《Real-Time Rendering 3rd》
提炼总结

毛星云（浅墨）

2018 年 3 月

前言

在实时渲染和计算机图形学领域,《Real-Time Rendering》系列书籍一直备受推崇。有人说，它是实时渲染的圣经，也有人说，它是绝世武功的目录。

其实《Real-Time Rendering》很像一整本图形学主流知识体系的论文综述，它涵盖了计算机图形和实时渲染的方方面面，可做论文综述合集了解全貌，也可作案头工具书日后查用。

正因如此，初学者直接一字一句读它，其实多少学习坡度会比较陡峭。而在了解了对应内容的大概概念之后，通过检索资料来进行延伸学习，或者在书中介绍晦涩难懂的时候，通过检索资料从侧面来学习，效果会更佳。

对此，我写了一个系列专栏《【《Real-Time Rendering 3rd》 提炼总结 】》，共 10 多篇文章，对《Real-Time Rendering 3rd》一书中渲染相关的章节进行了核心内容的梳理，也加上了不少个人的理解与总结。

该系列文章在知乎专栏和 CSDN 等站点发布以来,得到了不少朋友的赞许、鼓励与支持。在此，对他们表达我衷心的感谢。

而本书，即是系列专栏《【《Real-Time Rendering 3rd》 提炼总结 】》的合辑和汇编，全书共 9 万 7 千余字。你可以把它看做中文通俗版的《Real-Time Rendering 3rd》，也可以把它看做《Real-Time Rendering 3rd》的解读版与配套学习伴侣，或计算机图形和实时渲染相关内容的入门以及中阶读物，也可以作为快速检索的工具书之用。

本书的特点：

- 纯文字版 PDF，支持全文搜索、快速检索
- 按照纸质出版物的标准进行了排版
- 拥有高清的配图
- 有一点即达对应章节的详细目录
- 有精确到每章每节的书签，非常适合快速检索

在内容方面，全书按照系列专栏的顺序正序收录，分为十二章：

- 第一章 全书知识点总览
- 第二章 图形渲染管线

- 第三章 GPU 渲染管线与可编程着色器
- 第四章 图形渲染与视觉外观
- 第五章 纹理贴图及相关技术
- 第六章 高级着色：BRDF 及相关技术
- 第七章 延迟渲染的前生今世
- 第八章 全局光照：光线追踪、路径追踪与 GI 技术进化编年史
- 第九章 游戏开发中基于图像的渲染技术总结
- 第十章 非真实感渲染(NPR)相关技术总结
- 第十一章 游戏开发中的渲染加速算法总结
- 第十二章 渲染管线优化方法论：从瓶颈定位到优化策略
- 附录：《Real-Time Rendering 3rd》核心知识思维导图

也由于时间有限，水平有限，书中也许依然存在未发现的错误，敬请谅解。

当你觉得《Real-Time Rendering 3rd》英文原版硬啃不下来的时候，对照本书一起阅读，也许会事半功倍。而对于想快速入门实时渲染的朋友，翻翻本书，应该也会有所收获。

希望这本书，能对热爱游戏开发，计算机图形学和实时渲染的朋友们有所帮助。

最后，如果你希望找到我，以下是我常活跃的站点和联系方式。

知乎：https://www.zhihu.com/people/mao-xing-yun
知乎专栏：https://zhuanlan.zhihu.com/game-programming
GitHub：https://github.com/QianMo
CSDN 博客：http://blog.csdn.net/poem_qianmo
微博：https://weibo.com/u/1723155442
邮箱：happylifemxy#163.com（#替换成@）

祝好。

浅墨

2018 年 3 月于深圳

目录

第一章 全书知识点总览

1.1 《Real-Time Rendering 3rd》其书

在实时渲染和计算机图形学领域，《Real-Time Rendering 3rd》这本书一直备受推崇。有人说，它实时渲染的圣经。也有人说，它是绝世武功的目录。

诚然，《Real-Time Rendering 3rd》这本书的世界观架构宏大，基本涵盖了计算机图形学的方方面面，可谓包罗万象。概念讲得清楚明了，有丰富的论文引用，可供作为工具书查阅，深入某细分领域继续学习使用。

当然，如果我们吹毛求疵，那么也可以说，正因这本书包罗万象，由于篇幅受限，就会拥有一个缺点，就是大而不精。由于篇幅，很多知识点到为止，无法展开讲解，缺少更多范例，这就会让初学者读起来理解坡度稍为陡峭。但我们知道，图形学和实时渲染领域的知识浩如

烟海，就算是写个字典式的总览，这本书的篇幅也已经达到了惊人的 1047 页，要是再写详细一些，估计至少得 3000 页了。

总之，《Real-Time Rendering 3rd》这本书，可谓图形学界“九阴真经总纲”一般的存在，当世武功的心法口诀，尽数记载。它涵盖了计算机图形和实时渲染的方方面面，可做论文综述合集了解全貌，也可作案头工具书日后查用。

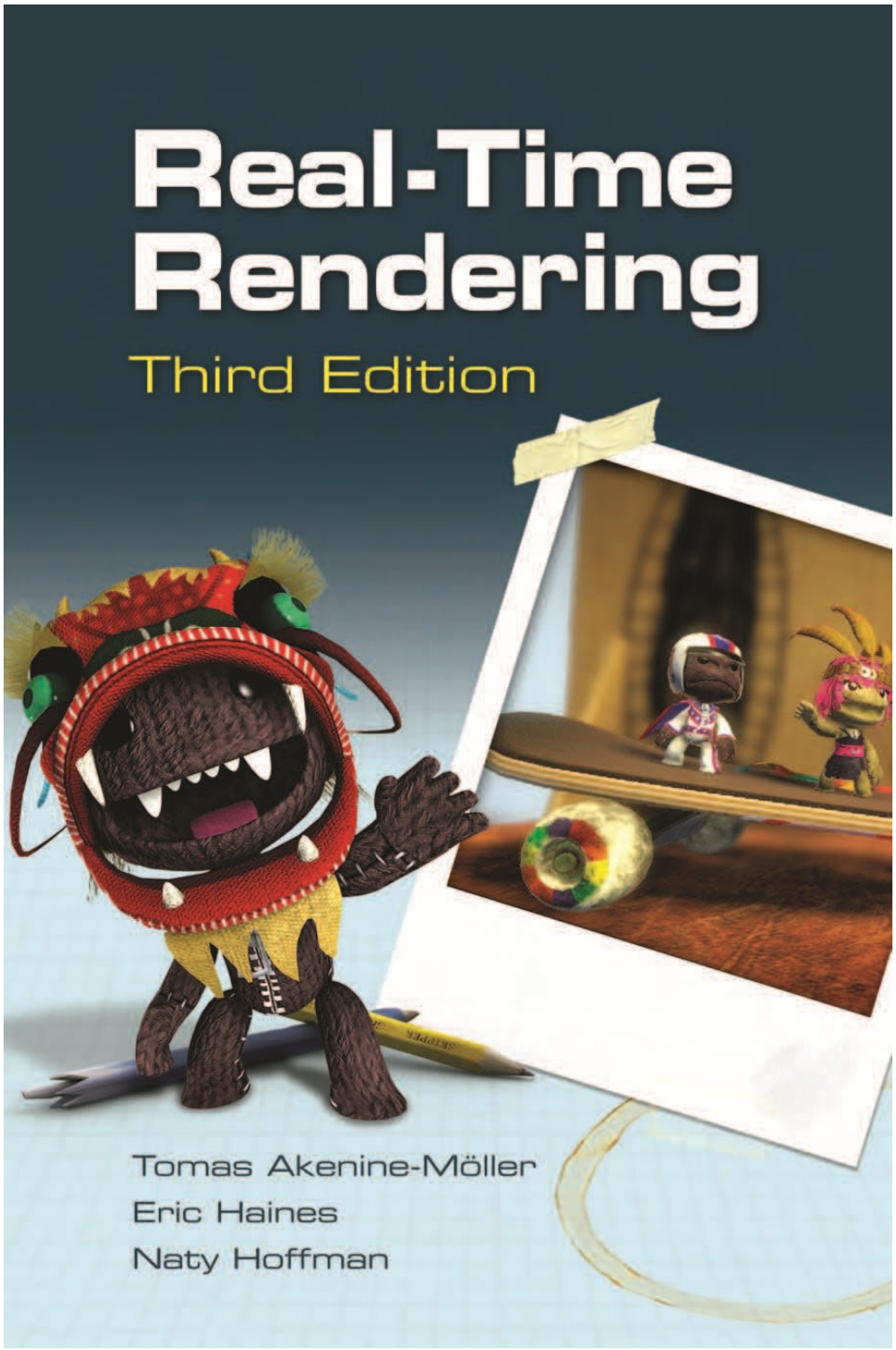


图 1 《Real-Time Rendering 3rd》 封面

1.2 相关背景

《Real-Time Rendering 3rd》出版于 2008 年，至今已经 9 年之久，但丝毫不能减弱它作为实时渲染界泰斗之作的重量。

这本书第三版目前没有中译版，只有第二版（英文原版出版于 2002 年）的中译版。网络上有一些第三版的翻译，但大多翻译到第四章后就没有下文。

另外，《Real-Time Rendering 4th》将于 2018 年 8 月 15 面世。目前 Amazon 上已经有了其预售页面: https://www.amazon.com/Real-Time-Rendering-Fourth-Tomas-Akenine-M%C3%B6ller/dp/1138627003/ref=sr_1_2?ie=UTF8&qid=1519278819&sr=8-2&keywords=Real-Time+Rendering

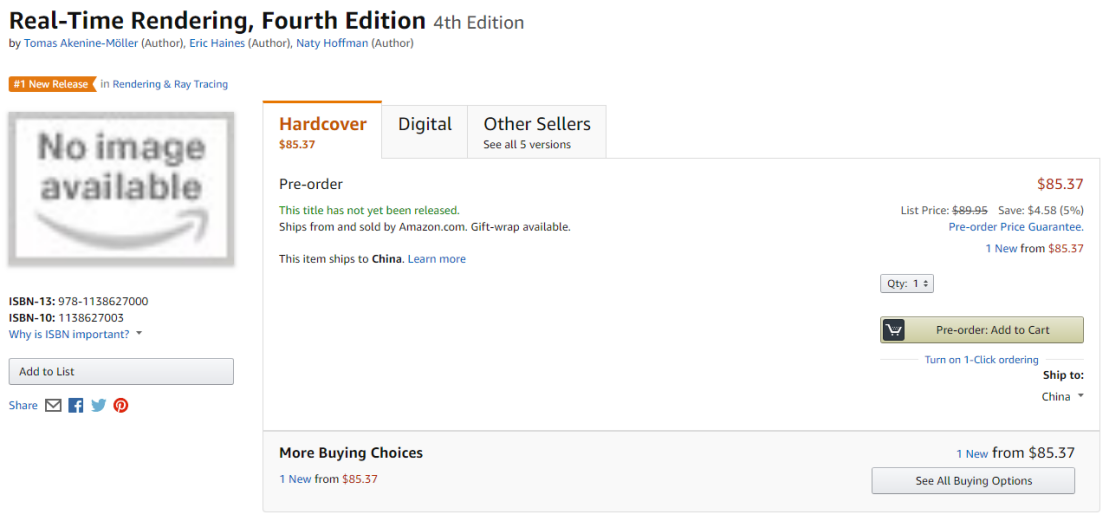


图 2 《Real-Time Rendering 4th》 的 Amazon 预售页面

1.3 《Real-Time Rendering 3rd》全书知识点总览

上文已经说到，《Real-Time Rendering 3rd》这本书，可谓图形学界“九阴真经总纲”一般的存在，当世武功的心法口诀，尽数记载。

而当我画完这张思维导图的时候，仔细看了看，还真有点“九阴真经总纲”图解的感觉。（建议另存为后放大查看）


Real-Time Rendering 3rd
实时渲染图形学 第三版
Chapter 1 Introduction 简介
1.1 Content 全书概览
1.2 Notation And Definitions 符号和定义
Chapter 2 The Graphics Rendering Pipeline 图形渲染管线
2.1 Architecture 架构
2.2 The Application Stage 应用程序阶段
2.3 The Geometry Stage 几何阶段
2.4 The Rasterizer Stage 光栅化阶段
2.5 Through The Pipeline 从流管线内容
2.6 Conclusion 总结
Chapter 3 The Graphics Processing Unit 图形处理器
3.1 Gpu Pipeline Overview Gpu 管线概述
3.2 The Programmable Shader Stage 可编程着色阶段
3.3 The Evolution Of Programmable Shading 可编程着色的进化
3.4 The Vertex Shader 顶点着色器
3.5 The Geometry Shader 几何着色器
3.6 The Pixel Shader 像素着色器
3.7 The Merging Stage合并阶段
3.8 Effects 特效
Chapter 4 The Graphics Processing Unit 图形处理器
4.1 Basic Transforms 基本变换
4.2 Special Matrix Transforms And Operations 特殊矩阵变换与运算
4.3 Quaternions四元数
4.4 Vertex Blending 顶点混合
4.5 Morphing 渐变
4.6 Projections 投影
Chapter 5 Visual Appearance 视觉外观
5.1 Visual Phenomena 视觉现象
5.2 Light Sources 光源
5.3 Material材质
5.4 Sensor 传感器
5.5 Shading 着色
5.6 Aliasing And Antialiasing 走样和反走样
5.7 Transparency, Alpha, And Compositing 透明度、Alpha值、以及合成
5.8 Gamma Correction 伽玛校正
Chapter 6 Texturing 纹理贴图
6.1 The Texturing Pipeline 纹理管线
6.2 Image Texturing 图像纹理贴图
6.3 Procedural Texturing 程序贴图
6.4 Texture Animation 纹理动画
6.5 Material Mapping 材质贴图
6.6 Alpha Mapping Alpha贴图
6.7 Bump Mapping 凹凸贴图
Chapter 7 Advanced Shading 高级着色
7.1 Radiometry 辐射度
7.2 Photometry 光度测定法
7.3 Colorimetry 比色测定法
7.4 Light Source Types 光源类型
7.5 Brdf Theory 双向反射分布函数理论
7.6 Brdf Models 双向反射分布函数模型
7.7 Brdf Acquisition And Representation Brdf的获取与表示
7.8 Implementing Brdfs 实现Brdf
7.9 Combining Lights And Materials 结合灯光与材质
Chapter 8 Area And Environmental Lighting 区域和环境光照
8.1 Radiometry For Arbitrary Lighting对于任意光照的辐射
8.2 Area Light Sources 区域光源
8.3 Ambient Light 环境光
8.4 Environment Mapping 环境映射
8.5 Glossy Reflections From Environment Maps 从环境映射中进行光泽反射
Chapter 9 Global Illumination 全局光照
9.1 Shadows 阴影
9.2 Ambient Occlusion 环境光遮蔽
9.3 Reflections 反射
9.4 Transmittance 透光率
9.5 Refractions 折射
9.6 Caustics 焦散
9.7 Global Subsurface Scattering 全局次表面散射
9.8 Full Global Illumination 完整全局光照
9.9 Precomputed Lighting 预计算光照
9.10 Precomputed Occlusion 预计算遮蔽
9.11 Precomputed Radiance Transfer 预计算辐射亮度传输
Chapter 10 Image-Based Effects 基于图像的效果
10.1 The Rendering Spectrum 绘制光谱
10.2 Fixed-View Effects 固定视角特效
10.3 Skyboxes 天空盒
10.4 Light Field Rendering 光场的渲染
10.5 Sprites And Layers 精灵与层级
10.6 Billboarding 公告板
10.7 Particle Systems 粒子系统
10.8 Displacement Techniques替代技术
10.9 Image Processing 图像处理
10.10 Color Correction 色彩校正
10.11 Tone Mapping色调映射
10.12 Lens Flare And Bloom镜头眩光和光晕
10.13 Depth Of Field 景深
10.14 Motion Blur 运动模糊
10.15 Fog 雾效
10.16 Volume Rendering 体积渲染
Chapter 11 Non-Photorealistic Rendering 非真实感渲染
11.1 Toon Shading 卡通着色
11.2 Silhouette Edge Rendering 轮廓边缘绘制
11.3 Other Styles 其他风格
11.4 Lines 线条
Chapter 12 Polygonal Techniques 多边形技术
12.1 Sources Of Three-Dimensional Data 三维数据的来源
12.2 Tessellation And Triangulation 细化和三角形剖分
12.3 Consolidation 合并
12.4 Triangle Fans, Strips, And Meshes 三角形扇、带、与网格
12.5 Simplification 网格简化
Chapter 13 Curves and Curved Surfaces 曲线和曲面
13.1 Parametric Curves 参数化曲线
13.2 Parametric Curved Surfaces 参数化曲面
13.3 Implicit Surfaces 隐式表面
13.4 Subdivision Curves 细分曲线
13.5 Subdivision Surfaces 细分曲面
13.6 Efficient Tessellation 高效细化
Chapter 14 Acceleration Algorithms 加速算法
14.1 Spatial Data Structures 空间数据结构
14.2 Culling Techniques 裁剪技术
14.3 Hierarchical View Frustum Culling 层次视域裁剪
14.4 Portal Culling 入口裁剪
14.5 Detail Culling 细节裁剪
14.6 Occlusion Culling 遮挡剔除
14.7 Level Of Detail 细节层次
14.8 Large Model Rendering 大型模型渲染
14.9 Point Rendering 点渲染
Chapter 15 Pipeline Optimization 管线优化
15.1 Profiling Tools 分析工具
15.2 Locating The Bottleneck 定位瓶颈
15.3 Performance Measurements 性能测试
15.4 Optimization 优化
15.5 Multiprocessing 多重处理
Chapter 16 Intersection Test Methods 相交测试方法
16.1 Hardware-Accelerated Picking 拾取硬件加速
16.2 Definitions And Tools 定义和工具
16.3 Bounding Volume Creation 包围体的创建
16.4 Geometric Probability 几何概率
16.5 Rules Of Thumb 重要规则
16.6 Ray/Sphere Intersection "射线 / 球体"相交测试
16.7 Ray/Box Intersection "射线 / 长方体"相交测试
16.8 Ray/Triangle Intersection "射线 / 三角形"相交测试
16.9 Ray/Polygon Intersection "射线 / 多边形"相交测试
16.10 Plane/Box Intersection Detection "平面 / 长方体"相交测试
16.11 Triangle/Triangle Intersection "三角形 / 三角形"相交测试
16.12 Triangle/Box Overlap "三角形 / 长方体"重叠测试
16.13 Bv/Bv Intersection Tests "Bv/Bv"相交测试
16.14 View Frustum Intersection 视锥体的相交测试
16.15 Shaft /Box And Shaft /Sphere Intersection "错 / 长方体"和"轴 / 球体"相交测试
16.16 Line/Line Intersection Tests "线 / 线"相交测试
16.17 Intersection Between Three Planes 三个面相交
16.18 Dynamic Intersection Testing 动态相交测试
Chapter 17 Collision Detection 碰撞检测
17.1 Collision Detection With Rays 使用射线进行碰撞检测
17.2 Dynamic Cd Using Bsp Trees 使用Bsp树进行动态碰撞检测
17.3 General Hierarchical Collision Detection 通用层次碰撞检测
17.4 Obbtree Obb树
17.5 A Multiple Objects Cd System 多物体碰撞检测系统
17.6 Miscellaneous Topics 其他主题
Chapter 18 Graphics Hardware 图形硬件
18.1 Buffers And Buffering 缓冲器和缓冲作用
18.2 Perspective-Correct Interpolation 透视校正插值
18.3 Architecture 架构

1.4 包含宝藏的书本主页

当然不能忘记书本主页，里面有一大堆图形学和渲染的书籍推荐，也有不少丰富的博文与资源。相信喜欢图形学和实时渲染的你，一定会爱不释手的。

地址在这里：Real–Time Rendering Resources

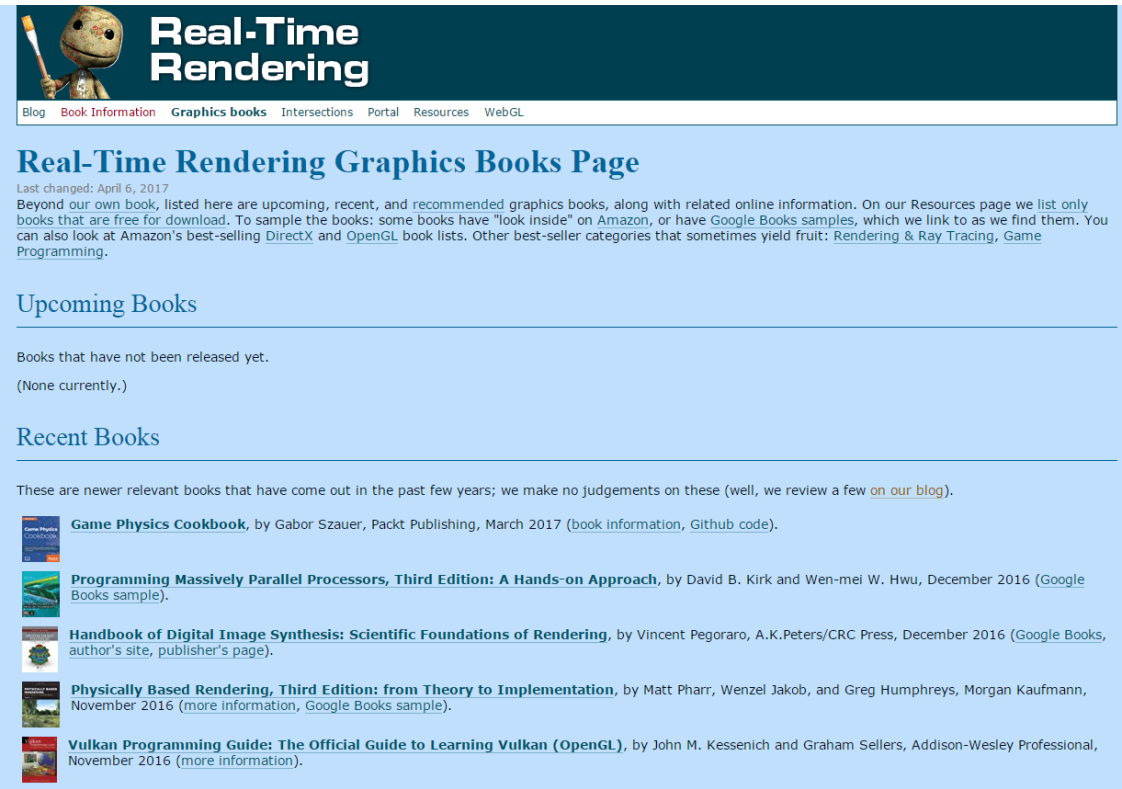


图 4　Real–Time Rendering Resources 主页截图

第二章 图形渲染管线

本章将带来 RTR3 第二章内容“Chapter 2 The Graphics Rendering Pipeline 图形渲染管线”的总结、概括与提炼。

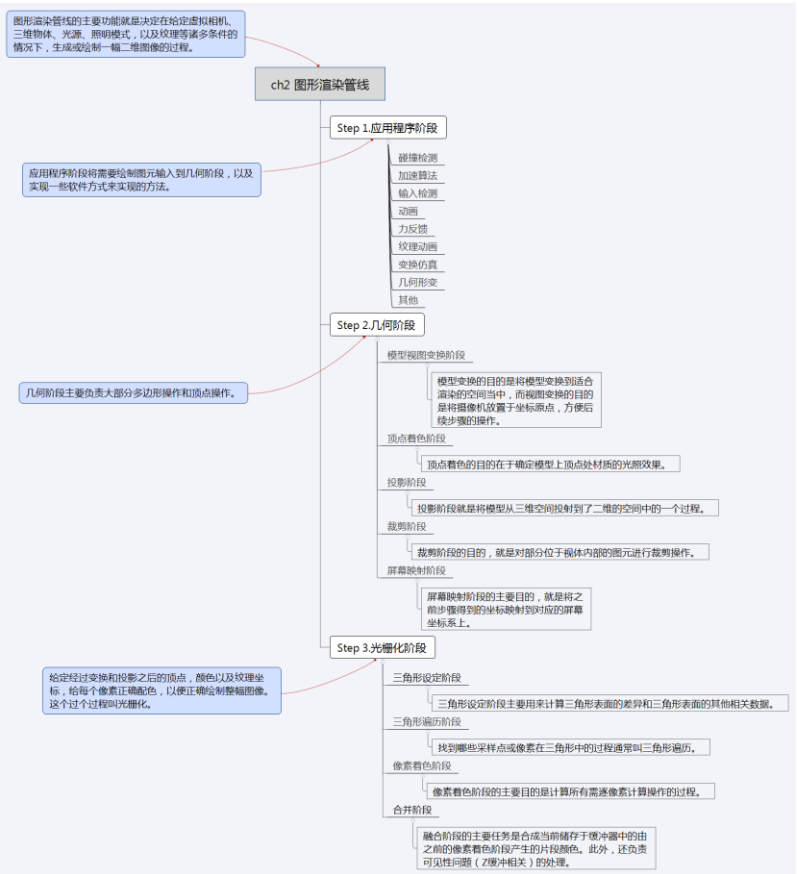
这章分为全文内容思维导图、核心内容分章节提炼、本章内容提炼总结三个部分来呈现，其中：

- 文章的第一部分，“全文内容思维导图”，分为“章节框架思维导图”和“知识结构思维导图”两个部分。
- 文章的第二部分，“核心内容分节提炼”，是按原书章节顺序分布的知识梳理。
- 而文章的第三部分“本章内容提炼总结”，则是更加精炼，只提炼出关键信息的知识总结。

2.1 本章内容思维导图

2.1.1 章节框架思维导图

2.1.2 知识结构思维导图

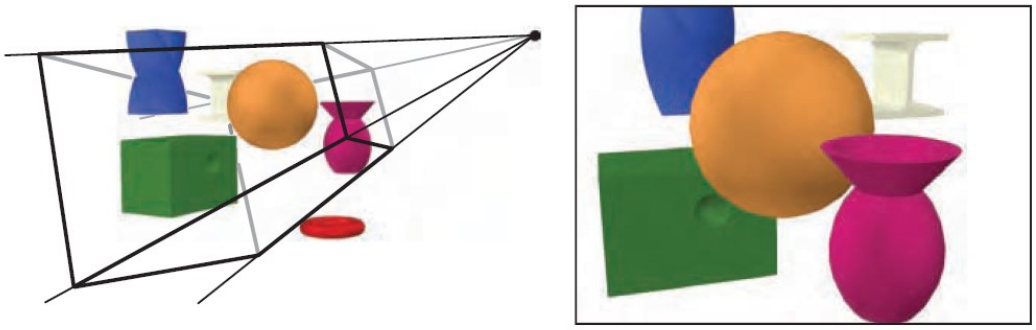


2.2 核心内容分节提炼

2.2.1 图像渲染管线架构概述

渲染管线的主要功能是决定在给定虚拟相机、三维物体、光源、照明模式，以及纹理等诸多条件的情况下，生成或绘制一幅二维图像的过程。对于实时渲染来说，渲染管线就是基础。因此，我们可以说，渲染管线是实时渲染的底层工具。

图 2.1 展示了使用渲染管线步骤。渲染出的图像的位置、形状是由它们的几何形状，环境特性，摄像机位置决定的。而物体的外观由材质特性，光源，纹理和着色模型确定。

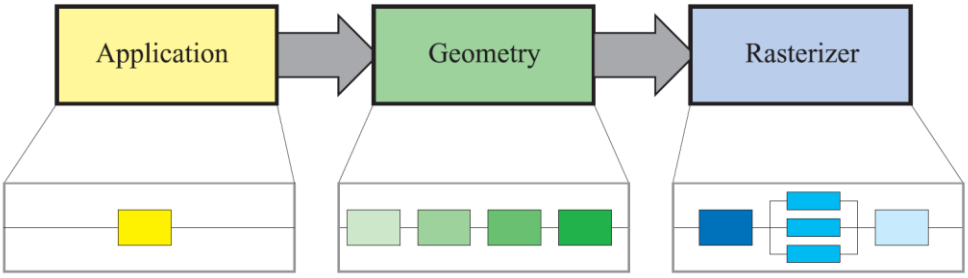


原书图 2.1 左图中，相机放在棱椎的顶端（四条线段的交汇点），只有可视体内部的图元会被渲染

在概念上可以将图形渲染管线分为三个阶段：

- 应用程序阶段（The Application Stage）
- 几何阶段（The Geometry Stage）
- 光栅化阶段（The Rasterizer Stage）

如下图：



原书图 2.2 绘制管线的基本结构包括 3 个阶段：应用程序、几何、光栅化

几个要点：

- 每个阶段本身也可能是一条管线，如图中的几何阶段所示。此外，还可以对有的阶段进行全部或者部分的并行化处理，如图中的光栅化阶段。应用程序阶段虽然是一个单独的过程，但是依然可以对之进行管线化或者并行化处理。
- 最慢的管线阶段决定绘制速度，即图像的更新速度，这种速度一般用 FPS 来表示，也就是每秒绘制的图像数量，或者用 Hz 来表示。

2.2.2 应用程序阶段 The Application Stage

- 应用程序阶段一般是图形渲染管线概念上的第一个阶段。应用程序阶段是通过软件方式来实现的阶段，开发者能够对该阶段发生的情况进行完全控制，可以通过改变实现方法来改变实际性能。其他阶段，他们全部或者部分建立在硬件基础上，因此要改变实现过程会非常困难。
- 正因应用程序阶段是软件方式实现，因此不能像几何和光栅化阶段那样继续分为若干个子阶段。但为了提高性能，该阶段还是可以在几个并行处理器上同时执行。在 CPU 设计上，称这种形式为超标量体系（superscalar）结构，因为它可以在同一阶段同一时间做不同的几件事情。
- 应用程序阶段通常实现的方法有碰撞检测、加速算法、输入检测，动画，力反馈以及纹理动画，变换仿真、几何变形，以及一些不在其他阶段执行的计算，如层次视锥裁剪等加速算法就可以在这里实现。
- 应用程序阶段的主要任务：在应用程序阶段的末端，将需要在屏幕上（具体形式取决于具体输入设备）显示出来绘制的几何体（也就是绘制图元，rendering primitives，如点、线、矩形等）输入到绘制管线的下一个阶段。
- 对于被渲染的每一帧，应用程序阶段将摄像机位置，光照和模型的图元输出到管线的下一个主要阶段——几何阶段。

2.2.3 几何阶段 The Geometry Stage

几何阶段主要负责大部分多边形操作和顶点操作。可以将这个阶段进一步划分成如下几个功能阶段：

- 模型视点变换 Model & View Transform
- 顶点着色 Vertex Shading
- 投影 Projection
- 裁剪 Clipping
- 屏幕映射 Screen Mapping

如图 2.3 所示。

原书图 2.3 几何阶段细分为的功能阶段管线

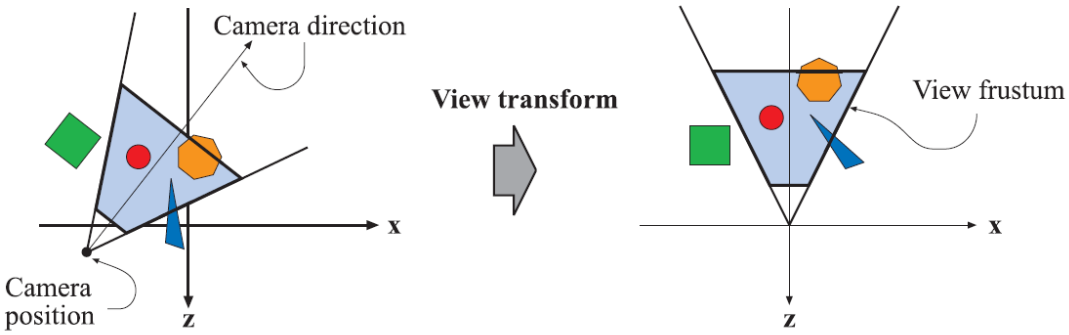
需要注意：

- 根据具体实现，这些阶段可以和管线阶段等同，也可以不等同。在一些情况下，一系列连续的功能阶段可以形成单个管线阶段（和其他管线阶段并行运行）。在另外情况下，一个功能阶段可以划分成其他更细小的管线阶段。
- 几何阶段执行的是计算量非常高的任务，在只有一个光源的情况下，每个顶点大约需要 100 次左右的精确的浮点运算操作。

2.2.3.1 模型和视图变换 Model and View Transform

- 在屏幕上的显示过程中，模型通常需要变换到若干不同的空间或坐标系中。模型变换的变换对象一般是模型的顶点和法线。物体的坐标称为模型坐标。世界空间是唯一的，所有的模型经过变换后都位于同一个空间中。
- 不难理解，应该仅对相机（或者视点）可以看到的模型进行绘制。而相机在世界空间中有一个位置方向，用来放置和校准相机。
- 为了便于投影和裁剪，必须对相机和所有的模型进行视点变换。变换的目的就是要把相机放在原点，然后进行视点校准，使其朝向 Z 轴负方向，y 轴指向上方,x 轴指向右边。在视点变换后，实际位置和方向就依赖于当前的 API。我们称上述空间为相机空间或者观察空间。

下图显示了视点变换对相机和模型的影响。



原书图 2.4 在左图中，摄像机根据用户指定的位置进行放置和定位。在右图中，视点变换从原点沿着 Z 轴负方向对相机重新定位，这样可以使裁剪和投影操作更简单、更快速。可视范围是一个平截椎体，因此可以认为它是透视模式。

【总结】模型和视图变换阶段分为模型变换和视图变换。模型变换的目的是将模型变换到适合渲染的空间当中，而视图变换的目的是将摄像机放置于坐标原点，方便后续步骤的操作。

2.2.3.2 顶点着色 Vertex Shading

为了产生逼真的场景，渲染形状和位置是远远不够的，我们需要对物体的外观进行建模。而物体经过建模，会得到对包括每个对象的材质，以及照射在对象上的任何光源的效果在内的一些描述。且光照和材质可以用任意数量的方式，从简单的颜色描述到复杂的物理描述来模拟。

确定材质上的光照效果的这种操作被称为着色（shading），着色过程涉及在对象上的各个点处计算着色方程（shading equation）。通常，这些计算中的一些在几何阶段期间在模型的顶点上执行（vertex shading），而其他计算可以在每像素光栅化（per-pixel rasterization）期间执行。可以在每个顶点处存储各种材料数据，诸如点的位置，法线，颜色或计算着色方程所需的任何其它数字信息。顶点着色的结果（其可以是颜色，向量，纹理坐标或任何其他种类的阴着色数据）计算完成后，会被发送到光栅化阶段以进行插值操作。

通常，着色计算通常认为是在世界空间中进行的。在实践中，有时需要将相关实体（诸如相机和光源）转换到一些其它空间（诸如模型或观察空间）并在那里执行计算，也可以得到正确的结果。

这是因为如果着色过程中所有的实体变换到了相同的空间，着色计算中需要的诸如光源，相机和模型之间的相对关系是不会变的。

【总结】顶点着色阶段的目的在于确定模型上顶点处材质的光照效果。

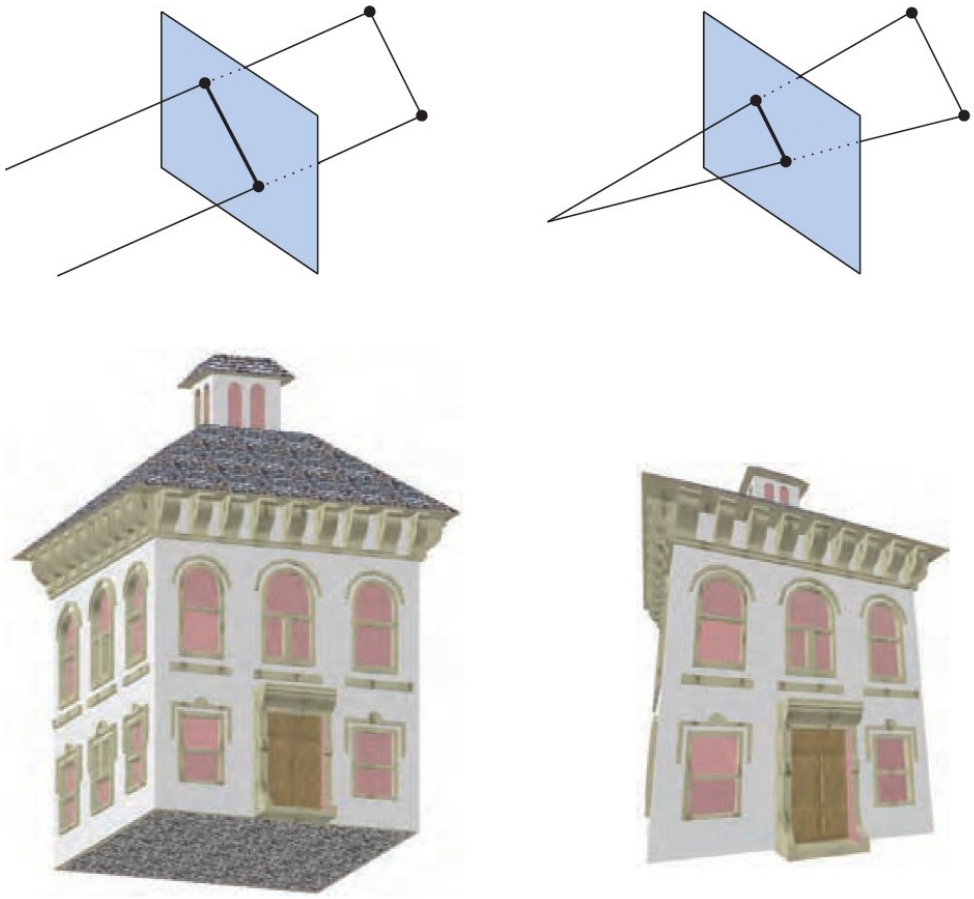
2.2.3.3 投影 Projection

在光照处理之后，渲染系统就开始进行投影操作，即将视体变换到一个对角顶点分别是(-1,-1,-1)和(1,1,1)单位立方体（unit cube）内，这个单位立方体通常也被称为规范立方体（Canonical View Volume，CVV）。

目前，主要有两种投影方法，即：

- 正交投影（orthographic projection，或称 parallel projection）。
- 透视投影（perspective projection）。

如下图所示。



原书图 2.5 左边为正交投影，右边为透视投影

两种投影方式的主要异同点：

- 正交投影。正交投影的可视体通常是一个矩形，正交投影可以把这个视体变换为单位立方体。正交投影的主要特性是平行线在变换之后彼此之间仍然保持平行，这种变换是平移与缩放的组合。
- 透视投影。相比之下，透视投影比正交投影复杂一些。在这种投影中，越远离摄像机的物体，它在投影后看起来越小。更进一步来说，平行线将在地平线处会聚。透视投影的变换其实就是模拟人类感知物体的方式。
- 正交投影和透视投影都可以通过 4 x 4 的矩阵来实现，在任何一种变换之后，都可以认为模型位于归一化处理之后的设备坐标系中。

虽然这些矩阵变换是从一个可视体变换到另一个，但它们仍被称为投影，因为在完成显示后，Z 坐标将不会再保存于的得到的投影图片中。通过这样的投影方法，就将模型从三维空间投影到了二维的空间中。

【总结】投影阶段就是将模型从三维空间投射到了二维的空间中的过程。

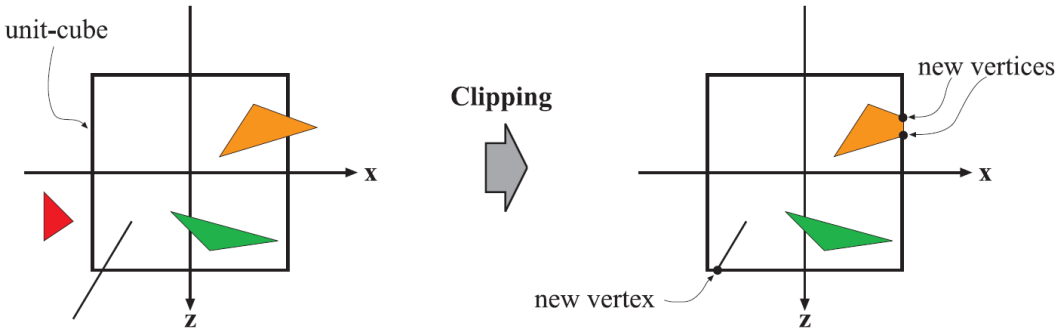
2.2.3.4 裁剪 Clipping

只有当图元完全或部分存在于视体（也就是上文的规范立方体，CVV）内部的时候，才需要将其发送到光栅化阶段，这个阶段可以把这些图元在屏幕上绘制出来。

不难理解，一个图元相对视体内部的位置，分为三种情况：完全位于内部、完全位于外部、部分位于内部。所以就要分情况进行处理：

- 当图元完全位于视体内部，那么它可以直接进行下一个阶段。
- 当图元完全位于视体外部，不会进入下一个阶段，可直接丢弃，因为它们无需进行渲染。
- 当图元部分位于视体内部，则需要对那些部分位于视体内的图元进行裁剪处理。

对部分位于视体内部的图元进行裁剪操作，这就是裁剪过程存在的意义。裁剪过程见下图。



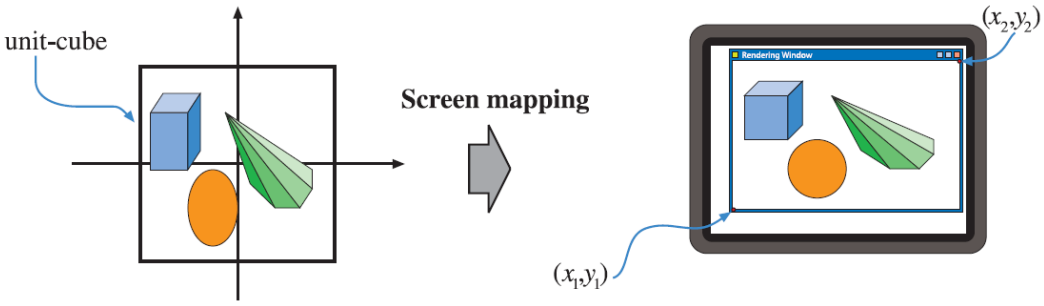
原书图 2.6 投影变换后，只对单位立方体内的图元（相应的是视锥内可见图元）继续进行处理，因此，将单位立方体之外的图元剔除掉，保留单位立方体内部的图元，同时沿着单位立方体将与单位立方体相交的图元裁剪掉，因此，就会产生新的图元，同时舍弃旧的图元。

【总结】裁剪阶段的目的，就是对部分位于视体内部的图元进行裁剪操作。

2.2.3.5 屏幕映射 Screen Mapping

只有在视体内部经过裁剪的图元，以及之前完全位于视体内部的图元，才可以进入到屏幕映射阶段。进入到这个阶段时，坐标仍然是三维的（但显示状态在经过投影阶段后已经成了二维），每个图元的 x 和 y 坐标变换到了屏幕坐标系中，屏幕坐标系连同 z 坐标一起称为窗口坐标系。

假定在一个窗口里对场景进行绘制，窗口的最小坐标为（x1，y1），最大坐标为（x2，y2），其中 x1<x2，y1<y2。屏幕映射首先进行平移，随后进行缩放，在映射过程中 z 坐标不受影响。新的 x 和 y 坐标称为屏幕坐标系，与 z 坐标一起（$-1 \leqq z \leqq 1$）进入光栅化阶段。如下图：



原书图 2.8 经过投影变换，图元全部位于单位立方体之内，而屏幕映射主要目的就是找到屏幕上对应的坐标

屏幕映射阶段的一个常见困惑是整型和浮点型的点值如何与像素坐标（或纹理坐标）进行关联。可以使用 Heckbert[书后参考文献第 520 篇]的策略，用一个转换公式进行解决。

【总结】屏幕映射阶段的主要目的，是将之前步骤得到的坐标映射到对应的屏幕坐标系上。

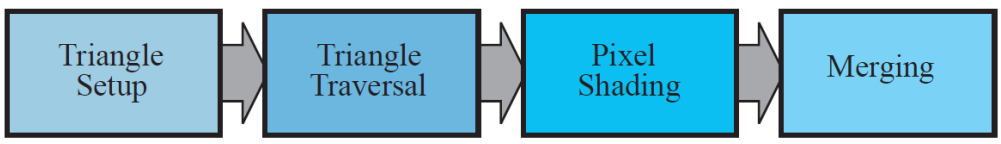
2.2.4 光栅化阶段 The Rasterizer Stage

给定经过变换和投影之后的顶点，颜色以及纹理坐标（均来自于几何阶段），给每个像素（Pixel）正确配色，以便正确绘制整幅图像。这个过个过程叫光珊化（rasterization）或扫描变换（scan conversion），即从二维顶点所处的屏幕空间（所有顶点都包含 Z 值即深度值，及各种与相关的着色信息）到屏幕上的像素的转换。

与几何阶段相似，该阶段细分为几个功能阶段：

- 三角形设定（Triangle Setup）阶段
- 三角形遍历（Triangle Traversal）阶段
- 像素着色（Pixel Shading）阶段
- 融合（Merging）阶段

如下图所示:



原书图 2.8 光栅化阶段一般细分为三角形设定，三角形遍历，像素着色和融合四个子阶段。

2.2.4.1 三角形设定 Triangle Setup

三角形设定阶段主要用来计算三角形表面的差异和三角形表面的其他相关数据。该数据主要用于扫描转换（scan conversion），以及由几何阶段处理的各种着色数据的插值操作所用。该过程在专门为其设计的硬件上执行。

2.2.4.2 三角形遍历 Triangle Traversal

在三角形遍历阶段将进行逐像素检查操作，检查该像素处的像素中心是否由三角形覆盖，而对于有三角形部分重合的像素，将在其重合部分生成片段（fragment）。

找到哪些采样点或像素在三角形中的过程通常叫三角形遍历（TriangleTraversal）或扫描转换（scan conversion）。每个三角形片段的属性均由三个三角形顶点的数据插值而生成（在第五章会有讲解）。这些属性包括片段的深度，以及来自几何阶段的着色数据。

【总结】找到哪些采样点或像素在三角形中的过程通常叫三角形遍历（TriangleTraversal）或扫描转换（scan conversion）。

2.2.4.3 像素着色 Pixel Shading

所有逐像素的着色计算都在像素着色阶段进行，使用插值得来的着色数据作为输入，输出结果为一种或多种将被传送到下一阶段的颜色信息。纹理贴图操作就是在这阶段进行的。

像素着色阶段是在可编程 GPU 内执行的，在这一阶段有大量的技术可以使用，其中最常见，最重要的技术之一就是纹理贴图（Texturing）。纹理贴图在书的第六章会详细讲到。简单来说，纹理贴图就是将指定图片“贴”到指定物体上的过程。而指定的图片可以是一维，二维，或者三维的，其中，自然是二维图片最为常见。如下图所示：

原书图 2.9 左上角为一没有纹理贴图的飞龙模型。左下角为一贴上图像纹理的飞龙。右图为所用的纹理贴图。

【总结】像素着色阶段的主要目的是计算所有需逐像素操作的过程。

2.2.4.4 融合 Merging

每个像素的信息都储存在颜色缓冲器中，而颜色缓冲器是一个颜色的矩阵列（每种颜色包含红、绿、蓝三个分量）。融合阶段的主要任务是合成当前储存于缓冲器中的由之前的像素着色阶段产生的片段颜色。不像其它着色阶段，通常运行该阶段的 GPU 子单元并非完全可编程的，但其高度可配置，可支持多种特效。

此外，这个阶段还负责可见性问题的处理。这意味着当绘制完整场景的时候，颜色缓冲器中应该还包含从相机视点处可以观察到的场景图元。对于大多数图形硬件来说，这个过程是通过 Z 缓冲（也称深度缓冲器）算法来实现的。Z 缓冲算法非常简单，具有 O(n)复杂度（n 是需要绘制的像素数量），只要对每个图元计算出相应的像素 z 值，就可以使用这种方法，大概内容是：

Z 缓冲器和颜色缓冲器形状大小一样，每个像素都存储着一个 z 值，这个 z 值是从相机到最近图元之间的距离。每次将一个图元绘制为相应像素时，需要计算像素位置处图元的 z 值，并与同一像素处的 z 缓冲器内容进行比较。如果新计算出的 z 值，远远小于 z 缓冲器中的 z 值，那么说明即将绘制的图元与相机的距离比原来距离相机最近的图元还要近。这样，像素的 z 值和颜色就由当前图元对应的值和颜色进行更新。反之，若计算出的 z 值远远大于 z 缓冲器中的 z 值，那么 z 缓冲器和颜色缓冲器中的值就无需改变。

上面刚说到，颜色缓冲器用来存储颜色，z 缓冲器用来存储每个像素的 z 值，还有其他缓冲器可以用来过滤和捕获片段信息。

- 比如 alpha 通道（alpha channel）和颜色缓冲器联系在一起可以存储一个与每个像素相关的不透明值。可选的 alpha 测试可在深度测试执行前在传入片段上运行。片段的 alpha 值与参考值作某些特定的测试（如等于，大于等），如果片断未能通过测试，它将不再进行进一步的处理。alpha 测试经常用于不影响深度缓存的全透明片段（见 6.6 节）的处理。

- 模板缓冲器（stencil buffer）是用于记录所呈现图元位置的离屏缓存。每个像素通常与占用 8 个位。图元可使用各种方法渲染到模板缓冲器中，而缓冲器中的内容可以控制颜色缓存和 Z 缓存的渲染。举个例子，假设在模版缓冲器中绘制出了一个实心圆形，那么可以使用一系列操作符来将后续的图元仅在圆形所出现的像素处绘制，类似一个 mask 的操作。模板缓冲器是制作特效的强大工具。而在管线末端的所有这些功能都叫做光栅操作（raster operations ，ROP）或混合操作（blend operations）。

- 帧缓冲器（frame buffer）通常包含一个系统所具有的所有缓冲器，但有时也可以认为是颜色缓冲器和 z 缓冲器的组合。

- 累计缓冲器（accumulation buffer），是 1990 年，Haeberli 和 Akeley 提出的一种缓冲器，是对帧缓冲器的补充。这个缓冲器可以用一组操作符对图像进行累积。例如，为了产生运动模糊（motion blur.，可以对一系列物体运动的图像进行累积和平均。此外，其他的一些可产生的效果包括景深（e depth of field），反走样（antialiasing）和软阴影（soft shadows）等。

而当图元通过光栅化阶段之后，从相机视点处看到的东西就可以在荧幕上显示出来。为了避免观察者体验到对图元进行处理并发送到屏幕的过程，图形系统一般使用了双缓冲（double buffering）机制，这意味着屏幕绘制是在一个后置缓冲器（backbuffer）中以离屏的方式进行的。一旦屏幕已在后置缓冲器中绘制，后置缓冲器中的内容就不断与已经在屏幕上显示过的前置缓冲器中的内容进行交换。注意，只有当不影响显示的时候，才进行交换。

【总结】融合阶段的主要任务是合成当前储存于缓冲器中的由之前的像素着色阶段产生的片段颜色。此外，融合阶段还负责可见性问题（Z 缓冲相关）的处理。

2.2.5 管线纵览与总结

在概念上可以将图形渲染管线分为三个阶段：应用程序阶段、几何阶段、光栅化阶段。

这样的管线结构是 API 和图形硬件十年来以实时渲染应用程序为目标进行演化的结果。需要注意的是这个进化不仅仅是在我们所说的渲染管线中，离线渲染管线（offline rendering pipelines）也是另一种进化的路径。且电影产品的渲染通常使用微多边形管线（micropolygon pipelines）。而学术研究和预测渲染的（predictive rendering）应用，比如建筑重建（architectural previsualization）通常采用的是光线跟踪渲染器（ray tracing renderers）

2.3 本章内容提炼总结

以下是对《Real Time Rendering 3rd》第二章“图形渲染管线”内容的文字版概括总结：

图形渲染管线的主要功能就是决定在给定虚拟相机、三维物体、光源、照明模式，以及纹理等诸多条件的情况下，生成或绘制一幅二维图像的过程。在概念上可以将图形渲染管线分为三个阶段：应用程序阶段、几何阶段、光栅化阶段。

2.3.1 应用程序阶段

应用程序阶段的主要任务，是将需要绘制图元输入到绘制管线的下一个阶段，以及实现一些软件方式来实现的方法。主要方法有碰撞检测、加速算法、输入检测，动画，力反馈以及纹理动画，变换仿真、几何变形，以及一些不在其他阶段执行的计算，如层次视锥裁剪等加速算法。

对于被渲染的每一帧，应用程序阶段将摄像机位置，光照和模型的图元输出到管线的下一个主要阶段，即几何阶段。

2.3.2 几何阶段

几何阶段主要负责大部分多边形操作和顶点操作，可以将这个阶段进一步划分成如下几个功能阶段：模型视点变换、顶点着色、投影、裁剪、屏幕映射。这些功能阶段的主要功能如下：

- 【模型和视图变换阶段】：模型变换的目的是将模型变换到适合渲染的空间当中，而视图变换的目的是将摄像机放置于坐标原点，方便后续步骤的操作。
- 【顶点着色阶段】：顶点着色的目的在于确定模型上顶点处材质的光照效果。
- 【投影阶段】：投影阶段是将模型从三维空间投射到二维的空间中的过程。投影阶段也可以理解为将视体变换到一个对角顶点分别是(–1,–1,–1)和(1,1,1)单位立方体内的过程。
- 【裁剪阶段】裁剪阶段的目的，是对部分位于视体内部的图元进行裁剪操作。
- 【屏幕映射阶段】屏幕映射阶段的主要目的，是将之前步骤得到的坐标映射到对应的屏幕坐标系上。

在几何阶段，首先，对模型的顶点和法线进行矩阵变换，并将模型置于观察空间中（模型和视图变换），然后，根据材质、纹理、以及光源属性进行顶点光照的计算（顶点着色阶段），接着，将该模型投影变换到一个单位立方体内，并舍弃所有立方体之外的图元（投影

阶段），而为了得到所有位于立方体内部的图元，接下来对与单位立方体相交的图元进行裁剪（裁剪阶段），然后将顶点映射到屏幕上的窗口中（屏幕映射阶段）。在对每个多边形执行完这些操作后，将最终数据传递到光栅，这样就来到了管线中的最后一个阶段，光栅化阶段。

2.3.3 光栅化阶段

给定经过变换和投影之后的顶点，颜色以及纹理坐标（均来自于几何阶段），给每个像素正确配色，以便正确绘制整幅图像。这个过个过程叫光栅化，即从二维顶点所处的屏幕空间（所有顶点都包含 Z 值即深度值，及各种与相关的着色信息）到屏幕上的像素的转换。光栅化阶段可分为三角形设定阶段、三角形遍历阶段、像素着色阶段、融合阶段。这些功能阶段的主要功能如下：

- 【三角形设定阶段】三角形设定阶段主要用来计算三角形表面的差异和三角形表面的其他相关数据。
- 【三角形遍历阶段】找到哪些采样点或像素在三角形中的过程通常叫三角形遍历。
- 【像素着色阶段】像素着色阶段的主要目的是计算所有需逐像素计算操作的过程。
- 【融合阶段】融合阶段的主要任务是合成当前储存于缓冲器中的由之前的像素着色阶段产生的片段颜色。此外，融合阶段还负责可见性问题（Z 缓冲相关）的处理。

在光栅化阶段，所有图元会被光栅化，进而转换为屏幕上的像素。首先，计算三角形表面的差异和三角形表面的其他相关数据（三角形设定阶段），然后，找到哪些采样点或像素在三角形中（三角形遍历阶段），接着计算所有需逐像素计算操作（像素着色阶段），然后，合成当前储存于缓冲器中的由之前的像素着色阶段产生的片段颜色，可见性问题可通过 Z 缓存算法解决，随同的还有可选的 alpha 测试和模版测试（融合阶段）。所有对像依次处理，而最后的图像显示在屏幕上。

第三章 GPU 渲染管线与可编程着色器

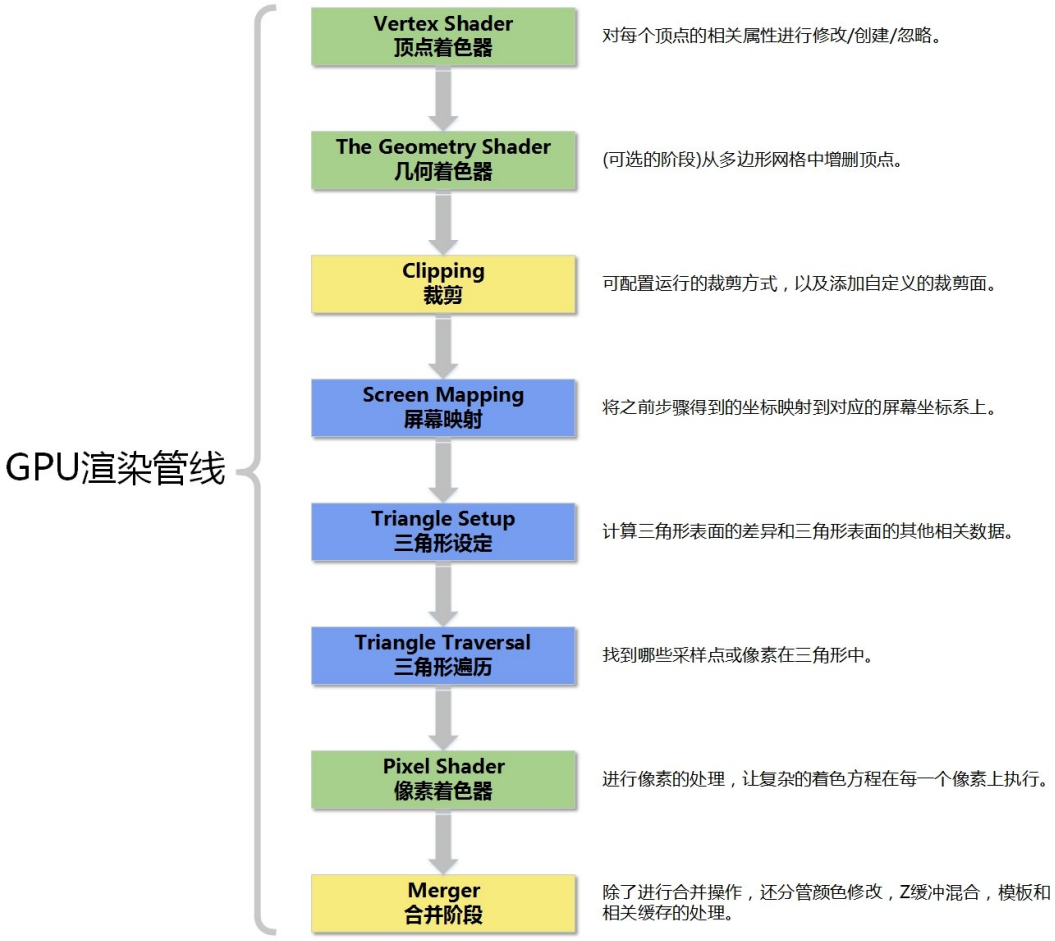
本章将带来 RTR3 第三章内容“Chapter 3 The Graphics Processing Unit 图形处理器”的总结、概括与提炼。

这章的主要内容是介绍 GPU 渲染管线的组成，以及可编程着色的进化史，顶点、几何、像素三种可编程着色器。

3.1 本章内容图示

3.1.1 章节框架图示

3.1.2 GPU 渲染管线流程图



其中：

- 绿色的阶段都是完全可编程的。
- 黄色的阶段可配置，但不可编程。
- 蓝色的阶段完全固定。

3.2 原书核心内容分节提炼

3.2.1 GPU 管线概述

- 第一个包含顶点处理，面向消费者的图形芯片（NVIDIA GeForce 256）发布于 1999 年，且 NVIDIA 提出了图形处理单元（Graphics Processing Unit，GPU）这一术语，将 GeForce 256 和之前的只能进行光栅化处理的图形芯片相区分。在接下来的几年中，GPU 从可配置的固定功能管线演变到了支持高度可编程的管线。直到如今，各种可编程着色器依然是控制 GPU 的主要手段。为了提高效率，GPU 管线的一部分仍然保持着可配置，但不是可编程的，但大趋势依然是朝着可编程和更具灵活性的方向在发展。
- GPU 实现了第二章中描述的几何和光栅化概念管线阶段。其被分为一些不同程度的可配置性和可编程性的硬件阶段。如图 3.1 所示。

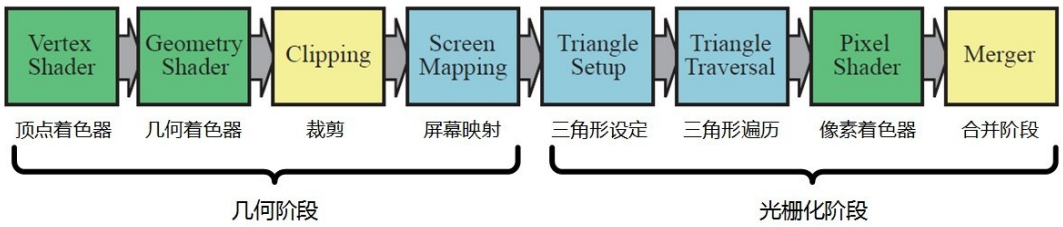


图 3.1 GPU 实现的渲染管线

上图中，不同颜色的阶段表示了该阶段不同属性。其中：

- 绿色的阶段都是完全可编程的。
- 黄色的阶段可配置，但不可编程。
- 蓝色的阶段完全固定。

需要注意，GPU 实现的渲染管线和第二章中描述的渲染管线的功能阶段在结构上略有不同。以下是对 GPU 渲染管线的一个流程概览：

- 顶点着色器（The Vertex Shader）是完全可编程的阶段，顶点着色器可以对每个顶点进行诸如变换和变形在内的很多操作，提供了修改/创建/忽略顶点相关属性的功能，这些顶点属性包括颜色、法线、纹理坐标和位置。顶点着色器的必须完成的任务是将顶点从模型空间转换到齐次裁剪空间。

- 几何着色器（The Geometry Shader）位于顶点着色器之后，允许 GPU 高效地创建和销毁几何图元。几何着色器是可选的，完全可编程的阶段，主要对图元（点、线、三角形）的顶点进行操作。几何着色器接收顶点着色器的输出作为输入，通过高效的几何运算，将数据输出，数据随后经过几何阶段和光栅化阶段的其他处理后，会发送给片段着色器。

- 裁剪（Clipping）属于可配置的功能阶段，在此阶段可选运行的裁剪方式，以及添加自定义的裁剪面。

- 屏幕映射（Screen Mapping）、三角形设置（Triangle Setup）和三角形遍历（Triangle Traversal）阶段是固定功能阶段。

- 像素着色器（Pixel Shader，Direct3D 中的叫法）常常又称为片断着色器，片元着色器(Fragment Shader，OpenGL 中的叫法)，是完全可编程的阶段，主要作用是进行像素的处理，让复杂的着色方程在每一个像素上执行。

- 合并阶段（The Merger Stage）处于完全可编程和固定功能之间，尽管不能编程，但是高度可配置，可以进行一系列的操作。其除了进行合并操作，还分管颜色修改（Color Modifying），Z 缓冲（Z-buffer），混合（Blend），模板（Stencil）和相关缓存的处理。

随着时间的推移，GPU 管线已经远离硬编码的运算操作，而朝着提高灵活性和控制性改进。编程着色器的引入是这个进化的最重要的一步。

3.2.2 可编程着色模型

- 现代着色阶段（比如支持 Shader Model 4.0，DirectX 10 以及之后）使用了通用着色核心（common-shader core），这就表明顶点，片段，几何着色器共享一套编程模型。

- 早期的着色模型可以用汇编语言直接编程，但 DX10 之后，汇编就只在调试输出阶段可见，改用高级着色语言。

- 目前的着色语言都是 C-like 的着色语言，比如 HLSL,CG 和 GLSL，其被编译成独立于机器的汇编语言语言，也称为中间语言（IL）。这些汇编语言在单独的阶段，通常是在驱动中，被转化成实际的机器语言。这样的安排可以兼容不同的硬件实现。这些汇编语言可以被看做是定义一个作为着色语言编译器的虚拟机。这个虚拟机是一个处理多种类型寄存器和数据源、预编了一系列指令的处理器。

- 着色语言虚拟机可以理解为一个处理多种类型寄存器和数据源、预编了一系列指令的处理器。考虑到很多图形操作都使用短矢量（最高四位），处理器拥有 4 路 SIMD（single-instruction multiple-data，单指令多数据）兼容性。每个寄存器包含四个独立的值。32 位单精度浮点的标量和矢量是其基本数据

类型；也随后支持 32 位整型。浮点矢量通常包含数据如位置（xyzw），法线，矩阵行，颜色（rgba），或者纹理坐标（uvwq）。而整型通常用来表示，计数器，索引，或者位掩码。也支持综合数据类型比如结构体，数组，和矩阵。而为了便于使用向量，向量操作如调和（swizzling，也就是向量分量的重新排序或复制），和屏蔽（masking，只使用指定的矢量元素），也能够支持。虚拟机的输入和输出见图 3.2。

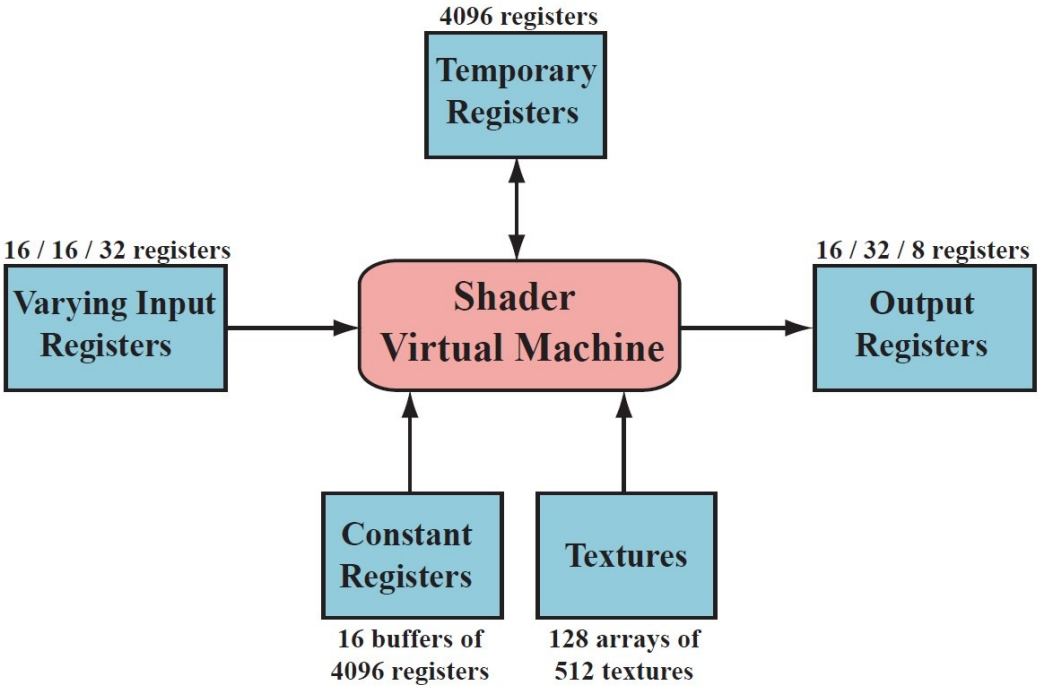


图 3.2 DX 10 下的通用 Shader 核心虚拟机架构以及寄存器布局。每个资源旁边显示最大可用编号。其中，用两个斜杠分开的三个数值，分别是顶点，几何、像素着色器对应的可用最大值。

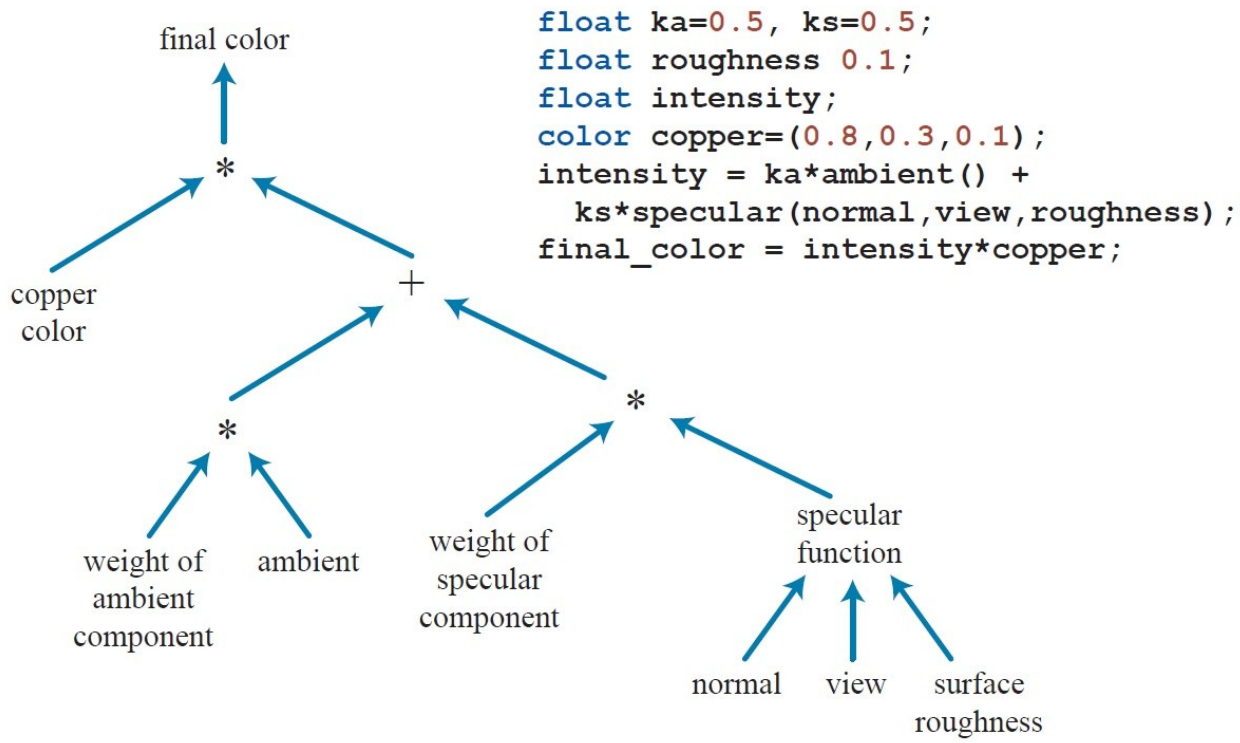
- 一个绘制调用（也就是喜闻乐见的 Draw Call）会调用图形 API 来绘制一系列的图元，会驱使图形管线的运行。
- 每个可编程着色阶段拥有两种类型的输入：
  - uniform 输入，在一个 draw call 中保持不变的值（但在不同 draw call 之间可以更改）；
  - varying 输入，shader 里对每个顶点和像素的处理都不同的值。纹理是特殊类型的 uniform 输入，曾经一直是一张应用到表面的彩色图片，但现在可以认为是存储着大量数据的数组。
- 在现代 GPU 上 ，图形运算中常见的运算操作执行速度非常快。通常情况下，最快的操作是标量和向量的乘法和加法，以及他们的组合，如乘加运算（multiply-add）和点乘 （dot-product）运算。其他操作，比如倒数（reciprocal），平方根（square root），正弦（sine），余弦（cosine），指数（exponentiation）、对数（logarithm）运算，往往会稍微更加昂贵，但依然相当快捷。纹理操作非常高效，但他们的性能可能受到诸如等待检索结果的时间等因素的限制。
- 着色语言表示出了大多数场常见的操作（比如加法和乘法通过运算符+和*来表示）。其余的操作用固有的函数，比如 atan() , dot() , log(),等。更复杂的操作也存在内建函数，比如矢量归一化（vector

normalization）、反射（reflection）、叉乘（cross products）、矩阵的转置（matrix transpose）和行列式（determinant）等。

- 流控制（flow control）是指使用分支指令来改变代码执行流程的操作。这些指令用于实现高级语言结构，如“if”和“case”语句，以及各种类型的循环。Shader 支持两种类型的流控制。静态流控制（Static flow control）是基于统一输入的值的。这意味着代码的流在调用时是恒定的。静态流控制的主要好处是允许在不同的情况下使用相同的着色器（例如，不同数量的光源）。动态流控制（Dynamic flow control）基于不同的输入值。但动态流控制远比静态流量控制更强大但同时也需更高的开销，特别是在调用 shader 之间，代码流不规律改变的时候。正如 18.4.2 节中讨论的，评估一个 shader 的性能，是评估其在一段时间内处理顶点或像素的个数。如果流对某些元素选择“if”分支，而对其他元素选择“else”分支，这两个分支必须对所有元素进行评估（并且每个元素的未使用分支将被丢弃）。
- Shader 程序可以在程序加载或运行时离线编译。和任何编译器一样，有生成不同输出文件和使用不同优化级别的选项。一个编译过的 Shader 作为字符串或者文本来存储，并通过驱动程序传递给 GPU。

3.2.3 可编程着色的进化史 The Evolution of Programmable Shading

可编程着色的框架的思想可以追溯到 1984 年 Cook 的 shade trees。一个简单的 Shader 以及其对应的 shade tree 如图 3.3。



原书图 3.3 一个简单的铜着色器（Cook 的 shade trees）和其对应的着色语言代码

- RenderMan 着色语言（Render Man Shading Language）是从 80 年代中后期根据这个可编程着色的框架思想开发出来的，目前仍然广泛运用于电影制作的渲染中。在 GPU 原生支持可编程着色之前，有一些通过多个渲染通道实现实时可编程着色的尝试。
- 在 1999 年,《雷神 III：竞技场》脚本语言是在这个领域上第一个成功广泛商用的语言。
- 在 2000 年，Peery 等人，描绘了一个将 Render Man Shader 翻译成在多通道上并且运行在图形硬件上的系统。
- 在 2001 年，NVIDIA's GeForce 3 发布，它是第一个支持可编程顶点着色器的 GPU，向 DirectX 8.0 开放，并以扩展的形式提供给 OpenGL。
- 2002 年， DirectX 9.0 发布，里面包括了 Shader Model 2.0（以及其扩展版本 2.X），Shader Model 2.0 支持真正的可编程顶点和像素着色。着色编程语言 HLSL 也是随着 DirectX 9.0 的发布而问世。
- 2004 年，Shader Model 3.0 推出。SM 3.0 是一个增量改进，将之前的可选功能进行了实现，进一步增加资源上限，并在顶点着色器中添加了有限的纹理读取支持。新一代主机的问世，2005 (Microsofts Xbox 360)，2006 (Sony Play Station)，它们配备配备了 Shader Model 3.0 级别的 GPU。
- 2006 年底 ，任天堂发布了搭载固定功能管线 GPU 的 Wii 主机 ，固定功能管线，并没有像预想中的会完全死亡（其实一直活得很好）。
- 2007 年，Shader Model 4.0 发布，包含于 DirectX 10.0 中，OpenGL 以扩展的方式支持。SM 4.0 新增了几个着色器和流输出等新特性。SM 4.0 包括支持所有 Shader 类型（顶点、几何、像素着色器）的一套统一着色模型（uniform programming model）

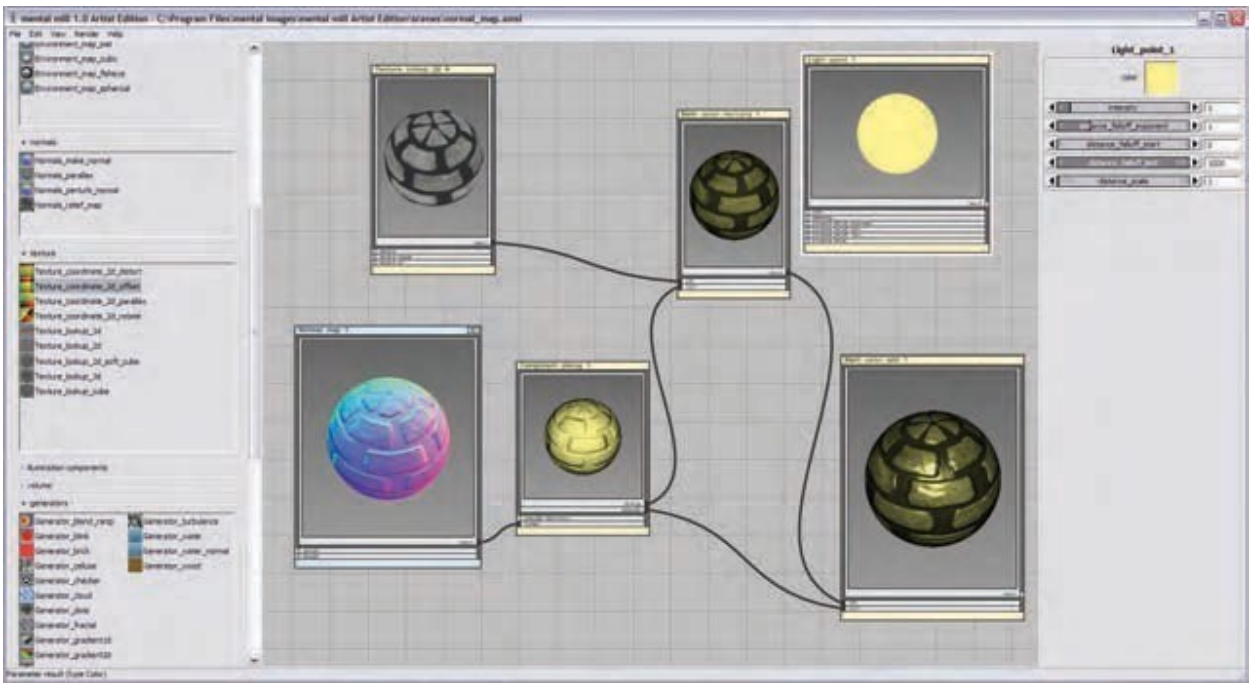


图 3.4 一个名为“mental mill”的可视化着色器设计系统。各种操作都封装在功能窗中，位于左侧。每个功能窗都有可调参数，每个功能框的输入和输出彼此连接，形成最终结果。

3.2.3.1 着色模型对比 Comparison of Shader Models

下表 3.1 比较了各种着色模型的能力。在这个表格中，VS”代表顶点着色器和“PS”代表像素着色器（而着色模型 4.0 引入了几何着色器，其能力与顶点着色相似）。如果没有出现“VS”和“PS”，那么该行适用于顶点和像素着色器。

\	SM 2.0/2.X	SM 3.0	SM 4.0
引入版本	DX 9.0, 2002	DX 9.0c, 2004	DX 10, 2007
VS 指令槽位	256	≥512	4096
VS 最大执行步长	65536	65536	∞
PS 指令槽位	≥96	≥512	≥65536
PS 最大执行步长	≥96	65536	∞
临时寄存器	≥12	32	4096
VS 常量寄存器	≥256	≥256	14×4096
PS 常量寄存器	32	224	14×4096
流程控制，判断	Optional	Yes	Yes
VS 纹理贴图	None	4	128×512
PS 纹理贴图	16	16	128×512
整数支持	No	No	Yes
VS 输入寄存器	16	16	16
插值寄存器	8	10	16 / 32
PS 输出寄存器	4	4	8

表 3.1 不同版本的着色模型能力对比，按 DirectX 着色模型版本列出

3.2.4 顶点着色器 Vertext Shader

- 顶点着色器（The Vertex Shader）的功能于 2001 年首次在 DirectX 8 中引入。由于它是流水线上的第一个阶段，可选是在 GPU 还是 CPU 上实现。而在 CPU 上实现的话，需将 CPU 中的输出数据发送到 GPU 进行光栅化。目前几乎所有的 GPU 都支持顶点着色。

- 顶点着色器是完全可编程的阶段，是专门处理传入的顶点信息的着色器，顶点着色器可以对每个顶点进行诸如变换和变形在内的很多操作。顶点着色器一般不处理附加信息，也就是说，顶点着色器提供了修改，创建，或者忽略与每个多边形顶点相关的值的方式，例如其颜色，法线，纹理坐标和位置。通常，顶点着色器程序将顶点从模型空间（Model Space）变换到齐次裁剪空间（Homogeneous Clip Space），并且，一个顶点着色器至少且必须输出此变换位置。

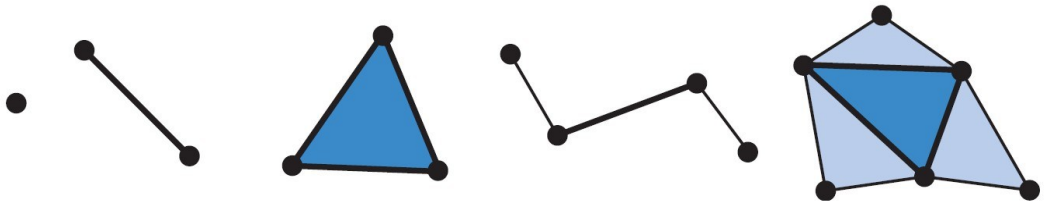
- 值得注意的是，在这个顶点着色阶段之前发生了一些数据操作。比如在 DirectX 中叫做输入装配（Input Assembler）的阶段，会将一些数据流组织在一起，以形成顶点和基元的集合，发送到管线。
- 顶点着色器本身与前面 3.2 节所述的通用核心虚拟机（Common-Shader Core Virtual Machine）非常相似。传入的每个顶点由顶点着色器程序处理，然后输出一些在三角形或直线上进行插值后获得的值。顶点着色器既不能创建也不能消除顶点，并且由一个顶点生成的结果不能传递到另一个顶点。由于每个顶点都被独立处理，所以 GPU 上的任何数量的着色器处理器都可以并行地应用到传入的顶点流上。
- 顶点着色器的输出可以以许多不同的方式来使用，通常是随后用于每个实例三角形的生成和光栅化，然后各个像素片段被发送到像素着色器，以便继续处理。而在 Shader Model 4.0 中，数据也可以发送到几何着色器（Geometry Shader）或输出流（Streamed Output）或同时发动到像素着色器和几何着色器两者中。

如图 3.5 使用顶点着色器的一些示例。

原书图 3.5 左图，一个普通茶壶。中图，经过顶点着色程序执行的简单剪切（shear）操作产生的茶壶。右图，通过噪声（noise）产生的发生形变的茶壶。

3.2.5 几何着色器 The Geometry Shader

- 几何着色器（Geometry Shader）是顶点和片段着色器之间一个可选的着色器，随着 2006 年底 DirectX 10 的发布被加入到硬件加速图形管线中。几何着色器作为 Shader Model 4.0 的一部分，不能在早期着色模型（<= SM 3.0）中使用。
- 几何着色器的输入是单个对象及对象相关的顶点，而对象通常是网格中的三角形，线段或简单的点。另外，扩展的图元可以由几何着色器定义和处理。如图 3.6。



原书图 3.6 几何着色器程序的输入是一个单独的类型：点，线段，三角形。两个最右边的图元，包括与线和三角形对象相邻的顶点也可被使用。

- 几何着色器可以改变新传递进来的图元的拓扑结构，且几何着色器可以接收任何拓扑类型的图元，但是只能输出点、折线（line strip）和三角形条（triangle strips）。
- 几何着色器需要图元作为输入，在处理过程中他可以将这个图元整个丢弃或者输出一个或更多的图元（也就是说它可以产生比它得到的更多或更少的顶点）。这个能力被叫做几何增长（growing geometry）。如上所述，几何着色器输出的形式只能是点，折线和三角形条。
- 当我们未添加几何着色器时，默认的行为是将输入的三角形直接输出。我们添加了几何着色器之后，可以在几何着色器中修改输出的图形，我们可以输出我们想要输出的任何图形。

图 3.7.一些几何着色器的应用。左图，使用几何着色器实现元球的等值面曲面细分（metaball isosurface tessellation）。中图，使用了几何着色器和流输出进行线段细分的分形（fractal subdivision of line segments）。右图，使用顶点和几何着色器的流输出进行布料模拟。（图片来自 NVIDIA SDK 10 的示例）

3.2.5.1 流输出 Stream Output

- GPU 的管线的标准使用方式是发送数据到顶点着色器，然后对所得到的三角形进行光栅化处理，并在像素着色器中处理它们。 数据总是通过管线传递，无法访问中间结果。流输出的想法在 Shader Model 4.0 中被引入。在顶点着色器（以及可选的几何着色器中）处理顶点之后，除了将数据发送到光栅化阶段之外，也可以输出到流，也就是一个有序数组中进行处理。事实上，可以完全关掉光栅化，然后管线纯粹作为非图形流处理器来使用。以这种方式处理的数据可以通过管线回传，从而允许迭代处理。如原书的第 10.7 节所述，这种操作特别适用于模拟流动的水或其他粒子特效。

3.2.6 像素着色器 Pixel Shader

- 像素着色器(Pixel Shader，Direct3D 中的叫法)，常常又称为片断着色器,片元着色器(Fragment Shader, OpenGL 中的叫法)，用于进行逐像素计算颜色的操作，让复杂的着色方程在每一个像素上执行。如图 3.1 GPU 渲染管线所示，像素着色器是光栅化阶段的主要步骤之一。在顶点和几何着色器执行完其操

作之后，图元会被裁剪、屏幕映射，结束几何阶段，到达光栅化阶段，在光栅化阶段中先经历三角形设定和三角形遍历，之后来到像素着色阶段。

- 像素着色器常用来处理场景光照和与之相关的效果，如凸凹纹理映射和调色。名称片断着色器似乎更为准确，因为对于着色器的调用和屏幕上像素的显示并非一一对应。举个例子，对于一个像素，片断着色器可能会被调用若干次来决定它最终的颜色，那些被遮挡的物体也会被计算，直到最后的深度缓冲才将各物体前后排序。

- 需要注意，像素着色程序通常在最终合并阶段设置片段颜色以进行合并，而深度值也可以由像素着色器修改。模板缓冲（stencil buffer）值是不可修改的，而是将其传递到合并阶段（Merge Stage）。在 SM 2.0 以及以上版本，像素着色器也可以丢弃（discard ）传入的片段数据，即不产生输出。这样的操作会消耗性能，因为通常在这种情况下不能使用由 GPU 执行的优化 。详见第 18.3.7 节。 诸如雾计算和 alpha 测试的操作已经从合并操作转移到 SM 4.0 中的像素着色器里计算。

- 可以发现，顶点着色程序的输出，在经历裁剪、屏幕映射、三角形设定、三角形遍历后，实际上变成了像素着色程序的输入。在 Shader Model 4.0 中，共有 16 个向量（每个向量含 4 个值）可以从顶点着色器传到像素着色器。当使用几何着色器时，可以输出 32 个向量到像素着色器中。像素着色器的追加输入是在 Shader Model 3.0 中引入的。例如，三角形的哪一面是可见的是通过输入标志来加入的。这个值对于在单个通道中的正面和背面渲染不同材质十分重要。而且像素着色器也可以获得片段的屏幕位置。

3.2.7 合并阶段 The Merging Stage

作为光栅化阶段名义上的最后一个阶段，合并阶段（The Merging Stage）是将像素着色器中生成的各个片段的深度和颜色与帧缓冲结合在一起的地方。这个阶段也就是进行模板缓冲（Stencil-Buffer）和 Z 缓冲（Z-buffer）操作的地方。最常用于透明处理（Transparency）和合成操作（Compositing）的颜色混合（Color Blending）操作也是在这个阶段进行的。

虽然合并阶段不可编程，但却是高度可配置的。在合并阶段可以设置颜色混合来执行大量不同的操作。最常见的是涉及颜色和 Alpha 值的乘法，加法，和减法的组合。其他操作也是可能的，比如最大值，最小值以及按位逻辑运算。

3.2.8 效果 Effect

- GPU 渲染管线中的可编程阶段有顶点、几何和像素着色器三个部分，他们需要相互结合在一起使用。正因如此，不同的团队研发出了不同的特效语言，例如 HLSL FX，CgFX，以及 COLLADA FX，来将他们更好的结合在一起。

- 一个效果文件通常会包含所有执行一种特定图形算法的所有相关信息，而且通常定义一些可被应用程序赋值的全局参数。例如，一个单独的 effect file 可能定义渲染塑料材质需要的 vs(顶点着色器)和 ps（像素着色器），它可能暴露一些参数例如塑料颜色和粗糙度，这样渲染每个模型的时候可以改变效

果而仅仅使用同一个特效文件。一个效果文件中能存储很多 techniques。这些 techniques 通常是一个相同 effect 的变体，每种对应于一个不同的 Shader Model（SM 2.0，SM 3.0 等等）。如图 3.9，使用 Shader 实现的效果。

原书图 3.9 可编程着色器可以实现各种材料和后处理效果

3.3 本章内容提炼总结

以下是对《Real Time Rendering 3rd》第三章“The Graphics Processing Unit 图形处理器”内容概括总结，有必要再次放出这张图：

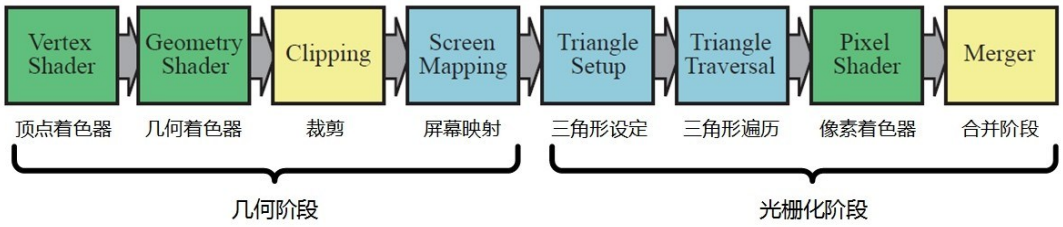


图 3.1 GPU 实现的渲染管线

上图中，不同颜色的阶段表示了该阶段不同属性。其中：

- 绿色的阶段都是完全可编程的。
- 黄色的阶段可配置，但不可编程。
- 蓝色的阶段完全固定。

对每个阶段的分别概述：

- 顶点着色器（The Vertex Shader）是完全可编程的阶段，顶点着色器可以对每个顶点进行诸如变换和变形在内的很多操作，提供了修改/创建/忽略顶点相关属性的功能，这些顶点属性包括颜色、法线、纹理坐标和位置。顶点着色器的必须完成的任务是将顶点从模型空间转换到齐次裁剪空间。
- 几何着色器（The Geometry Shader）位于顶点着色器之后，允许 GPU 高效地创建和销毁几何图元。几何着色器是可选的，完全可编程的阶段，主要对图元（点、线、三角形）的顶点进行操作。几何着色器接收顶点着色器的输出作为输入，通过高效的几何运算，将数据输出，数据随后经过几何阶段和光栅化阶段的其他处理后，会发送给片段着色器。
- 裁剪（Clipping）属于可配置的功能阶段，在此阶段可选运行的裁剪方式，以及添加自定义的裁剪面。
- 屏幕映射（Screen Mapping）、三角形设置（Triangle Setup）和三角形遍历（Triangle Traversal）阶段是固定功能阶段。
- 像素着色器（Pixel Shader，Direct3D 中的叫法）常常又称为片断着色器，片元着色器(Fragment Shader，OpenGL 中的叫法)，是完全可编程的阶段，主要作用是进行像素的处理，让复杂的着色方程在每一个像素上执行。
- 合并阶段（The Merger Stage）处于完全可编程和固定功能之间，尽管不能编程，但是高度可配置，可以进行一系列的操作。其除了进行合并操作，还分管颜色修改（Color Modifying），Z 缓冲（Z-buffer），混合（Blend），模板（Stencil）和相关缓存的处理。

第四章 图形渲染与视觉外观

本章将总结和提炼《Real-Time Rendering 3rd》（实时渲染图形学第三版）的第五章“Visual Appearance（视觉外观）”的内容。

4.1 导读

当我们渲染三维模型的图像时，模型不仅要有适当的几何形状，还应该有所需的视觉外观。《Real-Time Rendering 3rd》第五章内容，讨论光照和材质在现实世界的表现，关于光照和表面模型，着色方程，以及渲染出真实外观的一些额外技术。

简而言之，通过阅读这篇总结式文章，你将对图形渲染中的以下要点有所了解：

- 渲染与视觉物理现象
- 光照与材质
- 着色原理

- 抗锯齿
- 透明渲染
- 伽玛校正

当然，本文作为总结式文章，知识点会相对密集，很多地方对细节并不可能展开描述，对一些地方不太理解的朋友，自然还是推荐是去阅读《Real-Time Rendering 3rd》的对应原文与相应文献。

4.2 渲染与视觉物理现象

当渲染类似图 1 中的逼真场景时，可以帮助了解渲染相关的物理现象。一般情况，这些物理现象分为三种：

- 太阳光与其他光源（天然或人造光）发出光。
- 光与场景中的物体相互作用。部分被吸收；部分散射开来，向新的方向传播。
- 最终，光被传感器（人眼，电子传感器）吸收。

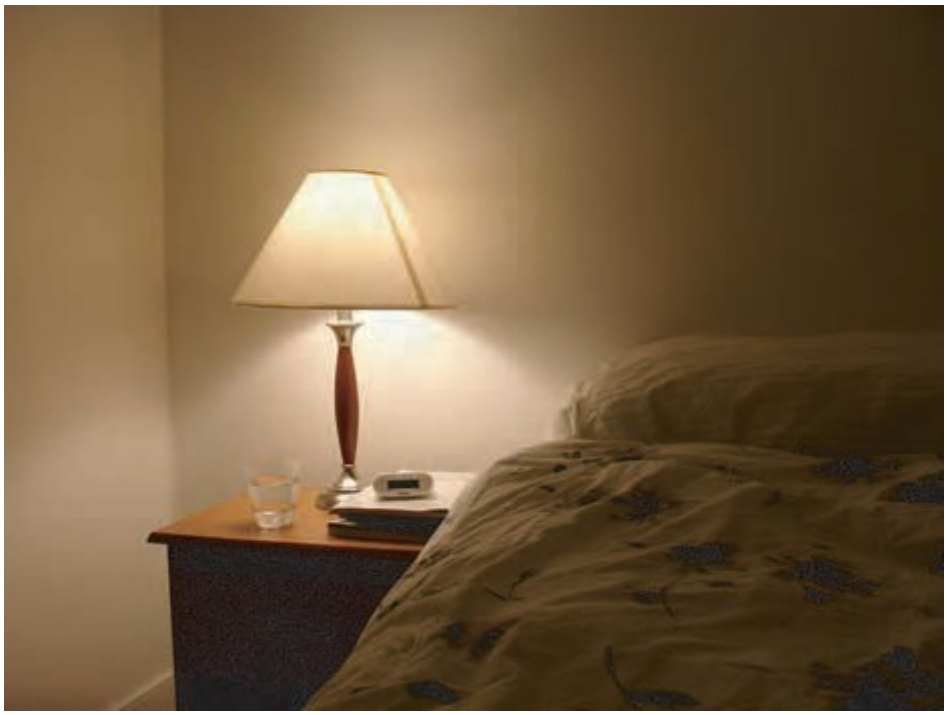


图 1 光源与各种物体交互的卧室照片

而在图 1 中，我们可以看到所有的如上三种物理现象：

- 光线从灯光发出并直接传播给房间里其他物体。

- 物体表面吸收一些物体，并将一些物体散射到新的方向。没有被吸收的光线继续在环境中移动，遇到其他物体。
- 通过场景的光一小部分光进入用于捕获图像的传感器（如摄像机）。

4.3 光照与材质

- 关于光源的特性。光被不同地模拟为几何光线，电磁波或光子（具有一些波特性的量子粒子）。无论如何处理，光都是电磁辐射能–通过空间传播的电磁能。光源发光，而不是散射或吸收光。根据渲染目的，光源可以以许多不同的方式来表示。光源可以分为三种不同类型：平行光源、点光源和聚光灯。
- 关于材质的特性。在渲染中，通过将材质附加到场景中的模型来描绘对象外观。每个材质都和一系列的 Shader 代码，纹理，和其他属性联系在一起，用来模拟光与材质相互作用。

4.3.1 光照现象：散射与吸收

- 从根本上来说，所有的光物质相互作用都是两种现象的结果： 散射（scattering）和吸收（absorption）。
- 散射（scattering）发生在当光线遇到任何种类的光学不连续性（optical discontinuity）时，可能存在于具有不同光学性质的两种物质分界之处，晶体结构破裂处，密度的变化处等。散射不会改变光量，它只是使其改变方向。光的散射（scattering）一般又分为反射（reflection）和折射（refraction）。

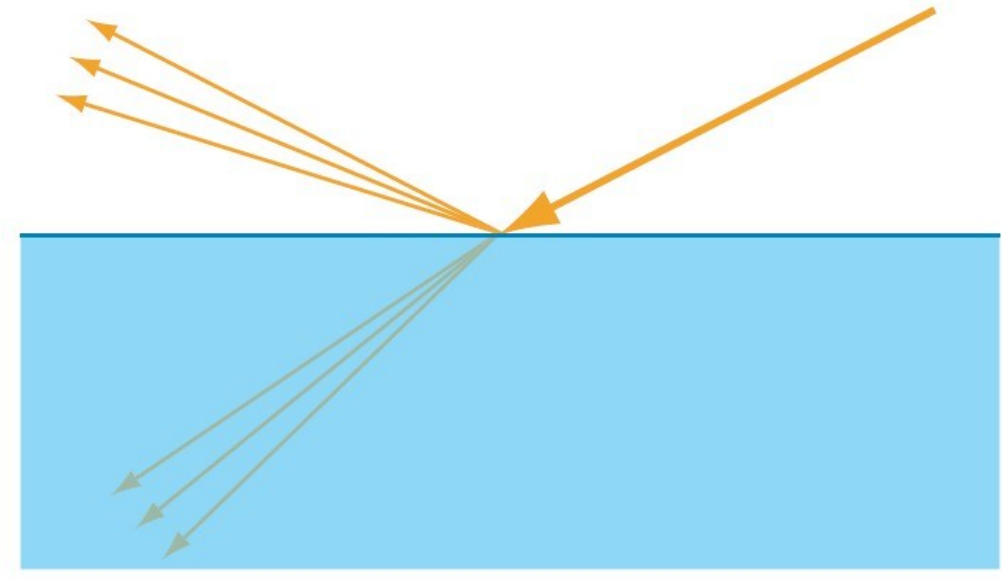


图 2 光的散射（scattering）——反射（reflection）和折射（refraction）

- 吸收（absorption）发生在物质内部，其会导致一些光转变成另一种能量并消失。 吸收会减少光量，但不会影响其方向。

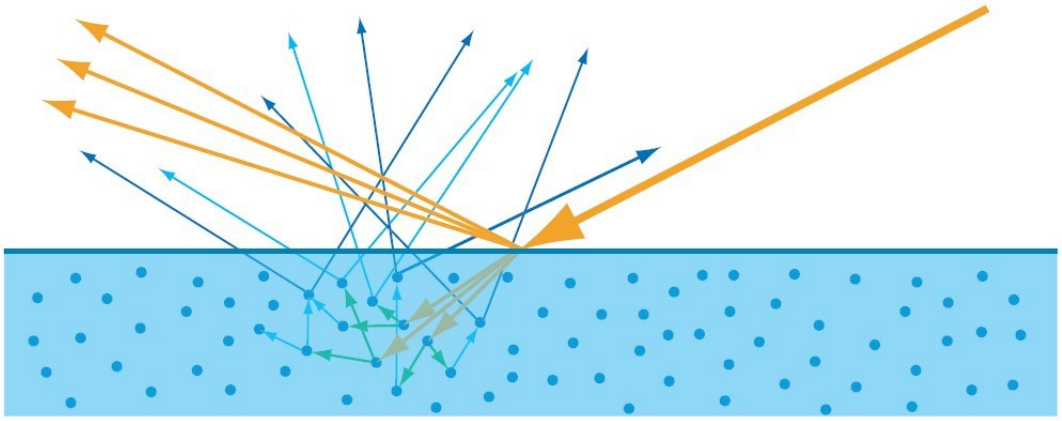


图 3 反射（reflected light）和透射光（transmitted light）的相互作用

- 镜面反射光表示在表面反射的光。而漫反射光表示经历透射（transmission），吸收（absorption）和散射（scattering） 的光。

- 入射光（Incoming illumination）通过表面辉度（irradiance）来度量。而出射光（outgoing light）通过出射率（exitance）来度量，类似于辉度是每单位面积的能量。光物质相互作用是线性的；使辉度加倍将会使出射率增加一倍。出射率除以辉度可以作为材质的衡量特性。对于不发光的表面，该比率为 0 到 1 之间。出射率和辉度的比率对于不同的光颜色是不同的，所以其表示为 RGB 矢量或者颜色，也就是我们通常说的表面颜色 c。

4.3.2 表面

- 镜面反射项的方向分布取决于表面粗糙度（roughness，其反义词是 smoothness，光滑度）。反射光线对于更平滑的表面更加紧密，并且对于较粗糙的表面更加分散。我们可以看到下图中的这种依赖关系，它显示了不同粗糙度的两个表面的反射效果。

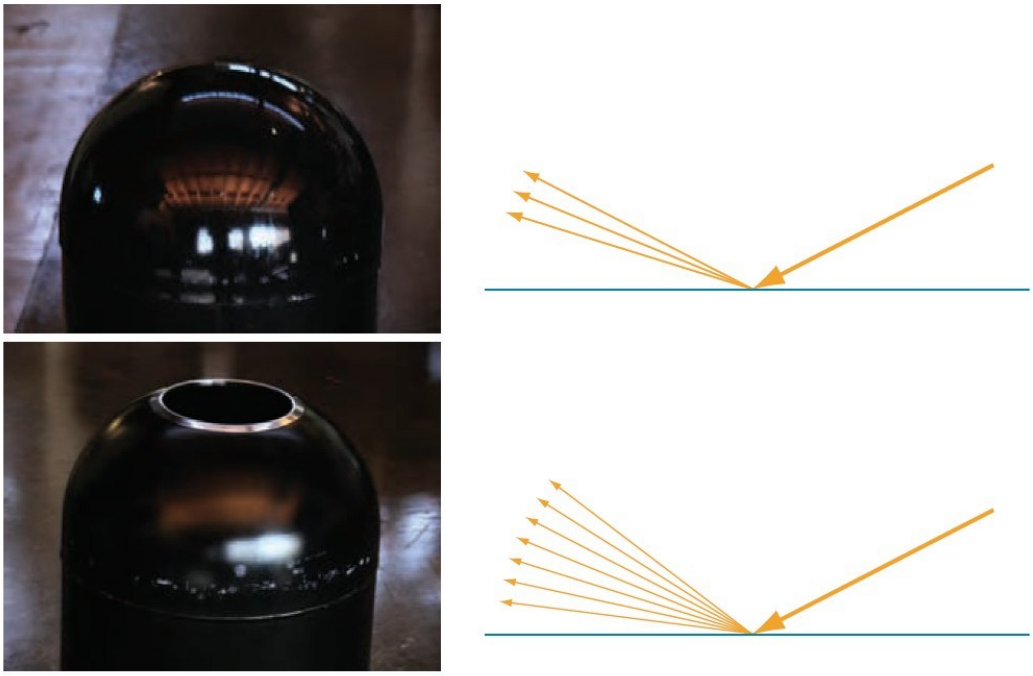


图 4 光在粗糙度不同表面的反射

4.4 着色

4.4.1 着色与着色方程

着色（Shade）是使用方程式根据材质属性和光源，计算沿着视线 v 的出射光亮度 *Lo* 的过程。我们使用的着色方程具有漫反射和镜面反射分量。

4.4.1.1 着色方程的漫反射分量

其中漫反射分量较为简单，书中推导出的对 Ldiff 的着色方程如下：

$$L_{\mathrm{diff}} = \frac{\mathbf{c}_{\mathrm{diff}}}{\pi} \otimes E_L \, \overline{\cos}\, \theta_i .$$

这种类型的漫反射着色也被叫做兰伯特（Lambertian）着色。兰伯特定律指出，对于理想的漫反射表面，出射光亮度与 cos θ i 成正比。注意，这种夹紧型 cos 因子（clamped dot product，可写作 max(n・l, 0)，通常称为 n 点乘 l 因子），不是兰伯特表面的特征;正如我们所见，它一

般适用于辉度（irradiance）的度量。兰伯特表面的决定性特征是出射光亮度（radiance）和辉度（irradiance）成正比。

4.4.1.2 着色方程的镜面反射分量

原书中推导出的镜面反射项的着色方程：

$$L_{\text{spec}}(\mathbf{v}) = \frac{m+8}{8\pi}\,\overline{\cos}^{m}\,\theta_h \mathbf{c}_{\text{spec}} \otimes E_L\,\overline{\cos}\,\theta_i.$$

4.4.1.3 着色方程

组合漫反射和镜面反射两个项，得到完整的着色方程，总出射光亮度 Lo：

$$L_o(\mathbf{v}) = \left(\frac{\mathbf{c}_{\text{diff}}}{\pi} + \frac{m+8}{8\pi}\,\overline{\cos}^{m}\,\theta_h \mathbf{c}_{\text{spec}}\right) \otimes E_L\,\overline{\cos}\,\theta_i.$$

这个着色方程与“Blinn–Phong”方程类似，“Blinn–Phong”方程是 Blinn 在 1977 年首次提出的。主要形式如下：

$$L_o(\mathbf{v}) = \left(\overline{\cos}\,\theta_i \mathbf{c}_{\text{diff}} + \overline{\cos}^{m}\,\theta_h \mathbf{c}_{\text{spec}}\right) \otimes B_L.$$

4.4.2 三种着色处理方法

着色处理是计算光照并由此决定像素颜色的过程，存在 3 种常见的着色处理方法：平滑着色、高洛德着色与冯氏着色。

- 平滑着色（Flat shading）：简单来讲，就是一个三角面用同一个颜色。如果一个三角面的代表顶点(也许是按在 index 中的第一个顶点)，恰好被光照成了白色，那么整个面都会是白的。
- 高洛德着色（Gouraud shading）：每顶点求值后的线性插值结果通常称为高洛德着色。在高洛德着色的实现中，顶点着色器传递世界空间的顶点法线和位置到 Shade( )函数(首先确保法线矢量长度为 1），然后将结果写入内插值。像素着色器将获取内插值并将其直接写入输出。 高洛德着色可以为无光泽表面产生合理的结果，但是对于强高光反射的表面，可能会产生失真（artifacts）。
- 冯氏着色（Phong shading）：冯氏着色是对着色方程进行完全的像素求值。在冯氏着色实现中，顶点着色器将世界空间法线和位置写入内插值，此值通过像素着色器传递给 Shade( )函数。而将 Shade( )函

数返回值写入到输出中。请注意，即使表面法线在顶点着色器中缩放为长度 1，插值也可以改变其长度，因此可能需要在像素着色器中再次执行此归一化操作。

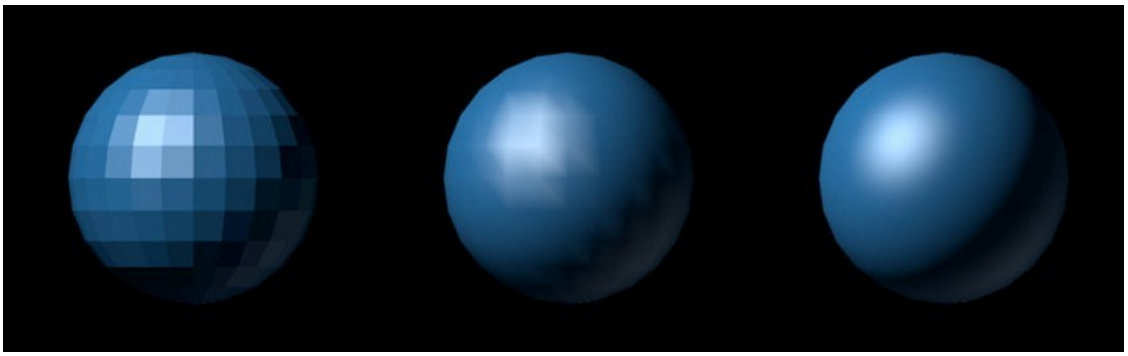


图 5 从左到右，平面着色（Flat shading），高洛德着色（ Gouraud shading），和冯氏着色（Phong shading）

- 注意 Phong Shading 和 Phong Lighting Model 的区别，前者是考虑如何在三个顶点中填充颜色，而后者表示的是物体被光照产生的效果。
- 注意冯氏着色可以说是三者中最接近真实的着色效果，当然开销也是最大的。因为高洛德着色是每个顶点（vertex）计算一次光照，冯氏着色是每个片元（fragment）或者说每像素计算一次光照，点的法向量是通过顶点的法向量插值得到的。所以说不会出现高洛德着色也许会遇到的失真问题。

4.5 抗锯齿与常见抗锯齿类型总结

抗锯齿（英语：anti-aliasing，简称 AA），也译为边缘柔化、消除混叠、抗图像折叠有损，反走样等。它是一种消除显示器输出的画面中图物边缘出现凹凸锯齿的技术，那些凹凸的锯齿通常因为高分辨率的信号以低分辨率表示或无法准确运算出 3D 图形坐标定位时所导致的图形混叠（aliasing）而产生的，抗锯齿技术能有效地解决这些问题。

下面将常见的几种抗锯齿类型进行总结介绍，也包括 RTR3 中没有讲到的，最近几年新提出的常见抗锯齿类型。

4.5.1 超级采样抗锯齿（SSAA）

超级采样抗锯齿（Super-Sampling Anti-Aliasing，简称 SSAA）是比较早期的抗锯齿方法，比较消耗资源，但简单直接。这种抗锯齿方法先把图像映射到缓存并把它放大，再用超级采样把放大后的图像像素进行采样，一般选取 2 个或 4 个邻近像素，把这些采样混合起来后，生成的最终像素，令每个像素拥有邻近像素的特征，像素与像素之间的过渡色彩，就变得近似，令图形的边缘色彩过渡趋于平滑。再把最终像素还原回原来大小的图像，并保存到帧缓

存也就是显存中，替代原图像存储起来，最后输出到显示器，显示出一帧画面。这样就等于把一幅模糊的大图，通过细腻化后再缩小成清晰的小图。如果每帧都进行抗锯齿处理，游戏或视频中的所有画面都带有抗锯齿效果。 超级采样抗锯齿中使用的采样法一般有两种：

- OGSS，顺序栅格超级采样（Ordered Grid Super-Sampling，简称 OGSS），采样时选取 2 个邻近像素。
- RGSS，旋转栅格超级采样（Rotated Grid Super-Sampling，简称 RGSS），采样时选取 4 个邻近像素。

另外，作为概念上最简单的一种超采样方法，全场景抗锯齿（Full-Scene Antialiasing,FSAA）以较高的分辨率对场景进行绘制，然后对相邻的采样样本进行平均，从而生成一幅新的图像。

4.5.2 多重采样抗锯齿（MSAA）

多重采样抗锯齿（Multi Sampling Anti-Aliasing，简称 MSAA），是一种特殊的超级采样抗锯齿（SSAA）。MSAA 首先来自于 OpenGL。具体是 MSAA 只对 Z 缓存（Z-Buffer）和模板缓存(Stencil Buffer)中的数据进行超级采样抗锯齿的处理。可以简单理解为只对多边形的边缘进行抗锯齿处理。这样的话，相比 SSAA 对画面中所有数据进行处理，MSAA 对资源的消耗需求大大减弱，不过在画质上可能稍有不如 SSAA。

4.5.3 覆盖采样抗锯齿（CSAA）

覆盖采样抗锯齿（Coverage Sampling Anti-Aliasing，简称 CSAA）是 NVIDIA 在 G80 及其衍生产品首次推向实用化的 AA 技术，也是目前 NVIDIA GeForce 8/9/G200 系列独享的 AA 技术。CSAA 就是在 MSAA 基础上更进一步的节省显存使用量及带宽，简单说 CSAA 就是将边缘多边形里需要取样的子像素坐标覆盖掉，把原像素坐标强制安置在硬件和驱动程序预先算好的坐标中。这就好比取样标准统一的 MSAA，能够最高效率的执行边缘取样，效能提升非常的显著。比方说 16xCSAA 取样性能下降幅度仅比 4xMSAA 略高一点，处理效果却几乎和 8xMSAA 一样。8xCSAA 有着 4xMSAA 的处理效果，性能消耗却和 2xMSAA 相同。

4.5.4 高分辨率抗锯齿（HRAA）

高分辨率抗锯齿方法(High Resolution Anti-Aliasing，简称 HRAA)，也称 Quincunx 方法，也出自 NVIDIA 公司。“Quincunx”意思是 5 个物体的排列方式，其中 4 个在正方形角上，第五个在正方形中心，也就是梅花形，很像六边模型上的五点图案模式。此方法中，采样模式是五点梅花状，其中四个样本在像素单元的角上，最后一个在中心。

4.5.5 可编程过滤抗锯齿（CFAA）

可编程过滤抗锯齿（Custom Filter Anti-Aliasing，简称 CFAA）技术起源于 AMD-ATI 的 R600 家庭。简单地说 CFAA 就是扩大取样面积的 MSAA，比方说之前的 MSAA 是严格选取物体边缘像素进行缩放的，而 CFAA 则可以通过驱动和谐灵活地选择对影响锯齿效果较大的像素进行缩放，以较少的性能牺牲换取平滑效果。显卡资源占用也比较小。

4.5.6 形态抗锯齿（MLAA）

形态抗锯齿（Morphological Anti-Aliasing，简称 MLAA），是 AMD 推出的完全基于 CPU 处理的抗锯齿解决方案。与 MSAA 不同， MLAA 将跨越边缘像素的前景和背景色进行混合，用第 2 种颜色来填充该像素，从而更有效地改进图像边缘的变现效果。

4.5.7 快速近似抗锯齿（FXAA）

快速近似抗锯齿(Fast Approximate Anti-Aliasing，简称 FXAA) ，是传统 MSAA(多重采样抗锯齿)效果的一种高性能近似。它是一种单程像素着色器，和 MLAA 一样运行于目标游戏渲染管线的后期处理阶段，但不像后者那样使用 DirectCompute，而只是单纯的后期处理着色器，不依赖于任何 GPU 计算 API。正因为如此，FXAA 技术对显卡没有特殊要求，完全兼容 NVIDIA、AMD 的不同显卡(MLAA 仅支持 A 卡)和 DirectX 9.0、DirectX 10、DirectX 11。

4.5.8 时间性抗锯齿（TXAA）

时间性抗锯齿（Temporal Anti-Aliasing，简称 TXAA），将 MSAA、时间滤波以及后期处理相结合，用于呈现更高的视觉保真度。与 CG 电影中所采用的技术类似，TXAA 集 MSAA 的强大功能与复杂的解析滤镜于一身，可呈现出更加平滑的图像效果。此外，TXAA 还能够对帧之间的整个场景进行抖动采样，以减少闪烁情形，闪烁情形在技术上又称作时间性锯齿。目前，TXAA 有两种模式：TXAA 2X 和 TXAA 4X。TXAA 2X 可提供堪比 8X MSAA 的视觉保真度，然而所需性能却与 2X MSAA 相类似；TXAA 4X 的图像保真度胜过 8XMSAA，所需性能仅仅与 4X MSAA 相当。

4.5.9 多帧采样抗锯齿（MFAA）

多帧采样抗锯齿（Multi-Frame Sampled Anti-Aliasing，MFAA）是 NVIDIA 公司根据 MSAA 改进出的一种抗锯齿技术。目前仅搭载 Maxwell 架构 GPU 的显卡才能使用。可以将 MFAA 理解为 MSAA 的优化版，能够在得到几乎相同效果的同时提升性能上的表现。MFAA 与 MSAA 最大的差别就在于在同样开启 4 倍效果的时候 MSAA 是真正的针对每个边缘像素周围的 4 个像素进行采样，MFAA 则是仅仅只是采用交错的方式采样边缘某个像素周围的两个像素。

4.6 透明渲染与透明排序

4.6.1 透明渲染

透明渲染是是图形学里面的常见问题之一，可以从《Real-Time Rendering 3rd》中总结出如下两个算法：

- Screen-Door Transparency 方法。基本思想是用棋盘格填充模式来绘制透明多边形，也就是说，以每隔一个像素绘制一点方式的来绘制一个多边形，这样会使在其后面的物体部分可见，通常情况下，屏幕上的像素比较紧凑，以至于棋盘格的这种绘制方式并不会露馅。同样的想法也用于剪切纹理的抗锯齿边缘，但是在子像素级别中的，这是一种称为 alpha 覆盖（alpha to coverage）的特征。screen-door transparency 方法的优点就是简单，可以在任何时间任何顺序绘制透明物体，并不需要特殊的硬件支持（只要支持填充模式）。缺点是透明度效果仅在 50%时最好，且屏幕的每个区域中只能绘制一个透明物体。

- Alpha 混合（Alpha Blending）方法。这个方法比较常见，其实就是按照 Alpha 混合向量的值来混合源像素和目标像素。当在屏幕上绘制某个物体时，与每个像素相关联的值有 RGB 颜色和 Z 缓冲深度值，以及另外一个成分 alpha 分量，这个 alpha 值也可以根据需要生成并存储，它描述的是给定像素的对象片段的不透明度的值。 alpha 为 1.0 表示对象不透明，完全覆盖像素所在区域; 0.0 表示像素完全透明。为了使对象透明，在现有场景的上方，以小于 1 的透明度进行绘制即可。每个像素将从渲染管线接收到一个 RGBA 结果，并将这个值和原始像素颜色相混合。

4.6.2 透明排序

要将透明对象正确地渲染到场景中，通常需要对物体进行排序。下面分别介绍两种比较基本的透明排序方法（深度缓存和油画家算法）和两种高级别的透明排序算法（加权平均值算法和深度剥离）。

4.6.2.1 深度缓存（Z-Buffer）

Z-Buffer 也称深度缓冲。在计算机图形学中，深度缓冲是在三维图形中处理图像深度坐标的过程，这个过程通常在硬件中完成，它也可以在软件中完成，它是可见性问题的一个解决方法。可见性问题是确定渲染场景中哪部分可见、哪部分不可见的问题。

Z-buffer 的限制是每像素只存储一个对象。如果一些透明对象与同一个像素重叠，那么单独的 Z-buffer 就不能存储并且稍后再解析出所有可见对象的效果。这个问题是通过改变加速器架构来解决的，比如用 A-buffer。A-buffer 具有“深度像素（deep pixels）”，其可以在单个像素中存储一系列呈现在所有对象之后被解析为单个像素颜色的多个片段。但需注意，Z-buffer 是市场的主流选择。

4.6.2.2 画家算法（Painter's Algorithm）

画家算法也称优先填充算法，效率虽然较低，但还是可以有效处理透明排序的问题。其基本思想是按照画家在绘制一幅画作时，首先绘制距离较远的场景，然后用绘制距离较近的场景覆盖较远的部分的思想。画家算法首先将场景中的多边形根据深度进行排序，然后按照顺序进行描绘。这种方法通常会将不可见的部分覆盖，这样就可以解决可见性问题。

4.6.2.3 加权平均值算法（Weighted Average）

使用简单的透明混合公式来实现无序透明渲染的算法，它通过扩展透明混合公式，来实现无序透明物件的渲染，从而得到一定程度上逼真的结果。

4.6.2.4 深度剥离算法（Depth Peeling）

深度剥离是一种对深度值进行排序的技术。它的原理比较直观，标准的深度检测使场景中的 Z 值最小的点输出到屏幕上，就是离我们最近的顶点。但还有离我们第二近的顶点，第三近的顶点存在。要想显示它们，可以用多遍渲染的方法。第一遍渲染时，按照正常方式处理，这样就得到了离我们最近的表面中的每个顶点的 z 值。在第二遍渲染时，把现在每个顶点的深度值和刚才的那个深度值进行比较，凡是小于等于第一遍得到的 z 值，把它们剥离，后面的过程依次类推即可。

图 6 每个深度剥离通道渲染特定的一层透明通道。左侧是第一个 Pass，直接显示眼睛可见的层，中间的图显示了第二层，显示了每个像素处第二靠近透明表面的像素。右边的图是第三层，每个像素处第三靠近透明表面的像素。

4.7 伽玛校正

伽马校正（Gamma correction） 又叫伽马非线性化（gamma nonlinearity），伽马编码（gamma encoding） 或直接叫伽马（gamma），是用来对光线的辉度（luminance）或是三色刺激值（tristimulus values）所进行非线性的运算或反运算的一种操作。为图像进行伽马编码的目的是用来对人类视觉的特性进行补偿，从而根据人类对光线或者黑白的感知，最大化地利用表示黑白的数据位或带宽。

4.8 Reference

[1] https://zh.wikipedia.org/wiki/%E5%8F%8D%E9%8B%B8%E9%BD%92

[2] 如何在 Unity 中分别实现 Flat Shading(平面着色)、Gouraud Shading(高洛德着色)、Phong Shading(冯氏着色)

[3] http://blog.csdn.net/wang15061955806/article/details/50564035

[4] http://blog.csdn.net/xoyojank/article/details/3918091

[5] Gouraud shading – Wikipedia

[6] Phong shading – Wikipedia

[7] http://www.nbb.cornell.edu/neurobio/land/OldStudentProjects/cs490-95to96/guo/report.html

[8] https://zh.wikipedia.org/wiki/%E6%B7%B1%E5%BA%A6%E7%BC%93%E5%86%B2

[9] https://zh.wikipedia.org/wiki/%E4%BC%BD%E7%91%AA%E6%A0%A1%E6%AD%A3

[10] 题图来自《刺客信条：枭雄》

The end.

第五章 纹理贴图及相关技术

在计算机图形学中，纹理贴图（Texturing）是使用图像、函数或其他数据源来改变物体表面外观的技术。这篇文章，将总结和提炼《Real-Time Rendering 3rd》（实时渲染图形学第三版）的第六章“Texturing（纹理贴图）”的内容，讲述纹理贴图与其相关技术的方方面面。

简而言之，通过阅读这篇总结式文章，你将对纹理贴图技术中的以下要点有所了解：

- 纹理管线 The Texturing Pipeline
- 投影函数 The Projector Function
- 映射函数 The Corresponder Function
- 体纹理 Volume Texture
- 立方体贴图 Cube Map
- 纹理缓存 Texture Caching
- 纹理压缩 Texture Compression
- 程序贴图纹理 Procedural Texturing

- Blinn 凹凸贴图 Blinn Bump Mapping
- 移位贴图 Displacement Mapping
- 法线贴图 Normal Mapping
- 视差贴图 Parallax Mapping
- 浮雕贴图 Relief Mapping

5.1 导读

《Real-Time Rendering 3rd》第六章内容“Chapter 6 Texturing”，讨论了纹理贴图的方方面面。在计算机图形学中，纹理贴图是使用图像、函数或其他数据源来改变物体表面外观的技术。例如，可以将一幅砖墙的彩色图像应用到一个多边形上，而不用对砖墙的几何形状进行精确表示。当观察这个多边形的时候，这张彩色图像就出现在多边形所在位置上。只要观察者不接近这面墙，就不会注意到其中几何细节的不足（比如其实砖块和砂浆的图像是显示在光滑的表面上的事实）。通过这种方式将图像和物体表面结合起来，可以在建模、存储空间和速度方面节省很多资源。

图 1 一幅使用了纹理颜色贴图、凹凸贴图和视差贴图等方法来增加画面复杂性和真实感的游戏截图（来自《黑暗之魂 3》）

5.2 纹理管线 The Texturing Pipeline

简单来说，纹理（Texturing）是一种针对物体表面属性进行“建模”的高效技术。图像纹理中的像素通常被称为纹素（Texels），区别于屏幕上的像素。根据 Kershaw 的术语，通过将投影方程（projector function）运用于空间中的点 ，从而得到一组称为参数空间值（parameter-spacevalues）的关于纹理的数值。这个过程就称为贴图（Mapping，也称映射 ）,也就是纹理贴图（Texture Mapping，也称纹理映射 ）这个词的由来。纹理贴图可以用一个通用的纹理管线来进行描述。纹理贴图过程的初始点是空间中的一个位置。这个位置可以基于世界空间，但是更常见的是基于模型空间。因为若此位置是基于模型空间的，当模型移动时，其纹理才会随之移动。

如图 2 为一个纹理管线（The Texturing Pipeline），也就是单个纹理应用纹理贴图的详细过程，而此管线有点复杂的原因是每一步均为用户提供了有效的控制。

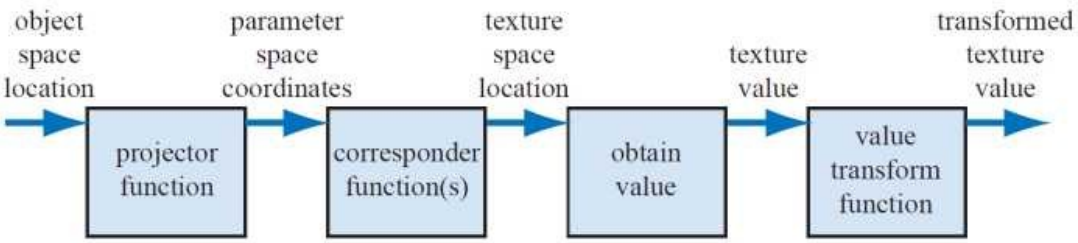


图 2 单个纹理的通用纹理管线

下面是对上图中描述的纹理管线的分步概述：

- 第一步。通过将投影方程（projector function）运用于空间中的点 ，从而得到一组称为参数空间值（parameter-space values）的关于纹理的数值。
- 第二步。在使用这些新值访问纹理之前，可以使用一个或者多个映射函数（corresponder function）将参数空间值（parameter-space values ）转换到纹理空间。
- 第三步。使用这些纹理空间值（texture-space locations）从纹理中获取相应的值（obtain value）。例如，可以使用图像纹理的数组索引来检索像素值。
- 第四步。再使用值变换函数（value transform function）对检索结果进行值变换，最后使用得到的新值来改变表面属性，如材质或者着色法线等等。

而如下这个例子应该对理解纹理管线有所帮助。下例将描述出使用纹理管线，一个多边形在给定一张砖块纹理时在其表面上生成样本时（如图 3）发生了哪些过程。

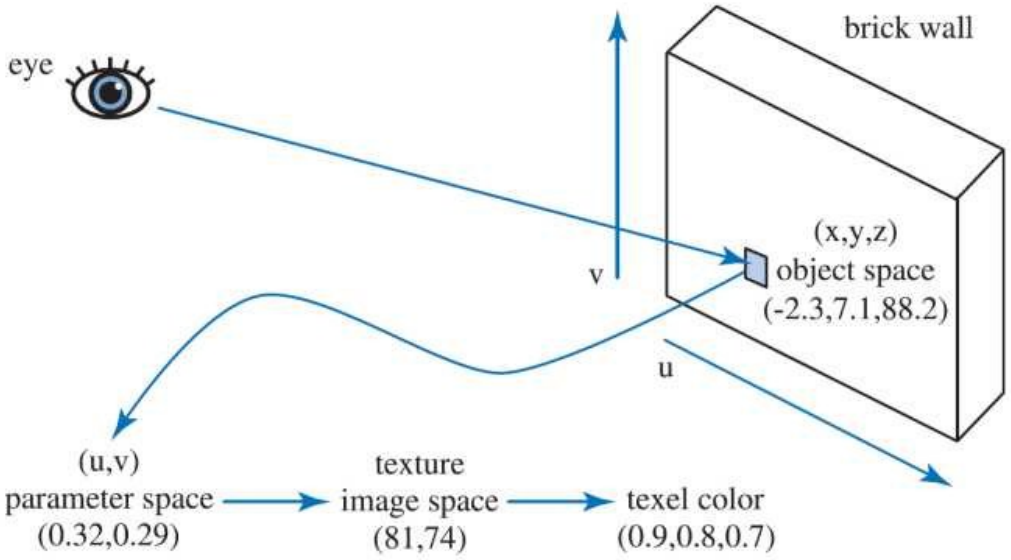


图 3 一个砖墙的纹理管线过程

如图 3 所示，在具体的参考帧画面中找到物体空间中的位置(x,y,z)，如图中点（–2.3,7.1,88.2），然后对该位置运用投影函数。这个投影函数通常将向量(x,y,z)转换为一个二元向量(u,v)。在此示例中使用的投影函数是一个正交投影，类似一个投影仪，将具有光泽的砖墙图像投影到多边形表面上。再考虑砖墙这边，其实这个投影过程就是将砖墙平面上的点变换为值域为 0 到 1 之间的一对(u,v)值，如图，(0.32,0.29)就是这个我们通过投影函数得到的 uv 值。而我们图像的分辨率是 256 x 256，所以，将 256 分别乘以(0.32,0.29)，去掉小数点，得到纹理坐标(81, 74)。通过这个纹理坐标，可以在纹理贴图上查找到坐标对应的颜色值，所以，我们接着找到砖块图像上像素位置为(81,74)处的点，得到颜色(0.9,0.8,0.7)。而由于原始砖墙的颜色太暗，因此可以使用一个值变换函数，给每个向量乘以 1.1，就可以得到我们纹理管线过程的结果——颜色值(0.99,0.88,0.77)。

随后，我们就可以将此值用于着色方程，作为物体的漫反射颜色值，替换掉之前的漫反射颜色。

下面对纹理管线中主要的两个组成，投影函数（The Projector Function）和映射函数（The Corresponder Function）进行概述。

5.2.1 投影函数 The Projector Function

作为纹理管线的第一步，投影函数的功能就是将空间中的三维点转化为纹理坐标，也就是获取表面的位置并将其投影到参数空间中。

在常规情况下，投影函数通常在美术建模阶段使用，并将投影结果存储于顶点数据中。也就是说，在软件开发过程中，我们一般不会去用投影函数去计算得到投影结果，而是直接使用在美术建模过程中，已经存储在模型顶点数据中的投影结果。但有一些特殊情况，例如：

1、OpenGL 的 glTexGen 函数提供了一些不同的投影函数，包括球形函数和平面函数。利用空闲时间可以让图形加速器来执行投影过程，而这样做的优点是不需要将纹理坐标送往图形加速器，从而可以节省带宽。

2、更一般的情况， 可以在顶点或者像素着色器中使用投影函数，这可以实现各种效果，包括一些动画和一些渲染方法（比如如环境贴图，environment mapping，有自身特定的投影函数，可以针对每个顶点或者每个像素进行计算）。

通常在建模中使用的投影函数有球形、圆柱、以及平面投影，也可以选其他一些输入作为投影函数。

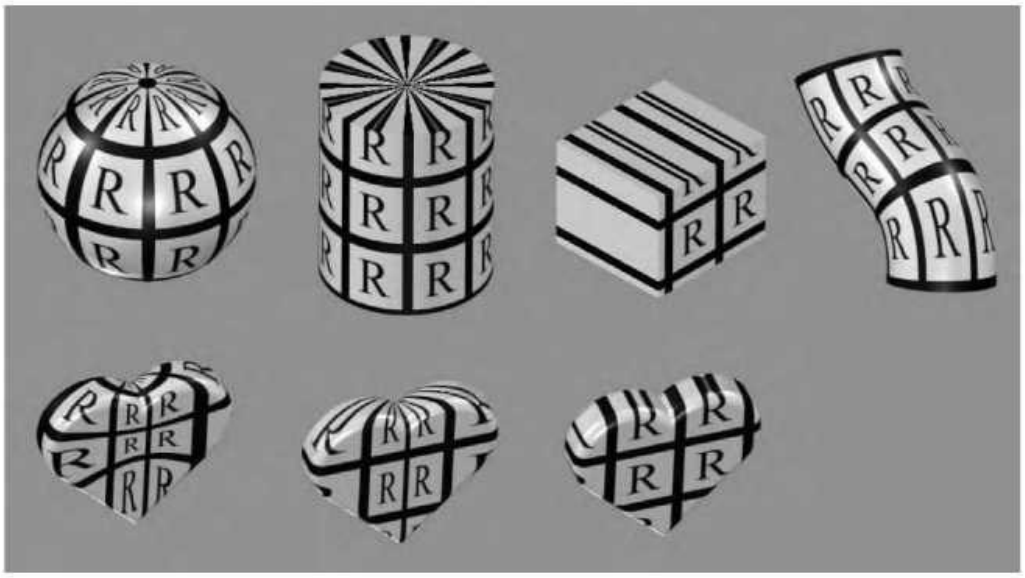


图 4 不同的纹理坐标，上面一行从左到右分别为球形、圆柱、平面，以及自然 uv 投影：下面一行所示为把不同的投影运用于同一个物体的情形。

非交互式渲染器（Noninteractive renderers）通常将这些投影方程称为渲染过程本身的一部分。一个单独的投影方程就有可能适用于整个模型，但其实实际上，美术同学不得不使用各种各样的工具将模型进行分割，针对不同的部分，分别使用不同的投影函数。如图 5 所示。

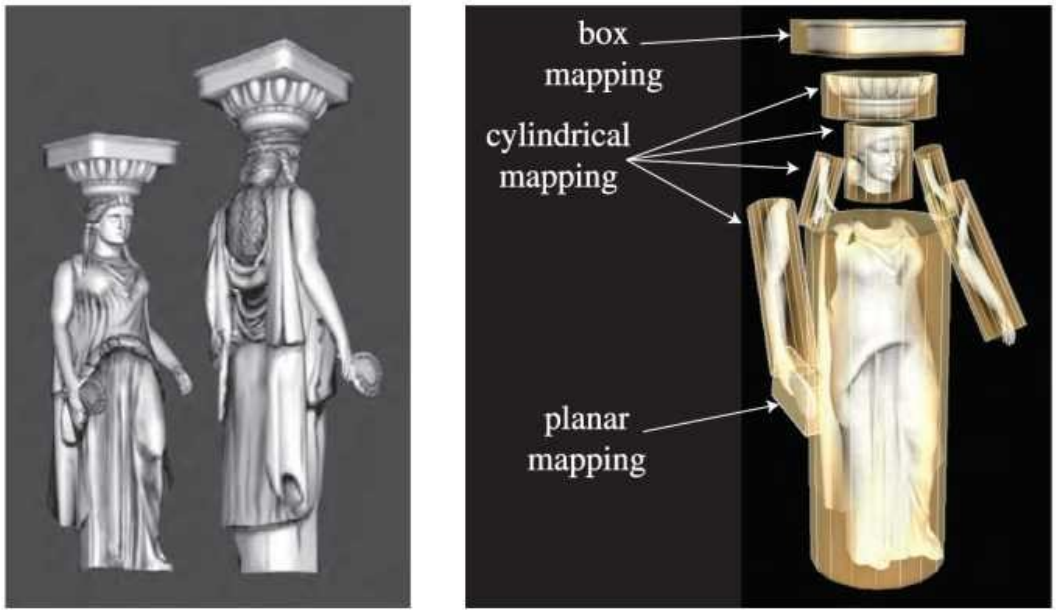


图 5 使用不同的投影函数将纹理以不同的方式投射到同一个模型上

各种常见投影的不同要点：

- 球形投影（The spherical projection）。球形投影将点投射到一个中心位于某个点的虚拟球体上，这个投影与 Blinn 与 Newell 的环境贴图方法相同。
- 圆柱投影（Cylindrical projection）。与球体投影一样，圆柱投影计算的是纹理坐标 u，而计算得到的另一个纹理坐标 v 是沿该圆柱轴线的距离。这种投影方法对具有自然轴的物体比较适用，比如旋转表面，如果表面与圆柱体轴线接近垂直时，就会出现变形。
- 平面投影（The planar projection）。平面投影非常类似于 x-射线幻灯片投影，它沿着一个方向进行投影，并将纹理应用到物体的所有表面上。这种方法通常使用正交投影，用来将纹理图应用到人物上，其把模型看作一个用纸做的娃娃，将不同的纹理粘贴到该模型的前后。

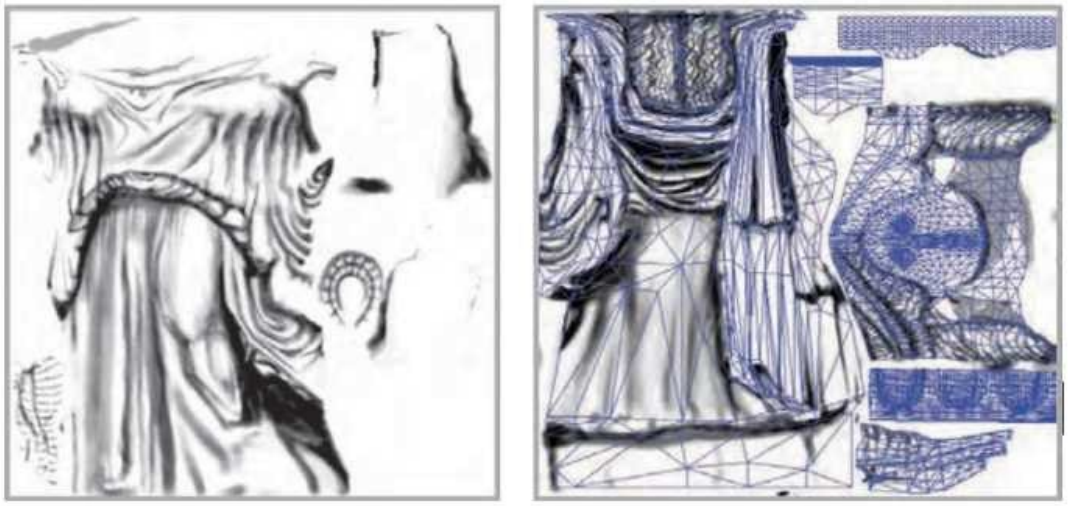


图 6 雕塑模型上的多个较小纹理，保存在两个较大的纹理上。右图显示了多边形网格如何展开并显示在纹理上的。

5.2.2 映射函数 The Corresponder Function

映射函数（The Corresponder Function）的作用是将参数空间坐标（parameter-space coordinates）转换为纹理空间位置（texture space locations）。

我们知道图像会出现在物体表面的(u,v)位置上，且 uv 值的正常范围在[0,1)范围内。超出这个值域的纹理，其显示方式便可以由映射函数（The Corresponder Function）来决定。

在 OpenGL 中，这类映射函数称为“封装模式（Warapping Mode）”，在 Direct3D 中，这类函数叫做“寻址模式（Texture Addressing Mode）”。最常见的映射函数有以下几种：

- 重复寻址模式，wrap (DirectX), repeat (OpenGL)。图像在表面上重复出现。
- 镜像寻址模式，mirror。图像在物体表面上不断重复，但每次重复时对图像进行镜像或者反转。
- 夹取寻址模式，clamp (DirectX) ,clamp to edge (OpenGL)。夹取纹理寻址模式将纹理坐标夹取在[0.0，1.0]之间，也就是说，在[0.0，1.0]之间就是把纹理复制一遍，然后对于[0.0，1.0]之外的内容，将边缘的内容沿着 u 轴和 v 轴进行延伸。
- 边框颜色寻址模式，border (DirectX) ,clamp to border (OpenGL)。边框颜色寻址模式就是在[0.0，1.0]之间绘制纹理，然后[0.0，1.0]之外的内容就用边框颜色填充。

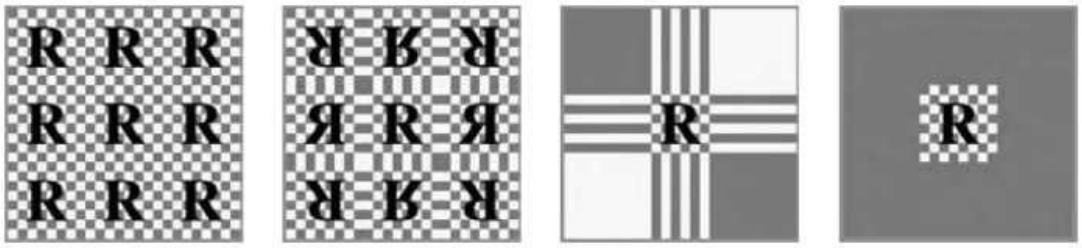


图 7 图像寻址模式，从左到右分别是重复寻址、镜像寻址、夹取寻址、边框颜色寻址

另外，每个纹理轴可以使用不同的映射函数。例如在 u 轴使用重复寻址模式，在 v 轴使用夹取寻址模式。

5.3 体纹理 Volume Texture

三维纹理（3D texture），即体纹理（volume texture），是传统二维纹理（2D texture）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，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 。

虽然体纹理具有更高的储存要求，并且滤波成本更高，但它们具有一些独特的优势：

- 使用体纹理，可以跳过为三维网格确定良好二维参数的复杂过程，因为三维位置可以直接用作纹理坐标，从而避免了二维参数化中通常会发生的变形和接缝问题。
- 体纹理也可用于表示诸如木材或大理石的材料的体积结构。使用三维纹理实现出的这些模型，看起来会很逼真，浑然天成。

劣势：

- 使用体纹理作为表面纹理会非常低效，因为三维纹理中的绝大多数样本都没起到作用。

5.4 立方体贴图 Cube Map

立方体纹理（cube texture）或立方体贴图（cube map）是一种特殊的纹理技术，它用 6 幅二维纹理图像构成一个以原点为中心的纹理立方体，这每个 2D 纹理是一个立方体（cube）的一个面。对于每个片段，纹理坐标(s, t, r)被当作方向向量看待，每个纹素(texel)都表示从原点所看到的纹理立方体上的图像。

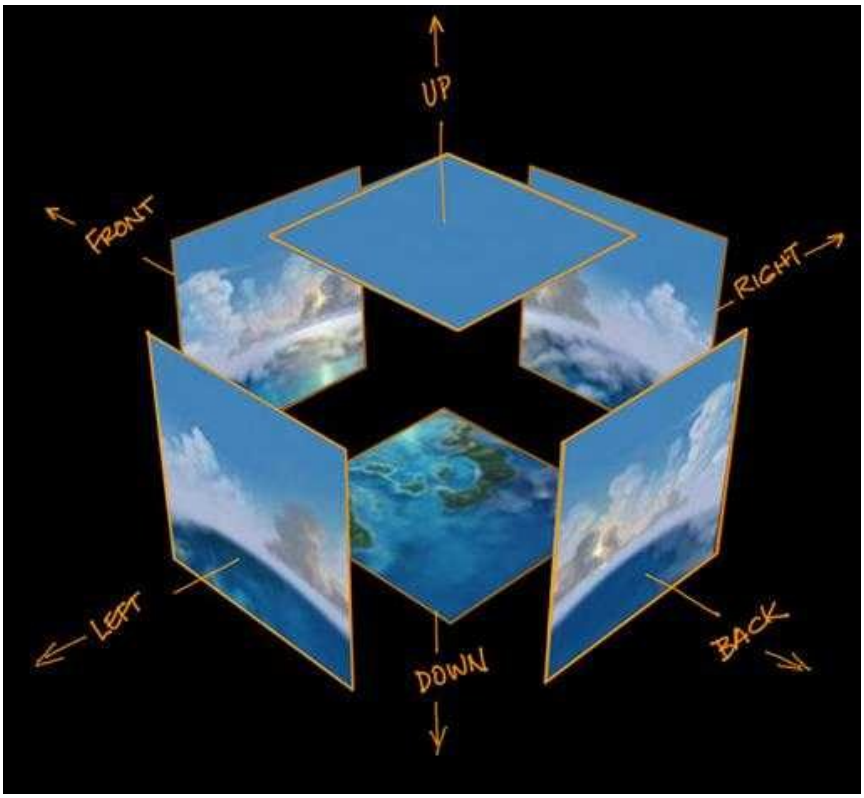


图 8 Cube Map 图示 1

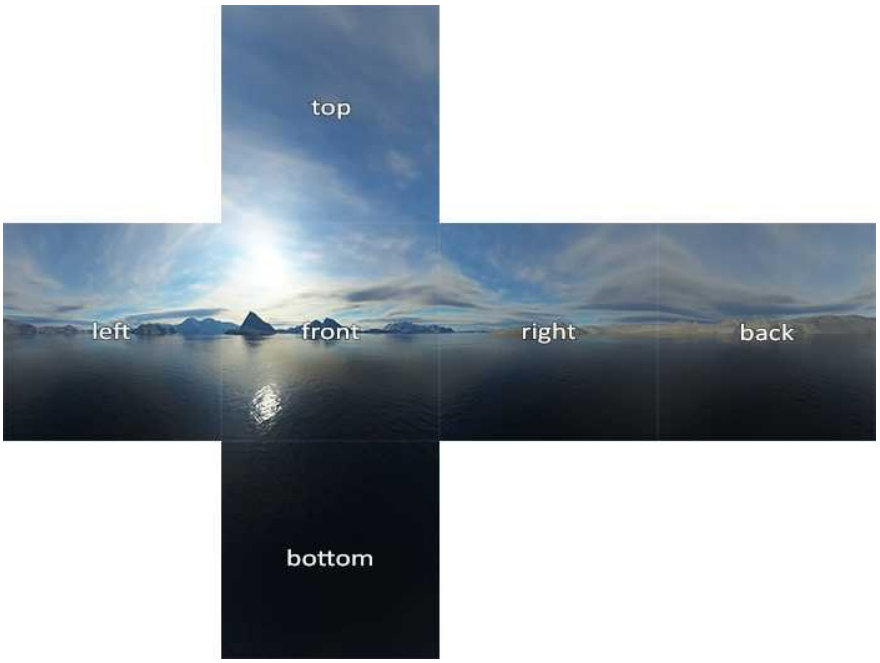


图 9 Cube Map 图示 2

可以使用三分量纹理坐标向量来访问立方体贴图中的数据，该矢量指定了从立方体中心向外指向的光线的方向。选择具有最大绝对值的纹理坐标对应的相应的面。（例如：对于给定的矢量(−3.2, 5.1, −8.4)，就选择–Z 面），而对剩下的两个坐标除以最大绝对值坐标的绝对值，即 8.4。 那么就将剩下的两个坐标的范围转换到了–1 到 1，然后重映射到[0，1]范围，以方便

纹理坐标的计算。例如，坐标（–3.2,5.1）映射到（（–3.2 / 8.4 + 1）/ 2，（5.1/ 8.4 + 1）/ 2）≈（0.31,0.80）。

立方体贴图支持双线性滤波以及 mip mapping，但问题可以可能会在贴图接缝处出现。有一些处理立方体贴图专业的工具在滤波时考虑到了可能的各种因素，如 ATI 公司的 CubeMapGen，采用来自其他面的相邻样本创建 mipmap 链，并考虑每个纹素的角度范围，可以得到比较不错的效果。如图 10。

图 10 立方体贴图过滤。最左边两个图像使用 2 x 2 和 4 x 4 的立方体贴图的纹理层次，采用标准立方体贴图生成 mipmap 链。因接缝显而易见，除了极端细化的情况，这些 mipmap 级别并不可用。两个最右边的图像使用相同分辨率的 mipmap 级别，通过在立方体面和采用角度范围进行采样生成。 由于没有接缝，不易失真，这些 mipmap 甚至可以用于显示在很大的屏幕区域的对象。

5.5 纹理缓存 Texture Caching

一个复杂的应用程序可能需要相当数量的纹理。快速纹理存储器的数量因系统而异，但你会发现它们永远不够用。有各种各样的纹理缓存（texture caching）技术，但我们一直在上传纹理到内存的开销和纹理单次消耗的内存量之间寻求一个好的平衡点。比如，一个由纹理贴图的多边形对象，初始化在离相机很远的位置，程序也许会只加载 mipmap 中更小的子纹理，就可以很完美的完成这个对象的显示了。

一些基本的建议是——保持纹理在不需要放大再用的前提下尽可能小，并尝试基于多边形将纹理分组。即便所有纹理都一直存储在内存中，这种预防措施也可能会提高处理器的缓存性能。

以下是一些常见的纹理缓存使用策略。

5.5.1 最近最少使用策略（Least Recently Used ,LRU）

最近最少使用（Least Recently Used ,LRU）策略是纹理缓存方案中常用的一种策略，其作用如下。加载到图形加速器的内存中的每个纹理都被给出一个时间戳，用于最后一次访问以渲染图像时。当需要空间来加载新的纹理时，首先卸载最旧时间戳的纹理。一些 API 还允许为每个纹理设置一个优先级：如果两个纹理的时间戳相同，则优先级较低的纹理首先被卸载。 设置优先级可以帮助避免不必要的纹理交换。

5.5.2 最近最常使用策略（Most Recently Used，MRU）

如果开发自己的纹理管理器，Carmack（就是那个游戏界大名鼎鼎的卡马克）提出了一种非常有用的策略，也就是对交换出缓冲器的纹理进行核查 [具体可见原书参考文献 374]。大概思想是这样：鉴于如果在当前帧中载入纹理，会发生抖动（Thrashing）的情况。这种情况下，LRU 策略是一种非常不好的策略，因为在每帧画面中会对每张纹理图像进行交换。在这种情况下，可以采用最近最常使用（Most Recently Used，MRU）策略，直到在画面中没有纹理交换时为止，再然后切换回 LRU。

5.5.3 预取策略（Prefetching）

加载纹理花费显着的时间，特别是在需将纹理转换为硬件原生格式时。 纹理加载在每个框架可以有很大的不同。在单个帧中加载大量纹理使得难以保持恒定的帧速率。一种解决方案是使用预取（prefetching），在将来需要预期的情况下，预计未来的需求然后加载纹理，将加载过程分摊在多帧中。

5.5.4 裁剪图策略（Clipmap）

对于飞行模拟和地型模拟系统，图像数据集可能会非常巨大。传统的方法是将这些图像分解成更小的硬件可以处理的瓦片地图（tiles）。Tanner 等人提出了一种一种称为裁剪图（clipmap）的改进数据结构。其思想是，将整个数据集视为一个 mipmap，但是对于任何特定视图，只需要 mipmap 的较低级别的一小部分即可。支持 DirectX 10 的 GPU 就能够实现 clipmap 技术。用这种技术制作的图像如图 11 所示。

图 11 高分辨率地形图访问海量图像数据库。使用 clipmapp 技术可以减少在同一时间所需数据量。

5.6 纹理压缩 Texture Compression

直接解决内存和带宽问题和缓存问题的一个解决方案是固定速率纹理压缩（Fixed-rate Texture Compression）。通过硬件解码压缩纹理，纹理可以需要更少的纹理内存，从而增加有效的高速缓存大小。至少这样的纹理使用起来更高效，因为他们在访问时消耗更少的内存带宽。

有多种图像压缩方法用于图像文件格式，如 JPEG 和 PNG，但在硬件上对其实现解码会非常昂贵。S3 公司开发一种名为 S3 纹理压缩（S3 Texture Compression，S3TC）的方法，目前已经被选为 DirectX 中的标准压缩模式，称为 DXTC。在 DirectX 10 中，这种方法称为 BC (Block Compression)。其优点是创建了一个固定大小，具有独立的编码片段，并且解码简单，同时速度也很快。每个压缩部分的图像可以独立处理，没有共享查找表（look-up tables）或其他依赖关系，这同样地简化了解码过程。

还有几种 S3/DXTC/BC 压缩方案的变种存在，他们有一些共同的特征。把纹理按 4x4 个单元（纹素）大小划分为块。每个块对应一张四色查找表，表中存有两个标准 RGB565 格式表示的 16 位颜色，另外使用标准插入算法在插入两个新的颜色值，由此构成四色查找表。4x4 大小的纹理块中每个单元（像素点）用两个 bit 表示，每一个都代表四色查找表中的一种颜色。可以看出，实质上是利用每个单元（像素点）中的两个 bit 来索引四色查找表中的颜色值。

这些压缩技术可以应用于立方体或体积图，以及二维纹理。而其主要缺点是它们是有损的压缩。 也就是说，原始图像通常不能从压缩版本检索。 仅使用四个或八个内插值来表示 16 个像素。 如果一个瓦片贴图有更大的数值，相较压缩前就会有一些损失。 在实践中，如果正常使用这些压缩方案，一般需给出可接受的图像保真度。

DXTC 的一个问题是用于块的所有颜色都位于 RGB 空间的直线上。例如，DXTC 不能在一个块中同时表示红色，绿色和蓝色。

下面对几种不同纹理压缩变体(S3/DXTC/BC 以及 ETC)在编码上的异同点分节概述。

5.6.1 DXT1

DXT1（DirectX 9.0）或 BC1（DirectX 10.0 及更高版本） – 每个块具有两个 16 位参考 RGB 值（5 位红，6 绿，5 蓝）的纹素，而每个纹素具有 2 位插值因子，以便从一个参考值或两个中间值之间选择。DXT1 作为五种变体中最精简的版本，块占用为 8 个字节，即每个纹素占用 4 位。 与未压缩的 24 位 RGB 纹理相比，有着 6：1 的纹理压缩率。

5.6.2 DXT3

DXT3（DirectX 9.0）或 BC2（DirectX 10.0 及更高版本） – 每个块都具有与 DXT1 块相同的 RGB 数据编码。另外，每个纹素都具有单独存储 4 位 alpha 值（这是唯一的直接存储数据的形式，而不是用插值的形式）。DXT3 块占用 16 个字节，或每个纹理元素 8 位。与未压缩的 32 位 RGBA 纹理相比，有着 4：1 的纹理压缩率。

5.6.3 DXT5

DXT5（DirectX 9.0）或 BC3（DirectX 10.0 及更高版本） – 每个块都具有与 DXT1 块相同的 RGB 数据编码。此外，alpha 数据使用两个 8 位参考值和一个每纹素 3 位的插值因子进行编码。每个纹素可以选择参考 alpha 值之一或六个中间值之一作为其值。DXT5 块具有与 DXT3 块相同的存储要求，也就是 DXT3 块占用 16 个字节，即每个纹理元素 8 位。与未压缩的 32 位 RGBA 纹理相比，有着 4：1 的纹理压缩率。

5.6.4 ATI1

ATI1（ATI 公司的特定扩展名）或 BC4（DirectX 10.0 及更高版本）– 每个块存储单个颜色的数据通道，以与 DXT5 中的 alpha 数据相同的方式进行编码。 BC4 块占用 8 个字节，即每个纹素占用 4 位。与未压缩的 8 位单通道纹理相比，有着 4：1 的纹理压缩率。仅在较新的 ATI 的硬件或任意供应商的 DirectX 10.0 硬件上才支持此格式。

5.6.5 ATI2

ATI2（ATI 公司的特定扩展名，也称为 3Dc）或 BC5（DirectX10.0 及更高版本） – 每个块存储两个颜色通道的数据，以与 BC4 块或 DXT5 中的 alpha 数据相同的方式进行编码。 BC5 块占用 16 个字节，即每个纹理元素 8 位。与未压缩的 16 位双通道纹理相比，有着 4：1 的纹理压缩率。也仅在较新的 ATI 的硬件或任意供应商的 DirectX 10.0 硬件上才支持此格式。

5.6.6 ETC

对于 OpenGL ES，选择了另一种称为 ETC（Ericsson texture compression，ETC）的压缩算法。方案与 S3TC 具有相同的特点，即快速解码，随机访问，无间接查找，速率固定。ETC 算法将 4×4 纹素的块编码为 64 位，即每个纹理元素 4 位。基本思想如图 12 所示。

图 12 ETC 算法对像素块的颜色进行编码，然后修改每像素的亮度以创建最终的纹理颜色

每个 2×4 块（或 4×2，取决于哪个质量更佳）存储基本颜色。每个块还从一个小的静态查找表中选择一组四个常量，并且块中的每个纹素可以选择在选定的查找表中添加一个值，添加的这个值就可以是每像素的亮度。

也可以这样理解：ETC 压缩算法将图像中的 chromatic 和 luminance 分开存储的方式，而在解码时使用 luminance 对 chromatic 进行调制进而重现原始图像信息。

两个要点：

- ETC 的图片压缩的质量和 DXTC 相当。
- ETC 也主要有两种方法：ETC1 和改进后的 ETC2。

5.7 程序贴图纹理 Procedural Texturing

程序贴图纹理（Procedural Texturing，也可译为过程纹理）是用计算机算法生成的，旨在创建用于纹理映射的自然元素（例如木材，大理石，花岗岩，金属，石头等）的真实表面或三维物体而创建的纹理图像。通常，会使用分形噪声（fractal noise）和湍流扰动函数（turbulence functions）这类“随机性”的函数来生成程序贴图纹理。

给定纹理空间位置，进行图像查找是生成纹理值的一种方法。另一种方法是对函数进行求值，从而得到一个程序贴图纹理（procedural texture）。

过程纹理主要用于模拟自然界中常见的 Marble,Stone,Wood,Cloud 等纹理。大多数的过程纹理都是基于某类噪声函数（Noise Function），比如说 perlin noise。在过去，由于过程纹理计算量很大，在实时绘制中很少使用。但是 GPU 的出现，促进了过程纹理在实时渲染中的广泛应用。

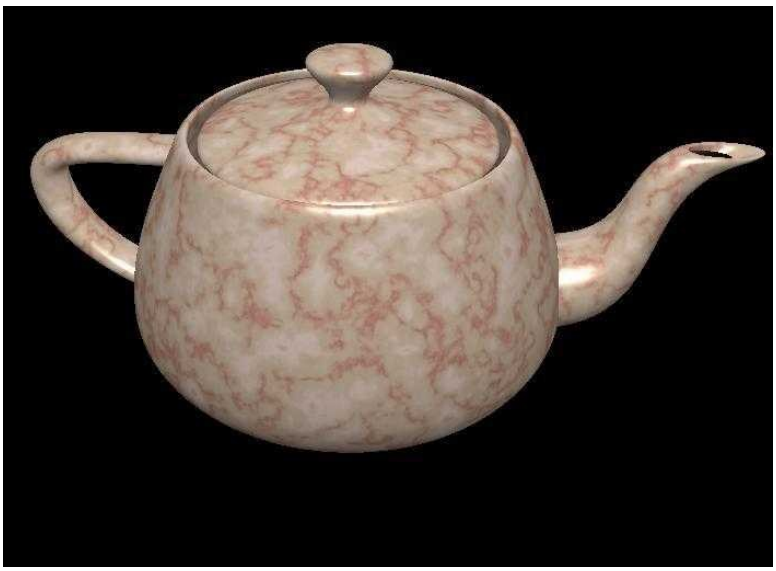


图 13 程序贴图纹理示例

程序贴图纹理通常用于离线渲染应用程序，而图像纹理在实时渲染中更为常见。这是由于在现代 GPU 中的图像纹理硬件有着极高效率，其可以在一秒钟内执行数十亿个纹理访问。然而，GPU 架构正在朝着更便宜的计算能力和（相对）更昂贵的存储器访问而发展。这将使程序纹理在实时应用程序中更常见，尽管它们不可能完全替代图像纹理。

考虑到体积图像纹理的高存储成本，体积纹理是用于程序贴图纹理是一项特别有吸引力的应用。这样的纹理可以通过各种技术来合成，最常见的是使用一个或多个噪声函数来产生纹理的值。

5.8 凹凸贴图与其改进

凹凸贴图（Bump Mapping）思想最早是由图形学届大牛中的大牛 Jim Blinn 提出，后来的 Normal Mapping，Parallax Mapping，Parallax Occulision Mapping，Relief Mapping 等等，均是基于同样的思想，只是考虑得越来越全面，效果也越来越逼真。

以下是几种凹凸贴图与其改进方法的总结对比。

贴图方式	思想概述	提出年代
Bump mapping 凹凸贴图	计算 vertex 的光强时，不是直接使用该 vertex 的原始法向量，而是在原始法向量上加上一个扰动得到修改法向量，经过光强计算，能够产生凹凸不平的表面效果。No self-occlusion, No self-shadow, No silhouette。	1978
Displacement Mapping 移位贴图	直接作用于 vertex，根据 displacement map 中相对应 vertex 的像素值，使 vertex 沿法向移动，产生真正的凹凸表面。	1984
Normal Mapping 法线贴图	normal map 需要法向量的信息，而法向量信息可由 height map 得到，且 texture 的 RGB 可以表示法向量的 XYZ，利用此信息计算光强，产生凹凸阴影的效果。No self-occlusion, No self-shadow, No silhouette。	1996
Parallax Mapping (Virtual Displacement Mapping) 视差贴图	没有修改 vertex 的位置，以视线和 height map 计算较陡峭的视角给 vertex 较多的位移，较平缓的视角给 vertex 较少的位移，透过视差获得更强的立体感，即利用 HeightMap 进行了近似的 Texture Offset。No self-occlusion, No self-shadow。	2001
Relief Mapping (Steep Parallax Mapping) 浮雕贴图	更精确地找出观察者的视线与高度的交点，对应的 texture 坐标则是位移的距离，所以能更正确地模拟立体的效果。Relief Mapping 实现了精确的 Texture Offset。Relief Mapping 可以产生 self-occlusion, self-shadowing, view-motion parallax, and silhouettes。	2005

除了 Displacement Mapping 方法以外，其他的几种改进一般都是通过修改每像素着色方程来实现，关键思想是访问纹理来修改表面的法线，而不是改变光照方程中的颜色分量。物体表面的几何法线保持不变，我们修改的只是照明方程中使用的法线值。他们比单独的纹理有更好的三维感官，但是显然还是比不上实际的三维几何体。

以下是各个方法分别的原理和特性说明。

5.8.1 凹凸贴图 Bump Mapping

凹凸贴图是指计算机图形学中在三维环境中通过纹理方法来产生表面凹凸不平的视觉效果。它主要的原理是通过改变表面光照方程的法线，而不是表面的几何法线，或对每个待渲染的像素在计算照明之前都要加上一个从高度图中找到的扰动，来模拟凹凸不平的视觉特征，如褶皱、波浪等等。

Blinn 于 1978 年提出了凹凸贴图方法。使用凹凸贴图，是为了给光滑的平面，在不增加顶点的情况下，增加一些凹凸的变化。他的原理是通过法向量的变化，来产生光影的变化，从而产生凹凸感。实际上并没有顶点（即 Geometry）的变化。

表示凹凸效果的另一种方法是使用高度图来修改表面法线的方向。每个单色纹理值代表一个高度，所以在纹理中，白色表示高高度区域，黑色是低高度的区域（反之亦然）。示例如图 14。

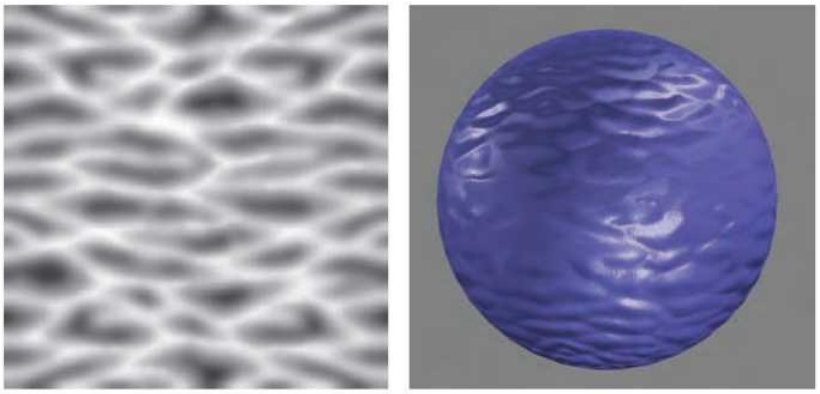


图 14 波浪高度凹凸贴图以及其在材质上的使用

5.8.2 移位贴图 Displacement Mapping

Displacement Mapping，移位贴图，也有人称为置换贴图，或称高度纹理贴图（Heightfield Texturing）。这种方法类似于法线贴图，移位贴图的每一个纹素中存储了一个向量，这个向量代表了对应顶点的位移。注意，此处的纹素并不是与像素一一对应，而是与顶点一一对应，因此，纹理的纹素个数与网格的顶点个数是相等的。在 VS 阶段，获取每个顶点对应的纹素中的位移向量，（注意，直到 3.0 版本的 vs 才支持纹理数据的获取，之前的版本只有 ps 才能获取纹理数据），施加到局部坐标系下的顶点上，然后进行世界视点投影变换即可。

5.8.3 法线贴图 Normal Mapping

法线贴图（Normal mapping）是凸凹贴图（Bump mapping）技术的一种应用，法线贴图有时也称为“Dot3（仿立体）凸凹纹理贴图”。凸凹与纹理贴图通常是在现有的模型法线添加扰动不同，法线贴图要完全更新法线。与凸凹贴图类似的是，它也是用来在不增加多边形的情况下在浓淡效果中添加细节。但是凸凹贴图通常根据一个单独的灰度图像通道进行计算，而法线贴图的数据源图像通常是从更加细致版本的物体得到的多通道图像，即红、绿、蓝通道都是作为一个单独的颜色对待。

简单来说，Normal Map 直接将正确的 Normal 值保存到一张纹理中去，那么在使用的时候直接从贴图中取即可。

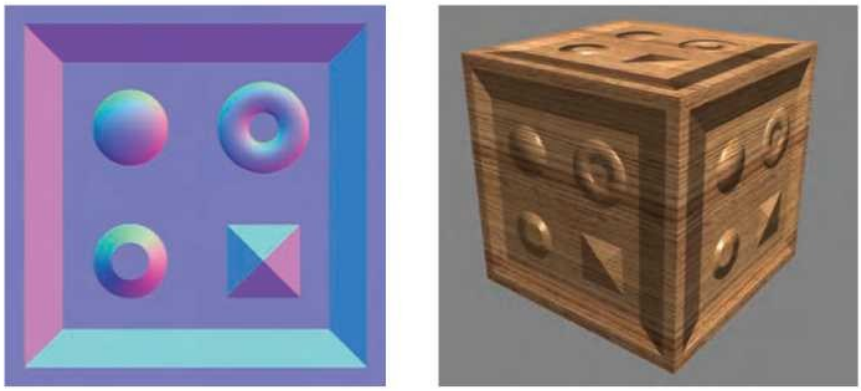


图 15 基于法线贴图的凹凸映射，每个颜色通道实际上是表面法线坐标。红色通道是 x 偏差; 红色越多，正常点越多。 绿色是 y 偏差，蓝色是 z。 右边是使用法线贴图生成的图像。 请注意立方体顶部的扁平外观。

5.8.4 视差贴图 Parallax Mapping

视差贴图 Parallax Mapping，又称为 Offset Mapping，以及 virtual displacement mapping)，于 2001 年由 Kaneko 引入，由 Welsh 进行了改进和推广。视差贴图是一种改进的 Bump Mapping 技术，相较于普通的凹凸贴图，视差贴图技术得到凹凸效果得会更具真实感（如石墙的纹理将有更明显的深度）。视差贴图是通过替换渲染多边形上的顶点处的纹理坐标来实现的，而这个替换依赖于一个关于切线空间中的视角（相对于表面法线的角度）和在该点上的高度图的方程。简单来说，Parallax Mapping 利用 Height Map 进行了近似的 Texture Offset。如图 6.32。

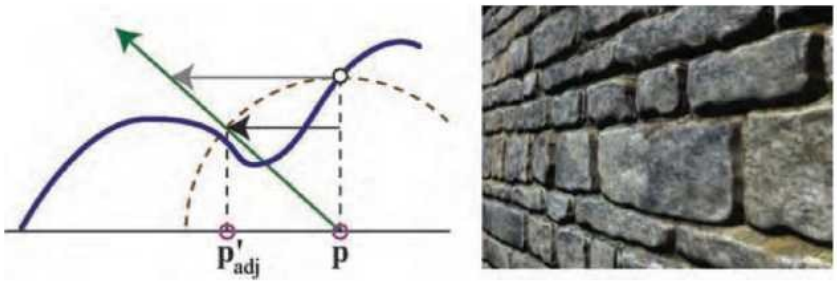


图 16 视差贴图

5.8.5 浮雕贴图 Relief Mapping

关于浮雕贴图（Relief Mapping），有人把它誉为凹凸贴图的极致。我们知道，Parallax Mapping 是针对 Normal Mapping 的改进，利用 HeightMap 进行了近似的 Texture Offset。而 Relief Mapping 是精确的 Texture Offset，所以在表现力上比较完美。

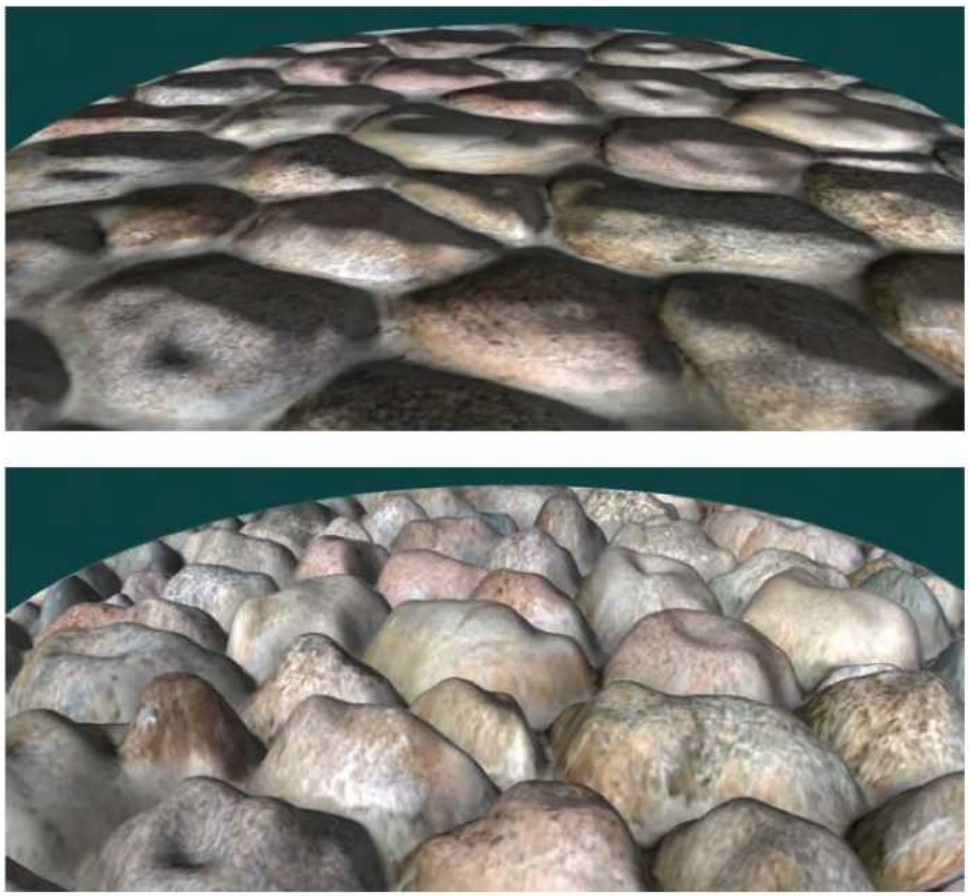


图 17 法线贴图和浮雕贴图的对比。法线贴图不发生自遮挡。

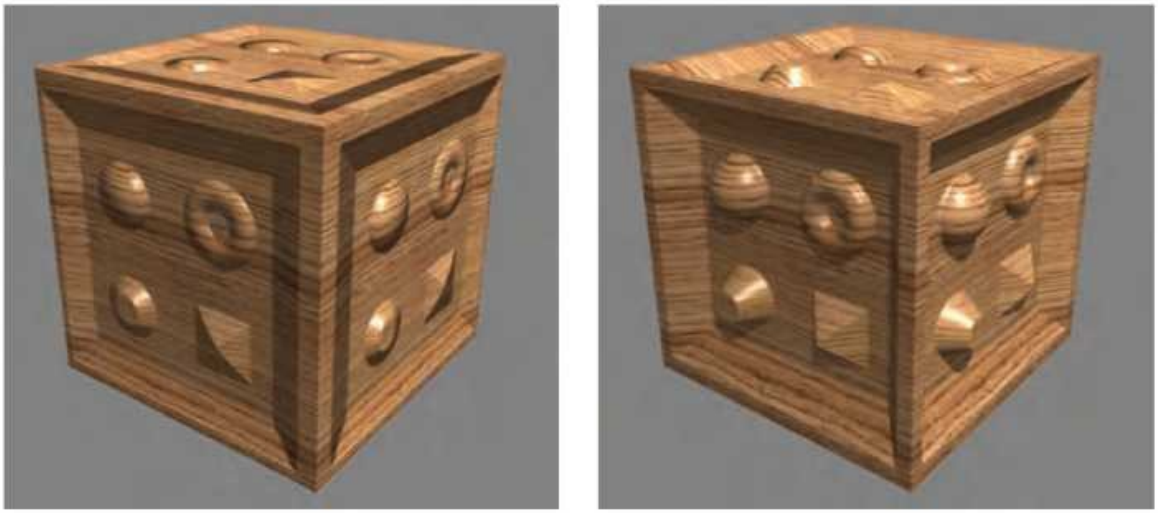


图 18 相较于视差贴图（左），浮雕贴图（右）可以实现更深的凹凸深度。

Parallax Mapping 能够提供比 Bump Mapping 更多的深度,尤其相比于小视角下,但是如果想提供更深的深度,Parallax Mapping 就无能为力了，Relief Mapping 则可以很好的胜任。相较于 Parallax Mapping，浮雕贴图（Relief Mapping）可以实现更深的凹凸深度。浮雕贴图方法不仅更容易提供更深的深度,还可以做出自阴影和闭塞效果,当然算法也稍稍有点复杂，具体细节可以参考这篇中文文献：Relief mapping:凹凸贴图的极致，而如果要用一句话概括 Relief Mapping，将会是：“在 Shader 里做光线追踪”。

图 19 浮雕贴图，使石块看起来更逼真

5.9 Reference

[1] Learn OpenGL, extensive tutorial resource for learning Modern OpenGL

[2] 体绘制（Volume Rendering）概述

[3] 几种主流贴图压缩算法的实现原理 – BugRunner 的专栏 – 博客频道 – CSDN.NET

[4] 过程纹理(Procedural Texture) [2005-11-20 update]

[5] https://zh.wikipedia.org/wiki/%E6%B3%95%E7%BA%BF%E8%B4%B4%E5%9B%BE

[6] Relief mapping:凹凸贴图的极致

[7] Bump Mapping 综述 – Just a Programer – 博客园

第六章 高级着色：BRDF 及相关技术

在计算机图形学中，BRDF（Bidirectional Reflectance Distribution Function，双向反射分布函数）是真实感图形学中最核心的概念之一，它描述的是物体表面将光能从任何一个入射方向反射到任何一个视点方向的反射特性，即入射光线经过某个表面反射后如何在各个出射方向上分布。BRDF 模型是绝大多数图形学算法中用于描述光反射现象的基本模型。

这篇文章，将专注于总结和提炼《Real-Time Rendering 3rd》（实时渲染图形学第三版）中第七章“Advanced Shading · 高级着色”的内容，并对这章中介绍 BRDF 的内容进行适当补充和引申，构成全文，成为一个对 BRDF 进行近乎系统式总结的文章。

图 1 基于 BRDF 渲染的场景图 ©Disney 2014.《超能陆战队》

6.1 导读

简而言之，通过阅读这篇总结式文章，你将对 BRDF 的以下要点有所了解：

- 一、BRDF 的前置知识 · 数学篇
  - 球面坐标 Spherical Coordinate
  - 立体角 Solid Angle
  - 投影面积 Foreshortened Area
- 二、BRDF 前置知识 · 辐射度量学篇
  - 辐射度量学基本参数表格
  - 辐射通量/光通量 Radiant Flux
  - 辐射强度/发光强度 Radiant Intensity
  - 辐射率/光亮度 Radiance
  - 辐照度/辉度 Irradiance
- 三、BRDF 的定义与理解
  - BRDF 的定义式
  - BRDF 的非微分形式
  - BRDF 与着色方程
  - BRDF 的可视化表示
- 四、BRDF 的性质
  - 亥姆霍兹光路可逆性
  - 能量守恒性质
  - 线性特征
- 五、BRDF 模型的分类
  - BRDF 经验模型
  - 数据驱动的 BRDF 模型
  - 基于物理的 BRDF 模型
- 六、基于物理的 BRDF · 前置知识

- 次表面散射 Subsurface Scattering
- 菲涅尔反射 Fresnel Reflectance
- 微平面理论 Microfacet Theory

• 七、基于物理的 BRDF · 常见模型
- Cook–Torrance BRDF 模型
- Ward BRDF 模型

• 八、BRDF 与其引申
- BSSRDF
- SBRDF(SVBRDF)
- BTDF 与 BSDF

6.2 BRDF 前置知识· 数学篇

在正式了解 BRDF 的概念之前，有必要先了解数学和辐射度量学相关的前置基础知识。

6.2.1 球面坐标 Spherical Coordinate

由于光线主要是通过方向来表达，通常用球面坐标表达它们比用笛卡尔坐标系更方便。

如图，球面坐标中的向量用三个元素来指定：

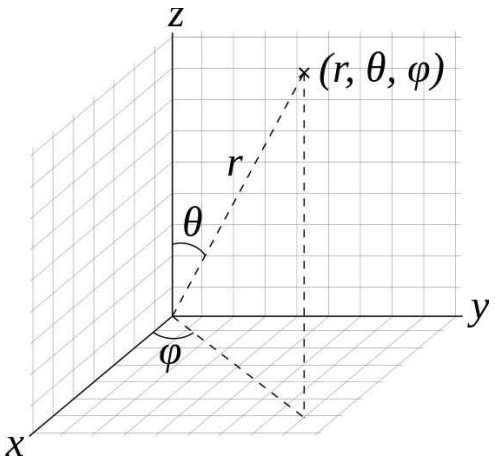


图 2 球面坐标

其中：

- r 表示向量的长度
- θ 表示向量和 Z 轴的夹角
- Φ表示向量在 x–y 平面上的投影和 x 轴的逆时针夹角。

6.2.2 立体角 Solid Angle

立体角描述了从原点向一个球面区域张成的视野大小，可以看成是弧度的三维扩展。

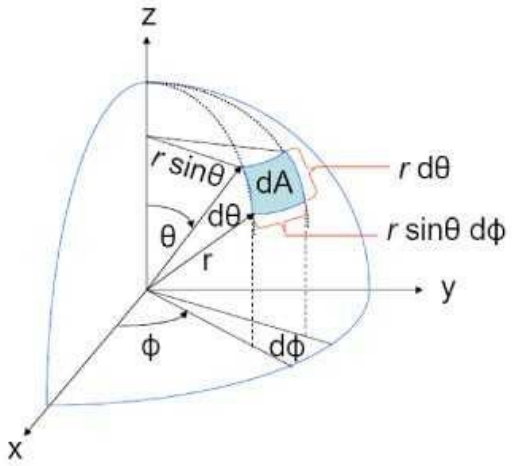


图 3 立体角

我们知道弧度是度量二维角度的量，等于角度在单位圆上对应的弧长，单位圆的周长是 2π，所以整个圆对应的弧度也是 2π 。立体角则是度量三维角度的量，用符号Ω表示，单位为立体弧度（也叫球面度，Steradian，简写为 sr），等于立体角在单位球上对应的区域的面积（实际上也就是在任意半径的球上的面积除以半径的平方 ω = s/r2 ），单位球的表面积是 4π ，所以整个球面的立体角也是 4π 。

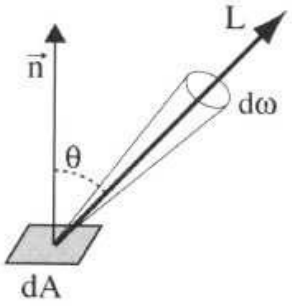


图 4 立体角

立体角 ω 有如下微分形式：

$$d\omega = \frac{dA}{r^2}$$

其中 *dA* 为面积微元。而面积微元 *dA* 在球面坐标系下可以写成：

$$dA = (rd\theta)(rsin\theta d\varphi) = r^2 sin\theta d\theta d\varphi$$

因此：

$$d\omega = \frac{dA}{r^2} = sin\theta d\theta d\varphi$$

6.2.3 投影面积 Foreshortened Area

投影面积描述了一个物体表面的微小区域在某个视线方向上的可见面积。

对于面积微元 A，则沿着与法向夹角为 θ 方向的 A 的可见面积为：

$$Area = Acos\theta$$

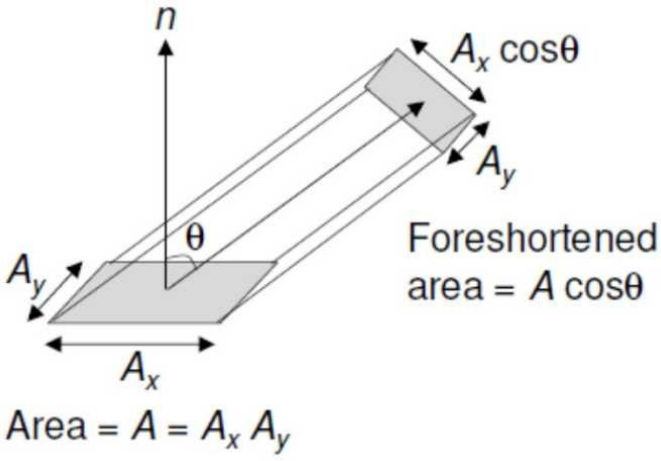


图 5 投影面积

6.3 BRDF 前置知识· 辐射度量学篇

6.3.1 辐射度量学基本参数表格

如下是截取的 wiki（https://en.wikipedia.org/wiki/Radiometry）上的辐射度量学国际单位制的辐射量参数表格：

物理量	符号	国际单位制	单位符号	注释
辐射出射度(Radiant exitance)	M_e	瓦特每平方米	$W \cdot m^{-2}$	表面出射的辐射通量
辐射度(Radiosity)	J_e or $J_{e\lambda}$	瓦特每平方米	$W \cdot m^{-2}$	表面出射及反射的辐射通量总和
辐射率(Radiance)	L_e	瓦特每球面度每平方米	$W \cdot sr^{-1} \cdot m^{-2}$	每单位立体角每单位投影面积的辐射通量
辐射能(Radiant energy)	Q_e	焦耳	J	能量
辐射强度(Radiant intensity)	I_e	瓦特每球面度	$W \cdot sr^{-1}$	每单位立体角的辐射通量
辐射通量(Radiant flux)	Φ_e	瓦特	W	每单位时间的辐射能量，亦作“辐射功率”
辐照度(Irradiance)	E_e	瓦特每平方米	$W \cdot m^{-2}$	入射表面的辐射通量

下面对几个核心的基本量进行分别介绍。

6.3.2 辐射通量/光通量 Radiant Flux

辐射通量（Radiant Flux，又译作光通量，辐射功率）描述的是在单位时间穿过截面的光能，或每单位时间的辐射能量，通常用Φ来表示，单位是 W，瓦特。

$$\Phi = \frac{dQ}{dt}$$

其中的 Q 表示辐射能(Radiant energy)，单位是 J，焦耳。

6.3.3 辐射强度/发光强度 Radiant Intensity

对一个点（比如说点光源）来说，辐射强度表示每单位立体角的辐射通量，用符号 I 表示，单位 $W \cdot sr^{-1}$ ：

$$I = \frac{d\Phi}{d\omega}$$

概括一下：辐射强度(Radiant intensity，又译作发光强度)，表示每单位立体角的辐射通量，通常用符号 I 表示，单位 $W \cdot sr^{-1}$ ，瓦特每球面度。

6.3.4 辐射率/光亮度 Radiance

辐射率（Radiance，又译作光亮度，用符号 L 表示）,表示物体表面沿某一方向的明亮程度，它等于每单位投影面积和单位立体角上的辐射通量，单位是 $W \cdot sr^{-1} \cdot m^{-2}$ ，瓦特每球面度每平方米。在光学中，光源的辐射率，是描述非点光源时光源单位面积强度的物理量，定义为在指定方向上的单位立体角和垂直此方向的单位面积上的辐射通量。光亮度 L 也可以理解为发光程度 I 在表面 dA 上的积分。

一种直观的辐射率的理解方法是：将辐射率理解为物体表面的微面元所接收的来自于某方向光源的单位面积的光通量，因此截面选用垂直于该方向的截面，其面积按阴影面积技术计算。

辐射率的微分形式：

$$L = \frac{d^2\Phi}{dAcos\theta d\omega}$$

其中：Φ是辐射通量，单位瓦特（W）；Ω是立体角，单位球面度（sr）。

另外需要注意的是，辐射率使用物体表面沿目标方向上的投影面积，而不是面积。

概括一下：辐射率（Radiance，又译作光亮度），表示每单位立体角每单位投影面积的辐射通量，通常用符号 L 表示,单位是 $W \cdot sr^{-1} \cdot m^{-2}$ ，瓦特每球面度每平方米。

6.3.5 辐照度/辉度 Irradiance

辐照度（Irradiance，又译作辉度，辐射照度，用符号 E 表示），指入射表面的辐射通量，即单位时间内到达单位面积的辐射通量，或到达单位面积的辐射通量，也就是辐射通量对于面积的密度，

用符号 E 表示，单位W/m^2，瓦特每平方米。

辐照度可以写成辐射率（Radiance）在入射光所形成的半球上的积分：

$$\frac{d\Phi}{dA} = E = \int_{\Omega} L(\omega) cos\theta d\omega$$

其中，Ω是入射光所形成的半球。L(ω)是沿ω方向的光亮度。

概括一下：辐照度（Irradiance，又译作辉度，辐射照度），表示单位时间内到达单位面积的辐射通量，也就是辐射通量对于面积的密度，通常用符号 E 表示，单位W/m^2，瓦特每平方米。

6.4 BRDF 的定义与理解

6.4.1 BRDF 的定义式

可以将给一个表面着色的过程，理解为给定入射的光线数量和方向，计算出指定方向的出射光亮度（radiance）。，在计算机图形学领域，BRDF （Bidirectional Reflectance Distribution Function，译作双向反射分布函数)是一个用来描述表面如何反射光线的方程。顾名思义，BRDF 就是一个描述光如何从给定的两个方向（入射光方向 l 和出射方向 v）在表面进行反射的函数。

BRDF 的精确定义是出射辐射率的微分（differential outgoing radiance）和入射辐照度的微分（differential incoming irradiance）之比：

$$f(l,v) = \frac{dL_o(v)}{dE(l)}$$

要理解这个方程的含义，可以想象一个表面被一个来自围绕着角度 l 的微立体角的入射光照亮，而这个光照效果由表面的辉度 dE 来决定。

表面会反射此入射光到很多不同的方向，在给定的任意出射方向 v，光亮度 dLo 与辐照度 dE 成一个比例。而两者之间的这个取决于 l 和 v 的比例，就是 BRDF。

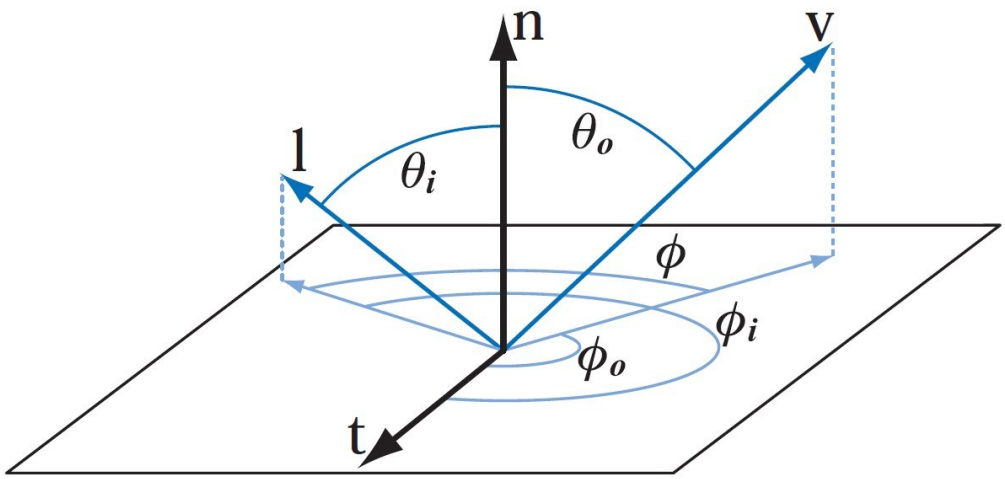


图 6 BRDF 图示

一个最常见的疑问是，BRDF 为什么要取这样的定义。BRDF 为什么被定义为辐射率（radiance）和辐照度（irradiance）之比，而不是 radiance 和 radiance 之比，或者 irradiance 和 irradiance 之比呢?

首先，我们分别重温它们的定义：

- 辐照度（Irradiance，又译作辉度，辐射照度），表示单位时间内到达单位面积的辐射通量，也就是辐射通量对于面积的密度，通常用符号 E 表示，单位W/m^2，瓦特每平方米。
- 辐射率（Radiance，又译作光亮度），表示每单位立体角每单位投影面积的辐射通量，通常用符号 L 表示,单位是 $W \cdot sr^{-1} \cdot m^{-2}$ ，瓦特每球面度每平方米。

那么，关于这个问题，我们可以这样理解：因为照射到入射点的不同方向的光，都可能从指定的反射方向出射，所以当考虑入射时，需要对面积进行积分。而辐照度 irradiance 正好表示单位时间内到达单位面积的辐射通量。所以 BRDF 函数，选取入射时的辐照度 Irradiance，和出射时的辐射率 Radiance，可以简单明了地描述出入射光线经过某个表面反射后如何在各个出射方向上分布。而直观来说，BRDF 的值给定了入射方向和出射方向能量的相对量。

概括一下：BRDF（Bidirectional Reflectance Distribution Function，译作双向反射分布函数 ），定义为出射辐射率的微分（differential outgoing radiance）和入射辐照度的微分（differential incoming irradiance）之比，描述了入射光线经过某个表面反射后如何在各个出射方向上分布，给定了入射方向和出射方向能量的相对量，单位是sr^{-1}，每球面度。

6.4.2 BRDF 的非微分形式

这里的讨论仅限于非区域光源，如点光源或方向光源。在这种情况下，BRDF 定义可以用非微分形式表示：

$$f(l,v)=\frac{L_o(v)}{E(l)\overline{cos}\theta_i}$$

其中：

- EL 是光源在垂直于光的方向向量 L 平面测量的辐照度（irradiance）。
- Lo（v）是在视图矢量 v 的方向上产生的出射辐射率（radiance）。

6.4.3 BRDF 与着色方程

根据上文所了解了 BRDF 的定义，现在，就很容易得到 BRDF 是如何用 n 个非区域光来拟合一般的着色方程的：

$$L_o(v)=\sum_{k=1}^{n}f(l_k,v)\otimes E_{L_k}cos\theta_{i_k}$$

其中 k 是每个光源的索引。使用⊗符号（分段向量乘法），是因为 BRDF 和辉度（irradiance）都是 RGB 向量。考虑到入射和出射方向都拥有两个自由度（通常参数化是使用两个角度：相对于表面法线的仰角 θ 和关于法线的旋转角度 ϕ），一般情况下，BRDF 是拥有四个标量变量的函数。

另外，各向同性 BRDFs（Isotropic BRDFs）是一个重要的特殊情况。这样的 BRDF 在输入和输入方向围绕表面法线变化（保持相同的相对夹角）时保持不变。所以，各向同性 BRDF 是关于三个标量的函数。

6.4.4 对 BRDF 的可视化表示

一种理解 BRDF 的方法就是在输入方向保持恒定的情况下对它进行可视化表示，如下图。对于给定方向的入射光来说，图中显示了出射光的能力分布：在交点附近球形部分是漫反射分量，因此出射光来任何方向上的反射概率相等。椭圆部分是一个反射波瓣（Reflectance

Lobe）。它形成了镜面分量。显然，这些波瓣位于入射光的反射方向上，波瓣厚度对应反射的模糊性。根据互易原理，可以将这些相同的可视化形成认为是每个不同入射光方向对单个出射方向的贡献量大小。

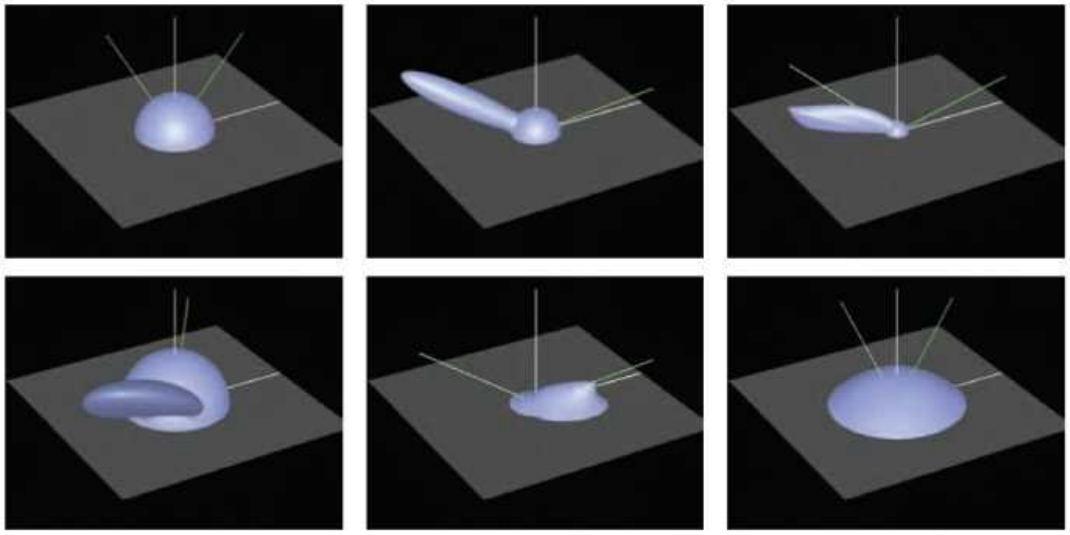


图 7 BRDF 的可视化表示

6.5 BRDF 的性质

6.5.1 可逆性

BRDF 的可逆性源自于亥姆霍兹光路可逆性（Helmholtz Recoprpcity Rule）。

BRDF 的可逆性即，交换入射光与反射光，并不会改变 BRDF 的值：

$$f(l,v) = f(v,l)$$

6.5.2 能量守恒性质

BRDF 需要遵循能量守恒定律。能量守恒定律指出：入射光的能量与出射光能量总能量应该相等。能量守恒方程如下：

$$Q_{incoming} = Q_{reflected} + Q_{absorb} + Q_{transmitted}$$

由此可知：

$$Q_{reflected} \leq Q_{incoming}$$

因此 BRDF 必须满足如下的积分不等式，也就是能量守恒性质：

$$\forall l, \int_{\Omega} f_{\mathrm{r}}(l, v)\, cosvdv \leq 1$$

6.5.3 线性特征

很多时候，材质往往需要多重 BRDF 计算以实现其反射特性。表面上某一点的全部反射辐射度可以简单地表示为各 BRDF 反射辐射度之和。例如，镜面漫反射即可通过多重 BRDF 计算加以实现。

6.6 BRDF 的模型分类

根据 BRDF 的定义来直接应用，会有一些无从下手的感觉。而为了方便和高效地使用 BRDF 数据，大家往往将 BRDF 组织成为各种参数化的数值模型。

有各式各样的 BRDF 模型，如：

- Phong (1975)
- Blinn–Phong (1977)
- Cook–Torrance (1981)
- Ward (1992)
- Oren–Nayar (1994)
- Schlick (1994)
- Modified–Phong (Lafortune 1994)

- Lafortune (1997)
- Neumann–Neumann (1999)
- Albedo pump–up (Neumann–Neumann 1999)
- Ashikhmin–Shirley (2000)
- Kelemen (2001)
- Halfway Vector Disk (Edwards 2006)
- GGX (Walter 2007)
- Distribution–based BRDF (Ashikmin 2007)
- Kurt (2010)
- etc.

这些 BRDF 的模型可以分为如下几类：

- 经验模型（Empirical Models）：使用基于实验提出的公式对 BRDF 做快速估计。
- 数据驱动的模型（Data–driven Models）：采集真实材质表面在不同光照角度和观察角将 BRDF 按照实测数据建立查找表，记录在数据库中，以便于快速的查找和计算。
- 基于物理的模型（Physical–based Models）：根据物体表面材料的几何以及光学属性建立反射方程，从而计算 BRDF，实现极具真实感的渲染效果。

6.6.1 BRDF 经验模型

关于 BRDF 的经验模型，有如下几个要点：

- 经验模型提供简洁的公式以便于反射光线的快速计算。
- 经验模型不考虑材质特性，仅仅提供一个反射光的粗糙近似。
- 经验模型不一定满足物理定律，比如 Helmholtz 可逆性或能量守恒定律。
- 经验模型因为其简洁和高效性被广泛运用。

常见的 BRDF 经验模型有：

- Lambert 漫反射模型
- Phong 模型
- Blinn–Phong 模型

- 快速 Phong 模型
- 可逆 Phong 模型

Lambert 模型，Phong 模型、Blinn-Phong 模型和其改进模型都是常见的光照模型，由于篇幅原因，就不展开论述了。

6.6.2 数据驱动的 BRDF 模型

数据驱动的 BRDF 模型可以理解为，度量一个大的 BRDF 材质集合，并将其记录为高维向量，利用降维的方法从这些数据中计算出一个低维模型，这样基于查表的方式，可以直接找到渲染结果，省去大量的实时计算。代表工作如：A Data-Driven Reflectance Model,ACM SIGGRAPH,2003 http://people.csail.mit.edu/wojciech/DDRM/index.html

另外，MERL 等实验室使用各类仪器测量了上多种真实材质表面在不同光照角度和观察角度下的反射数据，并记录在数据库中，如 MERL BRDF Database。

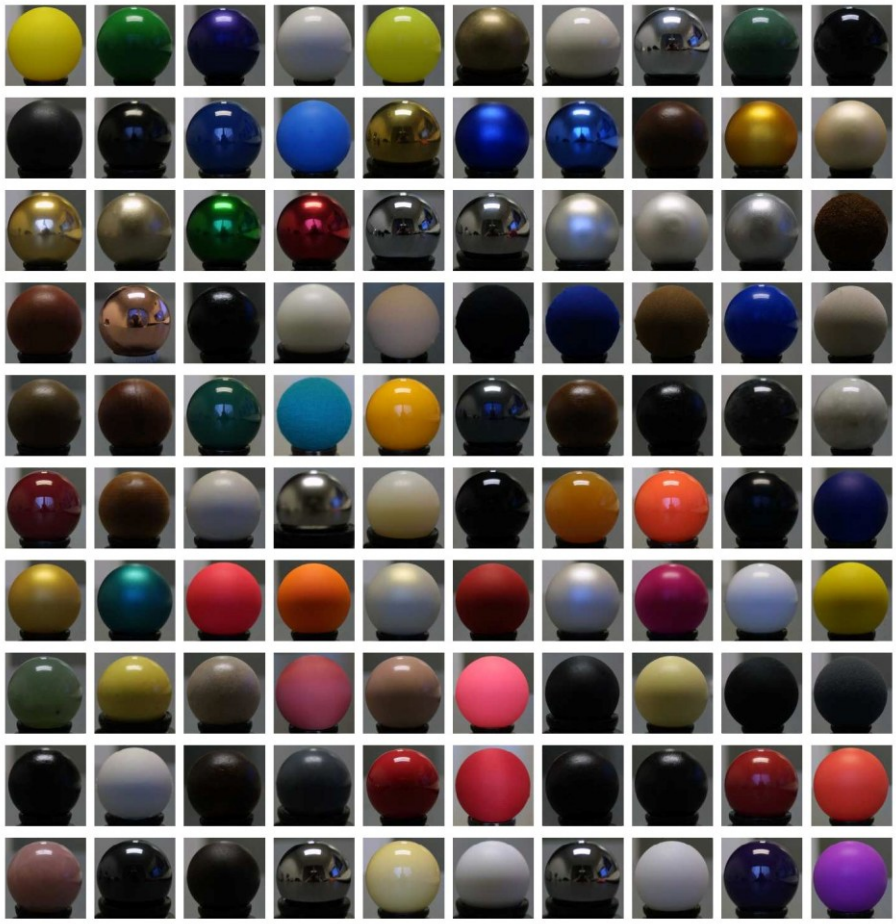


图 8 一个名为“MERL 100”的 BRDF 数据库。其中，含 50 种“光滑材质”（例如金属和塑料），和 50 种“粗糙材质”（例如纺织物）

需要注意的是，由于这些数据由于采集自真实材质，即便渲染出来的结果很真实，但缺点是没有可供调整效果的参数，无法基于这些数据修改成想要的效果，另外部分极端角度由于仪器限制，无法获取到数据，而且采样点密集，数据量非常庞大，所以并不适合游戏等实时领域，一般可用在电影等离线渲染领域，也可以用来做图形学研究，衡量其他模型的真实程度。

这边提供一些数据库的链接供参考：

- MERL BRDF Database： http://www.merl.com/brdf/
- MIT CSAIL: http://people.csail.mit.edu/addy/research/brdf/
- CAVE Database : http://www1.cs.columbia.edu/CAVE/databases/tvbrdf/about.php

6.6.3 基于物理的 BRDF 模型

基于物理的渲染(PBR, Physically-based rendering)是计算机图形学中用数学建模的方式模拟物体表面各种材质散射光线的属性从而渲染照片真实图片的技术，是近年来是实时渲染领域的大趋势。

基于物理的 BRDF 模型通过包含材质的各种几何及光学性质来尽可能精确的近似现实世界中的材料。而一个基于物理的 BRDF 要必须满足至少如下两条 BRDF 的特性：能量守恒、亥姆霍兹光路可逆性。

常见的基于物理的 BRDF 模型有：

- Cook-Torrance BRDF 模型
- Ward BRDF 模型

下文将先介绍基于物理的 BRDF 常常用到到的菲涅尔反射，次表面散射和微平面理论等理论，分别概括这两种基于物理的 BRDF 模型。

6.7 基于物理的 BRDF · 前置知识

6.7.1 次表面散射 Subsurface Scattering

在真实世界中许多物体都是半透明的，比如皮肤、玉、蜡、大理石、牛奶等。当光入射到透明或半透明材料表面时，一部分被反射，一部分被吸收，还有一部分经历透射。这些半透明

的材质受到数个光源的透射，物体本身就会受到材质的厚度影响而显示出不同的透光性，光线在这些透射部分也可以互相混合、干涉。

次表面散射，Subsurface Scattering，简称 SSS(又简称 3S)，就是光射入半透明材质后在内部发生散射，最后射出物体并进入视野中产生的现象，是指光从表面进入物体经过内部散射，然后又通过物体表面的其他顶点出射的光线传递过程。

图 9 次表面散射示例 1

图 10 次表面散射示例 2

又如文章开头，贴出的《超能陆战队》中大白的渲染照中，大白的白色身体，略微有些透明的感觉，就是典型的次表面散射。

简而言之：次表面散射，即光射入表面，在材质里散射，然后从与射入点不同的地方射出表面的一种现象。

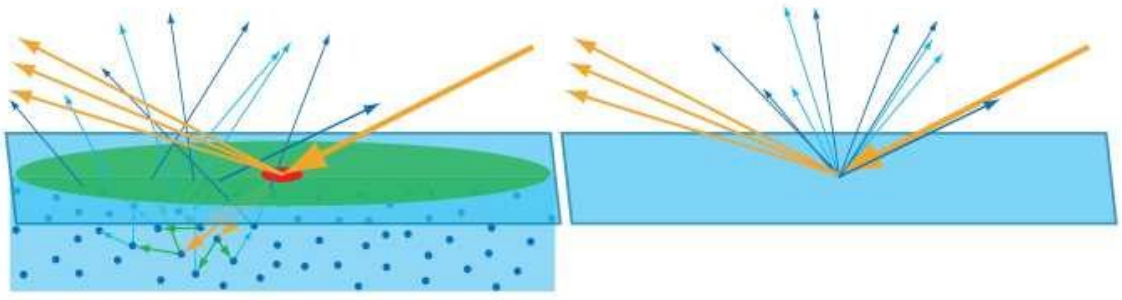


图 11 光与表面的相互作用。左图，可以看到次表面相互作用导致光从入口点重新射出。红色和绿色圆圈代表两个不同尺度下的像素所覆盖的区域。在右边，所有的次表面散射光都是从入口点发出的，忽略了表面之下的细节。

6.7.2 菲涅尔反射 Fresnel Reflectance

菲涅耳方程（Fresnel equations）是一组用于描述光在两种不同折射率的介质中传播时的反射和折射的光学方程。方程中所描述的反射被称作“菲涅耳反射”。

菲涅尔反射（Fresnel Reflectance）或者菲涅尔效果（Fresnel Effect），即当光入射到折射率不同的两个材质的分界面时，一部分光会被反射，而我们所看到的光线会根据我们的观察角度以不同强度反射的现象。

菲涅尔反射能够真实地模拟真实世界中的反射。在真实世界中，除了金属之外，其它物质均有不同程度的菲涅尔反射效果。

关于菲涅尔反射，一个很好的例子是一池清水。从水池上笔直看下去（也就是与法线成零度角的方向）的话，我们能够一直看到池底。而如果从接近平行于水面的方向看去的话，水池表面的高光反射会变得非常强以至于你看不到池底。

图 12 菲涅尔反射效果

简单来说，视线垂直于表面时，反射较弱，而当视线并非垂直表面时，夹角越小，反射越明显。

对于粗糙表面来说，在接近平行方向的高光反射也会增强但不够达到 100%的强度.为何如此是因为影响菲涅尔效应的关键参数在于每个微平面的法向量和入射光线的角度，而不是宏观平面的法向量和入射光线的角度。因此我们在宏观层面看到的实际上是微平面的菲涅尔效应的一个平均结果。

根据菲涅尔反射，若你看向一个圆球，那么圆球中心的反射会较弱，而靠近边缘是反射会较强。另外需注意，这种关系也受折射率影响。

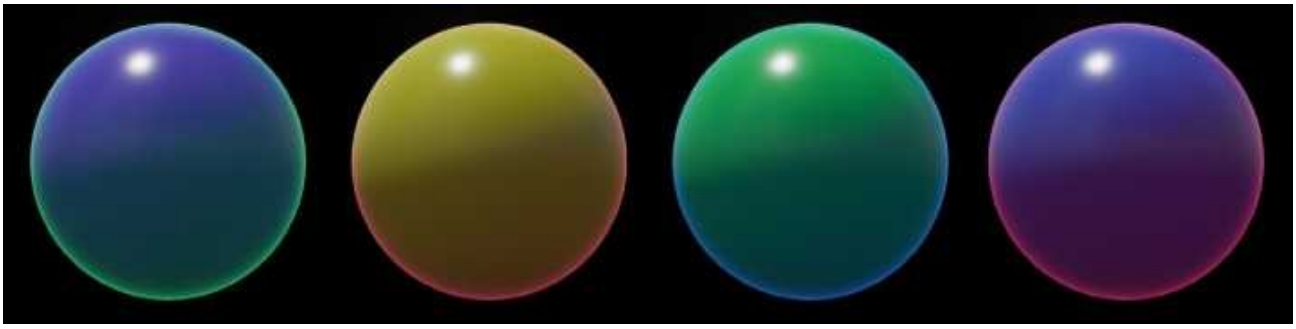


图 13 基于菲涅尔反射，在 Unreal Engine 4 中实现的边缘光 Shader

6.7.3 微平面理论 Microfacet Theory

微表面理论假设表面是由不同方向的微小细节表面，也就是微平面（microfacets）组成。每一个微小的平面都会根据它的法线方向在一个方向上反射光线。

表面法线朝向光源方向和视线方向中间的微表面会反射可见光。然而，不是所有的表面法线和半角法线（half normal）相等的微表面都会反射光线，因为其中有些会被遮挡，如下图所示。

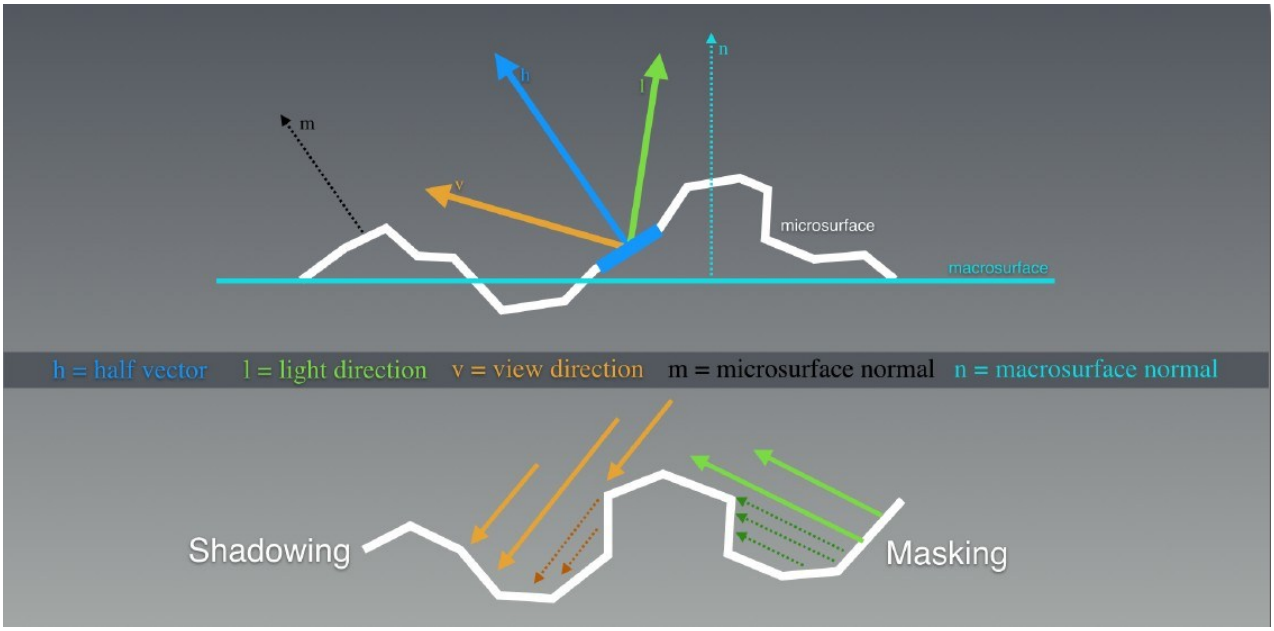


图 14 微平面理论图示

我们用法线分布函数（Normal Distribution Function，简写为 NDF），D(h)来描述组成表面一点的所有微表面的法线分布概率。则可以这样理解：向 NDF 输入一个朝向 h，NDF 会返回朝向是 h 的微表面数占微表面总数的比例，比如有 8%的微表面朝向是 h，那么就有 8%的微表面可能将光线反射到 v 方向。

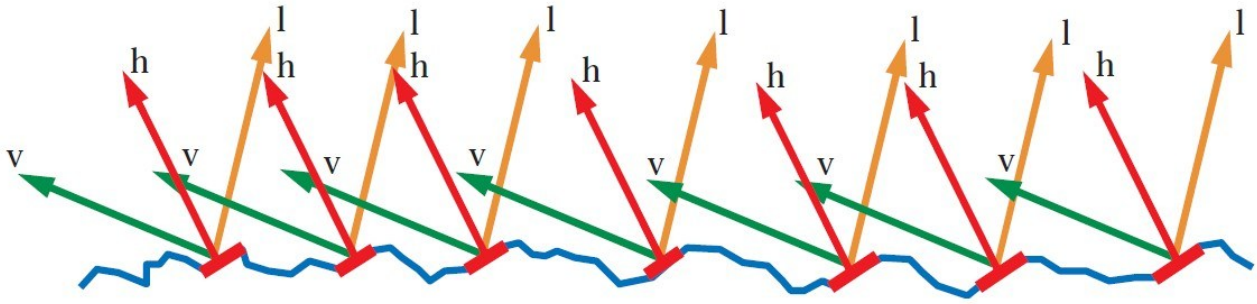


图 15 由微平面组成的表面。仅红色微平面的表面法线和半矢量 h 对齐，能参与从入射光线向量 l 到视线向量 v 的光线反射

NDF 的定义式：

$$D(h) = \frac{\alpha^2}{\pi((n \cdot h)^2(\alpha^2 - 1) + 1)^2}$$

在微观层面上不规则的表面会造成光的漫反射。例如，模糊的反射是由于光线的散射造成的。而反射的光线并不均匀，因此我们得到的高光反射是模糊的。如下图。

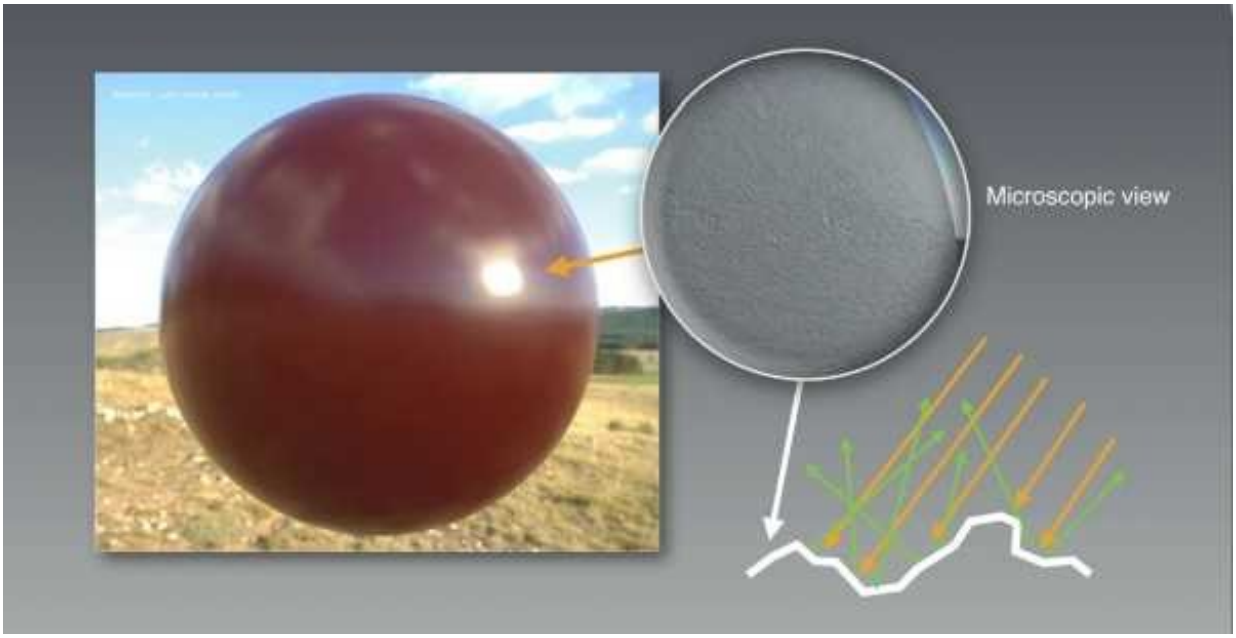


图 16 模糊的微平面高光反射

6.8 基于物理的 BRDF · 常见模型

6.8.1 Cook–Torrance BRDF 模型

Cook–Torrance 模型作为图形学中最早的基于物理的 BRDF 模型，由 Cook 和 Torrance 提出，是 Torrance–Sparrow 模型的一个应用版本。现今，Cook–Torrance 模型已经成为基于物理着色的标准模型之一。Cook–Torrance 模型将物理学中的菲涅尔反射引入了图形学，实现了比较逼真的效果。

Cook–Torrance 微平面着色模型（Cook–Torrance microfacet specular shading model），即 Microfacet Specular BRDF，定义为：

$$f(l,v) = \frac{F(l,h)G(l,v,h)D(h)}{4(n\cdot h)(n\cdot v)}$$

其中：

- F 为菲涅尔反射函数(Fresnel 函数)
- G 为阴影遮罩函数（ Geometry Factor，几何因子 ），即未被 shadow 或 mask 的比例
- D 为法线分布函数(NDF)

篇幅原因，就不展开论述了，若大家有兴趣，可以参考历年的 SIGGRAPH Course: Physically Based Shading in Theory and Practice 系列与其他相关文章。

如 SIGGRAPH 2013 Course: Physically Based Shading in Theory and Practice:

http://blog.selfshadow.com/publications/s2013-shading-course/

具体的课程中的两篇推荐：

1. Background:Physics and Math of Shading: http://blog.selfshadow.com/publications/s2013-shading-course/hoffman/s2013_pbs_physics_math_slides.pdf
2. Physically Based Lighting in Call of Duty: Black Ops ： http://blog.selfshadow.com/publications/s2013-shading-course/lazarov/s2013_pbs_black_ops_2_slides_v2.pdf

6.8.2 Ward BRDF 模型

一般情况下，我们可以将 BRDF 分为两类：

各项同性（ Isotropic ）的 BRDF

- 反射不受与给定表面法向夹角的约束
- 随机表面微结构

各向异性（ Anisotropic ）的 BRDF

- 反射比随着与某个给定的表面法向之间的夹角而变化
- 图案的表面微结构
- 金属丝，绸缎，毛发等

Phong 和 Cook-Torrance BRDF 模型都不能处理各项异性的效果，Ward 模型却可以。

Ward 模型由 Ward 于 1992 年提出[Measuringand Modeling Anissropic Reflection,ACM SIGGRAPH,1992]。Ward 模型介绍了更一般的表面法向表达方式：通过椭圆体（ellipsoids）这种允许各向异性反射的形式来表达。

然而，由于没有考虑菲涅耳因子（Fresnel factor）和几何衰减因子（geometric attenuation factor），该模型更像是一种经验模型，但还是属于基于物理的 BRDF 模型。

各向同性的 Ward 模型定义为：

$$\rho_{bd\ \ iso}(\theta_i,\phi_i,\theta_r,\phi_r)=\frac{\rho_d}{\pi}+\rho_s\cdot\frac{1}{\sqrt{\cos\theta_i\cos\theta_r}}\cdot\frac{\exp[-\tan^2\delta/\alpha^2]}{4\pi\alpha^2}$$

6.9 BRDF 与其引申

有不少与 BRDF 类似的函数：

- BSSRDF：Bidirectional Surface Scattering Reflectance Distribution，双向表面散反射分布函数
- SBRDF(SVBRDF): spatially varying BRDF(spatial BRDF) 空间 BRDF
- BTDF：Bidirectional Transmittance Distribution Function 双向透射分布函数
- BSDF : Bidirectional Scattering Distribution Function，双向散射分布函数

下面分别进行介绍。

6.9.1 BSSRDF

BRDF 只是更一般方程的一种近似,这个方程就是 BSSRDF（Bidirectional scattering-surface reflectance distribution function，双向表面散反射分布函数）。BSSRDF 描述了射出辐射率与入射通量之间的关系，BSSRDF 函数通过把入射和出射位置作为函数的输入，从而来包含这些现象，它描述了沿入射方向从物体表面的一点到另外一点，最后顺着出射方向出去的光线的相对量。注意，这个函数还考虑了物体表面的一点到另外一点，最好顺着出射方向出去的光线相对量。注意，这个函数还考虑了物体表面不一致的情况，因为随着位置的变化，反射系数也会发生变化。在实时绘制中，物体表面上的位置可以用来获取颜色纹理、光泽度，以及凹凸纹理图等信息。

图 17 BRDF 渲染图

图 18 BSSRDF 渲染图

6.9.2 SBRDF(SVBRDF)

一个捕获基于空间位置 BRDF 变化的函数被称为空间变化的 BRDF（Spatially Varying BRDF ,SVBRDF）或称空间 BRDF，空间双向反射分布函数（Spatial BRDF ，SBRDF）。

6.9.3 BTDF 与 BSDF

即使一般的 BSSRDF 函数，无论其多么复杂，仍然忽略了现实世界中非常重要的一些变量，比如说光的偏振。此外，也没有处理穿过物体表面的光线传播，只是对反射情况进行了处理。为了处理光线传播的问题，对物体表面定义了两个 BRDF 和两个 BTDF（T 表示传播“Transmittance”），每侧各有一个，这样就组成了 BSDF（S 表示散射“Scattering”）。

图 19 当用于镜面反射的 BRDF 和用于镜面透射的 BTDF 使用菲涅尔公式进行调制渲染时，得到了如真实玻璃视觉上精确的反射和透射的角度变化。来自《Physically Based Rendering, Third Edition》Figure 8.10

而在实践中，这些更复杂的函数很少使用，BRDF 和 SVBRDF 足以胜任一般情况下表面渲染的效果。

6.10 Reference

[1] http://graphics.stanford.edu/~henrik/images/subsurf.html

[2] http://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/reflection-refraction-fresnel

[3] http://wiki.nuaj.net/index.php?title=BRDF

[4] https://www.allegorithmic.com/pbr-guide

[5] http://www.icourses.cn/coursestatic/course_2987.html

[6] https://docs.unrealengine.com/latest/INT/Engine/Rendering/Materials/HowTo/Fresnel/index.html

[7] https://en.wikipedia.org/wiki/Fresnel_equations

[8] https://zhuanlan.zhihu.com/p/21376124

[9] https://en.wikipedia.org/wiki/Radiometry

[10] Real Shading in Unreal Engine 4：
https://cdn2.unrealengine.com/Resources/files/2013SiggraphPresentationsNotes-26915738.pdf

[11] SIGGRAPH 2013 Course: Physically BasedShading in Theory and Practice：
http://blog.selfshadow.com/publications/s2013-shading-course/

[12] Physically Based Rendering, Third Edition

第七章 延迟渲染的前生今世

题图为基于 Deferred Rendering 技术的渲染效果图。

在计算机图形学中，延迟渲染(Deferred Rendering) ，即延迟着色（ Deferred Shading），是将着色计算延迟到深度测试之后进行处理的一种渲染方法。延迟着色技术的最大的优势就是将光源的数目和场景中物体的数目在复杂度层面上完全分开，能够在渲染拥有成百上千光源的场景的同时依然保持很高的帧率，给我们渲染拥有大量光源的场景提供了很多可能性。

图 1 使用 Deferred Rendering 方法渲染的多光源场景

在《Real-Time Rendering 3rd》（实时渲染图形学第三版）的第七章“Advanced Shading · 高级着色”中，除了上篇文章中我们聊到的 BRDF，还有 Deferred Shading（延迟着色）这个重要概念我们没有聊到。这篇文章，就主要和大家一起聊一聊 Deferred Shading 和它的“前生今世”，以及文末简单提一提第八章“区域和环境光照 Area and Environmental Lighting”中的 Environment Mapping（环境映射）相关的内容。下篇文章，预计直接开始全局光照（Global Illumination）的内容。

简而言之，通过阅读这篇文章，你将对以下要点有所了解：

- 延迟着色/延迟渲染的概念 Deferred Shading / Deferred Rendering
- 几何缓冲区 G-buffer
- 延迟渲染的渲染过程
- 延迟渲染 vs 正向渲染
- 延迟渲染的优缺点
- 延迟光照 Light Pre-Pass / Deferred Lighting
- 分块延迟渲染 Tile-Based Deferred Rendering
- 延迟渲染 vs 延迟光照
- 实时渲染中常见的 Rendering Path 总结

- 环境映射 Environment Mapping

7.1 延迟渲染 Deferred Rendering

延迟渲染(Deferred Rendering)，即延迟着色（ Deferred Shading ），顾名思义，是将着色计算延后进行处理的一种渲染方法，在 2004 年的 GDC 上被正式提出 http://www.tenacioussoftware.com/gdc_2004_deferred_shading.ppt。

我们知道，正向渲染(Forward Rendering)，或称正向着色(Forward Shading)，是渲染物体的一种非常直接的方式，在场景中我们根据所有光源照亮一个物体，之后再渲染下一个物体，以此类推。

传统的正向渲染思路是，先进行着色，再进行深度测试。其的主要缺点就是光照计算跟场景复杂度和光源个数有很大关系。假设有 n 个物体，m 个光源，且每个每个物体受所有光源的影响，那么复杂度就是 O(m*n)。

正向渲染简单直接，也很容易实现，但是同时它对程序性能的影响也很大，因为对每一个需要渲染的物体，程序都要对每个光源下每一个需要渲染的片段进行迭代，如果旧的片段完全被一些新的片段覆盖，最终无需显示出来，那么其着色计算花费的时间就完全浪费掉了。

而延迟渲染的提出，就是为了解决上述问题而诞生（尤其是在场景中存在大量光源的情况下）。延迟着色给我们优化拥有大量光源的场景提供了很多可能性，因为它能够在渲染拥有成百上千光源的场景的同时还能够保持能让人接受的帧率。下面这张图展示了一个基于延迟着色渲染出的场景，这个场景中包含了 1000 个点光源，对于目前的硬件设备而言，用传统的正向渲染来实现几乎是不可能的。

图 2 基于 Deferred Rendering 渲染的含 1000 个点光源的场景 [J. Andersson, SIGGRAPH 2009 Beyond Programmable shading course talk] @ Frostbite 2 引擎

可以将延迟渲染(Deferred Rendering)理解为先将所有物体都先绘制到屏幕空间的缓冲（即 G-buffer，Geometric Buffer，几何缓冲区）中，再逐光源对该缓冲进行着色的过程，从而避免了因计算被深度测试丢弃的片元的着色而产生的不必要的开销。也就是说延迟渲染基本思想是，先执行深度测试，再进行着色计算，将本来在物空 间（三维空间）进行光照计算放到了像空间（二维空间）进行处理。

对应于正向渲染 O(m*n)的 复杂度，经典的延迟渲染复杂度为 O(n+m)。

图 3 Unreal Engine 3 中实现的 Deferred Shading @GDC 2011

Jimmikaelkael 在 2016 年 12 月 24 日发了一条推文（https://twitter.com/jimmikaelkael/status/812631802242273280），分享了一组在 Unity3D 中基于 Deferred Shading 渲染的 SpeedTree 场景，非常逼真：

图 4 SpeedTree deferred shading with translucency @Unity3D 引擎 by jimmikaelkael

图 5 SpeedTree deferred shading with translucency @Unity3D 引擎 by jimmikaelkael

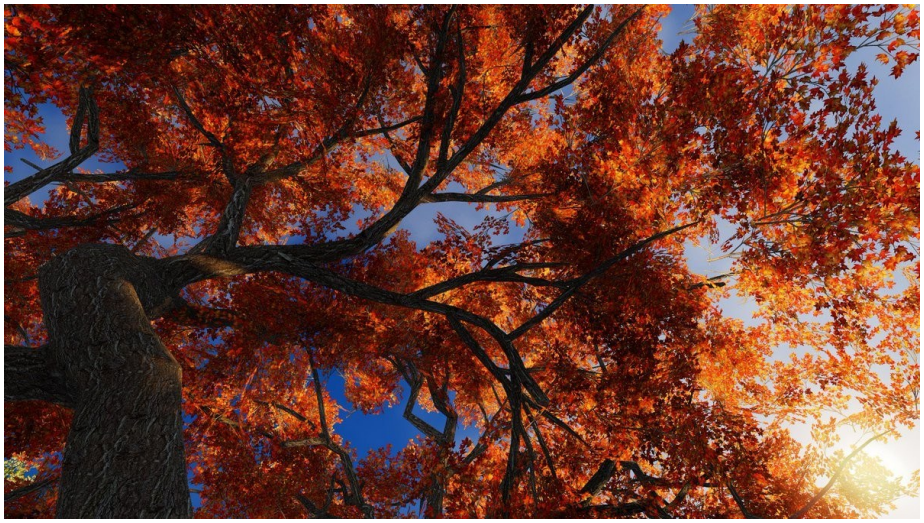


图 6 SpeedTree deferred shading with translucency @Unity3D 引擎 by jimmikaelkael

图 7 SpeedTree deferred shading with translucency @Unity3D 引擎 by jimmikaelkael

延迟着色中一个非常重要的概念就是 G-Buffer，下面先聊一下 G-Buffer。

7.2 几何缓冲区 G-buffer

G-Buffer，全称 Geometric Buffer ，译作几何缓冲区，它主要用于存储每个像素对应的位置（Position），法线（Normal），漫反射颜色（Diffuse Color）以及其他有用材质参数。根据这些信息，就可以在像空间（二维空间）中对每个像素进行光照处理。

图 8 一个典型的 G-buffer layout。Source: W. Engel, “Light-Prepass Renderer Mark III” @SIGGRAPH 2009Talks

下图是一帧中 G-buffer 中存储的内容：

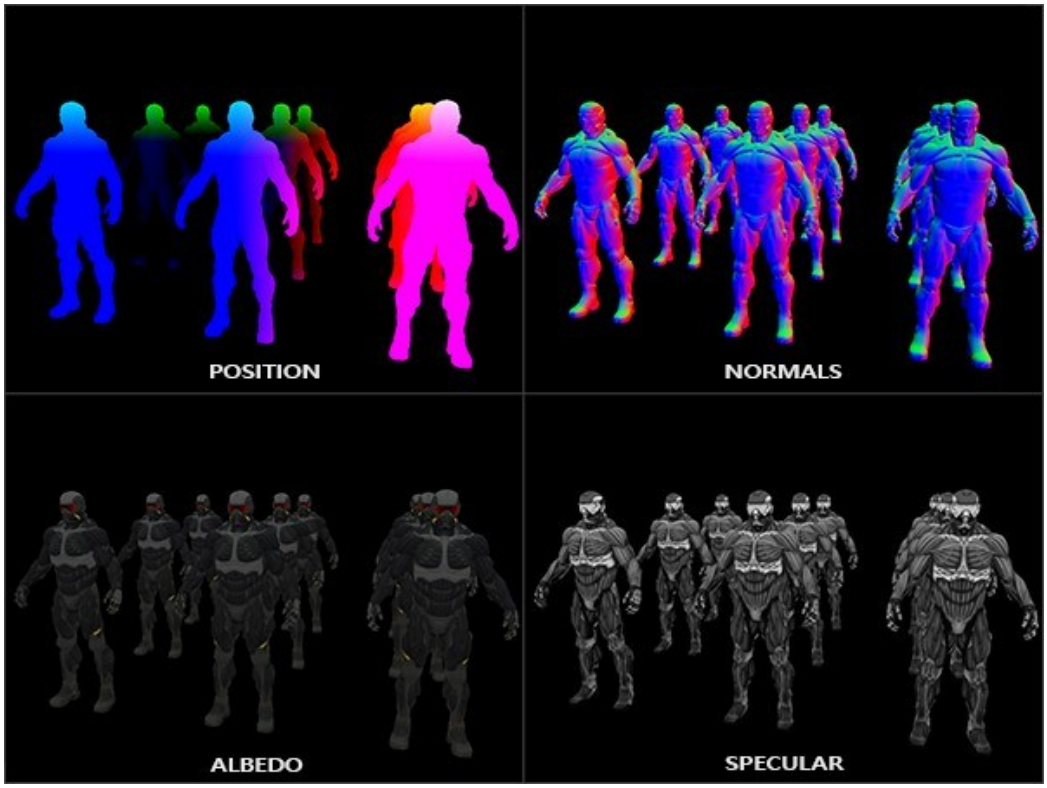


图 9 G-buffer 存储的信息

7.3 延迟渲染的过程分析

可以将延迟渲染理解为两个 Pass 的过程：

1、几何处理阶段(Geometry Pass)。这个阶段中，我们获取对象的各种几何信息，并将第二步所需的各种数据储存（也就是渲染）到多个 G-buffer 中；

2、光照处理阶段(Lighting Pass)。在这个 pass 中，我们只需渲染出一个屏幕大小的二维矩形，使用第一步在 G-buffer 中存储的数据对此矩阵的每一个片段计算场景的光照；光照计算的过程还是和正向渲染以前一样，只是现在我们需要从对应的 G-buffer 而不是顶点着色器(和一些 uniform 变量)那里获取输入变量了。

下面这幅图片很好地展示了延迟着色的整个过程：

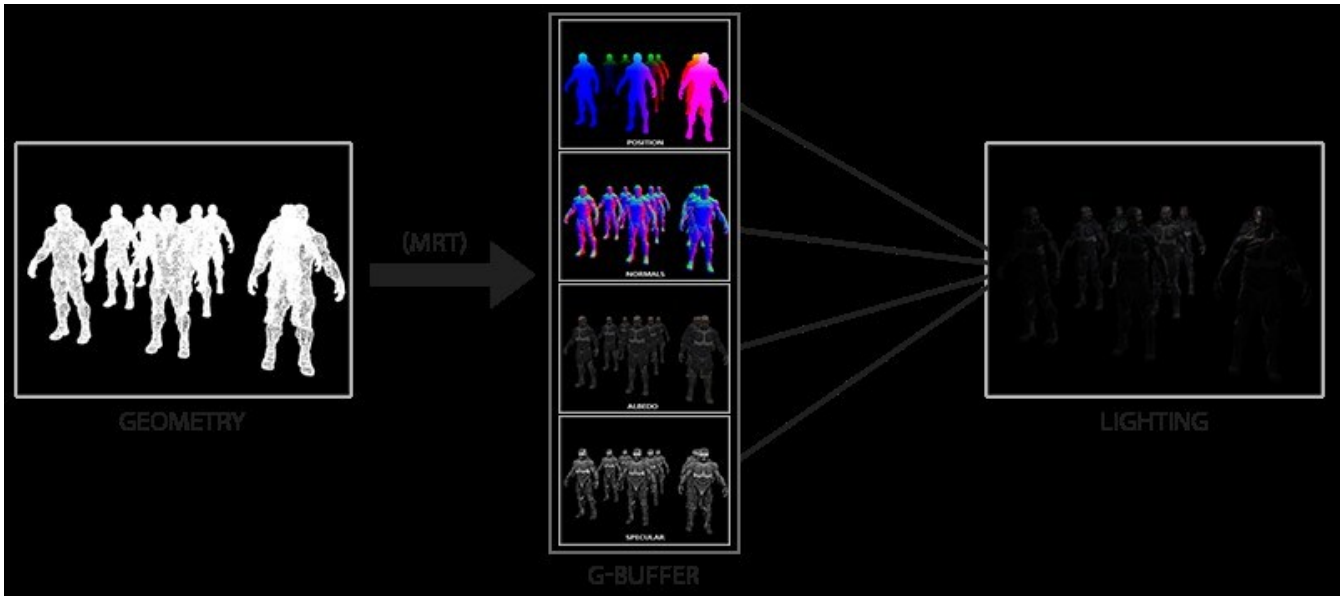


图 10 延迟着色的过程

延迟渲染方法一个很大的好处就是能保证在 G-buffer 中的片段和在屏幕上呈现的像素所包含的片段信息是一样的，因为深度测试已经最终将这里的片段信息作为最顶层的片段。这样保证了对于在光照处理阶段中处理的每一个像素都只处理一次，所以我们能够省下很多无用的渲染调用。除此之外，延迟渲染还允许我们做更多的优化，从而渲染更多的光源。

在几何处理阶段中填充 G-buffer 非常高效，因为我们直接储存位置，颜色，法线等对象信息到帧缓冲中，这个过程几乎不消耗处理时间。

而在此基础上使用多渲染目标(Multiple Render Targets, MRT)技术，我们可以在一个 Pass 之内完成所有渲染工作。

总结一下，典型的 Deferred Rendering 的渲染流程有两步：

1. 几何处理阶段：渲染所有的几何/颜色数据到 G-buffer

2. 光照处理阶段：使用 G-buffer 计算场景的光照。

7.4 延迟渲染的伪代码

为了便于理解，这里贴出一些各种描述版本的延迟渲染算法的伪代码：

1、通用版本的延迟着色算法伪代码：

```
For each object:
Render to multiple targets
For each light:
Apply light as a 2D postprocess
```

2、一个通用版本的 deferred shading 过程描述：

“Standard” deferred shading is a 2-stage process:

(1) draw (opaque) geometry storing its attributes (i.e. position as depth, normals, albedo color, specular color and other material properties) in a number of full screen buffers (typically 3 or 4)

(2) for each light source, draw its volume and accumulate lit surface color into final render target

3、两个 Pass 的延迟着色算法伪代码：

```
Two-pass deferred shading algorithm
Pass 1: geometry pass
- Write visible geometry information to G-buffer
Pass 2: shading pass
For each G-buffer sample, compute shading
- Read G-buffer data for current sample
- Accumulate contribution of all lights
- Output final surface color
```

4、多光源的延迟渲染的伪代码：

```
Many-light deferred shading algorithm
For each light:
- Generate/bind shadow/environment maps
- Compute light’s contribution for each G-buffer sample:
```

For each G-buffer sample
- Load G-buffer data
- Evaluate light contribution (may be zero)
- Accumulate contribution into frame-buffer

可以将这里的多光源计算过程理解为，对每个光源创建一个屏幕空间包围矩形，然后用光照 shader 渲染这个矩形。

7.5 延迟渲染 vs 正向渲染

这边对正向渲染和延迟渲染的特性做一个对照列举：

7.5.1 正向渲染

- 正向渲染（Forward Rendering），先执行着色计算，再执行深度测试。
- 正向渲染渲染 n 个物体在 m 个光源下的着色，复杂度为 O(n*m)次。
- Forward Rendering，光源数量对计算复杂度影响巨大，所以比较适合户外这种光源较少的场景。
- Forward Rendering 的核心伪代码可以表示为：

```
For each light:
    For each object affected by the light:
        framebuffer += object * light
```

Forward Rendering 的管线流程如下：

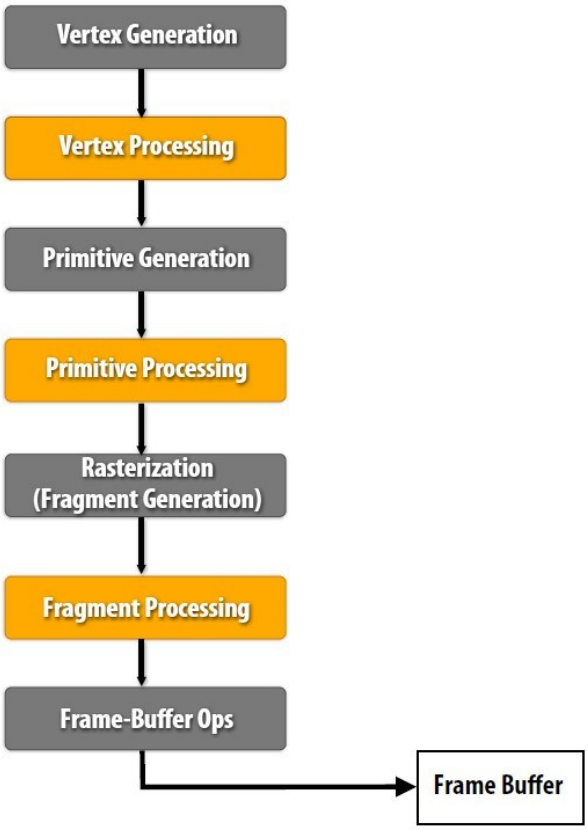


图 11 正向渲染（Forward Rendering）管线流程

7.5.2 延迟渲染

- 延迟渲染(Deferred Rendering)，先执行深度测试，再执行着色计算。
- 延迟渲染渲染 n 个物体在 m 个光源下的着色，复杂度为 O(n+m)次。
- Deferred Rendering 的最大的优势就是将光源的数目和场景中物体的数目在复杂度层面上完全分开。也就是说场景中不管是一个三角形还是一百万个三角形，最后的复杂度不会随 光源数目变化而产生巨大变化。
- Deferred Rendering 的核心伪代码可以表示如下，上文已经贴出过，这边再次贴出，方便对比：

  For each object: Render to multiple targets For each light: Apply light as a 2D postprocess

Deferred Rendering 的管线流程如下：

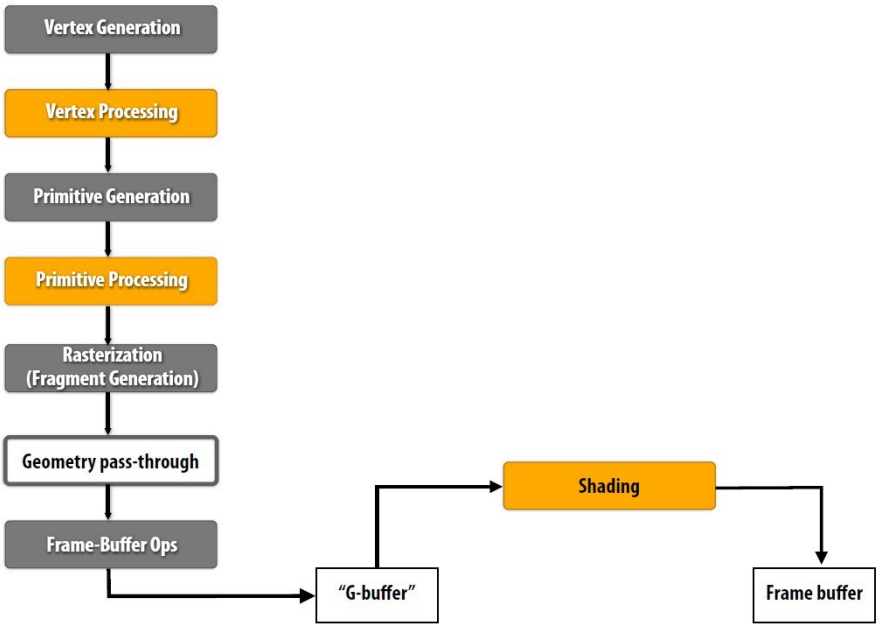


图 12 延迟渲染(Deferred Rendering)管线流程

7.6 延迟渲染的优缺点分析

这里列举一下经典版本的延迟渲染的优缺点。

7.6.1 延迟渲染的优点

Deferred Rendering 的最大的优势就是将光源的数目和场景中物体的数目在复杂度层面上完全分开。也就是说场景中不管是一个三角形还是一百万个三角形，最后的复杂度不会随光源数目变化而产生巨大变化。

一些要点：

- 复杂度仅 O(n+m)。
- 只渲染可见的像素，节省计算量。
- 用更少的 shader。
- 对后处理支持良好。
- 在大量光源的场景优势尤其明显。

7.6.2 延迟渲染的缺点

一些要点：

- 内存开销较大。
- 读写 G-buffer 的内存带宽用量是性能瓶颈。
- 对透明物体的渲染存在问题。在这点上需要结合正向渲染进行渲染。
- 对多重采样抗锯齿（MultiSampling Anti-Aliasing, MSAA）的支持不友好，主要因为需开启 MRT。

7.7 延迟渲染的改进

针对延迟渲染上述提到的缺点，下面简单列举一些降低 Deferred Rendering 存取带宽的改进方案。最简单也是最容易想到的就是将存取的 G-Buffer 数据结构最小化，这也就衍生出了 Light Pre-Pass（即 Deferred Lighting） 方法。另一种方式是将多个光照组成一组，然后一起处理，这种方法衍生了 Tile-Based Deferred Rendering。

也就是说，常见的两种 Deferred Rendering 的改进是：

- 延迟光照 Light Pre-Pass（即 Deferred Lighting）
- 分块延迟渲染 Tile-BasedDeferred Rendering

下面分别进行说明。

7.8 延迟光照 LightPre-Pass / Deferred Lighting

Light Pre-Pass 即 Deferred Lighting（延迟光照），旨在减少传统 Defferred Rendering 使用 G-buffer 时占用的过多开销（reduce G-buffer overhead），最早由 Wolfgang Engel 于 2008 年在他的博客(http://diaryofagraphicsprogrammer.blogspot.com/2008/03/light-pre-pass-renderer.html)中提到。

延迟光照的具体的思路是：

1、渲染场景中不透明（opaque ）的几何体。将法线向量 n 和镜面扩展因子（specular spread factor）m 写入缓冲区。这个 n/m-buffer 缓冲区是一个类似 G-Buffer 的缓冲区，但包含的信息更少，更轻量，适合于单个输出颜色缓冲区，因此不需要 MRT 支持。

2、渲染光照。计算漫反射和镜面着色方程，并将结果写入不同的漫反射和镜面反射累积缓冲区。这个过程可以在一个单独的 pass 中完成（使用 MRT），或者用两个单独的 pass。环境光照明可以在这个阶段使用一个 full-screen pass 进行计算。

3、对场景中的不透明几何体进行第二次渲染。从纹理中读取漫反射和镜面反射值，对前面步骤中漫反射和镜面反射累积缓冲区的值进行调制，并将最终结果写入最终的颜色缓冲区。若在上一阶段没有处理环境光照明，则在此阶段应用环境光照明。

4、使用非延迟着色方法渲染半透明几何体。

具体的流程图可以展示如下：

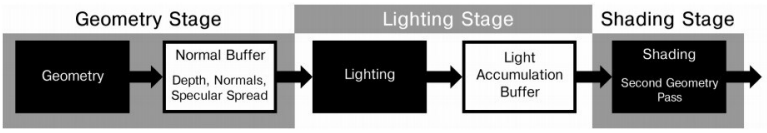


Figure 2: *The stages used in deferred lighting. The surface properties required to calculate the lighting are rendered to the normal buffer. The lighting is calculated using data from the normal buffer and stored in the light accumulation buffer. In the shading stage, all geometry is rendered again and shaded using information from only the light accumulation buffer.*

图 13 Deferred Lighting 流程图

相对于传统的 Deferred Render，使用 Light Pre-Pass 可以对每个不同的几何体使用不同 的 shader 进行渲染，所以每个物体的 material properties 将有更多变化。

这里我们可以看出对于传统的 Deferred Render，它的第二步是遍历每个光源，这样就增加了光源设置的灵活性，而 Light Pre-Pass 第三步使用的其实是 forward rendering，所以可以对每 mesh 设置其材质，这两者是相辅相成的，有利有弊。

另一个 Light Pre-Pass 的优点是在使用 MSAA 上很有利。虽然并不是 100%使用上了 MSAA（除非使用 DX10/11 的特性），但是由于使用了 Z 值和 Normal 值，就可以很容易找到边缘，并进行采样。

7.9 分块延迟渲染 Tile-Based Deferred Rendering

作为传统 Defferred Rendering 的另一种主要改进，分块延迟渲染（Tile-Based Deferred Rendering，TBDR）旨在合理分摊开销（amortize overhead），自 SIGGRAPH 2010 上提出以来逐渐为业界所了解。

实验数据表明 TBDR 在大量光源存在的情况下明显优于上文提到的 Light Pre-Pass。

我们知道，延迟渲染的瓶颈在于读写 G-buffer，在大量光源下，具体瓶颈将位于每个光源对 G-buffer 的读取及与颜色缓冲区（color buffer）混合。这里的问题是，每个光源，即使它们的影响范围在屏幕空间上有重叠，因为每个光源是在不同的绘制中进行，所以会重复读取 G-buffer 中相同位置的数据，计算后以相加混合方式写入颜色缓冲。光源越多，内存带宽用量越大。

而分块延迟渲染的主要思想则是把屏幕分拆成细小的栅格，例如每 32 × 32 象素作为一个分块（tile）。然后，计算每个分块会受到哪些光源影响，把那些光源的索引储存在分块的光源列表里。最后，逐个分块进行着色，对每像素读取 G-buffer 和光源列表及相关的光源信息。因此，G-buffer 的数据只会被读取 1 次且仅 1 次，写入 color buffer 也是 1 次且仅 1 次，大幅降低内存带宽用量。不过，这种方法需要计算光源会影响哪些分块，这个计算又称为光源剔除（light culling），可以在 CPU 或 GPU（通常以 compute shader 实现）中进行。用 GPU 计算的好处是，GPU 计算这类工作比 CPU 更快，也减少 CPU / GPU 数据传输。而且，可以计算每个分块的深度范围（depth range），作更有效的剔除。

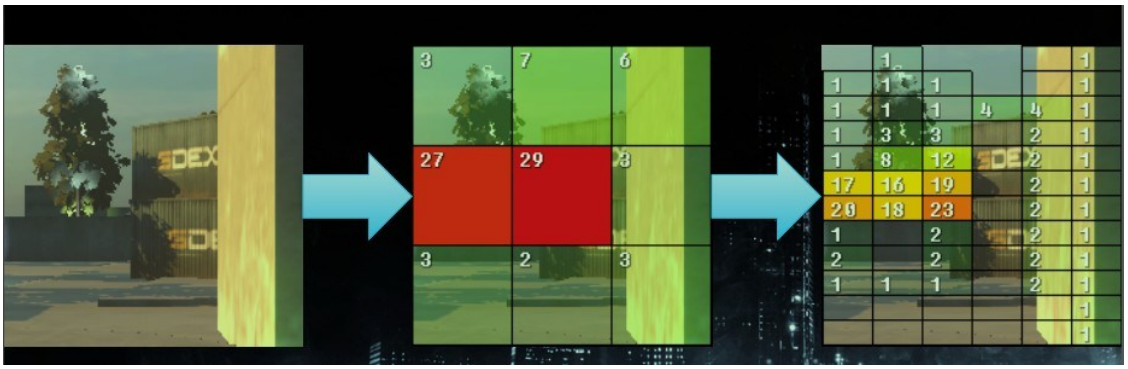


图 14 Tile-Based Deferred Rendering 图示 @GDC2011，SPU-based deferred shading for Battlefield 3 onPlaystation 3.

也就是说，TBDR 主要思想就是将屏幕分成一个个小块 tile。然后根据这些 Depth 求得每个 tile 的 bounding box。对每个 tile 的 bounding box 和 light 进行求交，这样就得到了对该 tile 有作用 的 light 的序列。最后根据得到的序列计算所在 tile 的光照效果。

对比 Deferred Rendering，之前是对每个光源求取其作用区域 light volume，然后决定其作用的的 pixel，也就是说每个光源要求取一次。而使用 TBDR，只要遍历每个 pixel，让其所属 tile 与光线求交，来计算作用其上的 light，并利用 G-Buffer 进行 Shading。一方面这样做减少了所需考虑的光源个数，另一方面与传统的 Deferred Rendering 相比，减少了存取的带宽。

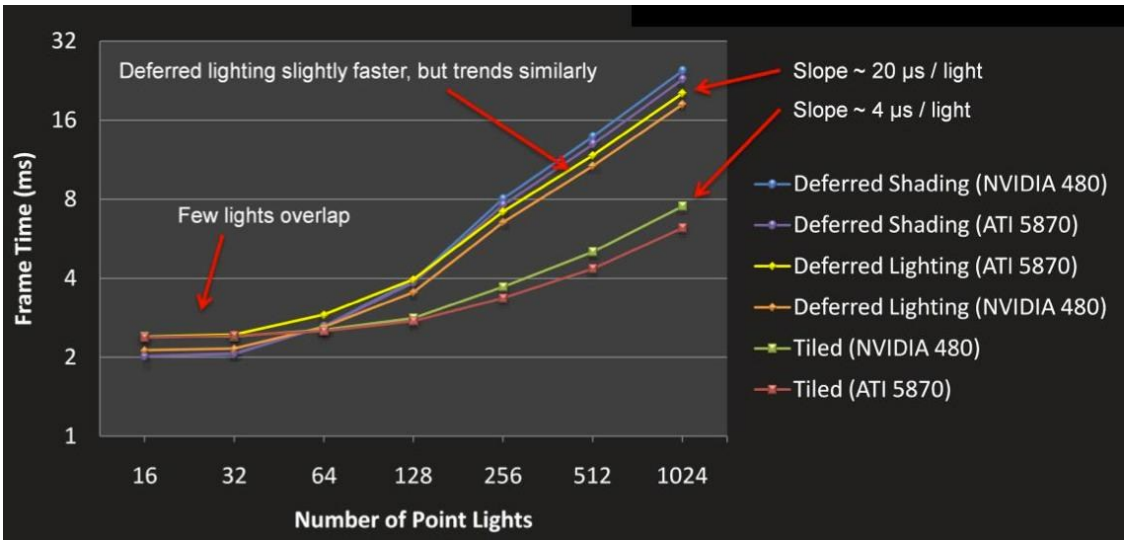


图 15 在 1920x1080 分辨率下，Tile-Based vs. 传统 deferred shading

图 16 使用 Tile-Based Deferred Rendering 思路渲染的场景，场景含 4096 个点光源

7.10 延迟渲染 vs 延迟光照

关于延迟着色和延迟光照，经常会被弄混，这边简单区分一下。

- 延迟渲染（Deferred Rendering）又称延迟着色（Deferred Shading），在 2004 年的 GDC 上被提出。
- 延迟光照（Deferred Lighting）又称 Light Pre-Pass，是延迟着色的一种改进，在 2008 年被提出。

Deferred Rendering 与 Deferred Lighting 在思想上的主要异同：

- DeferredShading 需要更大的 G-Buffer 来完成对 Deferred 阶段的前期准备，而且一般需要硬件有 MRT 的支持，可以说是硬件要求更高。
- DeferredLighting 需要两个几何体元的绘制过程来来完成整个渲染操作：G-Pass 与 Shading pass。这个既是劣势也是优势：由于 Deferred Shading 中的 Deffered 阶段是在完全基于 G-Buffer 的屏幕空间进行，这也导致了物体材质信息的缺失，这样在处理多变的渲染风格时就需要额外的操作；而 Deferred Lighting 却可以在 Shading 阶段得到物体的材质信息进而使这一问题的处理变得较简单。
- 两种方法的上述操作均是只能完成对不透明物体的渲染，而透明或半透明的物体则需额外的传统 Pass 来完成。

两者流程图的对比：

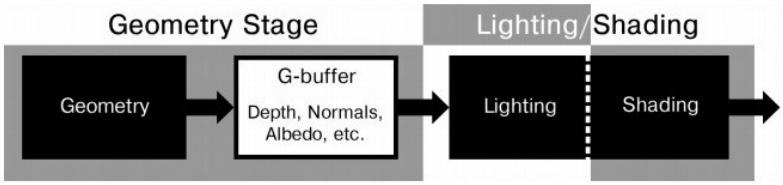


Figure 1: *The stages used in deferred shading. The first stage renders several properties of the opaque geometry to the G-buffer. The G-buffer is used to calculate the lighting and shading, which can be done in a single stage or two separate stages.*

图 17 Deferred Shading 流程图

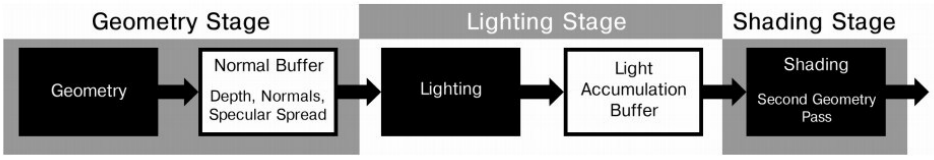


Figure 2: *The stages used in deferred lighting. The surface properties required to calculate the lighting are rendered to the normal buffer. The lighting is calculated using data from the normal buffer and stored in the light accumulation buffer. In the shading stage, all geometry is rendered again and shaded using information from only the light accumulation buffer.*

图 18 Deferred Lighting 流程图

7.11 实时渲染中常见的 Rendering Path 总结

这一节对常见实时渲染中常见的几种 Rendering Path 进行一个简单小节。

本文目前已经提到的 Rendering Path 有：

- 正向渲染 （Forward Rendering）
- 延迟渲染 （Deferred Rendering）
- 延迟光照 （Light Pre-Pass / Deferred Lighting）
- 分块延迟渲染（Tile-Based Deferred Rendering）

除此之外，还有如下一些后来提出的 Rendering Path 比较有趣：

- Forward+（即 Tiled Forward Rendering，分块正向渲染）
- 群组渲染 Clustered Rendering

篇幅原因，这边就不展开了，有兴趣的朋友不妨去查阅相关资料进行了解。

7.12 环境映射 Environment Mapping

最后简单聊一聊《Real-Time Rendering 3rd》第八章“区域和环境光照 Area and Environmental Lighting”中的 EnvironmentMapping 环境映射。下一篇文章，就直接开始提炼第九章“Global Illumination 全局光照”的内容。

Environment mapping（环境映射），又称 Reflection Mapping（反射映射）,是计算机图形学领域中使用基于图像的光照（Image-Based Lighting，IBL）技术，用预先计算的纹理图像模拟复杂镜面的一种高效方法。由 Blinn 和 Newell 在 1976 首次提出。

由于是事先准备好的数据，这种实现方法比传统的光线跟踪算法效率更高，但是需要注意的是这种方法是实际反射的一种近似，有时甚至是非常粗糙的近似。这种技术的一个典型的缺点是没有考虑自反射，即无法看到物体反射的物体自身的某一部分。

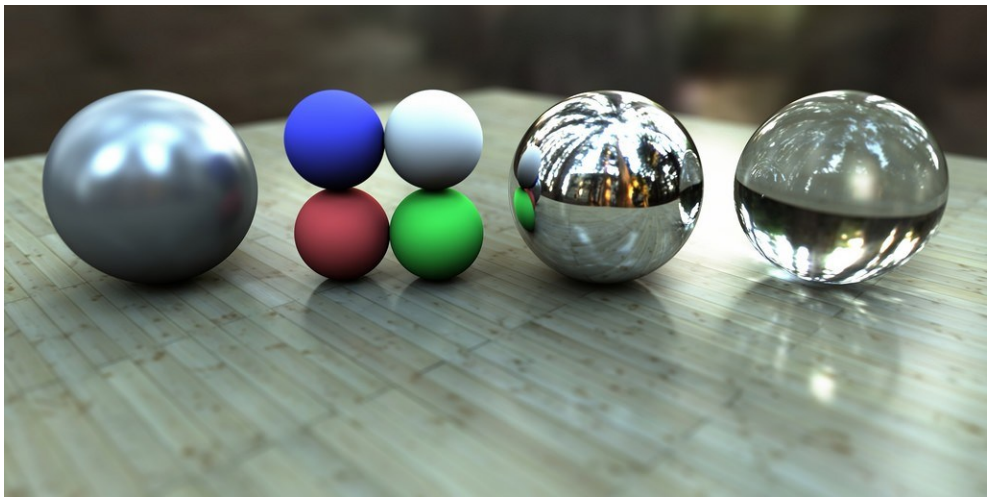


图 19 Image Based Lighting Environment Mapping 环境映射效果图

环境映射的常见类型有：

- 球型环境映射 Sphere Environment Mapping
- 立方体环境映射 Cubic Environment Mapping
- 抛物线环境映射 Parabolic Environment Mapping

环境映射的一些引申：

- 光泽反射环境映射(Glossy Reflections from Environment Maps)
- 基于视角的反射映射(View-Dependent Reflection Maps)
- 辐照度环境映射 (Irradiance Environment Mapping)

另外，推荐一个下载 CubeMap 资源的站点：

http://www.humus.name/index.php?page=Textures

7.13 Reference

[1] Lauritzen, Andrew. "Deferredrendering for current and future rendering pipelines." SIGGRAPH Course:Beyond Programmable Shading (2010): 1–34.

[2] Coffin, Christina. "SPU–based deferredshading for Battlefield 3 on Playstation 3." Game Developer ConferencePresentation. Vol. 8. 2011.

[3] Lee M. Pre–lighting in Resistance 2[J].GDC San Francisco, 2009.

[4] Valient M. Deferred rendering inKillzone 2[C]//The Develop Conference and Expo. 2007.

[5]Andersson, Johan. "Directx 11rendering in battlefield 3." Game Developers Conference. Vol. 2. 2011.

[6] http://www.cnblogs.com/ghl_carmack/p/4150232.html

[7] https://twitter.com/jimmikaelkael/status/812631802242273280

[8] http://www.realtimerendering.com/blog/deferred–lighting–approaches/

[9] http://miloyip.com/2014/many–lights/

[10] http://www.cnblogs.com/polobymulberry/p/5126892.html

[11] http://blog.csdn.net/bugrunner/article/details/7436600

[12] https://learnopengl-cn.readthedocs.io/zh/latest/05%20Advanced%20Lighting/08%20Deferred%20Shading/

[13] http://www.cnblogs.com/ghl_carmack/p/4150232.html

[14] http://gameangst.com/?p=141

[15] https://www.flickr.com/photos/mylaboratory/2332900823/in/photostream/

[16] https://gamedevcoder.wordpress.com/2011/04/11/light-pre-pass-vs-deferred-renderer-part-1/

[17] CMU 15869 lecture 12 Slides：

http://www.cs.cmu.edu/afs/cs/academic/class/15869-f11/www/lectures/12_deferred_shading.pdf

[18] Lauritzen A. Deferred rendering forcurrent and future rendering pipelines[J]. SIGGRAPH Course: Beyond ProgrammableShading, 2010: 1-34. https://software.intel.com/sites/default/files/m/d/4/1/d/8/lauritzen_deferred_shading_siggraph_2010.pdf

[19] Nguyen, Hubert. GPU Gems 3. Chapter 19. Addison-Wesley Professional, 2007.

第八章 全局光照：光线追踪、路径追踪与GI技术进化编年史

全局光照（Global Illumination,简称 GI），作为图形学中比较酷的概念之一，是指既考虑场景中来自光源的直接光照，又考虑经过场景中其他物体反射后的间接光照的一种渲染技术。

大家常听到的光线追踪，路径追踪等同样很酷的概念，都是全局光照中人气较高的算法流派。

而这篇文章将围绕全局光照技术，介绍的要点有：

- 全局光照的基本概念
- 全局光照的算法主要流派
- 全局光照技术进化编年史
- 光线追踪 Ray Tracing
- 路径追踪 Path Tracing

- 光线追踪、路径追踪、光线投射的区别
- 环境光遮蔽 Ambient Occlusion

8.1 行文思路说明

阅读过《Real-Time Rendering 3rd》第九章的读者们都会发现，作为一章关于全局光照的章节，作者讲了不少在严格意义上全局光照主线以外的内容，如 Reflections、Refractions、Shadow 等节，而这些内容在《Real-Time Rendering 2nd》中，其实是放在 Chapter 6 Advanced Lighting and Shading 一节的。

既然《Real-Time Rendering 3rd》第九章标题就叫全局光照，核心内容也是全局光照，本文即决定脱离原书安排的 100 来页的多余内容，以全局光照的主线内容为主，构成一篇包含全局光照基本概念，主要算法流派，以及全局光照技术进化编年史，和全局光照算法中人气较高的光线追踪、路径追踪等算法的综述式文章。

8.2 全局光照

全局光照，(Global Illumination,简称 GI), 或被称为 Indirect Illumination, 间接光照，是指既考虑场景中直接来自光源的光照（Direct Light）又考虑经过场景中其他物体反射后的光照（Indirect Light）的一种渲染技术。使用全局光照能够有效地增强场景的真实感。

即可以理解为：全局光照 = 直接光照(Direct Light) + 间接光照(Indirect Light)

图 1 Direct illumination

图 2 Global illumination = Direct illumination +Indirect illumination

上述两幅图片来自 CMU 15-462/15-662, Fall 2015 Slider，Lecture 14: Global Illumination,当然，细心的朋友也可以发现，它也被《Physically Based Rendering,Second Edition From Theory To Implementation》选作封面。

同样可以看到，加入了 Indirect illumination 的图 2，在直接光源（阳光）照射不到的地方，得到了更好的亮度和细节表现，从而使整张渲染效果更具真实感。

虽说实际应用中只有漫反射全局照明的模拟算法被称为全局照明算法，但其实理论上说反射、折射、阴影都属于全局光照的范畴，因为模拟它们的时候不仅仅要考虑光源对物体的直接作用还要考虑物体与物体之间的相互作用。也是因为，镜面反射、折射、阴影一般不需要进行复杂的光照方程求解，也不需要进行迭代的计算。因此，这些部分的算法已经十分高效，甚至可以做到实时。不同于镜面反射，光的漫反射表面反弹时的方向是近似“随机”，因此不能用简单的光线跟踪得到反射的结果，往往需要利用多种方法进行多次迭代，直到光能分布达到一个基本平衡的状态。

8.3 全局光照的主要算法流派

经过几十年的发展，全局光照现今已有多种实现方向，常见的全局光照主要流派列举如下：

- Ray tracing 光线追踪
- Path tracing 路径追踪
- Photon mapping 光子映射
- Point Based Global Illumination 基于点的全局光照
- Radiosity 辐射度
- Metropolis light transport 梅特波利斯光照传输
- Spherical harmonic lighting 球谐光照
- Ambient occlusion 环境光遮蔽
- Voxel-based Global Illumination 基于体素的全局光照
- Light Propagation Volumes Global Illumination
- Deferred Radiance Transfer Global Illumination
- Deep G-Buffer based Global Illumination
- 等。

而其中的每种流派，又可以划分为 N 种改进和衍生算法。

如光线追踪（Ray Tracing）派系，其实就是一个框架，符合条件的都可称为光线追踪，其又分为递归式光线追踪（Whitted-style Ray Tracing），分布式光线追踪（DistributionRay Tracing），蒙特卡洛光线追踪（Monte Carlo Ray Tracing）等。

而路径追踪（Path tracing）派系，又分为蒙特卡洛路径追踪（Monte Carlo Path Tracing），双向路径追踪（Bidirectional Path Tracing），能量再分配路径追踪（Energy Redistribution Path Tracing）等。

其中有些派系又相互关联，如路径追踪，就是基于光线追踪，结合了蒙特卡洛方法而成的一种新的派系。

8.4 全局光照技术进化编年史

这节以光线追踪和路径追踪派系为视角，简单总结一下全局光照技术发展早期（1968-1997）的重要里程碑。

8.4.1 光线投射 Ray Casting [1968]

光线投射（Ray Casting），作为光线追踪算法中的第一步，其理念起源于 1968 年，由 Arthur Appel 在一篇名为《 Some techniques for shading machine rendering of solids》的文章中提出。其具体思路是从每一个像素射出一条射线，然后找到最接近的物体挡住射线的路径，而视平面上每个像素的颜色取决于从可见光表面产生的亮度。

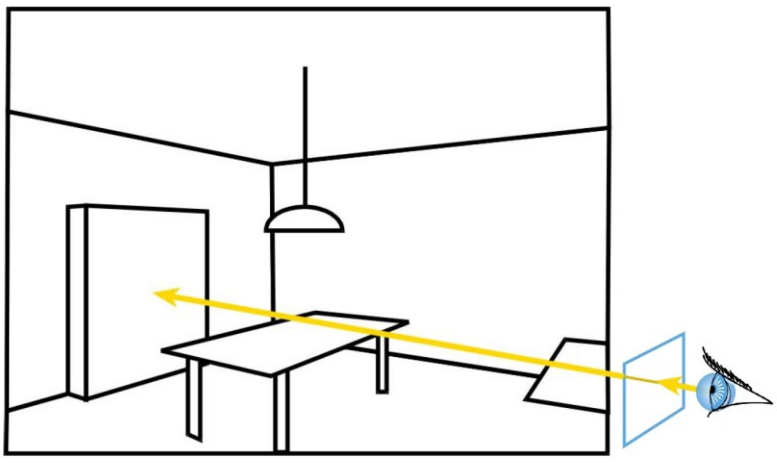


图 3 光线投射：每像素从眼睛投射射线到场景

8.4.2 光线追踪 Ray Tracing [1979]

1979 年，Turner Whitted 在光线投射的基础上，加入光与物体表面的交互，让光线在物体表面沿着反射，折射以及散射方式上继续传播，直到与光源相交。这一方法后来也被称为经典光线跟踪方法、递归式光线追踪（Recursive Ray Tracing）方法，或 Whitted-style 光线跟踪方法。

光线追踪方法主要思想是从视点向成像平面上的像素发射光线，找到与该光线相交的最近物体的交点，如果该点处的表面是散射面，则计算光源直接照射该点产生的颜色；如果该点处表面是镜面或折射面，则继续向反射或折射方向跟踪另一条光线，如此递归下去，直到光线逃逸出场景或达到设定的最大递归深度。

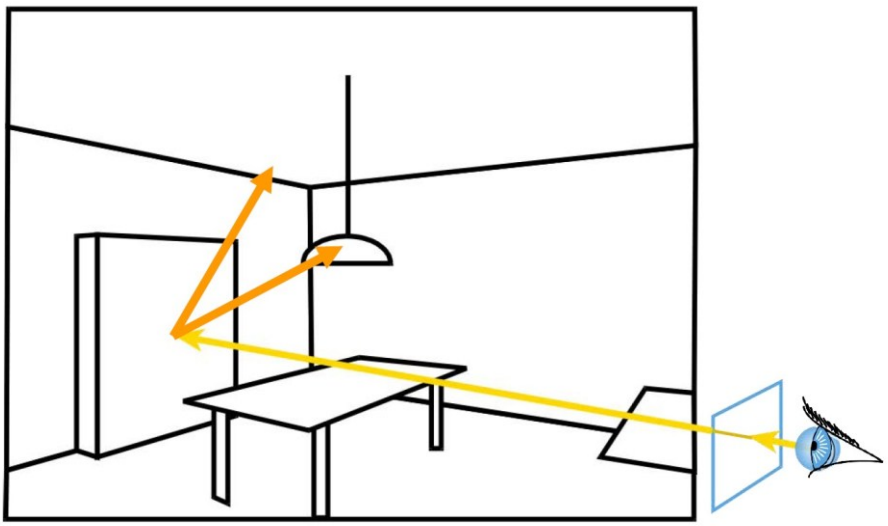


图 4 经典的光线追踪： 每像素从眼睛投射射线到场景，并追踪次级光线（shadow, reflection, refraction），并结合递归

8.4.3 分布式光线追踪 Distributed Ray Tracing [1984]

Cook 于 1984 年引入蒙特卡洛方法（Monte Carlo method）到光线跟踪领域，将经典的光线跟踪方法扩展为分布式光线跟踪算法（Distributed Ray Tracing），又称为随机光线追踪（stochasticray tracing），可以模拟更多的效果，如金属光泽、软阴影、景深（Depth of Field）、运动模糊等等。

8.4.4 渲染方程 The Rendering Equation [1986]

在前人的研究基础上，Kajiya 于 1986 年进一步建立了渲染方程的理论，并使用它来解释光能传输的产生的各种现象。这一方程描述了场景中光能传输达到稳定状态以后，物体表面某个点在某个方向上的辐射率（Radiance）与入射辐射亮度等的关系。

可以将渲染方程理解为全局光照算法的基础，Kajiya 在 1986 年第一次将渲染方程引入图形学后，随后出现的很多全局光照的算法，都是以渲染方程为基础，对其进行简化的求解，以达到优化性能的目的。渲染方程根据光的物理学原理，以及能量守恒定律，完美地描述了光能在场景中的传播。很多真实感渲染技术都是对它的一个近似。渲染方程在数学上的表示如下：

$$L_o(x,\vec{w}) = L_e(x,\vec{w}) + \int_\Omega f_r(x,\vec{w}',\vec{w}) L_i(x,\vec{w}')(\vec{w}' \cdot \vec{n}) d\vec{w}'$$

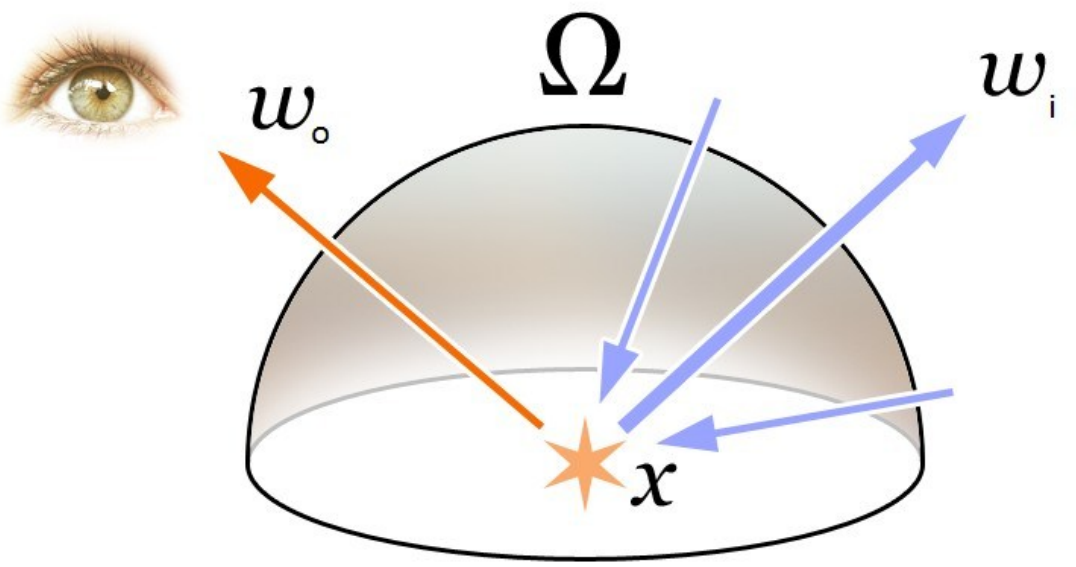


图 5 渲染方程描述了从 x 点沿某一方向看的光放射的总额。

8.4.5 路径追踪 Path Tracing [1986]

Kajiya 也于 1986 年提出了路径追踪算法的理念，开创了基于蒙特卡洛的全局光照这一领域。根据渲染方程， Kajiya 提出的路径追踪方法是第一个无偏（Unbiased）的渲染方法。路径追踪的基本思想是从视点发出一条光线，光线与物体表面相交时根据表面的材质属性继续采样一个方向，发出另一条光线，如此迭代，直到光线打到光源上（或逃逸出场景），然后用蒙特卡洛的方法，计算其贡献，作为像素的颜色值。

8.4.6 双向路径追踪 Bidirectional Path Tracing [1993，1994]

双向路径追踪（Bidirectional Path Tracing）的基本思想是同时从视点、光源打出射线，经过若干次反弹后，将视点子路径（eye path）和光源子路径（light path）上的顶点连接起来（连接时需要测试可见性），以快速产生很多路径。这种方法能够产生一些传统路径追踪难以采样到的光路，所以能够很有效地降低噪声。进一步， [Veach 1997]将渲染方程改写成对路径积分的形式，允许多种路径采样的方法来求解该积分。

8.4.7 梅特波利斯光照传输 Metropolis Light Transport [1997]

Eric Veach 等人于 1997 年提出了梅特波利斯光照传输（Metropolis Light Transport，常被简称为 MLT）方法。路径追踪（Path Tracing）中一个核心问题就是怎样去尽可能多的采样一些贡献大的路径，而该方法可以自适应的生成贡献大的路径，简单来说它会避开贡献小的路径，而在贡献大的路径附近做更多局部的探索，通过特殊的变异方法，生成一些新的路径，这些局部的路径的贡献往往也很高。 与双向路径追踪相比， MLT 更加鲁棒，能处理各种复杂的场景。比如说整个场景只通过门缝透进来的间接光照亮，此时传统的路径追踪方法因为难以采样到透过门缝的这样的特殊路径而产生非常大的噪声。

8.5 光线追踪 Ray Tracing

光线追踪（Ray tracing）是三维计算机图形学中的特殊渲染算法，跟踪从眼睛发出的光线而不是光源发出的光线，通过这样一项技术生成编排好的场景的数学模型显现出来。这样得到的结果类似于光线投射与扫描线渲染方法的结果，但是这种方法有更好的光学效果，例如对于反射与折射有更准确的模拟效果，并且效率非常高，所以当追求高质量的效果时经常使用这种方法。

上文已经提到过，Whitted 于 1979 年提出了使用光线跟踪来在计算机上生成图像的方法，这一方法后来也被称为经典光线跟踪方法、递归式光线追踪方法，或 Whitted-style 光线跟踪方法。其主要思想是从视点向成像平面上的像素发射光线，找到与该光线相交的最近物体的交点，如果该点处的表面是散射面，则计算光源直接照射该点产生的颜色；如果该点处表面是镜面或折射面，则继续向反射或折射方向跟踪另一条光线，如此递归下去，直到光线逃逸出场景或达到设定的最大递归深度。

以下这张图示可以很好的说明光线追踪方法的思路：

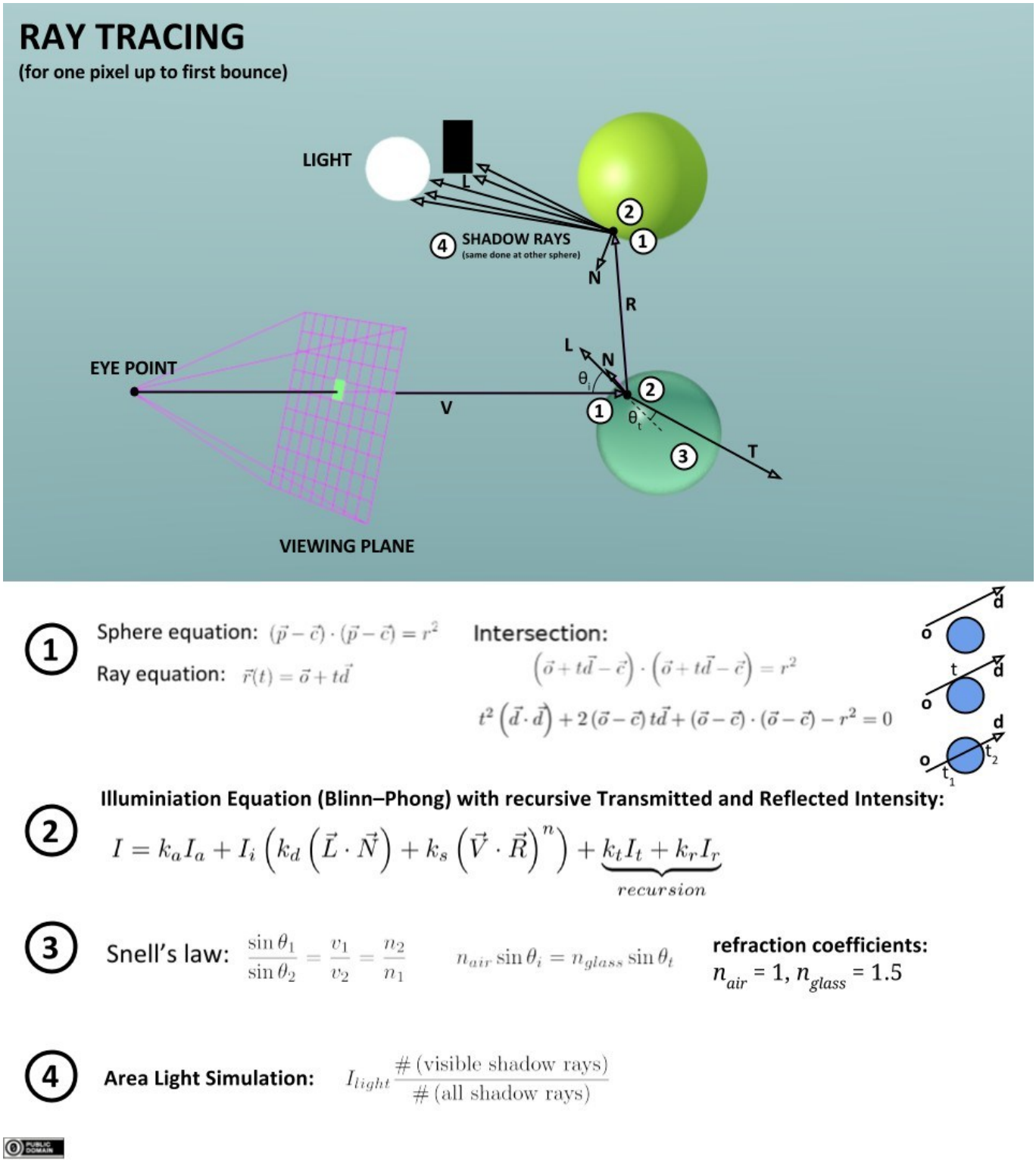


图 6 Ray Tracing Illustration First Bounce

图 7 基于光线追踪渲染出的效果图 1

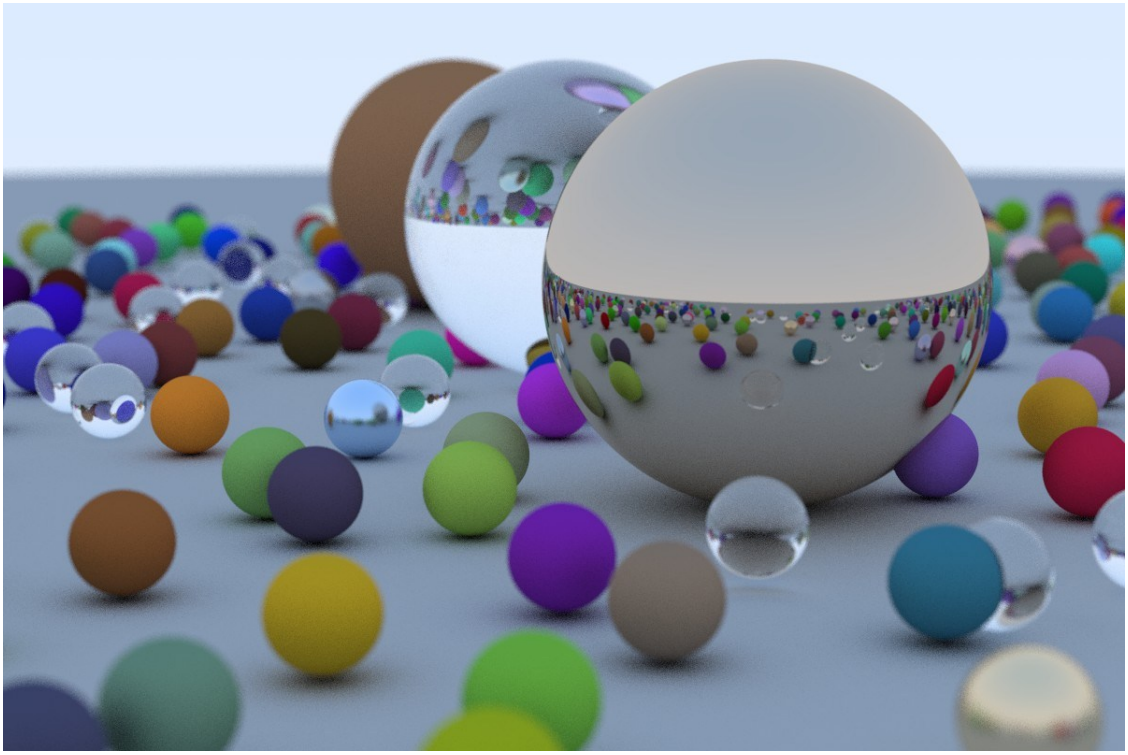


图 8 基于光线追踪渲染出的效果图 2

图 9 基于光线追踪渲染效果图 @Caustic-Graphics，Inc

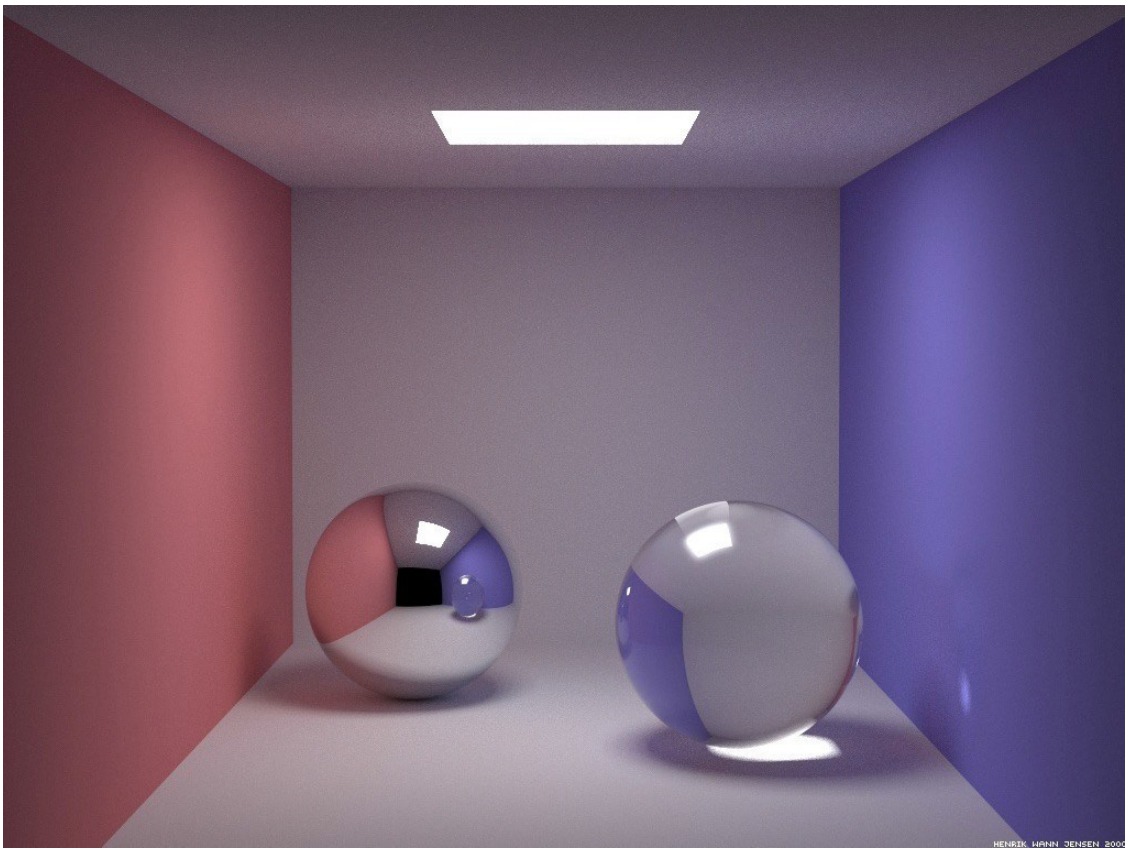


图 10 典型的光线追踪渲染效果图

光线跟踪的一个最大的缺点就是性能，需要的计算量非常巨大，以至于目前的硬件很难满足实时光线追踪的需求。传统的光栅图形学中的算法，利用了数据的一致性从而在像素之间共

享计算，而光线跟踪通常是将每条光线当作独立的光线，每次都要重新计算。但是，这种独立的做法也有一些其它的优点，例如可以使用更多的光线以抗混叠现象，并且在需要的时候可以提高图像质量。尽管它正确地处理了相互反射的现象以及折射等光学效果，但是传统的光线跟踪并不一定是真实效果图像，只有在非常近似或者完全实现渲染方程的时候才能实现真正的真实效果图像。由于渲染方程描述了每个光束的物理效果，所以实现渲染方程可以得到真正的真实效果，但是，考虑到所需要的计算资源，这通常是无法实现的。于是，所有可以实现的渲染模型都必须是渲染方程的近似，而光线跟踪就不一定是最为可行的方法。包括光子映射在内的一些方法，都是依据光线跟踪实现一部分算法，但是可以得到更好的效果。

用一套光线追踪的伪代码，结束这一节的内容：

```
for each pixel of the screen
{
    Final color = 0;
        Ray = { starting point, direction };
        Repeat
        {
            for each object in the scene
            {
                    determine closest ray object/intersection;
            }
            if intersection exists
            {
                    for each light inthe scene
                    {
                            if the light is not in shadow of anotherobject
                            {
                                    addthis light contribution to computed color;
                            }
                    }
            }
            Final color = Final color + computed color * previous reflectionfactor;
            reflection factor = reflection factor * surface reflectionproperty;
            increment depth;
    } until reflection factor is 0 or maximumdepth is reached
}
```

8.6 路径追踪 Path Tracing

路径追踪（Path Tracing）方法由 Kajiya 在 1986 年提出，该方法的基本思想是从视点发出一条光线，光线与物体表面相交时根据表面的材质属性继续采样一个方向，发出另一条光线，如此迭代，直到光线打到光源上（或逃逸出场景），然后用蒙特卡洛方法，计算光线的贡献，作为像素的颜色值。而使用蒙特卡洛方法对积分的求解是无偏的，只要时间足够长，最终图像能收敛到一个正确的结果。

简单来说，路径追踪 = 光线追踪+ 蒙特卡洛方法。

这边有一个用 99 行代码实现路径追踪算法的一个简易全局光照渲染器，有兴趣的朋友可以进行了解：

http://www.kevinbeason.com/smallpt/

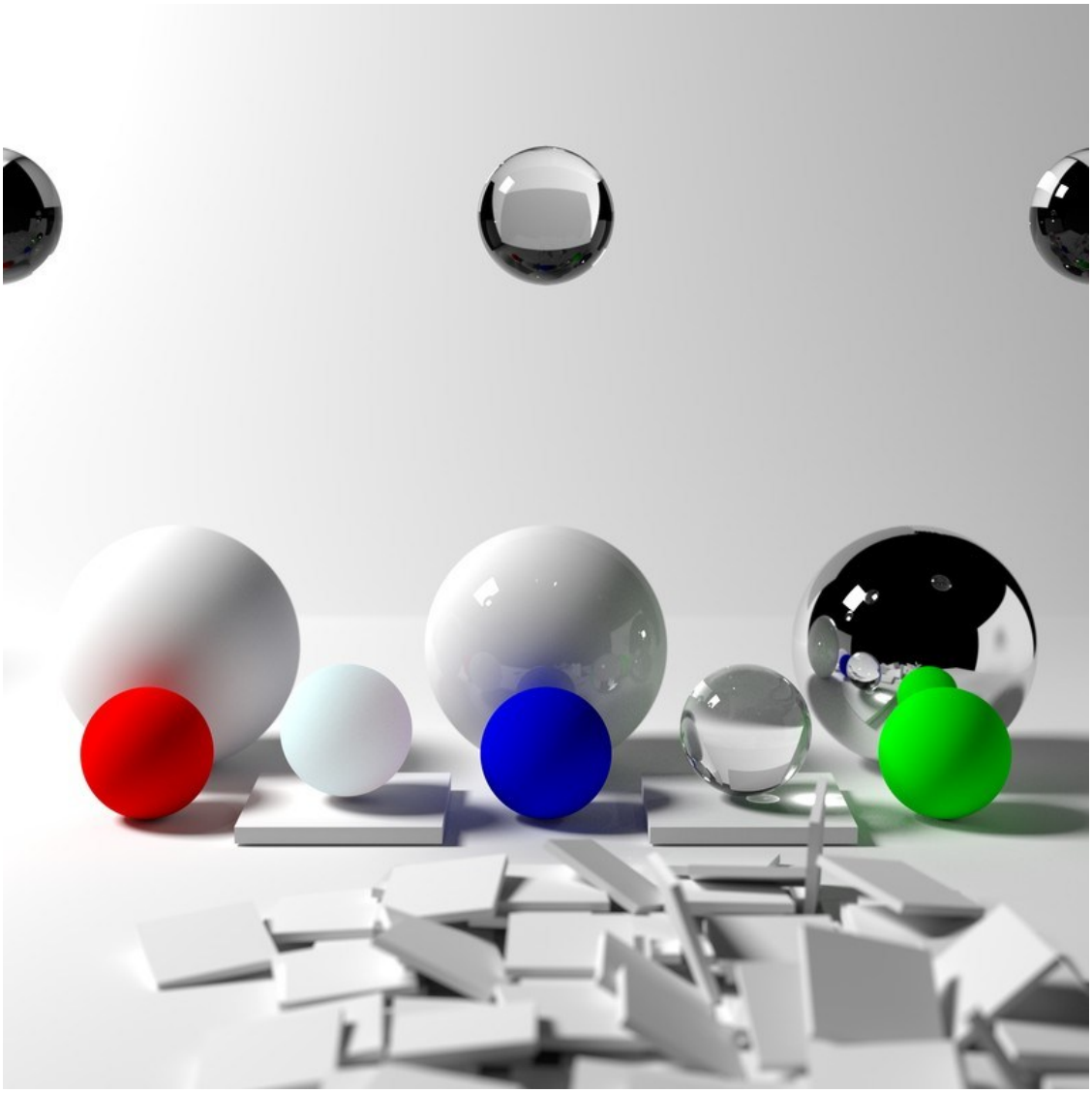


图 11 基于路径追踪渲染的效果图

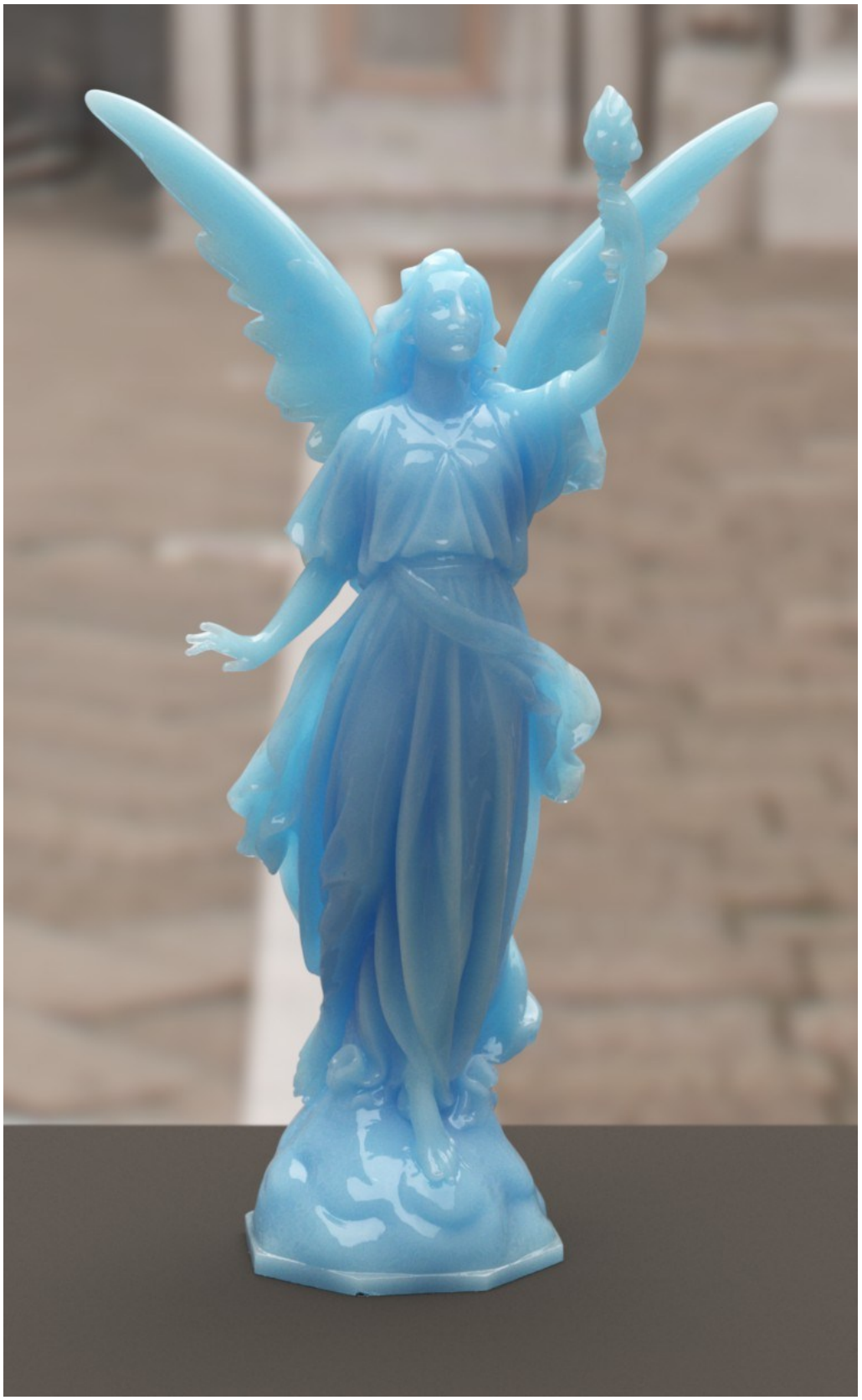


图 12 基于路径追踪实现的次表面散射渲染效果图 ©Photorealizer

图 13 基于路径追踪渲染的效果图 ©http://www.pathtracing.com

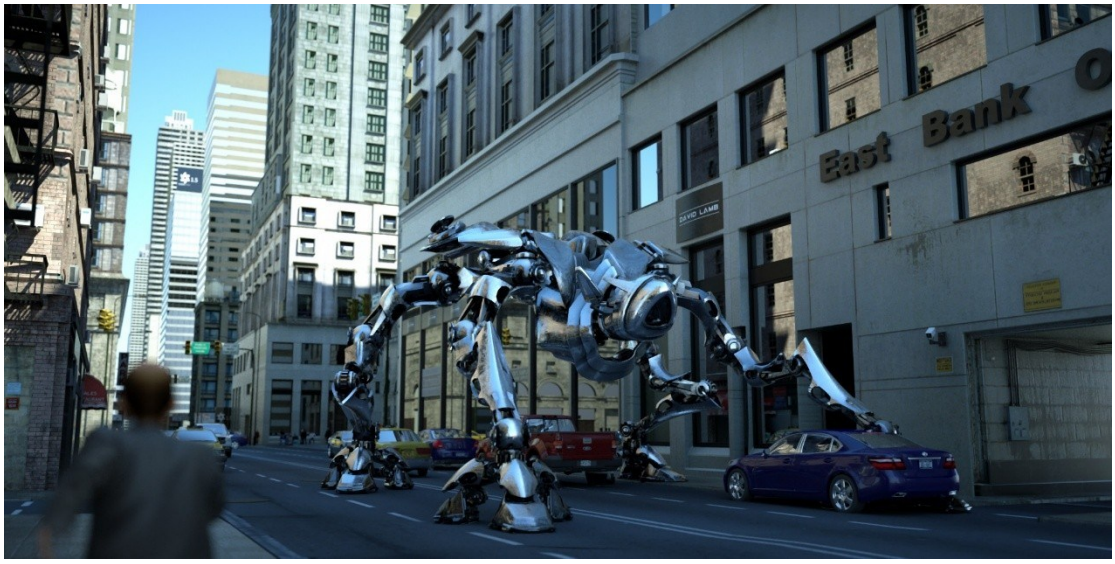


图 14 基于路径追踪渲染的效果图 ©NVIDIA

8.7 Ray Casting ， Ray Tracing，Path Tracing 区别

初学者往往会弄不明白光线投射（Ray Casting ），光线追踪（Ray Tracing），路径追踪（Path Tracing）三者的的区别，龚大 @叛逆者 在 https://www.zhihu.com/question/29863225 这个答案中的回答已经很精辟，本文就直接引用了过来：

- Ray Tracing：这其实是个框架，而不是个方法。符合这个框架的都叫 raytracing。这个框架就是从视点发射 ray，与物体相交就根据规则反射、折射或吸收。遇到光源或者走太远就停住。一般来说运算量不小。
- Ray Casting：其实这个和 volumetric 可以脱钩。它就是 ray tracing 的第一步，发射光线，与物体相交。这个可以做的很快，在 Doom 1 里用它来做遮挡。
- Path Tracing：是 ray tracing + 蒙特卡洛法。在相交后会选一个随机方向继续跟踪，并根据 BRDF 计算颜色。运算量也不小。还有一些小分类，比如 Bidirectional path tracing。

文末，简单聊一下环境光遮蔽，AO。

8.8 环境光遮蔽 Ambient Occlusion

环境光遮蔽（Ambient Occlusion，简称 AO）是全局光照明的一种近似替代品，可以产生重要的视觉明暗效果，通过描绘物体之间由于遮挡而产生的阴影， 能够更好地捕捉到场景中的细节，可以解决漏光，阴影漂浮等问题，改善场景中角落、锯齿、裂缝等细小物体阴影不清晰等问题，增强场景的深度和立体感。

可以说，环境光遮蔽在直观上给玩家的主要感觉体现在画面的明暗程度上，未开启环境光遮蔽特效的画面光照稍亮一些；而开启环境光遮蔽特效之后， 局部的细节画面尤其是暗部阴影会更加明显一些。

Ambient Occlusion 的细分种类有：

- SSAO-Screen space ambient occlusion
- SSDO-Screen space directional occlusion
- HDAO-High Definition Ambient Occlusion
- HBAO+-Horizon Based Ambient Occlusion+
- AAO-Alchemy Ambient Occlusion
- ABAO-Angle Based Ambient Occlusion
- PBAO
- VXAO-Voxel Accelerated Ambient Occlusion

一般而言，Ambient Occlusion 最常用方法是 SSAO，如 Unreal Engine 4 中的 AO，即是用 SSAO 实现。

最后，贴一些和 AO 相关的，较经典的渲染效果图，结束这篇文章。

图 15 Scene without Ambient Occlusion ©Unreal

图 16 Ambient Occlusion Only ©Unreal

图 17 Scene with Ambient Occlusion ©Unreal

图 18 使用环境光遮蔽制作人物的步骤

图 19 一张典型的环境光遮蔽的渲染图

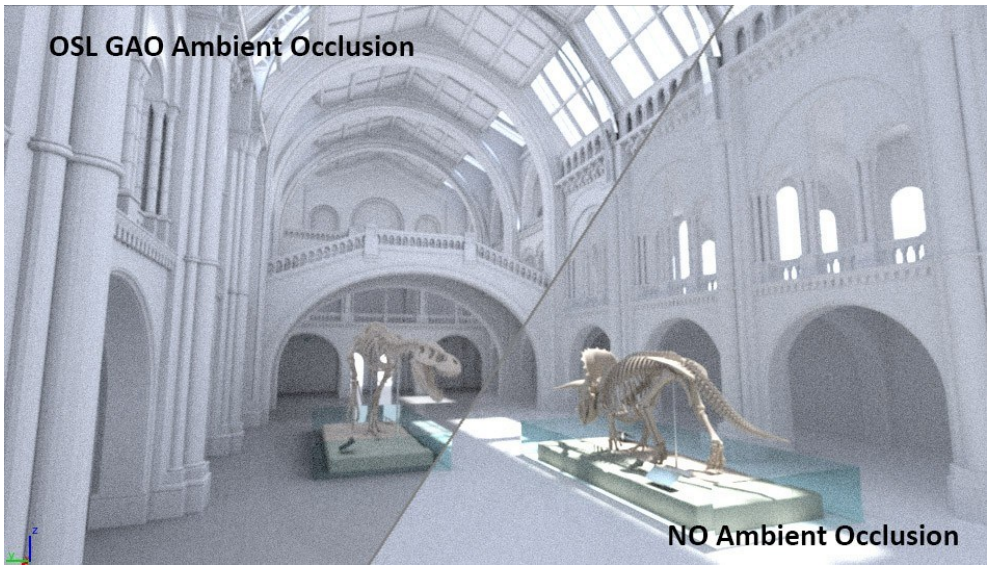


图 20 有无环境光遮蔽渲染效果对比图示

8.9 Reference

[1] http://15462.courses.cs.cmu.edu/fall2015/lecture/globalillum

[2] https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/AmbientOcclusion/

[3] https://en.wikipedia.org/wiki/Ambient_occlusion

[4] https://www.ics.uci.edu/~gopi/CS211B/RayTracing%20tutorial.pdf

[5] http://www.cnblogs.com/hielvis/p/6371840.html

[6] http://blog.csdn.net/thegibook/article/details/53058206

[7] http://www.di.ubi.pt/~agomes/cig/teoricas/02-raycasting.pdf

[8] https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-837-computer-graphics-fall-2012/

[9] Crytek on DX12, Vulkan, Async Compute, Global Illumination, Ray-tracing, Physically-based Rendering & E3 Demos

第九章 游戏开发中基于图像的渲染技术总结

本章是一篇近万字的总结式文章，关于游戏开发中基于图像的渲染（Image-Based Rendering，简称 IBR）技术的方方面面，将总结《RTR3》书中第十章提到的 16 种常用的 IBR 渲染技术。

他们包括：

- 渲染谱 The Rendering Spectrum
- 固定视角的渲染 Fixed-View Rendering
- 天空盒 Skyboxes
- 光场渲染 Light Field Rendering
- 精灵与层 Sprites and Layers
- 公告板 Billboarding

- 粒子系统 Particle System
- 替代物 Impostors
- 公告板云 Billboard Clouds
- 图像处理 Image Processing
- 颜色校正 Color Correction
- 色调映射 Tone Mapping
- 镜头眩光和泛光 Lens Flare and Bloom
- 景深 Depth of Field
- 运动模糊 Motion Blur
- 体渲染 Volume Rendering

在过去很多年里，基于图像的渲染（Image-Based Rendering ，简称 IBR），已经自成一派，逐渐发展成了一套广泛的渲染理念。正如其字面所表示的，图像是用于渲染的主要数据来源。用图像表示一个物体的最大好处在于渲染消耗与所要绘制的像素数量成正比，而不是几何模型的顶点数量。因此，使用基于图像的渲染是一种有效的渲染模型的方法。除此之外，IBR 技术还有其他一些更为广泛的用途，比如云朵，皮毛等很多很难用多边形来表示的物体，却可以巧妙运用分层的半透明图像来显示这些复杂的表面。

OK，下面开始正文，对这 16 种常见的基于图像的渲染技术，分别进行介绍。

9.1 渲染谱 The Rendering Spectrum

众所周知，渲染的目的就是在屏幕上渲染出物体，至于如何达到结果，主要依赖于用户的选择，白猫黑猫，抓到老鼠的就是好猫。而用多边形将三维物体显示在屏幕上，并非是进行三维渲染的唯一方法，也并非是最合适的方法。多边形具有从任何视角以合理的方式表示对象的优点，当移动相机的时候，物体的表示可以保持不变。但是，当观察者靠近物体的时候，为了提高显示质量，往往希望用比较高的细节层次来表示模型。与之相反，当物体位于比较远的地方时，就可以用简化形式来表示模型。这就是细节层次技术(Level Of Detail,LOD)。使用 LOD 技术主要目的是为了加快场景的渲染速度。

还有很多技术可以用来表示物体逐渐远离观察者的情形，比如，可以用图像而不是多边形来表示物体，从而减少开销，加快渲染速度。另外，单张图片可以很快地被渲染到屏幕上，用来表示物体往往开销很小。

如《地平线：黎明》远处的树木，即是采用公告板技术（Billboard）替换 3D 树木模型进行渲染。（关于公告板技术的一些更具体的总结，详见本文第六节）。

图 1 《地平线：黎明》中利用了 Billboard 进行画面的渲染

Lengyel 于 1998 在《The Convergence of Graphics and Vision》一文中提出了一种表示渲染技术连续性的方法，名为 The Rendering Spectrum 渲染谱，如下图所示。

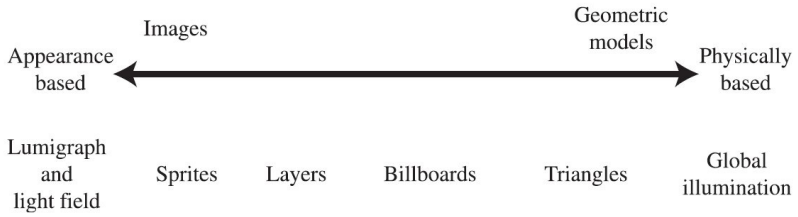


图 2 渲染谱 The Rendering Spectrum（RTR3 书中版本）

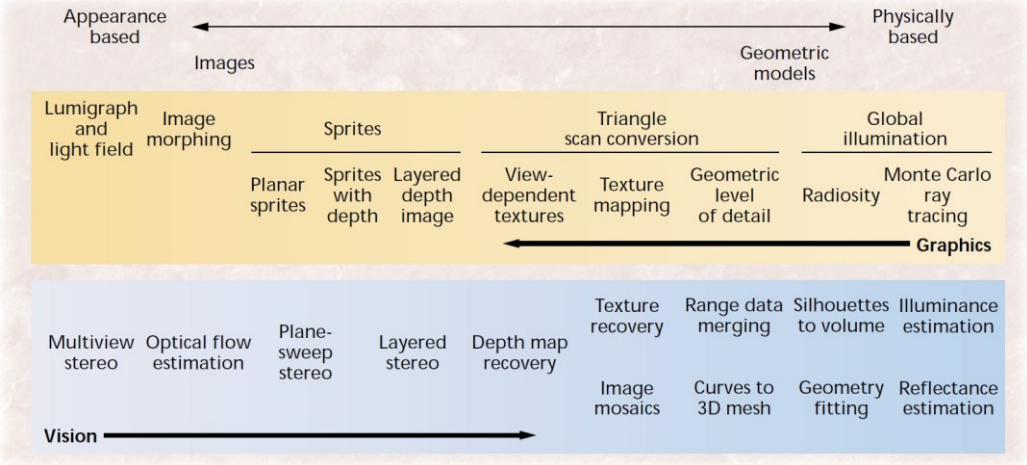


图 3 渲染谱 The Rendering Spectrum（Lengyel 1998 论文版本）

可以将渲染谱理解为渲染的金字塔。从左到右，由简单到复杂，由二维图像到几何模型，从外观特征到物理渲染。

9.2 固定视角的渲染 Fixed-View Rendering

固定视角的渲染（Fixed-View Rendering）技术，通过将复杂几何模型转换为可以在多帧中重复使用的一组简单的 buffer 来节省大量渲染时间与性能。

对于复杂的几何和着色模型，每帧去重新渲染整个场景很可能是昂贵的。可以通过限制观看者的移动能力来对渲染进行加速。 最严格的情况是相机固定在位置和方位，即根本不移动。而在这种情况下，很多渲染可以只需做一次。

例如，想象一个有栅栏的牧场作为静态场景，一匹马穿过它。牧场和栅栏渲染仅一次，存储其颜色和 Z 缓冲区。每帧将这些 buffer 复制到可显示的颜色和 Z 缓冲中。为了获得最终的渲染效果，马本身是需要渲染的。如果马在栅栏后面，存储和复制的 z 深度值将把马遮挡住。请注意，在这种情况下，马不能投下阴影，因为场景无法改变。可以进行进一步的处理，例如，可以确定出马影子的区域，根据需求进行处理。关键是对于要显示的图像的颜色何时或如何设置这点上，是没有限制的。固定视角的特效（Fixed-View Effects），可以通过将复杂几何模型转换为可以在多帧中重复使用的一组简单的 buffer 来节省大量时间。

在计算机辅助设计（CAD）应用程序中，所有建模对象都是静态的，并且在用户执行各种操作时，视图不会改变。一旦用户移动到所需的视图，就可以存储颜色和 Z 缓冲区，以便立即重新使用，然后每帧绘制用户界面和突出显示的元素。 这允许用户快速地注释，测量或以其他方式与复杂的静态模型交互。通过在 G 缓冲区中存储附加信息，类似于延迟着色的思路，可以稍后执行其他操作。 例如，三维绘画程序也可以通过存储给定视图的对象 ID，法线和纹理坐标来实现，并将用户的交互转换为纹理本身的变化。

一个和静态场景相关的概念是黄金线程(Golden Thread)或自适应（Adaptive Refinement）渲染。其基本思想是，当视点与场景不运动时，随着时间的推移，计算机可以生成越来越好的图像，而场景中的物体看起来会更加真实，这种高质量的绘制结果可以进行快速交换或混合到一系列画面中。这种技术对于 CAD 或其他可视化应用来说非常有用。而除此之外，还可以很多不同的精化方法。一种可能的方法是使用累积缓冲器（accumulation buffer）做抗锯齿（anti- aliasing），同时显示各种累积图像。另外一种可能的方法是放慢每像素离屏着色（如光线追踪，环境光遮蔽，辐射度）的速度，然后渐进改进之后的图像。

在 RTR3 书的 7.1 节介绍一个重要的原则，就是对于给定的视点和方向，给定入射光，无论这个光亮度如何计算或和隔生成这个光亮度的距离无关。眼睛没有检测距离，仅仅颜色。在现实世界中捕捉特定方向的光亮度可以通过简单地拍一张照片来完成。

QuickTime VR 是由苹果公司在 1995 年发布的 VR 领域的先驱产品，基本思路是用静态图片拼接成 360 度全景图。QuickTime VR 中的效果图像通常是由一组缝合在一起的照片，或者直接由全景图产生。随着相机方向的改变，图像的适当部分被检索、扭曲和显示。虽然这种方法仅限于单一位置，但与固定视图相比，这种技术具有身临其境的效果，因为观看者的头部可以随意转动和倾斜。

Kim，Hahn 和 Nielsen 提出了一种有效利用 GPU 的柱面全景图，而且通常，这种全景图也可以存储每个纹素的距离内容或其他值，以实现动态对象与环境的交互。

如下的三幅图，便是是基于他们思想的全景图（panorama），使用 QuickTime VR 来渲染出的全景视野范围。其中，第一幅是全景图原图，后两幅图是从中生成的某方向的视图。注意观察为什么这些基于柱面全景图的视图，没有发生扭曲的现象。

图 4 全景图原图

图 5 通过全景图得到的视图 1

图 6 通过全景图得到的视图 2

9.3 天空盒 Skyboxes

对于一些远离观众的物体，观众移动时几乎没有任何视差效果。换言之，如果你移动一米，甚至一千米，一座遥远的山本身看起来通常不会有明显的不同。当你移动时，它可能被附近的物体挡住视线，但是把那些物体移开，山本身看起来也依旧一样。天空盒就属于这种类型的物体。

图 7 基于天空盒渲染的场景 @mad max

环境贴图（environment map）可以代表本地空间入射光亮度。虽然环境贴图通常用于模拟反射，但它们也可以直接用来表示环绕环境的远处物体。任何独立于视图的环境地图表示都可以用于此目的；立方体贴图（cubic maps）是最为常见的一种环境贴图。环境贴图放置在围绕着观察者的网格上，并且足够大以包含场景中所有的对象。且网格的形状并不重要，但通常是立方体贴图。如下图，在该图所示的室内环境更像是一个 QuickTime VR 全景的无约束版本。观众可以在任何方向观察这个天空盒，得到很好的真实体验。但同样，任何移动都会破坏这个场景产生的真实感错觉，因为移动的时候，并不存在视差。

图 8 一个典型的立方体环境贴图

环境贴图通常可以包含相对靠近反射对象的对象。因为我们通常并没有多精确地去在乎反射的效果，所以这样的效果依然非常真实。而由于视差在直接观看时更加明显，因此天空盒通常只包含诸如太阳，天空，远处静止不动的云和山脉之类的元素。

图 9 玻璃球折射和反射效果的一个立方体环境贴图，这个 map 本身用可作天空盒。

为了使天空盒看起来效果不错，立方体贴图纹理分辨率必须足够，即每个屏幕像素的纹理像素。 必要分辨率的近似值公式：

$$\text{texture resolution} = \frac{\text{screen resolution}}{\tan(\text{fov}/2)},$$

其中，fov 表示视域。该公式可以从观察到立方体贴图的表面纹理必须覆盖 90 度的视域（水平和垂直）的角度推导出。并且应该尽可能隐藏好立方体的接缝处，最好是能做到无缝的衔接，使接缝不可见。一种解决接缝问题的方法是，使用六个稍微大一点的正方形，来形成一个立方体，这些正方形的每个边缘处彼此相互重叠，相互探出。这样，可以将邻近表面的样本复制到每个正方形的表面纹理中，并进行合理插值。

图 10 基于天空盒渲染的场景 @rage

9.4 光场渲染 Light Field Rendering

所谓光场（Light Field），可以理解为空间中任意点发出的任意方向的光的集合。

而光场渲染（Light Field Rendering），可以理解为在不需要图像的深度信息或相关性的条件下，通过相机阵列或由一个相机按设计的路径移动，把场景拍摄下来作为输出图像集。对于

任意给定的新视点，找出该视点邻近的几个采样点进行简单的重新采样和插值，就能得到该视点处的视图。

magic leap 公司目前的原型产品，Nvidia 公司的 near-eye light field display，Lytro 公司发布的光场相机，都是基于 Light Field 技术。

图 11 Lytro 公司的光场相机

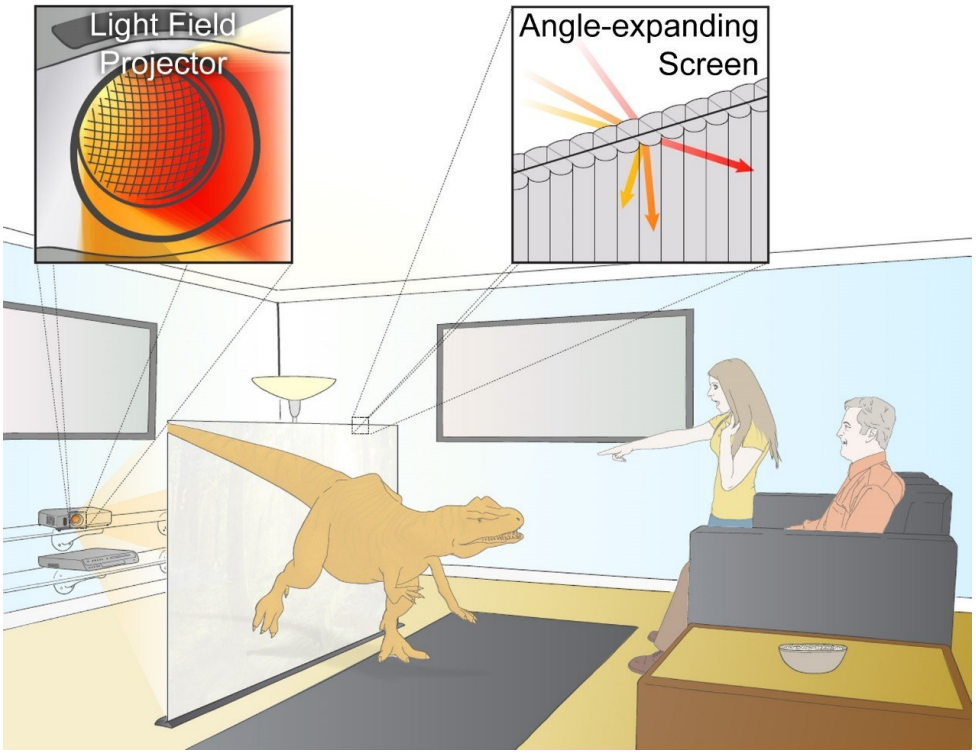


图 12 SIGGRAPH 2014 会议上，MIT’s Camera CultureGroup 介绍了一种基于开普勒望远镜中投影机和光学技术的无眼镜 3D 的新方法。 他们提出的“压缩光场投影（Compressive Light Field Projection）”新方法由单个设备组成，并没有机械移动的物件。

9.5 精灵与层 Sprites and Layers

最基本的基于图像的渲染的图元之一便是精灵（sprite）。精灵（sprite）是在屏幕上移动的图像，例如鼠标光标。精灵不必具有矩形形状，而且一些像素可以以透明形式呈现。对于简单的精灵，屏幕上会显示一个一对一的像素映射。存储在精灵中的每个像素将被放在屏幕上的像素中。可以通过显示一系列不同的精灵来生成动画。

图 13 基于 Sprite 层级制作的《雷曼大冒险》@UBISOFT

更一般的精灵类型是将其渲染为应用于总是面向观看者的多边形的图像纹理。图像的 Alpha 通道可以为 sprite 的各种像素提供全部或部分透明度。这种类型的精灵可以有一个深度，所以在场景本身，可以顺利地改变大小和形状。一组精灵也可以用来表示来自不同视图的对象。对于大型物体，这种用精灵来替换的表现效果会相当弱，因为从一个精灵切换到另一个时，会很容易穿帮。也就是说，如果对象的方向和视图没有显着变化，则给定视图中的对象的图像表示可以对多个帧有效。而如果对象在屏幕上足够小，存储大量视图，即使是动画对象也是可行的策略。

考虑场景的一种方法是将其看作一系列的层（layers），而这种思想也通常用于二维单元动画。每个精灵层具有与之相关联的深度。通过这种从前到后的渲染顺序，我们可以渲染出整个场景而无需 Z 缓冲区，从而节省时间和资源。

图 14 基于 Sprite 层级制作的《雷曼大冒险》@UBISOFT

9.6 公告板 Billboarding

我们将根据观察方向来确定多边形面朝方向的技术叫做公告板（Billboarding，也常译作布告板）。而随着观察角度的变化，公告板多边形的方向也会根据需求随之改变。与 alpha 纹理和动画技术相结合，可以用公告板技术表示很多许多不具有平滑实体表面的现象，比如烟，火，雾，爆炸效果，能量盾（Energy Shields），水蒸气痕迹，以及云朵等。如下文中贴图的，基于公告板渲染出的云朵。

图 15 一棵由公告板技术渲染出的树木

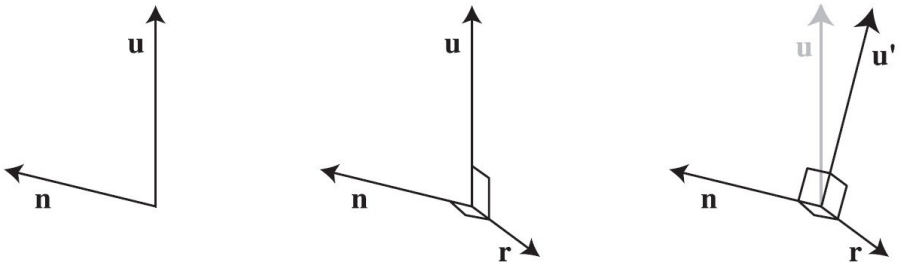


图 16 给定表面的法线向量 n 和近似向上方向的向量 u，通过创建一组由三个相互垂直的向量，就可以确定公告板的方向。其中，左图是互相垂直的 u 和 n。中图是 r 向量通过 u 和 n 的叉乘得到，因此同时垂直于 u 和 n，而在右图中，对固定向量 n 和 r 进行叉乘就可以得到与他们都垂直的的向上向量 u’

有三种不同类型的 Billboard，分别是：

- Screen-Aligned Billboard 对齐于屏幕的公告板
- World-Oriented Billboard 面向世界的公告板
- Axial Billboard 轴向公告板

其中：

- Screen-Aligned Billboard 的 n 是镜头视平面法线的逆方向,u 是镜头的 up。
- Axial Billboard 的 u 是受限制的 Axial, r = u* n,(n 是镜头视平面法线的逆方向,或,视线方向的逆方向),最后再计算一次 n' = r * u,即 n'才是最后可行的代入 M 的 n,表达了受限的概念。
- World-oriented billboard 就不能直接使用镜头的 up 做 up,因为镜头旋转了,并且所画的 billboard 原本是应该相对世界站立的,按 Screen-Aligned 的做法就会随镜头旋转,所以此时应该 r = u * n(u 是其在世界上的 up,n 是镜头视线方向的逆方向),最后再计算一次 u = r * n,即 u'才是最后的 up,即非物体本身相对世界的 up,亦非镜头的 up。

所以公告板技术是一种看似简单其实较为复杂的技术,它的实现变种较多。归其根本在于：

- View Oriented / View plane oriented 的不同
- Sphere/ Axial 的不同
- Cameraup / World up 的不同

如 View Oriented 和 View plane oriented 的不同，得到的公告板效果就完全不同：

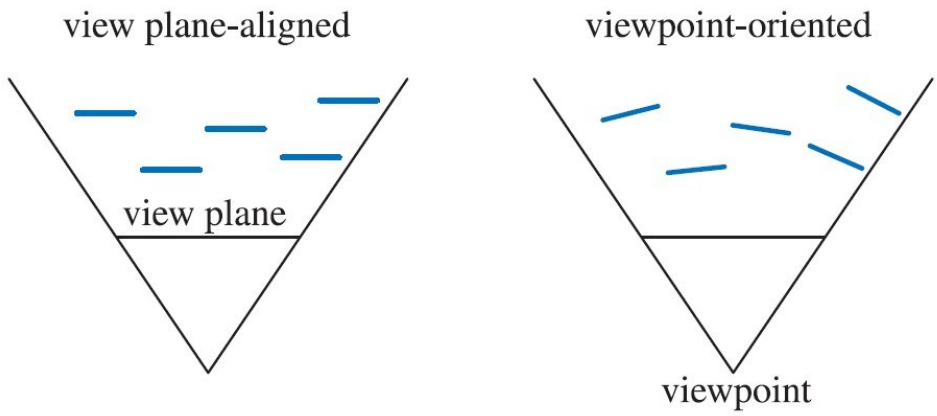


图 17 两种公告板对其技术的顶视图，左图为 view plane-aligned（视图平面对齐），右图为 viewpoint-oriented（视点对齐），其面向的方向根据算法的不同而有所不同。

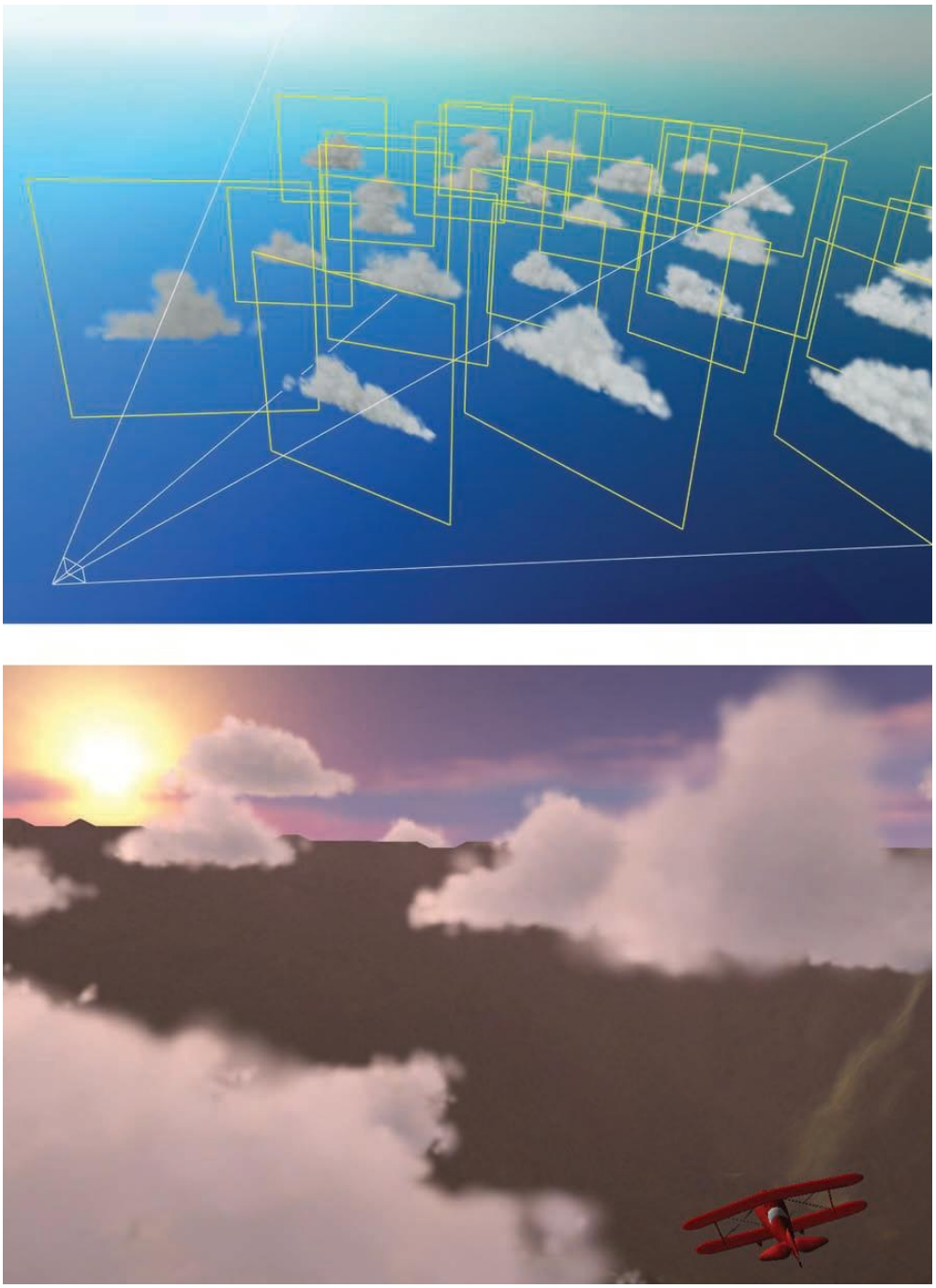


图 18 使用 world-oriented Billboard 创建的云层

在 Unreal 4 Engine 中，使用 Axial Billboard 作为树木 LOD 中的一级的一些图示 ：

图 19 使用 Axialbillboard 作为树木 LOD 中的一级 @Unreal 4 Engine

图 20 使用 Axialbillboard 作为树木 LOD 中的一级 @Unreal 4 Engine

9.7 粒子系统 Particle System

粒子系统（Particle System）是一组分散的微小物体集合，其中这些微小物体按照某种算法运动。粒子系统的实际运用包括模拟火焰，烟，爆炸，流水，树木，瀑布，泡沫，旋转星系和其他的一些自然现象。粒子系统并不是一种渲染形式，而是一种动画方法，这种方法的思想是值粒子的生命周期内控制他们的产生，运动，变化和消失。

可以用一条线段表示一个实例，另外，也可以使用轴向公告板配合粒子系统，显示较粗的线条。

除了爆炸，瀑布，泡沫以及其他现象以外，还可以使用粒子系统进行渲染。例如，可以使用粒子系统来创建树木模型，也就是表示树木的几何形状，当视点距离模型较近时，就会产生更多的粒子来生成逼真的视觉效果。

以下是一幅用粒子系统渲染树木的示例：

图 21 基于粒子系统渲染的树木

9.8 替代物 Impostors

作为一种公告板技术，替代物（Impostors）是通过从当前视点将一个复杂物绘制到一幅图像纹理上来创建的，其中的图像纹理用于映射到公告板上，渲染过程与替代物在屏幕上覆盖的像素点数成正比，而不是与顶点数或者物体的复杂程度成正比。替代物可以用于物体的一些实例上或者渲染过程的多帧上，从而使整体性能获得提升。

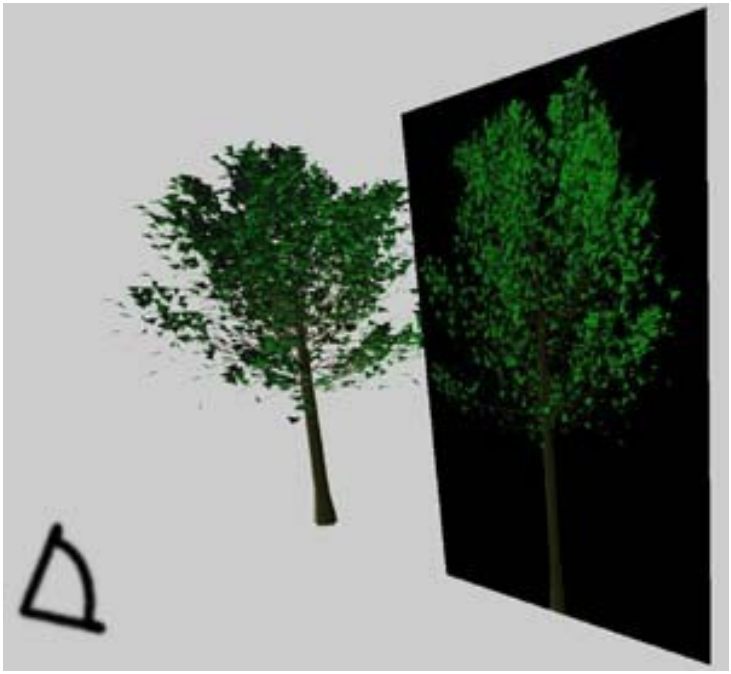


图 22 一幅树的视图和一个 Impostors（Impostors 的黑色背景是透明通道，在渲染时可以处理）

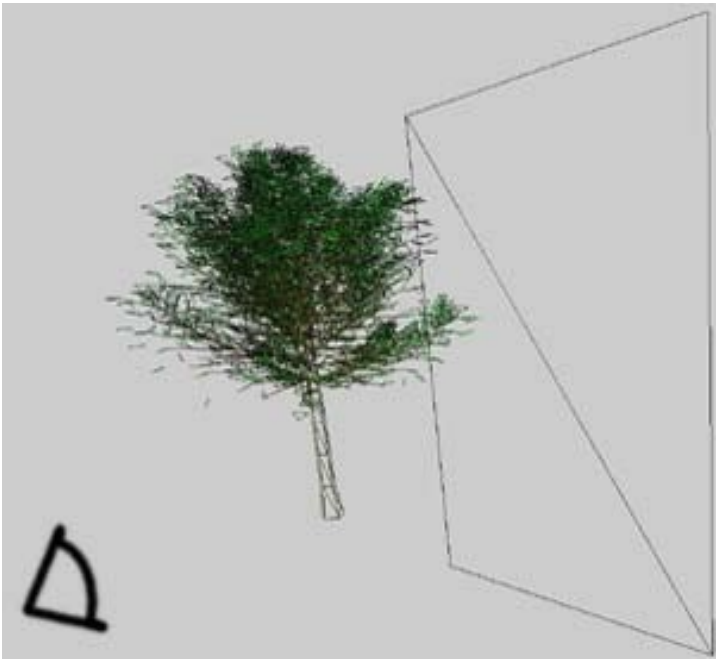


图 23 一幅相同的树和 Impostors 的线框视图

另外，Impostors 和 Billboard 的纹理还可以结合深度信息（如使用深度纹理和高度纹理）进行增强。如果对 Impostors 和 Billboard 增加一个深度分量，就会得到一个称为深度精灵（depth sprite）或者 nailboard（常被译作钉板）的相关绘制图元。也可以对 Impostors 和 Billboard 的纹理做浮雕纹理映射（relief texture mapping）。

关于 Impostors，一篇很好的文章是 William Damon 的《Impostors Made Easy》，有进一步了解兴趣的朋友可以进行延伸阅读：

https://software.intel.com/en-us/articles/impostors-made-easy

9.9 公告板云 Billboard Clouds

使用 Imposters 的一个问题是渲染的图像必须持续地面向观察者。如果远处的物体正在改变方向，则必须重新计算 Imposters 的朝向。而为了模拟更像他们所代表的三角形网格的远处物体，D´ecoret 等人提出了公告板云（Billboard Clouds）的想法，即一个复杂的模型通常可以通过一系列的公告板集合相互交叉重叠进行表示。我们知道，一个真实物体可以用一个纸模型进行模拟，而公告板云可以比纸模型更令人信服，比如公告板云可以添加一些额外的信息，如法线贴图、位移贴图和不同的表面材质。另外，裂纹沿裂纹面上的投影也可以由公告板进行处理。而 D´ecoret 等人也提出了一种在给定误差容限内对给定模型进行自动查找和拟合平面的方法。

如下是在 UNIGINE Engine（注意这不是虚幻引擎，经常会被看错）中基于 Billboard Clouds 技术创建云层效果的一个示例：

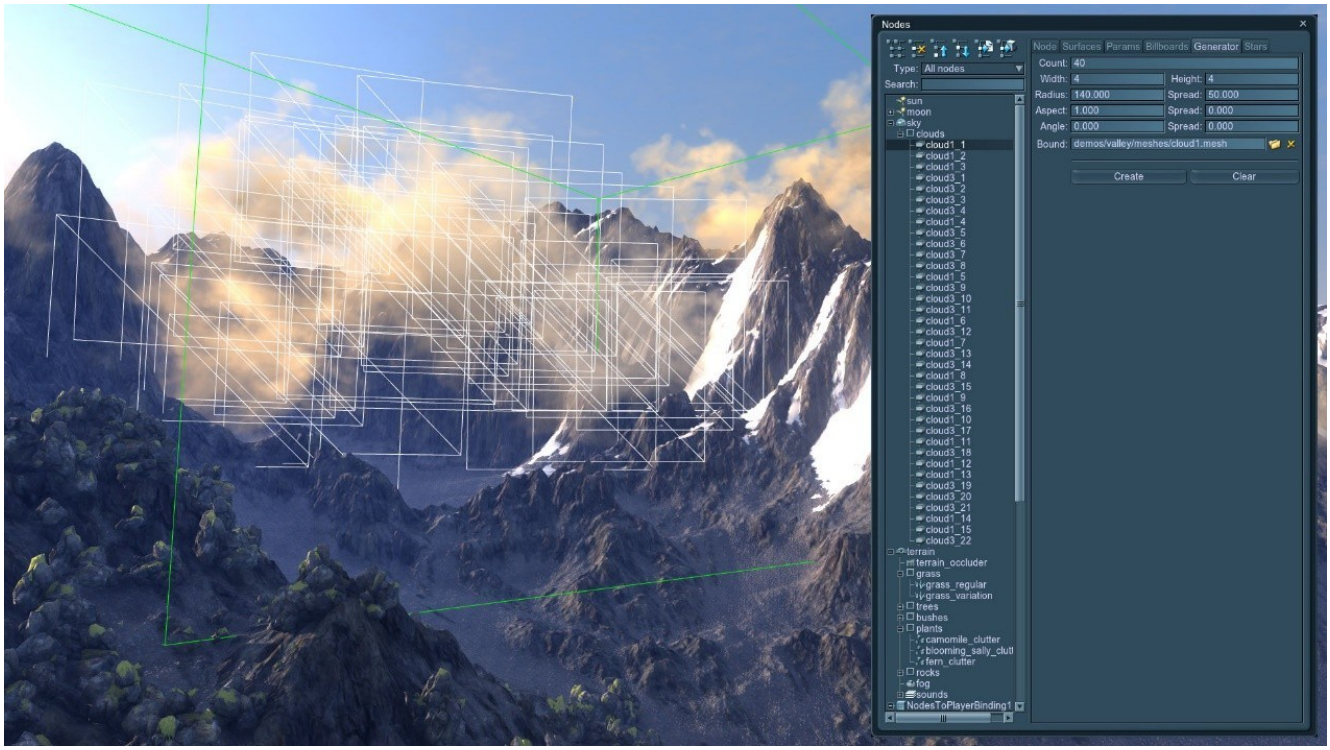


图 24 Billboard Clouds 技术创建云层示例图 @UNIGINE Engine

图 25 Billboard Clouds 技术创建云层的最终效果图 @UNIGINE Engine

9.10 图像处理 Image Processing

图像处理的过程，一般在像素着色器中进行，因为在像素着色器中，可以很好地将渲染过程和纹理结合起来，而且在 GPU 上跑像素着色器，速度和性能都可以满足一般所需。

一般而言，首先需要将场景渲染成 2D 纹理或者其他图像的形式，再进行图像处理，这里的图像处理，往往指的是后处理（post effects）。而下文将介绍到的颜色校正（Color Correction）、色调映射（Tone Mapping）、镜头眩光和泛光（Lens Flare and Bloom）、景深（Depth of Field）、运动模糊（Motion Blur），一般而言都是后处理效果。

图 26 使用像素着色器进行图像处理。左上为原始图像；右上角为高斯差分操作的图像，左下为经过边缘检测的图像，右下为边缘检测与原图像的混合。

9.11 颜色校正 Color Correction

色彩校正(Color correction)是使用一些规则来转化给定的现有图像的每像素颜色到其他颜色的一个过程。颜色校正有很多目的，例如模仿特定类型的电影胶片的色调，在元素之间提供一致的外观，或描绘一种特定的情绪或风格。一般而言，通过颜色校正，游戏画面会获得更好的表现效果。

图 27 左图是准备进行颜色校正的原图。右图是通过降低亮度，使用卷积纹理（Volume Texture），得到的夜间效果。@Valve

颜色校正通常包括将单个像素的 RGB 值作为输入，并向其应用算法来生成一个新的 RGB。颜色校正的另一个用途是加速视频解码，如 YUV 色彩空间到 RGB 色彩空间的转换。基于屏幕位置或相邻像素的更复杂的功能也可行，但是大多数操作都是使用每像素的颜色作为唯一的输入。

对于一个计算量很少的简单转换，如亮度的调整，可以直接在像素着色器程序中基于一些公式进行计算，应用于填充屏幕的矩形。

而评估复杂函数的通常方法是使用查找表（Look-Up Table，LUT）。由于从内存中提取数值经常要比复杂的计算速度快很多，所以使用查找表进行颜色校正操作，速度提升是很显著的。

图 28 原图和经过色彩校正后的几幅效果图 @Unreal 4 Engine

图 29 原图和经过颜色校正的效果图 @Crysis

9.12 色调映射 Tone Mapping

计算机屏幕具有特定的亮度范围，而真实图像具有更巨大的亮度范围。色调映射（Tonemapping），也称为色调复制（tone reproduction），便是将宽范围的照明级别拟合到屏幕有限色域内的过程。色调映射与表示高动态范围的 HDR 和 HDRI 密切相关：

- HDR，是 High-Dynamic Range（高动态范围）的缩写，可以理解为一个 CG 的概念，常出现在计算机图形学与电影、摄影领域中。
- HDRI 是 High-Dynamic Range Image 的缩写，即 HDR 图像，高动态范围图像。
- 而实际过程中，HDR 和 HDRI 两者经常会被混用，都当做高动态范围成像的概念使用，这也是被大众广泛接受的。

本质上来讲，色调映射要解决的问题是进行大幅度的对比度衰减将场景亮度变换到可以显示的范围，同时要保持图像细节与颜色等表现原始场景的重要信息。

根据应用的不同，色调映射的目标可以有不同的表述。在有些场合，生成“好看”的图像是主要目的，而在其它一些场合可能会强调生成尽可能多的细节或者最大的图像对比度。在实际的渲染应用中可能是要在真实场景与显示图像中达到匹配，尽管显示设备可能并不能够显示整个的亮度范围。

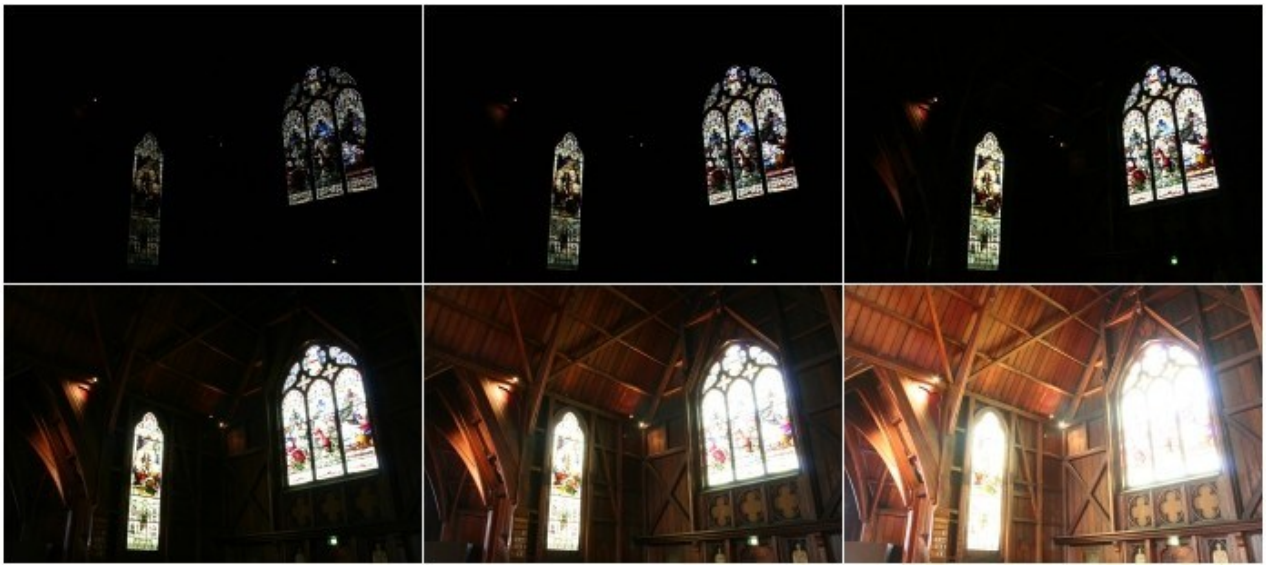


图 30 经过色调映射得到的高动态范围图像 @新西兰惠灵顿圣保罗教堂

9.13 镜头眩光和泛光 Lens Flare and Bloom

镜头眩光（Lens flare）是由于眼睛的晶状体或者相机的透镜直接面对强光所产生的一种现象，由一圈光晕（halo）和纤毛状的光环（ciliary corona）组成。光晕的出现是因为透镜物质（如三棱镜）对不同波长光线折射数量的不过而造成的，看上去很像是光周围的一个圆环，外圈是红色，内圈是紫红色。纤毛状的光环源于透镜的密度波动，看起来像是从一个点发射出来的光线。Lens flare 是近来较为流行的一种图像效果，自从我们认识到它是一种实现真实感效果的技术后，计算机便开始模拟此效果。

图 31 镜头眩光效果 @Watch Dogs

泛光（Bloom）效果，是由于眼睛晶状体和其他部分的散光而产生，在光源附近出现的一种辉光。在现实世界中，透镜无法完美聚焦是泛光效果的物理成因；理想透镜也会在成像时由于衍射而产生一种名为艾里斑的光斑。

常见的一个误解便是将 HDR 和 Bloom 效果混为一谈。Bloom 可以模拟出 HDR 的效果，但是原理上和 HDR 相差甚远。HDR 实际上是通过映射技术，来达到整体调整全局亮度属性的，这种调整是颜色，强度等都可以进行调整，而 Bloom 仅仅是能够将光照范围调高达到过饱和，也就是让亮的地方更亮。不过 Bloom 效果实现起来简单，性能消耗也小，却也可以达到不错的效果。

图 32 Bloom 效果 @ Battlefield3

图 33 《Battlefield 3》中的渲染效果，同时包含镜头眩光（Lens flare），泛光（Bloom）和 Dirty Lens

9.14 景深 Depth of Field

在光学领域，特别是摄影摄像领域，景深（Depth of field，DOF），也叫焦点范围（focus range）或有效焦距范围（effective focus），是指场景中最近和最远的物体之间出现的可接受的清晰图像的距离。换言之，景深是指相机对焦点前后相对清晰的成像范围。在相机聚焦完成后，在焦点前后的范围内都能形成清晰的像，这一前一后的距离范围，便叫做景深。

图 34 摄影中典型的景深效果

虽然透镜只能够将光聚到某一固定的距离，远离此点则会逐渐模糊，但是在某一段特定的距离内，影像模糊的程度是肉眼无法察觉的，这段距离称之为景深。当焦点设在超焦距处时，景深会从超焦距的一半延伸到无限远，对一个固定的光圈值来说，这是最大的景深。

景深通常由物距、镜头焦距，以及镜头的光圈值所决定（相对于焦距的光圈大小）。除了在近距离时，一般来说景深是由物体的放大率以及透镜的光圈值决定。固定光圈值时，增加放大率，不论是更靠近拍摄物或是使用长焦距的镜头，都会减少景深的距离；减少放大率时，则会增加景深。如果固定放大率时，增加光圈值（缩小光圈）则会增加景深；减小光圈值（增大光圈）则会减少景深。

景深的效果在计算机图形学中应用广泛，电影，游戏里面经常会利用景深特效来强调画面重点。相应的，已经有了很多成熟的算法在不同的渲染方法，而光栅化可以很高效的实现现有的景深算法。

图 35 景深效果 @Battlefield 4

图 36 景深效果 @ Witcher 2

9.15 运动模糊 Motion Blur

现实世界中，运动模糊（Motion Blur，或译为动态模糊)，是因为相机或者摄影机的快门时间内物体的相对运动产生的。在快门打开到关上的过程中，感光材料因为受到的是物体反射光

持续的照射成像。即在曝光的这个微小时间段内，对象依然在画面中移动，感光材料便会记录下这段时间内物体运动的轨迹，产生运动模糊。

我们经常在电影中看到这种模糊，并认为它是正常的，所以我们期望也可以在电子游戏中看到它，以带给游戏更多的真实感。

若无运动模糊，一般情况下，快速移动的物体会出现抖动，在帧之间的多个像素跳跃。这可以被认为是一种锯齿，但可以理解为基于时间的锯齿，而不是基于空间的锯齿。在这个意义上，运动模糊可以理解为是一种时间意义上的抗锯齿。

正如更高的显示分辨率可以减少但不能消除锯齿，提高帧速率并不能消除运动模糊的需要。而视频游戏的特点是摄像机和物体的快速运动，所以运动模糊可以大大改善画面的视觉效果。而事实表明，带运动模糊的 30 FPS 画面，通常看起来比没有带运动模糊的 60 FPS 画面更出色。

图 37 Motion Blur 效果 @GTA5

在计算机绘制中产生运动模糊的方法有很多种。一个简单但有限的方法是建模和渲染模糊本身。

实现运动模糊的方法大致分 3 种：

1、直接渲染模糊本身。通过在对象移动之前和之后添加几何体来完成，并通过次序无关的透明，避免 Alpha 混合。

2、基于累积缓冲区（accumulationbuffer），通过平均一系列图像来创建模糊。

3、基于速度缓冲器（velocity buffer）。目前这个方法最为主流。创建此缓冲区，需插入模型三角形中每个顶点的屏幕空间速度。通过将两个建模矩阵应用于模型来计算速度，一个用于

最后一个帧，一个用于当前模型。顶点着色器程序计算位置的差异，并将该向量转换为相对的屏幕空间坐标。图 10.34 显示了速度缓冲器及其结果。

图 38 Motion Blur 效果 @Battlefield4

运动模糊对于由摄像机运动而变得模糊的静态物体来说比较简单，因为往往这种情况下不需要速度缓冲区。如果需要的是摄像机移动时的运动感，可以使用诸如径向模糊（radial blur）之类的固定效果。如下图。

图 39 径向模糊可以增强运动感 @《刺客信条》Ubisoft

9.16 体渲染 Volume Rendering

体渲染（Volume Rendering），又称立体渲染，体绘制，是一种用于显示离散三维采样数据集的二维投影的技术。体渲染技术中的渲染数据一般用体素（Volumeric Pixel，或 Voxel）来表示，每个体素表示一个规则空间体。例如，要生成人头部的医学诊断图像（如 CT 或 MRI），同时生成 256 x256 个体素的数据集合，每个位置拥有一个或者多个值，则可以将其看做三维图像。因此，体渲染也是基于图像的渲染技术中的一种。

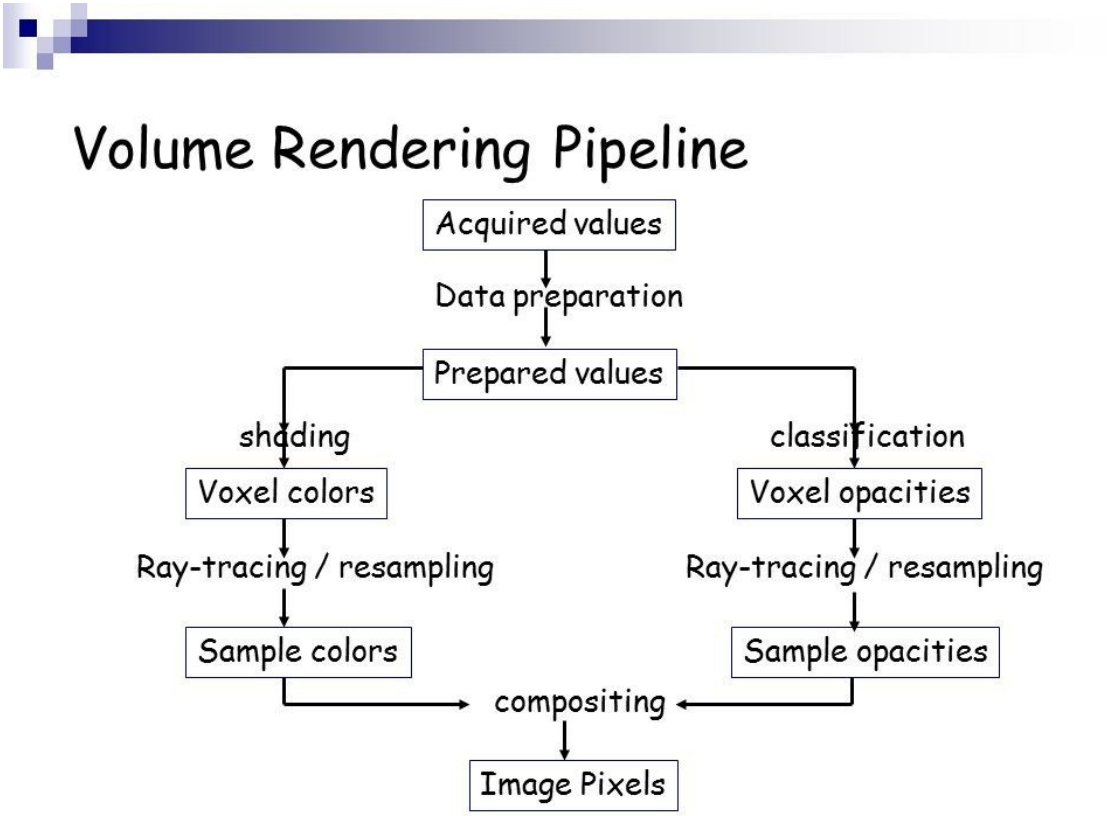


图 40 一个典型的体渲染 Pipeline

体渲染技术流派众多，常见的流派有：

- 体光线投射 Volume ray casting
- 油彩飞溅技术 Splatting
- 剪切变形技术 Shear warp
- 基于纹理的体绘制 Texture-based volume rendering
- 等。

图 41 基于 Splatting 和 voxel 在 Unreal 4 中进行的体渲染

图 42 Volume Cloud（体积云）效果 @Unity 5

图 43 Volume Fog（体积雾）效果 @CRY ENGINE 3

9.17 Reference

[1] https://udk-legacy.unrealengine.com/udk/Three/ColorGrading.html

[2] http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=DD633186A6579497B8B1434252979C80?doi=10.1.1.164.7222&rep=rep1&type=pdf

[3] http://blendermama.com/precise-distribution-of-trees-using-particles.html

[4] https://software.intel.com/en-us/articles/impostors-made-easy

[5] http://lightfield-forum.com/2014/05/mit-compressive-light-field-projection-system-for-new-glasses-free-3d-displays/

[6] https://bartwronski.com/2014/04/07/bokeh-depth-of-field-going-insane-part-1/

[7] http://blog.csdn.net/silangquan/article/details/17148757

[8] https://en.wikipedia.org/wiki/Skybox_(video_games)

[9] 胡孔明, 于瀛洁, 张之江. 基于光场的渲染技术研究[J].微计算机应用, 2008, 29(2): 22-27.

[10] https://forums.unrealengine.com/community/community-content-tools-and-tutorials/82804-free-trees-library

[11] http://unigine.com/cn/articles/procedural-content-generation2

[12] https://en.wikipedia.org/wiki/Tone_mapping

[13] https://en.wikipedia.org/wiki/Volume_rendering

第十章 非真实感渲染(NPR)相关技术总结

本章将带来 RTR3 第十一章内容“Chapter 11 Non-Photorealistic Rendering”的总结、概括与提炼。

与传统的追求照片真实感的真实感渲染不同，非真实感渲染（Non-Photorealistic Rendering，NPR）旨在模拟艺术式的绘制风格，常用来对绘画风格和自然媒体（如铅笔、钢笔、墨水、木炭、水彩画等）进行模拟。而卡通渲染（Toon Rendering）作为一种特殊形式的非真实感渲染方法，近年来倍受关注。

通过阅读这篇文章，你将对非真实感渲染技术的以下要点有所了解：

- 非真实感渲染的基本思想和相关领域
- 卡通渲染

- 轮廓描边的几种实现流派
- 1）基于视点方向的描边
- 2）基于过程几何方法的描边
- 3）基于图像处理生成的描边
- 4）基于轮廓边缘检测的描边
- 5）混和轮廓描边
- 其他风格的 NPR 渲染技术
- 1）纹理调色板（Palette of Textures）
- 2）色调艺术图（Tonal Art Maps，TAM）
- 3）嫁接（Graftals）
- 水彩风格的 NPR

10.1 非真实感渲染

正如变化的字体会给人不一样的感觉，不同的渲染风格会带给人们不同的心情，感受与意境。

非真实感渲染（Non-Photorealistic Rendering，NPR）， 亦被称为风格化渲染（Stylistic Rendering），是致力于为数字艺术提供多种表达方式的一种渲染流派。与传统的追求照片真实感的真实感渲染（Photorealistic Rendering）计算机图形学不同，非真实感渲染旨在模拟艺术式的绘制风格，也用于尝试新的绘制风格。

图 1 真实感渲染 vs. 非真实感渲染 @ruben3d

NPR 的目的之一就是创建类似技术示意图、技术图纸相关的图像，而另一个应用领域便是对绘画风格和自然媒体（如铅笔、钢笔、墨水、木炭、水彩画等）进行模拟。这是一个涉及内容非常之多的应用领域，为了捕捉各种媒体的真实效果，人们已经提出了各种不同的算法。

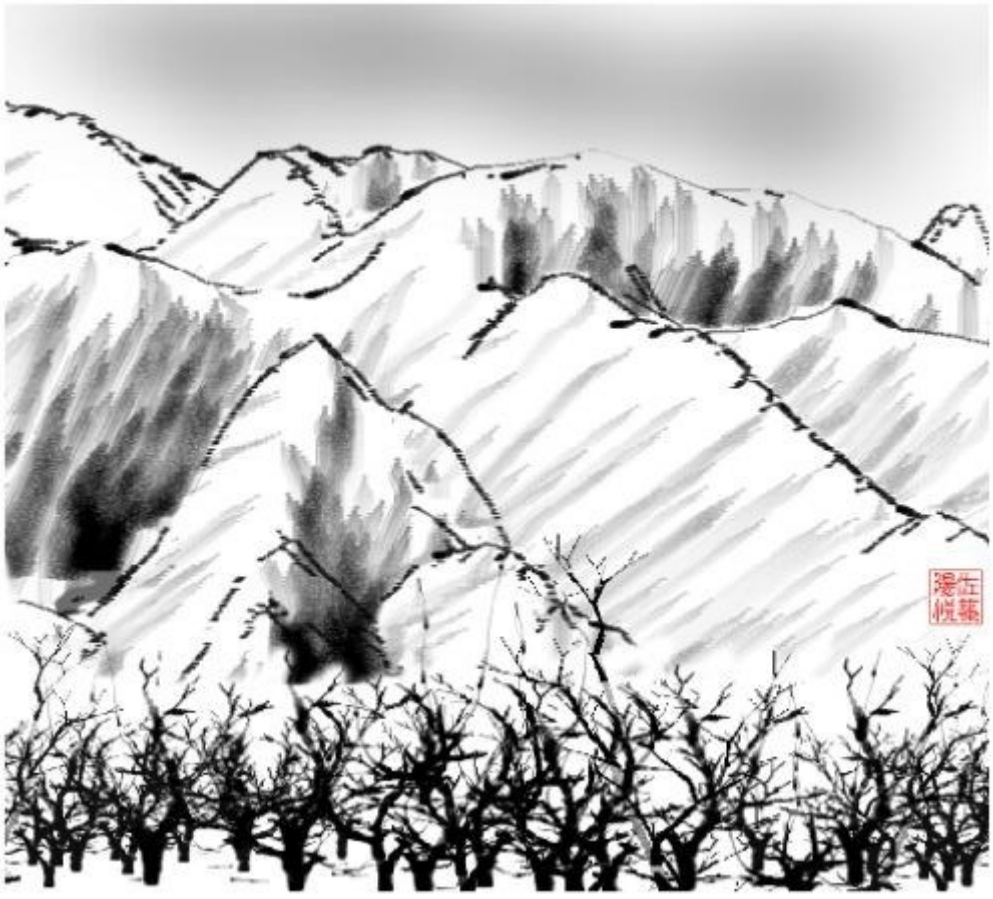


图 2 基于 NPR 渲染出的水墨画 @suiboku

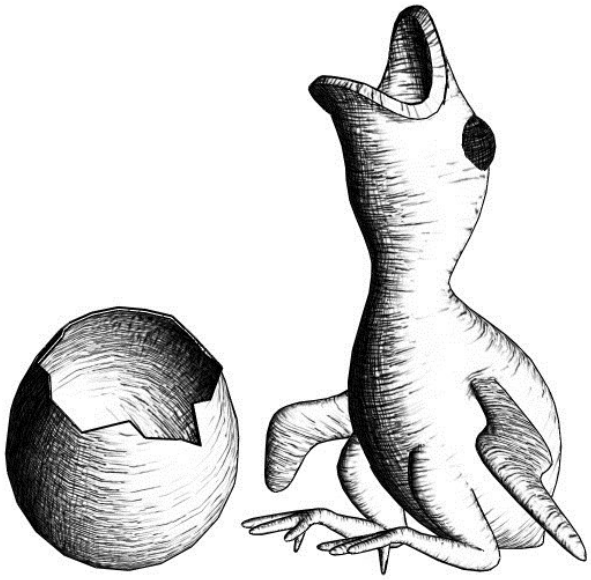


图 3 基于 NPR 渲染出的铅笔素描@ Real–Time Hatching . SIGGRAPH 2001

非真实感渲染与我们并不遥远，它早以“卡通着色（Toon Shading）”的形式出现在各式动漫和电影中。

图 4 基于卡通着色的 2016 年高分动漫电影《你的名字》

在游戏制作方面，各种涉及到非真实感渲染的作品数不胜数，《Ōkami(大神)》，《The Legend of Zelda（塞尔达传说）》系列，甚至到现在的《Dota2》、《英雄联盟》、《守望先锋》，都多多少少涉及到了 NPR。

图 5 非真实感渲染风格强烈的《塞尔达传说：荒野之息》

图 6 非真实感渲染风格强烈的《塞尔达传说：荒野之息》

10.2 卡通渲染

上文提到，一直以来，有一种特殊形式的 NPR 倍受关注，且和我们的生活息息相关，那就是卡通渲染（Toon Rendering，又称 Cel Rendering）。这种渲染风格能够给人以独特的感染力与童趣。

这种风格很受欢迎的原因之一是 McCloud 的经典著作《Understanding Comics》中所讲述到的“通过简化进行增强（Amplification Through Simplification）”。通过简化并剔除所包含的混杂部分，可以突出于主题相关的信息，而大部分观众都会认同那些用简单风格描绘出来的卡通形象。

在计算机图形学领域，大约在 20 世纪 90 年代就开始使用 toon 渲染风格来实现三维模型和二维 cel 动画之间的结合。而且和其他 NPR 风格相比，这种绘制方法比较简单，可以很容易地利用计算机进行自动生成。

可以将最卡通着色基本的三个要素概括为：

- 锐利的阴影（Sharp shadows）
- 少有或没有高亮的点（Little or no highlight）
- 对物体轮廓进行描边（Outline around objects）

关于 toon 渲染，有很多不同的实现方法。

- 对于含有纹理但没有光照的模型来说，可以通过对纹理进行量化来近似具有实心填充颜色的卡通风格。
- 对于明暗处理，有两种最为常见的方法，一种是用实心颜色填充多边形区域。但这种方式实用价值不大。另一种是使用 2-tone 方法来表示光照效果和阴影区域。也称为硬着色方法（Hard Shading），可以通过将传统光照方程元素重新映射到不同的调色板上来实现。此外，一般用黑色来绘制图形的轮廓，可以达到增强卡通视觉效果的目的。

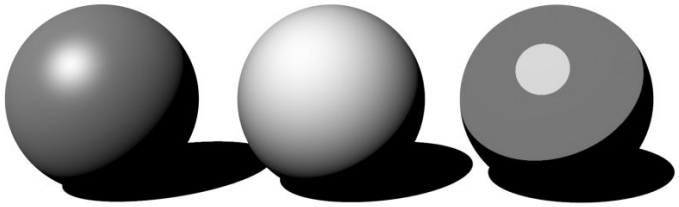


图 7 真实感光照模型和卡通着色模型

具体的着色方法，可以理解为在 Fragment shader 中测试每个像素漫反射 diffuse 中的 NdotL 值，让漫反射形成一个阶梯函数，不同的 NdotL 区域对应不同的颜色。下图显示了不同的漫反射强度值的着色部分阶梯指定了不同的像素颜色。

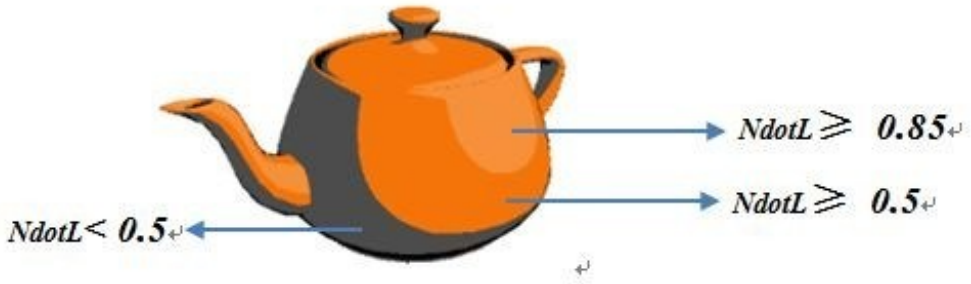


图 8 不同的漫反射强度值的着色部分阶梯指定不同的像素颜色

图 9 不同的卡通着色细节效果

10.3 轮廓描边的渲染方法小结

轮廓描边的渲染方法可以分为以下五种：

1）基于视点方向的描边

2）基于过程几何方法的描边

3）基于图像处理的描边

4）基于轮廓边缘检测的描边

5）混和轮廓描边

下面分别进行介绍。

10.3.1 基于视点方向的描边

基于视点方向的描边方法，即表面角描边（Surface Angle Silhouetting），其基本思想是使用视点方向（view point）和表面法线（surface normal）之间的点乘结果得到轮廓线信息。如果此点乘结果接近于零，那么可以断定这个表面极大概率是侧向（Edge-on）的视线方向，而我们就将其视做轮廓边缘，进行描边。

这种方法相当于用一个边缘为黑色圆环的环境贴图（Environment Map），对物体表面进行着色处理，如图所示。

图 10 使用球形图来绘制边缘轮廓，如果沿着球形图的边缘对圆进行加宽，即可产生较粗的轮廓

在实际应用中，通常使用一张一维纹理（一般我们称其为 ramp 图）来代替环贴图。也就是使用视角方向与顶点法向的点乘对该纹理进行采样。

需要注意，这种技术仅适用于一些特定的模型，这些模型必须保证法线与轮廓边缘之间存在一定关系。诸如立方体这样的模型，此方法并不太适用，因为往往无法得到轮廓边缘。但我们可以通过显式地绘制出折缝边缘，来正确地表现出这类比较明显的特征。

10.3.2 基于过程几何方法的描边

基于过程几何方法生成的描边，即过程几何描边（Procedural Geometry Silhouetting），基本思想是先渲染正向表面（frontfaces），再渲染背向表面（backfaces），从而使得轮廓边缘可见，达到描边的目的。

有多种方法用来渲染背向表面，且各有优缺点。但它们都是先渲染正向表面，然后打开正向表面裁剪（culling）开关，同时关闭背向裁剪开关。这样这个 pass 中的渲染结果便只会显示出背向表面。

一种基于过程几何方法生成的描边的方法是仅仅渲染出背向表面的边界线（而不是面），使用偏置（Biasing）或者其他技术来确保这些线条恰好位于正向表面之前。这样就可以将除轮

廓边缘之外的其他所有线条全部隐藏起来。这种方法非常适合单像素宽的线条，但如果线条的宽度超过这个值，那么通常会出现无法连接独立线段的情况，从而造成明显的缝隙。

另一种渲染较宽描边线条的方法是直接将背面表面本身渲染成黑色。但没有任何偏置操作，背向表面就会保持不可见，所以需要做的就是通过偏置将这些背向表面沿屏幕 Z 方向向前移动，这样，便只有背向表面的三角形边缘是可见的。

如下图，可以使用背向表面的斜率对对多边形进行向前偏置，但是线条宽度依然依赖于正向表面的角度。

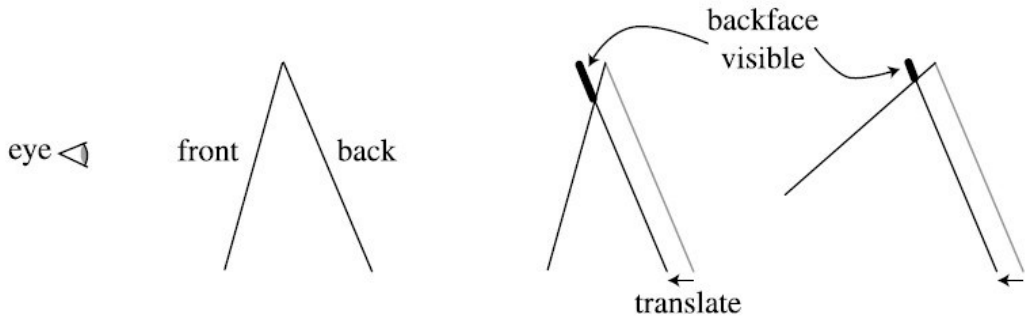


图 11 描边的 z 偏置方法，可以通过对背向表面进行向前平移来实现。如果正向表面的角度不同，那么背向表面的可见量也不同。

10.3.3 基于图像处理的描边

基于图像处理生成轮廓描边（Silhouetting by Image Processing），即通过在各种缓冲区上执行图像处理技术，来实现非真实渲染的方法。可以将其理解为一种后处理操作。通过寻找相邻 Z 缓冲数值的不连续性，就可以确定大多数轮廓线的位置。同样，借助邻接表面法线向量的不连续性，可以确定出分界线（往往也是轮廓线）边缘的位置。此外，利用环境色对场景进行绘制，也可以用来检测前两种方法可能会漏掉的边缘。

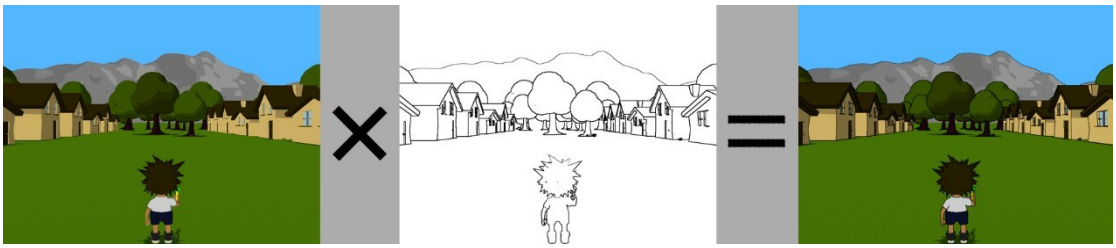


图 12 基于图像处理的描边

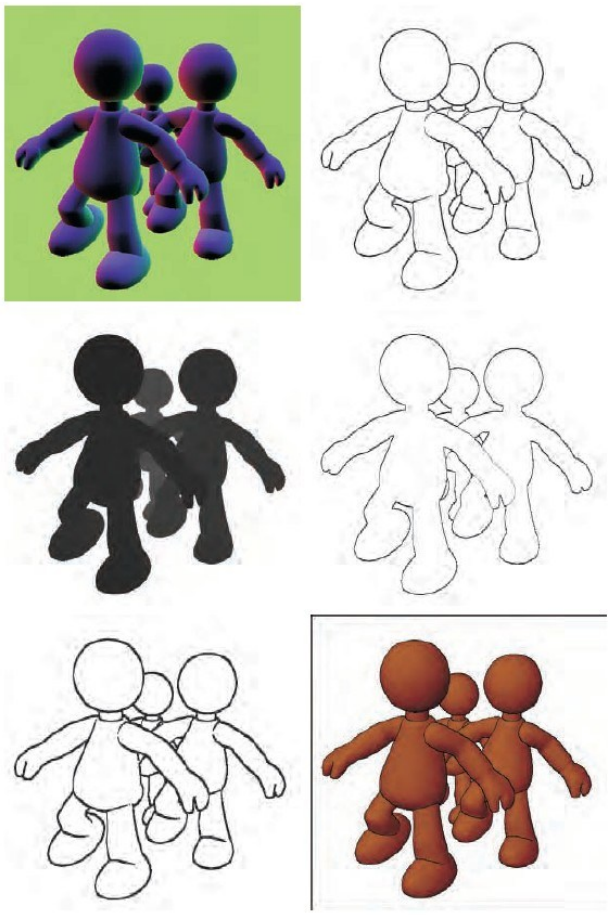


图 13 通过对场景的法线图和 z 深度值就行处理来进行边缘检测，右图所示为经过加粗的合成结果。注意，在这种情况下，使用法线图就主义检测出所有的边缘。

10.3.4 基于轮廓边缘检测的描边

上文提到的大多数渲染描边的方法都存在一个缺点，那就是他们都需要两个通道才能完成物体轮廓描边的渲染。

基于轮廓边缘检测的描边，通过检测出轮廓边缘（Silhouette EdgeDetection）），并直接对它们进行绘制，这种形式的描边，可以很好地控制线条绘制的过程。由于边缘独立于模型，因此这种方法还有另外一个优点，就是能够生成一些特殊的效果。例如，在网格密集的地方可以突现出轮廓边缘。

可以将轮廓边缘理解为朝向相反的相邻三角形的交接。也就是说，其中的一个三角形是朝向视点，另一个三角形背向视点。具体测试方法如下：

$$(\mathbf{n}_0 \cdot \mathbf{v} > 0) \neq (\mathbf{n}_1 \cdot \mathbf{v} > 0),$$

其中 n0 和 n1 分别表示两个三角形的表面法线向量，v 表示从视点到这条边缘（也就是其中任何一个端点）的视线方向向量。而为了确保这种测试的准确性，必须保证表面的取向一致。

10.3.5 混和轮廓描边

混和轮廓描边（Hybrid Silhouetting），即结合了图像处理方法和几何要素方法，来渲染轮廓的方法。

这种方法的具体思想是：首先，找到一系列轮廓边缘的列表。其次，渲染出所有物体的三角形和轮廓边缘，同时为他们指定一个不同的 ID 值（也就是说，赋予不同的颜色）。接着读取该 ID 缓冲器并从中判断出可见的轮廓边缘，随之对这些可见线段进行重叠检测，并将它们连接起来形成平滑的笔划路径。最后就可以对这些重建起来的路径进行风格化笔划渲染，其中，这些笔划本身可以用很多方法来进行风格化处理，包括变细、火焰、摆动、淡化等效果，同时还有深度和距离信息。如下图。

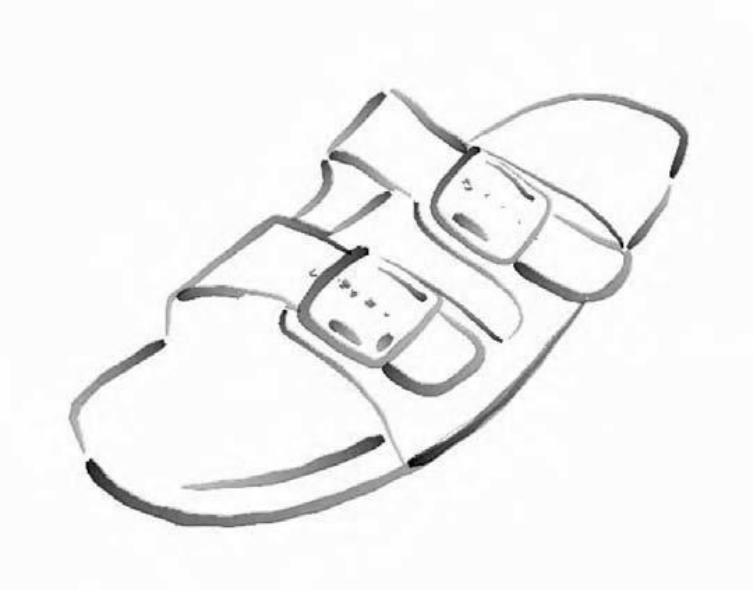


图 14 使用混合轮廓描边方法生成的图像，其中可以将找到的轮廓边缘连接起来作为笔划进行渲染。

10.4 其他风格的 NPR 渲染技术小结

除了 toon 渲染这种比较受欢迎的模拟风格之外，还存在其他各式各样的风格。NPR 效果涵盖的范围非常广泛，从修改具有真实感效果的纹理，到使用算法一幅幅画面的几何修饰。RTR3 中主要谈了 3 种不同的其他风格的 NPR 渲染技术：

- 纹理调色板（Palette of Textures）
- 色调艺术图（Tonal Art Maps，TAM）
- 嫁接（Graftals）

下面分别进行介绍。

10.4.1 纹理调色板（Palette of Textures）

纹理调色板（palette of textures）由 Lake 等人讨论提出，基本思想是通过反射着色项（diffuse shading term）的不同，来选择应用于物体表面上的不同纹理。随着漫反射项逐渐变暗，可以选用相应更暗的纹理，而为了能够产生手绘的效果，可以使用屏幕空间坐标来采样纹理。同时，为了增强绘制效果，可以在屏幕空间上的所有表面上运用纸纹理。随着物体的运动，他们就可以在纹理之间进行穿梭。原因在于这个纹理是在屏幕空间中实现的。此外，也可以在世界空间中运用这个纹理，这样就能够得到一个与屏幕空间完全不同的效果。

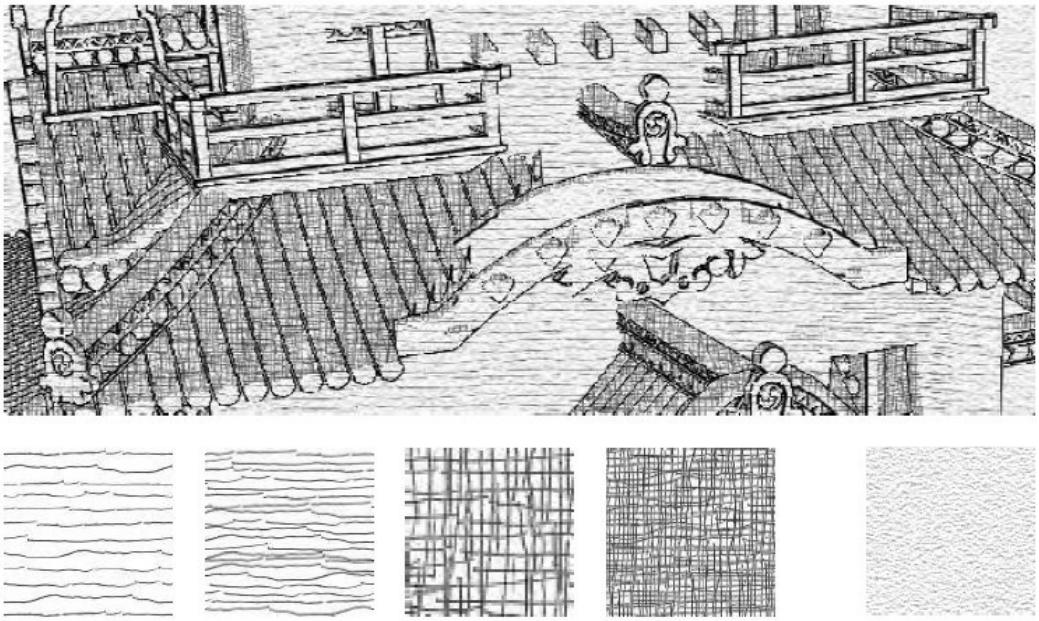


图 15 使用纹理调色板（paletteof textures）、纸纹理，以及轮廓边缘绘制生成的一幅图像。

10.4.2 色调艺术图（Tonal Art Maps，TAM）

通过在纹理之间进行切换形成的硬着色效果和 toon 着色效果之间的一种混合，Praun 等人（https://www.dimap.ufrn.br/~motta/dim102/Projetos/p581-praun.pdf）提出了一种可以实时生成笔划纹理分级细化图的方法，并可以将其以平滑的方式运用到物体表面上。第一步是生成即时使用的纹理，称为色调艺术图（Tonal Art Maps，TAM）,主要思想是将笔划绘制为分级细分图层次，如图。

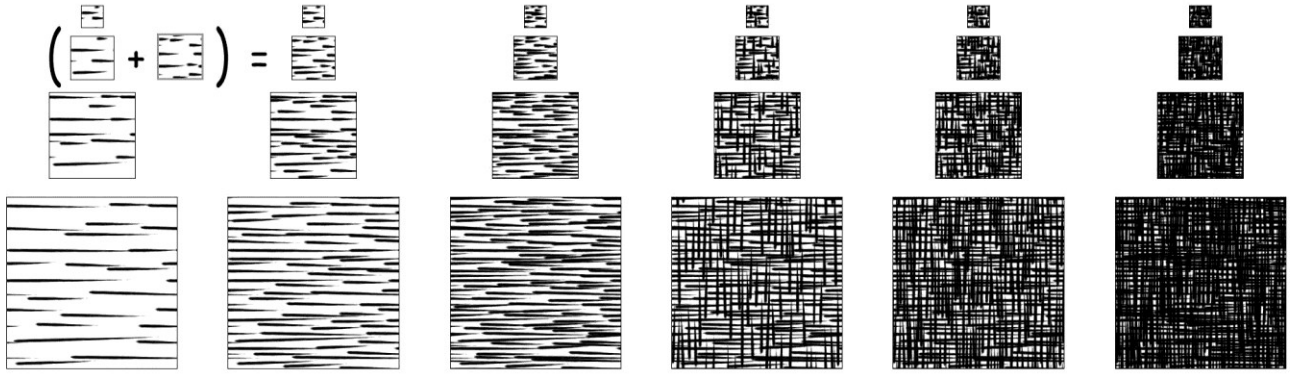


图 16 TAM 将笔划绘制到细分图层次中，每个分级细化图层次包含图中左边和上边纹理中的所有笔划，这样，在细化图层次之间和相邻纹理之间的插值就比较平滑

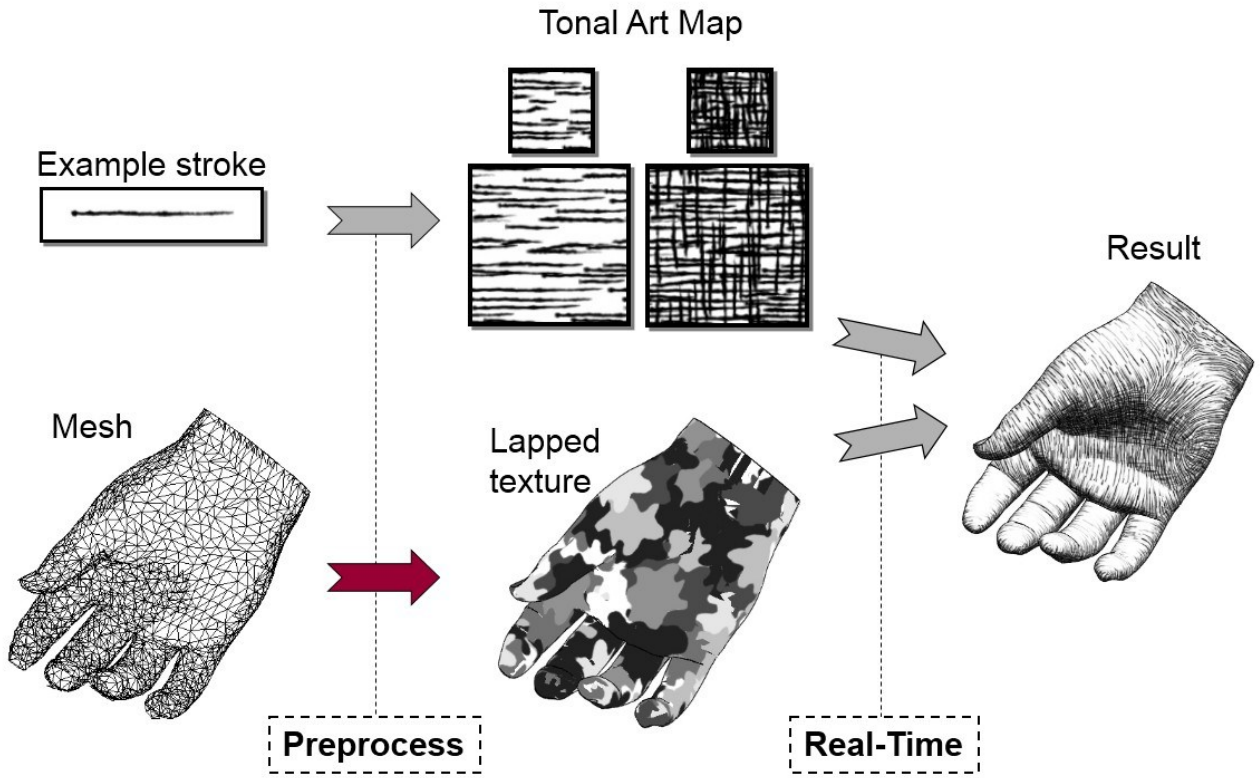


图 17 使用 TAM 渲染一副素描图的过程

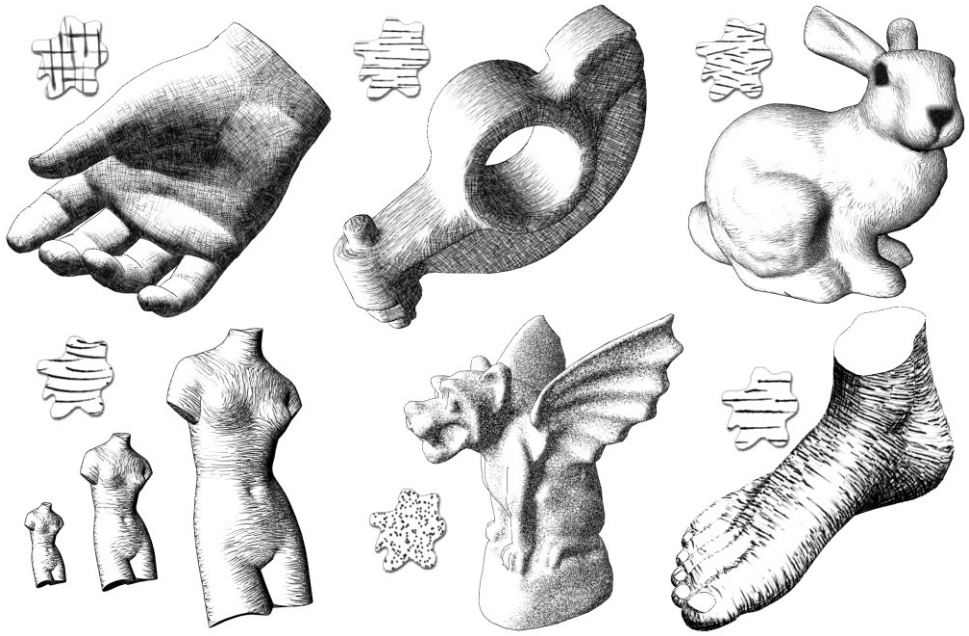


图 18 6 种不同的 TAM 渲染出的 6 组不同模型

10.4.3 嫁接（Graftals）

嫁接（Graftals）的基本思想，是将几何或者贴花纹理应用到物体表面，从而产生某种特殊效果。可以通过所需要的细节层次，物体表面相对视点的方位或者其他因素，对纹理进行控制。这种方法可以用来模拟钢笔或者画刷的笔刷，如下图。

图 19 使用两种不同的嫁接（Graftals）风格绘制出来的 Standord 小兔

10.5 关于水彩风格的 NPR

在写这篇文章查阅 NPR 相关资料的过程中，发现了非常酷的一个水彩风格化渲染的框架，Maya Non-photorealistic Rendering Framework，简称 MNPR.http://artineering.io/docs/mnprDocsWC/。

MNPR 在 SIGGRAPH 2017 有亮相，可以实现非常棒的水彩（Watercolor）渲染效果。有兴趣的朋友可以了解一下。以下是相关的一些精彩的图示。

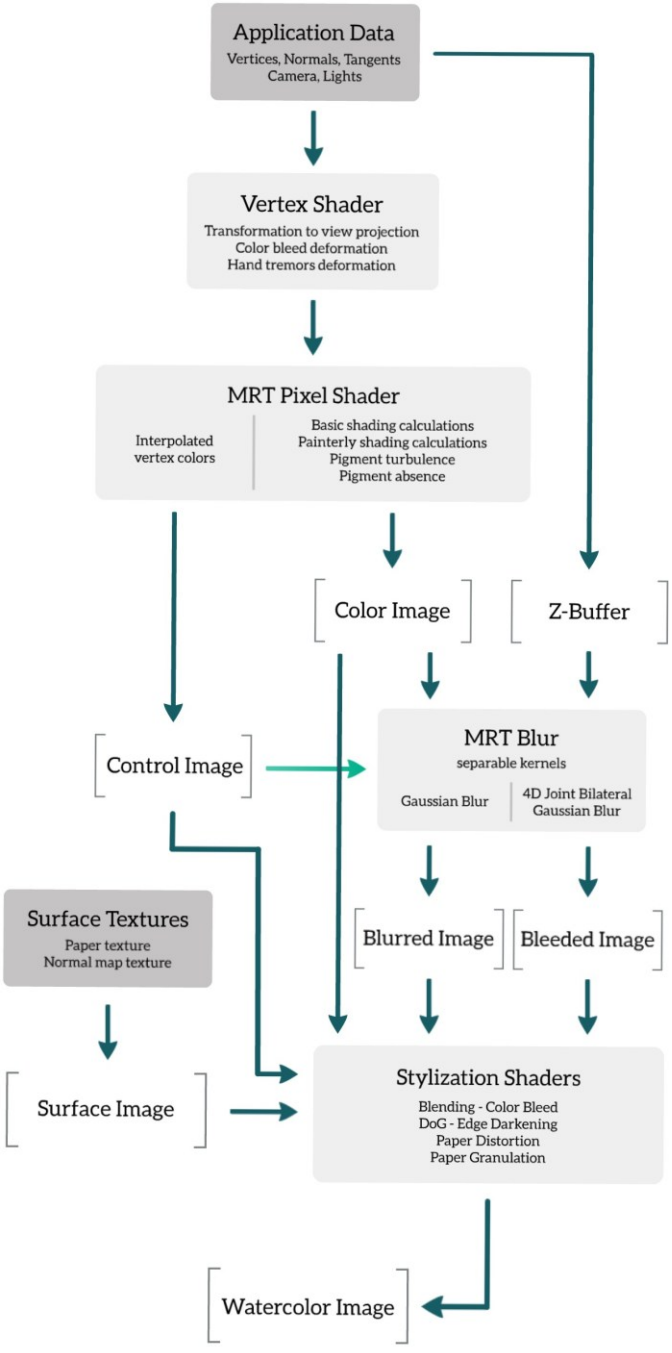


图 20 一个典型水彩风格渲染流程图

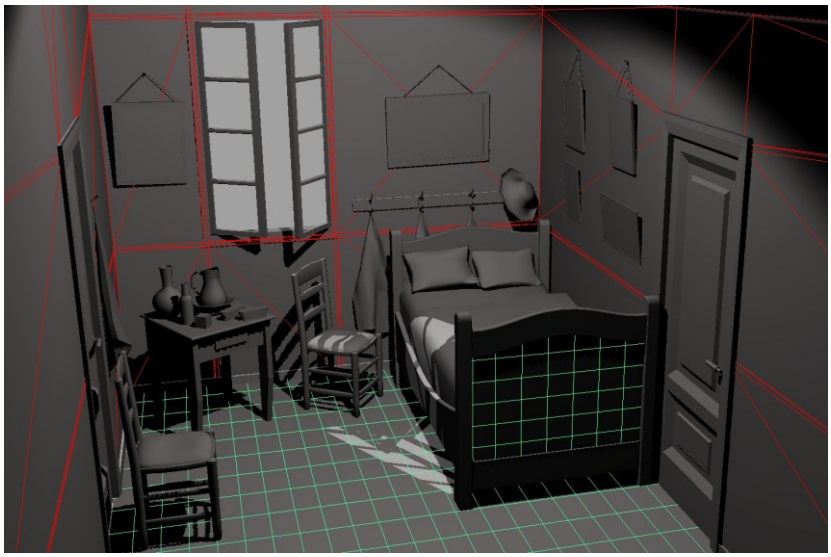


图 21 水彩风格的 NPR 建模过程

图 22 水彩风格的 NPR 效果图

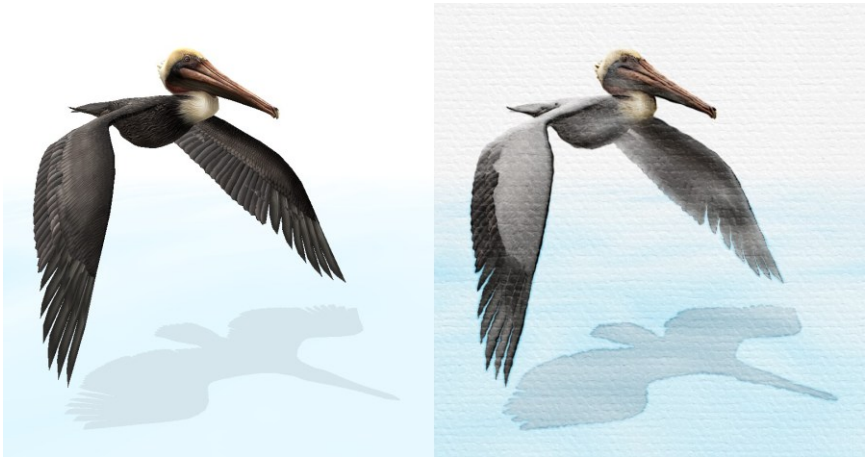


图 23 真实感渲染 vs. 水彩风格 NPR 渲染

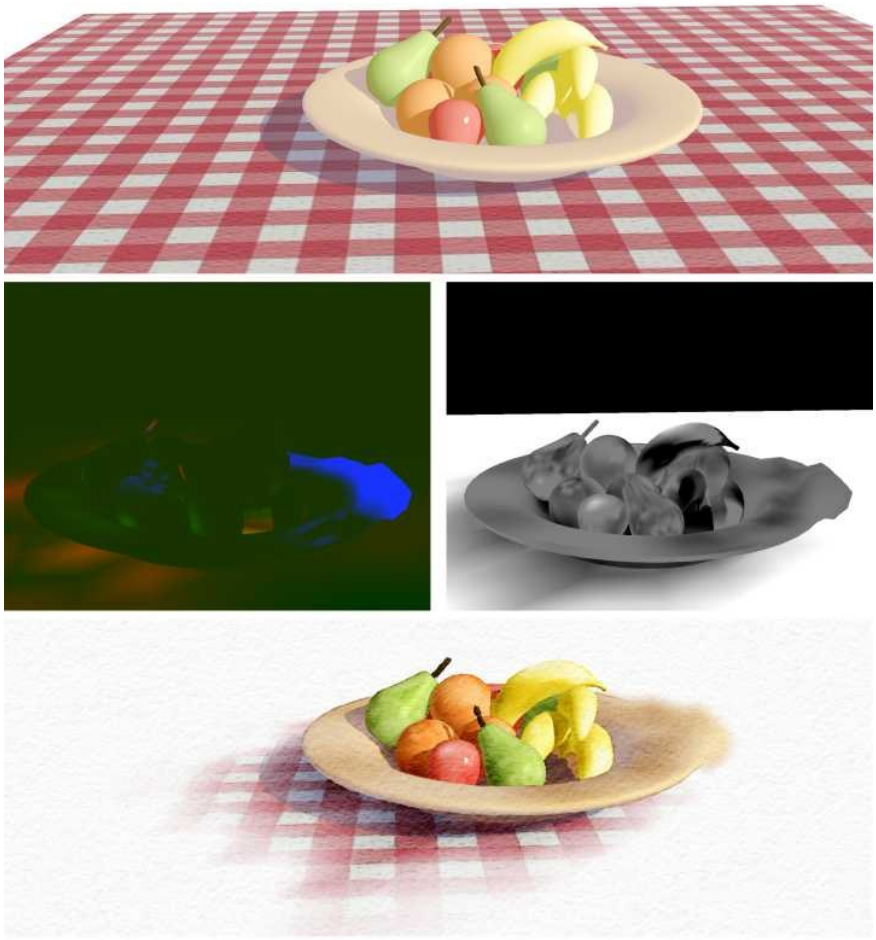


图 24 水彩 NPR 的渲染过程

10.6 NPR 相关著作

如下两本书从技术和 NPR 绘画算法两个方面，对非真实感渲染有了一个系统的涵盖，可谓 NPR 界的泰斗之作，有兴趣的朋友不妨进一步深入阅读。

- Gooch, Bruce or Amy, and Amy or Bruce Gooch, Non-Photorealistic Rendering,A K Peters Ltd., 2001.
- Strothotte, Thomas, and Stefan Schlechtweg ,Non-Photorealistic Computer Graphics: Modeling, Rendering, and Animation, Morgan Kaufmann, 2002.

10.7 NPR 相关延伸资料推荐

- 链接（http://kesen.realtimerendering.com/）中的 Non-Photorealistic Animation and Rendering Proceedings 一栏里可以找到 NPR 业界前沿的一些发展近况，即 NPAR 会议相关的资源。
- SIGGRAPH 2010 上 Stylized Rendering in Games 课程（Stylized Rendering in Games）里有不少值得了解的 NPR 的内容。
- NPR resources page（http://www.red3d.com/cwr/npr/）里的材料也值得一看。

10.8 Reference

[1] http://www.gatheryourparty.com/2013/05/30/crash-course-cel-shading-in-video-games/

[2] http://www-cg.cis.iwate-u.ac.jp/lab/suiboku.html

[3] https://zhuanlan.zhihu.com/p/26409746

[4] https://www.zhihu.com/question/32078473

[5] http://www.valvesoftware.com/publications/2007/NPAR07_IllustrativeRenderingInTeamFortress2.pdf

[6] http://www.4gamer.net/games/216/G021678/20140703095/

[7] http://www.4gamer.net/games/216/G021678/20140714079/

[8] http://www.4gamer.net/games/216/G021678/20150317055/

[9][https://en.wikipedia.org/wiki/Cel_shading](https://link.zhihu.com/?target=https%3A//en.wikipedia.org/wiki/Cel_shading)

[10] Praun E, Hoppe H, Webb M, et al.Real-time hatching[C]//Proceedings of the 28th annual conference on Computergraphics and interactive techniques. ACM, 2001: 581. http://gfx.cs.princeton.edu/proj/hatching/

[11] http://blog.csdn.net/silangquan/article/details/17184807

[12] http://blog.csdn.net/candycat1992/article/details/45577749

[13] http://blog.csdn.net/thenile/article/details/8858702

[14] http://www.cs.duke.edu/courses/compsci344/spring15/classwork/16_npr/cmu.pdf

[15] http://libregraphicsworld.org/blog/entry/freestyle-jot-and-the-future-of-non-photorealistic-rendering

[16] http://www.cartoonbrew.com/tag/non-photorealistic-rendering

[17] Art-directed watercolor stylization of3D animations in real-time

http://artineering.io/articles/Art-directed-watercolor-stylization-of-3D-animations-in-real-time/

[18] https://expressivesymposium.com/

第十一章 游戏开发中的渲染加速算法总结

本章将带来 RTR3 第十四章内容“Chapter 14 Acceleration Algorithms”的总结、概括与提炼。

这是一篇 1 万 3 千余字的总结式文章，通过阅读，你将对游戏开发与实时渲染中加速渲染算法的以下要点有所了解：

- 常用空间数据结构（Spatial Data Structures）
  - 层次包围盒（BVH ,Bounding Volume Hierarchies）
  - BSP 树（BSP Trees）
  - 八叉树（Octrees）
  - 场景图（Scene Graphs）
- 各种裁剪技术（Culling Techniques）
  - 背面裁剪（Backface Culling）
  - 视锥裁剪（View Frustum Culling）
  - 遮挡剔除（Occlusion Culling）

  - 层次视锥裁剪（Hierarchical View Frustum Culling）
  - 入口裁剪（Portal Culling）
  - 细节裁剪（Detail Culling）
- 各种层次细节（LOD，Level of Detail）技术
  - 几种 LOD 切换技术（Discrete Geometry LODs、Blend LODs、Alpha LODs、CLODs and Geomorph LODs）
  - 几种 LOD 的选取技术（Range-Based、Projected Area-Based、Hysteresis）
- 大型模型的渲染（Large Model Rendering）
- 点渲染（Point Rendering）

11.0 引言

《Real-Time Rendering 3rd》书中提到，实时渲染领域有四大目标，激励着游戏开发者们不断进步，它们是：

- 更高的每秒帧数
- 更高的分辨率
- 渲染更多的物体与更具真实感的场景
- 实现更高的复杂度

而要不断地追逐这四大目标，需要持续不断的优化算法，进行技术革新和硬件的升级。其中，加速渲染相关的算法一直是追逐这四大目标的重要一环。

这篇文章将基于《Real-Time Rendering 3rd》第十四章“Acceleration Algorithms”的内容，介绍计算机图形学和游戏开发中常用的对渲染进行加速的算法，尤其是对大量几何体的渲染，而很多这类算法的核心都是基于空间数据结构（Spatial Data Structures）。所以，本文将先介绍一些游戏开发中常用的空间数据结构，再进行各种加速算法，不同种类的裁剪算法，LOD 相关的介绍。

11.1 空间数据结构 | Spatial Data Structures

空间数据结构（Spatial Data Structures）是将几何体组织在 N 维空间中的一系列数据结构，而且我们可以很容易地将二维和三维的一些概念扩展到高维之中。这些空间数据结构可以用于很多实时渲染相关操作的加速查询中，如场景管理，裁减算法、相交测试、光线追踪、以及碰撞检测等。

空间数据结构的组织通常是层次结构的。宽泛地说，即最顶层包含它之下的层次，后者又包含更下层的层次，以此类推。因此，这种结构具有嵌套和递归的特点。用层次结构的实现方式对访问速度的提升很有帮助，复杂度可以从 O(n)提升到 O(log n)。但同时，使用了层次结构的大多数空间数据结构的构造开销都比较大，虽然也可以在实时过程中进行渐进更新，但是通常需要作为一个预处理的过程来完成。

一些常见的空间数据结构包括：

- 层次包围盒（Bounding Volume Hierachy，BVH）
- 二元空间分割树（Binary Space Partitioning，BSP），
- 四叉树 （QuadTree）
- kd 树（k-dimensional tree）
- 八叉树（Octree）
- 场景图 （Scene Graphs）

其中，BSP 树和八叉树都是基于空间细分（Space Subdivision）的数据结构，这说明它们是对整个场景空间进行细分并编码到数据结构中的。例如，所有叶子节点的空间集合等同于整个场景空间，而且叶子节点不相互重叠。

BSP 树的大多数变种形式都是不规则的，而松散地意味着空间可以被任意细分。

八叉树是规则的，意味着空间是以一种均匀的形式进行分割，虽然这种均匀性限制比较大，但这种均匀性常常是效率的源泉。另外值得注意的是，八叉树是四叉树的三维空间推广。

另一方面，层次包围盒不是空间细分结构，它仅将几何物体周围的空间包围起来，所以包围层次不需要包围所有的空间。

下文将对其中的层次包围盒、二元空间分割树、八叉树进行近一步介绍，并还将简单提到场景图(SceneGraph)，这是一种比较高层次的，相较渲染性能更关注模型关系的数据结构。

当然，限于篇幅原因，这里的每种数据结构都无法介绍得事无巨细，但已在每种数据结构介绍的最后备好了一些延伸的阅读材料，方便希望进一步了解的朋友们进行延伸阅读。

11.1.1 层次包围盒 | Bounding Volume Hierarchies , BVH

层次包围盒（Bounding Volume Hierarchies, BVH）方法的核心思想是用体积略大而几何特征简单的包围盒来近似地描述复杂的几何对象，从而只需对包围盒重叠的对象进行进一步的相交测试。此外，通过构造树状层次结构，可以越来越逼近对象的几何模型，直到几乎完全获得对象的几何特征。

对于三维场景的实时渲染来说，层次包围体（Bounding Volume Hierarchy，BVH）是最常使用的一种空间数据结构。例如，层次包围体经常用于层次视锥裁减。场景以层次树结构进行组织，包含一个根节点（root）、一些内部节点（internal nodes），以及一些叶子节点（leaves）。顶部的节点是根，其无父节点。叶子节点（leaf node）包含需渲染的实际几何体，且其没有子节点。

相比之下，内部节点包含指向它子节点的指针。因此，只要根节点不是这颗树唯一的一个节点，那么它就是一个内部节点。树中的每一个节点，包括叶子节点，都有一个包围体可以将其子树中的所有几何体包围起来，这就是包围体层次的命名来源，同时，也说明了根节点有一个包含整个场景的包围体。

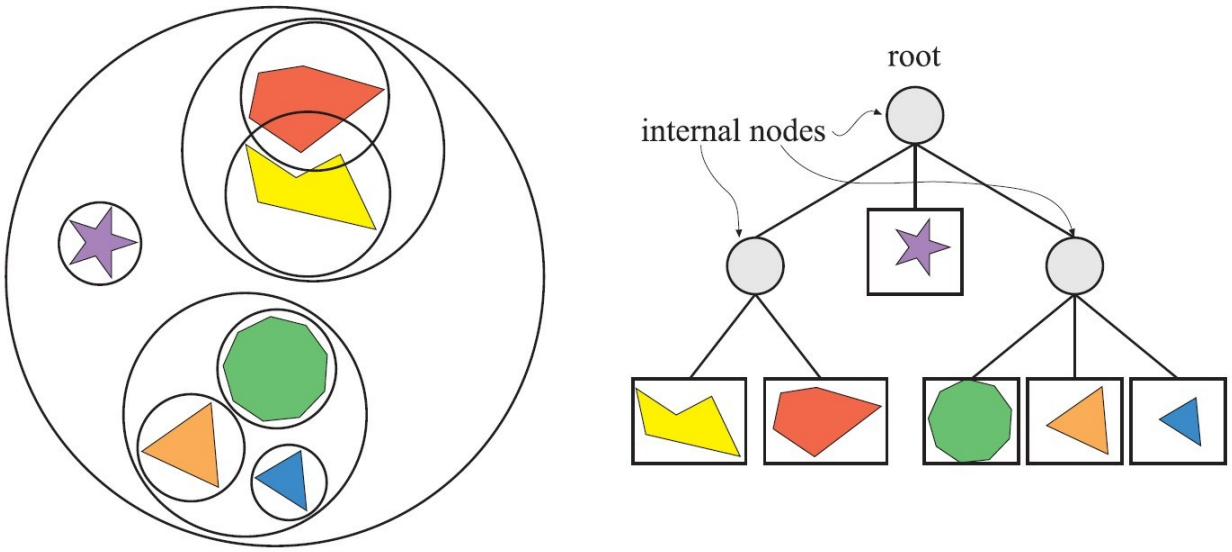


图 1 左图为一个包含 6 个物体的简单场景，每个物体由一个包围的球体封闭起来，其中可以将包围球体归组为一个更大的包围球体，如此内推，直到所有的物体被最大的球体包围，右图所示为层次包围体（树），可以用来表示左图的物体层次、根节点的包围体包含场景中的所有物体。

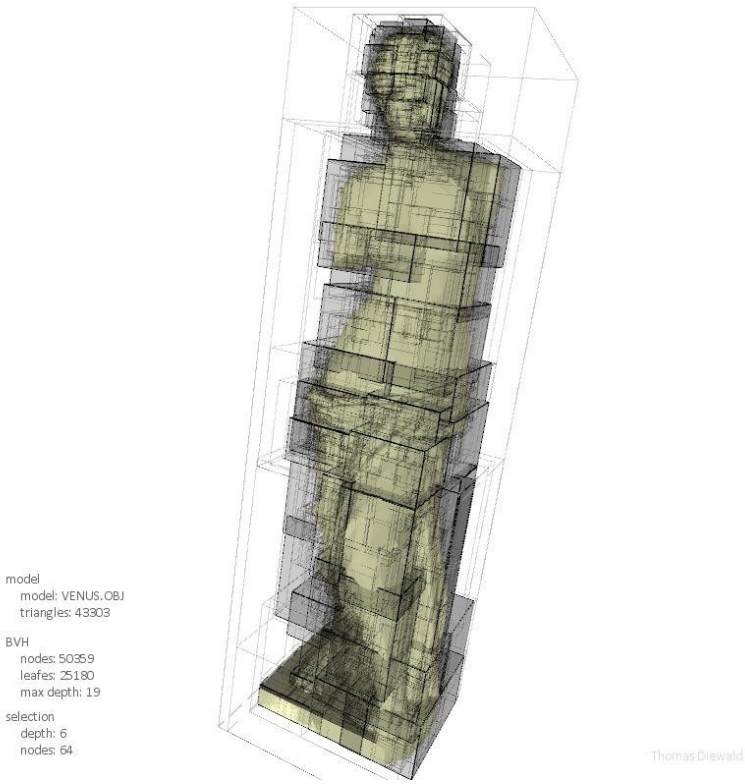


图 2 层次包围盒的实现 @http://thomasdiewald.com/blog/?p=1488

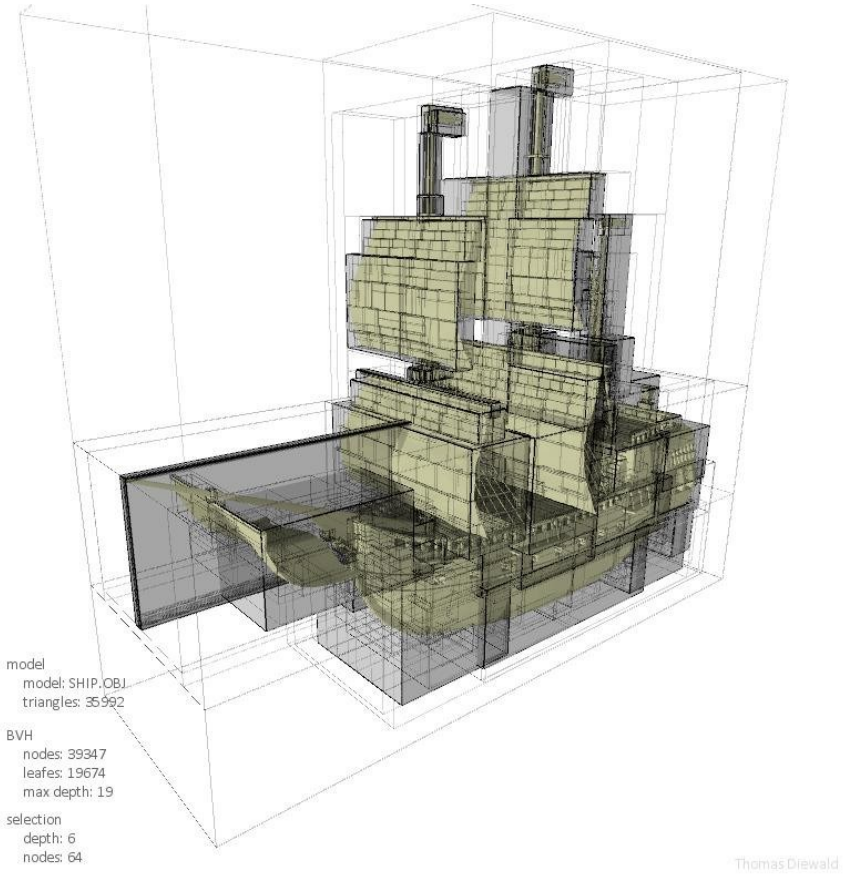


图 3 层次包围盒的实现 @http://thomasdiewald.com/blog/?p=1488

11.1.1.1 BVH 的延伸阅读材料

[1] https://hal.inria.fr/inria-00537446/file/bounding_volume_hierarchies.pdf

[2] https://www.codeproject.com/Articles/832957/Dynamic-Bounding-Volume-Hiearchy-in-Csharp

[3] Wald I, Boulos S, Shirley P. Raytracing deformable scenes using dynamic bounding volume hierarchies[J]. ACM Transactions on Graphics (TOG), 2007, 26(1): 6.

11.1.2 BSP 树 | BSP Trees

BSP 树(二叉空间分割树，全称 Binary Space Partitioning Tree)是一种常用于判别对象可见性的空间数据结构。类似于画家算法，BSP 树可以方便地将表面由后往前地在屏幕上渲染出来，特别适用于场景中对象固定不变，仅视点移动的情况。

其中，BSP 是 Binary SpacePartitioning（二叉空间划分法）的缩写。这种方法递归地将空间使用超平面划分为凸面体集合。而这种子划分引出了借助于称之为 BSP 树的树形数据结构的场景表示。

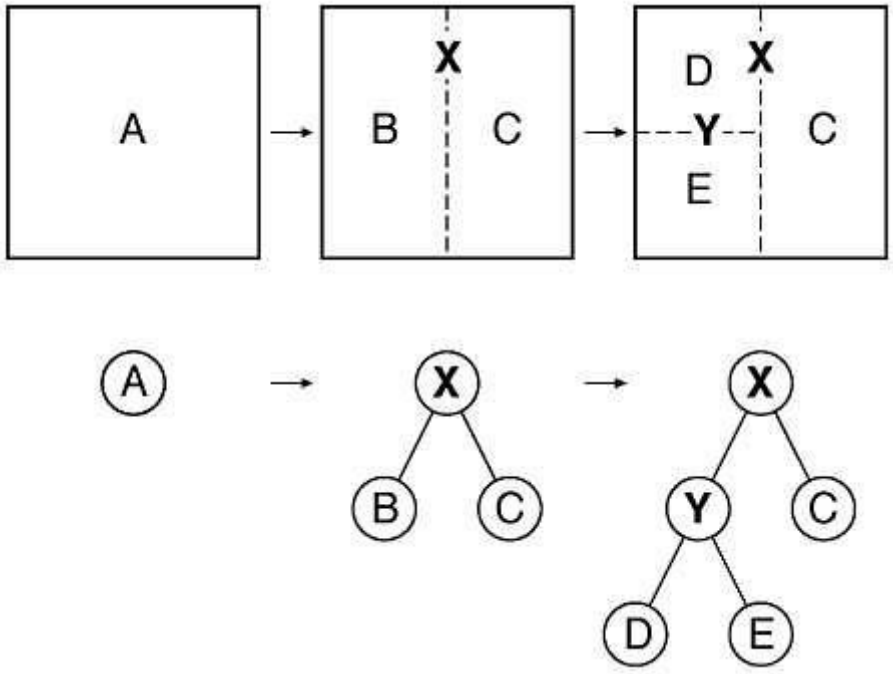


图 4 一个 BSP 树的构造

BSP 树是一棵二叉树，每个节点表示一个有向超平面，其将当前空间划分为前向（front）和背向（back）两个子空间，分别对应当前节点的左子树和右子树。且 BSP 树已经在游戏工业

中应用了许多年（Doom 是第一个使用 BSP 树的商业游戏）。尽管在现今 BSP 树已经没像过去那么受欢迎了，但使用依然广泛。

BSP 树的一个有趣特性是，如果用一种特定的方式遍历，树的几何内容可以从任何角度进行前后排序。这个排序可以近似轴对齐，精确对齐多边形 BSP。而 BVH 与之不同，因为 BVH 通常不包含任何形式的排序。

11.1.2.1 BSP 树的构造

- 从空树开始，每次选择一个面片作为节点插入树中
- 每次插入一个新节点，从树的根节点开始遍历
  - 如果新节点面片与当前结点片面相交，将新面片分割成两个面片
  - 新节点在当前节点前向空间，插入左子树
  - 新节点在当前节点背向空间，插入右子树
  - 当前节点为空，直接插入新节点
- 直到所有面片都被插入树中

简单来说，若要创建 BSP 树，需递归将一个平面空间一分为二，并将几何体归类到这两个空间中来完成。

11.1.2.2 BSP 树的遍历

从根节点开始，判断输入位置与当前分割平面的“前”、“后”关系，

“前”则遍历左子树，“后”则遍历右子树，递归到叶子节点终止。

用平面方程 $Ax + By + Cz + D = 0$ 判断前后位置，可用 $D(x0, y0, z0) = Ax0 + By0 + Cz0 + D$ 进行判别，其中：

- $D > 0$：在平面前面
- $D = 0$：在平面上
- $D < 0$：在平面后面

这里贴出从后向前遍历 BSP 的示例代码：

```
traverse_tree(bsp_tree* tree,point eye)
{
```

```
    location = tree->find_location(eye);

    if(tree->empty())
        return;

    if(location > 0)          // if eyeinfront of location
    {
        traverse_tree(tree->back,eye);
        display(tree->polygon_list);
        traverse_tree(tree->front,eye);
    }
    else if(location < 0) // eye behind location
    {
        traverse_tree(tree->front,eye);
        display(tree->polygon_list);
        travers_tree(tree->back,eye);
    }
    else                          // eyecoincidental with partition hyperplane
    {
        traverse_tree(tree->front,eye);
        traverse_tree(tree->back,eye);
    }
}
```

11.1.2.3 BSP 树的种类

在计算机图形学中，BSP 树有两大类别，分别是为轴对齐（Axis-Aligned）BSP 树和多边形对齐（Polygon-Aligned）BSP 树。下面分别进行介绍。

11.1.2.4 轴对齐 BSP 树 | Axis-aligned BSP tree

轴对齐 BSP 树可以按如下方式来创建。首先，将整个场景包围在一个 AABB（轴对齐包围盒，Axis-Aligned Bounding Box）中，然后以递归的方式将这个包围盒分为若干个更小的盒子。

现在，考虑一下任何递归层次的盒子。选取盒子的一个轴，生成一个与之垂直的平面，将盒子一分为二。有一些方法可以将这个分割平面固定，从而将这个盒子分为完全相同的两部分，而也有其他的一些方法，允许这个平面在位置上有一些变化。与分割平面相交的物体，要么存储在这个层次上，成为两个子集中的一员，要么被这个平面分割为两个不同的物体。

经过这个过程，每个子集就处于一个比较小的盒子中，重复这个平面分割的过程，就可以对每个 AABB 进行递归细分，直到满足某个标准才终止这个分割过程。而这个标准，通常是用户定义的树最大深度，或者是盒子里面所包含的几何图元数量，需低于用户定义的某个值。

分割平面的轴线和位置对提高效率至关重要。一种分割包围盒的方法就是轴进行循环。即在根节点，沿着 x 轴对盒子进行分割，然后再沿着 y 轴对其子盒子进行分割，最后沿 z 轴对其孙盒子进行分割。这样，就完成了一个循环周期。使用这种分割策略的 BSP 树常被称为 k-d 树。而另一种常见策略是找到盒子的最长边，沿着这条边的方向对盒子进行分割。

下图展示了一种轴对齐 BSP 树的分割过程。

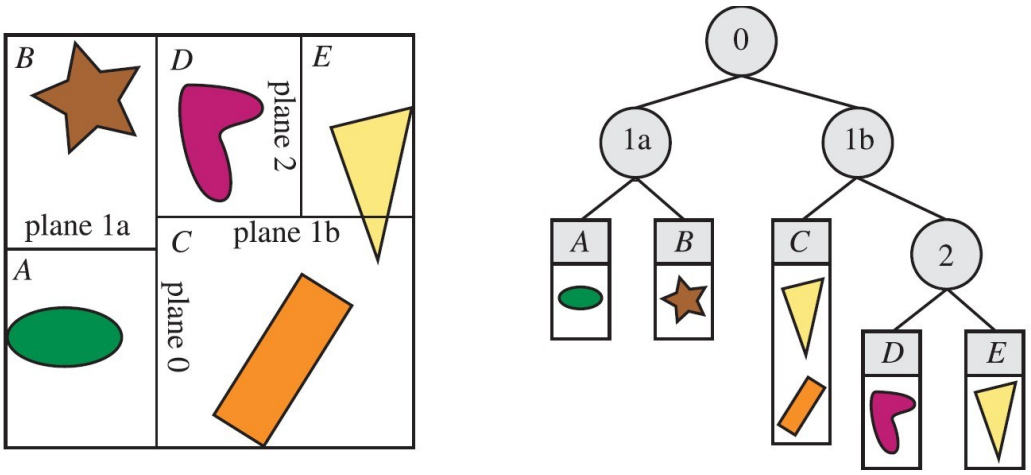


图 5 轴对齐 BSP 树。在这个示例中，允许空间分割位于轴上的任意位置，不一定必须在中点位置，形成的空间体分别用 A~E 来标志。右图所示的树是当前的 BSP 树数据结构，每个叶子节点表示一个区域，区域内容显示在下方。注意，黄色三角形在物体列表中含有 C 和 E 两个区域，因为它同时覆盖了这两个区域。

值得一提的是，从前到后的粗排序（Rough Front-to-Back Sorting）是轴对齐 BSP 树的一种应用示例，这种方法对于遮挡剔除算法非常有用。而在视点的另一侧进行遍历，可以得到从后向前的粗排序（Rough Fack-to-Gront Sorting）,这对于透明排序非常有用。且还可以用来测试射线和场景几何体相交的问题，只需将视点位置换为射线原点即可，另外还可以用于视锥裁剪。

11.1.2.5 多边形对齐 BSP 树 | Polygon-aligned BSP tree

多边形对齐 BSP 树（Polygon-aligned BSP tree）是 BSP 树的另一大类型，其中将多边形作为分隔物，对空间进行平分。也就是说，在根节点处，选取一个多边形，用这个多边形所在平面将场景中剩余多边形分为两组。对于与分割平面相交的多边形来说，沿着其中的交线将这个

多边形分为两部分。然后，在分割平面的每个半空间中，选取另外一个多边形作为分隔物，只对这个分隔物所在平面的多边形进行继续分割，直到所有的多边形都在 BSP 树中为止。

需要注意，多边形对齐 BSP 树的创建是一个非常耗时的过程，这些树通常只需计算一次，可以存储起来进行重用。

下图是一个多边形对齐 BSP 树的图示。

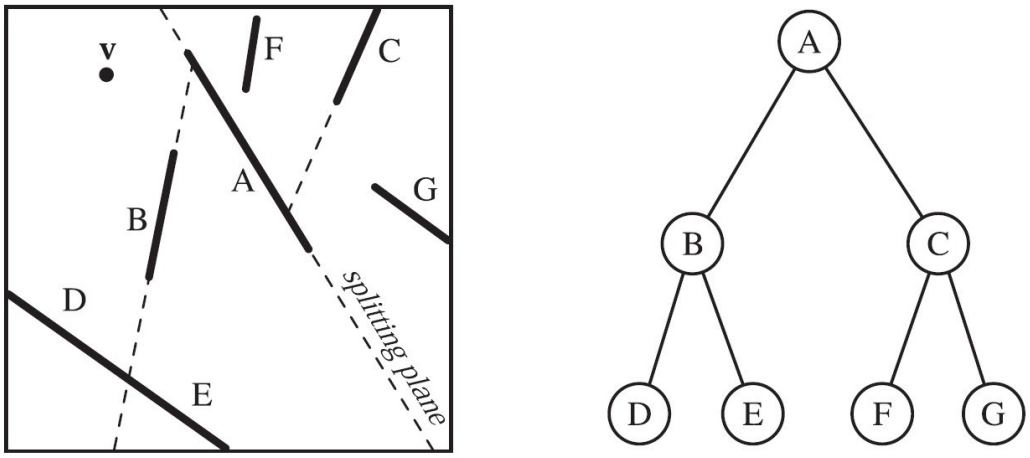


图 6 多边形对齐 BSP 树。左图中，多边形分别用 A~G 表示。首先，用多边形 A 对空间进行分割，生成的两个半空间分别由多边形 B 和 C 分割，由 B 形成的分割平面与左下角的多边形相交，将其分割为多边形 D 和 E。最后形成的 BSP 书如右图所示。

因为完全不平衡树的效率非常低，所以多边形对齐 BSP 树分割时最好是形成平衡树，即每个叶子节点的深度相同或者相差一个层次的树。

多边形对齐 BSP 树的一个典型性质就是对于一个给定的视点来说，可以对该结构按照从后往前（或者从前往后）的顺序进行严格遍历，而轴对齐的 BSP 通常只能给出粗略的排序顺序。所以，基于多边形对齐 BSP 树的此性质，建立了严格的前后顺序，可以配合画家算法来绘制整个场景，而无需 Z 缓冲。

多边形对齐 BSP 树的其他应用也包括相交测试和碰撞检测等。

11.1.2.6 BSP 树的延伸阅读材料推荐

[1] http://web.cs.wpi.edu/~matt/courses/cs563/talks/bsp/bsp.html

[2] https://pdfs.semanticscholar.org/90e4/c4a65b4b04d9e2374e5753659c102de4c0eb.pdf

[3] https://en.wikipedia.org/wiki/Binary_space_partitioning

[4] http://archive.gamedev.net/archive/reference/programming/features/bsptree/bsp.pdf

11.1.3 八叉树 | Octrees

八叉树（octree），或称八元树，是一种用于描述三维空间的树状数据结构。八叉树的每个节点表示一个正方体的体积元素，每个节点有八个子节点，这八个子节点所表示的体积元素加在一起就等于父节点的体积。一般中心点作为节点的分叉中心。

简单来说，八叉树的空间划分方式很简单，即递归地进行规整地 1 分为 8 的操作。如下图，把一个立方体分割为八个同样大小的小立方体，然后递归地分割出更的小立方体。这个就是八叉树的命名来源。这种分割方式可以得到比较规则的结构，从而使得查询变得高效。

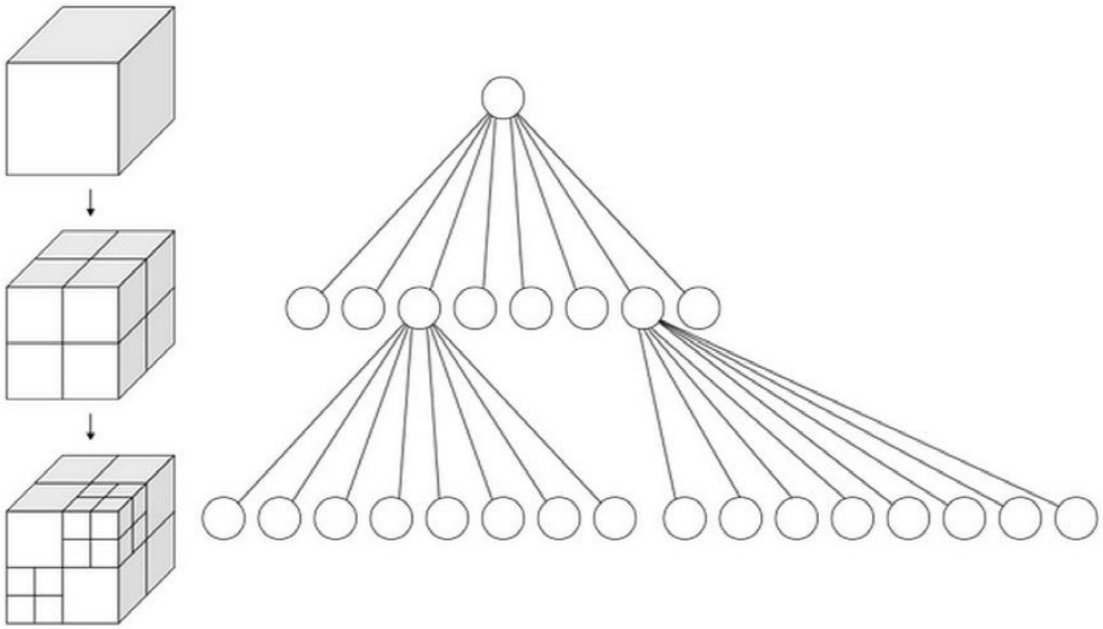


图 7 八叉树的构成 @wiki

相似地，四叉树是把一个二维的正方形空间分割成四个小正方形。而八叉树是四叉树的三维空间推广。

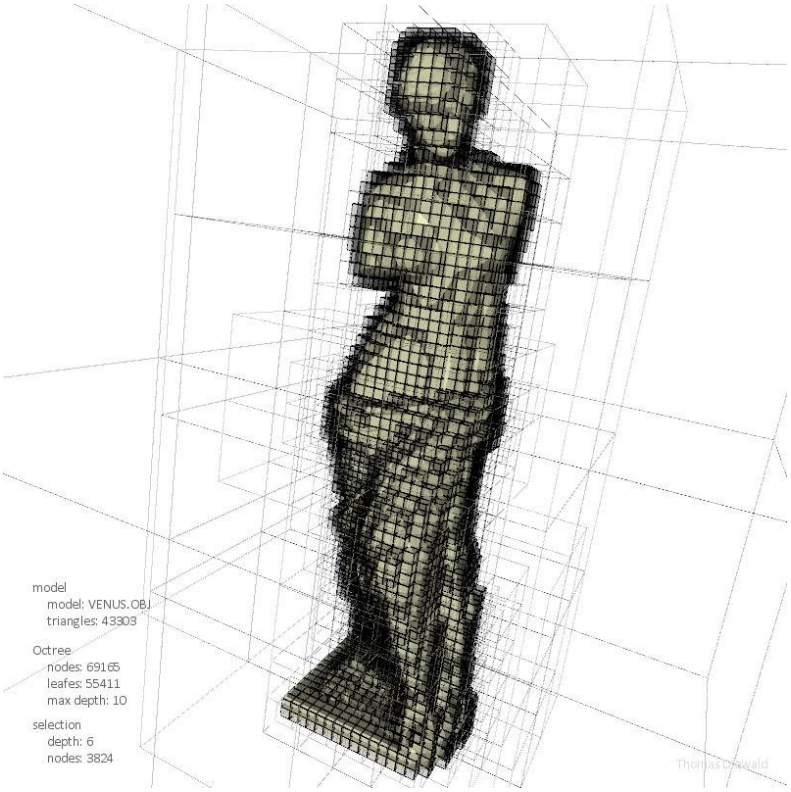


图 8 八叉树的实现 @http://thomasdiewald.com/blog/?p=1488

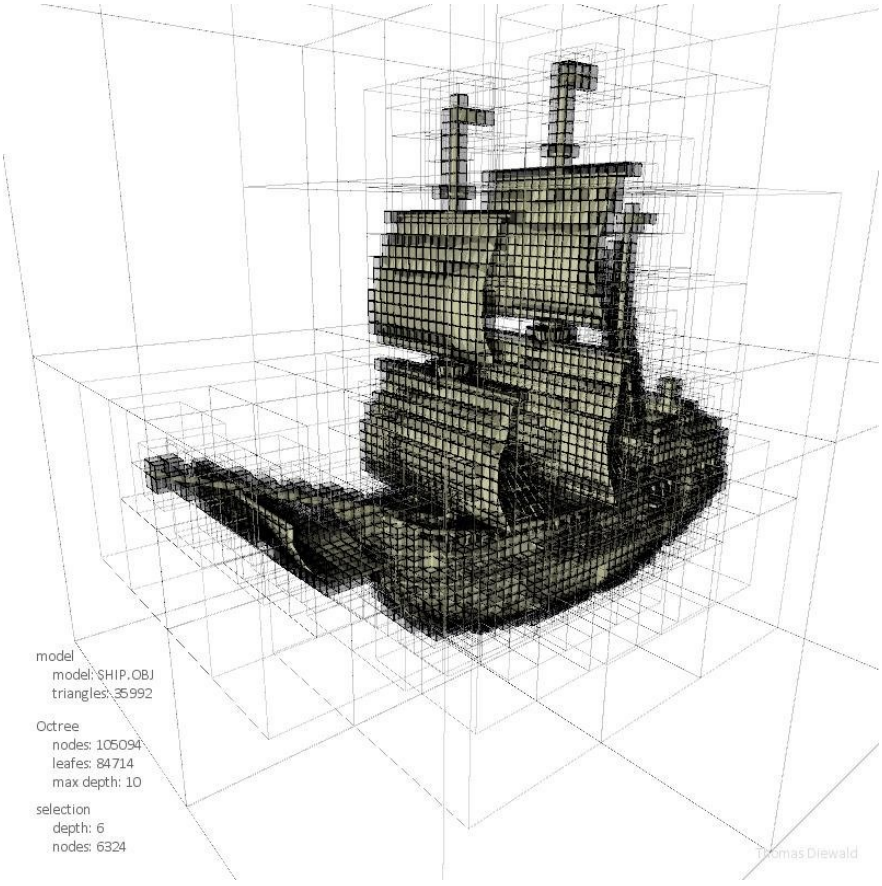


图 9 八叉树的实现 @http://thomasdiewald.com/blog/?p=1488

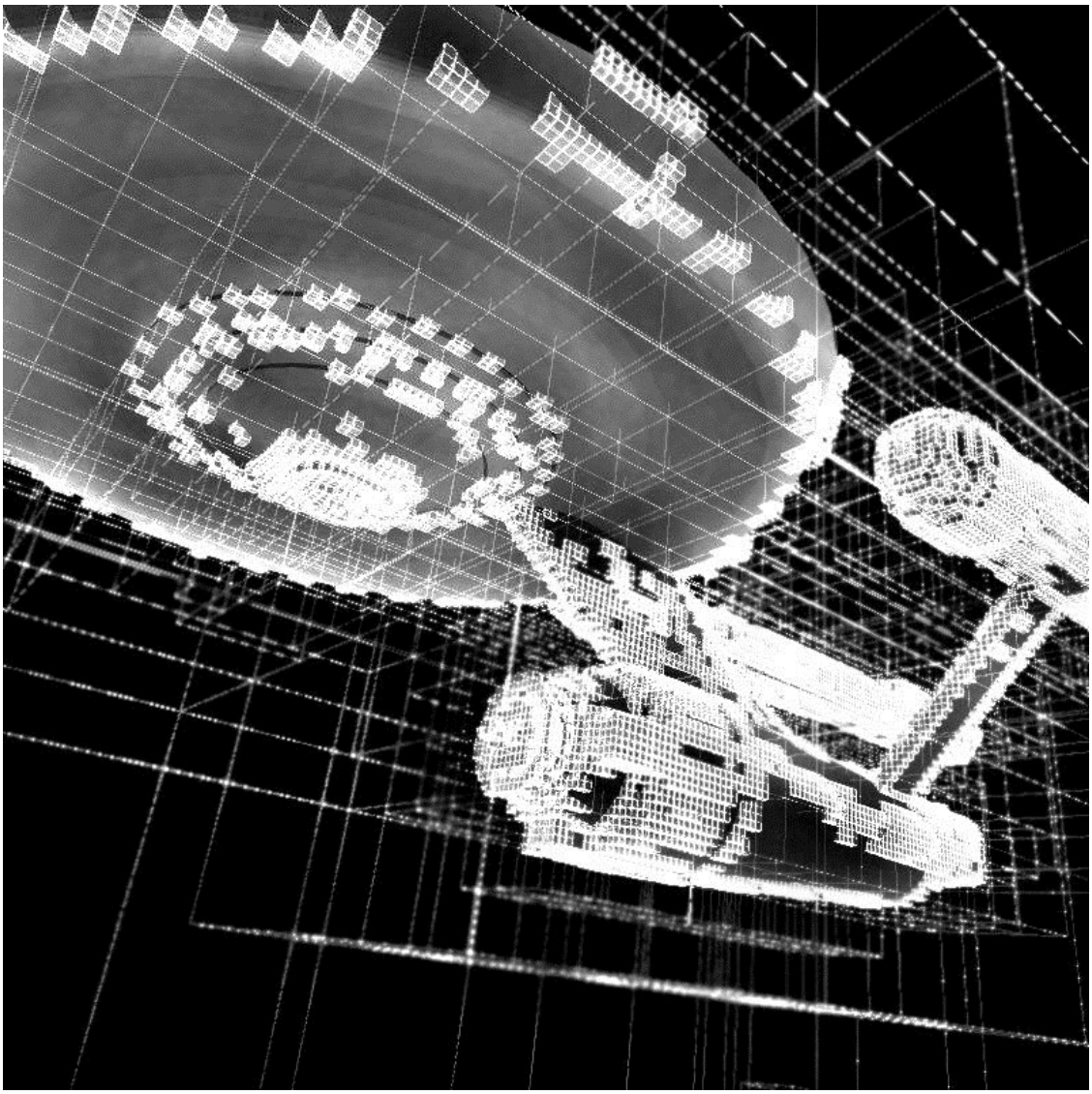


图 10 八叉树的实现 @http://thomasdiewald.com/blog/?p=1488

上述三幅图也均来自《Space Partitioning: Octree vs.BVH》http://thomasdiewald.com/blog/?p=1488 一文，这是一篇比较八叉树和 BVH 的有趣的文章，有兴趣的朋友可以阅读一下。

11.1.3.1 松散八叉树 Loose Octrees

松散八叉树的基本思想和普通八叉树一样，但是每个长方体的大小选中比较宽松。而如果一个普通长方体的边长为 l，那么可以用 kl 来代替，其中 k>1，如下图所示。

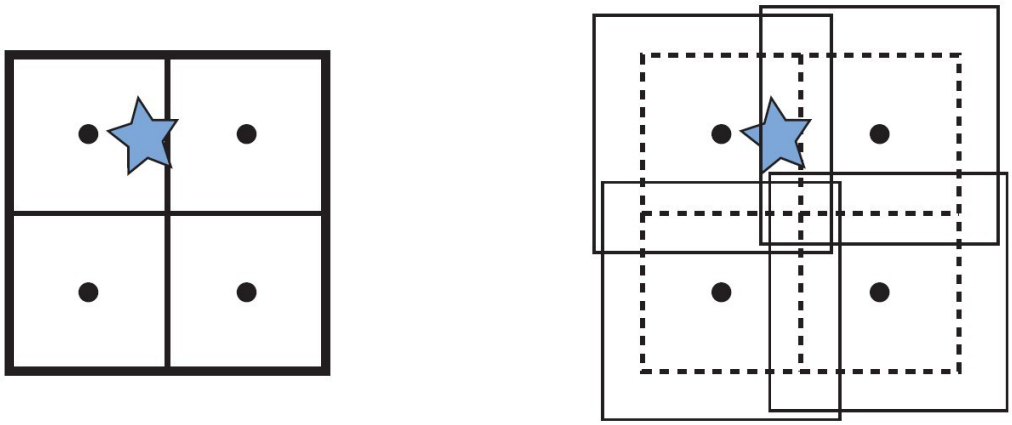


图 11 一个普通八叉树和松散八叉树的比较。图中黑色的原点表示长方形的中心点（第一次细分）。在左图中，星形物体刺穿了一个普通八叉树的一个分割平面。，这样，一种选择就是将这个星型物体放在最大的长方形中（根节点的长方体）。而右图所示为一个 k=1.5 的松散八叉树，也就是将长方体放大了 50%，如果将这些长方体稍微移动，就可以保证区分出它们。这样，这个星型多边形就完全位于左上角的长方形之中。

11.1.3.2 八叉树延伸阅读材料

[1] http://web.cs.wpi.edu/~matt/courses/cs563/talks/color_quant/CQoctree.html

[2] https://en.wikipedia.org/wiki/Octree

[3] Losasso F, Gibou F, Fedkiw R.Simulating water and smoke with an octree data structure[C]//ACM Transactionson Graphics (TOG). ACM, 2004, 23(3): 457–462.

11.1.4 场景图 | Scene Graphs

BVH、BSP 树和八叉树都是使用某种形式的树来作为基本的数据结构，它们的具体区别在于各自是如何进行空间分割和几何体的存储，且他们均是以层次的形式来保存几何物体。然而三维场景的绘制不仅仅是几何体。

然而，渲染三维场景不仅仅只是渲染出几何图形，对动画，可见性，以及其他元素的控制，往往需要通过场景图（Scene Graphs）来完成。

场景图被誉为“当今最优秀且最为可重用的数据结构之一。”Wiki 中的对场景图的定义是“场景图（Scene Graph）是组织和管理三维虚拟场景的一种数据结构，是一个有向无环图（Directed Acyclic Graph， DAG）。”

场景图是一个面向用户的树结构，可以通过纹理、变换、细节层次、渲染状态（例如材质属性）、光源以及其他任何合适的内容进行扩充。它由一棵以深度优先遍历来渲染整个场景的树来表示。

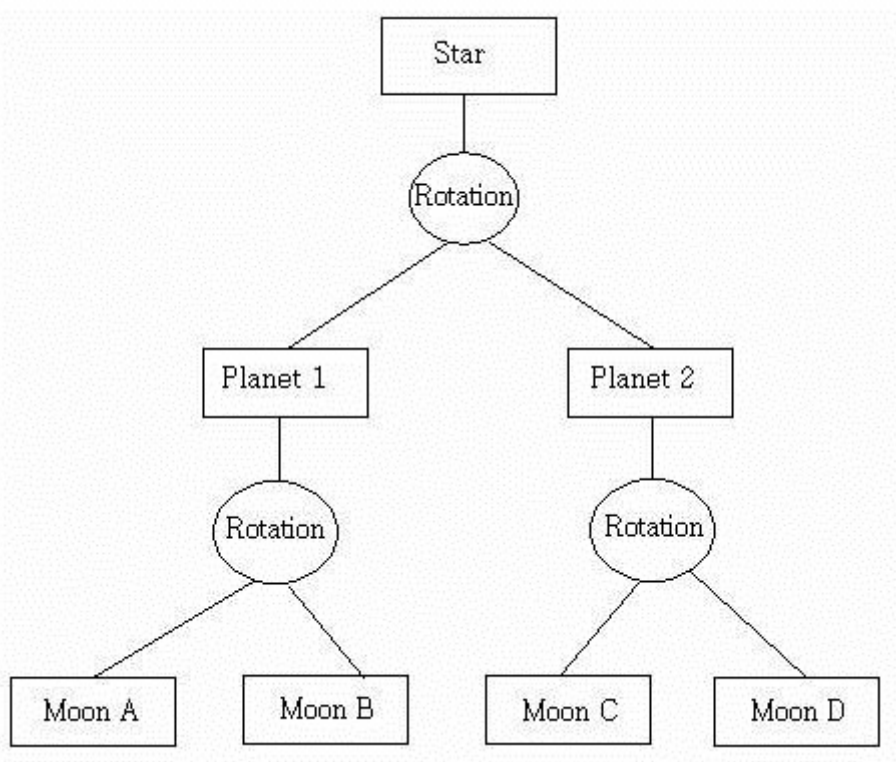


图 12 通过创建场景图来表示对象

另外提一句，开源的场景图有 Open Scene Graph 和 OpenSG 等，有兴趣的朋友们可以进行进一步了解。

11.1.4.1 场景图的延伸阅读材料

[1] http://www.openscenegraph.org/index.php/documentation/knowledge-base/36-what-is-a-scene-graph

[2] https://en.wikipedia.org/wiki/Scene_graph

[3] http://archive.gamedev.net/archive/reference/programming/features/scenegraph/index.html

11.2 裁剪技术 | Culling Techniques

裁剪（Culling）的字面意思是“从大量事物中进行删除”。在计算机图形学中，相对应的就是裁剪技术（Culling Techniques）所要做的工作——“从大量游戏事物中进行删除”。所谓的“大量事物”就是需要绘制的整个场景，删除的是对最终图像没有贡献的场景部分，然后将剩余场景发送到渲染管线。因此，在渲染方面通常使用“可见性裁剪（Visibility Culling）”这个术语。但其实，裁剪也可以用于程序的其他部分，如碰撞检测（对不可见物体进行不十分精确的计算）、物理学计算，以及人工智能（AI）领域。

与渲染相关的裁剪技术，常见的有背面裁剪（Backface Culling），视锥裁剪（View Frustum Culling），以及遮挡裁剪（Occlusion Culling，也常常称作遮挡剔除）：

- 背面裁剪即是将背向视点的物体删除，是一种非常简单直观的操作，只能一次一对个单一多边形进行操作。
- 视锥裁剪是将视锥之外的多边形删除，相对而言，这种操作比背面裁剪稍微复杂。
- 遮挡裁剪，是将被其他物体遮挡的物体进行删除，这种操作在三者中最为复杂，因为其需要聚集一个或者多个物体，同时还需使用其他物体的位置信息。

理论上，裁剪操作可以发生在渲染管线的任何一个阶段，而且对于一些遮挡裁剪算法来说，甚至可以预先计算出来。对于在硬件中实现的裁剪算法来说，有时只需启动/禁止或者设置一些裁剪函数即可。而为了进行完全控制，我们可以在应用程序阶段在（CPU 上）实现一些裁剪算法。假设瓶颈位置不在 CPU 上，渲染最快的多边形就是没有送到图形加速器管线上的多边形，裁剪通常可以使用几何计算来实现，但也不局限于此。例如，某算法也可以使用帧缓冲中的内容。而理想的裁剪算法预期是只发送所有图元中通过管线的精确可见集(Exact Visible Set, EVS)。

下图是三种裁剪技术的对比图示。

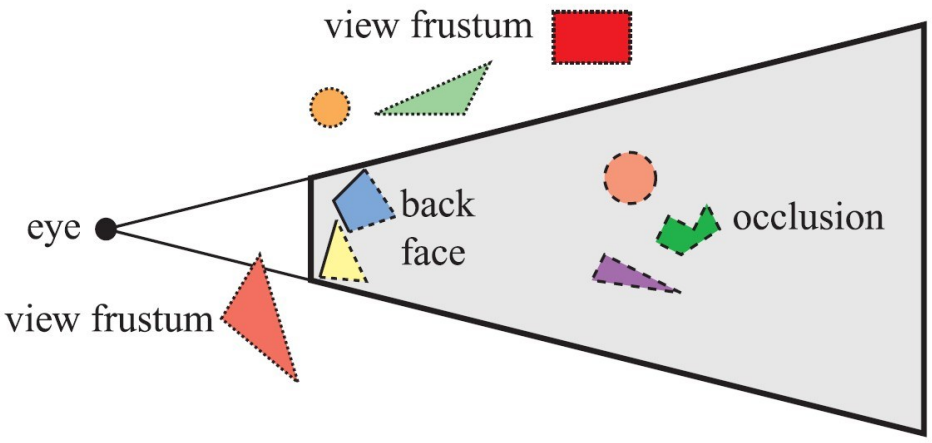


图 13 三种裁剪技术的对比，其中被裁剪掉的几何体用虚线表示

下文接下来将分别介绍背面裁剪、层次视锥裁剪、入口裁剪、细节裁剪、遮挡剔除等几种裁剪技术。

11.3 背面裁剪 | Backface Culling

假设你正在观察一个场景中不透明的球体。大约有一半的球体是不可见的。那么，可以从中这个例子里得到一个众所周知的结论，那就是，对不可见的内容不需要进行渲染，因为它们对最终的渲染图像没有贡献。不需要对球体的背面进行处理，这就是背面裁剪的基本思想。对于一组物体来说，还可以一次性地进行背面裁剪，这也称为聚集背面裁剪（Clustered Backface Culling）。

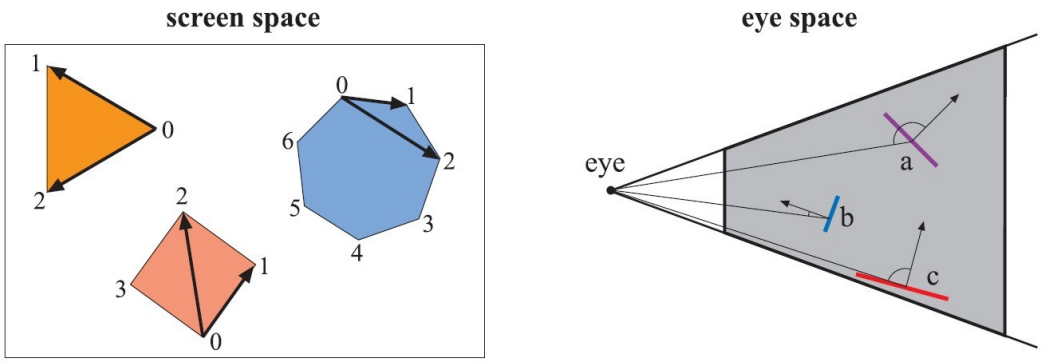


图 14 确定多边形是否背向的两种不同测试。左图所示为屏幕空间的测试情形，三角形和四边形是正向，而七边形是背向。对背向的七边形，无需进行光栅化。

右图为视点空间中背面测试情形，多边形 A 是背向的，而 B 和 C 是正向的。多背向的多边形 A，无需进行光栅化。

11.4 层次视锥裁剪 | Hierarchical View Frustum Culling

如上文所示，只需对完全或者部分在视锥中的图元进行渲染。一种加快渲染速度的方法便是将每个物体的包围体与视锥进行比较，如果包围体位于视锥之外，那么便不需要渲染包围体中的几何体。由于这些计算在 CPU 上进行，因此包围体中的几何体不需要通过管线中的几何和光栅阶段。相反，如果包围体在视锥内或者与视锥相交，那么包围体中的内容就是可见的，所以必须发送到渲染管线中去。

利用空间数据结构，可以分层地来应用这种裁剪。例如，对于层次包围体 BVH 来说，从根节点进行先序遍历（Preorder Transversal），就可以完成这一任务。

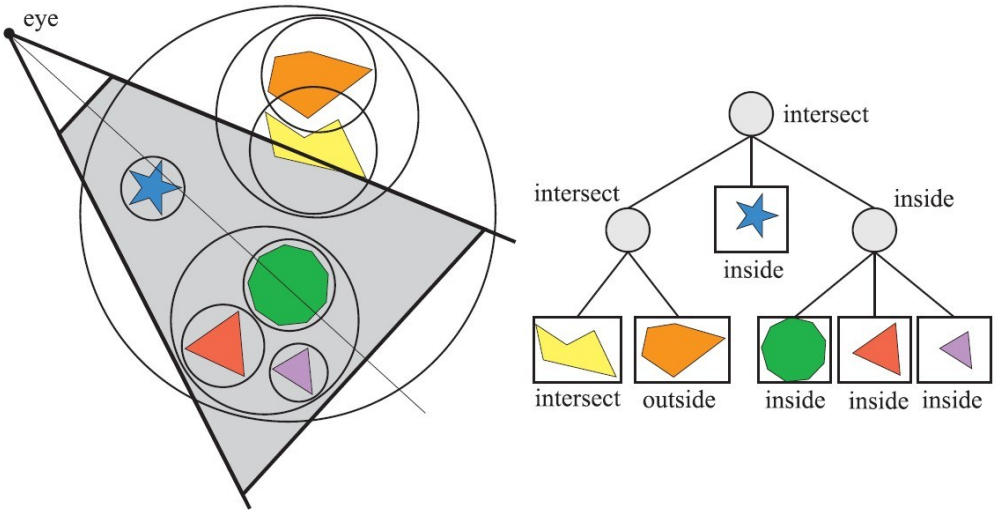


图 15 左图所示为一组几何体和相应的包围体（球体），从视点位置使用视锥裁剪来渲染场景。右图所示为层次包围体，视节点的包围体与视锥相交，对子包围体进行遍历测试，左子树的包围体与视锥相交，而其中只有一个子节点与视锥相交，另外一个子包围体体外边，无需发送到管线。根节点中间子树的包围体完全位于视锥内部，可以立即进行渲染，右边的子树的包围体也完全位于视锥内部，所以不需要进一步测试就可以渲染整个子树。

视锥裁剪操作位于应用程序阶段（CPU），这意味着几何阶段和光栅阶段都可以从中受益，对于大场景或者一定的相机视线来说，场景只有一小部分是可见的，只需要将这部分发送到渲染管线。可期望获得一定的加速效果，视锥裁剪技术利用了场景中的空间相关性，因为可以将彼此靠近的物体包围在一个包围体中，而且几乎所有包围体都是以层次形式聚集在一起。

除了层次包围体，其他的空间数据结构同样也可以用于视锥裁剪，包括上文提到的八叉树和 BSP 树。但是当渲染动态场景时，这些方法便会显得不够灵活,不如层次包围体。

11.5 入口裁剪 | Portal Culling

对建筑物模型来说，很多裁剪方面的算法可以归结为入口裁剪（Protal Culling）。在这个方向，最早的算法由 Airey 提出，随后 Teller 和 Sequin，以及 Teller 和 Hanrahan 构造出了更高效，更复杂的算法。

入口裁剪算法的基本思想是，在室内场景中，建筑物墙面通常充当大的遮挡物，通过每个入口（如门或者窗户）进行视锥裁剪。当遍历入口的时候，就减小视锥。

使得与入口尽可能紧密贴合。因此，可以将入口裁减算法看作是视锥裁剪算法的一种扩展，且需将位于视锥之外的入口丢弃。

入口裁剪方法以某种方式对场景进行预处理，可以是自动形式，也可以是手动形式，可以将场景分割为一系列单元（Cells），其通常对应于建筑物中的房间或者走廊；链接进阶房间的门和窗口称为入口（Protals）。单元中的每个物体和单元的墙面可以存储在一个与单元关联的数据结构中，还可以将邻接单元和链接这些单元的入口信息保存在一个临接图中。

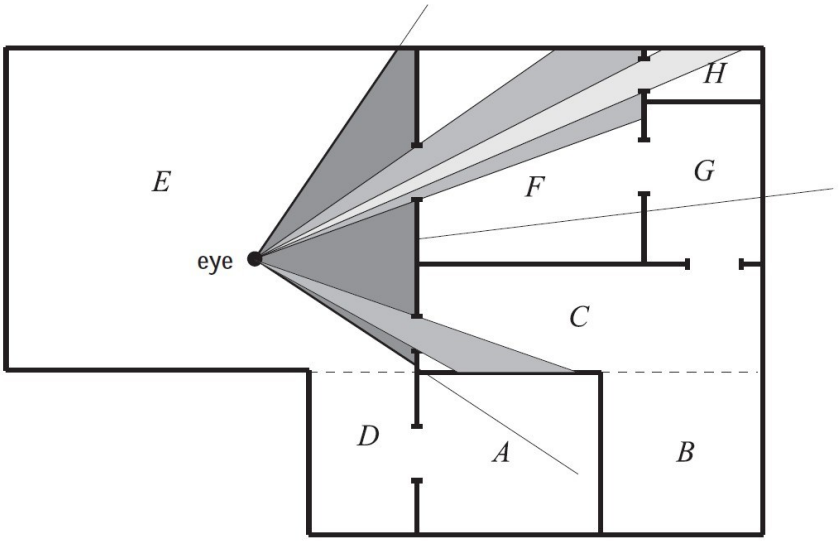


图 16 入口裁剪。单元分别从 A 到 H，入口是连接单元的通路，只对穿过入口能看到的几何体进行渲染。

图 17 入口裁剪，左图为房间顶视图，白线表示每一个入口的截锥体减少的方式。红线是在镜子上反射圆台来产生的。实际视图显示在右侧的图像中。

11.6 细节裁剪 | Detail Culling

细节裁剪（Detail Culling）是一种通过牺牲质量换取速度的技术。其基本原理是，当视点处于运动的时候，场景中的微小细节对渲染出的图像贡献甚微。且当视点停下来的时候，通常禁止细节裁剪。

考虑一个具有包围体的问题，将这个包围体投射到投影平面，然后以像素为单位来估算投影面积，如果像素的数量小于用户定义的阈值，那么不对这个物体进行进一步处理。基于这个原因，细节裁剪也往往被称为屏幕尺寸裁剪（Screen-Size Culling）。另外，细节裁剪也可以在场景图上以层次形式来实现，几何阶段和光栅阶段都可以从这个算法中受益。

细节裁剪还可以作为一种简化的 LOD 技术来实现，其中一个 LOD 是整个模型，另外一个 LOD 是空物体。

11.7 遮挡剔除 | Occlusion Culling

遮挡裁剪（Occlusion Culling），也常被称作遮挡剔除。

聊一聊遮挡剔除必要性。不难理解，可见性问题可以通过 Z 缓冲器的硬件构造来实现，即使可以使用 Z 缓冲器正确解决可见性问题，但其中 Z 缓冲并不是在所有方面都不是一个很“聪明”的机制。例如，假设视点正沿着一条直线观察，其中，在这条直线上有 10 个球体，虽然这 10 个球体进行了扫描转换，同时与 Z 缓冲器进行了比较并写入了颜色缓冲器和 Z 缓冲器，但是这个从这个视点渲染出的图像只会显示一个球体，即使所有 10 个球体都将被光栅化并与 Z 缓冲区进行比较，然后可能写入到颜色缓冲区与 Z 缓冲区。

下图中间部分显示了在给定视点处场景的深度复杂度，深度复杂度指的是对每个像素重写的次数。对于有 10 个球体的情形，最中间的位置，深度复杂度为 10，因为在这个地方渲染了 10 个球体（假设背面裁剪是关闭的），而且这意味着其中有 9 次像素写入是完全没有必要的。

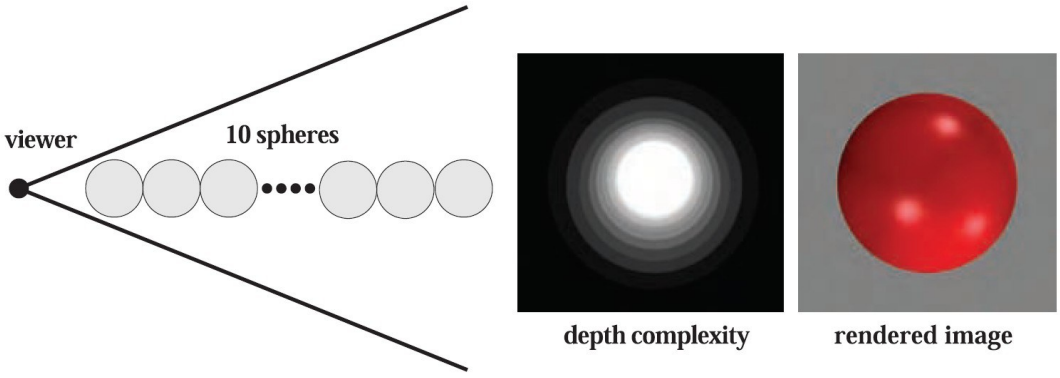


图 18 展示遮挡剔除必要性的图示

像上图这样无聊极端的场景，现实生活中很难找到，但其描述的这种密集性很高的模型的情形，在现实生活中却很常见，如热带雨林，发动机，城市，以及摩天大楼的内部。下图显示了曼哈顿式城市的示例。

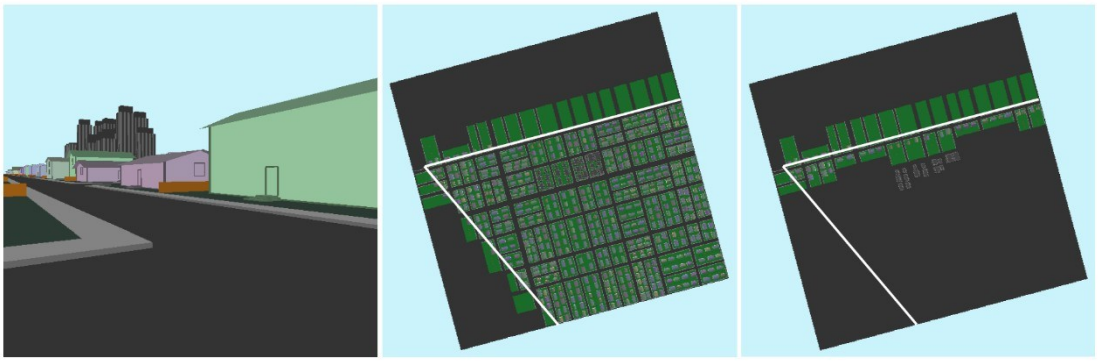


图 19 城市鸟瞰图，左图为视锥裁剪后的图示，中图为视锥裁剪后的图示，右图所示为遮挡剔除和视锥裁剪后的图示

从上面给出的示例可以看出，这种用来避免低效率的算法可以带来速度上的补偿，具体可以将这些方法归类为遮挡剔除算法（Occlusion Culling Algorithms），因为它们都试图裁剪掉被遮挡的部分，也就是被场景中其他物体遮挡的物体，最优的遮挡裁剪算法只选择其中可见得的部分。

有两种主要形式的遮挡裁剪算法，分别是基于点的遮挡裁剪和基于单元的遮挡裁剪。如下图所示。

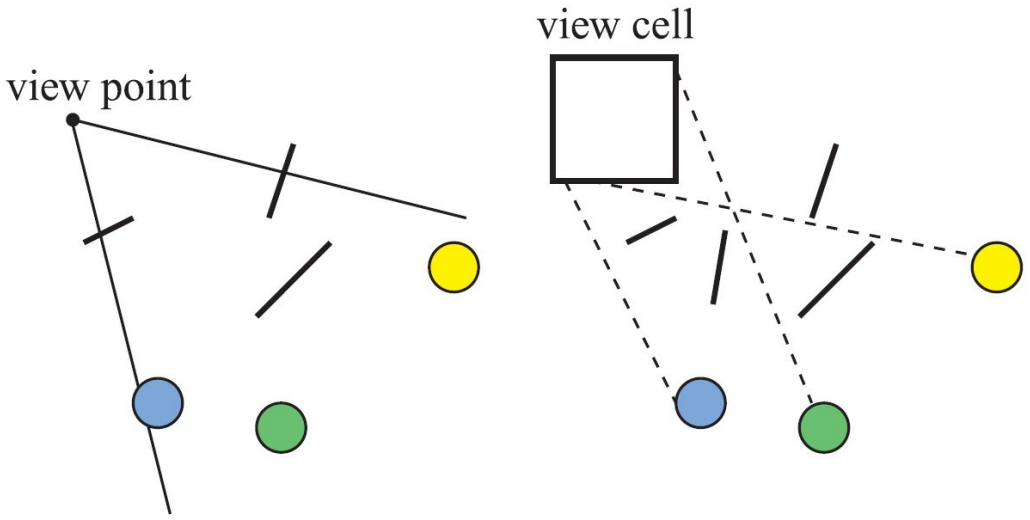


图 20 左图所示为基于点的可见性，右图所示为基于单元的可见性，其中单元是一个长方形，从中可以看出，从视点左边看上去，有些圆被遮挡了，但是从右边看上去，这些圆却是可见的，因为可以从单元的某个位置到这些圆画一些射线，这些射线没有和任何遮挡物相交。

下图所示为一种遮挡剔除算法的伪代码。

```
1:    OcclusionCullingAlgorithm(G)
2:    O_R =empty
3:    P =empty
4:    for each object g ∈ G
5:       if(isOccluded(g,O_R))
6:          Skip(g)
7:       else
8:          Render(g)
9:          Add(g, P)
10:         if(LargeEnough(P))
11:            Update(O_R, P)
12:            P =empty
13:         end
14:      end
15:   end
```

前人在遮挡剔除方面已经做了大量的工作，有多种不同种类的遮挡剔除算法：

- Hardware Occlusion Queries 硬件遮挡查询
- Hierarchical Z–Buffering 层次 Z 缓冲
- Occlusion Horizons 遮挡地平线
- Occluder Shrinking 遮挡物收缩
- Frustum Growing 视锥扩张
- Virtual occluder 虚拟遮挡物算法
- Shaft Occlusion Culling 轴遮挡裁剪
- The HOM algorithm 层次遮挡映射算法
- Ray Space Occlusion Culling 射线空间遮挡剔除

下面将对其中的常见几种介绍。

11.7.1 硬件遮挡查询 | Hardware Occlusion Queries

现代 GPU 可以以一种特殊的渲染模式来支持遮挡剔除。通过硬件遮挡查询（Hardware Occlusion Queries），我们能够直接获得所提交的物体是否被绘制到场景中。

简单来说，硬件遮挡查询的基本思想是，当和 Z 缓冲器中内容进行比较时，用户可以通过查询硬件来找到一组多边形是否可见的，且这些多边形通常是复杂物体的包围体（如长方体或者 k-DOP）。如果其中没有多边形可见，那么便可将这个物体裁剪掉。硬件实现对查询的多边形进行光栅化，并且将其深度和 Z 缓冲器进行比较。

更多细节，可以参考这篇论文：

Bittner J, Wimmer M, Piringer H, et al. Coherent hierarchical culling: Hardware occlusion queries made useful[C]//Computer Graphics Forum. Blackwell Publishing, Inc, 2004, 23(3): 615-624.

11.7.2 层次 Z 缓冲 | Hierarchical Z-Buffering

层次 Z-缓冲算法（Hierarchical Z-Buffering ，HZB）是由 Greene 等人提出的一种算法，对遮挡剔除的研究有着显著的影响。尽管其在 CPU 上很少使用，但该算法是 GPU 上做 Z-Culling（深度裁剪）的基础。

层次 Z-缓冲算法用八叉树来维护场景模型，并将画面的 Z 缓冲器作为图像金字塔（也称为 Z-金字塔（Z-pyramid）），该算法因此在图像空间中进行操作。其中，八叉树能够对场景的遮挡区域进行层次剔除，而 Z-金字塔则可以对单个基元和边界体积进行层次 Z 缓冲。 因此 Z-金字塔可以作为此算法的遮挡表示。

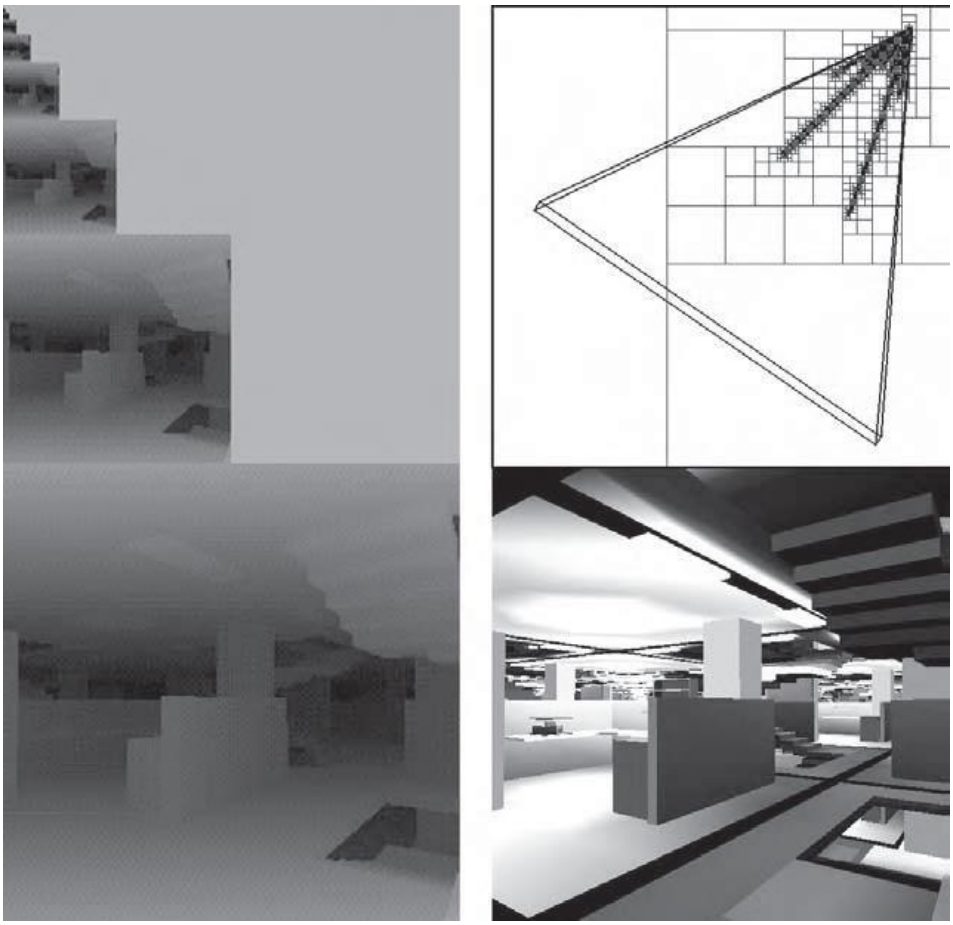


图 21 使用 HZB 算法的遮挡裁剪示例，显示了一个复杂的场景（右下），相应的 Z-pyramid(左图)，以及八叉树细分（右上）。通过从前到后遍历八叉树并裁剪遇到的八叉树节点，此算法可以仅访问可见的八叉树节点及其子节点（右上角的节点），的容器只对可见包围体中的多边形进行渲染。在这个例子中，遮挡八叉树节点的裁剪可以将深度复杂度从 84，降低到了 2.5。

更多细节，可以参考这篇论文：

Greene N, Kass M, Miller G. Hierarchical Z-buffer visibility[C]//Proceedings of the 20th annual conference on Computer graphics and interactive techniques. ACM, 1993: 231-238.

11.7.3 其他遮挡剔除技术 | Other Occlusion Culling Techniques

前人在遮挡剔除方面已经做了大量的工作，但由于 GPU 的性能早已超过了 CPU，所以这些算法中的大部分已经不再受青睐。因此这边只对一些常见的方案做一些简单介绍，至少他们还是值得传递下去的一些知识，因为架构和硬件的不断发展。

而随着多核系统的崛起，CPU 端有了额外的资源，但难以直接给渲染本身带来提升，但同时使用单核或多核来执行基于单元的可见性测试或者其他进行方案，也已变得可以想象。

11.7.3.1 层次遮挡映射算法 | Hierarchical Occlusion Map

层次遮挡映射（Hierarchical Occlusion Map ，HOM)）算法，类似层次 Z 缓冲算法，是一种启用分层图像空间剔除的方法。但其也不同于层次 Z 缓冲，因为它提供了使用近似遮挡剔除的能力。 HOM 算法的基本思想是，每帧建立一个分层深度缓冲区，用于遮挡测试。并且在每个级别使用不透明度阈值来确定是否有足够的要渲染的对象是可见的。如果只有一小部分的对象是可见的，那么该对象被剔除。但作为一个基于 CPU 的算法系统，这个算法已经不受欢迎了。

对此算法感兴趣的朋友，可以进一步参考这篇论文：

Zhang H, Manocha D, Hudson T, et al. Visibility culling using hierarchical occlusion maps[C]//Proceedings of the 24th annual conference on Computer graphics and interactive techniques. ACM Press/Addison-Wesley Publishing Co., 1997: 77-88.

11.7.3.2 遮挡地平线算法 | Occlusion Horizons

遮挡地平线（Occlusion Horizons）算法是一种非常简单的、基于点的可见性算法，可以对遮挡物进行融合，在基于点的可见性算法演示方面非常有用。由 Wonka 和 Schmalstieg 等人首先提出，并通过图形硬件将其进行了实现，随后 Downs 等人使用几何计算的方法将其独立开发实现，最早于 1995 年在电脑游戏中使用。

顾名思义，遮挡地平线算法的基本思想是裁剪掉位于地平线之间和之下的物体。 这种类型的算法经常被用来高效绘制如城市和村庄一样的城市环境。

通过从前到后渲染一个场景，我们可以定位到地平线在哪里进行渲染，而任何在当前地平线之后和之下的物体都可以被裁剪掉。

对此算法感兴趣的朋友，可以进一步参考这篇论文：

Downs L, Möller T, Séquin CH. Occlusion horizons for driving through urban scenery[C]//Proceedings of the2001 symposium on Interactive 3D graphics. ACM, 2001: 121-124.

11.7.3.3 遮挡物收缩与视锥扩张算法 | Occluder Shrinking and Frustum Growing

上文给出的遮挡地平线算法是基于点的可见性来判断的。有些时候采用基于单元的可见性方法更合适，但基于单元通常比基于点的可见性计算复杂度要高得多。Wonka 等人提出了一种称为遮挡物收缩（Occluder Shrinking）的方法，可以使用基于点的遮挡算法来生成基于单元的

可见性，根据给定的量来缩小场景中所有遮挡物来达到延伸有效可见点的目的。他们也提出了一种视锥扩张（Frustum Growing）技术，通常与 Occluder Shrinking 算法一起配合使用。

对此算法感兴趣的朋友，可以进一步参考如下三篇论文：

[1] Wonka P, Wimmer M, Schmalstieg D. Visibility preprocessing with occluder fusion for urban walk throughs[M]//Rendering Techniques 2000. Springer, Vienna, 2000: 71–82.

[2] Wonka P, Wimmer M, Sillion F X. Instantvisibility[C]//Computer Graphics Forum. Blackwell Publishers Ltd, 2001, 20(3):411–421.

[3] Wonka, Peter, Occlusion Culling for Real–Time Rendering of Urban Environments,Ph.D. Thesis, The Institute of Computer Graphics and Algorithms, Vienna University of Technology, June, 2001. Cited on p. 679

11.8 层次细节 | LOD，Level of Detail

细节层次（Level of Detail,LOD）的基本思想是当物体对渲染出图像贡献越少，使用越简单的形式来表达该物体。这是一个已经在各种游戏中广泛使用的基本优化技术。

例如，考虑一个包含 1 万个三角形的汽车，其中所包含的细节信息比较丰富。当视点靠近物体时，可以使用详细的细节表示，而当视点远离物体时，比如仅需覆盖 200 个像素，则完全无需渲染出 1 百万个三角形，相反，我可以使用诸如只有 1000 个三角形的简化模型。而由于距离的原因，简化后的模型与细节较丰富的模型看上去其实很接近。以这种方式，可以显著地提高渲染的性能开销。

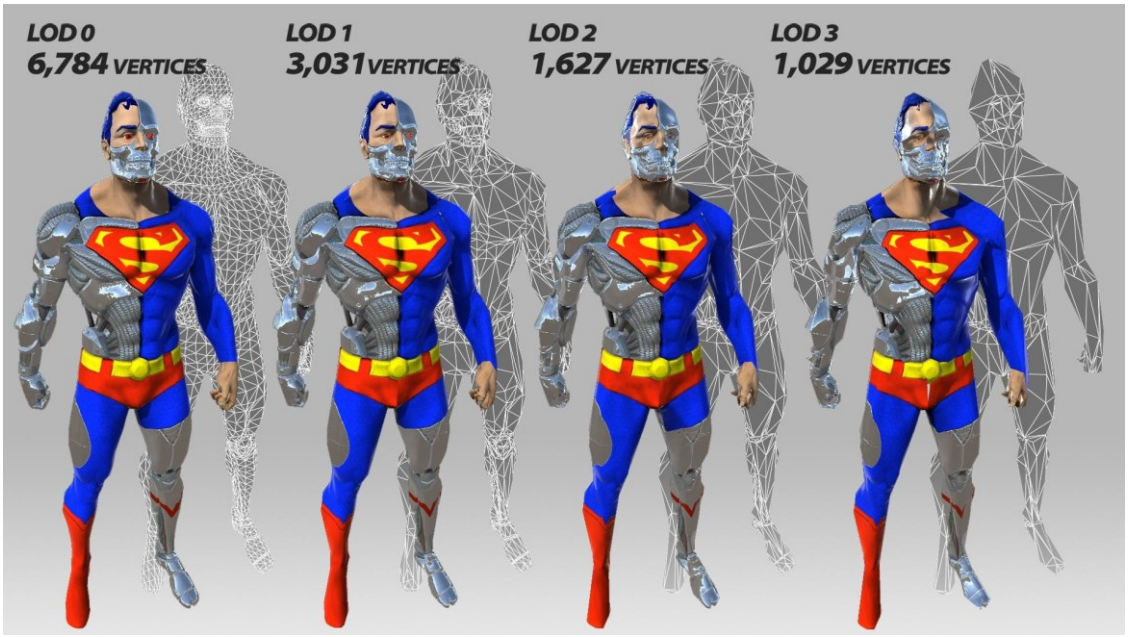


图 22 LOD 图示

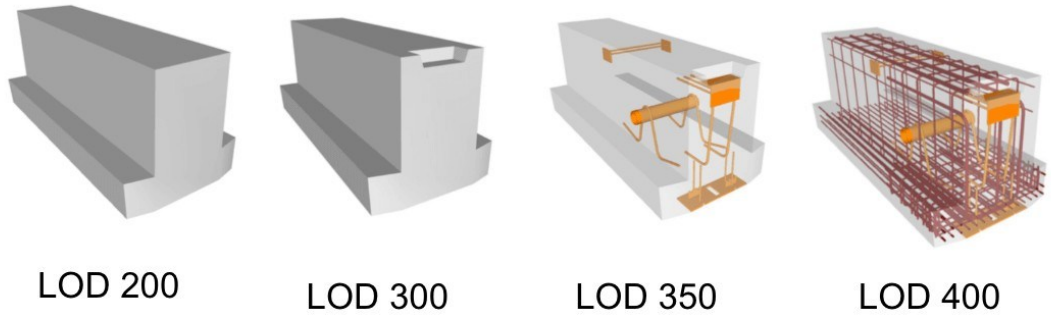


图 23 LOD 图示

NVIDIA 提供了一个非常有趣的网站，可以在网页上自己拖动分界线进行交互，查看《古墓丽影:崛起》的游戏画面不同级别的 LOD 显示情况，可以对 LOD 有一个很直观的认识，感兴趣的同学不妨试试：

http://images.nvidia.com/geforce-com/international/comparisons/rise-of-the-tomb-raider/alt/rise-of-the-tomb-raider-level-of-detail-interactive-comparison-001-very-high-vs-low-alt.html

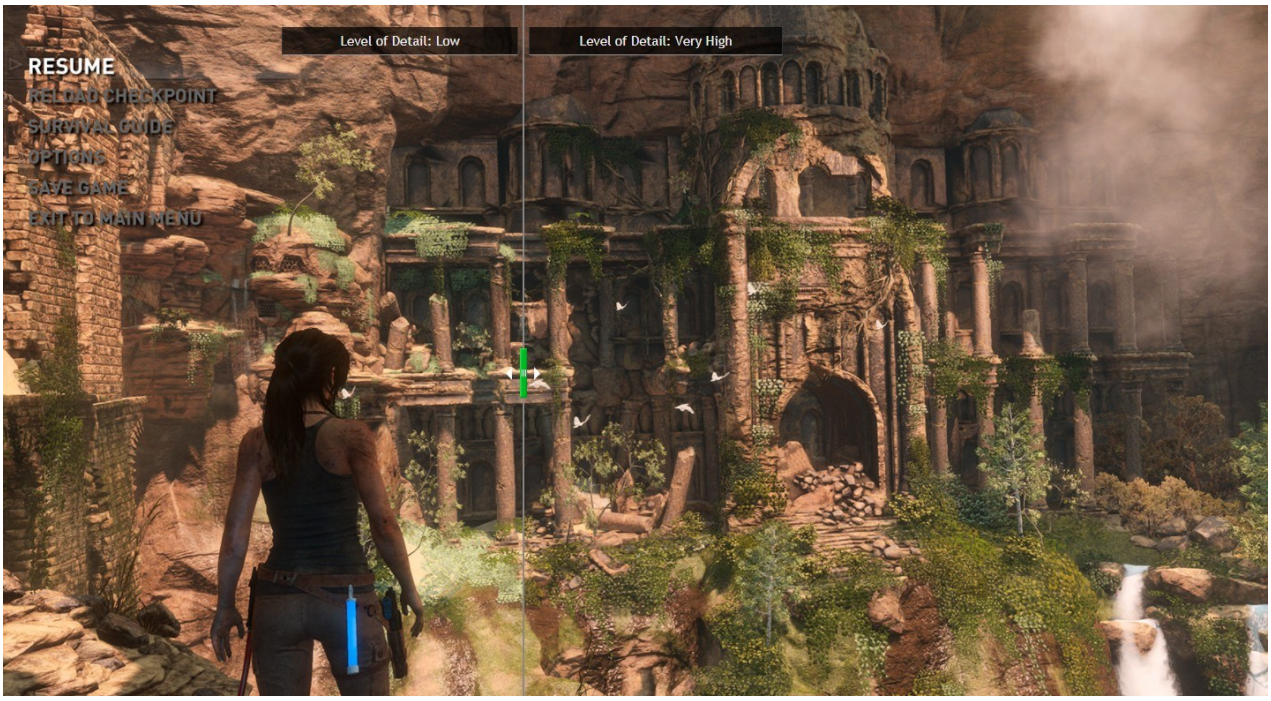


图 24 《古墓丽影:崛起》不同 LOD 的在线对比网站 @NVIDIA

通常情况下，雾效会与 LOD 一起使用。这样我们可以完全跳过对一些物体的渲染，直接用不透明的雾来进行遮挡。另外，雾效的机制可以实现下文所介绍到的时间临界 LOD 渲染

（Time-Critical LOD Rendering）。通过将元平面移近观察者，可以更早地剔除对象，并且可以实现更快速的渲染以保持帧速率。

一般情况下，完整的 LOD 算法包含 3 个主要部分：

- 生成 Generation
- 选择 Selection
- 切换 Switching

其中，LOD 的生成就是生成不同细节的模型表示。RTR3 书中 12.5 节中讨论的简化方法可用于生成所需数量的 LOD。另一种方法是手工制作具有不同数量的三角形模型。选择机制就是基于某种准则选取一个层次细节模型，比如屏幕上的评估面积。最后，我们还需要从一个细节层次转换到另一个细节层次，而这个过程便称为 LOD 切换。

下面对 LOD 的切换和选取相关的算法进一步说明。

11.8.1 LOD 的切换 | LOD Switching

当从一个 LOD 切换到另一个 LOD 的时候，忽然的模型替换往往会引起观察者的注意。这种现象被称为突越（Poping）。这里有几种不同的 LOD 切换方法，有着不同的特性：

- 离散几何 LOD | Discrete Geometry LODs
- 混合 LOD | Blend LODs
- 透明 LOD | Alpha LODs
- 连续 LOD 和几何形变 LOD | CLODs and Geomorph LODs

11.8.1.1 离散几何 LOD | Discrete Geometry LODs

离散几何 LOD 是最简单的 LOD 算法，不同的表示是不同图元数量的同一模型，但这种方法突越现象严重。

11.8.1.2 混合 LOD | Blend LODs

在概念上，完全可能存在一种直观的方法，从一个 LOD 切换到另一个 LOD，只需要在较短的时间内在两个 LOD 之间执行一个线性混合，这种方法无疑可以得到一种比较平滑的切换，但是这种混合操作的代价较高。渲染两个 LOD 要比一个 LOD 需要更大开销，因此也就违背了

LOD 的初衷。但 LOD 切换通常发生在较短时间内容，在同一时间也不是对场景中所有物体进行切换，所以依然可以从中获益。

Giegl 等人在《Unpopping: Solvingthe Image-Space Blend Problem for Smooth Discrete LOD Transition 》这篇文章中提出了一种方法，实际应用的效果较为出色。思路是在两个 LOD 之间有一个 alpha 值的过渡，有兴趣的朋友可以进一步了解。

11.8.1.3 透明 LOD | Alpha LODs

完全避免突越现象的一种简单方法便是使用 alpha LOD。其中并没有使用同一物体很多不同细节的实例，而且每个物体只有一个实例。

随着 LOD 选取度量值（如与物体之间的距离）的增大，物体整体透明度也随之增大（也就是 alpha 值减小），当完全透明时，物体最终就会消失。

这种方法的优点是，比离散几何 LOD 方法上感觉更连续一些，可以避免突跃现象。此外，由于物体最终会完全消失而不需要进行渲染，可以得到很好的加速效果。

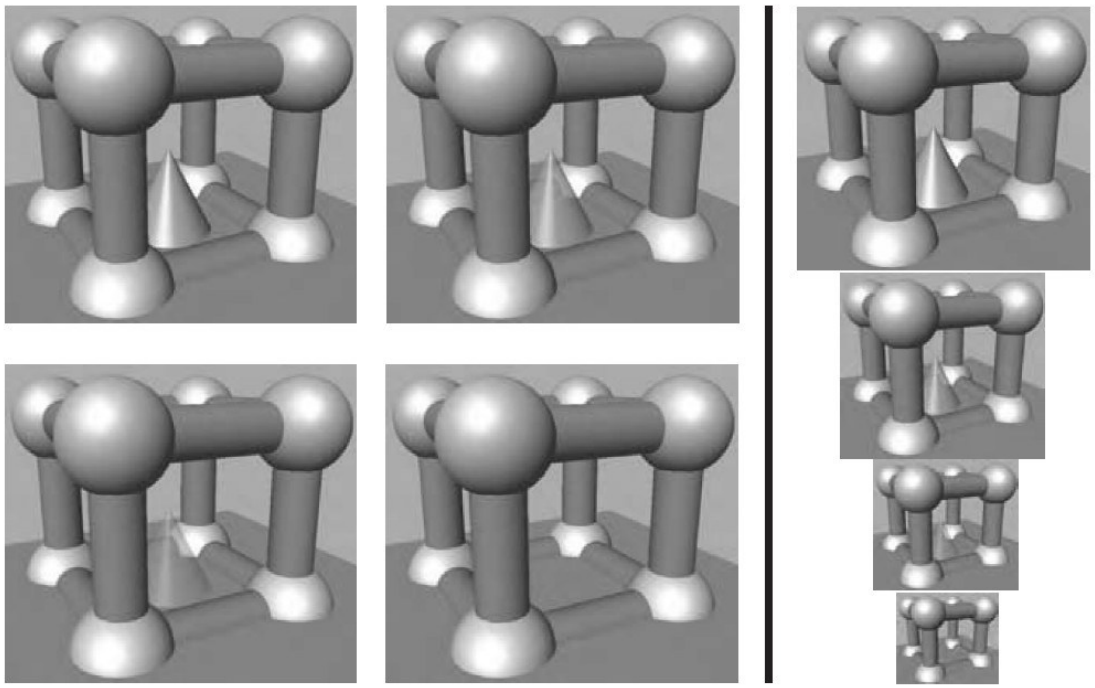


图 25 使用 Alpha LOD 对图中的圆锥体进行渲染，当距离圆锥体较远时，就提高它的透明度，直到最后消失。直线左边的图像是从同一距离处进行的观察，而直线右边的图像是左边图像不同尺寸的情形。

11.8.1.4 连续 LOD 和几何形变 LOD | CLODs and Geomorph LODs

连续细节层次（Continuous Level of Detail , CLOD）的基本思想是基于 LOD 选取值来精确决定可见多边形的数量。在 100m 远处，模型包含 1000 个多边形，当移动的到 101m 的地方时，模型减少到 998 个多边形。

几何形变层次细节（Geomorph LODs）是基于简化生成的一组离散模型，且其中模型顶点之间的链接关系保持不变。而网格简化的过程可以从一个复杂的物体中创建各种不同的 LOD 模型，具体做法可以参见《Real-Time Rendering 3rd》12.5.1 一节，一种方法就是创建一组离散的 LOD，然后按照上文中提到的方法来使用。这里的边塌陷（Edge Collapse Methods）方法有一个有趣的性质，即允许在不同的 LOD 之间采用其他过渡方法。

图 26 几何形变 LOD 的简化模型图示。左边和右边的图像所示分为为低细节层次和高细节层次的模型，中间的图像是在左右模型中间进行插值生成的几何变形模型。注意。中间的牛模型和右边的模型具有相同数量的顶点和三角形。

11.8.2 LOD 的选取 | LOD Selection

给定一个物体不同细节层次，必须做一个选择，决定渲染或者混合其中的哪一个层次，这就是 LOD 选择（LOD selection）的任务。有几种不同的 LOD 选择方案，这些方案也可以用于遮挡裁剪算法。

常见的三种 LOD 选取技术是：

- 基于距离的 LOD 选取(Range-Based)
- 基于投影面积的 LOD 选取(Projected Area-Based)
- 基于滞后的 LOD 选取(Hysteresis)

依然是分别进行简要概述。

11.8.2.1 基于距离的 LOD 选取 |Range-Based

选取 LOD 的一种常用方法是将物体的不同 LOD 于不同距离联系起来。细节最丰富的 LOD 的距离从 0 到一个用户定义值 r1 之间，下次层次的 LOD 的距离位于 r1~r2 之间，以此类推，如下图：

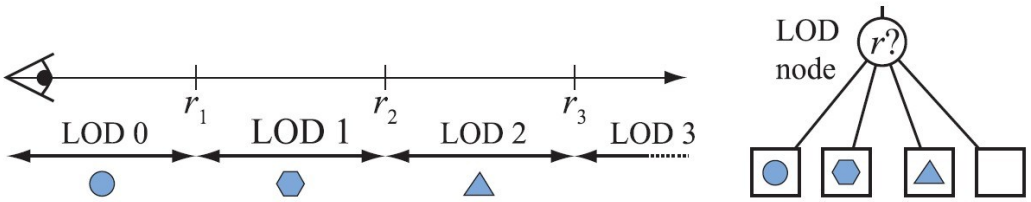


图 27 左图为基于距离的 LOD 的原理示例图。其中，LOD3 是一个空物体，也就是表示当物体大于 r3 时，就不渲染任何物体，因为物体对图像的贡献度不够。右图为场景中的一个 LOD 节点，它只有一个子节点基于 r。

11.8.2.2 基于投影面积的 LOD 选取 | Projected Area-Based

基于投影面积的 LOD 选取，顾名思义，即投影面积越大，就选取细节越丰富的 LOD。

11.8.2.3 基于滞后的 LOD 选取 | Hysteresis

若用于确定 LOD 度量标准围绕某个值 ri 在画面之间是变化的，那么就会出现不必要的突跃现象，也就会在不同的 LOD 之间来回快速切换。对此，可以引入一个围绕 ri 值的滞后来解决这个问题。如下图，这是一个基于距离的 LOD，可以应用于任何类型，当 r 增大时，使用上一行的 LOD 距离；当 r 减小时，使用下面一行的 LOD 距离。

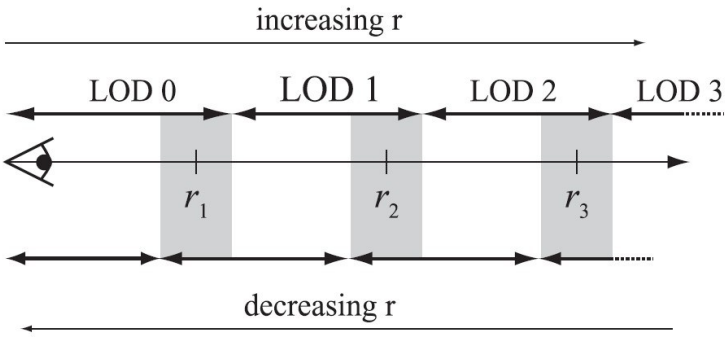


图 28 灰色区域表示的是基于滞后的 LOD 选取方法的滞后区域

11.8.2.4 其他 LOD 选取方法

基于距离和基于投影面积的 LOD 选取最常用。除了投影面积，Funk houser 等人《Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments》一文中还提出了使用物体的重要程度，运动，以及焦点等方法来作为 LOD 的选取方案。

观察者的注意力是一个重要的因素。例如，在一个体育游戏中，控制球的图像是用户最注意的地方，那么其他的部分就可以相对来说低的层次细节，具体可以参见论文《Never Let ' Em See You Pop — Issues in Geometric Level of Detail Selection,"》。

另外，也可以使用整体的可见性，如通过密集的叶子看到的附近对象可以用较低的 LOD 呈现。以及限制整体高级别 LOD 的数量以控制渲染的三角形总数的开销。其他的一些 LOD 选取的因素有颜色、以及纹理等。此外，也可以使用感知尺度来选择 LOD。

11.8.3 时间临界 LOD 渲染 | Time-Critical LOD Rendering

我们通常希望渲染系统有一个固定的帧率，实际上这就是通常所说的"硬实时（Hard Real Time）"或者时间临界（Time-Critical）。通常给定这类系统一个特定时间段（如 30ms），必须在这段时间内完成相应的任务（如图像渲染）；当时间到的时候，必须停止处理。如果场景中的物体用 LOD 来表示，则可以实现硬实时渲染算法。

Funk houser 等人在《Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments》一文中提出了一种启发式算法（heuristic algorithm），对于场景中的所有可见物体，可以自适应选取细节层次，从而满足固定帧率的要求。这个算法在场景中具有预测性，因为可见物体的 LOD 选取基于预期帧率和可见物体。这种启发式算法与对应的反应性算法（reactive algorithm）形成了鲜明对比，后者的 LOD 选取基于前一帧画面的渲染时间。

11.9 大型模型的渲染 | Large Model Rendering

人们一直都认为所渲染的模型是适合存放到计算机主内存中的，但通常的情况其实并非如此。一个简单的例子便是渲染一个地球模型。这是一个非常复杂的话题，《Real-Time Rendering 3rd》中也仅提了一下，然后列了一些相关文献，这里也仅简单说一下。

为了简单起见，大型模型的渲染通常会使用多个嵌套的数据结构，使用一个四叉树形式的数据结构来覆盖地球表面。而在每个叶节点内部可以根据具体内容使用不同的数据结构。此

外，为了保持合适的帧率，即将进入视野中的模型区域，在需要之前从磁盘中分页，而四叉树也可以在这里使用的。

值得一提的是，在 RTR3 书中 6.2.5 节(对应本系列文章第五篇的 5.4 节)讨论了裁剪图（clip-mapping）策略，便是管理大型纹理的一种技术。

将不同的加速算法进行结合是不容易的事情。Aliaga 等人将几种算法结合起来，用于非常大型的场景，具体可以参考如下 3 篇文章：

[1] Akeley, K., and T. Jermoluk, “High-Performance Polygon Rendering,” Computer Graphics (SIGGRAPH ’88 Proceedings), pp. 239 - 246, August 1988. Cited on p.22

[2] Akeley, K., P. Haeberli, and D. Burns, tomesh.c, a C-program on the SGI Developer’s Toolbox CD, 1990. Cited on p. 543, 553, 554

[3] Akeley, Kurt, “RealityEngine Graphics,” Computer Graphics (SIGGRAPH 93 Proceedings), pp. 109 - 116, August 1993. Cited on p. 126

而关于大型模型的渲染，有兴趣的朋友可以近一步参考这篇 SIGGRAPH 笔记：

Manocha D, Aliaga D. Interactive walkthroughs of large geometric datasets[J]. SIGGRAPH 00 Course notes, 2000.

11.10 点渲染 | Point Rendering

在 1985 年，Levoy 和 Whitted 写了一篇具有开创性的技术报告《The use of points asa display primitive》。在这篇报告中，他们提出点作为一种新的图元来进行渲染，基本思想是用一个大的点集来表示物体表面并予以渲染。在随后的通道中，使用高斯滤波来填充渲染点之间的间隙。而高斯滤波器的半径取决于表面上点的密度和屏幕上的投影密度。有兴趣的朋友可以进一步了解。PDF 地址在这里：

https://www.cs.princeton.edu/courses/archive/spring01/cs598b/papers/levoy85.pdf

图 29 根据点渲染的方法渲染出来的模型，使用原型油彩（circular splats）。左图为名为 Lucy 的天使模型，拥有 10 万个顶点。但在渲染中只用到了 300 万个油彩，中图、和右图是对左边图的放大。在渲染中，中间的图像使用 4 万个油彩，当视点停止移动时，就变成了右图，使用了 60 万个油彩（此图由 Szymon Rusinkiewicz 的 QSPlat program 产生，Lucy 的模型来自斯坦福图形实验室）

11.11 Reference

[1] Bittner J, Wimmer M, Piringer H, et al. Coherent hierarchical culling: Hardware occlusion queries made useful[C]//Computer Graphics Forum. Blackwell Publishing, Inc, 2004, 23(3): 615–624.

[2] Zhang H, Manocha D, Hudson T, et al. Visibility culling using hierarchical occlusion maps[C]//Proceedings of the 24th annual conference on Computer graphics and interactive techniques. ACM Press/Addison–Wesley Publishing Co., 1997: 77–88.

[3] 实时计算机图形学第二版[J]. 2004.

[4] Wonka P, Wimmer M, Schmalstieg D. Visibility preprocessing with occluder fusion for urban walkthroughs[M]//Rendering Techniques 2000. Springer, Vienna, 2000: 71–82.

[5] http://thomasdiewald.com/blog/?p=1488

[6] http://blog.csdn.net/skybreaker/article/details/1828104

[7] https://en.wikipedia.org/wiki/Bounding_volume_hierarchy

[8] Wonka P, Wimmer M, Sillion F X. Instant visibility[C]//Computer Graphics Forum. Blackwell Publishers Ltd, 2001, 20(3):
411–421.

[9] Wonka, Peter, Occlusion Culling for Real-Time Rendering of Urban Environments,Ph.D. Thesis, The Institute of Computer Graphics and Algorithms, Vienna University of Technology, June, 2001. Cited on p. 679

[10] http://insaneguy.me/attachments/spatial_ds--bsp_tree-octree-kd-tree.pdf

[11] http://book.51cto.com/art/201008/220506.htm

[12] http://blog.csdn.net/silangquan/article/details/17386353

[13] Akeley, K., and T. Jermoluk, "High-Performance Polygon Rendering," Computer Graphics (SIGGRAPH '88 Proceedings), pp. 239 - 246, August 1988. Cited on p.22

[14] Akeley, K., P. Haeberli, and D. Burns, tomesh.c, a C-program on the SGI Developer's Toolbox CD, 1990. Cited on p. 543, 553, 554

[15] Akeley, Kurt, "RealityEngine Graphics," Computer Graphics (SIGGRAPH 93 Proceedings),pp. 109 - 116, August 1993. Cited on p. 126

[16] 题图来自《最终幻想 XV》

第十二章 渲染管线优化方法论：从瓶颈定位到优化策略

本章将带来 RTR3 第十五章内容“Chapter 15 Pipeline Optimization”的总结、概括与提炼。

12.0 导读

这篇文章约 1 万 8 千字，构成主要分为上篇（渲染管线瓶颈定位策略），下篇（渲染管线优化策略），以及常用的性能分析工具的列举三部分，详细目录如下。

文中列举了渲染管线各个阶段中用到的几十种主流的优化策略。其中，个人印象比较深刻的优化方法有使用实例（Instance）结合层次细节和 impostors 方法来对多人同屏场景的渲染进行优化，以及使用纹理页（Texture Pages）来进行批次的尺寸最大化。

这篇文章会是《Real-Time Rendering 3rd》第十五章“Pipeline Optimization”和《GPU Gem I》第 28 章“Graphics Pipeline Performance”的一个结合，而不是之前一贯的《Real-Time Rendering 3rd》的单篇章节为主线。

需要吐槽的是，如果你对照阅读《GPU Gem I》的英文原版和中文翻译版，会发现中文翻译版中有一些不合理甚至曲解英文原文意思的地方，在第五部分性能与实这一部分尤其明显。

OK，正文开始。

12.1 渲染管线的构成

通常，可以将渲染管线的流程分为 CPU 和 GPU 两部分。下图显示了图形渲染管线的流程，可以发现，在 GPU 中存在许多并行运算的功能单元，本质上它们就像独立的专用处理器，其中存在许多可能产生瓶颈的地方。包括顶点和索引的取得、顶点着色（变换和照明，Transform & Lighting，即 T&L）、片元着色和光栅操作(Raster Operations ，ROP)。

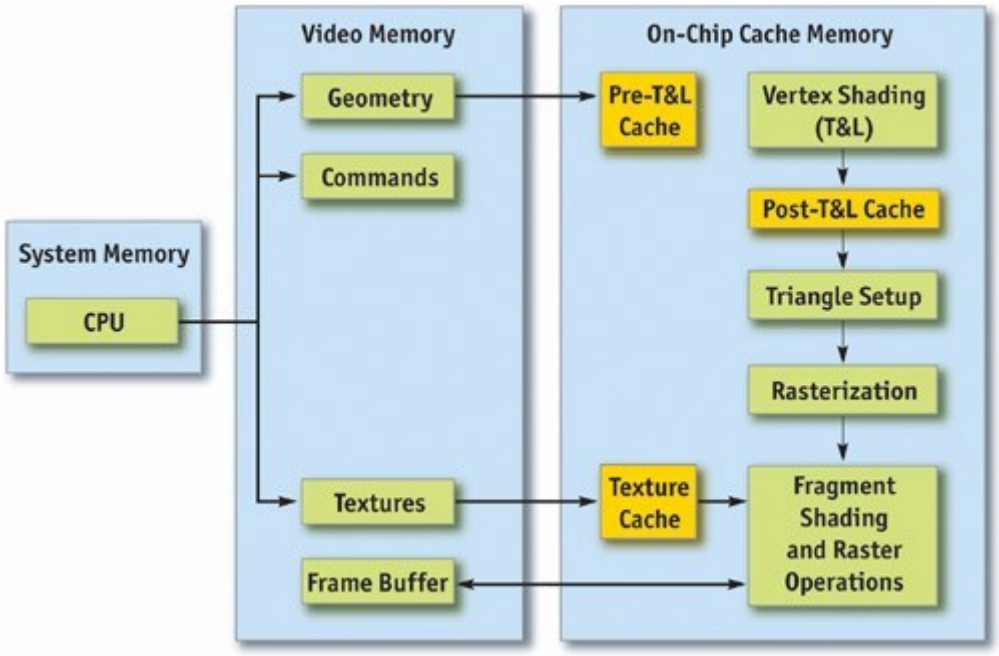


图 1 图形渲染管线

如《Real-Time Rendering 3rd》第二章所述， 图形的渲染过程基于由三个阶段组成的管线架构：

- 应用程序阶段（The Application Stage）
- 几何阶段（The Geometry Stage）
- 光栅化阶段（The Rasterizer Stage）

基于这样的管线架构，其中的任意一个阶段，或者他们之间的通信的最慢的部分，都可能成为性能上的瓶颈。瓶颈阶段会限制渲染过程中的整个吞吐量，从而影响总结渲染的性能，所以不难理解，瓶颈的部分便是进行优化的主要对象。

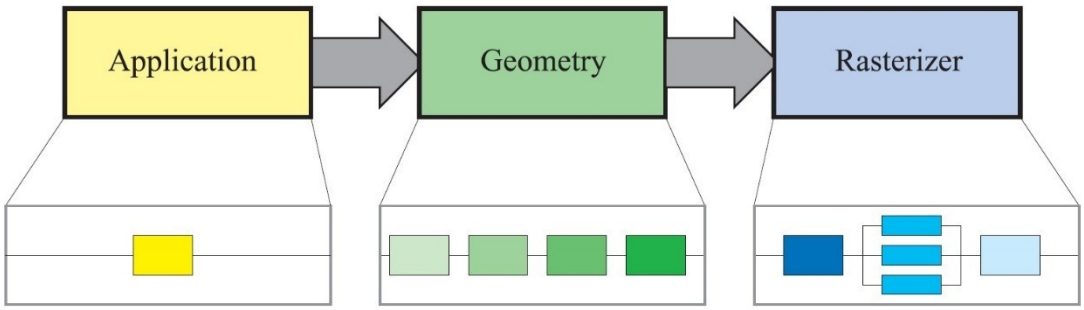


图 2 渲染管线架构

若有对这个过程不太熟悉的朋友，具体可以移步回看这个系列的第二篇文章《【《Real-TimeRendering 3rd》 提炼总结】(二) 第二章 · 图形渲染管线 The Graphics Rendering Pipeline》

12.2 渲染管线的优化概览

准确定位瓶颈是渲染管线优化的关键一步。若没有很好确认瓶颈就进行盲目优化，将造成大量开发的工作的无谓浪费。

根据以往的优化经验，可以把优化的过程归纳为以下基本的确认和优化的循环：

- Step 1. 定位瓶颈。对于管线的每个阶段，改变它的负载或计算能力（即时钟速度）。如果性能发生了改变，即表示发现了一个瓶颈。
- Step 2. 进行优化。指定发生瓶颈的阶段，减小这个阶段的负载，直到性能不再改善，或者达到所需要的性能水平。
- Step 3. 重复。重复第 1 步和第 2 步，直到达到所需要的性能水平。

需要注意的是，在经过一次优化步骤后，瓶颈位置可能依然在优化前的位置，也可能不在。比较好的想法是，尽可能对瓶颈阶段进行优化，保证瓶颈位置能够转移到另外一个阶段。在这个阶段再次成为瓶颈之前，必须对其他阶段进行优化处理，这也是为什么不能在一个阶段上进行过多优化的原因。

同一帧画面中，瓶颈位置也有可能改变。由于某个时候要渲染很多细小的三角形，这个时候，几何阶段就可能是瓶颈；在画面后期，由于要覆盖屏幕的大部分三角形单元进行渲染，因此这时光栅阶段就可能成为瓶颈。因此，凡涉及渲染瓶颈问题，即是指画面中花费时间最多的阶段。

在使用管线结构的时候应该意识到，如果不能对最慢的阶段进行进一步优化，就要使其他阶段与最慢阶段的工作负载尽可能一样多（也就是既然都要等瓶颈阶段，不妨给其他阶段分配更多任务来改善最终的表现，反正是要等）。由于没有改变最慢阶段的速度，因此这样做并没有改变最终的整个性能。例如，假定应用程序阶段成为瓶颈，需要花费 50ms，而其他阶段仅需要花费 25ms。这意味着，在不改变管线渲染速度(50ms，即每秒 20 帧)的情况下，几何阶段和光栅化阶段可以在 50ms 内完成各自任务。这时，可以使用一个更高级的光照模型或者使用阴影和反射来提高真实感（在不增加应用程序阶段工作负载的前提下）。

管线优化的一种大致思路是，先将渲染速度最大化，然后使得非瓶颈部分和瓶颈部分消耗同样多的时间（如上文所述，这里的思想是，既然要等，不等白不等，不妨多给速度快的部分分配更多工作量，来达到更好的画面效果）。但这种想法已经不适于不少新架构，如 XBOX 360，因其为自动加载平衡计算资源。

因为优化技术对于不同的架构有很大的不同，且不要过早地进行优化。在优化时，请牢记如下三句格言：

- “KNOW YOUR ARCHITECTURE（**了解你所需优化的架构**）”
- “Measure（**去测量，用数据说话**）”

- “We should forget about small efficiencies, say about 97% of the time: Premature optimization is the root of all evil.”（我们应该忘记一些小的效率，比如说 97%的时间：**过早的优化是万恶之源。**）– Donald Knuth

OK，下面开始，本文的上篇，渲染管线的瓶颈定位。

12.3 上篇：渲染管线的瓶颈定位策略

正确定位到了瓶颈，优化工作就已完成了一半，因为可以针对管线上真正需要优化的地方有的放矢 。

提到瓶颈定位，很多人都会想到 Profiler 工具。Profiler 工具可以提供 API 调用耗时的详细信息，由此可以知道哪些 API 调用是昂贵费时的，但不一定能准确地确定管道中哪些阶段正在减慢其余部分的速度。（PS:本文文末提供了一系列常用的 profiler 工具的列表）

确定瓶颈的方法除了用 Profiler 查看调用耗时的详细信息这种众所周知的方法外，也可以采用基于工作量变化的控制变量法。**设置一系列测试，其中每个测试减少特定阶段执行的工作量。如果其中一个测试导致每秒帧数增加，则已经找到瓶颈阶段。**

而上述方法的排除法也同样可行，即在不降低测试阶段的工作量的前提下减少其他阶段的工作量。如果性能没有改变，瓶颈就是工作负载没有改变的此阶段。

下图显示了一个确认瓶颈的流程图，描述了在应用程序中精确定位瓶颈所需要的一系列步骤。

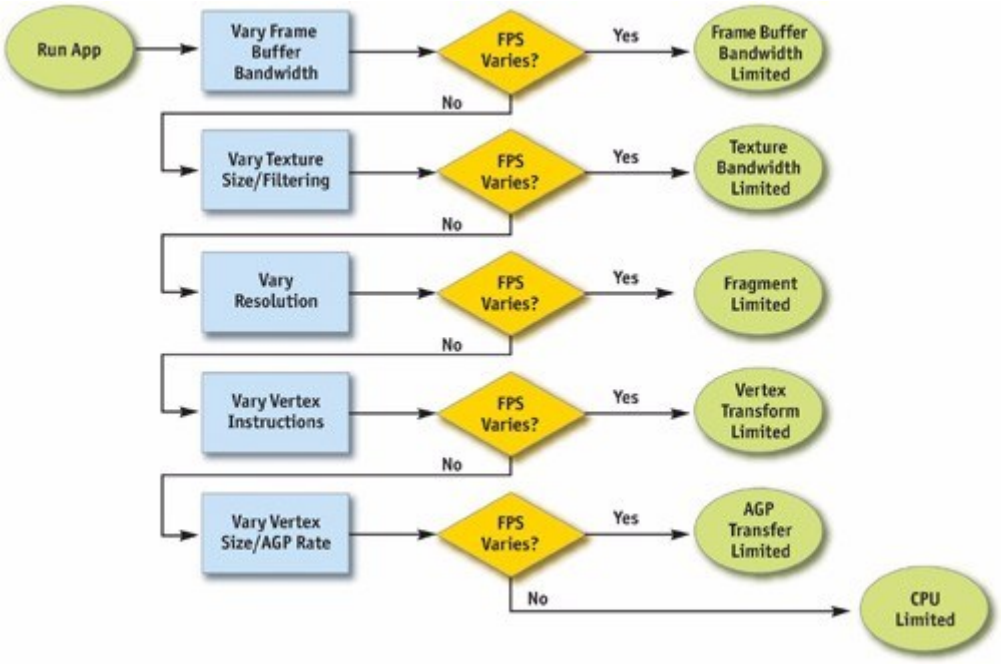


图 3 确认渲染管线瓶颈流程图 @ 《GPU GEMS I》

整个确认瓶颈的过程从渲染管线的尾端，光栅化阶段开始，经过帧缓冲区的操作（也称光栅操作），终于 CPU（应用程序阶段）。虽然根据定义，某个图元（通常是一个三角形）只有一个瓶颈，但在帧的整个流程中瓶颈有可能改变。因此，修改流水线中多个节点的负载常常会影响性能。例如，少数多边形的天空包围盒经常受到片元着色或帧缓冲区存取的限制：只映射为屏幕上几个像素的蒙皮网络时常受到 CPU 或顶点处理的约束。因此，逐个物体地改变负载，或逐个材质地改变负载时常是有帮助的。

另外，管线的每个阶段都依赖于 GPU 频率（分为 GPU Core Clock ，GPU 核心频率，以及 GPU Memory Lock，GPU 显存频率），这个信息可以配合工具 PowerStrip（EnTech Taiwan 2003），减小相关的时钟速度，并在应用中观察性能的变化。

下文将按照按照优化定位的一般顺序（即上述图中的流程），按光栅化阶段、几何阶段、应用程序阶段的的顺序来依次介绍瓶颈定位的方法与要点。

12.3.1 光栅化阶段的瓶颈定位

众所周知，光栅化阶段由三个独立的阶段组成： 三角形设置，像素着色器程序，和光栅操作。

其中三角形设置阶段几乎不会是瓶颈，因为它只是将顶点连接成三角形。**而测试光栅化操作是否是瓶颈的最简单方法是将颜色输出的位深度从 32（或 24）位减少到 16 位**。如果帧速率大幅度增加，那么此阶段为瓶颈所在。

一旦光栅化操作被排除，**像素着色器程序的是否是瓶颈所在可以通过改变屏幕分辨率来测试**。如果较低的屏幕分辨率导致帧速率明显上升，像素着色器则是瓶颈，至少在某些时候会是这样。当然，如果是渲染的是 LOD 系统，就需斟酌一下是否瓶颈确实是像素着色器了。

另一种方法与顶点着色器程序所采用的方法相同，可以添加更多的指令来查看对执行速度的影响。当然，也要确保这些额外的指示不会被编译器优化。

下文将对光栅化阶段三个常常可能是瓶颈的地方进行进一步论述。

12.3.1.1 光栅化操作的瓶颈定位

光栅化操作的瓶颈主要与帧缓冲带宽（Frame-Buffer Bandwidth）相关。众所周知，位于管线末端的光栅化操作（Raster
Operations，常被简称为 ROP），用于深度缓冲和模板缓冲的读写、深度缓冲和模板缓冲比较，读写颜色，以及进行 alpha 混合和测试。而光栅化操作中许多负载都加重了帧缓冲带宽负载。

测试帧缓冲带宽是否是瓶颈所在，比较好的办法是改变颜色缓冲的位深度，或深度缓冲的位深度（也可以同时改变两者）。如果此操作（比如将颜色缓冲或深度缓冲的深度位从 32 位减少到 16 位）明显地提高了性能，那么帧缓冲带宽必然是瓶颈所在。

另外，帧缓冲带宽也与 GPU 显存频率（GPU memory clock）有关，因此，修改该频率也可以帮助识别瓶颈。

12.3.1.2 纹理带宽的瓶颈定位

在内存中出现纹理读取请求时，就会消耗纹理带宽(Texture Bandwidth)。尽管现代 GPU 的纹理高速缓存设计旨在减少多余的内存请求，但纹理的存取依然会消耗大量的内存带宽。

在确认光栅化操作阶段是否是瓶颈所在时，修改纹理格式比修改帧缓冲区的格式更麻烦。所以，**比较推荐使用大量正等级 mipamap 细节层次(LOD)的偏差，让纹理获取访问非常粗糙的 mipmap 金字塔级别，来有效地减小纹理尺寸。**同样，如果此修改显著地改善性能，则意味着纹理带宽是瓶颈限制。

纹理带宽也与 GPU 显存频率相关。

12.3.1.3 片元着色的瓶颈定位

片元着色关系到产生一个片元的实际开销，与颜色和深度值有关。这就是运行“像素着色器（Pixel Shader ）”或“片元着色器（Fragment Shader ）”的开销。片元着色（Fragment shading）和帧缓冲带宽（Frame-Buffer Bandwidth）由于填充率（Fill Rate）的关系，经常在一起考虑，因为他们都与屏幕分辨率相关。尽管它们在管线中位于两个截然不同的阶段，区分两者的差别对有效优化至关重要。

在可编程片元处理的高级 GPU 出现之前，片元着色几乎没有什么局限性，时常是帧缓冲带宽引起的屏幕分辨率和性能之间不可避免的瓶颈。但随着开发者利用新的灵活性制造出一些新奇的像素，片元着色的性能问题也就出现了。

改变分辨率是确定片元着色是否为瓶颈的第一步。因为在上述光栅化操作步骤中，已经通过改换不同的深度缓冲位，排除了帧缓冲区带宽是瓶颈的可能性。所以，**如果调整分辨率使得性能改变，片元着色就可能是瓶颈所在**。而辅助的鉴别方法可以是修改片元长度，看这样是否会影响性能。但是要注意，不要添加可以被一些“聪明”的设备驱动轻松优化的指令。

片元着色的速度与 GPU 核心频率有关。

12.3.2 几何阶段的瓶颈定位

几何阶段是最难进行瓶颈定位的阶段。这是因为如果在这个阶段的工作负载发生了变化，那么其他阶段的一个或两个阶段的工作量也常常发生变化。为了避免这个问题，Cebenoyan [1]提出了一系列的试验工作从光栅化阶段后的管线。

在几何阶段有两个主要区域可能出现瓶颈：顶点与索引传输(Vertex and Index Transfer)和顶点变换阶段（ Vertex Transformation Stage ）。要看瓶颈是否是由于顶点数据传输的原因，可以增加顶点格式的大小。这可以通过每个顶点发送几个额外的纹理坐标来实现，例如。如果性能下降，这个部分就是瓶颈。

顶点变换是由顶点着色器或固定功能管线的转换和照明功能完成的。对于顶点着色器瓶颈，测试包括使着色器程序更长。为了确保编译器没有优化这些附加指令，必须采取一些注意事项。对于固定功能管线，可以通过打开附加功能（如镜面高光）或将光源转换成更复杂的形式（例如聚光灯）来提高处理负荷。

下文将对几何阶段两个常可能是瓶颈的阶段的定位方法进行进一步论述。

12.3.2.1 顶点与索引传输的瓶颈定位

GPU 渲染管线的第一步，是让 GPU 获取顶点和索引。而 GPU 获取顶点和索引的操作性能取决于顶点和索引的实际位置。其位置通常是在系统内存中（通过 AGP 或 PCI Express 总线传送到 GPU），或在局部帧缓冲内存中。通常，在 PC 平台上，这取决于设备驱动程序而不是应用程序，而现代图形 API 允许应用程序提供使用提示，以帮助驱动程序选择正确的内存类型。

可以通过调整顶点格式的大小，来确定得到顶点或索引传输是否是应用程序的瓶颈。

如果数据放在系统内存内，得到顶点或索引的性能与 AGP 或 PCI Express 总线传输速率有关；如果数据位于局部缓冲内存，则与内存频率有关。

如果上述测试对性能都没有明显影响，那么顶点与索引传输阶段的瓶颈也可能位于 CPU 上。我们可以通过对 CPU 降频来确认这一事实，如果性能按比例进行变化，那么 CPU 就是瓶颈所在。

12.3.2.2 顶点变换的瓶颈定位

渲染管线中的顶点变换阶段（ Vertex Transformation Stage ）负责输入一组顶点属性（如模型空间位置、顶点法线、纹理坐标等等），以及生产一组适合裁剪和光栅化的属性（如齐次裁剪

空间位置，顶点光照结果，纹理坐标等等）。当然，这个阶段的性能与每个顶点完成的工作，以及正在处理的顶点数量有关。

对于可编程的顶点变换，只要简单地改变顶点程序的长度，就能确定顶点处理是否是瓶颈。如果此时发生性能的变化，就可以判定顶点处理阶段是瓶颈所在。如上文提到过的，如果要增加指令，在添加富有意义的指令时需要留心，
以防止被编译器或驱动将指令优化掉。例如，因为驱动程序通常不知道程序编译时常量的值，没有被常量寄存器引用的空操作指令（no-ops）不能被优化（如加入一个含有值为零的常量寄存器）。

对于固定功能的顶点变换，判定瓶颈则有点麻烦。试着通过改变顶点的工作，例如修改镜面光照或纹理坐标生成的状态来修改负载。

另外需要注意，顶点处理的速度与 GPU 核心频率有关。

12.3.3 应用程序阶段的瓶颈定位

以下是应用程序阶段的瓶颈定位的一些策略的总结：

- **可以用 Profiler 工具查看 CPU 的占用情况。**主要是看当前的程序是否使用了接近 100%的 CPU 占用。比如 AMD 出品的 Code
  Analyst 代码分析工具，可以对运行在 CPU 上的代码进行分析和优化。Intel 也出品了一个称为 Vtune 的工具，可以分析在应用程序或驱动器（几何处理阶段）中时间花费的位置情况。

- **一种巧妙的方法是发送一些其他阶段工作量极小甚至根本不工作的数据。**对于某些 API 而言，可以通过简单地使用一个空驱动器（就是指可以接受调用但不执行任何操作）来取代真实驱动器来完成。这就有效地限制了整个程序运行的速度，因为我们没有使用图形硬件，因此 CPU 始终是瓶颈。通过这个测试，我们可以了解在应用阶段没有运行的阶段有多大的改进空间。也就是说，请注意，使用空驱动程序还隐藏了由于驱动程序本身和阶段之间的通信所造成的瓶颈。

- **另一个更直接的方法是对 CPU 进行降频(Underclock)。**如果性能与 CPU 速率成正比，则应用程序的瓶颈与 CPU 相关。但需要注意，降频的方法可以帮助识别瓶颈，也有可能导致一个之前不是瓶颈的阶段成为瓶颈。

- 另外，则是排除法，**如果 GPU 阶段没有瓶颈，那么 CPU 就一定是瓶颈所在。**

11.4 下篇：渲染管线的优化策略

一旦确定了瓶颈位置，就可以对瓶颈所处阶段进行优化，以改善我们游戏的性能。下面根据解决问题的不同阶段，对一些优化策略进行了分类整理，将分为六个部分来进行呈现：

- 对 CPU 的优化策略
- 应用程序阶段的优化策略
- API 调用的优化策略
- 几何阶段的优化策略
- 光照计算的优化策略
- 光栅化阶段的优化策略

12.4.1 对 CPU 的优化策略

许多应用的瓶颈都位于 CPU，有的是正当理由（如复杂的物理或 AI 运算）导致，有的是因为不好的批处理或资源管理导致。如果已经发现应用程序受到 CPU 限制，可以试行下列建议，以对渲染管线中 CPU 的性能进行优化。

12.4.1.1 减少资源锁定

每当执行一个需要访问 GPU 的同步操作，就可能严重堵塞 GPU 管线，这将消耗 CPU 和 GPU 两者的周期。CPU 必须保持在一个循环中，等待 GPU 管线工作，直到它闲下来并返回所请求的资源，这种等待会造成 CPU 周期的浪费。然后 GPU 等待对管线的再填充，这种等待又造成 GPU 周期的浪费。

上述的锁定发生在以下情况下：

- 对前面正在渲染的表面进行锁定或读出时
- 对 GPU 正在读的表面进行写入，例如纹理或顶点缓冲区

而减少资源锁定的方法，可以尝试避免访问渲染期间 GPU 正在使用的资源。

12.4.1.2 批次的尺寸最大化

这个策略也可以称为“将批次的数量减到最小”。

批次（batch）是调用单个 API 渲染所做的一组基本渲染。用来绘制几何体的每个 API 函数调用，都有对应的 CPU 消耗。因此最大限度地增加每次调用所提交的三角形的数量，CPU 渲染给定数目三角形的消耗就可以减到最小。也即批次的尺寸乘以批次数量得到的工作总量一定，此消彼长。

使批次最大化的技巧列举如下：

- **若使用了三角形带（Triangle Strips），则使用退化三角形（Degenerate Triangles）将不相交的条带拼接起来**。这样就能够一次发送多条三角形带，以便能在单个 Draw Call 中共享材质。
- **使用纹理页（Texture Pages）**。不同物体使用不同纹理时，批次时常会被打破，若通过把多个纹理安排进单个的 2D 纹理内并适当设定纹理坐标，就能在单个 Draw Call 中发送使用了多个纹理的几何体。此技术可能存在 mipmapping 和反走样的问题，而回避大部分这类问题的技术是，把单个的 2D 纹理打包到一个立方体贴图的各个面内。
- **使用 Shader 分支来增加单个批次大小从而合批。**现代 GPU 具有灵活的顶点和片元处理管线。允许 Shader 里有分支。例如，若两个分开的批次，因为其中一个需要四个骨骼蒙皮顶点着色器，而另一个需要两个骨骼蒙皮顶点着色器，你可以编写一个顶点着色器来遍历所需的骨骼数量，累积混合权重，然后在权重相加为一个时跳出循环。这样两个批次就可以合并为一个。在不支持 shader 分支的架构上，可以实现相似的功能，方法是上述两种情况都使用 4 块骨骼的顶点着色器，对骨骼数量不足 4 块的顶点，将其骨骼权重因子设置为 0。
- **将顶点着色器常量内存（vertex shader constant memory）作为矩阵查找表（Lookup Table of matrices）使用**。通常，当许多小对象共享所有的属性，但仅矩阵状态不同时（例如，含相似树木的森林，或一个粒子系统），批次就会遭到破坏。这时，可以把 n 个不同的矩阵加载到顶点着色器常量内存中，并将索引以每个对象的顶点格式存储在常量内存中。然后使用此索引查询顶点 shader 的常量内存，并使用正确的变换矩阵，从而一次渲染 n 个对象。
- **尽可能远地往管线下端推迟决策。**若要速度更快，应该使用纹理的 alpha 通道作为发光值，而不是打破批次，为光泽设定一个像素 shader 常量。同样地，把着色数据放入纹理和顶点可以使单个批次的提交量更大。

12.4.2 应用程序阶段的优化策略

对应用程序阶段的优化，可以通过提高代码的执行速度，以及提到程序的存储访问速度（或者减少存储访问的次数）来实现。下面将给出一些通用的优化技术，适用于大多数的 CPU。

最基本的代码优化策略包括为编译器打开优化标志。通常有很多不同的标志，一般需要检查哪些标志可以应用于程序代码中，而且对所使用的优化选项一般不做任何假设。例如，可以将编译器的开关设置为“最小代码大小（minimize code size）”而不是“速度优化（optimizing for speed）”，这样可以导致代码执行速度的提高，因为缓冲性能提高了。此外，如果可能的话，可以尝试不同的编译器，因为不同编译器一般是按照不同的方式进行优化的。

对于代码优化来说，定位大部分时间花在哪部分代码是很关键的。一个好的代码 profiler 是找到大部分运行时间都花费在代码哪里的关键。然后在这些地方进行优化工作。而这些位置通常是内部循环，或是每帧执行多次的代码片段。（PS:本文文末提供了一系列常用的 profiler 工具的列表）

优化的基本原则是尝试多种策略，包括重新检查算法，假设，以及代码语法等，也就是尽可能多的尝试各种变化情况。

下文将从内存层面和代码层面进一步说明。

12.4.2.1 内存层面的优化

对于存储层次结构来说，如何在各种不同的 CPU 体系结构上编写执行速度较快的代码变得越来越重要。在编写程序时，应该注意下列准则：

- **在代码中连续访问的存储内容在内存中也应保持连续存储。**例如，当渲染一个三角形网格的时候，如果访问的顺序是：纹理坐标#0、法线#0、颜色#0、顶点#0、纹理坐标#1、法线#1 等，那么在内存中也应该按这个顺序连续存储。**尽量避免指针的间接、跳转，以及函数调用，因为它们很容易显著降低 CPU 中缓冲的性能。**比如当一个指针指向另一个指针，而这个指针又指向其他指针时，以此类推，类似典型的链表和树结构，而这将导致数据缓存未命中（cache misses for data）。为了避免这种情况，应该尽可能使用数组来代替。

PS: 上述条准则的思想有点类似《Game Programming Patterns》书中讲到的数据局部性模式（Data Locality pattern），具体可以参考《Game Programming Patterns》这本书的 web 版关于数据局部性模式的讲解：http://gameprogrammingpatterns.com/data-locality.html

- **某些系统中，默认的内存分配和删除功能可能比较慢，因此，在启动时最好为相同大小的对象分配一个大的内存池，然后使用自己分配或空闲部分来处理该池的内存。**
- **尽量尝试去避免在渲染循环中分配或释放内存**。例如，可以单次分配暂存空间(scratch space)，并且使用栈、数组等其他仅增长的数据结构（也可以使用标志位来标识哪些元素应该被视为已删除）。
- **对数据结构尝试用不同的组织形式。**例如，Hecker[2]指出，对于一个简单的矩阵乘法器而言，通过不同的矩阵结构可以节省大量的计算开销。例如，一个结构数组如下：

```
struct Vertex {float x,y,z;}

Vertex myvertices[1000];
```

或者为：

```
struct VertexChunk {float x[1000],y[1000],z[1000];}

VertexChunk myvertices;
```

对于给定的体系结构而言，上述第二种结构对于 SIMD 命令来说要更好一些。但是随着顶点数目的增多，高速缓存的命中失误率也会随之增多。当数组大小增加到一定程度时，下面这种混合方案可能是最好的一种选择：

```
struct Vertex4 {float x[4],y[4],z[4];}
```

Vertex4 myvertices[250];

12.4.2.2 代码层面的优化

下面的会列出编写与计算机图形相关的高效代码的一些技术。这些方法随着编译器和不断发展的 CPU 而有所不同，但大多数已经保存了很多年（主要是针对 C/C++而言）：

- **善用 SIMD。**单指令多数据流（Single Instruction Multiple Data，SIMD），例如 Intel 的 MMX 或 SSE，以及 AMD 的 3D Now!指令集，在很多情形下能获得很好的性能，可以并行计算多个单元，且比较适合用于顶点操作。
- **使用 float 转 long 转换在奔腾系列处理器上速度较慢。**如果可以请尽量避免。
- **尽可能避免使用除法。**相对于其他大多数指令而言，执行除法指令所需要的时间大约是执行其他指令所需时间的 2.5 倍或更多。
- **许多数学函数，如 sin、cos、tan、exp、arcsin 等，计算开销较高，使用的时候必须小心。如果可以接受低精度，那么只需要使用麦克劳林（MacLaurin）或泰勒（Taylor）级数的前几项即可。**由于现代 CPU 对内存的访问的代价依然很高，因此使用级数的前几项比使用查找表（Lookup Tables）强得多。
- **条件分支会有一定的开销，Shader 中的条件分支开销尤甚。**尽管大多数处理器都有分支预测功能，但是这意味着只有准确地进行分支预测，才有可能降低计算开销。错误的分支预测对一些体系结构、特别是对于具有深管线的体系结构来说，计算开销通常会较高。
- **对于经常调用的小函数使用内联（Inline）。**
- **在合理的情况下减少浮点精度，比如用 float 代替 double。**而当选用 float 型来代替 double 型数据时，需要在常数末尾加上一个 f。否则，整个表达式就会被强制转换为 double 型；因此，语句 float x =2.42f；要比 float x = 2.42；执行得更快。
- **尽可能使用低精度数据，让发送到图形管线的数据量更少。**
- **虚函数方法、动态转换、（继承）构造，以及按值传递结构体（passing structs by value）都会对效率造成一定影响。**据了解，一帧画面中大约有 40%的时间花费在用于模型管理的虚拟继承层次结构上。Blinn 提出了一种技术[3]，可以避免计算 C++中向量表方面的一部分开销。

12.4.3 API 调用的优化策略

上文已经提到，批次（batch）是调用单个 API 渲染所做的一组基本渲染。用来绘制几何体的每个 API 函数调用，都有对应的 CPU 消耗。改进批次过小问题的方法有很多种，且它们都有共同的目标——更少的 API 调用。以下是一些要点。

- **一种减少 API 调用的方法是使用某种形式的实例（Instance）**。大多数 API 都支持在一次调用中拥有一个对象并进行多次绘制。因此，与其为森林中的每一棵树单独调用 API，不如使用单次调用来渲染树模型的许多副本。如下图。

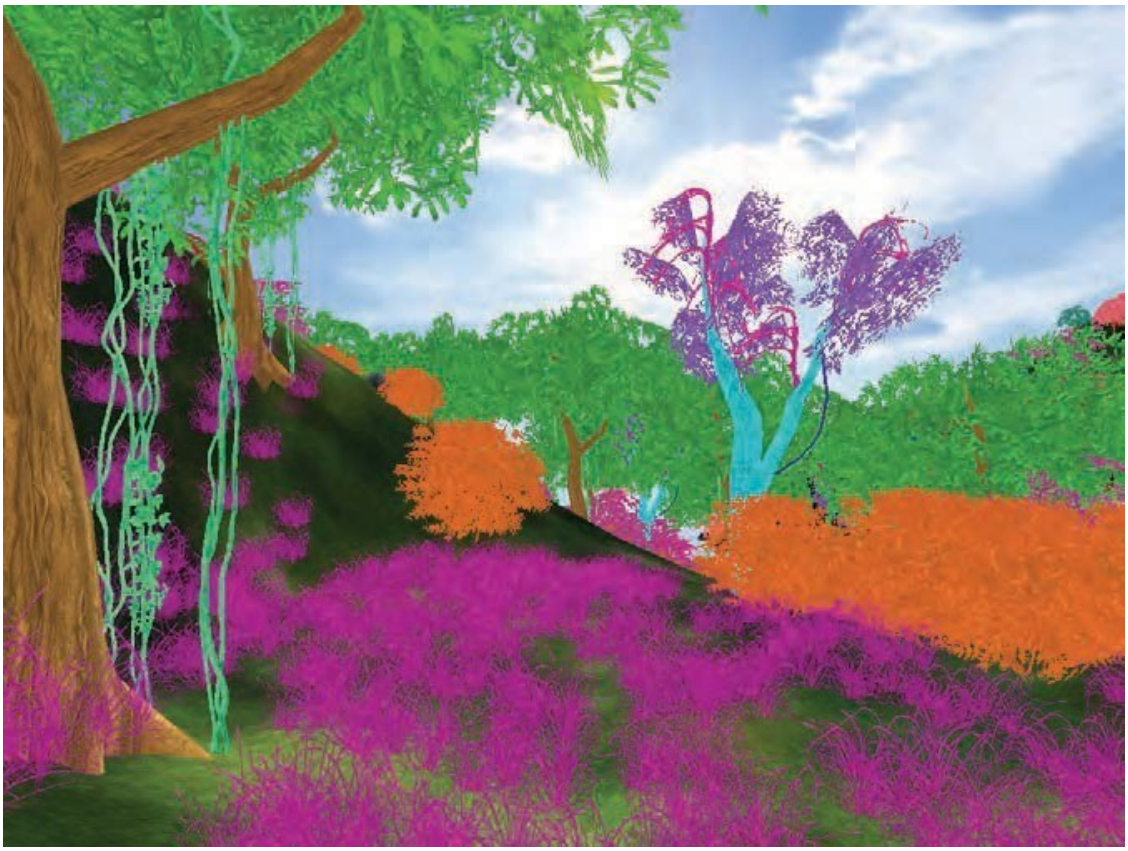


图 4 植被实例（Vegetation instancing）。所有同样颜色的物体在一个 Draw Call 中进行渲染。

PS: 此思想有点类似设计模式中的享元模式（flyweight pattern）。具体可以参考《Game Programming Patterns》这本书的 web 版关于享元模式的精彩讲解：
http://gameprogrammingpatterns.com/flyweight.html

- **进行批处理（batching）**。批处理的基本思想是将多个对象合并成一个对象，因此只需要一个 API 调用便可以渲染整个集合。批处理中的合并可以一次性完成，且缓冲区对静态对象集合都能每帧进行重用。对于动态对象，可以使用多个网格填充单个缓冲区。但这种基本方法的局限性是，网格中的所有对象都需要使用一组相同的着色器程序，即相同的材质。

- **可以用不同的颜色来合并对象，例如，通过用标识符对每个对象的顶点进行标记。**着色器程序可以根据此标识符，查找使用什么颜色来遮挡物体。同样的想法可以扩展到其他表面属性。类似地，附于表面的纹理也可以用于标识使用哪种材质。而单独物体的光照贴图需合并成纹理图集（texture atlases）或纹理数组（texture array）。

- **多人同屏的场景很适合使用实例进行渲染，其中每个角色都拥有独特的一套外表。**而进一步的变化可以添加随机的肤色和贴花。这种基于实例的方法也可以结合 LOD 技术进行。如下图。

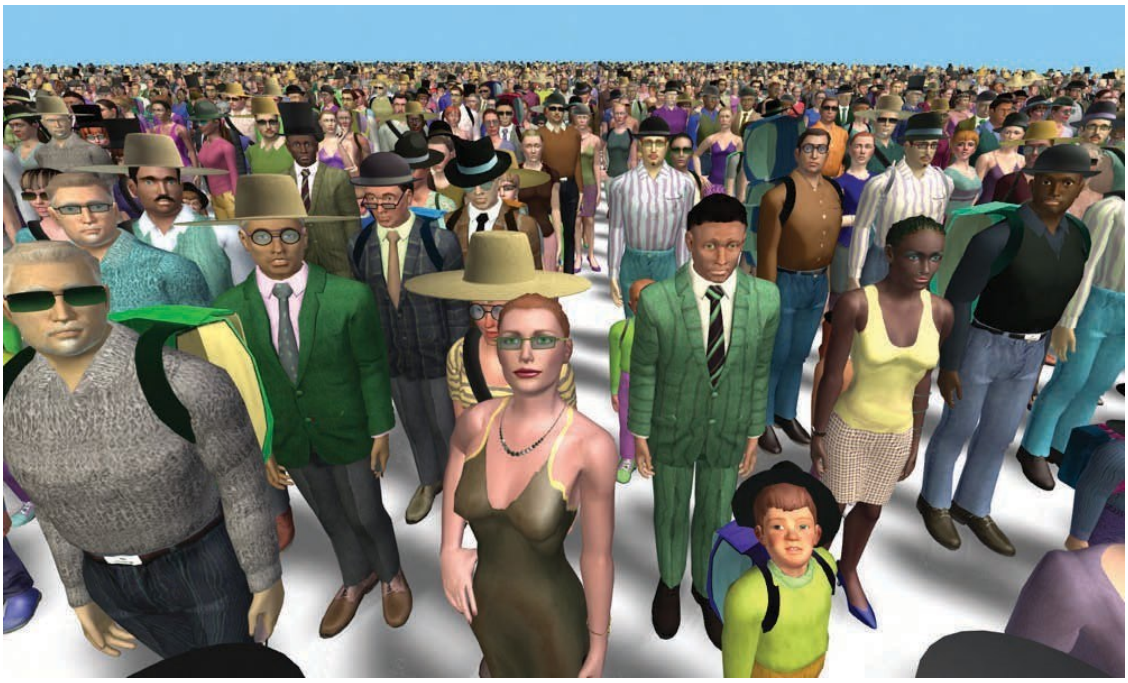


图 5 多人同屏场景（crowd scene）。使用实例（instancing）来减少 Draw Call，也可以结合 LOD 技术使用，比如对于远处的模型，使用 impostors 进行渲染。

- **提高性能的另一种方法是通过将具有类似渲染状态的对象（顶点和像素着色器、纹理、材质、光照、透明度等）分组并将它们按顺序渲染，从而最小化状态更改**。

- 改变状态时，有时需要完全或部分地清理管线。出于这个原因，改变着色器程序或材质参数可能非常昂贵。**对使用共享材质（shared material）的节点可以进行分组，以获得更好的性能，而采用共享纹理（shared texture）绘制多边形可以减小纹理缓存的抖动。**另外，正如上文提到的，一种减少纹理改变的变化的方法便是把一些纹理图像到一个大的纹理图集或纹理数组中。

- **理解对象缓冲（object buffer）在渲染时的分配和存储方式也同样重要。**对于一个含 CPU 和 GPU 的系统，GPU 和 CPU 有各自的内存，而图形驱动程序通常控制对象所在的位置，它也可以给出存储在何处是更优的建议。一个常见的类型分类是静态与动态缓冲区。如果一个物体不形变，或者形变可以完全由着色器程序（如蒙皮）完成，那么在 GPU 内存中存储对象的数据是较为合适的。而该对象的不变属性可以通过将其存储在为静态缓冲区中。通过这种方式，不必在渲染的每帧在总线上发送这些不变的数据，从而避免在管线的这一阶段出现瓶颈。

12.4.4 几何阶段的优化策略

几何阶段主要负责变换、光照、裁剪、投影，以及屏幕映射。其中，变换和光照过程比较容易优化，其他几个部分的优化稍显困难。以下是一些要点：

- 变换、光照、裁剪、投影，以及屏幕映射操作可以使用较低精度的数据，以减小开销。

- 合理地使用索引和顶点缓冲区可以帮助几何阶段减小计算量。

- **可以简化模型来减小整个管线的顶点和绘制图元的数量，以降低顶点数据传输和顶点变换成本。**而诸如视锥裁剪和遮挡剔除之类的技术避免了将全部的图元发送到管线。
- 可以使用缓存感知（cache-oblivious）布局算法，其中顶点以某种形式排列，以最大限度地提高缓存重用性，从而节省处理时间。（具体可见 RTR3 原文 12.4.4 节）
- 同理，**为了节省内存和访问时间，尽可能在顶点、法线、颜色和其他着色参数上，选择更低精度的数据格式。**有时我们会在 half、single、double。float 精度之间做选择，需要注意，除了其中因为精度更低带来的速度提升外，有些格式也会因为是硬件内部使用的原生格式（native format）而更快。
- **减少内存使用的另一种方法是将顶点数据存储在压缩格式中。**对此，Deering [4]深入讨论了这种技术. Calver [5]提出了各种方案，使用顶点着色器进行解压。zarge [6]也指出，数据压缩也有助于调整顶点格式缓存线。而 Purnomo 等人[7]结合简化方法和顶点的量化技术，使用图像空间的度量，提出了为一个给定的目标网格尺寸优化网格的方案。

12.4.4.1 减少顶点传输的开销

顶点传递是瓶颈的可能性很小，但也偶有发生。假如顶点或索引（索引是瓶颈的可能性更小）的传递是应用瓶颈，可以试着使用下列各项策略：

- **在顶点格式中使用尽可能少的位。**位数足够即可，不需要对所有数据都使用浮点格式（例如对颜色）。
- **在顶点程序中产生可推导的顶点属性，而不是把他们存储在输入顶点格式中。**例如，正切线(tangent)、法线(normal)和副法线(binormal)通常不需要都存储。给出任意两个，能用 Vertex-program 简单叉积推导出第三个。这项技术，即为用顶点处理速度去换取顶点传输速率。
- **使用 16 位的索引代替 32 位的索引。**16 位索引更容易查找。移动起来更轻量，而且占用的内存更少。
- **以相对连续的方式访问顶点数据。**当访问顶点数据时现代 GPU 可以进行缓存。因为在任意内存层次中，引用的空间局部性有助于最大化缓存的命中率，这可以减少对带宽的要求。

12.4.4.2 顶点处理的优化

顶点处理是现代 GPU 的瓶颈可能性很小，但是也偶有发生，这取决于所使用的模式和目标硬件。如果发现顶点处理是瓶颈所在，可以试用如下列举的各项策略：

- **对变换和照明(T&L)后的顶点存储进行优化。**现代 GPU 有一个小的先入先出(FIFO)的缓存，用于存储最近所转换的顶点结果：命中这个高速缓冲器可以保存所有的变换和照明，以及所有流水线早先完成的工作。为了利用这个缓存的优势，必须使用经过索引的图元，而且必须对顶点进行排序，以最大化网格上的引用局部性。可以帮助完成这个任务的工具有 D3DX 和 NVTriStrip 等。

- **减少所处理的顶点数。**这是能想到的很基本的解决方案。但是使用简单的层次细节方案，例如一组静态的 LOD，确实有助于减少顶点处理的负担。
- **使用顶点处理 LOD。**在使用层次细节减少所处理的顶点数时，可以试着把层次细节用于顶点计算本身。例如，对远处的任务没必要完全做 4 块骨骼的蒙皮，或许可以使用更轻量的光照近似。而如果当前材质的 shader 是多通道的，那么减少位于远处低 LOD 级别的渲染通道数量，也会减少顶点处理的成本。
- **把每个物体的计算留给 CPU 去做。**每个物体或每帧都改变的计算时常在顶点着色器中进行。例如，将方向光矢量转换到视点空间的通常在顶点 shader 中进行，虽然计算结果只是每帧改变一次。
- **使用正确的坐标空间。**坐标空间的选择时常影响计算视点程序值所需的指令数。例如，计算顶点光照时，如果顶点法线存储在物体空间中，而方向光矢量存储在视图空间中，就须在顶点 shader 中转换其中之一，将两者转换到统一空间下。而如果改为在 CPU 上对每个物体一次性地把光矢量转换到物体空间，再进行逐个顶点的转换就没有必要了，这样就节省了 GPU 顶点处理的运算量。
- **使用顶点分支来“提前结束”计算**。例如，若在顶点着色器中循环多个光源，然后进行法线、[0，1] 低动态范围的光照，你可以判断饱和度到 1，或者远离光源的顶点，来 break 掉，避免进一步无用的计算。对于蒙皮阶段有一个类似的优化，当权重之和达到 1 时，停止计算（因此所有后来加权的值是 0）。需要注意，这个方法是否生效，依赖于 GPU 如何实现顶点分支，无法在所有架构上保证性能的改善。

12.4.5 光照计算的优化策略

考虑光照的影响可以每顶点，每像素的进行计算，光照计算可以通过多种方式进行优化：

- **首先，应该考虑使用的光源类型，以及可以考虑是否所有的多边形都需要光照。**有时模型只需纹理贴图，或者在顶点使用纹理，或只需要顶点颜色。那么很多多边形就无需进行光照计算。
- **如果光源是静态的，且照明对象是几何体，那么漫反射光照和环境光可以预先计算并存储在顶点颜色中。**这样做通常被称为烘焙照明（baking lighting）。一个预光照（prelighting）更复杂的形式是使用辐射度（Radiosity）方法预先计算场景中的漫反射全局光照，而这样的光照可以存储在顶点颜色或光照贴图（lightmaps）中。
- **控制光源的数量。**光源的数量影响几何阶段的性能，更多的光源意味着更少的速度。此外，双面的光照可以比单面光照更为昂贵。当对光源使用固定功能距离衰减时，根据物体与光源的距离来关闭/打开光源是有较为有用，且几乎不会被察觉。而距离足够大时，可以关掉光源。
- **一种常见的优化方法是根据光源的距离来进行剔除，只渲染受本地光源影响的对象。**
- **另一种减少工作的方法是禁用光源，取而代之的是使用环境贴图（environment map）**
- **如果场景拥有大量光源，可以使用延迟着色技术来限制计算量和避免状态的变化。**

12.4.6 光栅化阶段的优化策略

光栅化阶段可以以多种方式进行优化。现将主流的优化策略列举如下：

- **善用背面裁剪**。对封闭（实心）的物体和无法看到背面的物体（例如，房间内墙的背面）来说，应该打开背面裁剪开关。这样对于封闭的物体来说，可以将需光栅化处理的三角形数量减少近 50%。但需要注意的是，虽然背面裁剪可以减少不必要的图元处理，但需要花费一定的计算量来判断图元是否朝向视点。例如，如果所有的多边形都是正向的，那么背向裁剪计算就会降低几何阶段的处理速度。

- **一种光栅化阶段的优化技术是在特定时期关闭 Z 缓冲（Z–buffering）。**例如，在清楚帧缓冲之后，必须要进行深度测试也可以直接渲染出任何背景图像。如果屏幕上的每个像素保证被某些对象覆盖（如室内场景，或正在使用背景天空图），则不需要清楚颜色缓冲区。同样，确保只有在需要时才使用混合模式（blend modes）。

- **值得一提的是，如果在使用 Z 缓冲，在一些系统上使用模板缓冲不需要额外的时间开销。**这是因为 8 位的模板缓冲的值是存储为 24 位 z 深度值的同一个 word 中。

- **优先使用原生的纹理和像素格式。**即使用显卡内部使用的原生格式，以避免可能会有的从一种格式到另一种格式的昂贵转换。

- **另一种适用于光栅化阶段的优化技术是进行合适的纹理压缩。**如果在送往图形硬件之前已经将纹理压缩好，那么将它发送到纹理内存中的速度将会非常迅速。压缩纹理的另一个优点是可以提高缓存使用率，因为经过压缩的纹理会使用更少的内存。

- **另一种有用的相关优化技术是基于物体和观察者之间的距离，使用不同的像素着色器。**例如，在场景中有三个飞碟模型，最接近摄像机的飞碟的可能用详细的凹凸贴图来进行渲染，而另外两个较远的对象则不需要渲染出细节。此外，对最远的飞碟可以使用简化的镜面高光，或者直接取消高光，来简化了计算量以及减少采样次数。

- **理解光栅化阶段的行为。**为了很好地理解光栅阶段的负荷，可以对深度复杂度进行可视化，所谓的深度复杂度就是指一个像素被接触的次数。生成深度复杂度图像的一种简单方法就是，使用一种类似于 OpenGL 的 glBlendFunc(GL ONE,GL ONE)调用，且关闭 Z 缓冲。首先，将图像清除成黑色；然后，对场景中所有的物体，均使用颜色(0,0,1)进行渲染。而混合函数（blend function）设置的效果即是对每个渲染的图元来说，可以将写入的像素值增加(0,0,1)。那么，深度复杂度为 0 的像素是黑色，而深度复杂度为 255 的像素为全蓝色（0, 0, 255）。

- **可以通过计数得到通过 Z 缓冲与否的像素的像素的数量，从而确定需进一步优化的地方。**使用双通道的方法对那些通过或没通过 Z 缓冲深度测试的像素进行计数。在第一个通道中，激活 Z 缓冲，并对那些通过深度测试的像素进行计数。而对那些没有通过深度测试的像素进行计数，可以通过增加模板缓冲的方式。另一种方法是关闭 Z 缓冲进行渲染来获得深度复杂度，然后从中减去第一个通道的结果。

通过上述方法得到结果后，可以确认：

（1）场景中深度复杂度的平均值、最小值和最大值

（2）每个图元的像素数目（假定已知场景中图元的数目）；

（3）通过或没有通过深度测试的像素数目。

而上述这些像素数量对理解实时图形应用程序的行为、确定需要进一步优化处理的位置都非常有用。

通过深度复杂度可以知道每个像素覆盖的表面数量，重复渲染的像素数量与实际绘制的表面的多少是相关的。假设两个多边形覆盖了一个像素，那么深度复杂度就是 2。如果开始绘制的是远处的多边形，那么近处的多边形就会重复绘制整个远处的多边形，重绘数量也就为 1。如果开始绘制的是近处的多边形，那么远处的多边形就不会通过深度测试，从而也就没有重绘问题。假设有一组不透明的多边形覆盖了一个像素，那么平均绘制数量就是调和级数：

$$H(n) = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n}. \qquad (15.2)$$

上式背后所包含的逻辑是：第一个绘制的多边形是一次绘制：第 2 个多边形在第一个多边形之前绘制的概率是 1/2：第三个多边形在前两个多边形前绘制的概率是 1/3。依次类推，当 n 取极极限时：

$$\lim_{n\to\infty} H(n) = \ln(n) + \gamma, \qquad (15.3)$$

其中，γ=0.57721…是 Euler–Mascheroni 常量。当深度复杂度很低时，重绘量会急剧增加，但增加速度也会逐渐减少。深度复杂度为 4，平均绘制 2.08 次，深度复杂度为 11，平均绘制 3.02 次，但深度复杂度为 12367，平均绘制 10 次。

通过进行粗排序，并从前向后场景的渲染对性能提升会有帮助。这是因为后面绘制的被遮挡物体无需写入颜色缓冲区或 Z 缓冲区中。此外，在到达像素着色器程序之前，像素片元也可以被遮挡剔除硬件丢弃掉。

- **另一种称为“early z pass”的技术对带复杂片元着色器的表面很有用**。即首先渲染 z 缓冲，然后再对整个场景进行渲染。此方法对于避免不必要的像素着色器的计算非常有用，因为只有可见的表面才会进行像素着色的计算。而通过 BSP 树遍历或显式地排序提供了一个粗略的前后顺序，可以提供很多优势，而不需要额外的 Pass。

12.4.6.1 加速片元着色

如果你正在使用长而复杂的片元着色器，那么往往瓶颈就处于片元着色器中。若果真如此，那么可以试试如下这些建议：

- **优先渲染深度**。在渲染主要着色通道（Pass）前，先进行仅含深度的通道（depth-only (no-color) pass）的渲染，能显著地提高性能，尤其是在高深度复杂性的场景中。因为这样可以减少需要执行的片元着色量，以及帧缓冲存储器的存取量，从而提高性能。而为了发挥仅含深度的通道的全部优势，

仅仅禁用颜色写入帧缓冲是远远不够的，同时也应该禁用所有向片元的着色，甚至禁用影响到深度以及颜色的着色（比如 alpha test）。

- **帮助 early-z 优化（即 Z 缓冲优化），来避免多余片元处理** 。现代 GPU 配有设计良好的芯片，以避免对被遮挡片元的着色，但是这些优化依赖场景知识。而以粗略地从前向后的顺序进行渲染，可以明显提高性能。以及，先在单独的 pass 中先渲染深度（见前一条 tip），通过将着色深度复杂度减少到 1，可以有效地帮助之后的 pass（主要的昂贵的 shader 计算的位置）进行加速。

- **在纹理中存储复杂功能。**纹理作为查找表(lookup tables)其实非常好用，而且可以无消耗地过滤它们的结果。一个典型例子便是单位立方体贴图，它仅允许以一个单一纹理查找的代价来高精度地对任意向量进行标准化。

- **将更多每片元的工作移到顶点着色器。**对于优化的大方向而言，正如顶点着色器中的每个物体的计算量工作应该尽可能地移到 CPU 中一样，每顶点的计算也应该尽量被移到顶点着色器（连同在屏幕空间中线性插值计算）。常见的例子包括计算向量和坐标系之间的变换向量。

- **使用必需的最低精度。**诸如 DirectX 之类的 API 允许您在着色器代码中指定精度，以减少精度高所带来的额外计算量。很多 GPU 都可以利用这些提示来减少内部精度以及提高性能。

- **避免过度归一化（Normalization）。**在写 shader 时，对每个步骤的每个矢量都进行归一化的习惯，常常被调侃为“以归一化为乐（Normalization-Happy）”。这个习惯通常来说其实是不太好的习惯。我们应该意识到不改变长度的变换（例如标准正交基上的变换）和不依赖矢量长度的计算（例如正方体贴图的查询）是完全没必要进行归一化后再进行的。

- **考虑使用片元着色器的 LOD 层次细节。**虽然片元着色器的层次细节不像顶点着色器的层次细节影响那么大（由于投射，在远处物体本身的层次细节自然与像素处理有关），但是减少远处着色器的复杂性和表面的通道数，可以减少片元处理的负载。

- **在不必要的地方禁用三线性过滤**。在现代 GPU 结构的片元着色器中计算三线性过滤(Trilinear filtering)，即使不消耗额外的纹理带宽，也要消耗额外的循环。在 mip 级别转换不容易辨别的纹理上，关掉三线性过滤，可以节省填充率。

- **使用尽可能简单的 Shader 类型。**在 Direct3D 和 OpenGL 中，对片元进行着色都有多种方法。举例来说，在 Direct3D 9 中，可以指定片元着色的使用，随着复杂性和功率的增加，有纹理阶段、像素 shader 版本 1.x、像素 shader 版本 2.x，以及像素 shader 3.0 等。一般而言，应该使用最简单的着色器版本来创建预期的效果。更简单的着色版本提供了更多的一些隐式编译选项，通常可以用来让它们更快地被 GPU 驱动程序编译成处理像素的原生代码。

12.4.6.2 减少纹理带宽

如果发现内存带宽是瓶颈，但是大部分结果又要从纹理中取得，那么可以考虑以下方面的优化。

- **减少纹理尺寸。**考虑目标分辨率和纹理坐标。如果玩家是不是真的会看到最高级别的 mip 级别，如果不是，就应该考虑缩减纹理大小。此方法在超载的帧缓冲存储器从非本地存储器（例如系统存储器，

通过 AGP 或 PCI Express 总线）强制进行纹理化时会非常有用。一个 NVIDIA 在 2003 年出品的名叫 NVPerfHUD 的工具可以帮助诊断这个问题，其显示了各个堆（heaps）中由驱动所分配的内存量。

- **压缩所有的彩色纹理**。应该压缩作为贴花或细节的一切纹理，根据特定纹理 alpha 的需要，选用 DXT1、DXT3 或 DXT5 进行压缩。这个步骤将会减少内存使用，减少纹理带宽需求，并提高纹理缓存效率。

- **避免没必要的昂贵纹理格式。**64 位或 128 位浮点纹理格式，显然要花费更多带宽，仅在不得已时才可以使用它们。

- **尽可能地在缩小的表面上使用 mipmapping**。mipmapping 除了可以通过减少纹理走样改善质量外，还可以通过把纹理内存访问定位在缩小的纹理上来改善纹理缓存效用。如果发现某个 mipmapping 使表面看起来很模糊，不要禁用 mipmapping，或增加大的 LOD 级别的基准偏移，而是使用各向异性过滤（anisotropic filtering），并适当调整每个批次各向异性的级别。

12.4.6.3 优化帧缓冲带宽

管线的最后阶段，光栅化操作，与帧缓冲存储器直接衔接，是消耗帧缓冲带宽的主要阶段。因此如果带宽出了问题，经常会追踪到光栅化操作。下面几条技巧将讲到如何优化帧缓冲带宽。

- **首先渲染深度。**这个步骤不但减少片元着色的开销(见上文)，也会减少帧缓冲带宽的消耗。

- **减少 alpha 混合。**当 alpha 混合的目标混合因子非 0 时，则要求对帧缓冲区进行读取和写入操作，因此可能消耗双倍的带宽。所以只有在必要时才进行 alpha 混合，并且要防止高深度级别的 alpha 混合复杂性。

- **尽可能关闭深度写入。**深度写入会消耗额外的带宽，应该在多通道的渲染中被禁用（且多通道渲染中的最终深度已经在深度缓冲区中了）。比如在渲染 alpha 混合效果（例如粒子）时，也比如将物体渲染进阴影映射时，都应该关闭深度写入。另外，渲染进基于颜色的阴影映射也可以关闭深度读取。

- **避免无关的颜色缓冲区清除**。如果每个像素在缓冲区都要被重写，那么就不必清除颜色缓冲区，因为清除颜色缓冲区的操作会消耗昂贵的带宽。但是，只要是可能就应该清除深度和模板缓冲区，这是因为许多早期 z 值优化都依赖被清空的深度缓冲区的内容。

- **默认大致上从前向后进行渲染。**除了上文提到的片元着色器会从默认大致上从前向后进行渲染这个方法中受益外，帧缓冲区带宽也会得到类似的好处。早期 z 值硬件优化能去掉无关的帧缓冲区读出和写入。实际上，没有优化功能的老硬件也会从此方法中受益。因为通不过深度测试的片元越多，需要写入帧缓冲区的颜色和深度就越少。

- **优化天空盒的渲染。**天空盒经常是帧缓冲带宽的瓶颈，因此必须决定如何对其进行优化，以下有两种策略：

（1）最后渲染天空盒，读取深度，但不写入深度，而且允许和一般的深度缓冲一起进行早期 early-z 优化，以节省带宽。

（2）首先渲染天空盒，而且禁用所有深度读取和写入。

以上两种策略，究竟哪一种会节省更多开端，取决于目标硬件的功能和在最终帧中有多大部分的天空盒可见。如果大部分的天空盒被遮挡，那么策略（1）更好，否则，策略（2）可以节省更多带宽。

- **仅在必要时使用浮点帧缓冲区。**显然，这种格式比起较小的整数格式来说，会消耗更多的带宽，所以，能不用就不用。对多渲染目标(Multiple Render Targets，MRT)也同样如此。
- **尽可能使用 16 为的深度缓冲区**。深度处理会消耗大量带宽，因此使用 16 位代替 32 位是极有好处的，且 16 位对于小规模、不需要模板操作的室内场景往往就足够了。对于需要深度的纹理效果，16 位深度缓冲区也常常足够渲染，如动态的立方体贴图。
- **尽可能使用 16 位的颜色**。这个建议尤其适用于对纹理的渲染效果，因为这些工作的大多数，用 16 位的颜色能工作得很好，例如动态立方体贴图和彩色投射阴影贴图。

综上，现代 GPU 能力和可编程性的增强，使得改善机器性能变得更复杂。无论是打算加速应用程序的性能，还是希望无成本地改善图像质量，都需要对渲染管线的内部工作原理有深刻理解。而 GPU 管线优化的基本思路是，通过改变每个单位的负荷或计算能力来识别瓶颈，然后运用每个传递单元工作原理的理解，系统地解决那些瓶颈。

12.5 主流性能分析工具列举

有很多不错的分析图形加速器和 CPU 使用的的工具，以及性能优化相关的 Profiling 工具，在这里，将主流的工具进行列举：

- Adreno Profiler
- GPA
- Tegra Graphics Debuger
- Xcode Profiler
- Xcode Instruments
- PIX for Windows (for DirectX)
- gDEBugger (for OpenGL)
- NVIDIA’s NVPerfKit suite of tools
- ATI’s GPU PerfStudio
- Apple’s OpenGL Profiler

- Linux 上的 Valgrind http://valgrind.org/
- NVIDIA 出品的 Nsight 系列性能优化套件
- https://developer.nvidia.com/gameworks-tools-overview
- CPU 端内循环优化工具 Vtune https://software.intel.com/en-us/intel-vtune-amplifier-xe
- AQTime – 代码 profilers 工具 https://smartbear.com/product/aqtime-pro/overview/

现今主流游戏引擎提供的 Profiler 有：

- Unreal Engine 的一列系列 Profiler 工具集 https://docs-origin.unrealengine.com/latest/INT/Engine/Performance/
- Unity 的 Profiler 和后续新加入的 Frame Debugger
  - Unity - Profiler https://docs.unity3d.com/Manual/ProfilerWindow.html
  - Unity – Frame Debugger https://docs.unity3d.com/Manual/FrameDebugger.html

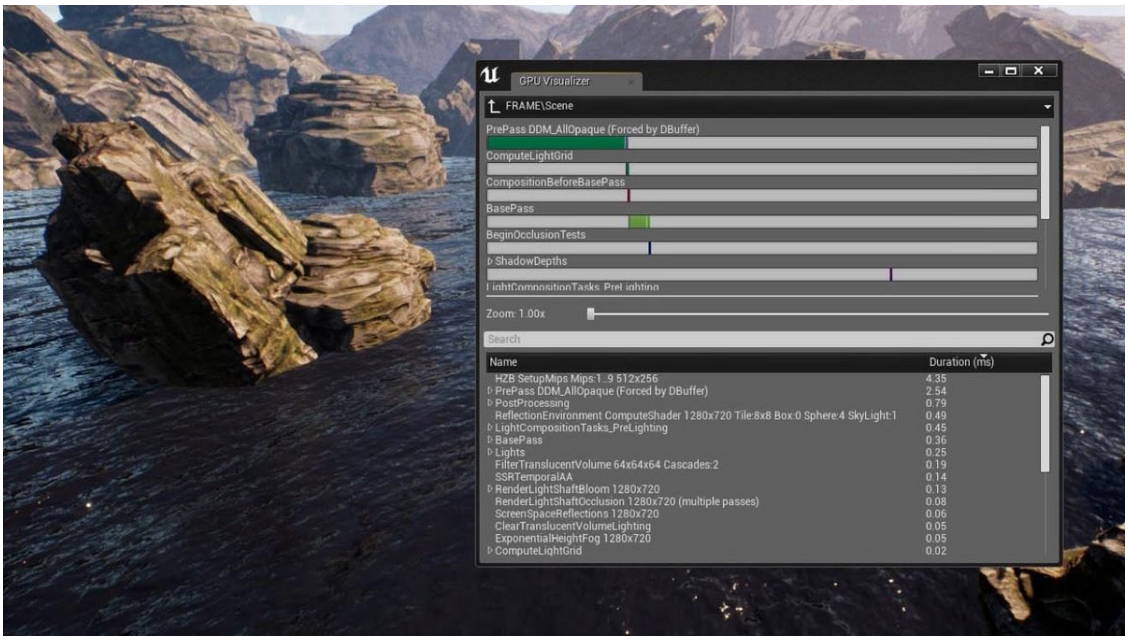


图 6 Unreal Engine 的 GPU Visualizer

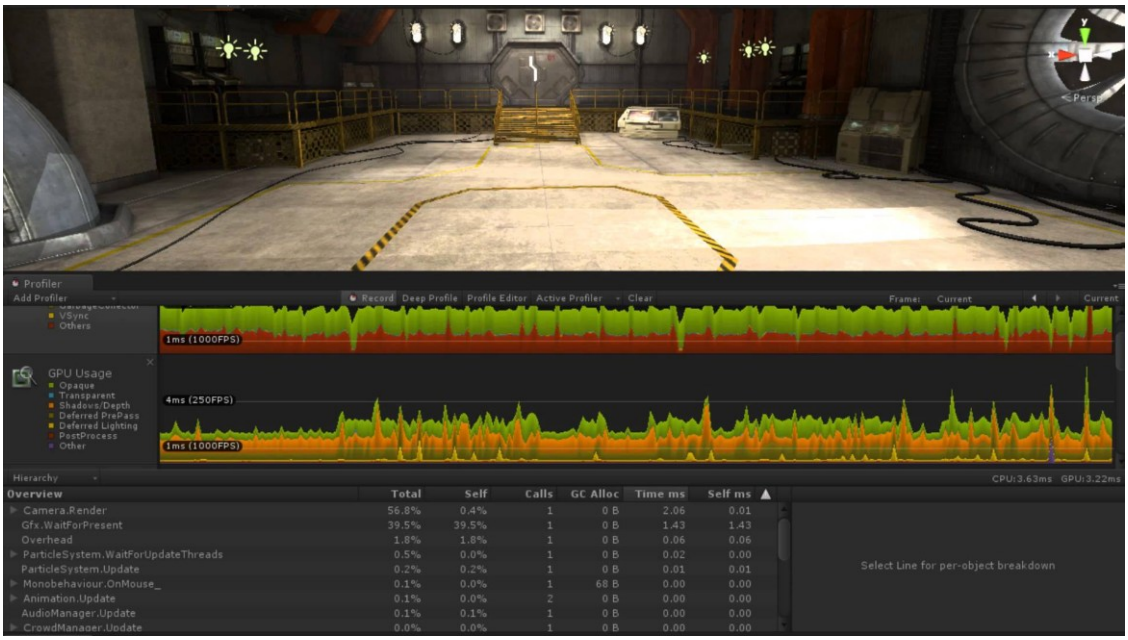


图 7 Unity 的 Profiler

12.7 更多性能优化相关资料

- 虽然有点过时，Cebenoyan 的文章[1]概述了如何找到提高效率的瓶颈和技术。
- 《NVIDIA's extensive guide》[8]包含了相关的各种主题。
- 一些很赞的 C++优化指南包括 Fog 的文章[9]和 Isensee 的文章[10]。

12.8 Reference

[1] Cebenoyan, Cem, “Graphics Pipeline Performance,” in Randima Fernando, ed.,GPU Gems, Addison–Wesley, pp. 473 – 486,2004. Cited on p. 681, 699, 701, 716,722

[2] Hecker, Chris, “More Compiler Results, and What To Do About It,” Game Developer, pp. 14 – 21, August/September 1996. Cited on p. 705

[3] Blinn, Jim, “Optimizing C++ Vector Expressions,” IEEE Computer Graphics &Applications, vol. 20, no. 4, pp. 97 – 103, 2000. Also collected in [110], Chapter 18.Cited on p. 707

[4] Deering, Michael, “Geometry Compression,” Computer Graphics (SIGGRAPH 95 Proceedings), pp. 13 – 20, August 1995. Cited on p. 555, 713

[5] Calver, Dean, "Vertex Decompression Using Vertex Shaders," in Wolfgang Engel, ed., ShaderX, Wordware, pp. 172 – 187, May 2002. Cited on p. 713

[6] Zarge, Jonathan, and Richard Huddy, "Squeezing Performance out of your Game with ATI Developer Performance Tools and Optimization Techniques," Game Developers Conference, March 2006. http://ati.amd.com/developer/gdc/2006/GDC06-ATI Session-Zarge-PerfTools.pdf Cited on p. 270, 699, 700, 701,702, 712, 713, 722, 847

[7] Purnomo, Budirijanto, Jonathan Bilodeau, Jonathan D. Cohen, and Subodh Kumar, "Hardware-Compatible Vertex Compression Using Quantization and Simplification," Graphics Hardware, pp. 53 – 61, 2005. Cited on p. 713

[8] NVIDIA Corporation, "NVIDIA GPU Programming Guide," NVIDIA developer website, 2005. http://developer.nvidia.com/object/gpu programming guide.htmlCited on p. 38, 282, 699, 700, 701, 702, 712, 722

[9] Fog, Agner, Optimizing software in C++, 2007. Cited on p. 706, 722

[10] Isensee, Pete, "C++ Optimization Strategies and Techniques," 2007. Cited on p.706, 722

附录:《Real-Time Rendering 3rd》核心知识思维导图

考虑到图片过大，建议另存为本地图片后放大查看。