GPU Programming And Cg Language Primer 1rd Edition

GPU 编程与 CG 语言之阳春白雪下里巴人

康玉之

kang_yu_zhi@yahoo.cn

半山工作室

2009 年 9 月

半山工作室出品

题　　目 **GPU 编程与 CG 语言之阳春白雪下里巴人**

英　　文 **GPU Programming And Cg Language Primer 1rd Edition**

作者姓名：康玉之

专　　业：计算机科学与技术　　研究方向：多媒体与图形学
邮　　箱：kang_yu_zhi@yahoo.cn

书籍报告提交日期 2009 年 9 月

独 创 性 声 明

本人声明所呈交的书籍是我个人在学习和研究工作中取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，书中中不包含其他人已经发表或撰写过的研究成果。

签名：_____ 日期：_____

关于本书使用授权的说明

任何人和组织有权保留本书的复印件，允许本书被查阅和借阅；任何人或组织可以公布论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存论文。

签名：_____ 日期：_____

序

26 年前，江汉平原的一个农家女人在身边无人的情况下生下了一名男婴，有人看到后叫来了男孩的外婆，外婆剪断了脐带。起初很多人都以为会是一个女孩，都劝她打掉这个孩子，而她还是坚持生了下来。

男孩 4 岁时，由于家里忙没有功夫照看，于是女人用一卷毯子裹着他去学校报名读书，说他已经 7 岁了。

男孩 17 岁那年，考上了武汉大学。爸爸妈妈一起送他到武汉读书，爸爸妈妈还很有力气，扛着几十斤的行李很轻松，男孩那个时候不知道麦当劳，不知道电脑如何开机，不知道什么叫上网。他们对男孩说，要好好读书，多学东西，和同学好好相处，到了大三可以找女朋友。

5 年前，他来到了北京读研。爸爸妈妈一起送他到北京，那个时候他们头发还没有完全白，扛着行李也觉得有点吃力了，到颐和园玩之前先带上水、方便面和几个馒头。他们对男孩说，等他毕业时，再来北京玩，要慢慢适应北方的天气，多喝水，用菊花和甘草泡水喝，要注意穿衣，不要让人瞧不起。

2 年前，他毕业了，在一家研究所找了一份工作，拿到了北京户口，爸爸妈妈来北京玩了一次，打算一家人在北京过年，他们的头发白了不少，拿着行李很辛苦，行李里面是吃的、锅、中草药、被子等。他们说，你要做一个好人，亲戚有事情，你要帮忙，不要骄傲，要攒钱买房子，准备结婚。

你们赐予我生命，一直坚持要我和姐姐都上大学。对于一个普通的家庭而言，同时负担两个孩子上大学，是一件多么艰难的事情！

二十多年过去有如瞬刹的流水，感谢你们，我的父亲母亲。

前言

在我读研期间，对国内的研究风气经历了从期待、到失望的过程。原因在于，工程化的迹象太过明显，我的观点并不排斥工程化，但是以研究的名义，行工程化之实，并将工程化中根深蒂固的“急功近利”带到整个过程中，最后不得不在匆忙中以“略可充数的软件”和“流于表面的技术论文，粉饰以复杂深奥的术语和图表”来结尾，这实在是一种悲哀。对于广大的研究人员和学生而言，已经是一种无奈。

很多人都说，这是教授的错，也有很多人都说，这是学生的错，学风不正！我最开始也这么认为，随着所见愈多，却又困惑。其实大部分的教授都是希望有一番作为的，也是希望可以严格要求学生的，但是现在的大趋势是“权力决定学术”以及“钻营决定利益”，于是很少有人可以在“所有人都进行利益争食”时，还静心研究，实际上，就算静心研究，也没有这个条件。研究过程需要经费进行保证，而不去钻营，不去附和哪些狗屁不懂的人，就没有办法拿到经费。大量的课题原本应该从实际需求中提炼出来的，但随着行政体制的流转，最后演变成为了“媚上”和“附和”而进行的选美比赛。而申请材料的撰写目标，也变成了为了好看和吸引眼球而组建的“花哨”课题。

除此之外，研究人员难！很多研究工作需要依靠博士，但是中国博士的待遇低，前途未卜是众所周知的。一个花哨的课题，加上一个每天想着结婚、吃饭的穷博士，结果是可想而知的。

我工作之后发现，与学校的科研情况相比，社会上的 IT 混乱情况有过之而无不及。刚工作初，曾在一次很大的项目验收中听到这么一句话“软件鉴定会就是软件追悼会”。再加上国人不重基础研究，在我工作之初，一个很大的苦恼是，很多人都觉得计算机图形学不属于计算机科学，他们认为计算机图形学只是“使用软件去建模，是美工”，当有人发现我从本科到研究生都是计算机系的学生时，还非常夸张的表达了他们的惊讶。更有一个领导对我说“计算机图形学是一门很小的学科……”，她似乎觉得看过人月神话、软件工程之类的书，就是大牛了，才是真正的计算机科学。

这些情况让我困惑了很久，我的师兄师弟师姐师妹，目前还在计算机图形学领域的人，已经屈指可数。回想这 5 年来的所经所历，得出的结论是：中国的科学研究已然被功利化、制度化。

被功利化的学术，失去了方向，变得鼠目寸光，无法容忍长时间的基础积累，一味幻想空中楼阁式的成功。是呀，没有学术的成功，哪有领奖的资格，哪有白花花的银子？我坚决认为，中国必然存在很多像爱迪生一样的天才，但是中国没有一个上位者可以容忍一千次的失败，所以中国的爱迪生无法发明电灯，也无法用智慧照亮别人和自己！

被制度化的学术失去了灵性和创造性，而奴性则越来越重！处于这种氛围的人会从最初的挣扎到慢慢的习惯，到最后成了坚定的捍卫者，因为他们已然失去了活力，已然要依赖这种制度而生存，就好比长在大树上的寄生植物一般。这些人每天说着“假大空”的“真善美”，创造性实际上变成了嘴巴上的大跃进。也确实是嘴巴上的大跃进，从东吃到西，从南吃到北，从河鲜吃到海鲜，油乎乎的嘴巴中一个一个世界空白被填补。

这种现状和柏林墙很像呀！曾看过一段关于柏林墙的评价，堪称经典：

世界上的围墙都是防止外面的人闯进来，只有一种围墙是防止里面的人出去的，那是什么？那就是监狱的围墙！在这样的墙里面是什么地方？那不就是监狱吗？

我无意去批评任何人，这种现状的产生并非一朝一夕，一人之力。在这个漩涡中的每个人都身不由己，被动或主动的去忽略科学，正视利益。

科学工作者没有独立的人格和意愿，面对的只是官本位和市场经济的强烈压力，在这种情况下，中国想要获得诺贝尔奖实在是天方夜谭！

请原谅我发这么多牢骚，不过此书正是在有点怨气的情况下写出的。我想让更多的人了解计算机图形学，了解 GPU 编程，而不是认为那是很遥远很飘渺的事情。当我查询 Cg 语言的文献时，NVIDIA 公司出的 CgUsersManual 一书已经有了

日文版的翻译，而中文版的却还未出现；当我研究体绘制算法时，到处寻找用于实验的体数据，在中文网站上未曾发现。

虽然我一向要求自己做一块坚硬的石头，但是面对不确定的未来难免有惴惴不安之感。正所谓人生失意无南北，或许有一天我的职业生涯走到了无路可回的境地，不得不放弃自己的专业，今天留下点回忆也不错。

最后，愿中国的学术可以“天地开辟，日月重光”!

天地漫漫，浩瀚史诗
金黄，那神性灿烂的光芒
没有一种爱
没有一种情
可以在自由之上
星辰如海啊
请听我倾说
我的爱人很不一般
那是天地之外的逍遥自在

第 1 章　绪论 13
1.1 Programmable Graphics Processing Unit 发展历程 13
1.2 GPU VS CPU 15
1.3 国内外研究现状 18
1.4 本书主要内容和结构 18
第 2 章 GPU 图形绘制管线 21
2.1 几何阶段 22
2.1.1 从 object space 到 world space 23
2.1.2 从 world space 到 eye space 24
2.1.3 从 eye space 到 project and clip space 25
2.2 Primitive Assembly && Triangle setup 26
2.3 光栅化阶段 27
2.3.1 Rasterization 27
2.3.2 Pixel Operation 28
2.4 图形硬件 30
2.4.1 GPU 内存架构 30
2.4.2 Z Buffer 与 Z 值 31
2.4.3 Stencil Buffer 33
2.4.4 Frame Buffer 34
2.5 本章小节 35
第 3 章 Shader Language 36
3.1 Shader Language 原理 37
3.2 Vertex Shader Program 39
3.3 Fragment Shader Program 40
3.4 CG VS GLSL VS HLSL 41
3.5 本章小结 43
第 4 章 Cg 语言概述 44
4.1 开始 Cg 之旅 44
4.2 CG 特性 45

4.3 CG 编译 45
4.3.1 CG 编译原理 45
4.3.1 CGC 编译命令 47
4.4 CG Profiles 50
第 5 章 CG 数据类型 53
5.1 基本数据类型 53
5.2 数组类型 55
5.3 结构类型 56
5.4 接口（Interfaces）类型 58
5.5 类型转换 58
第 6 章 CG 表达式与控制语句 60
6.1 关系操作符（Comparison Operators） 60
6.2 逻辑操作符（Logical Operators） 61
6.3 数学操作符（Math Operators） 62
6.4 移位操作符 62
6.5 Swizzle 操作符 63
6.6 条件操作符（Conditional Operators） 64
6.7 操作符优先顺序 65
6.8 控制流语句（Control Flow Statement） 66
第 7 章 输入\输出与语义绑定 67
7.1 Cg 关键字 67
7.2 uniform 68
7.3 const 69
7.4 输入\输出修辞符（in\out\inout） 69
7.5 语义词（Semantic）与语义绑定（Binding Semantics） 70
7.5.1 输入语义与输入语义的区别 71
7.5.2 顶点着色程序的输入语义 72
7.5.3 顶点着色程序的输出语义 73
7.5.4 片段着色程序的输出语义 75

7.5.5 语义绑定方法.....75
第 8 章 函数与程序设计.....79
8.1 函数.....79
8.1.1 数组形参.....80
8.2 函数重载.....81
8.3 入口函数.....82
8.4 CG 标准函数库.....83
8.4.1 数学函数（Mathematical Functions）.....84
8.4.2 几何函数（Geometric Functions）.....87
8.4.3 纹理映射函数（Texture Map Functions）.....88
8.4.4 偏导函数（Derivative Functions）.....91
8.4.5 调试函数（Debugging Function）.....94
8.5 在未来将被解决的问题.....94
开篇语：.....96
第 9 章 经典光照模型（illumination model）.....97
9.1 光源.....97
9.2 漫反射与 Lambert 模型.....98
9.2.1 漫反射渲染.....99
9.3 镜面反射与 Phong 模型.....102
9.3.1 phong 模型渲染.....103
9.4 Blinn-Phong 光照模型.....107
9.5 全局光照模型与 Rendering Equation.....109
9.6 本章小结.....110
第 10 章 高级光照模型.....111
10.1 Cook-Torrance 光照模型.....112
10.1.1 Cook-Torrance 光照模型渲染实现.....116
10.2 BRDF 光照模型.....118
10.2.1 什么是 BRDF 光照模型.....118
10.2.2 什么是各向异性.....119

10.3 Bank BRDF 经验模型.....120
10.4 本章小结.....123
第 11 章 透明光照模型与环境贴图.....124
11.1 Snell 定律与 Fresnel 定律.....125
11.1.1 折射率与 Snell 定律.....125
11.1.2 色散.....126
11.1.3 Fresnel 定律.....127
11.2 环境贴图.....128
11.3 简单透明光照模型.....132
11.4 复杂透明光照模型与次表面散射.....136
第 12 章 投影纹理映射（Projective Texture Mapping）.....137
12.1 投影纹理映射的优点.....137
12.2 齐次纹理坐标（Homogeneous Texture Coordinates）.....139
12.3 原理与实现流程.....139
12.4 本章小结.....143
第 13 章 Shadow Map.....144
13.1 什么是 depth map.....144
13.2 Shadow map 与 shadow texture 的区别.....145
13.3 Shadow map 原理与实现流程.....146
第 14 章 体绘制（Volume Rendering）概述.....152
14.1 体绘制与科学可视化.....153
14.2 体绘制应用领域.....153
14.3 体绘制与光照模型.....154
14.4 体数据（Volume Data）.....155
14.4.2 体素（Voxel）.....156
14.4.1 体纹理（Volume Texture）.....157
14.5 体绘制算法.....159
第 15 章 光线投射算法（Ray Casting）.....160
15.1 光线投射算法原理.....160

15.1.1 吸收模型.....161
15.2 光线投射算法若干细节之处.....161
15.2.1 光线如何穿越体纹理.....161
15.2.2 透明度、合成.....163
15.2.3 沿射线进行采样.....164
15.2.2 如何判断光线投射出体纹理.....166
15.3 算法流程.....168
15.4 光线投射算法实现.....169
15.5 本章小结.....172
附录 A 齐次坐标.....173
附录 B：体绘制的医学历程.....175
附录 C：模板阴影（Stencil Shadow）.....178
参考文献.....182

Alice 羞涩地说：“咄牙猫，请问你能告诉我应该走哪条路吗？”咄牙猫说：“这取决于你想去哪儿。”Alice 说：“我不怎么介意到哪儿去。”咄牙猫回答说：“那你走哪条路都可以”Alice 说：“只要我能到某个地方”咄牙猫说：“只要你走的足够远，你肯定能到某个地方”。

-------Lewis Carroll 的 Alice’s adventures in Wonderland

借用上面的一句话，只要你走的足够远，你肯定能到某个地方。加油！

第1章　绪论

面纱掩盖了过去、现在和将来，历史学家的使命是发现它现在是什么，而不是过去是什么。

------Henry David Thoreau

1.1 Programmable Graphics Processing Unit 发展历程

Programmable Graphics Processing Unit（GPU），即可编程图形处理单元，通常也称之为可编程图形硬件。

GPU 概念在 20 世纪 70 年代末和 80 年代初被提出，使用单片集成电路（monolithic）作为图形芯片，此时的 GPU 已经被用于视频游戏和动画方面，它能够很快地进行几张图片的合成（仅限于此）。在 20 世纪 80 年代末到 90 年代初这段时间内，基于数字信号处理芯片（digital signal processor chip）的 GPU 被研发出来，与前代相比速度更快、功能更强，当然价格是非常的昂贵。在 1991 年，S3 Graphics 公司研制出第一个单芯片 2D 加速器，到了 1995 年，主流的 PC 图形芯片厂商都在自己的芯片上增加了对 2D 加速器的支持。与此同时，固定功能的视窗加速器（fixed-function Windows accelerators）由于其高昂的价格而慢慢退出 PC 市场。

1998 年 NVIDIA 公司宣布 modern GPU 的研发成功，标志着 GPU 研发的历史性突破成为现实。通常将 20 世纪 70 年代末到 1998 年的这一段时间称之为 pre-GPU 时期，而自 1998 年往后的 GPU 称之为 modern GPU。在 pre-GPU 时期，一些图形厂商，如 SGI、Evans & Sutherland，都研发了各自的 GPU，这些 GPU 在现在并没有被淘汰，依然在持续改进和被广泛的使用，当然价格也是非常的高昂。

modern GPU 使用晶体管（transistors）进行计算，在微芯片（microchip）中，GPU 所使用的晶体管已经远远超过 CPU。例如，Intel 在 2.4GHz 的 Pentium IV 上使用 5 千 5 百万（55 million）个晶体管；而 NVIDIA 在 GeForce FX GPU 上使用超过 1 亿 2 千 5 百万（125 million）个晶体管，在 NVIDDIA 7800 GXT 上的晶

体管达到 3 亿 2 百万（302 million）个。

回顾 Modern GPU 的发展历史，自 1998 年后可以分为 4 个阶段。NVIDIA 于 1998 年宣布 Modern GPU 研发成功，这标志着第一代 Modern GPU 的诞生，第一代 Modern GPU 包括 NVIDIA TNT2，ATI 的 Rage 和 3Dfx 的 Voodoo3。这些 GPU 可以独立于 CPU 进行像素缓存区的更新，并可以光栅化三角面片以及进行纹理操作，但是缺乏三维顶点的空间坐标变换能力，这意味着“必须依赖于 GPU 执行顶点坐标变换的计算”。这一时期的 GPU 功能非常有限，只能用于纹理组合的数学计算或者像素颜色值的计算。

从 1999 到 2000 年，是第二代 modern GPU 的发展时期。这一时期的 GPU 可以进行三维坐标转换和光照计算（3D Object Transformation and Lighting, T&L），并且 OpenGL 和 DirectX7 都提供了开发接口，支持应用程序使用基于硬件的坐标变换。这是一个非常重要的时期，在此之前只有高级工作站（workstation）的图形硬件才支持快速的顶点变换。同时，这一阶段的 GPU 对于纹理的操作也扩展到了立方体纹理（cube map）。NVIDIA 的 GeForce256，GeForce MAX，ATI 的 Radeon 7500 等都是在这一阶段研发的。

2001 年是第三代 modern GPU 的发展时期，这一时期研发的 GPU 提供 vertex programmability（顶点编程能力），如 GeForce 3，GeForce 4Ti，ATI 的 8500 等。这些 GPU 允许应用程序指定一个序列的指令进行顶点操作控制（GPU 编程的本质！），这同样是一个具有开创意义的时期，这一时期确立的 GPU 编程思想一直延续到 2009 年的今天，不但深入到工程领域帮助改善人类日常生活（医疗、地质勘探、游戏、电影等），而且开创或延伸了计算机科学的诸多研究领域 （体绘制、光照模拟、人群动画、通用计算等）。同时，Direct8 和 OpenGL 都本着与时俱进的精神，提供了支持 vertex programmability 的扩展。不过，这一时期的 GPU 还不支持像素级的编程能力，即 fragment programmability（片段编程能力），在第四代 modern GPU 时期，我们将迎来同时支持 vertex programmability 和 fragment programmability 的 GPU。

第四代 modern GPU 的发展时期从 2002 年末到 2003 年。NVIDIA 的 GeForce FX 和 ATI Radeon 9700 同时在市场的舞台上闪亮登场，这两种 GPU 都支持 vertex

programmability 和 fragment programmability。同时 DirectX 和 OpenGL 也扩展了自身的 API，用以支持 vertex programmability 和 fragment programmability。自 2003 年起，可编程图形硬件正式诞生，并且由于 DirectX 和 OpenGL 锲而不舍的追赶潮流，导致基于图形硬件的编程技术，简称 GPU 编程，也宣告诞生。恭喜 GeForce 和 ATI 的硬件研发人员，你们终于可以歇口气了，不用较着劲地出显卡了，同时也恭喜 DirectX 和 OpenGL 的研发人员，你们也可以休息下了，不用斗鸡一般的工作了，最后恭喜广大工作在图形图像领域的程序员，你们可以继续学而不倦。

目前最新的可编程图形硬件已经具备了如下功能：

1. 支持 vertex programmability 和 fragment programmability；
2. 支持 IEEE32 位浮点运算；
3. 支持 4 元向量，4 阶矩阵计算；
4. 提供分支指令，支持循环控制语句；
5. 具有高带宽的内存传输能力（>27.1GB/s）；
6. 支持 1D、2D、3D 纹理像素查询和使用，且速度极快；
7. 支持绘制到纹理功能（Render to Texture，RTT）。

关于 GPU 发展历史的相关数据参考了 Feng Liu 的“Platform Independent Real-time X3D Shaders and Their Applications in Bioinformatics Visualization”一文

1.2 GPU VS CPU

从上节阐述了 GPU 的发展历史，那么为什么在 CPU 之外要发展 GPU？GPU 的 vertex programmability 和 fragment programmability 究竟在何处有着怎样的优势？引用在文献【2】第 6 页的一段话为：

Modern GPUs implement a number of graphics primitive operations in a way that make running them much faster than drawing directly to the screen with the host CPU. They are efficient at manipulating and displaying computer graphics, and their highly parallel structure makes them more effective than typical CPUs for a range of

complex algorithms.

这段话的意思是，由于 GPU 具有高并行结构（highly parallel structure），所以 GPU 在处理图形数据和复杂算法方面拥有比 CPU 更高的效率。图 1 GPU VS CPU 展示了 GPU 和 CPU 在结构上的差异，CPU 大部分面积为控制器和寄存器，与之相比，GPU 拥有更多的 ALU（Arithmetic Logic Unit，逻辑运算单元）用于数据处理，而非数据高速缓存和流控制，这样的结构适合对密集型数据进行并行处理。CPU 执行计算任务时，一个时刻只处理一个数据，不存在真正意义上的并行（请回忆 OS 教程上的时间片轮转算法），而 GPU 具有多个处理器核，在一个时刻可以并行处理多个数据。

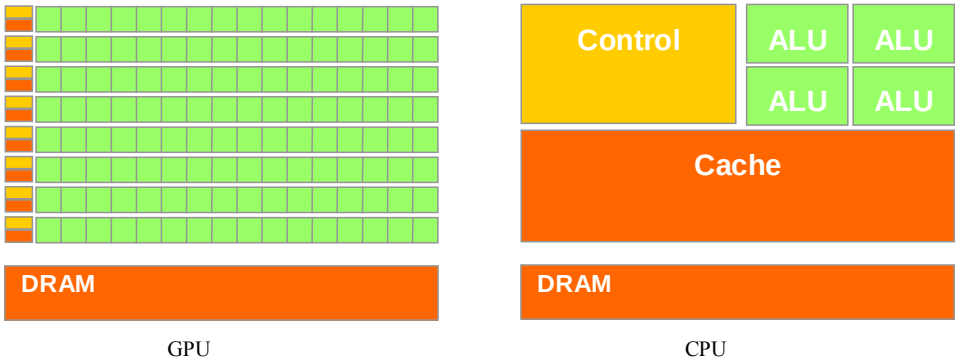


图 1 GPU VS CPU

GPU 采用流式并行计算模式，可对每个数据进行独立的并行计算，所谓“对数据进行独立计算”，即，流内任意元素的计算不依赖于其它同类型数据，例如，计算一个顶点的世界位置坐标，不依赖于其他顶点的位置。而所谓“并行计算”是指“多个数据可以同时被使用，多个数据并行运算的时间和 1 个数据单独执行的时间是一样的”。图 2 中代码目的是提取 2D 图像上每个像素点的颜色值，在 CPU 上运算的 C++代码通过循环语句依次遍历像素；而在 GPU 上，则只需要一条语句就足够。

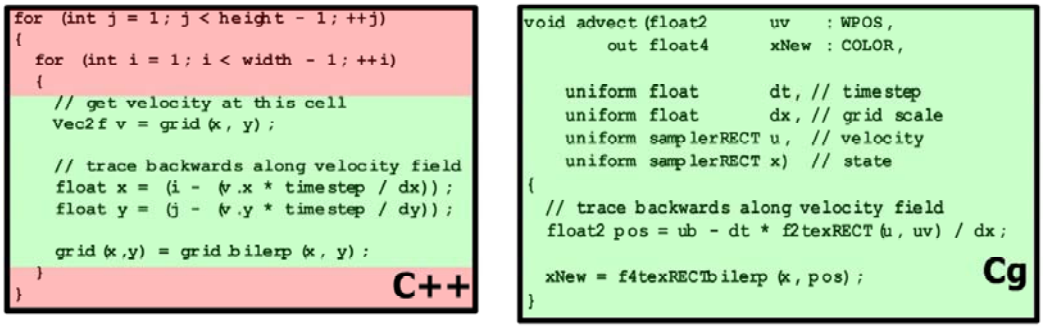


图 2 CPU 和 GPU 上的代码比较

可能有人会问道：既然 GPU 在数据处理速度方面远胜 CPU，为什么不用 GPU 完全取代 CPU 呢？实际上，关于 GPU 取代 CPU 的论调时有出现，但是作者本人并不同意这种观点，因为 GPU 在许多方面与 CPU 相比尚有不如。

首先，虽然 GPU 采用数据并行处理方式极大加快了运算速度，但正是由于“任意一个元素的计算不依赖于其它同类型数据”，导致“需要知道数据之间相关性的”算法，在 GPU 上难以得到实现（但在 CPU 上则可以方便的实现），一个典型的例子是射线与不规则物体的求交运算。

此外，GPU 在控制流方面弱于 CPU，在图中可以看到，GPU 中的控制器少于 CPU，而控制器的主要功能是取指令，并指出下一条指令在内存中的位置，控制和协调计算机的各个部件有条不紊地工作。 在早期的 OpenGL fp2.0，fp3.0 以及 DirectX 的 ps_4_0 之前的 profile 版本都不支持或不完全支持循环控制流语句（目前在软硬件方面都已得到改进）。由于 GPU 编程完全依赖于图形硬件，故而较早版本的 GPU 并不支持一些常用的编程需要，而现在很多个人电脑或者公司的电脑的更新换代并没有那么快（考虑个人电脑的使用寿命在 4-6 年，所以在 2012 之后，旧式显卡的更新换代会基本结束），这也制约了 GPU 编程技术的使用。

最后进行 GPU 编程必须掌握计算机图像学相关知识，以及图形处理 API，入门门槛较高，学习周期较长，尤其国内关于 GPU 编程的资料较为匮乏，这些都导致了学习的难度。在早期，GPU 编程只能使用汇编语言，开发难度高、效率低，不过，随着高级 Shader language 的兴起，在 GPU 上编程已经容易多了。

1.3 国内外研究现状

基于 GPU 的科学可视化计算（Visualization in Scientific Computing），在研究和工程运用上都取得了卓越的成果。由于科学可视化计算处理的数据量极大（人体 CT、地质勘探、气象数据、流体力学等），仅仅基于 CPU 进行计算完全不能满足实时性要求，而在 GPU 上进行计算则可以在效率上达到质的突破，许多在 CPU 上非常耗时的算法，如体绘制中的光线投射算法，都可以成功移植到 GPU 上，所以基于 GPU 的科学可视化研究目前已经成为主流。

近年来，基于GPU进行通用计算的研究逐渐成为热点，被称之为GPGPU（General-Purpose Computing on Graphics Processing Units,也被称为GPGP，或 GP^2 ），很多数值计算等通用算法都已经在GPU上得到了实现，并有不俗的性能表现，目前，线性代数（linear algebra）【kruger and westermann 2003】，物理仿真（physical simulation）【Harris et al. 2003】和光线跟踪算法（ray tracer）【Purcell et al.2002； Carr et al. 2002】都已经成功的移植到GPU上。在国内，中国科学院计算技术研究所进行了基于GPU的串匹配算法的实现【29】。关于GPGPU的更多知识点可以参阅网站http://gpgpu.org/

旨在降低 GPU 编程难度，设计基于 GPU 的高级程序语言的研究同样进行的如火如荼。2004 年，斯坦福大学研究的 BROOKGPU 项目设计了一个实时的编译器，编程人员不需掌握图形学知识，只需掌握与 C 语言类似的流处理语言 BROOK，即可进行基于 GPU 的通用编程开发。目前 BROOKGPU 已经在 AMD 公司进行深入研发。国内浙江大学计算机学院针对高级着色语言的编译系统【30】，以及可编程图形硬件的加速等技术进行了研究。

1.4 本书主要内容和结构

本书旨在引导初级 GPU 学习者步入 GPU 编程的大堂，并普及一些在国内资料中较少见到的 GPU 算法，例如光照渲染中的 bank BRDF，以及体绘制中的光线投射（ray-casting）算法。在 GPU 编程方面有一定基础的同学，可以将本书的一些观点作为参考。

本书并非网络小说，也非“立意新奇，饰以深奥文字，佐以华丽图表的国际论文”，而是作者有感于国内 GPU 研究现状堪忧，而抛砖引玉之作，故而以通俗的语言说出自己明白的事情，是我写作的原则。

本书的框架划分，也是希望可以循序渐进、深入浅出的让大家了解 GPU 世界。

本书由四大部分组成，第一部分阐述“GPU 的发展历史、GPU 和 CPU 的优劣比较、GPU 的图形绘制管线，以及在 GPU 上使用的 shader language”，这一部分由 3 章组成，尤为重要的是第二章“GPU 图形绘制管线”和第三章“Shader Language”。GPU 图形绘制管线描述了 GPU 的工作原理，这部分知识是 GPU 编程的铸基之石；而 Shader Language 章节阐述了 vertex program（顶点编程）和 fragment program（片段编程）在 GPU 管线中的位置、作用和工作机制。

本书的第二部分讲述 Cg 语言的使用方法，由五章组成（第四章到第八章）。这部分的知识以 NVIDIA 出版的 The Cg Tutorial The Definitive Guide to Programmable Real-Time Graphics 和 Cg ToolKit User’s Manual 作为基础，并加入了作者本人在实践中的一些总结。The Cg Tutorial The Definitive Guide to Programmable Real-Time Graphics 已经有了中文版本，名为《Cg 教程_可编程实时图形权威指南》，不过个人觉得，由于此书的英文原版在组织形式上有些不合理之处，且在知识点上并不完善，故此，并不推荐初学者先阅读此书。Cg ToolKit User’s Manual 一书，目前还没有中文版（却存在日文版!），此书在语法的阐述上胜于前者，英文好的读者可以尝试着阅读一下。

本书的第三部分阐述光照模型知识，由三章组成（第九章到第十一章）。这部分首先以较为简单的光照模型作为 GPU 编程的实践理论，让读者从实际编程中学习 Cg 语言的使用方法，然后介绍较为高级的 BRDF 光照模型，以及透明光照模型。BRDF 光照模型的知识点在国内的书籍中并不常见，实现代码更是没有看到过，希望这一章节对这方面的研究人员略有帮助。

本书的第四部分针对投影纹理映射和阴影算法进行讲解，由两章组成（第十二章和第十三章），这部分的知识希望可以引起大家足够的重视，因为投影纹理映射和深度值的使用方法，都是既基础又重要的知识，在更高一级的前沿研究课

题（如体绘制、软阴影渲染）中常被使用。

本书的第五部分阐述了体绘制知识点以及基于 GPU 的光线投射算法。体绘制是我花费研究时间较多的地方，不但因为技术本身较为复杂，而且因为体数据收集和使用较为困难。体绘制技术的中文资料少之又少，优秀硕博士论文库上一般也是人云亦云，可能高手都比较含蓄，所以我姑且写出两章，如有可用之处，则可慰我心，如不堪入目，聊供方家一笑。

第 2 章 GPU 图形绘制管线

图形绘制管线描述 GPU 渲染流程，即“给定视点、三维物体、光源、照明模式，和纹理等元素，如何绘制一幅二维图像”。本章内容涉及 GPU 的基本流程和实时绘制技术的根本原理，在这些知识点之上才能延伸发展出基于 GPU 的各项技术，所以本章的重要性怎么说都不为过。欲登高而穷目，勿筑台于浮沙！

本章首先讨论整个绘制管线（不仅仅是 GPU 绘制）所包含的不同阶段，然后对每个阶段进行独立阐述，最后讲解 GPU 上各类缓冲器的相关知识点。

在《实时计算机图形学》一书中，将图形绘制管线分为三个主要阶段：应用程序阶段、几何阶段、光栅阶段。

应用程序阶段，使用高级编程语言（C、C++、JAVA 等）进行开发，主要和 CPU、内存打交道，诸如碰撞检测、场景图建立、空间八叉树更新、视锥裁剪等经典算法都在此阶段执行。在该阶段的末端，几何体数据（顶点坐标、法向量、纹理坐标、纹理等）通过数据总线传送到图形硬件（时间瓶颈）；数据总线是一个可以共享的通道，用于在多个设备之间传送数据；端口是在两个设备之间传送数据的通道；带宽用来描述端口或者总线上的吞吐量，可以用每秒字节（b/s）来度量，数据总线和端口（如加速图形端口，Accelerated Graphic Port,AGP）将不同的功能模块“粘接”在一起。由于端口和数据总线均具有数据传输能力，因此通常也将端口认为是数据总线（实时计算机图形学 387 页）。

几何阶段，主要负责顶点坐标变换、光照、裁剪、投影以及屏幕映射（实时计算机图形学 234 页），该阶段基于 GPU 进行运算，在该阶段的末端得到了经过变换和投影之后的顶点坐标、颜色、以及纹理坐标（实时计算机图形学 10 页）。

光栅阶段，基于几何阶段的输出数据，为像素（Pixel）正确配色，以便绘制完整图像，该阶段进行的都是单个像素的操作，每个像素的信息存储在颜色缓冲器（color buffer 或者 frame buffer）中。

值得注意的是：光照计算属于几何阶段，因为光照计算涉及视点、光源和物体的世界坐标，所以通常放在世界坐标系中进行计算；而雾化以及涉及物体透明度的计算属于光栅化阶段，因为上述两种计算都需要深度值信息（Z 值），而深

度值是在几何阶段中计算，并传递到光栅阶段的。

下面具体阐述从几何阶段到光栅化阶段的详细流程。

2.1 几何阶段

几何阶段的主要工作是“变换三维顶点坐标”和“光照计算”，显卡信息中通常会有一个标示为“T&L”硬件部分，所谓“T&L”即 Transform & Lighting。那么为什么要对三维顶点进行坐标空间变换？或者说，对三维顶点进行坐标空间变换有什么用？为了解释这个问题，我先引用一段文献【3】中的一段叙述：

Because, your application supplies the geometric data as a collection of vertices, but the resulting image typically represents what an observer or camera would see from a particular vantage point.

As the geometric data flows through the pipeline, the GPU’s vertex processor transforms the continuant vertices into one or more different coordinate system, each of which serves a particular purpose. CG vertex programs provide a way for you to program these transformations yourself.

上述英文意思是：输入到计算机中的是一系列三维坐标点，但是我们最终需要看到的是，从视点出发观察到的特定点（这句话可以这样理解，三维坐标点，要使之显示在二维的屏幕上）。一般情况下，GPU 帮我们自动完成了这个转换。基于 GPU 的顶点程序为开发人员提供了控制顶点坐标空间转换的方法。

一定要牢记，显示屏是二维的，GPU 所需要做的是将三维的数据，绘制到二维屏幕上，并到达“跃然纸面”的效果。顶点变换中的每个过程都是为了这个目的而存在，为了让二维的画面看起具有三维立体感，为了让二维的画面看起来“跃然纸面”。

根据顶点坐标变换的先后顺序，主要有如下几个坐标空间，或者说坐标类型：Object space，模型坐标空间；World space，世界坐标系空间；Eye space，观察坐标空间；Clip and Project space，屏幕坐标空间。图 3 表述了 GPU 的整个处理流程，其中茶色区域所展示的就是顶点坐标空间的变换流程。大家从中只需得到一个大概的流程顺序即可，下面将详细阐述空间变换的每个独立阶段。

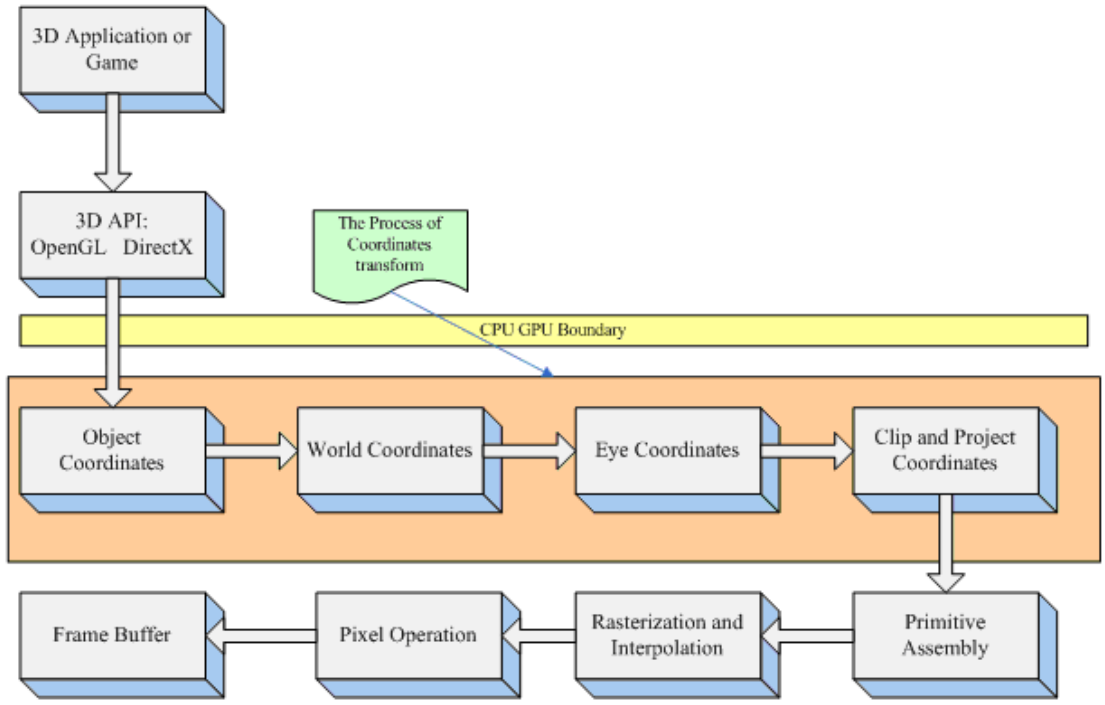


图 3 GPU 处理流程

2.1.1 从 object space 到 world space

When an artist creates a 3D model of an object, the artist selects a convenient orientation and position with which to place the model's continent vertices.

The object space for one object may have no relationship to the object space of another object.【3】

上述语句表示了 object space 的两层核心含义：其一，object space coordinate 就是模型文件中的顶点值，这些值是在模型建模时得到的，例如，用 3DMAX 建立一个球体模型并导出为.max 文件，这个文件中包含的数据就是 object space coordinate；其二，object space coordinate 与其他物体没有任何参照关系，注意，这个概念非常重要，它是将 object space coordinate 和 world space coordinate 区分开来的关键。无论在现实世界，还是在计算机的虚拟空间中，物体都必须和一个固定的坐标原点进行参照才能确定自己所在的位置，这是 world space coordinate 的实际意义所在。

毫无疑问，我们将一个模型导入计算机后，就应该给它一个相对于坐标原点

的位置，那么这个位置就是 world space coordinate，从 object space coordinate 到 world space coordinate 的变换过程由一个四阶矩阵控制，通常称之为 world matrix。

光照计算通常是在 world coordinate space（世界坐标空间）中进行的，这也符合人类的生活常识。当然，也可以在 eye coordinate space 中得到相同的光照效果，因为，在同一观察空间中物体之间的相对关系是保存不变的。

需要高度注意的是：顶点法向量在模型文件中属于 object space，在 GPU 的顶点程序中必须将法向量转换到 world space 中才能使用，如同必须将顶点坐标从 object space 转换到 world space 中一样，但两者的转换矩阵是不同的，准确的说，法向量从 object space 到 world space 的转换矩阵是 world matrix 的转置矩阵的逆矩阵（许多人在顶点程序中会将两者的转换矩阵当作同一个，结果会出现难以查找的错误）。（参阅潘李亮的 3D 变换中法向量变换矩阵的推导一文）

可以阅读电子工业出版社的《计算机图形学（第二版）》第 11 章，进一步了解三维顶点变换具体的计算方法，如果对矩阵运算感到陌生，则有必要复习一下线性代数。

2.1.2 从 world space 到 eye space

每个人都是从各自的视点出发观察这个世界，无论是主观世界还是客观世界。同样，在计算机中每次只能从唯一的视角出发渲染物体。在游戏中，都会提供视点漫游的功能，屏幕显示的内容随着视点的变化而变化。这是因为 GPU 将物体顶点坐标从 world space 转换到了 eye space。

所谓 eye space，即以 camera（视点或相机）为原点，由视线方向、视角和远近平面，共同组成一个梯形体的三维空间，称之为 viewing frustum（视锥），如图 4 所示。近平面，是梯形体较小的矩形面，作为投影平面，远平面是梯形体较大的矩形，在这个梯形体中的所有顶点数据是可见的，而超出这个梯形体之外的场景数据，会被视点去除（Frustum Culling，也称之为视锥裁剪）。

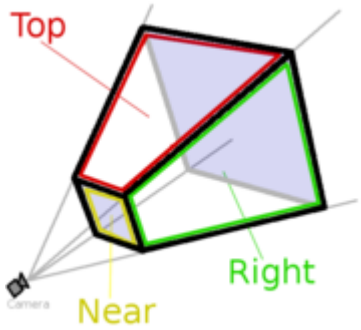


图 4 视锥体

2.1.3 从 eye space 到 project and clip space

Once positions are in eye space, the next step is to determine what positions are actually viewable in the image you eventually intend trend.【3】

即：一旦顶点坐标被转换到 eye space 中，就需要判断哪些点是视点可见的。位于 viewing frustum 梯形体以内的顶点，被认定为可见，而超出这个梯形体之外的场景数据，会被视点去除（Frustum Culling，也称之为视锥裁剪）。这一步通常称之为“clip（裁剪）”，识别指定区域内或区域外的图形部分的过程称之为裁剪算法。

很多人在理解该步骤时存在一个混乱，即“不清楚裁减与投影的关系和两者发生的先后顺序”，不少人觉得是“先裁减再投影”，不过事实并非如此。因为在不规则的体（viewing frustum）中进行裁剪并非易事，所以经过图形学前辈们的精心分析，裁剪被安排到一个单位立方体中进行，该立方体的对角顶点分别是(-1,-1,-1)和(1,1,1)，通常称这个单位立方体为规范立方体（Canonical view volume, CVV）（实时计算机图形学第 9 页）。CVV 的近平面（梯形体较小的矩形面）的 X、Y 坐标对应屏幕像素坐标（左下角是 0、0），Z 坐标则是代表画面像素深度。

多边形裁剪就是 CVV 中完成的。所以，从视点坐标空间到屏幕坐标空间（screen coordinate space）事实上是由三步组成：

1. 用透视变换矩阵把顶点从视锥体中变换到裁剪空间的 CVV 中；

2. 在 CVV 进行图元裁剪；

3. 屏幕映射：将经过前述过程得到的坐标映射到屏幕坐标系上。

在这里，我们尤其要注意第一个步骤，即把顶点从 viewing frustum 变换到 CVV 中，这个过程才是我们常说或者听说的“投影”。主要的投影方法有两种：正投影（也称平行投影）和透视投影。由于投影投影更加符合人类的视觉习惯，所以在附录中会详细讲解透视投影矩阵的推导过程，有兴趣的朋友可以查阅潘宏（网名 Twinsen）的“透视投影变换推导”一文。更详细全面的投影算法可以近一步阅读《计算机图形学（第二版）》第 12 章第 3 节。

确定只有当图元完全或部分的存在于视锥内部时，才需要将其光栅化。当一个图元完全位于视体（此时视体以及变换为 CVV）内部时，它可以直接进入下一个阶段；完全在视体外部的图元，将被剔除；对于部分位于视体内的图元进行裁减处理。详细的裁剪算法可以近一步阅读《计算机图形学（第二版）》第 12 章第 5 节。

附 1：透视投影矩阵的推导过程，建议阅读潘宏（网名 Twinsen）的“透视投影变换推导”一文。

附 2：视点去除，不但可以在 GPU 中进行，也可以使用高级语言（C\C++）在 CPU 上实现。使用高级语言实现时，如果一个场景实体完全不在视锥中，则该实体的网格数据不必传入 GPU，如果一个场景实体部分或完全在视锥中，则该实体网格数据传入 GPU 中。所以如果在高级语言中已经进行了视点去除，那么可以极大的减去 GPU 的负担。使用 C++进行视锥裁剪的算法可以参阅 OGRE（Object-Oriented Graphics Rendering Engine，面向对象的图形渲染引擎）的源码。

2.2 Primitive Assembly && Triangle setup

Primitive Assembly，图元装配，即将顶点根据 primitive（原始的连接关系），

还原出网格结构。网格由顶点和索引组成，在之前的流水线中是对顶点的处理，在这个阶段是根据索引将顶点链接在一起，组成线、面单元。之后就是对超出屏幕外的三角形进行裁剪，想象一下：一个三角形其中一个顶点在画面外，另外两个顶点在画面内，这是我们在屏幕上看到的就是一个四边形。然后将该四边形切成两个小的三角形。

此外还有一个操作涉及到三角形的顶点顺序（其实也就是三角形的法向量朝向），根据右手定则来决定三角面片的法向量，如果该法向量朝向视点（法向量与到视点的方向的点积为正），则该面是正面。一般是顶点按照逆时针排列。如果该面是反面，则进行背面去除操作（Back-face Culling）。在 OpenGL 中有专门的函数 enable 和 disable 背面去除操作。所有的裁剪剔除计算都是为了减少需要绘制的顶点个数。

附：在 2.2 和 2.3 节都提到了裁减的概念，实际裁减是一个较大的概念，为了减少需要绘制的顶点个数，而识别指定区域内或区域外的图形部分的算法都称之为裁减。裁减算法主要包括：视域剔除（View Frustum Culling）、背面剔除（Back-Face Culling）、遮挡剔除（Occlusing Culling）和视口裁减等。

处理三角形的过程被称为 Triangle Setup。到目前位置，我们得到了一堆在屏幕坐标上的三角面片，这些面片是用于做光栅化的（Rasterizing）。

2.3 光栅化阶段

2.3.1 Rasterization

光栅化：决定哪些像素被集合图元覆盖的过程（Rasterization is the process of determining the set of pixels covered by a geometric primitive）。经过上面诸多坐标转换之后，现在我们得到了每个点的屏幕坐标值（Screen coordinate），也知道我们需要绘制的图元（点、线、面）。但此时还存在两个问题，

问题一：点的屏幕坐标值是浮点数，但像素都是由整数点来表示的，如果确定屏幕坐标值所对应的像素？

问题二：在屏幕上需要绘制的有点、线、面，如何根据两个已经确定位置的2个像素点绘制一条线段，如果根据已经确定了位置的3个像素点绘制一个三角形面片？

首先回答一下问题一，“绘制的位置只能接近两指定端点间的实际线段位置，例如，一条线段的位置是（10.48，20.51），转换为像素位置则是（10，21）”（计算机图形学（第二版）52页）。

对于问题二涉及到具体的画线算法，以及区域图元填充算法。通常的画线算法有DDA算法、Bresenham画线算法；区域图元填充算法有，扫描线多边形填充算法、边界填充算法等，具体请参阅《计算机图形学（第二版）》第3章。

这个过程结束之后，顶点(vertex)以及绘制图元（线、面）已经对应到像素(pixel)。下面阐述的是“如何处理像素，即：给像素赋予颜色值”。

2.3.2 Pixel Operation

Pixel operation又称为Raster Operation（在文献【2】中是使用Raster Operation），是在更新帧缓存之前，执行最后一系列针对每个片段的操作，其目的是：计算出每个像素的颜色值。在这个阶段，被遮挡面通过一个被称为深度测试的过程而消除，这其中包含了很多种计算颜色的方法以及技术。Pixel operation包含哪些事情呢？

1：消除遮挡面

2：Texture operation，纹理操作，也就是根据像素的纹理坐标，查询对应的纹理值；

3：Blending

混色，根据目前已经画好的颜色，与正在计算的颜色的透明度（Alpha），混合为两种颜色，作为新的颜色输出。通常称之为 alpha 混合技术。 当在屏幕上绘制某个物体时，与每个像素都相关联的哟一个 RGB 颜色值和一个 Z 缓冲器深度值，另外一个称为是 alpha 值，可以根据需要生成并存储，用来描述给定像素处的物体透明度。如果 alpha 值为 1.0，则表示物体不透明；如果值为 0，表示该物体是透明的，

从绘制管线得到一个 RGBA，使用 over 操作符将该值与原像素颜色值进行混合，公式如下：

$c_d = a \bullet c_a + (1-a)c_s$ 【over 操作符】

a 是透明度值（alpha），c_a 表示透明物体的颜色，c_s 表示混合前像素的颜色值，c_d 是最终计算得到的颜色值。Over 操作可以用于照片混合和物体合成绘制方面，这个过程称为合成（compositing）。可以联想一下，OGRE 中有一种技术称为 compositor（合成器）。

此外还需要提醒的一点是：为了在场景中绘制透明物体，通常需要对物体进行排序。首先，绘制不透明的物体；然后，在不透明物体的上方，对透明物体按照由后到前的顺序进行混合处理。如果按照任意顺序进行混合，那么会产生严重的失真。既然需要排序，那么就需要用到 z buffer。关于透明度、合成的相关知识点，可以在《实时计算机图形学（第二版）》第四章 4.5 节（59 页）中得到更多详尽的知识。

4：Filtering，将正在算的颜色经过某种 Filtering（滤波或者滤镜）后输出。可以理解为：经过一种数学运算后变成新的颜色值。

该阶段之后，像素的颜色值被写入帧缓存中。图 5 来自文献【2】1.2.3，说明了像素操作的流程：

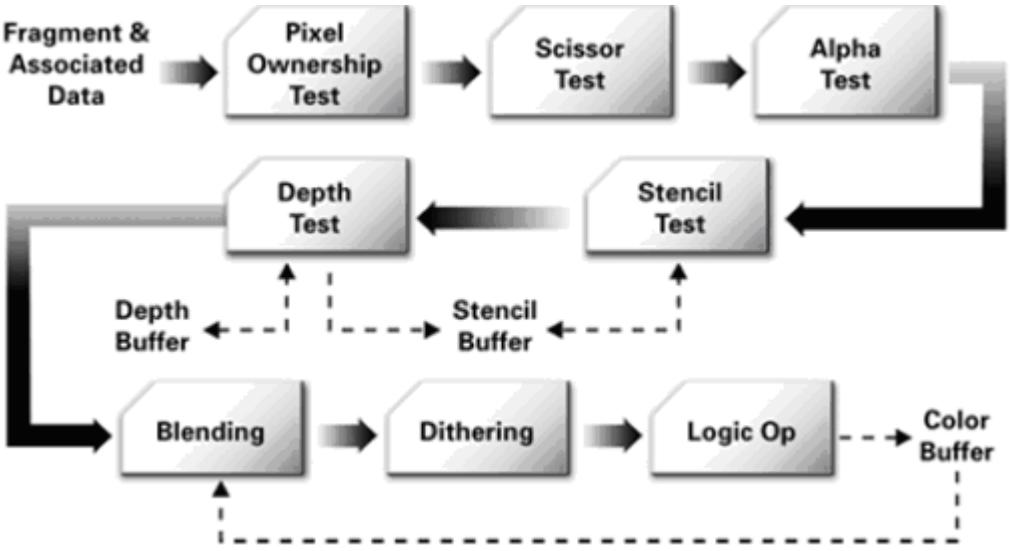


图 5 OpenGL 和 Direct3D 中的 Raster Operations

2.4 图形硬件

这一节中主要阐述图形硬件的相关知识，主要包括 GPU 中数据的存放硬件，以及各类缓冲区的具体含义和用途，如：z buffer（深度缓冲区）、stencil buffer（模板缓冲区）、frame buffer（帧缓冲区）和 color buffer（颜色缓冲区）。

2.4.1 GPU 内存架构

寄存器和内存有什么区别？

从物理结构而言，寄存器是 cpu 或 gpu 内部的存储单元，即寄存器是嵌入在 cpu 或者 gpu 中的，而内存则可以独立存在；从功能上而言，寄存器是有限存储容量的高速存储部件，用来暂存指令、数据和位址。Shader 编成是基于计算机图形硬件的，这其中就包括 GPU 上的寄存器类型，glsl 和 hlsl 的着色虚拟机版本就是基于 GPU 的寄存器和指令集而区分的。

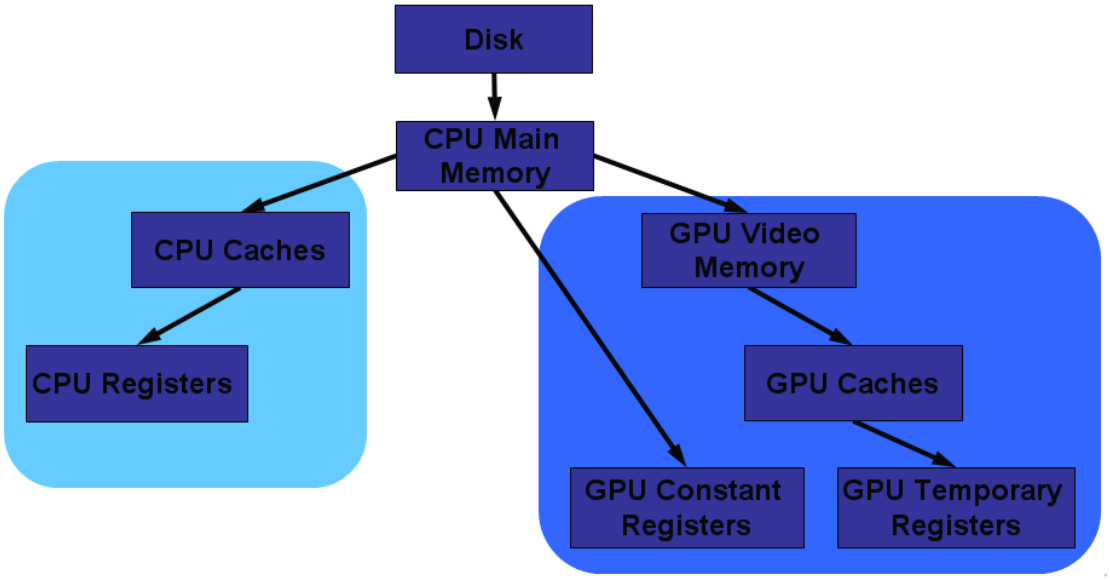


图 6 GPU 存储架构

2.4.2 Z Buffer 与 Z 值

Z buffer 应该是大家最为熟悉的缓冲区类型，又称为 depth buffer，即深度缓冲区，其中存放的是视点到每个像素所对应的空间点的距离衡量，称之为 Z 值或者深度值。可见物体的 Z 值范围位于【0，1】区间，默认情况下，最接近眼睛的顶点（近裁减面上）其 Z 值为 0.0，离眼睛最远的顶点（远裁减面上）其 Z 值为 1.0。使用 z buffer 可以用来判断空间点的遮挡关系，著名的深度缓冲区算法（depth-buffer method，又称 Z 缓冲区算法）就是对投影平面上每个像素所对应的 Z 值进行比较的。

Z 值并非真正的笛卡儿空间坐标系中的欧几里德距离（Euclidean distance），而是一种“顶点到视点距离”的相对度量。所谓相对度量，即这个值保留了与其他同类型值的相对大小关系。在 steve Baker 撰写的文章“Learning to love your Z-buffer”中将 GPU 对 Z 值的计算公式描述为：

$$z\_buffer\_value = (1 << N) * \frac{a * z + b}{z}$$

$$a = \frac{f}{f - n}$$

$$b = \frac{f * n}{n - f}$$

其中 f 表示视点到远裁减面的空间距离，n 表示视点到近裁减面的空间距离，z 表示视点到顶点的空间距离，N 表示 Z 值精度。

大多数人所忽略的是，z buffer 中存放的 z 值不一定是线性变化的。在正投影中同一图元相邻像素的 Z 值是线性关系的，但在透视投影中却不是的。在透视投影中这种关系是非线性的，而且非线性的程度随着空间点到视点的距离增加而越发明显。

当 3D 图形处理器将基础图元（点、线、面）渲染到屏幕上时，需要以逐行扫描的方式进行光栅化。图元顶点位置信息是在应用程序中指定的（顶点模型坐标），然后通过一系列的过程变换到屏幕空间，但是图元内部点的屏幕坐标必须由已知的顶点信息插值而来。例如，当画三角形的一条扫描线时，扫描线上的每个像素的信息，是对扫描线左右端点处已知信息值进行插值运算得到的，所以内部点的 Z 值也是插值计算得到的。同一图元相邻像素点是线性关系（像素点是均匀分布的，所以一定是线性关系），但对应到空间线段上则存在非线性的情况，如图 7 所示。所示：线段 AE 是某三角面片的两个顶点，投影到屏幕空间对应到像素 1 和像素 5；光栅化时，需要对像素 2、3、4 进行属性插值，从视点引射线到空间线段上的交点分别为 B、C、D。从图中可以看出，点 B、C、D 并不是均匀分布在空间线段上的，而且如果离视点越远，这种差异就越发突出。即，投影面上相等的步长，在空间中对应的步长会随着离视点距离的增加而变长。所以如果对内部像素点的 Z 值进行线性插值，得到的 Z 值并不能反应真实的空间点的深度关系。Z 值的不准确，会导致物体显示顺序的错乱，例如，在游戏中常会看到远处的一些面片相互交叠。

为了避免或减轻上述的情况，在设置视点相机远裁减面和近裁减面时，两者的比值应尽量小于 1000。要想解决这个问题，最简单的方法是通过将近截面远离眼睛来降低比值，不过这种方法的副作用时可能会将眼前的物体裁减掉。

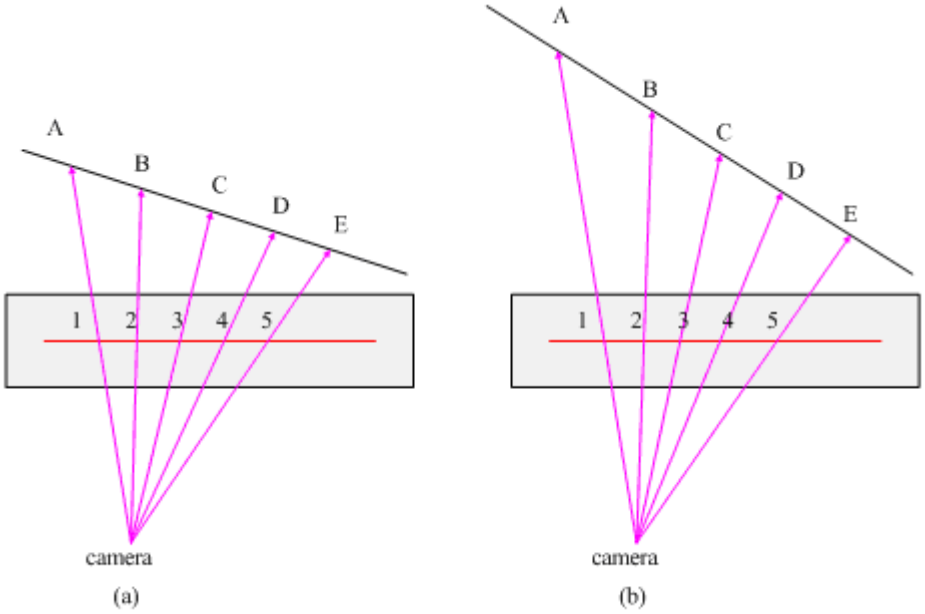


图 7 Z 值的非线性关系

很多图形硬件使用 16 位的 Z buffer，另外的一些使用 24 位的 Z buffer，还有一些很好的图形硬件使用 32 位的。如果你拥有 32 位的 Z buffer，则 Z 精度（Z-precision）对你不是一个问题。但是如果你希望你的程序可以灵活的使用各种层次的图形硬件，那么你就需要多思考一下。

Z 精度之所以重要，是因为 Z 值决定了物体之间的相互遮挡关系，如果没有足够的精度，则两个相距很近的物体将会出现随机遮挡的现象，这种现象通常称为“flimmering”或”Z-fighting”。

2.4.3 Stencil Buffer

A stencil buffer is an extra buffer, in addition to the color buffer and depth buffer found on modern computer graphics hardware. The buffer is per pixel, and works on integer values, usually with a depth of one byte per pixel. The depth buffer and stencil buffer often share the same area in the RAM of the graphics hardware.

Stencil buffer,中文翻译为“模板缓冲区”，它是一个额外的 buffer，通常附加到 z buffer 中，例如：15 位的 z buffer 加上 1 位的 stencil buffer(总共 2 个字节)；或者 24 位的 z buffer 加上 8 位的 stencil buffer（总共 4 个字节）。每个像素对应

一个 stencil buffer(其实就是对应一个 Z buffer)。 Z buffer 和 stencil buffer 通常在显存中共享同一片区域。Stencil buffer 对大部分人而言应该比较陌生，这是一个用来“做记号”的 buffer，例如：在一个像素的 stencil buffer 中存放 1，表示该像素对应的空间点处于阴影体（shadow volume）中。

2.4.4 Frame Buffer

A framebuffer is a video output device that drives a video display from a memory buffer containing a complete frame of data. The information in the buffer typically consists of color values for every pixel on the screen. Color values are commonly stored in 1-bit monochrome, 4-bit palettized, 8-bit palettized, 16-bit highcolor and 24-bit truecolor formats. An additional alpha channel is sometimes used to retain information about pixel transparency. The total amount of the memory required to drive the framebuffer depends on the resolution of the output signal, and on the color depth and palette size.

Frame buffer，称为帧缓冲器，用于存放显示输出的数据，这个 buffer 中的数据一般是像素颜色值。Frame buffer 有时也被认为是 color buffer（颜色缓冲器）和 z buffer 的组合（《实时计算机图形学（第二版）》12 页）。那么 frame buffer 位于什么地方呢？在 webMediaBrands 网站上摘录了一段英文说明，即 frame buffer 通常都在显卡上，但是有时显卡会集成到主板上，所以这种情况下 frame buffer 被放在内存区域（general main memory）。

Frame buffer: The portion of memory reserved for holding the complete bit-mapped image that is sent to the monitor. Typically the frame buffer is stored in the memory chips on the video adapter. In some instances, however, the video chipset is integrated into the motherboard design, and the frame buffer is stored in general main memory.

2.5 本章小节

本章介绍了 GPU 图形绘制管线，并对相关的图形硬件进行了阐述。图形绘制管线是 GPU 编程的基础，事实上顶点着色程序和片段着色程序正是按照图形绘制管线而划分的。

国内的 openGPU 网站上有一些资料可以进一步阅读，北京大学出版社出版的《实时计算机图形学（第二版）》的第 2 章和第 3 章，以及 Blinn 的《A Trip Down the Graphics Pineline》在学习绘制管线方面也是非常好的资料。

第 3 章 Shader Language

In the last year I have never had to write a single HLSL/GLSL shader. Bottom line, I can't think of any reason NOT to use CG.

---

shader language，称为着色语言，shade 在英语是阴影、颜色深浅的意思，Wikipedia 上对 shader language 的解释为"The job of a surface shading procedure is to choose a color for each pixel on a surface, incorporating any variations in color of the surface itself and the effects of lights that shine on the surface(Marc Olano)"，即，shader language 基于物体本身属性和光照条件，计算每个像素的颜色值。

实际上这种解释具有明显的时代局限性，在 GPU 编程发展的早期，shader language 的提出目标是加强对图形处理算法的控制，所以对该语言的定义亦针对于此。但随着技术的进步，目前的 shader language 早已经用于通用计算研究。

shader language 被定位为高级语言，如，GLSL 的全称是"High Level Shading Language"，Cg 语言的全称为"C for Graphic"，并且这两种 shader language 的语法设计非常类似于 C 语言。不过高级语言的一个重要特性是"独立于硬件"，在这一方面 shader language 暂时还做不到，shader language 完全依赖于 GPU 构架，这一特征在现阶段是非常明显的！任意一种 shader language 都必须基于图形硬件，所以 GPU 编程技术的发展本质上还是图形硬件的发展。在 shader language 存在之前，展示基于图形硬件的编程能力只能靠低级的汇编语言。

目前，shader language 的发展方向是设计出在便捷性方面可以和 C++\JAVA 相比的高级语言，"赋予程序员灵活而方便的编程方式"，并"尽可能的控制渲染过程"同时"利用图形硬件的并行性，提高算法的效率"。Shader language 目前主要有 3 种语言：基于 OpenGL 的 GLSL，基于 Direct3D 的 HLSL，还有 NVIDIA 公司的 Cg 语言。

本章的目的是阐述 shader language 的基本原理和运行流程，首先从硬件的角度对 Programmable Vertex Processor（可编程顶点处理器，又称为顶点着色器）

和 Programmable Fragment Processor（可编程片断处理器，又称为片断着色器）的作用进行阐述，然后在此基础上对 vertex program 和 fragment program 进行具体论述，最后对 GLSL、HLSL 和 Cg 进行比较。

3.1 Shader Language 原理

使用 shader language 编写的程序称之为 shader program（着色程序）。着色程序分为两类：vertex shader program（顶点着色程序）和 fragment shader program（片断着色程序）。为了清楚的解释顶点着色和片断着色的含义，我们首先从阐述 GPU 上的两个组件：Programmable Vertex Processor（可编程顶点处理器，又称为顶点着色器）和 Programmable Fragment Processor（可编程片断处理器，又称为片断着色器）。文献[2]第 1.2.4 节中论述到：

The vertex and Fragment processing broken out into programmable units. The Programmable vertex processor is the hardware unit that runs your Cg Vertex programs, whereas the programmable fragment processor is the unit that runs your Cg fragment programs.

这段话的含义是：顶点和片段处理器被分离成可编程单元，可编程顶点处理器是一个硬件单元，可以运行顶点程序，而可编程片段处理器则是一个可以运行片段程序的单元。

顶点和片段处理器都拥有非常强大的并行计算能力，并且非常擅长于矩阵（不高于 4 阶）计算，片段处理器还可以高速查询纹理信息（目前顶点处理器还不行，这是顶点处理器的一个发展方向）。

如上所述，顶点程序运行在顶点处理器上，片段程序运行在片段处理器上，哪么它们究竟控制了 GPU 渲染的哪个过程。图 8 展示了可编程图形渲染管线。

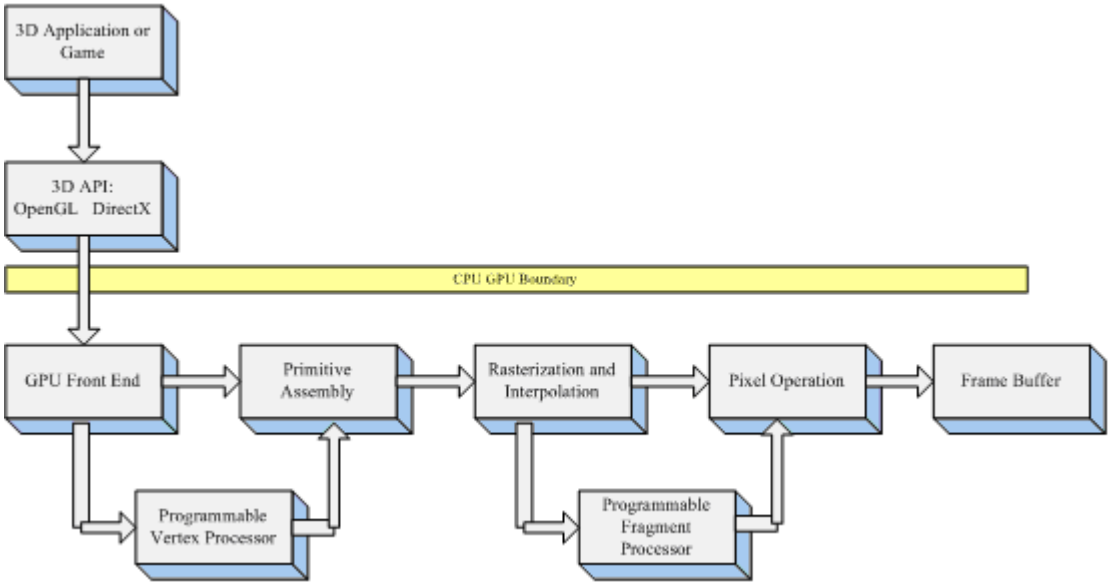


图 8 可编程图形渲染管线

对比上一章图 3。中的 GPU 渲染管线，可以看出，顶点着色器控制顶点坐标转换过程；片段着色器控制像素颜色计算过程。这样就区分出顶点着色程序和片段着色程序的各自分工：Vertex program 负责顶点坐标变换；Fragment program 负责像素颜色计算；前者的输出是后者的输入。

图 9 展示了现阶段可编程图形硬件的输入\输出。输入寄存器存放输入的图元信息；输出寄存器存放处理后的图元信息；纹理 buffer 存放纹理数据，目前大多数的可编程图形硬件只支持片段处理器处理纹理；从外部宿主程序输入的常量放在常量寄存器中；临时寄存器存放着色程序在执行过程中产生的临时数据。

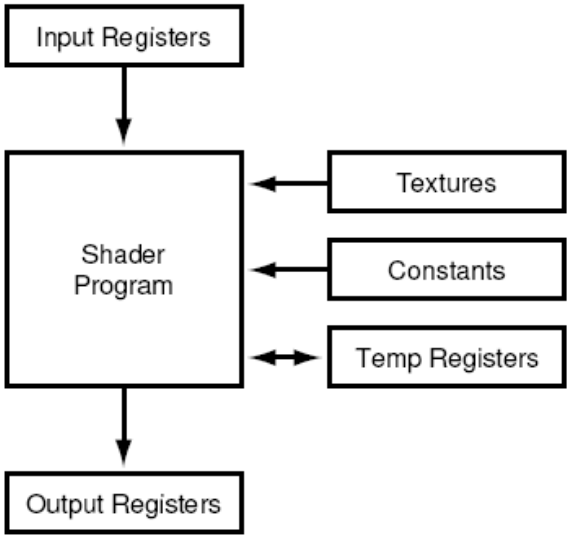


图 9 可编程图形硬件输入\输出

3.2 Vertex Shader Program

Vertex shader program（顶点着色程序）和 Fragment shader program（片断着色程序）分别被 Programmable Vertex Processor（可编程顶点处理器）和 Programmable Fragment Processo（可编程片断处理器）所执行。

顶点着色程序从 GPU 前端模块（寄存器）中提取图元信息（顶点位置、法向量、纹理坐标等），并完成顶点坐标空间转换、法向量空间转换、光照计算等操作，最后将计算好的数据传送到指定寄存器中；然后片断着色程序从中获取需要的数据，通常为“纹理坐标、光照信息等”，并根据这些信息以及从应用程序传递的纹理信息（如果有的话）进行每个片断的颜色计算，最后将处理后的数据送光栅操作模块。

图 10 展示了在顶点着色器和像素着色器的数据处理流程。在应用程序中设定的图元信息（顶点位置坐标、颜色、纹理坐标等）传递到 vertex buffer 中；纹理信息传递到 texture buffer 中。其中虚线表示目前还没有实现的数据传递。当前的顶点程序还不能处理纹理信息，纹理信息只能在片断程序中读入。

顶点着色程序与片断着色程序通常是同时存在，相互配合，前者的输出作为后者的输入。不过，也可以只有顶点着色程序。如果只有顶点着色程序，那么只

对输入的顶点进行操作，而顶点内部的点则按照硬件默认的方式自动插值。例如，输入一个三角面片，顶点着色程序对其进行 phong 光照计算，只计算三个顶点的光照颜色，而三角面片内部点的颜色按照硬件默认的算法（Gourand 明暗处理或者快速 phong 明暗处理）进行插值，如果图形硬件比较先进，默认的处理算法较好（快速 phong 明暗处理），则效果也会较好；如果图形硬件使用 Gourand 明暗处理算法，则会出现马赫带效应（条带化）。

而片断着色程序是对每个片断进行独立的颜色计算，并且算法由自己编写，不但可控性好，而且可以达到更好的效果。

由于 GPU 对数据进行并行处理，所以每个数据都会执行一次 shader 程序程序。即，每个顶点数据都会执行一次顶点程序；每个片段都会执行一次片段程序。

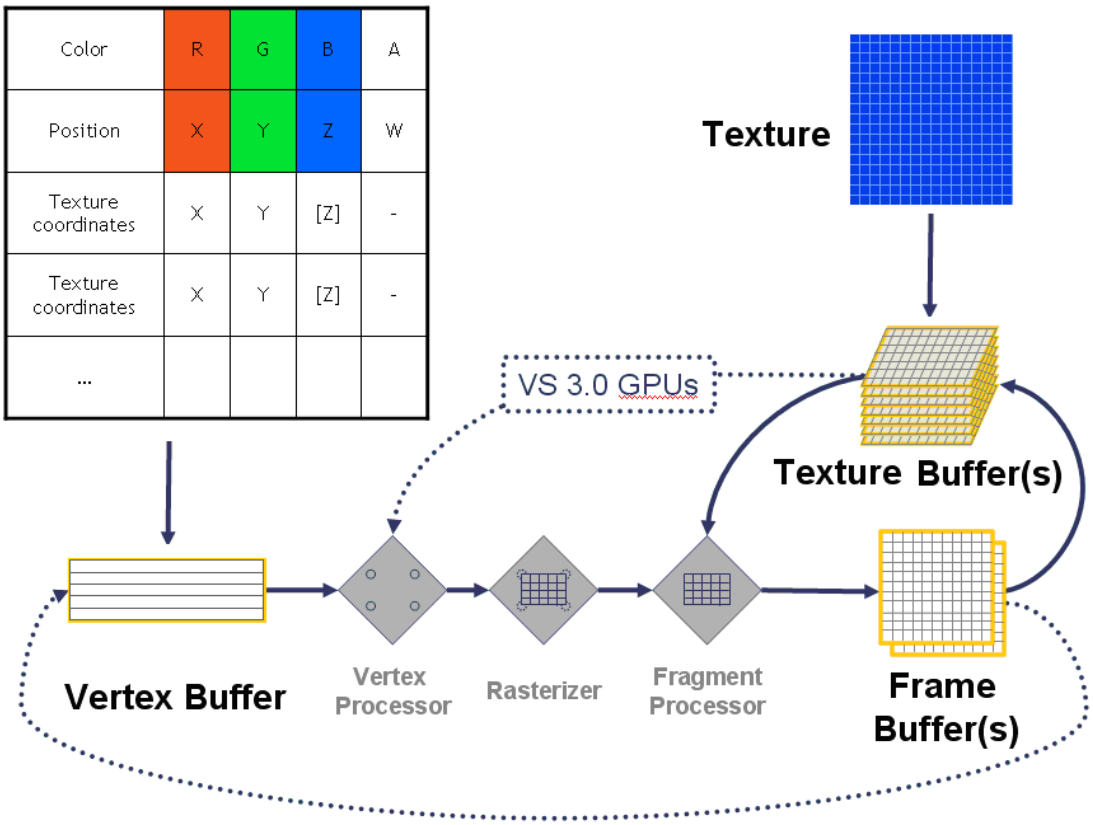


图 10 顶点着色器和像素着色器的数据处理流程

3.3 Fragment Shader Program

片断着色程序对每个片断进行独立的颜色计算，最后输出颜色值的就是该片

段最终显示的颜色。可以这样说，顶点着色程序主要进行几何方面的运算，而片段着色程序主要针对最终的颜色值进行计算。

片段着色程序还有一个突出的特点是：拥有检索纹理的能力。对于 GPU 而言，纹理等价于数组，这意味着，如果要做通用计算，例如数组排序、字符串检索等，就必须使用到片段着色程序。让顶点着色器也拥有检索纹理的能力，是目前的一个研究方向。

附：什么是片断？片断和像素有什么不一样？所谓片断就是所有的三维顶点在光栅化之后的数据集合，这些数据还没有经过深度值比较，而屏幕显示的像素都是经过深度比较的。

3.4 CG VS GLSL VS HLSL

Shader language 目前有 3 种主流语言：基于 OpenGL 的 GLSL（OpenGL Shading Language，也称为 GLslang），基于 Direct3D 的 HLSL（High Level Shading Language），还有 NVIDIA 公司的 Cg （C for Graphic）语言。

GLSL 与 HLSL 分别提基于 OpenGL 和 Direct3D 的接口，两者不能混用，事实上 OpenGL 和 Direct3D 一直都是冤家对头，曹操和刘备还有一段和平共处的甜美时光，但 OpenGL 和 Direct3D 各自的东家则从来都是争斗不休。 争斗良久，既然没有分出胜负，那么必然是两败俱伤的局面。

首先 ATI 系列显卡对 OpenGL 扩展支持不够，例如我在使用 OSG（Open Scene Graphic）开源图形引擎时，由于该引擎完全基于 OpenGL，导致其上编写的 3D 仿真程序在较老的显卡上常常出现纹理无法显示的问题。其次 GLSL 的语法体系自成一家，而 HLSL 和 Cg 语言的语法基本相同，这就意味着，只要学习 HLSL 和 Cg 中的任何一种，就等同于学习了两种语言。不过 OpenGL 毕竟图形 API 的曾经领袖，通常介绍 OpenGL 都会附加上一句“事实上的工业标准”，所以在其长期发展中积累下的用户群庞大，这些用户当然会选择 GLSL 学习。此外，GLSL 继承了 OpenGL 的良好移植性，一度在 unix 等操作系统上独领风骚（已是曾经

的往事）。

微软的 HLSL 移植性较差，在 windows 平台上可谓一家独大，可一出自己的院子（还好院子够大），就是落地凤凰不如鸡。这一点在很大程度上限制了 HLSL 的推广和发展。目前 HLSL 多半都是用于游戏领域。我可以负责任的断言，在 Shader language 领域，HLSL 可以凭借微软的老本成为割据一方的诸侯，但，决不可能成为君临天下的霸主。这和微软现在的局面很像，就是一个被带刺鲜花簇拥着的大财主，富贵已极，寸步难行。

上面两个大佬打的很热烈，在这种情况下可以用一句俗话来形容，“鹬蚌相争，渔翁得利”。NVIDIA 是现在当之无愧的显卡之王（尤其在 AMD 兼并 ATI 之后），是 GPU 编程理论的奠基者，GeForce 系列显卡早已深入人心，它推出的 Cg 语言已经取得了巨大的成功，生生形成了三足鼎立之势。NVIDIA 公司深通广告之道，目前最流行的 GPU 编程精粹一书就出自该公司，书中不但介绍了大量的 GPU 前沿知识，最重要的是大部分都用 Cg 语言实现。凭借该系列的书籍，NVIDIA 不光确定了在青年学子间的学术地位，而且成功的推广了 Cg 语言。我本人就是使用 Cg 语言进行研发，基于如下理由：

其一，Cg 是一个可以被 OpenGL 和 Direct3D 广泛支持的图形处理器编程语言。Cg 语言和 OpenGL、DirectX 并不是同一层次的语言，而是 OpenGL 和 DirectX 的上层，即，Cg 程序是运行在 OpenGL 和 DirectX 标准顶点和像素着色的基础上的；

其二，Cg 语言是 Microsoft 和 NVIDIA 相互协作在标准硬件光照语言的语法和语义上达成了一致，文献[1]在 1.3.1 节的标题就是“Microsoft and NVIDIA’s Collaboration to Develop Cg and HLSL”，所以，HLSL 和 Cg 其实是同一种语言（参见 Cg 教程_可编程实时图形权威指南 29 页的致谢部分）。很多时候，你会发现用 HLSL 写的代码可以直接当中 Cg 代码使用。也就是说，cg 基于知识联盟（Microsoft 和 NVIDIA），且拥有跨平台性，选择 cg 语言是大势所趋。有心的读者，可以注意市面上当前的 GPU 编程方面的书籍，大都是基于 CG 语言的。（附：Microsoft 和 NVIDIA 联手推出 Cg，应该是一种经济和技术上的双赢，通过这种

方式联手打击 GLSL）

此外，Cg，即 C for graphics，用于图形的 C 语言，这其实说明了当时设计人员的一个初衷，就是“让基于图形硬件的编程变得和 C 语言编程一样方便，自由”。正如 C++和 Java 的语法是基于 C 的，cg 语言本身也是基于 C 语言的。如果您使用过 C、C++、Java 其中任意一个，那么 Cg 的语法也是比较容易掌握的。Cg 语言极力保留了 C 语言的大部分语义，力图让开发人员从硬件细节中解脱出来，Cg 同时拥有高级语言的好处，如代码的易重用性，可读性提高等。使用 cg 还可以实现动画驱动、通用计算（排序、查找）等功能。

在曾经的一段时间中有一种流言：NVIDIA 将要抛弃 Cg 语言。并且在网上关于 Cg、GLSL、HLSL 的优劣讨论中，Cg 的跨平台性也受到过广泛的质疑。我在 2007 年 12 月参加朱幼虹老师 OSG 培训班时，他曾专门对 Cg、GLSL、HLSL 进行了比较，说道：尽管目前还有一些关于 Cg 和 GLSL 之间的争议，不过主流的 3D 图形厂家都开始支持 Cg 语言。市场经济的选择可以说明一切，时间可以明辨真伪，到 2009 年末，Cg 语言不但没有被抛弃，而且越来越受欢迎。

我在 OGRE 官方论坛上，搜索过有关使用 Cg 和 GLSL 的讨论帖子，套用其中一个帖子的结尾语来结束本章：

In the last year I have never had to write a single HLSL/GLSL shader. Bottom line, I can’t think of any reason NOT to use CG.

3.5 本章小结

本章首先阐述了了着色程序的工作原理，着色程序分为顶点着色程序和片段着色程序；然后对三种主流的着色语言，Cg、GLSL、HLSL，进行了对比论证。虽然我本人比较推崇 Cg 语言，但并不排斥 GLSL，事实上学习任意一种语言总会有用武之地，无非是“地”的大小有别而已。套用武侠小说中的一句话，语言无高低，用法有高下。着色程序中的算法才是精髓所在！

第 4 章 Cg 语言概述

Cg（C for Graphcis）语言，是 NVIDIA 与 Microsoft 合作研发，旨在为开发人员提供一套方便、跨平台（良好的兼容性），控制可编程图形硬件的高级语言。Cg 语言的语法结构与 C 语言非常类似，使用 Cg 编写的着色程序默认的文件后缀是*. Cg。

4.1 开始 Cg 之旅

在 NVIDIA 的 http://developer.nvidia.com/object/cg_toolkit.html 网页上下载 Cg Toolkit ，截止到 2009 年 10 月，Cg 语言的版本为 2.2。下载之后直接安装即可。在安装目录的 bin 目录下一个可执行程序：cgc.exe。这是 NVIDIA 提供的 Cg 程序编译器。

Cg 语言规范是公开和开放的，并且 NVIDIA 开放了 Cg 编译器技术的源代码，使用无限制的、免费的许可证。

目前还没有一个主流的专门为编写着色程序而开发的 IDE，很多人都是直接在文本中写好程序后，然后将文件后缀改为.cg。在网上有一个名为 NShader（http://nshader.codeplex.com/）的 Visual Studio2008 插件，安装之后可以支持编写着色程序。图 11 展示了使用该插件之后的使用效果。

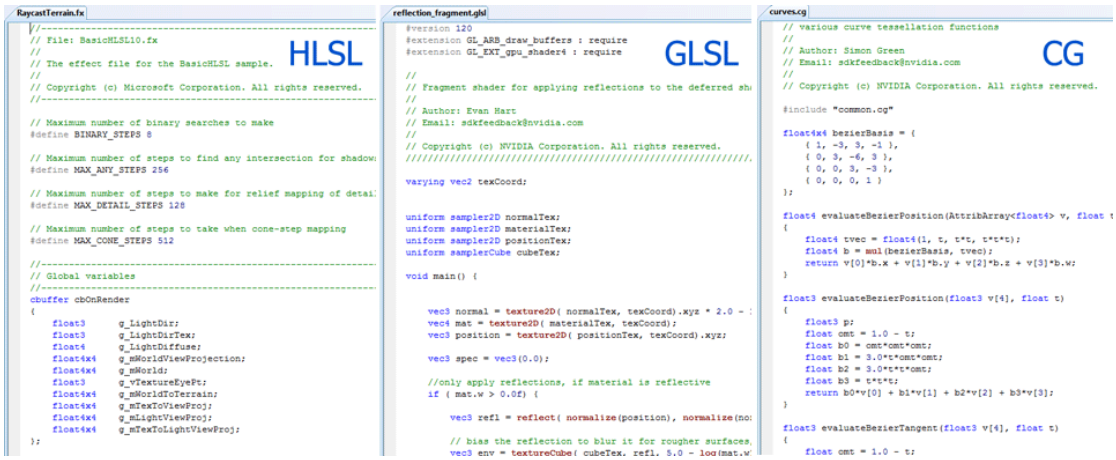


图 11 NShader 插件的使用效果

4.2 CG 特性

Cg 同时被 OpenGL 与 Direct3D 两种编程 API 所支持。这一点不但对开发人员而言非常方便，而且也赋予了 Cg 程序良好的跨平台性。一个正确编写的 Cg 应用程序可以不做任何修改的同时工作在 OpenGL 和 Direct3D 之上。

4.3 CG 编译

4.3.1 CG 编译原理

计算机只能理解和执行由 0、1 序列（电压序列）构成的机器语言，所以汇编语言和高级语言程序都需要进行翻译才能被计算机所理解，担负这一任务的程序称为语言处理程序，通常也被称为编译程序。例如 C 或者 C++ 编写的程序，需要首先编译成可执行文件（.exe 文件），然后才能在 GPU 上运行，且一旦编译后，除非改变程序代码，否则不需要重新编译，这种方式称为静态编译（static coompilation）。静态编译最重要的特征是：一旦编译为可执行文件，在可执行文件运行期间不再需要源码信息。而动态编译（dynamic compilation）与之相反，编译程序和源码都要参与到程序的运行过程中。

Cg 语言通常采用动态编译的方式，即，在宿主程序运行时利用 Cg 运行库(Cg

Runtime library)动态编译 Cg 代码，使用动态编译的方式，可以将 Cg 程序当作一个脚本，随时修改随时运行，节省大量的时间，在 OGRE 图形引擎中就是采用这样的方法。在文献[2]的 1.4.2 章节中提到 Cg 语言同样支持静态编译方式，即，Cg 源码编译成汇编代码后，这部分目标代码被链接到宿主程序最后的可执行程序中。使用静态编译的好处是只要发布可执行文件即可，源码不会被公开。

Cg 编译器首先将 Cg 程序翻译成可被图形 API（OpenGL 和 Direct3D）所接受的形式，然后应用程序使用适当的 OpenGL 和 Direct3D 命令将翻译后的 Cg 程序传递给图形处理器，OpenGL 和 Direct3D 驱动程序最后把它翻译成图形处理器所需要的硬件可执行格式。NVIDIA 提供的 Cg 编译器为 cgc.exe。

Cg 程序的编译不但依赖于宿主程序所使用的三维编程接口，而且依赖于图形硬件环境，因为图形硬件自身的限制，不一定支持某种 Cg 语句，例如，如果你所使用的 GPU 并不支持循环控制指令，那么在 Cg 程序中编写的循环控制语句将无法通过编译。被特定的图形硬件环境或 AIP 所支持的 Cg 语言子集，被称为 Cg Profiles。需要注意的是： profile 分为顶点程序的 profile 和片段程序的 profile，这是因为顶点着色器和片段着色器原本就不是工作在同一个硬件。

Cg Profiles 是 Cg 语言的重要组成部分，在使用 Cg 语言编写着色程序时，首先要考虑的一点就是“当前的图形硬件环境支持那个 Cg Profile”，这直接关系到您所编写的着色程序是否可以在当前的图形硬件上运行。

4.3.1 CGC 编译命令

图 12 cgc -h 命令

如果 Cg Toolkit 安装正确，在 NVIDIA Corporation\Cg\bin 文件夹下会看到 cgc.exe 文件。首先打开命令行窗口，输入“cgc –h”（引号不用输入），如果安装正确，则会出现图 12 所示的提示信息。

Cg 程序编译的命令形式为：

cgc [options] file

[options]表示可选配置项，file 表示 Cg 程序文件名。可选配置项包括编译时选择使用的 profile、着色程序的入口函数名称，以及着色程序文件名。比

较典型的编译方式是：

cgc –profile glslv –entry main_v test.cg

-profile 是 profile 配置项名;glslv 是当前所使用的 profile 名称;-entry 着色程序的入口函数名称配置项； main_v 是顶点着色程序的入口函数名；test.cg 是当前的着色程序文件名。编译器指定的着色程序入口函数名默认为 main，通常为了将顶点\片段着色程序入口函数名区别开来，而并不使用默认名称。在下面所有的例子中，main_v 表示顶点着色程序入口函数名,main_f 表示片段着色程序入口函数名。

需要强调如下几点：

1. 着色程序分为顶点着色程序和片段着色程序，profile 也分为顶点 profile 和片段 profile，所以编译顶点着色程序时必须选用当前图形硬件支持的顶点 profile，同理，编译片段着色程序时必须选用当前图形硬件支持的片段 profile。下面所示使用片段 profile fp20 编译顶点着色程序是不对的。

   cgc –profile fp20 glslv –entry main_v test.cg

   所以，如果您的着色程序中同时存在顶点着色程序和片段着色程序，在编译前切记分别选择各自的 profile。

2. 选择 profile 如果不被当前图形硬件所支持，编译时会出现错误。被编译的着色程序中，如果存在不被所选择的 profile 所支持的语句，则编译时会出现错误。例如，tex2D(sampler2D tex , float3 sz ,float2 dsdx , float2 dsdy)不被 fp20 所支持，如果你的编译形式为：

   cgc –profile fp20 –entry main_f test.cg

   则会出现错误提示信息：

   error C3004: function “tex2D” not supported in this profile。

改用fp30，进行编译就会通过。

cgc –profile fp30 –entry main_f test.cg

尤其需要注意的是，循环语句for,while只被vs_2_x, vp30, vp40,fp40等少量的profiles所支持。在CgUsersManual中提到“In other profiles, **for** and

while loops may only be used if the compiler can fully unroll them (that is, if the compiler can determine the iteration count at compile time)”，这句话的意思是“在其他的profiles中，for和while循环只有当确切的知道循环次数时才能被使用”。但经过试验，通常在其他profiles编译含义for,while语句时会出现错误提示信息：

error c6003：instruction limit of exceeded……

因此，如果没有确切的把握，不要在低级的profiles中使用循环控制语句。

3. 被编译的着色程序文件名必须加上.cg 后缀。如果没有加后缀，写成如下的形式：

cgc –profile glslv –entry main_v

则会出现错误提示信息：

fatal error C9999: Can’t open file:test

4. 另外 cgc 还提供一种比较特殊的功能：就是将 Cg 语言所写的着色程序转换为使用 GLSL 或 HLSL 所编写的程序。例如，将代码写成如下形式，表示编译文件 test.cg 中的顶点着色程序，入口函数名为 main_v，并将顶点着色程序转换为 glsl 程序，然后保存成文件 direct.glsl。

cgc –profile glslv –o direct.glsl –entry main_v test.cg

5. 还有一个非常隐蔽的编译情况是：如果着色程序中的某些变量并没有为最终的输出做出贡献，则编译时会将该部分代码忽略（会检查语法错误，但并不编译成汇编代码）。通常这一点不会造成太大的影响，但是如果这些变量刚好是从外部宿主程序中传入的变量，并且在着色程序中没有被使用，则宿主

程序传入变量的接口函数可能会报错“找不到该变量”。这种情况比较少遇到，但并非不存在，且一旦遇上问题的原因难以查明，故而我在此写上，希望可以有所帮助。

基于 GPU 编程，最令人崩溃的一点是：无法跟踪调试着色程序！这一点目前还没有解决方案出现。对于一个着色程序，语法错误可以通过编译器发现，而代码逻辑错误只能是人为查找。常会遇到这种情况，一段代码编译通过，但是运行结果不在预期之中，如果是 C++\JAVA 程序就可以进行跟踪调试，但是着色程序不能被调试，只能一行代码一行代码的进行逻辑分析。

所以，编译着色程序要非常注意逻辑的严密性，和代码的组织结构，这是为了更加容易的暴露错误和维护代码。一个良好的习惯是加入注释语句。

4.4 CG Profiles

Profile 在英文中的意思是“侧面、轮廓”， 文献[1]第三页写到：A Cg profile defines a subset of the full Cg language that is supported on a particular hardware platform or API（CgUsersManual 21 页）。即一个 Cg profile 定义了一个“被特定图形硬件或 API 所支持的 Cg 语言子集”，从前面的分析我们可以知道，任意一种 shader language 都是基于可编程图形硬件的（寄存器、指令集等），这也就意味着:不同的图形硬件对应着不同的功能子集。Profile 按照功能可以划分为顶点 Profile 和片断 Profile，而顶点 profile 和片段 profile 又基于 OpenGL 和 DirectX 的不同版本或扩展，划分为各种版本。从某种意义上而言，OpenGL 和 DirectX 的发展历程成就了 Cg 语言。

当前 Cg compiler 所支持的 profiles 有：

- OpenGL ARB vertex programs
  Runtime profile: CG_PROFILE_ARBVP1
  Compiler option: _profile arbvp1
- OpenGL ARB fragment programs
  Runtime profile: CG_PROFILE_ARBFP1
  Compiler option: _profile arbfp1

- OpenGL NV40 vertex programs

  Runtime profile: CG_PROFILE_VP40

  Compiler option: _profile vp40
- OpenGL NV40 fragment programs

  Runtime profile: CG_PROFILE_FP40

  Compiler option: _profile fp40
- OpenGL NV30 vertex programs

  Runtime profile: CG_PROFILE_VP30

  Compiler option: _profile vp30
- OpenGL NV30 fragment programs

  Runtime profile: CG_PROFILE_FP30

  Compiler option: _profile fp30
- OpenGL NV2X vertex programs

  Runtime profile: CG_PROFILE_VP20

  Compiler option: _profile vp20
- OpenGL NV2X fragment programs

  Runtime profile: CG_PROFILE_FP20

  Compiler option: _profile fp20
- DirectX 9 vertex shaders

  Runtime profiles: CG_PROFILE_VS_2_X

  CG_PROFILE_VS_2_0

  Compiler options:-profile vs_2_x

  -profile vs_2_0
- DirectX 9 pixel shaders

  Runtime profiles: CG_PROFILE_PS_2_X

  CG_PROFILE_PS_2_0

  Compiler options: -profile ps_2_x

  -profile ps_2_0
- DirectX 8 vertex shaders

  Runtime profiles: CG_PROFILE_VS_1_1

Compiler options:-profile vs_1_1

- DirectX 8 pixel shaders

  Runtime profiles: CG_PROFILE_PS_1_3

  CG_PROFILE_PS_1_2

  CG_PROFILE_PS_1_1

  Compiler options: -profile ps_1_3

  -profile ps_1_2

  -profile ps_1_2

  -profile ps_1_1

附：截止到 2009 年 10 月，出现的 profile 已经不止上面这些种类了，尤其是现在 DirectX 已经出到了 11 的版本。上面的 profile 是可以在当前大多数机器上使用的。

第 5 章 CG 数据类型

本章将着重介绍Cg语言中预定义的内置（built in）的、或称为基本（primitive）的数据类型。然后介绍可以用来声明对象的各类类型，主要是数组和结构类型。学习本章时，需要体会内置向量类型和数组类型的区别。

5.1 基本数据类型

Cg 支持 7 种基本的数据类型：

1. float，32 位浮点数据，一个符号位。浮点数据类型被所有的 profile 支持（但是 DirectX8 pixel profiles 在一些操作中降低了浮点数的精度和范围）；

2. half，16 为浮点数据；

3. int，32 位整形数据，有些 profile 会将 int 类型作为 float 类型使用；

4. fixed，12 位定点数，被所有的 fragment profiles 所支持；

5. bool,布尔数据，通常用于 if 和条件操作符（?:），布尔数据类型被所有的 profiles 支持；

6. sampler*,纹理对象的句柄（the handle to a texture object），分为 6 类：sampler, sampler1D, sampler2D, sampler3D, samplerCUBE,和 samplerRECT。DirectX profiles 不支持 samplerRECT 类型，除此之外这些类型被所有的 pixel profiles 和 NV40 vertex program profile 所支持（CgUsersManual 30 页）。由此可见，在不远的未来，顶点程序也将广泛支持纹理操作；

7. string，字符类型，该类型不被当前存在的 profile 所支持，实际上也没有必要在 Cg 程序中用到字符类型，但是你可以通过 Cg runtime API 声明该类型变量，并赋值；因此，该类型变量可以保存 Cg 文件的信息。

前 6 种类型会经常用到，事实上在 Wikipedia 有关 Cg 语言的阐述中只列举了前 6 种类型，而并没有提到 string 数据类型。除了上面的基本数据类型外，Cg 还提供了内置的向量数据类型(built-in vector data types)，内置的向量数据类型基于基础数据类型。例如：float4，表示 float 类型的 4 元向量；bool4，表示 bool 类型 4 元向量。

注意：向量最长不能超过 4 元，即在 Cg 程序中可以声明 float1、float2、float3、float4 类型的数组变量，但是不能声明超过 4 元的向量，例如：

```
float5 array;//编译报错
```

向量初始化方式一般为：

```
float4 array = float4(1.0, 2.0, 3.0, 4.0);
```

较长的向量还可以通过较短的向量进行构建：

```
float2 a = float2(1.0, 1.0);
float4 b = float4(a, 0.0, 0.0);
```

此外，Cg 还提供矩阵数据类型，不过最大的维数不能超过 4*4 阶。例如：

```
float1x1 matrix1;//等价于 float matirx1; x 是字符，并不是乘号！
float2x3 matrix2;// 表示 2*3 阶矩阵，包含 6 个 float 类型数据
float4x2 matrix3;// 表示 4*2 阶矩阵，包含 8 个 float 类型数据
float4x4 matrix4;//表示 4*4 阶矩阵，这是最大的维数
```

矩阵的初始化方式为：

```
float2x3 matrix5 = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0};
```

注意：Cg 中向量、矩阵与数组是完全不同，向量和矩阵是内置的数据类型（矩阵基于向量），而数组则是一种数据结构，不是内置数据类型！这一点和 C\C++中不太一样，在 C\C++中，这三者同属于数据结构，数组可以构建向量和矩阵。下一节中将详细阐述 Cg 中的数组类型。

5.2 数组类型

“General-purpose arrays can only be used as uniform parameters to a vertex program. The intent is to allow an application to pass arrays of skinning matrices and arrays of light parameters to a vertex program”(文献【3】的 Array 章节)。

在着色程序中，数组通常的使用目的是：作为从外部应用程序传入大量参数到 Cg 的顶点程序中的形参接口，例如与皮肤形变相关的矩阵数组，或者光照参数数组等。

简而言之，数组数据类型在 Cg 程序中的作用是：作为函数的形参，用于大量数据的转递。

Cg 中声明数组变量的方式和 C 语言类似：例如：

```
float a[10];//声明了一个数组，包含 10 个 float 类型数据
float4 b[10];//声明了一个数组，包含 10 个 float4 类型向量数据
```

对数组进行初始化的方式为：

```
float a[4] = {1.0, 2.0, 3.0, 4.0}; //初始化一个数组
```

要获取数组长度，可以调用“.length”，例如：

```
float a[10];                    //声明一个数组
int length = a.length;//获取数组长度
```

声明多维数组以及初始化的方式如下所示：

```
float b[2][3] = {{0.0, 0.0, 0.0},{1.0, 1.0, 1.0}};
```

对多维数组取长度的方式为：

```
int length1 = b.length;   // length1 值为 2
int length2 = b[0].length; // length2 值为 3
```

数组和矩阵有些类似，但是并不是相同。 例如 4*4 阶数组的的声明方式为：float M[4][4];4 阶矩阵的声明方式为：float4x4 M。前者是一个数据结构，包含 16 个 float 类型数据，后者是一个 4 阶矩阵数据。float4x4 M[4]，表示一个数组，包含 4 个 4 阶矩阵数据。

进行数组变量声明时，一定要指定数组长度，除非是作为函数参数而声明的形参数组。并且在当前的 profiles 中，数组的长度和所引用的数组元素的地址必须在编译时就知道。

“Unsized arrays may only be declared as function parameters-they may not be declared as variables. Furthermore, in all current profiles, the actual array length and address calculations implied by array indexing must be known at compile time”（文献【3】）。

由于形参数组的概念与函数的概念紧密结合，所以将在第 8 章的 8.1 函数章节中进行统一阐述。

5.3 结构类型

Cg 语言支持结构体（structure），实际上 Cg 中的结构体的声明、使用和 C++非常类似（只是类似，不是相同）。一个结构体相当于一种数据类型，可以定义该类型的变量。引入结构体机制，赋予了 Cg 语言一丝面向对象的色彩。还记得 C++中，结构体与类的区别吗？没有区别，除了默认访问属性在结构体中为 public，类中为 private，所以结构体与类是非常近似的，由此可以看出 shader 语言的发展趋势还是向着具有面向对象特性的高级语言。不过目前的 Cg 语言中的

结构体以展现“封装”功能为主，并不支持继承机制。

结构体的声明以关键字 struct 开始，然后紧跟结构体的名字，接下来是一个大括号，并以分号结尾（不要忘了分号）。大括号中是结构体的定义，分为两大类：成员变量和成员函数。例如，定义一个名为 myAdd 的结构体，包含一个成员变量，和一个执行相加功能的成员函数，然后声明一个该结构体类型的变量，代码为：

```
struct myAdd
{
    float val;
    float add(float x)
    {
       return val + x;
     }
};
myAdd s;
```

使用符号“•”引用结构体中的成员变量和成员函数。例如：

```
float a = s.value;
float b = s.add(a);
```

注意：在当前的所有的 profile 版本下，如果结构体的一个成员函数使用了成员变量，则该成员变量要声明在前。此外，成员函数是否可以重载依赖于使用的 profile 版本。（文献[3] 的 structures and Member functions 章节）

一般来说,Cg 的源代码都会在文件首部定义二个结构体，分别用于定义输人和输出的类型，这二个结构体定义与普通的 C 结构定义不同，除了定义结构体成员的数据类型外，还定义了该成员的绑定语义类型(Binding Semantics)，所谓绑定语义类型是为了与宿主环境进行数据交换的时候识别不同数据类型的。目前

Cg 支持的绑定语义类型包括 POSTION 位置)，COLOR(颜色)，NORMAL(法向量)，Texcoord(纹理坐标)等类型。

当顶点着色程序向片段着色程序传递的数据类型较多的情况下，使用结构体可以大大的方便代码的编写和维护。总而言之，使用结构体是一个好习惯，高智商的孩子都使用。

5.4 接口（Interfaces）类型

Cg 语言提供接口类型，实际上，我本人很少见到使用接口类型的着色程序，原因在于，Cg 语言中的接口类型还不完善，不能被扩展和包含。此外，目前的 GPU 编程大多只是针对独立的算法进行编码，规模较小，使用接口类型没有太大的优势。所以这里对该类型并不多做说明。有兴趣的读者可以参阅文献【3】的 Interfaces 章节。

5.5 类型转换

Cg 中的类型转换和 C 语言中的类型转换很类似。C 语言中类型转换可以是强制类型转换，也可以是隐式转换，如果是后者，则数据类型从低精度向高精度转换。在 Cg 语言中也是如此。例如：

```
float a = 1.0;
half b = 2.0;
float c = a+b;  //等价于 float c = a + (float)b;
```

当有类型变量和无类型常量数据进行运算时，该常量数据不做类型转换，举例如下：

```
float a = 1.0;
float b = a + 2.0; //2.0 为无类型常量数据，编译时作为 float 类型
```

Cg 语言中对于常量数据可以加上类型后缀，表示该数据的类型，例如：

float a = 1.0;

float b = a + 2.0h; //2.0h 为 half 类型常量数据，运算是需要做类型转换

常量的类型后缀（type suffix）有 3 种：

- f ：表示 float;
- h: 表示 half;
- x: 表示 fixed

第 6 章 CG 表达式与控制语句

在上一章中，我们已经介绍了Cg语言的基础数据类型（7种）、内置数据类型，以及数组、结构、接口等类型，本章将在此基础上讨论Cg中的表达式，表达式由操作符（operator）关联一个或多个操作数（operand）构成，我们首先阐述各种类型的操作符，并结合数据类型讲解操作符的具体使用方法。

Cg中的操作符与C语言中的类似（操作符的功能和写法与C相同，但用法不尽相同），按照操作符的功能可以划分为：关系操作符、逻辑操作符、条件操作符。Cg中有一类较为独特的操作符，称为Swizzle操作符， 这个操作符用于取出向量类型变量中的分量。此外，与C语言不同的是，Cg允许在向量类型变量上使用操作符，例如>操作符可以用来比较两个向量各个分量的大小关系。Cg中的表达式还有很多与C语言不同的细节之处，将在本章中一一分说。

6.1 关系操作符（Comparison Operators）

关系操作符，用于比较同类型数据（不同类型的基础数据需要进行类型转换，不同长度的向量，不能进行比较）之间的大小关系或者等价关系。 Cg中有6种关系操作符，如表 1所示, 关系操作符运算后的返回类型均为bool类型。

关系操作符	功能	用法
<	小于	expr < expr
<=	小于或等于	expr <= expr
!=	不等于	expr != expr
==	等于	expr == expr
>=	大于或等于	expr >= expr
>	大于	expr > expr

表 1 关系操作符

在Cg中，由于关系操作符以及下节会讲到的逻辑操作符，都返回bool类型结果，所以这两种操作符有时也被统一称为boolean operator。

Cg语言表达式允许对向量使用所有的boolean operator，如果是二元操作符，则被操作的两个向量的长度必须一致。表达式中向量的每个分量都进行一对一的运算，最后返回的结果是一个bool类型的向量，长度和操作数向量一致。例如：

```
float3 a = float4(0.5, 0.0, 1.0);
float3 b = float4(0.6, -0.1, 0.9);
bool3 c = a<b;
```

运算后向量c的结果为float3(true, false, true);

6.2 逻辑操作符（Logical Operators）

Cg语言中有3种逻辑操作符(也被称为boolean Operators)，如表 2所示，逻辑操作符运算后的返回类型均为bool类型。

逻辑操作符	功能	用法
&&	逻辑与	expr && expr
||	逻辑或	expr || expr
!	逻辑非	!expr

表 2 逻辑操作符

正如上节所说，逻辑操作符也可以对向量使用，返回的变量类型是同样长度的内置bool向量。

有一点需要注意：Cg中的逻辑与（&&）和逻辑或（||）不存在C中的短路现象（short-circuiting，即只用计算一个操作数的bool值即可），而是参与运算的操作数据都进行bool分析。

6.3 数学操作符（Math Operators）

Cg语言对向量的数学操作提供了内置的支持，Cg中的数学操作符有：*乘法；/除法；-取反；+加法；—减法；%求余；++；——；*=；/=；+=；-=；后面四种运算符有时被归纳入赋值操作符，不过它们实际上进行数学计算，然后进行赋值，所以这里也放入数学操作符中进行说明。

在文献【2】第3.3节Math Expressions中，其行文意思容易让人觉得“好像只有加减乘除等运算可以对向量进行”，实际上经过我的测试，++、——等数学运算符同样可以使用在向量上。所以“Cg语言对向量的数学操作提供内置支持”这句话是非常准确的。

需要注意的是：求余操作符%。只能在int类型数据间进行，否则编译器会提示错误信息： error C1021: operands to “%” must be integral.

当使用这些数学操作符对一个标量和一个向量进行运算时，标量首先被复制到一个长度相同的向量中，然后进行运算，例如下面的代码形式是正确的：

```
void function()
{
    float2 a = float2(1.0, 1.0);

    float b = 2.0;

    f *= d;
    f *= 2.0;
}
```

6.4 移位操作符

Cg语言中的移位操作符，功能和C语言中的一样，也可以作用在向量上，但是向量类型必须是int类型。例如：

```
int2 a = int2(0.0,0.0);
int2 b = a>>1;
```

如果使用如下代码，会出现错误提示信息：error C1021:operands to “shr” must be integral.

```
float2 a = int2(0.0,0.0);
float2 b = a>>1;
```

6.5 Swizzle 操作符

可以使用Cg语言中的swizzle操作符（.）将一个向量的成员取出组成一个新的向量。swizzle操作符被GPU硬件高效支持。swizzle操作符后接x、y、z、w，分别表示原始向量的第一个、第二个、第三个、第四个元素；swizzle操作符后接r、g、b和a的含义与前者等同。不过为了程序的易读性，建议对于表示颜色值的向量，使用swizzle操作符后接r、g、b和a的方式。

举例如下：

```
float4(a, b, c, d).xyz      等价于   float3(a, b, c)
float4(a, b, c, d).xyy      等价于   float3(a, b, b)
float4(a, b, c, d).wzyx     等价于   float4(d, c, b, a)
float4(a, b, c, d).w        等价于   float d
```

值得注意的是，Cg语言中float a 和float1 a是基本等价的，两者可以进行类型转换；float、bool、half等基本类型声明的变量也可以使用swizzle操作符。例如：

```
float a = 1.0;
float4 b = a.xxxx;
```

注意：swizzle操作符只能对结构体和向量使用，不能对数组使用，如果对数组使用swizzle操作符则会出现错误信息：error C1010: expression left of .”x” is

not a struct or array（其实个人觉得，提示的错误信息中array换成vector更加合适）。

要从数组中取值必须使用[]符号。例如：

```
float a[3] = {1.0,1.0,0.0};
float b = a[0]; //正确
float c = a.x; //编译会提示错误信息
```

6. 6 条件操作符（Conditional Operators）

条件操作符的语法格式为：

expr1 ? expr2 : expr3;

expr1的计算结果为true或者flase，如果是true,则expr2执行运算，否则expr3被计算。

条件操作符为简单的if-else语句提供了一种便利的替代方式，例如我们可以不必写：

```
if(a < 0){b = a}
else{c = a}
```

而改写为：

（a < 0）?(b = a) :(c = a);

Cg中的条件操作符一个独特的性能是：支持向量运算。即，expr1的计算结果可以是bool型向量，expr2和expr3必须是与expr1长度相同的向量。举例如下：

```
float3 h = float3(-1.0,1.0,1.0);
float3 i = float3(1.0,0.0,0.0);
float3 g = float3(1.0,1.0,0.0);
float3 k;

k = (h < float3(0.0,0.0,0.0))?(i):(g);
```

三元向量h与float3(0.0, 0.0, 0.0)做比较运算后结果为（true, false, false）,所以i的第一个数据赋值给K的第一个数据，g的第二个和第三个数据赋值给k的第二个和第三个数据，K的值为(1.0, 1.0, 0.0)。

6.7 操作符优先顺序

Cg语言中操作符的优先顺序如表 3所示，从上到下表示从高级到低级的优先级；同一行的操作符具有同等优先级。该表参考了Cg教程_可编程实时图形权威指南第3.3.1节。

操作符	结合律	功能
() [] -> .	从左到右	函数调用、数组引用、结构引用、成员选择
! ~ ++ - + - * & (type) sizeof	从右到左	一元操作符：取反、增加、减少、正号、负号、间接、地址、转换
* / %	从左到右	乘法、除法、余数
+ -	从左到右	加法、减法
<< >>	从左到右	移位操作符
< >= > >=	从左到右	关系操作符
== !=	从左到右	等于，不等
&	从左到右	位操作符与
^	从左到右	位操作符异或
|	从左到右	位操作符或
&&	从左到右	逻辑与
||	从左到右	逻辑或
?:	从右到左	条件表达式
= += -= *= /= %= &= ^= != <<= >>=	从右到左	赋值、赋值表达式
,	从左到右	逗号操作符

表 3 操作符优先级

6.8 控制流语句（Control Flow Statement）

程序最小的独立单元是语句（statement），语句一般由分号结尾，缺省情况下，语句是顺序执行的，但是当涉及逻辑判断控制时，就要求有控制流程序语句。控制流程序语句分为条件语句和循环语句，在C语言中，条件语句有if、if-else、switch等，而循环过程则由while、do-while和for语句支持。Cg中的控制流语句和循环语句与C语言类似：条件语句有：if、if-else；循环语句有：while、for。break语句可以和在for语句中使用。

Cg语言中的控制流语句要求其中的条件表达式返回值都是bool类型，这一点是与C语言不同之处（C语言中，条件表达式返回值可以是0、1）

vs_2_x, vp30和vp40这些profile支持分支指令（又称转移指令，branch instruction），for和while循环指令在这些profile中被完全支持。在文献【3】中提到：

“In other profiles, for and while loops may only be used if the compiler can fully unroll them (that is, if the compiler can determine the iteration count at compile time)”。

这句话的意思是“在其他的profiles中，for和while循环只有当确切的知道循环次数时才能被使用”。但经过试验，如果使用“在fp40和ps_3_0之前的”片段profiles编译含义for, while语句时会出现错误提示信息：error c6003：instruction limit of exceeded……。因此，如果没有确切的把握，不要在低级的profiles中使用循环控制语句。

同样，return只能作为最后一条语句出现。函数的递归调用（recursion）在Cg语言中是被禁止的。Switch 、case和default在Cg中作为保留关键字存在，但是它们目前不被任何profile所支持。

第 7 章 输入\输出与语义绑定

第三章从 GPU 运行原理和数据流程的角度阐述了顶点着色程序和片段着色程序的输入输出，即，应用程序（宿主程序）将图元信息（顶点位置、法向量、纹理坐标等）传递给顶点着色程序；顶点着色程序基于图元信息进行坐标空间转换，运算得到的数据传递到片段着色程序中；片段着色程序还可以接受从应用程序中传递的纹理信息，将这些信息综合起来计算每个片段的颜色值，最后将这些颜色值输送到帧缓冲区（或颜色缓冲区）中。

这些是顶点着色程序和片段着色程序的基本功能和数据输入输出，实际上现在的着色程序已经可以接受多种数据类型，并灵活的进行各种算法的处理，如，可以接受光源信息（光源位置、强度等）、材质信息（反射系数、折射系数等）、运动控制信息（纹理投影矩阵、顶点运动矩阵等），可以在顶点程序中计算光线的折射方向，并传递到片段程序中进行光照计算。

这一章节中，我们将讲解Cg语言通过何种机制确定数据类型和传递形式。读者要抱着如下几个问题阅读本章节：

1. 从应用程序传递到GPU的数据，分为图元信息数据（在GPU处理的基本数据如顶点位置信息等）和其他的离散数据（在GPU运行流程中不会发生变化，如材质对光的反射、折射信息），这两种输入数据如何区分？

2. 从应用程序传递到GPU中的图元信息如何区分类型，即，顶点程序怎么知道一个数据是位置数据，而不是法向量数据？

3. 顶点着色程序与片段着色程序之间的数据传递如何进行？

7.1 Cg 关键字

关键字是语言本身所保留的一个字符串集合，用于代表特定的含义，如前面

所讲到的数据类型关键字 int、float 等，以及结构体关键字 struct。Cg 中的关键字很多都是照搬 C\C++中的关键字，不过 Cg 中也创造了一系列独特的关键字，这些关键字不但用于指定输入图元的数据含义（是位置信息，还是法向量信息），本质也则对应着这些图元数据存放的硬件资源（寄存器或者纹理），称之为语义词（Semantics），通常也根据其用法称之为绑定语义词（binding semantics）。

除语义词外，Cg 中还提供了三个关键字，in、out、inout，用于表示函数的输入参数的传递方式，称为输入\输出关键字，这组关键字可以和语义词合用表达硬件上不同的存储位置，即同一个语义词，使用 in 关键字修辞和 out 关键词修辞，表示的图形硬件上不同的寄存器。

Cg 语言还提供两个修辞符：uniform，用于指定变量的数据初始化方式；const 关键字的含义与 C\C++中相同，表示被修辞变量为常量变量。

下面将分别对上述的关键字进行详细阐述。这一章非常关键，尤其是语义词的使用方法和含义，再小的 Cg 程序都需要使用到语义词。

7.2 uniform

Cg 语言将输入数据流分为两类（参见文献[3]Program inputs and Outputs ）：

1. Varying inputs, 即数据流输入图元信息的各种组成要素。从应用程序输入到 GPU 的数据除了顶点位置数据，还有顶点的法向量数据，纹理坐标数据等。Cg 语言提供了一组语义词，用以表明参数是由顶点的哪些数据初始化的。

2. Uniform inputs，表示一些与三维渲染有关的离散信息数据，这些数据通常由应用程序传入，并通常不会随着图元信息的变化而变化，如材质对光的反射信息、运动矩阵等。Uniform 修辞一个参数，表示该参数的值由外部应用程序初始化并传入；例如在参数列表中写：

```
uniform float brightness,
uniform float4x4 modleWorldProject
```

表示从“外部”传入一个 float 类型数据，和一个 4 阶矩阵。“外部”的含义通常是用 OpenGL 或者 DirectX 所编写的应用程序。

使用 Uniform 修辞的变量，除了数据来源不同外，与其他变量是完全一样的。

需要注意的一点是：uniform 修辞的变量的值是从外部传入的，所以在 Cg 程序（顶点程序和片段程序）中通常使用 uniform 参数修辞函数形参，不容许声明一个用 uniform 修辞的局部变量！否则编译时会出现错误提示信息：

Error C5056:’uniform’not allowed on local variable

7.3 const

Cg 语言也提供 const 修辞符，与 C\C++中含义一样，被 const 所修辞的变量在初始化之后不能再去改变它的值。下面的例子程序中有一个声明为 const 的变量被赋值修改：

```
const float a = 1.0;
a = 2.0;   //错误
float b = a++; //错误
```

编译时会出现错误提示信息：error C1026: assignment to const variable。

const 修辞符与 uniform 修辞符是相互独立的，对一个变量既可以单独使用 const 或者 uniform，也可以同时使用。

7.4 输入\输出修辞符（in\out\inout）

参数传递是指：函数调用实参值初始化函数形参的过程。在 C\C++中，根据

形参值的改变是否会导致实参值的改变，参数传递分为“值传递（pass-by-value）”和“引用传递（pass-by-reference）”。按值传递时，函数不会访问当前调用的实参，函数体处理的是实参的拷贝，也就是形参，所以形参值的改变不会影响实参值；引用传递时，函数接收的是实参的存放地址，函数体中改变的是实参的值。C\C++采取指针机制构建引用传递，所以通常引用传递也称为“指针传递”。

Cg 语言中参数传递方式同样分为“值传递”和“引用传递”，但指针机制并不被 GPU 硬件所支持，所以 Cg 语言采用不同的语法修辞符来区别“值传递”和“引用传递”。这些修辞符分别为：

1. in：修辞一个形参只是用于输入，进入函数体时被初始化，且该形参值的改变不会影响实参值，这是典型的值传递方式。

2. out：修辞一个形参只是用于输出的，进入函数体时并没有被初始化，这种类型的形参一般是一个函数的运行结果；

3. inout：修辞一个形参既用于输入也用于输出，这是典型的引用传递。

举例如下：

```
void myFunction(out float x);   //形参 x，只是用于输出
void myFunction(inout float x);   //形参 x，即用于输入时初始化，也用于输出数据
void myFunction(in float x);   //形参 x，只是用于输入
void myFunction(float x);   /等价与  in float x，这种用法和 C\C++完全一致
```

也可以使用 return 语句来代替 out 修辞符的使用。输入\输出修辞符通常和语义词一起使用，表示顶点着色程序和片段着色程序的输入输出。

7.5 语义词（Semantic）与语义绑定（Binding Semantics）

语义词，表示输入图元的数据含义（是位置信息，还是法向量信息），也表明这些图元数据存放的硬件资源（寄存器或者纹理缓冲区）。顶点着色程序和片

段着色程序中 Varying inputs 类型的输入，必须和一个语义词相绑定，这称之为绑定语义（binding semantics）。

7.5.1 输入语义与输入语义的区别

语义概念的提出和图形流水线工作机制大有关系。从前面所讲的 GPU 处理流程中可以看出，一个阶段处理数据，然后传输给下一个阶段，那么每个阶段之间的接口是如何确定的呢？例如：顶点处理器的输入数据是处于模型空间的顶点数据（位置、法向量），输出的是投影坐标和光照颜色；片段处理器要将光照颜色做为输入，问题是“片段处理器怎么知道光照颜色值的存放位置”？

在高级语言中（C/C++），数据从接口的一端流向另一端，是因为提供了数据存放的内存位置（通常是指针信息）；由于 Cg 语言并不支持指针机制，且图形硬件处理过程中，数据通常暂存在寄存器中，故而在 Cg 语言中，通过引入语义绑定（binding semantics）机制，指定数据存放的位置，实际上就是将输入\输出数据和寄存器做一个映射关系（在 OpenGL Cg profiles 中是这样的，但在 DirectX-based Cg profiles 中则并没有这种映射关系）。根据输入语义，图形处理器从某个寄存器取数据；然后再将处理好的数据，根据输出语义，放到指定的寄存器。

记住这一点：语义，是两个处理阶段（顶点程序、片段程序）之间的输入\输出数据和寄存器之间的桥梁，同时语义通常也表示数据的含义，如 POSITION 一般表示参数种存放的数据是顶点位置。

语义，只对两个处理阶段的输入\输出数据有意义，也就是说，语义只有在入口函数中才有效，在内部函数（一个阶段的内部处理函数，和下一个阶段没有数据传递关系）的无效，被忽略（Semantics attached to parameters to non-main functions are ignored(261 页)）；

语义，分为输入语义和输入语义；输入语义和输出语义是有区别的。虽然一些参数经常会使用相同的绑定语义词，例如：顶点 Shader 的输入参数，POSITION

指应用程序传入的顶点位置，而输出参数使用 POSITION 语义就表示要反馈给硬件光栅器的裁剪空间位置，光栅器把 POSITION 当成一个位置信息。虽然两个语义都命名为 POSITION，但却对应着图形流水线上不同的寄存器。

> 说明：在 OpenGL Cg profiles 中，语义绑定指定了输入\输出数据和图形硬件寄存器之间的对应关系；但是在 DirectX Cg profiles 中，则并非如此。在文献【3】的第 25 页写到：
>
> In the OpenGL Cg profiles, binding semantics implicitly specify the mapping of varying inputs to particular hardware registers. However, in DirectX based Cg profiles there is no such implied mapping.
>
> 在文献【3】的第 260 页写到：
>
> Often, these predefined names correspond to the names of hardware registers or API resources

7.5.2 顶点着色程序的输入语义

图 13 所示的这组绑定语义关键字被 Cg 语言的所有 vertex profile 所支持，一些 profile 支持额外的语义词。

POSITION	BLENDWEIGHT
NORMAL	TANGENT
BINORMAL	PSIZE
BLENDINDICES	TEXCOORD0---TEXCOORD7

图 13 定点着色程序输入语义词

语义词 POSITION0 等价于 POSITION，其他的语义词也有类似的等价关系。为了说明语义词的含义，举例如下：

```
in float4 modelPos: POSITION
```

表示该参数中的数据是的顶点位置坐标（通常位于模型空间），属于输入参数，语义词 POSITION 是输入语义，如果在 OpenGL 中则对应为接受应用程序传

递的顶点数据的寄存器（图形硬件上）。

in float4 modelNormal: NORMAL

表示该参数中的数据是顶点法向量坐标（通常位于模型空间），属于输入参数，语义词 NORMAL 是输入语义，如果在 OpenGL 中则对应为接受应用程序传递的顶点法向量的寄存器（图形硬件上）。

注意，上面的参数都被声明为四元向量，通常我们在应用程序涉及的顶点位置和法向量都是三元向量，至于为什么要将三元向量便为四元向量，又称齐次坐标，具体请看附录 A。顶点位置坐标传入顶点着色程序中转化为四元向量，最后一元数据为 1，而顶点法向量传入顶点着色程序中转化为四元向量，最后一元数据为 0。

7.5.3 顶点着色程序的输出语义

顶点程序的输出数据被传入到片断程序中，所以顶点着色程序的输出语义词，通常也是片段程序的输入语义词，不过语义词POSITION除外。

下面这些语义词适用于所有的Cg vertex profiles作为输出语义和Cg fragment profiles的输入语义：POSITION，PSIZE，FOG,COLOR0-COLOR1, TEXCOORD0-TEXCOORD7。

顶点着色程序必须声明一个输出变量，并绑定POSITION语义词，该变量中的数据将被用于，且只被用于光栅化！ 如果没有声明一个绑定POSITION语义词的输出变量，如下所示的代码：

```
void main_v(float4 position: POSITION,
            //out float4 oposition : POSITION,
            uniform float4x4 modelViewProj)
{
     //oposition = mul(modelViewProj,position);
}
```

在使用vp20和vp30编译时会提示错误信息：error C6014: Required output ‘HPOS’ not written。在使用vs_2_0和vs_3_0编译时会提示错误信息：error C6014：Required output ‘POSITION’ not written.

为了保持顶点程序输出语义和片段程序输入语义的一致性，通常使用相同的struct类型数据作为两者之间的传递，这是一种非常方便的写法，推荐使用。例如：

```
struct VertexIn
{
   float4 position : POSITION;
   float4 normal     : NORMAL;

};


struct VertexScreen
{
     float4 oPosition : POSITION;
     float4 objectPos         : TEXCOORD0;
     float4 objectNormal     : TEXCOORD1;
};
```

注意：当使用struct结构中的成员变量绑定语义时，需要注意到顶点着色程序中使用的POSITION语义词，是不会被片段程序所使用的。

如果需要从顶点着色程序向片段程序传递数据，例如顶点投影坐标、光照信息等，则可以声明另外的参数，绑定到TEXCOORD系列的语义词进行数据传递，实际上TEXCOORD系列的语义词通常都被用于从顶点程序向片段程序之间传递数据

当然，你也可以选择不使用struct结构，而直接在函数形参中进行语义绑定。无论使用何种方式，都要记住vertex program中的绑定语义（POSITION除外）的输出形参中的数据会传递到fragment program中绑定相同语义的输入形参中。

7.5.4 片段着色程序的输出语义

片段着色程序的输出语义词较少，通常是COLOR。这是因为片段着色程序运行完毕后，就基本到了GPU流水线的末端了。 片段程序必须声明一个out向量（三元或四元），绑定语义词COLOR，这个值将被用作该片断的最终颜色值。例如：

```
void main_f(out float4 color          : COLOR)
{

   color.xyz = float3(1.0,1.0,1.0);
   color.w = 1.0;
}
```

一些fragment profile支持输出语义词DEPTH，与它绑定的输出变量会设置片断的深度值；还有一些支持额外的颜色输出，可以用于多渲染目标（multiple render targets , MRTs）。

和顶点着色程序一样，片断着色程序也可以将输出对象放入一个结构体中。不过，这种做法未必方便，理由是：片断着色程序的输出对象少，最常用的就是颜色值（绑定输出语义词COLOR），单独的一个向量没有必要放到结构体中。而顶点着色程序输出的对象很多，在有些光照或阴影计算中，往往要输出顶点的世界坐标、法向量、光的反射方向、折射方向、投影纹理坐标等数据，这些数据统一放到结构体中方便管理。

7.5.5 语义绑定方法

入口函数输入\输出数据的绑定语义有 4 四种方法（文献【3】第 260 页）

1. 绑定语义放在函数的参数列表的参数声明后面中：

[const] [in | out | inout]<type><identifier> [:

<binding-semantic>][=<initializer>]

其中，const 作为可选项，修辞形参数据；in、out、inout 作为可选项，说明数据的调用方式；type 是必选项，声明数据的类型；identifier 是必选项，形参变量名；一个冒号“：”加上一个绑定语义，是可选项；最后是初始化参数，是可选项。如下代码所示。形参列表中的参数一、参数二绑定到输入语义；参数三、参数四绑定到输出语义；尽管参数一和参数 3 的绑定语义词一样，但前者是输入语义，后者是输出语义，所以这两个参数数据所对应的硬件位置是不一样的。

```
void mian_v(float4 position_obj : POSITION,
            float3 normal_obj : NORMAL,

            out float4 oPosition : POSITION,
            out float4 oColor    : COLOR,

           uniform float4x4 modelViewProj)
{
    ……
}
```

2. 绑定语义可以放在结构体（struct）的成员变量后面：

struct <struct-tag>

{

<type><identifier> [:<binding-semantic >];

};

举例如下，结构 C2E1v_Outpu 中的 2 个成员变量分别绑定到语义 POSITION 和 COLOR，然后在 C2E1v_green 顶点程序入口函数中输出，所以 C2E1v_Outpu 中的语义是输出语义。

```
struct C2E1v_Output {
   float4 position : POSITION;
   float3 color    : COLOR;
};

C2E1v_Output C2E1v_green(float2 position : POSITION)
{
   C2E1v_Output OUT;
   OUT.position = float4(position,0,1);
   OUT.color = float3(0,1,0);

   return OUT;
}
```

3. 绑定语义词可以放在函数声明的后面，其形式为：

<type> <identifier> (<parameter-list>) [:<binding-semantic]

{

<body>

}

如下代码所示，顶点入口函数的声明后带有“COLOR”语义词，表示该函数需要反馈一个颜色值，所以函数的返回类型为 float4，函数体也必须以 return 语句结束。

```
float4 main_v(float4 position: POSITION,
              out float4 oposition : POSITION,
              uniform float4x4 modelViewProj):COLOR
{
    oposition = mul(modelViewProj,position);

    float4 ocolor = float4(1.0,0,0,0);

    return ocolor;
}
```

4. 最后一种语义绑定的方法是，将绑定语义词放在全局非静态变量的声明后面。其形式为：

<type> <identifier> [:<binding-semantic>][=<initializer>];

这种形式的结构很不紧凑，也不利于代码的维护和阅读，所以并不常见，不建议读者使用。事实上，我在学习和研究过程中也很少碰到这种形式。

第 8 章 函数与程序设计

通过第 5 章到第 7 章的阅读，我们已经知道了怎么声明变量（第 5 章），怎么写表达式和语句（第 6 章），怎么将输入\输出参数绑定到语义词（第 7 章），本章将首先描述 Cg 语言中函数的写法，以及函数是否可以重载；然后阐述顶点\片段着色程序中入口函数的概念（类似 C\C++中的 main()函数）；最后，以 Cg 标准函数库来结束本章。

8.1 函数

函数可以被看作一个由用户定义的操作。Cg 语言中的函数声明形式与 C\C++中相同，由返回类型（return type）、函数名、形参列表（parameter list，位于括号中，并用逗号分隔的参数表）和函数体组成。函数体包含在花括号中。如果没有返回值，则函数的返回类型是 void。下面是函数定义的例子：

```
void myFunc(inout float val )
{

    ……
    val += 10.0;
}
```

```
float myFunc(float vals[])
{
    float sum = 0.0;
     ……..
    return sum;
}
```

注意：如果函数没有返回值，函数的返回类型一定要是 void，否则编译会出现大量的错误，错误信息的大概形式是：

error C0000: syntax error, unexpected’(’at token “(”

error C0501: type name expected at token “(”

error C1110: function “main_v” has no return statement

参数传递机制是函数概念中的重中之重，请参阅 7.4 节“输入\输出修辞符”

中的论述。此外，有一个比较特殊的函数形参类型，不论在 C\C++中还是在 Cg 语言中，都是一个令人头疼的话题，它就是数组形参。

8.1.1 数组形参

在 C\C++中，当一个数组作为函数的形参时，实际上传入的只是指向首元素的指针，并且数组边界被忽略（参阅 stephen C.Dewhurst 所著的《C++必知必会》）。而在 Cg 语言中不存在指针机制（图形硬件不支持），数组作为函数形参，传递的是数组的完整拷贝。关于 Cg 中形参数组的原始资料可以在文献[3]Array 章节中查到：

“The most important difference from C is that arrays are first-class types. That means array assignments actually copy the entire array, and arrays that are passed as parameters are passed by value, rather than by reference”.

这段英文中描述道，Cg 语言中数组是“first-class types”，中文翻译为“第一类数据类型”，所谓“第一类（first-class）”的含义是，强调该类型数据是“不可分解的、最高级别的、不被重述的”，即“第一类数据类型”和“基础数据类型”的概念是近同的。如有兴趣深入了解“first-class”概念，可参阅 Matthieu Sozeau and Nicolas Oury 所著的“First-Class Type Classes”一文。

数组类型变量作为函数形参，可以是一维的也可以是多维的，并且不必声明数组长度，即 Unsized Array。例如：

```
float myFunc( float vals[])
{
    float sum = 0.0;
    for(int i = 0; i< vals.length; i++)
    {
        sum += vals[i];
    }
        return sum;
}
```

myFunc 是一个函数，输入一个数组变量，计算数组中所有数据之和，然后返回 float 类型数据。请注意：数组形参不必指定长度。如果指定了函数中形参数组的长度，那么在调用该函数时实参数组的长度和形参数组的长度必须保持一致，如果没有保持一致，编译时会出现错误提示信息：error C1102: incompatible type for parameter…。

```
float myFunc( float vals[3])
{
    float sum = 0.0;
    for(int i = 0; i< vals.length; i++)
    {
        sum += vals[i];
    }
        return sum;
}

void main(…)
{
    float a[2] = {0.0, 1.0};
    float b[3] = {0.0, 1.0, 2.0};
    myFunc(a); //错误调用，会导致编译错误
    myFunc(b); //正确调用
}
```

对于函数的形参数组最好不要指定长度，这样就可以片配任意长度的参数数组。

如果函数的形参数组是多维数组，其声明方式和上面是一样的，可以不指定长度；如果指定形参数组长度，则实参数组长度必须保持一致。

8.2 函数重载

Cg 语言支持函数重载（Functon Overlaoding），其方式和 C++基本一致，通过形参列表的个数和类型来进行函数区分。例如：

```
bool function(float a, float b)      {return    ( a == b);}
bool function(boo a, bool b)      {return    ( a == b);}
```

Cg 语言标准函数库中绝大部分函数都被重载过。

8.3 入口函数

所谓入口函数，即一个程序执行的入口，例如 C\C++程序中的 main()函数。

通常高级语言程序中只有一个入口函数，不过由于着色程序分为顶点程序和片断程序，两者对应着图形流水线上的不同阶段，所以这两个程序都各有一个入口函数。

顶点程序和片段程序有且只有一个入口函数，当程序进行编译时，需要指定入口函数名称（参阅 4.4 节 CG 编译），除非入口函数名为 main。当我们编写或阅读 Cg 代码时，如何区分哪个函数是入口函数呢？或者哪个入口函数对应着顶点程序或片段程序？事实上，顶点程序和片段程序的入口函数形式，已经完全由它们在渲染管线中所处的阶段所决定。在前面已经阐述过，顶点程序接收应用程序传递的顶点数据（通常位于模型坐标空间），然后进行坐标空间转换和光照处理，最后输出投影坐标和计算得到的光照颜色；而片段程序接收从顶点程序输出的数据，并进行像素颜色计算。在片段程序中往往涉及到纹理颜色的处理，其输入参数中常有纹理形参的声明。所以通过观察程序的输入输出语义绑定（参阅 7.5 节语义词与语义绑定），就可以区分入口函数对应到顶点程序还是片段程序。而内部函数则忽略任何应用到形参上的语义，通常也没有人会在内部函数使用语义词，除非他\她的目的是练习打字速度。

下面的代码展示了一个顶点程序的入口函数，名称为 C2E1v_green，这个顶点着色程序接收二维顶点数据，然后转换为齐次坐标（请思考，顶点和向量的齐次坐标有什么不同？齐次坐标的本质是什么？），并将该顶点设置为绿色，最后使用 return 语句输出。如果电脑安装了 Cg，该程序文件位于“NVIDIA Corporation\Cg\examples\OpenGL\basic\ 01_vertex_program\C2E1v_green.cg”目录

下。

```
struct C2E1v_Output {
    float4 position : POSITION;
    float3 color    : COLOR;
};

C2E1v_Output C2E1v_green(float2 position : POSITION)
{
    C2E1v_Output OUT;

    OUT.position = float4(position,0,1);
    OUT.color = float3(0,1,0);

    return OUT;
}
```

8.4 CG 标准函数库

和 C 的标准函数库类似，Cg 提供了一系列内建的标准函数。这些函数用于执行数学上的通用计算或通用算法（纹理映射等），例如，需要求取入射光线的反射光线方向向量可以使用标准函数库中的 reflect 函数，求取折射光线方向向量可以使用 refract 函数，做矩阵乘法运算时可以使用 mul 函数。

有些函数直接和 GPU 指令相对应，所以执行效率非常高。绝大部分标准函数都被重载过，用于支持不同长度的数组和向量作为输入参数。

Cg 标准函数会随着未来 GPU 硬件的发展而不断优化，所以基于标准函数库写的程序是可以用在以后的 GPU 硬件上的。

Cg 标准函数库主要分为五个部分：

1. 数学函数（Mathematical Functions）；
2. 几何函数(Geometric Functions)；
3. 纹理映射函数(Texture Map Functions)；（作为单独的一章进行讲解）
4. 偏导数函数(Derivative Functions)；

5. 调试函数(Debugging Function)；

8.4.1 数学函数（Mathematical Functions）

表 4 中列举了 Cg 标准函数库中所有的数学函数，这些数学函数用于执行数学上常用计算，包括：三角函数、幂函数、园函数、向量和矩阵的操作函数。这些函数都被重载，以支持标量数据和不同长度的向量作为输入参数。

函数	功能
abs(x)	返回输入参数的绝对值
acos(x)	反余切函数，输入参数范围为[-1,1]， 返回$[0,\pi]$区间的角度值
all(x)	如果输入参数均不为 0，则返回 ture； 否则返回 flase。&&运算
any(x)	输入参数只要有其中一个不为 0，则返回 true。||运算
asin(x)	反正弦函数,输入参数取值区间为$[-1,1]$，返回角度值范围为$\left[-\frac{\pi}{2},\frac{\pi}{2}\right]$
atan(x)	反正切函数，返回角度值范围为$\left[-\frac{\pi}{2},\frac{\pi}{2}\right]$
atan2(y,x)	计算 y/x 的反正切值。实际上和 atan(x)函数功能完全一样，至少输入参数不同。atan(x) = atan2(x, float(1))。
ceil(x)	对输入参数向上取整。例如：ceil(float(1.3)) ，其返回值为 2.0
clamp(x,a,b)	如果 x 值小于 a，则返回 a；如果 x 值大于 b，返回 b；否则，返回 x。
cos(x)	返回弧度 x 的余弦值。返回值范围为$[-1,1]$
cosh(x)	双曲余弦（hyperbolic cosine）函数，计算 x 的双曲余弦值。
cross(A,B)	返回两个三元向量的叉积(cross product)。注意，输入参数必须是三元向量！
degrees(x)	输入参数为弧度值(radians)，函数将其转换为角度值(degrees)

determinant(m)	计算矩阵的行列式因子。
dot(A,B)	返回 A 和 B 的点积(dot product)。参数 A 和 B 可以是标量，也可以是向量（输入参数方面，点积和叉积函数有很大不同）。
exp(x)	计算 e^x 的值，e= 2.71828182845904523536
exp2(x)	计算 2^x 的值
floor(x)	对输入参数向下取整。例如 floor(float(1.3)) 返回的值为 1.0；但是 floor(float(-1.3))返回的值为-2.0。该函数与 ceil(x)函数相对应。
fmod(x,y)	返回 x/y 的余数。如果 y 为 0，结果不可预料。
frac(x)	Returns the fractional portion of a scalar or each vector component
frexp(x, out exp)	将浮点数 x 分解为尾数和指数，即 x = m* 2^exp，返回 m，并将指数存入 exp 中；如果 x 为 0，则尾数和指数都返回 0
isfinite(x)	判断标量或者向量中的每个数据是否是有限数，如果是返回 true；否则返回 false;无限的或者非数据(not-a-number NaN)，
isinf(x)	判断标量或者向量中的每个数据是否是无限，如果是返回 true；否则返回 false;
isnan(x)	判断标量或者向量中的每个数据是否是非数据(not-a-number NaN)，如果是返回 true；否则返回 false;
ldexp(x, n)	计算 $x*2^n$ 的值
lerp(a, b, f)	计算 $(1-f)*a+b*f$ 或者 $a+f*(b-a)$ 的值。即在下限 a 和上限 b 之间进行插值，f 表示权值。注意，如果 a 和 b 是向量，则权值 f 必须是标量或者等长的向量。
lit(NdotL, NdotH, m)	N 表示法向量；L 表示入射光向量；H 表示半角向量；m 表示高光系数。 函数计算环境光、散射光、镜面光的贡献，返回的 4 元向量： X 位表示环境光的贡献，总是 1.0； Y 位代表散射光的贡献，如果 $N \bullet L < 0$，则为 0；否则为 $N \bullet L$ Z 位代表镜面光的贡献，如果 $N \bullet L < 0$ 或者 $N \bullet H < 0$，则位 0；否则为 $(N \bullet H)^m$； W 位始终位 1.0

log(x)	计算 $\ln(x)$ 的值，x 必须大于 0
log2(x)	计算 $\log_2^{(x)}$ 的值，x 必须大于 0
log10(x)	计算 $\log_{10}^{(x)}$ 的值，x 必须大于 0
max(a, b)	比较两个标量或等长向量元素，返回最大值。
min(a,b)	比较两个标量或等长向量元素，返回最小值。
modf(x, out ip)	在 Cg Reference Manual 中没有查到
mul(M, N)	计算两个矩阵相乘，如果 M 为 AxB 阶矩阵，N 为 BxC 阶矩阵，则返回 AxC 阶矩阵。下面两个函数为其重载函数。
mul(M, v)	计算矩阵和向量相乘
mul(v, M)	计算向量和矩阵相乘
noise(x)	噪声函数，返回值始终在 0，1 之间；对于同样的输入，始终返回相同的值（也就是说，并不是真正意义上的随机噪声）。
pow(x, y)	x^{y}
radians(x)	函数将角度值转换为弧度值
round(x)	Round-to-nearest，或 closest integer to x 即四舍五入。
rsqrt(x)	X 的反平方根，x 必须大于 0
saturate(x)	如果 x 小于 0，返回 0；如果 x 大于 1，返回 1；否则，返回 x
sign(x)	如果 x 大于 0，返回 1；如果 x 小于 0，返回 01；否则返回 0。
sin(x)	输入参数为弧度，计算正弦值，返回值范围为 $[-1,1]$
sincos(float x, out s, out c)	该函数是同时计算 x 的 sin 值和 cos 值，其中 s=sin(x)，c=cos(x)。该函数用于“同时需要计算 sin 值和 cos 值的情况”，比分别运算要快很多!
sinh(x)	计算双曲正弦（hyperbolic sine）值。
smoothstep(min, max, x)	值 x 位于 min、max 区间中。如果 x=min，返回 0；如果 x=max，返回 1；如果 x 在两者之间，按照下列公式返回数据： $-2*(\frac{x-\min}{\max-\min})^3+3*(\frac{x-\min}{\max-\min})^2$
step(a, x)	如果 x<a，返回 0；否则，返回 1。
sqrt(x)	求 x 的平方根，$\sqrt{x}$，x 必须大于 0。

tan(x)	输入参数为弧度，计算正切值
tanh(x)	计算双曲正切值
transpose(M)	M 为矩阵，计算其转置矩阵

表 4 Cg 标准函数库中的数学函数

8.4.2 几何函数（Geometric Functions）

几何函数，如表 5 所示，用于执行和解析几何相关的计算，例如根据入射光向量和顶点法向量，求取反射光和折射光方向向量。Cg 语言标准函数库中有 3 个几何函数会经常被使用到，分别是：normalize 函数，对向量进行归一化；reflect 函数，计算反射光方向向量；refract 函数，计算折射光方向向量。大声呐喊，并要求强烈注意：

1. 着色程序中的向量最好进行归一化之后再使用，否则会出现难以预料的错误；

2. reflect 函数和 refract 函数都存在以“入射光方向向量”作为输入参数，注意这两个函数中使用的入射光方向向量，是从外指向几何顶点的；平时我们在着色程序中或者在课本上都是将入射光方向向量作为从顶点出发。

函数	功能
distance(pt1, pt2)	两点之间的欧几里德距离（Euclidean distance）
faceforward(N,I,Ng)	如果 $Ng \bullet I < 0$，返回 N；否则返回-N。
length(v)	返回一个向量的模，即 sqrt(dot(v,v))
normalize(v)	归一化向量
reflect(I, N)	根据入射光方向向量 I，和顶点法向量 N，计算反射光方向向量。其中 I 和 N 必须被归一化，需要非常注意的是，这个 I 是指向顶点的；函数只对三元向量有效。
refract(I,N,eta)	计算折射向量，I 为入射光线，N 为法向量，eta 为折射系数；其中 I 和 N 必须被归一化，如果 I 和 N 之间的夹角太大，则返回（0，0，0），也就是没有折射光线；I 是指向顶点的；函数只对三

	元向量有效。

表 5 Cg 标准函数库几何函数

8.4.3 纹理映射函数（Texture Map Functions）

下表提供 Cg 标准函数库中的纹理映射函数。这些函数被 ps_2_0、ps_2_x、arbfp1、fp30 和 fp40 等 profiles 完全支持（fully supported）。所有的这些函数返回四元向量值。

函数
tex1D(sampler1D tex, float s) 一维纹理查询
tex1D(sampler1D tex, float s, float dsdx, float dsdy) 使用导数值（derivatives）查询一维纹理
Tex1D(sampler1D tex, float2 sz) 一维纹理查询，并进行深度值比较
Tex1D(sampler1D tex, float2 sz, float dsdx,float dsdy) 使用导数值（derivatives）查询一维纹理， 并进行深度值比较
Tex1Dproj(sampler1D tex, float2 sq) 一维投影纹理查询
Tex1Dproj(sampler1D tex, float3 szq) 一维投影纹理查询，并比较深度值
Tex2D(sampler2D tex, float2 s) 二维纹理查询
Tex2D(sampler2D tex, float2 s, float2 dsdx, float2 dsdy) 使用导数值（derivatives）查询二维纹理
Tex2D(sampler2D tex, float3 sz) 二维纹理查询，并进行深度值比较
Tex2D(sampler2D tex, float3 sz, float2 dsdx,float2 dsdy) 使用导数值（derivatives）查询二维纹理，并进行深度值比较
Tex2Dproj(sampler2D tex, float3 sq) 二维投影纹理查询
Tex2Dproj(sampler2D tex, float4 szq) 二维投影纹理查询，并进行深度值比较
texRECT(samplerRECT tex, float2 s)
texRECT (samplerRECT tex, float2 s, float2 dsdx, float2 dsdy)
texRECT (samplerRECT tex, float3 sz)
texRECT (samplerRECT tex, float3 sz, float2 dsdx,float2 dsdy)

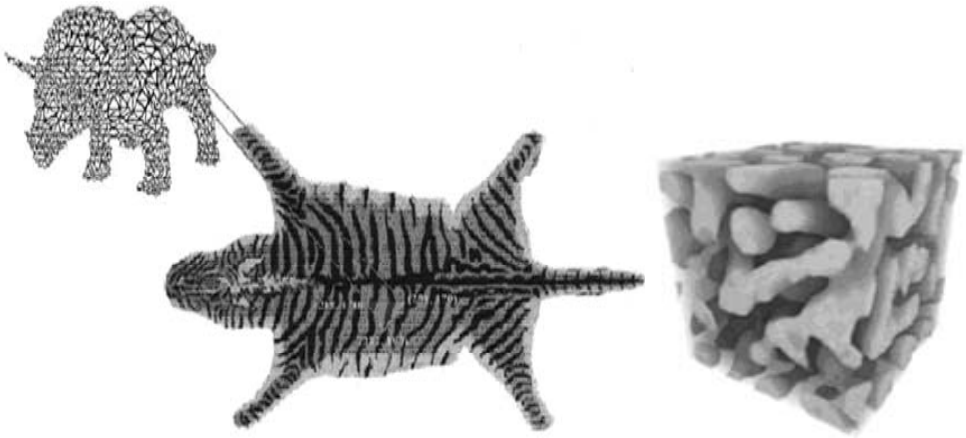
texRECT proj(samplerRECT tex, float3 sq)
texRECT proj(samplerRECT tex, float3 szq)
Tex3D(sampler3D tex, float s) 三维纹理查询
Tex3D(sampler3D tex, float3 s, float3 dsdx, float3 dsdy) 结合导数值（derivatives）查询三维纹理
Tex3Dproj(sampler3D tex, float4 szq) 查询三维投影纹理，并进行深度值比较
texCUBE(samplerCUBE tex, float3 s) 查询立方体纹理
texCUBE (samplerCUBE tex, float3 s, float3 dsdx, float3 dsdy) 结合导数值（derivatives）查询立方体纹理
texCUBEproj (samplerCUBE tex, float4 sq) 查询投影立方体纹理

表 6 Cg 标准函数库纹理映射函数

s 象征一元、二元、三元纹理坐标；z 代表使用“深度比较（depth comparison）”的值；q 表示一个透视值（perspective value,其实就是透视投影后所得到的齐次坐标的最后一位），这个值被用来除以纹理坐标（S），得到新的纹理坐标（已归一化到 0 和 1 之间）然后用于纹理查询。

纹理函数非常多，总的来说，按照纹理维数进行分类，即：1D 纹理函数，2D 纹理函数，3D 纹理函数，已经立方体纹理。需要注意，TexREC 函数查询的纹理实际上也是二维纹理。

3D 纹理，另一个比较学术化的名称是“体纹理（Volume Texture）”，体纹理通常用于体绘制，体纹理用于记录空间中的体细节数据。通常的二维纹理如图 14 左边所示，记录每个三角面皮上的颜色信息，但并不会记录体内的信息；体纹理记录三个方向，从里到外的信息，如图右边所示。在第 15 章和 16 章将会详细阐述体绘制算法和体纹理知识。



二维纹理　　三维纹理

图 14 二维纹理与三维纹理比较

还有一类较为特殊的纹理查询函数以 proj 结尾，主要是针对投影纹理进行查询。所谓投影纹理是指：将纹理当做一张幻灯片投影到场景中，使用投影纹理技术需要计算投影纹理坐标，然后使用投影纹理坐标进行查询。使用投影纹理坐标进行查询的函数就是投影纹理查询函数。

本质来说，投影纹理查询函数和普通的纹理查询函数没有什么不同，唯一的区别在于“投影纹理查询函数使用计算得到的投影纹理坐标，并在使用之前会将该投影纹理坐标除以透视值”。举例而言，计算得到的投影纹理坐标为 float4 uvproj，使用二维投影纹理查询函数：

```
tex2Dproj(texture,uvproj);
```

等价于按如下方法使用普通二维纹理查询函数：

```
float4 uvproj = uvproj/uvproj.q;
tex2D(texture,uvproj);
```

8.4.4 偏导函数（Derivative Functions）

函数	功能
ddx(a)	参数 a 对应一个像素位置，返回该像素值在 X 轴上的偏导数
ddy(a)	参数 a 对应一个像素位置，返回该像素值在 X 轴上的偏导数

表 7 Cg 标准函数库偏导函数

偏导函数的用法很容易让人困惑，因为找不到非常信息的解释说明，当我查找这两个函数的中文资料时，给出的解释通常都是“ddx：返回关于屏幕坐标 x 轴的偏导数；ddy：返回关于屏幕坐标 y 轴的偏导数”。这个简单且单纯的解释完全可以让人陷入长时间的迷惑中，并进而怀疑自己的智商。于是我查找 Cg Refernece Manual 中关于 ddx 函数的定义(ddy 函数的定义与之类似)，原文是“returns approximate partial derivative with respect to window-space X”。看来是典型的上梁不正下梁歪。学习一个函数，最重要的是明白其输入\输出参数的意义，以及有何作用。下面我们带着问题来学习这两个函数：

1. 函数 ddx 和 ddy 是做什么用的，即对输入参数进行了哪些处理？
2. 函数 ddx 和 ddy 的输入参数的意义？是纹理坐标，还是颜色值？
3. 函数 ddx 和 ddy 输出量表达了何种含义？

下面这段话引自某个论坛上的技术讨论贴，作者不详：

If you evaluate ddx (myVar), the GPU will give you the difference between the value of myVar at the current pixel and its value at the pixel next door. It's a straight linear difference, made efficient by the nature of GPU SIMD architectures (neighboring pixels will be calculated simultaneously).The derivative of any uniform value will always be zero.Because these derivatives are always linear, the second derivatives—for example, ddx(ddx(myVar))—will always be zero.

上面这句话的意思是，如果函数 ddx 的参数为 myVar，该参数对应的像素点记为 $p(i,j)$，则 ddx(myVar)的值为“像素点 $p(i+1,j)$ 的值减去 myVar”。同理，

ddy(myVar)的值为“像素点 $p(i, j+1)$ 的值减去 myVar”。如果函数 ddx 和 ddy 的输入参数为常数，则函数返回值永远为 0。

这里面存在像素点所对应数据的类型问题。从前面的知识可知，传入片段程序的顶点属性一般有：屏幕坐标空间的顶点齐次坐标、纹理坐标、法向量、光照颜色等。假设传递给 ddx\ddy 函数的参数 myVar 是纹理坐标，则，ddx(myVar)的值为，纹理上像素点 $p(i+1, j)$ 的纹理颜色值减去 myVar 对应的纹理颜色值。

现在我们可以回答上面的三个问题：

1. 函数 ddx 和 ddy 用于求取相邻像素间某属性的差值；
2. 函数 ddx 和 ddy 的输入参数通常是纹理坐标；
3. 函数 ddx 和 ddy 返回相邻像素键的属性差值；

下面的这段文字来自于 GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics，24.2. Understanding GPU Derivatives，by Randima Fernando.

Complex filtering depends on knowing just how much of the texture (or shading) we need to filter. Modern GPUs such as the GeForce FX provide partial derivative functions to help us. For any value used in shading, we can ask the GPU: "How much does this value change from pixel to pixel, in either the screen-x or the screen-y direction?"

These functions are ddx() and ddy(). Although they are little used, they can be very helpful for filtering and antialiasing operations.

中文含义是：现代的 GPU 中提供了计算 partial derivative 的指令（通常被称为梯度指令或者偏导数指令，DDX 或 DDY），Cg 中所对应的函数为 ddx()和 ddy()(注：glsl 和 hlsl 中也有该函数)。按照屏幕相关空间 x 或 y 计算偏导数，对纹理滤镜以及抗锯齿等非常有用（注：也可以用于 TXD 纹理查找的参数计算）。

学过高等数学应该知道，偏导数的物理含义是：在某一个方向上的变化快慢。所以 ddx 求的是 X 方向上，相邻两个像素的某属性值的变化量；ddy 球的是 Y 方向上，相邻两个像素的某属性值的变化量。

正是由于 ddx 和 ddy 指令是作用于像素级的，所以 ddx 和 ddy 函数只被片段程序所支持。

当在纹理查询函数中使用 ddx 和 ddy 是可以进行图形过滤。所谓图像过滤，一个比较简单的定义是：对于给定的输入图像 A，要创建新的图像 B，把源图像 A 变换到目标图像 B 的操作就是图像滤波。最一般的变换是调整图像大小、锐化、变化颜色，以及模糊图像等。复杂的过滤有赖于知道究竟需要过滤多少纹理。象 GeForce 那样的现代 GPU 提供的偏导函数可以帮助我们。

图 15展示了对二维纹理正常采样时的效果：

图 15 二维纹理正常采样效果图

图展示了tex2D(sampler2D tex , float3 sz ,float2 dsdx , float2 dsdy)函数进行纹理采样时，分别取x,y方向的偏导数为（0.01，0.01）、（0.02，0.02）、（0.0，0.015）时的效果。

图 16 使用偏导函数进行采样的效果图

8.4.5 调试函数（Debugging Function）

函数	功能
void debug(float4 x)	如果在编译时设置了 DEBUG，片段着色程序中调用该函数可以将值 x 作为 COLOR 语义的最终输出；否则该函数什么也不做。

表 8 Cg 标准函数库调试函数

首先顶点 profiles 不支持调试函数。总的来说，目前所有的 shader language 中的调试功能都非常有限，表 8 中列出的 debug 函数实际上十分鸡肋，能起到的作用非常有限的，随着而来的影响是“难以查找程序逻辑错误”。目前对 Cg 程序无法像 C++程序一样进行运行调试，步步跟踪。

8.5 在未来将被解决的问题

我们设想一下，未来的着色语言会解决哪些问题。要回答这个问题，首先需要知道“我们想让着色语言提供那些特性”。首先，我们希望着色语言可以做到

方便进行通用计算编程，而不被数据之间的独立性所限制；其次，我们希望着色语言可以在控制语句方面有所加强；此外，着色语言还需要引入继承机制，让其面向对象的特征更加明显；要加强调试功能，可以对程序执行过程进行调试（现阶段好像比较困难），如果没有很好的调试功能，也希望可以有一个比较通用的IDE 提供给编程者；

此外，未来的可编程图形硬件的顶点处理器要能进行纹理信息的查询；我们还希望越来越的领域可以被 GPU 编程所介入，而不会因为其学习的难度被限制。

总的来说，写到这里，我觉得自己就是一个书呆子，希望的太多了，还是先去吃饭吧。

开篇语：

从第9章到13章，针对光照模型和阴影算法的相关知识点进行讲解，这些算法都用于面绘制，即计算物体表面某点的颜色值；从第14章到15章，将重点阐述体绘制算法。面绘制和体绘制的区别从字面意思上就可以理解一二，体绘制专注于表现物体内部结构，例如一个房子模型，使用面绘制算法，可以让用户观察到房子的外部特征，而使用体绘制算法，则可以让用户观察到墙体中的电缆水管。不过，光照算法、阴影算法等也可以用于体绘制，加强真实感。

光照模型（illumination model），也称为明暗模型，用于计算物体某点处的光强（颜色值）。从算法理论基础而言，光照模型分为两类：一种是基于物理理论的，另一种是基于经验模型的。基于物理理论的光照模型，偏重于使用物理的度量和统计方法，比较典型的有ward BRDF模型，其中的不少参数是需要仪器测量的，使用这种光照模型的好处是“效果非常真实”，但是“计算复杂，实现起来也较为困难”；经验模型更加偏重于使用特定的概率公式，使之与一组表面类型相匹配，所以经验模型大都比较简洁，效果偏向理想化。其实两者之间的界限并不是明确到“非黑即白”的地步，无论何种光照模型本质上还是基于物理的，只不过在求证方法上各有偏重而已。

从使用角度而言，光照模型分为局部光照模型和全局光照模型。所谓局部光照模型，是将光照的种类进行分解，在计算时只考虑其中的一种；而全局光照模型则是考虑到所有的光照种类。一个比较类似的例子是物理力学，牛顿力学最初是考虑理想状态下的运动方式，无摩擦力；然后慢慢的会在力的条件中考虑到摩擦力因素；接着会学习弹性系数。总之是将一个原本复杂的过程分解为各种子过程，然后渐进叠加。

多说一句，光照模型的英文缩写IM最好还是记住，在openGPU网站上注册时，会被要求填写该英文缩写。

第 9 章 经典光照模型（illumination model）

天地开辟 日月重光
---司马懿，征东辽歌

这一章起两个作用，其一阐述经典光照模型算法，其二，阐述如何使用CG语言实现这些算法，让读者在实践中学习到CG编程的方法。这种原理加算法程序实现的讲述方法，同样会用在下面的章节中。原理阐述的过程难免是枯燥的，我会尽量在逻辑安排和讲解方式上减轻这种感觉。

当光照射到物体表面时，一部分被物体表面吸收，另一部分被反射，对于透明物体而言，还有一部分光穿过透明体，产生透射光。被物体吸收的光能转化为热量，只有反射光和透射光能够进入眼睛，产生视觉效果。通过反射和透射产生的光波（光具有波粒二相性）决定了物体呈现的亮度和颜色，即反射和投射光的强度决定了物体表面的亮度，而它们含有的不同波长光的比例决定了物体表面的色彩。

所以，物体表面光照颜色由入射光、物体材质，以及材质和光的交互规律共同决定。

光与物体最基本的交互方式就是反射，遵循反射定律：反射光与入射光位于表面法向两侧，对理想反射面（如镜面），入射角等于反射角，观察者只能在表面法向的反射方向一侧才能看到反射光。 不过世界上并不存在真正的理想反射体，正如物理学中绝对的匀速状态是不存在的。

9.1 光源

环境光(Ambient Light):从物体表面所产生的反射光的统一照明，称为环境光或背景光（计算机图形学第二版 389 页）。例如房间里面并没有受到灯光或者太阳光的直接照射，而是由墙壁、天花板、地板及室内各物体之间光的多次反射进行自然照明。通常我们认为理想的环境光具有如下特性：没有空间或方向性；在

所有方向上和所有物体表面上投射的环境光强度是统一的恒定值。

由于环境光给予物体各个点的光照强度相同，且没有方向之分，所以在只有环境光的情况下，同一物体各点的明暗程度均一样，因此，只有环境光是不能产生具有真实感的图形效果。

环境光是对光照现象的最简单抽象，局限性很大。它仅能描述光线在空间中无方向并均匀散布时的状态。真实的情况是：光线通常都有方向。点光源是发光体的最简单的模型，光线由光源出发向四周发散。还有一种是平行光，即光线都从同一个方向照射。通过模拟方向光和物体表面的交互模式，可以渲染出具有高真实感（明暗变化、镜面反射等）的三维场景。

9.2 漫反射与 Lambert 模型

粗糙的物体表面向各个方向等强度地反射光，这种等同地向各个方向散射的现象称为光的漫反射（diffuse reflection）。产生光的漫反射现象的物体表面称为理想漫反射体，也称为朗伯（Lambert）反射体。

对于仅暴露在环境光下的朗伯反射体，可以用公式(9-1)表示某点处漫反射的光强：

$$I_{ambdiff} = k_d I_a \qquad (9\text{-}1)$$

其中 I_a 表示环境光强度（简称光强），k_d（0< k_d <1）为材质对环境光的反射系数，$I_{ambdiff}$ 是漫反射体与环境光交互反射的光强。

即使一个理想的漫反射体在所有方向上具有等量的反射光线，但是表面光强还依赖于光线的入射方向（方向光）。例如，入射光方向垂直的表面与入射光方向成斜角的表面相比，其光强要大的多。这种现象可以用 Lambert 定律进行数学上的量化。

即，当方向光照射到朗伯反射体上时，漫反射光的光强与入射光的方向和入

射点表面法向夹角的余弦成正比，这称之为 Lambert 定律，并由此构造出 Lambert 漫反射模型：

$$I_{ldiff} = k_d I_l \cos\theta \qquad (9\text{-}2)$$

I_l 是点光源强度，θ 是入射光方向与顶点法线的夹角，称为入射（$0 \leqslant \theta \leqslant 90°$），$I_{ldiff}$ 是漫反射体与方向光交互反射的光强。入射角为零时，说明光线垂直于物体表面，漫反射光强最大；90° 时光线与物体表面平行，物体接收不到任何光线。

若 N 为顶点单位法向量，L 表示从顶点指向光源的单位向量（注意，是由顶点指向光源，不要弄反了），则 $\cos\theta$ 等价于 N 与 L 的点积。所以公式(9-2)可以表示为公式(9-3)：

$$I_{ldiff} = k_d I_l (N \bullet L) \qquad (9\text{-}3)$$

综合考虑环境光和方向来，Lambert 光照模型可写为：

$$I_{diff} = I_{ambdiff} + I_{ldiff} = k_d I_a + k_d I_l (N \bullet L) \qquad (9\text{-}4)$$

9.2.1 漫反射渲染

这一节给出漫反射模型的渲染代码。正如前面所言，着色程序分为顶点着色程序和片段着色程序，顶点着色程序工作在顶点着色器上，只对传入的顶点数据进行处理，也就是说“不对面片的内部点进行处理”。顶点着色程序和片段着色程序可以写在同一个文件中，也可以写在不同的文件中。也可以只有顶点着色程序，而没有片段着色程序。

由于这是本书的第一个着色程序，也就是 Hello Cg World，所以首先只使用顶点着色程序进行漫反射光照渲染，待读者掌握顶点着色程序的写法后，写片段着色程序就会感觉比较轻松。本节同时也会给出在 Cg 语言中使用结构体的编码方法。漫反射光照模型的顶点着色代码如下所示：

代码 1 漫反射光照模型顶点着色程序

```
void main_v(float4 position    : POSITION,
            float4 normal      : NORMAL,

            out float4 oPosition : POSITION,
            out float4 color     : COLOR,

            uniform float4x4 modelViewProj,
            uniform float4x4 worldMatrix,
            uniform float4x4 worldMatrix_IT,
            uniform float3 globalAmbient,
            uniform float3 lightPosition,
            uniform float3 lightColor,
            uniform float3 Kd)
{
    oPosition = mul(modelViewProj, position);

    float3 worldPos = mul(worldMatrix, position).xyz;
    float3 N = mul(worldMatrix_IT, normal).xyz;
    N = normalize(N);

    //计算入射光方向
    float3 L = lightPosition - worldPos;
    L = normalize(L);

    //计算方向光漫反射光强
    float3 diffuseColor = Kd*lightColor*max(dot(N, L), 0);

    //计算环境光漫反射光强
    float3 ambientColor = Kd*globalAmbient;

    color.xyz = diffuseColor+ambientColor;
    color.w = 1;
}
```

图 17 展示了使用漫反射光照模型的渲染效果。

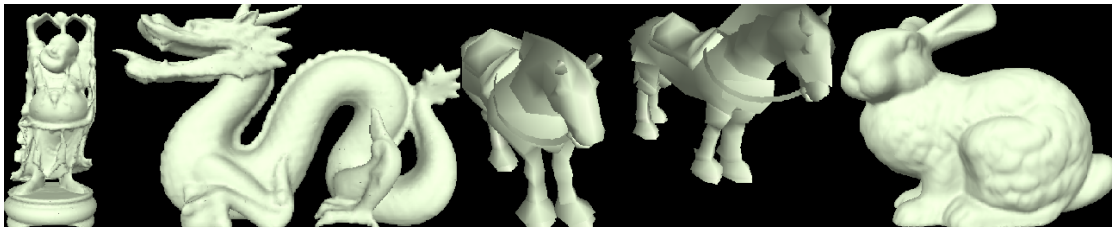


图 17 漫反射光照模型渲染效果

下面给出使用结构体的代码形式：

代码 2 漫反射光照模型顶点着色程序（使用结构体）

```
struct VertexIn
{
   float4 position : POSITION;
   float4 normal     : NORMAL;
};
struct VertexScreen
{
    float4 oPosition : POSITION;
    float4 color         : COLOR;
};
void main_v(VertexIn posIn,
                out VertexScreen posOut,

                uniform float4x4 modelViewProj,
                uniform float4x4 worldMatrix,
                uniform float4x4 worldMatrix_IT,
                uniform float3 globalAmbient,
                uniform float3 lightPosition,
                uniform float3 lightColor,
                uniform float3 Kd)
{
   posOut.oPosition = mul(modelViewProj, posIn.position);

   float3 worldPos = mul(worldMatrix, posIn.position).xyz;
   float3 N = mul(worldMatrix_IT, posIn.normal).xyz;
   N = normalize(N);

   //计算入射光方向
   float3 L = lightPosition - worldPos;
   L = normalize(L);

   //计算方向光漫反射光强
   float3 diffuseColor = Kd*lightColor*max(dot(N, L), 0);

   //计算环境光漫反射光强
   float3 ambientColor = Kd*globalAmbient;

   posOut.color.xyz = diffuseColor+ambientColor;
   posOut.color.w = 1;
}
```

前面已经说过，在着色程序中使用结构体会使得代码易编写、易维护。本书下面的代码中都会使用结构体形式。

9.3 镜面反射与 Phong 模型

Lambert 模型较好地表现了粗糙表面上的光照现象，如石灰粉刷的墙壁、纸张等，但在用于诸如金属材质制成的物体时，则会显得呆板，表现不出光泽，主要原因是该模型没有考虑这些表面的镜面反射效果。一个光滑物体被光照射时，可以在某个方向上看到很强的反射光，这是因为在接近镜面反射角的一个区域内，反射了入射光的全部或绝大部分光强，该现象称为镜面反射。

故此，Phong Bui Tuong 提出一个计算镜面反射光强的经验模型，称为 phong 模型，认为镜面反射的光强与反射光线和视线的夹角相关，其数学表达如公式(9-5)所示：

$$I_{spec} = k_s I_l (V \bullet R)^{n_s} \qquad (9-5)$$

k_s 为材质的镜面反射系数，n_s 是高光指数，V 表示从顶点到视点的观察方向，R 代表反射光方向。

高光指数反映了物体表面的光泽程度。n_s 越大，反射光越集中，当偏离反射方向时，光线衰减的越厉害，只有当视线方向与反射光线方向非常接近时才能看到镜面反射的高光现象，此时，镜面反射光将会在反射方向附近形成亮且小的光斑；n_s 越小，表示物体越粗糙，反射光分散，观察到的光斑区域小，强度弱。

反射光的方向 R 可以通过入射光方向 L（从顶点指向光源）和物体法向量 N 求出：

$$R + L = (2N \bullet L)N \qquad (9-6)$$

所以有：

$$R = (2N \bullet L)N - L \qquad (9\text{-}7)$$

实际上，Cg 语言标准函数库中有求取反射光方向的函数（参阅 8.3.2 节），不过掌握求取反射光方向的方法是有必要的，很多公司在考计算机图形学算法时会要求你写出这个公式，到时候诸如“我知道某某函数可以实现这个功能”之类的话语是不会打动面试官的。自从微软经常拿 VS 中的一些函数要求面试者写出实现时，这种笔试方法就形成了一种潮流，不过如果有可能的话，我很想对这种笔试方法说：“滚”！

9.3.1 phong 模型渲染

Phong 光照模型的渲染代码如代码 3 所示。图 18 展示了在顶点着色程序中进行 phong 光照渲染的效果：

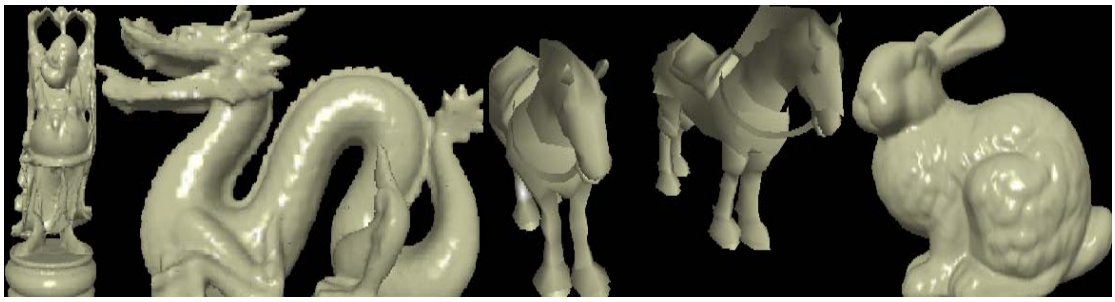


图 18　phong 光照模型的顶点着色程序渲染效果

从图 18 可以看出，与漫反射模型的渲染效果相比，phong 光照模型的渲染效果要圆润很多，明暗界限分明，光斑效果突出。不过请注意图 18 中马的渲染效果，可以很清楚的发现，马的渲染效果没有其他三个模型好，原因在于马模型的面片少，是低精度模型，而顶点着色渲染只对几何顶点进行光照处理，并不会对内部点进行处理。

为了使得低精度模型也能得到高质量的渲染效果，就必须进行片段渲染，所以本节中我们还将给出使用片段着色程序的 phong 光照模型渲染代码和效果。

代码 3 phong 光照模型的顶点着色程序实现

```
struct VertexIn
{
   float4 position : POSITION;            // Vertex in object-space
   float4 normal     : NORMAL;

};
struct VertexScreen
{
    float4 oPosition : POSITION;
    float4 color         : COLOR;

};
void main_v(    VertexIn posIn,
                    out VertexScreen posOut,
                      uniform float4x4 modelViewProj,
                      uniform float4x4 worldMatrix,
                      uniform float4x4 worldMatrix_IT,
                      uniform float3 globalAmbient,
                      uniform float3 eyePosition,
                      uniform float3 lightPosition,
                      uniform float3 lightColor,
                      uniform float3 Kd,
                      uniform float3 Ks,
                      uniform float    shininess)
{
    posOut.oPosition = mul(modelViewProj, posIn.position);

    float3 worldPos = mul(worldMatrix, posIn.position).xyz;
    float3 N = mul(worldMatrix_IT, posIn.normal).xyz;
    N = normalize(N);

   //计算入射光方向、视线方向、反射光线方向
   float3 L = normalize(lightPosition - worldPos);
   float3 V = normalize(eyePosition - worldPos);
   float3 R = 2*max(dot(N, L), 0)*N-L;
   R = normalize(R);

   // 计算漫反射分量
   float3 diffuseColor = Kd * globalAmbient+Kd*lightColor*max(dot(N, L), 0);
   //计算镜面反射分量
   float3 specularColor = Ks * lightColor*pow(max(dot(V, R), 0), shininess);

   posOut.color.xyz = diffuseColor + specularColor;
   posOut.color.w = 1;
}
```

下面给出同时使用顶点着色程序和片段着色程序的 phong 光照模型代码。依然是首先定义结构体，用来包含输入、输出数据流，不过这里使用的结构体（代码 4）和代码 3 中的有所不同，在 VertexScreen 结构体中有两个绑定到 TEXCOORD 语义词的变量，objectPos 和 objectNormal，这两个变量用于传递顶点模型空间坐标和法向量坐标到片段着色器中。

代码 4 phong 光照模型片段着色实现的结构体

```
struct VertexIn
{
   float4 position    : POSITION;
   float4 normal      : NORMAL;
};

struct VertexScreen
{
    float4 oPosition          : POSITION;
    float4 objectPos          : TEXCOORD0;
    float4 objectNormal       : TEXCOORD1;
};
```

代码 5 展示了当前的顶点着色程序代码，其所做的工作有两点：首先将几何顶点的模型空间坐标转换为用于光栅化的投影坐标；然后将顶点模型坐标和法向量模型坐标赋值给绑定 TEXCOORD 语义词的变量，用于传递到片段着色程序中。

代码 5 phong 光照模型顶点着色程序

```
void main_v(VertexIn posIn,
            out VertexScreen posOut,
            uniform float4x4 modelViewProj)
{
  posOut.oPosition = mul(modelViewProj, posIn.position);
  posOut.objectPos = posIn.position;
  posOut.objectNormal = posIn.normal;
}
```

代码 6 展示了使用 phong 光照模型渲染的片段着色程序。我将反射光方向、视线方向、入射光方向都放在片段着色程序中计算，实际上这些光照信息也可以放到顶点着色程序中计算，然后传递到片段着色程序中。

代码 6 phong 光照模型片段着色程序

```
void main_f(    VertexScreen posIn,
                out float4 color         : COLOR,

                  uniform float4x4 worldMatrix,
                  uniform float4x4 worldMatrix_IT,
                  uniform float3 globalAmbient,
                  uniform float3 eyePosition,
                  uniform float3 lightPosition,
                  uniform float3 lightColor,
                  uniform float3 Kd,
                  uniform float3 Ks,
                  uniform float    shininess)
{
    float3 worldPos = mul(worldMatrix, posIn.objectPos).xyz;
    float3 N = mul(worldMatrix_IT, posIn.objectNormal).xyz;
    N = normalize(N);

   //计算入射光方向、视线方向、反射光线方向
   float3 L = normalize(lightPosition - worldPos);
   float3 V = normalize(eyePosition - worldPos);
   float3 R = 2*max(dot(N, L), 0)*N-L;
   R = normalize(R);

   //  计算漫反射分量
   float3 diffuseColor = Kd * globalAmbient+Kd*lightColor*max(dot(N, L), 0);

   //计算镜面反射分量
   float3 specularColor = Ks * lightColor*pow(max(dot(V, R), 0), shininess);

   color.xyz = diffuseColor + specularColor;
   color.w = 1;
}
```

图 19 展示了同时使用顶点着色程序和片段着色程序的 phong 光照模型渲染效果。

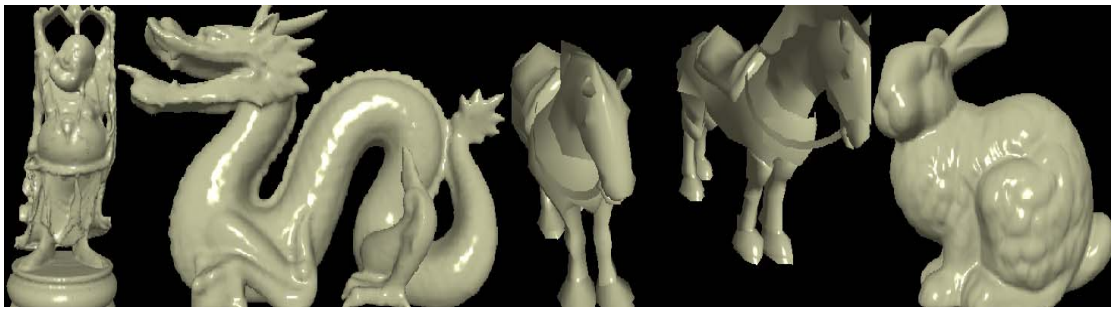


图 19 phong 光照模型的片段着色程序渲染效果

9.4 Blinn-Phong 光照模型

Blinn-Phong 光照模型，又称为 Blinn-phong 反射模型（Blinn–Phong reflection model）或者 phong 修正模型（modified Phong reflection model），是由 Jim Blinn 于 1977 年在文章“Models of light reflection for computer synthesized pictures”中对传统 phong 光照模型基础上进行修改提出的。和传统 phong 光照模型相比，Blinn-phong 光照模型混合了 Lambert 的漫射部分和标准的高光，渲染效果有时比 Phong 高光更柔和、更平滑，此外它在速度上相当快，因此成为许多 CG 软件中的默认光照渲染方法。此外它也集成在了大多数图形芯片中，用以产生实时快速的渲染。在 OpenGL 和 Direct3D 渲染管线中，Blinn-Phong 就是默认的渲染模型。

phong 光照模型中，必须计算$V \bullet R$的值，其中R为反射光线方向单位向量，V为视线方向单位向量，但是在 Blinn-phong 光照模型中，用$N \bullet H$的值取代了$V \bullet R$。Blinn-phong 光照模型公式为：

$$I_{spec} = k_s I_l (N \bullet H)^{n_s} \tag{9-8}$$

其中N是入射点的单位法向量，H是“光入射方向L和视点方向V的中间向量”，通常也称之为半角向量。注意：半角向量被广泛用于各类光照模型，原因不但在于半角向量蕴含的信息价值，也在于计算半角向量是一件简单、耗时不多的工作。

$$H = \frac{L+V}{|L+V|} \qquad (9\text{-}9)$$

通常情况下，使用 Blinn-phong 光照模型渲染的效果和 phong 模型渲染的效果没有太大的区别，有些艺术工作者认为 phong 光照模型比 blinn-phong 更加真实，实际上也是如此。Blinn-phong 渲染效果要更加柔和一些，Blinn-phong 光照模型省去了计算反射光线方向向量的两个乘法运算，速度更快。由于 Blinn-phong 和 phong 模型的唯一区别一个使用半角向量，一个使用反射光方向向量，所以下面只给出 Blinn-phong 模型的片段着色程序代码。

代码 7 Blinn-phong 模型片段着色程序

```
void main_f(VertexScreen posIn,
                    out float4 color          : COLOR,
                       uniform float4x4 worldMatrix,
                       uniform float4x4 worldMatrix_IT,
                       uniform float3 globalAmbient,
                       uniform float3 eyePosition,
                       uniform float3 lightPosition,
                       uniform float3 lightColor,
                       uniform float3 Kd,
                       uniform float3 Ks,
                       uniform float    shininess)
{
    float3 worldPos = mul(worldMatrix, posIn.objectPos).xyz;
    float3 N = mul(worldMatrix_IT, posIn.objectNormal).xyz;
    N = normalize(N);

    //计算入射光方向\视线方向\半角向量
    float3 L = normalize(lightPosition - worldPos);
    float3 V = normalize(eyePosition - worldPos);
    float3 H = normalize(L + V);

    // 计算漫反射分量
   float3 diffuseColor = Kd * globalAmbient+Kd*lightColor*max(dot(N, L), 0);

   //计算镜面反射分量
   float3 specularColor = Ks * lightColor*pow(max(dot(N, H), 0), shininess);

   color.xyz = diffuseColor + specularColor;
   color.w = 1;
}
```

Blinn-phong 光照模型的渲染效果如：

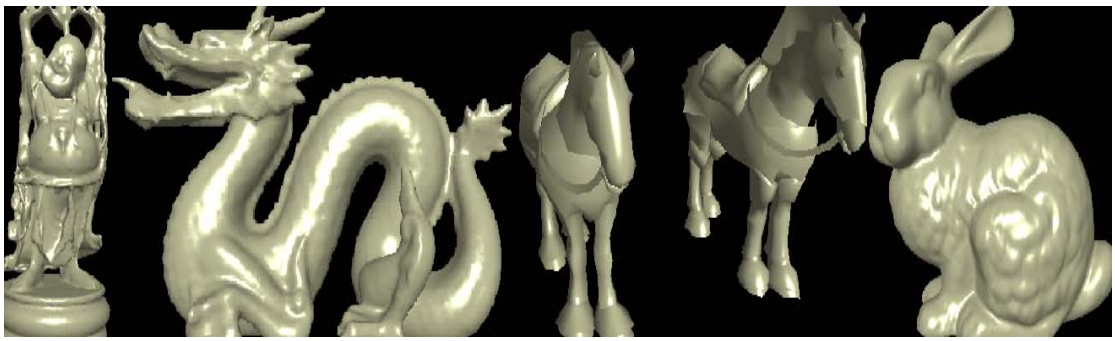


图 20 Blinn-phong 光照模型渲染效果

图形截屏，加上文档大小的限制，好像导致图 19 和图 20 不能形成明显的对比。实际上，我在实现算法时认真对比过，phong 光照模型确实要比 blinn-phong 渲染效果要真实。与 phong 光照模型相比，使用 blinn-phong 进行光照渲染，在同样的高光系数下，高光领域覆盖范围较大，明暗界限不明显。

9.5 全局光照模型与 Rendering Equation

Kajia 在 1986 年提出 rendering equation：

$$L_o(x,w_o)=L_e(x,w_o)+\int_{\Omega} f_r(x,w_i,w_o)L_i(x,w_i)(n\bullet w_i)dw_i \qquad (9\text{-}10)$$

其中 x 表示入射点；$L_o(x,w_o)$ 即从物体表面 x 点，沿方向 w_o 反射的光强；$L_e(x,w_o)$ 表示从物体表面 x 以方向 w_o 发射出去光强，该值仅对自发光体有效；$f_r(x,w_i,w_o)$ 为，入射光线方向为 w_i，照射到点 x 上，然后从 w_o 方向反射出去的 BRDF 值；$L_i(x,w_i)$ 为入射方向为 w_i，照射到点 x 上入射光强；n 表示点 x 处的法向量。然后对入射方向进行积分（因为光线入射的方向是四面八方的，积分的意义是对每个方向进行一遍计算后进行相加），计算的结果就是“从观察方向上看到的辐射率”。

该公式基于物理光学，对观察方向上辐射率的进行了本质上的量化，前面所讲的漫反射光照模型和 phong 高光模型，其实是公式在单一光源，特定 BRDF 下的推导。

对于单个点光源照射到不会自发光的物体上，公式可以简化为：

$$L_o(x,w_o) = f_r(x,w_i,w_o)L_i(x,w_i)(n \bullet w_i) \qquad (9\text{-}11)$$

这个公式非常有用，通常会将该公式分解为漫反射表达式和镜面反射表达式之和。对于漫反射表面，BRDF 可以忽略不计，因为它总是返回某个恒定值（实时计算机图形学第二版 112 页），所以公式可以写成下面的形式：

$$L_o(x,w_o) = I_{diff} + f_{rs}(x,w_i,w_o)L_i(x,w_i)(n \bullet w_i) \qquad (9\text{-}12)$$

其中 I_{diff} 表示漫反射分量，使用公式的计算方法，$f_{rs}(x,w_i,w_o)$ 表示镜面反射的 BRDF 函数。前面所讲的 phong 高光模型，其实是 rendering equation 在单一光源下针对理想镜面反射的特定推导。而对于 Phong 高光而言：

$$f_{rs}(x,w_i,w_o) = \frac{k_s(n \bullet h)^{n_s}}{n \bullet w_i} \qquad (9\text{-}13)$$

在第 10 章中将要被阐述的 BRDF，就是 $f_{rs}(x,w_i,w_o)$ 的一种函数类型。

9.6 本章小结

这一章主要对比较经典的光照模型进行了论述，这些光照模型虽然简单，却是目前所有光照算法和图形硬件光照处理的基础。正是由于前辈们做出的贡献，才有了今天疯狂玩魔兽的庞大人群。虽然目前光线跟踪算法和辐射度算法已经在 GPU 上得到了实现，但本章并没有对光线跟踪算法和辐射度算法做出论述，主要是因为考虑到技术的复杂度，我打算将其放到本书的下一版再进行阐述。而且，我个人每天下班后再饿着肚子写到 7 点，然后吃饭，再回来写，也颇觉辛苦，也就偷偷懒吧。

第 10 章 高级光照模型

在过去的一段时间中，光照模型得到了深入的发展，人们不再满足于只是对漫反射现象和镜面反射现象进行模拟，而是希望可以模拟更多特殊材质上的光照效果，如同向异性、各项异性等。上一章已经提到过，无论是漫反射，还是镜面反射，都属于材质和光交互的理想状态，就好像物理学中的无阻力状态和恒温状态，真实的情况是：漫反射和镜面反射都需要依据材质特征和物体表面微平面特征。图 21是实际漫反射、镜面反射与理想漫反射、镜面反射的示意图。

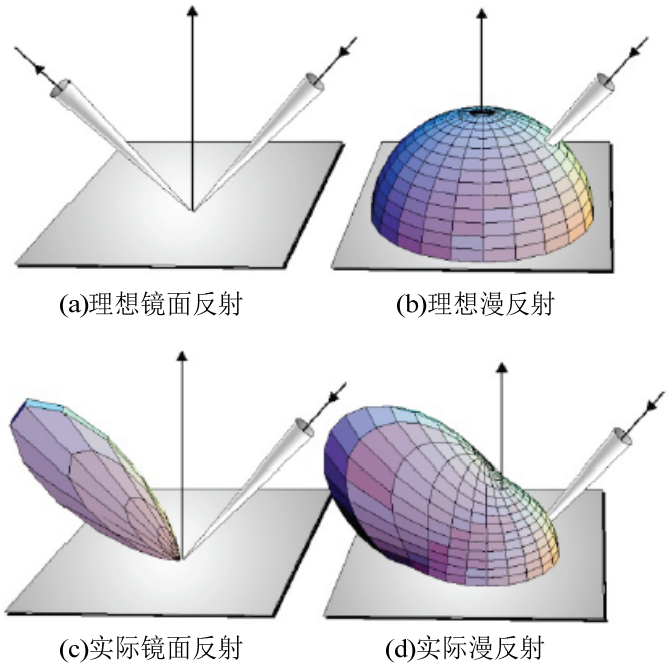


图 21 理想发射和实际反射之对比

材质和光的交互方式是多种多样的，例如，我们可以在光盘上看到如扇面般的光带分布，在光滑的圆柱形炊具上可以看到光的条纹。为了衡量材质的光学特征和物体表面微平面特征，BRDF（Bidirectional Reflectance Distribution Function）模型应运而生。

本章将要讲到的以BRDF模型为代表的一些新的光照模型，它们的共同特点是，扩展了材质和光的交互方式，充分考虑材质微平面对光的影响。

首先讲解Cook-Torrance光照模型，并给出GPU上的实现代码；然后重点阐述BRDF模型，BRDF表示双向反射分布函数（Bidirectional Reflectance Distribution Function），用来描述光线如何在物体表面进行反射。BRDF模型的概念较为复杂，其中亦有许多不同的分枝模型，中文资料中少有对该类型模型给出详细讲解和实现代码的，不过该领域的研究已经是国际上一个热点，确有学习和研究的价值。

10.1 Cook-Torrance 光照模型

使用 phong 和 blinn-phong 光照模型渲染出来的效果都存在一个问题：效果过于艺术化，不太真实。这是因为这两种模型都对材质细节方面没有进行考虑。

1981 年，Robert L. Cook 和 Kenneth E. Torrance 发表了名为“A Reflectance Model For Computer Graphics”的论文，首次提出了 Cook-Torrance 光照模型。

Cook-Torrance 光照模型将物体粗糙表面（rough surface）看作由很多微小平面（微平面）组成，每一个微平面都被看作一个理想的镜面反射体，物体表面的粗糙度由微平面斜率的变化来衡量。一个粗糙表面由一系列斜率变化很大的微平面组成，而在相对平滑的表面上微平面斜率变化较小。

Cook-Torrance 模型将光分为两个方面考虑：漫反射光强和镜面反射光强。如公式（10-1）所示：

$$I_{c-t} = I_{diff} + I_{spec} = I_{diff} + k_s I_l R_s \qquad (10\text{-}1)$$

其中 I_{diff} 是漫反射光强，该部分的计算方法和前面所讲的相同，$k_s I_l R_s$ 是镜面反射光强的计算方法。从公式可以看出：cook-Torrance 模型与 phong、blinn-phong 模型的不同之处在于 R_s 的计算方法。实际上，cook-Torrance、phong 和 blinn-phong 三种光照模型的本质区别都在于“使用不同数学表达式计算 R_s”。R_s 在英文中称之为“specular term”。

关于 Cook-Torrance 模型的 R_s，我找到了两个不同的数学描述。在 Wikipedia

的 specular highlight 网页中提供的计算方法为公式（10-2）：

$$R_s = \frac{F * D * G}{V \bullet N} \qquad (10\text{-}2)$$

在《3D 游戏与计算机图形学中的数学方法》第 117 页和 D3DBook (Lighting) Cook-Torrance 一文中给出的数学表达均为公式（10-3）：

$$R_s = \frac{F * D * G}{(N \bullet V) * (N \bullet L)} \qquad (10\text{-}3)$$

我在“A Reflectance Model For Computer Graphics”一文中对公式的原始出处进行了查询，确定公式（10-3）是正确的数学表达。

F 是 Fresnel 反射系数（Fresnel reflect term），表示反射方向上的光强占原始光强的比率；D 表示微平面分布函数（Beckmann distribution factor），返回的是“给定方向上的微平面的分数值”； G 是几何衰减系数（Geometric attenuation term），衡量微平面自身遮蔽光强的影响。N 、V 、L 分别表示法向量、视线方向（从顶点到视点）和入射光方向（从顶点向外）。

schlick 给出了 Fresnel 反射系数的一个近似（参阅第 11 章），精度在 1%范围内，如公式（10-4）所示：

$$F = f_0 + (1 - f_0)(1 - V \bullet H)^5 \qquad (10\text{-}4)$$

f_0 为入射角度接近 0（入射方向靠近法向量）时的 Fresnel 反射系数，V 是指向视点的向量，H 为半角向量。

微平面分布函数：根据给定的半角向量 H，微平面分布函数返回微平面的分数值。最常使用的微平面分布函是 Backmann 分布函数：

$$D = \frac{1}{m^2 \cos^4 \alpha} e^{-\frac{\tan^2 \alpha}{m^2}} \qquad (10\text{-}5)$$

m 值用于度量表面的粗糙程度，较大的 m 值对应于粗糙平面，较小的 m 值

对应与较光滑的表面；α 是顶点法向量 N 和半角向量 H 的夹角。其中

$$-\frac{\tan^2\alpha}{m^2}=-\frac{\frac{1-\cos^2\alpha}{\cos^2\alpha}}{m^2}=\frac{\cos^2\alpha-1}{m^2*\cos^2\alpha}=\frac{(N\bullet H)^2-1}{m^2*(N\bullet H)^2} \qquad (10\text{-}6)$$

所以 Backmann 微平面分布函数的最终数学表达为公式（10-7）所示：

$$D=\frac{1}{m^2\cos^4\alpha}e^{\frac{(N\bullet H)^2-1}{m^2*(N\bullet H)^2}}=\frac{1}{m^2(N\bullet H)^4}e^{\frac{(N\bullet H)^2-1}{m^2*(N\bullet H)^2}} \qquad (10\text{-}7)$$

微平面上的入射光，在到达一个表面之前或被该表面反射之后，可能会被相邻的微平面阻挡，未被遮挡的光随机发散，最终形成了表面漫反射的一部分。这种阻挡会造成镜面反射的轻微昏暗，可以用几何衰减系数来衡量这种影响。

微平面上反射的光可能出现三种情况：入射光未被遮挡，此时到达观察者的光强为 1；入射光部分被遮挡；反射光部分被遮挡。几何衰减系数被定义为：到达观察者的光的最小强度。所以：

$$G=\min(1,G_1,G_2) \qquad (10\text{-}8)$$

$$G_1=\frac{2(N\bullet H)(N\bullet L)}{V\bullet H} \qquad (10\text{-}9)$$

$$G_2=\frac{2(N\bullet H)(N\bullet V)}{V\bullet H} \qquad (10\text{-}10)$$

综上所述，Cook-Torrance 光照模型的 specular term 的最终数学表达为：

$$I_{c-t}=I_{diff}+I_{spec}=k_dI_l(N\bullet L)+k_sI_lR_s$$

$$=k_dI_l(N\bullet L)+k_sI_l\frac{\left(f_0+(1-f_0)(1-V\bullet H)^5\right)\frac{1}{m^2\cos^4\alpha}e^{\frac{(N\bullet H)^2-1}{m^2*(N\bullet H)^2}}\min(1,\frac{2(N\bullet H)(N\bullet L)}{V\bullet H},\frac{2(N\bullet H)(N\bullet V)}{V\bullet H})}{V\bullet N} \qquad (10\text{-}11)$$

附：《在 3 D游戏与计算机图形学中的数学方法》第 119 页将 beckmann 分布函数，错写为“Backmann”，此外该页中给出的 beckmann 分布函数为公式（10-12）：

$$D=\frac{1}{4\pi m^2(N\bullet H)^4}e^{\frac{(N\bullet H)^2-1}{m^2*(N\bullet H)^2}} \qquad (10\text{-}12)$$

我查阅论文《A Reflectance Model For Computer Graphics》，确认文章中给出的 D 的求法应该为公式（10-7）中的形式。给出图片为证。

The facet slope distribution function D represents the fraction of the facets that are oriented in the direction H. Various facet slope distribution functions have been considered by Blinn [5,6]. One of the formulations he described is the Gaussian model [22]:

$$D = ce^{-(\alpha/m)^2},$$

where c is an arbitrary constant.

In addition to the ones mentioned by Blinn, other facet slope distribution models are possible. In particular, models for the scattering of radar and infrared radiation from surfaces are available, and are applicable to visible wavelengths. For example, Davies [8] described the spatial distribution of electromagnetic radiation reflected from a rough surface made of a perfect electrical conductor. Bennett and Porteus [3] extended these results to real metals, and Torrance and Sparrow [21] showed that they apply to nonmetals as well. Beckmann [2] provided a comprehensive theory that encompasses all of these materials and is applicable to a wide range of surface conditions ranging from smooth to very rough. For rough surfaces, the Beckmann distribution function is

$$D = \frac{1}{m^2\cos^4\alpha}\, e^{-\left\{\frac{\tan^2\alpha}{m^2}\right\}}.$$

不过，我也在国外的一些技术资料上看到类似于公式般的写法（只是把下面的π去掉了），这种写法应该是一种变体形式，本质上不会影响光强，只是对光强做一种缩放而已。Cook-Torrance 提出之前，微平面分布函数使用的是高斯分布（即正态分布）函数。

此外，所谓的 beckmann 分布函数相关资料非常少，通过反复查询，可以确认这是电磁学中的一个分布函数，有兴趣深入研究的，可以参考如下两个资料：

1. Beckmann, Petr and Spizzichino, Andre, The Scattering of Electromagnetic Waves from Rpugh Surfaces, MacMillan, pp.1-33, 70-98, 1963;

2. Computer graphics: principles and practice, James D. Foley - 1995 - Computers - 1175 页

10.1.1 Cook-Torrance 光照模型渲染实现

Cook-Torrance 光照模型的渲染效果为图 22：

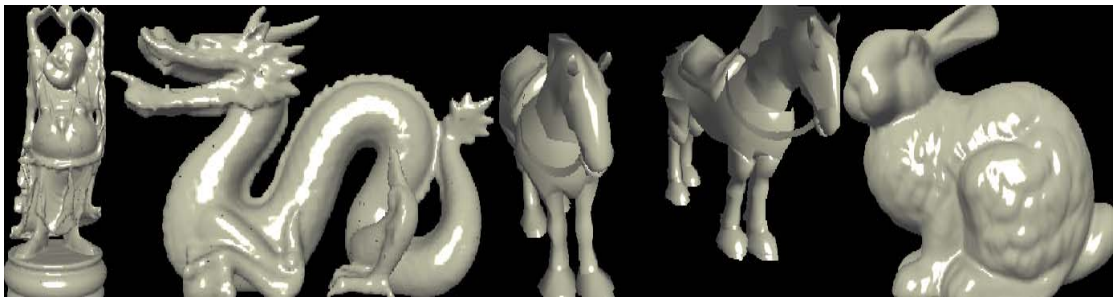


图 22 Cook-Torrance 光照模型的渲染效果

本节给出 Cook-Torrance 光照模型的实现代码，顶点着色程序如下所示：

代码 8 Cook-Torrance 光照模型顶点着色程序

```
void main_v( float4 position     : POSITION,
             float4 normal       : NORMAL,

             out float4 oPosition : POSITION,
             out float3 worldPos : TEXCOORD0,
             out float3 oNormal  : TEXCOORD1,

             uniform float4x4 worldMatrix,
             uniform float4x4 worldMatrix_IT,
             uniform float4x4 worldViewProj)
{
    oPosition = mul(worldViewProj, position);
    worldPos = mul(worldMatrix, position).xyz;
    oNormal = mul(worldMatrix_IT,normal).xyz;
    oNormal = normalize(oNormal);
}
```

片段着色程序为：

代码 9　Cook-Torrance 光照模型片段着色程序

```
void main_f(float3 position    : TEXCOORD0,
            float3 normal      : TEXCOORD1,
            out float4 color   : COLOR,
            uniform float3 globalAmbient,
            uniform float3 lightColor,
            uniform float3 lightPosition,
            uniform float3 eyePosition,
            uniform float3 Ka,
            uniform float3 Kd,
            uniform float3 Ks,
            uniform float  f,
            uniform float  m)
{
  float3 P = position.xyz;
  float3 N = normalize(normal);

  float3 ambient = Ka * globalAmbient; // 计算环境光分量

  float3 L = normalize(lightPosition - P);
  float nl = max(dot(L, N), 0);
  float3 diffuse = Kd * lightColor * nl; // 计算漫反射光分量

  float3 V = normalize(eyePosition - P);
  float3 H = normalize(L + V);
  float3 specular = float3(0.0,0.0,0.0);

  float nv = dot(N,V);
  bool back = (nv>0) && (nl>0);
  if(back)
  {
     float nh = dot(N,H);
     float temp = (nh*nh-1)/(m*m*nh*nh);
    float roughness = (exp(temp))/(pow(m,2)*pow(nh,4.0)); //粗糙度，根据 beckmann 函数

    float vh = dot(V,H);
    float a = (2*nh*nv)/vh;
    float b = (2*nh*nl)/vh;
    float geometric = min(a,b);
    geometric = min(1,geometric); //几何衰减系数

    float fresnelCoe=f+(1-f)*pow(1-vh,5.0); //fresnel 反射系数
    float rs = (fresnelCoe*geometric*roughness)/(nv*nl);
    specular = rs * lightColor * nl*Ks; // 计算镜面反射光分量（这是重点）
  }
  color.xyz =   ambient + diffuse + specular;
  color.w = 1;
}
```

10.2 BRDF 光照模型

10.2.1 什么是 BRDF 光照模型

1965 年，Nicodemus, Fred 在论文“Directional reflectance and emissivity of an opaque surface” 中提出了 BRDF 的概念。BRDF，Bidirectional Reflectance Distribution Function，中文翻译为“双向反射分布函数”。该函数描述了入射光线在非透明物体表面如何进行反射。

BRDF 的结果是一个没有单位的数值，表示在给定入射条件下，某个出射方向上反射光的相对能量，也可以理解为“入射光以特定方向离开的概率”（实时计算机图形学第二版 111 页）。如图 23 所示，w_i 表示光线入射方向，w_o 表示光线出射方向（入射点到视点），则该情况下的 BRDF 值表示：光线以 w_i 方向入射，然后以 w_o 方向出射的概率，或者光强。这些信息也可以用仪器进行测试记录，并存放在图片上，称为 polynomial texture map。

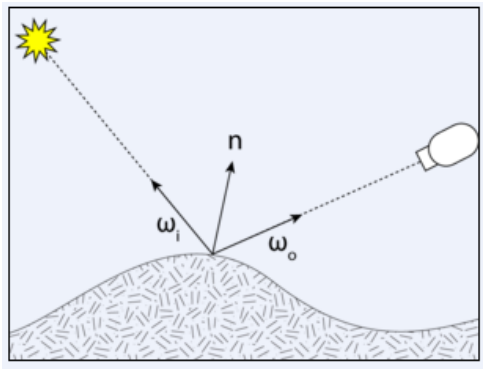


图 23 光的反射

依据光学原理，BRDF 的计算公式为：

$$f_r(w_i, w_o) = \frac{dL_r(w_o)}{dE_i(w_i)} = \frac{dL_r(w_o)}{L_i(w_i)\cos\theta_i dw_i} \qquad (10\text{-}13)$$

其中 $L_r(w_o)$ 表示从 w_o 方向反射的光线的辐射亮度（Radiance）；$E_i(w_i)$ 表示从 w_i 方向入射的光线在辐射照度（Irradiance）。辐射亮度和辐射照度是表示光照性质的光学量，辐射亮度是每单位立体角在垂直于给定方向的平面上的单位正投影面积上的功率。辐射照度则是整个入射表面的功率，等于投射在包括该点的一个面元上的辐射通量 dφ 除以该面元的面积 dA。故而，从物理光学上我们可以将公式理解为：BRDF 函数计算的是“特定反射方向的光强与入射光强的比例”。

所以给定一个具体的 BRDF 数学描述后，就可以放到 rendering equation 中使用（参阅 9.4 节）。

10.2.2 什么是各向异性

各向异性(anisotropy)与均向性相反，是指在不同方向具有不同行为的性质，也就是其行为与方向有关。如在物理学上，沿着材料做不同方向的量测，若会出现不同行为，通常称该材料具有某种“各向异性”，这样的材料表面称为各向异性表面（anisotropic surface）；

特殊的晶体结构会导致各向异性，材质表面上存在有组织的细小凹凸槽也会导致各向异性。各向异性反射是指：各向异性表面反射光的一种现象。在生活中我们经常见到各向异性光照效果，例如光滑的炊具上的扇面光斑（图 24 所示）。

图 24 炊具呈现的各向异性光照效果

由于材质有组织的细微凹凸结构的不同，各向异性也分为基本的三种类型（如图 25 所示）：

1. 线性各向异性；
2. 径向各向异性；
3. 圆柱形各向异性，实际上线性各向异性，单被映像为圆柱形。

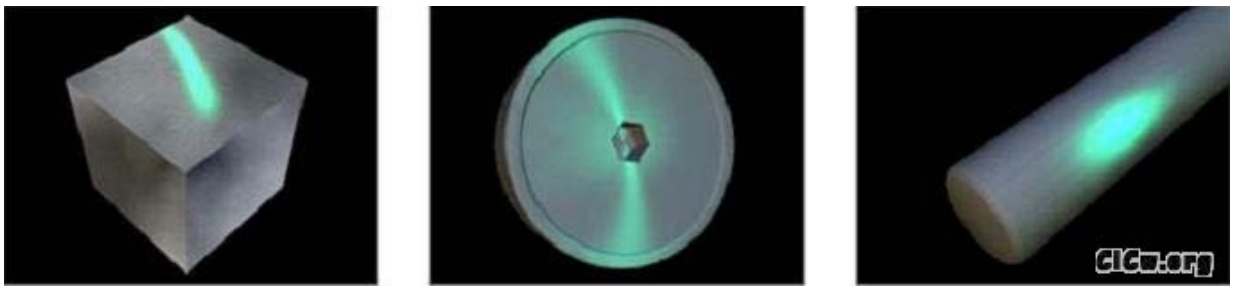


图 25 三种典型的各向异性光照效果

10.3 Bank BRDF 经验模型

Bank BRDF 属于经验模型，由于其计算简单，且效果良好，所以该模型在各向异性光照效果的模拟方面非常有用。Bank BRDF 的镜面反射部分可以表达为公式（10-14）的形式：

$$f = k_s(\sqrt{1-(L \bullet T)^2}\sqrt{1-(V \bullet T)^2} - (L \bullet T)(V \bullet T))^{n_s} \quad (10\text{-}14)$$

k_s、n_s 分别表示镜面反射系数和高光系数；L 表示入射光线方向、V 表示实现观察方向、T 表示该点的切向量。尤其要注意切向量的计算方法，因为一个三维空间点可能存在无数个切向量，通常我采用“顶点的法向量和视线方向做叉积，其结果作为 T。

Bank BRDF 模型渲染效果如图 26、图 27 所示。图 27 的渲染图非常明显的呈现出各向异性的光照效果。

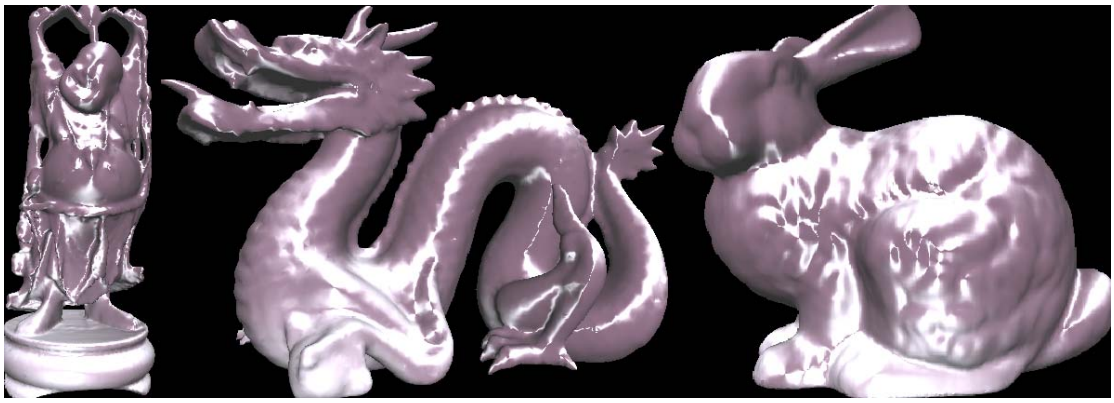


图 26 Bank BRDF 渲染效果 1

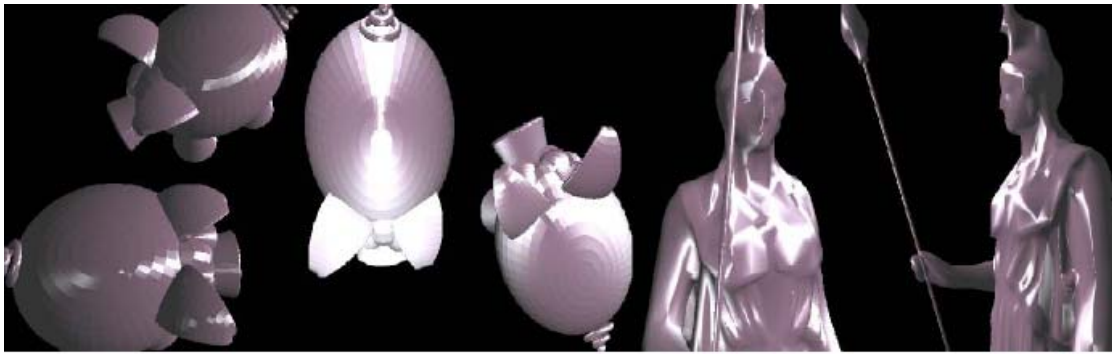


图 27 Bank BRDF 渲染效果 2

下面分别给出 Bank BRDF 的顶点着色程序和片段着色程序代码。

代码 10 Bank BRDF 的顶点着色程序

```
void main_v(float4 position    : POSITION,
            float4 normal      : NORMAL,

            out float4 oPosition : POSITION,
            out float3 worldPos : TEXCOORD0,
            out float3 worldNormal     : TEXCOORD1,

            uniform float4x4 modelViewProj,
            uniform float4x4 worldMatrix,
            uniform float4x4 worldMatrix_IT)
{
    oPosition = mul(modelViewProj, position);
    worldPos = mul(worldMatrix, position).xyz;
    worldNormal = mul(worldMatrix_IT, normal).xyz;
}
```

代码 11 Bank BRDF 片段着色程序

```
void main_f(float4 position    : TEXCOORD0,
            float3 normal      : TEXCOORD1,

            out float4 color       : COLOR,

            uniform float3 globalAmbient,
            uniform float3 lightColor,
            uniform float3 lightPosition,
            uniform float3 eyePosition,
            uniform float3 Ka,
            uniform float3 Kd,
            uniform float3 Ks,
            uniform float   shininess)
{
  float3 P = position.xyz;
  float3 N = normalize(normal);

  float3 ambient = Ka * globalAmbient; //计算环境光分量

  float3 L = normalize(lightPosition - P);
  float ln = max(dot(L, N), 0);
  float3 diffuse = Kd * lightColor *ln; // 计算有向光漫反射分量

  // 计算镜面反射分量
  float3 V = normalize(eyePosition - P);
  float3 H = normalize(L + V);
  float3 specular = float3(0.0,0.0,0.0);
  bool back = (dot(V,N)>0) && (dot(L,N));
  if(back)
  {
    float3 T = normalize(cross(N,V));   //计算顶点切向量
    float a = dot(L,T);
    float b = dot(V,T);
    float c = sqrt(1-pow(a,2.0))* sqrt(1-pow(b,2.0)) - a*b; //计算 Bank BRDF 系数
    float brdf= Ks* pow(c, shininess);

    specular = brdf * lightColor *ln;
  }
  color.xyz = ambient + diffuse + specular;
  color.w = 1;
}
```

10.4 本章小结

BRDF 模型有很多分支模型，如 HTSG BRDF 模型擅长模拟很多物理现象，是现今最完整的 BRDF 模型，但是同时需要昂贵的计算开销；Ward BRDF 用于各向异性表面的经验模型有些复杂，并且需要从实际物体表面来获取 BRDF 数据。这些数据可以通过测角仪、图像双向反射计来得到，国外网站也有一些公开的数据库。

第 11 章 透明光照模型与环境贴图

材质和光的交互除了反射现象，对于透明物体还存在透射现象。模拟光的透射现象通常是一个比较头痛的问题，因为需要至少计算光的两次透射方向，首先计算光从介质一进入介质二的透射方向，然后计算光从介质二进入介质一的透射方象。此外，光在透明物体内穿越的距离以及被穿越的材质，直接关系到光的衰减程度；加上，还有很复杂的透明材质的次表面散射现象，即光线渗透到透明材质中，在内部发生散射，最后射出物体并进入视野中产生的现象。总而言之，不论在CPU上还是在GPU上，想要精确完善的模拟光透现象是一件相当复杂的事情。

我们常听说的光线跟踪算法，虽然可以跟踪模拟出光透现象，但是对于次表面散射的模拟无能为力，并且基于CPU的光线跟踪算法是无法达到实时要求的。基于GPU的光线跟踪算法已经得到实现，不过我并不打算在本书中进行阐述，而是转而讲解较为基础的简单透明光照模型，以及其基于GPU的实现方法。待读者掌握了这个基础实现后，我会在本套书的第二版中再阐述基于GPU的复杂光透模拟的相关算法。

在一个场景中，光滑的物体表面往往会映射出周围环境，想象一下，一款赛车游戏中，当你的车呼啸而过时，车身同时映射出周围的景色，流光飞扬。为了达到这样的效果，通常使用环境贴图技术。

本章中，首先给出与光透现象相关的两个重要光学定律：Snell定律和Fresnel定律；然后阐述如何使用环境贴图方法模拟光滑表面对周围场景的映射效果，并给出GPU实现代码；最后讲解简单透明光照模型，并给出GPU实现代码。这些技术非常有用，在大家所玩的3D游戏中，为了在效率和效果上达到平衡，百分之百是采用这些技术或者这些技术的变体。

11.1 Snell 定律与 Fresnel 定律

11.1.1 折射率与 Snell 定律

光在真空中的速度c与在透明介质中的速度v之比，如公式（11-1）所示，称之为该介质的绝对折射率，简称折射率。光在真空中的折射率等于 1，通常我们认为光在空气中的折射率也近视为 1。

$$n=\frac{c}{v} \qquad (11\text{-}1)$$

折射率较大介质的称为光密介质，折射率较小的介质称为光疏介质。下面给出一些常用介质的折射率，如表所示。

材质	折射率值
真空\空气	1.0\ 1.0003
水	1.333
玻璃	1.5-1.7
钻石	2.417
冰	1.309

表 9 常用介质折射率表

Snell 定律描述光线从一个介质传播到另一个介质时，入射角、折射角和介质折射率的关系。如图 28 所示，假设光线从空气射入水面，入射角度为θ_i，空气对光线的折射率为n_i，折射率角度为θ_t，水对光线的折射率为n_t，则存在：

$$\sin\theta_i * n_i = \sin\theta_t * n_t \qquad (11\text{-}2)$$

通过 snell 定律，我们可以根据入射光的方向向量求取折射光的方向向量

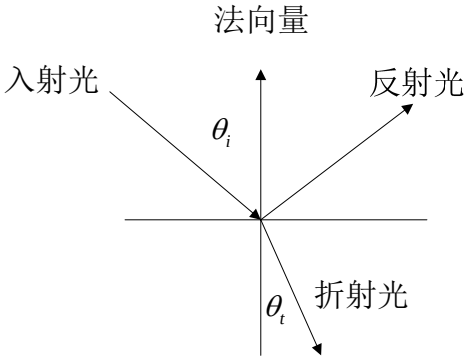


图 28 Snell 定律描述图

需要注意：折射率和下面将要阐述的 fresnel 折射系数并不是同样的东西。折射率本质上反映的是光在介质中的运行速度，以及折射方向；而 fresnel 折射系数反映的是，光在透明介质表面被折射的光通量的比率。

11.1.2 色散

色散分为正常色散和反常色散，通常我们所说的色散都是指反常色散，即，对光波透明的介质，其折射率随着波长的增加而减小。1672 年，牛顿利用三棱镜将太阳光分解为光谱色带（红橙黄绿蓝靛紫），这是人类首次所做的色散试验。天空中的彩虹也是由于光与水蒸气交互形成的色散现象（图 29 所示）。法国数学家柯西在 1936 年首先给出了正常色散的经验公式，称为柯西公式。

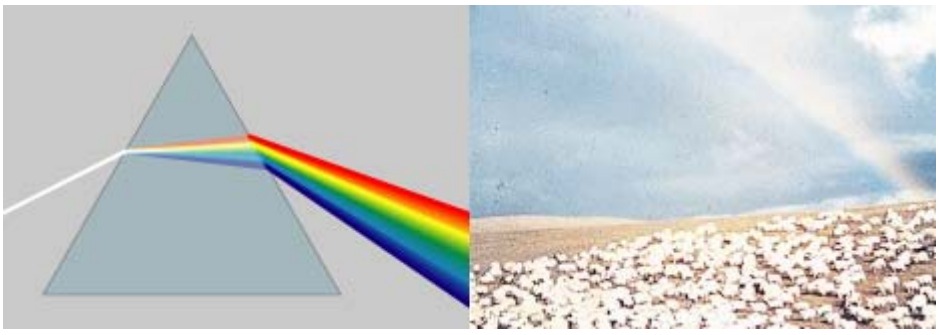


图 29 色散效果

计算机中的颜色是三原色，即只有红、绿、蓝三种颜色，给出光的红、绿、蓝分量的不同折射系数，可以近似模拟色散效果。

11.1.3 Fresnel 定律

光线照射到透明物体上时，一部分发生反射，一部分进入物体内部并在介质交界处发生折射，被反射和折射的光通量存在一定的比率关系，这个比率关系可以通过 Fresnel 定律进行计算。

根据 Fresnel 定律计算得出的数据称为 Fresnel 系数。严格而言，fresnel 系数分为 fresnel 反射系数和 fresnel 折射系数，通常我们所说的 fresnel 系数指“反射系数”。例如，实时计算机图形学第二版 139 页中写到：fresnel 系数，如果 fresnel 为 0.7，那么被反射光线就减少奥 70%，穿过表面的折射光线则是 30%。下文中我们用 k_r 表示 fresnel 反射系数，用 k_t 表示折射系数，如果不做详细说明，则通常所写的 fresnel 系数指反射系数。

一个完整的 fresnel 公式依赖于折射率、消光率和入射角度等因素，该公式的推导本质上是属于物理光学的部分，在《3D 游戏与计算机图形学中的数学方法》一书的 117 页给出了 Fresnel 系数的简易推导方式，有兴趣的可以阅读。

schlick 给出了 Fresnel 反射系数的一个近似，精度在 1%范围内：

$$F = f_0 + (1 - f_0)(1 - V \bullet H)^5 \qquad (11\text{-}3)$$

f_0 为入射角度接近 0（入射方向靠近法向量）时的 Fresnel 反射系数，V 是指向视点的观察方向，H 为半角向量。观察公式（11-3），可以得出一个结论：随着入射角趋近 90，反射系数趋近 1，即擦地入射时，所有入射光都被反射。

在 fresnel equations-Wikepedia 中列举了当入射角度接近 0 时的 fresnel 反射系数的计算方法：

$$f_0 = \frac{(n_i - n_t)^2}{(n_i + n_t)^2} \qquad (11\text{-}4)$$

求出 fresnel 反射系数后，用 1 减去该系数，就得到了折射系数，所以当入射角度接近 0 时的 fresnel 折射系数的计算方法为：

$$f_{t0} = 1 - f_0 = \frac{4n_1 n_2}{(n_i + n_t)^2} \qquad (11\text{-}5)$$

综合公式（11-3）和公式（11-4），fresnel 反射系数的计算公式为：

$$F \approx \frac{(n_i - n_t)^2}{(n_i + n_t)^2} + \frac{4n_1 n_2}{(n_i + n_t)^2} * (1 - V \bullet H)^5 \qquad (11\text{-}6)$$

计算公式（11-6）比较消耗时间，所以通常在程序中使用入射角接近 0 时的 fresnel 系数。下面的公式（11-7）计算精度不高，但是胜在计算速度快，利于硬件实现，所以如果想动态的计算 fresnel 系数，而又不希望消耗太多的时间，可以采用公式（11-7）。

$$F \approx (1 - V \bullet H)^4 \qquad (11\text{-}7)$$

普通玻璃的反射系数大约是 4%。

11.2 环境贴图

环境贴图（Environment Mapping, EM）也称为反射贴图（Reflection Mapping），用于模拟光滑表面对周围场景的映射效果，这项技术由 Blinn 和 Newell 于 1976 在文献【10】中提出。

本节中首先说明什么是环境贴图，然后阐述如何使用环境贴图模拟光滑表面对周围场景的映射效果。

环境贴图，顾名思义，在一副图片上展现周围的环境。环境贴图假设进行反射的光源和物体都位于很远的位置，同时反射体不会反射自身。最常被用到的环境贴图是立方体环境贴图，由 Greece 于 1986 年在文献【11】中提出。该方法是

将相机放置在环境的最中央，然后从上、下、左、右、前、后，6 个方向拍摄周围环境，最后将这些信息投影到立方体的 6 个面上，所得到的纹理称之为立方体环境贴图。

环境贴图的文件表现形式是一种纹理类型，例如立方体环境贴图在 DirectX 中可以保存为.dds 格式，也很多开源图形引擎中使用 6 个方向上的二维纹理动态组成环境贴图。

如下图 29 所示，这是 OGRE 图形引擎中提供的 2 组环境贴图。

图 30 OGRE 图形引擎中提供的 2 组环境贴图

除了立方体环境贴图，还有球面贴图、抛物面贴图。不过这两种贴图类型，与立方体环境贴图相比没有明显的优势。有兴趣的同学，可以参阅《实时计算机图形学第 2 版第 5.7.4 节》。

使用环境贴图，是为了模拟光滑表面对周围场景的映射效果。光滑表面对周围场景的映射，是由从场景出发的光线投射到光滑表面上然后被反射到人眼所形成的视觉效果，如图 31 所示。

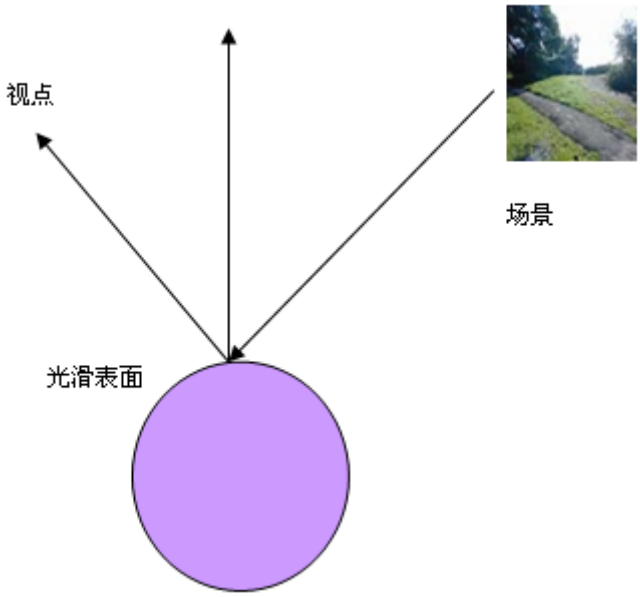


图 31 光滑表面环境映射示意图

我们将这个过程倒退回去，从视点发射一束射线到反射体上的一个点，然后这束射线以这个点为基准进行反射，并根据反射光线的方向向量检索环境图像的颜色。这就是环境贴图算法的基本思想。

环境贴图算法的步骤如下：

1. 首先根据视线方向和法向量计算反射向量；
2. 然后使用反射向量检索环境贴图上的纹理信息；
3. 最后将该纹理信息融合到当像素颜色中。

环境贴图在 Cg 语言中属于 smaplerCUBE 类型变量，标准函数库中的 texCUBE 函数提供依据反射方向向量检索环境贴图的功能。

在代码 12 是使用立方体环境贴图实现环境映射效果的着色程序代码。在顶点着色程序中计算反射光线方向向量，然后传递到片段着色程序中，在片段着色程序中利用该方向检索环境贴图，获取纹理颜色，并进行片段赋值。整个实现过程非常简练。

代码 12 实现环境映射效果的着色代程序

```
void main_v( float4 position: POSITION,
                    float4 normal: NORMAL,

                    out float4 oposition : POSITION,
                    out float3 R : TEXCOORD1,

                    uniform float3 eyePosition,
                    uniform float4x4 modelViewProj,
                    uniform float4x4 modelToWorld)
{
   oposition = mul( modelViewProj, position);

   //计算世界空间中的物体坐标和法向量
   float3 positionW = mul( modelToWorld, position).xyz;
   float3 normalW = mul( modelToWorld_IT, normal).xyz;
   normalW = normalize( normalW);
   float3 I = positionW - eyePosition;

   //计算反射光线 R.
   R = reflect( I, normalW );
}

void main_f( float3 R : TEXCOORD1,
                 out float4 color : COLOR,
                  uniform samplerCUBE environmentMap)
{
   color = texCUBE( environmentMap, R);
}
```

渲染效果如图 32 所示：

图 32 环境映射实现效果

11.3 简单透明光照模型

简单透明光照模型不考虑透明物体对光的第二次折射、次表面散射，以及光在穿越透明物体时的强度衰减，只是简单的使用颜色调和的方法，即我们最终所看到的颜色，是物体表面的颜色和背景颜色的叠加。

如图 33 所示，透明物体位于视点与另一个不透明物体之间，透明物体的不透明度为 t，点 A 为透明物体上的一点，点光源直接照射到 A 点上产生的反射光强为i_a，视线穿过透明体与另一个物体相交处的光强为i_b，则点 A 处的最终可观察的光强为：

$$i = (1-t) * i_b + t * i_a \qquad (11\text{-}8)$$

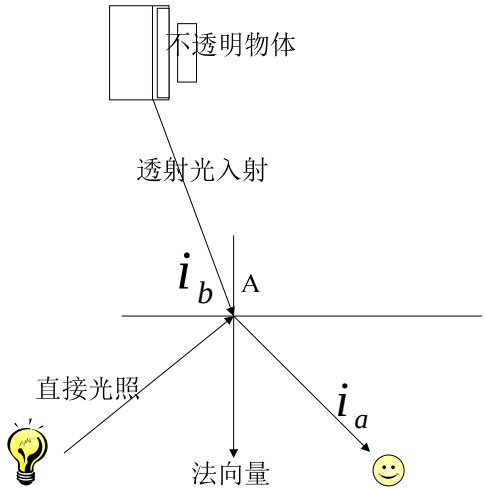


图 33 简单透明光照模型示意图

i_a 和 i_b 都可以用前面所讲的光照模型进行计算。通过透射光方向计算透射光强，首先需要进行光线和空间物体的求交运算，以确定透射光的来源，这是非常消耗时间的，如果真的这样做，其实就是演变为光线跟踪算法了。为了保证实时性，在实际使用中，通常是根据入射光方向向量和法向量求取折射光方向，然后根据折射光方向检索环境纹理上的颜色值作为 i_b。 简单透明光照模型渲染效果如图 34 所示。

图 34 简单透明光照模型渲染效果

简单透明光照模型的着色程序如下所示。依然是首先定义结构体，用来包含输入、输出数据流，然后在顶点着色程序中计算顶点投影坐标，并将顶点模型坐标和法向量坐标传递到片段着色程序中，然后在片段着色程序中计算折射光方向向量，并根据折射光向量检索环境贴图，最后按照公式（11-8）进行颜色合成。Cg 标准函数库中的 lerp 函数可以提供颜色合成功能，具体函数使用方法参见 8.3.1 节。

注意：11.2 节所阐述的环境贴图方法，是使用反射光方向向量检索环境贴图；而在本节中，是使用折射光方向向量检索环境贴图。尽管都是检索环境贴图，但还是有区别的。

代码 13 结构体

```
struct VertexIn
{
   float4 position    : POSITION;
   float4 normal      : NORMAL;
};

struct VertexScreen
{
    float4 oPosition          : POSITION;
    float4 objectPos          : TEXCOORD0;
    float4 objectNormal       : TEXCOORD1;
};
```

顶点着色程序和片段着色程序为：

代码 14 简单透明光照模型顶点着色程序

```
void main_v(VertexIn posIn,
                out VertexScreen posOut,
                uniform float4x4 modelViewProj)
{
   posOut.oPosition = mul(modelViewProj, posIn.position);
   posOut.objectPos = posIn.position;
   posOut.objectNormal = posIn.normal;
}
```

代码 15 简单透明光照模型片段着色程序

```
void main_f(      VertexScreen posIn,
                  out float4 color         : COLOR,

                   uniform float4x4 worldMatrix,
                   uniform float4x4 worldMatrix_IT,
                   uniform float3 globalAmbient,
                   uniform float3 eyePosition,
                   uniform float3 lightPosition,
                   uniform float3 lightColor,
                   uniform float3 Kd,
                   uniform float3 Ks,
                   uniform float    shininess,
                   uniform float etaRatio,                    //折射系数
                   uniform float            transmittance, //透明度
                   uniform samplerCUBE environmentMap //环境贴图
                   )
{
    float3 worldPos = mul(worldMatrix, posIn.objectPos).xyz;
    float3 N = mul(worldMatrix_IT, posIn.objectNormal).xyz;
    N = normalize(N);

   //计算入射光方向\视线方向\半角向量
   float3 L = normalize(lightPosition - worldPos);
   float3 V = normalize(eyePosition - worldPos);
   float3 H = normalize(L + V);

   // 计算漫反射分量、镜面反射分量
   float3 diffuseColor = Kd * globalAmbient+Kd*lightColor*max(dot(N, L), 0);
   float3 specularColor = Ks * lightColor*pow(max(dot(N, H), 0), shininess);
   float3 reflectColor =    diffuseColor+specularColor;

   //计算折射光线的方向,注意 refract 的输入参数!
   float3 I = normalize(worldPos - eyePosition);
   float3 T = refract(I, N, etaRatio);

   //根据折射光线的方向，检索环境贴图上的颜色信息
   float3 refractedColor = texCUBE(environmentMap, T).xyz;

   color.xyz = lerp(reflectColor, refractedColor, transmittance);
   color.w = 1;
}
```

11.4 复杂透明光照模型与次表面散射

光射入透明物体时会发生一次反射和折射，光从透明物体内射出时，又会发生一次反射和折射。透明光照的简单模型实际上只是通过计算了第一次反射和折射，近似的模拟光透效果。2005 年，Wyman 在 ACM SIGGRAPH 大会上提出了在 GPU 中用近似的方法实现两次折射的透明物体绘制算法（Interactive image-space refraction of nearby geometry）。

次表面散射是光射入半透明物体后再内部发生散射，最后射出物体并进入视野中产生的现象。次表面散射材质是高质量渲染中最复杂的材质之一，一个重要原因在于此表面散射物体内部的任何一点的光照度取决于体内其他点的光照度和材质本身的透光率。抛开材质本身的性质不说，这一特性使得次表面散射的光照方程变成一个复杂的微分方程，求出此方程的准确解是十分困难的，另一方面，材质本身可能具有复杂的各向异性和不均匀密度等性质，因此计算这样的积分变得非常困难。GPU 编程精粹第一部的第 16 章给出了一种次表面散射的实时近似模拟算法。

第 12 章 投影纹理映射（Projective Texture Mapping）

投影纹理映射（Projective Texture Mapping）最初由 Segal 在文章“Fast shadows and lighting effects using texture maaping”中提出，用于映射一个纹理到物体上，就像将幻灯片投影到墙上一样。该方法不需要在应用程序中指定顶点纹理坐标，实际上，投影纹理映射中使用的纹理坐标是在顶点着色程序中通过视点矩阵和投影矩阵计算得到的，通常也被称作投影纹理坐标(coordinates in projective space)。而我们常用的纹理坐标是在建模软件中通过手工调整纹理和 3D 模型的对应关系而产生的。

投影纹理映射的目的是将纹理和三维空间顶点进行对应，这种对应的方法好比“将纹理当作一张幻灯片，投影到墙上一样”。如图 35 投影纹理映射所示。

本章我们针对投影纹理映射的原理和实现方法进行详细的阐述。这一章的地位很高，在一些阴影算法以及体绘制算法中都需要用到投影纹理映射技术。严格的说，只要涉及到“纹理实时和空间顶点对应”，通常都会用到投影纹理映射技术。

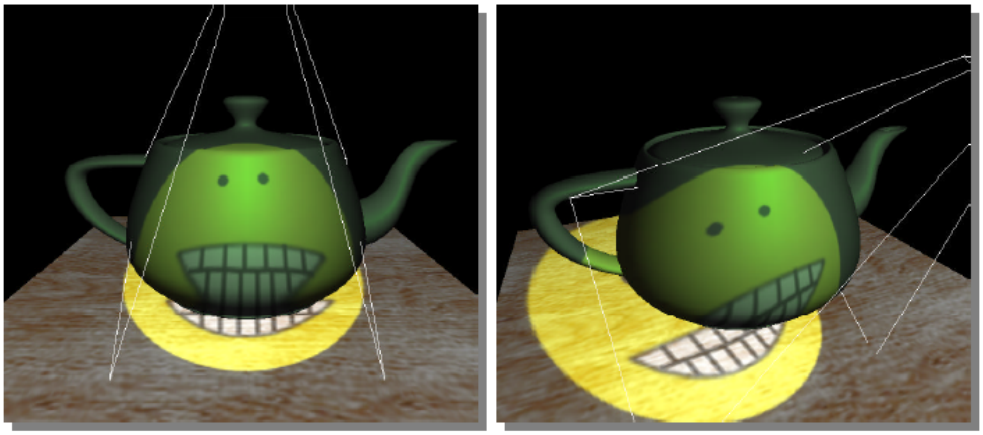


图 35 投影纹理映射(引用自文献【12】)

12.1 投影纹理映射的优点

投影纹理映射有两大优点：其一，将纹理与空间顶点进行实时对应，不需要

预先在建模软件中生成纹理坐标；其二，使用投影纹理映射，可以有效的避免纹理扭曲现象。

为了说明第一个优点，先举一个简要的例子：很多情况下，我们需要将场景渲染两遍，第一遍是为了获取场景信息，得到的场景信息通常保存为一张纹理（例如深度图）；然后基于“存放场景信息”的纹理进行第二次渲染；第二次渲染结果才是最终显示到屏幕上的效果。为了在第二次渲染中使用到“存放场景信息”的纹理（无预先设置的纹理坐标），需要时时进行纹理计算，这时就可以使用投影纹理映射技术。实际上，这也是投影纹理映射技术的最广泛的应用了。

可能大家对于上一段文字还不能理解得很清楚，不过在第 13 章的阴影贴图算法以及第 15 章的体绘制光线投射算法中，大家会明白其含义。一个算法只有理论加实践，才可能真正的被理解，只会照本宣科的朗诵术语，基本上都是鲁迅先生所说的“泥塘”似的人。

投影纹理映射的第二个优点是：可以有效的避免纹理扭曲现象。如图 36 所示，将一张纹理投影到两个三角面片上，它们的顶点纹理坐标相同，但是由于三角面片形状不同，插值出来的内部点的纹理坐标也会产生不同的梯度（gradient），最后纹理颜色在两个三角面片上的分布也是不一样的。

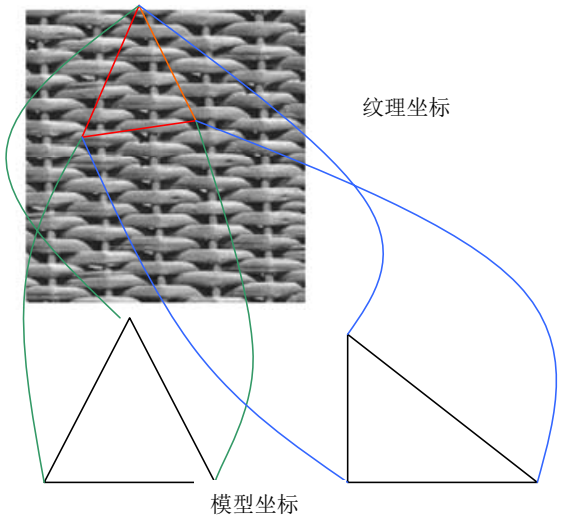
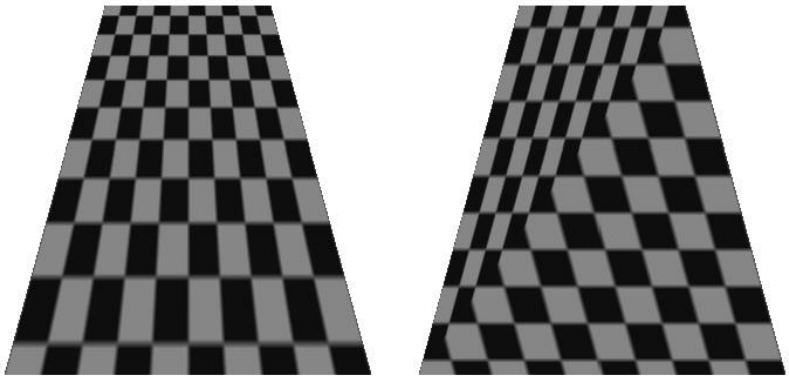


图 36 纹理与几何顶点的对应

图 37 右边所示的是将一张纹理贴到一个正方形上，左边所示的是将同样的纹理贴到一个梯形上，正方形和梯形的顶点纹理坐标相同，但两者的贴图效果是不同的。梯形上的纹理会出现明显的扭曲现象。这是因为几何体的变换，导致插值出来的内部纹理坐标分布不均衡。



(a)投影纹理插值效果　　(b)实际纹理插值效果

图 37 投影纹理映射与普通纹理映射效果对比

12.2 齐次纹理坐标（Homogeneous Texture Coordinates）

齐次纹理坐标（homogeneous texture coordinates）的概念对大多数人来说比较陌生，纹理坐标一般是二维的，如果是体纹理，其纹理坐标也只是三维的。齐次纹理坐标的出现是为了和三维顶点的齐次坐标相对应，因为本质上，投影纹理坐标是通过三维顶点的齐次坐标计算得到的。

齐次纹理坐标通常表示为（s,t,r,q），以区别于物体位置齐次坐标(x, y, z, w)。一维纹理常用 s 坐标表示，二维纹理常用(s, t)坐标表示，目前忽略 r 坐标，q 坐标的作用与齐次坐标点中的 w 坐标非常类似。值一般为 1。

12.3 原理与实现流程

对投影纹理映射，很多教程上都是这么解释的：纹理好比一张幻灯片，灯光好比投影机，然后将纹理投影到一个物体上，类似于投影机将幻灯片投影到墙上。这个比喻没有太大的问题，也找不到更加形象的比喻了。问题是：这个解释刚好

颠倒了算法的实现流程。

投影纹理映射真正的流程是“根据投影机（视点相机）的位置、投影角度，物体的坐标，求出每个顶点所对应的纹理坐标，然后依据纹理坐标去查询纹理值”，也就是说，不是将纹理投影到墙上，而是把墙投影到纹理上。投影纹理坐标的求得，也与纹理本身没有关系，而是由投影机的位置、角度，以及3D模型的顶点坐标所决定。所以，我一直觉得“投影纹理映射”这个术语具有很强的误导性，总让人觉得是把纹理投射出去。

根据顶点坐标获得纹理坐标的本质是将顶点坐标投影到 NDC 平面上，此时投影点的平面坐标即为纹理坐标。如果你将当前视点作为投影机，那么在顶点着色程序中通过 POSTION 语义词输出的顶点投影坐标，就是当前视点下的投影纹理坐标没有被归一化的表达形式。

“Projective texture mapping”文章中有一幅比较著名的图片，说明计算纹理投影坐标的过程，如图 38所示。

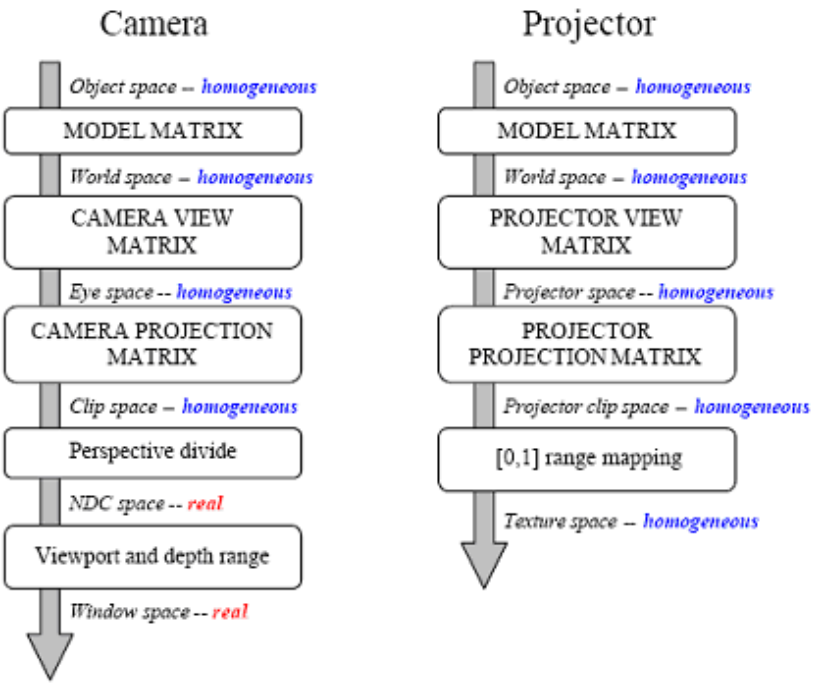


图 38 顶点投影过程与纹理投影坐标计算过程之对比

图 38左边是正常的顶点坐标空间转换流程，无非是顶点从模型坐标空间转

换到世界坐标空间，然后从世界坐标空间转换到视点空间，再从视点空间转换到裁剪空间，然后投影到视锥近平面，经过这些步骤，一个顶点就确定了在屏幕上的位置。图的右边是将视点当作投影机，根据模型空间的顶点坐标，求得投影纹理坐标的流程。通过比较，可以发现这两个流程基本一样，唯一的区别在于求取顶点投影坐标后的归一化不一样：计算投影纹理坐标需要将投影顶点坐标归一化到【0，1】空间中，实现这一步，可以在需要左乘矩阵 *normalMatrix*，也可以在着色程序中对顶点投影坐标的每个分量先乘以1/2然后再加上1/2。

$$normalMatrix = \begin{bmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (12\text{-}1)$$

所以求取投影坐标矩阵的公式为：

$$texView\Pr ojMatrix = biasMatrix \times projectionMatrix \times viewMatrix \times worldMatrix \quad (12\text{-}2)$$

求得纹理投影矩阵后，便可以使用该矩阵将顶点坐标转换为纹理投影坐标。

$$texView\Pr ojCoordinate = texViw\Pr ojMatrix \times \mathrm{mod}\, elCoordinate \qquad (12\text{-}3)$$

使用投影纹理坐标之前，别忘了将投影纹理坐标除以最后一个分量q。到此，你就可以使用所求得的投影纹理坐标的前两个分量去检索纹理图片，从中提取颜色值。还记得Cg标准函数库中有的纹理映射函数的表达形式为：

tex2DProj(sampler2D tex, float4 szq)

tex2DProj函数与tex2D函数的区别就在于：前者会对齐次纹理坐标除以最后一个分量q，然后再进行纹理检索！

注意：上面常被提到的“投影机”只是一种形象化的比喻，本质是视点相机，很多教程上都说“将灯光当作投影机”，这是一种错误的表达（并非是这些教程的作者不懂，而是语言组织上的错误），他们真正的意思是“在当前灯光所在的位置放置一个相机，相机的观察方向和光线投射方向一致”，这个相机就作为投

影机使用。在一些阴影算法中，根据光源信息设置投影机，并从投影机的角度渲染出场景信息纹理（如，阴影纹理），然后把这个纹理放到正常的场景渲染相机中使用，这时就需要投影机的矩阵信息来建立投影纹理矩阵了。

附：投影纹理矩阵的计算通常不需要开发人员自己动手，常用的图形API中都给出了获取各种矩阵（视点矩阵、投影矩阵等）的函数，不过偏移矩阵需要自己设置。在应用程序中获取这些矩阵信息后，再传递到着色程序中使用。

顶点着色程序和片段着色程序如下所示：

代码 16 投影纹理映射顶点着色程序

```
void main_v(
    float4 position             : POSITION,
    float4 normal               : NORMAL,

    out float4 outPos           : POSITION,
    out float4 outShadowUV      : TEXCOORD0,

    uniform float4x4 worldMatrix,
    uniform float4x4 worldViewProj,
    uniform float4x4 texViewProj //投影纹理矩阵
    )
{
    outPos = mul(worldViewProj, position);

    // 计算投影纹理坐标
    float4 worldPos = mul(worldMatrix, position);
    outShadowUV = mul(texViewProj, worldPos);
}
```

代码 17 投影纹理映射片段着色程序

```
void main_f(
                float4 shadowUV                 : TEXCOORD0,
                out float4 result               : COLOR,
                uniform sampler2D projectiveMap    //用于投影的纹理
               )
{
    shadowUV = shadowUV / shadowUV.w;

    float4 mapColor ;

    //归一化到 0-1 空间
    shadowUV.x = (shadowUV.x +float(1.0))/float(2.0);
    shadowUV.y = (shadowUV.y +float(1.0))/float(2.0);
    mapColor = tex2D(projectiveMap, shadowUV.xy);

    result = mapColor;
}
```

12.4 本章小结

本章对投影纹理映射的基本原理和实现流程进行了阐述，这项技术虽然并不复杂，但是却很基础。在下一章的 Shadow Map 以及第 16 章的光线投射算法中都需要使用到。

第 13 章 Shadow Map

Shadows are created by testing whether a pixel is visible from the light source, by comparing it to a z-buffer or depth image of the light source's view, stored in the form of a texture.

Shadow Map 是一种基于深度图（depth map）的阴影生成方法，由 Lance Williams 于 1978 年在文章“Casting curved shadows on curved surfaces”中首次提出。该方法的主要思想是：在第一遍渲染场景时，将场景的深度信息存放在纹理图片上，这个纹理图片称为深度图；然后在第二次渲染场景时，将深度图中的信息 $lenth_1$ 取出，和当前顶点与光源的距离 $lenth_2$ 做比较，如果 $lenth_1$ 小于 $lenth_2$，则说明当前顶点被遮挡处于阴影区，然后在片段着色程序中，将该顶点设置为阴影颜色。

13.1 什么是 depth map

深度图是一张 2D 图片，每个像素都记录了从光源到遮挡物（遮挡物就是阴影生成物体）的距离，并且这些像素对应的顶点对于光源而言是“可见的”。这里的“可见”像素是指，以光源为观察点，光的方向为观察方向，设置观察矩阵并渲染所有遮挡物，最终出现在渲染表面上的像素。

Depth map 中像素点记录的深度值记为 $lenth_1$；然后从视点的出发，计算物体顶点 v 到光源的距离，记为 $lenth_2$；比较 $lenth_1$ 与 $lenth_2$ 的大小，如果 $lenth_2 > lenth_1$，则说明顶点 v 所对应的 depth texure 上的像素点记录的深度值，并不是 v 到光源的距离，而是 v 和光源中间某个点到光源的距离，这意味着“v 被遮挡”。

在一些教程中，往往将 depth map 翻译成阴影贴图（shdaow texture），这实在是一个误解，不光误解了两个名称，也混淆了 2 种阴影算法。阴影贴图的英文

为 Shadow texture，就是将日常所见的阴影保存为纹理图片； Depth texture 保存的是“从视点到物体顶点的距离，通常称为深度值”。图 39 左边的子图来自 wikipid 上 shadow map 网页，请注意，下面表述为 depth map；右边的子图则是一张普通的 shadow texture。

图 39depth map 与 shadow texture 之对比

此外， Shadow texture 不但表示阴影贴图，也代表了一种阴影渲染方法，其实就是将阴影贴图作为纹理投影到物体上，投影的方法采用前面所讲述的 texture projective 方法。

13.2 Shadow map 与 shadow texture 的区别

在很多中文资料中，论述 Shadow map 技术时，容易将 Shadow map 与 shadow texture 这两个不同的概念混淆；

在英文中 map 有映射和图片的双重含义在内，shadow map 技术称为“shadow map”在英文中应该是准确的。中文翻译 shadow map 为阴影图，例如“实时计算机图形学第二版 153 页，第 6.12.4 节便将 shadow map 翻译为阴影图”，这种翻译已经是既成事实，那么我们也延续这种翻译方式。但是一定要知道“阴影图”和 shadow texture 所谓的阴影贴图是完全不同的两个概念。Shadow map 以 depth map 为技术基础，通过比较“光源可见点到光源的深度”和“任何点到光源的深

度”来判断点是否被物体遮挡；而 shadow texture 技术，将生成的阴影图形作为投影纹理来处理，也就是将一张阴影图投影映射到一个物体上（阴影接收体）。这种方法的缺点在于：设计者必须确认哪个物体是遮挡物，哪个物体是阴影接受体，并且不能产生自阴影现象（将一个物体的阴影贴图贴到物体身上，这是多么怪异）。

13.3 Shadow map 原理与实现流程

使用 Shadow Map 技术渲染阴影主要分两个过程：生成 depth map(深度图)和使用 depth map 进行阴影渲染。

生成 depth map 的流程为：

1. 以光源所在位置为相机位置，光线发射方向为观察方向进行相机参数设置；

2. 将世界视点投影矩阵 worldViewProjMatrix 传入顶点着色程序中，并在其中计算每个点的投影坐标，投影坐标的 Z 值即为深度值（将 Z 值保存为深度值只是很多方法中的一种）。在片段 shadow 程序中将深度值进行归一化，即转化到【0，1】区间。然后将深度值赋给颜色值（Cg 最的颜色值范围在 0-1 之间）。

这里有一点要留心：depth map 中保存的深度值到底是什么？很多文献都将 depth map 深度值解释成 Z Buffer 中的 Z 值，我对这种解释一直持怀疑态度！并不是说这种解释不对，而是指“这种解释有以偏概全的嫌疑”。我们通常所说的距离是指笛卡尔坐标空间中的欧几里得距离（Euclidean distance），Z 值本身并不是这个距离（参阅第 2.4.2 节），此外我在研究 GPU 算法的过程中，看到的关于 depth map 中保存的深度值的计算方法远不止一种，有些直接计算顶点到视点的距离，然后归一化到【0，1】空间，同样可以有效的用于深度比较。由此可见，depth map 中保存的深度值，是衡量“顶点到视点的距离”相对关系的数据，计算深度值的重点在于“保证距离间相对关系的正确性”，至于采用什么样的计算方法

倒在其次。

3. 从 frame buffer 中读取颜色值，并渲染到一张纹理上，就得到了 depth map。注意：在实际运用中，如果遇到动态光影，则 depth map 通常是实时计算的，这就需要场景渲染两次，第一次渲染出 depth map，然后基于 depth map 做阴影渲染。渲染 depth map 的顶点着色程序和片段着色程序分别为：

代码 18 渲染 depth map 的顶点着色程序

```
void main_v(float4 position              : POSITION,
            out float4 oPosition         : POSITION,
            out float2 depth             : TEXCOORD0,
             uniform float4x4 worldViewProj )
{
    oPosition = mul(worldViewProj, position);

    // 存放深度值
    depth.x = oPosition.z;
    depth.y = oPosition.w;
}
```

代码 19 渲染 depth map 的片段着色程序

```
void main_f(float2 depth             : TEXCOORD0,
            out float4 result        : COLOR,
             uniform float pNear ,
             uniform float pFar,
             uniform float depthOffset )
{
    float depthNum = 0.0;

   //归一化到 0-1 空间
    depthNum = (depth.x - pNear) / (pFar - pNear);

    depthNum += depthOffset;

    result.xyz = depthNum.xxx;
    result.w = 1.0;
}
```

在代码 19 的片段着色程序中，有一个外部输入变量 depthOffset，该变量表示深度值的偏移量，这时因为：将深度值写入纹理颜色，会导致数据精度的损失，所以需要加上一个深度偏移量。这个偏移量自己设定，通常是 0.01 之类的微小数据。

使用 depth map 进行阴影渲染的流程为：

1. 将纹理投影矩阵传入顶点着色程序中。注意，这个纹理投影矩阵，实际上就是产生深度图时所使用的 worldViewProjMatrix 矩阵乘上偏移矩阵（具体参见第 13 章），根据纹理投影矩阵，和模型空间的顶点坐标，计算投影纹理坐标和当前顶点距离光源的深度值 $lenth_2$（深度值的计算方法要和渲染深度图时的方法保持一致）。

2. 将 depth map 传入片段着色程序中，并根据计算好的投影纹理坐标，从中获取颜色信息，该颜色信息就是深度图中保存的深度值 $lenth_1$。

3. 比较两个深度值的大小，若 $lenth_2$ 大于 $lenth_1$，则当前片断在阴影中；否则当前片断受光照射。顶点着色程序和片段着色程序如下所示：

代码 20 使用 depth map 进行阴影渲染的顶点着色程序

```
void main_v(float4 position          : POSITION,
            float4 normal            : NORMAL,
            float2 tex               : TEXCOORD,

            out float4 outPos        : POSITION,
            out float4 outShadowUV   : TEXCOORD0,

            uniform float4x4 worldMatrix,
            uniform float4x4 worldViewProj,
            uniform float4x4 texViewProj)
{

    outPos = mul(worldViewProj, position);
    float4 worldPos = mul(worldMatrix, position);

    // 计算投影纹理坐标
    outShadowUV = mul(texViewProj, worldPos);
}
```

代码 21 使用 depth map 进行阴影渲染的片段着色程序

```
void main_f(float4 position             : POSITION,
            float4 shadowUV             : TEXCOORD0,
            out float4 result         : COLOR

            uniform sampler2D shadowMap ,
            uniform float pNear ,
            uniform float pFar,
            uniform float depthOffset,
            uniform int pixelOffset)
{
    //计算当前顶点和光源之间的距离（相对）
    float lightDistance = (shadowUV.z - pNear) / (pFar - pNear);
    lightDistance = lightDistance - depthOffset;

    shadowUV.xy = shadowUV.xy/ shadowUV.w;

    //进行多重采样，减小误差
    float4 depths = float4(
            tex2D(shadowMap, shadowUV.xy + float2(-pixelOffset, 0)).x,
            tex2D(shadowMap, shadowUV.xy + float2(pixelOffset, 0)).x,
            tex2D(shadowMap, shadowUV.xy + float2(0, -pixelOffset)).x,
            tex2D(shadowMap, shadowUV.xy + float2(0, pixelOffset)).x);
    float centerdepth = tex2D(shadowMap, shadowUV.xy).x;

    //进行深度比较
    float l_Lit = (lightDistance >= centerdepth? 0 : 1);
    l_Lit += (lightDistance >= depths.x? 0 : 1);
    l_Lit += (lightDistance >= depths.y? 0 : 1);
    l_Lit += (lightDistance >= depths.z? 0 : 1);
    l_Lit += (lightDistance >= depths.w? 0 : 1);
    l_Lit *= 0.2f;

    result = float4(l_Lit, l_Lit, l_Lit, 1.0);
}
```

图 40 展示了使用 shadow map 方法得到的阴影渲染效果。

图 40　shadow map 渲染效果图

Shadow map 方法的优点是可以使用一般用途的图形硬件对任意的阴影进行绘制，而且创建阴影图的代价与需要绘制的图元数量成线性关系，访问阴影图的时间也固定不变。此外，可以在基于该方法进行改进，创建软阴影效果。所谓软阴影就是光学中的半影区域。如果实时渲染软阴影，并运用到游戏中，是目前光照渲染领域的一个热门研究方向。

但 Shadow map 方法同样存在许多不足之处：

其一：阴影质量与阴影图的分辨率有关，所以很容易出现阴影边缘锯齿现象；

其二：深度值比较的精确度和正确性，有赖于 depth map 中像素点的数据精度，当生成深度图时肯定会造成数据精度的损失。要知道，深度值最后都被归一化到 0，1 空间中，所以看起来很小的精度损失也会影响数据比较的正确性，尤其是当两个点相聚非常近时，会出现 z-fighting 现象。所以往往在深度值上加上一个偏移量，人为的弥补这个误差；

其三：自阴影走样（Self-shadow Aliasing）,光源采样和屏幕采样通常并不一定在完全相同的位置，当深度图保存的深度值与观察表面的深度做比较时，其数值可能会出现误差，而导致错误的效果，通常引入偏移因子来避免这种情况；

其四：这种方法只适合于灯类型是聚光灯（Spot light ）的场合。如果灯类型是点光源（Point light）的话，则在第一步中需要生成的不是一张深度纹理，是一个立方深度纹理（cube texture）。如果灯类型是方向光（Directional light）的话,，则产生深度图时需要使用平行投影坐标系下的 worldViewProjMatrix 矩阵；

当前广泛使用的阴影算法中有一种被称之为模板（stencil）阴影算法。模板阴影算法在游戏中得到广泛的使用，在当前主流的开源图形引擎中，基本都集成了该算法。为了对比 shadow map 方法，特地在本书的附录 C 中对其进行阐述。

第 14 章 体绘制（Volume Rendering）概述

1982 年 2 月，美国国家科学基金会在华盛顿召开了科学可视化技术的首次会议，会议认为“科学家不仅需要分析由计算机得出的计算数据，而且需要了解在计算过程中的数据变换，而这些都需要借助于计算机图形学以及图像处理技术”。

----《三维数据场可视化》1.1 节科学计算可视化概述

自 20 世纪 80 年代科学计算可视化（Visualization in Scientific Computing）被提出后，三维体数据的可视化逐渐称为发展的重点，并最终形成了体绘制技术领域。

一些文章，甚至是优秀硕博士论文库上的文章，解释体绘制概念时，通常都说“体绘制技术是直接根据三维体数据场信息产生屏幕上的二维图像”，这种说法太过含糊，如果根据三维体数据场信息随便产生一张图像，难道也是体绘制吗？我查找相关文献后，发现这种说法是国外一些文献的误译，例如，M.Levoy 在文章“Display of surfaces from volume data”(文献【14】)中提到“volume rendering describes a wide range of techniques for generating images from three-dimensional scalar data” ，翻译过来就是“体绘制描述了一系列的“根据三维标量数据产生二维图片”的技术”。注意，人家文章中用的是“描述”，而不是对体绘制下定义。老实说，老外的这种说法虽然挑不出毛病，但是我依然感觉没有落实到重点。

体绘制的核心在于“展示体细节！而不是表面细节”。我给出的定义是：依据三维体数据，将所有体细节同时展现在二维图片上的技术，称之为体绘制技术。利用体绘制技术，可以在一幅图像中显示多种物质的综合分布情况，并且可以通过不透明度的控制，反应等值面的情况。

例如，CT 图片中展示的是人体的肌肉和骨骼信息，而不是表面信息（那是照片）。所以理解体绘制和面绘制技术的区别的,一个很直观的比喻是：普通照相机照出的相片和 CT 仪器拍出的 CT 照片，虽然都是二维图片，但是展现的对象

是不同的！

国外自上世纪 80 年代末以来，在体绘制技术方面已经取得了长足的进步，西门子、东芝、通用电器，都有对 GPU 编程领域以及体绘制技术进行研究，并将体绘制技术运用到医疗器材中。然而，体绘制技术在中国的发展，如果说还处于萌芽阶段，实不为过！我在学习和研究过程中，在国内网站上甚至只找到了一个可用的体数据，还是国外代码中附带的演示数据，而国内的 openGPU 网站上关于体绘制的论坛板块则是根本是空白。国外已经常用的医疗器材和算法，在中国还没有成形，这实在是一种悲哀。一个讽刺的现象是，外国公司从事体绘制算法研究的却不乏中国人，这更是一种悲哀。写到这里，作为一名以 server the people 为毕生理想的有志青年，我有点伤感，有些沮丧，所以，还是先洗洗睡了，明天再来写下面的章节。

14.1 体绘制与科学可视化

科学可视化技术是运用计算机图形学、图像处理、计算机视觉等方法，将科学、工程学、医学等计算、测量过程中的符号、数字信息转换为直观的图形图像，并在屏幕上显示的理论、技术和方法。

体绘制是科学可视化领域中的一个技术方向。如前所述，体绘制的目标是在一副图片上展示空间体细节。举例而言，你面前有一间房子，房子中有家具、家电，站在房子外面只能看到外部形状，无法观察到房子的布局或者房子中的物体；假设房子和房子中的物体都是半透明的，这样你就可以同时查看到所有的细节。这就是体绘制所要达到的效果。

14.2 体绘制应用领域

人类发展史上的重大技术带来的影响大致分为两种：其一，技术首先改变生活本身，然后改变人类对世界的看法，例如电视、电话等；还有一种技术，是首先改变人类对世界的看法，然后改变生活本身，例如伦琴射线、望远镜。

体绘制技术应该属于后者，通过改变所见，而改变生活。体绘制计算的重要意义，首先在于可以在医疗领域 server the people,有助于疾病的诊断，这一点应该不用多说，计算机断层扫描（CT）已经广泛应用于疾病的诊断。医疗领域的巨大需求推动了体绘制技术的告诉发展，如果了解 CT 的工作原理，也就大致了解了体绘制技术原理和流程，所以本书在附录 B 给出了医学体绘制的有关文献，作为补充阅读资料，当您对体素、光线投射等术语缺乏感性认识时，可以参阅理解；其二，体绘制计算可以用于地质勘探、气象分析、分子模型构造等科学领域。我在工作期间承担的一个较大的项目便是有关“三维气象可视化”，气象数据通常非常庞大，完全可以号称海量数据，每一个气压面上都有温度、湿度、风力风向等格点数据，气象研究人员希望可以同时观察到很多气压面的情况，这时就可以采用体绘制技术，对每个切面（气压面）进行同时显示。

体绘制技术也能用于强化视觉效果，自然界中很多视觉效果是不规则的体，如流体、云、烟等，它们很难用常规的几何元素进行建模，使用粒子系统的模拟方法也不能尽善尽美，而使用体绘制可以达到较好的模拟效果。如图 41 所示，这是使用体绘制技术进行烟的模拟效果。

图 41 体绘制技术渲染的烟

14.3 体绘制与光照模型

尽管光照模型通常用于面绘制，但是并不意味着体绘制技术中不能使用光照模型。实际上,体绘制技术以物体对光的吸收原理为理论基础，在实现方式上，最终要基于透明度合成计算模型。此外，经典的光照模型，例如 phong 模型，

cook-torrance 模型都可以做为体绘制技术的补充，完善体绘制效果，增强真实感。

往往有初学者会分不清“体绘制技术”以及“透明光照模型”之间的区别。这个问题很有意思。实际上，体绘制技术与透明光照模型在感性认识上十分类似，在很多教程中对体绘制技术的阐述也涉及到透明物体。但是，透明光照模型，一般侧重于分析光在透明介质中的传播方式（折射，发散，散射，衰减等），并对这种传播方式所带来的效果进行模拟；而体绘制技术偏重于物体内部层次细节的真实展现。举例而言，对于一个透明的三棱镜，使用透明光照模型的目的在于“模拟光的散射，折射现象（彩虹）”；而对于地形切片数据或者人体数据，则需要使用体绘制技术观察到其中的组织结构。此外，在实现方式上，透明光照模型一般是跟踪光线的交互过程，并在一系列的交互过程中计算颜色值；而体绘制技术是在同一射线方向上对体数据进行采样，获取多个体素的颜色值，然后根据其透明度进行颜色的合成。

总的来说，透明光照模型侧重于光照效果展现，并偏向艺术化；而体绘制技术侧重展现物质内部细节，要求真实！

不过，现在体绘制技术实际上也可以用于艺术领域，因为体绘制技术所使用的方法，实际上具有很强的通用性，尤其是传统的 ray-cast 方法，完全可以应用到透明光照模型中（绘制烟雾等）。不同的技术之间会存在共融性，将技术和领域的关系近固化，是研究人员的大忌。科学史上很多前例都说明了一个事实：不同领域的交合点，往往会出现重大发现或发明。在爱因斯坦之前，又有谁知道时间、空间和质量之间的关系呢？

14.4 体数据（Volume Data）

学习任何一门技术，首先要弄清楚这项技术的起源以及数据来源。技术的起源也就是技术最原始的需求，最原始的发展动力，了解了这一点就了解了这项技术的价值。而了解一门技术的数据来源，就把握了技术的最初脉络，是“持其牛耳”的一种方法，正如软件工程中的数据流分析方法一般。

我很想说，体数据与面数据的区别，就好像一个实心的铁球和一个空心的兵乓球的区别。不过这个比喻很显然有点俗，很难让人相信作者（我）是一个专业人士。于是我决定还是将与体数据相关的专业术语都阐述一遍。

不过，在此之前，我需要先消除大家的恐惧感，研究表明，动物对于未知事物总是存在恐惧感，这也是阻碍进一步学习的关键所在。体数据不是什么特别高深的火星符号，它是对一种数据类型的描述，只要是包含了体细节的数据，都可以称之为体数据。举个例子，有一堆混凝土，其中包含了碳物质（C）若干，水分子（H20）若干，还有不明化学成分的胶状物，你用这种混凝土建造了块方砖，如果存在一个三维数组，将方砖 X、Y、Z 方向上的物质分布表示出来，则该数组可以被称为体数据。不要小看上面这个比喻，体数据本质上就是按照这个原理进行组织的！

体数据一般有 2 种来源：

1. 科学计算的结果，如：有限元的计算和流体物理计算；
2. 仪器测量数据，如：CT 或 MRI 扫描数据、地震勘测数据、气象检测数据等。

与体数据相关的专业术语有：体素（Voxel）、体纹理（Volume Texture）。尤其要注意：所谓面数据，并不是说二维平面数据，而是说这个数据中只有表面细节，没有包含体细节，实际上体数据和面数据的本质区别，在于是否包含了体细节，而不是在维度方面。

14.4.2 体素（Voxel）

Wikipedia中对体属voxel的介绍为：

A voxel (a portmanteau of the words volumetric and pixel) is a volume element, representing a value on a regular grid in three dimensional space. This is analogous to a pixel, which represents 2D image data in a bitmap。

即“体素，是组成体数据的最小单元，一个体素表示体数据中三维空间某部分的值。体素相当于二维空间中像素的概念”。图 42中每个小方块代表一个体素。体素不存在绝对空间位置的概念，只有在体空间中的相对位置，这一点和像素是一样的。

通常我们看到的体数据都会有一个体素分布的描述，即，该数据由n*m*t个体素组成，表示该体数据在X、Y、Z方向上分别有n、m、t个体素。在数据表达上，体素代表三维数组中的一个单元。假设一个体数据在三维空间上256*256*256个体素组成，则，如果用三维数组表示，就必须在每一维上分配256个空间。

在实际的仪器采样中，会给出体素相邻间隔的数据描述，单位是毫米（mm），例如0.412mm表示该体数据中相邻体素的间隔为0.412毫米。

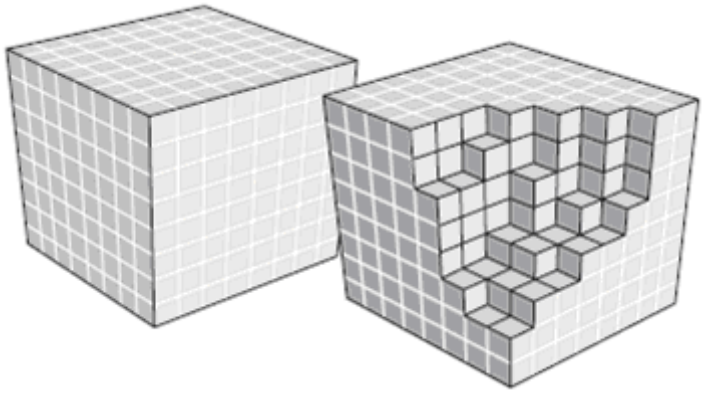


图 42 体数据中的体素

14.4.1 体纹理（Volume Texture）

体数据最主要的文件格式是“体纹理（volume texture）”！故而，非常有必要对体纹理的概念进行详细的阐述。

目前，学术性文章中关于体纹理的概念描述存在不小的混乱，很多书籍或者网页资料没有明确的区分2d texture，3d texture，volume texture之间的区别。导致不少人认为“只要是用于三维虚拟或仿真技术中的纹理都称之为3d texture”。这是一个误会。纹理上的2，3维之分本质上是根据其所描述的数据维数而定的，所谓2d texture指的是纹理只描述了空间的面数据，而3d texture则是描述了空间中的三

维数据。3d texture另一个较为学术化的名称是：volume texture。文献【22】上对体纹理的定义是：

3D texture (Three Dimensional Texture), also known as "volume texture," is a logical extension of the traditional (and better known) 2D texture. In this context, a texture is simply a bitmap image that is used to provide surface coloring for a 3D model. A 3D texture can be thought of as a number of thin pieces of texture used to generate a three dimensional image map. 3D textures are typically represented by 3 coordinates.

翻译成中文就是“三维纹理，即体纹理，是传统2D纹理在逻辑上的扩展。二维纹理是一张简单的位图图片，用于为三维模型提供表面点的颜色值；而一个三维纹理，可以被认为由很多张2D纹理组成的，用于描述三维空间数据的图片。三维纹理通过三维纹理坐标进行访问”。

从上面这句话，可以得到两点信息：

1. 三维纹理和体纹理是同一概念；三维纹理和二维纹理是不同的；
2. 三维纹理通过三维纹理坐标进行访问。

这时可能会有人提出问题了，图片都是平面的，怎么能表示三维数据？请注意，我们通常所看到的图片确实都是平面的，但是并不意味着x,y平面上的像素点不能存放三维数据，举一个例子：在高级语言编程中，我们完全可以用一维数组去存放三维数组中的数据，只要按照一定规则存放即可！

按照一定规则将三维数据存放在XY像素平面所得到的纹理，称之为volume texture。

体数据通常是由CT仪器进行扫描得到的，然后保存在图片的像素点上。目前国际上比较常用的体纹理格式有，基于DirectX的.dds格式和.raw格式。注意，很多人会将.raw格式当作摄像器材使用的那种格式，其实这两个格式的后缀虽然都是.raw，但是其数据组织形式是不同的。用于体纹理的.raw格式，存放的是三

维数据，用于摄像器材的.raw格式只是普通的二维图片。图 43从左到右分别是University of Tübingen（Germany）、Viatronix Inc.(USA)、Walter Reed Army Medical Center (USA)三家机构的通过仪器扫描得到的体纹理数据的体绘制图片。

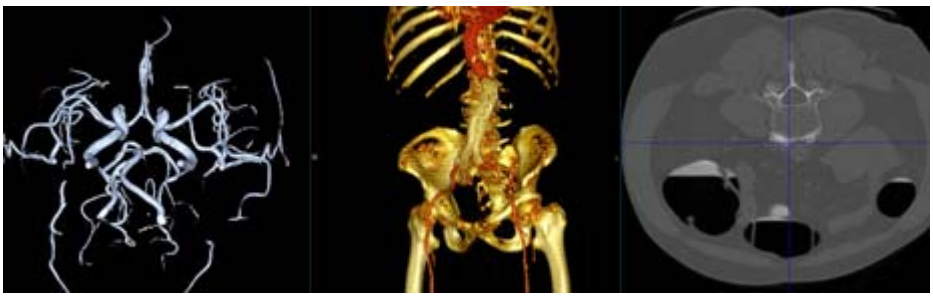


图 43 体纹理数据

这三个体纹理数据的描述分别是：256 x 320 x 128 \0.66, 0.66, 0.66；512 x 512 x 174\0.8398, 0.8398, 3.2；512 x 512 x 463\0.625, 0.625, 1.0。

由于在国内的网站上很难找到体数据，所以下面我给出几个国外的网址，这些网址提供用于教学和研究只用的体纹理数据（只能用于教学和研究）。

http://wwwvis.informatik.uni-stuttgart.de/~engel/pre-integrated/data.html

http://www9.informatik.uni-erlangen.de/External/vollib/

http://www.volren.org/

14.5 体绘制算法

国际上留下的体绘制算法主要有：光线投射算法（Ray-casting）、错切-变形算法（Shear-warp）、频域体绘制算法（Frequency Domain）和抛雪球算法（Splatting）。其中又以光线投射算法最为重要和通用。

究其原因，无外乎有三点：其一，该算法在解决方案上基于射线扫描过程，符合人类生活常识，容易理解；其二，该算法可以达到较好的绘制效果；其三，该算法可以较为轻松的移植到 GPU 上进行实现，可以达到实时绘制的要求。

本书的第 15 章将重点阐述光线投射算法。

第 15 章 光线投射算法（Ray Casting）

15.1 光线投射算法原理

光线投射方法是基于图像序列的直接体绘制算法。从图像的每一个像素，沿固定方向（通常是视线方向）发射一条光线，光线穿越整个图像序列，并在这个过程中，对图像序列进行采样获取颜色信息，同时依据光线吸收模型将颜色值进行累加，直至光线穿越整个图像序列，最后得到的颜色值就是渲染图像的颜色。

为什么在上面的定义是穿越“图像序列”，而不是直接使用“体纹理”？原因在于，体数据有多种组织形式，在基于 CPU 的高级语言编程中，有时并不使用体纹理，而是使用图像序列。在基于 GPU 的着色程序中，则必须使用体纹理。这里所说的图像序列，也可以理解为切片数据。

尤其要注意：光线投射算法是从视点到“图像序列最表面的外层像素”引射线穿越体数据，而不少教程中都是糊里糊涂的写到“从屏幕像素出发”，这种说法太过简单，而且很容易让人误解技术的实现途径，可以说这是一种以讹传讹的说法！从屏幕像素出发引出射线，是光线跟踪算法，不是光线投射算法。

体绘制中的光线投射方法与真实感渲染技术中的光线跟踪算法有些类似，即沿着光线的路径进行色彩的累计。但两者的具体操作不同。首先，光线投射方法中的光线是直线穿越数据场，而光线跟踪算法中需要计算光线的反射和折射现象。其次，光线投射算法是沿着光线路径进行采样，根据样点的色彩和透明度，用体绘制的色彩合成算子进行色彩的累计，而光线跟踪算法并不刻意进行色彩的累计，而只考虑光线和几何体相交处的情况；最后，光线跟踪算法中光线的方向是从视点到屏幕像素引射线，并要进行射线和场景实体的求交判断和计算，而光线投射算法，是从视点到物体上一点引射线（16.1.2 节会进行详细阐述），不必进行射线和物体的求交判断。

上述文字，对于光线投射算法的描述可能太过简单，会引起一些疑惑，不过

这是正常的，有了疑惑才会去思考解决之道，最怕看了以后没有任何疑惑，那只是浮光掠影似的一知半解，而不是真正的了然于胸。

15.1.1 吸收模型

几乎每一个直接体绘制算法都将体数据当作“在某一密度条件下，光线穿越体时，每个体素对光线的吸收发射分布情况”。这一思想来源于物理光学，并最终通过光学模型（Optical Models）进行分类描述。为了区别之前的光照渲染模型，下面统一将Optical Model翻译为光学模型。

文献【15】中对大多数在直接体绘制算法中使用的重要光学模型进行了描述，这里给出简要概述。

1. 吸收模型（Absorption only）：将体数据当作由冷、黑的体素组成，这些体素对光线只是吸收，本身既不发射光线，也不反射、透射光线；
2. 发射模型（Emission only）：体数据中的体素只是发射光线，不吸收光线；
3. 吸收和发射模型（Absorption plus emission）：这种光学模型使用最为广泛，体数据中的体素本身发射光线，并且可以吸收光线，但不对光线进行反射和透射。
4. 散射和阴影模型（Scattering and Shading/shadowing）：体素可以散射（反射和折射）外部光源的光线，并且由于体素之间的遮挡关系，可以产生阴影；
5. 多散射模型（Multiple Scattering）：光线在被眼睛观察之前，可以被多个体素散射。

通常我们使用吸收和发射模型（Absorption plus emission）。为了增强真实感，也可以加上阴影（包括自阴影）计算。

15. 2 光线投射算法若干细节之处

15.2.1 光线如何穿越体纹理

这一节中将阐述光线如何穿越体纹理。这是一个非常重要的细节知识点，很

多人就是因为无法理解“体纹理和光线投射的交互方式”而放弃学习体绘制技术。

前面的章节似乎一直在暗示这一点：通过一个体纹理，就可以进行体渲染。我最初学习体绘制时，也被这种暗示迷惑了很久，后来查找到一个国外的软件，可以将体纹理渲染到立方体或者圆柱体中，这时我才恍然大悟：体纹理并不是空间的模型数据，空间体模型（通常是规则的立方体或圆柱体）和体纹理相互结合才能进行体渲染。

举例而言，我们要在电脑中看到一个纹理贴图效果，那么至少需要一张二维的纹理和一个面片，才能进行纹理贴图操作。这个面片实际上就是纹理的载体。

同理，在体绘制中同样需要一个三维模型作为体纹理的载体，体纹理通过纹理坐标（三维）和模型进行对应，然后由视点向模型上的点引射线，该射线穿越模型空间等价于射线穿越了体纹理。

通常使用普通的立方体或者圆柱体作为体绘制的空间模型。本章使用立方体作为体纹理的载体。

注意：体纹理通过纹理坐标和三维模型进行对应，考虑到OpenGL和Direct3D使用的体纹理坐标并不相同，所以写程序时请注意到这一点。

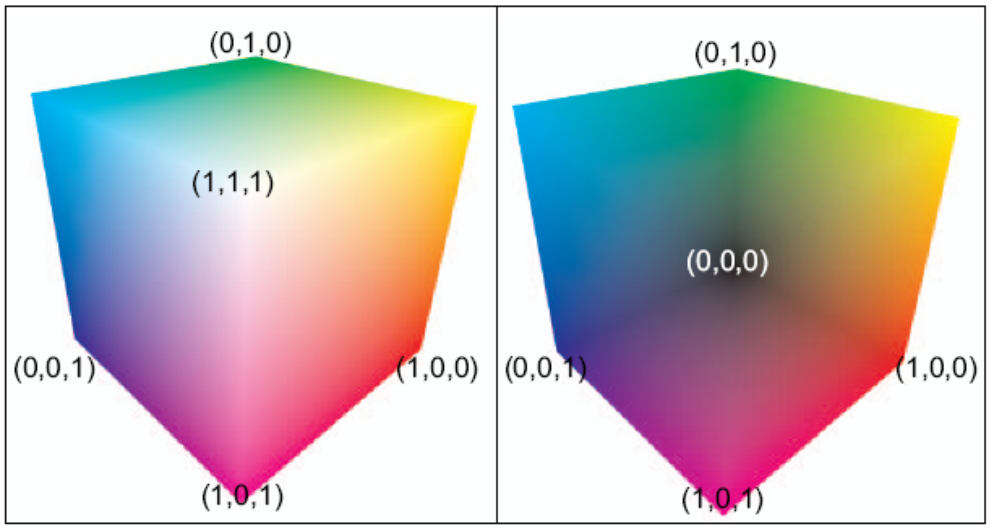


图 44 体纹理坐标分布

图 44 展示了体纹理坐标在立方体上的分布，经过测试，这种分布关系是基于 OpenGL 的。在宿主程序中确定立方体 8 个顶点的体纹理坐标，注意是三元向量，然后传入 GPU，立方体 6 个面内部点的体纹理坐标会在 GPU 上自动插值得到。

根据视点和立方体表面点可以唯一确定一条射线，射线穿越整个立方体等价于穿越体数据，并在穿越过程中对体数据等距采样，对每次得到的采样数据按照光透公式进行反复累加。这个累加过程基于 11 章讲过的透明合成公式，不过之前只是进行了简单的讲解，在本章中将针对透明度，透明合成，以及排序关系做全面阐述。

15.2.2 透明度、合成

透明度本质上代表着光穿透物体的能力，光穿透一个物体会导致波长比例的变化，如果穿越多个物体，则这种变化是累加的。所以，透明物体的渲染，本质上是将透明物体的颜色和其后物体的颜色进行混合，这被称为 alpha 混合技术。图形硬件实现 alpha 混合技术，使用 over 操作符。Alpha 混合技术的公式如下所示：

$$c_0 = a_s c_s + (1 - a_s) c_d \qquad (15\text{-}1)$$

其中，a_s 表示透明物体的透明度，c_s 表示透明物体的原本颜色，c_d 表示目标物体的原本颜色，c_0 则是通过透明物体观察目标物体所得到的颜色值。

如果有多个透明物体，通常需要对物体进行排序，除非所有物体的透明度都是一样的。在图形硬件中实现多个透明物体的绘制是依赖于 Z 缓冲区。在光线投射算法中，射线穿越体纹理的同时也就是透明度的排序过程。所以这里存在一个合成的顺序问题。可以将射线穿越纹理的过程作为采样合成过程，这是从前面到背面进行排序，也可以反过来从背面到前面排序，毫无疑问这两种方式得到的效果是不太一样的。

如果从前面到背面进行采样合成，则合成公式为：

$$\begin{aligned} C_i^{\Delta} &= (1-A_{i-1}^{\Delta})C_i + C_{i-1}^{\Delta} \\ A_i^{\Delta} &= (1-A_{i-1}^{\Delta})A_i + A_{i-1}^{\Delta} \end{aligned} \qquad (15\text{-}2)$$

其中，C_i 和 A_i 分别是在体纹理上采样所得到的颜色值和不透明度，其实也就是体素中蕴含的数据；C_i^{Δ} 和 A_i^{Δ} 表示累加的颜色值和不透明度。

注意，很多体纹理其实并没有包含透明度，所以有时是自己定义一个初始的透明度，然后进行累加。

如果从背面到前面进行采样合成，则公式为：

$$\begin{aligned} C_i^{\Delta} &= (1-A_i)C_{i+1}^{\Delta} + C_i \\ A_i^{\Delta} &= (1-A_i)A_{i+1}^{\Delta} + A_i \end{aligned} \qquad (15\text{-}3)$$

15.2.3 沿射线进行采样

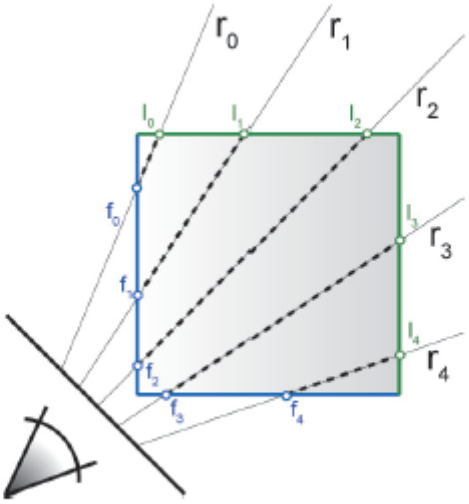


图 45 射线穿越体数据

如图 45 所示，假定光线从 F 点投射到立方体中，并从 L 点投出，在立方体

中穿越的距离为 m。当光线从 F 点投射到立方体中，穿越距离为 n（n<m）时进行采样，则存在公式：

$$t = t_{start} + d * delta \qquad (15\text{-}4)$$

其中 t_{start} 表示立方体表面被投射点的体纹理坐标；*d* 表示投射方向；*delta* 表示采样间隔，随着 n 的增加而递增；t 为求得的采样纹理坐标。通过求得的采样纹理坐标就可以在体纹理上查询体素数据。直到 n>m，或者透明度累加超过 1，一条射线的采样过程才结束。

下面总结一下：首先需要一个确定了顶点纹理坐标的三维立方体，光线穿越立方体的过程，就是穿越体纹理的过程，在整个穿越过程中，计算采样体纹理坐标，并进行体纹理采样，这个采样过程直到光线投出立方体或者累加的透明度为 1 时结束。

我想这个过程应该不复杂，大家一定要记住：纹理坐标是联系三维模型和体纹理数据之间的桥梁，通过计算光线穿越三维模型，可以计算体纹理在光线穿越方向上的变化，这就是计算采样纹理坐标的方法。

高中时学习物理的力学部分，最初一直处于浑浑噩噩的状态，遇到应用题不知道从何处入手，后来看一本参考资料讲到“加速度是联系力和运动状态的桥梁，遇到题目首先分析加速度的求法”，由此举一反三，不再感觉物理难学。所以在此我也借用那句话，总结纹理坐标的作用。

现在还存在一个问题：如何知道光线投射出了立方体？这个问题等价于计算光线在立方体中穿越的距离 m。在下一节中将进行阐述。

附：在 OpenGL 和 DirectX 中，体纹理坐标的分布规则是不一样的，所以要针对自己当前使用的 profile 来确定顶点体纹理坐标的设置。这也从侧面说明了，Cg 语言是基于 OpenGL 和 DirectX 的。

15.2.2 如何判断光线投射出体纹理

上一节阐述过：光线投射出体纹理，等价于光线投射出立方体。所以如何判断光线投射出体纹理，可以转换为判断光线投射出立方体。

首先计算光线在立方体中入射到出射的行进距离 m，然后当每次采样体纹理时同时计算光线在立方体中的穿越距离 n，如果 n>=m，则说明光线射出立方体。给定光线方向，以及采样的距离间隔，就可以求出光线在立方体中的穿越距离 n。

如果是在 CPU 上，距离 m 很容易通过解析几何的知识求得，直接求出光线和几何体的两个交点坐标，然后计算欧几里德距离即可。但是在 GPU 上计算光线和几何体的交点是一个老大难的问题，尤其在几何体不规则的情况下；此外，就算是规则的几何体，光线与其求交的过程也是非常消耗时间，所以通过求取交点然后计算距离的方法不予采用。

请思考一下，在 GPU 中确定点和点之间顺序关系的还有哪个量？深度值（我自问自答）。

在 GPU 中可以间接反应点和点之间关系的有两个量，一个是纹理坐标，另一个就是深度值。通常在渲染中会进行深度剔除，也就是只显示深度值小的片段。不过也存在另外一个深度剔除，将深度值小的片段剔除，而留下深度值最大的片段（深度值的剔除方法设置，在 OpenGL 和 Direct 中都有现成函数调用）。如果使用后者，则场景中渲染显示的是离视点最远的面片集合。

所以，计算距离 m 的方法如下：

1. 剔除深度值较大的片段（正常的渲染状态），渲染场景深度图 frontDepth（参阅第 14 章），此时 frontDepth 上的每个像素的颜色值都代表“某个方向上离视点最近的点的距离”；
2. 剔除深度值较小的片段，渲染场景深度图 backDepth，backDepth 上的每

个像素的颜色值都代表“某个方向上离视点最远的点的距离”；

3. 将两张深度图上的数据进行相减，得到的值就是光线投射距离 m。

如果认真实现过第 14 章讲的 shadow Map 算法，对这个过程应该不会感到太复杂。可能存在的问题是：背面渲染很多人没有接触过。这里对背面渲染的一些细微之处进行阐述，以免大家走弯路。

通常，背面的面片（不朝向视点的面片）是不会被渲染出来的，图形学基础比较好的同学应该知道，三个顶点通常按逆时针顺序组成一个三角面，这样做的好处是，背面面片的法向量与视线法向量的点积为负数，可以据此做面片剔除算法（光照模型实现中也常用到），所以只是改变深度值的比较方法还不够，还必须关闭按照逆\顺时针进行面片剔除功能，这样才能渲染出背面深度图。图 46 是立方体的正面和背面深度图。

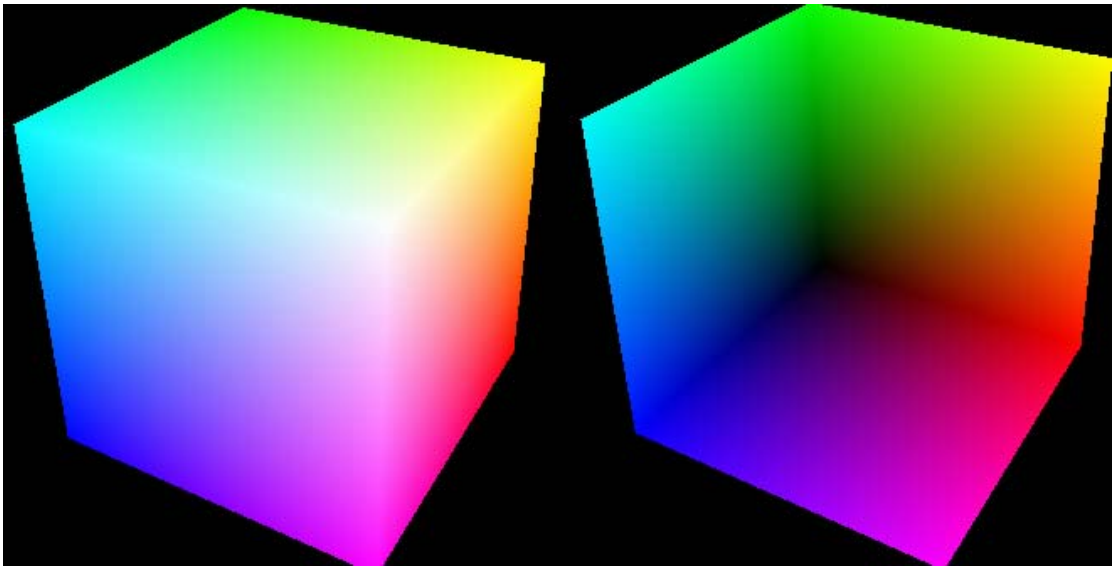


图 46 单位立方体的正面和背面深度图

附：在很多教程上，都是将 frontDepth 和 backDepth 相减后的值，保存为另外一个纹理，称之为方向纹理，每个像素由 r、g、b、a 组成，前三个通道存储颜色值，最后的 a 通道存放距离值，我觉得这个过程稍微繁琐了些，此外由于方向向量可能存在负值，而颜色通道中只能保存正值，所以必须将方向向量归一化到【0，1】空间，这个过程有可能导致数据精度的损失。基于如上的考虑，我将方向向量的计算放到片段着色程序中，通过视点和顶点位置进行计算。

15.3 算法流程

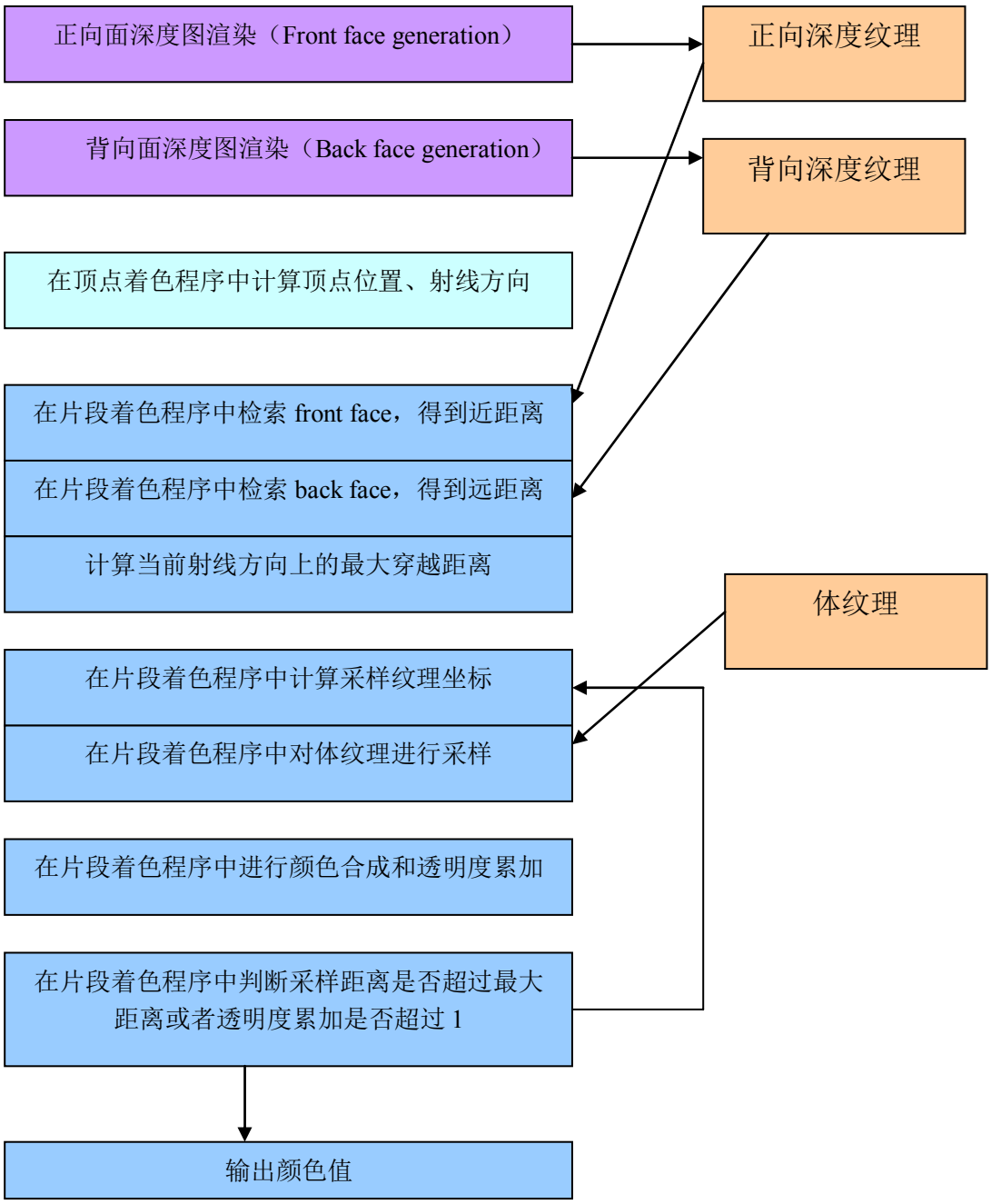


图 47 光线投射算法流程

图 47 展示了使用光线投射算法进行体绘制的实现流程。

首先要渲染出正向面深度图和背向面深度图，这是为了计算射线穿越的最大

距离，做为循环采样控制的结束依据；然后在顶点着色程序中计算顶点位置和射线方向，射线方向由视线方向和点的世界坐标决定，其实射线方向也可以放在片段着色程序中进行计算。然后到了最关键的地方，就是循环纹理采样、合成。

每一次循环都要计算新的采样纹理坐标和采样距离，然后进行颜色合成和透明度累加，如果采样距离超过了最大穿越距离，或者透明度累加到 1，则循环结束。将合成得到的颜色值输出即可。

图 48 给出了使用光线投射算法进行体绘制的效果图：

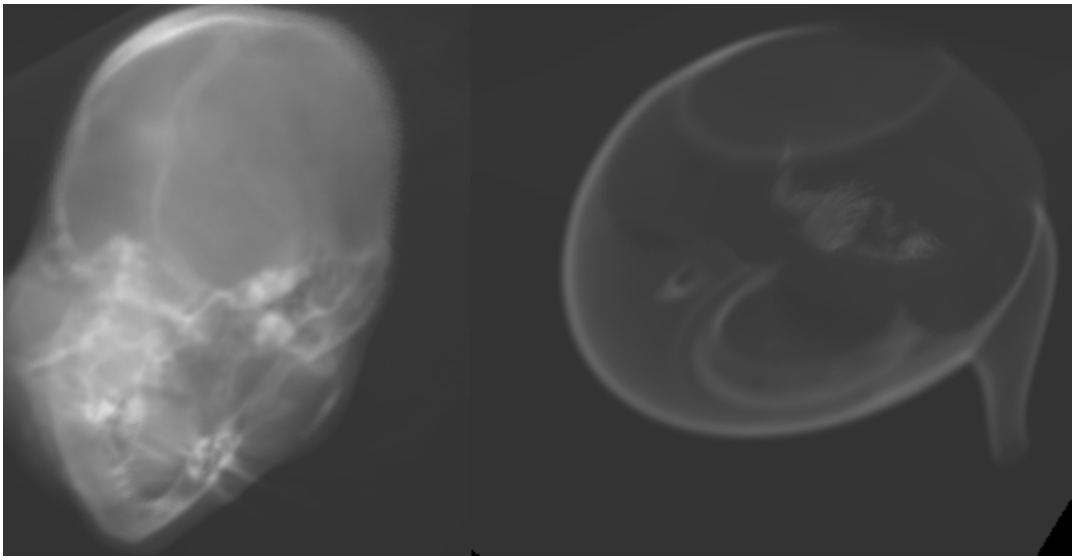


图 48 光线投射算法体绘制效果

15.4 光线投射算法实现

本节给出光线投射算法的着色程序实现代码。依然是分为三个部分：结构体、顶点着色程序和片段着色程序。

代码 22 光线投射算法结构体

```
struct VertexIn
{
   float4 position : POSITION;
   float4 texCoord:    TEXCOORD;
};

struct VertexScreen
{
   float4 position      : POSITION;
   float4 worldPos      : TEXCOORD0;
   float4 projPos       : TEXCOORD1;
   float4 texCoord      : TEXCOORD2;
};
```

代码 23 光线投射算法顶点着色程序

```
VertexScreen main_v(VertexIn posIn,
                          uniform float4x4 world,
                         uniform float4x4 worldViewProj,
                         uniform float4x4 texViewProj)
{
      VertexScreen posOut;

      posOut.position = mul(worldViewProj, posIn.position);
      posOut.worldPos = mul(world,posIn.position);
      posOut.projPos = mul(texViewProj, posOut.worldPos);
      posOut.texCoord = posIn.texCoord;

      return posOut;
}
```

代码 24 光线投射算法片段着色程序

```
void main_f(VertexScreen posIn,
            uniform float3 eyePosition,
            uniform sampler3D volumeTex: register(s0),
            uniform sampler2D frontDepthTex: register(s1) ,
            uniform sampler2D backDepthTex: register(s2) ,
             out float4 result       : COLOR)
{
    //根据视点和当前顶点世界坐标计算方向
    float3 dir = posIn.worldPos.xyz-eyePosition;
    dir = normalize(dir);
    float3 deltaDir = float3(0.0, 0.0, 0.0);

    //获取当前顶点的三维纹理坐标
    float3 tex = posIn.texCoord.xyz;
    float2 uvDelta;
    uvDelta.x = 0.0;//ddx( tex ).x;
    uvDelta.y = 0.0;//ddy( tex ).y;

    //取出深度间隔值,并设置采样间隔
    float2 uv= posIn.projPos.xy/posIn.projPos.w;
    float frontDis = tex2D(frontDepthTex,uv).x;
    float backDis = tex2D(backDepthTex,uv).x;
    float len = backDis-frontDis;

    //初始化颜色值、采样值、透明度
  float3 norm_dir = normalize(dir);
  float stepsize = 0.01;
  float delta = stepsize;
  float3 delta_dir = norm_dir * delta;
  float delta_dir_len = length(delta_dir);
  float3 vec = posIn.texCoord.xyz;
  float4 col_acc = float4(0,0,0,0);
  float alpha_acc = 0;
  float length_acc = 0;
  float4 color_sample;
  float alpha_sample;

  for(int i = 0; i < 800; i++){
        color_sample = tex3D(volumeTex,vec);
      alpha_sample = color_sample.a * stepsize;
      col_acc     += (1.0 - alpha_acc) * color_sample * alpha_sample * 3;
      alpha_acc += alpha_sample;
      vec += delta_dir;
      length_acc += delta_dir_len;
      if(length_acc >= len || alpha_acc > 1.0) break; //  采样循环控制条件
 }
    result.xyz = col_acc.xyz*2.0+float3(0.2,0.2,0.2);
    result.w = col_acc.w;
}
```

15.5 本章小结

本书的第 14、15 章阐述了体绘制中光线投射算法的基本原理和实现流程。实际上，在此基础上可以对光线投射算法加以扩展，例如将光线投射算法和阴影绘制算法相结合，可以渲染出真实感更强的图像。

此外，有些体数据是中间是空的，在射线方向上进行采样时需要跳过空区域，这其中也需要额外的算法处理，在英文中称为“Object-Order Empty Space Skipping”。

目前我所发现关于体绘制以及光线投射算法最好的教材是 Markus Hadwiger 等人所写的“Advanced Illumination Techniques for GPU-Based Volume Raycasting”。此书发表在 SIGGRAPH ASIA2008 上，是目前所能找到最新也是非常权威的教材，共 166 页。英文阅读能力比较好的同学可以尝试着看一下。

本章已经是此书的最后一章，最后希望中国的计算机科学可以真正上升到科学研究的层次，而不是一直在混沌中热衷做泥瓦匠的工作。

附录 A 齐次坐标

“齐次坐标是表示计算机图形学的重要手段之一，它既能够用来明确区分向量和点，同时也更加易于进行仿射（线性）集合变换”

----F.S.Hill,JR

在讲述齐次坐标之前，首先请大家思考两个问题：

(a,b,c)表示空间向量还是空间点？

空间向量与空间点有什么不同？

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \qquad \text{（A-1）}$$

该公式表示一个空间点绕三维 Z 轴旋转θ角度（计算机图形学第二版 317 页）。想想为什么空间三维点要用四元向量表示？为什么第四个量为 1，而不是为 0，或者其他值？

向量代表方向，对一个向量进行平移和缩放操作，向量本身所表示的方向性是不会发生变化；而空间点坐标是相对于坐标系的原点而言的。

对于一个向量，如果能被其他向量通过加法和乘法表示出来，这就是线性代数中所说的向量的线性组合：则对于一个向量v，可以找到一组坐标$v1$、$v2$、$v3$，使得公式（A-2）成立。

$$v = a \bullet v1 + b \bullet v2 + c \bullet v3; \qquad \text{（A-2）}$$

而对于一个点 p，可以找到一组坐标 $p1$、 $p2$、 $p3$，以及坐标原点 p_o 使得公式（A-3）成立：

$$p = a \bullet p1 + b \bullet p2 + c \bullet p3 + p_o ; \qquad \text{（A-3）}$$

公式（A-2）与公式（A-3）都可以转换为 2 个四元向量的点积形式，如公式（A-4）和（A-5）所示：

$$v = \begin{pmatrix} a & b & c & 0 \end{pmatrix} \bullet \begin{pmatrix} v1 & v2 & v3 & p_0 \end{pmatrix} \qquad \text{（A-4）}$$

$$p = \begin{pmatrix} a & b & c & 1 \end{pmatrix} \bullet \begin{pmatrix} p1 & p2 & p3 & p_0 \end{pmatrix} \qquad \text{（A-5）}$$

即：可以通过将三元向量扩展到四元向量，将向量与点的偏移转换公式统一到一起；如果第四元向量是 1，则代表点；如果是 0，则代表 1。

例如，如果表达式为$\begin{pmatrix} 100 & 100 & 100 & 1 \end{pmatrix}$，则说明是一个空间点，其坐标距离原点在三个坐标轴上的偏移量都为 100；如果表达式为$\begin{pmatrix} 100 & 100 & 100 & 0 \end{pmatrix}$，则说明是一个空间向量。

此外，在计算机图形学中引入四元数计算，可以将平移运算和缩放、旋转等运算统一到四阶矩阵的乘法运算中。点 $p\begin{pmatrix} x_p & y_p & z_p \end{pmatrix}$，平移$\begin{pmatrix} t_x & t_y & t_z \end{pmatrix}$，其新位置的计算公式为：

$$\begin{pmatrix} x'_p & y'_p & z'_p \end{pmatrix} = \begin{pmatrix} x_p & y_p & z_p \end{pmatrix} + \begin{pmatrix} t_x & t_y & t_z \end{pmatrix} ; \qquad \text{（A-6）}$$

我们可以将公式等价的转换到矩阵运算：

$$\begin{bmatrix} x'_p \\ y'_p \\ z'_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} \qquad \text{（A-7）}$$

总结一下：将三元向量扩充到四元向量，可以用来区分空间向量与空间点的数学表达；此外，在此基础上通过四阶矩阵运算，将空间点的平移、旋转、缩放等操作全部统一到矩阵的乘法运算中。

附录 B：体绘制的医学历程

（参考百科，现代医学，影像部分的内容）

1、CT的发明

自从X射线发现后，医学上就开始用它来探测人体疾病。但是，由于人体内有些器官对X线的吸收差别极小，因此X射线对那些前后重叠的组织的病变就难以发现。于是，美国与英国的科学家开始了寻找一种新的东西来弥补用X线技术检查人体病变的不足。 1963年，美国物理学家科马克发现人体不同的组织对X线的透过率有所不同，在研究中还得出了一些有关的计算公式，这些公式为后来CT的应用奠定了理论基础。1967年，英国电子工程师亨斯费尔德在并不知道科马克研究成果的情况下，也开始了研制一种新技术的工作。他首先研究了模式的识别，然后制作了一台能加强X射线放射源的简单的扫描装置，即后来的CT，用于对人的头部进行实验性扫描测量。后来，他又用这种装置去测量全身，获得了同样的效果。1971年9 月，亨斯费尔德又与一位神经放射学家合作，在伦敦郊外一家医院安装了他设计制造的这种装置，开始了头部检查。10月4日，医院用它检查了第一个病人。患者在完全清醒的情况下朝天仰卧，X线管装在患者的上方，绕检查部位转动，同时在患者下方装一计数器，使人体各部位对X线吸收的多少反映在计数器上，再经过电子计算机的处理，使人体各部位的图像从荧屏上显示出来。这次试验非常成功。1972年4月，亨斯费尔德在英国放射学年会上首次公布了这一结果，正式宣告了 CT的诞生。这一消息引起科技界的极大震动，CT的研制成功被誉为自伦琴发现X射线以后，放射诊断学上最重要的成就。因此，亨斯费尔德和科马克共同获取1979年诺贝尔生理学或医学奖。而今，CT已广泛运用于医疗诊断上。

CT是“计算机X线断层摄影机”或“计算机X线断层摄影术”的英文简称，是从1895年伦琴发现X线以来在X线诊断方面的最大突破，是近代飞速发展的电子 计算机控制技术和X线检查摄影技术相结合的产物。我国也在70年代末引进了这一

新技术，在短短的30年里，全国各地乃至县镇级医院共安装了各种型号的CT机数千台，CT检查在全国 范围内迅速地层开，成为医学诊断中不可缺少的设备。

CT是从X线机发展而来的，它显著地改善了X线检查的分辨能力，其分辨率和定性诊断准确率大大高于一般X线机，从而开阔了X线检查的适应范围，大幅度地提高了x线诊断的准确率。

但CT与传统X线摄影有所不同，在CT中使用的X线探测系统比摄影胶片敏感，是利用计算机处理探测器所得到的资料。CT的特点在于它能区别差异极小的X 线吸收值。与传统X线摄影比较，CT能区分的密度范围多达2000级以上，而传统X线片大约只能区分20级密度。这种密度分辨率，不仅能区分脂肪与其他软 组织，也能分辨软组织的密度等级。这种革命性技术显著地改变了许多疾病的诊断方式。

2、CT的工作原理

CT的工作程序是这样的：它根据人体不同组织对X线的吸收与透过率的差异，应用灵敏度极高的仪器对人体进行测量，然后将测量所获取的数据输入电子计算机，电子计算机对数据进行处理后，就可摄下人体被检查部位的断面或立体的图像，发现体内任何部位的细小病变。

CT是用X线束对人体的某一部分按一定厚度的层面进行扫描，当X线射向人体组织时，部分射线被组织吸收，部分射线穿过人体被检测器官接收，产生信号。因为人体各种组织的疏密程度不同，X线的穿透能力不同，所以检测器接收到的射线就有了差异。将所接收的这种有差异的射线信号，转变为数字信息后由计算机进行处理，输出到显示的荧光屏上显示出图像，这种图像被称为横断面图像。CT的特点是操作简便，对病人来说无痛苦，其密度、分辨率高，可以观察到人体内 非常小的病变，直接显示X线平片无法显示的器官和病变，它在发现病变、确定病变的相对空间位置、大小、数目方面非常敏感而可靠，具有特殊的价值，但是在疾 病病理性质的诊断上则存在一定的限制。

在进行CT检查时， 目前最常应用的断层面是水平横断面，断层层面的厚度与部位都可由检查人员决定。常用的层面厚度在1~10毫米间，移动病人通过检查机架后，就能陆续获得能 组合成身体架构的多张相 接影像。利用较薄的切片能获得较准确的资料，但这时必须对某一体积的构造进行较多切片扫描才行

CT的图像实际上是人体某一部位有一定厚度的体层图像。我们将成像的体层分成按矩形排列的若干个小的基本单元。而以一个CT值综合代表每个小单元内的物质的密度。这些小单元我们称之为体素。

同样，一幅CT图像是由许多按矩阵排列的小单元组成，这些组成图像的基本单元被称之为像素。

体素是一个三维概念，而像素是一个二维概念，像素实际上是体素在成像时的表现。像素越小，图像细节越清晰。

3、CT的成像基本原理

CT是用X线束对人体某部一定厚度的层面进行扫描，由探测器接收透过该层面的X线，转变为可见光后，由光电转换变为电信号，再经模拟/数字转换器（analog/digital converter）转为数字，输入计算机处理。图像形成的处理有如对选定层面分成若干个体积相同的长方体，称之为体素（voxel）。扫描所得信息经计算而获得每个体素的X线衰减系数或吸收系数，再排列成矩阵，即数字矩阵（digital matrix），数字矩阵可存贮于磁盘或光盘中。经数字/模拟转换器（digital/analog converter）把数字矩阵中的每个数字转为由黑到白不等灰度的小方块，即象素（pixel），并按矩阵排列，即构成CT图像。所以，CT图像是重建图像。每个体素的X线吸收系数可以通过不同的数学方法算出。

强度为 I0 的 X 射线,穿过厚度为 d 的物体后，由于物体对 X 射线的吸收或衰减作用，X 射线强度变为 I，其衰减符合下列公式

$$I = I_0 e^{-ud} \qquad \text{（B-1）}$$

其中 μ 为吸收系数或衰减系数。人体组织的 μ 是不均匀的，但如果将人体均匀分解成许多足够小的方块，每个小块组织(称为体素)的 μ 可以看作是均匀的。如果能测定每个体素的吸收系数,则可重建出人体某一断面的 X 射线吸收系数分布图，即 CT 图像。图像上对应体素的小单元，称像素。

CT 扫描时将一束准直 X 射线束从与人体纵轴垂直的方向透射人体，得到一个投影，此投影是 X 射线束通路上各个体素衰减作用的总和。如果每个体素的吸收系数分别为 μ1、μ2、μ3……，且厚度 d 均相等，则

$$I = I_0 e^{-(u_1+\ldots+u_n)d} \qquad \text{（B-2）}$$

I 为投影强度,可以用检测器测定，并用模－数转换器转变成数字（称为投影数据或扫描数据），输入电子计算机。不断改变投射角度，可以得到数万到数十万个数据。电子计算机对这些数据按照一定算法进行计算，就可求得所扫描层面上的每一个体素的 μ，通过数－模转换器，即可重建出该层面的图像。图像上每个像素都有一定的 μ 值，用不同的灰度表示，μ 代表组织的密度，在 CT 上称 CT 值。某一组织的 CT 值，是其 μ 值与水的 μ 值相比得出的，公式为

$$CT = \left(\frac{u - u_{water}}{u_{water}} \right) * 1000 \qquad \text{（B-3）}$$

附录 C：模板阴影（Stencil Shadow）

模板阴影（Stencil Shadow）算法是基于网格几何体和模板缓冲区（stencil buffer）的阴影渲染算法。其主要优势在于：不需要高端显卡支持，只需要基本的模板缓冲区即可；阴影精度高，边缘清晰（既是优点，也是缺电），无锯齿现象；光源的类型和位置对性能没有影响，而 shadow map 技术会受到光源类型的

影响。

正因为上述优点，该算法被广泛的应用于游戏中，目前主流的图形引擎基本都集成了该算法，例如 OGRE。

理解模板阴影首先需要理解两个概念：阴影体（shadow volume）和模板缓冲区。实际上模板阴影名称的由来就是因为该算法要基于 stencil buffer 实现。模板缓冲区的概念在 2.4.3 节进行了详细阐述。

什么是阴影体？阴影体是 Frank Crow 在 1977 在文献【19】中提出的，本质上一个几何区域，用于描述光线被遮挡的空间区域。请看图 49,三角形 ABC 遮挡了来自光源的光线，然后在地面上形成了一个投影三角面面 DEF，ABC 与 DEF 之间的空间体称为阴影体（shadow volume，也称之为阴影锥）。

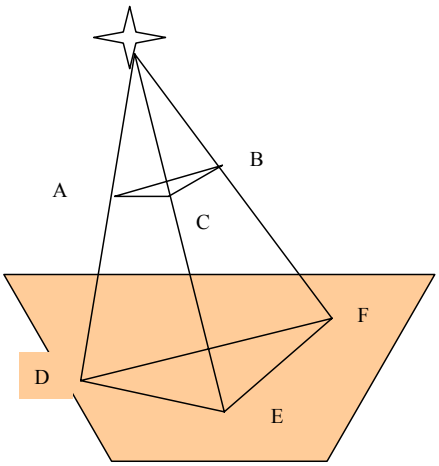


图 49 阴影体

模板阴影算法的基本原理是：找出场景中哪些点处于阴影体中，然后将这些点所对应的像素赋予阴影的颜色。很明显该算法有两个关键之处：

1：计算空间点与阴影体的关系（包含或者非包含）。

2：在阴影体中的空间点所对应的像素进行标记，并在绘制的时候对这些标记的像素点绘制暗色调。

计算空间点与阴影体的关系，等价于判断一个像素点所代表的 3D 空间点是否在阴影体中。一般使用经典的直线穿越次数判断算法，即：通过计算视点到该点的连线穿越阴影体面的次数，判断点是否在阴影体中。想象从视点出发射向空间点，如果射线和阴影体完全没有交集，那么点肯定不在阴影体中，该点所对应的像素不必做阴影处理；如果射线与阴影体有交集，并不表示该点一定在阴影体中，因为射线有可能会射入后再射出。所以只有在射线射入阴影体后，并终止于阴影体中，才表示该点在阴影体内。

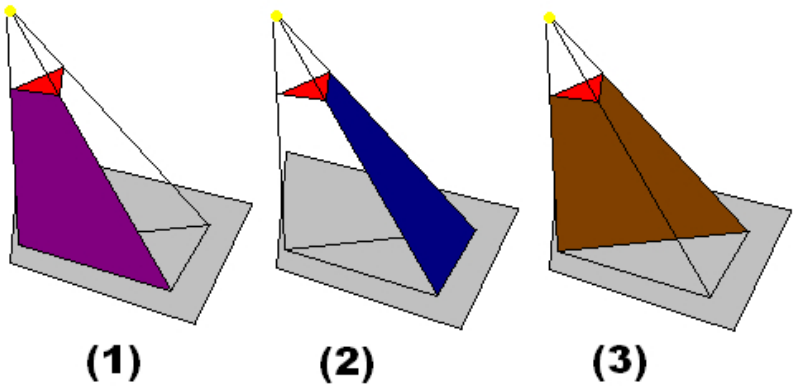


图 50 阴影体各面与视线的关系

图 50 展示了阴影体各个面与视线的关系（该图来自于网络，已无法查到原始作者，特此说明）。图 50（1）与图 50（2）都是面向观察者的面，也就是射线射入阴影体的面；图 50（3）是背对观察者的面，所以是射线射出阴影体的面。所以计算空间点与阴影体关系，等价于计算“从视点到空间点的”射线进出阴影体的次数。具体的算法有 Z Pass 算法和 Z Fail 算法。先看看 Z Pass 算法。

Z Pass 算法：从视点向空间点引一条射线，当射向进入阴影体时，stencil 加一；当射线离开阴影体时，stencil 减一。如果 stencil 值大于零，则表示该空间点在阴影体中；如果 stencil 值为零，则表示射线进入阴影体和离开阴影体的次数相等，即，空间点不在阴影体中；stencil 值不可能小于零，因为只有进入阴影体才可能离开，不可能存在离开的次数大于进入的次数。

注意：stencil 是一个标记值，初始为零，之所以取名为 stencil，是因为该值存放在模板缓冲中（参阅 2.4.3 节）。

明白了 Z Pass 算法后，如何实现它呢？首先需要两个缓存器：z buffer（深度缓冲器）和 stencil buffer（模板缓冲器）。算法流程如下：

1：开启深度写（Z Enable，允许向 z buffer 中写入值），关闭光源，渲染整个场景，目的是为了获得深度值；

2：关闭深度写（Z Disable，不允许向 z buffer 中写入值），只渲染阴影体的正面（面向视线的面），如果深度测试通过则 stencil 值加一。所谓深度测试，即“第一步得到的像素深度值，与渲染阴影体正面得到的深度值进行比较，如果前者小于后者（场景像素更接近视点），则谓之深度测试失败”，反之，如果深度测试通过，则说明场景像素所对应的空间点在阴影体正面之后，stencil 加一，即射线进入阴影体中。

3： 渲染阴影体背面（背向观察者的面），深度通过（场景像素深度大于阴影体背面深度），即表示射线出阴影体，stencil 减一；

4：模板值大于零的像素点对应的空间点在阴影体中。现在开启深度写，并开启光源，重新渲染场景，考察每个像素的 stencil 值（存放在 stencil buffer 中），如果大于零，则设置像素颜色值为暗色调。

这里需要补充说明一点：为什么在进行第一步渲染时要关闭光源？原因在于：之前渲染场景完全是为了获得场景像素的深度值，而开启光源只会花更多时间进行光照渲染，但这是不必要的。这也说明了一点：计算阴影时，要两次渲染场景。

Z Pass 算法有一个严重的缺点：当视点在阴影体中时，会导致 stencil 值计算错误，因为视点在阴影体中，会使得视线失去一次进入 shadow volume 的机会。

Z Fail 算法弥补了 Z Pass 算法的缺点，即：解决了视点进入阴影体后，z pass 失效的问题。Z Fail 算法的流程为：

1： 开启深度写，关闭光源，渲染整个场景，获取深度值；

2： 关闭深度写，渲染阴影体背面，深度测试失败（场景像素深度值小于阴影体背面深度值），则 stencil 加一；

3：渲染阴影体正面，深度测试失败（场景像素深度值小于阴影体正面深度值场

景像素），则 stencil 减一；

4： 最后 stencil 值不为零的像素点处于阴影体中。开启阴影写，开启光源，重新渲染场景，并查看每个像素点的 stencil 值是否为零；如果不为零，则对像素颜色值赋予暗色调。

Z Fail 算法要求阴影体是闭合的（有时候视锥裁剪会导致阴影体是敞开的）。

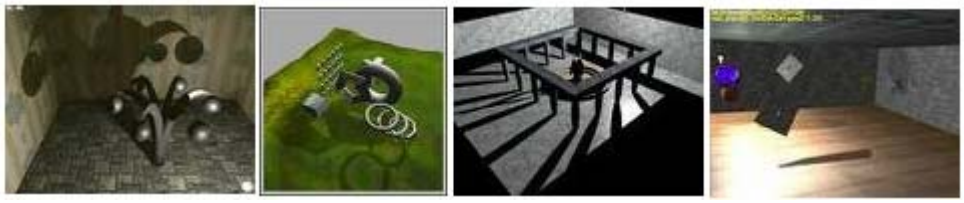


图 51 模板阴影算法渲染效果

模板阴影的好处在于：可以精确表现动态光影技术（如果是基于纹理或者像素做阴影计算，会由于精度的原因导致锯齿现象），适用性广，其在 Doom3 中有广泛的使用；不足在于：确立阴影体，需要提前对实体进行物体勾边（object Outlining），即，计算出物体网格的轮廓，同时渲染的阴影比较硬，无法渲染软阴影。

参考文献

1 Feng Liu, Platform Independent Real-time X3D Shaders and Their Applications in Bioinformatics Visualization, Georgia State University, 2005, ISBN: 0-542-56173-5, Order Number: AAI3207182.

2 Randima Fernando, Mark J. Kilgard, The Cg Tutorial The Definitive Guide to Programmable Real-Time Graphics, Addison-Wdsley Professional, March 8, 2003, ISBN-10: 0321194969.

3 Cg Toolkit User's Manual A Developer's Guide to Programmable Graphics. Release 1.4, September 2005.

4 Kr uger, J., and Westermann, R. 2003. Linear algebra operators for GPU implementation of numerical algorithms. ACM Trans. Graph. 22, 3, 908-916.

5 Harris, M. J., Baxter, W. V., Scheuermann, T., and Lastra, A. 2003. Simulation of cloud dynamics on graphics hardware. In Proceedings of Graphics hardware, Eurographics Association, 92-101.

6 Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P. 2002. Ray tracing on programmable graphics hardware. ACM Trans. Graph., 703-712.

7 Alexander J.Faaborg, A Psychophysical Study of BRDF-Based Lighting, May 10, 2001.

8 Wynn, C. (2000). An Introduction to BRDF-Based Lighting. nVIDIA Corporation.

9 Michael Ashikhmin, Peter Shirley, An Anisotropic Phong BRDF Model. August 13, 2000.

10 Blinn, J.F., and M.E. Newell, "Texture and reflection in computer generated images", Communications of the ACM, vol.19, no. 10, pp.542-547, October 1976.

11 Greene, Ned, "Environment Mapping and Other Applications of World Projections", IEEE Computer Graphics and Application, vol, 6, no, 11, pp21-29, November 1986.

12 Cass Everitt, Projective Texture Mapping, 04/24/2001

13 Lance Williams, Casting curved shadows on curved surfaces, 1978, ISSN:0097-8930, page: 270-274.

14 Levoy,M. Display of surfaces from volume data. May, 1988, Computer Graphics and Applications, IEEE, Volume: 8, Issue: 3.

15 Nelson Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995.

16 Matthieu Sozeau , Nicolas Qury, First-Class Type Classes.

17 steve Baker, Learning to love your Z-buffer.

18 Markus Hadwiger, Patric Ljung, Christof Rezk Salama, Timo Ropinski. Advanced Illumination Techniques for GPU-Based Volume Raycasting. SIGGRAPH ASIA2008.

19 Crow,Franklin C:"Shadow Algorithms for Computer Graphics", Computer Graphics (SIGGRAPH 77 Proceedings), vol. 11, no. 2, 242-248.
20 http://nshader.codeplex.com/
21 http://www.opengpu.org/
22 www.tech-faq.com3d-texture.shtml#
23 http://wwwvis.informatik.uni-stuttgart.de/~engel/pre-integrated/data.html
24 http://www9.informatik.uni-erlangen.de/External/vollib/
25 http://www.volren.org/
26 Randima Fernando, Mark J.Kilgard, 洪伟、刘亚妮、李骑、丁莲玲翻译，Cg 教程_可编程实时图形权威指南，2004 年 9 月，人民邮电出版社，ISBN: 711512430。
27 Tomes Akenine-Moller, Eric-Haines, 普建涛（翻译），实时计算机图形学第二版，北京大学出版社，2004-8-18，书号 7301071051。
28 Stephen C.Dewhurst，荣耀翻译，C++必知必会，2005 年 12 月 16 号，人民邮电出版社，书号 7115141010。
29 张庆丹，基于 GPU 的串匹配算法的实现，中国科学院研究生院，2006 年 4 月。
30 须晖，高级着色语言及其优化编译，浙江大学，2004 年 2 月。
31 Frangois X.Sillion, James R.Arvo, A Global Illumination Solution for General Reflectance Distributions. ACM O-89791-436-8/91/007/0187, 1991.
32 Robert L. Cook, A Reflectance Model For Computer Graphics, ACM O-8971-045-I/81-0800-0370. 1981.
33 Xin Sun, Kun Zhou, Yanyun Chen, Interactive Relighting with Dynamic BRDFs.
34 Wolfgang Heidrich , Hans-Peter Seidel. Efficient Rendering of Anisotropic Surfaces Using Computer Graphics Hardware.
35 David C.Banks. Illumination In Diverse Codimensions.
36 Zhao Dong, Jan Kautz, Christian Theobalt. Interactive Global Illumination Using Implicit Visibility.
37 Daniel Reiter Horn, Jeremy Sugerman, Mike Houston. Interactive k-D Tree GPU Raytracing.
38 Benthin,C., Wald,I., Scherbaum, M., And Friedrich,H. 2006. Ray Tracing on the Cell Processor. In Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing.
39 Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., And Hanrahan, P. 2004. Brook for GPUs: Stream computing on graphics hardware. In

Proceedings of ACM SIGGRAPH 2004.

40 Xin Sun, Kun Zhou, Eric Stollnitz. Interactive Relighting of Dynamic Refractive Objects.

41 Timothy J.Purcell, Ian Buck, William R.Mark. Ray Tracing on Programmable Graphics Hardware.

42 Ziyad S.Hakura, John M.Snyder. Realistic Reflections and Refractions on Graphics Hardware With Hybrid Rendering and Layered Environment Maps.

43 Turner Whitted, An Improved Illumination Model for Shaded Display. 1980 ACM 0001-0782/80/0600-0343.

44 Mark Segal, carl Korobkin, Rolf van Widenfelt. Fast Shadows and Lighting Effects Using Texture Mapping. 1992 ACM-0-89791-479-1/92/007/0249.

45 Mark J.Kilgard. Improving Shadows and Reflections via the Stencil Buffer. NVIDIA Corporation.

46 Randima Fernando. Percentage-Closer Soft Shadows. NVIDIA Corporation.

47 Fan Zhang, Hanqiu Sun, Leilei Xu. Parallel-Split Shadow Maps for Large-scale Virtual Environments.

48 Uwe Behrens, Ralf Rastering. Adding shadows to a texture-based volume renderer. In VVS'98: Proceedings of the 1998 IEEE symposium on Volume visualization, pages 39-46. ACM Press, 1998.

49 Johanna Beyer, Markus Hadwiger. Smooth Mixed-Resolution GPU Volume Rendering. In IEEE/EG International Symposium on Volume and Point-Based Graphics, pages 163-170, 2008.

50 Praveen Bhaniramka, Yves Demange. OpenGL Volumizer: A Toolkit for High Quality Volume Rendering of Large Data Sets. In Proceedings IEEE Visualization 2002, pages 45-53, 2002.

51 Imma Boada, Isabel Navazo, Roberto Scopigno. Multiresolution volume visualization with a texture-based octree. The Visual Computer, 17:185-197, 2001.

52 N. Carr, J.Hall, J.Hart. GPU Algorithms for Radiosity and Subsurface Scattering. In proc. Graphics Hardware, 2003.

53 Nathan A. Carr, Jesse D.hall, and John C.Hart. GPU algorithms for radiosity and subsurface scattering. In HWWS'03: Proceedings of the conference on Graphics Hardware '03, pages 51-59. Eurographics Association, 2003.

54 R.A.Drebin, L.Carpenter, and P. Hanrahan. Volume rendering. In Proceedings of SIGGRAPH '88, pages 65-74, 1988.