硕 士 学 位 论 文

论文题目 基于可编程图形硬件加速的若干技术研究

作者姓名 董 朝

指导教师 彭群生 教授 陈为 副教授

学科(专业) 计 算 机 应 用 技 术

所在学院 计 算 机 学 院

提交日期 2005 年 3 月

Dissertation Submitted to Zhejiang University
For Master Degree of Science

Relevant Technology Study on Programmable Graphics Hardware

Written by
Zhao Dong

Majoring in
Computer Science

Supervised by
Professor Qunsheng Peng
Associate Professor Wei Chen

College of Computer Science
Zhejiang University
March 2005

目　录

摘　　要

目前图形硬件中的图形处理器(GPU)计算能力的增长速度已经超过了中央处理器(CPU)计算能力的增长速度，主流图形硬件制造商声称，现在每隔 12 个月 GPU 的性能就会增长一倍。图形硬件技术一个最主要的突破就是在图形硬件中引入了可编程功能，此功能允许用户编制自定义的着色器程序(Shader program)来替换原来固定流水线中的某些功能模块，使得 GPU 在功能上更像一个通用处理器. 虽然 GPU 具有非常高的计算速度，但并不能直接将以前在 CPU 中实现的算法照搬到 GPU 中来执行, 这是因为 GPU 的指令执行方式和 CPU 不一样，GPU 的体系结构是一种高度并行的单指令多数据(SIMD)指令执行体系。所以要基于可编程图形硬件实现一些在 CPU 中效率较低的算法，就必须重新组织算法实现的数据结构和步骤，以充分利用 GPU 并行处理体系结构带来的性能优势。本文中的几种算法都基于可编程图形硬件实现，在达到实时效率的同时保证了结果的质量。

本文中的研究工作主要包括以下几个方面：

1. 实时体素化及其应用

提出了一种面向复杂几何模型的高效体素化方法。算法首先将几何模型依据各面片的朝向将它们分别变换到三个离散的体空间，然后将每个体空间中生成的体素以二维纹理的方式存储在三张工作表格(worksheet)中，三张工作表格最终合并成为一张包含全部体模型数据的工作表格。算法整个运行过程中只需要遍历初始几何模型一次。由于整个运行过程全部在GPU中实现，对于两百万面片数的几何模型算法能够达到实时。该算法实现简单并且易于扩展到体建模、透明绘制、碰撞检测等许多具体应用中。

2. 大尺寸点模型实时高质量绘制

提出了一种大尺寸点模型的自适应绘制算法。该算法在预处理阶段首先将点模型分割为很多点片，建立每一个点片的层次结构并以线性二叉树的方式保存；在接下来的绘制过程中对点模型分片进行处理，通过快速的可见性测试剔除掉不可见的点片，可见的点片则会依据距离视点的远近选取合适的绘制模式在 GPU 中实时绘制。算法不仅充分发挥了 GPU 的性能并且有效地均衡了 GPU 和 CPU 之间的负载。为解决大尺寸模型数据量过大的问题，我们还提出了一种快速的压缩/解压缩技术，可以将显存中的绘制数据压缩 8 倍以上。基于以上算法，可以在普通 PC 平台上实现百万数量级

点采样模型的实时高质量绘制。.

3. 实时阴影映射

阴影映射是一种基于图像空间的阴影绘制算法。该算法基于图形硬件提供的纹理(texture)和深度缓存(depth buffer)等技术实现，依靠 GPU 加速可以达到很高的绘制效率。文中会详细介绍两种实时阴影映射的实现方法：普通基于 GPU 实现的阴影映射和硬件阴影映射。

在本文的最后，作者总结了自己关于可编程图形硬件技术的一些经验和体会，并提出了一些未来的研究方向。

关键词： GPU；可编程图形硬件；体素化；实时绘制；点绘制；阴影

Abstract

The computation power of the Graphics Processing Unit (GPU) in current commodity graphics hardware is increasing at a much faster rate than that of the Central Processing Unit (CPU) in computer systems. The projected time to double in efficiency for the GPU is quoted to be roughly 12 months by the leading graphics card manufacturers. A recent major breakthrough in graphics hardware technology has been the introduction of programmability; this allows the user to replace portions of the fixed graphics pipeline with customized shader programs exposing the ability of GPU to function more like a general processing unit. In spite of all the rendering power, it is not possible or meaningful to use algorithms designed with CPU in mind on graphics hardware. The essential difference is that GPU provides a highly parallel Single Instruction Multiple Data Set (SIMD) architecture. The key to harnessing this resource is reengineering the computationally expensive algorithms to take advantage of this architecture as well as making use of rendering optimizations built into the programmable graphics pipeline. This thesis presents several novel graphics approaches which utilize programmable graphics hardware to obtain both real-time frame rate performance and high quality result.

Our research works in this thesis mainly focus on the following aspects:

1. Real-time Voxelization for Complex Models

We present an efficient voxelization algorithm for complex polygonal models by exploiting newest programmable graphics hardware. We first convert the model into three discrete voxel spaces according to its surface orientation. The resultant voxels are encoded as 2D textures and stored in three intermediate sheet buffers called directional sheet buffers. These buffers are finally synthesized into one worksheet, which records the volumetric representation of the target. The whole algorithm traverses the geometric model only once and is accomplished entirely in GPU, achieving real-time frame rate for models with up to 2 million triangles. The algorithm is simple to implement and can be integrated easily into diverse applications such as volume based modeling, transparent rendering and collision detection.

2. High Quality Real-time Rendering of Large Scale Point Model

Here we introduce an adaptive rendering algorithm for large scale point models. The algorithm first subdivide the target model into multiple patches in preprocess. A hierarchical structure is built for each patch and then converted into a linear binary tree. During rendering, the model is processed patch by patch. Fast visibility decision is made to cull invisible patches. Visible patches are displayed in GPU by choosing appropriate rendering mode, i.e, a distance-dependent strategy. Our algorithm takes full advantage of GPU and effectively balances the workload between CPU and GPU. We also propose a fast compression/decompression technique which achieves 8 times compression ratio. The results demonstrate high performance and image quality rendering for large scale point models in consumer PC.

3. Real-time shadow mapping

Shadow mapping is an image-based shadowing technique. It is particularly amenable to hardware implementation because it makes use of hardware functionality– texturing and depth buffering existed. Here we present the implementation process of two real-time shadow mapping methods in detail: common GPU-based shadow mapping and hardware shadow mapping.

Finally, I summarize my own research experience of programmable graphics pipeline and propose some potential research topics in the future.

Key words: GPU; Programmable Graphics Hardware; Voxelization; Real-time rendering; Point-based rendering; Shadow mapping

第一章 可编程图形硬件技术综述

1.1 可编程图形硬件的发展

多年来计算机图形处理器(Graphics Processor Unit, GPU)以大大超过摩尔定律的速度高速发展，极大地提高了计算机图形处理的速度和图形质量，并促进了计算机图形相关应用领域的发展。

目前微机平台的图形处理器已经达到了非常高的性能，2004 年，Nvidia GeForce 6800 Ultra 图形处理器的处理能力已经可以达到 40 Gigaflops，而 Intel 3GHz Pentium 4 采用最新的 SSE 指令集也只能达到 6 Gigaflops。自从 1993 年以来，GPU 的性能以每年 2.8 倍的速度增长，估计这样的增长速度还可以维持 5 年左右的时间，现在每隔半年左右，新的一代 GPU 的处理速度便会提高一倍[1]。

图形处理器技术的迅速发展带来的并不只是处理速度的提高，还产生了很多全新的图形硬件技术，其中最引人注目的便是在图形硬件处理管道的顶点处理和象素处理模块中引入了可编程性，使得用户可以通过程序方式控制图形流水线的执行，极大地扩展了图形处理器的能力和应用范围。图 1.1 为一个最基本的可编程图形硬件的框架，其中阴影模块代表可编程模块。

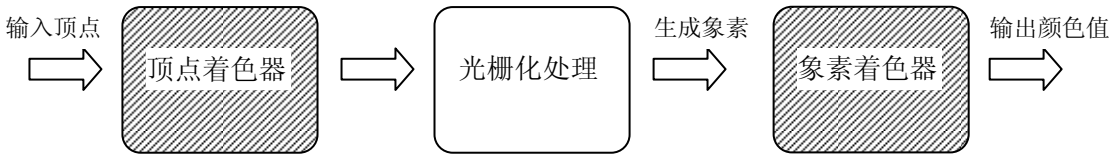


图 1.1 可编程的图形硬件框架

从上图可以看出，在顶点级操作上，引入了顶点着色器(vertex shader)处理每个顶点，用户可以通过自己编写代码实现专门的光照明模型。经过光栅化后，在象素处理阶段，引入的象素着色器(pixel shader)可以实现对每个象素的可编程操作。顶点着色器和象素着色器都是典型的流处理机(stream processor)，这种流处理机和传统的向量处理机的主要区别在于，它不具有大容量的快存/存储器可以读写，只是直接在芯片上利用临时寄存器做流数据的操作。对于 GPU 而言，图形流数据分别是顶点图元以及

光栅化后的象素；根据图形处理的特点，GPU 流出里的元素为 4 个单元的向量，可以利用它来表示三维齐次坐标，三维空间齐次向量，颜色等数据，正是这种流处理机的并行结构，实现了指令的并行处理，目前绝大部分的 GPU 都拥有多条可以并行的 Shader 管线，这种体系结构使得其不仅可以用于高效图形绘制，而且可以成为通用并行计算平台。

自从 1998 年以来，GPU 的功能迅速更新，平均每一年就有新的一代 GPU 问世，在现代 GPU 概念出现以前，只有 Silicon Graphics(SGI)等图形工作站上的特殊的硬件才具有基于硬件的顶点变换和纹理映射功能。第一代图形处理器出现在 1998 年后期，主要代表为 Nvidia TNT2, ATI Rage 和 3DFX Voodoo3，这些处理器主要处理光栅化部分，部分芯片支持多纹理，可以在光栅化过程中完成多幅纹理的融合操作；从 1999 年后期开始，第二代 GPU(Nvidia GeForce 256,GeForce 2 和 ATI Radeon 7500)可以处理顶点的矩阵变换和进行光照计算，但此时还没有出现真正的可编程功能；第三代 GPU(Nvidia GeForce 3, GeForce 4, ATI Radeon 8500, 大约 2001 年至 2002 年早期)有了重要的技术变革，此时图形硬件的流水线可以作为流处理器来解释，顶点级出现了可编程性，象素级也出现了有限的可编程性，但在象素级程序中，访问纹理的方式和格式受到一定限制，只有定点数可用；第四代 GPU(Nvidia GeForce FX series, ATI Radeon 9700/9800)的顶点和象素可编程性更加通用化，依赖纹理更为灵活，可以索引方式访问数据，GPU 具备了浮点功能，纹理中保存的值不再依赖于[0,1]范围，可以读写一般的浮点数；最新的第五代 GPU 以 Nvidia GeForce 6800 为代表[10,12]，功能相对以前更为丰富，灵活，顶点级程序可以访问纹理，支持程序的动态条件分支，象素级程序也开始支持分支操作，如循环，if/else 等，支持子函数调用，在纹理滤波和融合过程中支持 64 位的浮点精度，同时支持多个渲染目标[2-5]。

目前最新的可编程图形硬件已经具备了下列功能：

(1) 在顶点级和像素级提供了灵活的可编程特性。

(2) 在顶点级和像素级运算上都支持IEEE 32位浮点运算，可进行高精度的绘制。

(3) 完全支持4元向量的数据格式（齐次坐标，法向等），方便了图形程序的设计与开发。

(4) 具有高带宽的内存传输能力(>27.1GB/s),具备强大的数据吞吐能力。

(5) 支持绘制到纹理的功能(Render to texture)，从而避免将中间绘制结果拷贝到纹理这个费时的过程；

(6) 支持依赖性纹理访问功能，以方便数据的索引访问，可以将纹理作为内存来使用。

1.2 可编程图形流水线功能介绍

可编程图形流水线的总体框架如图 1.2 所示，左边用实线表示的流程就是传统的图形流水线的流程；在这种通用的流水线中，首先经过顶点级的光照计算和坐标变换，求出每个顶点的光照颜色值，同时还将顶点坐标从物体坐标系转换到裁剪空间(Clip Space)。然后，对每个三角形进行光栅化处理并将对三角形顶点的颜色进行双线性插值，得到了三角形中每一个象素的颜色值。接着进行纹理映射，即根据每一个象素的纹理坐标值将纹理图颜色分配到每个象素上。最后进行颜色混合计算(Blending)和雾化效果计算，得到的结果将会放进帧缓存(Frame buffer)并显示到屏幕上。

目前的可编程的图形硬件中，除了光栅化这一部分依然保持固化的硬件实现不变以外，其他部分都引入了可编程性。也就是图 1.2 中的虚线部分[11]。

顶点着色器(Vertex Shader)的功能正是为了实现顶点的光照计算和坐标变换，在过去的硬件中，人们只能实现一些固定的光照模型和坐标系转换计算，但是在现在的可编程的图形硬件中，用户可以通过编写代码自由地设计自己所需要的光照模型和坐标系转换计算公式，只要不超出硬件的功能即可。在 1.2.1 节中，将专门对顶点着色器作进一步的介绍。

至于象素着色器(Pixel Shader)，功能相对比较复杂，同时限制较多。它几乎包括了所有光栅化以后的操作：采样纹理，颜色混合计算(Blending)和雾化效果计算等，传统图形硬件中每个顶点的纹理坐标值需要人工指定或者根据空间坐标来计算，而在象素着色器中，允许采用多种纹理访问方式；每个象素的绘制信息(例如，象素颜色值、纹理坐标等)均保存在寄存器中，象素着色器通过完成这些寄存器之间的加法，乘法或者点积等运算操作，从而在象素级别上实现光照明模型的计算(per-pixel lighting)[6-9]。

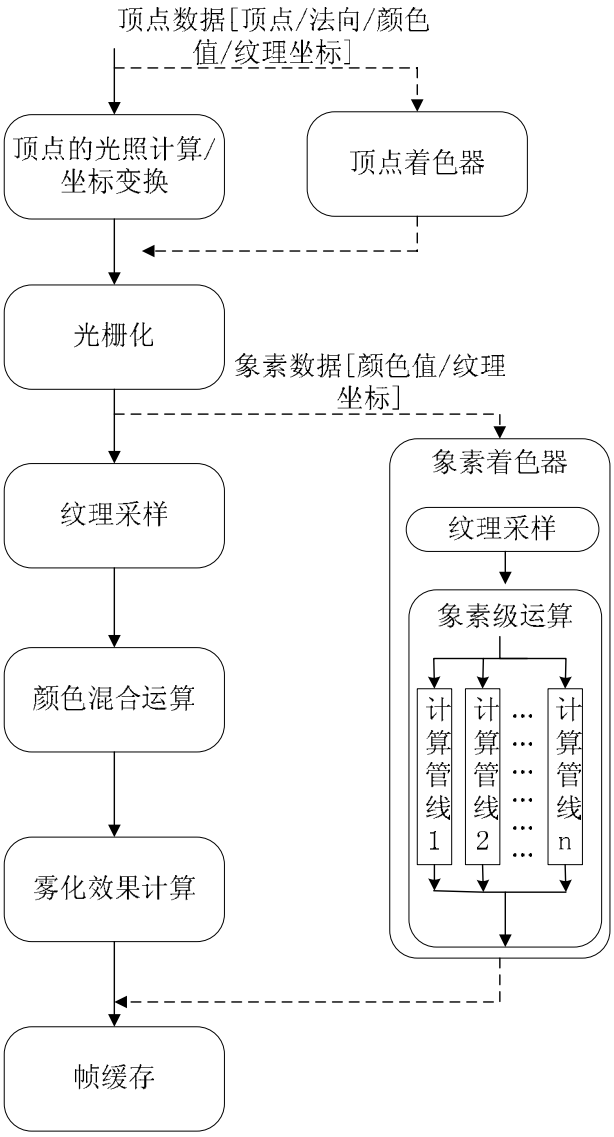


图 1.2 可编程图形流水线的总体框架

1.2.1 顶点着色器

下面以 2.0 版本的顶点着色器为例来分析其架构(如图 1.3 所示[3])。其中所有的寄存器长度都为 4×4 字节，即 RGBA 四个值分别用 32 位的浮点数(即 4 个字节)来存储。输入的 16 个寄存器通常用来保存顶点的绘制信息，包括顶点位置、法向或者颜色值，也可以是由用户自己定义的信息，例如:权重、位移或者速度等等。至于输出的 15 个寄存器通常用来保存顶点着色器的运算结果，包括裁剪坐标系下的顶点位置、颜

色值，雾化坐标值和纹理坐标等等，同样输出寄存器也可以存放用户自己定义的绘制信息，比如：光照方向、视线方向等等，这些信息在三角形光栅化后经双线性插值生成象素级的绘制信息。此外，还有 96 个只读的常量数据寄存器以及 12 个可读写寄存器，其中只读的常量数据寄存器的赋值操作由 CPU 控制，也就是通过外界函数调用进行赋值，而可读写的寄存器则可在顶点绘制编程器的计算过程中作为临时寄存器使用[12-15]。

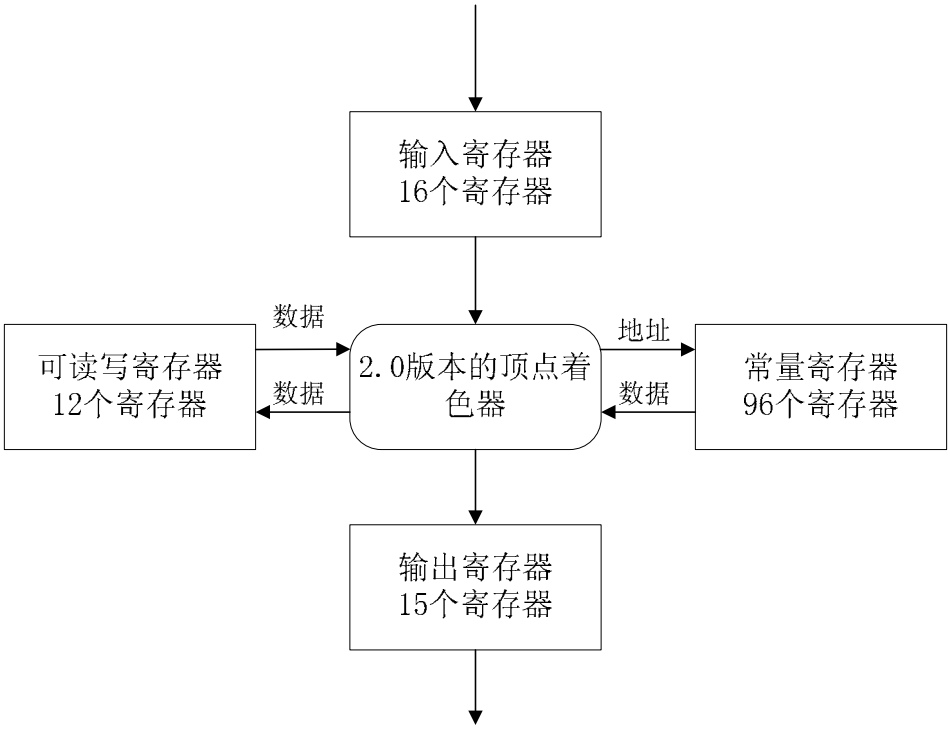


图 1.3 顶点着色器(2.0)系统框架

这些寄存器通常有固定的表达方式：输入寄存器为 v[n]，输出寄存器为 o[n]，常量寄存器为 c[n]，而可读写寄存器则为 Rn，这里的 n 就是具体的寄存器编号。如第十号的输入寄存器就写作 V[10]。而且每个寄存器都可以用 xyzw 后缀名来访问寄存器中的 RGBA 各个分量，比如说:R1.w 表示的就是第 1 号可读写寄存器的第四个分量－A 分 量。

寄存器为 Shader 程序的执行提供了变量空间，但程序功能实现最核心的还是指令集。不同版本的着色器对于指令长度的限制是不一样的，早期的顶点着色器只支持 128 条指令长度内的程序，也就是说，顶点着色器对于程序的复杂度是有限制的，2.0

版本的顶点着色器支持的程序指令长度已经达到 256 条。

顶点着色器的指令集中包含的通用运算指令有：加法(ADD)、乘法(MUL)、先加后乘(MAD)、三维的点积运算(DP3)、四维的点积运算(DP4)、求倒数(RCP)、求平方根的倒数(RSQ)、指数运算(EXP)和对数运算(LOG)，求最大值(MAX)，求最小值(MIN)等；还有寻址指令(ARL)，移动数据指令(MOV)，判断指令：判断是否小于(SLT)，判断是否大于等于(SGE)，以及两个专门为了实现光照计算公式的指令：一个是根据光源距离计算衰减系数(DST)，另一个是实现 Phong 光照模型公式的计算(LIT)等。具有丰富功能的指令集为顶点处理绘制信息的计算提供了强大的支持 [16-19]。

有了顶点着色器，就拥有了一个强大的顶点级绘制信息的计算工具。但是，由于颜色计算只是在三角形的顶点上进行，三角形内部各个象素的颜色只能通过插值得到，这样将无法充分表现三角形内部的细节。下面要介绍的象素着色器则可以弥补这点不足，其操作对象是每个象素，可以在象素级别上进行绘制，从而充分表现三角形的内部细节 [20-22]。

1.2.2 象素着色器

如图 1.2 所示，象素着色器的功能主要体现为两个部分：纹理采样和象素级运算 [23-26]。其具体结构图如下：

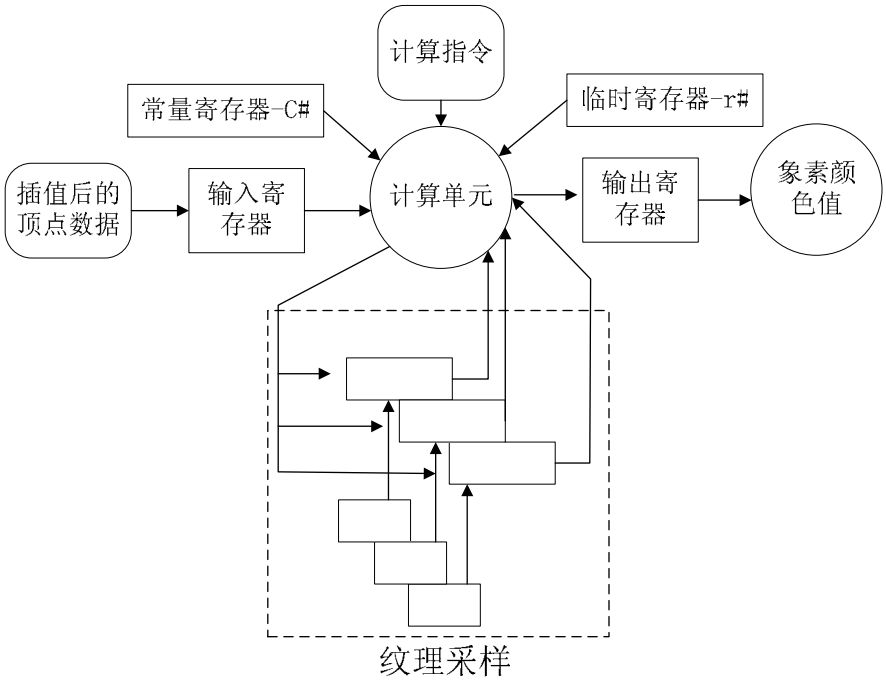


图 1.4 象素着色器结构图

1.2.2.1 纹理采样

前面已经提到纹理采样就是通过不同的纹理访问方式，由每个象素的纹理坐标得到纹理颜色值。其中纹理坐标值(s_i,t_i,r_i,q_i)和颜色值(R_i,G_i,B_i,A_i)均可以是 32 位的浮点值。请注意如图中虚线所示，通过纹理采样得到的颜色值同样可以再次作为纹理坐标进行采样，这就是前面提到的依赖性纹理访问功能，该功能增加了对于象素处理的灵活性，为把象素处理的中间结果保存在纹理中打下了基础[27-30]。

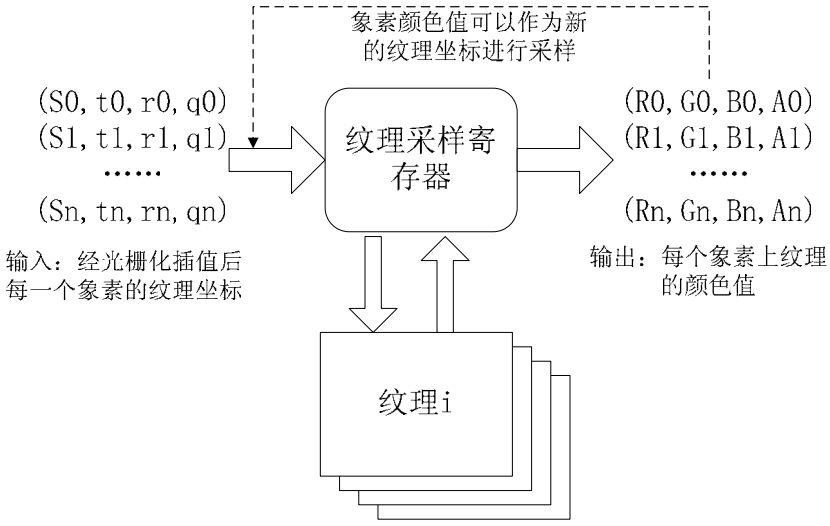


图 1.5 纹理采样的功能示意图

1.2.2.2 象素级运算

象素着色器提供了象素级上的加法，乘法以及点积等运算功能，丰富的运算功能可支持象素级上的各种绘制运算。目前 GPU 大都提供了可以并行计算的 8 条甚至 16 条象素着色器计算管线，在每一计算管线上都可以对取自多个输入寄存器的数据进行基于指令集的各种运算，象素着色器的指令集和顶点着色器一样提供了丰富的计算功能，此处就不再赘述，可参见具体的开发文档。图 1.5 以其中任意一条计算管线为例简单图示其结构。

图中计算管线 1，2…n 的功能和结构是一样的，以管线 1 为例，首先选择 4 个输入寄存器，其数据分两组经过输入映射后进行乘法或者点积运算，得到中间结果，中间结果可以直接由输出映射传送到输出寄存器中，也可以作为下一次加法或者乘法运

算的输入；最终输出寄存器中的结果值将送入帧缓存中[33-35]。

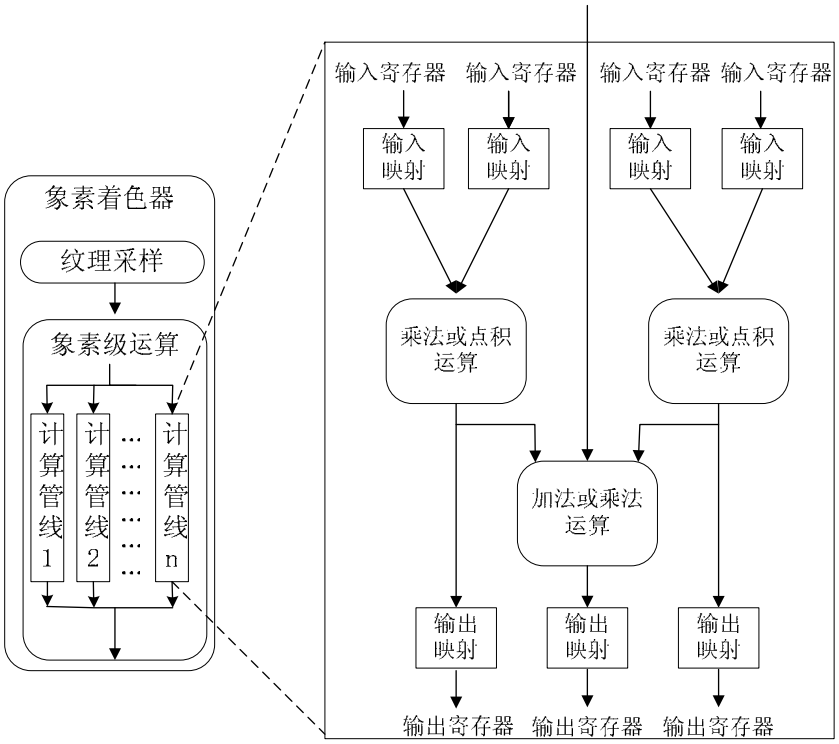


图 1.5 象素计算管线结构图

1.3 可编程图形流水线的软件开发技术

在可编程图形硬件迅猛发展的同时，其相关的软件开发技术也在不断进步。目前 GPU 的应用途径主要是通过图形 API(OpenGL 或者 Direct3D)扩充新 GPU 的功能，其扩充功能或由 GPU 厂家提供，或由 API 软件开发者提供[39]。利用高级语言不依赖于具体的 GPU 硬件及计算平台的编程一直是 GPU 应用的努力目标，朝此目标而在绘制语言(shading language)和实时绘制语言方面所作的研究一直在进行之中。运用 GPU 进行通用计算最终需要摆脱图形流水线及其绘制的概念，从流处理机的角度利用高级语言编程，这方面的研究工作在 2004 年开始有了新的进展[40]。

GPU 所提供的可编程功能以顶点处理器和像素处理器的操作形式完成，由每个处理器执行用户定义的汇编级的绘制程序(shader program)[10]，对流数据(顶点/像素)执行绘制程序的操作。对于标准图形界面用户来说，这些 GPU 的新功能一般通过 GPU 设计者或用户界面的设计者以扩充函数库的形式提供给用户。

OpenGL 作为事实上的工业标准已为学术界和工业界所普遍接受，因而绝大部分与图形有关的应用产品一直以 OpenGL 作为实现界面。对于使用 GPU 的扩充函数，OpenGL 中包括了 GPU 设计者(如 Nvidia)以及 OpenGL“架构委员会(ARB)”所扩充的函数。Direct3D 作为微软视窗的标准，其图形界面 Direct3D 从新世纪开始亦得到广泛接受和应用，特别是作为游戏软件的实现界面。

为适应 GPU 应用的需求，Direct3D 根据 GPU 新产品功能的扩充与进展及时地定义新的版本以扩充 Vertex Shader 和 Pixel Shader 的新功能，Direct3D 软件接口所提供的功能几乎与 GPU 提供的功能同步。对于熟悉 OpenGL 或 Direct3D 的用户或软件专业用户来说，直接使用其扩充的接口软件应该是比较好的选择，因为这里可以从底层实现更灵活的控制和对 GPU 编程。

1.3.1 高级绘制语言及实时绘制语言

使用类似于 C 的高级语言对 GPU 直接编程一直是图形界努力追求的目标，也是将来应该达到的目标。在这方面，已取得了一些进展。使用绘制语言及高级语言进行绘制编程可以方便用户书写各种不同功能的绘制程序以及对其绘制程序提供各种控制，从而使 GPU 硬件的具体功能对于用户而言更具有透明性。绘制程序(shader)设计的思想源自于早年 Pixar 设计的 RenderMan[41-43] 绘制软件。此软件多年来广泛应用于好莱坞电影中特技效果的绘制。关于新型标准绘制语言的研究工作，近年来具有较大影响的是：OpenGL shading language[44,45]，斯坦福大学的 RTSL (real-time shading language)[46,47]，Microsoft 的 HLSL(high-level shading language)[48] 以及 Nvidia 的 Cg[49]。尽管还未形成统一的绘制语言，这些语言的研究和应用为用户提供了直接基于 API (OpenGL 或 Direct3D)编程的较为方便和高层次的工具。

1.3.2 流处理机编程环境及工具 [50]

然而，使用绘制语言编程仍然存在着不少缺陷，对使用者来说仍然是一件十分麻烦的工作。首先，用户必须编写控制图形流水线的许多任务，如分配纹理存储、读入绘制程序、构造图形元素等，为此，用户对最新的 API 以及图形处理器硬件的特点与限制需要有详细的了解 [51-54]。此外，用户仍然需要利用纹理、三角形等图形素元表达他们的算法，这就使得 GPU 的通用计算编程工作仍然只能由资深的图形开发者进行。如果这一问题不能很好地解决，GPU 用于通用计算的前景将会受到很大的限制 [55,56]。因为，熟悉 GPU 图形编程的人毕竟太少，而这些资深的图形开发者对通用计算的各个应用领域又比较陌生。为解决这一问题，当前的努力方向主要是将 GPU 的结构纯粹纳入流处理机的模型而以高级语言编程，使得程序说明、运算操作、模块化定义等

一系列运算和控制规范化，使用户在实现高效率利用 GPU 的同时，不必考虑 GPU 的具体图形结构。这一研究工作的代表是最近发表于 ACM SIGGRAPH2004 的斯坦福大学的 Brook-for-GPU 系统[57]和加拿大 Waterloo 大学的 Shader Algebra 系统[58]。

1.4 本文工作的主要研究工作介绍及章节安排

本文以可编程图形硬件技术作为文章的主线，介绍了作者硕士生学习期间的主要工作。

第一章、 可编程图形硬件技术的综述。介绍可编程硬件相关的软硬件知识和可编程图形流水线的工作原理及架构。

第二章、 实时体素化。介绍了一种基于可编程图形硬件技术实现的实时体素化算法，该算法完全在 GPU 中运行，对于两百万面片以上的复杂几何模型仍然可以达到实时，该算法便于实现且具有很好的可扩展性，在体建模，透明绘制和碰撞检测等很多领域中都有重要的应用前景。

第三章、 高质量大尺寸点模型实时绘制算法。介绍了一个针对大规模点模型(数百万级)的自适应实时高质量绘制算法，该算法根据视点与模型的距离远近和视角大小，对具有较好绘制效果的 EWA 滤波反走样算法采用不同的近似方式，为模型选择合适的细节层次，对模型进行自适应绘制。通过一种面向保留模式图形硬件加速的点模型压缩和解压缩算法，该算法可以在可编程图形硬件中实现，在普通微机上实时高质量绘制了千万数量级的点模型。

第四章、 实时阴影映射。介绍阴影映射的原理以及如何基于可编程图形流水线实现实时阴影映射，着重介绍了一种基于完全硬件功能实现的硬件阴影映射方法。

第五章、 结论与展望。总结现有工作，简单介绍了未来可能的研究方向。

参考文献

[1] Macedonia M. The GPU enters computing’s mainstream. IEEE Computer, 2003,36(10): 106-108.

[2] Hopgood FRA, Duce DA, Gallop JR, Sutcliffe DC. Introduction to the Graphics Kernel System

(GKS). Academic Press, 1983.
[3] Enderle G, Kansy K, Pfaff G. Computer Graphics Programming: GKS-The Graphics Standard. Berlin: Springer-Verlag, 1984.
[4] Howard TLJ, Hewitt WT, Hubbold RJ, Wyrwas KM. A Practical Introduction to PHIGS and PHIGS Plus. Addition-Wesley, 1991.
[5] clark JH. The geometry engine: A VLSI geometry system for graphics. In: Proc. Of the SIGGRAPH'82. 1982. 127-133.
[6] Fuchs H, Poulton J. Pixel-Planes: A VLSI-oriented design for a raster graphics engine. VLSI Design, 1981, 2(3): 20-28.
[7] Eyles J, Austin J, Fuchs H, Greer T, Poulton J. Pixel-Plane 4: A summary, advances in computer graphics hardware II. In: Eurographics Seminars Tutorials and Perspectives in Computer Graphics. 1988. 183-208.
[8] Fuchs H, Israel L, Poulton J, Eyles J, Greer T, Goldfeather J, Ellsworth D, Molnar S, Turk G, Tebbs B. Pixel-Planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. In: Proc. of the SIGGRAPH'89. 1989. 79-88.
[9] Molnar S, Eyles J, Poulton J, Greer T. PixelFlow: High-Speed rendering using image composition. In: Proc. of the SIGGRAPH'92. ACM Press, 1992. 231-240.
[10] Lindholm E, Kilgard MJ, Moreton H. A user-programable vertex engine. In: Proc. of the SIGGRAPH 2001. Los Angeles, 2001. 149-158.
[11] Owens JD, Dally WJ, Kapasi UJ, Rixner S, Mattson P, Mowery B. Polygon redering on a stream architecture. In: Proc. of the Eurographics/SIGGRAPH Workshop on Graphics Hardware. 2000. 23-32.
[12] 吴恩华，柳有权，基于图形处理器(GPU)的通用计算. 计算机辅助设计与图形学学报，2004，16(5):601-612.
[13] govindaraju NK, Sud A, Yoon SE, Manocha D. SWITCH: Parallel occlusion culling for interactive walkthroughs using multiple GPUs. Technical Report, TR02-027, UNC-CH, 2002.
[14] Govindaraju NK, Redon S, Lin M, Manocha D. CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. In: Proc. of the Eurographics/SIGGRAPH Workshop on Graphics Hardware. 2003. 25-32.
[15] Sud A, Otaduy MA, Manocha D. DiFi: Fast 3D distance field computation using graphics hardware. In: Proc. of the Eurographics, 2004.
[16] Tomov S, McGuigan M, Bennett R, Smith G, Spiletic J. Benchmarking and implementation of probability-based simulations on programmable graphics cards. Computer & Graphics, 2005, 29(1).
[17] Larsen ES, Mcallister D. Fast matrix multiplies using graphics hardware. In: Proc. of the Supercomputing. 2001. 55-60.
[18] Thompson CJ, Hahn S, Oskin M. Using modern graphics architectures for general-purpose computing: A framework and analysis. In: Proc. of the Int'l Oymp. On Microarchitecture. 2002. 306-317.
[19] Krüger J, Westermann R. Linear algebra operators for GPU implementation of numerical algorithms. ACM Trans. On Graphics, 2003, 22(3): 908-916.
[20] Hall JD, Carr NA, Hart JC. Cache and bandwidth aware matrix multiplication on the GPU. UIUCDCS-R-2003-2328, Champaign: University of Illinois at Urbana-Champaign, 2003.
[21] Rumpf M, Strzodka R. Using graphics cards for quantized FEM computations. In: Proc. of the VIIP 2001. 2001. 98-107.

[22] Harris MJ, Coombe G, Scheuermann T, Lastra A. Physically-Based visual simulation on graphics hardware. In: Proc. of the Graphics Hardware 2002. 2002. 109-118.
[23] Li W, Wei XM, Kaufman A. Implementing lattice Boltzmann computation on graphics hardware. The Visual Computer, 2003, 19(7-8): 444-456.
[24] Bolz J, Farmer I, Grinspun E, Schröoder P. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. ACM Trans. on Graphics, 2003, 22(3): 917-924.
[25] Goodnight N, Woolley C, Luebke D, Humphreys G. A multigrid solver for boundary balue problems using programmable graphics hardware. In: Proc. of the Graphics Hardware 2003. 2003. 102-111.
[26] Harris MJ, Baxter W-V III, Scheuermann T, Lastra A. Simulation of cloud dynamics on graphics hardware. In: Proc. of the Graphics Hardware 2003. 2003. 92-101.
[27] Li W, Fan Z, Wei XM, Kaufman A. GPU-Based flow simulation with complex boundaries. Technical Report, 031105, Computer Science Department, SUNY at Stony Brook, 2003.
[28] Kim T, Lin MC. Visual simulation of ice crystal growth. In: Proc. of the SIGGRAPH/Eurographics Symp. on Computer Animation. 2003. 86-97.
[29] Lefohn AE, Kniss JM, Hansen CD, Whitaker RT. Interactive deformation and visualization of level set surfaces using graphics Hardware. In: IEEE visualization. 20003. 75-82.
[30] Stam J. Stable fluids. In: Proc. of the SIGGRAPH'99. 1999. 121-128.
[31] Wu EH, Liu YQ, Liu XH. An improved study of real-time fluid simulation on GPU. Journal of Computer Animation & Virtual World (invited paper of CASA2004), 2004, 15(3-4): 139-146.
[32] Liu YQ, Wu EH, Liu XH. Real-Time 3D fluid dimulation on GPU with complex obstacles. In: Proc. of the Pacific Graphics 2004. 2004.
[33] Liu YQ, Wu EH, Liu XH. Fluid simulations on GPU with complex boundary conditions. In: ACM Workshop on General-Purpose Computing on Graphics Processors (GP2). 2004.
[34] Govindaraju NK, Lloyd B, Wang W, Lin M, Manocha D. Fast computation of database operations using graphics processors. In: Proc. of the SIGMOD. 2004.
[35] http://lava.cs.virgina.edu/bpred.html
[36] Moreland K, Angel A. The FFT on a GPU. In: Proc. of the Graphics Hardware 2003. 2003.
[37] Strang G, Nguyen T. Wavelets and Filter Bands. California: Wellesley-Cambridge Press, 1996.
[38] Hopf M, Ertl T. Hardware accelerated wavelet transformations. In: Proc. of the EG/IEEE TCVG Symp. on Visualization 2000. 2000. 93-103.
[39] Wang JQ, Wong TT, Heng PA, Leung CS. Discrete wavelet transform on GPU. In: ACM Workshop on General-Purpose Computing on Graphics Processors (GP2). 2004.
[40] Wang JQ. Exploiting the GPU power for intensive geometric and imaging data computation [MPhil Thesis]. Chinese University of Hong Kong, 2004.
[41] Hanrahan P, Lawson J. A language for shading and lighting calculations. Computer Graphics, 1990, 24(4): 289-298.
[42] Upstill S. The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics. Addison-Wesley, 1990.
[43] Apodaca AA, Gritz L. Advanced RenderMan: Creating CGI for Motion Pictures. Morgan Kaufmann Publishers, 2000.
[44] Kessenich JD, Baldwin RR. OpenGL 2.0 Shading Language. 1.05I Ed, 2003.
[45] Rost RJ. OpenGL Shading Language. Addison-Wesley, 2004.
[46] Proudfoot K, Mark WR, Hanrahan P, Tzvetkov S. A real time procedural system for programmable

graphics hardware. Computer Graphics, 2000. 159-170.
[47] Mark WR, Proudfoot K. Compiling to a VLIW fragment pipeline. In: Proc. of the Graphics Hardware 2001. 2001.
[48] Peeper C, Mitchell JL. Introduction to the DirectX 9 High-Level Shader Language. 2003. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnhlsl/html/shaderx2_introductionto.asp
[49] Mark WR, Glanville S, Akeley K, Kilgard MJ. Cg: A system for programming graphics hardware in a C-like language. ACM Trans. on Graphics, 2003, 22(3): 896-907.
[50] 吴恩华，图形处理器用于通用计算的技术、现状及其挑战. 软件学报 2004, 15(10): 1493-1504.
[51] Dally WJ, Hanrahan P, Erez M, Knight TJ, Labont F, Ahn JH, Jayasena N, Kapasi UJ, Das A, Gummaraju J, Buck I. Merrimac: Supercomputing with streams. In: Proc. of the SC 2003. ACM Press, 2003.
[52] Talor MB, Kim J, Miller J, Wentzlaff D, Ghodrat F, Greenwald B, Hoffmann H, Johnson P, Lee JW, Lee W, Ma A, Saraf A, Seneski M, Shnidman N, Strumpen V, Frank M, Amarasinghe S, Agarwal A. The raw microprocessor: A computational fabric for software circuits and general purpose programs. IEEE Micro, 2002.
[53] Wu EH. Challenge of highly detail object modeling and illumination calculation in virtual reality (Keynote Speech). In: Proc. of the ACM/VRST2002. Hong Kong, 2002.
[54] 吴恩华. 复杂虚拟场景的造型与绘制(大会特邀报告). 见：第 3 届全国虚拟现实与可视化学术会议(CCVRV 2003). 长沙：国防科学技术大学，2003。
[55] Fan Z, Qiu F, Kaufman A, Yoakum-Stover S. GPU cluster for high performance computing. In: Proc. of the ACM/IEEE SC2004 Conf. 2004.
[56] Gulde R, Weeks M, Owen S, Pan Y. Parallel computing with multiple GPUs on a single machine to achieve performance gains. In: ACM GP2: Workshop on General Purpose Computing on Graphics Processors. 2004.
[57]Buck I, Foley T, Horn D, Sugerman J, Fatahalian K, Houston M, Hanrahan P. Brook for GPUs: Stream computing on graphics hardware. ACM Trans. on Graphics, 2004, 23(3): 777-786.
[58] McCool M, Toit SD, Popa T, Chan B, Moule K. Shader algebra. ACM Trans. on Graphics, 2004, 23(3): 787-795.

第二章 实时体素化及其应用

体素(voxel = volume + element)是体图形学(volume graphics)中描述体模型的基本数据单元。相对于传统的三角面片表示，以体素的方式来描述一个几何物体具有简单、稳定及不需要拓扑信息等特点，在进行 CSG 求交和碰撞检测等经典应用中具有明显优势。体素化(voxelization)指在保证精度的前提下，将由三角面片或其他边界表示组成的几何模型转化为离散的体素集合表示的过程。体素化的概念最先由 Arie Kaufman[1, 2] 于 1986 年提出，在此之上出现了多种优秀的研究成果，并已广泛应用于体模型建模[3]、虚拟医学[4]、混合体绘制[5]、几何模型可视化[6]、CSG 建模[7]、碰撞检测[8-10]等。

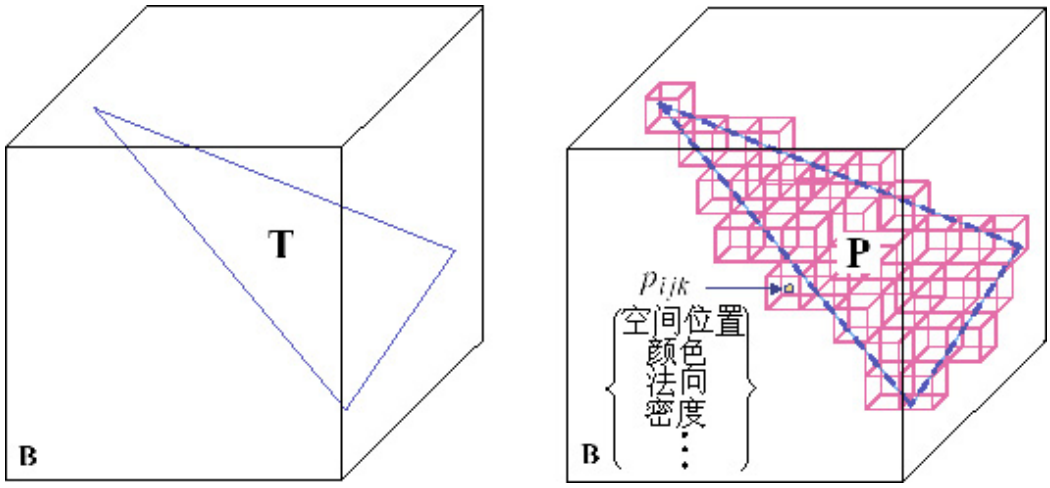


图2.1 三角面片体素化过程.

图2.1即为对一个三角面片**T**进行体素化生成其体模型**P**的过程。其中三角面片**T**由一系列包含位置、法向、颜色和纹理坐标等信息的顶点和表示三角形拓扑关系的索引信息构成，经体素化后，连续的空间结构转化为了离散的体模型**P**，其中每一个小立方体表示空间中一个体素如$P_{ijk}(0<i<L,0<j<M,0<k<N$，$L, M, N$ 分别为体模型**P**在X, Y, Z 三个方向上的分辨率)，每一个体素包含的属性有空间位置、密度、颜色、法向和纹理坐标等。

2.1 已有的体素化工作介绍

已有的体素化研究可以分为表面体素化 [11-13](surface voxelization)和实体体素化[14,15](solid voxelization)，前者将几何物体的外表面转化为离散的体素表示，后者则同时转化物体表面及其内部。另一种经典的体素化分类方法是根据生成体素所包含的信息分为二值体素化(binary voxelization)和多值体素化(non-binary voxelization)，前者所产生的每一个体素仅仅表示几何物体在空间中的有无，而后者除此之外还可能包含法向、纹理坐标和密度等很多其他信息，具有更广阔的应用范围。

上世纪九十年代以来的体素化相关工作主要集中于生成体模型过程中的稳定性和鲁棒性问题。1993年，S.Wang等人[16]提出将滤波器引入体素化过程。实现精确体建模的法向估计方法，此后大量的工作[12,13,17]围绕如何设计更高效的滤波器和改进法向估计方法等方面展开，力图实现更好的体绘制效果。2000年，Dachille和Kaufman[11]提出了一种高效的增量式三角面片体素化方法，可以快速完成三角面片模型的表面体素化。Haumont和Warzee[15]提出了一种基于三维种子填充方式实现的实体体素化方法。Widjaya等[18]在通用二维网格上的基础上，实现了包括六边形网格和三维立方体网格等网格模型的体素化。2003年，Varadhan等人[19]利用最大法向距离估算法来有效地判断几何物体表面是否与空间体素相交，有效地提高了体素化结果的精确性。

另一部分工作则主要关注体素化的效率，其中一个很有效的方法是在基于共享内存的多CPU体系结构硬件平台下，充分利用数据相关性原理实现高效的体素化[20]。体素化本质上是一个三维空间数据扫描转化的过程，人们很自然联想到利用光栅化图形硬件加速体素化，在这方面比较有代表性的工作主要有三类。基于切片的体素化算法[21]利用图形硬件在流水线中设定适当的裁剪平面，分割模型得到切片并存储在帧缓存(Frame buffer)中，所有产生的切片构成最终的体模型。此方法因充分利用了光栅化图形硬件，实现了可交互的体素化，后来被广泛应用于各种三维物体的体素化[7,22,23]以及三维空间分析[24]和碰撞检测[9]等应用。但是，基于切片的体素化算法的复杂度与最终体模型的分辨率成正比，因此体模型分辨率越高，体素化过程中所需要生成切片数越多，算法的效率越低。基于外表面投影算法[25]将几何物体的外表面依次投影到其空间包围盒的6个面，然后通过读取深度缓存(depth buffer)完成体素化的过程；这种方法的主要缺陷在于无法正确处理某些情况下一些奇异模型的凹凸问题，从而限制了它的应用范围。受到层次深度图像（LDI）[26]思想的启发，Heidelberger等人[27,28]提出了一种利用图形硬件生成层次深度图像而有效实现体素化的方法，此法虽然可以产生比较精确的结果，但是其性能同样受到场景复杂度和景深的限制，所使用的深度列表每一帧都需要从GPU中读出并在CPU中进行排序。

通过对已有的利用图形硬件的体素化算法进行分析，我们发现现有方法的三处不

足：

1. 已有方法在效率上受限于场景复杂度和体模型的最终分辨率，原因是在体素化过程中，输入的几何模型需要被遍历多次，遍历次数与场景复杂度和体模型分辨率成正比，极大地增加了体素化所需耗费的时间。

2. 已有方法需频繁访问帧缓冲器，这会加大内存和显存之间的带宽传输负担，影响图形硬件的工作效率。

3. 已有方法产生的体模型结果均直接存储在颜色缓冲器或者深度缓冲器中，耗费大量的显存。

根据综合文献报告，目前的体素化方法很难在中等体模型分辨率如256×256×256的条件下达到交互的帧率。为了解决以上存在的一系列问题，我们提出了一种全新的基于可编程图形硬件的实时体素化算法。

2.2 实时体素化算法原理

2.2.1 算法核心思想

在标准的光栅化图形硬件中，空间中三角面片被扫描转化生成二维图像存贮到帧缓冲器中，这个过程中依靠硬件提供 z-buffer 功能，只有距离视点最近的那部分三角面片对最终图像产生贡献，其后面的几何信息都被裁剪剔除了。然而，体素化是一个三维的扫描转化过程，其目标是得到一个离散的三维体模型，因此需要保存不仅是距离视点最近的，而是整个 Z 方向上的空间几何信息，由三维扫描转换生成的体素所组成的阵列即为需要的体模型。在图形流水线中，体素所组成的阵列可以由二维纹理(2D texture)或者三维纹理(3D texture)的数据形式来表示。由于目前直接对三维纹理进行赋值的操作还未得到 PC 平台图形硬件的广泛支持，我们选择使用二维纹理作为体素化过程中中间数据的载体和最终体模型结果的保存形式。我们称保存体素化过程中间数据的纹理为工作表格(worksheet)，它记录了构成最后体模型结果的一部分体素。通常，在纹理中的每一个纹元 (texel) 都包含 red, green, blue 和 alpha 四个颜色通道(component)，根据体模型中每个体素所需要的信息量的不同，一个纹元可以记录一个或者多个体素信息，如果我们仅需要表示一个体素在空间某一位置是否存在，那么仅需要用 1 个 bit (0 或 1) 即可表示一个体素，这样一个 8 bits 的 red 颜色通道就可以表示 8 个体素，而一个纹元则可以表示 32 个体素，这样的表示方法大大压缩了算法所需要的显存量，提高了效率。依靠以上方法，一张2048×2048的二维纹理即可包含生

成 512^3 分辨率的体模型数据信息。请注意，算法中使用的工作表格的长宽尺寸都很大（如 2048×2048 ）,因此我们通常会将其分割为很多数据块 (patch)，每一个数据块的长和宽都和最终生成的体模型在空间中两个轴向上的分辨率一致，对应于最终生成体模型中的一个切片 (slab)，如图 2.2 所示。当体模型的分辨率较高时，算法通常需要多个工作表格记录数据。

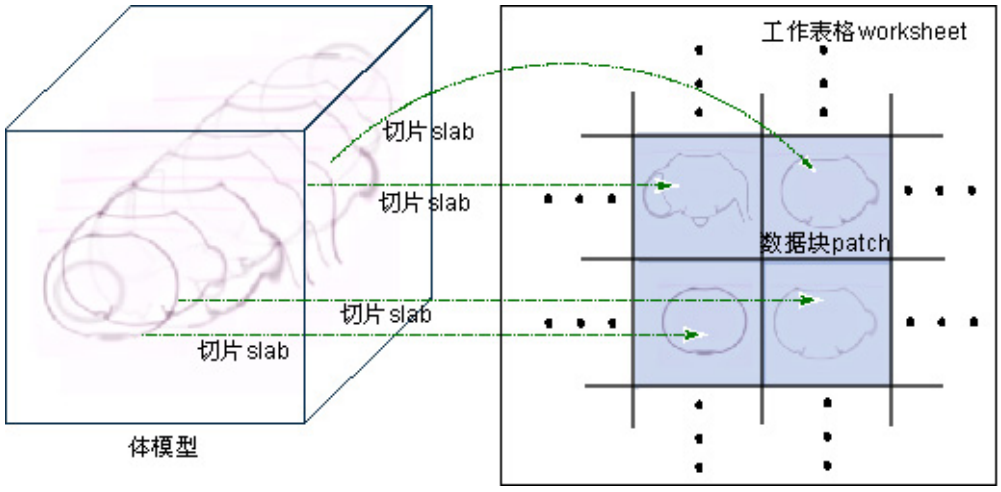


图2.2 工作表格和体模型切片的对应关系

三角面片转化为离散的体素表示的过程可以利用可编程图形硬件完成，在此过程中，工作表格以数据块的形式被填充，而体模型则以逐个切片的方式生成。对于每个切片，只有和它相交的三角面片才会被扫描转化，为了保证扫描转化的鲁棒性，每一个三角面片在转化时需要按照其自身投影面积的最大的轴向进行光栅投影。每个体素的三维空间坐标先被变换到体模型所在的三维空间，我们称为目标体空间，以便用来在纹理编码(texelization)时在 worksheet 中找到对应的存储位置。请注意这里的目标体空间只是一个算法抽象概念，在具体实现过程中算法使用 alpha 混合的方式实现将目标体空间中的信息编码到一张二维纹理之中。

由图形硬件完成的二维光栅化过程包含对相关几何顶点属性进行双线性插值得到每个象素(pixel)相应属性的过程。但当三角形平行于光栅化的方向时，其光栅化所得到象素将会构成一条线而不再是原来的三角形，这会导致体素信息的丢失，产生错误的体素化结果，因此在进行三角形的投影变换时，应该注意将每个三角形向其投影面积最大的轴向进行光栅投影。算法使用空间中三个正交轴向作为投影方向，分别进行光栅化和纹理编码，生成了三张代表不同投影方向工作表格 (directional sheet

buffers) 纹理，它们分别代表了最终生成体模型的一部分体素数据。三张纹理生成后算法通过一个合并的过程(synthesis)将其合成为一张包含全部体模型体素的工作表格。在合并过程中，每一个体素首先将其空间坐标变换到目标体空间中之后再进行一次从 3D 空间到 2D 空间的坐标变换，最终保存到工作表格中，如图 2.3 所示。

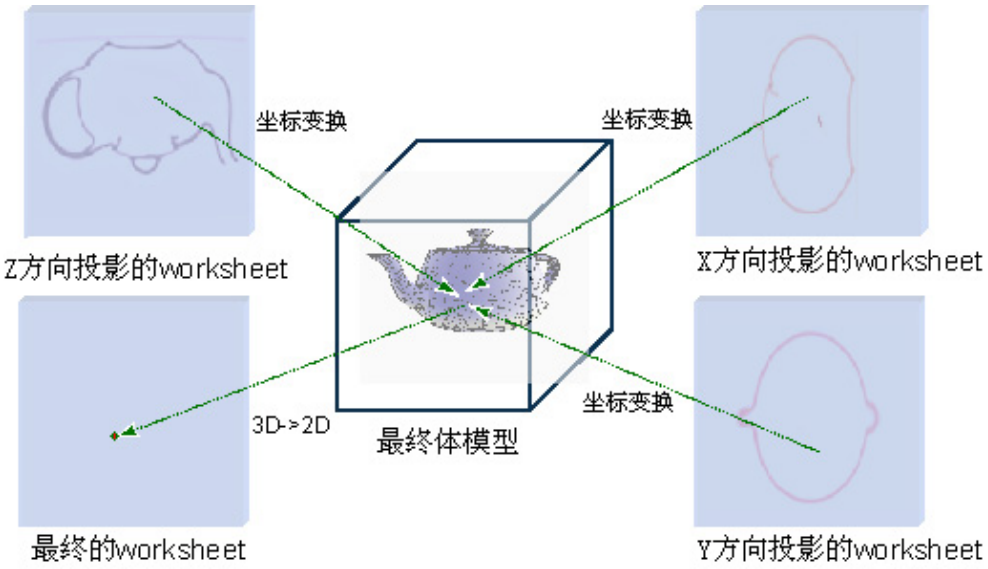


图2.3 三张代表不同投影方向的工作表格(worksheet)的合成

总体而言，实时体素化算法可以分为三个步骤：

1. 光栅化(rasterization)：三角面片通过光栅化变换到离散目标体空间中。
2. 纹理编码(texelization)：经过一系列空间坐标变换，每一个体素都被编码并通过 alpha 混合的方式存储到工作表格(worksheet)中。
3. 合成(synthesis)：三张代表不同投影方向工作表格合成为一张包含最终体模型数据的工作表格。

算法的核心流程图如图 2.4 所示：

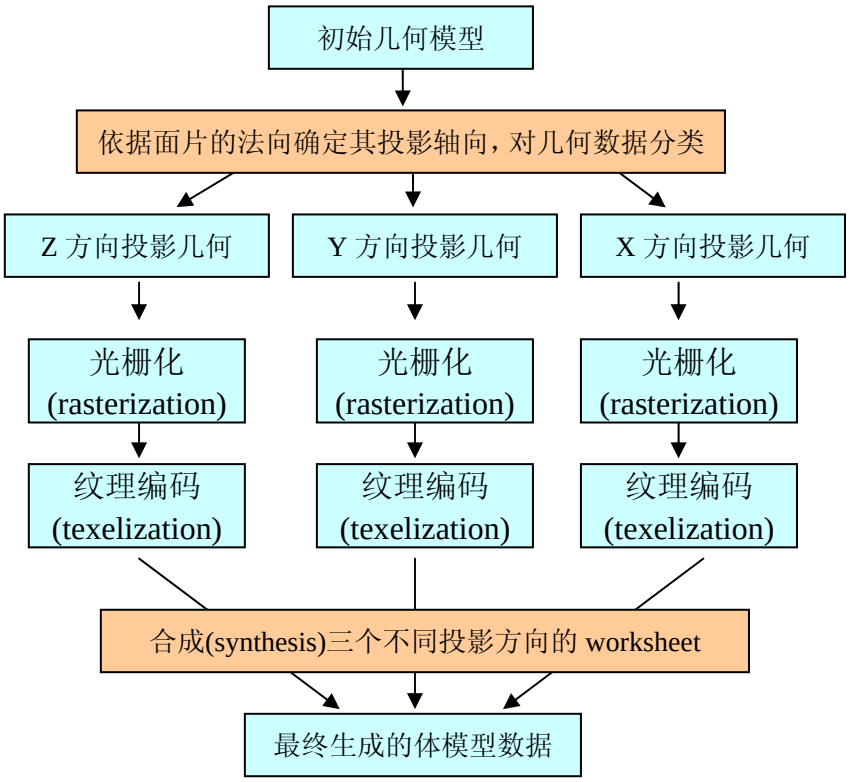


图 2.4 实时体素化算法流程示意图

2.2.2 体素编码和寻址

最简单的体素化结果是仅仅表示空间位置信息由二值体素化产生的体模型，每一个体素的值要么是空，要么是实。因此只需要 1 个 bit 即可以表示一个体素，而对于多值体素，则需要多个比特甚至字节才能表示一个体素，例如可以用多个 byte 来存储离散量化的法向量，纹理坐标或者颜色属性等多种信息。

假设目标体空间的分辨率为 $2^L \times 2^M \times 2^N$，工作表格的分辨率为 $2^W \times 2^H$，由数据量不变得到等式 $2^L \times 2^M \times 2^N = 2^W \times 2^H \times C$，其中 C 即为每一个纹元所包含的体素数目。如对于二值体素化 C 等于 32，因为每个纹元均包含 4 个 8bits 的颜色通道。而对于每张工作表格，其 X, Y 方向的数据块数目分别为 2^{W-L} 和 2^{H-M}。在光栅化过程中，每一个数据块都对应于最终体模型的一个切片并按顺序存储在工作表格中。当需要绘制体模型时，需要首先创建一系列简单的代理几何模型（如平面），而后数据块会被作为纹理被采样以生成连续的层次矩形空间，从而得到正确的体绘制结果。

由于一个纹元可能包含多个体素，因此需要有一种在工作表格中高效地存储和读取体素的寻址方法，以提高体素化的整体效率。算法设计了如下寻址方式：令体素的体空间坐标为 (px, py, pz)，首先通过用 pz 除上 C 得到当前体素应该位于哪个数据块，然后找到该数据块中 (px, py) 处的纹元，接下来再次使用 pz 和 C 得到体素在该纹元中 bit 位移，最后利用已知信息使用一张查找表准确定位该体素存储或者读取的位置。

2.2.3 实体体素化

实时表面体素化算法同样适用于实体体素化。为实现实体体素化，算法采用了一种类似于二维扫描线填充的三维扫描线方法来获得几何体表面内部的体素信息。算法在体素化过程中，在每次生成每个体模型一个切片时，会逐条扫描线遍历该切片，对于每条扫描线从左至右逐个处理该线上的所有体素，在此过程中每个体素有一个标记值用于表示该体素是否在几何模型的内部，标记值的初值设为 false，在当前扫描线上遇到一个和几何模型外表面相交的体素时，标记值开始变为 true，表示进入模型内部，当再次遇到一个和表面相交的体素时再次恢复为 false，表示离开模型内部。其处理非常类似于大家熟悉的二维扫描线方法，算法同样取三个正交的轴向作为投影方向生成三张工作表格，再合并生成最终的实体体素化结果。

2.3 图形硬件实现细节

可编程图形硬件技术使得我们可以完全在 GPU 中实现体素化算法，实现了实时体素化的目标。在具体实现过程中，运用了大量旨在提高算法效率的图形硬件特性和技巧。

(1) 大尺寸的纹理：算法使用的纹理尺寸为 2048×2048，这样大尺寸的纹理可以为中等分辨率的目标体模型提供足够的数据存储空间，可以减少整个算法过程中工作表格的数目，减少中间过程耗费的时间。纹理所能使用的最大尺寸在不同的图形硬件条件下是不一样的，我们在实现过程中使用了 ATI 系列显卡，它所能支持的最大纹理尺寸为 2048×2048，但 Nvidia 系列[29]显卡所能支持的最大纹理尺寸可达到 4096×4096。

(2) 动态顶点缓存和索引缓存：是指将几何数据存储在 AGP 缓存中，在每一帧可以对其数据进行动态更新并绘制，这种方式比原来使用静态顶点缓存和索引缓存时每帧更改数据的代价要小很多，特别适合于动态变形物体的绘制，可以极大地提高绘制

的效率。

(3) 多个渲染目标(render target)：目前最新的 GPU 已经可以同时生成 4 个 Render target，也就意味着程序运行过程中可以同时生成 4 张工作表格或者 4 个数据块，这会大量节省绘制时状态设置等的时间消耗，提高效率。

(4) 依赖性地纹理采样：这项技术允许在 Shader 代码中，采样一张纹理得到的纹元数据可以作为纹理坐标去采样一张新的纹理得到新的纹元。Shader 2.0 对这项技术有一定的限制，要求这种依赖关系不得超过 3 层以上，但最新的 Shader 3.0 对此技术已经没有了任何限制。

2.3.1 动态更新索引缓存

前面分析实时体素化算法的时候已经提到，初始的三角面片需要依照其投影面积最大的方向分别向三个正交的轴向投影。一种耗费时间的方法是每一帧都遍历整个几何模型 3 次，每次都在 Shader 代码中通过比较顶点的法向选择当前投影方向的三角面片光栅化结果并编码到当前投影方向的工作表格中；这样带来的最直接问题就是几何数据处理量加大，增加了 Vertex Shader 的负担。有一种补救的办法就是绘制之前，预处理几何模型数据，依据表面法向将其分为三部分，然后每次绘制送入一部分几何数据；这个办法对于静态的模型是可行的，因为静态模型的几何数据是不会改变的，但如果模型是动态变形的，则这样的预处理会因为效率过低导致算法不能实时运行，这时我们需要动态索引缓存更新技术，GPU 将索引数据存放在 AGP 缓存内进行动态更新，每一帧都可以高效完成所需的预处理，AGP 8x 的高速传输速率保证了动态更新的实时性，而且使得几何模型在每一次体素化过程中在光栅化阶段只遍历一遍。

更进一步，我们还可以把每个投影方向的几何数据分成更小的数据组，以减少在生成体模型的切片时需要遍历所有的几何数据。我们通过设定裁剪平面的方法达到生成更小的数据组的目的，借助同时生成多个 render target 的技术，一次生成多个目标体空间的切片，从而加快速度。

2.3.2 查找表纹理

在算法实现过程中，大量的 Shader 计算都被简化为对一些预处理好的纹理的查找，以减少 Shader 计算的指令数目从而提高效率，我们统称这些纹理为查找表。以下会按照不同的功能分别介绍：

(1) 读取一个二值化体素

即读取一个 bit 的数据，为了得到一个 8bits 颜色通道中第 k 个 bit 的值，我们创建了一张 256×8 大小的查找表，表中纹元填充 0 或者 1，查找表中 (s,t) 位置的纹元表示某个能用一个 byte 表示的数字 s 的第 t 位是 0 还是 1。在具体实现时可以通过简单几个采样操作完成取得一个二值化体素。

(2) 存储一个二值化体素：

即存一个 bit 的数据，此时创建一张 8×1 的查找表，该表中第 s 个纹元保存数值 2^s。这样通过将 alpha 混合的操作方式设成相加，源/目标混合因子设为 1/1，则所需要的位的值可以被正确地存储。

(3) 合并工作表格：

在体素化的合并阶段(synthesis)，三个不同投影方向的工作表格应该合并成为一张包含最终体模型数据的工作表格。这个过程中包含两个坐标变换过程，如图 2.3 所示，其中一个过程是从不同二维工作表格坐标空间变换到目标体空间，另一个过程则是从三维目标体空间再次变换到最终合并生成的工作表格二维空间，完成纹理编码存储。这些坐标变换在 Shader 中都会耗费大量的运算指令，我们创建了多张查找表以简化运算。我们取 Z 方向投影的工作表格作为最终合并的目标空间，创建了两张查找表用来分别将 X 方向工作表格空间和 Y 方向工作表格空间中的每一个位置映射到 Z 方向工作表格二维空间中正确的对应位置；实际上，工作表格纹理中的每一个纹元往往对应通常对应多个在空间位置上相邻连续的体素，因此在 Shader 中可以通过偏移已变换得到的体素位置来实现连续多个体素位置的快速映射变换，这样会大量减轻 Pixel Shader 的负担，明显提高算法的效率。

2.3.3 结果体模型的二维纹理表示

经过体素化后最终生成的体模型是以工作表格的形式存放在显存中的，所以对它的使用如同对于一张二维纹理的操作。例如体模型的绘制非常方便地如 2.2.2 中已叙述的通过采样纹理的方式完成，如果同时体素化多个几何物体，可以保存多张包含体模型数据的纹理在显存中，这样可以方便地实现布尔运算等操作。当然这些纹理也可以读回内存中并保存到磁盘上，需要注意的是目前 AGP 总线结构属于不对称结构，即它的下行速率非常快，所以向 GPU 发送几何数据时速度不是问题，但是它的上行

速率比较慢，我们在 ATI 9800 上测试从显存中读回一张512×512的二维纹理所需要的时间大约为 75ms，这样会极大地降低整个程序的运行速度。目前最新的图形硬件技术提供了一种上下行速度对称的 PCI-Express 总线，已经可以解决回读速度慢的问题。

2.3.4 负载平衡

在可编程图形硬件环境中对于纹理的操作是很方便的，所以实时体素化算法可以扩展支持三角面片以外别的输入几何数据的体素化，只需要对其流水线稍作修改还可以支持很多其他的体图形操作。光栅化，纹理编码，合成三个步骤组成了一个可以灵活配置的体素化引擎。

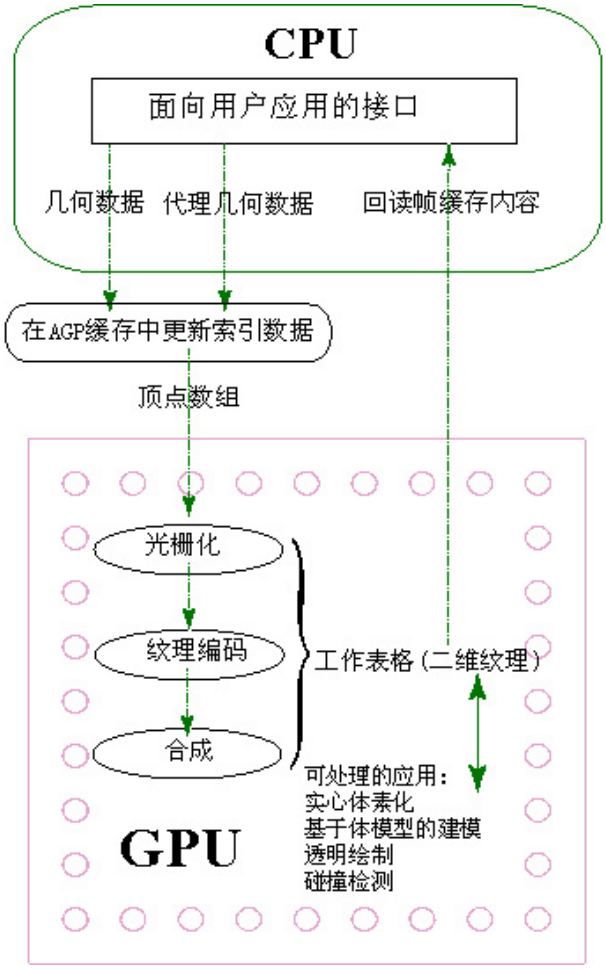


图 2.5 体素化引擎

该引擎前端接受输入的几何数据，并将其转化为准备进入 Vertex Shader 的顶点数

组，很多情况下，用户会只对几何数据的一部分感兴趣，此时可以通过 CPU 动态修改 AGP 缓存中的索引数据，接着 CPU 利用输入顶点的法向为每个顶点选择正确的投影轴向，此后几何数据便由 AGP 总线传入 GPU 中，接下来便是光栅化，纹理编码，合成等步骤的执行完成体素化。如果需要绘制生成的体模型，则需要创建一系列简单的代理几何并对包含体模型数据的工作表格纹理进行采样，从而完成绘制。通过以上分析可以看出，体素化引擎需要在 CPU 和 GPU 之间进行数据交换，实现两者工作负载的平衡是提升性能的关键之处。图 2.4 表示了体素化引擎的工作模块和流程。

2.4 算法扩展与应用

2.4.1 其他输入形式数据的体素化

(1) 隐式曲面：

对于隐式曲面的体素化，算法可以创建一系列的简单矩形代理几何并从前至后地处理，我们将每个矩形都光栅化为 render target，render target 中的每个象素则代表了三维空间中对于该曲面的一个采样点，可以使用该象素的空间坐标结合隐式曲面的表达式来判断它所代表的采样点是在曲面的外部，内部还是刚好在曲面上，接着我们便可以将满足要求的采样点纹理编码到一张工作表格中得到该曲面体素化的结果。因为可以依据曲面表达式判断象素点是否在曲面内部，不论是表面体素化还是实体体素化对于隐式曲面都不会存在问题。下图为对隐式曲面 $x^4-5x^2+y^4-5y^2+z^4-5z^2+11.8=0$ 分别进行表面体素化和实体体素化的结果，两者的体素化时间均为 25ms。

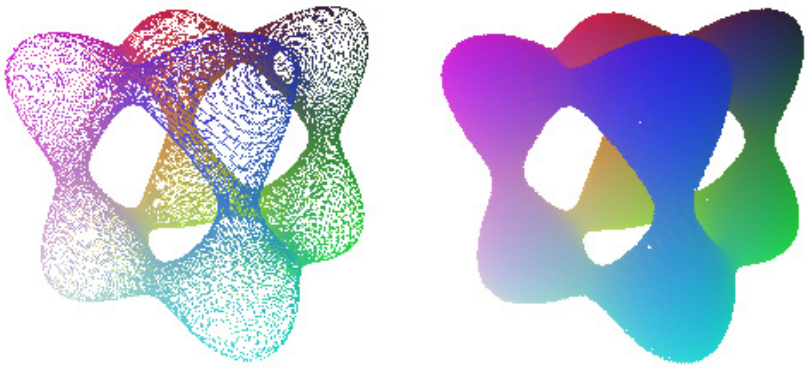


图 2.6 隐式曲面体素化结果

(2) 参数曲面：

体素化参数曲面需要在 Vertex Shader 中进行处理，而不是通常使用的 Pixel Shader。算法创建一系列覆盖整个参数域的点作为输入，每个点在参数域上的纹理坐标会被用来在 Vertex Shader 中直接计算该点在三维空间的坐标，之后经过光栅化和 Pixel Shader，符合要求的点会被纹理编码到工作表格中。

(3) CSG 模型：

CSG 模型的体素化是非常直观的，首先将各个模型的几何顶点数据体素化并保存在各自的工作表格中，接下来交，并，减等布尔操作都可以利用 GPU 中二维纹理混合的不同方式实时完成。

2.4.2 透明绘制

透明绘制一直都是图形学中一个难题，实时体素化为此提供一个简便的解决方法。基于多值体素化的原理，对于每个体素使其包含离散量化的法向，纹理坐标和材质等多种属性并保存在工作表格中，从而可以在后继的绘制阶段利用已有数据进行光照效果的计算。本文提出的体素化引擎可以和基于切片的体绘制方法结合起来，实现复杂体模型的半透明绘制，此项技术在虚拟医学等领域有广阔应用。下图左边为一个具有 36,758 个三角面片和 29,371 个顶点的大脑模型的透明绘制效果，右边为将大脑模型放入一个 MRI 头部体数据模型（体分辨率为 128^3）进行透明的混合体绘制的效果，其绘制效率为 10 帧每秒。

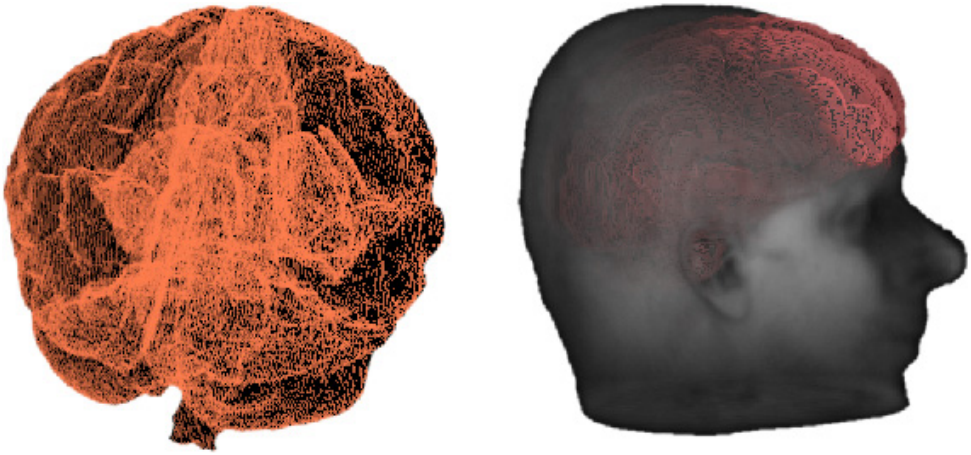


图 2.7 基于实时体素化的透明绘制

2.4.3 碰撞检测

碰撞检测是虚拟现实应用中的关键技术之一，基于实时体素化的碰撞检测算法能将碰撞检测中涉及的大量求交运算转化到离散的图像空间完成，并采用可编程图形硬件加速。我们的算法首先将需要进行检测的几何物体分别体素化为体模型，接下来几何模型公共部分的判断就变成了在图像空间中对多张二维纹理的操作。由于我们的体素化算法可以达到实时，因此对于复杂的几何物体都可实现实时的碰撞检测。下图图示了碰撞检测的结果，其中红线表示的部分为检测得到的两个几何模型的公共部分。

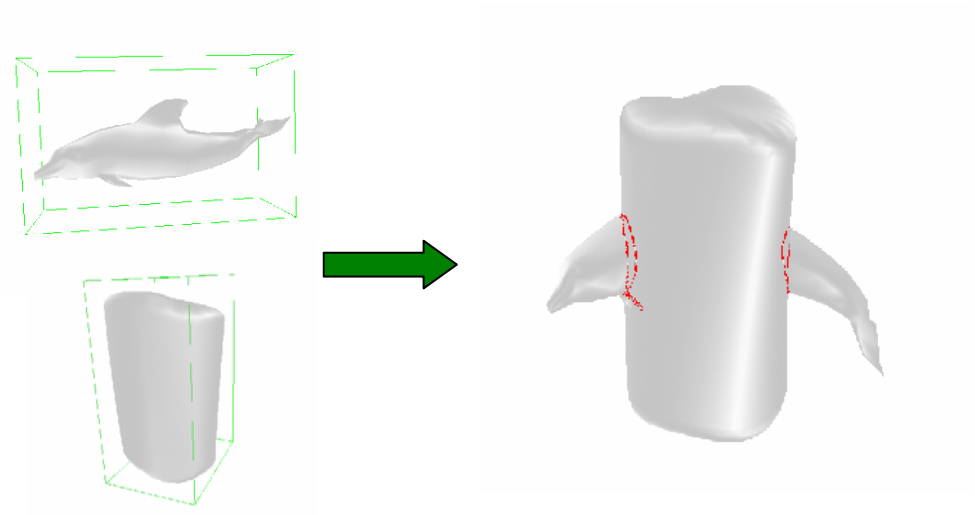


图 2.8 基于实时体素化的碰撞检测

2.5 实验结果及性能比较

上述算法的实验硬件平台是：一台具有 2G 内存的 2.4GHz Pentium IV CPU 的主机，配备具有 256M 显存的 ATI Radeon 9800 Pro 图形显卡。所有的实验结果和数据都在此平台上测试得到，使用 DirectX 9.0b SDK[30] 结合 Vertex/Pixel Shader 2.0 版本实现算法程序。

2.5.1 性能

表 2.1 列出了在 256^3 分辨率下不同尺寸模型体素化的时间消耗，各个模型体素化结果的绘制图像在图 2.9 中。实时体素化算法分为光栅化，纹理编码，合成三步，表格中给出了每一步具体的时间消耗。光栅化的速度主要受限于几何模型的三角面片个数，如 Blade 模型，它有 1,765,388 个三角面片因此光栅化需要大约 75ms；而纹理编码和合成均为在二维纹理空间的操作，其主要受限于目标体空间的分辨率和每个纹元中包含体素的数目。表 2.1 中的第 7 列包含的是预处理对几何数据按照投影方向分类的时间统计，很明显预处理时间和场景复杂度密切相关，但对于实时体素化算法，如果初始几何模型是静态的，则这个预处理过程仅需要执行一次。表格中没有列出绘制生成的体模型的时间，显然这个时间仅仅和目标体空间的分辨率相关，在 256^3 分辨率下表中所有模型的绘制时间均约为 8ms。

表 2.1 对于不同尺寸模型的体素化效率统计. 体模型分辨率: 256^3

模型	三角面片数目	顶点数目	光栅化	纹理编码	合成	预处理
Duck	1,254	947	9.0ms	8.0ms	10ms	0.8ms
Hugo	16,928	8,634	9.7ms	8.3ms	10ms	7.0ms
Bunny	69,451	34,834	12.0ms	8.0ms	11ms	27.0ms
Dragon	871,326	439,370	38.0ms	8.0ms	11ms	330.0ms
Buddha	1,087,514	550,868	47.0ms	7.8ms	10ms	415.0ms
Blade	1,765,388	898,796	75.0ms	8.0ms	12ms	660.0ms

我们还对同一个模型在不同体分辨率下的二值体素化作了测试，实验选取了 Wagner 模型，这个几何模型具有 60,246 个三角面片和 30,215 个顶点，统计结果在表 2.2 中，其对应的体素化结果的绘制图像在图 2.10 中，为了可以用一张工作表格完整包含其体模型数据，在 512^3 分辨率下算法使用一个 bit 来表示一个体素，而在 256^3，128^3 等分辨率下可以使用一个 byte 来表示一个体素，从表中数据我们可以看出，基于 bit 寻址存取的操作显然要比基于 byte 的操作要耗费更多时间。

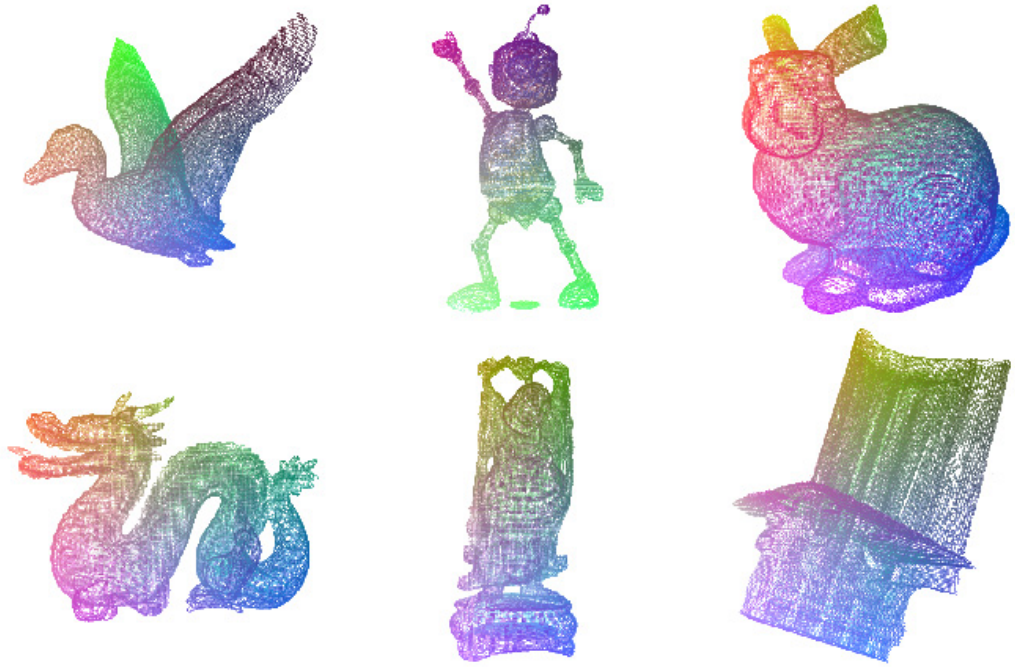


图 2.9 不同尺寸模型体素化结果的绘制图像

表2.2 对于Wagner模型在不同体分辨率下进行二值体素化时间统计

体分辨率	每个体素所需bit数目	耗费显存量	体素化时间	产生体素数目	绘制时间
512^3	1	16MB	500ms	620,381	240ms
256^3	8	16MB	30ms	151,347	10ms
128^3	16	4MB	24ms	38,012	6ms
64^3	8	256KB	21ms	9,379	4ms

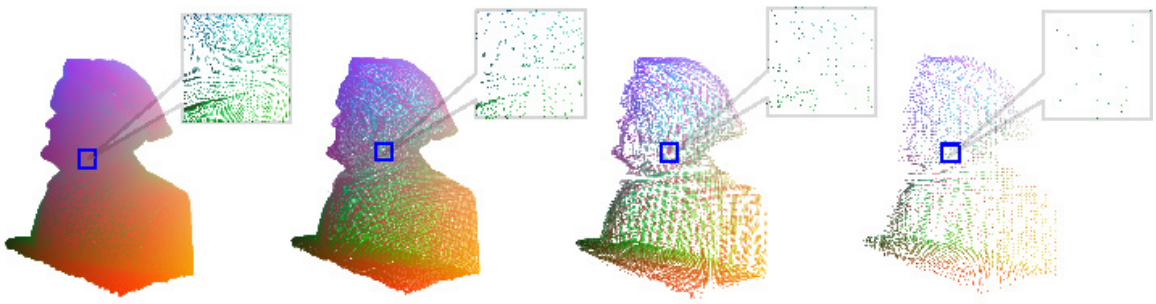


图 2.10 Wagner 模型在不同分辨率下体素化结果的绘制图像

2.5.2 质量

体素化生成体模型的精确性正比于其分辨率，也就是说目标体空间的分辨率越高，体模型的精确度越高；但同时分辨率越高也意味着需要更多的显存空间，特别对于多值体素化时，在具体实现过程中因为所使用硬件功能的限制，不支持浮点数据类型纹理的 alpha 混合操作（目前最新的图形硬件已经支持此项功能），往往需要多张工作表格纹理才能够存储。经过实现我们发现利用目前主流的 GPU，使用 256^3 的分辨率，用一个 byte 表示一个体素的方式可以获得较好的性能和质量。

在实现过程中，我们还发现实体体素化时对于闭合的多边形模型在某些情况下会产生走样，会产生一些不属于模型范围的奇异线。经测试发现，由于一小部分的体素在光栅化的时候会因为精度的问题而缺失，从而导致实体体素化的结果不正确。这个现象也说明了基于 GPU 硬件流水线的实时体素化方法和以往利用软件方式实现的方法的差异。随着图形硬件技术的不断发展，这个问题将很快得到解决。

2.5.3 比较

目前存在两种典型的利用图形硬件加速的体素化方法，一种是基于切片的体素化[31,32]，另一种是层次深度图像抽取 [27] 的方法。基于切片的体素化方法的主要问题在于需要多次光栅化输入几何数据，其次数等于目标体空间在 Z 方向上的分辨率，如生成 256^3 分辨率的体模型，需要光栅化 256 次初始几何数据，这极大地影响了算法的效率；虽然它也利用了专用图形硬件将结果绘制到三维纹理中，但到目前位置 PC 平台的图形硬件还不支持直接对 3D texture 进行读写；我们实现了该算法并用 2.5.1 中的实验数据测试算法性能，发现基于切片的体素化方法在 256^3 分辨率下，完成对 Blade 模型的体素化需要耗费 5s 的时间，而实时体素化方法只需要不到 0.1s。而层次深度图像抽取的方法，虽然可以减少对初始几何模型的遍历次数，但它是与视点相关的，每一帧都需要从显存中回读深度图像数据到内存中，依靠 CPU 对深度列表进行排序，因此当目标体空间的分辨率增大时，其算法性能受到很大的影响，很难达到实时的要求。

2.6 结论和未来工作

高效的体素化在交互图形学的研究中有广泛的应用，我们充分利用现有可编程图形硬件的强大功能，提出了一种利用流水线中对于二维纹理的操作来实现实时体素化

的方法，该方法简单稳定而且易于实现，我们相信随着图形硬件功能的日益增强，它会在很多领域取得广泛的应用。

就未来工作而言，首要的问题是提高体素化结果的质量，增加体模型的精确度，力求实现高性能且高精确的实时体素化。最新的图形硬件 Nvidia Geforce 6800 Ultra 已经可以支持浮点格式纹理的 alpha 混合操作，此项功能可以明显提高多值体素化的效率，因为如法向，纹理坐标等浮点数据不再需要编码解码而是可以直接读取，我们准备尽快利用新的图形硬件实现已有的体素化算法。在近期，图形硬件厂商还在考虑使普通 PC 级图形硬件支持直接对 3D texture 进行存取，实时体素化算法同样可以利用这项新功能获得更高的绘制效率和质量。

参考文献

[1] A. Kaufman and E. Shimony. 3d scan-conversion algorithms for voxel-based graphics. In *Proceedings of ACM Workshop on Interactive 3D Graphics*, pages 45–76, Chapel Hill, NC, USA, October 1986. ACM Press.

[2] A. Kaufman. Efficient algorithms for 3d scan-conversion of parametric curves, surfaces, and volumes. In *Proceedings of ACM SIGGRAPH 1987*, pages 171–179, USA, July 1987. ACM Press.

[3] S. Wang and A. Kaufman. Volume-sampled 3d modeling. *IEEE Computer Graphics and Applications*, 14(5):26–32, 1994.

[4] K. Kreeger and A. Kaufman. Mixing translucent polygons with volumes. In *Proceedings of IEEE Visualization 1999*, pages 191–198, USA, October 1999.

[5] W. McNeely, K. Puterbaugh, and J. Troy. Six degree-offreedom haptic rendering using voxel sampling. In *Proceedings of ACM SIGGRAPH 1999*, pages 401–408, 1999.

[6] R. Westermann, O. Sommer, and T. Ertl. Decoupling polygon rendering from geometry using rasterization hardware. In *Proceedings of the 10th Eurographics Workshop on Rendering*, pages 53–64, 1999.

[7] S. Fang and D. Liao. Fast csg voxelization by frame buffer pixel mapping. In *Proceedings of the ACM/IEEE Volume Visualization and Graphics Symposium 2000*, pages 43–48, Salt Lake City, UT, USA, October 2000.

[8] T. He and A. Kaufman. Collision detection for volumetric objects. In *Proceedings of IEEE Visualization 1997*, pages 27–35. IEEE Computer Society Press, 1997.

[9] M. Boyles and S. Fang. Slicing-based volumetric collision detection. *ACM Journal of Graphics*

Tools, 4(4):23–32, 2000.

[10] N. Gagvani and D. Silver. Shape-based volumetric collision detection. In *Proceedings of the IEEE Symposium on Volume visualization 2000*, pages 57–61. ACM Press, 2000.

[11] F. Dachille and A. Kaufman. Incremental triangle voxelization. In *Proceedings of Graphics Interface*, pages 205–212, May 2000.

[12] N. Stolte. Robust voxelization of surfaces. *Technical Report TR.97.06.23, State University of New York at Stony Brook*, 1997.

[13] J. Huang, R. Yagel, V. Filippov, and Y. Kurzion. An accurate method for voxelizing polygon meshes. In *IEEE Symposium on Volume Visualization*, pages 119–126, 1998.

[14] M. Sramek and A. Kaufman. Alias-free voxelization of geometric objects. *IEEE Transactions on Visualization and Computer Graphics*, 5(3):251–267, 1999.

[15] D. Haumont and N.Warzee. Complete polygonal scene voxelization. *ACM Journal of Graphics Tools*, 7(3):27–41, 2002.

[16] S. Wang and A. Kaufman. Volume sampled voxelization of geometric primitives. In *Proceedings of IEEE Visualization 1993*, pages 78–84. IEEE Computer Society Press, October 1993.

[17] M. W. Jones. The production of volume data from triangular meshes using voxelisation. *Computer Graphics Forum*, 15(5):311–318, 1996.

[18] H. Widjaya, T. Mueller, and A. Entezari. Voxelization in common sampling lattices. In *Proceedings of Pacific Graphics 2003*, pages 497–501, Canmore, Alberta, Canada, October 2003.

[19] G. Varadhan, S. Krishnan, Y. J. Kim, S. Diggavi, and D. Manocha. Efficient max-norm distance computation and reliable voxelization. In *Proceedings of the Eurographics/ ACM SIGGRAPH symposium on Geometry processing*, pages 116–126. Eurographics Association, 2003.

[20] C. E. Prakash and S. Manohar. Shared memory multiprocessor implementation of voxelization for volume visualization. *HPC for Computer Graphics and Visualization*, 17(3):135–145, 1995. (Proc. Eurographics'98).

[21] H. Chen and S. Fang. Fast voxelization of 3d synthetic objects. *ACM Journal of Graphics Tools*, 3(4):33–45, 1999.

[22] S. Fang and H. Chen. Hardware accelerated voxelization. *Computers and Graphics*, 24(3):433–442, 2000.

[23] S. Fang and H. Chen. Hardware accelerated voxelization. *Volume Graphics*, pages 301–315, 2000.

[24] S. Beckhaus, J. Wind, and T. Strothotte. Hardware-based voxelization for 3d spatial analysis. In *Proceedings of the 5th International Conference on Computer Graphics and Imaging*, pages 15–20, Canmore, Alberta, Canada, August 2002. ACTA Press.

[25] G. P. Evaggelia-Aggeliki Karabassi and T. Theoharis. A fast depth-buffer-based voxelization algorithm. *ACM Journal of Graphics Tools*, 4(4):5–10, 1999.

[26] C. Everitt. Interactive order-independent transparency. *Technical report, NVIDIA Corporation.*, May 2001.

[27] B. Heidelberger, M. Teschner, and M. Gross. Real-time volumetric intersections of deforming objects. In *Proceedings of Vision, Modeling, Visualization 2003*, pages 461–468, Munich, Germany, November 2003.

[28] B. Heidelberger, M. Teschner, and M. Gross. Volumetric collision detection for deformable objects. April 2003.

[29] NVIDIA Corporation. *Cg specification*, August 2002.

[30] Microsoft Corporation. *DirectX 9.0 SDK*, December 2002.

[31] S. Fang and H. Chen. Hardware accelerated voxelization. *Computers and Graphics*, 24(3):433–442, 2000.

[32] S. Fang and H. Chen. Hardware accelerated voxelization. *Volume Graphics*, pages 301–315, 2000.

第三章 高质量大尺寸点模型实时绘制算法

点模型用几何体表面密集采样的离散点来隐式地表示几何体。与传统的三角（多边形）网格模型相比，点模型不需要任何拓扑信息的支持，便于重采样，可以建立非常灵活的层次结构。实践表明，点模型非常适合于表达雕像等形状复杂且不规则的物体。对于这类景物，采用点模型，不仅能获得更高的绘制速度[1]，也能获得更高的绘制质量[2]。

3.1 点绘制相关工作介绍

运用点模型表示几何物体的思想最早可以追溯到 1985 年[3]。2000 年的 SIGGRAPH 大会上展示的 QSplat 技术巧妙地利用点模型的灵活性，建立了一种层次包围球的结构。与传统层次结构的最大的不同在于，每个结点自身可以被看作是在一定分辨率下的一个表面重采样点，而这个点代表了其局部区域所有采样点的信息。当不影响绘制精度，或是绘制实时性更为重要的时候，可通过绘制该点来代替对它下属的所有子结点的绘制，从而提高绘制效率。此外，Qsplat 可通过从上至下遍历层次树的方式对各结点所代表的曲面局部区域进行视域裁剪和背面剔除，如果上层结点被确定位于视域外或背向视点，则无需对子树中的每一个结点分别进行判断。然而，由于绘制算法基于层次结构的遍历，因此无法利用可编程图形硬件基于流的工作模式进行加速。2001 年，Rusinkiewicz 等人[4]提出 Streaming Qsplat 的概念，将模型序列化为线性结构，从而能在网络中快速传输点模型。由于其绘制仍通过层次树遍历来完成，因此只能由软件实现。2003 年，Stamminger 研究小组[5]提出了一种将点模型的层次结构序列化为一个线性数组的方法，并在此基础上将绘制算法由层次树的遍历转化为线性数组的遍历，从而巧妙地实现了 Qsplat 的 GPU 硬件加速，极大地提高了绘制效率。他们报告的效率为每秒绘制约 5 千万个点。

以上算法都侧重于绘制的效率，而不考虑绘制的质量。它们在绘制的时候把每个点用一个不透明的正方形表示，能用 GL_POINT 简单实现。这类算法虽然高效，但通常会产生严重的图形走样 [2]。因此点图形学中另一部分重要的工作着眼于提高点模型绘制的质量。2000 年，Pfister 等人[6]率先提出用点面元(Surfel)来代表一个采样点。这个点被表示为位于点切向上的一个圆盘，各点的圆盘相互重叠并形成紧密(water-tight)

的物体表面。在绘制时，首先通过可见性预处理去除被遮挡的点，然后将可见点投影到屏幕空间，并在屏幕空间进行二维图像重建而得到光滑图像。2001 年，Zwicker 等人在 Surface Splatting 一文中将信号处理的思想引入到点绘制中，并采用了 EWA 反走样滤波[7]的技术，获得了高质量的绘制结果。2002 年 Ren 等人[8]在此文的基础上推导出物体空间的 EWA 滤波算子，并基于可编程图形硬件获得了每秒约 150 万个点的绘制效率。2003 年，Kobbelt 等人[9]提出了一种基于高斯滤波的点模型图形硬件加速方法，取得了更高的效率。他们的算法在绘制效果上逊色于 EWA，并且没有考虑点模型的层次结构，从而缺乏处理大规模的点模型场景的手段。

注意到高效的基于层次结构的算法，如 QSplat 和 Sequential Point Tree 等，在模型距视点较远时将多个点合并为一个点处理，从而大大提高了绘制效率。但它们不考虑绘制质量，在距离视点较近时以及在模型侧影轮廓线区域处会产生严重的图形走样。高质量的算法如 EWA 滤波算法可以减少这些走样现象，但计算代价太大, 导致绘制速度降低。一个自然的想法是，依据模型中的点与视点的远近关系自适应地选取不同的细节层次和绘制模式, 对于远处的点，将它们“合并”为较大的点，用较模糊的绘制模式绘制; 对于近处的点，则同时选取用较精细的层次结构和绘制模式来绘制。这就引入了一个自适应绘制的概念。我们的工作通过建立一种算法机制，对模型的不同部分自动地采用不同的细节层次和适宜的绘制模式，并充分利用强大的可编程图形硬件技术，实现了大尺寸点模型的高质量实时绘制。

3.2 自适应绘制算法及其分析

3.2.1 自适应算法总体框架

本章提出的自适应绘制算法包含两个方面的含义，即绘制时对模型的不同部分选择不同的细节层次和不同的绘制模式。算法在预处理阶段首先将模型剖分为若干点片，点片是点模型中一组空间上相邻点的集合, 每一点片代表模型表面上一片连续区域。对每一点片分别建立一个独立的层次结构，然后将每个点片中的层次结构序列化为线性二叉树保存。在自适应绘制阶段，首先，对所有点片进行视域裁剪和背面剔除，快速剔除完全不可见的点片。然后对于每一可见的点片，再根据其与视点的远近及视线与点片平均法向的夹角，选取不同的几何细节层次和不同的绘制模式进行绘制。算法流程如下图所示：

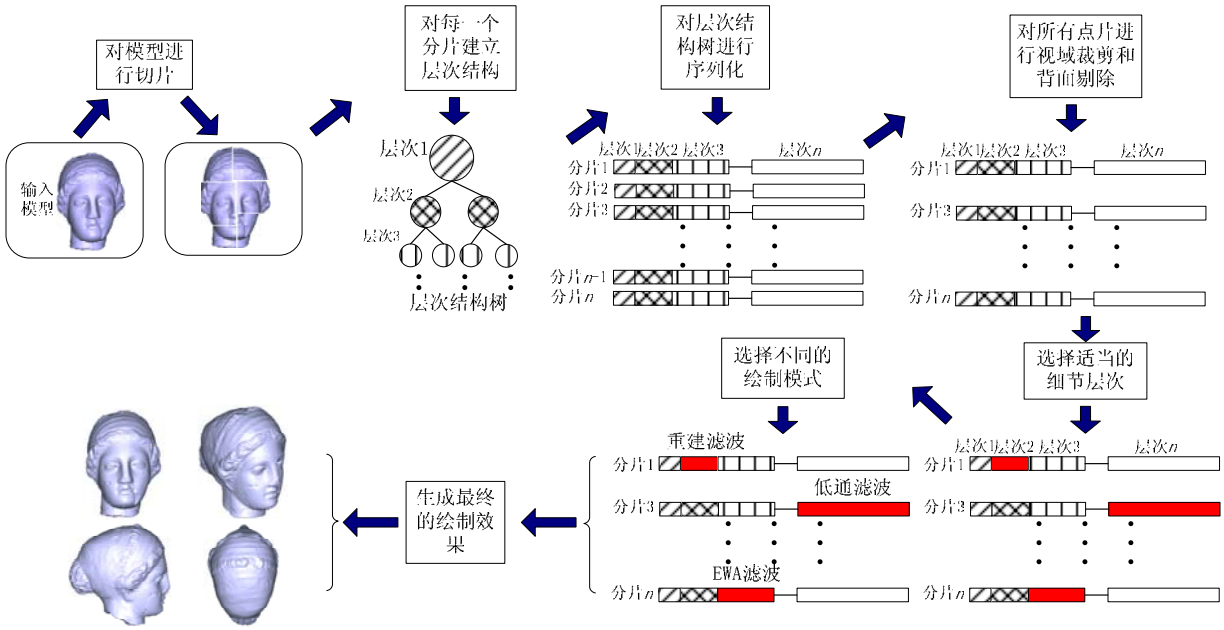


图 3.1 自适应绘制算法流程

3.2.2 点模型的预处理分片

本文的原始数据中包含位置、法向及半径等信息。所采取的实现点模型分片的方法是，首先，由模型中所有的点，建立一棵完整的二叉树。然后，把二叉树剖分为若干子树，则每一棵子树就是一个点片。我们利用协方差分析的方法[10]，计算最佳剖分平面，采用自适应的剖分算法剖分子树。对于模型较为平坦、细节较少的部分，剖分较少的点片；对于模型较为尖锐，细节比较丰富的部分，切割为较多的点片。我们同样使用协方差分析的方法来估计二叉树每个结点的曲率，并综合考虑它的法向锥以实现这样的自适应切割。

为了避免层次遍历，我们将每一棵子树序列化为一个线性表。Sequential Point Tree[11]是一种由八叉树序列化而来的线性表，其对应的绘制算法由层次结构的根结点开始遍历，在绘制时会处理很多额外的点，带来绘制效率的损失。本文提出使用线性二叉树来实现二叉树的序列化，将子树中的数据按照广度优先的顺序存放到一个线性数组中，并记下每一层次的数据的索引号和该层次中的数据的相关信息，如该层次结点的最大半径、最小半径等；在绘制时根据每一片预先记录的信息快速地选择该片的线性二叉树中恰当的一层进行绘制，这样就无需遍历其上层的结点，保证了不会处理大量额外的点。图 3.2 形象地说明了我们建立分片的点模型结构的流程。

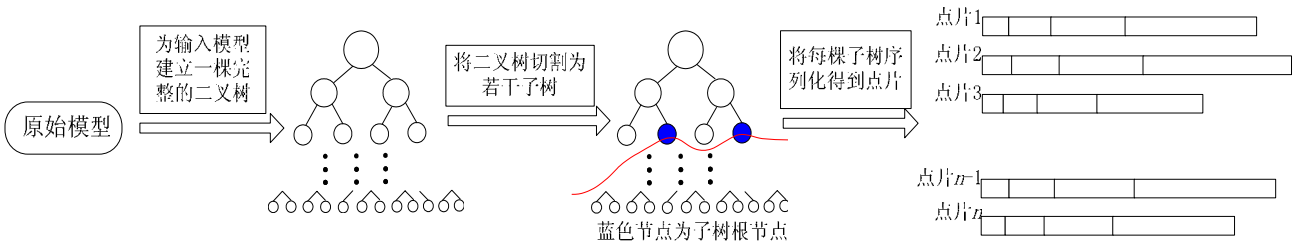


图 3.2 建立分片的点模型结构

3.2.3 自适应反走样点绘制策略

为了获得高质量的绘制效果，我们采用具有良好反走样效果的 EWA 滤波对模型进行绘制。但是，计算 EWA 滤波开销较高，影响绘制效率。经过严格的理论推导，我们发现，EWA 所需的繁杂的计算并不总是必须的，根据视点与模型的距离远近和视角大小，可以采用 EWA 的不同近似方式，大大简化 EWA 的计算，在不损失或损失很少的质量的前提下，大幅提高绘制效率。 这是本文自适应绘制算法的重要理论依据。这一节首先简要回顾一下 EWA 滤波的概念，然后推导出 EWA 简化的条件表达式。

3.2.3.1 椭圆加权滤波

EWA（Elliptical Weighted Average:椭圆加权）滤波[7]通过引入二维屏幕空间的低通滤波解决了由于透视变换带来的走样问题，并已成功应用到点和体绘制[12]。

设 $\{\mathbf{P}_k\}_{k=1}^n$ 是点模型的点集，则二维屏幕空间的 EWA 重采样函数 $g_c'(\mathbf{x})$ 是点数据的重建滤波函数 $g_c(\mathbf{x})$ 与二维屏幕低通滤波函数 $h(\mathbf{x})$ 的卷积（图 3.3）：

$$g_c'(\mathbf{x}) = g_c(\mathbf{x}) \otimes h(\mathbf{x}) = \int_{\mathbb{R}^2} g_c(\xi) h(\mathbf{x} - \xi) d\xi \tag{1}$$

EWA 滤波的二维形式是高斯函数 $G_V(\mathbf{x})$：

$$G_V(\mathbf{x}) = \frac{1}{2\pi |V|^{\frac{1}{2}}} e^{-\frac{1}{2}\mathbf{x}^T V^{-1}\mathbf{x}} \tag{2}$$

其中，V 是高斯函数的方差矩阵。

记空间某点 $\mathbf{P}_k$ 的重建滤波的方差矩阵为 V_k^r，低通滤波的方差矩阵为 V_h。通常，两者都取为对角阵。对应 $\mathbf{P}_k$ 的 EWA 重采样滤波 $\rho_k(\mathbf{x})$ 为

$$\rho_k(\mathbf{x}) = G_{J_k V_k^r J_k^T + V_h}(\mathbf{x} - \mathbf{x}_k) \tag{3}$$

式中 $\mathbf{x}_k$ 是点 $\mathbf{P}_k$ 通过空间变换 $\mathbf{m}$ 在屏幕的投影点，J_k 是坐标变换 $\mathbf{m}$ 的 Jacobi 矩阵（泰勒一级展开），$H_k = J_k V_k^r J_k^{\mathrm{T}} + V_h$ 是 EWA 重采样滤波对应的方差矩阵。

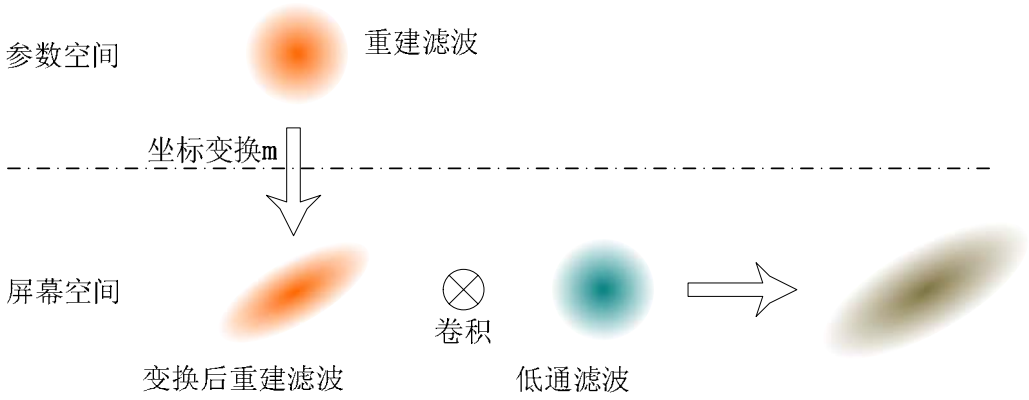


图 3.3 EWA 滤波原理示意图

3.2.3.2 自适应椭圆加权滤波

EWA能获得高质量绘制结果，但是逐点计算(3)式开销较大，影响绘制效率。注意到EWA重采样滤波由重建滤波与低通滤波卷积而成，两者分别对EWA的权值产生部分影响。当某点距离视点近的时候，重建滤波占的比例大。当距离非常近的时候，重建滤波占绝对地位，低通滤波的影响可以忽略不计。反之，当距离很远的时候，重建滤波的影响微乎其微，EWA重采样滤波完全可以用低通滤波来代替。

我们对(3)式进行细致的分析。虽然高斯滤波是无限支撑的，但实际使用时设置截断半径(cutoff radius)，令对应 V_k^r 和 V_h 的 cutoff 半径为 r_k 和 r_h，通常它们被取为点 $\mathbf{P}_k$ 的半径和屏幕像素间的距离，则 V_k^r 和 V_h 可表述为：

$$V_k^r = \begin{pmatrix} r_k^2 & 0 \\ 0 & r_k^2 \end{pmatrix}, \quad V_h = \begin{pmatrix} r_h^2 & 0 \\ 0 & r_h^2 \end{pmatrix},$$

由于 V_k^r 是对称阵，因此 $J_k V_k^r J_k^{\mathrm{T}}$ 是对称阵。根据矩阵理论，存在正交(旋转)矩阵 R_θ，使得

$$J_k V_k^r J_k^{\mathrm{T}} = R_\theta^T \begin{pmatrix} \lambda_0 & 0 \\ 0 & \lambda_1 \end{pmatrix} R_\theta \qquad (4)$$

此处 λ_0 与 λ_1 是矩阵 $J_k V_k^r J_k^{\mathrm{T}}$ 的特征值。注意到 $V_h = R_\theta^{\mathrm{T}} \cdot V_h \cdot R_\theta$，因此有

$$H_k = J_k V_k^r J_k^{\mathrm{T}} + V_h = R_\theta^T \begin{pmatrix} \lambda_0 + r_h^2 & 0 \\ 0 & \lambda_1 + r_h^2 \end{pmatrix} R_\theta \tag{5}$$

令 $\mathbf{y} = (y_1, y_2)^{\mathrm{T}} = R_\theta \cdot (\mathbf{x} - \mathbf{x}_k)$，并代入(3)式，得

$$G_{J_k V_k^r J_k^{\mathrm{T}} + V_h}(\mathbf{x} - \mathbf{x}_k) = \frac{1}{2\pi\sqrt{(\lambda_0 + r_h^2)(\lambda_1 + r_h^2)}} e^{-\frac{1}{2}\left[\left(\lambda_0 y_1^2 + \lambda_1 y_2^2\right) + r_h^2\left(y_1^2 + y_2^2\right)\right]} \tag{6}$$

由上式可以看出，当 $\min(\lambda_0, \lambda_1) \gg r_h^2$ 时，重建滤波占主导；当 $\max(\lambda_0, \lambda_1) \ll r_h^2$ 时，低通滤波占主导。注意到 $\left|J_k V_k^r J_k^{\mathrm{T}}\right|$ 等于 λ_0 与 λ_1 的乘积，因此，当 $\left|J_k V_k^r J_k^{\mathrm{T}}\right| \gg r_h^2$ 时，低通滤波的影响可以忽略不计；当 $\left|J_k V_k^r J_k^{\mathrm{T}}\right| \ll r_h^2$ 时，则重建滤波的影响可以忽略不计。由于 $\left|J_k V_k^r J_k^{\mathrm{T}}\right| = r_k^4 \left|J_k\right|^2$，下面我们考察 $\left|J_k\right|$。令 $\tilde{\mathbf{s}}_k$，$\tilde{\mathbf{t}}_k$ 是 $\mathbf{P}_k$ 处的任意两个互相垂直的单位切向，则

$$u_s = (\mathbf{P} - \mathbf{P}_k) \cdot \tilde{\mathbf{s}}_k, \quad u_t = (\mathbf{P} - \mathbf{P}_k) \cdot \tilde{\mathbf{t}}_k, \tag{7}$$

给出了一个 $\mathbf{P}_k$ 的邻域（圆盘）的局部参数化。

设坐标变换 m 由平移、旋转和投影变换组成，则 Jacobi 矩阵

$$J_k = \frac{1}{o_z} \begin{bmatrix} s_x - s_z \cdot \dfrac{o_x}{o_z} & t_x - t_z \cdot \dfrac{o_x}{o_z} \\ s_z \cdot \dfrac{o_y}{o_z} - s_y & t_z \cdot \dfrac{o_y}{o_z} - t_y \end{bmatrix} \tag{8}$$

其中，(o_x, o_y, o_z)、(s_x, s_y, s_z) 与 (t_x, t_y, t_z) 分别是 $\mathbf{P}_k$、$\tilde{\mathbf{s}}_k$ 与 $\tilde{\mathbf{t}}_k$ 变换到视点坐标系下的向量。计算 J_k 的行列式，得到

$$\left|J_k\right| = \frac{1}{o_z^3} \cdot \left(o_x \begin{vmatrix} t_y & t_z \\ s_y & s_z \end{vmatrix} + o_y \begin{vmatrix} t_z & t_x \\ s_z & s_x \end{vmatrix} + o_z \begin{vmatrix} t_x & t_y \\ s_x & s_y \end{vmatrix} \right) \tag{9}$$

记 (n_x, n_y, n_z) 是 $\mathbf{P}_k$ 在视点坐标系下的单位法向量，则有

$$(n_x, n_y, n_z) = -\left(\begin{vmatrix} t_y & t_z \\ s_y & s_z \end{vmatrix}, \begin{vmatrix} t_z & t_x \\ s_z & s_x \end{vmatrix}, \begin{vmatrix} t_x & t_y \\ s_x & s_y \end{vmatrix} \right) \tag{10}$$

将（11）式代入（10）式，得到

$$\left|J_k\right| = \frac{\cos\theta}{o_z^2} \cdot \sqrt{\left(\frac{o_x}{o_z}\right)^2 + \left(\frac{o_y}{o_z}\right)^2 + 1} \tag{11}$$

θ 是视线与点法线的夹角，因此

$$\left|J_k V_k^r J_k^{\mathrm{T}}\right| = \left[\frac{r_k^2 \cdot \cos\theta}{o_z^2} \cdot \sqrt{\left(\frac{o_x}{o_z}\right)^2 + \left(\frac{o_y}{o_z}\right)^2 + 1}\right]^2 = \left[\frac{S_{proj}}{\pi} \cdot \sqrt{\left(\frac{o_x}{o_z}\right)^2 + \left(\frac{o_y}{o_z}\right)^2 + 1}\right]^2 \quad (12)$$

令 $S_{proj} = \pi \cdot r_k^2 \cdot \cos\theta$ 是点 $\mathbf{P}_k$ 在屏幕上的投影面积，则有

$$\frac{S_{proj}}{\pi} \cdot \le \sqrt{\left|J_k V_k^r J_k^{\mathrm{T}}\right|} \le \frac{S_{proj}}{\pi} \cdot \sqrt{2 \cdot \tan^2\left(\frac{fov}{2}\right) + 1} \quad (13)$$

其中，fov 为视角，在给定相机参数的情况下，对于所有点是常数。

记 $C_\eta = \sqrt{2 \cdot \tan^2\left(\frac{fov}{2}\right) + 1}$。令 $S_h = \pi \cdot r_h^2$ 是低通滤波在屏幕上的有效范围面积，那么 S_{proj} 与 S_h 的比值决定了重建滤波与低通滤波在 EWA 重采样滤波中的权重。结合(13)式，我们可采取以下的的自适应绘制策略：

ChooseRenderMode

if $S_{proj} / S_h > c_{\max}$ **then** 用重建滤波绘制，$H_k = J_k V_k^r J_k^{\mathrm{T}}$；

else if $C_\eta S_{proj} / S_h < c_{\min}$ **then** 用低通滤波绘制，$H_k = V_h$；

else 用 EWA 滤波绘制。

其中 $c_{\min}$，$c_{\max}$ 是调节效率与质量的参数。一般我们取 $c_{\min} = 0.3$, $c_{\max} = 3.0$。自适应绘制选择策略的示意图见图3.4。我们选择不同的滤波策略的好处在于单独计算重建滤波或低通滤波远比EWA重采样滤波简单。

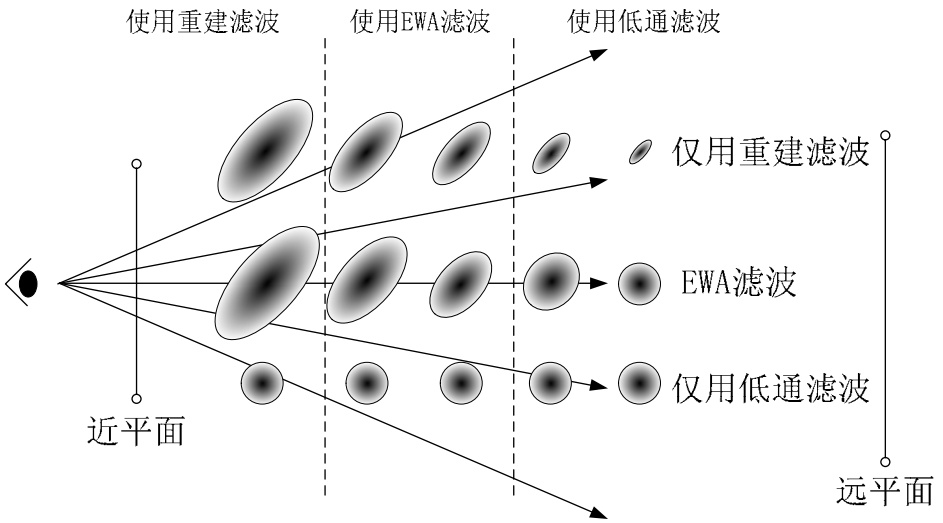


图3.4 自适应EWA示意图

3.3 自适应绘制算法的具体实现

3.3.1 算法实现细节

3.3.1.1 视域裁剪和背面剔除

在预处理阶段，我们求出模型的每一个点片的包围盒、平均法向和法向锥。在绘制某点片时，首先利用该片的包围盒进行视域裁剪，快速判断它是否位于视域之外。然后利用该点片的平均法向和法向锥，快速判断它是否背向视点。实验证明，通过视域裁剪和背面剔除可以大大减少实际绘制的点的数量，从而提高绘制效率。由于我们所采用的视域裁剪和背面剔除的方法不需要遍历层次结构，因此便于用当前面向流的可编程图形硬件实现。

3.3.1.2 细节层次的选取

对于可见的点片，首先为它选择恰当的细节层次。我们计算出它的包围盒的八个顶点距离视点的最近距离，用这个最近距离来保守地估计该点片到视点的距离。然后以该点片中所包含点的最大半径估算其中每一点元投影到屏幕的面积。如果某一点片的平均曲率较大，或是它位于侧影轮廓线附近，或者它的法向锥较大，则对它采用较精细的层次，使得它包含的每个点投影到屏幕上的面积足够小；否则我们对它选择较为粗略的层次，使得每个点投影到屏幕上的较大。

3.3.1.3 绘制模式的选择

一旦选择了点片的某一层次进行绘制，就可以依据该层结点所包含的点的平均半径来估算投影面积, 然后用前面提出的选择策略来选择最佳的绘制模式。一个例外的情况是，对于位于侧影轮廓线附近的点片，我们总是选择生成最佳质量的 EWA 进行绘制。

3.3.2 大尺寸点模型几何数据的压缩

我们采用 Ren 的方法 [8]，用带高斯纹理的四边形来绘制一个点。但是，采用自适应 EWA 滤波虽然取得了较好的绘制效果，却产生了模型数据量太大的问题。为了计算 EWA 滤波，每个点需要记录位置、法向、两个切向、半径以及高斯纹理坐标等数

据。其中位置、法向、切向均为包含 3 个浮点型数据分量的矢量，纹理坐标和半径总共包含 3 个浮点型数据，由此一个 EWA 滤波的点的数据总量为 60 个字节。若点模型包含 1 千万个点，绘制算法所需要的点的数据量将达到 600M 字节。另一方面，GPU 硬件为了避免在 GPU 和 CPU 之间频繁交换数据和指令，选择将需要绘制的点的数据全部存储在硬件本身的显存中的保留模式，而显存本身的容量一般不会大于 256M 字节。因此为了利用 GPU 实现千万级点模型的高效绘制，我们对点的数据进行了压缩，压缩比接近 8:1，各部分的压缩情况如表 3.1 所示。

表 3.1 点数据的压缩统计

数据类型	压缩前数据大小(bit)	压缩后数据大小(bit)	最终处理数据大小(bit)
位置	96	32	32
法向	96	16	16
切向	96×2	2×2	8
高斯纹理坐标	64	2	
半径	32	8	8
总计	480 = 60 Bytes	60	64=8 Bytes

3.3.2.1 点云位置量化压缩

对于点的位置，将点片的包围盒空间均分为$256\times256\times256$的网格，计算出每个点所处的网格的位置，用 3 个字节型数据分别保存网格位置的空间坐标，然后我们对于每个网格进一步的细分为$8\times8\times4$的精确网格，定位每个点的位置，用 1 个字节型数据保存，其中 3 个比特表示精确网格位置的 X 位置，3 个比特表示其 Y 位置，剩余 2 个比特表示其 Z 位置；经过这样的空间剖分，包围盒空间实际被分成了$2048\times2048\times1024$个空间网格，点所在的网格的位置在精度上已经完全可以替代点的原始位置。

3.3.2.2 参数化法向压缩

对于法向，我们使用一个单位半径的球面坐标系统，法向的三个分量可以用球面坐标中的θ和φ两个角度的三角函数形式表示，因此我们采用对θ和φ进行离散化的方式进行压缩，将两个角度在其定义域中分成 256 等份，对于每个点的法向计算其所

对应的θ和φ，然后存储其所对应角度的在 0-255 中的索引值，这样可以用 2 个字节型数据表示法向。采用这种压缩方式实际可以表示的法向类型为 65536 种，从原理上说这样会带来一定的法向信息的损失，但在实际绘制过程中我们发现效果是完全可以接受的。

3.3.2.3 切向压缩

考虑切向和法向的关系，其压缩方法比较直观，对于法向(n_x, n_y, n_z)，其切向可从$(0, -n_z, n_y)$、$(n_z, 0, -n_x)$及$(-n_y, n_x, 0)$三种中选择其一，因此我们只需要用 2 个比特就可以确定点的一个切向。

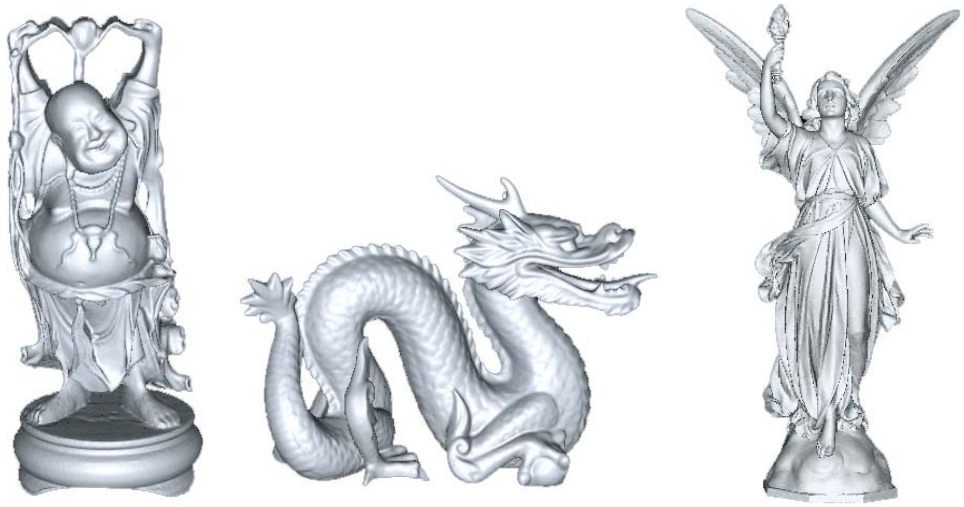
3.3.2.4 基于查找表的纹理坐标压缩

对于高斯纹理坐标，只有（0, 0）、（0, 1）、（1, 1）、（0, 1）四种情况，因此可以采用 2 个比特表示。我们把表示切向的 4 个比特和纹理坐标的 2 个比特存储在同一个字节型数据中。

3.3.2.5 半径几何查找表

对于模型中点的半径，我们采用了一种基于聚类分析的选择方法。首先统计出模型中所有点的半径的分布情况，生成 256 种半径值作为候选，制成半径的查找表；然后对于每个点利用折半查找的方法查找出其半径最接近的半径值，存储它的索引号。这样可以用 1 个字节表示点的半径。

经过上述的压缩方法之后每个点的数据所占用的空间为 8 个字节，约为原来 60 个字节的 1/8 ，大大减少了绘制所需的数据量，使得我们的算法程序可以利用 GPU 高速绘制千万级的点模型。图 3.5 给出了 Lucy 的绘制结果。



a b c

图 3.5 大规模点模型的实时绘制结果：a：buddha，1.06M 个点，13.06FPS；c：Lucy，10.07M 个点，10.25FPS；b：dragon，1.28M 个点，11.77FPS。

3.4 实验结果及分析

我们利用 Visual C++ 和 DIRECTX 9.0b SDK 在一台配备Intel P4 2.4GHz、ATI 9800Pro 显卡、1G内存的微机上实现了本文的算法。以下是我们的实验结果和统计列表。

3.4.1 绘制效率

表 3.2 自适应绘制算法的绘制效率统计(绘制窗口的分辨率为 512x512)

模型	绘制方式	点的数量	实际绘制点	帧率(FPS)
Lucy（图 3.5）	Point Sprite	10.073M	3.071M	10.25
Dragon（图 3.5）	自适应 EWA	1.28M	0.42M	10.75
Buddha（图 3.5）	自适应 EWA	1.06M	0.31M	15.12
Hip（图 3.9）	自适应 EWA	0.53M	0.15M	38.12
Hand（图 3.8）	自适应 EWA	0.33M	0.11M	55.23
Lion（图 3.10）	自适应 EWA	0.18M	0.07M	80.5

由表 3.2 可以看出，自适应 EWA 的绘制速度约为 4.5M 个点/秒（实际绘制点 × 帧率），是 Ren[8] 报告的绘制速度的 3 倍。考虑到因层次结构的采用而带来的性能提升，我们的算法实际每秒约能绘制 18M 个点。由表 2 中 Lucy 模型的统计数据可以看出如果仅采用 PointSprite 进行绘制，则我们的自适应绘制约能达到 1 亿个点/秒。

3.4.2 绘制质量

通过采用自适应反走样策略，本文给出的自适应绘制算法在提高绘制效率的同时，也保证了绘制的质量。图 3.6 给出了 buddha 模型用本文的自适应绘制与 QSplat 绘制结果的比较。图 3.7 还给出了仅用重建滤波绘制，仅用低通滤波绘制与自适应绘制的结果比较。

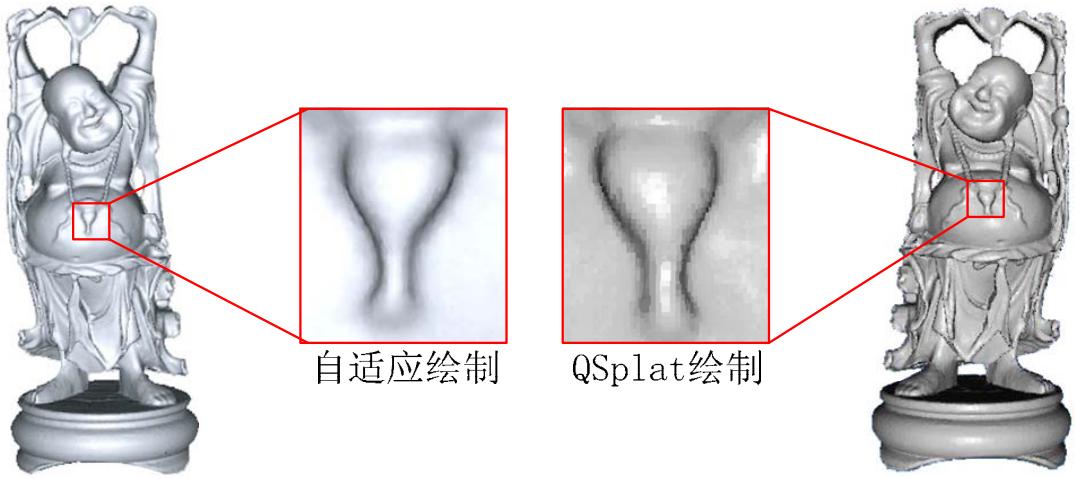
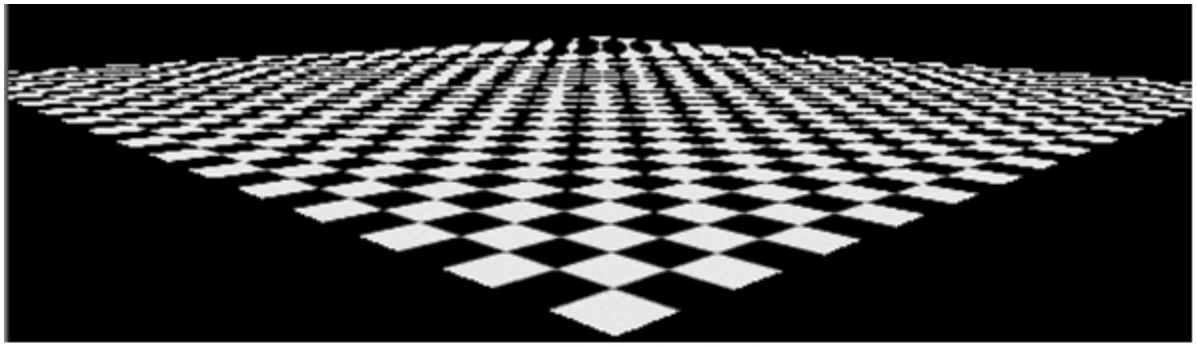
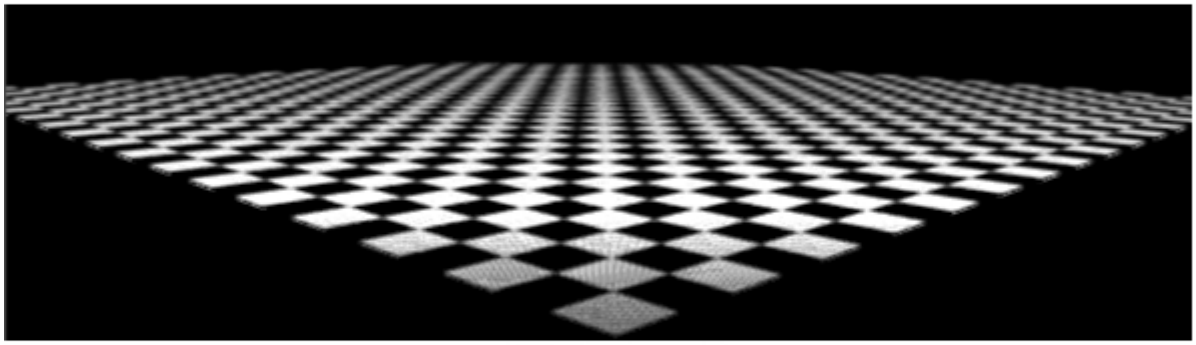


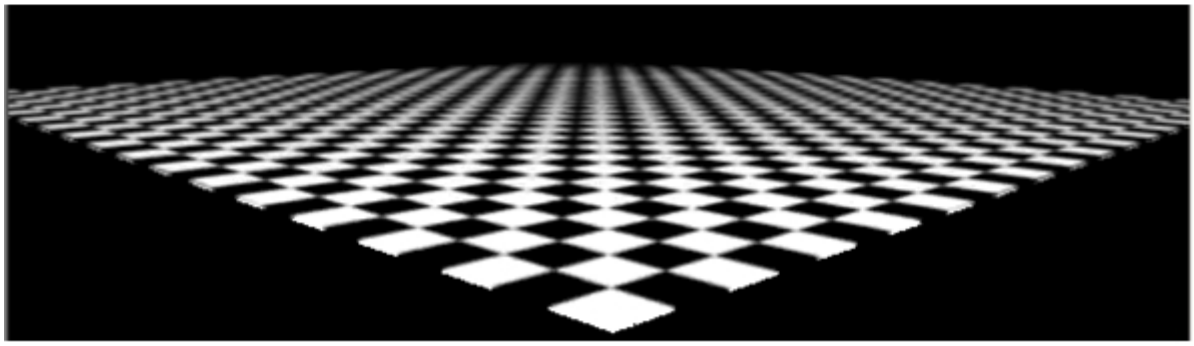
图 3.6 本文自适应绘制算法与 QSplat 绘制的比较



(a) 仅用重建滤波绘制



(b) 仅用低通滤波绘制



(c) 自适应绘制

图 3.7 自适应 EWA 反走样效果比较

由图 3.6 可以看出，我们的绘制结果比 QSplat 的绘制结果更加光滑。这是本文采用自适应 EWA 滤波进行反走样处理的结果。图 3.8，图 3.9，图 3.10 给出了 Hand, Hip, Lion 三个中等数据量的点模型通过自适应绘制算法绘制得到的结果。图 3.11 是百万级点模型 Dragon 的绘制和局部放大图，可以清晰的反映自适应算法的精度。图 3.12 是千万级点模型 Lucy 的侧面绘制图，图 3.13 和 3.14 通过使用自适应算法绘制精细的人头部模型，可以很好反映绘制的准确和逼真。

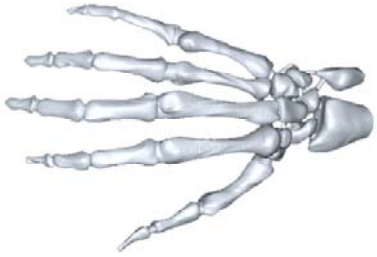


图3.8 Hand
0.33M Points, 55.23 FPS

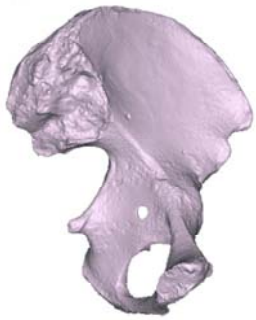


图3.9 Hip
0.53M Points, 38.12 FPS

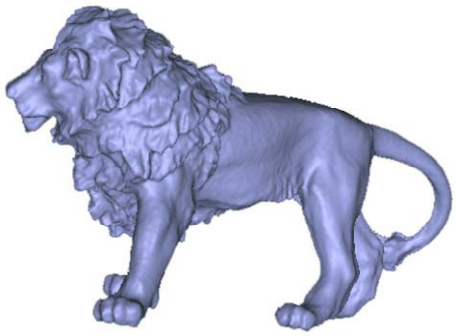
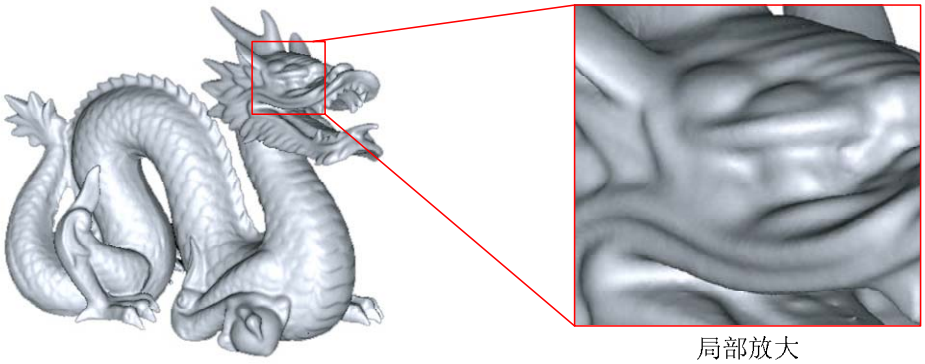


图3.10 Lion
0.18M Points, 80.5 FPS



局部放大

图3.11 Dragon
1.28M Points, 10.75 FPS

图3.14 Male
0.3M Points, 56.36 FPS

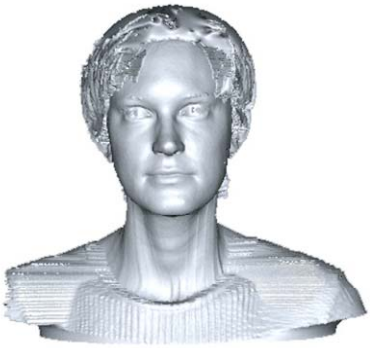


图3.13 Female
0.3M Points, 57.12FPS

图3.12 Lucy
10.073M Points,
10.25 FPS

3.5 结论和未来工作

本章提出了一种面向大规模点模型的自适应高质量实时绘制策略。算法首先将点模型剖分为若干点片，然后对每一个点片建立一个层次结构，并将每一个分片中的层次结构序列化为一个线性二叉表，便于图形硬件的流水线操作；在绘制过程中，基于本章提出的 EWA 滤波近似方式的理论，结合已有的层次结构，依据距离和视角等条件，为每一可见点片自适应地选择不同的细节层次和适宜的绘制模式。此外，算法利用量化的思想压缩超大规模的点几何数据，使得可以利用保留模式一次性将点模型装入显存，并实时解压，避免由于 CPU 和 GPU 之间的频繁交换和简化层次带来的效率和质量损失。

未来的工作将集中在三个方面。首先，结合层次结构进行压缩算法，实现面向网络的渐进式传输；其次，对于带颜色的大规模点模型，提出新的结合颜色空间和几何空间的层次结构，以及新的考虑颜色和几何特征的自适应绘制策略。最后，我们将考虑大规模基于广义点云表示（包括三维扫描数据、深度图像数据和粒子系统）的室外场景的实时建模与绘制算法。

参考文献

[1] S. Rusinkiewicz and M. Levoy. Streaming QSplat: A Viewer for Networked Visualization of Large, Dense Models. In *Siggraph 2000 Conference Proceedings*, pages 343.352, 2000

[2] M. Zwicker, H. P_ster, J. van Baar, and M. Gross. Surface splatting. In *Siggraph 2001 Conference Proceedings*, pages 371.378, 2001.

[3] M. Levoy and T. Whitted. The use of points as display primitives. Technical report, CS Departement, university of North Carolina at Chapel Hill, January 1985.

[4] S. Rusinkiewicz and M. Levoy. QSplat: a multiresolution point rendering system for large meshes. In *Siggraph 2001Symposium on Interactive 3D Graphics*, 2001.

[5] C. Dachsbacher, C. Vogelgsang, and M. Stamminger. Sequential point trees. In *Siggraph 2003 Conference Proceedings*, 2003.

[6] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: Surface elements as rendering primitives. In *Siggraph 2000 Conference Proceedings*, pages 335.342, 2000.

[7] P. S. Heckbert. Fundamentals of Texture Mapping and Image Warping. Master's thesis, University of california at Berkley, 1989.

[8] L. Ren, H. Pfister, and M. Zwicker. Object space ewa surface splatting: A hardware accelerated approach to high quality point rendering. In *Eurographics 2002 Conference Proceedings*, pages 461.470, 2002.

[9] M. Botsch and L. Kobbelt. High-Quality Point-Based Rendering on Modern GPUs. In Pacific

graphics 2003 Conference Proceedings, 2003.

[10] M. Pauly, M. Gross, and L. Kobbelt. Efficient simplification of point-sampled surfaces. In *Proc. IEEE Visualization 2002*, 2002.

[11] C. Dachsbacher, C. Vogelgsang, and M. Stamminger. Sequential point trees. In *Siggraph 2003 Conference Proceedings*, 2003.

[12] M. Zwicker, M. Pauly, O. Knoll, and M. Gross. PointShop 3D: An Interactive System for Point-Based Surface Editing. In *Siggraph 2002 Conference Proceedings*, 2002.

[13] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, and C. Silva. Point set surfaces. In *Proc. IEEE Visualization 2001*, pages 21.28, 2001.

[14] B. Chen and M. X. Nguyen. Pop: a hybrid point and polygon rendering system for large data. In *Proc. IEEE Visulization 2001*, pages 45.52, 2001.

[15] L. Coconu and H.-C. Hege. Hardware-accelerated pointbased rendering of complex scenes. In *Proc. Eurographic Workshop on Rendering 2002*, pages 41.51, 2002.

[16] J. D. Cohen, D. G. Aliaga, andW. Zhang. Hybrid simplification: combining multi-resolution polygon and point rendering. In *Proc. IEEE Visualization 2001*, pages 37.44, 2001.

[17] O. Deussen, C. Colditz, M. Stamminger, and G. Drettakis. Interactive visualization of complex plant ecosystems. In *Proc. IEEE Visualization 2002*, 2002.

[18] S. Fleishman, D. Cohen-Or, M. Alexa, and C. T. Silva. Progressive point set surfaces. *ACM Transaction on Graphics*, 2003.

[19] J. P. Grossman and W. J. Dally. Point sample rendering. In *Eurographics Workshop on Rendering 1998*, pages 181-192, 1998.

[20] L. Kobbelt and M. Botsch. An interactive approach to point cloud triangulation. In Eurographics 2000 Conference Proceedings, 2000.

[21] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The digital michelangelo project: 3d scanning of large statues. In *Siggraph 00 Conference Proceedings*, pages 131-144, 2000.

[22] M. Pauly and M. Gross. Spectral Processing of Point-Sampled Geometry. In *Siggraph 2001 Conference proceedings*, 2001.

[23] M. Pauly, R. Keiser, L. Kobbelt, and M. Gross. Shape Modeling with Point-Sampled Geometry. In *Siggraph 2003 ConferenceProceedings*, 2003.

[24] M. Stamminger and G. Drettakis. Interactive sampling and rendering for complex and procedural geometry. In *Eurographic Workshop on Rendering 2001*, pages 151.162, 2001.

[25] L. Westover. Footprint evaluation for volume rendering. In *Siggraph 1990 Conference Proceedings*, pages 367.376, 1990.

[26] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. EWA Splating. In IEEE Transactions On Visualization And Graphics. 2002

[27] M. Botsch, A. Wiratanaya, and L. Kobbelt. Efficient high quality rendering of point sampled geometry. In *Proc. Eurographics Workshop on Rendering 2002*, 2002.

第四章 实时阴影映射

阴影效果对于增强画面的真实感有着非常重要的作用，它可以反映画面中景物之间的相对位置，增加绘制效果的立体感和场景层次感。在计算机图形学领域中对于阴影绘制已经有了大量的研究工作，绘制高质量阴影的最主要问题是对于光栅化后的每一个象素如何正确地判断其可见点与场景中每一个光源的可见性。可编程图形硬件技术已经广泛应用于高质量阴影绘制的研究中，阴影映射(shadow mapping)是一种基于图像空间的阴影绘制技术，该方法原理简单，实现过程中可以通过充分发挥现有可编程图形硬件的强大功能，高效地完成实时阴影的绘制。本章着重介绍阴影映射的原理以及如何基于可编程图形流水线实现实时阴影映射。

4.1 阴影映射原理

阴影映射算法最早是由 Lance Williams[1] 于 1978 年提出的，其基本思想十分直观：一个物体之所以会处在阴影当中，是由于在它和光源之间存在着遮蔽物，或者说遮蔽物离光源的距离比物体要近，因此只要在绘制时能快速判断出场景中物体和光源之间的远近关系即可获得实时的阴影。它的算法流程分为两个步骤(Pass):

步骤 1：以光源作为视点，或者说在光源坐标系下面对整个场景进行绘制，目的是要得到一张场景中所有物体相对于光源的深度表(depth map，即通常所说的 shadow map)，深度表中每个象素的值记录了沿该光源入射方向场景中距离光源最近的景物采样点的深度值(z-buffer 的作用)。在这个步骤中我们感兴趣的只是每个象素上记录的深度值，所以不需要做任何光照计算，只要打开深度测试(z-test)和写深度值(z-write)的绘制状态即可。

步骤 2：将视点恢复到原来的正常位置，绘制整个场景，对每个象素在 pixel shader 中计算其可见点和光源之间的距离，然后将距离值和深度表中对应的值比较，以确定这个可见点是否处在阴影当中，接着根据比较的结果，对阴影区象素和非阴影区象素分别进行不同的光照计算，完成阴影效果的绘制。

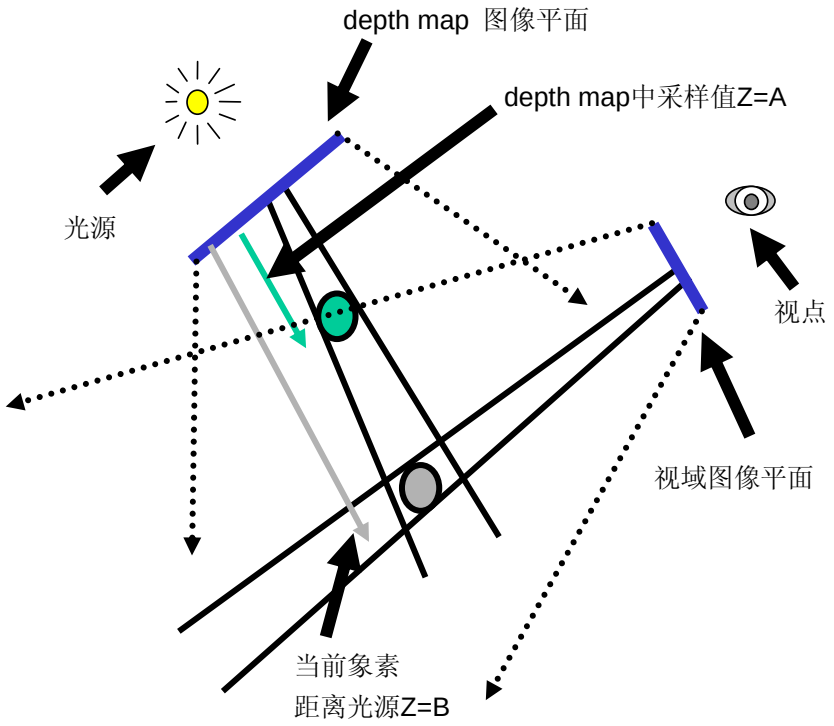


图 4.1[2] 当 A<B 时，当前象素处于阴影区中

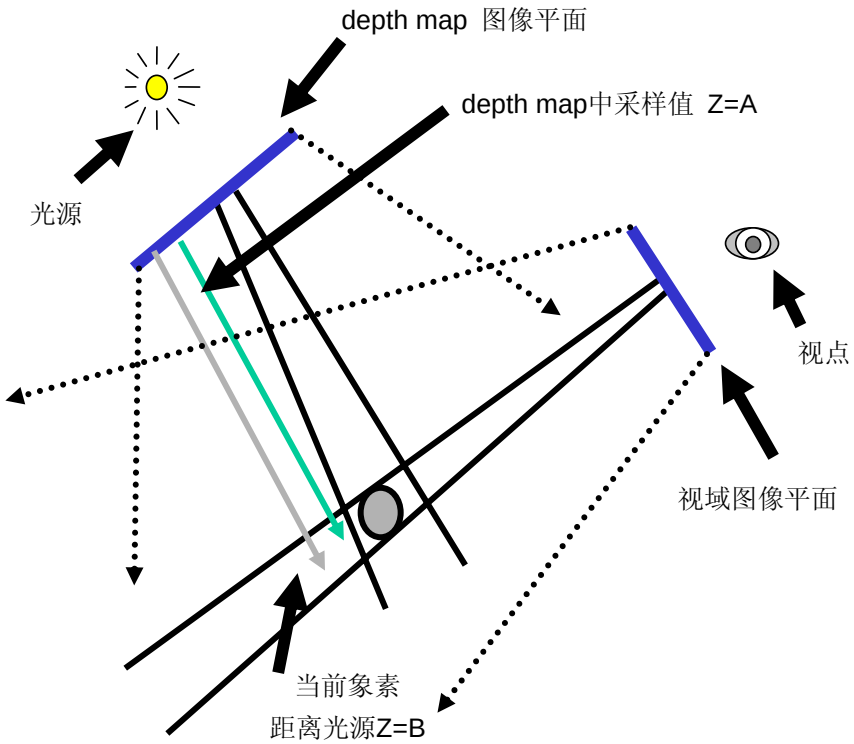


图 4.2[2] 当 A≅B 时 当前象素不在阴影区中

从以上算法步骤可以看出，步骤 1 以光源为视点绘制整个场景其实是对光源视域四棱锥进行了一次光源可见性的预计算，所生成的深度表为步骤 2 的深度判断做好准备。因为深度表的生成只和光源的位置以及场景中物体的位置有关，无论视点怎么运动，只要光源和物体的相互位置关系不变，深度表就可以被重复使用。步骤 2 中的核心任务是完成阴影判断，确定当前视域范围内的阴影区和非阴影区，如果当前象素上的可见点距离光源的距离比从深度表中采样得到的值大时，说明当前象素上可见点在光源坐标系中被其前面的景物采样点所遮挡，应该在阴影区中，判断过程如图 4.1 所示，反之如果当前象素距离光源的距离小于等于从深度表中采样得到的值时，则应该在非阴影区，判断过程如图 4.2 所示。

4.2 实时阴影映射的实现

4.2.1 实现步骤

目前的图形硬件已经提供了纹理(texture)和深度缓存(depth buffer)等很多基于图像空间的相关技术可以用来实现实时阴影映射，算法的具体实现步骤可分为：

(1) 首先创建一张作为绘制目标的纹理作为深度表。

当前 GPU 大都支持 16 位甚至 32 位浮点格式类型的纹理，而且在图形程序的执行过程中提供浮点精度的计算，因此可以选择 32 位浮点格式创建深度表纹理；当生成深度表时，对于每个象素只需要保存一个浮点深度值，所以只需要创建如 D3DFMT_32F 这样仅有一个颜色通道的纹理即可满足要求。

(2) 以光源位置为视点绘制场景生成深度表。

将(1)生成的深度表 设为渲染目标(render target)，算法基于 Shader 代码实现。在 vertex shader 中，首先将输入的模型顶点坐标变换到以光源位置为视点的投影坐标系下，然后 vertex shader 将经过投影变换的顶点坐标的 z 和 w 值以纹理坐标的方式传送给 pixel shader，这样保证经过光栅化的双线性插值后，每个象素都能有相应的 z 和 w 值。

在 pixel shader 中，计算深度 z / w 并将其输出，该值范围为 [0,1]，其中 0 表示以光源位置为视点的近裁减平面，1 表示远裁减平面。每个象素的深度值计算并输出后即生成了所需的深度表。

(3) 恢复正常视点位置，利用深度表绘制场景生成阴影效果。

恢复以前默认的渲染目标，将(2)中生成的深度表设为采样纹理。

在 vertex shader 中，为了实现深度判断，必须将当前顶点位置经过坐标变换到以光源为视点的投影坐标系下，坐标变换矩阵 M 如下：

$$M = M_{world} \bullet M_{LightView} \bullet M_{Light\,Proj}$$

上式中 M_{world} 是变换至世界坐标系的变换矩阵，$M_{LightView}$ 为以光源为视点的视域坐标变换矩阵，$M_{Light\,Proj}$ 为以光源为视点的投影坐标变换矩阵

将变换后的坐标以纹理坐标的形式输出给 pixel shader，由于后面还需要实现基于象素的光照计算(per-pixel lighting)，每个顶点的法向，纹理坐标等信息也需要以纹理坐标的方式输出，使光栅化后的每个象素都包含这些数据。

在 pixel shader 中，把每个象素上的可见点在光源坐标系下的坐标值的 x 和 y 变换至纹理坐标空间中生成深度表纹理的坐标 tex：

$$tex.x = 0.5 \cdot x / w + 0.5$$
$$tex.y = 0.5 - 0.5 \cdot y / w$$

式中 w 为坐标的齐次项，上式可以将在 [-1,1] 范围的投影坐标变换到 [0,1] 的纹理坐标空间内。之后使用 tex 采样深度表纹理，用当前象素上可见点在光源坐标系下的坐标的 z / w 和深度表上记录的深度值进行比较，如果 z / w 的值大于表上记录的深度值，则当前象素可见点处于阴影中，反之当前象素可见点不被阴影遮蔽。在正确判断出阴影后，将其效果作为乘积因子带入光照明计算公式中即可。

4.2.2 实现过程中存在的问题

(1) 阴影效果的绘制

绘制阴影的一般同时伴随着如 per-pixel lighting 的局部光照明效果绘制，一般使用如下局部光照明公式为：

$$Final\ Effect = (Ambient + Diffuse) * texture + Specular$$

式中 *Ambient, Diffuse, Specular* 分别代表环境光，漫反射和镜面反射的光照效果，而 *texture* 表示采样颜色纹理的结果。

假定已经确定当前象素是否位于阴影中，假定 *Shadow Term* 为代表阴影效果的因子，在阴影中 *Shadow Term* = 0，不在阴影中则 *Shadow Term* = 1。如果使用 *Final Effect = Final Effect * Shadow Term* 进行绘制会导致处于阴影中的象素是全黑的，但真实世界中的光源并没有完全被遮蔽，处于阴影中的象素可见点还可以通过周围环境的反射，散射等的作用获得光照，因此为了得到比较真实的阴影效果，在绘制时建议使用如下

局部光照明公式:

$$Final\ Effect = (Ambient + Diffuse * Shadow\ Term) * texture + Specular * Shadow\ Term$$

(2) 浮点精度

因为深度表中存储的深度值是浮点数且精度是有限的，同时深度的比较即两个浮点数的比较本身就存在精度问题，特别是判断两者是否相等时；当场景中阴影投射者和阴影接收者距离很近的情况下，深度表的精度就可能无法正确判断两者的深度值的差异，可能导致错误的阴影效果，如下图所示:

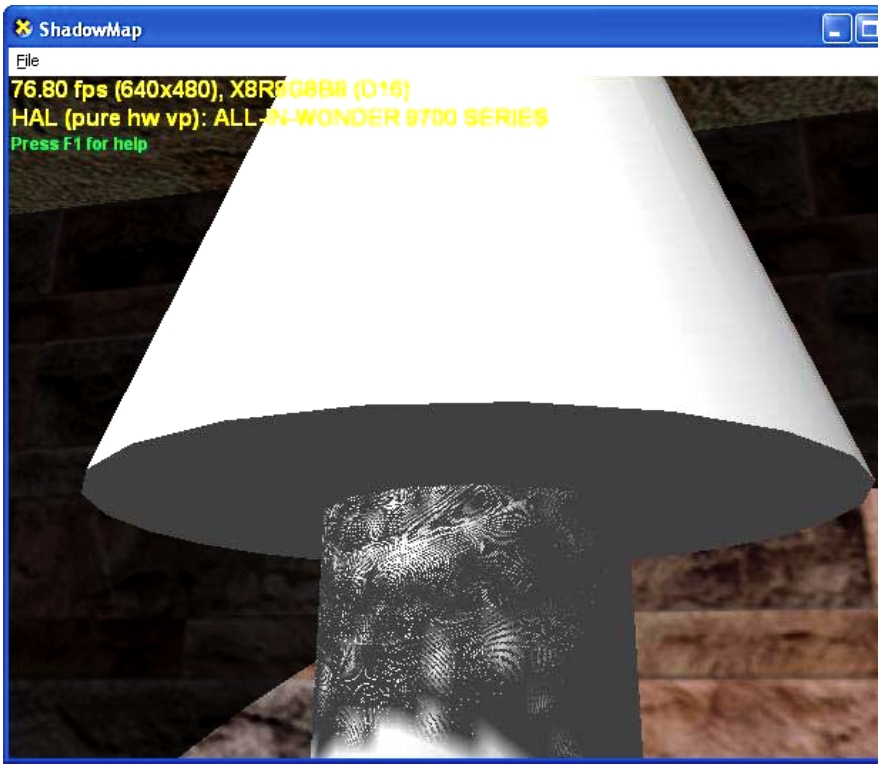


图 4.3 因精度不足而产生的阴影错误

解决上述问题的一个方法是在步骤 2 中进行深度比较时，人为在判断等式的任意一边加上或者减去一个很小的浮点数。

(3) 阴影的边缘走样

阴影映射还存在着所有纹理相关技术面临的共同问题－走样。根据采样定理，只有纹理分辨率小于或者等于物体的实际分辨率时才不会失真，而当一幅很大的纹理被

贴到尺寸比它小的物体上时，会出现一个象素(pixel)覆盖多个纹元(texel)的情况，这时要准确的再现这个象素的颜色信息，就要综合考虑所有被它覆盖的纹元产生的影响，这是各种纹理滤波方法最基本的原理。

但是由于深度表中每一象素记录的深度值并非沿该方向局部区域的平均深度值，所以不能像一般的纹理那样把各个 mip-map 层次都预先计算好放在显存中，而必须实时处理出现的走样。有一种方法是在 pixel shader 中进行深度判断时，使用复杂的运算指令同时将当前象素的深度值和深度表中对应的 4 个相邻纹元的深度值都进行比较，实现双线性插值滤波，其效果如下图所示：

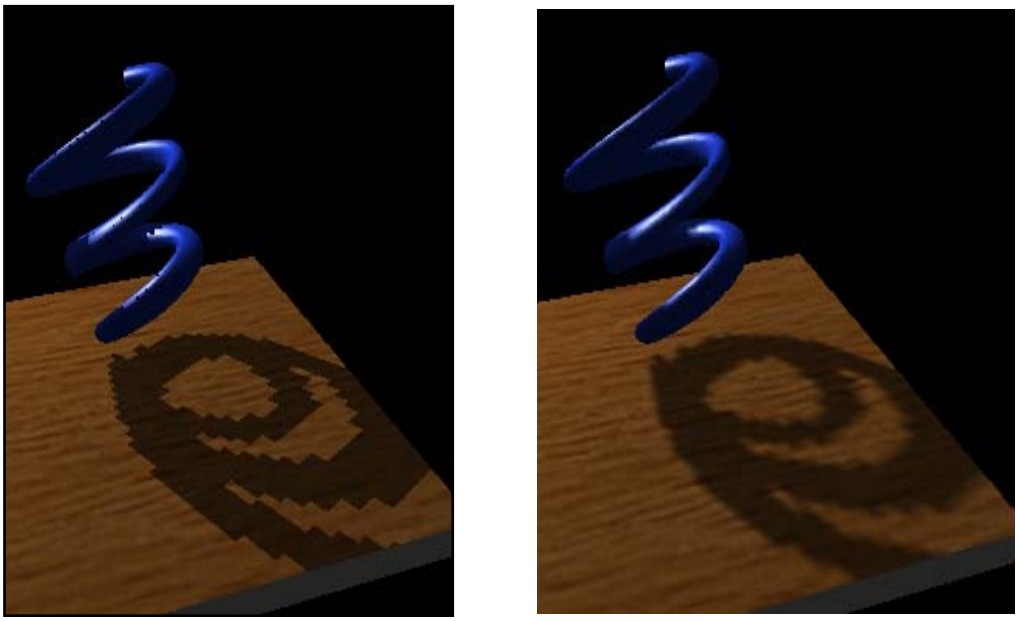


图 4.4 阴影映射效果：未使用双线性滤波的效果 (左图)和使用了双线性滤波的效果(右图)。为了对比清晰，绘制时使用分辨率非常低的 深度表。

但是这样 Shader 的计算量很大，会大幅降低绘制的效率。走样的问题在纹理分辨率小于屏幕分辨率的时候仍然存在，这时多个象素会被投射到同一个纹元上面，虽然从再现纹理的角度来说并不存在失真，但是由于多个象素共用同一个纹元值，走样问题还是存在。

到目前为止仍然没有一种纹理滤波技术可以从根本上解决这样的走样问题，因为从数学上讲，不可能通过增加运算步骤得到比原始数量更多的信息。近年来，为了解决阴影映射的走样问题，出现了大量优秀的研究工作，效果比较好的是 adaptive shadow map(ASM)[10] 和 perspective shadow map(PSM)[11]。两者的基本原理都是在可能产生锯齿的地方人为提高采样率，保证一个象素尽量对应一个纹元， 两种方法的区别在于 ASM 是在 shadow 边缘处增加采样率，而 PSM 则是在靠近视点的地方。

4.3 硬件阴影映射实现

我们可以充分利用可编程图形流水线所提供的功能实现阴影映射算法，完成阴影的实时绘制。4.2 节中所提到的阴影映射实现过程中是使用 Shader 代码实现的，其中当进行深度判断时所使用的代码如下：

if (*z* / *w* > 采样*depthmap*得到的深度值)
ShadowTerm = 0.0;
else
ShadowTerm = 1.0

分析这部分代码可以看出，上述 Shader 代码在执行时需要进行条件判断。在 2.0 及其以下版本的 Shader 中条件分支指令的两部分代码其实是都会被执行的，只是执行完后通过一个寄存器值的判断来选取满足分支条件的结果，因此当条件分支的两部分指令比较复杂或者 Shader 中存在大量条件分支时，会影响 Shader 代码执行的效率，当在 Shader 中进行双线性滤波时，需要执行 4 次这样的条件判断来完成与深度表中 4 个相邻纹元深度值的比较，会明显降低绘制速度。

为了解决这个问题，图形硬件厂商为显卡提供新的特性以通过硬件而不是软件的方式来进行深度判断，Nvidia 公司从 GeForce3 系列 GPU 开始就支持一种称为硬件阴影映射的特性 [5,6]，并为 OpenGL 和 Direct3D 均提供了实现的方法。其中 OpenGL 的实现是依靠 SGIX_shadow 和 SGIX_depth_texture 两个新的扩展来完成，而 Direct3D 的实现则需要通过创建一种新的纹理格式来创建深度表。下面会对该方法基于 Direct3D 的具体实现进行介绍。

Nvidia 公司的驱动程序从 21.81 版本之后开始支持创建两种特殊格式的纹理，它们是 24-bit 的 D3DFMT_D24S8 和 16-bit 的 D3DFMT_D16，利用这两种格式可以实现硬件阴影映射方法。其步骤为：

(1) 首先可以使用以下代码检查所使用的图形硬件是否支持创建 D3DFMT_D24S8 格式的纹理，其功能是作为深度模板缓存(depth-stencil surface)。

```
HRESULT hr = pD3D->CheckDeviceFormat(
D3DADAPTER_DEFAULT,   //默认的显卡
D3DDEVTYPE_HAL,       //纯硬件支持标志
```

```
D3DFMT_X8R8G8B8,        //显示模式
D3DUSAGE_DEPTHSTENCIL, //功能是作为深度模板缓存
D3DRTYPE_TEXTURE,       //创建对象是纹理
D3DFMT_D24S8            //所创建纹理的格式
);
```

一旦 hr 返回值为 S_OK，则表明硬件支持该功能，可以使用以下代码创建一张纹理作为深度表：

```
pD3DDev->CreateTexture(texWidth, texHeight, 1,
D3DUSAGE_DEPTHSTENCIL, D3DFMT_D24S8,
D3DPOOL_DEFAULT, &pTex);
```

请注意此处所创建深度表纹理的功能是 D3DUSAGE_DEPTHSTENCIL，而不是 D3DUSAGE_RenderTarget，并不能直接作为渲染目标(render target)，因此需要创建一张用做渲染目标的颜色纹理，同时在绘制时将创建的深度表设为该颜色纹理的深度模板缓存。

(2) 以光源位置为视点绘制场景生成深度表。

这一步和以前步骤 1 的思想是一样的，但因为深度表纹理此处是深度模板缓存，而不是 render target，所以此处可以不需要输出颜色值到颜色缓存(Frame buffer)，而只需要渲染场景得到深度就可以了，可以通过将绘制状态 D3DRS_COLORWRITEENABLE 的值设为 false 来实现，这样的实现可以极大地降低总线的传输和 Shader 代码的负担，Nvidia 官方数据显示在只渲染深度不渲染颜色值的情况下，GPU 的绘制速度会提高一倍。vertex shader 只需要简单地输出 oPos，pixel shader 可被设成 null，硬件便会自动在深度表中记录绘制结果中每个象素的深度值。

(3) 恢复正常视点位置，利用深度表绘制场景生成阴影效果。

前面已经提到在绘制阴影效果时 Shader 的深度判断使用条件分支会带来性能的下降，硬件阴影映射同样需要进行条件判断，但不再需要执行复杂的 Shader 指令而是直接利用硬件提供的功能来完成，可以使用 Shader 指令 tex2DProj(HLSL)来实现该功能，该指令如下：

ret = tex2DProj(s , t)

式中 s 为纹理采样寄存器，t 为纹理坐标寄存器，对于 tex2DProj 来说 t 应为 4 元寄存器，当对二维纹理采样时，该指令会使用 t.x / t.w 和 t.y / t.w 作为纹理坐标去采样得到返回值 ret。Nvidia 支持的硬件阴影映射对该指令功能做了扩展，当采样满足硬件阴影映射格式(如 D3DFMT_D24S8)要求的深度表时，tex2DProj 会采用 t.x / t.w 和 t.y / t.w 作为纹理坐标去采样深度表，同时将采样得到的深度值和 t.z / t.w 进行比较，如果采样得到的深度小于 t.z / t.w，则返回值 ret.xyz 三个通道的值均为 0.0，否则 ret.xyz 三个通道的值均为 1.0。因此可以仅通过一个纹理采样的指令 ShadowTerm= tex2DProj(s, t).r 便可完成以前冗长的条件分支判断，这就是硬件阴影映射可以提高绘制速度的最主要的原因。以下为具体实现：

在 vertex shader 中，需要对顶点进行坐标变换。首先将顶点坐标利用矩阵 $M = M_{world} \bullet M_{LightView} \bullet M_{Light\,Proj}$ 变换到以光源为视点的投影坐标系下。令变换后的坐标为 $T(x, y, z)$；然后做如下变换：

$$
\begin{aligned}
T.x &= T.x \bullet 0.5 + T.w \bullet 0.5 + T.w \bullet TexBias\_x \\
T.y &= T.w \bullet 0.5 - T.y \bullet 0.5 + T.w \bullet TexBias\_y \\
T.z &= T.z - T.w \bullet shadow\_epsilon
\end{aligned}
$$

上式中 $TexBias\_x$ 和 $TexBias\_y$ 是纹理坐标的在 x, y 方向上的偏移，加上偏移量的原因是在 Direct3D 中屏幕象素坐标和纹理坐标并不是一一对应的，两者之间存在半个象素长度上的偏移 [3]。$shadow\_epsilon$ 是为了避免深度判断过程中的精度误差而引入的一个很小的浮点数。经过以上变换后的坐标即为 pixel shader 中 tex2DProj 指令将使用的 4 元纹理坐标。

在 pixel shader 中，仅需要利用 ShadowTerm= tex2DProj(s, t).r 即可确定当前象素是否处于阴影中，然后将 ShadowTerm 带入光照明计算公式即可得到正确的阴影效果。如果要实现双线性滤波，硬件阴影映射还提供了一个加速功能，只需在采样深度表纹理时将采样寄存器的纹理滤波方式设为 D3DFILTER_LINEAR，且仅使用一条纹理采样指令，图形硬件便会自动帮我们完成双线性滤波的操作，返回滤波后的值，在提高阴影质量的同时保证了非常高的绘制效率。

4.4 实验结果和讨论

我们利用 Visual C++ 和 DIRECTX 9.0b SDK 在一台配备 Intel P4 2.8GHz、Nvidia GeForce 5950 显卡、1G 内存的微机上实现了普通的阴影映射算法和硬件阴影映射算法。其效率对比表格如下：

表 4.1 两种阴影映射方法效率对比

场景	光源数目	普通阴影映射(FPS)	硬件阴影映射(FPS)
场景 1	1	160	220
场景 2	1	125	176

由表中数据可以看出在相同场景和光源条件下，硬件阴影映射算法比普通阴影映射算法的效率大约提高 30%-40%。我们还对使用双线性滤波的情况下两者的效率做了测试，结果如下表所示：

表 4.2 两种阴影映射方法均采用双线性滤波时的效率对比

场景	光源数目	普通阴影映射(FPS)	硬件阴影映射(FPS)
场景 1	1	110	217
场景 2	1	85	173

由表中数据可以看出相同场景和光源条件下，使用双线性滤波的硬件阴影映射方法效率上基本和以前一样，而普通阴影映射方法的效率则明显下降，此时硬件阴影映射方法的优势体现的十分明显，效率几乎普通阴影映射方法的 2 倍。这主要得意于图形硬件本身将双线性滤波功能固化，硬件阴影映射方法只需将采样深度表的模式改为线性插值模式，经过一次硬件纹理采样就可以实现双线性滤波。

下表将两种阴影映射方法进行总结和对比：

表 4.3 两种阴影映射实现方法的对比

绘制过程	普通阴影映射	硬件阴影映射
步骤 1	创建一张 D3DFMT_32F 格式纹理作为深度表，将其设为渲染目标。以光源位置为视点绘制整个场景，在 Shader 中将每个象素相对于光源的距离以颜色信息的方式记录到深度表中。	创建一张 D3DFMT_D24S8 格式的纹理作为深度表，将其设渲染目标的深度模板缓存(depth-stencil surface)。以光源位置为视点绘制整个场景，可将绘制状态 COLORWRITEENABLE 的值设为 false，不要输出颜色信息，绘制时深度信息会记录到深度表。
步骤 2	恢复正常视点对场景进行绘制，将步骤 1 中生成的深度表设成纹理，在 Shader 中采样深度表并进行深度判断后确定阴影区域，然后将阴影项带入光照明计算。	恢复正常视点对场景进行绘制，将步骤 1 中生成的深度表设成纹理，使用 Shader 指令 tex2DProj 采样深度表，将得到的值直接作为阴影项带入光照明计算即可。

总的来说，硬件阴影映射的确是一种高效的阴影绘制技术，在虚拟现实、游戏等很多方面都有广泛应用，虽然目前只有 Nvidia 的图形硬件支持这项功能，还不具备跨硬件平台的能力，但将来 OpenGL 和 Direct3D 很可能将这项技术纳入其标准中，要求所有硬件厂商支持，这样使用起来会更加方便，基于它开发的程序也更具备扩展性。

参考文献

[1] Lance Williams. Casting curved shadows on curved surfaces. In *Proceedings of SIGGRAPH '78*, pages 270-274, 1978.

[2] Mark Kilgard. Shadow Mapping with Today's Hardware. Technical presentation: http://developer.nvidia.com/view.asp?IO=cedec_shadowmap.

[3] Craig Duttweiler. Mapping Texels to Pixels in Direct3D. http://developer.nvidia.com/view.asp?IO=Mapping_texels_Pixels.

[4] Cass Everitt. Interactive Order-Independent Transparency. Whitepaper: http://developer.nvidia.com/view.asp?IO=Interactive_Order_Transparency.

[5] Mark Segal and Kurt Akeley. The OpenGL Graphics System: A Specification (Version 1.2.1). www.opengl.org

[6] OpenGL Extension Registry. http://oss.sgi.com/projects/ogl-sample/registry/.

[7] Mark Segal, et al. Fast shadows and lighting effects using texture mapping. In *Proceedings of SIGGRAPH '92*, pages 249-252, 1992.

[8] Yulan Wang and Steven Molnar. Second-Depth Shadow Mapping. UNC-CS Technical Report TR94-019, 1994.

[9] Andrew Woo, P. Poulin, and A. Fournier. "A Survey of Shadow Algorithms," IEEE Computer Graphics and Applications: vol 10(6), pages 13-32, 1990.

[10] Randima Fernando, Sebastian Fernandez, Kavita Bala, Donald P. Greenberg. Adaptive Shadow Maps. In Proceedings of SIGGRAPH '2001: 387-390

[11] Marc Stamminger and George Drettakis. Perspective Shadow Maps. In Proceedings of SIGGRAPH '2002: 557-562

[12] Cass Everitt, Ashu Rege and Cem Cebenoyan. Hardware Shadow Mapping. Nvidia Technical Reprot

第五章 总结与未来工作

5.1 工作总结和体会

在硕士研究生工作期间，作者一直从事可编程图形硬件相关技术的研究工作，同时阅读了大量与之相关的文献和资料，在此总结一些自己的体会和感受。

(1) 对于计算机图形学领域特别是绘制领域而言，可编程图形硬件可以说是必不可少的实现工具；目前绝大部分的显卡在硬件结构中已经不再有固定流水线，而只有可编程流水线，其固定流水线的绘制其实是用可编程流水线模拟出来的，因此在进行绘制时使用可编程图形流水线肯定会带来更好的性能表现；但即便最新的显卡 Nvidia GeForce 6800 仍然存在着一些功能上的限制，如该 GPU 上只有 8 级的堆栈结构，所以 Shader 代码中函数调用的层次不能超过 8 层，这样就限制了递归程序的执行，经典的图形学应用－光线跟踪现在虽已可以利用 GPU 实现，然而将递归展开执行的实现方式导致对于中等复杂度的场景，实现的结果仍只能达到交互的速率；实际上 GPU 执行很快也是因为它的流水线比较简单且具有并行性，不像 CPU 那么复杂，毕竟绘制效果是给肉眼看的，无需达到非常高的精确程度。

(2) 可编程图形硬件具有并行执行的体系结构，所以很多通用计算领域的算法都希望能借助 GPU 的强大运算性能来执行，目前已有了大量这方面的研究和应用。可惜现在的可编程图形硬件还只能提供 IEEE 单精度的浮点支持，一些精度要求较高的通用计算往往需要 IEEE 双精度的浮点支持，这是当前限制可编程图形硬件在高性能计算方面得到实质性应用的一个最主要原因；此外现在的可编程图形硬件的显存容量最大仅为 512M，这对于很多计算应用也是不够的，而且很多应用程序都需要在内存和显存之间频繁地交换数据，目前的 AGP8x 总线技术在回读数据时仍然效率不高。令人欣慰的是，随着可编程图形硬件的飞速发展，硬件厂商两年内就会提供 IEEE 双精度的浮点支持，显存容量也会不断增加，同时现在已经出现了 PCI-Express 这样传输速率对称的总线技术可以解决回读速度低下的问题，还有一个更有利的驱动因素就是类似 Brook，Sh 这样的流处理机编程环境的兴起，使得不熟悉图形流水线的人同样可以充分利用 GPU 提供的强大计算能力，这样会吸引更多不同领域的研究者关注和利用 GPU。可以想象，出现基于可编程图形硬件的高性能计算系统只是时间长短的问题。

(3) 就图形 API 方面，OpenGL 和 Direct3D 在相当长一段时间内会是 GPU 编程的主要方式，OpenGL 作为事实上的工业标准，多年来已经广泛为学术界和工业界所接受。2000 年后，伴随着可编程图形硬件的发展，Direct3D 开始迅速崛起，到目前为止它对可编程硬件特性的支持应该说已经领先于 OpenGL。对于开发人员而言，Direct3D 虽然编程方式较 OpenGL 复杂，但它的文档资料和例程非常齐全，特别是将 GPU 的操作封装成了 COM 对象的属性，使用起来非常方便，这对于很多刚接触 GPU 的新手是十分友好的；当然 OpenGL 1.5 后对于可编程硬件的支持取得了很明显的进步，全面提供对高级语言 OpenGL shading language 的支持，但目前相比于 Direct3D 而言仍有不完善的地方，如不支持 Shader 指令的调试模式等。希望马上就要推出的 OpenGL 2.0 标准能够做的更好。

(4) 可编程图形硬件在计算机图形学领域将会成为一种普遍的实现工具，同时目前的可编程图形硬件在功能上越来越容易理解和使用，很多世界一流的图形学实验室均已开设了相关课程，并列入学生的培养计划；可编程图形硬件技术也是促进图形领域学术界和工业界之间沟通的桥梁，学术界的研究者在使用可编程硬件过程中不停地对 GPU 的功能提出新的要求，从而推动硬件厂商们不断推出功能更加强大的 GPU；功能更强大的 GPU 反过来又会促进更多新的研究成果，这样相辅相成的良性循环一定会推动整个计算机图形学领域的进步。

5.2 未来工作

作者的未来工作仍然会以可编程图形硬件技术为基础，将可能主要集中在以下几个方面：

(1) 基于可编程图形硬件的 High dynamic range(HDR)图像或者视频压缩。目前 GPU 的强大处理能力已经可以对视频进行实时处理，浮点格式纹理为 HDR 数据提供了载体，同时 GPU 提供的丰富的二维纹理操作可以很容易地过渡到对图像或者视频的处理，如果可以实现实时对 HDR 图像或者视频进行频谱压缩，将会很有意义。

(2) 实时软阴影(soft shadow)。前面第四章中详细介绍了基于可编程硬件的阴影映射算法，该算法产生的是硬阴影(hard shadow)，但我们真实世界中见到的阴影一般都是软阴影，因此实时绘制软阴影的效果是有实际意义的，目前已经有了一些关于这方面的研究工作，作者准备以阴影映射的原理出发考虑一种全新的基于可编程图形硬件

的实时软阴影绘制方法。

(3) 基于可编程图形硬件的全局光照明算法的研究。全局光照明可以真实地再现原始场景，但大部分全局光照明的实现技术在目前的绘制实现中仍然不能达到实时，如光线跟踪(ray tracing)，光子投射(photon mapping)等，极大地限制了它们应用的范围，借助日益强大的 GPU 功能，希望可以找到实现实时全局光照明算法的有效途径。

致　　谢

本文是在导师彭群生教授和陈为副教授的悉心指导下完成的。在攻读硕士学位期间，两位导师在学习和研究方面给了我精心的指导和莫大的帮助。导师严谨的治学态度、渊博的学识、为人师表的品格、孜孜不倦的进取精神、敏捷的科学思维，以及和学生共同探讨科研难点等等，这些都一直感染并激励着我，并将使我终身受益。同时，两位导师对本人生活、学习和思想上也给予了朋友般的关心和帮助，让人如沐春风，深为感动，值此论文完成之际，谨向彭老师和陈老师致以衷心感谢和诚挚敬意！

在硕士科研工作过程中，还得到了鲍虎军教授的倾力指导和热情帮助，对我研究的方向提出了很有价值的意见和建议，在此向鲍老师表示由衷的感激和深深的敬意！在微软亚洲研究院实习期间，刘新国研究员，郭百宁研究员在科研和学习等很多方面都对我悉心指导，使我受益匪浅，在此表示由衷的感谢。在学习期间和论文撰写过程中，还得到了王章野副教授，郑文庭副教授，胡敏老师，王进老师的指导和帮助，在此深表谢意！我还要感谢江钟鼎，刘刚，汤峰等多位师兄在我读研期间给予的大量帮助和鼓励！

感谢江照意、金剑秋、吴向阳、胡国飞、苗兰芳、王长波、缪永伟、柴登峰、郭延文、肖春霞、王锐、刘春晓、宋成芳、刘世光、张龙、管宇、张涛、马瑞金、刘峰、曾阳艳、周廷芳、武凤霞、林先茂、曾翔、应逸亭、谢仲毅、吴岚、张晓等课题组的同学对本人的论文工作提供了大力帮助。特别感谢马瑞金、张龙时常和我一起调试 GPU 相关的程序。感谢室友付超、黄成、范汉生和我一起度过了快乐时光，使我的硕士研究生生活更加丰富多彩！

感谢所有关心我的老师、同学和亲朋好友！

感谢父母一直以来对我的关心和爱护，他们给了我莫大的鼓励和默默的支持。最后要感谢我的女友柴燚在我完成学业期间所给予的无私帮助和热情鼓励！

最后祝愿我的恩师们身体健康、祝愿实验室的师兄师姐师弟师妹们学业有成！

董朝 于求是园

二零零五年三月

攻读硕士学位期间发表论文情况

(1) Zhao Dong, Wei Chen, Hujun Bao, Hongxin Zhang, Qunsheng Peng, "Real-time Voxelization for Complex Polygonal Models", In *Proceedings of the 12th Pacific Confe*rence on Computer Graphics and Applications (Pacific Graphics 2004).

(2) Zhao Dong, Wei Chen, Long Zhang and Qunsheng Peng. “Balancing CPU and GPU: Real-time Visualization of Large Scale 3D Scene”. In *The* T*hird International Conference on Grid and Cooperative Computing (GCC 2004), VVS workshop (Visualization and Visual Steering),* Wuhan, October. 2004. Lecture Notes in Computer Science, Springer Verlag. (Cited by SCI)

(3) 张龙, 董朝, 陈为, 彭群生. “大尺寸点模型自适应实时高质量绘制”. 《计算机学报》 2005 , 28(2).