[redacted]

本科毕业设计（论文）

基于 WinGDI 的 3D 图形软光栅化渲染器研究与设计

学生姓名 [redacted]

院系名称 [redacted]

专业名称 [redacted]

班　　级 [redacted]

学　　号 [redacted]

指导教师 [redacted]

完成时间 2023 年 4 月 20 日

摘 要

图形化界面的出现是计算机发展的一大飞跃，从而使计算机能够真实地还原现实世界，模拟一些现实世界中难以实现的内容，就离不开基于计算机图形学的 3D 渲染技术。3D 渲染技术在影视制作、游戏开发、建筑设计、工业设计和医学应用等方面有着非常广泛的应用，并且也在高质量实时渲染、AI 智能渲染、AR /VR 技术上有着非常好的前景，因此对于 3D 渲染方面的相关研究和优化算法设计也是络绎不绝。

本文在对现代计算机图形学与 3D 图形渲染的发展现状，进行了简单的了解与分析的基础上，拟根据渐进式图形渲染管线的设计思路来设计一个 3D 图形软光栅化渲染器，将模型文件通过坐标-投影变换，屏幕投影，纹理映射，然后加上光照反射模型与阴影映射来渲染出一张 RGB 三通道图像，并调用 Windows 的原生图形设备接口 WinGDI 来渲染在图形窗口中。

另外，本文还将试图从渲染质量和渲染速度两个方面对本渲染器的图形渲染做出优化，通过引入超采样、三线性插值等反走样技术以及 CUDA 并行编程技术，来提高渲染器的渲染精度和渲染效率。

关键词：图形；渲染；3D

Abstract

The emergence of graphical interface is a great leap in the development of computer, and the computer can truly restore the real world and simulate some difficult to achieve in the real world, which is inseparable from the 3D rendering technology based on computer graphics. 3D rendering technology in the film and television production, game development, architectural design, industrial design and medical application has a very wide application, and also in high quality real-time rendering, AI intelligent rendering, AR / VR technology has a very good prospect, so for 3D rendering related research and optimization algorithm design is also an endless stream.

Based on a brief understanding and analysis of the development status of modern computer graphics and 3D rendering, this article intends to design a 3D graphics software rasterization renderer based on the design concept of progressive graphics rendering pipeline. The renderer will take the model file and perform coordinate-projection transformation, screen projection, texture mapping, and then add lighting reflection models and shadow mapping to render an RGB image. The rendering will be done using Windows' native graphics device interface, WinGDI, and displayed in a graphics window.

In addition, this article will attempt to optimize the graphics rendering of the renderer in terms of rendering quality and rendering speed. This will be achieved by introducing anti-aliasing techniques such as supersampling, trilinear interpolation, as well as CUDA parallel programming technology to improve the rendering accuracy and efficiency of the renderer.

Keywords: graphics; render; 3D

目 录

第 1 章 概述

1.1 研究背景

图形化界面的出现是计算机发展的一大飞跃，而使计算机能够真实地还原现实世界，模拟一些现实世界中难以实现的内容，就离不开基于计算机图形学的 3D 渲染技术。计算机图形学是一门研究使用计算机生成、处理、显示图形的学科。它涉及计算机科学、数学、物理学和艺术等多个领域，广泛应用于计算机游戏、动画、视频、图像处理、工业设计、医学影像、科学可视化等领域[1]。在电脑绘图中，渲染（英语：render，又称为绘制或彩现）是指利用软件生成图像的过程，该过程依靠模型以语言或数据结构进行严格定义，描述了三维物体或虚拟场景的几何、视点、纹理、照明和阴影等信息。图像是数字图像或位图图像[2, 3]。渲染技术广泛应用于视频编辑软件中，用于计算视频效果并生成最终输出。渲染是三维计算机图形学的重要研究领域之一，与其他技术密切相关，而在图形流水线中，渲染通常是最后一个关键步骤，用于生成模型和动画的最终显示效果[4]。随着计算机图形的不断复杂化，渲染技术变得越来越重要，自 20 世纪 70 年代以来得到了广泛的应用[5]。

计算机图形学的发展可以追溯到 20 世纪 50 年代，当时计算机的运算能力非常有限，只能处理简单的图形。1963 年 1 月，MIT 林肯实验室的萨瑟兰完成了一篇关于图形系统人机通信的博士论文，当时他只有 24 岁。在论文中，萨瑟兰引入了分层存储符号的数据结构，开发了交互技术，可以使用键盘和激光笔实现定位、选项和绘图，并提出了很多至今仍在使用的图形学的其他基本思想和技术。因此，萨瑟兰的博士论文被认为是计算机图形学的奠基之作，也是现代计算机辅助设计的肇始[6, 7]。随着计算机技术的发展，计算机图形学也不断发展，涉及许多方面的研究，如图形输入、输出、处理、显示、建模、动画、渲染、交互等。在计算机图形学的研究中，数学在很大程度上起到了支撑作用。例如，计算机图形学中常用的坐标系统、几何变换、三角函数、向量计算、曲线、曲面等都涉及数学知识。在计算机图形学中，经常使用图形算法来处理图形信息。图形算法是指能够解决图形问题的算法。例如，图形算法可以用来求解图形的交点、判断图形是否相交、求解图形的面积、求解图形的几何变换等。计算机图形学中还使用许多图形数据结构来存储、表示图形信息。图形数据结构是指能够用来存储图形信息的数据结构。例如，可以使用点、线、多边形、三维模型等图形数据结构来表示图形。

计算机图形学的发展也促进了计算机硬件的发展。例如，为了支持计算机图形学的应用，计算机硬件需要拥有足够的图形处理能力，包括图形输入、输出、处理、显示等功能。为了满足这些需求，计算机硬件不断发展，例如出现了图形加速器、图形处理器等硬件设备[8, 9]。

1.2 研究意义

计算机图形学和 3D 渲染技术的研究之一在于创造有效的视觉交流方式。在科学领域，通过图形学可将科学成果通过可视化的方式展示给公众；在娱乐领域，如在 PC 游戏、手机游戏、3D 电影和电影特效中，计算机图形学和 3D 渲染技术的作用越来越重要；在创意、艺术创作、商业

广告、产品设计等领域，图形学也扮演着重要的基础角色[10]。然而，这种思想在科学领域中直到 1987 年才被正式提出，即在有关科学计算可视报告的报告中。该报告引用了 Richard Hamming 1962 年的经典观点：“计算的目的是为了洞察事物的本质，而不是获得数字。”报告指出，计算机图形学在帮助人们从图形图像的角度理解事物本质方面起着重要作用，因为图形图像比单纯数字具有更强的直观性[11]。

计算机图形学和渲染技术的应用也极为广泛。例如，在科学计算中，它可以帮助人们完成可视化的任务。在当今医学领域中，许多重要的操作通常需要依靠精密的机械，并在实施之前进行可视化处理。在手术过程中，只有通过可视化才能判断机械操作的准确性。计算机图形学的可视化应用在医学领域有许多成功的实例，例如将 CT 扫描数据转化为图像，以直观地显示患者的情况。此外，计算机图形学还广泛应用于平面设计与动画领域，如广告设计、网页设计、电脑绘画艺术设计等。这些领域中的从业人员必须掌握计算机图形学和渲染技术[12, 13]。计算机图形学和 3D 渲染技术不仅在视觉传达方面具有广泛应用，还被广泛应用于工业设计和制造领域。例如，在 CAD 领域，计算机图形学的应用已经扩展到了集成电路、印刷电路板、电子线路和网络分析等领域，其优势十分明显。随着计算机网络的发展，异地异构系统的协同设计已经成为 CAD 领域的热门课题之一[14, 15]。

1.3 国内外研究现状

计算机图形学与渲染是一个充满活力和不断发展的领域。自光栅显示器的诞生以来，图形学算法迅速发展，70 年代应用于 CAD 等方面。80 年代中期以来，大规模集成电路和光线跟踪算法的提出推动了真实感图形的成熟。80-90 年代图形学应用范围更广，提出真实性和实时性要求。进入千禧年后，图形处理单元的成熟和 3D 图形 GPU 等技术的发展推动了 CGI 的普及。到了 2010 年后，预渲染的图形已经达到真实照片级别，纹理映射等算法集成到渲染引擎中使用着色器实现多种复杂操作[16]。目前，随着硬件性能和算力的不断提升，人们对图形渲染的速度和质量也提出了更高的要求。图形渲染技术由传统的光栅化技术逐渐过渡为现代的不同类型的光线追踪技术，包括路径追踪（Path Tracing）、光子映射（Photon Mapping）、梅特罗波利斯光传输（Metropolis Light Transport）等等，致力于模拟出更真实的光线传播路径从而获得更真实的光照效果[17-20]。以 Nvidia 为代表的专用图形处理器也大大加速了图形渲染过程。除此之外，随着人工智能与深度学习的不断发展，已经出现了许多关于 AI 渲染的相关研究，譬如通过 Tensor Core 硬件加速的深度学习对实时渲染的图片实现非常高质量的超分辨率，从而大幅提升游戏渲染的性能[21, 22]。

然而在图形学渲染领域发展的过程中，仍然存在着许多需要通过研究去解决的问题。例如，对于各种各样类型的曲线的仿真，对于透明物体和水波的渲染，光线在微观领域上的物理性质（波粒二象性），对动物和人体的毛发渲染，以及基于这些理论研发出的渲染引擎与各种各样图形加速硬件的适配问题，等等[23-26]。就图形学中较为热门的光线追踪算法而言，传统的 Whitted-Style 光线追踪的运算速度太慢，因此便诞生了基于蒙特卡洛积分和概率学的路径追踪。但路径追踪在光照强度剧烈变化的场景下会出现许多噪点，于是又有了梅特罗波利斯光传输解决这一

问题，但目前这还不能算是完美的解决方案。又如在物体表面材质的研究中，研究者们提出了许许多多的针对不同材质表面的 BRDF（双向反射分布函数）模型。例如将模型的表面无限细分所得到的微表面模型（Microfacet BRDF），可以很真实地渲染出粗糙表面；以及针对各向异性表面提出的 BRDF（Anisotropic BRDF）模型和针对动物和人体的毛发提出的双圆柱体反射模型（Double Cylinder Model），等等[27-29]。对于工业界来说，如何加快图形渲染的速度的同时保证渲染的真实性一直是各大渲染引擎所不懈追求的目标。所谓搏二兔不得一兔，目前在渲染精度与渲染速度上仍需要作出一些取舍。

随着计算机视觉，深度学习等相应学科近些年的发展，计算机图形学也出现了许多与这些学科相交叉的研究，并且都在不断地探索新的技术和应用。例如图像风格转换[30]，它利用深度学习模型可以学习不同风格的图像特征，并将这些特征应用于新的图像中，以生成具有新风格的图像；以及虚拟现实技术，它从现实世界中获取图像或视频，并将其应用于虚拟场景中。深度学习技术也可以用于提高虚拟场景的逼真度，等等。未来，随着技术的不断进步，计算机图形学将会在更多领域得到应用，并创造出更加出色的成果。

第 2 章 图形学光栅化渲染理论研究

2.1 视图变换

当向世界中添加了模型文件，并确定好模型和相机的位置后，接下来就是要想办法将相机所看到的画面呈现到屏幕上了，这一步可以通过对模型的视图变换来完成。

视图变换主要分为以下几个部分：

(1) 模型位置变换(model location transformation)：模型位置变换得到的是物体与摄像机的相对位置，因此它可以表现摄像机(或者说眼睛)所看到的东西。

(2) 相机方向变换(camera direction transformation)：根据摄像机所注视的方向，继续调整物体与摄像机的相对位置。

(3) 投影变换(projection transformation)：在摄像机变换之后，就可以得到所有可视范围内的物体相对于摄像机的相对位置坐标，之后根据具体情况选择平行投影或是透视投影，将三维空间投影至标准二维平面([-1, 1]^2)之上 （tips：这里的 z 并没有丢掉，为了之后的遮挡关系检测）。

(4) 视口变换(viewport transformation)：将处于标准平面映射到屏幕分辨率范围之内，即[-1, 1]^2⇒[0, width]*[0, height], 其中 width 和 height 指屏幕分辨率大小[31]。

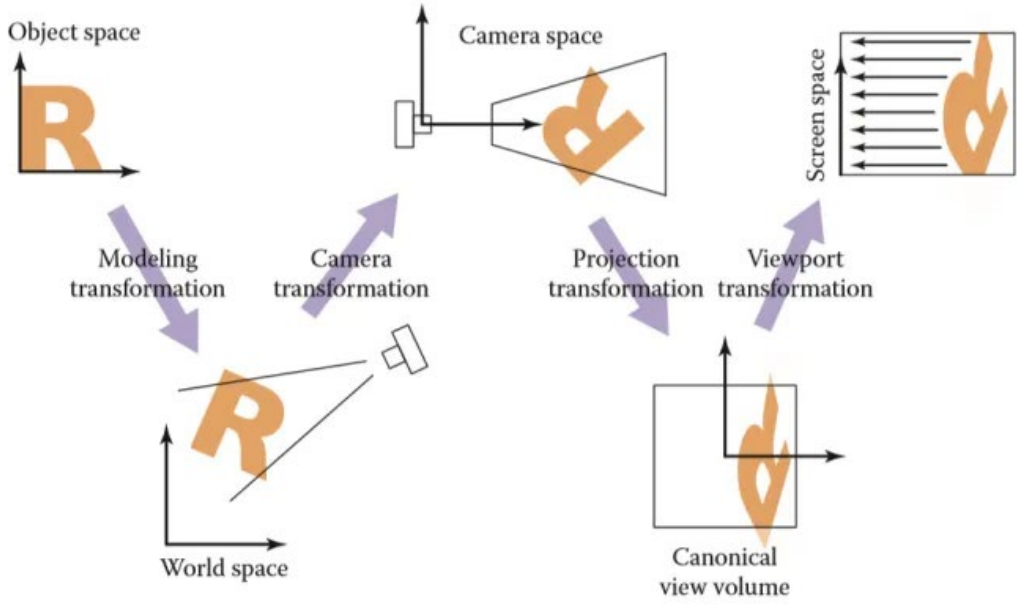


图 2-1 视图变换的大致流程

2.1.1 基于线性代数的空间变换

2.1.1.1 使用矩阵来表示空间中的缩放和旋转

众所周知，矩阵在线性代数中的意义是对向量进行线性变换，当矩阵被某一向量左乘时，它就像一个输入和输出都为向量的线性函数。

$$V_{out} = f(V_{in}) \tag{2-1}$$

可以将被左乘的矩阵看作是某一新的向量空间的基向量集合，而向量左乘该矩阵后所得到的向量就是该向量在新的向量空间中的表示。以 3 维为例，对向量$V_{in}(1,2,3)^T$，它所在的向量空间的基向量为 3 个互相正交的单位向量$V_x(1,0,0)^T$，$V_y(0,1,0)^T$，$V_z(0,0,1)^T$。想要在一个基向量分别为$V_x(1,1,0)^T$，$V_y(0,1,1)^T$，$V_z(1,0,1)^T$的向量空间中表示原向量，就需要对原向量进行左乘。

$$\begin{bmatrix}1 & 0 & 1\\ 1 & 1 & 0\\ 0 & 1 & 1\end{bmatrix}\begin{bmatrix}1\\ 2\\ 3\end{bmatrix}=\begin{bmatrix}4\\ 3\\ 5\end{bmatrix} \tag{2-2}$$

此时$(4,3,5)^T$即为新的向量空间中所表示的$(1,2,3)^T$向量。借此可以轻松通过改变基向量的大小和方向来实现空间中的缩放和旋转。将基向量的长度缩短或伸长，可以实现空间中的缩放；将两个相互正交的基向量沿着垂直于另一基向量的平面上旋转，可以实现空间中平行于某一坐标轴的旋转。如果想要实现沿空间中任意一个方向的旋转，则需要用到著名的罗德里格旋转公式（Rodrigues' rotation formula）。

$$A=\begin{bmatrix}x_u & x_v & x_w\\ y_u & y_v & y_w\\ z_u & z_v & z_w\end{bmatrix}\begin{bmatrix}\cos\varphi & -\sin\varphi & 0\\ \sin\varphi & \cos\varphi & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x_u & y_u & z_u\\ x_v & y_v & z_v\\ x_w & y_w & z_w\end{bmatrix} \tag{2-3}$$

简单来说，就是先将该向量需要沿着旋转的轴旋转到某一坐标轴上（这里是 xoy 轴），然后沿着这个轴作旋转，最后再将旋转好了的轴还原到原来的方向，罗德里格旋转公式也就是由上述操作复合而成的三个矩阵。

2.1.1.2 引入齐次坐标来表示仿射变换

目前已经可以使用 3 维矩阵来表示空间向量的缩放和旋转了，理论上空间向量的所有变换操作都可以使用 3 维矩阵来完成，但是却不能表示一个非常重要的变换：位移。

但为什么需要位移呢？这是因为无法只使用向量来表示空间中的除了原点以外的任何一个点的同时，又能使用矩阵来表示该点的变换。所有的空间向量的出发点都是基于原点的，那么如果想要对一个空间上的直线（不与坐标轴重合）作绕该直线旋转的运动时，3 维向量和矩阵就无法表示这一变换了。如果想要解决这一点，就需要坐标系能够在表示向量大小，方向的同时，还能够将点的位移融合进来。而齐次坐标很好地做到了这一点。

齐次坐标使用 N+1 个数来表示 N 维坐标，在 3 维空间中，齐次坐标使用 4 维矩阵和向量来表示 3 维空间中的变换。齐次坐标中的向量分为两种类型：

(1) 当最后一维的值为 0 时，该向量表示一个起点在原点的普通向量，这与之前的 3 维向量一样，例如$V_h(1,2,3,0)^T$表示 $V_o(1,2,3)^T$向量。

(2) 当最后一维的值为非 0 时，该向量在齐次化（同除以最后一位）后，表示一个位于向量终点的点。例如$V_h(1,2,3,1)^T$表示位于$(1,2,3)^T$的没有方向的一个点。

引入齐次坐标之后，就可以通过齐次坐标来表示点的位移了。而之前所说的例子“对一个空间上的直线（不与坐标轴重合）作绕该直线旋转的运动”，就可以通过“将该直线平移回原点——利用罗德里格旋转公式进行旋转——再将直线平移回去”来实现。而在实现渲染器的过程中，齐次坐标还被用来实现空间中的投影变换。

2.1.2 模型位置变换

这一步可以说是最简单的一步，利用基础的变换矩阵将世界当中的物体调整至合适的地方。假设相机的位置是$(x_c，\ y_c，\ z_c，\ 1)^T$（以后都将使用 4 维齐次坐标来表示 3 维向量和空间点），那么对于空间中的所有的点，应用以下变换。

$$A_1 = \begin{bmatrix} 1 & 0 & 0 & -x_c \\ 0 & 1 & 0 & -y_c \\ 0 & 0 & 1 & -z_c \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2-4}$$

相当于将相机摆到了坐标原点后，其他点的位置发生了相应的位移。此时其他点的坐标到原点的距离即为到相机的相对距离。

2.1.3 相机方向变换

在经过模型位置变换以后，相机的位置到了原点，但是此时相机的方向并不确定，为此先规定：

(1) 相机的方向为 z 轴负方向。

(2) 相机的正上方为 y 轴正方向。

此时就可以利用矩阵来将相机旋转到正确的方向了。

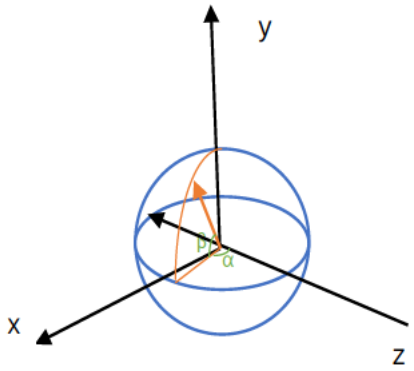


图 2-2 相机在空间坐标系中的方向

如图，假设橙色箭头为相机的朝向，利用 α 和 β 两个角来表示该朝向，那么要想使它指向 z 轴的负方向，就需要先让它先向下旋转 β 度到平面上，再向右旋转 α 度到 z 轴的负方向。那么如何使用矩阵表示这一变换呢？直接表示显然是不方便的，因为第一个向下旋转的变换并不与坐标轴平行。因此可以变换一下顺序，先让它向右旋转 α 度到 yoz 平面上，再向下旋转 β 度到 z 轴的负方向，这样效果是一样的，并且两次变换都是与坐标轴平行的变换。所以有：

$$A_{21} = \begin{bmatrix} cos\ \alpha & 0 & sin\ \alpha & 0 \\ 0 & 1 & 0 & 0 \\ -sin\ \alpha & 0 & cos\ \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2-5}$$

$$A_{22} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\,\beta & sin\,\beta & 0 \\ 0 & -sin\,\beta & cos\,\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2-6}$$

其中A_{21}表示向右旋转 α 度，A_{22}表示向下旋转 β 度。

当相机的方向指向 z 轴后，它的正上方应当平行于 xoy 面，像这样：

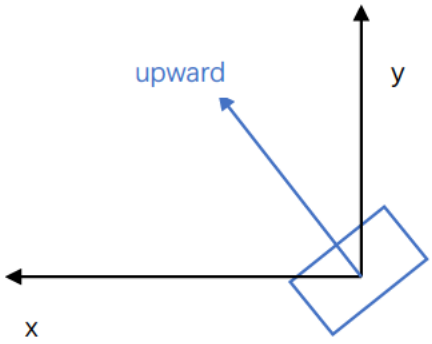


图 2-3 相机的正上方

因此还需要再将相机的正上方旋转到 y 轴：

$$A_{23} = \begin{bmatrix} cos\,\gamma & -sin\,\gamma & 0 & 0 \\ sin\,\gamma & cos\,\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2-7}$$

最后将这三个变换复合，就得到了相机方向变换矩阵A_2：

$$A_2 = A_{23} * A_{22} * A_{21} \tag{2-8}$$

2.1.4 投影变换

投影变换分为正交投影变换和透视投影变换。正交投影是相对简单的一种，坐标的相对位置都不会改变，所有光线都是平行传播；透视投影就是最类似人眼所看东西的方式，遵循近大远小的原则，这里采用透视投影来更真实的模拟物体投影到照相机上的情况。

透视投影变换属于前文所提到的仿射变换，每一个不同的点上的变换都不尽相同，可以把透视变换大致概括为“将近大远小的透视空间‘压缩’成水平空间”。

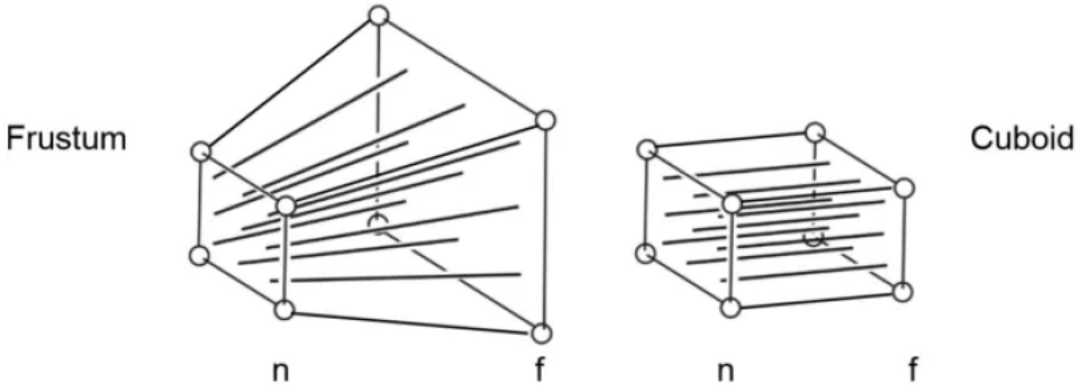


图 2-4 透视变换示意图

透视变换之后，所形成的透视空间中的物体就非常接近人的眼中所看到的物体了。下面讨论如何得到透视变换的矩阵A_3。

很容易看出透视投影变换有一近一远两个平面，不妨设相机到近平面的距离为 n（near），到远平面的距离为 f（far），以垂直于 yoz 平面方向看过去会是这个样子：

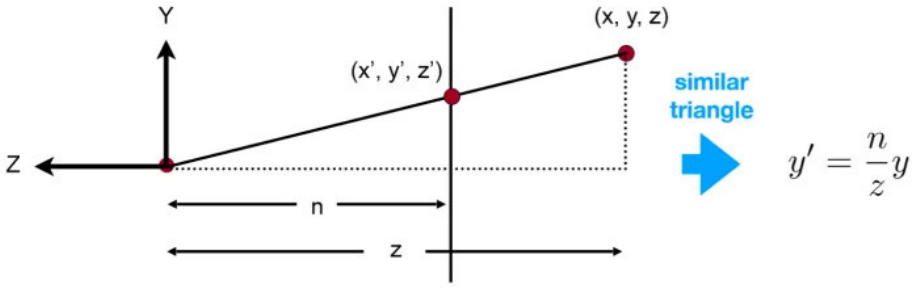


图 2-5 透视变换的平面分析图

由相似三角形性质，在远平面上：

$$\begin{cases} x' = \frac{n}{f}x \\ y' = \frac{n}{f}y \end{cases} \tag{2-9}$$

经过透视变换后有：

$$[x \quad y \quad f \quad 1]^T \Rightarrow [x' \quad y' \quad f \quad 1]^T \tag{2-10}$$

由此可知：

$$A_3 * [x \quad y \quad f \quad 1]^T = \left[\frac{n}{f}x \quad \frac{n}{f}y \quad f \quad 1\right]^T \tag{2-11}$$

在近平面上所有点位置不发生改变，因此有：

$$A_3 * [x \quad y \quad n \quad 1]^T = [x \quad y \quad n \quad 1]^T \tag{2-12}$$

这样就得到了三个与A_3相关的等式，但是却没有办法解出来，这时就需要用到齐次坐标相乘同一数后不变的性质：

$$\left[\frac{n}{f}x \quad \frac{n}{f}y \quad f \quad 1\right]^T = [nx \quad ny \quad f^2 \quad f]^T \tag{2-13}$$

$$[x \quad y \quad n \quad 1]^T = [nx \quad ny \quad n^2 \quad n]^T \tag{2-14}$$

由(2-10)(2-12)和(2-11)(2-13)，当 x 和 y 只分别出现在向量的前面两个维度时，可假定A_3如下：

$$A_3 = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & c & d \end{bmatrix} \tag{2-15}$$

其中 a，b，c，d 为待确定的未知量。根据(2-10)(2-12)和(2-11)(2-13)的后两个维度，有：

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} f & n \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} f^2 & n^2 \\ f & n \end{bmatrix} \tag{2-16}$$

可知：

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} f^2 & n^2 \\ f & n \end{bmatrix} \begin{bmatrix} f & n \\ 1 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} f+n & -fn \\ 1 & 0 \end{bmatrix} \tag{2-17}$$

综上可得：

$$A_3 = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2-18}$$

这样就得到了所需要的的透视变换矩阵[17, 32]。

2.1.5 视口变换

视口变换就是将处于标准 xoy 平面映射到屏幕分辨率范围之内，即$[-1,1]^2 \Rightarrow [0,w] * [0,h]$，其中 w 和 h 指屏幕分辨率的高度和宽度，这一步利用一个简单的矩阵便可实现：

$$A_4 = \begin{bmatrix} \frac{w}{2} & 0 & 0 & \frac{w}{2} \\ 0 & -\frac{h}{2} & 0 & \frac{h}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2-19}$$

最终所有的视图变换工作就完成了：

$$A = A_4 * A_3 * A_2 * A_1 \tag{2-20}$$

2.2 屏幕投影

将世界对象中所有的顶点乘以视图矩阵后，就可以将其逐三角形投影到屏幕上了。在世界对象中所有的模型都由三角形面组成，而三角形的位置和大小等也都由顶点决定，这里需要将视图变换过后的三角形按照一定顺序来投影到屏幕上，读者很容易知道这个顺序是根据三角形位置的先后来的，先投影靠后面的三角形，再投影靠前面的三角形，得到的结果才符合遮挡关系，下面讨论实现三角形投影的方案。

2.2.1 判断像素与三角形面的位置关系

对于每一个三角形而言，当它所有的顶点以及顶点法线经过了视图变换之后，它就已经位于屏幕空间中了，它的 x 与 y 的值对应的是屏幕上的像素，而它的 z 值则能够描述三角形的前后关系。因此可以逐像素判断这个三角形在屏幕上的投影是否覆盖了该像素，如果该像素位于三角形的投影上面，就把它显示出来，这就是三角形的光栅化过程。下面讨论判断某一个像素是否位于三角形上。

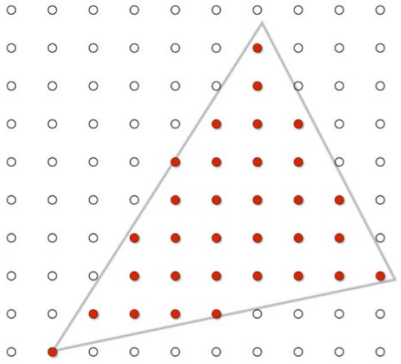


图 2-6 三角形的光栅化

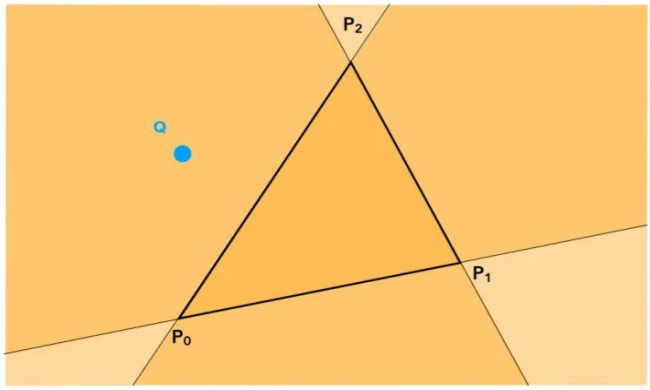


图 2-7 判断点与三角形的关系

如上图所示，有三角形$P_0P_1P_2$，以及一个三角形外的点 Q。假设三角形由三个逆时针向量$\overrightarrow{P_0P_1}$，$\overrightarrow{P_1P_2}$，$\overrightarrow{P_2P_0}$组成，且P_0，P_1，P_2各自有指向点 Q 的向量$\overrightarrow{P_0Q}$，$\overrightarrow{P_1Q}$，$\overrightarrow{P_2Q}$。可以观察到，当 Q 点位于三角形内时，$\overrightarrow{P_0Q}$，$\overrightarrow{P_1Q}$，$\overrightarrow{P_2Q}$分别位于$\overrightarrow{P_0P_1}$，$\overrightarrow{P_1P_2}$，$\overrightarrow{P_2P_0}$的左侧，而当 Q 点位于三角形外时（譬如说图中位置），$\overrightarrow{P_0Q}$，$\overrightarrow{P_1Q}$分别位于$\overrightarrow{P_0P_1}$，$\overrightarrow{P_1P_2}$的左侧，而$\overrightarrow{P_2Q}$却位于$\overrightarrow{P_2P_0}$的右侧！多次尝试后，我们发现了如下规律：仅当 Q 点位于三角形内时，它与三角形三个逆时针向量才满足全部位于其左侧的关系。

那么如何量化这种“全部位于其左侧的关系”呢？这需要使用到向量的叉乘性质。由于是在平行于屏幕的平面上判断，所以使用二维向量即可，而二维向量叉乘的结果刚好是一个数值，符合右手定则（逆时针为正），因此有如下结论：

- 仅当$\begin{cases}\overrightarrow{P_0P_1}\times\overrightarrow{P_0Q}>0\\ \overrightarrow{P_1P_2}\times\overrightarrow{P_1Q}>0\\ \overrightarrow{P_2P_0}\times\overrightarrow{P_2Q}>0\end{cases}$同时成立，即可判断该点位于三角形内。

2.2.2 深度缓存投影

判断好三角形与像素的位置关系后，现在可以顺利地将一个三角形投影到屏幕上了，不过有些时候投影出来的效果可能是下图左侧这样的。

图 2-8 遮挡关系不正确的渲染

渲染效果不尽人意，有些看不到的三角形也跑到前面来了，其实这就是三角形的遮挡关系出现了问题，渲染器需要正确地根据三角形的前后位置关系来渲染三角形，才能出现如右图一样正确的效果。

每一个三角形的顶点的 z 坐标表示它目前在屏幕空间到屏幕的距离，可以据此把三角形上任何一个点在屏幕空间中的深度表示出来，然后再把它记录到缓存上。之后当有新的三角形与原来的三角形出现遮挡关系时，就可以比较缓存上的值与新的三角形在该像素处的深度，以此来决定是否覆盖原来的三角形。将所有的三角形遍历一遍，最后得到的缓存中便是所有没有被遮挡的三角形。将深度值按照颜色深浅渲染到屏幕上的效果如下。

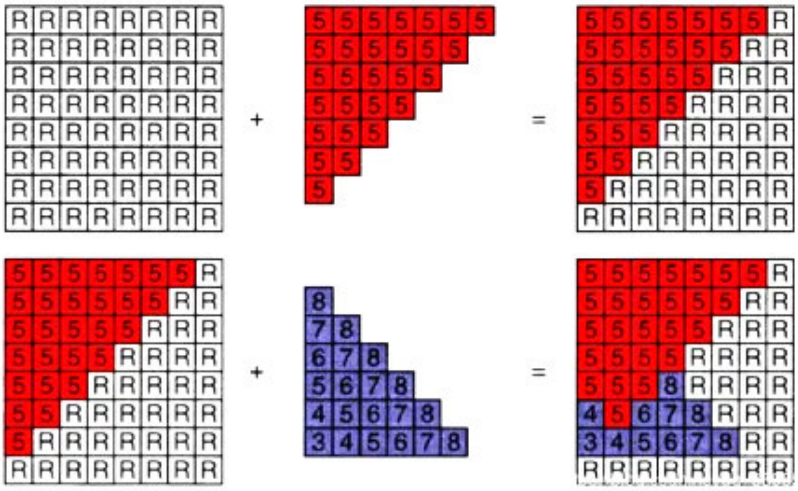


图 2-9 深度缓存示例

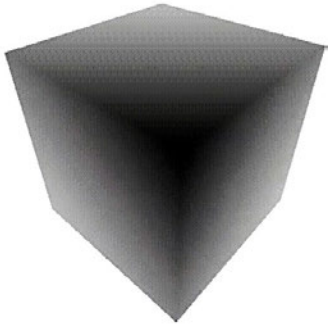


图 2-10 使用深度缓存后的效果

2.3 纹理映射

纹理映射说白了就是“把一张张图片贴到一个个三角形上”，来让模型获得真实的颜色和材质。一些图片规定了模型的颜色，而另一些图片则通过法线规定了模型的表面粗糙程度，无论如何，都需要通过纹理映射来将图片映射到物体上去。

2.3.1 纹理坐标与重心坐标插值

纹理坐标又称 UV 坐标，规定了一个图片上的某个点与模型中的顶点的一一映射关系，并规定所有的纹理都位于1×1的小正方形中。屏幕中的三角形需要根据三个顶点来找到纹理上的三个点，并将这三个点所酿成的三角形再映射到屏幕中。而对于每一个像素而言，相当于需要根据它来找到世界空间中的相应位置，再根据纹理坐标找到它所对应的纹理位置，最后把该位置上点的颜色再显示出来，如下图。

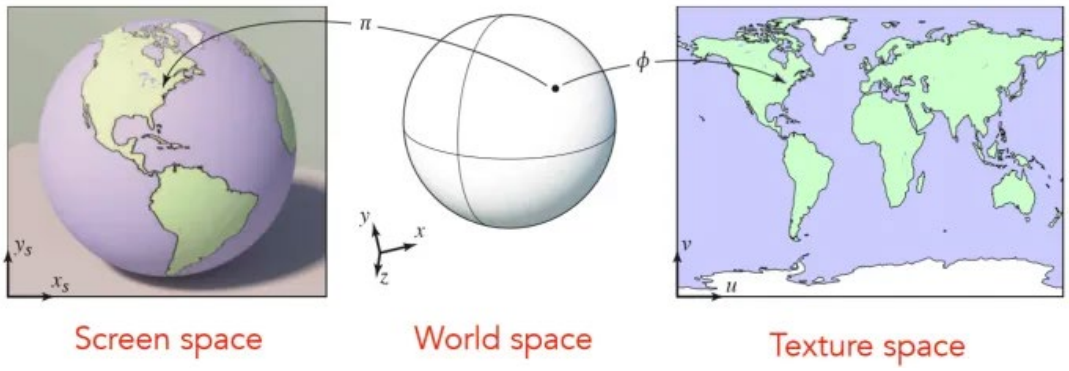


图 2-11 屏幕空间，世界空间与纹理空间的对应

显然这里有一个亟需解决的问题，由于世界空间与纹理都只有三角形的顶点坐标，确定了某个点在三角形中的相对位置，才能确定它在世界空间以及纹理中的位置，那么如何确定某个点在三角形中的位置？这就需要使用到三角形的重心坐标来插值了。假如三角形三个顶点 P，Q，R，待表示的点 M，重心坐标$A = [a, b, c]^T$,则重心坐标满足如下方程。

$$\begin{bmatrix} x_P & x_Q & x_R \\ y_P & y_Q & y_R \\ z_P & z_Q & z_R \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} x_M \\ y_M \\ z_M \end{bmatrix} \tag{2-21}$$

获得重心坐标 A 之后，根据三角形对应顶点的纹理坐标，就可以根据 A 得到它的纹理了（但

这里得到的并不是真正的纹理，之后会进行矫正）。

$$\begin{bmatrix} u_P & u_Q & u_R \\ v_P & v_Q & v_R \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} u_M \\ v_M \end{bmatrix} \tag{2-22}$$

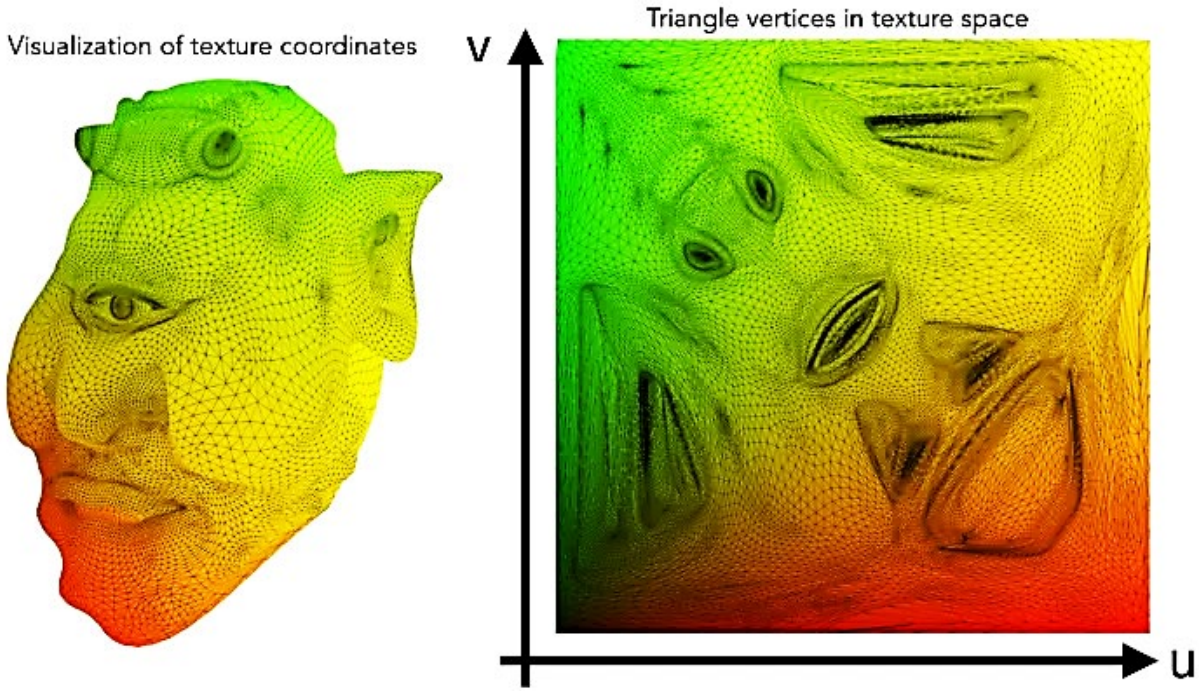


图 2-12 世界空间和屏幕空间中的纹理

2.3.2 透视矫正

上一节通过重心坐标插值来将纹理映射到了三角形上，现在试试将一个黄蓝相间的格子图案映射到一个地板上，结果如下。

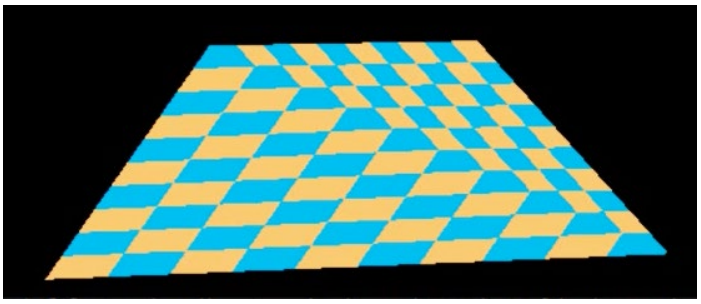


图 2-13 不正确的透视

可以很明显的看到纹理映射后的结果并不符合预期，格子图案被撕裂成了两个部分。回顾之前得到纹理的整个过程，就会发现一个致命的问题：渲染器是在透视变换之后的屏幕空间中获取的重心坐标，而这与世界空间中的重心坐标不一致！因此需要先还原出世界空间中点的重心坐标，才能获得正确的纹理。[31]

现在试着寻找一下屏幕空间与世界空间点的对应关系，根据之前所述的透视投影变换，不难发现如下。

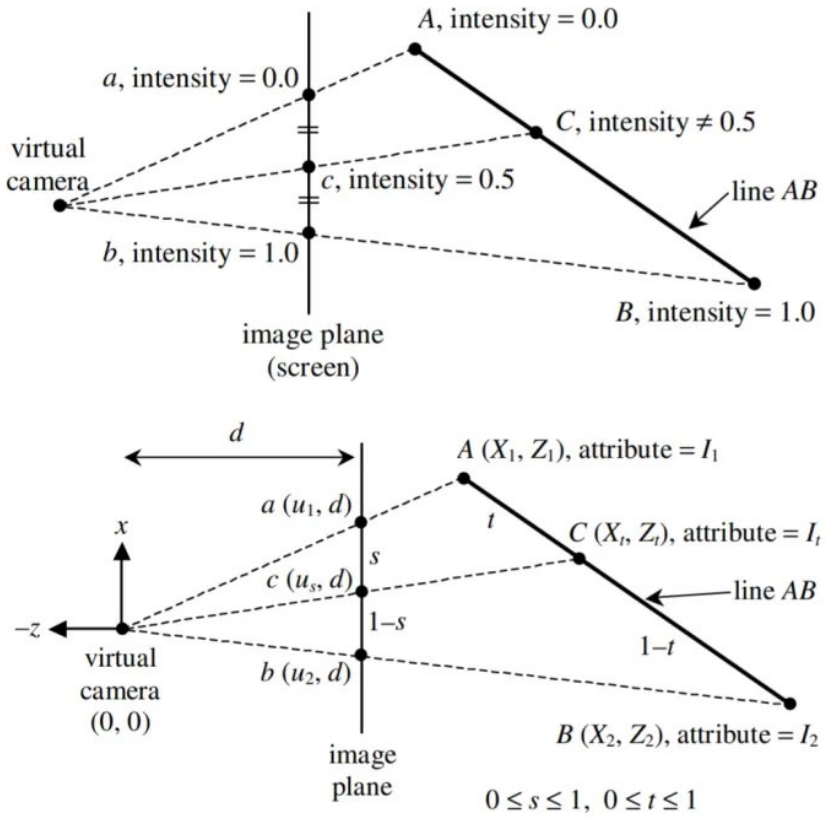


图 2-14 透视投影变换的平面分析

如图，从垂直于 x 轴方向看，世界空间中的 C 点重心坐标并不是 0.5，而它经过透视变换后在屏幕上的重心坐标却变成了 0.5，这也就是误差的来源。可以根据透视关系得到如下等式：

$$\frac{X_1}{Z_1}=\frac{u_1}{d} \tag{2-23}$$

$$\frac{X_2}{Z_2}=\frac{u_2}{d} \tag{2-24}$$

$$\frac{X_t}{Z_t}=\frac{u_s}{d} \tag{2-25}$$

根据屏幕空间以及世界空间的线性插值可以得到：

$$u_s=u_1+s(u_2-u_1) \tag{2-26}$$

$$X_t=X_1+t(X_2-X_1) \tag{2-27}$$

$$Z_t=Z_1+t(Z_2-Z_1) \tag{2-28}$$

其中X_1,Z_1,X_2,Z_2,d,s为已知量，Z_1,Z_2为世界空间下的坐标（注意不是屏幕空间的 Z!）。

这里以 Z 轴坐标为例，需要得到Z_t与Z_1,Z_2的表达式,并且使用已知量 s 来表示未知量 t，将(2-26)(2-25)带入(2-24)得：

$$Z_t=\frac{d\big(X_1+t(X_2-X_1)\big)}{u_1+s(u_2-u_1)} \tag{2-29}$$

再将(2-22)(2-23)带入(2-28)：

$$Z_t=\frac{d\left(\frac{u_1Z_1}{d}+t\left(\frac{u_2Z_2}{d}-\frac{u_1Z_1}{d}\right)\right)}{u_1+s(u_2-u_1)}=\frac{u_1Z_1+t(u_2Z_2-u_1Z_1)}{u_1+s(u_2-u_1)} \tag{2-30}$$

将(2-27)带入(2-29)等式左侧,得到了 s 与 t 的关系：

$$Z_1 + t(Z_2 - Z_1) = \frac{u_1 Z_1 + t(u_2 Z_2 - u_1 Z_1)}{u_1 + s(u_2 - u_1)} \Rightarrow t = \frac{sZ_1}{sZ_1 + (1-s)Z_2} \tag{2-31}$$

这样就可以得到想要的Z_t与Z_1, Z_2的表达式了：

$$Z_t = \frac{1}{\frac{s}{Z_2} + \frac{1-s}{Z_1}} \tag{2-32}$$

以上是平行于 yoz 平面的二维结果，可以把它推广到三维，此时有屏幕重心坐标(α, β, γ)：

$$Z_t = \frac{1}{\frac{\alpha}{Z_A} + \frac{\beta}{Z_B} + \frac{\gamma}{Z_C}} \tag{2-33}$$

得到了Z_t的真实坐标之后可以利用Z_t与Z_A, Z_B, Z_C的关系来其他属性的透视矫正：

$$I_t = \alpha \frac{Z_t}{Z_A} I_A + \beta \frac{Z_t}{Z_B} I_B + \gamma \frac{Z_t}{Z_C} I_C \tag{2-34}$$

由此可见通过 Z 轴坐标矫正后的重心坐标为$\left(\alpha \frac{Z_t}{Z_A}, \beta \frac{Z_t}{Z_B}, \gamma \frac{Z_t}{Z_C}\right)$。利用这个重心坐标来矫正纹理，得到的结果就正常了：

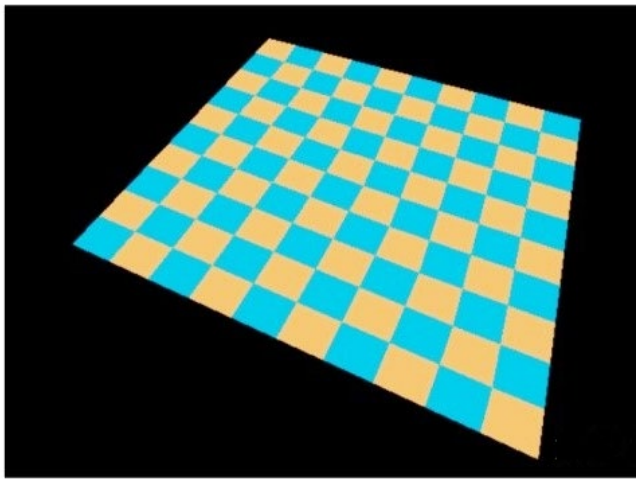


图 2-15 透视校正后的纹理

2.4 表面着色

当对物体进行纹理映射以后，终于不用只是根据物体的大致形状来分辨是什么物体了，而是可以很清晰的一眼看过去就能够分辨出物体的类型。譬如绿色的草地，蓝色的大海……

等等，草地就一定是绿色的吗？大海就一定是蓝色的吗？当太阳西下，黄昏来临的时候，绿色的草地变成了夕阳红，而当夜晚来临时，大海也由白天的湛蓝变成了伸手不见五指的黑……而这一切都是因为光的存在，没有光的世界将毫无生气。而图形学中的光照也相当重要，如果一个物体呈现在人们眼前时没有光线的明暗变化，那么它看起来会显得非常失真，因此这一节主要研究如何给世界空间中加上光照，并在物体表面形成反射。

在图形学中，通过光照反射模型来模拟光照，光照模型可以分为局部光照模型和全局光照

模型。局部光照模型只计算一次光线反射，而全局光照模型可以模拟多次光线反射，但是计算复杂度更高[33]。本文重点介绍局部光照模型，虽然并不完全准确，但具有计算速度快、效果可接受等优点，因此在各种游戏中广泛应用至今。

2.4.1 点光源与反射光

要在图形世界中添加光照，首先需要添加光照源。光照源一般有如下几种类型：

(1) 像电灯泡一样，由一个点出发并沿四面八方均匀辐射光线的点光源。

(2) 像太阳一样，由于离得足够远导致光线几乎平行的平行光源或定向光源。

(3) 像聚光灯一样将光线定向集中的聚光灯光源。

这里主要介绍最通用的点光源[34]。点光源由一个点出发并沿四面八方均匀辐射光线，世界中的点光源一般有如下属性：

(1) 坐标（位置）。

(2) 辐射通量（功率），表示点光源在单位时间内向外辐射的能量大小。

(3) 颜色（RGB 三通道）。

和其他光源一样，当来自点光源的光线打在一个反射点上并形成反射时，由反射点的纹理决定它将以怎样的比例分别吸收红、绿、蓝三种光线。当接收到的光线为纯白时，反射点的反射光颜色与纹理色相同，否则将根据光线颜色来进行反射。可以根据反射点的纹理计算出其对红、绿、蓝三种光线的吸收率，然后再将光线颜色分别乘以吸收率，即可得到正确的反射光线。

2.4.2 Blinn-Phong 反射模型

局部光照模型中最有名的当属 Blinn-Phong 反射模型了，是由 Jim Blinn 于 1977 年在文章中对传统 Phong 光照模型基础上进行修改提出的，它将进入摄像机的光线分为三个部分，每个部分使用一种方法来计算它的贡献度，这三个部分分别是环境光(Ambient)、漫反射(Diffuse)和高光反射(Specular)。[17]

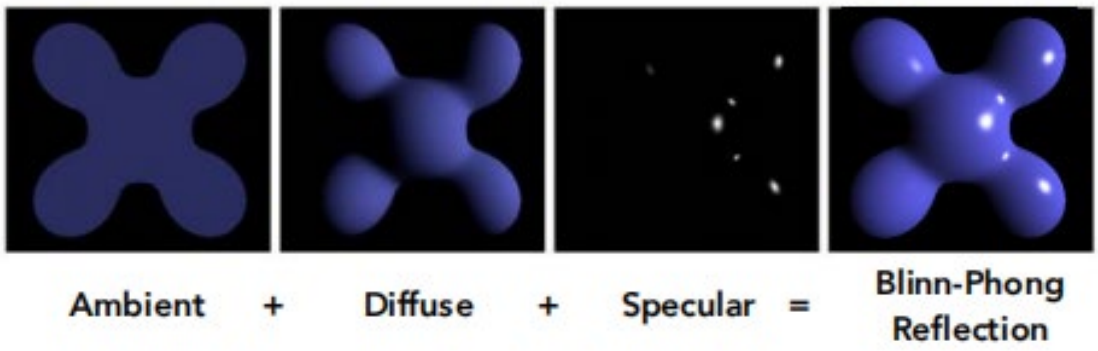


图 2-16 Blinn-Phong 反射模型

下面将分别讨论并计算这三种反射。

2.4.2.1 环境光

环境光也称间接光，是光线经过周围环境表面多次反射后形成的，利用它可以描述一块区域的亮度，在光照模型中，通常用一个常量来表示。只考虑环境光的反射模型被称为泛光模型，其表达式如下：

$$I_a = K_a I \tag{2-35}$$

其中K_a代表物体表面对环境光的反射率，I代表入射环境光的亮度，I_a代表人眼所能看到从物体表面反射的环境光的亮度。效果如下：

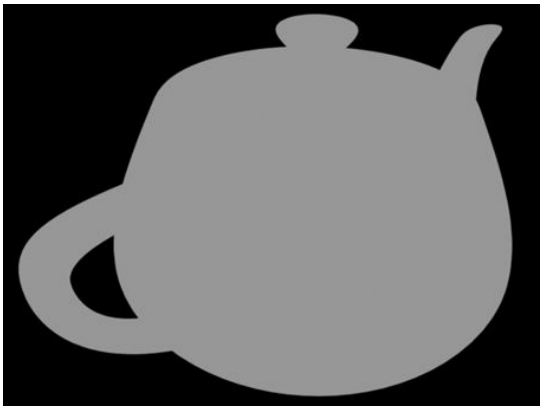


图 2-17 泛光模型

2.4.2.2 漫反射

泛光模型只能呈现物体的平面形状，无法表现其体积感。为了增加体积感，需要使用漫反射模型，如 Lambert 漫反射模型。该模型在泛光模型的基础上增加了漫反射项。漫反射是指光从一定角度入射到物体表面后，向各个方向均匀地反射，且每个方向反射光的强度相等。物体表面的粗糙程度是产生漫反射的原因，这种物理现象的发生为物体增加了体积感。[31]

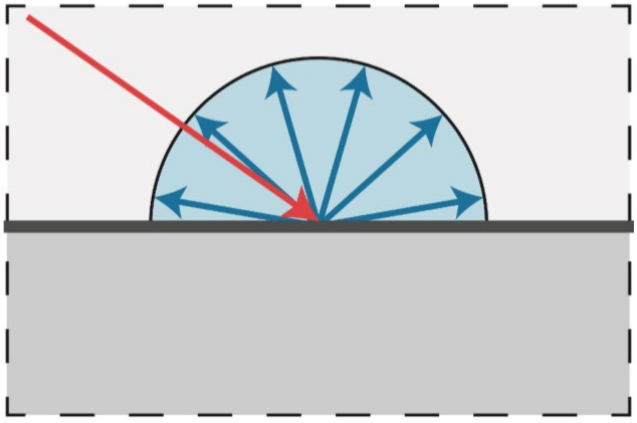


图 2-18 漫反射现象

漫反射得到的反射光强主要受到两个方面的影响，一个是反射点离光源的距离，反射点离光源越近反射光强就越强；另一个则是反射点处光线的入射角度，入射角度越正则反射光强越强。

首先分析反射点离光源的距离与反射光强的关系。

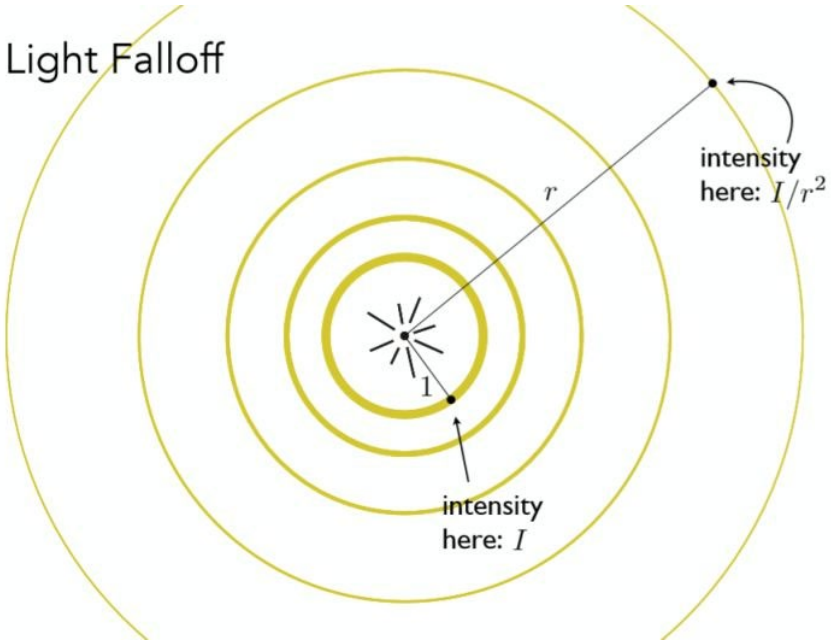


图 2-19 反射点离光源的距离与反射光强的关系

如图所示，由简单的辐射度量学可以知道，反射点离光源的距离 r 与反射光强I_d有如下关系：

$$I_d \propto \frac{1}{r^2} \tag{2-36}$$

分析反射点处光线的入射角度与反射光强的关系：

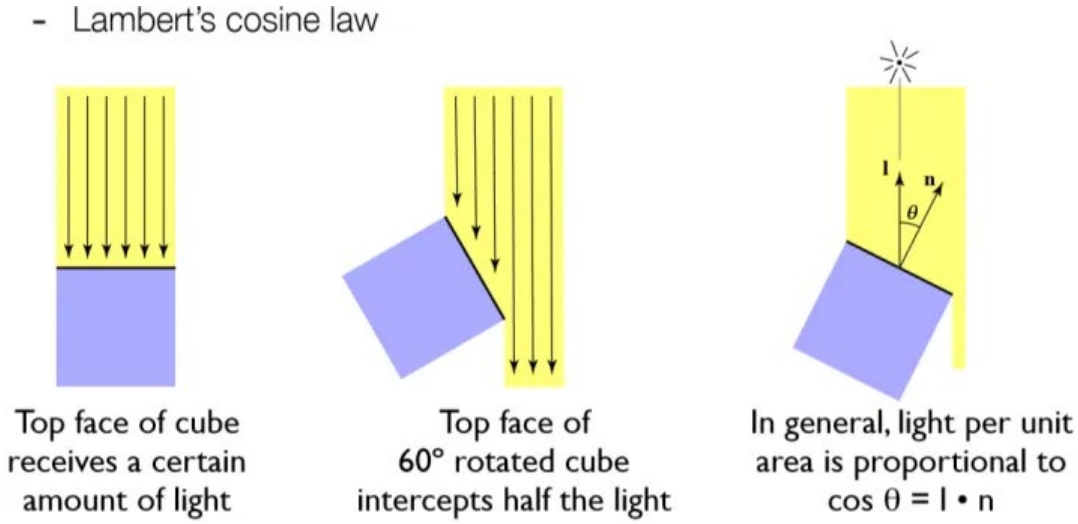


图 2-20 Lambert 余弦法则

根据 Lambert 余弦法则，反射光强I_d与反射点处光线的入射角度的余弦$cos\theta$成正比，即与光线入射方向的反方向与反射点法线的点积$l \cdot n$成正比：

$$I_d \propto cos\theta \propto l \cdot n \tag{2-37}$$

综上，可以得到反射光强的大致表达式：

$$I_d = K_d \left(\frac{I}{r^2}\right) max(0, l \cdot n) \tag{2-38}$$

其中K_d为漫反射系数，I为入射光强，n, l分别如图中所示为法线向量和入射方向，max 是为了

剔除夹角大于 90°的光。将环境光与漫反射一起考虑之后，得到的就是 Lambert 模型：

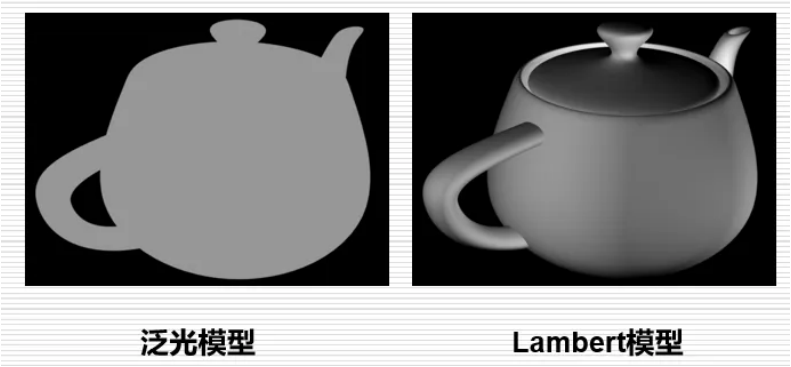


图 2-21 泛光模型与 Lambert 模型

因为漫反射的存在，茶壶的体积感便很容易看出来了。

2.4.2.3 高光反射

高光反射也就是镜面反射，只有在入射光反射出的方向才能看到高光。

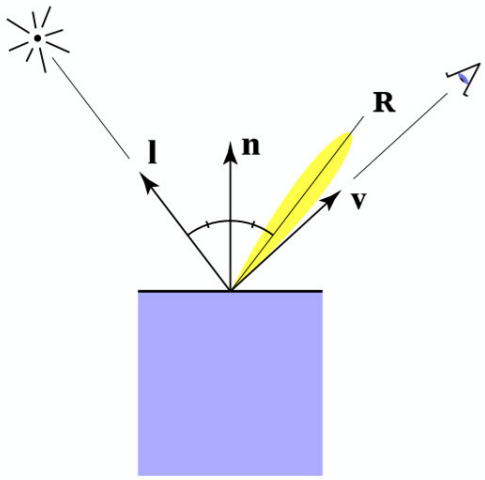


图 2-22 镜面反射

这里只需要在漫反射的基础上再考虑入射光的反射方向 R 与目视方向 v 的夹角 α，并且不需要再考虑光的入射角度了（因为不管光的入射角度怎么变，其反射方向上的光强不会再变），因此有如下表达式。

$$I_s = K_s\left(\frac{I}{r^2}\right)max(0, cos\alpha)^p \tag{2-39}$$

其中K_s为镜面反射系数，I为入射光强，r为光源到入射点距离，注意这里在 max 剔除大于 90°的光之后，还乘了一个指数 p，添加该项的原因很直接，因为离反射光越远就越不应该看见反射光，需要一个指数 p 加速衰减。这就是 Phong 模型，而 Blinn-Phong 模型则在 Phong 模型的基础上将 cos α 改为了入射光方向 l 与目视方向 v 的半程向量 h 与反射点法线的夹角。[17, 31]

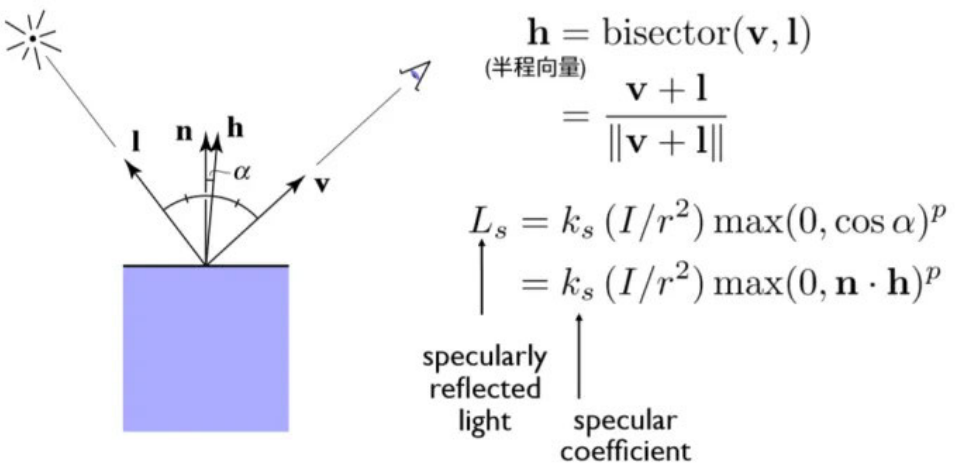


图 2-23 Blinn-Phong 模型的镜面反射

这样做的好处是大大加快了计算速度，同时该夹角与之前夹角的大小关系比始终是 1：2。最后将三个部分的反射光线性相加，就可以得到最终的反射光了。

$$I_R = I_a + I_d + I_s \tag{2-40}$$

三种反射光相加后效果如下：

图 2-24 泛光模型，Lambert 模型与 Phong 模型

2.4.3 阴影映射

借由 Blinn-Phong 反射模型可以简单地在世界中加入光源后对模型上色。如果世界中只有一个模型，那么效果是不错的，但当有两个及以上的模型并且他们相对于官员出现了遮挡关系时，就会存在一个很严重的失真：模型居然没有影子！本来一个模型可以通过判断反射点的光线入射角来判断它是否受到光线的直接光照，但当两个模型出现遮挡关系时，后面被遮挡的模型就无法判断其是否受到前面的遮挡了，自然影子也不会出现。为了解决这个问题，必须想办法对物体是否受到光源直接照射进行判断。于是便有了阴影映射技术（Shadow Mapping）。

简单而言，阴影映射就是对所有的反射点作这样一个判断：

(1) 如果该点（A）能够同时被相机和点光源“看到”，那么这个点就没有阴影。

(2) 如果该点（C）不能摄像机看到，那么这个点显然不可见。

(3) 如果这个点（B）只能被相机看到而不能被点光源看到，那么这个点有阴影。

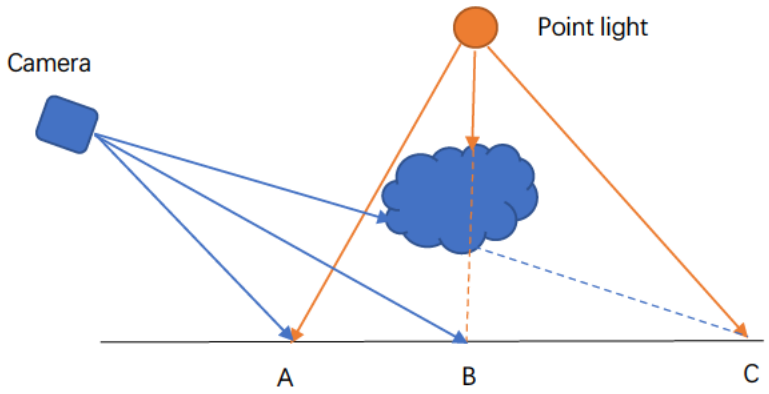


图 2-25 对阴影区域的判断

这样从每一个像素出发，可以到达相机视野中所有的反射点，然后只需逐点判断点光源到该点的连线是否有其他物体穿过即可。世界中的模型由三角形组成，这样可以把问题化为空间中的三角形与直线的相交问题。可以先将直线用点向式表示出来：

$$l:\frac{x-x_0}{v_x}=\frac{y-y_0}{v_y}=\frac{z-z_0}{v_z} \tag{2-41}$$

其中直线的方向向量 v 可以利用两点相减得到。

当该直线与三角形所在平面法向量垂直时，该直线除非位于平面上，否则与平面平行，没有交点。此时默认这种情况不存在（对整体影响几乎没有），之后假定直线与三角形存在一个交点，此时可以联立直线方程与三角形所在的面方程：

$$\begin{cases}\dfrac{x-x_0}{v_x}=\dfrac{y-y_0}{v_y}=\dfrac{z-z_0}{v_z}\\ ax+by+cz-1=0\end{cases} \tag{2-42}$$

$$\Rightarrow\begin{cases}v_yv_zx-v_xv_zy-x_0v_yv_z+y_0v_xv_z=0\\ v_yv_zx-v_xv_yz-x_0v_yv_z+z_0v_xv_y=0\\ ax+by+cz-1=0\end{cases} \tag{2-43}$$

$$\Rightarrow\begin{bmatrix}v_yv_z & -v_xv_z & 0\\ v_yv_z & 0 & -v_xv_y\\ a & b & c\end{bmatrix}\begin{bmatrix}x\\y\\z\end{bmatrix}=\begin{bmatrix}x_0v_yv_z-y_0v_xv_z\\ x_0v_yv_z-z_0v_xv_y\\ 1\end{bmatrix} \tag{2-44}$$

可解出相交点$\begin{bmatrix}x\\y\\z\end{bmatrix}$。

之后在三角形所在平面上判断相交点是否位于三角形内，具体方法参考 2.2.1 判断像素与三角形面的位置关系。最终效果如下：

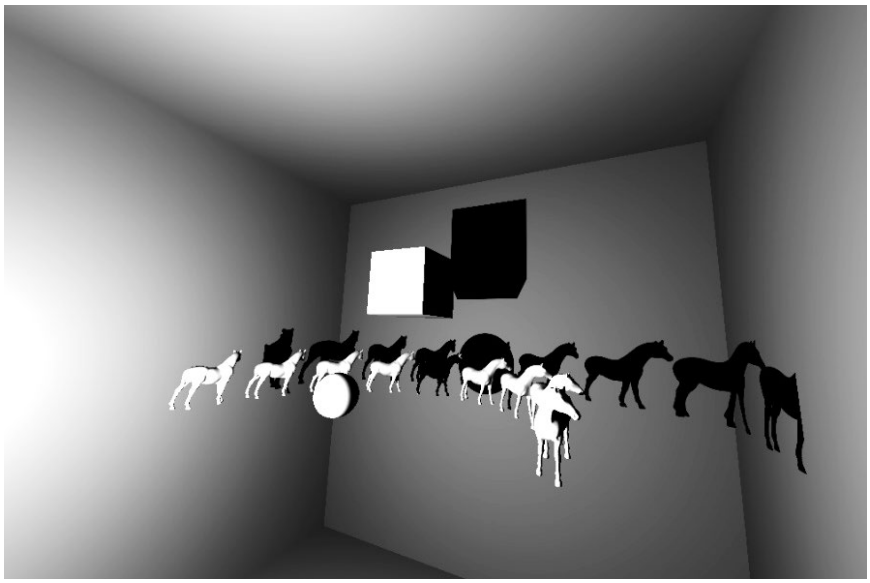


图 2-26 阴影映射效果

2.5 反走样技术

本章主要介绍几种图形学中常见的反走样技术，并将使用它们来尝试解决常见的走样问题。

2.5.1 超采样

所谓超采样（Super sampling），有点类似于数学中极限的思想：使用更多的采样点来使得结果更加接近真实值。由于屏幕上的像素点是离散、有限的，它无法表达尺度比它精细的多的纹理或模型，因此会出现锯齿和摩尔纹失真现象，下面将通过超采样来尝试解决它。

2.5.1.1 利用超采样抗锯齿

当渲染器在屏幕上光栅化一个尺寸仅为几像素的三角形时，得到的结果可能是像下面这样。

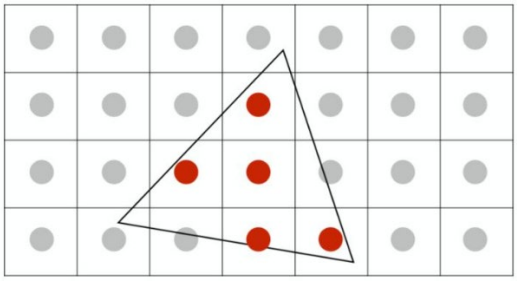


图 2-27 三角形光栅化（无抗锯齿）

完全看不出来这是一个三角形，而这正是因为屏幕像素表达能力不够的问题。这张图中只判断了像素的中点是否位于三角形内，并且仅仅只是区分了两种情况：像素中点在三角形内或像素中点不在三角形内，而没有考虑到那些“部分”在三角形内的像素。这时可以将每个像素分成四个部分（或更多），并判断每一个部分是否位于三角形内，将每个像素点内部所细分的采样点的颜色值全部加起来再求均值，作为该像素点的抗走样之后的颜色值，这样做的结果如下。

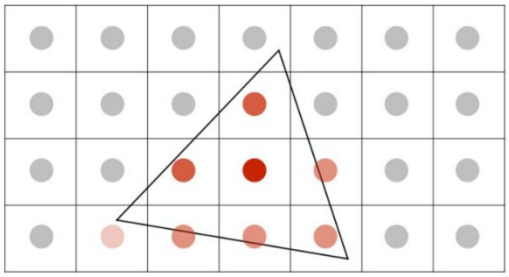


图 2-28 三角形光栅化（超采样颜色值）

仔细观察可以发现因为将 4 个采样点的颜色求均值的之后，靠近三角形边缘的像素点有的变淡了，从宏观角度来看的话，这个锯齿就会变得不那么明显了。

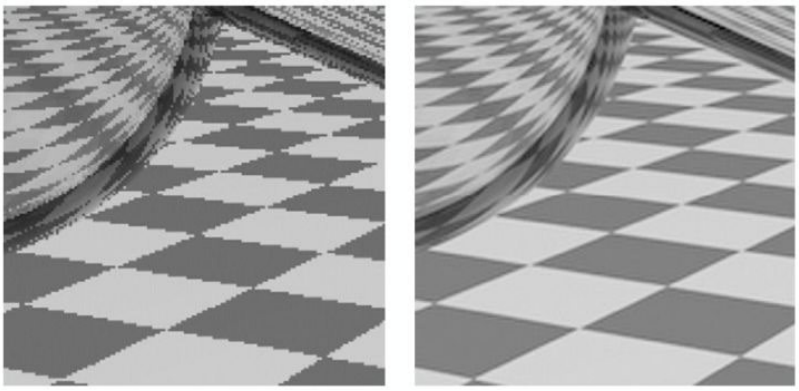


图 2-29 超采样抗锯齿效果

2.5.1.2 利用超采样处理纹理映射中的摩尔纹

想象一张很大的地板，在上面铺满了重复的方格贴图，渲染出来的结果会在近处出现锯齿，在远处出现摩尔纹。

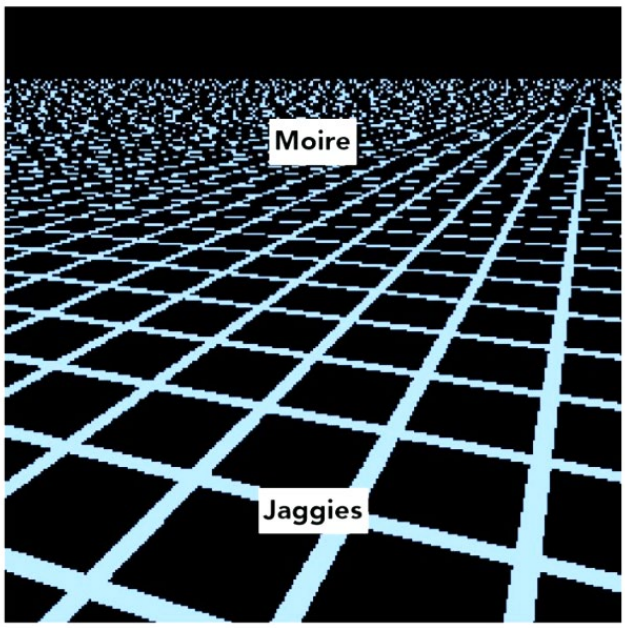


图 2-30 摩尔纹与锯齿

可以来将屏幕空间和纹理空间的像素放在一起对比，这是一张屏幕像素在纹理空间里的footprint。

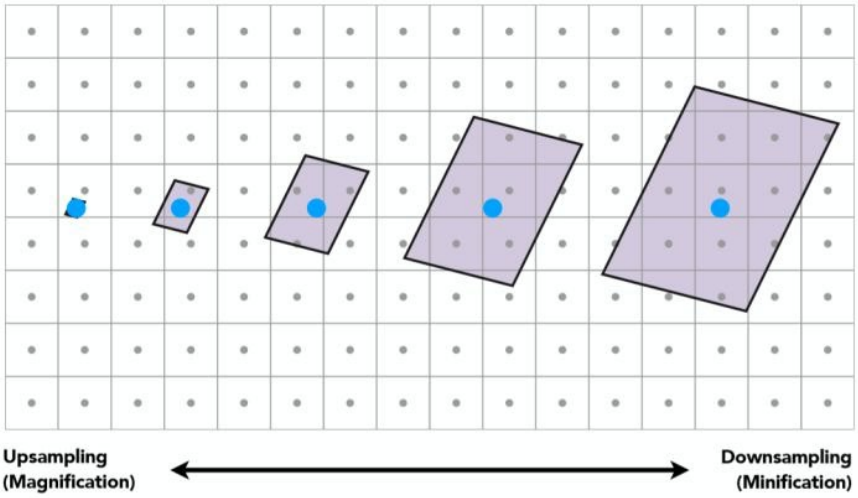


图 2-31 过采样与欠采样

容易发现，当纹理离屏幕过近，就会出现纹理不足以反映屏幕的采样的问题，也就是过采样；反之，当纹理离屏幕过远，屏幕的采样不足以反映纹理，也就会出现欠采样的问题，换而言之，过采样和欠采样分别是近处出现锯齿，远处出现摩尔纹的根本原因。对于纹理的欠采样，一种直观的解决方法就是超采样，如果一个像素点不足以代表一个区域的颜色信息，那么便把一个像素细分为更多个小的采样点就可以解决这个问题了。可以看看如下图 512×512 超采样的结果。

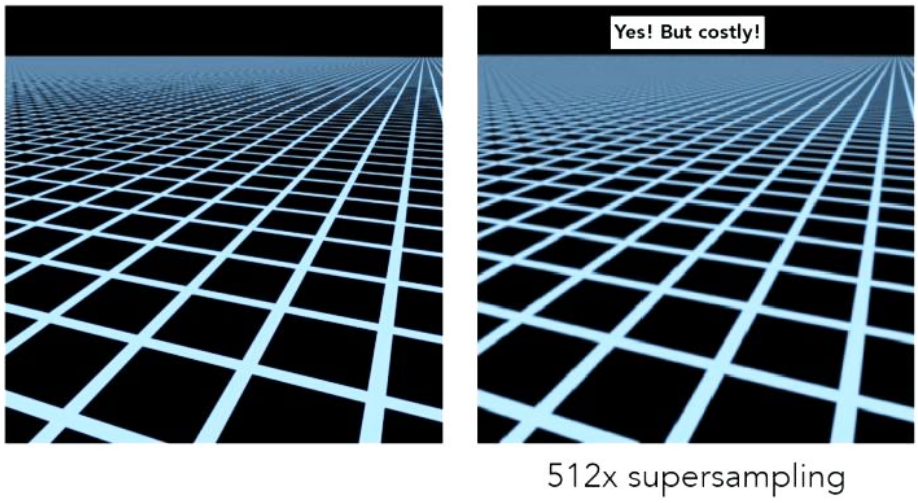


图 2-32 超采样消除摩尔纹

2.5.2 线性插值

线性插值则可以用来解决纹理过小，以及上述纹理离屏幕过近导致的过采样问题，这里选取双线性插值来解决这个问题。

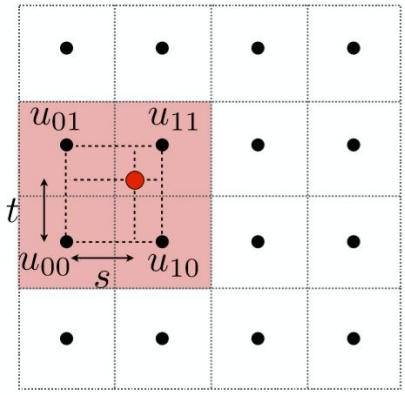


图 2-33 对采样点的双线性插值

如图所示，当屏幕坐标采样到红点时，如果直接将u_{11}所在的纹理直接返回，那么照样会出现之前说过的锯齿问题。现在以先水平后垂直插值为例（反过来也可以）,将$u_{00},u_{10},u_{01},u_{11}$按照红点的 u 坐标对 RGB 三通道插值，可以得到：

$$\begin{cases} u_0 = su_{10} + (1-s)u_{00} \\ u_1 = su_{11} + (1-s)u_{01} \end{cases} \tag{2-45}$$

之后再根据红点的 v 坐标插值：

$$u = tu_1 + (1-t)u_0 \tag{2-46}$$

就可以得到差值后的纹理颜色 u。将其运用到有锯齿的图像上：

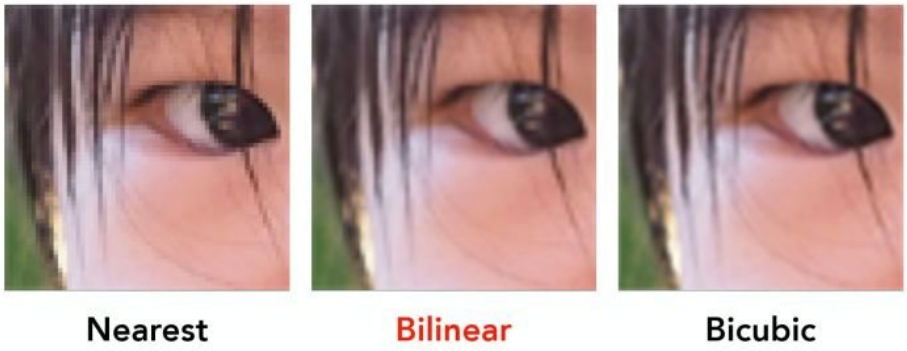


图 2-34 双线性插值抗锯齿效果

中间就是双线性插值后的结果，效果不错。

2.5.3 Mipmap 贴图技术

在 2.5.1 中利用 512×512 超采样来去除了纹理上的摩尔纹，但这样一来就导致采样次数也变成了之前的512^2次，这样计算量太大，并且随着屏幕空间的点离相机距离更远，更多的 texels(纹理空间的像素)会在屏幕像素的一个 footprint 里面，会需要更高的超采样频率，因此并不是一个很好的解决方案。那么如果不去超采样，仅仅是求出每个屏幕像素所对应的 footprint 里所有 texels 的颜色均值呢？这也就是接下来所要介绍的 Mipmap 技术了。

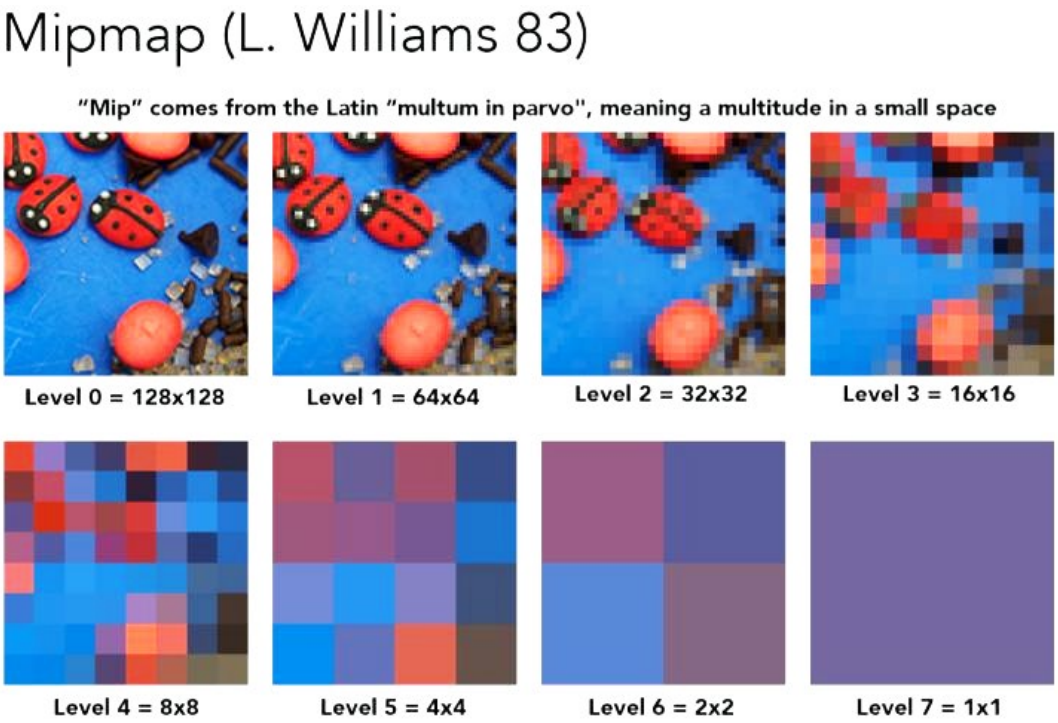
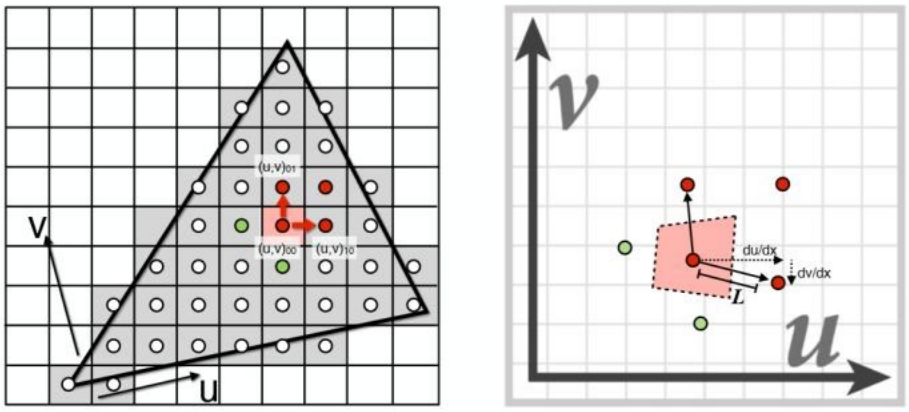


图 2-35 不同 Level 的 Mipmap

level 0 表示原始纹理，也是最精确的纹理。随着 level 的提升，每一级将 4 个相邻像素点的平均值合并为一个像素点，因此越高的 level 代表的区域查询越大。接下来，需要根据屏幕像素的区域大小选择不同 level 的纹理，再进行点查询。这实际上相当于在原始纹理上进行了区域查询。可以利用屏幕像素的相邻像素点估算 footprint 大小再确定 level D。



$$D = \log_2 L \qquad L = \max\left(\sqrt{\left(\frac{du}{dx}\right)^2 + \left(\frac{dv}{dx}\right)^2}, \sqrt{\left(\frac{du}{dy}\right)^2 + \left(\frac{dv}{dy}\right)^2}\right)$$

图 2-36 利用 footprint 确定 level D

通过在屏幕空间中选取当前像素点的右侧和上方的两个相邻像素点（也可以全取四个像素点），分别查询这三个点在纹理空间中的坐标，计算出当前像素点与右方像素点以及上方像素点在纹理空间中的距离，取两者之间的最大值。计算公式如图所示，因此 level D 是该距离的 log2 值。[31]

但是这里 D 值算出来是一个连续值，并不是一个整数，有两种对应的方法：

(1) 四舍五入取得最近的那个 level D。

(2) 利用 D 值在 向下和向上取整的两个不同 level 作三线性插值。

第一个方法很容易理解，具体讲述一下第二个方法，如图。

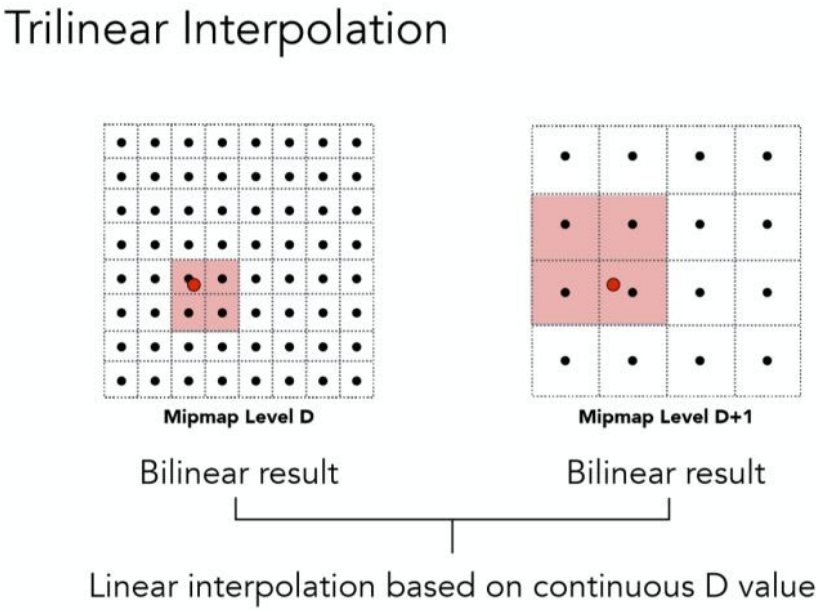


图 2-37 利用 level D 作三线性插值

三线性插值指的是在向下取整的 D level 上进行一次双线性插值，再在 D+1 level 上进行一次双线性插值，然后根据实际连续 D 值在向下和向上取整的两个不同 level 之间的比例进行一次线性插值，将两次插值的结果进行插值。这样就得到了一个完整的三线性插值。

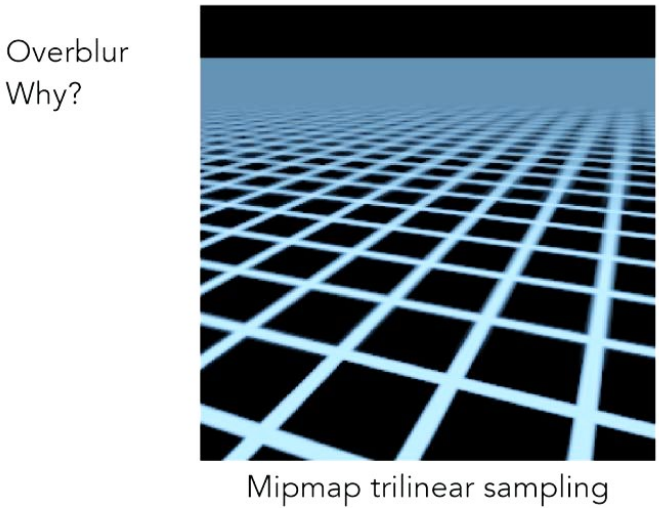


图 2-38 Mipmap 三线性插值效果

尽管与最初的点采样相比有了很大的进步，但是存在一个严重的问题，就是在远处的地面会产生过曝现象，使得地面完全模糊在一起。这种现象的原因是，所采用的不同 level 的 Mipmap 默认都采用正方形区域的范围查询，然而实际情况并非如此。

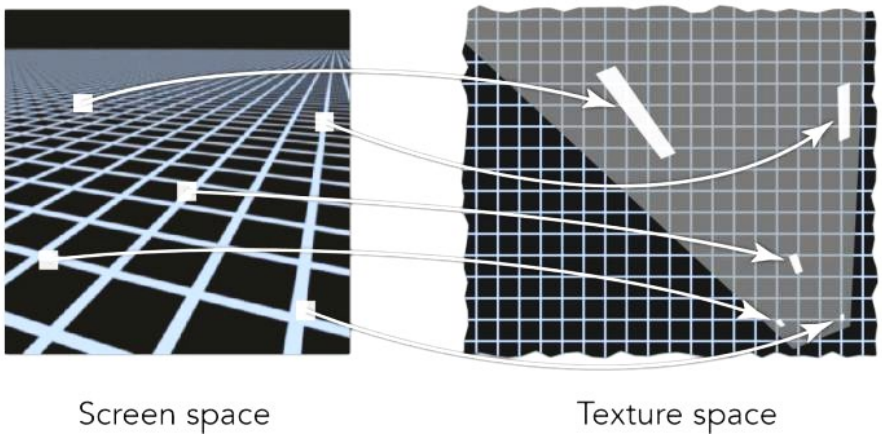


图 2-39 不同类型的范围查询

可以发现，不同屏幕空间的像素点对应的 footprint 是不同的，有长方形，甚至有不规则图形。因此，针对这种情况，有些情况只需要水平方向上的高 level，而有些情况则只需要竖直方向上的高 level。为了解决这个问题，可以使用经过各向异性滤波的 Mipmap[35]。

图 2-40 各向异性 Mipmap

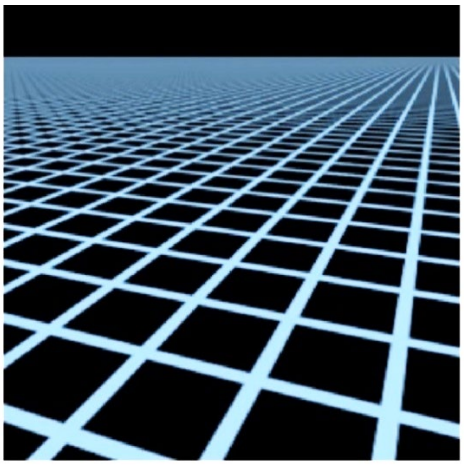


图 2-41 启用各向异性 Mipmap 后的效果

通过使用不同的纹理图像，可以更精细地选择后处理结果，这将显著提升图像的质量。

第 3 章 3D 图形软光栅化渲染器设计与实现

3.1 线性代数框架

使用 C++实现线性代数，最重要的就是设计好矩阵和向量对象的数据结构，以及实现基本的运算方法。为了实现的方便，这里把所有的值都存进了一维数组中，并使用一个无符号整形数来表示它的维度。

```
class Vector
{
    private:
    // The pointer indicated to vector.
    VectorElemType _V[Vector$::MAX_SUPPORTED_DIMENSION];
    // The length of the vector.
    std::size_t _N = Vector$::NOT_INITIALIZED_N;
};
class SMatrix
{
    private:
    // The pointer indicated to square matrix.
    SMatrixElemType _SM[SMatrix$::MAX_SUPPORTED_DIMENSION * SMa-
trix$::MAX_SUPPORTED_DIMENSION];
    // The length of the square matrix.
    std::size_t _N = SMatrix$::NOT_INITIALIZED_N;
};
```

图 3-1 矩阵与向量的数据结构的 C++设计

此外还需要实现的成员函数如下表所示：

表 3-1 矩阵与向量成员函数设计

函数原型	函数作用
Vector(std::initializer_list<VectorElemType> list);	向量的初始化
std::size_t N();	获取向量的维度
VectorElemType operator[](size_t n); VectorCode Set(size_t index, VectorElemType value);	根据索引获取/设置向量中每一个元素的值
VectorCode Unitization();	向量的单位化
SMatrix(std::initializer_list<SMatrixElemType> list);	矩阵的初始化
std::size_t N();	获取矩阵的维度
SMatrixElemType Get(size_t row, size_t col); SMatrixCode Set(size_t row, size_t col, SMatrixElemType value);	根据索引获取/设置矩阵中每一个元素的值

续表

SMatrixCode operator*=(SMatrix const &sm);	矩阵与矩阵的乘法
SMatrixCode operator*(Vector &v);	矩阵与向量的乘法
SMatrix operator+();	求转置矩阵
SMatrix operator-();	求逆矩阵
SMatrix operator*();	求伴随矩阵
SMatrixElemType Determinant();	求行列式
SMatrixElemType AComplement(size_t row, size_t col);	求代数余子式

3.2 模型文件处理

3.2.1 使用 obj 文件作为模型文件

渲染器通常需要若干个模型文件作为输入，每一个模型文件代表一个 3D 模型，存储了该模型的一些基本属性，例如几何顶点，顶点纹理，顶点法线，由顶点构成的面等，本渲染器选择了 obj 文件作为输入的模型文件。

Obj 文件格式是一种简单的文本文件格式，用于单独表示 3D 几何图形元素。该文件包含顶点坐标、每个顶点对应的纹理 UV 坐标、顶点法向量以及构成多边形面的顶点坐标和纹理 UV 坐标序列。默认情况下，多边形面的顶点按逆时针顺序排列，且法向量不是必需的。obj 文件不是归一化的，但可以在注释中加入缩放信息[36]。

Obj 文件主要由以下几个部分组成：

(1) v(vertices)：几何形状的顶点，是构成物体的基本元素，每个顶点由三个坐标值表示（x、y、z）。有些应用程序支持顶点颜色，可以在顶点坐标后面加上红、绿、蓝三个值来表示颜色，取值范围在 0 到 1 之间。

(2) vt(vertex texture)：顶点纹理坐标，表示当前顶点对应纹理图像中的哪个像素。通常取值范围为 0 到 1，如果超过 1，就相当于将纹理图像重新扩展并取值。可以通过镜像填充、翻转填充等方式进行纹理坐标的扩展，然后根据纹理图像的宽高来计算具体的像素位置。

(3) vn(vertex normal)：顶点法线，是入射光线与物体表面垂直的向量，用于计算光照效果。在渲染时，可以利用法线来计算反射光线的方向，从而实现真实感的渲染效果。

(4) f(face)：大部分几何体都由面构成，常见的面是由三个顶点构成的三角形面。在 Obj 文件中，每个面由其对应的三个顶点的索引值组成，通常是按照逆时针方向给出顶点索引，这样可以确保面的正面朝向观察者。

3.2.2 使用 tga 文件作为纹理

除了加载 obj 文件作为模型文件之外，还需要载入一种图像文件来作为纹理。所谓纹理就是模型的外表材质与颜色，这里选择 tga 文件来作为模型的纹理。

TGA（Targa）文件格式使用.tga 作为文件扩展名，它支持压缩和不失真的压缩算法，可以包

含通道图，并支持行程编码压缩。该格式最初是由美国 Truevision 公司为其显示卡开发的，已被国际图形图像工业广泛接受，成为数字化图像和运用光线跟踪算法产生高质量图像的常见格式。相对于 BMP 格式具有更好的图像质量，相对于 JPEG 格式则具有更小的文件体积。此外，TGA 格式还具有通道效果和方向性等特点，在 CG 领域中常作为影视动画序列输出格式，因为同时具有体积小和清晰效果的特点。

3.2.3 将模型文件与纹理文件加载到内存中

Obj 文件作为文本格式文件，其内容可以通过 std::ifstream 读取，这里以读取 floor.obj 文件为例：

```
v -1 -1 -1
v  1 -1 -1
v  1 -1  1
v -1 -1  1

vt 0 0
vt 1 0
vt 1 1
vt 0 1

vn 0 1 0

f 3/3/1 2/2/1 1/1/1
f 4/4/1 3/3/1 1/1/1
```

图 3-2 floor.obj

(1) 打开文件读取流 std::ifstream，并判断读取文件的合法性。

(2) 逐行读取文件，判断行开头的字符：

1) 如果是 vt，则判断该行为顶点纹理，将其添加到顶点纹理的动态数组（vertex_textures）中。

2) 如果是 vn，则判断该行为顶点法线，将其添加到顶点法线的动态数组（vertex_normals）中。

3) 如果是 v，则判断该行为顶点，将其添加到顶点的动态数组（vertices）中。

4) 如果是 f，则判断该行为面，如果该面由 3 个以上的顶点组成，则将其拆为多个由三角形组成的面，然后添加到面的动态数组（faces）中。

(3) 关闭文件流。

对于二进制格式的 tga 文件，其读取方式比较复杂，它的结构由一个控制头部加 rgb 数据体组成，头部规定了该 tga 文件的所有结构如长度，宽度，颜色表等，因此首先要从文件流中将控制头读取到一个结构体中。

```
// This pragma preprocessive instruct is used to align memory
#pragma pack(push, 1)
struct TGAHeader
{
    // 图像信息长度、颜色表类型、图像类型码
    std::uint8_t id_length{};       // 指出图像信息字段长度，取值范围0~255
    std::uint8_t color_map_type{}; // 0：不使用颜色表 1：使用颜色表
    std::uint8_t data_type_code{}; // 0：没有图像数据 1：未压缩的颜色表图像 2：
未压缩的真彩图像 3：未压缩的黑白图像 9：RLE压缩的颜色表图像 10：RLE压缩的真彩图像
11：RLE压缩的黑白图像
    // 颜色表规格字段
    std::uint16_t color_map_origin{}; // 颜色表首址    2   颜色表首的入口索引
    std::uint16_t color_map_length{}; // 颜色表长度    2   颜色表表项总数
    std::uint8_t color_map_depth{};   // 颜色表项位数  1   位数，16代表16位
TGA，24代表24位TGA，32代表32位TGA
    //图像规格字段
    std::uint16_t x_origin{};       // 图像X坐标起始位置 2   图像左下角X坐标
    std::uint16_t y_origin{};       // 图像Y坐标起始位置 2   图像左下角Y坐标
    std::uint16_t width{};          // 图像宽度 2    以像素为单位
    std::uint16_t height{};         // 图像高度 2    以像素为单位
    std::uint8_t bits_per_pixel{}; // 图像每像素存储占用位数2   值为8、16、24、
32等
    std::uint8_t image_descriptor{};
};
#pragma pack(pop)
```

图 3-3 TGAHeader 结构体

其中#pragma pack 预处理指令的目的是让该结构体的内存对齐，不然从文件流中直接读取出来的时候数据是错位的，然后便可以使用

```
in.read(reinterpret_cast<char *>(_data.data()), nbytes);
```

来直接将头部载入到结构体了。之后再根据文件头的定义来将数据体载入到动态数组中。

3.3 图形接口封装

本渲染器使用 windows 操作系统提供的原生图形接口 WinGDI 来完成图形的绘制，其大致分为以下几个步骤：

(1) 设计并注册窗口。每一个窗口就是一个由渲染程序打开的窗口实例，将一些基本参数以及窗口处理函数放入 WNDCLASS 结构体后，调用 RegisterClass 宏函数注册该窗口。

(2) 创建窗口并获取句柄。调用 CreateWindow 宏函数将窗口创建出来，并将返回值作为窗口句柄。

(3) 将句柄与本窗口对象使用 map 映射储存。

(4) 使用 while 循环来处理窗口消息。

```
// 对于创建窗口的每个线程，操作系统都会为窗口消息创建一个队列。此队列保存在该线程上创建的所有窗口的消息。
// 队列本身对程序是隐藏的。不能直接操作队列。但是，可以通过调用 GetMessage 函数从队列中提取消息。GetMessage从调用线程的消息队列中取得一个消息并放于msg。
while (GetMessage(&msg, NULL, 0, 0))
{
    //将虚拟键消息转换为字符消息
    TranslateMessage(&msg);
    //将消息分发给窗口处理函数
    DispatchMessage(&msg);
}
```

图 3-4 使用 while 循环处理窗口消息

在全局的窗口处理函数（WindowProc）中，从参数获取到消息类型并使用 switch 进行分支处理。在（1）中将该函数注册到窗口中，操作系统会在有事件发生时（如鼠标点击）自动回调该函数，此时便可以进行处理。

```
/// @brief This WinGDI_Window class is a adapter of WinGDI interfaces.
class WinGDI_Window
{
private:
    // The world
    Renderer::World::World3D& _world;

    friend LRESULT CALLBACK WindowProc(HWND hWnd, UINT uMsg, WPARAM
wParam, LPARAM lParam);
    // handle of window
    HWND _h_wnd;
    // window message process callback
    LRESULT (*_WindowProc)(HWND, UINT, WPARAM, LPARAM);
    // delegate chain
    Utils::Delegate<WinGDI_Window$::WinGDI_Message> _procedure_chain;
    unsigned int _window_width;
    unsigned int _window_height;
};
```

图 3-5 WinGDI 图形设备接口封装类

p.s. 这里使用了委托调用链的思想来代替了 switch 分支以提高可拓展性。

另外，想要通过图形设备接口在屏幕中绘制图像一般有两种方法：

(1) 先逐一将像素点上的颜色设置到兼容位图上，再将整个兼容位图传输到屏幕上。

(2) 直接将一块内存作为位图传输到屏幕上。

在进行性能测试后，发现第二种方案较第一种效率提升很大，所以最后渲染器的实现采用了第二种方案。在 WinGDI 中可以调用 StretchDIBits 来实现这一点：

```
// StretchDIBits 函数将 DIB、JPEG 或 PNG 图像中像素矩形的颜色数据复制到指定的目标矩形。
// 如果目标矩形大于源矩形，此函数将拉伸颜色数据的行和列以适应目标矩形。
// 如果目标矩形小于源矩形，则此函数使用指定的光栅操作压缩行和列。
WINGDIAPI int WINAPI StretchDIBits(
    _In_ HDC hdc, // 目标设备上下文的句柄。
    _In_ int xDest, // 目标矩形左上角的 x 坐标（以逻辑单位为单位）。
    _In_ int yDest, // 目标矩形左上角的 y 坐标（以逻辑单位为单位）。
    _In_ int DestWidth, // 目标矩形的宽度（以逻辑单元表示）。
    _In_ int DestHeight, // 目标矩形的高度（以逻辑单元表示）。
    _In_ int xSrc, // 图像中源矩形的 x 坐标（以像素为单位）。
    _In_ int ySrc, // 图像中源矩形的 y 坐标（以像素为单位）。
    _In_ int SrcWidth, // 图像中源矩形的宽度（以像素为单位）。
    _In_ int SrcHeight, // 图像中源矩形的高度（以像素为单位）。
    _In_opt_ CONST VOID * lpBits, // 指向图像位的指针，图像位存储为字节数组。
    _In_ CONST BITMAPINFO * lpbmi, // 指向包含 DIB 相关信息的 BITMAPINFO 结构的指针。
    _In_ UINT iUsage, // 指定是否提供了 BITMAPINFO 结构的 bmiColors 成员
    _In_ DWORD rop // 一个光栅操作代码，指定如何组合源像素、目标设备上下文的当前画笔和目标像素以形成新图像。
);
```

图 3-6 StretchDIBits 函数

3.4 渲染器架构与渲染管线设计

渲染器对象架构设计如下所示：

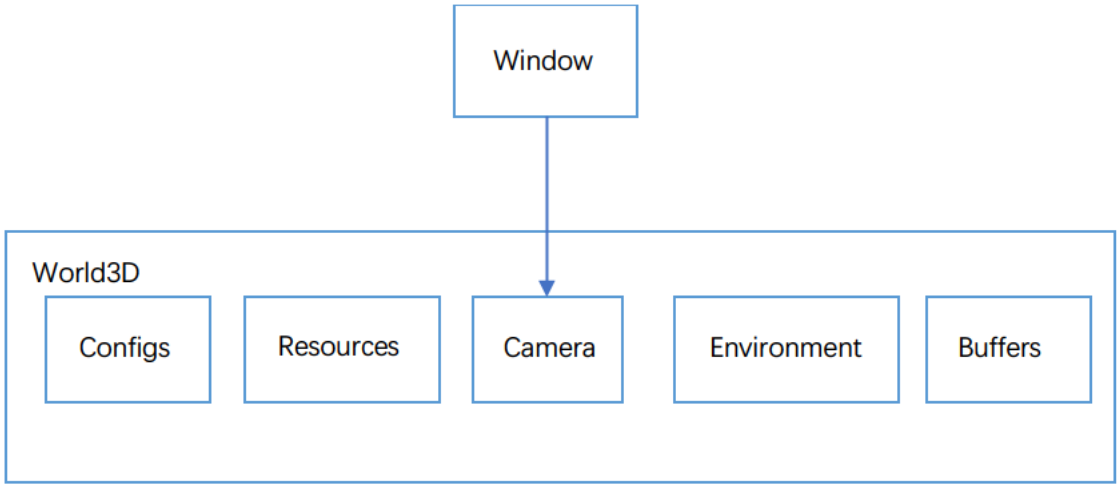


图 3-7 渲染器对象架构设计

其中：

(1) Window 为窗口对象，在本渲染器中为 WinGDI_Window，负责处理与窗口有关的事务。

(2) World3D 为 3D 世界对象，其包括五大部分：

1) Configs，负责存储世界对象的所有配置，如是否使用 CUDA 加速。

2) Resources，负责存储世界中所有对象的属性，如顶点，顶点法线，顶点纹理，由三角形构成的面等。

3) Camera，为该世界中的相机，存有关于目视点的属性，利用它的位置来观测世界。

4) Environment，负责存储世界中环境的属性，如光照模型。

5) Buffers，负责存储世界中的所有以像素为单位的缓存，包括最后生成的位图。

渲染管线设计如下所示：

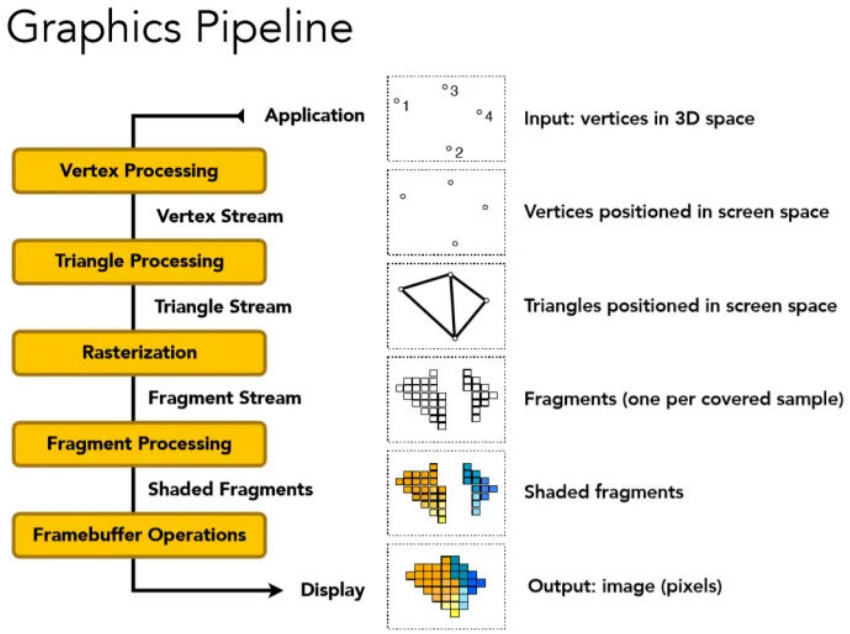


图 3-8 渲染管线示意图

(1) 顶点处理。渲染器通过读取模型文件得到 3D 世界中的一系列顶点，通过对顶点的视图变换，得到这些顶点对应的屏幕空间坐标。

(2) 三角形处理。将屏幕空间中的点所形成的三角形与屏幕像素在 z 方向延长的直线作相交性判断，从而得到三角形在屏幕空间中的位置。

(3) 光栅化。通过深度缓存，按照遮挡关系将三角形逐一投影到屏幕上。

(4) 着色。将纹理映射到模型上，并且根据光照反射模型向世界添加光照。

(5) 帧缓冲区操作。将得到的帧进行处理，如利用反走样技术抗锯齿。

3.5 核心渲染算法实现

3.5.1 视图变换

视图变换的主要工作是将几个变换矩阵与空间中的所有顶点、顶点法线、以及点光源作矩阵乘法。需要注意的是，由于最后在屏幕空间中展现的只有模型顶点所组成的三角形，所以只有模型顶点需要进行投影变换与视口变换。代码如下：

```
// 定义模型位置变换矩阵
 SMatrix model_location_transform =
 {
     1, 0, 0, -lx,
     0, 1, 0, -ly,
     0, 0, 1, -lz,
     0, 0, 0, 1
 };
// 定义相机方向变换矩阵
 SMatrix view_direction_transform =
 {
     cos_a, 0, sin_a, 0,
     -sin_a_sin_b, cos_b, cos_a_sin_b, 0,
     -sin_a_cos_b, -sin_b, cos_a_cos_b, 0,
     0, 0, 0, 1
 };
// 定义相机正上方变换矩阵
 SMatrix view_upward_transform =
 {
     cos_g, -sin_g, 0, 0,
     sin_g, cos_g, 0, 0,
     0, 0, 1, 0,
     0, 0, 0, 1
 };
// 定义投影变换矩阵
 SMatrix projection_transform =
 {
     _nearest_dist, 0, 0, 0,
     0, _nearest_dist, 0, 0,
     0, 0, _nearest_dist + _furthest_dist, -_nearest_dist *
_furthest_dist,
     0, 0, 1, 0
 };
// 定义视口变换矩阵
 SMatrix screen_fit_transform =
 {
     _screen_width / 2, 0, 0, _screen_width / 2,
     0, -_screen_height / 2, 0, _screen_height / 2,
     0, 0, 1, 0,
     0, 0, 0, 1
 };
```

图 3-9 视图变换代码

```
// _p_resources表示存有顶点信息的指针，使用for循环来将所有顶点迭代一遍
for(std::size_t i = 0; i != _p_resources->vertices.size(); i++)
{
    // _p_resources->vertices_transformed表示经过所有变换的顶点集合，
_p_resources->vertices_model_view_transformed表示只经过模型位置变换和相机方向、
正上方变换的顶点集合，_p_resources->vertices表示未经任何变换的初始顶点集合
    // 先将它们设为未经任何变换的初始顶点
    _p_resources->vertices_transformed[i] = _p_resources->vertices[i];
    _p_resources->vertices_model_view_transformed[i] =
_p_resources->vertices[i];

    // 这里model_view_transform是model_location_transform、
view_direction_transform与view_upward_transform的复合矩阵，
projection_screen_transform是projection_transform与screen_fit_transform的复
合矩阵
    // 将复合矩阵与顶点作矩阵乘法
    model_view_transform * _p_resources->vertices_transformed[i];
    model_view_transform *
_p_resources->vertices_model_view_transformed[i];

    projection_screen_transform * _p_resources->vertices_transformed[i];

    // 将变换后的顶点坐标齐次化
    _p_resources->vertices_transformed[i] *= (1 /
(_p_resources->vertices_transformed[i][3]));

}

// 再迭代一遍所有的顶点法线
for(std::size_t i = 0; i != _p_resources->vertex_normals.size(); i++)
{
    _p_resources->vertex_normals_model_view_transformed[i] =
_p_resources->vertex_normals[i];
    model_view_transform *
_p_resources->vertex_normals_model_view_transformed[i];
}

// 将model_view_transform应用到点光源上
_p_bpr_model->ModelViewTransform(model_view_transform);
```

图 3-10 视图变换代码

3.5.2 屏幕投影

屏幕投影的主要工作是对于每一个三角形与每个像素，判断该三角形是否覆盖了该像素点，

并存入深度缓存。代码如下：

```
void Triangle3D::WriteToPixel(size_t x, size_t y, FrameBuffer&
frame_buffer, double nearest_dist, Object* cuda_objects) const
{
    using namespace __Triangle3D;
    // 剪枝，不考虑大于三角形顶点最大x，y或小于三角形顶点最小x，y的所有像素，三角形一
定不会覆盖这些像素
    if(x < Min(_s_v1_x, _s_v2_x, _s_v3_x) || x > Max(_s_v1_x, _s_v2_x,
_s_v3_x)) return;
    if(y < Min(_s_v1_y, _s_v2_y, _s_v3_y) || y > Max(_s_v1_y, _s_v2_y,
_s_v3_y)) return;
    // 剪枝，不考虑未被三角形覆盖的像素
    if(!IsScreenCover((double)x, (double)y)) return;

    // 剪枝，不考虑比深度缓存中的z值还要大的三角形
    if(world_z < frame_buffer.location[2] || world_z > nearest_dist)
return;
    // 将该三角形的z值存入深度缓存中
    frame_buffer.triangle_index = _index;
    frame_buffer.location =
    {
        world_x,
        world_y,
        world_z,
        1
    };

}
```

图 3-11 屏幕投影代码

其中 IsScreenCover 函数实现如下：

```
bool Triangle3D::IsScreenCover(double x, double y) const
{
    // 利用行列式得到各个叉乘
    v1_v2_xy_determinant = Determinant
    (
        _s_v2_x - _s_v1_x, x - _s_v1_x,
        _s_v2_y - _s_v1_y, y - _s_v1_y
    );
```

```
    v2_v3_xy_determinant = Determinant
    (
        _s_v3_x - _s_v2_x, x - _s_v2_x,
        _s_v3_y - _s_v2_y, y - _s_v2_y
    );
    v3_v1_xy_determinant = Determinant
    (
        _s_v1_x - _s_v3_x, x - _s_v3_x,
        _s_v1_y - _s_v3_y, y - _s_v3_y
    );
    // 如果叉乘都同号，则判定覆盖
    if (v1_v2_xy_determinant * v2_v3_xy_determinant >= 0 &&
v2_v3_xy_determinant * v3_v1_xy_determinant >= 0 && v3_v1_xy_determinant *
v1_v2_xy_determinant >= 0)
    {
        return true;
    }

    return false;
}
```

图 3-12 屏幕投影代码

图 3-13 屏幕投影效果图

3.5.3 纹理映射

纹理映射的工作基于屏幕投影。需要先根据每个像素获取其在对应的三角形上的点的屏幕空间中的重心坐标，并还原出它在世界空间中的重心坐标，最后通过透视矫正得到该点在纹理上对应的颜色。代码如下：

```
// 获得该像素投影在三角形上的点在屏幕空间中的重心坐标
Vector screen_areal_coordinates(3);
ScreenArealCoordinates((double)x, (double)y, screen_areal_coordinates);

// 根据屏幕空间中重心坐标与三角形三个顶点的世界坐标还原出该点的世界坐标z值
double world_z = PerspectiveUndo(screen_areal_coordinates, _w_v1_z,
_w_v2_z, _w_v3_z);
// 通过透视矫正得到世界坐标x，y值
double world_x = PerspectiveCorrect(screen_areal_coordinates, _w_v1_x,
_w_v2_x, _w_v3_x, _w_v1_z, _w_v2_z, _w_v3_z, world_z);
double world_y = PerspectiveCorrect(screen_areal_coordinates, _w_v1_y,
_w_v2_y, _w_v3_y, _w_v1_z, _w_v2_z, _w_v3_z, world_z);

// 通过透视矫正得到uv纹理坐标
double img_u = PerspectiveCorrect(screen_areal_coordinates, _vt1_x,
_vt2_x, _vt3_x, _w_v1_z, _w_v2_z, _w_v3_z, world_z);
double img_v = PerspectiveCorrect(screen_areal_coordinates, _vt1_y,
_vt2_y, _vt3_y, _w_v1_z, _w_v2_z, _w_v3_z, world_z);

// 通过透视矫正得到该点的法线向量
frame_buffer.vertex_normal =
{
PerspectiveCorrect(screen_areal_coordinates, _vn1_x, _vn2_x, _vn3_x,
_w_v1_z, _w_v2_z, _w_v3_z, world_z),
PerspectiveCorrect(screen_areal_coordinates, _vn1_y, _vn2_y, _vn3_y,
_w_v1_z, _w_v2_z, _w_v3_z, world_z),
PerspectiveCorrect(screen_areal_coordinates, _vn1_z, _vn2_z, _vn3_z,
_w_v1_z, _w_v2_z, _w_v3_z, world_z),
0
};
// 单位化法线向量
frame_buffer.vertex_normal.Unitization();

// 根据纹理坐标从纹理中得到颜色值
frame_buffer.color  =  _p_objects->at(_object_index).GetImage().Get(img_u,
img_v).rgb;
```

图 3-14 纹理映射代码

图 3-15 纹理映射效果图

3.5.4 表面着色

本渲染器使用基于 Blinn-Phong 表面反射模型的表面着色，需要判断每一个反射点是否存在镜面反射以及是否被遮挡（是否有漫反射），然后计算出具体的镜面反射，漫反射与环境光的贡献值。具体代码如下：

```
// 对于每一个像素与每一个三角形
auto& light_buffer_item = _lights_buffer[LightBufferLoc(_screen_width,
_screen_height, point_light_index, x, y)]; // 获取深度缓存引用
// 获取点光源位置
auto& light_location =
_point_lights[point_light_index].location_model_view_transformed;
// 计算点光源到反射点的距离
auto light_point_distance = light_location - buffer.location;
if (triangle.Index() == buffer.triangle_index)
{ // 表示该三角形在该像素上的投影对于相机可见
    if (light_point_distance < light_buffer_item.distance)
light_buffer_item.distance = light_point_distance; // 更新点光源缓存中的距离
    // 判断是否有镜面反射，此时相机位置(0, 0, 0, 1)，计算半程向量与法线向量的余弦值
    auto cos_theta = point_camera_add_point_light_vector *
buffer.vertex_normal;
    // 如果该余弦值小于最小阈值，则判定为存在镜面反射，计算镜面反射光线
    if (cos_theta >= _specular_min_cos)
    {
        light_buffer_item.is_specular = true;
        light_buffer_item.specular_factor =
SpecularTransition(_specular_min_cos, cos_theta);
    }
}
```

```
else
{ // 此时需要判断三角形是否遮挡了像素反射点的直接光照
    auto light_point_direction = buffer.location;
    light_point_direction -= light_location;
    double light_triangle_distance;
    if (triangle.IsThrough(light_location, light_point_direction,
light_triangle_distance))
    {
        if (light_triangle_distance < light_point_distance)
        { // 如果三角形与光线相交，且它到光线的距离小于反射点到光线的距离，判定为遮挡
            light_buffer_item.distance = light_triangle_distance;
            light_buffer_item.is_exposed = false;
        }
    }
}
```

图 3-16 表面着色代码

```
// 计算三种光线的贡献值，并写入缓存
void BlinnPhongReflectionModel::WriteToPixel(size_t x, size_t y,
FrameBuffer& buffer, DWORD& pixel)
{
    // 初始化rgb，辐照度与三种光线
    buffer.r = buffer.g = buffer.b = 0;
    buffer.power = 0;
    buffer.specular_color = buffer.diffuse_color = buffer.ambient_color =
0;

    for(size_t i = 0; i < _point_lights.size(); i++)
    {
        // 得到点光源缓存，点光源到反射点的距离和方向
        auto& light_buffer_item =
_lights_buffer[LightBufferLoc(_screen_width, _screen_height, i, x, y)];
        auto distance = _point_lights[i].location_model_view_transformed -
buffer.location;
        auto direction = _point_lights[i].location_model_view_transformed;
        direction -= buffer.location;
        direction.Unitization();
```

```
        // 计算辐照度，power = Φ / S * cos(theta)
        auto cos_theta = (buffer.vertex_normal * direction);
        if (cos_theta <= 0) continue;
        auto power = (_point_lights[i].power / (4 * Maths::PI * pow(distance,
2))) * cos_theta;
        buffer.power += power;

        // 根据辐照度与纹理rgb值计算最终的反射颜色
        auto receive_light_color = RGBMul(_point_lights[i].color, power);
        DivideRGB(
            RGBReflect(receive_light_color, buffer.color),
            buffer.r, buffer.g, buffer.b,
            [](unsigned int& y, RGB x){ y += x; }
        );

        // 根据之前的判断计算镜面反射与漫反射
        buffer.specular_color += GenerizeReflection(buffer.r, buffer.g,
buffer.b, power * light_buffer_item.specular_factor *
light_buffer_item.is_specular * light_buffer_item.is_exposed);
        buffer.diffuse_color += GenerizeReflection(buffer.r, buffer.g,
buffer.b, power * _diffuse_factor * light_buffer_item.is_exposed);
    }

    // 根据漫反射系数计算漫反射
    buffer.ambient_color += RGBMul(buffer.color, _ambient_factor);

    // 线性叠加三种光线，得到最终的rgb值
    pixel = RGBAdd(buffer.ambient_color, buffer.diffuse_color,
buffer.specular_color);

}
```

图 3-17 表面着色代码

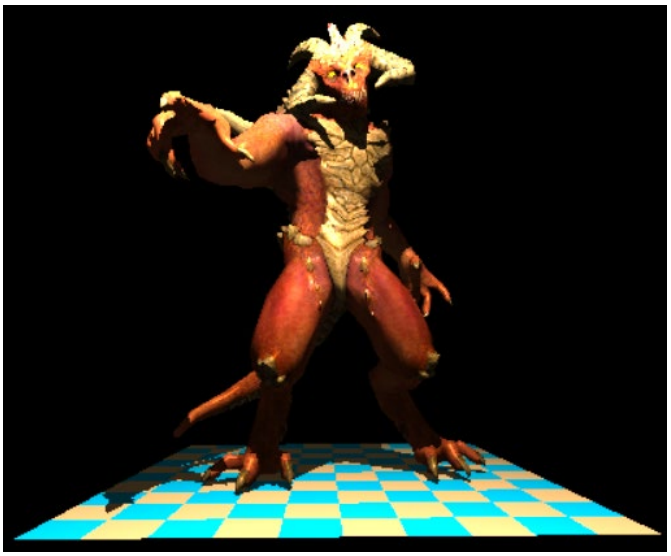


图 3-18 表面着色示意图

3.6 利用 CUDA 并行框架加速

3.6.1 核函数与设备函数

CUDA 从逻辑上将 GPU 线程分成了三个层次——线程格（grid）、线程块（block）和线程（thread）。每个核函数对应一个线程格，一个线程格中有一个或多个线程块，一个线程块中有一个或多个线程。在一维的情况下，三者关系如下。

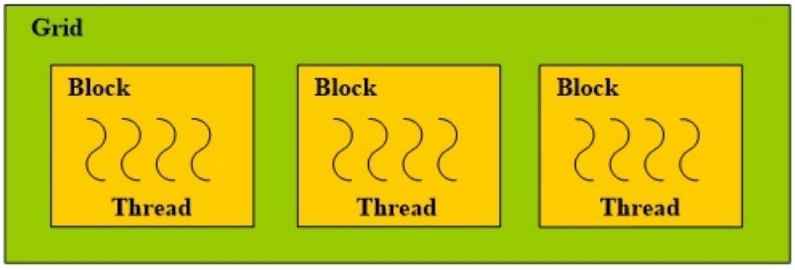


图 3-19 CUDA 线程层次

可以将 Grid 想象为一栋楼，将 Block 想象为楼里面的房间，而 Thread 就是房间里面的工作人员。由于 GPU 的运行独立于 CPU，它们分别位于设备运行空间（Device）和主机运行空间（Host），因此想要通过 CPU 启动 GPU 线程时，就需要一个入口函数来调用，这个入口函数就是核函数。启动一个核函数就像将一项任务交给一栋楼来完成，楼将任务分解给各个房间，房间再将任务分解给各个工作人员。

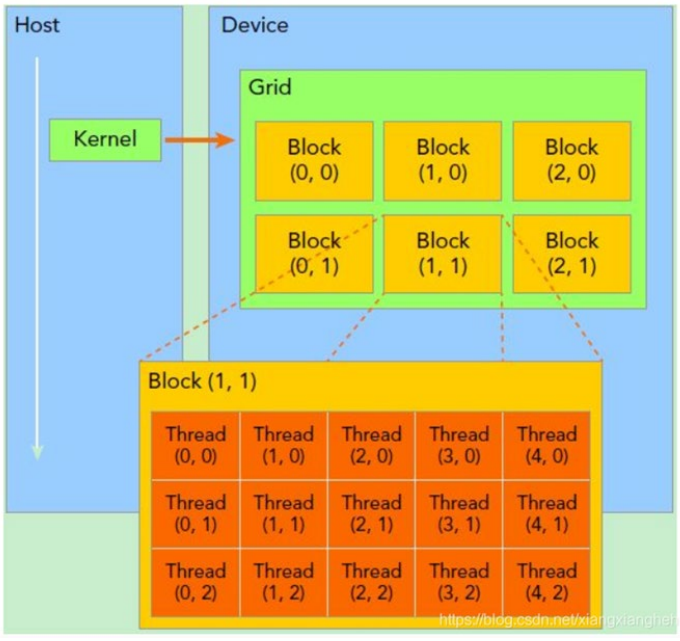


图 3-20 从 CPU 启动 GPU 线程

在 CUDA C++源文件中可以在函数定义前加上“__global__”来将它标记为核函数，然后在 CPU 中调用它。

```
// Running on GPU
__global__ void Kernal(int param)
{
    // Do something or call other device functions.
}

// Running on CPU
void HostFunction()
{
    Kernal<<<block_num, thread_num>>>(param);
}
```

图 3-21 __global__标记核函数

可以看到，主机函数通过调用`Kernal<<<block_num, thread_num>>>(param);`来调用核函数。其中核函数的配置项 block_num 表示使用的线程块数，thread_num 表示每个块使用的线程数。另外还可以透过填入核函数调用所在的 stream 将每个核函数调用当作一个流，然后利用流水线技术来充分发挥 GPU 的并行能力。

GPU 中的每个线程独立执行核函数，那么如果想在核函数中直接调用 CPU 中的宿主函数（例如 stl 库函数）是否可行呢？事实告诉我们不可行，因为 GPU 运行的设备代码是经过特别编译后放在了 GPU 的设备代码区的，GPU 没有办法调用 CPU 代码区的函数，自然对于 GPU 所调用的函数也需要特殊处理，可以通过在函数定义前加上“__device__”来将它标记为设备函数，这样在编译器编译函数时，就会将它编译到 GPU 的代码区中了。

```
// Running on GPU
__device__ void Device(int param)
{
    // Do something.
}


// Running on GPU
__global__ void Kernal(int param)
{
    Device();
    // Do something or call other device functions.
}

// Running on CPU
void HostFunction()
{
    Kernal<<<block_num, thread_num>>>(param);
}
```

图 3-22 __device__标记设备函数

在调用核函数时，可以像普通函数一样传入简单对象的参数，也可以传入指向一个复杂对象的指针。但要注意，由于 GPU 内存独立于 CPU，所以需要先把位于 CPU 内存的对象拷贝到 GPU 内存中才能在 GPU 中使用它，具体操作如下：

(1) 在主机代码中调用 cudaMalloc()函数在设备内存中分配指定大小的内存。

(2) 在主机代码中调用 cudaMemcpy()函数将指定主机内存拷贝到设备内存中。

(3) 将设备内存指针传入核函数，并在设备代码中使用指针[37]。

3.6.2 使用 nvcc 编译设备代码

Nvcc 编译器用于处理输入的所有 CUDA C++源文件（.cu）。首先，它将这些文件中的设备代码编译为.ptx 汇编文件，并将它们分别汇编为二进制.cubin 文件，并将它们放在 fatbinary 文件中。然后，编译器将处理主机代码，使用 cudafe++将其转换为标准 C++结构。主机编译器将主机代码与 fatbinary 结合编译为.o/.obj 文件。在主机设备启动代码时，CUDA run-time 系统将检测 fatbinary 文件，以获取一个适合 GPU 设备的映像。如果之前的 cubin 文件与当前 GPU 相匹配，则 run-time 系统将通过 nvlink 对多个设备端文件进行链接，并通过 host linker 链接主机端文件成可执行文件。如果 cubin 文件所表示的虚拟设备与当前 GPU 不符或者 fatbinary 文件中只包含 ptx 文件，则将对 fatbinary 文件中的 ptx 文件进行即时编译，并对其进行链接（nvlink），最后通过 host linker 链接主机端文件成可执行文件[38]。

这里使用 nvcc 编译 CUDA C++源文件，编译过程如下图所示。

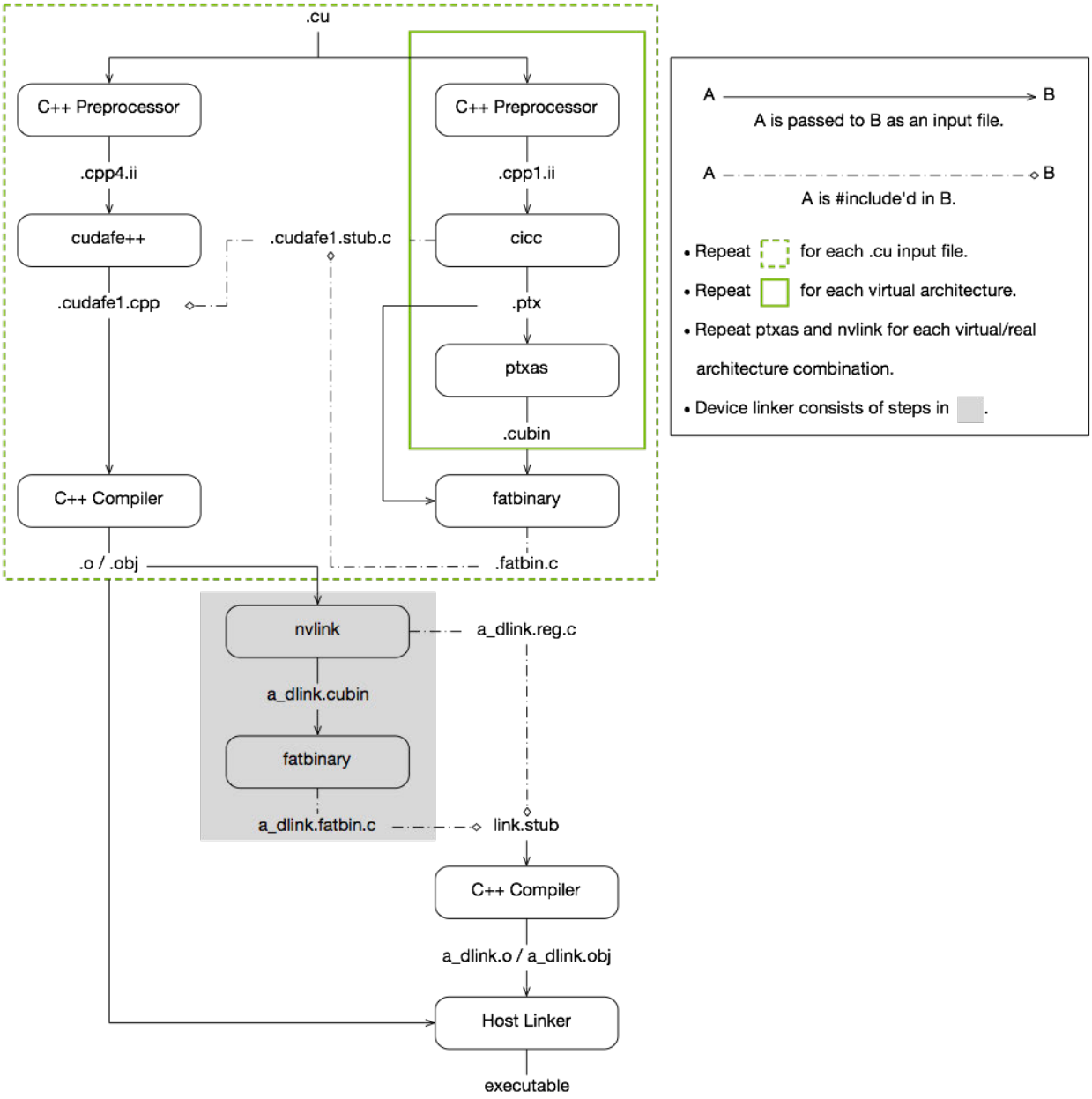


图 3-23 nvcc 编译流程

3.6.3 将 CPU 代码移植到 GPU

现在需要将原来在 CPU 中调用的代码移植到 GPU 中，首先需要判断应当有哪些代码需要被移植到 GPU 上，渲染器通过世界对象中的 Build()成员函数来启动渲染，因此可以通过寻找 Build()函数的所有引用，来获得需要移植的函数名单，在 Visual Studio 中寻找如下。

图 3-24 利用 Visual Studio 寻找函数依赖

这仅仅是 Build()函数的直接引用，而我需要寻找到它所有的直接引用与间接引用，然后将他们的实现移植到 CUDA C++源文件中。此时这部分函数的头文件声明有可能同时被主机端代码和设备端代码实现在不同的 obj 文件中，因此需要对这种情况加以判断，如果该声明被设备端代码实现，则判断位于 CUDA 环境中，并且将其声明为设备函数，可以简单用预编译条件指令实现：

```
#ifdef __CUDA_RUNTIME_H__
                __device__
#endif
```

图 3-25 通过预编译指令判断主机/设备环境

将其加在函数声明的前面即可，当被 CUDA C++源文件包括时，__CUDA_RUNTIME_H__将被定义，因此会在函数声明的前面声明__device__。

当设备代码编译完后，可以选择将其直接与主机代码静态链接，也可以将其单独封装为一个动态链接库，然后从主机端动态调用它。由于我选择了类成员函数在主机端和设备端中的双定义，如果采用静态链接会出现重定义错误，因此这里采用了动态链接的方式，这样做也能很清楚地将主机代码和设备代码作区分。

这里对动态调用函数做一个宏封装：

```
#pragma once
#include <Windows.h>
#include "string.hpp"
#include "logs.hpp"

#define c_export extern "C" __declspec(dllexport)

#define func_type(func) Type_##func

#define func_p(func) (*func_type(func))

#define dll HINSTANCE
```

```
#define load_dll(dll_src, mount, log_name) mount =
LoadLibrary(STR(dll_src.dll)); \
if(!mount) { Log::Error(log_name, "Failed to load %s, error code: %d.\n",
STR(dll_src), GetLastError()); PRINT_LOCATION; }

#define import_func(func, dll_src, mount, log_name) mount =
(func_type(func))GetProcAddress(dll_src, STR(func)); \
if(!dll_src) { Log::Error(log_name, "Invalid dll source %s\n", STR(dll_src));
PRINT_LOCATION; } \
if(!mount) { Log::Error(log_name, "Failed to import %s from %s, error
code: %d.\n", STR(func), STR(dll_src), GetLastError()); PRINT_LOCATION; }
```

图 3-26 宏封装动态库调用

其中：

(1) c_export 为 C 函数动态链接库导出的声明。

(2) func_type 为对导出函数类型的通用命名。

(3) func_p 为导出函数对应的函数指针类型。

(4) load_dll 根据 dll_src 给出的库名，通过 LoadLibrary()动态导入库，并赋予一个 HINSTANCE 实例变量。

(5) import_func 通过 HINSTANCE 实例变量从库中根据函数名导出想要的函数。

之后在需要被主机端包含头文件中声明导出的函数和函数指针类型：

```
#pragma once
#include "kamanri/utils/imexport.hpp"
// used models
#include "kamanri/renderer/world/world3d.hpp"

// export functions codes
typedef unsigned int BuildWorldCode;

// export functions types
typedef BuildWorldCode func_p(BuildWorld)
(Kamanri::Renderer::World::World3D* p_world, unsigned int width, unsigned
int height);
```

图 3-27 声明导出函数与函数指针类型

并在另一个头文件中包括它，声明要导出的函数（这样做是为了避免函数声明出现在主机代码中，与函数指针类型的命名出现冲突）：

```
#pragma once
#include "cuda_dll/exports/build_world.hpp"

c_export    BuildWorldCode    BuildWorld(Kamanri::Renderer::World::World3D*
p_world, unsigned int width, unsigned int height);
```

图 3-28 声明导出函数

这样就可以在主机代码中愉快地调用库函数了，然后再由库函数来负责调用核函数。

3.7 键盘交互

交互也是渲染器中一个必不可少的部分，许多专业的渲染器有着各种各样的参数，可以通过 UI 框架来设定。这里作最简单的简化：通过监听键盘事件来移动相机，从而改变观察位置与方向。

WinGDI 中通过 GetMessage()来循环获取消息事件，这里设定几个按键来移动相机，并得到它在 message 中的消息类型。

表 3-2 WinGDI 消息类型

按键	作用	消息类型
W	向前移动	0x57
A	向左移动	0x41
S	向后移动	0x53
D	向右移动	0x44
Q	向上移动	0x51
E	向下移动	0x45
↑	视角向上	0x26
↓	视角向下	0x28
←	视角向左	0x25
→	视角向右	0x27

之后设置好单位移动距离和单位移动角度，并在 switch 语句中根据消息类型对相机作出适当变换。这里只以向前移动为例，当 W 键按下时，将相机的位置乘以向相机朝向移动单位距离的矩阵，其他消息也与此类似。此时设置渲染线程循环渲染，然后设置消息监听线程循环监听消息，这样就可以及时地将键盘消息反馈到屏幕上去了。

```
if (message.u_msg == WM_KEYDOWN)
{

    auto& location = message.world.GetCamera().Location();
    auto& direction = message.world.GetCamera().Direction();
    auto& upward = message.world.GetCamera().Upward();

    switch (message.w_param)
    {
    case W_KEY:
        _move =
        {
            1, 0, 0, direction[X] * _min_move_location,
            0, 1, 0, direction[Y] * _min_move_location,
            0, 0, 1, direction[Z] * _min_move_location,
            0, 0, 0, 1
        };
        _move * location;
        break;

    // other cases..

    default:
        break;
    }

}
```

图 3-29 设置消息处理

第 4 章 总结与展望

4.1 总结

本文对现代计算机图形学当中的光栅化渲染理论进行了详细阐述和梳理，并在此基础上设计了一套渐进式渲染管线的实现流程，最后使用 C++实现了基本框架以及所有的核心渲染算法，实现了一个 3D 软渲染器。该渲染器能够从文件中读取模型与纹理，并在图形窗口中将 3D 模型渲染出来，并且能够通过键盘来控制相机的方向，从而能从不同角度对模型进行观察。本渲染器将论文中除了高级反走样技术之外的所有理论均进行了代码实现。除此之外，本渲染器还利用了论文中未提及的 AABB 轴对齐边界包围盒技术，大大加快了渲染速度。本渲染器在设计之初就被设计为可以跨平台使用的渲染器，但目前仅利用了 Windows 的图形接口实现了 Windows 端的代码调用，之后仍然可以就实际情况拓展到其他平台。

本渲染器仍然存在着许多的不足之处，例如，作为一个光栅化渲染器，它很可能没有充分利用好图形处理器的硬件资源，导致渲染速度仍不尽如人意（1.5s-2s 每帧）。在使用 Visual Studio 的性能探查器和 Nvidia 的 Nsight Compute 后，我也发现了许多算法上可以改进和加速的地方。另外本渲染器读取的模型种类不够多样，交互方式较为简单，日后我也会考虑添加更多的模型种类，添加 GUI 交互方式，将渲染器做得更完善。

4.2 展望

计算机图形学的应用前景非常广泛，也可以与许多其他的领域去结合，渲染技术固然只是整个图形学中小小的一环，但它却也如磐石一般的不可或缺。未来我会在我的渲染器的基础上继续深入对计算机图形学的学习与研究，包括但不限于：

(1) 利用各种光线追踪技术实现全局光照渲染。

(2) 研究高质量实时渲染。

(3) 研究深度学习在图形渲染领域中的应用。

(4) 探索图形渲染的应用方向，譬如元宇宙的应用。

不管怎么说，图形渲染所发展的方向始终是清晰且明确的，即，更高的精度，更快的速度，更真实的画面。图形学的发展也逐渐从符合人类直觉的“经验模型”转向了符合自然的“物理规律”，正如科学能比宗教更好地解释自然现象一样，物理规律也肯定能比经验模型更能反映真实世界，而这不得不说是人类算力的伟大进步。我期待一定会有未来的一天，自然规律能够完美地融入虚拟世界之中，身处于其中的人类，尤其是孩子，已经完全无法区分虚拟与现实，虚拟世界也能够帮助我们完成更多现实生活中所不能完成的事情。

致 谢

我在此想要表达我最深切的谢意和感激之情。在我的学术之旅中，我收获了很多知识和经验，这些都离不开您们的支持和帮助。

首先，感谢我的指导老师[redacted]，从毕业设计选题到正式开发实现再到论文的撰写，只要我有什么不懂或者不清楚的地方，他都会耐心而细致的给我讲解，还会针对我的实际情况给我提出很多建设性的建议。感谢[redacted]老师，不辞辛苦、严谨认真而负责，即使再忙也会抽出时间来为我答疑解惑。

其次，感谢我的家人和朋友。他们一直支持我，鼓励我在学术上不断取得进步。没有他们的支持，我不可能走到今天的地步。

最后，我还要感谢我的学校和所有为我提供帮助的机构和人员。他们的支持和帮助为我提供了更好的学习和研究环境。感谢学校，感谢四年来任课的所有老师。是学校合理、细致而周到的教学课程安排，是各位老师们认真细致的教学，让我从一个懵懂无知的青涩少年，蜕变成了一个计算机行业的开发者。

在此，我要再次向所有支持和帮助我的人表示最真诚的感谢。在未来的学术道路上，我将会继续不断努力，为学术事业贡献自己的力量。

谢谢！

参考文献

[1] FOLEY, DAM, VAN, FEINER, et al. ComputerGraphics: Principles and Practice [M]. Addison Wesley, 1990.

[2] 唐春芳. 计算机图形学与图形图像处理技术 [Z]. 电子技术与软件工程. 2018.

[3] 刘明. 基于 OpenGL 的大规模场景实时渲染技术的研究 [D]. 华中科技大学, 2007.

[4] 高海峰. 基于球面调和理论及其相关技术的高级光照模型研究 [D]. 合肥工业大学, 2011.

[5] PHARR, HUMPHREYS. Physically Based Rendering [M]. Morgan Kaufmann, 2004.

[6] 张彻. 机械工程简史 [M]. 清华大学出版社, 2015.

[7] 杨旻. 基于 MultiGen 的真实感虚拟场景绘制研究 [D]. 北京交通大学, 2008.

[8] 邓军勇, 李涛, 蒋林, et al. 面向 OpenGL 的图形加速器设计与实现 [J]. 西安电子科技大学学报, 2015, 42(06): 124-130.

[9] 阮航, 张义伟, 王炜. 舰载显控系统中二维矢量图形加速器的设计; proceedings of the 中国造船工程学会电子技术学术委员会 2017 年装备技术发展论坛, 中国浙江杭州, F, 2017 [C].

[10] 鲁英春, 绵阳师范学院信息工程学院. 计算机图形图像处理相关技术探讨 [J]. 信息与电脑(理论版), 2018, (16): 133-135.

[11] 裴云强, 吴亚东, 王赋攀, et al. 基于改进 LK 光流的 WebAR 信息可视分析方法 [J]. 图学学报, 2020, 41(6): 962.

[12] 张范文, 李中, 陆淑蕾, et al. 基于纹理的流场可视化技术在水下生产系统中的应用与展望 [J]. 中国海洋平台, 2023, 38(02): 103-108.

[13] 于尚平, 侯冠宇, 王艳, et al. 近 10 年重症护理超声研究热点的 CiteSpace 可视化分析 [J]. 现代临床护理, 2022, 21(11): 28-36.

[14] 鄂金龙, 何林. 基于异构算力节点协同的高效视频分发 [J]. 计算机研究与发展, 2023, 60(04): 772-785.

[15] 程锦, 叶虎强, 冯劲松, et al. 三维 CAD 软件性能自动化测试与变粒度可视评价 [J]. 西安交通大学学报: 1-13.

[16] Introduction to Computer Graphics and Ray-Tracing Using the WebGPU API [Z]. ACM SIGGRAPH Asia 2022 Courses. Daegu, Republic of Korea; Association for Computing Machinery. 2023: Article 1.10.1145/3550495.3558218

[17] 闫令琪, UCSB. GAMES101——现代计算机图形学入门 [Z]. 加州大学圣巴巴拉分校. 2020

[18] POLYCHRONAKIS, ANDREAS, KOULIERIS, GEORGE ALEX, MANIA, KATERINA. Emulating Foveated Path Tracing [Z]. Proceedings of the 14th ACM SIGGRAPH Conference on Motion, Interaction and Games. Virtual Event, Switzerland; Association for Computing Machinery. 2021: Article 10.10.1145/3487983.3488295

[19] 王欣. 大规模场景分布式并行光子映射方法研究 [D]. 山东大学, 2019.

[20] PANTALEONI, JACOPO. Charted metropolis light transport [J]. ACM Trans Graph, 2017, 36(4): Article 75.

[21] JIN, BONGJUN, IHM, INSUNG, CHANG, BYUNGJOON, et al. Selective and adaptive supersampling for real-time ray tracing [Z]. Proceedings of the Conference on High Performance Graphics 2009. New Orleans, Louisiana; Association for Computing Machinery. 2009: 117–125.10.1145/1572769.1572788

[22] XIAO, LEI, NOURI, SALAH, CHAPMAN, MATT, et al. Neural supersampling for real-time rendering [J]. ACM Trans Graph, 2020, 39(4): Article 142.

[23]路勇良, 张伟, 赵军, et al. 基于弧长参数的自由曲线实时误差估计算法 [J]. 山东大学学报(工学版): 1-9.
[24]KAUKER, D., KRONE, M., PANAGIOTIDIS, A., et al. Rendering molecular surfaces using order-independent transparency [Z]. Proceedings of the 13th Eurographics Symposium on Parallel Graphics and Visualization. Girona, Spain; Eurographics Association. 2013: 33–40
[25]ZHAO, SHUANG, JAKOB, WENZEL, LI, TZU-MAO. Physics-based differentiable rendering: from theory to implementation [Z]. ACM SIGGRAPH 2020 Courses. Virtual Event, USA; Association for Computing Machinery. 2020: Article 14.10.1145/3388769.3407454
[26]UKAJI, HIROAKI, KOSAKA, TAKAHIRO, HATTORI, TOMOHITO, et al. Acquiring shell textures from a single image for realistic fur rendering [Z]. ACM SIGGRAPH 2012 Posters. Los Angeles, California; Association for Computing Machinery. 2012: Article 100.10.1145/2342896.2343013
[27]SCHÜSSLER, VINCENT, HEITZ, ERIC, HANIKA, JOHANNES, et al. Microfacet-based normal mapping for robust Monte Carlo path tracing [J]. ACM Trans Graph, 2017, 36(6): Article 205.
[28]KURT, MURAT, SZIRMAY-KALOS, LÁSZLÓ, KŘIVÁNEK, JAROSLAV. An anisotropic BRDF model for fitting and Monte Carlo rendering [J]. SIGGRAPH Comput Graph, 2010, 44(1): Article 3.
[29]YAN, LING-QI, TSENG, CHI-WEI, JENSEN, HENRIK WANN, et al. Physically-accurate fur reflectance: modeling, measurement and rendering [J]. ACM Trans Graph, 2015, 34(6): Article 185.
[30]朱静洁. 基于深度学习的图像风格转换方法 [D]. 杭州电子科技大学, 2020.
[31]孙小磊. 计算机图形学系列笔记 [Z]. 电子科技大学. 2020.
[32]SHIRLEY, STEVE MARSCHNER, PETER. Fundamentals of Computer Graphics 5th [M]. Taylor & Francis Group, 2022.
[33]曹岳. 基于 3DMM 的三维人脸重建 [D]. 中国石油大学(华东), 2019.
[34]HOFFMAN, NATY, BAKER, DAN. Reflectance rendering with point lights [Z]. ACM SIGGRAPH 2006 Courses. Boston, Massachusetts; Association for Computing Machinery. 2006: 4–es.10.1145/1185657.1185758
[35]YUMEFX. 纹理映射之插值与 Mipmap [Z]. yumefx. 2022.
[36]韩清. 基于图片生成具有纹理贴图的三维人物化身研究 [D]. 合肥工业大学, 2020.
[37]王玉龙. 基于 GPU 的视频大数据处理方法研究 [D]. 大连理工大学, 2017.
[38]HARRIS, MARK. Many-core GPU computing with NVIDIA CUDA [Z]. Proceedings of the 22nd annual international conference on Supercomputing. Island of Kos, Greece; Association for Computing Machinery. 2008: 1.10.1145/1375527.1375528