两个级别上编程实现灵活的处理，满足不同的要求。顶点和片元程序在图形应用中统称为渲染程序（Shader）。以及后来的 GeForce 6x、7x 系列和相应的 ATI 5xx 核心的处理器都是沿着扩展图形可编程能力的方向发展，这一阶段图形处理器的特点是“可编程图形管线（Programable Pipleline）”。

可编程管线（Programable Pipleline），顾名思义，就是说管线中的某些环节是可以被控制的。人们可以通过对 GPU 中的着色器进行编程的方式，来控制、管理加速卡的渲染效果。着色器分为顶点着色器和像素着色器。顶点着色器是在进行坐标变换和光照计算时工作，像素着色器是在光栅化环节工作。人们对着色器进行自定义编程时，这个流水线就叫做可编程管线。

至此，GPU 的发展已经是基本抛离了依靠硬件固定单元实现诸如环境映射等复杂效果的阶段，现在这些效果都是由可编程的 shader 来完成。

随着通用可编程能力的提高，图形处理器迎来新的变革与发展，顶点处理和片元处理在统一的硬件处理单元上执行，体系结构更加通用化，如 Nvidia G8x 和 AMD/ATI R6xx。GeForce 8800 GTX 采用了统一着色器架构，它由由 8 组着色器矩阵组成，每个着色器矩阵内包含 16 个标量流处理器。在执行图形渲染的时候，同一时间内每个着色器矩阵可以同时运行 VS+PS、VS+GS 的指令，各个着色器矩阵可以在线程处理器的分配下执行包括不同类型指令的线程。例如在同一时间里：着色器矩阵 1 执行 VS+PS，着色器矩阵 2 执行 VS+GS，着色器矩阵 3 完全用来执行 VS，着色器矩阵 4 完全用来执行 PS，开发人员无须为 VS、PS 代码的轻重比例而过分操心。由图形处理器衍生出的协处理器，将会提升个人电脑的并行计算能力。这个阶段的图形处理器的特点是“数据流通用计算（GPGPU）”。

在过去的 10 年，图形渲染流水线上的工段依然都是：顶点-Setup-像素/纹理操作-光栅输出-内存操作，但是正如上面所说的，每一代新的图形芯片问世总会给这些工段注入新的活力。

2.2.2 Shader 语言

在历史上，图形硬件都是从非常低层上进行开发的。通过设置状态，例如贴

图合并(Texture-combining)模式，来设定固定功能的管线(Pipelines)。可编程管线（Programable Pipleline）的出现，程序员可以通过使用汇编语言层的编程接口来设置可编程管线。理论上来说，这些低层的编程语言提供了极大的灵活性。在实际应用中，它们在使用起来却是很痛苦并且在有效使用硬件上设置了极大的障碍。

用高级编程语言，而不是使用以前的低层编程语言，有以下几个优势：

- 当着色器(shader)被开发出来的时候，高级语言可以加速运行周期。对于着色器的最终测试就变成了“它看起来对吗？”。到最后，快速原型和修改着色器在高质量效果的快速开发上就变得至关重要。
- 编译器可以自动优化代码并执行低层任务，例如注册地址分配，那是很容易出错并且乏味的操作。
- 着色代码用高级语言编写更容易被阅读和理解。这样也允许通过修改以前编写的着色器，从而轻松的创建新的着色器。
- 用高级语言编写的着色器要比用汇编代码更能够适用于广泛的平台。

目前常用的高级着色语言有：CG(C for Graphics)、HLSL（与 D3D 结合使用）、GLSL（与 OpenGL 结合使用），为 GPU 编程特别设计的新的高级语言。这些高级着色语言提供了刚才提到的所有优点，经过增强和调整使它们可以轻松的编程并编译成为高度优化的 GPU 代码。它最终使程序员使用这个语言轻松的进行 GPU 编程并发挥了 GPU 固有的强大动力。

2.3 光照模型介绍

光照模型（Illumination Model）是计算机图形学中生成真实感图形的基础。光照模型根据物体的表面材质和光源特性，按照光学物理的有关定律，计算几何物体表面上任一点上的光亮度和色彩组成的数学计算公式。

根据对光学模型的研究方法，通常把光学模型分为两大类模型：经验模型和物理模型。经验模型是基于光照效果分析和实验数据总结得出的光照模型。早期的光照模型多为经验模型，如 Lambert 漫反射光照模型、Phong 镜面反射光照模型、Blinn 光照模型等等。物理模型是基于物理学中的热能辐射原理发展起来的更加符合光能物理属性的光照模型，如 Torrance-Sparrow 光照模型、Cook-Torrance 光照模型等。

另外，从所能计算的光照类型的完整性的角度出发，通常又将光照模型划分为局部光照模型（Local Illumination Model）和全局光照模型（Global Illumination

Model)。简单的说，局部光照模型只能计算直接光照下几何物体表面是上的每个像素点上的光亮度和色彩组成；而全局光照模型不仅具备局部光照光照模型的功能，还可以完整的计算出光反射所引起的间接照明作用。全局光照[10]用来指的一类模型，这类模型通过估算从某点 x 处反射的光线，并且考虑所有到达该点的照明从而绘制一个场景。也就是不仅要考虑直接从光源到达该点的照明，还要考虑可能穿过其他物体的来自光源（折射光、反射光）的间接照明。现在有两个主要的全局光照算法，辐射度算法和光线跟踪算法。而局部反射模型[11]是指在光照计算中只考虑直接照明，而不考虑可以导致阴影的与其他物体的相互作用和物体内部的反射。其中，最著名的局部反射模型就是 Phong 明暗处理模型[12]。

经验性光照模型多为局部关照模型，而物理性光照模型典型代表就是光线追踪算法所采用的 Whitted 光照模型和光能辐射算法中的辐射度光照模型。

下面我们将对光照模型的基本概念阐述：

1、光照明模型 （illumination model）

图形对象上某点处的光强度的物理描述；其包括光的反射、折射、透明度和阴影等。主要分为物理模型和经验模型。

2、面的明暗处理（surface rendering）

通过光照模型中的光强计算，确定场景中物体表面的所有投影像素点的光强，也称为面绘制。

3、环境光(ambient light)

在基本光照模型中，为场景设置的一个基准光亮度，用以简单模拟一种从不同物体表面所产生的反射光的统一照明，称为环境光(ambient light)或背景光。其特点如下：

- 是全局漫反射光照的一种近似；
- 场景中每个物体都有相同强度的入射环境光照；

环境光的计算如下：

- 每个面上的漫反射光为常数，与观察方向无关；
- 用参数 Ia 表示场景中入射环境光的大小，物体上某点处的环境光 I 的近似计算公式为：

$$I = K_a \bullet I_a$$

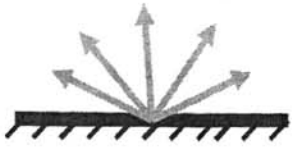


图 2-3 环境光示意图

其中 ka 是漫反射系数，与物体表面性质相关。

4、漫反射(diffuse reflection)

光源照射到物体表面产生的漫反射 Diffuse Reflection。设点光源入射光强为 Il，入射角为 θ，物体表面漫反射系数为 Kd （也称漫反射率 diffuse reflectivity ），漫反射光强的计算公式（兰伯特余弦定理）：

$$I_{l,diff} = k_d I_l \cos\theta \qquad \text{或} \qquad I_{l,diff} = k_d I_l (N \bullet L)$$

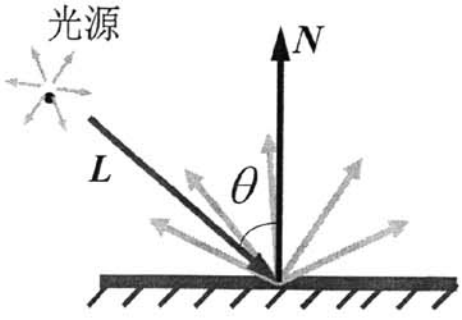


图 2-4 漫反射光计算示意图

漫反射有以下几个特征：

- 漫反射率 kd 与物体的表面性质相关；
- 为计算方便通常假设 kd 与波长及视点位置无关；
- θ 为入射光与表面法矢量之间的夹角，入射角越小，物体表面的受光越多；
- 仅当入射角 θ 在 0～90°之间，点光源才照亮表面，若 cosθ 为负值，则光源位于表面之后；
- 可将环境光与点光源所产的光强度合并，得到一个完整的漫反射表达式：$I = k_a I_a + k_d I_l \cos\theta$

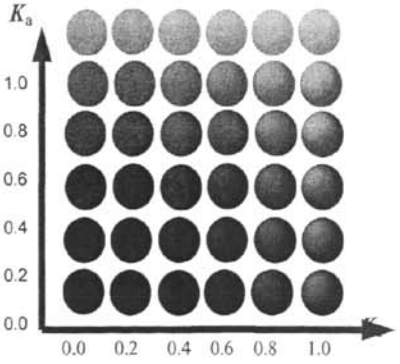


图 2-5 环境光和漫反射光

上图描述了在一个点光源下，光照模型中只有环境光和点光源漫反射的情况，图中小球的颜色代表着球的亮度。

5、镜面反射与 Phong 模型

(1) 镜面反射

光照下的光滑物体表面会在某个观察方向上产生高光，这种现象称为镜面反射（Specular Reflection）。当发生镜面反射时，在接近镜面反射角的一个汇聚区域内，入射光的全部或绝大多数成为反射光。

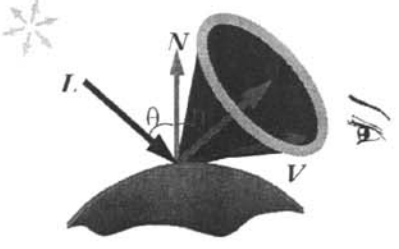


图 2-6 镜面反射计算示意图

对理想反射体（镜子），只有在 R 方向才能观察到反射光，即 $\varphi=0$；对非理想反射体，反射光分布在 R 周围的有限范围内。

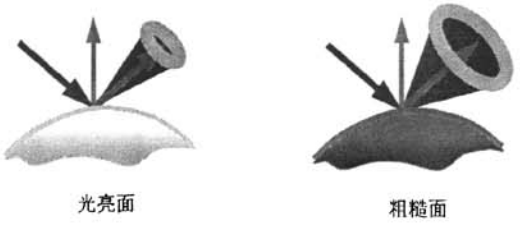


图 2-7 不同材质的镜面反射汇聚区域示意图

(2) Phong 模型

Phong 模型是计算镜面反射的经验模型。计算公式如下（参考图 2-6）：

$$I_{spec} = W(\theta) I_l \cos^{n_s} \varphi ;$$

Phong 模型具有如下特征：

- 镜面反射光强度与 cosφ 成正比， φ 介于 0～90°之间；
- 镜面反射参数 ns 与物体表面性质相关，光滑表面 ns 较大，粗糙表面的 ns 较小；
- 镜面反射系数 W(θ)与物体表面材质相关，并且是入射角 θ 的函数：透明材质如玻璃仅当 θ 角接近 90°时才有明显的镜面反射，当 θ＝0°时，约 4 %的入射光被玻璃表面反射；

对于许多不透明的材质， W(θ)几乎为常数，随 θ 角的变化很小，因此可以用一个镜面反射系数 ks 来代替。

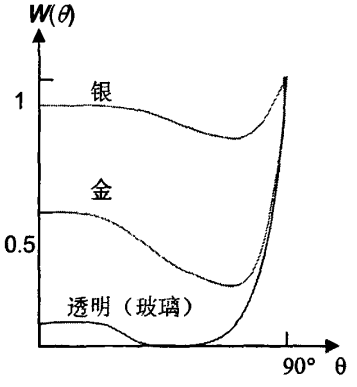


图 2-8 不同材质镜面反射系数 W(θ)与入射角 θ 的关系图

Phong 镜面反射模型的矢量形式为：

$$I_{spec} = k_s I_l (N \bullet H)^{n_s}, \text{ 其中 } H = (L+V)/|L+V|$$

H 为半角向量。若观察者离物体表面足够远，可以认为 L 和 V 都为常数，对于非平面，用 H 向量的 Phong 模型计算量小。

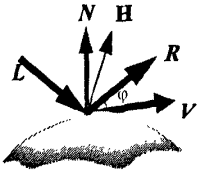


图 2-9 Phong 模型计算图

6、多光源

对单个点光源，包含漫反射和镜面反射的光照模型可表示为：

$$I = I_{amb} + I_{diff} + I_{spec} = k_a I_a + k_d I_l (N \bullet L) + k_s I_l (N \bullet H)^{n_s}$$

场景中设置多个点光源，物体表面的光照是各个光源产生的光照效果的叠加：

$$I = I_{amb} + I_{diff} + I_{spec} = k_a I_a + \sum_{i=1}^{n} I_i (k_d (N \bullet L_i) + k_s (N \bullet H_i)^{n_s})$$

7、光强衰弱

辐射光线从点光源出发在空气中传播时，其强度将按因子 1/d2 进行衰减，d 为光线经过的路程长度；一个常用的衰减函数为：

$$f(d) = \frac{1}{a_0 + a_1 d + a_2 d^2}$$

考虑光线衰减的基本光照模型计算公式为：

$$I = k_a I_a + \sum_{i=1}^{n} f(d_i) I_i [k_d (N \cdot L_i) + k_s (N \cdot H_i)^n]$$

8、颜色

用 RGB 各分量表示光源强度和物体表面颜色，并根据光照模型计算反射光线中的 RGB 分量；

方法一：将反射系数标识为三元向量，例如设置漫反射系数为(k_dR, k_dG, k_dB)。

$$I_B = k_a I_{aB} + \sum_{i=1}^{n} f(d_i) I_{iB} [k_{dB} (N \cdot L_i) + k_{sB} (N \cdot H_i)^n]$$

方法二：为每个表面定义漫反射和镜面反射的颜色向量(S_dR, S_dG, S_dB)和(SsR, SsG, SsB)，而将反射系数定义为单值常数。

$$I_B = k_a S_{dB} I_{aB} + \sum_{i=1}^{n} f(d_i) I_{iB} [k_{dB} S_{dB} (N \cdot L_i) + k_{sB} S_{sB} (N \cdot H_i)^n]$$

2.4 景深及运动模糊介绍

景深（DOF，depth of field）是人眼视觉系统中成像的重要特征。人眼对现实

世界成像时，自动调节焦距以适应不同的取景距离，眼睛注视的物体便处于聚焦平面（focus plane）上，因此清晰成像于视网膜；而处于聚焦平面之外的物体，成像便模糊不清。透镜（瞳孔）的焦距、直径、以及物距共同决定了物体成像时的模糊程度。然而，目前的虚拟现实系统中，几乎都未引入景深效果，整个场景成像后都是清晰的，这样，整个场景则显得不够真实、自然，并且缺少景深所带的深度暗示[9]。加入景深效果，有助于立体照片的合成以及缓解虚拟现实系统中常有的眼睛疲劳，增强场景的真实感、沉浸感。

快速物体的运动模糊是虚拟环境和 3D 用户界面真实感的表现之一。运动模糊在游戏、训练模拟等计算机图形学的实时应用中有着重要的地位，它是一种表述运动物体真实感以及增强用户沉浸感的一种方法。

运动模糊描述了非无限小的相机快门速度带来的效果。当物体运动速度比快门速度快、快门还未关闭时，物体在胶片上的成像会发生移动，因此最终相片上物体显得半透明模糊。减缓物体速度或者提高相机快门速度都可以减少模糊，但是可能会产生易碎的、静止的图片。

运动模糊大量存在于摄影和运动照片中。例如，摄影师会有意的将运动模糊作为一种技巧来表达运动的物体。运动相片的照相机，由于它的快门速度相对慢，快速运动的物体在电影每一张胶片上都成像模糊。由于运动模糊是胶片摄影的内在属性，因此拥有运动模糊的图像更具有真实感。

在三维场景中的应用中，运动模糊可以表现一个飞奔的运动物体，同时也将用户的注意力集中到运动的某个物体上；在二维场景、图像或单个帧缓存中，没有运动模糊的图像是不足以表述运动信息的，拥有运动模糊的图像可以表述出正在发生的运动或动作。因此运动模糊可以增强用户沉浸感。

2.5 高动态范围介绍

2.5.1 高动态范围的概念

近几年来，高动态范围（High Dynamic Range, HDR）技术在计算机图形学、虚拟现实等邻域开始有着越来越多的应用。HDR（high dynamic range）是与传统的 LDR（low dynamic range）相对的，LDR 是采用 8 位纹理格式（24/32 位颜色每像素），HDR 这种全新的模型将每个像素的 RGB 以及亮度值用实际物理参数或是线性函数来表示，参数不再限于整数，可以达到更大的范围和更高精确度。

HDR 本身是 High-Dynamic Range（高动态范围）的缩写，这是一个 CG 概念。计算机在表示图像的时候是用 8bit(256)级或 16bit(65536)级来区分图像的亮度的，但这区区几百或几万的数量根本无法再现真实自然的光照情况。而 HDR 文件是一种特殊图形文件格式，它的每一个像素除了普通的 RGB 信息外，还有该点的实际亮度信息。普通的图形文件每个像素只有 0～255 的灰度范围，这实际上是不够的。试想一下太阳的发光强度和一个纯黑的物体之间的灰度范围或者说亮度范围的差别，远远超过了 256 个级别。因此，一张普通的白天风景图片看上去白云和太阳可能都呈现是同样的灰度/亮度，都是纯白色，但实际上白云和太阳之间实际的亮度不可能一样，它们之间的亮度差别是巨大的。因此，普通的图形文件格式的表现是很不精确的，远远没有记录到现实世界的实际状况。但是 HDR 文件却能提供更多的亮度信息，让图像表现得更加逼真。

一幅图像亮度级的最大值和最小值之比被称为动态范围（Dynamic Range）。定义如下：

$$a = I_{\max} / I_{\min}$$

通过眼睛瞳孔的自动调节，从明亮的日光到星光，人眼能分辨物体的动态范围可以达到 100000000：1，即使在同一个适应场景内，不需调节，人眼也能分辨 10000：1 的亮度范围。然而，常规显示设备能重建的亮度动态范围仅仅是 100：1。

2.5.2 高动态范围图像

2.5.2.1 HDRI 与 HDRI 文件

HDRI 是 High-Dynamic Range (HDR) image 的缩写，是一种亮度范围非常广的图像，它比其它格式的图像有着更大亮度的数据贮存，而且它记录亮度的方式与传统的图片不同，不是用非线性的方式将亮度信息压缩到 8bit 或 16bit 的颜色空间内，而是用直接对应的方式记录亮度信息，它可以说记录了图片环境中的照明信息，因此我们可以使用这种图象来“照亮”场景。有很多 HDRI 文件是以全景图的形式提供的，我们也可以用它做环境背景来产生反射与折射。这里强调一下 HDRI 与全景图的区别，全景图指的是包含了 360 度范围场景的普通图像，可以是 JPG 格式、BMP 格式、TGA 格式等等，属于 Low-Dynamic Range Radiance Image，它并不带有光照信息。

HDRI 文件是一种文件，扩展名是 hdr 或 tif 格式，有足够的能力保存光照信息，但不一定是全景图。一张 HDR 图片，它记录了远远超出 256 个级别的实际

场景的亮度值，超出的部分在屏幕上是显示不出来的。可以这样想象：在 photoshop 里打开一张从室内往窗外外拍的图片，窗外的部分处在强烈的阳光下，曝光过度，呈现的是一片白色，没有多少细节。用户将毫无办法，调暗只会把白色变成灰色而已，并不会呈现更多的细节。但如果同一场景是由 HDR 纪录的话，如果减低曝光度，原来纯白的部分将会呈现更多的细节。如下图：

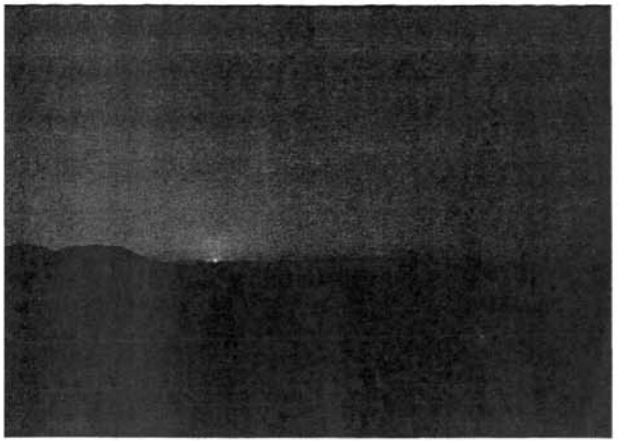


图 2-10 曝光度低的普通图片

图 2-11 曝光度高的普通图片

图 2-12 HDR 图片

2.5.2.2 高动态范围图像的获取

目前，对 HDR 的研究主要是围绕两个方向来进行的，首先是 HDR 图像的获取。HDR 图像的获取有两种途径：根据 LDR 的相机照出的相片，然后经过一定的处理得到 HDR 图像。另一种途径是，用专业的 HDR 相机照出 HDR 图像。

对于第一种途径的研究是从普通的图片恢复技术开始的。1997 年，Paul Debevec 等[13]人从数字图片中恢复了高动态范围的光照。只需选择图片上的一小部分的像素即可，并对感应函数做了优化。2003 年，Robertson 等[14]人在用多曝光的方法提高了图片的动态范围。这种方法是使用感应函数来计算 HDR 图像。用感应函数把一些不同曝光的照片，变成线性的图像序列，在通过缩放、权值函数，成为浮点的 HDR 图像。

第二种途径是直接用 HDR 照相机就可以得到 HDR 图像。现在已经有两种 HDR 照相机（IMS-CHIPS 和 LarsIII），还有一种高质量的低动态范围相机 Jenoptik C14。从最近在数字摄像技术方面的进展来看[15,16]，很可能未来的数码相机和数码摄像机会直接获取高动态范围图像。

2.5.3 高动态范围技术与游戏

在很多电影发烧友大聊 HDTV，感受高清画质的同时，不少游戏玩家也在津津乐道地谈论着游戏中的高清画质，那就是 HDR。随着越来越多的游戏问世，高质量的游戏画面成为玩家讨论的热门话题。目前，HDR 已经成为玩家评价一款游戏是否优秀的标准之一。从游戏的发展史以及未来的发展状况来看，随着科学技术的发展，游戏越来越趋向于真实化，玩家不但要求游戏中的每一种物体都有真

实的外观，真实的物理特性，同时还要有真实的光照和阴影效果。所以说游戏的操作和画面是同等重要的，因此，本小节将侧重于介绍 HDR 与游戏的结合。

2.5.3.1 HDR 与游戏——HDR 特效

HDR 在游戏中特指 HDR 特效。HDR 特效是通过 Shader Model 实现的图像渲染特效。HDR 的格式分为三个种类，即 FP16 HDR、FP24 HDR 和 FP32 HDR。FP16 HDR 代表 RGB 每个颜色通道采用 16 位浮点数表示；FP24 HDR 代表 RGB 每个颜色通道采用 24 位浮点数表示；FP32 HDR 代表 RGB 每个颜色通道采用 32 位浮点数表示。

要实现 HDR 特效，首先，游戏开发者要在游戏开发过程中，利用开发工具（就是游戏引擎）将实际场景用 HDRI 记录下来；也可直接用开发工具（比如 3D MAX 的某些特效插件）创造 HDRI 图像；其次，我们的显卡必须支持显示 HDR 特效，nVIDIA 的显卡必须是 GeForce 6 系列或更高，ATI 显卡至少是 Radeon 9550 或以上。HDR 特效最早是在 nVIDIA 的显卡实现的，但是 ATI 的 SMARTSHADER 技术也包含 HDR 技术，不过这种 HDR 从实现原理上与 nVIDIA 的是有区别的。具体的 HDR 特效实现将在第四章第三、四节进行详细介绍。

2.5.3.2 HDR 与 Bloom 效果的区别与关系

游戏中 HDR 和 Bloom 技术都统属于特效。从游戏表现出的画面效果来看，两者的差别不是很大，但是他们的技术成分就相差千里，具体如下：

第一，HDR 效果就是超亮的光照与超暗的黑暗的某种结合，这个效果是光照产生的，强度、颜色等方面是游戏程序可动态控制的；Bloom 效果则是物体本身发出的光照，仅仅是将光照范围调高到过饱和，是游戏程序无法动态控制的。

第二，Bloom 效果无需 HDR 就可以实现，但是 Bloom 效果是很受限的，它只支持 8 位 RGBA，而 HDR 最高支持到 32 位 RGBA。

第三，Bloom 效果的实现很简单，而且 Bloom 效果不受显卡的规格的限制，甚至可以在 TNT 显卡上实现 bloom 效果，当然这样做效果很差。然而 HDR，必须是 GeForce6XXX 以上的显卡才能够实现。

事实上，游戏开发者往往会将两种特效一起使用以达到一个最终的效果。两者的区别可以举一个最简单的例子：用户在游戏中，从黑暗的房间中走到太阳地中，用户眼前的景物会很刺眼，随后亮度会降低，完全就同现实中的情况一致。这就是 HDR 特效的威力。而 Bloom 效果实现的光照强度可能不会很真实，同时

也是不可变的。

2.6 本章小结

这一章，我们主要对本文将要描述的特效作详细的背景知识导引。首先，我们详细的介绍了三维编程接口 OpenGL 和 Direct3D 以及我们图形渲染引擎所采用的开源库 OGRE；接着，我们介绍了图形处理器的发展及 GPU 编程所采用的 Shader 语言；我们继续介绍了光照模型；最后，我们对高动态范围、景深和运动模糊的概念进行了介绍。

第三章 游戏引擎架构

3.1 游戏引擎设计与架构

本课题来源于电子科技大学数字媒体技术研究所的国家 863 项目“网络游戏公共技术平台关键技术研究”中“实时特效的渲染研究“部分。该项目完成了一个大型实时网络三维游戏引擎的设计，整合了网络、图形、物理、AI 和游戏逻辑、音效等功能模块，开发了以场景编辑器为主的数字内容创作工具，并且实现了多通道大视景的大型室外场景实时模拟。

这一节主要描述了我们的网络三维游戏引擎架构设计，它是基于多种开源引擎的高层引擎。网络三维游戏引擎是在 OGRE 图形引擎的基础上，结合 Newton 物理引擎、OpenAL 音效引擎、OpenSteer 行为引擎以及 RakNet 等开源引擎而建立的。具体的层次结构图如图 3-1 所示：

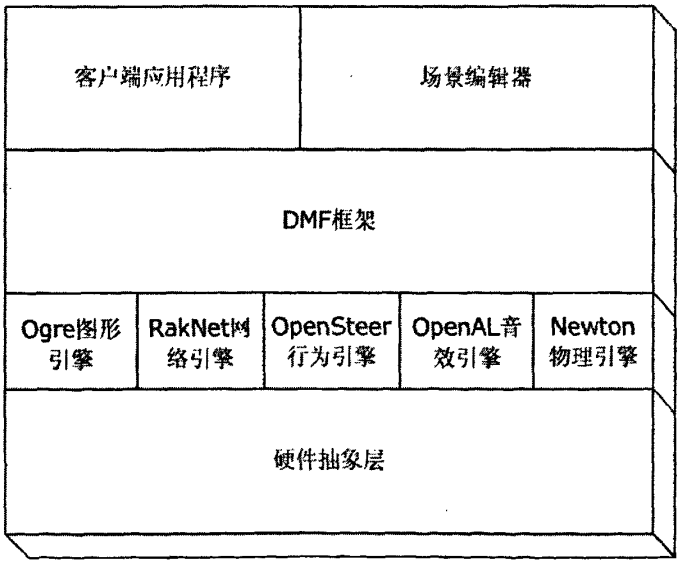


图 3-1 高层引擎结构图

我们所基于的开源引擎有：(1)、OGRE，OGRE（Object-oriented Graphics Rendering Engine）是一款开源的专注于架构一个强大的图形渲染引擎，关于 OGRE 我们在 2.1 节已做了详细介绍。(2)、RankNet，RakNet 是一个基于 Windows 和 Linux 操作系统、面向游戏的网络引擎。RakNet 提供一组网络 API，可实现可

靠的、有序的 UDP 通信和基于 Windows、Linux 和 Unix 系统的高层次网络编程。它可以应用在任何进程之间进行网络通信，RakNet 可用于任何网络应用，但它主要是针对网络游戏的开发。Raknet 不光解决了网络游戏一般的需求，而且为网络游戏的编程提供了一些额外的功能。(3)、Newton，Newton 物理引擎是一种对物理世界实时模拟的综合解决方案。它提供了场景管理、碰撞检测、动态行为等多种功能，是一款小型、快速、稳定且易于使用的物理引擎，并支持对地形这种复杂数据结构的物理建模。(4)、OpenSteer，OpenSteer 是一个为游戏中智能个体建立导航行为的 C++底层库，它定义了智能体的一些基本行为：漫游、追逐、逃避、分离、列队等，灵活运用这些行为，可以为智能体创建丰富多样的 AI 行为。(5)、OpenAL，OpenAL（Open Audio Library）是音频硬件的一个软件接口，为程序员提供产生高质量多通道输出能力的音效引擎。OpenAL 是在三维环境中模拟产生声音的一种重要方法，它跨平台，并且易于使用，在风格和规范上与 OpenGL 相似，它可应用于多种音频程序，但它的设计主要针对游戏音频的开发。

在多个开源引擎的基础上建立的 DMF 框架是我们系统的核心层，为了方便迭代开发和模块化设计，整个 DMF 框架设计采用了分层构建，如图 3-2 所示。

由上图可以看到，最底层的组件是核心组件 DMFCore 和轻量级单元测试库 DMFTest。接着中层的组件包括了各种子系统插件，这里没有全部列出，其他子系统还包括如地形子系统与脚本子系统等。最后是上层的 GUI 组件，可以支持多种 GUI 库，如 wxWidget，MFC，Qt 等。目前项目中的场景编辑器使用的是 MFC 作为 GUI 端开发库。以后如果需要进行跨平台移植可以很方便的转换到 wxWidget 或是 Qt 库。

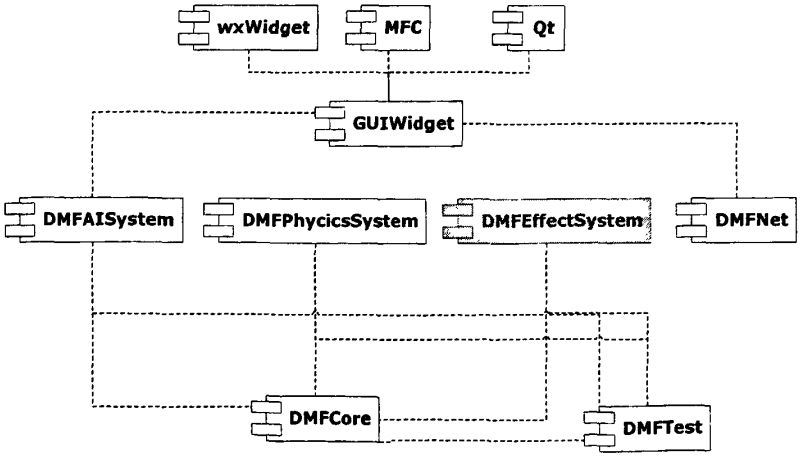


图 3-2 DMF 分层框架图

DMFCore：包含 DMF 的核心抽象类以及管理类，所有相关的派生类在 dll 载入的时候向管理类注册其创建工厂等信息。

DMFTest：包含一个用于自动化单元测试的库。

DMFPhycicsSystem：是 DMF 框架中为了支持系统中刚体物理运动的模拟的子系统，主要实现了大型复杂场景的碰撞检测。

DMFAISystem：是 DMF 框架中的 AI 子系统，实现了游戏人工智能系统的系统架构，并模拟了鸟群、鹿群等群体行为。

DMFEffectSystem：是框架中的特效子系统，实现了柔体与流体动画的实时模拟，同时提供了一些如爆炸、闪电、光晕、雨滴等高级特效的简单接口。

DMFNet：包含 DMF 的网络库，这个组件是基于 RakNet 开发的，为 DMF 框架支持网络协同编辑提供了底层支持。

本文所做的部件是属于整个系统框架的图形特效模块，即 DMFEffectSystem 模块。在下面的章节我们将详细的介绍本文的三个部件的设计与实现。

3.2 本课题在引擎中的位置

本课题主要研究了高动态范围技术以及景深和运动模糊特效的实时模拟技术。并且本课题实现的系统是对整个游戏引擎框架的一种在光照系统及成像方面的特效的扩展，其基于 OGRE 引擎开发，位于 DMF 框架之中，以插件的形式在 DMF 框架中实现（图 3-2）。所以，本课题在引擎中的位置如下图所示：

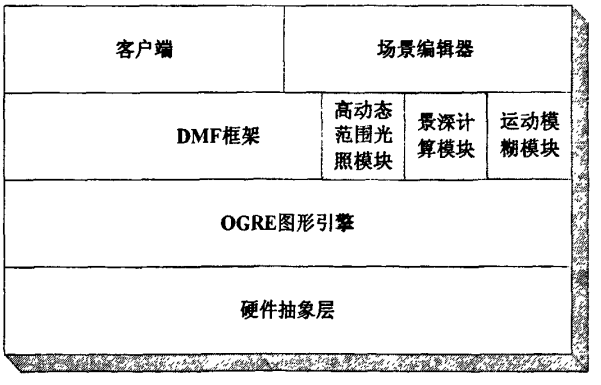


图 3-3 本课题与引擎的关系图

3.3 模块的实现类图

本文实现了实时的高动态范围光照、景深模拟和运动模糊插件，并实现了三者的结合使用。插件的内部实现如下图所示，下图为系统类图：

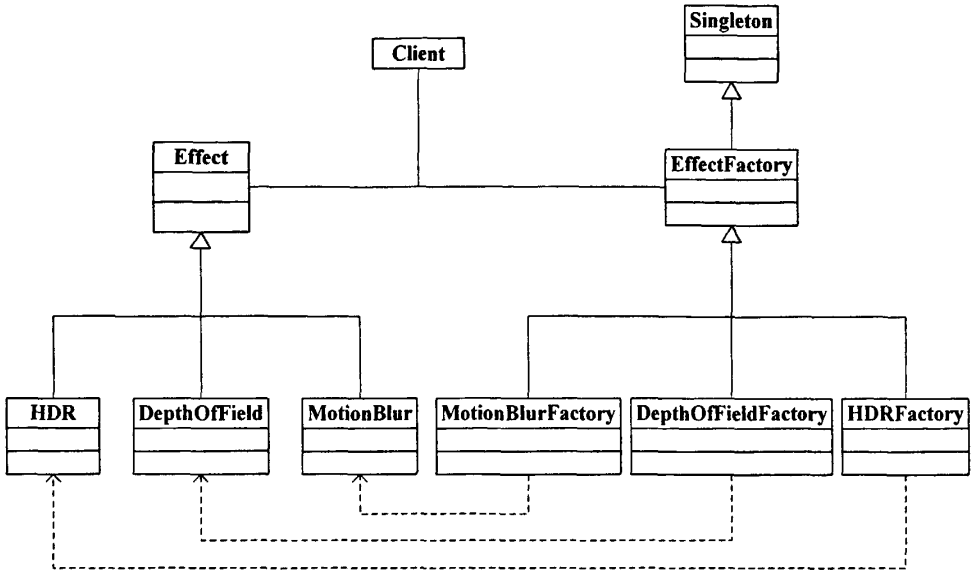


图 3-4 系统实现类图

本系统的实现主要是采用工厂模式和单件模式来实现的。由于本系统主要实现三个特效插件，一是高动态范围（High Dynamic Range），二是景深模拟（Depth of Field），三是运动模糊（Motion Blur）。使用单件模式（Singleton）可以保证工厂类全局唯一；使用工厂方法能很好把几个不同的类结合在一起，用工厂方法来实现各自的创建。HDRFactory 对应 HDR 对象的创建；DepthOfFieldFactory 对应 DepthOfField 对象的创建；MotionBlurFactory 对应 MotionBlur 对象的创建。这样为以后的扩展留了接口，例如要是以后要实现其他的特效，可以直接继承于 Effect 类，再写对应的工厂方法即可。

3.4 本章小结

这一章是我们系统整体框架设计的概要介绍，我们首先给出了一个宏观的框架，指出我们的大规模室外场景渲染引擎框架所处的位置，并对我们依赖的底层

开源库进行了介绍，这部分是数字媒体研究所网络游戏公共技术平台项目组的04级学长实现的。接着，我们对 DMF 框架内部的组件进行了介绍，着重介绍了本文的特效系统在游戏引擎中的位置。在后面的章节里，会对本文涉及的三种特效分别进行详细介绍。

第四章 高动态范围光照系统的设计与实现

真实、自然的光照是计算机图形学中经常模拟的现象，尤其是在计算机游戏中，光照总是必不可少的，它们的出现能够为玩家建立一个更加真实可信的游戏世界。但是，只有高动态范围的光照才能对场景进行真实的模拟，然后它却非一件非常容易的事情，不仅要考虑恰当的光照模型和物体材质，还要着重考虑怎样将高动态范围映射到低动态范围，即色阶重建。在我们的高动态范围光照系统的设计与实现中，我们采用了一种改进的 Phong 光照模型，并实行逐像素光照，以及用一种高效的方法进行色阶重建的计算。

4.1 色阶映射算法的相关研究

HDR 的场景在现实生活中大量的存在着，因此需要以某种方式将图像的动态范围进行缩放，使之匹配只能输出 LDR 的现实设备。这种方式称为色阶重建（Tone Reproduction）或色调映射（Tone Mapping），它提供一种方式将现实场景的亮度值缩放或者映射到现实设备能显示的范围。除了压缩亮度范围，还必须保留原始图像的感观质量（Perceptual Quality），如重建算法必须保留原始图像对比度、明亮程度、细节等信息。

动态范围的重建算法大致分为两种，一种是空域不变（Spatially Uniform），或者叫全局范围动态映射（压缩）[17,18,19]。此类算法在对图像进行动态范围变换时，每个像素（Pixel）上使用同一条变换曲线，变换曲线可以预先指定或者根据图像的内容获取。

从 1984 年起，Miller 和 Ngai 等人[20]在设计建筑图像渲染系统的时候，第一次根据实验数据制定图像亮度变换映射方法，尝试将现实场景中的亮度和显示图像的亮度进行匹配。接着，Upstill[21]在他 1985 年的博士学位论文中采用一种明确的感觉模型，强调了色阶重建技术中在寻求映射曲线时，必须要遵循人眼的知觉模型。在 1993 年，Tumblin 和 Rushmeier 等人[22]针对保留观察者对图像与场景的亮度信息在感觉上的一致性，提出了一种基于人类视觉系统（HVS）亮度感觉模型的图像变换理论基础。1994 年，Ward 等人[17]提出了一种保留对比度信息而不是亮度信息的变换处理算法。参考人眼对亮度相对变化的敏感模型，该算法能够

通过一个缩放因子以最低的计算开销将真实场景的亮度值变换到显示设备上的亮度值。

空域不变这类算法中以 Ward Larson 等人[18]在 1997 年提出的基于直方图调整的动态范围重建技术为标志，其考虑到人眼是对图像亮度的相对变化敏感而不是绝对变化，因此在图像中只要亮的区域被显示得较亮而暗的区域被显示得较暗就行了，并不需要考虑确切的亮度的绝对强度值。亮度级别（灰度级）的跨度在整个图像中并不是一个固定值，而表现成一系列灰度级的族，且各族的跨度不同。直方图均衡是在图像处理领域用来调整图像的对比度和可见性的方法之一，Ward Larson 等人[18]采用了这种调整直方图的方式，根据人眼感观模型来定义亮度级别的改变，以达到模拟效果，而非最大化图像的可见性。其不足在于不变的变换曲线不能自适应图像的各个区域，导致了结果图像在细节、颜色、明亮程度上损失。

2000 年，Scheel 与 Stamminger 等人[23]为了在交互式应用系统中利用动态范围重建技术，将亮度图像表示为纹理。他们将亮度图像的四个顶点映射成纹理坐标，在渲染这些纹理的时候，根据 Ward 等人[17]和 Ward Larson 等人[18]的变换算法，将纹理象素的值映射成显示象素的值。这样使得对于大型场景中交互，每一帧图像显示过程中进行全局亮度动态范围调整时，变换映射函数可以得到适应场景的修改[24]。

另一种是空域变化的（Spatially Varying），或者叫局部动态范围映射（压缩）。该类算法针对图像不同的区域进行不同的变换。根据人类视觉系统(HVS)的不同模型，各种不同算法在压缩动态范围的同时都以保留图像质量的某一方面为标准。早在 60 年代，Oppenheim 等人[25]在研究非线性滤波器的工作中建立了一种图像多层（Multi Scale）亮度模型，他们将图像分成两个部分：光照成分（有效的光照）和反射成分（场景中物体反射光线的能力）。光照成分包含着亮度强度的大范围变化，主要由低频分量构成；反射成分（细节）主要由高频分量构成。因此，图像中的低频成分趋向于高动态范围，高频率成分则趋向于低动态范围。通过在频域上对低频分量进行衰减，使得高动态信息被压缩而同时保留了高频成分[24]。

一直到 90 年代末期，各种算法都是在多层模型上针对不同的 HVS 模型进行调整，但是由于低频图像上采用的滤波函数特性不佳，在结果图像中物体边缘会产生严重光晕（Halo）一直是困扰该类算法多年的问题[26,25,27,28]。1999 年，Tumblin 等人[29]提出了 LCIS 算法（Low Curvature Image Simplifier），通过对图像不同细节的定义，提高了结果图像质量，但却使得算法变得非常低效，速度过慢。

在 2002 年的 ACM SIGGRPH 会议上，三篇论文在该领域同时发表。Durand

和 Dorsey[30]在基于分层模型的基础上采用具备边缘检测的双边滤波技术，避免了 LCIS 的缺陷，是基于分层模型的动态范围压缩算法获得了较高的性能和令人满意的结果。Fattal 等人[31]则从梯度域上对亮度图像进行多尺度的衰减，再以新梯度图像恢复出亮度图像。Reinhard 等人[32]则取法于摄影技术，将亮度范围分成不同区域（Zone），将 HDR 的不同区域单独映射到 LDR 的对应区域。

4.2 高动态范围光照系统的设计

高动态范围光照系统是基于整个游戏引擎实现，所以要满足游戏引擎的一些基本的要求，包括实时性、逼真性、动态性。实时性是最基本的要求，因为在游戏中，无论多复杂的场景都是不允许出现停顿感的，让可以顺畅的漫游其中。逼真性是一个目标，恰好我们的系统是高动态范围光照的，因此所做的场景具有非常的真实感和逼真的效果。动态性，即可以动态的移动视点和光源。在移动视点和光源条件下，整个场景还是满足实时性和逼真性的要求。

一个真实、逼真的场景是离不开灯光效果的，所以下面将首先阐述我们在 GPU 上的数据结构的设计。然后阐述我们的光照模型的设计及在 GPU 上的逐像素光照的实现。最后进一步阐述高动态范围光照系统的绘制流程及色阶重建的具体实现。

4.2.1 数据结构

由于 GPU 的特殊设计，其内存是以纹理这样的特殊形式表示的，因此我们采用纹理来存储向量。在图形处理器中，一维向量 V 可以采用一维纹理来存储，但是一维纹理的大小限制使得具有大量元素的向量难以表示。另一方面，就图形硬件而言，渲染一维纹理的速度要比渲染相同纹理元素的二维纹理慢[24]。因此，我们用二维纹理来存储向量，向量的每个元素对应纹理的每个纹理元素。这种向量在图形硬件内部存储及表示的方法能够使得前述数值算法中涉及到的基本运算操作在图形硬件加速下得以高效的进行。

针对以上情况，我们在 GPU 上的运算中，所有向量均由二维纹理图像来表示，如下图所示：

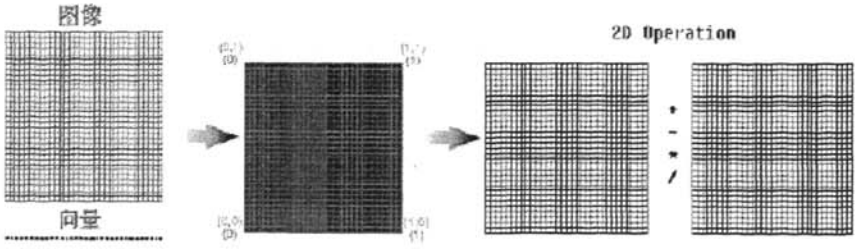


图 4-1 纹理示意图（图片来自参考文献[24]）

为了有助于 GPU 对方形纹理的处理，实现向量的基本运算，我们使用长宽相等的正方形二维纹理来代表向量，并且使其宽度等于 2n。高动态范围场景的色阶重建算法是在后处理（Post Processing）完成，其输入数据和输出数据本身就是一幅 2D 图像，因此直接由纹理图来存储也避免了由于数据的输入输出带来的附加拷贝和转换操作。同时，我们可以目前图形处理器支持的渲染到纹理（Render to Texture）操作将向量纹理化。

4.2.2 光照模型的设计

在一个普通亮度的场景中增加一个光源，会引起许多明暗程度的变化。对光的恰当使用，会产生许多有趣、梦幻、朦胧等感觉。这就是电影导演、舞台灯光师为什么对灯光非常重视的原因。同样的，光照模型（Illumination Model）是计算机图形学中生成真实感图形的基础。光照模型根据物体的表面材质和光源特性，按照光学物理的有关定律，计算几何物体表面上任一点上的光亮度和色彩组成。

在过去的研究中，为了生成有较强真实感的图形，人们已经提出了很多光照模型。其中较为成功的多个基于点光源的光照模型有：Lmabert 漫反射模型、Phong 模型、Blinn 模型、Cook-Torrnaee 模型、Whitted 模型、Hall 模型等。在图像处理系统中，综合考虑模型与实际的吻合程度和模型处理所需的代价，Phong 模型是一个可接受的“标准”模型，它综合反映了漫射、反射和环境光对表面作用的结合。

在可编程图形管线（Programable Pipleline）出现前，我们无论是使用 OpenGL 还是 DirectX，都需要通过对图形管线的状态设置来改变光源的属性，并且，固定图形管线（Fix Function Pipeline）只能支持一个光照模型——Phong 模型[33]。

现在，我们可以通过高级着色语言（HLSL、CG 等）编写程序来实现需要的复杂光照模型。我们通过对经典 Phong 模型的扩展，考虑物体放射（emissive）、

环境反射（ambient reflection）、漫反射（diffuse reflection）及镜面反射（specular reflection）等光照作用来计算物体表面颜色，其计算公式如下：

$$Color = K_{emissive} + k_a I_a + \sum_{i=1}^{M} f_i(d) I_i [k_d (N \bullet L_i) + k_s (N \bullet H_i)^n] \quad (4\text{-}1)$$

其中 $H = \frac{L+V}{|L+V|}$，$N \bullet H_i = R \bullet V_i$，其余符号见表 4-1：

表 4-1 颜色计算符号表

符号	描述
Kemissive	物体的放射光，为一个 RGB 值
Ia	入射环境光的光强
Ka	物体表面对环境光的漫反射系数
Fi(d)	第 i 个光源的强度衰减因子
Ii	第 i 个光源发出的入射光光强
Kd	物体表面的漫反射系数
Ks	物体表面的镜面反射系数
N	点（x,y）处的物体表面单位法向量
Li	第 i 个点光源发射方向的单位向量
Hi	将入射光反射到观察者方向的理想镜面的单位法向量
R	入射光的反射单位向量
V	从点（x,y）处到视点的单位向量

4.2.3 光照模块的 GPU 实现

在图形渲染管线（Pipeline）中，顶点着色器（Vertex Shader）中是一个在图形卡的 GPU 上执行的程序，它替换了在固定渲染管线中的变换（Transformation）和光照（Lighting）阶段。

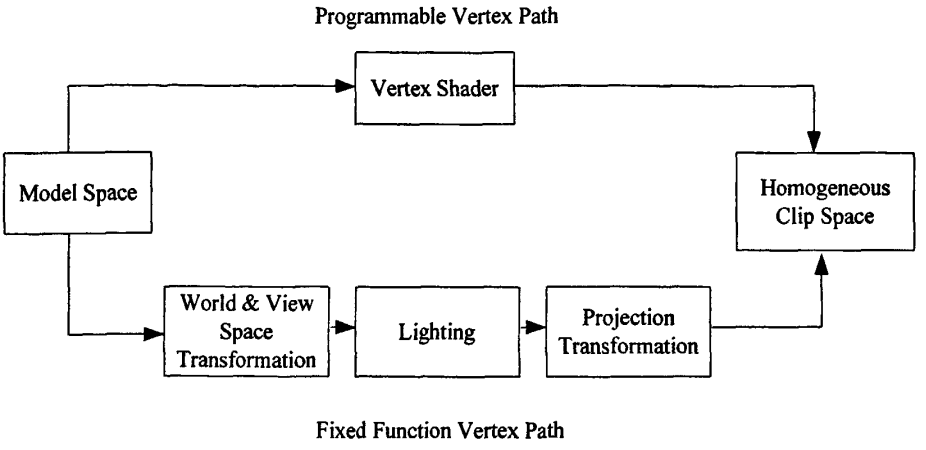


图 4-2 顶点着色器的替换部分

从上图可以知道，顶点以自身坐标输入到顶点着色器，并且进行坐标变换和光照之后输出在齐次裁剪空间（Homogeneous Clip Space）中的顶点位置。因此，要把一个顶点从本地空间（Object Space）变换到齐次裁剪空间，必须应用下列变换序列：世界变换（World Transformation）、视图变换（View Transformation）和投影变换（Projection Transformation）（它们分别由世界矩阵、视图矩阵和投影矩阵来完成）。

因此，一般来讲，我们应该在顶点着色器中完成光照计算。但是，由于光照计算在顶点级完成，光照计算只是在每个三角形的每个顶点进行；然后，光照为每个三角形所生成的片段（像素）插值；最终到达像素级时，计算出的光照颜色是被插值后使用的，这种方法被称为光滑颜色插值或 Gouraud 着色。因为光照计算不是真正地为每个片段（像素）进行的，因此它会丢失一些细节。

为了得到一个更好、更精确的效果，我们在顶点着色器将输入顶点坐标从物体空间转换到裁减空间，并传送顶点在视图空间中的位置和法向量传送给像素着色器，将光照计算延迟到像素着色器。

下面为根据光照模型使用高级着色语言 HSLS 编写的代码，其中顶点渲染部门的程序如下：

顶点着色器的输入结构为：

```
struct VS_INPUT
{
    float3 vObjPos : POSITION;        // 顶点在物体空间中的坐标
```

```
    float3 vObjNor : NORMAL;            // 顶点在物体空间中的法向量
    float2 vObjTex : TEXCOORD0; // 顶点的纹理坐标
};
```

顶点着色器的输出结构为：

```
struct VS_OUTPUT
{
    float4 Pos : POSITION;      // 顶点在裁剪空间中的坐标
    float2 Tex0 : TEXCOORD0; // 顶点的纹理坐标
    float3 Tex1 : TEXCOORD1; // 顶点在视图空间中的坐标，以纹理形式保存
    float3 Tex2 : TEXCOORD2; // 顶点在视图空间中的法向量，以纹理形式保存
};
```

顶点着色器程序为：

```
VS_OUTPUT main( VS_INPUT in )
{
  VS_OUTPUT Out = (VS_OUTPUT)0;
  // 顶点在视图空间中的坐标声明
  float4 vViewPosition;
  // 顶点在视图空间中的法向量声明
  float3 vViewNormal;
  // 将顶点的坐标和法向量从物体空间转换到视图空间
  vViewPosition = mul(float4(in.vObjPos, 1.0f), view _matrix);
  vViewNormal = normalize(mul(in.vObjNor, (float3x3)view_matrix));
  Out.Pos = mul(vViewPosition, proj_matrix); //将顶点的坐标和法向量从视图空间转换到裁剪
  空间
  Out.Tex0 = in.vObjTex; // 输出纹理坐标
  Out.Tex1 = vViewPosition.xyz; // 输出顶点在视图空间中的坐标，以纹理形式保存
  Out.Tex2 = vViewNormal; //输出顶点在视图空间中的法向量，以纹理形式保存
  return Out;
}
```

在图形渲染管线（Pipeline）中，像素着色器（Pixel Shader）中是一个在图形卡的 GPU 上的，在对每个像素进行光栅化处理期间的一个程序。它实际上替换了固定渲染管线中的多纹理阶段化（Multitexturing）阶段，并赋予我们直接操纵单独的像素和访问每个像素的纹理坐标的能力。

我们并没有在顶点着色器中完成光照计算，而是通过纹理将顶点在视图空间的坐标和法向量输出。经过图元装配和光栅化后，像素着色器得到的便是插值后的表面坐标和法向量，而不是插值最后的光照颜色。然后根据每个像素在视图空间中的坐标和法向量计算光照，这种技术被称为 Phong 着色，或逐像素光照（Per Pixel Lighting）。

这里，我们再次回忆光照模型示意图及计算公式：

$$Color = K_{emissive} + k_a I_a + \sum_{i=1}^{M} f_i(d) I_i [k_d (N \bullet L_i) + k_s (R \bullet V_i)^n]$$

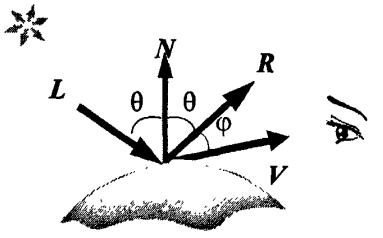


图 4-3 光照计算图

像素着色器程序为：

```
float4 main( in float2 vTex : TEXCOORD0, // 像素的纹理坐标
    in float3 vPos : TEXCOORD1,    // 像素在视图空间的坐标
    in float3 N : TEXCOORD2   // 像素在视图空间的法向量   ) : COLOR
{
    float3 V = normalize(-vPos); // 计算点（x,y）到视点的向量，记为 V，见本小节的表
    float3 color = float3(0.01f, 0.01f, 0.01f); // 首先计算环境光
    color += Emissive;        // 加入物体自身放射光
    for(int i=0; i < NUM_LIGHTS; i++)    // 处理多个光源
    {
        float3 L = normalize(vViewPosition - g_lihgtPos[i]);  // 计算光的入射向量
        float3 R = reflect( L, N ); // 计算光的反射向量
        float   fDiff = Kd * saturate(dot( N, L)); // 计算漫反射光
        // 计算镜面反射光，Ns 为镜面反射参数，根据材质不同
        float   fSpec       = Ks * pow( saturate(dot( R, V )), Ns);
        float fDis = distance(g_lihgtPos[i], vPos);   // 计算光源到物体点的距离
        color += (fDiff + fSpec) * I[i]/(fDistance*fDistance);  // 根据衰减计算最终光强
    }
    color *= tex2D(s0, vTex);    // 最后加上纹理颜色
    return float4(color, 1.0f);
}
```

4.2.4 高动态范围的绘制流程

高动态范围技术的诞生让场景中亮的地方变得真正的亮，暗的地方变得真正的暗；同一场景中可以同时绘制出高亮和黑暗局部的细节。不幸的是，目前的图形显示设备都是属于低动态范围，亮度范围在 0 到 255 之间，要在低动态范围的

显示设备上显示高动态范围的场景或图片，需要在后处理阶段经过以下步骤处理（如图 4-4）：

1) 将高动态范围场景渲染至高精度缓存，一般为浮点缓存或浮点纹理（Floating Render Target）；
2) 对第一步得到的纹理进行 1/4 缩小采样（Downsample），即分别在 x 方向和 y 方向缩小 1/2，并过滤掉处于低动态范围的颜色值；
3) 对第二步得到的缩小后的纹理进行 Bloom 滤波处理，一般采用可分离高斯滤波；
4) 将第三步得到的经过模糊处理后的纹理与第一步得到的原始高精度纹理叠加，进行色阶重建。

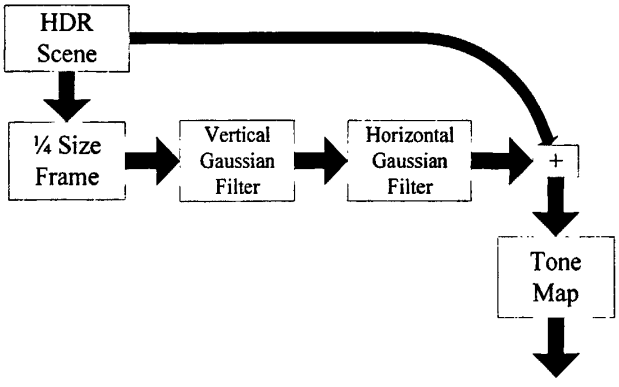


图 4-4 高动态范围的绘制流程

4.3 高动态范围在游戏引擎中的实现

4.2.4 节介绍了高动态范围场景的绘制流程，按照此流程，本节将详细介绍在我们的游戏引擎中实现基于 GPU 的高动态范围渲染的方法。由于高动态范围绘制的实现都是在 GPU 上实现，因此我们使用了高级着色语言 HLSL 来编写代码实现绘制。

4.3.1 将场景渲染至高精度缓存

由于高动态范围场景的特点是存在亮度很高的区域，普通的低动态范围的缓存一般采用 A8R8G8B8 格式，即动态范围在 0 到 255 之间，为了更好的保存更多的细节，需要采用了浮点 16 位高精度缓存(A16B16G16R16F)来保存高动态范围场

景的原始信息。

因此，进行高动态范围场景处理的第一步是在一个有相同比例尺寸的高精度纹理上进行场景渲染，并且使用 4.2 节的实现的逐像素光照模型。下图显示了高精度缓存的内容。由于整个场景不是很亮，大多数像素具有的每个色彩通道不在 0.0 到 1.0 的图形卡显示范围之内。因此，这些纹理直接观察时看起来大部分是比较暗的。但是，0.0 以上的色彩信息并没有丢失，只不过需要色调映射到 0.0 到 1.0 的范围。

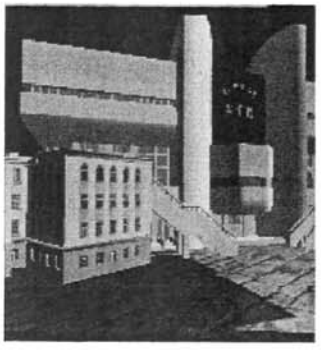


图 4-5 高精度缓存内的图像

4.3.2 计算平均亮度 $\bar{L}_w$

和许多色阶重建算法 Reinhard[32]一样，在本文中将亮度的对数平均值视为重要因素之一，在进行色阶重建前，需要计算场景的平均亮度值，采用了高动态范围图片的计算平均亮度的公式：

$$\bar{L}_w = \exp[\frac{1}{N}\sum_{x,y}\log(\delta + L_w(x,y))] \qquad (4\text{-}2)$$

其中，N 是图片像素的数量。x，y 是每个像素的 2D 坐标。δ 是个很小的常数，以避免当某像素点亮度为 0 时，log 的取值正常。L_w 是像素的亮度值。

在读取了 RGB 值后，需要把 RGB 值转换成光强值 L_w，使用以下公式：

$$L_w = 0.27R + 0.67G + 0.06B \qquad (4\text{-}3)$$

公式 4-2 要在 GPU 上实现，并不太容易。需要在帧缓存中读入每个像素。如果在一个绘制管道中，读入所有像素，这即使在现在 PS3.0 的硬件上也是不能实现的。在很多情况下，即使在两个管道下也不一定能实现。因此，本文采用了以

下步骤来计算场景平均亮度：

1. 创建另一个高精度缓存（float render target），记为 smallScene，对高精度场景纹理使用高斯滤波进行缩小采样（downsample）。由于场景将被多次采样以便实现快速处理，最好将场景缩小到像素着色器取样的最小值，因此将 x 和 y 方向分别缩小到 1/4；
2. 建立一串的按比例缩小的纹理：64×64，16×16，4×4，1×1，以便快速测量场景的平均亮度。纹理格式为 D3DFMT_R16F 或 D3DFMT_R32F（以 DX 为例）；
3. 首先从 smallScene 中获取颜色值，然后根据公式 4-3 转化成灰度/亮度，渲染到 64×64 中；
4. 然后，从上一级获取颜色值，然后渲染到下一级中，直至 1×1，这样，就在 1×1 的纹理中得到了场景平均亮度。在这种缩小采样（downsample）的情形下，采用高斯滤波进行，而不是点取。

下面，我们示例了三个像素着色器函数，它们都是独立的像素着色器代码，在不同的技术中（Technique）执行。

首先，我们需要计算亮度值公式 4-3 和公式 4-2 中的 log 对数计算，第一个像素着色器代码完成此功能。

```
float4 GetLumLog ( in float2 postion_screen : TEXCOORD0   //纹理坐标   ) : COLOR{
    float3 v_Sample = 0.0f;      float   f_LogLum = 0.0f;
    for(int i = 0; i < 9; i++)      {
        // 计算所有采样点的亮度值公式 4-2，然后计算 log 平均值
        v_Sample = tex2D(s0, postion_screen + v_SampleOffsets[i]);
        f_LogLum += log( dot(v_Sample, LUMINANCE_VECTOR) + 0.0001f );
    }
    // 取采样平均值
    f_LogLum /= 9;      return float4(fLogLum, fLogLum, fLogLum, 1.0f);
}
```

接着，我们需要按照步骤 2、3、4 不断的对纹理逐级向下采样的计算，第二个像素着色器代码通过的采样点周围像素的融合，并迭代的执行此像素着色器代码便可完成此功能。

```
float4 DownSampleLum ( in float2 postion_screen : TEXCOORD0   //纹理坐标   ) : COLOR{
    float f_sum = 0.0f;
    for(int i = 0; i < 16; i++) {          // 对采样点周围像素采样后求平均
        f_sum += tex2D(s0, postion_screen + v_SampleOffsets[i]);
    }
```

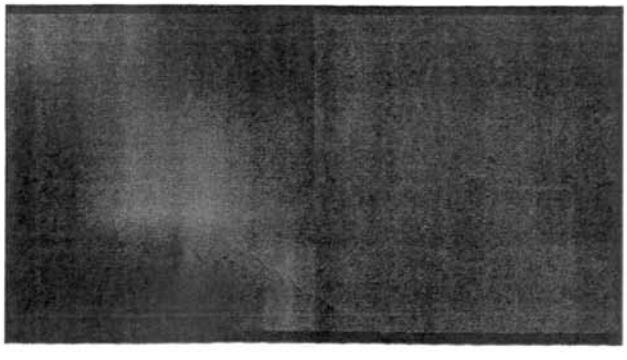
```
    f_sum /= 16;     return float4(f_sum, f_sum, f_sum, 1.0f);
}
```

最后，为了完成公式 4-2 对平均亮度的计算，并且按照我们对平均亮度的计算步骤，需要对以上计算的结果做 exp()运算，再将亮度信息渲染到一个 1×1 大小的纹理。第三个像素着色器代码完成此功能。

```
float4 GetAvgLum( in float2 postion_screen : TEXCOORD0    //纹理坐标   ) : COLOR{
    float f_sum = 0.0f;
    for(int i = 0; i < 16; i++) {
      f_sum += tex2D(s0, postion_screen + v_SampleOffsets[i]);// 对所有采样点的值进行求和
    }
    f_sum = exp(f_sum/16); // 对求和后的值求平均，再做 exp()运算，得到平均亮度值
    return float4(f_sum, f_sum, f_sum, 1.0f);
}
```

下图显示了亮度计算的过程。

(a)64×64　　(b)16×16



(c)4×4　　(d)1×1

图 4-6 亮度计算过程图示

4.3.3 亮度过滤及辉光处理

在真实场景中，由于光线在大气及人眼中的散射作用，当人眼注视高亮物体的时候会有光晕的出现，感觉高亮物体的光会发散到它周围，因此需要模拟这种高亮物体周围具有光晕的真实感效果。要将高亮部分的光强扩散到周围，就需要对场景进行辉光处理。辉光处理是通过在水平和垂直方向上做可分离高斯滤波实现。通过高斯滤波，可达到亮度到四周扩散从而模拟高亮部分的光晕效果，但是在非高亮会引起场景的模糊。所以在进行高斯滤波前需要进行亮度过滤处理。

亮度过滤的计算过程如下：

1. 将每个像素的亮度值用公式 4-4（4.3.4 节）进行处理；
2. 设置阀值，将亮度值减去阀值以去掉非高亮区域；
3. 将亮度进行归一化处理。

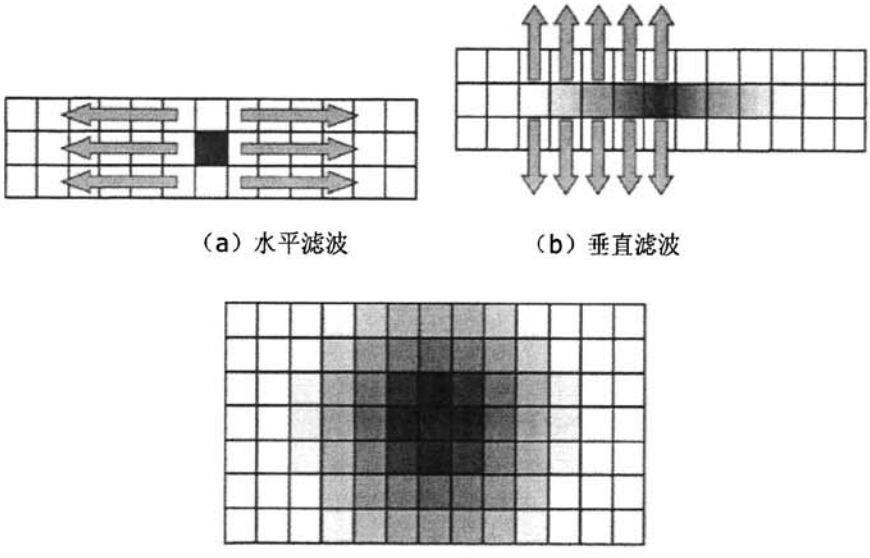
模糊辉光源，可以将其展开成为一个平滑而又自然的辉光图案。模糊是在硬件中使用一个二维的图像处理滤波器完成的。创建辉光效果的速率取决于模糊的执行效率。执行模糊所需的时间则取决于应用滤波的范围（以纹素计）。当增加模糊滤波的范围、覆盖更多的纹素时，我们必须读写更多的纹素，这与二维的模糊面积成正比。而面积与模糊直径的平方成正比，或 d^2。如果把辉光的直径加倍，则需要处理 4 倍的纹素。对于包含 50×50 个纹素的模糊形状，为了所创建辉光的每个像素，我们不得不读 2500 个纹素。这对制作大面积的辉光是非常不实际的，但是幸运的是，通过一个两步操作（称为分布卷积）执行模糊，可以避免讨厌的直径平方的麻烦。分布卷积把消耗从 d^2 减少到 2d，因此对每个像素只读 100 个纹素，却能产生一个 50×50 的辉光[34]。这个在现代图形硬件上能很快地执行。

分步卷积设计的目的是为了减少某种特殊情况的计算，即卷积核心是几个项的乘积，而每项在各个轴上彼此独立。在这种情况下，一个 n×m 个元素的二维卷积，就能被转化为两个分别的 n 和 m 的一维卷积。这极大地减少了卷积的计算消耗。不再需要在每个点计算并加和 n×m 个样本，卷积被简化成两步，总共只需要处理 n+m 个样本[34]。第一步，对结果中的每个点沿着一根轴取样并加和 n 个样本，创建一个中间图像。第二步，对中间结果的邻近地区沿着另一根轴取 m 个样本，创建最终结果的每个点。n 或 m 样本的每个加权因子是沿着每个轴卷积的轮廓。我们的主要想法是选择一维的卷积轮廓，使得两步法可以实现。即使一个特殊的二维模糊形状不可能在数学上分解，我们也可以使用两个一维的轮廓去近似那个形状。仅通过做两个一维的模糊，我们可以创建许多各种不同的二维模

糊形状。

我们在 GPU 上进行渲染到纹理的操作，先在一根轴上模糊，然后在另一根轴上模糊第一个模糊。渲染取出围绕每个像素的纹素的局部相邻的区域，而且把卷积核心权数加到这个相邻区域的样本上。如果 GPU 能读出一遍中所有相邻的样本，一个卷积就能在一个单个的渲染遍中执行；不然就使用相加的混合，一次累积几个相邻的样本，在几个渲染遍上得到结果。因此，我们在 GPU 上采用可分离高斯滤波处理来完成分步卷积，即首先沿 x 方向对辉光源进行水平滤波（水平模糊），再对得到的模糊纹理沿 y 方向进行垂直滤波（垂直模糊），这样便得到了模糊的辉光效果，以在后期处理中混合进场景。

可分离高斯滤波处理示意图如下：



（a）水平滤波　　（b）垂直滤波

（c）结果

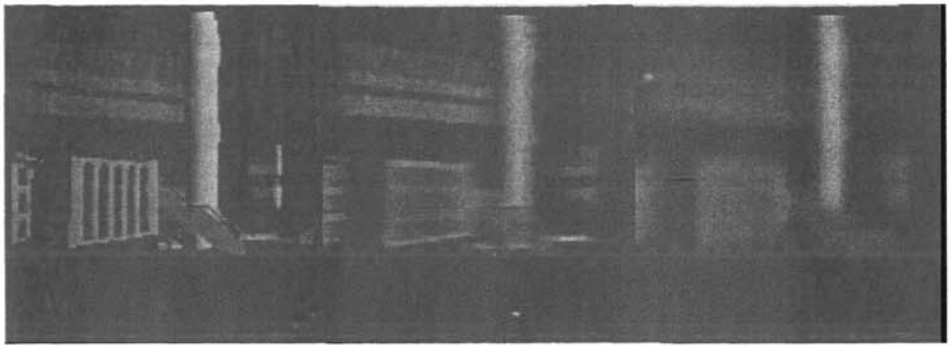
图 4-7 可分离高斯滤波

可分离高斯滤波的高级着色语言 HLSL 示例，以 x 方向为例，y 方向同理：

```
// Pixel shader
float2 BlurOffset      = float2( 1/1024, 0 );  //根据纹理大小设置
float PixelWeight[8] = { 0.2537, 0.2185, 0.0821, 0.0461, 0.0262, 0.0162, 0.0102, 0.0052 };
float4 ps_main( float2 inTex: TEXCOORD0 ) : COLOR0{
    float4 color = tex2D( BlurXSampler, inTex );
    // 在该象素点的左右两侧采样，并乘以权值
    for( int i = 0; i < 8; ++i ) {
        color += tex2D( BlurXSampler, inTex + ( BlurOffset * i ) ) * PixelWeight[i];
```

```
        color += tex2D( BlurXSampler, inTex - ( BlurOffset * i ) ) * PixelWeight[i];
    }
    return color;
}
```

亮度过滤过程在显存中的示意图如下：



(a)亮度过滤　　(b)水平高斯滤波　　(c)垂直高斯滤波

图 4-8 亮度过滤及 Bloom 处理结果示意图

4.3.4 曝光控制

暴光控制是找出一个适当的低动态范围的 HDR 场景视图的方法。在物理镜头系统中，光圈调整光线进入系统的数量极限。该方法使用计算机图像来模拟，通常叫做色调映射的暴光控制。它引用映射一个高动态范围图像空间到低动态范围空间中以适合于视频显示的方法。不同于普通照片的暴光控制，色调映射时所有高范围数据是有效的；所以，图像比使用传统的摄像机可能包含更多的来自亮区和暗区的细节。

在平均场景亮度计算之后，HDR 场景纹理能被按照目标平均亮度缩放，表现在下面方程式的 a 上。

$$L(x,y)=\frac{aL_w(x,y)}{\bar{L}_w} \tag{4-4}$$

其中 a 是一个关键值，在 0 至 1 之间（一般亮度的场景建议使用 0.18），用于场景曝光度的控制，a 值越大，其曝光度越高，场景越明亮；反之，若 a 值越小，则曝光度越低，场景越暗。

4.3.5 色阶重建计算

色阶重建主要分为全局色阶映射和局部色阶映射，全局色阶映射由于对所有像素点都采用同一映射函数性能较高，这里我们也采用全局映射函数，早期著名的色阶映射函数：

$$L_d(x,y)=\frac{L(x,y)}{1+L(x,y)} \tag{4-5}$$

通过此映射，高亮的像素值被缩放了大约 1/L 倍，低亮的像素值被缩放了大约 1 倍。它将所有范围的亮度值都缩放到了可显示的低动态范围。但是，有时我们需要对场景的亮度进行控制，采用了以下公式：

$$L_d(x,y)=\frac{L(x,y)[1+\frac{L(x,y)}{L_{white}^2}]+\sigma}{1+L(x,y)} \tag{4-6}$$

其中 L_d 是被映射的像素的光强值，L_{white} 是最大光强值，这是一个 HDR 值，亮度值大于 L_{white} 的像素都会映射为全白。L_{white} 能被设成一个无限值，用于把产生的光强值映射为一个可以显示的值，σ 为自定义值，当场景不需要全黑的局部时，可将 σ 设置为非零值。L(x,y)是由公式 4-4 产生。

总之，先从纹理中读入 RGB 值，公式 4-3 转换成实际光强值 L_w，然后用 L_w 和公式 4-2 计算出平均光强值 $\bar{L}_w$。通过公式 4-4，使用 $\bar{L}_w$ 和 L_w 计算出 L（x, y)。最后用公式 4-5 或者公式 4-6 计算出最后的值，这个值即使可以在显示其上显示的对应高动态光强的值。

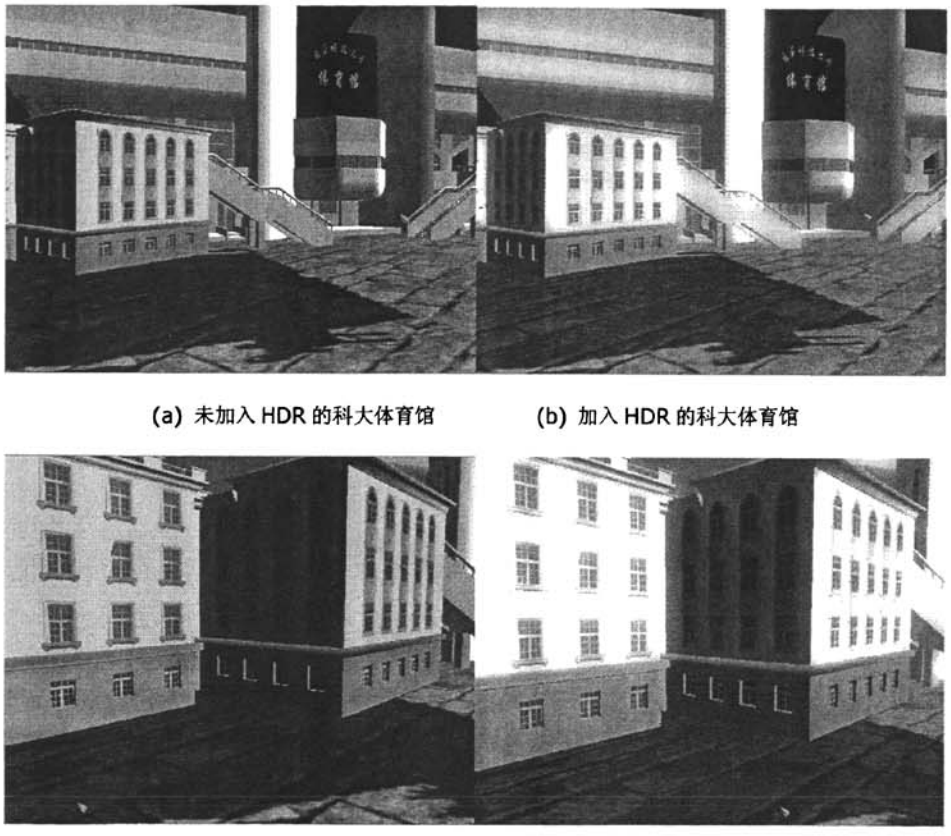
色阶重建的高级着色语言 HLSL 代码示意如下：

```
float4 ToneMapping (   in float2 pos_screen: TEXCOORD0    ) : COLOR{
  float4 v_HDR = tex2D(s0, pos_screen); // 取得该像素在原始高动态范围纹理的颜色
  float4 v_bloom = tex2D(s1, pos_screen); // 取得该像素在 bloom 处理后的颜色
  float f_AvgLum = tex2D(s3, float2(0.5f, 0.5f)); // 取得该场景的平均亮度值
  v_HDR.rgb *= middleGray/(f_AvgLum + 0.001f); // 公式 4-4 的计算，根据平均亮度进行缩放
  // 色阶重建的计算，将高动态范围光照信息映射到可显示的低动态范围
  v_HDR.rgb /= (1.0f+ v_HDR);
  v_HDR += f_bloomScale * v_bloom; // 将在后处理阶段得到的 bloom 效果加上
  return v_HDR;
}
```

4.4 实验结果与分析

下面我们给出系统模拟产生的数据，并对它进行分析。我们在 IBM 图形工作站上实现，图形工作站的配置为：Intel Xeon™ CPU 3.0GHz，2.0GB 的内存，NVIDIA Quadro FX 3400/4400。操作系统为 Microsoft Windows XP Professional（SP2）。

我们的试验是在用色阶重建（Tone Mapping）技术将高动态范围场景映射到低动态范围进行渲染。实验结果如图 4-9 所示。



(a) 未加入 HDR 的科大体育馆　　(b) 加入 HDR 的科大体育馆

(c) 未加入 HDR 的科大主楼　　(d) 加入 HDR 的科大主楼

图 4-9 高动态范围光照系统试验截图

通过对高动态范围场景的色阶重建，将之输入到低动态范围的显示设备上。试验表明开启 HDR 的场景、材质、整体环境对光线的比未开启 HDR 的表现的更加生动，具有更加真实的光线，保留了其真实的更多的细节，具有更好的对比度，

实现了“亮的地方真正的亮起来”的效果，给人种亲临现场的真实感。在未开启HDR特效之前只能看到光线的平淡表现，但开启HDR特效之后却可以得到刺眼光线的真实体验。这说明在开启HDR后，当画面的亮度更加趋近于现实的时候，以往游戏中昏暗的画面风格得到了本质的改善，玩家因此获得更加逼真的游戏体验，增强了玩家的沉浸感。

表 4-2 实验结果

帧率	HDR	非HDR
当前	66.9 fps	100.1 fps
平均	66.5 fps	100.5 fps
最坏	60.2 fps	90.3 fps
最好	67.8 fps	103.7 fps

图中场景分别是电子科技大学体育馆及主楼，两个模型分别有10万多个三角形面片。在没有高动态范围渲染的实现下，我们的平均帧率可以达到100.5fps以上，在开启高动态范围光照时，平均帧率有所降低，但仍然有66.5fps，这说明我们的试验是满足游戏实时性要求的。

4.5 本章小结

本章主要介绍了高动态范围场景渲染技术，首先介绍了在GPU上对Phong光照模型模型的扩展。接着介绍了高动态范围的关键技术——色阶重建的相关研究以及具体方法。在进行色阶映射时，需要将场景渲染至高精度缓存保存；缩小纹理以便采样；取得场景的平均亮度；进行亮度过滤以实现高亮部分的光晕甚至耀光效果；将原始纹理以及各种效果实现叠加进行色阶重建将场景映射到可显示的低动态范围。最后，详细介绍了高动态范围场景渲染的具体步骤，并对实验结果进行了分析，说明我们的实验满足逼真性和实时性。

第五章 景深模拟系统

上一章从光照模型的角度在 GPU 上实现了对 Phong 模型的扩展，并从光强的动态范围的角度实现了高动态范围光照系统。简单说，是从光源模型、光照计算和场景亮度范围来实现了具有真实感的光照。

本章将从光照效果的成像机制上研究，首先对透镜成像模型与针孔成像模型进行研究，然后在此基础上提出一种在计算机三维场景成像中实时地模拟出景深效果的算法。该算法利用了 MRT（Multiple Render Targets）技术及 GPU 的可编程性，在渲染时将场景存为纹理，并输出了像素的深度值和模糊因子，利用模糊因子计算每像素模糊圈大小并对图像滤波，最终在模糊圈内对清晰图像和模糊图像进行融合，模拟出景深效果，并以一个场景的景深模拟为例，展示了不同聚焦时的景深效果。

5.1 景深数学模型的建立

三维计算机图形系统模型采用的是针孔照相机模型，如图 5-1。理想针孔照相机的小孔很小，所以从点光源发出的光只有一条光线能进入小孔，是点对点的线性映射关系，因此理想针孔照相机的景深是无穷远，在视域范围内任何点都在聚焦范围之内[35]。

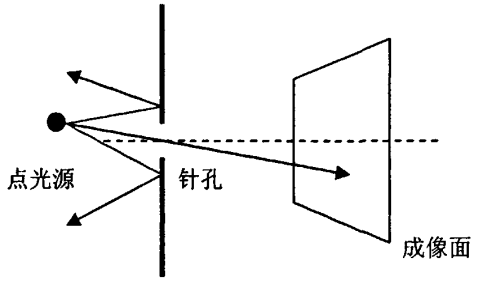


图 5-1 针孔照相机模型

人眼视觉系统可用图 5-2 所示的薄透镜模型[9][36]表示，此模型也广被使用。从物点 O 发散出进入透镜的光线被折射成像于像点 I。透镜焦距 f，物距 u ，相距 v 间关系构成透镜公式 5-1：

$$\frac{1}{f}=\frac{1}{u}+\frac{1}{v} \tag{5-1}$$

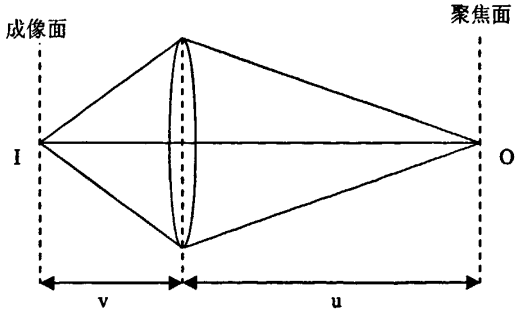


图 5-2 薄透镜成像模型

透镜对光线的折射取决于它的焦距 f，人眼通过调节自身的焦距 f 以让目光所注视的物体(聚焦面上的物体)在视网膜（成像面）上成像清晰。然而，在聚焦面两侧的物体上的一点其在视网膜上的成像则不止一个点，是一个模糊圈(CoC, circle of confusion)，如图 5-3，聚焦面外的物点 O 通过人眼晶体折射后在视网膜的成像为一个直径为 Cr 的模糊圈。模糊圈（CoC）的直径 Cr 用公式 5-2[9]计算：

$$C_r=\left|V_d-V_f\right|\left(D-V_d\right) \tag{5-2}$$

$$V_d=\frac{P\bullet d}{d-P} \qquad d>P$$

$$V_f=\frac{P\bullet d_f}{d_f-P} \qquad d_f>P$$

$D=$ 透镜直径；　　$d=$ 物体到透镜的距离；$d_f=$ 聚焦面到透镜的距离；

$$P=\frac{1}{\frac{1}{d_f}+\frac{1}{d_r}}=f$$

$d_r=$ 视网膜到透镜的距离；

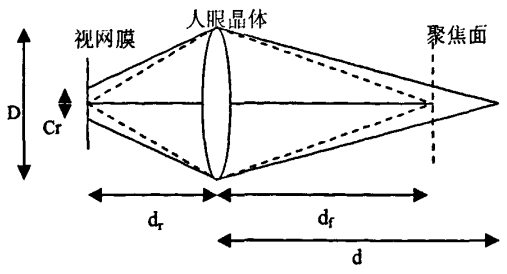


图 5-3 模糊圈的计算图示

5.2 相关的研究现状

近年来，在计算机图形学领域中出现了许多关于景深渲染的算法研究，这些算法主要归为三类。

（一）、后处理滤波。Petmesil 等学者[37][38]是后处理滤波的代表，也是最早研究景深算的学者。其算法采用标准针孔相机模型渲染场景，并输出每个像素的深度值 z；根据深度值 z、光圈、焦距等将每个采样点转化为不同大小、强度分布的模糊圈（CoC）；每个像素的最终值由覆盖它的所有模糊圈（CoC）的加权平均值确定。Poetmesil 采用了 Lommel 强度分布函数计算点对周围像素的影响。Chen 也采用了类似的方法来计算强度分布[39]。周强等[40]采用了均值滤波得到模糊图像，再与清晰图像融合，实时的模拟了景深效果。

Poetmesil 和 Chen 的算法都是采用软件实现，运算全由 CPU 负担，运行时间较长，难以满足虚拟现实等高实时性的要求。Rokita[41][36]提出使用特殊的数字硬件滤波器以加速 DOF 效果的产生。它采用多次高斯卷积滤波将像素值融合到周围像素中，达到模糊的效果。由于采用了卷积滤波技术，引起像素的强度渗漏，引起前、后景物体混合模糊、聚焦面上的物体与前景或后景的模糊物体混合模糊等。周强等[40]的算法虽实时性满足，但均值滤波会引起强度渗漏，且融合时未考虑模糊圈内的像素，精度不高。

（二）、多次渲染。其采用针孔相机模型，通过每次细微的改变投影中心，并保持聚焦面不变[42]，然后将渲染结果累积保存，最终便得到一幅具有景深效果的图像[43]。但是，多次渲染所得的景深效果重影较重，缺乏真实感。

（三）、反向光线跟踪。Kolb 等[44]的几何透镜模型和 Rebort 等[45]的分布式光线跟踪采用对每个像素反向跟踪若干条光线,取这些光线颜色的加权和作为像素颜色,成功地模拟了聚焦、景深等效果。这样图像上的点是多束光线叠加的效果，

避免了针孔相机模型中一对一映射而无景深效果的缺陷。但此类算法绘制速度很慢,且当透镜参数改变时,需重新计算[46]。吴向阳等[46]采用基于正向光线跟踪模拟底片的成像机制，生成了真实的摄像效果，并且当透镜参数改变时无须重新求取光线与场景的交和光亮度，但算法仅限于几何场景，并采用真实透镜模型，因此，速度难以满足虚拟现实系统要求。

目前很多算法需要大量的计算或者缺乏精度而不能应用于虚拟现实系统。本文所采用的景深算法，充分利用了目前 GPU（图形处理器）的并行性和可编程能力，将大量代数运算从 CPU 转移至 GPU，不仅释放了 CPU，并且减少了 CPU 与 GPU 的通信量，大大提高了景深模拟的速度，适用于大规模复杂场景；同时，根据薄透镜系统成像时产生模糊的原理（根据公式 5-2）求得的模糊圈进行融合，提高了景深模拟的精确度。

5.3 景深模拟系统的设计

5.3.1 设计目标

景深模拟系统也是基于整个游戏引擎实现,所以同高动态范围光照系统一样，景深模拟系统也同样要满足游戏引擎的一些基本的要求，包括实时性、逼真性、动态性。实时性是最基本的要求，因为在游戏中，无论多复杂的场景都是不允许出现停顿感的，让可以顺畅的漫游其中。逼真性是一个目标，我们的景深模拟需要将处于聚焦面附近上的物体成像清晰，在聚焦面两侧（包括前、后）的物体成像根据其离聚焦面的距离进行模糊，距离越远则成像越模糊，并且在对非聚焦面上的物体成像进行模糊的时候，避免将前景物体与后景物体成像混合，以模拟真实相机拍摄的景深效果。动态性，即可以动态的移动视点和聚焦面，并且在移动视点和聚焦面的条件下，满足实时性和逼真性。

在我们的景深模拟系统中，性能的表现需要满足下面的要求：

- 在视点关注的距离（聚焦面）要能提供清晰的渲染效果；
- 在视点没有关注的距离（非聚焦面）要能提供较好的景深（模糊）渲染效果；
- 在同时进行高动态范围光照的情况下，帧速（FPS）不低于 30。

基于上面的原则，我们实现并评估了景深渲染算法。下面介绍我们实现的算法流程。

5.3.2 绘制流程

图形真实感的研究主要集中在光照效果的生成上，而成像机制大多基于针孔模型，针孔模型所生成的图像往往超现实。我们在对透镜成像模型与针孔成像模型进行研究后，提出一种在计算机三维场景成像（针孔相机模型）中实时地模拟出薄透镜成像模型下景深效果的算法。算法的步骤如下：

1）渲染场景至纹理（清晰纹理），同时将每个像素的深度信息及模糊因子输出到另一张纹理；
2）将第一步得到的清晰纹理进行模糊处理，得到模糊纹理；
3）根据模糊因子计算模糊圈大小，然后对清晰纹理和模糊纹理采样，根据模糊圈大小进行融合，得到具有景深效果的场景图。

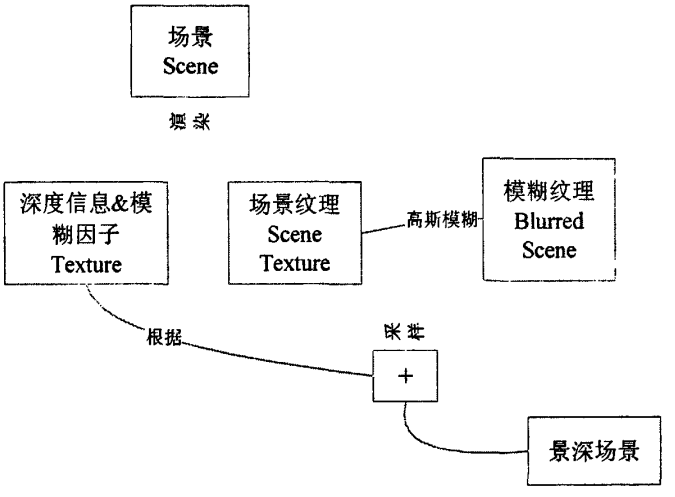


图 5-4 景深模拟的绘制流程

5.3.3 数据结构

景深模拟的计算同样需要在图形处理器上进行，由于 GPU 的特殊设计，其显存是以纹理这样的特殊形式表示的，因此同高动态范围光照系统一样我们采用纹理来存储向量。

5.4 景深模拟在引擎中的实现

一幅具有景深（DOF）的图，在其聚焦面上的成像是清晰可见的，而在聚焦

面两侧的成像是模糊的，其模糊程度和模糊圈（CoC）成正比关系。因此，我们可利用清晰和模糊场景纹理的融合（Blend）来模拟景深（DOF）效果，融合因子采用归一化的模糊圈（CoC）大小。下面为算法的具体实现流程：

5.4.1 利用顶点编程以及像素编程获得深度信息和模糊信息

在虚拟现实系统（VR）中，光圈、焦距及透镜参数可以随意调节，因此物体各点的模糊程度需要实时计算（公式 5-4）。同时，为了防止前景像素流入后景，在保存场景物体各点的模糊程度之外，还需要保存场景各点的深度信息，因此，可采用多渲染目标（Multiple Render Target）。

多渲染目标是一种硬件功能，最早支持它的硬件是 ATi 的 Rdaoen9700。多渲染目标 (MRT)技术允许像素着色器将每像素的数据保存到不同的渲染缓冲区当中。随着渲染方式越来越复杂，单一的渲染目标不足以存储所有的渲染结果，比如在延时渲染(DeferredRendering)中，最后所有的光照计算都是在屏幕空间进行的，光照计算不仅需要颜色信息，还需要法线和镜面光指数等信息，这时显然单一的渲染目标是不够的，这就是 MRT 技术提出的背景。

这里，我们采用多渲染目标（MRT）将场景输出到一张纹理，深度信息和模糊信息归一化后输出到另一张纹理。（如下图）

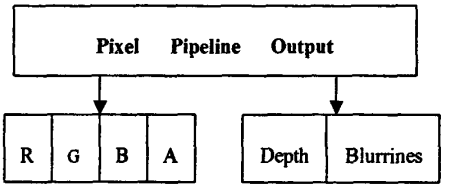


图 5-5 像素管线输出示意

在视图空间中，设物体上点的坐标为 Pos(x, y, z)，近剪截平面为 znear，远剪截平面为 zfar，则物点的深度值为 z，归一化输出为 depth=z/zfar；根据公式 5-2，可计算得出近剪截面的模糊圈（CoC）直径为 Cnear，远剪截面的模糊圈（CoC）直径为 Cfar，取它们的最大值作为 maxCoC，即：

$$\max CoC = \max(C_{near}, C_{far}) \tag{5-3}$$

物体上的点的模糊圈大小可有公式 5-2 得到，归一化后模糊因子为：

$$Blurriness = \left| \frac{f \bullet z}{z-f} - \frac{f \bullet d_f}{d_f - f} \right| \bullet (D - \frac{f \bullet z}{z-f}) \Big/ \max(C_{near}, C_{far}) \tag{5-4}$$

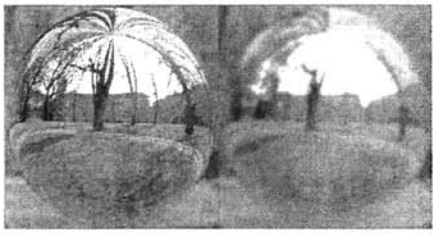
5.4.2 图像预模糊

处于聚焦平面外的物体上的点，经过透镜折射后，在成像面上形成一个模糊圈，在屏幕上，则表现为多个像素组成的圆形区域。它是像素与周围多个像素相互作用的结果。在针孔照相机模型中，可以采用平滑滤波处理得到。在图像处理中，常用的平滑算子有高斯滤波、均值滤波等，为了取得更好的效果，这里，采用可分离 2D 高斯滤波。

$$F = \frac{\sum_{i=1}^{n}\sum_{j=1}^{n} P_{ij} \bullet C_{ij}}{S} \qquad (5\text{-}5)$$

F 表示目标像素滤波后的值；Pij 表示 2D 高斯矩阵中的像素；C 表示像素 Pij 对应的高斯系数；n 表示矩阵的维数；S 表示高斯矩阵中所有系数的和。

在高斯滤波的规模（scale）确定时，可采用标准卷积的计算方法来计算高斯滤波，因此计算分解为两部分：沿 x 方向的计算 1D 卷积；再在此基础上沿 y 方向计算 1D 卷积。



(a) (b)

图 5-6 高斯滤波（a 为原图，b 为高斯滤波图）

5.4.3 模拟模糊圈（CoC）、融合清晰和模糊的图像

经过5.4.1和5.4.2两个处理（Pass）后，得到了一幅清晰的图像和一幅模糊的图像，可以直接融合这两幅图得到具有景深效果的场景图。如下：

$$color = a * blur\ \mathrm{Im}age + (1-a) * original\ \mathrm{Im}age \qquad (5\text{-}6)$$

其中，a为blurriness，由公式5-4求得。

为了达到更好的景深效果，在融合每个像素前，应根据此像素对应的模糊因子（blurriness），确定一个*CoC*大小。在模糊圈范围内取1个中心点以及按照泊松圆盘分布（Poisson Disk Distribution）随机的取12个像素进行采样（pixel shader2.0中，一个pass最低支持13个采样器）。

模糊圈的大小是根据模糊圈的中心采样点的模糊因子（blurriness）以及*maxCoC*计算的，如下公式：

$$Radius = Blurriness * \max CoC/2 \quad (5\text{-}7)$$

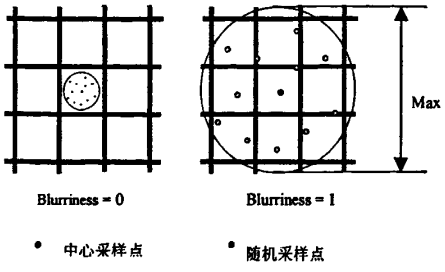


图 5-7 Blurriness 和模糊圈大小的关系

图5-7左边的模糊因子（blurriness）等于0，说明该点处于聚焦面上，由公式5-7计算得模糊半径为0，此点成像清晰；右边的模糊因子（blurriness）等于1，由公式5-7计算得模糊半径为*maxCoC/2*，此点成像模糊。根据模糊采样圈中的13个采样点，逐个对原始的清晰图像和模糊后的图像分别采样，得到*colorOriginal*、*colorBlur*；再根据原始图像中此像素对应的模糊因子（blurriness），进行融合，如公式5-8；最后，求13个采样点融合后的几何平均数就得到了最终此像素点的颜色值。

$$color = lerp(colorOriginal, colorBlur, Blurriness) \quad (5\text{-}8)$$

在求采样点的几何平均时，如果将所有融合后的颜色值求算术平均，会导致前景物体的颜色流入到后景中。所以，在5.4.1节的处理中，保留了每个像素对应的深度值，在对模糊圈（CoC）求平均时，将每个采样点的深度值与中心点（Center Point）的深度值做比较。如果此点深度值小于中心点的深度值，则将其滤去，但这样会引起“Poping”效果，即前景与后景的变换很突然，缺乏一种柔和过渡的效果。为了解决“Poping”，对前景物体上的点，采用它的模糊因子作为其权值与其它点相加，就可达到既无前景流入后景的问题，又无“Poping”问题。算法代码描述如下（HLSL）：

```
.....
/****变量说明
texCoord：中心采样点的纹理坐标
tSource：5.4.1 节得到的存储原始场景的纹理
tBlurred：5.4.2 节得到的存储模糊场景的纹理
depthBlurSample：5.4.1 节得到的存储对应于原始场景中每个像素的深度和模糊值的纹理
```

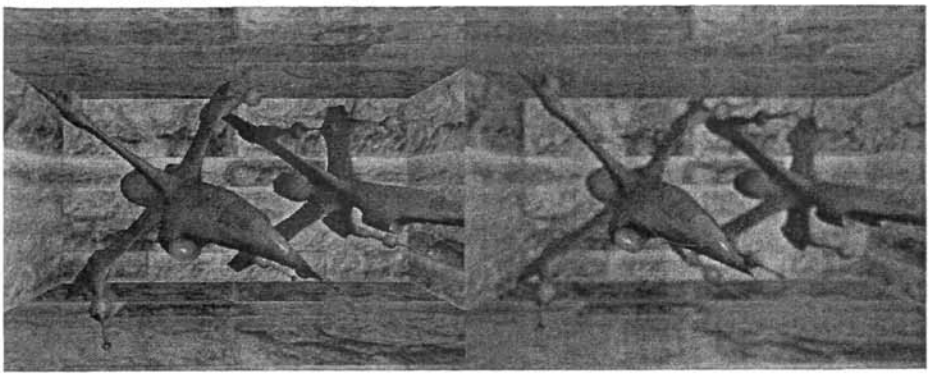
```
pixelSize：像素的大小
poisson[]：存储泊松分布的数组
radius：CoC 的半径
****/
//定义最终（输出）颜色值
float4 colorOut = float4( 0, 0, 0, 0 );
//取得中心点的深度值 centerDepth
float4 centerColor= tex2D(depthBlurSample,texCoord );
float centerDepth = centerColor.x;
//对 13 个采样点循环处理
for( int i=0; i<=13; i++ ){
//计算每一个采样点的纹理坐标
float2 coord = texCoord + pixelSize*poisson[i]*radius;
//取得原始颜色值
float4 colorOriginal = tex2D( tSource, coord );
//取得模糊颜色值
float4 colorBlurred = tex2D( tBlurred, coord );
//取得深度和模糊值
float4 depthBlur = tex2D( depthBlurSample, coord);
//根据深度值对原始和模糊纹理进行融合
float4 color = lerp( colorOriginal, colorBlurred, depthBlur.y );

float blur;
//处理前景点
blur = ( centerDepth <= depthBlur.x ) ? 1 : depthBlur.y ;
cout.rgb += color.rgb * blur;
cout.a += blur;
}
cout = cout / cout.a ;
  ......
```

5.4.4 实验结果分析

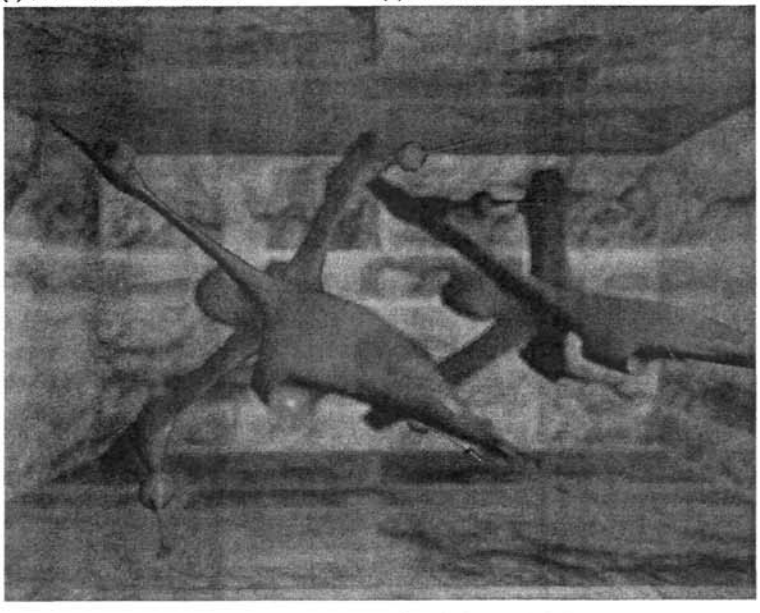
下面我们给出系统模拟产生的数据，并对它进行分析。我们在 IBM 图形工作站上实现，图形工作站的配置为：Intel Xeon™ CPU 3.0GHz，2.0GB 的内存，NVIDIA Quadro FX 3400/4400。操作系统为 Microsoft Windows XP Professional（SP2）。

我们的试验是根据本章提出的景深模拟算法，对聚焦面两侧（包括前、后）的物体成像根据其离聚焦面的距离进行模糊，并保持处于聚焦面附近上的物体成

像清晰。实验结果如图 5-8 所示。



(a)未加入景深，其成像清晰　　(b)加入景深，聚焦面（椭圆框内）成像清晰



(c)加入景深，聚焦面（椭圆框内）成像清晰

图 5-8 景深效果图

表 5-1 实验结果

帧率	景深	无景深处理
当前	78.0 fps	109.1 fps
平均	77.5 fps	108.5 fps
最坏	70.2 fps	98.3 fps
最好	79.8 fps	110.7 fps

图如上三幅为实验的效果图，红色线框所围的为处在聚焦面上的点，其成像清晰；在聚焦面两侧（包括前、后）的物体成像则根据所处的深度不同，即离聚焦面越远或者越近的而越模糊，并且在对非聚焦面上的物体成像进行模糊的时候，并无前景物体与后景物体的成像混合，接近真实相机拍摄的景深效果。

图中场景为两架飞机，一前一后，分别处于不同的深度。在进行景深模拟时，可以手动调节聚焦深度，图 5-8(a)和图 5-8(b)分别展示了聚焦面处于不同飞机的头部，实验平均帧率可达 77.5fps。以上说明我们的实验是满足游戏实时性、逼真性及动态性要求的。

5.5 本章小结

本章主要介绍了在 GPU 上实现景深模拟的原理和技术。首先将场景渲染到纹理保存，与其它算法不同的是，我们利用了现代管线的多渲染目标技术（MRT，Multiple Render Targets）在将场景渲染至纹理的同时输出该点的深度信息；然后在后处理阶段通过每个像素点的深度信息来对原始清晰图像和模糊图像进行融合得到具有景深的结果。最后对实验结果进行了分析，说明我们的实验满足逼真性和实时性。在下一章，我们会将景深系统加入高动态范围光照环境下进行实现。

第六章 运动模糊模拟系统

第四章从是从光源模型、光照计算和场景亮度范围的角度实现了具有真实感的高动态范围光照，第五章从光照效果的成像机制上实现了对远景和近景（非聚焦面上）的物体模糊，即景深模拟。

本章我们同样从光照效果的成像机制着手，对非无限小的相机快门成像系统研究发现，当物体运动速度比快门速度快、快门还未关闭时，物体在胶片上的成像会发生移动，因此最终相片上物体显得半透明模糊，即“运动模糊”。而计算机采用了针孔相机模型的快门时间几乎等于零，因此不会产生运动模糊效果，从而让本该具有动感、真实性的图像变得静止。由于变速的、非直线的运动在成像瞬间可以视为匀速直线运动[47]，匀速直线运动模糊图像在运动模糊图像的研究领域内很具一般性和代表性。为了提高执行效率，我们在 GPU 上实现了对运动模糊的实时模拟，并以一个匀速直线运动物体为例，展示了本系统的模拟效果。在本章的最后，我们将运动模糊与高动态范围光照系统和景深模拟结合，实现了基于高动态范围的实时景深及运动模糊模拟。

6.1 运动模糊的相关研究现状

近年来，计算机图形学中出现了许多关于运动模糊模拟算法的相关研究。本节，我们将对这些算法进行讨论。

计算机图形学对运动模糊的研究中，Korein 与 Badler[48]、Potmesil 与 Chakravarty[50]、Dippe 与 Wold[49]等人对运动模糊算法进行了早期的基础性研究。大多数研究都集中于用光线追踪渲染来产生运动模糊效果。

Potmesil 与 Chakravarty [50]采用了三步计算运动模糊。第一，光线追踪器决定了可见性以及运动的像素；第二，算法根据图像路径函数（Image Path Function）对运动的像素作卷积运算，产生适当的模糊；第三及最后，将静态图片和做了卷积的图片合并，并考虑深度关系。从那以后，学者将运动模糊归为时间混淆问题。

Korein 与 Badler[48]、Dachille 与 Kaufman[51]的算法采用了随机采样抗锯齿（Temporal Anti-Aliasing），其模拟效果最好并在计算机电影制作中得到了广泛应用。他们是通过在时间域上对场景进行超采样来对运动模糊进行模拟；具体来讲

就是对每一帧，在离散的时间点上多次渲染场景，然后再求以上渲染结果的平均并输出至显示设备。这类算法的关键在于，在某一时间段（运动的时间间隔）里对帧缓存内的像素值求平均。

随着计算机图形硬件的进步，为了节省空间，以上渲染和求平均可放在硬件的累加缓存（Accumulation buffer）中进行[52]。但是，每一帧最终的效果依赖于渲染的次数。由于整个场景必须渲染多次，这种方法的扩展性并不好。如果场景中多边形数量很大时，对场景进行多次渲染会降低帧率（Frame Rate）。当即物体的运动速度很快时，每一帧对场景渲染的次数小于场景中动作发生的数量，此时渲染结果就会显得不流畅，并产生重影（double vision）或者拖影（ghosting）[53]，如图 6-1。这种效果被叫作“随机采样锯齿（temporal aliasing）”。存在快速运动的虚拟现实系统需要每帧对运动至少渲染 20 次以消除拖影。

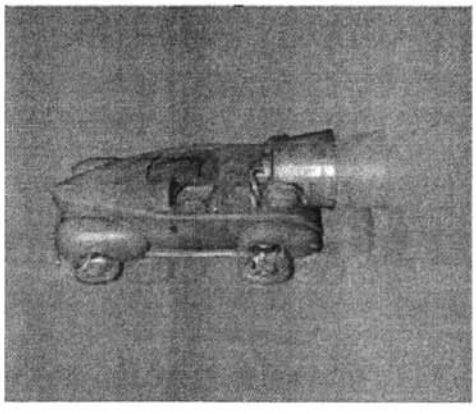


图 6-1 使用累积缓冲模拟运动模糊（图片来自参考文献[53]）

Wloka 和 Zeleznik[54]在物体空间中提出了解决运动模糊问题的方法。在物体空间中，运动模糊是由运动物体在曝光时间内占据了大于物体体积的空间，他们把这个被占据了的大于物体体积的空间称作“运动体积”（motion volume）。Wloka 和 Zeleznik 通过物体分割成为后表面（trailing surface）和前表面（leading surface），并构造连接面（joining surface）来近似物体的“运动体积”。由于“运动体积”表示物体在[t0, t1]时间间隔内占据的体积，“运动体积”的后表面（trailing surface）等效与物体出发时（即 t0 时刻）的背面（考虑到物体的运动路径）；“运动体积”的前表面（leading surface）等效与物体到达时（即 t1 时刻）的正面（再次考虑到物体的运动路径）；“运动体积”的连接面（joining surface，between leading and trailing surface）代表在[t0, t1]内物体的轮廓（contour）扫过的体积。

构造“运动体积”的基本的算法由 4 步构成。一、计算物体的运动向量；二、

根据运动向量，将物体分为两部分：前和后；三、从物体的前半部分到后半部分，构造“运动体积”的前、后、连接表面；四、由前、后、连接表面汇成“运动体积”。

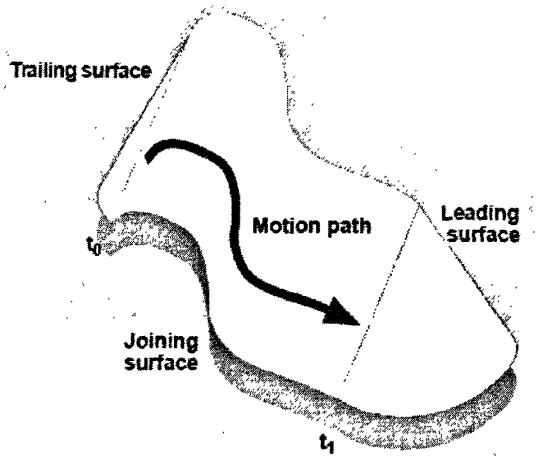


图 6-2 物体的运动产生的“运动体积”

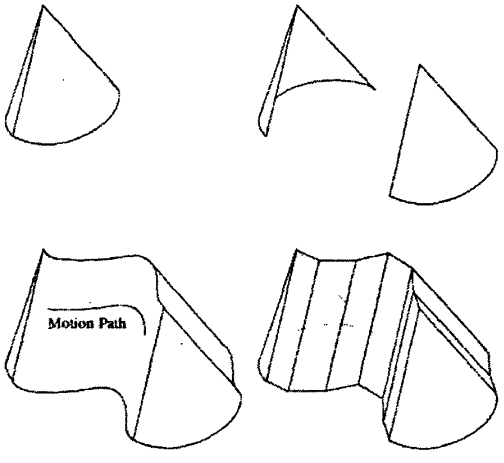


图 6-3 “运动体积”示意图

Wloka 和 Zeleznik[54]的方法在三维物体空间中对物体的运动体积进行了较好的模拟，并能够在计算机硬件上实时的处理。但是这种方法不易扩展，并且不能很好的处理曲线轨迹的运动模糊。

本章所采用的运动模糊模拟算法，将构建物体“运动体积”（motion volume）的计算从三维空间中映射到图像空间（2.5D），并利用目前 GPU（图形处理器）

的并行性和可编程能力，对渲染后的场景进行后处理（Post-Processing），有效提高了运动模糊模拟的速度，适用于大规模复杂场景。

6.2 运动模糊系统的设计

6.2.1 设计目标

运动模糊模拟系统也是基于整个游戏引擎实现，所以同高动态范围光照系统和景深系统一样，运动模糊模拟系统也同样要满足游戏引擎的一些基本的要求，包括实时性、逼真性、动态性。实时性是最基本的要求，因为在游戏中，无论多复杂的场景都是不允许出现停顿感的，让可以顺畅的漫游其中。逼真性是一个目标，我们的运动模糊模拟系统需要根据物体的运动速度、方向来进行模糊，并且对未运动的物体要保持清晰，以求达到真实相机拍摄运动物体的模糊效果。动态性，即可以动态的移动视点和聚焦面，并且在移动视点和聚焦面的条件下，满足实时性和逼真性。

6.2.2 绘制流程

计算机仿真运动模糊图像从方法上讲，大致可以分为两类：一是在三维空间中，根据物体的运动速度及方向，构造出运动体积，即运动物体在曝光时间内在三维空间中所经过的体积。简单说，是在三维空间中通过对顶点的操作实现运动模糊。二是在 2.5D 空间中进行处理，将运动物体在三维空间的速度等映射到二维空间形成光流（Optic Flow），然后在二维空间中通过对像素的操作实现运动模糊。这种方法通常需要硬件（GPU）加速，在渲染的后处理阶段上执行。

显然，在三维空间中的运动模糊模拟的正确度比在 2.5D 空间的高，但是其模拟速度和场景的复杂度成反比，而 2.5D 的方法由于是在图像空间完成的模拟，因此效率不受场景复杂度的影响。为了满足游戏引擎对实时性的要求，我们采用了一种在 2.5D 空间中完成运动模糊计算的方法，其算法的步骤如下：

1） 渲染场景至纹理（清晰纹理）；

2） 进行后处理（Post-Processing）对场景进行第二次渲染，将后表面（Trailing Surface，即法向量方向与运动方向相反的顶点）上的顶点平移到前一帧该点所处的位置并转化为窗口坐标，同时将物体在三维空间中的速度表示映射到图像空间；

3）利用像素编程根据速度及坐标对清晰纹理进行采样，实现运动物体的模糊。

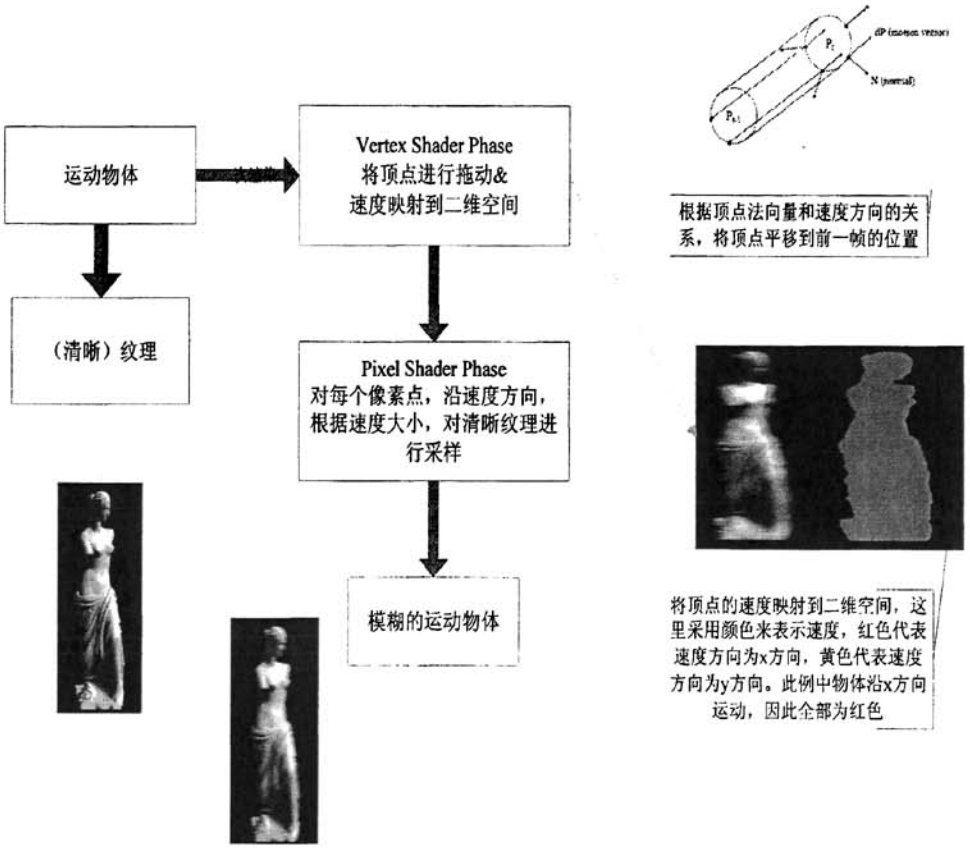


图 6-4 运动模糊的绘制流程

6.2.3 数据结构

运动模糊模拟的计算同样需要在图形处理器上进行，由于 GPU 的特殊设计，其显存是以纹理这样的特殊形式表示的，因此同高动态范围光照系统和景深模拟系统一样，我们采用纹理来存储向量。

6.3 运动模糊在引擎中的实现

6.2.2 节介绍了运动模糊在 2.5D 空间上的绘制流程，按照此流程，本节将详细介绍在我们的游戏引擎中实现基于 GPU 的运动模糊渲染的方法。下面，将对运

动模糊算法及实现步骤进行详细介绍。

6.3.1 将场景渲染至纹理

如上面所阐述的，本文采用的方法是在 2.5D 空间计算来近似运动模糊效果。在第二次渲染的时候，对第一次渲染得到的场景纹理在 2D 空间进行操作以提高计算的性能，因此需要首先将场景渲染至纹理保存。

6.3.2 利用顶点编程求得顶点的窗口坐标及速度

我们需要利用图形显卡的顶点着色器（Vertex Shader）对运动物体的顶点进行处理。首先，根据顶点当前坐标及前一帧坐标，计算出这个顶点的运动向量（运动方向）。根据运动向量与顶点法向量的夹角大小，可以判断出顶点是属于前表面（Leading Surface）或者是后表面（Trailing Surface）。类似于 Wloka 和 Zeleznik 对物体表面的划分，我们将顶点运动方向与顶点法向量间夹角小于 90 度的称为前表面，将顶点运动方向与顶点法向量间夹角大于或等于 90 度的称之为后表面。对于处于前表面的顶点，保持它的坐标不变；对于处于后表面的顶点，将顶点拖回上一帧它所处的位置，以产生拖尾的模糊效果。

在求得顶点的坐标同时，计算出顶点映射到 2D 空间的速度，以便在像素着色器中采样使用。具体过程如下：

顶点着色器的输入结构为：

```
struct VS_IN {
  float4 coord    : POSITION0;      // 顶点的坐标
  float4 prevCoord : POSITION1;     // 顶点前一帧的坐标
  float3 normal   : NORMAL;
  float2 texture : TEXCOORD0;
};
```

顶点着色器的输出结构为：

```
struct v2f {
  vector pos      : POSITION;  // 窗口坐标
  float2 velocity : TEXCOORD0; // 2D 速度
};
```

为了计算顶点的窗口坐标和运动速度，我们需要知道当前帧的视图矩阵 MV（ModelView）、视图投影矩阵 MVP（ModelViewProjection）及前一帧的视图矩阵 PreMV、视图投影矩阵 PreMVP。

因此，我们可以计算得到物体顶点在视图空间的当前位置 Pos、前一帧位置 PrePos 及运动向量 MotionVector。

$Pos = in.coord * MV$

$\mathrm{Pr}ePos = in.preCoord * \mathrm{Pr}eMV$

$MotionVector = Pos - \mathrm{Pr}ePos$

再得到运动向量后，将处于物体前表面的顶点保持位置不变，而处于物体后表面的顶点平移到前一帧所在的位置；并计算出该顶点在图像空间中的速度 Velocity。

$Pos = in.coord * MVP$

$\mathrm{Pr}ePos = in.preCoord * \mathrm{Pr}eMVP$

$Velocity = Pos - \mathrm{Pr}ePos$

$Pos = (MotionVector \bullet in.normal > 0)?Pos : \mathrm{Pr}ePos$

通过 GPU 的顶点着色器程序，将物体顶点在图像空间的运动速度作为纹理坐标输出，并输出顶点在图像空间中经过位移后的坐标，然后将它们传递给像素着色器进行处理。

6.3.3 利用像素编程实现模糊

在顶点着色器输出了顶点的速度和窗口坐标后，图形处理器管线（Pipeline）会进行图元装配及光栅化处理，到像素着色器输入时，每个顶点的速度及坐标被插值为每个像素的速度及坐标。

此步属于后处理（Post-processing），即将场景作为纹理贴在一个与窗口大小成比例的矩形上，然后在 2D 空间对场景进行操作。运动物体的动态模糊效果，是在曝光时间内物体在运动方向上位移而产生的拖影。因此可根据每个像素的窗口坐标和速度值，在物体映射到屏幕的窗口坐标上，沿此像素的运动方向（即运动速度的方向），按运动速度的大小对第一步渲染得到的场景纹理进行采样，得到运动模糊的模拟效果。为了使模糊程度可以得到用户控制，可以引入模糊因子，用模糊因子与速度大小的乘积来对场景纹理进行采样。模糊因子一般取值范围为：0 到 1.0，也可根据需要扩大到大于 1。

像素着色器的输入为：

```
struct PS_IN {
  vector   pos      : COLOR0;  //窗口坐标
  float2   velocity : TEXCOORD0;
```

};

像素着色器的输出为此像素的 24 位颜色值。

其具体步骤如下：

①. 根据运动向量（速度）和模糊因子乘积，沿速度方向对场景纹理多次查询；

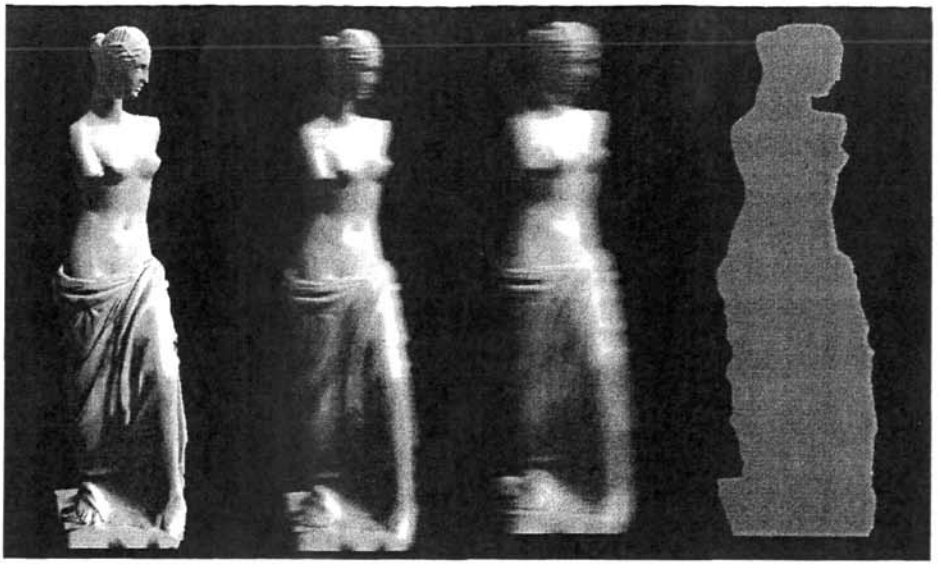
②. 产生采样点的加权总和；这里可采用均值滤波（Box Filter）或高斯滤波（Gaussian filter）等；

```
// 沿速度方向，根据速度大小，进行采样
half2 velocity = In.velocity.xy * blurScale;  //模糊因子
  const fixed w = 1.0 / samples;  // 权值，一般为 8；这里采用均值采样
  float4 color = 0;
  for(float i=0; i<samples; i+=1) {
    half t = i / (samples-1);
    color += tex2D(sceneTex, in.pos + velocity*t) * w;    //对像素坐标沿速度方向采样
  }
```

需要注意的是采样点的数量取决于运动的数量，一般来讲 8 个采样点便可，16 个采样点会达到更好的效果，采样点越多会降低帧的渲染率。

6.3.4 模拟效果及分析

下面我们给出系统模拟产生的数据，并对它进行分析。我们在 IBM 图形工作站上实现，图形工作站的配置为：Intel Xeon™ CPU 3.0GHz，2.0GB 的内存，NVIDIA Quadro FX 3400/4400。操作系统为 Microsoft Windows XP Professional（SP2）。



（a）原始图像　（b）模糊因子 0.5　（c）模糊因子 1.0　（d）速度可视化（红色代表沿 X 方向的速度）

图 6-5 模糊与原始图像

6.4 基于高动态范围的景深及运动模糊的实现

5.4 节和 6.3 节分别对景深计算和运动模糊模拟算法进行了详细描述，并在可编程图像硬件上进行了实现。本节将会把景深计算及运动模糊融入到高动态范围场景进行模拟，并对其性能进行分析。

第四章讲述了高动态范围场景的渲染流程，首先需要将高动态范围场景渲染至高精度缓存保存，然后再在后处理过程进行色阶重建。第五章讲述了景深计算在普通场景中的模拟过程，它需要先将场景渲染至纹理，同时保留深度信息，然后在根据计算得到的模糊圈（CoC）来融合清晰和模糊的纹理，以至得到最终的具有景深效果的图像。

由于高动态范围场景的渲染主要是进行色阶重建（Tone Mapping），而这一步是在后处理阶段完成。因此，为了在具有高动态范围的场景中加入景深模拟效果，可以首先在高精度缓存中完成景深模拟的计算，将具有景深效果的场景保存至高精度纹理；然后再将具有景深效果的场景的高精度纹理再进行色阶重建，便可得到高动态范围场景下具有景深效果的模拟。

同样的，6.3 节讲述了普通场景中运动模糊效果在可编程硬件(GPU)上的计算

过程。它需要将场景先渲染到纹理保存，然后再根据每个像素点在图像空间中的速度来实现运动物体的模糊效果。

因此，同在高动态范围场景中实现景深模拟一样，在高动态范围场景下进行运动模糊的模拟，可以首先在高精度缓存中完成运动模糊的计算，将具有运动模糊效果的场景保存至高精度纹理；然后再将具有运动模糊效果的场景的高精度纹理再进行色阶重建，便可得到高动态范围场景下具有运动模糊效果的模拟。

同理，在高动态范围场景中同时加入景深计算及运动模糊效果，应该先在高精度缓存中先进行景深计算和运动模糊计算，最后再进行色阶重建，将具有景深和运动模糊效果的高动态范围场景映射到低动态范围。

以下分别在高动态范围场景下对景深计算、运动模糊及同时对二者的模拟。其背景部分为电子科技大学主楼（左边）、电子科技大学体育馆（右边），中间分别放置了一架 F4U1 飞机与一辆 M2A2 坦克，前方放置了一架 F22 分机，并在进行旋转。

实验是在如下配置的 IBM 图形工作站上进行的：

- CPU：双 CPU，Intel Xeno 3.0GHz，
- 系统内存：2GB，
- GPU：NVIDIA 的 Quadro Fx3400，显存为 256MB

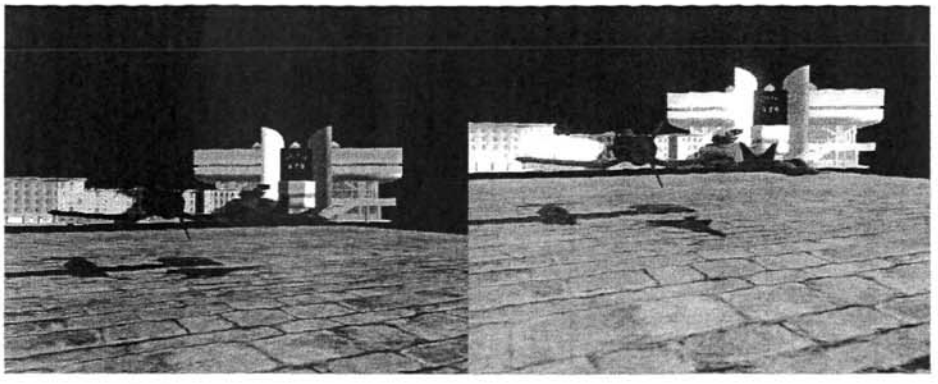
编译环境为：

- 操作系统：Windows XP Professional(SP2)，
- 开发工具：Microsoft VisualC ++ 7.1
- 图形渲染：OGRE 图形引擎

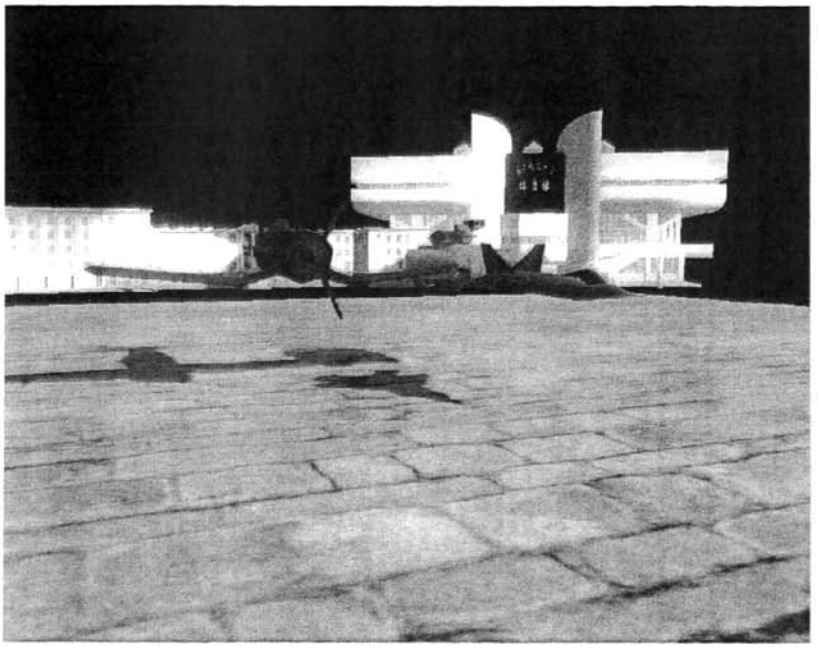
其运行分辨率为 1024×768。我们针对不同设置分别记录了整个场景渲染耗费的帧频，如表 6-1 所示。

表 6-1 帧率对比

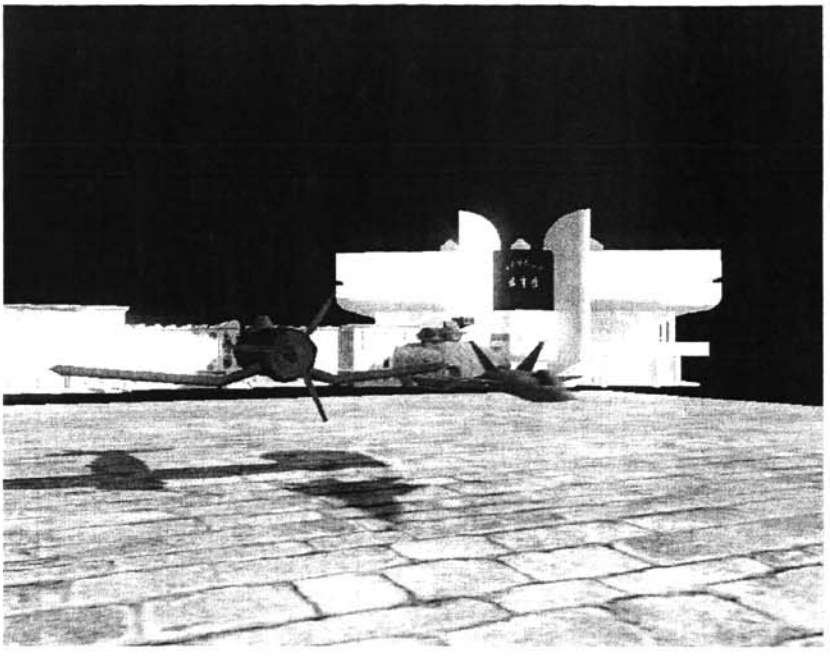
环境	平均帧率
非 HDR 环境	100.5 fps
HDR 环境	66.5 fps
HDR ＋ 景深	48.1 fps
HDR ＋ 运动模糊	60.1 fps
HDR ＋ 景深 ＋ 运动模糊	46.1 fps



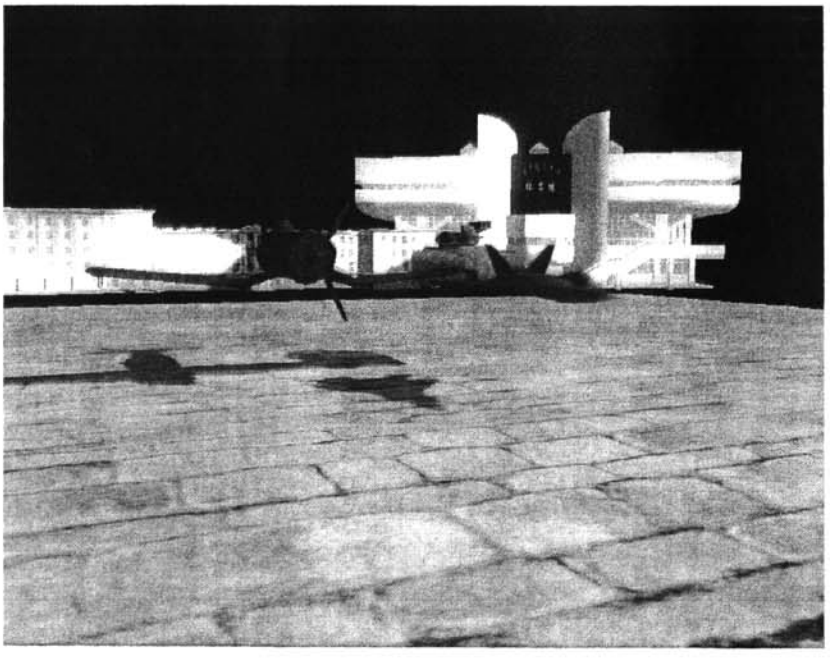
(a) 非高动态范围下场景　　(b) HDR 效果



(c) HDR 加景深效果



(d) HDR 加运动模糊效果



(e) HDR 加景深和运动模糊效果

图 6-6 各种特效综合效果

实验结果表明，没有加入高动态范围的场景的渲染帧率为 100.5fps，在高动态范围场景中加入景深模拟和运动模糊后，帧率从 66.5fps 分别降低到 48.1fps 和 60.1fps；在高动态范围场景中加入景深模拟和运动模糊后，帧率降低到 46.1fps。在本实验中，加入景深模拟比加入运动模糊对帧率的影响更大，这是因为在本场景中只有一个物体处于运动状态，而远景物体比较多。但是从帧率可以发现，我们可以满足实时渲染的要求。

6.5 本章小结

本章主要介绍了运动模糊的原理及在可编程图形硬件上的计算方法，通过将场景渲染至纹理，再将速度映射到图像空间，根据速度将原始图像的运动部分模糊达到运动模糊的效果。最后介绍了在高动态范围场景中对景深、运动模糊及两者的模拟，介绍了系统的实现流程以及实验数据和分析，实验表明在场景中加入高动态范围光照可以让场景中亮的地方真正亮起来，暗的的地方真正暗，并同时能够显示高亮和漆黑，增强了视觉效果；景深模拟实现了用户聚焦面上物体的成像清晰，远处和近处的物体成像模糊，给用户带来深度暗示和缓解虚拟现实系统中常有的眼睛疲劳，让场景显得更真实、更自然；运动模糊则实现了快速运动物体的真实模拟，可以应用于游戏和训练模拟等计算机图形学的实时应用中；高动态范围、景深和运动模糊三个特效的同时使用，更是增强了整个场景的真实感效果及用户的沉浸感，是未来增强虚拟场景真实感应用的一个方向。

第七章 总结与展望

本文结合网络游戏公共技术平台关键技术研究和支持数字媒体内容创作的集成环境项目，对真实感特效技术进行了深入的研究和探索，主要内容涉及：图形引擎技术、基于图形硬件的通用计算技术、高动态范围映射技术、实时景深模拟算法、实时运动模糊模拟算法，主要工作如下：

- 高动态范围绘制技术是真实感绘制的一项重要技术，它能增强场景的光照亮度和真实度。本文研究了高动态范围技术的相关理论和绘制方法，在游戏引擎中实现了高动态范围光照，增强了场景的视觉效果。
- 景深是人眼视觉系统中成像的重要特征，它能够使场景变得真实、自然且具有深度暗示。本文研究了景深的数学模型、相关理论以及绘制流程，以插件形式在游戏引擎中实现了景深模拟，有助于缓解虚拟现实系统中常有的眼睛疲劳，增强场景的真实感、沉浸感。
- 运动模糊是非无限小的相机快门的成像系统对物体运动物体进行拍摄时产生模糊图像的效果，是运动物体成像真实感的体现。本文在编程图形硬件上的实现了实时景深模拟，扩展了引擎特效。

当然，本文的工作也存在一些局限性，有很多方面可以作进一步研究，主要包括：

- 高动态范围的色阶映射技术方面。色调映射（Tone Mapping），它提供一种方式将现实场景的亮度值缩放或者映射到现实设备能显示的范围，除了压缩亮度范围，还必须保留原始图像的感观质量，即 HVS 敏感信息。本文采用了全局色阶映射技术，图像在细节、颜色、明亮程度上由一定的损失。因此，未来可对局部色阶重建技术作进一步研究，并集成到系统中。
- 景深模拟和运动模糊的计算精度方面。本文提出的景深模拟算法及采用的运动模糊模拟算法，都是在后处理阶段对纹理进行的操作，其精度与三维空间直接操作相比不高，针对这一情况，在日后的工作中，需要对模拟的精度进行进一步的提高。
- 场景建模方面。场景的建模是构建三维虚拟环境的关键之一和前提。在整个系统的开发过程中，我们采用了传统的基于几何的建模方法，速度

较慢。目前，基于图像和几何的混合建模方法已经有了广泛应用[55]，因此未来我们可以对此方法做进一步研究，应用到开发流程中。

致谢

在论文即将完成之际，我首先要感谢我的导师卢光辉以及孙世新教授。感谢卢老师和孙老师在我攻读硕士期间在学术上给予我的悉心指导，您们一丝不苟的工作作风、严谨的治学态度和对工作的无私奉献精神，深深地影响了我，使我受益匪浅。在此，谨向近三年来培养、帮助和关怀我的 X 老师和孙老师致以由衷的敬意和诚挚的谢意。

同时，我要特别感谢教研室的何明耘老师在科研过程中给予我的指导和支持，何老师平易近人但又学风严谨，是他引导我认识计算机图形学，逐步开展研究工作，享受科研的艰辛和快乐，这些足以使我受益终身。我也特别感谢数字媒体研究所的陈雷霆老师、蔡洪斌老师、房春兰老师、白忠建老师和邱航老师。感谢他们在工作和学习上给予我无私的帮助和支持。

同时我也要感谢数字媒体研究所游戏引擎小组与我一起工作和学习的所有同学：刘洋、杨洋、张钰靖、赵彬如、陈健、吴磊、罗恒希、刘煜岗，感谢他们在工作和学习上给予我的帮助和支持。另外，我还要感谢我的好朋友：李杨、何子昂、涂宇、郑义、林格非、邱毅川、余豪、郭学平、朱磊、周宇，感谢他们多年来与我一起追求人生的理想。

最后我要感谢我的父母，谢谢你们这么多年来的付出与教育。同时还要感谢我最亲爱的女朋友周祥，感谢你这四年来对我的支持和理解，没有你，我做的一切都没有任何意义。

参考文献

[1] Leonard, M. and B. Gary, Plenoptic modeling: an image-based rendering system, in Proceedings of the 22nd annual conference on Computer graphics and interactive techniques. 1995, ACM Press.

[2] H.-Y, S., et al., A review of image-based rendering techniques. SPIE proceedings series (SPIE proc. ser.) International Society for Optical Engineering proceedings series, 2000. 4067(3): p. 2-13.

[3] Marc, L., et al., Light field rendering, in Proceedings of the 23rd annual conference on Computer graphics and interactive techniques. 1996, ACM Press University of British Columbia, p36~49

[4] Fournier, et al., Common Illumination between Real and Computer Generated Scenes. 1992, University of British Columbia, p19~39

[5] Steven, J.G., et al., The lumigraph, in Proceedings of the 23rd annual conference on Computer graphics and interactive techniques. 1996, ACM Press, p 45~61

[6] Paul, D., Rendering synthetic objects into real scenes: bridging traditional and image-based graphics with global illumination and high dynamic range photography, in Proceedings of the 25th annual conference on Computer graphics and interactive techniques. 1998, ACM Press, p 46~78

[7] Mann and P.R. W., On Being `Undigital' With Digital Cameras: Extending Dynamic Range By Combining Differently Exposed Pictures Perceptual Computing Section Technical Report, 1995: p. TR-323.

[8] Mitsunaga, T. and S.K. Nayar, Radiometric Self Calibration IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'99), 1999. 1: p. 1374.

[9] Jurriaan D. Mulde, Robert van Liere. *Fast Perception-Based Depth of Field Rendering, in Proceedings of ACM VRST2000,* Seoul: ACM Press, 2000: 129~133

[10] Watt, A.H., 3D Computer Graphics third ed. p275-292.

[11] Bui Tuong, P., Illumination for computer-generated images:[docor thesis]. 1973

[12] James, F.B., Models of light reflection for computer synthesized pictures, in Proceedings of the 4th annual conference on Computer graphics and interactive techniques. 1977, ACM Press:

San Jose, California, p 125~137

[13] Paul, E.D. and M. Jitendra, *Recovering high dynamic range radiance maps from photographs*, in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. ACM Press/Addison-Wesley Publishing Co. 1997, p521-530.

[14] Robertson M A, Borman S, Stevenson R L. Estimation-theoretic approach to dynamic range enhancement using multiple exposures. Electronic Imaging, 2003, 12(2): 219-228.

[15] Aggarwal,M., Ahuja,N. High dynamic range panoramic imaging. In Proc.IEEE ICCV, 2001, 1: 2–9.

[16] Nayar,S.K., Mitsunaga,T. High dynamic range imaging:Spatially varying pixel exposures.In Proc.IEEE CVPR. 2000.

[17] Greg, W., A contrast-based scalefactor for luminance display, in Graphics gems IV. Boston: Academic Press, 1994: 415-421.

[18] Larson, G.W., H. Rushmeier, and C. Piatko, A Visibility Matching Tone Reproduction Operator for High Dynamic Range Scenes. IEEE Transactions on Visualization and Computer Graphics, 1997. 3: 12~34

[19] Tumblin, J., J.K. Hodgins, and a.B.K. Guenter, Two methods for display of high contrast images. ACM Trans. Graph, 1999. 18: 56-94.

[20] N.J.Miller,P.Y.Ngai,and D.D.Miller, The application of computer graphics in lighting design, Journal of the IES,1984, 14:6–26

[21] S.D.Upstill, The Realistic Presentation of Synthetic Images. PhD thesis, Computer Science Division,University of California,Berkeley,1985

[22] Jack Tumblin, Holly E.Rushmeier. Tone reproduction for realistic images. IEEE Computer Graphics&Applications, 1993, 13(6):42–48

[23] A.Scheel,etc.Tone reproduction for interactive walkthroughs. Computer Graphics Forum, 2000, 19(3):301–312

[24] 彭韬. 高动态图像视觉保真实时显示变换技术：[硕士学位论文]. 成都：电子科技大学，2005

[25] A.Oppenheim,etc. Nonlinear filtering of multiplied and convolved signals. In Proceedings of the IEEE, 1968, 56: 1264–1291

[26] Sumanta N.Pattanaik.etc. A multiscale model of adaptation and spatial vision for realistic image display. In Proceedings of SIGGRAPH 98, Computer Graphics Proceedings, Annual Conference Series, Orlando, Florida: ACM SIGGRAPH, 1998, 287–298

[27] K.Chiu, etc. Spatially nonuniform scaling functions for high contrast images. In Graphics Interface'93, Toronto,Ontario,Canada: Canadian Information Processing Society, 1993, 245–253

[28] D.J.Jobson,etc. A multiscale retinex for bridging the gap between color images and the human observation of scenes. IEEE Transactions on Image Processing, 1997, 6(7): 965–976

[29] Jack Tumblin and Greg Turk. Lcis:A boundary hierarchy for detail-preserving contrast reduction. In Proceedings of SIGGRAPH 99, Computer Graphics Proceedings, Annual Conference Series, Los Angeles, California: ACM SIGGRAPH/Addison Wesley Longman, 1999, 83–90

[30] Fr, D. do, and D. Julie, Fast bilateral filtering for the display of high-dynamic-range images, in Proceedings of the 29th annual conference on Computer graphics and interactive techniques. Texas: ACM Press, 2002, 14~25

[31] Raanan, F., L. Dani, and W. Michael, Gradient domain high dynamic range compression, in Proceedings of the 29th annual conference on Computer graphics and interactive techniques. Texas: ACM Press, 2002, 31~48

[32] Erik, R., et al., Photographic tone reproduction for digital images. ACM Trans. Graph., 2002. 21(3): 267-276.

[33] 袁亚杰. 基于可编程图形硬件的实时图形技术研究：[硕士学位论文]. 上海：上海师范大学，2006

[34] Randima Fernando. GPU 精粹——实时图形编程的技术、技巧和技艺, 姚勇，王小琴 译. 北京：人民邮电出版社, 2006, 238-249

[35] Edward Angel, Interactive Computer Graphics:A Top-Down Approach with OpenGL,Third Edition. Addison Wesley, Pearson, 2002: 11-12

[36] P. Rokita. Generating depth-of-field effects in virtual reality applications. IEEE Computer Graphics and Applications, 1996, 16(2): 18~21

[37] M. Potmesil and I. Chakravarty. A lens and aperture camera model for synthetic image generation, Dallas, 1981. Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH, 1981: 297~305

[38] M. Potmesil and I. Chakravarty. Synthetic image generation with a lens and aperture camera model. ACM Transactions on Graphics, 1982, 1(2) : 85~108

[39] Y.C. Chen. Lens effect on synthetic image generation based on light particle theory. The Visual Computer, 1987, 3(3):125~136

[40]周强, 彭俊毅, 戴树岭. 基于可编程图形处理器的实时景深模拟. 系统仿真学报, 2006, 18(8): 2219~2221,2238

[41]P.Rokata. Fast generation of depth of field effects in computer graphics. Computer Graphics, 1993, 17(5): 593~595

[42]J. Neider, T. Davis, and M. Woo. OpenGL Programming Guide: The Official Guide to Learning OpenGL. Reading Massachusetts: Addison-Wesley, 1993

[43]P. Haeberli and K. Akeley. The accumulation buffer: Hardware support for high-quality rendering, Dallas, 1990. Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH, 1990: 309~318

[44]Kolb C, Mitchell D, Pat Hanrahan. A realistic camera model for computer graphics, Los angles, 1995. Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH, 1995. 317~324

[45] Robert C , et al. Distributed ray tracing . Computer Graphics, 1984 , 18 (3) : 137~145

[46] 吴向阳, 鲍虎军, 陈为, 彭群生. 采用正向光线跟踪的照相机成像实时模拟. 计算机辅助设计与图形学学报, 2005, 17(7) : 1427-1433

[47] 张秉仁等, 运动模糊图像的降质过程分析与恢复技术研究，中国图像图形学报, 2004, 9(7):815-819

[48] Korein, J, and Badler, N, Temporal Anti-Aliasing in Computer Generated Animation, In *Proceedings of SIGGRAPH 1983*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM, 1983

[49] Dippe, M., and Wold, E, Antialiasing Through Stochastic Sampling, *In Proceedings of SIGGRAPH 1985*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM, 1985

[50] M. Potmesil and I. Chakravarty. Modelling Motion Blur In Computer-Generated Images, Computer Graphics (SIGGRAPH'83 Proceedings), 1983, 17(3):389–399

[51] Dachille, F., and Kaufman, A, High-Degree Temporal Antialiasing, *Proceedings of Computer Animation 2000*, 2000: 49-54

[52] Haeberli P., and Akeley K., The Accumulation Buffer: Hardware Support for High-Quality Rendering, SIGGRAPH 1990, 1990: 309-318.

[53] Shimizu C., Shesh A., Chen B.: Hardware Accelerated Motion Blur Generation. EUROGRAPHICS 2003, 22(3)

[54] Wloka, M., and Zeleznik, R, Interactive Real-Time Motion Blur. *The Visual Computer*, 1996, 12(6): 283-295.

[55] 潘志庚，姜晓红等, 分布式虚拟环境综述, 软件学报, 2000，11(4):461-467

个人简历及硕士期间发表的论文

个人简历

黄蓝枭(1982-)，男，四川大邑人。2005 年 7 月毕业于电子科技大学软件学院，获工学学士学位。于 2005 年 9 月保送进入电子科技大学计算机科学与工程学院攻读工学硕士学位。主要研究方向为：计算机图形学、3D 游戏引擎。

科研情况

1. 国家 863 项目：支持数字媒体内容创作的集成环境
2. 国家 863 项目：网络游戏公共技术平台关键技术研究
3. 合作项目：基于 P2P 的实时流媒体系统

获奖情况

2007 年 10 月 获电子科技大学优秀研究生三等奖学金

发表论文

[1]黄蓝枭，卢光辉等，基于 GPU 的实时景深模拟，计算机应用研究(已录用).