# 3渲染管道

Direct3D 11渲染管道是用于使用GPU将内存资源处理为渲染图像的机制。管道本身由许多较小的逻辑单元组成，称为管道阶段。数据通过管道一次处理一个阶段，并在每个阶段以某种方式进行操作。通过了解管道的各个阶段是如何操作的，以及使用它们的语义，我们可以将管道作为一个整体来实现实时执行的各种算法。

随着GPU随着每一代新体系结构的出现而变得越来越强大，流水线的规模和能力都有了显著的扩展。此外，每个管道阶段的复杂性和可配置性稳步增加。当前渲染管道具有固定的功能阶段以及可编程着色器阶段。本章将首先考虑这些类型的管道阶段之间的差异。特别是，它将重点关注使它们运行所需的状态，以及它们可以执行的处理。在澄清这一区别之后，我们将考虑管道如何调用的更高级细节，以及每个流水线阶段如何与其邻居进行通信。

然后，我们将详细探讨每个管道阶段。这包括每个阶段执行的单个功能、阶段如何配置的详细讨论以及它们带来的一般语义。通过对管道的每一个单独组件的理解，我们可以考虑在各种配置中由流水线级组实现的一些高级功能。我们还将讨论整个管道的几个高级数据处理概念，包括如何管理这样一个复杂的处理体系结构。渲染管道已经演变成一套复杂的API，可用于实现多种算法。完成本章后，我们将深入透彻地了解如何使用渲染管道来开发用于实时渲染应用程序的高效有趣的渲染技术。

## 3.1管道状态

为了理解管道是如何运作的，我们只需要看看它的名字。数据在管道的一端作为输入提交，然后由第一个管道阶段进行处理。该数据由最多四个分量的基于向量的变量组成。第一阶段的处理完成后，修改后的输出数据将传递到下一阶段。然后将下一组数据带入第一阶段。这意味着前两个阶段同时处理不同的数据段。重复此过程，直到整个管道在输入数据的不同部分上同时运行。管道体系结构特别允许通过不同的管道阶段同时执行多个操作，这允许在单个数据项通过管道时对其执行许多专门的过程。一旦数据项到达管道的末端，它就会存储到输出资源中，稍后可以根据主机应用程序的需要使用该资源。这种流水线概念是一种简单但功能强大的处理技术，本质上类似于装配线。图3.1显示了管道如何处理数据。

开发人员的任务是正确配置管道的每个阶段，以便在数据从管道末端出现时获得所需的结果。管道配置过程通过操纵管道的每个单独阶段的状态来执行。通过将管道组织为多个阶段，Direct3D 11有效地将相关的状态集组合在一起，并整合它们的操作方式。有两种不同类型的管道级，固定功能级和可编程着色器级。这两种阶段类型都有一些关于数据如何流经它们以及应用程序如何操纵它们的状态的共同概念。以下各节将探讨这两种状态，并提供如何使用它们的一些一般概念。

### 3.1.1固定管道级

固定功能阶段对传递给它们的数据执行一组固定的操作。它们执行特定的操作，因此提供了可用功能的“固定”范围。它们可以以各种方式进行配置，但它们的性能始终相同*管道*

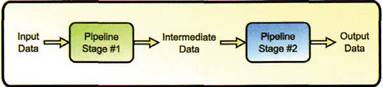


图3.1。Direct3D 11渲染管道。

对传递给它们的数据的操作。这个概念的一个有用的类比是考虑正则函数如何在C++中工作。首先向函数传递预定义的参数列表。然后，它处理数据并将结果作为其输出返回。函数体表示固定函数阶段实现的操作，而输入参数是可用配置和实际输入数据。我们不能更改函数体，但可以控制在处理输入数据期间使用的选项。

前几代Direct3D中此类固定函数的一些示例包括更改顶点剔除顺序（现在是光栅化器阶段的一部分）、选择深度测试函数（现在是输出合并阶段的一部分）和设置alpha混合模式（也是输出合并阶段的一部分）的功能。这种对单个功能领域的关注主要是出于性能考虑。如果特定管道阶段是为特定任务设计的，则通常可以对其进行优化，以比通用解决方案更有效地执行该任务。

在Direct3D 9中，通过为每个要更改的单独设置调用API函数来更改这些固定函数阶段的状态。这需要执行许多API调用来为每个阶段配置特定的处理设置。根据场景内容和渲染配置，API调用的数量很容易增加，并开始导致性能问题。Direct3D 10引入了状态对象的概念，以替换这些单独的状态设置。状态对象用于通过单个API调用配置完整的功能状态。这大大减少了配置状态所需的API调用数量，还减少了运行时所需的错误检查量。Direct3D 11遵循这种状态对象范例。应用程序必须使用描述结构描述所需的状态，然后从中创建状态对象，该对象可用于控制固定功能管道阶段。

一次性创建完整状态的要求将状态验证从运行时API调用转移到状态创建方法。如果同时配置了不兼容状态，则会在创建时返回错误。创建后，状态对象是不可变的，不能修改。因此，无法为固定函数管道设置无效状态，这将有效地消除状态设置API调用的验证负担。总的来说，这种使用状态对象来表示管道状态的系统允许更精简的管道配置，只需要最少的应用程序交互。

### 3.1.2可编程管道级

可编程管道阶段包括管道的其余阶段。这里的“可编程”一词意味着这些阶段可以执行用高级着色语言（HLSL）编写的程序。通过使用上面的C++类比，可编程阶段允许您定义所需的输入参数和函数的主体。事实上，在这些阶段运行的程序是作为HLSL中的函数编写的。

120                                                                                                                                                     3.渲染管道

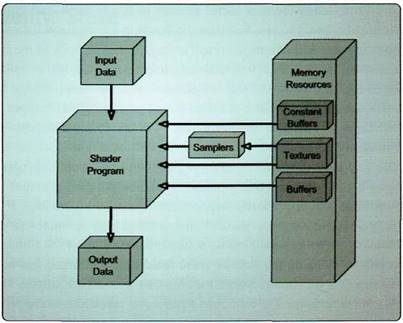


 图3,2。公共着色器核心的框图。

执行程序的能力使这些管道阶段可用于各种各样的处理任务。这与固定功能阶段形成对比，固定功能阶段用于非常特定的任务，只提供少量的可配置性。在这些可编程管道阶段中执行的程序通常被称为着色器（shader）。当像素着色器阶段最初用于修改对象对照明的反应时，该名称继承自可编程性的早期步骤。随着越来越多的可编程阶段被添加到管道中，着色器的名称被用来指代所有可编程管道阶段。我们将在整本书中交替使用这个术语和可编程阶段。在以下各节中，我们将首先从较高的层次上了解这些可编程着色器阶段，然后在建立基本概念后深入了解其体系结构的细节。*程序，*

公共着色器核心

所有可编程着色器阶段都建立在公共功能基础之上，称为公共着色器核心。公共着色器核心定义管道级的通用输入和输出设计，提供所有可编程着色器级支持的一组内在函数，以及可编程着色器可以使用的资源接口。图3.2提供了公共着色器核心操作方式的可视化表示。如上所述，数据流入舞台顶部，在着色器核心内进行处理，然后作为输出流出舞台

舞台的底部。在这些阶段中执行的着色器程序是为特定目的而用HLSL编写的函数。在着色器阶段内处理数据时，着色器程序可以访问由应用程序绑定到该阶段的常量缓冲区、采样器和着色器资源视图。一旦着色器程序完成了对其数据的处理，该数据将从后台传递出去，并引入下一段数据以重新开始该过程。

可编程管道级的可配置性不限于着色器程序内执行的处理。这些阶段的输入和输出数据结构也由着色器程序指定，该程序提供了一种从阶段到阶段传递数据的灵活方式。在这些阶段之间的接口上施加了一些规则，例如要求在特定阶段的接口中包含某些类型的数据。所需输出的一个典型示例是光栅化器之前的一个阶段提供位置输出，该位置输出可用于确定渲染目标的哪些像素被基本体覆盖。根据传递数据的阶段，参数要么未经修改提供，要么可以在传递到下一阶段之前进行插值。

虽然所有可编程阶段共享一组共同的功能，但每个阶段还可以提供其特有的其他专门特性和功能。这些通常与输入和输出语义相关，因此仅适用于特定阶段。当我们描述管道的每个阶段时，将在本章后面更详细地讨论每个阶段的单独行为。

由于具有如此大的灵活性和各种不同的管道阶段，有许多算法不需要所有阶段都处于活动状态。因此，可以通过清除其着色器程序来禁用可编程着色器阶段。此外，由于在每个阶段中进行的处理具有很大的灵活性，因此很可能在所有可编程阶段的不同组合中执行部分或全部所需工作。这种选择在何处执行特定计算的能力可用于我们的优势。如果可以在任何数据放大之前进行计算，则可以用少得多的操作计算相同的数据。我们将在第8章“网格渲染”中看到这方面的一个很好的示例，其中在对模型进行细分之前执行顶点蒙皮，以减少蒙皮顶点的数量。（有关顶点蒙皮的详细信息，请参见第8章。）

着色器核心体系结构

在上一节中，我们已经了解了可编程阶段如何运行的一般概念。它接收来自前一个阶段的输入，在其上执行HLSL程序，并将结果传递到下一个管道阶段。然而，这只是对实际情况的概述。在可编程阶段执行的着色器程序实际上是从HLSL编译成基于向量寄存器的汇编语言，设计用于GPU中的专用着色器处理器核心。即使所有着色器程序

122                                                                                                                                                     3.渲染管道

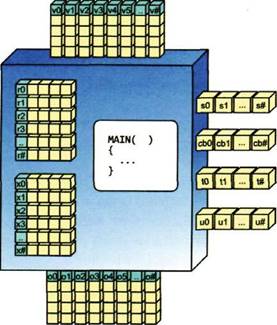


图3.3。公共着色器核心的汇编语言视图。

必须用HLSL编写，但在渲染管道中使用之前，它们仍必须编译为此程序集字节码。12

我们可以从这种汇编语言中学到很多信息。它定义了一组特定的寄存器，编译器可以使用这些寄存器将HLSL程序映射到汇编语言。寄存器通常是四分量向量寄存器，可以使用单个分量来提供标量寄存器功能。着色器核心有用于接收其输入数据的寄存器、用于执行计算的临时寄存器、用于与资源交互的寄存器以及用于将数据传递出后台的寄存器。汇编语言指令使用这些寄存器执行各自的操作。通过了解汇编程序如何使用这些寄存器，我们可以深入了解着色器阶段的操作方式。

为了开始，我们将考虑图3.2中所示的常见着色器核心概述，但这次我们将从汇编语言的角度来查看它。图3.3显示了公共着色器核心的组件版本。

1 编译HLSL程序生成的汇编程序不会直接在GPU中执行。视频驱动程序将其进一步处理为特定于机器的指令，这些指令可能因GPU而异。即便如此，汇编语言还是提供了一个共同的参考点，我们可以借助它深入了解着色器处理器的操作。

2 编译过程的细节可以在第6章“高级着色语言”中找到

3.1管道状态123

如图3.3所示，着色器核心的输入在v#寄存器中提供。由于它们向后台提供输入，因此它们自然是只读的。当执行着色器程序时，其输入数据在v#寄存器中可用。读取数据后，可以对其进行操作并与其他数据以及任何中间数据组合e计算可以存储在r#和x#[n]寄存器中。这些寄存器称为临时寄存器，由于它们持有中间值，它们都可以被着色器程序读写。纹理寄存器（t#）、常量缓冲寄存器（cb#[n]）、即时常量缓冲寄存器（icb[index]）和无序访问寄存器（u#也可用作数据源。这些寄存器用于提供对设备内存资源的访问，如第2章所述，除无序访问寄存器外，它们都是只读的。最后，将传递到下一管道级的计算值写入输出寄存器（o#）。当着色器程序终止时，存储在输出寄存器中的值将传递到下一阶段的输入寄存器，并在下一阶段重复该过程。其他一些特殊用途寄存器仅在某些阶段可用，因此我们将推迟到本章后面讨论。3

通常，开发人员不需要检查已编译着色器程序的程序集列表，除非存在性能问题。这使得了解程序集指令如何操作的细节不那么重要。即使如此，了解基于程序集的世界的基本知识仍然很有帮助。例如，当开发人员定义对于着色器程序的输入和输出数据结构，他们必须知道每个阶段可以使用多少输入和输出向量的限制。这取决于该特定阶段可用的输入和输出寄存器的数量。类似地，恒定缓冲区、纹理和无序访问资源的可用数量是这是非常重要的信息，在我们进行管道阶段的每一次讨论时都应该加以考虑。4

GPU架构

即使有一个严格定义的汇编语言规范，实际的GPU硬件也不需要直接实现该规范。有许多不同的体系结构实现，不同的供应商之间可能会有很大的差异。事实上，即使是来自同一供应商的连续几代GPU硬件也可能会因rom。这使得预测给定着色器程序在当前或未来GPU上的执行效率变得非常困难。根据执行该程序的GPU的体系结构，一种特定的内存访问模式可能比另一种更有效，但在不同的体系结构中，情况可能相反。

3 #符号表示有多个可用寄存器，这些寄存器由整数索引标识。例如，v0和vl是前两个可用的输入寄存器。

4有关如何编译着色器并查看其程序集列表的详细信息，请参见第6章。

124                                                                                                                                                    3.渲染管道

由于各种实现之间存在巨大的差异，因此在此处提供详细信息是不切实际的。我们邀请读者更详细地探讨这一引人入胜的主题，并在（Fatahalian）中提供一个良好的起点但是，我们仍然可以提供GPU是如何组织的一般概念，以便本书后面的讨论将有一个上下文基础。GPU已经演变成一个大规模并行处理器，容纳数百个单独的ALU处理核心。这些处理器可以运行自定义程序，并且可以访问大量内存，这允许高带宽数据传输。每个GPU都使用某种形式的内存缓存系统来减少内存请求的有效延迟，尽管缓存系统是特定于供应商的，其低级设计通常不会公开。在过去，有必要了解特定目标的各个体系结构细节这一趋势可能会持续一段时间，因为GPU仍在以非常快的速度发展。

## 3.2管道执行

固定功能阶段与可编程阶段结合在一起，提供了一个有趣且多样的管道，我们可以使用该管道渲染图像。执行管道的过程包括配置所有管道阶段状态，将输入和输出资源绑定到管道，然后调用其中一个绘制方法进行绘制执行中。所有这些任务都通过ID3DllDeviceContext接口的方法执行。为生成渲染帧而执行的管道执行次数取决于应用程序和当前场景，但我们可以假设每个帧至少执行一次管道执行。

使用这些draw方法之一调用管道后，数据将从输入内存资源读入管道的第一阶段。从输入资源读取的数据量取决于用于调用管道的draw调用的类型和参数。每一段数据都经过处理并传递到下一个阶段，直到它到达管道的末端，在那里它将被写入一个输出内存资源。处理完这些绘制调用中的所有数据后，可以将生成的输出资源（通常是2D渲染目标）显示到输出窗口或保存到文件中。

在本章中，我们将看到如何配置管道的每个阶段，以及理解每个绘制方法之间的差异。有七种不同的绘制方法可用于调用渲染管道。这里列出它们是为了便于参考，并演示有相当广泛的方法可用于调用渲染管道。

3.2管道执行125

ſŜŜŜƀ

ſŜŜŜƀ

ſŜŜŜƀ

ſŜŜŜƀ

ſŜŜŜƀ

ſŜŜŜƀ

ſŜŜŜƀ

这些方法中的每一种都指示管道以不同的方式解释其输入数据。每个方法还提供不同的参数，以进一步配置要处理的输入数据量和输入数据。一旦启动管道执行，draw方法中指定的输入数据将在被引入管道时逐段处理。

在规划将在一个管道执行期间执行的工作时，最重要的考虑因素之一是每个阶段（固定和可编程）具有不同的输入和输出类型。此外，使用这些类型的语义具有相对较大的性能影响。例如，在一个简单的渲染序列中，我们可以将四个顶点作为两个三角形提交到管道。这将导致四个顶点着色器调用，这些调用将其输出传递到光栅化器阶段，该阶段将生成大量片段。然后将这些片段传递到像素着色器，最后传递到输出合并。如果三角形靠近查看器，可能会生成许多片段，从而导致相应的大量像素着色器调用。在这种情况下，顶点的数量是恒定的，而片段的数量取决于场景的条件。也可能出现相反的情况，其中细分阶段产生可变数量的要光栅化的顶点，但对象在屏幕上保持相同的大小，因此像素数保持不变。在设计算法时必须考虑这一点，因为平衡GPU上的工作量是至关重要的。

在硬件上实际执行管道期间，每个单独的管道阶段必须在可用的GPU硬件上尽快执行其计算。在许多情况下，GPU可以在不同的处理器组中同时执行多种类型的计算。在上面的示例中，GPU可以处理第一个三角形并将其光栅化以生成片段。然后，它可以使用它的一些处理核心来处理像素着色器调用，并使用一些来执行其余的顶点着色器工作。另一方面，它可以先处理所有顶点，然后再切换到处理所有片段。不保证GPU处理数据的顺序，除非它遵守管道中呈现的逻辑顺序。

126                                                                                                                                                      3.渲染管道

### 3.2.1阶段间沟通

由于数据由各个管道级处理，因此必须从一个级传输到下一个级。为此，每个可编程阶段定义其着色器程序所需的输入和输出。我们将这些输入和输出称为属性，属性由向量变量（1到4个元素）组成。每个向量中的所有元素都由一种标量类型组成（整型和浮点型都可用）。我们还可以将这些属性称为语义或绑定语义，即为每个输入/输出属性指定的基于文本的标识符的名称。一般来说，当我们通过这些名称中的任何一个引用这些类型的阶段到阶段变量时，应该从使用它们的上下文中清楚地看到。

着色器程序的输入属性定义该程序执行所需的信息。确切地说，输入属性是着色器程序函数的输入参数。同样，函数的返回值定义其输出属性。因此，每个前一阶段必须产生与下一阶段所需输入参数相匹配的输出参数。这些输入和输出属性是通过我们在“着色器核心体系结构”一节中讨论的输入和输出寄存器实现的。清单3.1中的简单示例提供了两个着色器函数的示例声明，每个函数定义其输入和输出。请注意，第一个函数的输出与第二个函数的输入相匹配。

ɏ

Ƈ

ɪ ś Ś

ɩ ś Ś

ś ɏ Ś ƈŚ

ɏ

Ƈ

ɫ ś ɏ Ś ɫ ś Ś ƈ

ɏ ſ ɏ ƀ

Ƈ

ŵŵ ŜŜŜ

ƈ

ɫ ſ ƀ ś ɏ

Ƈ

ŵŵ ŜŜŜ

ƈ

清单3.1。两个着色器函数定义其必须匹配的输入和输出属性。

3.2输入汇编程序127

在每个函数的输入和输出声明中，使用结构定义将所有输入和输出分组在一起。输入和输出结构的每个属性都有一个关联的语义，一个文本名称，用于在阶段之间将两个数据项链接在一起。语义位于属性名称之后，后面是分号。您还可以在组件名称的左侧看到每个组件的类型声明，就像您在C/C++中看到的类似声明一样。

从本例中可以看出，第一个着色器函数的输出几乎与第二个函数的输入匹配，vertexID参数除外。此特定参数旁边列出的语义是SV\_VertexID。除了清单3.1中所示的用户定义的语义属性外，还有一组参数可供可编程着色器在其输入/输出签名中使用。这些被称为系统值语义。它们表示运行时生成或使用的属性，具体取决于它们的声明位置以及使用的系统值语义。这些系统值语义提供了在HLSL程序中使用的有用信息，还用于向管道指示所需的值。系统值始终以SV\_u作为前缀，表示它们具有特殊意义，并且不是用户定义的。我们将看到每个系统值，以及它们如何在管道的每个阶段中使用。

理解如何将数据提供给管道，并了解每个管道阶段的状态如何影响发生的处理类型，是成功使用渲染管道的关键。在以下章节中，我们将按照管道中出现的顺序详细检查每个管道阶段。

## 3.3输入汇编程序

输入汇编阶段是渲染管道中的第一站。这是一个固定的功能阶段，负责将所有顶点放在一起，这些顶点将在管道中进一步处理，因此得名。作为管道的入口点，输入汇编程序必须创建具有下一管道阶段顶点着色器所需属性的顶点。从一个或多个顶点缓冲区资源组装顶点的过程实际上可以覆盖大量不同的配置，我们将在本节后面详细介绍。该应用程序为输入汇编程序提供了使用顶点布局对象构建顶点的路线图，这也允许在创建顶点缓冲区资源时在应用程序端采用灵活的策略。图3.4突出显示了输入汇编程序在管道中的位置。

除了构造输入顶点外，输入汇编器还通过指定正在渲染的几何体的拓扑来确定这些顶点之间的连接方式。这将标识定义的基本体或控制点的类型

128                                                                                                                                                      3.渲染管道

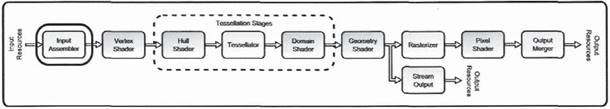


图3.4。输入汇编阶段。

如何在管道中稍后将各个顶点分组在一起。通过指定正在渲染的几何体的拓扑，输入汇编程序可以对管道其余部分如何解释同一组输入顶点产生重大影响。如果一个几何对象由三角形组成，但它被渲染为一组点，那么即使输入顶点缓冲区完全相同，结果渲染也会完全不同。

输入汇编程序也是启用程序，允许渲染管道执行几种不同类型的渲染操作。渲染管道支持标准绘图调用、索引绘图、实例化绘图、间接绘图以及这些不同操作的多种组合，所有这些操作都需要不同的输入资源配置。输入汇编程序与它的输入资源和draw调用中提供的信息一起工作，将这些不同的输入配置转换为管道其余部分可用的格式。在许多情况下，输入汇编程序是针对不同的绘图调用执行不同操作的唯一阶段。它对管道的其余部分有效地隐藏了数据提交细节，并以一致的格式提供其输出数据。由于输入汇编程序是应用程序模型数据和渲染管道之间的主要链接，因此它起着非常重要的作用。我们将更详细地研究这个管道阶段，以准确地了解如何正确地使用它，以及可以使用哪些选项来调用管道执行。

### 3.3.1输入汇编程序管道输入

在将数据传递到管道之前，输入汇编程序必须连接到适当的资源以获取输入几何数据。该数据由顶点数据和索引数据组成，它们都以缓冲区资源的形式提供给管道。以下两部分描述如何将这些缓冲区资源绑定到输入汇编程序。

顶点缓冲区使用

输入汇编器的主要职责是正确地组装顶点，以便在管道中进一步使用，因此需要从应用程序访问顶点数据。该数据存储在一个或多个顶点缓冲区资源中，可以以各种不同的方式组织

3.3输入汇编程序129

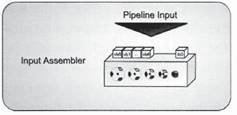


图3,5。输入汇编程序的可用输入插槽概述。

布局。从概念上讲，我们可以将输入汇编程序视为具有多个输入插槽，这些插槽可以用顶点缓冲区资源填充。图3.5显示了这种可视化。

每个插槽都可以使用包含完整顶点的一个或多个属性的顶点缓冲区资源填充。例如，一个顶点缓冲区可以包含顶点的位置数据，第二个顶点缓冲区可以包含所有法向量数据。也可以在结构数组中的单个顶点缓冲区中提供所有顶点数据。开发人员可以选择如何最好地组织输入数据。当前有16个可供应用程序配置的输入插槽，这允许为所需的顶点数据提供相当灵活的存储选项。使用单独存储的各个属性，可以动态决定特定渲染需要哪些顶点组件。通过消除顶点中未使用的属性，这可能会减少读取顶点数据所需的带宽。我们将在本章后面讨论如何在多个缓冲区中存储各种数据的细节。

为了将顶点缓冲区绑定到输入汇编阶段，我们使用设备上下文接口。正如我们在第1章中看到的，设备上下文是完整管道的主要接口。使用ID3D11 DeviceContext:：IASetVertexBuffers方法将多个顶点缓冲区绑定到输入汇编程序的过程如清单3.2所示。在调用set vertex buffer方法之前，列表中声明的每个变量都将填充相应的缓冲区引用、步长和偏移量。

Ś Ś

ɪƋ ƃ ɪ ɨ ɨ ɏ ɏ ɏ ɏ ɥ ɏ ɥ ɏ ɥ Ƅ Ś ƃ ɪ ɏ ɏ ɏ ɏ ɏ ɏ Ƅ Ś ɥ ƃ ɪ ɏ ɏ ɏ ɏ ɏ ɏ Ƅ Ś

ŵ ŵ Ŝ Ŝ Ŝ

Ş ʴ ſ ř ř ř ř ƀ Ś

清单3.2。如何将多个顶点缓冲区绑定到输入汇编程序阶段。

130                                                                                                                                                      3.渲染管道

StartSlot和NumBuffers参数中指定了要开始绑定到的第一个输入汇编程序顶点缓冲区插槽的索引以及应绑定的后续缓冲区插槽的数量。顶点缓冲区资源指针列表必须组合成一个连续数组，并在ppVertexBuffers参数中传递。最后两个参数pStrides和poffset允许应用程序指定每个顶点缓冲区的逐顶点步长，以及开始使用每个顶点缓冲区到所需位置的偏移量。这些数组元素中的每一个都将开始在StartSlot索引中使用，并将用于填充NumBuff ens数量的插槽。这意味着数组的大小和填充必须适当，否则函数将访问无效的内存位置。此外，要将顶点缓冲区绑定到输入汇编程序，不能绑定缓冲区以在管道中的另一个位置写入。这是一个合理的要求，因为它会导致看似不可预测的结果，同时读取和写入缓冲区。

索引缓冲区使用情况

当单独获取顶点数据时，输入汇编程序将按照顶点数据在顶点缓冲区中出现的顺序对其进行解释。这意味着，如果顶点数据用于渲染三角形，则前三个顶点定义第一个三角形，然后是下一个三角形的下三个顶点，依此类推。在两个相邻三角形实际引用同一顶点的情况下，这可能会导致重复的顶点数据。事实上，大多数三角形网格通常都有大量的相邻三角形。在顶点缓冲区中多次指定单个顶点数据时，可能会浪费大量内存。这种低效率可以通过使用索引缓冲区来消除。索引缓冲区用于通过向顶点缓冲区提供索引偏移来定义每个图元的顶点顺序。在这种情况下，每个顶点可以指定一次，然后指向它的索引可以用于它所属的任何三角形。使用索引渲染时，三角形中要使用的三个顶点由索引缓冲区中的三个连续索引指定。5

要使用索引呈现，应用程序必须将索引缓冲区绑定到输入汇编程序。只有一个插槽可用于绑定索引缓冲区，并且它仅在一个索引绘制调用期间用于指定用于生成基本体的顶点顺序。要将索引缓冲区绑定到输入汇编程序，应用程序必须使用ID3DllDeviceContext:：IASetIndexBuffer方法。清单3.3演示了如何执行此绑定操作。该方法提供了三个简单的参数：指向索引缓冲区的指针、索引存储的格式（16位或32位无符号整数），最后是索引缓冲区中的偏移量，以便从中开始构建原语。

5 顶点和索引的顺序因不同的基本体类型而异。本节后面将更详细地描述这些。

3.3输入汇编程序131

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ID3DllBuffer\* | | | 布弗 | | = 0; |  |  |
| 单位偏移量= | | | 0; |  |  |  |  |
| // | 设置 | 布弗 | 到 | 这个 | 渴望的 | 指数 | 此处为缓冲区引用，以及 |
| // | 设置 | 偏移量 | | 到 | 适当的 | | 缓冲区中的位置。 |
| m\_pContext->IASetIndexBuffer( | | | | | | pBuffer、DXGI格式、R32单元、偏移量）； | |
|  |  |  |  |  |  |  |  |

清单3.3。如何将索引缓冲区绑定到输入汇编程序阶段。

顶点或索引缓冲区绑定到管道后，可以通过将空指针值设置到相应的输入槽来解除绑定。记住这一点很重要，尤其是在draw调用之间重新配置管道时。如果在以前的绘图调用中使用了多顶点缓冲区配置，而新的绘图调用使用的顶点缓冲区较少，则未使用的插槽应填充NULL以解除未使用缓冲区的绑定。这就是为什么顶点缓冲区绑定方法总是使用一个大小为顶点缓冲区最大数量的数组，并在填充所需配置之前将其初始化为空值。

### 3.3.2输入汇编程序状态配置

将适当的资源绑定到顶点和索引缓冲区插槽后，在使用输入汇编程序之前，必须在输入汇编程序中设置另外两个配置。第一个是Input Layout对象，输入汇编程序使用该对象了解从哪个输入槽读取逐顶点数据以构建完整的顶点。第二个参数是基本体拓扑，输入汇编程序应使用该拓扑来确定顶点如何组合成基本体。这两种状态与用于执行管道的绘制方法相结合，确定管道如何解释顶点和基本体数据。我们将详细探讨每一个状态，然后考虑它们如何与可用的绘图方法交互以产生不同的输入配置。

输入布局

输入布局对象可以看作是一个配方，告诉输入汇编程序如何创建顶点。每个顶点由向量属性集合组成，每个属性最多有四个组件。由于多达16个顶点缓冲区可用于绑定，输入汇编程序需要知道从何处读取这些组件，以及了解将它们放入最终组装顶点的顺序。输入布局对象将此信息提供给输入汇编程序。

要创建一个输入布局对象，应用程序必须创建一个D3D11\_uuinput\_E element\_DESC结构数组，其中所需顶点的每个组件都有一个结构

132                                                                                                                                                     3.渲染管道

配置D3D11\_INPUT\_ELEMENT\_DESC的成员如清单3.4所示。我们将检查每个结构成员代表什么，以及它们如何定义输入汇编程序执行其工作所需的信息。

结构D3D11\u输入\u元素\u描述{

LPCSTR语义名称；

UINT语义索引；

DXGI\_格式；

单元输入槽；

UINT对齐字节偏移量；

D3D11\u输入\u分类输入SLOTCLASS；

UINT瞬时数据速率；

}

清单3.4。D3D11输入元素描述结构成员。

**逐顶点元素。**第一个参数SemanticName标识顶点属性的文本名称，该名称必须与顶点着色器程序中提供的相应名称匹配。顶点着色器的每个输入必须在其HLSL源代码中定义语义。该语义名称用于将顶点着色器输入与输入汇编程序提供的顶点数据相匹配。Semanticlndex是一个整数，它允许SemanticName被多次使用。例如，如果在单个顶点布局中使用多组纹理坐标，则每组坐标可以使用相同的SemanticName，但会使用增加的Semanticlndex值来区分它们。

下一个结构成员是组件的格式。这将指定什么数据类型，以及属性中包含多少元素。可用的格式范围从1到4个元素，浮点和整数类型都可用。下一个参数是InputSlot，它指示应从16个顶点缓冲槽中读取此组件的数据。AlignedByteOffset表示此D3D11\_INPUT\_element\_DESC描述的项到顶点缓冲区中第一个元素的偏移量。这告诉输入汇编程序顶点缓冲区中从何处开始读取输入数据。

**每个实例元素。**最后两个结构成员声明了有关实例化的顶点功能。实例化绘制方法基本上使用单个绘制调用将模型提交到管道。然后多次“实例化”此模型，但在每个实例级别（而不是每个顶点级别）指定顶点数据的子集。当同一对象的多个副本必须在整个场景的不同位置渲染时，经常使用此选项。顶点格式将在其定义中包含一个变换矩阵，对于模型的每个实例，变换矩阵仅递增到下一个值。然后在顶点中提供模型所有实例的变换矩阵

3.3输入汇编程序133

缓冲区并绑定到输入汇编程序。执行绘制调用时，顶点数据一次提交给顶点着色器一个副本，并在模型实例之间更新变换矩阵。

如果顶点组件提供每个实例的数据，则必须使用InputSlotClass参数指定该数据。在这种情况下，还可以选择仅在向管道提交一定数量的实例后，才递增到顶点缓冲区中的下一个元素。这在InstanceDataStepRate中指定，并为每个实例参数的更改频率提供粗略控制。由于这是在单个属性级别指定的，因此可以使用不同的属性以不同的速率前进到下一个数据成员。识别所有所需的顶点属性并填写描述结构数组后，应用程序可以使用ID3DllDevice:：CreateInputl\_ayout（）方法创建ID3DllInputLayout对象。清单3.5演示了如何执行此过程。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| // | 创建数组 | | 这里有很多元素 | 对于 | 美国石油学会。 | | |  |
| D3D11输入元素描述 | | | | | = | 新的D3D11输入元素描述元素。 | | |
|  | c o u n t（）]； | |  |  |  |  |  |  |
| // | 在 | 元素数组 | | (t h i s) | | w i l | 视情况而定 | 他们是怎样的 |
| // | 提交 | 到 | t h i s方法）。 |  |  |  |  |  |
| 因为= | | 0; | i<elements.count（）； | | | 我++ | ) |  |
|  | Pelments[i] | | =元素[i]j； |  |  |  |  |  |
| // | 尝试从输入信息创建输入布局（ | | | | | | | |
| // | 编译着色器字节码存储 | | | | w i l l也随时间而变化 | | | 数据 |
| // | 结构 | 使用）。 | |  |  |  |  |  |

ID3DBlob\*pCompiledShader=m\_vShaders[ShaderID]->pCompiledShader；ID3DLLInputOut\*播放=0；

*二,*创建输入布局对象，然后检查HRESULT。HRESULT hr=m\_pDevice->CreateInputLayout（元素，元素.计数（），

pCompiledShader->GetBufferPointer（），pCompiledShader->GetBufferSize（），

&播放）；

清单3.5。创建输入布局对象。

这里我们可以看到D3D11\_INPUT\_ELEMENT\_DESC结构的数组被分配，然后从输入容器对象填充。ID3D11 Device:：CreateInputLayout（）的前两个参数提供描述数组和数组中存在的元素数。第三个和第四个参数分别是指向编译顶点着色器源代码所产生的着色器字节代码的指针以及该字节代码的大小。这用于将元素描述的输入数组与将用于向其提供数据的HLSL着色器进行比较。比较这两个对象

134                                                                                                                                                      3.渲染管道

当着色器与特定输入布局一起使用时，创建输入布局可降低运行时所需的验证量。

本原拓扑

必须在输入汇编程序中设置的最终状态是基本拓扑。我们可以将输入汇编程序的工作视为两个任务：首先，它必须使用输入布局对象从提供的输入资源中组装顶点；其次，它必须将顶点流组织成一个基本体流。每个基本体使用一小组连续的顶点（其顺序由它们在顶点缓冲区中的出现顺序或索引渲染的索引缓冲区中的索引顺序决定）来定义组成它的组件。基本拓扑的常见示例有三角形列表、三角形带、线列表和线带。在三角形列表的示例中，将为顶点流中的每三个顶点创建一个基本体。

基本体拓扑设置告诉输入汇编程序如何从组装的顶点流构建各种基本体。每个基本体中使用的顶点数量，以及如何从组合的顶点流中选择顶点，是基本体拓扑设置的函数。清单3.6中提供了可用的基元类型，并演示了如何使用ID3DllDeviceContext:：IASetPrimitiveTopology（）方法设置基元拓扑。

枚举D3D11\_原语\_T0P0L0GY{

D3Dll\_原语\_T0POL0GY\_未定义，

D3D11\_原语\_T0P0L0GY\_P0INTLIST，

D3D11\_原语\_T0P0L0GY\_行列表，

D3D11\_原语\_T0P0L0GY\_线条带，

D3D11\_原语\_T0P0L0GY\_三角形列表，

D3D11\_原语\_T0P0L0GY\_三角形条带，

D3D11\_原语\_T0P0L0GY\_行列表\_调整，

D3D11原语T0P0L0GY线条带调整，

D3D11原语T0P0L0GY三角形列表ADJ，

D3D11\_原语\_T0P0L0GY\_三角形trip\_ADJ，

D3D11原语T0P00GY 1 NTR0L P0INT补丁列表，D3D11原语拓扑图控制点补丁列表，D3D11原语拓扑图控制点补丁列表，D3D11原语拓扑图控制点补丁列表，D3D11原语拓扑图控制点补丁列表，D3D11原语拓扑图控制点补丁列表，D3D11原语拓扑图控制点补丁列表\_P0INT\U补丁列表、D3D11\U基元T0P00GY\U 8\U C0NTR0L\U P0INT\U补丁列表、D3D11\U基元T0P00GY\U 9\U C0NTR0L\U P0INT\U补丁列表、D3D11\U基元拓扑10\U控制点补丁列表、D3D11\U基元T0P00GY\U 11\U C0NTR0L\U p0T补丁列表、D3D11\U基元拓扑12\U控制点补丁列表、D3D11\U基元拓扑\_T0P0L0GY\U 14\U C0NTR0L\U P0INT\U配线架，

3.3输入汇编程序135

D3Dll\U基元\U拓扑\U 15\U控制点\U补丁列表，

D3Dll\U基元\U拓扑\U 16\U控制点\U补丁列表，

D3Dll\U基元\U拓扑\U 17\U控制点\U补丁列表，

D3D11\_基本\_拓扑\_18\_控制\_点\_补丁列表，

D3Dll\u基本\u拓扑\u 19\u控制\u点\u补丁列表，

D3D11\_基本\_拓扑\_20\_控制\_点\_补丁列表，

D3Dll\U基元\U拓扑\U 21\U控制点\U补丁列表，

D3D11\_基本\_拓扑\_22\_控制\_点\_补丁列表，

D3D11\_基本\_拓扑\_23\_控制\_点\_补丁列表，

D3D11\_原语\_T0P0L0GY\_24\_C0NTR0L\_P0INT\_补丁列表，

D3Dll\u基本\u拓扑\u 25\u控制\u点\u补丁列表，

D3Dll\_基本\_拓扑\_26\_控制\_点\_补丁列表，

D3Dll\_基本\_拓扑\_27\_控制\_点\_补丁列表，

D3D11\_基本\_拓扑\_28\_控制\_点\_补丁列表，

D3D11\_基本\_拓扑\_29\_控制\_点\_补丁列表，

D3D11\_基本\_拓扑\_30\_控制\_点\_补丁列表，

D3D11\_基本\_拓扑\_31\_控制\_点\_补丁列表，

D3Dll\U基元\U拓扑\U 32\U控制\U点\U补丁列表

}

//    指定m\_pContext->iSetPrimitiveTopology所使用的几何图形类型(

我们会做交易的。

primType）；

清单3.6。可用的基本拓扑类型，以及如何使用设备上下文接口设置拓扑类型的示例。

如清单3.6所示，有多种可用的基本类型。如果您以前使用过Direct3D 10，则列表中的前九个条目看起来应该很熟悉，因为它们在Direct3D 11之前就已经使用过了。但是，列表的其余部分提供了一系列不同的控制点修补程序列表，每个列表中包含的控制点修补程序数量不同，最多32个。引入这些基本类型是为了支持Direct3D 11中引入的新细分阶段。为了更好地理解所有这些基本拓扑如何组织顶点流，我们将创建一个具有有序顶点序列的示例，然后检查每个拓扑类型如何使用该顶点流创建基本拓扑。图3.6显示了顶点的样本集，每个顶点根据其在顶点流中的位置进行编号。

**点原语。**点列表基本体拓扑是所有基本体类型中最简单的。顶点被分组为单顶点基本体，这意味着顶点流产生大小相等的基本体流。因此，原语的输出组看起来与图3.6中所示的相同。

**行原语。**“线列表”基本体拓扑稍微复杂一些，每对顶点都会生成一个“线”基本体。这本质上表明，对于要渲染的每条线，包含一条线的两个顶点必须包含在顶点缓冲区中。“线带基本体”拓扑提供了线基本体更紧凑的表示形式，

136                                                                                                                                                      3.渲染管道

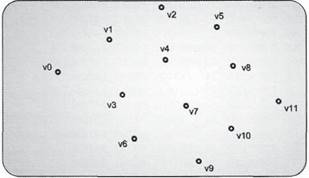


图3.6。用于创建各种基本体类型的多个顶点。

但它只能表示一个连接的行列表。输入汇编程序从顶点流中的前两个顶点生成第一行。前两个之后的每个顶点定义一条新线，其中该线的另一个顶点是流中的上一个顶点。这在要绘制的线端到端连接的情况下提供了更密集的顶点表示。这两种基本拓扑如图3.7所示。

**三角形原语。**标准三角形基本拓扑遵循与直线基本拓扑相同的范例。三角形列表拓扑从顶点流中的每三个顶点创建一个三角形基本体。“三角形带”从流中的前三个顶点创建三角形基本体，然后为每个后续顶点创建新的三角形基本体，使用流中的前两个顶点定义三角形的其余部分。与线带基本体一样，三角形带基本体拓扑只能表示三角形的连接“带”。这些基本拓扑及其顺序如图3.8所示。

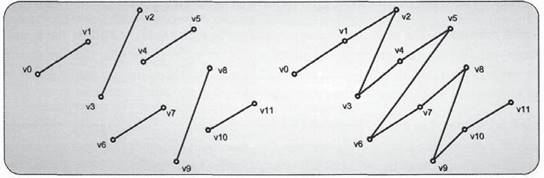


图3.7，从输入顶点创建的线列表和线带原语。

|  |  |
| --- | --- |
| 3.3输入汇编程序 | 137 |

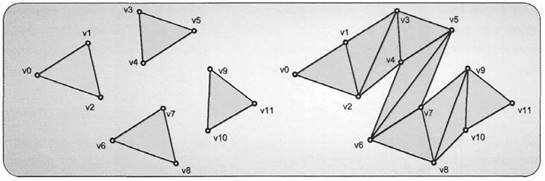


图3.8。从输入顶点创建的三角形列表和三角形条原语。

具有邻接关系的基元。在某些情况下，希望在管道中使用相邻的基本信息。这意味着每个原语都将被提供给管道，并且可以访问紧邻它的原语。有四种不同的基本拓扑提供邻接信息：具有邻接的线列表、具有邻接的线带、具有邻接的三角形列表和具有邻接的三角形带。这些表示为管道提供了更多的信息，以便在原语级别进行处理，但每个原语的内存和带宽成本较高。图3.9和3.10显示了这些基本拓扑中的每一种。

控制点原语。为了便于在细分管道中使用高阶基本体，提供了其他基本体拓扑选择。每个数量的控制点有一个基本类型，范围从1到32。这提供了执行大量不同控制补丁方案的能力。图3.11显示了一个样本控制补丁原语。

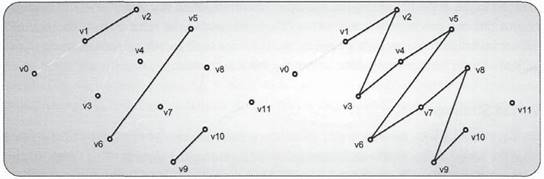


图3.9。从输入顶点创建具有邻接关系的线列表和线带基本体。

138                                                                                                                                                     3.渲染管道

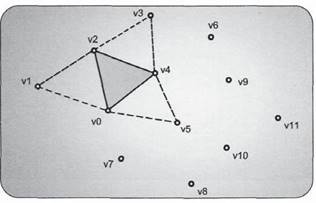


图3.10。三角形列表基本体，具有从输入顶点创建的邻接。

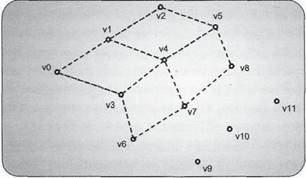


图3.11。控制从输入顶点创建的面片基本体。

### 3.3.3输入汇编程序阶段处理

对于输入汇编程序的所有可用设置，有各种各样的配置，可以生成不同的输出数据，以便在管道的其余部分中使用。在研究这些配置产生的内容之前，我们将更仔细地了解输入汇编程序中正在做什么。

顶点流

我们已经了解了输入汇编程序如何通过从绑定顶点缓冲区读取数据并将其组合到各个顶点中来创建顶点流。此顶点流以两个有序序列之一生成顶点。顶点要么按其顶点缓冲区中的现有顺序保留，要么按指定顺序重新排列

3.3输入汇编程序139

通过索引缓冲区中的索引。使用哪个顺序的确定来自用于调用管道执行的draw调用的类型。无论排序来自何处，输入汇编程序都会生成一个顶点流来表示输入几何体。

原始河流

顶点流进一步细化为基本体流。管道的各个部分旨在仅对顶点（顶点着色器）、顶点和基本体（外壳着色器和域着色器）或完整基本体（几何体着色器）进行操作。在每种情况下，原语流都由选定的原语拓扑和用于启动管道执行的绘制调用类型决定。

绘制调用效果

那么，绘制调用产生的顶点和基本体流究竟如何受到所用调用类型的影响呢？这可以通过检查每种类型的绘图调用的输入汇编程序的输出来最好地证明。以下各节提供了有关每个绘制调用类的信息，以及它们对结果数据流的影响。

**标准绘图方法。**要考虑的最简单的绘制方法是DRAW（）和DrawAuto（）方法。这两种方法都会触发输入汇编器根据其输入布局设置来组装顶点。这将创建一个顶点流，然后将其传递到管道中。这些绘制调用生成的基本体信息由输入汇编程序中当前指定的基本体拓扑确定，基本体的构造取决于顶点缓冲区中顶点的顺序。这个过程可以看作是基本顶点和基元的构造过程。

**索引绘制方法。**基本绘制调用的第一个变体是添加索引渲染。正如我们已经讨论过的，索引渲染使用索引缓冲区的索引来确定哪些顶点用于构造基本体，而不是简单地使用顶点缓冲区中的顶点顺序。顶点流保持不变，但其中的顶点由索引缓冲区的索引选择。此外，基元流使用索引缓冲区顺序而不是顶点缓冲区顺序创建其基元。有几个draw调用支持索引呈现，包括DrawIndexed（）、DrawIndexedlnstanced（）和DrawIndexedlnstancedlndirect（）。

**实例绘制方法。**除了索引渲染，Direct3D 11还允许实例化渲染。在描述D3D11\_INPUT\_ELEMENT\_DESC结构的可用成员时，我们简要讨论了实例渲染。如果顶点组件声明为逐实例组件，并且使用其中一个实例调用管道

140                                                                                                                                                     3.渲染管道

在绘制方法中，顶点和基本体流的创建方式与基本渲染和索引渲染的创建方式基本相同。但是，顶点和基本体流对于对象的每个实例都是重复的，但每个完整实例的逐实例顶点组件都会更新。实例的数量由传递给draw调用的参数确定，每个实例的顶点组件从绑定到输入汇编程序的一个或多个顶点缓冲区中获取。

总的来说，使用实例渲染方法的结果是顶点和基本体流乘以正在渲染的实例数。有四个不同的实例绘制调用：DrawInstanced（）、DrawIndexedlnstanced（）、DrawInstancedlndirectQ和Drawlndexedlnstancedlndirect（）。

间接绘制方法。除了基本、索引和实例渲染方法外，还有另一种类型的绘制调用：间接渲染。这种技术实际上并不修改顶点和基本体流，但这里应该讨论它，以获得可用绘制方法的完整视图。相反，间接呈现方法允许将缓冲区资源作为绘制调用参数传递。缓冲区包含通常由应用程序传递的所需输入信息。间接渲染方法是DrawInstancedlndirectQ和DrawIndexedlnstancedlndirect（）。例如，Drawlnstancedlndirect（）方法的顶点数、起始顶点数、实例数和起始实例数将包含在缓冲区资源中，而不是直接传递给函数。

这些调用的目的是允许GPU填充缓冲区，然后缓冲区可以控制如何执行绘制序列。这将draw调用的控制转移到GPU而不是应用程序，并提供了使GPU在其操作中更加自治的第一步。但是，间接渲染操作不会修改顶点和基本体流的构造，它们只会修改将绘制方法参数传递到运行时的方式。

混合渲染绘制方法。正如我们所看到的，每一类渲染操作并不是相互排斥的。绘制方法的名称通常包括几种渲染技术，并提供多种不同的方式来执行管道。在每种混合情况下，生成的顶点和基本体流都是独立渲染类型的混合。

### 3.3.4输入汇编程序管道输出

用户定义的属性

随着理解输入汇编器可以产生什么，以及如何操作它的输出，我们需要考虑输出流如何与管道的其余部分交互。用于输入汇编程序生成与所需

3.3输入汇编程序141

顶点着色器，用于创建输入布局对象的D3D11\_INPUT\_ELEMENT\_DESC结构数组必须与顶点着色器声明的输入签名匹配。因此，各个顶点属性必须与顶点着色器所期望的匹配，包括数据类型和语义名称。清单3.7显示了HLSL中的顶点着色器程序示例。

s t r u c t VS\_输入

{

浮动位置3：位置；

浮动2 tex:TEXCOORDS；

3正常：正常；

};

VS\_0输出VSMAIN（在VS\_输入v中）

{

}

清单3.7。顶点着色器输入声明示例。

对于本例，应用程序将需要提供顶点数据，该数据由具有三个浮点元素的位置语义属性、具有两个浮点元素的TEXCOORDS语义组件和具有三个浮点元素的普通语义组件组成。创建输入布局对象有助于开发人员确保顶点数据和顶点着色器签名彼此匹配。由于顶点数据是由应用程序提供的，因此可以使用与特定顶点着色器一起使用的正确格式创建或加载顶点数据。然后顶点着色器使用这些输入来执行自己的计算，然后它还将定义一些输出属性，并将这些属性写出以供下一个活动管道阶段使用。这个过程对每个后续的管道阶段重复，每个阶段声明其输出签名，然后在其中提供其输出数据。输入汇编程序生成的输入数据的一些典型示例包括但不限于以下内容：

浮动3定位输入顶点的位置

float3法线输入顶点的法向量

float 3切线输入顶点的切线向量

float 3 bitangent-输入顶点的bitangent向量

uint4 BoneId用于蒙皮的顶点的四个骨骼ID

float4骨骼权重用于蒙皮顶点的四个骨骼权重

142                                                                                                                                                     3.渲染管道

输入汇编程序还可以生成三种不同的系统值语义：SV\_VertexID、SV\_PrimitiveID和SV\_InstanceID。这些系统值语义不是由用户提供的，而是由输入汇编程序在创建输出数据流时生成的。SV\_VertexID系统值是一个无符号整数，它唯一标识集合顶点流中的每个顶点。它首先在顶点着色器中可用，并提供了一种稍后在管道中区分顶点的简单方法。SV\_原语ID为原语流中的每个原语提供类似的唯一标识符。它首先在外壳着色器阶段可用，因为顶点着色器不使用基本体级别信息。最后，SV\_InstanceID在实例化绘制调用中唯一标识几何体的每个实例。它首先可用于顶点着色器阶段。这三个系统值一起提供了一种相当广泛的方法来识别输入汇编程序生成的每个数据元素。

## 3.4顶点着色器

渲染管道中的第一个可编程着色器阶段是顶点着色器。如上所述，可编程着色器阶段执行用HTSL编写的自定义函数。在顶点着色器阶段的情况下，顶点着色器程序是一个函数，对于输入汇编程序生成的顶点流中的每个顶点调用一次。每个输入顶点都作为顶点着色器程序的参数接收，处理后的顶点作为函数的结果返回。每个顶点着色器调用都是在与其他调用完全隔离的情况下执行的，它们之间不可能进行通信。由于对每个输入顶点执行一次，因此将存在从顶点着色器阶段到输出顶点的输入顶点的一对一映射。管道中顶点着色器的位置在图3.12中高亮显示。

此外，顶点着色器的每次调用都不知道输入汇编程序生成的基元流。它的唯一目的是处理顶点数据，并将任何更高级别的处理留给管道下游的各个阶段。在不了解基本体的情况下，顶点着色器对所有不同拓扑类型的操作方式相同。无论是否正在处理作为点、线提交的几何图形，

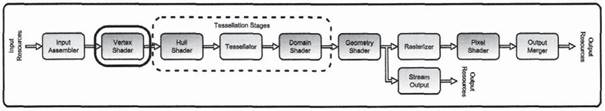


图3.12。顶点着色器阶段。

3.4顶点着色器143

三角形或面片、各个顶点都由顶点着色器进行相同的处理。顶点着色器的处理输出与管道中稍后由输入汇编程序阶段生成的基本体信息相结合。这简化了顶点着色器阶段的工作，允许它执行完全相同的操作，而不管基本体类型如何。

顶点着色器位于输入汇编程序阶段和外壳着色器阶段之间。这意味着它将在几何体数据插入管道之后以及在执行任何细分之前立即处理几何体数据。通常在顶点着色器中执行的一些操作示例包括将变换矩阵应用于输入几何体、执行基于骨骼的动画变换（也称为顶点蒙皮）以及执行逐顶点照明计算。在本节的其余部分中，我们将更精确地研究顶点着色器阶段通常用于什么，以及它有哪些工具可供使用。我们首先考虑顶点着色器接收什么数据作为输入，然后处理它执行的处理类型，然后最后考虑它产生什么数据作为输出。6

### 3.4.1顶点着色器管道输入

由于顶点着色器直接位于管道中输入汇编程序的后面，因此它自然会从那里接收输入。配置输入汇编程序的输入布局所做的所有工作都旨在使创建的顶点与顶点着色器阶段的当前程序所期望的格式相匹配。这就是为什么在创建ID3DllInputl\_布局对象时需要编译着色器字节代码作为输入，以确保组装的顶点与执行顶点着色器程序所需的匹配。

如输入汇编器输出的描述所示，顶点着色器输入除了使用集合顶点的属性外，还可以使用系统值语义。可作为顶点着色器输入的两个系统值语义是SV\_VertexID和SV\_InstanceID。这两者都为顶点着色器程序提供了有用的信息，开发人员只需将它们包含在顶点着色器输入签名中即可获取它们。例如，SV\_VertexID可以用作查找表的索引，用于向着色器提供伪随机值。类似地，SV\_InstanceID也可以用作索引机制。但是，它只针对输入汇编程序生成的每个实例进行更新，因此任何查找表值都将统一应用于实例的所有顶点。这可用于在网格实例之间引入变化，使每个实例看起来更独特。

我们已经描述了如何将这些系统值语义作为输入提供给顶点着色器。它们也可以提供给管道中的后期，但是

6顶点蒙皮在第8章“网格渲染”中有详细描述

144                                                                                                                                                     3.渲染管道

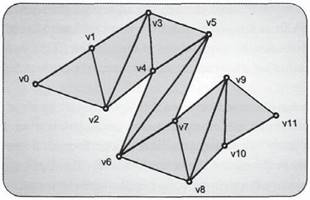


图3.13。属于多个基本体的三角形条带顶点的描述。

如果以后要使用，则必须通过顶点着色器输出。这是一个合理的限制，如果系统值是在输入汇编程序中生成的，则它们直接可用于顶点着色器。如果该着色器决定不将系统值沿管道进一步传播，则它们不会简单地在需要它们的后续管道阶段重新出现。开发人员必须确保以后需要的任何系统值语义在使用之前从一个阶段传递到另一个阶段。

顶点着色器不可用的一个系统值语义是SV\_uuPrimitiveId属性。SV\_PrimitiveID由输入汇编程序生成，并为原语流中的每个原语提供标识符。顶点由顶点着色器阶段一次处理一个顶点，而与渲染的基本体类型无关。此外，顶点着色器生成的单个经过处理的输出顶点可以在多个基本体中使用，例如在三角形条带中，或者如我们讨论的索引渲染中所示。这使得为输入顶点属性拥有唯一的基元ID不合逻辑，因为它可以在多个基元中使用。图3.13显示了作为三角形条带提交的一组非常简单的几何图形，突出显示了这种情况。

当管道处理此类几何体时，顶点着色器阶段将单独处理每个顶点。如果顶点着色器中允许SV\_原语ID，则哪个原语索引将分配给顶点v37，它属于多个原语，因此共享顶点应使用哪个原语ID并不清楚。

### 3.4.2顶点着色器状态配置

作为可编程着色器阶段，顶点着色器实现公共着色器核心功能。这意味着它提供了一组标准的资源接口方法，允许应用程序向着色器程序提供对所需资源的访问。与所有管道操作一样，所有可以更改顶点着色器阶段的

3.4顶点着色器145

状态属于ID3DllDeviceContext接口。我们将回顾这些可用资源中的每一个，并了解它们如何在顶点着色器阶段的上下文中使用。

着色器程序

顶点着色器阶段的第一个，可能也是最重要的状态是着色器程序本身。编译着色器并从中创建着色器对象的过程在第6章中有详细描述，因此我们将在这里假设编译的着色器已用于创建有效的顶点着色器对象。应用程序可以并且通常会创建多个顶点着色器对象，用于渲染各种对象。要使用所需的着色器程序配置顶点着色器阶段，应用程序必须使用ID3DllDevice Context:：VSSetShader方法。清单3.8显示了如何执行此操作的示例。

//pNextShader必须与所需的着色器对象保持一致。

ID3DllVer texShader\*pNextShader=pShaderl；

m\_pCorttext->VSSetShader（pNextShader，0,0）；

清单3.8。设置顶点着色器程序。

设置所需着色器对象的方法需要三个参数。第一个是指向要绑定到舞台的着色器对象的指针。第二个和第三个参数用于传递要在着色器程序中使用的着色器类实例数组。这些类实例是可用于为着色器程序的一部分提供外部对象的对象，旨在减少支持各种类似渲染场景所需的单个着色器对象的数量。还有一个对应的ID3DllDeviceContext7

：：VSGetShader方法，可用于检索当前绑定到顶点着色器阶段的着色器和类实例对象。这对于读取现有着色器对象设置非常有用，以便在执行临时操作后恢复该设置。

恒定缓冲区

应用程序可以提供给顶点着色器程序的下一个配置是常量缓冲区数组。第2章详细介绍了常数缓冲区。这些基本上是缓冲区资源，应用程序使用着色器程序可用的参数加载这些资源，着色器程序可以直接在HLSL代码中使用数据。这用于将参数传递给着色器程序，该程序将在整个管道执行过程中保持不变，因此称为常量缓冲区。第2章还详细描述了用数据填充常量缓冲区的过程，因此我们将在这里重点介绍

7 第6章简要描述了这些类实例，但没有详细讨论。读者可以参考DXSDK文档，以获得有关如何使用这些对象的进一步指导。

146                                                                                                                                                     3.渲染管道

如何将这些资源绑定到顶点着色器阶段。清单3.9显示了如何将一个或多个常量缓冲区绑定到顶点着色器阶段。

ID3DllBuffer\*cbuffePS[D3Dll\_COMMONSHADER\_CONSTANT\_BUFFER\_API\_SLOT\_COUNT]；

//  在这里用缓冲区pContext->VSSetConstantBuffers（0计数，cbuffers）填充数组的每个元素；*,*



清单3.9。将常量缓冲区数组绑定到顶点着色器阶段。

清单3.9显示了如何使用单个设备上下文方法设置多个常量缓冲区。实际上，这个例子展示了如何设置所有可用的常量缓冲区插槽。如果ID3D11Buffer\*数组的所有元素都初始化为NULL，则这是一种有利的技术。然后，当NULL值被绑定到其位置时，先前附加到此阶段的任何常量缓冲区将自动分离。图3.14描述了“完成”状态设置前后顶点着色器中的常量缓冲区插槽。当前在顶点着色器阶段中设置的常量缓冲区指针数组也可以使用相应的ID3D11DeviceContext:：VSGetConstant检索

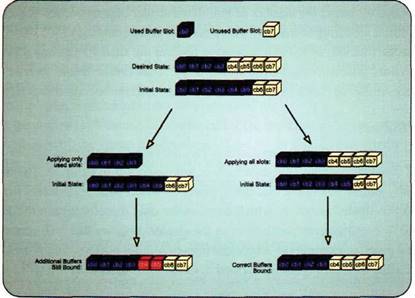


图3.14。使用初始化为NULL的缓冲区资源指针数组解除未使用常量缓冲区的绑定。

|  |  |
| --- | --- |
| 3.4顶点着色器 | 147 |

缓冲区方法。同样，这对于在执行临时操作后恢复现有设置非常有用。

着色器资源视图

要使顶点着色器程序能够访问其可用的各种类型的只读资源，必须使用着色器资源视图将资源绑定到顶点着色器阶段。缓冲区资源和纹理资源以完全相同的方式绑定到管道，使用ID3D11DeviceContext:：VSSetShaderResources方法。与常量缓冲区一样，可以使用相同的方法调用同时设置一个或多个着色器资源视图。清单3.10演示了如何使用此方法。

，&apos;&apos;OO6KDGHU5HVRXUFH9LHZ

6KDGHU5HVRXUFH9LHZV>“B&200216+$”（5B，1387B5（6285和（B6/27B&2817@

)我的工作是超高VrXUFH YLHZV S&RQWH[W！966HW6KDGHU5HVRXUFHV VWDUW FRXQW 6KDGHU5HVRXUFH9LHZV

清单3.10.将着色器资源视图数组绑定到顶点着色器阶段。

着色器资源视图数组的操作方式与常量缓冲区中描述的方式大致相同。一旦在其中一个插槽中设置了着色器资源视图，它将保持绑定在此位置，直到被另一个着色器资源视图或空指针替换。清除所有未使用的着色器资源视图的技术管道执行之间使用NULL在应用于资源时具有更大的重要性。如果GPU动态生成的资源在另一个渲染过程中用作着色器资源，则首先在第一个过程中由管道写入，然后在第二个过程中读入着色器程序。写入资源是不正确的e使用渲染目标视图或无序访问视图，同时使用着色器资源视图执行读取。如果资源被绑定为使用着色器资源视图读取，并且被错误地左绑定到顶点着色器阶段，则任何绑定资源以进行写入的尝试都将生成错误，因为无法同时读取资源由多个视图读取和写入。这种情况如图3.15所示。确保不会发生这种情况的最简单方法是，每次为新渲染效果配置管道时，从顶点着色器阶段清除所有不需要的着色器资源视图。

采样器

应用程序可以提供给顶点着色器阶段的最终配置是采样器，它提供了对纹理执行各种类型过滤的能力*采样器对象。*

148                                                                                                                                                      3.渲染管道

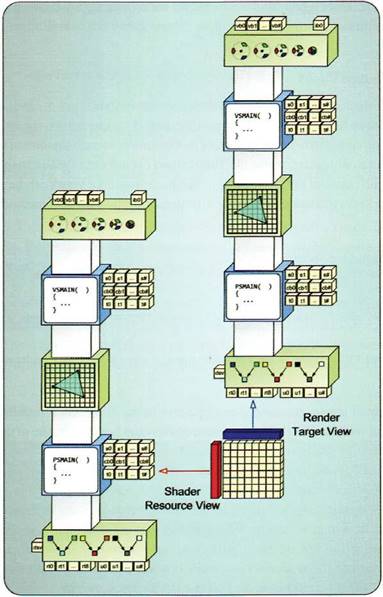


图3.15.在两个不同的渲染过程中用于读取和写入的资源。

当它们从内存中读取时。这通常会导致采样值的性能和/或质量显著提高，因为GPU通常具有专门用于加速此过程的额外专用硬件。清单3.11演示了ID3DllSamplerState实例如何绑定到管道。我们再次看到iar8

8第2章详细介绍了取样器。

3.4顶点着色器149

同时设置采样器状态数组的方法，以减少配置渲染管道所需的与API的交互次数。

I D 3 D l S a m p l e r t a t e\*采样器状态[D3D11\_c0; u SAMPLER\_SLOT\_C0UNT]；

/ / F i l l i n t h采样器s t a t e r e…pContext->vSetSamplers（s t a r t，计数，采样器状态）；

清单3.11.将采样器状态对象数组绑定到顶点着色器阶段。

总的来说，我们可以将这四种配置分组在一起，并将它们视为可用的顶点着色器阶段状态。配置顶点着色器需要适当的着色器程序，以及所有着色器程序中调用的所有常数缓冲器、着色器资源和采样器。不同的用途和要求，这些状态可能有许多不同的组合。

### 3.4.3顶点着色器阶段处理

现在，我们知道顶点着色器可以从输入汇编程序接收哪些数据作为输入数据，以及主机应用程序可以绑定哪些资源。我们还知道顶点着色器程序提供了对单个顶点的自定义处理，而不管调用管道的拓扑是什么。因此，执行哪些类型的操作在顶点着色器程序中的MED？某些类型的操作更适合于这个阶段的语义吗？我们将考虑顶点着色器阶段通常使用的操作，并将评论如何最好地使用流水线的这个阶段。

几何操纵

顶点着色器传统上用于在输入数据稍后在管道中光栅化之前对其执行几何操作。这是因为模型的顶点包含表示正在渲染的对象的几何曲面的信息。由于所有顶点都在管道的早期通过顶点着色器，因此这是执行几何操作的合理位置。几个例子可以很容易地阐明这个主题。

通常在顶点着色器中执行的原型操作是将变换矩阵应用于顶点位置。此操作修改顶点的“输入位置”（input position）属性，以通过执行顶点位置与由提供的变换矩阵的矩阵乘法来反映场景中几何体的位置

3.渲染管道

在恒定缓冲区中的应用程序。此外，对模型坐标系敏感的任何顶点属性也必须转换为具有类似矩阵乘法的新坐标系。顶点的位置及其法向量都定义了模型的物理表示，模型由变换矩阵修改。通过在“顶点着色器”（vertex shader）中执行此计算，我们可以确定在进一步执行后续处理之前，模型的所有几何体都将转换为所需的坐标空间。9

顶点着色器阶段中经常执行的另一个操作是顶点蒙皮。该操作在模型上执行不同类型的坐标变换，其中骨骼在骨骼系统中按层次相互链接，每个顶点指定给一个或多个骨骼。这些骨骼以不同的方式移动模型的不同部分，例如，模拟移动手臂时骨骼如何移动皮肤。与上面的标准变换示例类似，此操作修改模型的物理结构以提供所需的姿势。在进一步处理之前，也可以在顶点着色器中执行此操作。*10*

顶点照明

几何操纵和顶点蒙皮可以在顶点着色器中高效执行，因为它们可以直接在顶点表示要渲染的几何体的物理形状、结构、方向和位置的数据上执行。当然，这些不是可以在顶点着色器中执行的唯一操作类型。由于此阶段是可编程的，因此可以在顶点着色器中执行许多特殊类型的处理，然后将其传递到后续阶段。此类操作的常见示例是逐顶点照明。逐顶点照明计算从顶点反射的光量，通常使用某种形式的简化照明模型（Hoxley）。照明数据可以以输出顶点格式存储为三分量或四分量浮点属性，其中值1.0表示全部颜色可用，0.0表示完全没有该颜色。

这些逐顶点照明值沿管道传递，最终到达光栅化器阶段，该阶段将它们作为基本体中每个顶点之间的属性进行插值。这些插值随后应用于光栅化器生成的每个像素，并最终用于确定写入渲染目标的最终颜色。通过在“顶点着色器”（vertex shader）中执行这些计算，将仅在稀疏的点集（顶点处）计算照明量，而不是在模型生成的每个像素处计算照明量。只要顶点在屏幕上的距离不太远，生成的值将在顶点之间插值，从而降低计算的分辨率。

9 第8章提供了变换矩阵以及坐标空间的基本介绍。“网格渲染。”

10顶点蒙皮也将在第8章“网格渲染”中详细讨论

3.4顶点着色器151

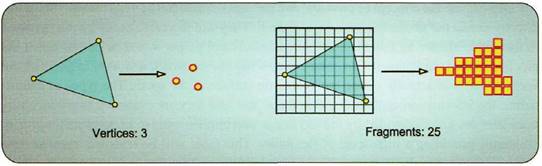


图3.16。一对三角形中逐顶点和逐像素计算之间的差异。

一般逐顶点计算

许多其他类型的计算可以遵循与上述照明类似的模型。如果可以在顶点着色器阶段执行计算，并且可以在每像素级别使用插值版本的数据，则在顶点着色器中执行计算是有意义的，因为每个顶点只执行一次计算。如果在光栅化后执行计算，则会执行更多次。图3.16显示了逐顶点计算和逐像素计算之间的计算频率差异。

从数学上讲，如果计算是其输入的线性组合，则由于顶点属性之间的插值，逐顶点计算的结果将与逐像素计算的结果相同。即使计算不是线性的，逐顶点计算仍然可以提供完整的逐像素计算的相当好的近似值。此近似的有效性取决于计算类型，以及最终光栅化基本体在最终渲染目标中显示的大小。如果顶点之间生成的像素数量非常小，则不太可能注意到插值的任何不精确性。这也意味着，如果创建的几何体输入集具有更多的顶点，则插值效果将不那么明显，而要处理的顶点数量将增加。没有简单的规则来确定特定模型所需的详细程度。这是可用处理能力和所需图像质量的函数。作为本章后面内容的一个小提示，新的细分阶段旨在通过动态生成适当数量的顶点（仅在需要它们的地方（它们可见的地方））在这方面取得平衡。这降低了顶点处理的总体成本，同时仍然可以生成小屏幕空间大小的原语，从而有效地提高了所需处理量相同的可用图像质量。

控制点处理

使用顶点着色器阶段的另一种方法是处理实际上不是顶点的数据，而是高阶基本体或控制面片的控制点。这些控制点

152                                                                                                                                                     3.渲染管道

仍然封装了位置的概念，可能还有方向的概念，即使它们没有直接定义网格的几何曲面。因此，可以在顶点着色器中轻松操纵它们。在这种情况下，经过处理的控制点将沿着管道传递到细分阶段，在细分阶段将对其进行评估并用于生成实际定义网格几何曲面的顶点。控制点的类型、包含在其中的信息以及如何对其进行评估以稍后生成顶点，都由开发人员决定，并在可编程着色器程序中实现，提供了高度的灵活性。这一主题将在本书中多次重温（特别是在第4章“细分管道”以及一些示例算法章节中）。

顶点缓存

在本章前面，我们讨论了如何为每个顶点调用一次顶点着色器程序。虽然顶点着色器阶段每个顶点有效执行一次，但单个处理的顶点结果可能会在多个基本体之间共享。例如，如果一个顶点由三角形条带中的两个三角形基本体共享，则可以对其进行一次处理，其结果可以在管道中稍后的顶点着色器阶段之后在两个三角形基本体中使用。使用任何“条带”基本拓扑类型都将允许这种类型的顶点重用。此外，使用索引渲染时，即使是“列表”基本体拓扑类型也可以定义共享相同顶点的多个基本体。与无法共享顶点的渲染技术相比，这可以显著减少要处理的顶点数。作为一个例子，考虑图3.17所示的几何结构。在这三种情况中的每一种情况下，使用不同的基本拓扑来定义所需的模型，需要处理的顶点数量也不同。

关于顶点缓存的另一个注意事项是，此操作依赖于硬件。缓存的大小（甚至它的存在）在不同的CPU之间可能会有很大的差异。因此，最佳做法是确保在顶点或输入中始终尽可能紧密地引用任何共享顶点。就

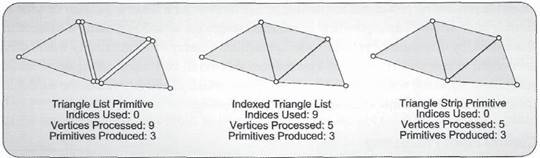


图3.17。具有不同拓扑的不同几何体集，以及由此产生的需要处理的顶点数。

3.4顶点着色器153

“条带”基本体拓扑类型，这是自动完成的，因为每个基本体都使用其前面的顶点来创建新的基本体。在索引渲染的情况下，引用相同顶点的索引应尽可能紧密地放置在一起，以确保缓存的顶点可以重用。

### 3.4.4顶点着色器管道输出

在决定在输出顶点结构中包含哪些信息时，需要考虑如何使用管道的其余部分。图3.18显示了渲染管道的框图。

顶点着色器阶段之后是一组细分阶段（外壳着色器、细分器和域着色器阶段），然后是几何体着色器，然后是光栅化器阶段。根据顶点着色器和光栅化器阶段之间这些阶段中的哪一个处于活动状态，必须满足不同的要求。如果顶点着色器直接连接到光栅化器阶段，则它必须在SV\_位置系统值中生成每个顶点的最终剪辑空间位置。如果细分阶段（所有三个阶段同时处于活动或非活动状态）或几何体着色器阶段处于活动状态，则顶点着色器生成剪辑空间位置是可选的。但是，光栅化器级之前的最后一个活动级必须提供SV\_位置系统值。

类似地，如果细分阶段处于活动状态，则顶点着色器阶段必须为外壳着色器阶段提供控制点。输出顶点和控制点之间的区别实际上仅取决于所实现的细分方案，但在这两种情况下，数据仍然由顶点着色器生成。如果禁用了细分阶段并启用了几何体着色器，则顶点着色器输出将直接发送到几何体着色器。在这种情况下，几何体着色器必须向光栅化器提供SV\_位置语义，尽管它可以在顶点着色器中计算，然后传递给几何体着色器。仅生成一个位置计算的各种选项强调了管道提供的灵活性，以及开发人员以最有利的方法实现算法的相应自由度。

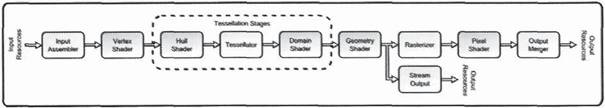


图3.18。渲染管道的框图。

154                                                                                                                                                      3.渲染管道

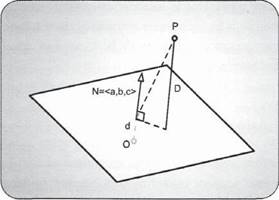


图3.19。计算从点到平面的距离。

系统值

除了这些基于阶段的注意事项外，顶点着色器阶段还可以写入两个新的系统值：SV\_clipdance[n]和SV\_CullDistance[n]。它们都可以在光栅化器之前的阶段中写入，用于执行两种不同类型的操作：剪切和剔除。管道的剪裁和剔除在光栅化器中实现，并在本章的“光栅化器”部分进行更详细的讨论。然而，我们将在这里简要描述这些操作，以解释这两个系统值中提供的功能。

我们首先简要介绍点到平面距离的计算。一般来说，点到平面的最短距离可以通过点的位置和平面的归一化法向量的点积减去平面到其坐标空间原点的最短距离来求出。当熟悉的平面方程适用于平面时，很容易得到这个结果。方程式（3.1）显示了提供该特性的方程式，其中a、b和c是法向长度法向量的分量，d是从原点到平面的最短距离。该方程产生一个标量结果，可以采用三个不同的值范围。如果该点位于法线指向的平面一侧，则该值可以为正值；如果该点正好位于该平面上，则该值可以为零值；如果该点位于远离法向量的平面一侧，则该值可以为负值。图3.19显示了平面和点的计算：

|  |  |
| --- | --- |
| *D=ax1* +借*1+cz1-d。* | (3.1) |

3.4顶点着色器155

要保守地从进一步处理中剔除完整的基本体，所有顶点位置必须完全位于定义剪辑空间的六个平面中的至少一个之外。“顶点位置和每个平面之间的每个距离都可以如上所述进行计算。如果一个基本体的所有顶点导致六个平面中的任何一个平面的距离值为负值，则可以放弃该基本体，因为它不可见。SV\_CullDistance[n]系统值语义的行为方式与上面提到的保守剔除形式大致相同。使用此系统值声明的每个属性表示到剔除平面的距离。当光栅化器阶段解释这些系统值时，它将消除顶点在同一寄存器中均为负值的任何基本体。清单3.12显示了此类系统值使用的示例。如果“剪辑”属性的同一组件对于基本体的所有顶点都为负值，则将对其进行剔除，而不进行光栅化。这种安排允许为该属性存储多达四个不同剔除方程结果的结果。

结构VS\_OUT

{

浮动4位：SV位；

float4clips:SV\_距离；

};

清单3.12。使用四分量SVCullDistance系统值语义的示例输出顶点结构。

SV\_ClipDistance系统值语义以类似方式操作，并定义到剪裁平面的距离。如果基本体的一个或多个顶点在此属性中具有负值，则其值将用于确定应剪裁当前基本体的哪个部分。实际剪裁可以在光栅化之前或之后执行，这是开发人员无法事先知道的特定于实现的细节。无论原语是如何剪裁的，结果都是光栅化后生成的片段都不会在插值SV\_ClipDistance系统值中包含负值。这与SV\_CullDistance系统变量不同，SV\_CullDistance系统变量对整个基元执行剔除测试，而不是试图“剪裁”基元，使其仅具有正的SV\_ClipDistance属性。

这些剪辑和消隐系统值的最大数量总共是两个属性，最多包含四个组件，每个组件都可以以SV\_clipdance和SV\_CullDistance的任意组合声明。这意味着总共有八个平面可以被剪裁（如果两个float4用作SV\_Clipdance）或者被剔除（如果两个float4用作SV\_CullDistance），或者两者的某种组合。

11 1我们将在本章的光栅化器和像素着色器部分更详细地讨论剪辑空间及其属性。

156                                                                                                                                                     3.渲染管道

## 3.5外壳着色器

我们的下一站将带我们进入新的细分阶段。这三个阶段共同实现一个灵活且可配置的系统。这些阶段按管道中的外观顺序为外壳着色器阶段、细分器阶段和域着色器阶段。在这些阶段中，外壳着色器和域着色器阶段是可编程的，而镶嵌器阶段执行固定功能操作。可以将外壳着色器阶段视为执行设置操作，为进一步处理准备输入原语。它指示细分器阶段如何细分输入几何体，并为域着色器阶段提供已处理的控制点原语，域着色器阶段将用数据填充新细分的顶点。因此，外壳着色器在配置整体细分方案中起着非常重要的作用。图3.20中突出显示了管道中外壳着色器的位置。12

外壳着色器阶段的职责分为两个单独的HLSL功能。第一个是外壳着色器程序。为创建输出控制面片所需的每个控制点调用一次外壳着色器程序。它接收由顶点着色器生成的控制点构造的输入控制面片原语，并且必须为每次调用外壳着色器程序创建一个经过处理的输出控制点。与顶点着色器不同，外壳着色器程序可以感知通过它的几何体的输入和输出基本体拓扑。事实上，外壳着色器的每次调用都可以访问属于其输入控制补丁的所有输入控制点。通过访问完整的输入数据，“外壳着色器”阶段还可以在控制修补程序传递到管道中的下一阶段之前，静态扩展或减少控制修补程序中的控制点数量。控制点计数的此更改在HLSL中使用特殊属性静态声明，传递到外壳着色器的所有控制面片将以相同的方式减少或扩展。这是我们第一次看到管道阶段，它可以对通过它的原始数据流执行放大。

在外壳着色器阶段内执行操作的第二个HLSL函数称为面片常量函数。与外壳着色器程序不同，面片

图3.20。外壳着色器阶段。

12控制补丁基本类型在本章前面的“基本拓扑”一节中介绍。

常量函数对于每个完整的控件补丁只执行一次。此函数的主要目的是定义几个细分因子，这些因子告诉细分器阶段细分输入基本体的精细程度。它还可以计算完整控制点面片的恒定附加属性，这些属性随后将在管道中提供给域着色器。外壳着色器阶段是在HLSL中提供两个功能所需的唯一可编程管道阶段。

在本章的其余部分中，我们将进一步探讨这种函数组合如何允许外壳着色器阶段启用多种细分算法。我们首先检查阶段的典型输入，以及可用于配置的状态类型。接下来讨论适用于外壳着色器的典型处理操作。最后，我们将查看由外壳着色器阶段生成的数据，以及它在管道中传递到的位置。

### 3.5.1外壳着色器管道输入

外壳着色器阶段位于管道中顶点着色器阶段之后。其输入数据是顶点着色器生成的一组经过处理的顶点，这些顶点与输入汇编程序阶段生成的基本流信息相结合，形成控制面片基本体。严格地说，外壳着色器程序在完整的控制面片上运行，这些面片由控制点而不是顶点组成。但是，控制点和顶点之间几乎没有区别。顶点和控制点都定义了位置以及一些附加属性，因此它们的内容没有太大差异。控制点和顶点之间的唯一区别实际上是由输入汇编阶段中指定的基本拓扑决定的。如果特定绘制调用的管道配置将使用细分阶段，则必须声明一种控制点修补程序列表基本类型。如果改为传递等效拓扑（例如，三角形列表拓扑代替三点控制修补程序列表），则运行时将发出错误。因此，我们可以假设，如果细分阶段处于活动状态，则可以说顶点着色器处理控制点。如果细分阶段未处于活动状态，我们称顶点着色器为顶点处理。13

如上所述，对每个所需输出面片的控制点执行一次外壳着色器程序。每次调用外壳着色器程序都完全了解当前输入控制补丁中的所有控制点。这些点作为管道输入属性接收，该属性以类似模板的方式声明。属性中声明了两个模板参数：顶点着色器生成的数据结构和输入控制面片中的点数。控制补丁中的点数必须与输入汇编程序的拓扑类型中指定的点数相匹配。

几何体着色器阶段还能够接收控制面片以及细分阶段。

158                                                                                                                                                    3.渲染管道

清单3.13显示了三点控制面片列表拓扑的外壳着色器程序的示例属性声明。

HS\u控制点输出HSMAIN（InputPatch<HS\u控制点输入，3>ip，uint i:SV\u输出控制点ID，

uint PatchID:SV\_基本ID）

{

HS\_控制\_点\_输出；

//  在此处插入计算输出的代码，Output.WorldPosition=ip[i].WorldPosition；

返回输出；

}

清单3.13。示例外壳着色器程序，演示其属性声明。

清单3.13还显示了一对系统值属性。外壳着色器调用正在处理的特定控制点在SV\_OutputControlPointID系统值中指示，该值必须声明为输入属性。此系统值语义提供单个组件无符号整数，用于标识指定输出控制面片大小内的当前外壳着色器调用控制点。通过使用此属性，外壳着色器程序可以确定在生成输出控制点时要使用的输入控制点的哪一部分。第二个系统值是SV\_PrimitiveID，它为每个控制补丁提供唯一的单组件无符号整数标识符。这是该系统值语义可用的第一个管道阶段，因为顶点着色器没有基本体的概念。

对于传递到外壳着色器阶段的完整控制面片基本体，“面片常数”函数仅运行一次。与外壳着色器程序一样，它可以访问完整的输入控制点集及其所有属性。它还可以使用SV\_PrimitiveID系统值，但不能使用SV\_OutputControlPointID系统值语义。这是有道理的，因为该函数在整个补丁中只运行一次，因此它不会使用单个控制点的标识符。

### 3.5.2外壳着色器状态配置

因为外壳着色器阶段是可编程着色器阶段之一，并且可以访问顶点着色器讨论过的同一组资源。为了完整起见，这里讨论了它们，但实际上，用于设置和获取资源的方法的名称只是简单地更新，以指示哪个管道阶段应该受到该方法的影响，因此我们不会在这里重复代码清单。此处提供的方法名称仅供参考。

3.5船体着色器159

ɪś śſ ƀ

ɪ ś ś ſ ƀ ɪ ś ś ſ ƀ ɪ ś ś ſ ƀ

除了应用程序可以绑定到外壳着色器阶段的资源外，还可以在HLSL代码中指定许多其他函数属性。下文也对这些问题进行了说明。

着色器程序

外壳着色器程序的编译方式与任何其他HLSL程序相同。区别在于，外壳着色器程序的前面必须有面片常量函数，前面必须紧跟函数级属性列表。这些项目必须全部位于同一HLSL文件中，并与主外壳着色器程序一起编译为着色器字节码。生成的字节码随后用于创建ID3DllHullShader对象的实例，然后通过设备上下文将其绑定到管道。14

恒定缓冲区

将数据传送到外壳着色器程序的主要方法是常量缓冲机制。这与顶点着色器阶段的操作方式完全相同。关于在管道执行之间持久设置常量缓冲区的建议也适用于这里。

着色器资源视图

外壳着色器程序的另一个可能的只读数据源是着色器资源视图。如第2章所述，它们提供对大量资源的只读访问。与常量缓冲区相比，资源为外壳着色器提供了更大的可用内存池，但它们的访问速度也可能较慢。

采样器

同样，采样器提供了一种对纹理资源执行过滤操作的机制。外壳着色器阶段可以访问此功能，正如对顶点着色器所讨论的那样。

14 有关如何编译和创建着色器对象的详细信息，请参见第6章“高级着色语言”。

160                                                                                                                                                      3.渲染管道

功能属性

要控制三个细分阶段中使用的细分方案，在外壳着色器的HLSL代码中需要几个函数属性。这些属性不同于在管道阶段之间传递的输入和输出管道数据流属性。相反，这些是执行细分系统特定配置的单独语句。这些属性必须位于HLSL源代码中外壳着色器程序的上方，因为它们用于验证着色器程序本身。清单3.14中提供了属性列表。在本节中讨论每个属性的主题区域时，将对其进行描述。我们在这里列出它们，然后是一个简短的描述。

//   指定要细分的基本体的“域”。[域名（“tri”）]

//   指定要细分的方法。[分割（“分数偶数”）

//指定要从细分创建的基本体的类型。[输出拓扑（“三角形\_cw”）]

//指定由外壳着色器程序创建的点数（该程序

//    也是HU l着色器程序执行的次数。[outputcontrolpoints（3）]

//    指定修补常量函数的名称。[patchconstantfunc（“PassThroughConstantHS”）l

|  |  |  |  |
| --- | --- | --- | --- |
| // | 指定面片常数所对应的最大细分因子 | | |
| // | 功能 | w i l | 能够生产。这是可选的，作为 |
| / / | 提示 | 这个 | d r i v r。 |

[修正系数（5）]

清单3.14.外壳着色器程序的函数属性。

### 3.5.3外壳着色器阶段处理

用于外壳着色器阶段的处理任务在应用程序必须提供的两个HLSL函数之间分割。由于外壳着色器程序创建沿管道传递的输出控制点，其主要职责是读取管道中先前生成的输入控制补丁，并将将它们转换为所需的输出控制补丁格式。补丁常量函数负责根据开发人员定义的度量确定必要的细分因子。在以下部分中，我们将研究这些函数可用于执行的一些更高级别的概念。

3.5船体着色器161



图3.21.更改传递到域着色器阶段的控制补丁中的控制点数量。

外壳着色程序

如上所述，外壳着色器对需要为所选控制面片输出大小生成的每个输出控制点操作一次。输出控制点的数量由应用程序通过在outputcontrolpoints函数属性中指定来配置。系统值语义输入属性SV\_OutputControlPoint id标识在每次调用外壳着色器程序时创建的输出控制点。其值范围为0到n-\，其中n是outputcontrolpoints函数属性中定义的输出控制点数量。此函数属性是外壳着色器可以扩展或减少程序中点数量的机制输出控制补丁。为了生成这些输出控制点，外壳着色器程序接收作为输入属性包含在输入控制补丁原语中的一整套控制点。对于接收16点控制补丁并生成四点控制补丁的外壳着色器，此概念如图3.21所示。

细分算法设置。此输出控制补丁是稍后将在域着色器阶段中使用的数据。由于此配置，外壳着色器可以被视为细分系统的数据设置阶段。正如我们刚才看到的，可以在外壳着色器。此功能可用于生成与域着色器阶段中实现的所需细分算法兼容的控件面片类型。考虑到这一点，外壳着色器可用于使用标准控件面片基本类型（可能由为给定项目选择的内容创建工具强制执行）然后为所需类型的细分算法生成所需的输出面片格式。这充当细分系统的隔离机制，因为可以提供标准输入，并且外壳着色器阶段的输出将始终以适当的格式显示

162                                                                                                                                                      3.渲染管道

当前算法，就好像它是在输入汇编程序阶段传递到管道中的原始几何体一样。可以在不影响管道其余部分的情况下交换不同的细分算法，只要它能够使用可用的输入控制面片几何体。

**细分算法的细节层次。**这种交换细分算法的能力也可以用作细节级别的一种形式，在这种情况下，可以为靠近摄影机的对象选择更复杂或更激进的细分方案，而可以为更远的对象选择不太复杂的细分方案。这将被视为比实际使用varyi的过程更粗糙的LOD形式调整细分量（使用面片常数函数执行）。

**通用控制点计算。**外壳着色器程序还可用于与细分无关的各种其他处理任务。由于它是实际发生细分之前的最后一个阶段，因此它是执行将由多个细分点共享的计算的良好候选者。这可以有效地将计算拉回到某个区域of流水线前的数据放大细分程序，节省了大量的处理成本。

镶嵌因子计算

patch constant函数背后的思想是，它将计算两个细分因子，这两个因子被写入特殊的系统值语义。这两个系统值语义告诉固定函数tessellator stage细分从输入控制面片生成的基元的精细程度。细分因子的数量required受上述“域函数”属性的影响。在该属性中指定了在概念上由镶嵌器阶段分割的基元类型。根据指定的域类型（等值线、三角形或四边形），所需的边缘镶嵌因子数量将有所不同。

patch constantfunc函数本身是用另一个属性patch constantfunc声明的。这只是标识应该为patch constant函数执行哪个HLSL函数。上面列出的其余函数属性用于配置细分阶段的执行。partitioning属性配置细分的类型要执行此操作，outputtopology属性定义将由细分创建的输出原语的类型，可选的maxtessf actor属性告诉驱动程序应该期望的最大细分量，以便它可以分配适当的内存量来保存结果。

这些属性及其可用值将在第4章“细分管道”中详细讨论然而，从这些配置中去掉的重要概念是，面片常数函数及其指定的函数属性用于精确确定细分器阶段产生的结果

3.5船体着色器163

与讨论的外壳着色器程序的设置功能类似。函数属性配置将细分的内容（使用域属性）、细分方式（使用分区属性）以及输出格式（使用outputtopology属性）。然后，“面片常数”函数为每个面片生成细分因子，指示细分器阶段如何细分选定的域基元。如果外壳着色器程序设置细分算法的数据，则“面片常数”函数将设置实际的细分机制。

**精细的细节。**细分因子的计算可以基于任意数量的不同标准。例如，可以根据输入控制面片到查看器的距离修改细分量。当然，与面片距离较远时相比，更靠近查看器的细分量更高，以减少数量r所需的计算量。但是，在其他情况下，这种方法可能不合适或不理想。例如，当对象相对平滑，只有一些锐利的细节时，最好将细分因子基于曲面中变化量的某种度量。这可以预先计算并存储在控件中ol点，或者可能存储在纹理中以供以后查找。也可以（事实上，更可能）组合多个启发式方法来解决此问题。第9章“动态细分”中介绍了此类分析的示例

**图像质量。**在确定所需的细分密度时，应考虑许多其他类型的属性。例如，如果控制面片位于网格的轮廓边缘上，则细分面片将比垂直于视图方向时更能提高图像质量。另一个提高图像质量的非常好的示例是在生成双抛物面阴影贴图或环境贴图时使用不同的细分级别。众所周知，抛物面投影使用覆盖抛物面贴图大面积的三角形生成时会产生伪影。这是因为每个顶点都使用抛物面投影进行变换，但顶点用来产生的三角基元被线性光栅化。这种情况如图3.22所示。1

随着基本体覆盖抛物面贴图越来越多的纹理，这种情况变得越来越麻烦。缓解此问题的一种简单方法是在面片常数函数中计算三角形的屏幕空间面积，然后根据此屏幕空间大小增加细分因子。然后，可以选择合适的细分点在“域着色器”阶段中进行精确投影，以最小化此问题的影响。

**通用控制补丁计算。**除细分因子外，“面片常数”功能还可以生成用户定义的属性，这些属性将应用于当前控制面片中的所有细分点。这意味着所有用户都可以共享任何数据

5有关抛物面贴图的更多详细信息，请参见第13章“多线程示例”

164                                                                                                                                                      3.渲染管道



图3.22.在不保留直线的投影中使用线性光栅化产生的伪影。

控制面片的控制点（如完整面片法向量或面片位置的屏幕空间导数）可以计算一次并在以后多次使用。这再次减少了执行计算所需的计算数量。

### 3.5.4外壳着色器管道输出

我们已经看到，执行外壳着色器程序的结果是要在细分中使用的控制面片的输出控制点。这些控制点是其沿管道传递信息的主要方式。最终，这些控制点将被域着色器阶段使用，在那里它们将被用于生成p从tessellator阶段开始的每个新细分点的位置。对外壳着色器输出属性的要求非常灵活。没有需要写入的强制系统值。相反，由开发人员提供后期处理所需的信息。但是，虽然要求非常灵活，通常包括一些公共属性。控制点的位置是

3.6镶嵌器165

通常由外壳着色器程序提供，通常从输入控制点复制或从中提取。此外，未从资源中读取的任何材质属性通常会包含在控制点中。这些属性可能包括每个控制点的颜色、光泽度或材质标识符，仅举几例。

相反，patch constant函数的输出更为预定义。它必须生成一组大小适当的细分因子，以供细分器阶段使用。这些因子被写入输出属性，系统值语义为SV\_tessefactor和SV\_insidetessesfactor。此外，任何patch constant信息也会被传递这些属性不会发送到镶嵌器阶段，因为它不执行自定义处理，而是作为输入属性直接传递给域着色器程序的每次调用。

## 3.6镶嵌器

细分器阶段是细分系统的固定功能管道阶段。它位于外壳着色器阶段和域着色器阶段之间的某个位置，该位置定义其输入和输出的源和目标。当在渲染管道中使用细分阶段时，它们将一起使用，无法单独操作。因此，细分器阶段在实现任何细分算法中都起着重要作用。图3.23中突出显示了细分器阶段的位置。

细分器阶段的工作是将在外壳着色器阶段面片常数函数的细分因子中确定的所请求的细分量转换为当前“域”内的一组坐标点。生成这些点的域可以是等值线、三角形、，或四元组，以在外壳着色器程序的“域函数”属性中指定的为准。这应该有点令人惊讶，因为上面描述的操作没有使用控制点或控制补丁。相反，镶嵌器阶段只生成一组点，指示应在指定域内创建顶点的位置。这些点稍后将传递到域着色器阶段，在该阶段，它们将与外壳着色器生成的控制点重新连接



图3.23。镶嵌器阶段。

166                                                                                                                                                      3.渲染管道

程序为这些点赋予物理意义。了解棋盘级如何操作，以及为什么它以这种违反直觉的方式来实现，将为构建镶嵌算法提供坚实的基础。

### 3.6.1细分器级管道输入

为了理解细分器阶段，我们将首先考虑流入细分器的数据类型。顶点着色器阶段接收来自输入汇编器阶段的已装配顶点/控制点，外壳着色器阶段接收来自顶点着色器阶段的输出控制点创建的控制面片（以及来自输入汇编器的基本体信息）。这一基本概念如图3.24所示。

在管道早期的这两种情况下，流入该阶段的数据以某种方式进行处理，然后传递到下一阶段。在细分器阶段，情况并非如此。相反，它接收由外壳着色器阶段面片常量函数生成的细分因子。这些因素是简单的浮点数，表示对域对象的不同部分进行细分的程度。无论在“外壳着色器”阶段中如何计算细分因子，如果两个控制面片生成相同的细分因子集，则它们看起来与细分器阶段相同。由于细分器阶段无法区分两组输入，因此在这两种情况下，它将生成完全相同的输出。

严格地说，如果给定两组相同的输入，所有其他管道级将产生相同的输出。细分器之所以独特，是因为它更可能遇到来自多个不同控制面片的相同输入值，这是由于将输入数据“展平”为简单因子所致。这与其他管道阶段有着深刻的区别。这将是重要的考虑，因为我们进一步探索在细分器阶段执行的处理类型。



图3.24。流经管道的数据的表示形式。

3.6镶嵌器167

### 3.6.2细分器级状态配置

镶嵌器阶段不是由应用程序直接配置的，而是由外壳着色器程序（由应用程序配置）中指定的属性配置的。可用的配置集是“外壳着色器”部分中提到的函数属性集合。由于大多数函数属性都用于配置细分器阶段（即使它们是在外壳着色器阶段中指定的），因此我们将更仔细地了解它们的用途。

域属性

我们从domain属性开始。我们已经提到，细分器阶段实际上并不细分控制面片，而是细分域对象。此属性中指定的域类型可以有三个值之一：等值线、t r i或四边形。在较高的级别上，我们可以说细分器阶段确定了该域中需要由域着色器阶段实现的点。此属性是必需的，因为域的类型也用于验证面片常数函数生成的细分因子的数量。

分区属性

下一个属性是分区属性。这可以采用四个值之一：整数、分数偶数、分数奇数或pow2。每个值都标识一个不同的方案，用于如何分割指定的域。正如我们将在第4章“细分管道”中更详细地看到的那样，每个分区方案所选择的细分点存在显著差异。此属性也是必需的。

输出拓扑属性

一旦指定的域根据指定的分区方案被分割，镶嵌器阶段还需要知道如何从这些单独的点组装原语。此属性的可用选项有三角形cw、三角形ccw和直线。这一属性也是强制性的；否则，细分阶段生成的原语可能会产生与预期显著不同的结果。例如，如果生成了不正确的三角形缠绕，则渲染的几何体将从内到外显示！

最大细分因子属性

配置细分阶段的最后一个属性是MaxTessesFactor。此属性是向驱动程序提供的关于最大可能数据放大量的可选提示。使用此细分因子上限，驱动程序可以有效地预先分配足够的内存以接收细分操作的结果。

168                                                                                                                                                     3.渲染管道

### 3.6.3细分器阶段处理

那么，细分器阶段使用上述细分因子和配置执行的具体操作是什么？简单的答案是，它选择给定域中的一系列点，然后生成基本数据，这些数据稍后在管道中从这些点创建输出基本数据。然而，关于如何执行此操作以及如何控制生成的输出的细节并不十分清楚。在本节中，我们将更详细地了解细分过程以及细分器阶段正在执行的操作。

样本位置

细分器阶段更像是数据生成阶段，而不是数据处理阶段。指定域类型后，内部细分因子和边细分因子将指定要执行的细分量。细分本身分两步进行。首先，选择将生成适当数量三角形的细分点。如果边缘镶嵌因子高于内部因素，则更多的点将被选择接近域的边缘而不是在中间。如果接收到更高的内部细分因子，则反转也是正确的，在域的内部部分将选择更多的点。图3.25中描述了四元域。

选定点仅由当前域中的一组坐标标识。首先，当考虑到镶嵌器阶段正在做什么时，这有点令人惊讶。这些坐标是细分器生成的唯一数据，并且它们仅提供通用域形状中的位置，该区域形状应在以后成为顶点，基于外壳着色器生成的控制面片。这就是将细分器阶段称为数据生成阶段而不是数据处理阶段的意思。它获取域配置和一组细分因子，然后生成所有必需的采样位置。输出数据量通常会大于



图3.25。受细分因子影响的四边形区域。

3.6镶嵌器169



图3.26。原始数据通过管道流动，直至通过细分阶段。

这仅仅是因为此阶段最多只使用五个浮点值作为细分因子。

原始世代

细分器阶段的第二步是生成所需的基本体信息，以便稍后在管道中将采样位置用作可渲染几何体。您可能想知道为什么必须在这个阶段生成基本数据，因为基本拓扑是在输入汇编阶段指定的。事实上，输入汇编程序创建的基本体连接信息跳过顶点着色器阶段，并在细分系统处于活动状态时传递到外壳着色器阶段，这是正确的。但是，当细分阶段处于活动状态时，来自输入汇编程序阶段的输入原语必须是控制面片类型之一。这将顶点的连接性指定为控制点，并定义控制面片，而不是可以直接光栅化的基本体类型。图3.26显示了原始数据如何通过管道流向细分系统末端的区别。

我们已经看到输出原语的类型是由outputtopology属性指定的。生成直线时，直线没有正面或背面，因此在基本体中顶点的排列顺序并不重要。但是，当生成三角形时，指定顶点的顺序肯定很重要。正如我们将在光栅化器阶段中看到的，背向当前视点的三角形将被丢弃，并且不会影响最终渲染图像。三角形面向哪个方向是通过取两个向量的叉积来确定的，这两个向量是由接触第一个顶点的边创建的。如图3.27所示。

图3.27。确定三角形所面对的方向。



170                                                                                                                                                      3.渲染管道

由于确定哪些曲面基本体可见，哪些不可见的重要性，因此确保输出拓扑设置与光栅化器阶段中使用的消隐配置兼容至关重要。一旦选择了坐标点，基本体信息基本上就是一个列表，其中包含两个（直线）或三个（三角形）到相应坐标点的引用。

### 3.6.4细分器管道数据流输出

一旦完成了对细分器阶段的特定调用的细分过程，结果将通过系统值语义传递到域着色器阶段。SV\_Domain Location系统值一次为域着色器阶段提供一个域坐标点，并且为每个输出点调用一次域着色器程序。在域着色器中处理点之前，它们是当前域中的简单坐标。

由于为新细分的几何体创建的基本体拓扑信息不被域着色器阶段使用，因此从概念上讲，它将从域着色器阶段传递到几何体着色器阶段。几何体着色器阶段知道完整的基本体，因此可以利用这些信息。需要记住的一个重要注意事项是，在细分阶段中，没有可用于创建的具有邻接信息的基本拓扑。这意味着当细分阶段处于活动状态时，几何体着色器阶段无法接收具有邻接性的基本体类型。这限制了可在几何体着色器程序中实现的可用算法，但通常，需要邻接信息的任何计算都可以合并到外壳或域着色器程序中，因为它们具有不同级别的邻接信息。如果“几何体着色器”阶段未处于活动状态，则在“域着色器”阶段中创建的基本体信息和顶点将直接传递到光栅化器阶段，并在该阶段中进行进一步处理并用于生成像素。

## 3.7域着色器

域着色器阶段是细分系统中的最后一站。这是一个可编程的管道阶段，它接收来自外壳着色器阶段和细分器阶段的输入，并生成输出顶点，这些顶点可在管道中稍后由光栅化阶段进一步处理。根据这一职责，域着色器阶段可以被视为细分算法的核心，该算法由外壳着色器和由细分器阶段设置的生成几何体的粒度设置。域着色器阶段的位置在图3.28中高亮显示。

3.7域着色器171



图3.28。域着色器阶段。

### 3.7.1域着色器管道输入

域着色器阶段需要生成顶点以完成细分过程并生成输出几何体。对细分器阶段生成的每个坐标点调用一次。要创建这些顶点，它将接收由外壳着色器阶段生成的完整控制面片。这可以从1到32个控制点不等，但必须与外壳着色器程序的outputcontrolpoints属性中声明的输出点数量相匹配。此处声明的控制面片的方式与在外壳着色器程序中使用的方式类似，其中各个控制点在模板样式属性中提供。每个控制点的结构类型是第一个参数，控制补丁中的点数是第二个参数。清单3.15提供了一个示例域着色器程序输入签名，它演示了这个模板样式属性。

T U S V D U

)4@$0/530-@10\*/5@065165 \

GMPBU 8PSME 1PTJ UJPO 104\*5\*0/

^

4 U S V D U）4@$0/45“/5%/5”@065165

\

GMPBU和EHFT<>47@5FTT"BDUPS"

GMPBU\*OTJEF 47@\*OTJEF5FTT&apos;BDUPS

^

<EPNBJO USJ>

%4@065165 %4."\*/

DPOTU 0VUQVU1BUDI）4@$0/530-@10\*/5@0651655SJBOHMF1BUDI GMPBU#BSZDFOUSJD$PPSEJOBUFT 47@%PNBJO-PDBUJPO

)0/45美元/5%“5”@065165乔基伍

\

^

清单3.15.一个示例域着色器声明，演示了输入属性声明。

172                                                                                                                                                      3.渲染管道

如清单3.15所示，输入控制补丁声明为常量输入参数，因为它无法修改。由于此只读访问，必须包含在域着色器程序的控制补丁中以创建输出顶点的任何修改或附加数据必须在此阶段之前由外壳着色器程序提供。

除了控制补丁之外，域着色器程序还从镶嵌器阶段接收它应该实现的域位置。该数据作为SV\_Domainl\_location系统值语义提供。该属性的格式根据指定的域类型而不同，其中float2用于等值线和四元d域和t r i域的浮点3。这些值表示域着色器调用应在域内的何处生成顶点数据。在这方面，坐标位置定义了由域着色器程序实现的函数确定的虚拟曲面上的采样点。我们将在中进一步研究此概念本节的处理部分。

域着色器程序接收的最后一组输入数据是由hull shader stage的patch constant函数生成的常量。此结构中包含的数据根据patch constant函数中计算的内容而有所不同，但对于特定时间内的每个域着色器程序调用，该数据将保持不变给定控制补丁。

为了提供域着色器程序需要执行的操作的高级视图，我们可以检查其来自管道的输入。所有输入控制补丁数据都提供给域着色器程序的每次调用。此外，从patch constant函数接收的数据在整个控制补丁中也是常量。这意味着对于一个完整的控制面片，在域着色器调用之间变化的唯一输入是域位置。这是有意义的，因为细分器阶段产生的点的数量会变化，但控制面片数据输入不会随细分级别的变化而变化ull shader stage在这方面提供了明确的区别。

### 3.7.2域着色器状态配置

作为可编程管道阶段，域着色器阶段可以使用所有可编程阶段可用的标准配置集。同样，这些配置由应用程序使用熟悉的方法名称通过ID3D11DeviceContext接口设置，但方法名称的开头使用DS表示它们将影响域着色器阶段。由于已经讨论了这些方法及其使用特征，因此我们不会在此处重复代码列表。有关使用这些方法的更多详细信息，请参见“顶点着色器”部分。此处列出了这些方法以供参考。

\*%%MM%FWJDF$POUFYU%44FU4IBEFS

\*%%MM%FWJDF$POUFYU%44FU$POTUBOU#VGGFOT

3.7域着色器173

ID3DllDeviceContext:：DSSetShaderResources（）

ID3DllDeviceContext:：DSSetSamplers（）

### 3.7.3域着色器阶段处理

域着色器阶段的主要工作是通过使用外壳着色器阶段生成的控制面片和面片常数，使用从细分器阶段生成的坐标点集创建顶点。创建这些顶点的最重要任务是确定它们的位置，因为这将直接影响到最终结果镶嵌几何体将被光栅化。考虑到这一点，我们将更仔细地了解如何在域着色器程序中计算位置，并提供一些高级概念来考虑此过程。

控制面片内的点中保存的数据可以表示各种不同类型的高阶曲面。控制点可以用于实现Bezier曲面（Akenine Moeller，2002），就像它们可以表示曲面三角形中使用的法线顶点一样容易（Vlachos，2001）。唯一的要求是外壳着色器程序必须生成预期的数据格式，并且域着色器必须实现兼容的算法，以便为每个坐标点生成输出位置。16

在更广义的视图中，我们可以将控制面片视为定义虚拟曲面的一组参数。该曲面必须在控制面片的整个域内有效，因为这是所有输入坐标点将驻留的区域。然后，我们可以考虑由细分器生成的每个坐标点阶段作为我们希望对虚拟表面进行采样的选定位置。这可以通过简化的案例进行可视化，如图3.29所示，



我们看到的是等值线域、其控制点和它所代表的虚拟曲面。沿等值线域选择的采样点指示沿虚拟曲面采样的位置。

从这个概念中得到的重要信息是，无论细分器生成的点位于何处或有多少点，虚拟曲面都保持不变。这有助于我们打破总体平衡

图3.29。描述控制面片如何定义虚拟曲面的参数，以及细分器生成的点，这些点可被视为该曲面的采样位置。

16 Bezier曲面将在第4章中详细讨论，而弯曲点三角形将在第9章中进一步讨论。

174                                                                                                                                                      3.渲染管道



图3.30。在指定采样点从虚拟曲面构造几何图形的过程。

图3.31。将影响曲面单个四边形的控制点集。

将细分算法设计任务分成几个不同的组件。首先，我们的控件面片必须定义我们实际需要的曲面。如果对象是由艺术家预先创建的，则这通常在内容创建阶段完成。其次，必须从控制面片计算细分因子，以确保所需曲面得到充分采样，从而在当前查看条件下正确显示，同时最小化实现该视觉质量所需的顶点数量。最后，当提供一组坐标时，我们的算法必须能够

从控制面片数据中正确采样虚拟曲面。这些步骤如图3.30中的方框图所示。

贝塞尔曲线

考虑到这些步骤，我们可以考虑一个传统的镶嵌算法，并看看它将如何适应这个完整的镶嵌范式。我们将考虑一个最著名的表面表示贝塞尔表面。Bezier曲面的一般概念是通过移动控制点网格来塑造所需曲面。点的数量可能会有所不同，但在我们的示例中，我们将假设控制点的4x4栅格来定义曲面四边形部分的曲面形状。此设置如图3.31所示。

将曲面编辑为所需形状后，控制点将作为顶点存储在顶点缓冲区中，索引缓冲区用于将控制点组定义为控制面片。接下来，当要渲染对象时，输入汇编程序将控制点构建为顶点，并将其传递给顶点着色器。同时,，

3.7域着色器175



图3.32。通过输入汇编程序、顶点着色器和外壳着色器阶段时的4x4贝塞尔曲面。

它根据索引缓冲区中提供的索引及其基本类型设置（本例中为16点控制补丁），确定哪些控制点属于特定的控制补丁。控制面片点由顶点着色器读取并传递给外壳着色器。然后，外壳着色器读取4x4组控制点，并以相同的4x4控制面片配置传递给域着色器。到目前为止的过程代表了图3.30方框图中的第一步。图3.32描述了这一过程。

接下来，面片常数函数必须计算所需的细分量，以便在给定的观察条件下充分表示贝塞尔曲面。这可以考虑控制面片的大小、与当前视点的距离或任何数量的其他度量。所需的细分因子将传递给细分阶段，细分阶段将该数据转换为域内的一系列坐标。然后将这些点传递到域着色器阶段。这表示细分过程中的第二步，如图3.33所示。



图3.33。根据当前查看条件确定特定四边形所需的边缘因子。

176                                                                                                                                                     3.渲染管道



图3.34。在4x4 Bezier曲面示例的域着色器中生成顶点位置。

最后，域着色器程序必须能够接收控制面片，以及域着色器每次调用的一个坐标点，并生成表示贝塞尔曲面的顶点。这基本上是通过计算Bezier曲面的方程来实现的，其中每个控制点都对顶点的计算位置有贡献。贡献量由坐标确定，坐标或多或少地指定了我们评估的每个控制点的接近程度。一旦计算出该位置，该位置将由域着色器生成，并通过管道进行进一步处理。如图3.34所示。

当然，域着色器通常需要计算的不仅仅是每个细分顶点的位置。在大多数情况下需要的一个属性是法向量，用于管道中的进一步照明计算。然而，这与我们讨论的位置确定过程相同。唯一的区别是，除了位置之外，我们还必须在域着色器程序中计算法向量。渲染所需的任何其他属性也是如此。它必须根据可用信息计算，即使只是从每个控制点属性读取。

### 3.7.4域着色器管道输出

计算所需的属性数据后，域着色器程序将返回新创建的顶点，并将其传递到下一阶段。此阶段的位置输出

3.8几何体着色器177

通常以SV\_位置系统值语义形式写入，该语义必须出现在光栅化阶段的输入中。此外，确定最终像素颜色所需的任何其他逐顶点属性也将添加到域着色器输出。

几何体着色器阶段可能处于活动状态，也可能处于非活动状态，具体取决于所需的管道配置。如果它处于活动状态，则域着色器的输出将传递到几何体着色器阶段，在该阶段，它将作为完整的基本体使用（请记住，在本例中，基本体信息是在细分器阶段生成的）。否则，输出将直接传递到光栅化器阶段，在那里它也将作为基本体使用。

## 3.8几何体着色器

几何体着色器是最后一个管道阶段，可以在光栅化之前操纵通过管道的几何体。它是一个可编程阶段，具有在任何其他阶段中都找不到的几个独特功能，包括以编程方式在管道中插入/删除几何体的能力，通过流输出阶段将几何体信息传递给顶点缓冲区的能力，以及生成不同于传递给它的原语类型的能力。这为这个阶段提供了一些非常有趣的用例，这些用例超出了传统的管道模型，从保存处理过的几何体数据到文件，管道操作调试，当然还有渲染操作。“几何体着色器”阶段还可对完整的基本体（包括相邻的基本体信息）进行操作，这提供了分析和测试每个基本体的各个方面并执行自定义计算的附加功能，具体取决于几何体及其直接相邻体。图3.35突出显示了几何体着色器阶段在管道中的位置。

几何体着色器阶段接收表示输入基本体的顶点列表。然后可以自由地将这些顶点传递到输出流，然后在输出流中重新解释它们



图3.35。几何体着色器阶段。

178                                                                                                                                                      3.渲染管道

在传递到管道中的下一个阶段时作为基本体。根据为接收此输出而声明的流对象的类型，流顶点将用于构造不同的基本体。此外，最多可以声明四个输出流与流输出阶段一起使用，这使得生成用于光栅化的几何图形以及保存到缓冲区资源中的几何图形变得非常简单。此流模型是几何体着色器操作的核心，我们将在本节中了解如何在各种情况下使用它。

### 3.8.1几何体着色器管道输入

几何体着色器阶段对由顶点数组组成的完整基本体进行操作。根据要传递的基本体类型，数组的大小将有所不同，对于具有邻接信息的三角形，从一个顶点一直到六个顶点。几何体着色器程序的输入声明它将在输入属性定义中接收的基本体的类型。清单3.16中显示了一个示例几何体着色器函数签名。

<JOTUBODF>

<NBYWFSUFYDPVOU>

WPJE（44DFOF U S J B O H M F B E K）（44DFOF mo J O Q V U<>

JOPVU 5SJBOHMF4USFBN 144DFOF\*O 0VUQVU4USFBN

{

144DFOFMO PVUQVU 144DFOF\*O

全球定位系统

output.Pos=输入[i].Pos；

output.Norm=input[i].Norm；

output.Tex=输入[i].Tex；

追加（输出）；

}

OutputStream.RestartStrip（）；

}

清单3.16.一个示例几何体着色器程序，演示如何声明输入流属性。

在此列表中，input属性表示一个具有邻接性的三角形，它从六个顶点中生成四个三角形。此输入数组中顶点的顺序取决于传递给函数的基元类型。各种基元类型、它们的顶点计数以及它们的顶点的提供顺序如下：表3.1中列出。此外，图3.36中显示了每种基本类型，以便于参考。

3.8几何体着色器179



-图3.36.几何体着色器可接收的各种基本排列。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **关键词** | **原始类型** | **数量** | **顶点排序** |  |
| ,,  . |  |
| *1* | *}五* | 顶点 |  |  |
|  |  | 顶点 |  |  |
| **指向** | **点列表** | **1** | **[V**O] 0 |  |
|  |  |  |  |  |
| **线** | **行列表，行s**旅行 | **2** | **[V**0，V1O.l] |  |
|  |  |  |  |  |
| **三角形** | **三角形列表，三角形s**旅行 | **3** | **[V0、V1、V2]** |  |
|  |  |  |  |  |
| **L**我的 | **带调整的线路列表，带调整的线路带**/形容词。 | **4** | **[A0、V0、V1、A1]** |  |
|  |  |  |  |  |
| **三角导线J** | **带调整的三角形列表，带调整的三角形条。** | **6** | [V0、A0、V1、A1、V2、A2] |  |
|  |  |  |  |  |

表3.1.用于确定如何评估几何体着色器输入顶点流的表。

使用可用的基本体信息，几何体着色器程序可以检查几何体、执行可见性测试或计算基本体作为一个整体的高级信息。有两种可能的管道配置可以将基本体数据发送到几何体着色器阶段。如果镶嵌处于非活动状态（外壳和域着色器程序设置为NULL），输入顶点直接从顶点着色器接收，具有输入汇编程序指定的基本体连接。在这种情况下，基本体拓扑可以是输入汇编程序可以使用的任何基本体拓扑

180                                                                                                                                                      3.渲染管道



图3.37.在顶点着色器和几何体着色器中处理顶点级别数据之间的差异。

指定。除了标准的点、线和三角形类型外，还包括控制面片基本类型，包括它们的各种邻接形式。

另一种可能的管道配置是当细分阶段处于活动状态时。在这种情况下，输入顶点从域着色器接收，基本体拓扑由细分阶段的配置确定。由于细分阶段只能生成直线或三角形，因此这些是唯一可用的基本体拓扑当细分处于活动状态时，几何体着色器可以接收的PE。这就是为什么在细分阶段处于活动状态时不支持具有邻接性的基本体。

无论管道配置如何，几何体都将作为顶点列表接收，这些顶点表示管道中此时存在的基本体。这对性能有一定影响，因为减少顶点处理量的任何渲染模式都不会在几何体着色器中产生相同的节省。例如，如果具有四个顶点的三角形条在顶点着色器中处理，顶点着色器的四次调用将处理完整的几何体集。但是，如果在几何体着色器中处理每个顶点，则所有三个顶点都将提交给每个几何体着色器调用，即正在处理的基元数。在在这种情况下，将有两个几何体着色器调用，每个调用将接收正在处理的三角形的三个顶点。因此，在顶点着色器中处理顶点只需要四组计算，而在几何体着色器中执行的相同操作将需要六组计算。此差异如图3.37所示。

我们还可以从清单3.16中看到如何为几何体着色器声明流输出对象。该对象实际上是几何体着色器用于将其输出传递出后台的机制，而不是直接返回其输出值的函数。即使如此，流还是作为输入参数传递，因此将在他的输入讨论。声明流输出对象的语法以inout关键字开始，该关键字指定该对象既是输入又是输出。之后是三种可能的流类型之一。流输出对象可用于输出点列表、线带或三角形带；每种类型由可用的

3.8几何体着色器181

流声明。参数可以声明为PointStream<T>.LineStream<T>或TriangleStream<T>，其中每个T表示将传递到流中的顶点结构的格式。我们将在“几何体着色器”部分进一步讨论这些输出流的使用机制。

最后一个输入注意事项是使用两个系统值语义，一个是我们以前见过的，另一个是新的。SV\_PrimitiveID系统值属性可用于唯一标识传递给几何体着色器的每个原语。我们以前在外壳着色器阶段见过此系统值，可用于唯一标识控制补丁。由于每个基本体调用一次几何体着色器，因此此系统值语义也唯一标识几何体着色器调用。

几何体着色器可以用作输入的另一个系统值语义是SV\_GSInstanceID。该系统值语义的工作原理与SV\_InstanceID类似，我们在顶点着色器阶段中看到了SV\_InstanceID，只是实例实际上是在几何体着色器阶段本身而不是在输入汇编阶段中实例化的。可以re为几何体着色器接收的每个基本体创建的静态实例数。SV\_GSInstanceID用于区分各个基本体实例，并在着色器程序中采取适当的操作。此系统值必须声明为几何体着色器的独立输入，因为它不能在o中声明输入流的ne。由于标识符指示正在处理的几何体着色器调用，因此不允许此为逐顶点流属性是有意义的。启用此实例化机制的技术将在“几何体着色器状态配置”中讨论而“几何体着色器阶段处理”一节将更详细地讨论该功能的使用。

### 3.8.2几何体着色器状态配置

与所有其他可编程管道阶段一样，几何体着色器阶段提供了一组常见的着色器核心资源配置，可由应用程序通过ID3DllDeviceContext接口进行操作。由于几何体着色器阶段的关闭，在创建几何体着色器对象的方式上存在一些关键差异与流输出阶段的关系，我们将在本节中简要介绍，并在“流输出”阶段一节中进行更详细的检查。

几何体着色器程序

可以使用两种方法之一创建几何体着色器对象。必须使用哪种方法取决于是否使用流输出阶段。如果不使用流输出阶段，则几何体着色器对象的创建方式与我们使用的方式相同17

17 “几何体着色器阶段处理”部分介绍了几何体着色器如何将数据写入流输出阶段的特定细节。

3.渲染管道

已经看到使用ID3DllDevice:：CreateGeometry着色器（…）方法的其他可编程阶段。但是，如果要使用流输出阶段，则必须使用ID3DllDevice:：CreateGeometryShaderWithStreamOutput（…）创建着色器对象方法。此方法需要几个额外的参数，这些参数配置流输出阶段以及将数据流传输到连接到该阶段的缓冲区的方式。如何为流输出版本配置这些参数的详细信息将在“流输出”中讨论本章的阶段部分。请记住，根据是否使用流输出，有两种不同的方法创建着色器对象。

创建几何体着色器对象后，可以在几何体着色器阶段使用通常的ID3DllDeviceContext方法对其进行设置。同样，几何体着色器程序所需的常量缓冲区、着色器资源视图和采样器在标准通用着色器核心方法中进行操作。此处列出了这些方法以供参考。

ID3D11DeviceContext:：GSSetShader（）

ID3D11DeviceContext:：GSSetConstantBuffers（）

ID3D11DeviceContext:：GSSetShaderResources（）

ID3D11DeviceContext:：GSSetSamplers（）

功能属性

geometry shader stage还支持两个必须在HLSL源文件中的geometry shader函数之前声明的函数属性。第一个属性是maxvertexcount参数，它允许开发人员仅指定调用geometry shader将发射到其输出中的最大顶点数t stream。这是确保几何体着色器在出现逻辑错误时不会输出错误的顶点数所必需的。它还允许GPU为每次调用几何体着色器时预期产生的顶点数正确分配内存。此属性是必需的，必须为几何体着色器提供这个函数属性的一个例子可以在清单3.16中看到。

单个几何体着色器调用可以生成的数据量是有限制的。标量值的数量不能超过1024。这意味着必须将顶点结构中使用的标量值的数量相加，然后乘以maxvertexcount属性中提供的最大顶点数。此值必须为less大于或等于1024。稍后我们将看到，在使用流输出功能时，最多可以同时使用四个不同的流，但为了进行最大输出计算，我们可以简单地获取所使用的所有输出流的最大顶点大小，并将其乘以maxvertexcount属性。

3.8几何体着色器183

几何体着色器允许的第二个函数属性是实例属性。此属性激活几何体着色器实例化机制，在该机制中，传递到几何体着色器中的基本体将重复实例属性声明中指定的次数。然后，几何体着色器可以声明SV\_GSInstanceID（如上所述），以提供实例的唯一标识符。以这种方式创建的实例的最大数量是32个，这允许显著的数据放大。此实例化技术旨在简化几何体着色器程序，这些程序需要处理多个不同输出的几何体，例如同时渲染多个视点时。例如，在使用单个渲染过程生成立方体贴图时，可以使用几何体着色器实例化来生成基本体数据的六个副本，然后可以使用SV\_GSInstanceID系统值语义选择适当的变换矩阵。18

### 3.8.3几何体着色器阶段处理

几何体着色器处理流

在研究可在几何体着色器中执行的可能操作之前，必须首先阐明它如何使用流对象生成其输出。清单3.16中显示了一个示例几何体着色器，我们将在整个讨论中引用它。正如我们在前面的部分中所看到的，几何体着色器程序接收作为顶点数组的输入原语。顶点的顺序将有所不同，具体取决于管道在几何体着色器阶段之前生成的输入基本体的类型，分别如表3.1和图3.36所示。

一旦顶点数据在几何体着色器中可用，将执行一些计算以修改输入顶点或创建全新的顶点。清单3.16中的示例代码接收一个具有邻接关系的三角形作为其输入，然后简单地将主三角形传递给其流输出对象，忽略邻接关系信息。用于生成输出原语的方法是调用流输出对象的Append（）方法，该方法将输出顶点结构的实例作为其参数。有三种不同的流类型可用（一种用于三角形带，一种用于直线带，一种用于点列表），附加到流的一系列顶点将创建不同的基本体。例如，如果使用一个三角形流，并向其添加5个顶点，则“条带”顶点中将总共创建3个三角形。每个附加顶点将使用前两个顶点生成一个新三角形，以完成基本体。对于线条带流对象也存在相同的方法，这将为附加在第一个顶点之后的每个附加顶点创建附加的线条基本体。19

18 立方体贴图在第2章“Direct3D 11资源”的“使用2D纹理”一节中有简要介绍

19 这与本章“输入汇编程序”一节中解释的“条带”排序原则相同。

184                                                                                                                                                    3.渲染管道

如果需要重新启动三角形/线条带以创建分离的几何体组，则几何体着色器可以调用输出流的RestartStrip（）方法。这将重置原语生成，以允许启动另一个未连接到先前流几何体的条带。虽然在单个条带中生成几何图形更有效，但如果所需的输出几何图形必须位于多个断开连接的段中，则必须重新启动条带。通过将顶点附加到输出流来生成基本体的过程将继续，直到几何体着色器完成，这标志着着色器程序的此特定调用的结束。

多个流对象

Direct3D 11中的一个新功能是几何体着色器程序能够使用多达四个不同的流输出对象。正确配置后，这些对象不仅可用于将基本数据传递到光栅化器阶段，还可用于将其数据发送到流输出阶段，在该阶段数据可存储在缓冲区资源中。由于应用程序可以访问缓冲区资源，这意味着管道数据流可直接用于读取或存储在文件中。这是一个功能强大的工具，可用于一些有趣的数据分析场景。要在几何体着色器程序中声明多个流，只需将其他流对象作为输入参数包含到函数中即可。清单3.17演示了这种技术。使用AppendQ和RestartStripQ方法时，这些流的使用方式与单个输出流相同。每个输出流可以彼此独立地写入，它们不必具有相同的输出频率。20

WPJE.Z（4\*O7FSUFY W F S U T<>

JOPVU 1pJU4USFBN 0VU7FSUFYM NZ4USFBN JOPVU 1pJU4USFBN VU7FSUFY NZ4USFBN

{

OutVertexl myVert1=TransformVertexl（verts[0]）；0utVertex2 myVert2=TransformVertex2（verts[l]）；myStreaml.Append（myVertl）；

myStream2.Append（myVert2）；

}

清单3.17.声明多个输出流的几何体着色器，具有不同的顶点结构类型。

通常，当几何体着色器仅使用单个输出流时，该流的结果将传递到光栅化器阶段，在该阶段对其进行处理并拆分为多个输出流

20 第2章：Direct3D 11参考资料中提供了操作资源的技术，包括如何从C/C++读取资源的内容。

/

|  |
| --- |
|  |

|  |
| --- |
| Rgure 3.38.几何体着色器的多个流输出。 |

3.8几何体着色器185

片段。当几何体着色器程序中使用多个流时，只能选择其中一个声明流，以便在编译几何体着色器程序时通过应用程序设置的流配置传递到光栅化器阶段。必须将其余流配置为流到缓冲区（每个流一个缓冲区）连接到流输出阶段。传递到光栅化器的流也可以选择性地流到缓冲区。有关如何从应用程序配置这些流的详细信息将在“流输出”阶段部分中介绍。图3.38显示了几何体着色器的多个流输出。



当使用多个输出流时，一个非常重要的考虑因素是所有流类型都必须是点流-

传递到缓冲区资源的CE会单独收集，这也意味着不可能生成共享顶点的三角形或线。在这种情况下，给定的几何体集可能会消耗比通常用于线条或三角形条带的内存更多的内存。但是，如果几何体着色器中的算法生成已知的在顶点模式中，可以使用预加载的索引缓冲区将顶点引用为所需的基本体。除此限制外，如果将流发送到光栅化器阶段，它将仅显示为点。因此，具有多个输出流的大多数几何体着色器程序都不会将任何流光栅化，除非需要ss点渲染。

另一种可能的配置是使用单个流输出对象（可以使用三种流类型中的任何一种）并将其传递到光栅化器和流输出阶段。发送到流输出缓冲区的数据仍然作为非索引几何体存储，因此会导致如上所述的内存消耗增加。但是，如果对象正在光栅化，则可以使用三角形和线，而不仅仅是点。这些都是r开发人员，每种情况可能需要使用这些流的不同组合。

使用多个输出流时，最后要考虑的是对每个几何体着色器调用施加的标量值输出限制。我们已经看到有1024个标量值的限制，并且该限制适用于传递到输出流的标量总数。在多个流的情况下，我们取最大顶点ex size并使用它来确定可以传递到输出流中的最大顶点数。该阶段中只有一组输出寄存器，并且这些寄存器在所有输出流中共享。选择最大顶点大小，因为

186                                                                                                                                                      3.渲染管道



图3.39使用相同输出寄存器的多顶点结构。

用于接收输出顶点数据的输出寄存器将相互重叠。如图3.39所示。

由于流的输出频率不要求相等，因此没有简单的方法来确保动态分支不会产生一个执行路径，该路径将所有顶点输出的最大顶点流出来。因此，确保仅输出1024个标量的最简单方法是在编译时确定“最坏情况”顶点输出，只允许流式传输有效数量的顶点。流式输出的任何其他顶点只会减少生成的标量总数，确保遵守1024个标量的限制。

原始操作

几何体着色器的处理流程的另一个有趣结果是，虽然必须将基本体传递到该阶段，但它们可能传递到下一阶段，也可能不传递到下一阶段。这使得几何体着色器可以有选择地丢弃不需要的几何体，减少光栅化器要处理的几何体数量，并允许进行特殊的渲染操作hat只需要渲染模型的一个子集。一个很好的例子是渲染模型的轮廓以指示其高亮显示。

阴影体积

阴影体积是一种阴影技术，用于从光源中提取模型的轮廓。提取的轮廓向外展开，远离灯光，并将此展开的几何体栅格化到模具缓冲区中，以指示阴影体积内的像素。这是extrac算法的核心ted轮廓几何体形成一个体积。如果观看者在特定像素处仅看到阴影体积的一侧，则表示场景中的特定点位于阴影内部，不会被灯光照亮。如果观看者可以看到两侧21

21模具缓冲区及其操作将在本章后面的“输出合并阶段”部分中介绍。

3.8几何体着色器187



图3.40.将三角形拆分为三条线，以提供自定义线框渲染。

对于该像素处的阴影体积，则场景中的该点位于阴影体积的远侧，因此应被照亮。

在几何体着色器存在之前，必须在CPU上执行阴影体积挤出，然后将其传递到GPU以光栅化到模具缓冲区。但是，几何体着色器可以通过将三角形的面法向量与相邻三角形的面法向量相比较，轻松检测三角形何时位于轮廓上。如果当前三角形面向摄影机，且至少有一个相邻三角形背对着摄影机，则两个三角形共享的边应拉伸为阴影体积的一部分。如果三角形不是轮廓的一部分，则几何体着色器可以通过不输出其顶点将其丢弃。这样，几何体着色器通过从进一步处理中消除不必要的几何图形，减少管道下游的工作量。

点精灵

由于几何体着色器在其输入顶点数组中接收基本体，并且可以声明所需的输出流类型，因此输入和输出基本体类型没有限制。结合几何体着色器阶段生成可变数量顶点的能力，这允许对基本体进行完全控制ve类型转换。例如，传入几何体着色器的三角形可以转换为表示三角形边的三条线。这是实现线框渲染方案的简单方法。单个三角形边的创建如图3.40所示。

另一种流行的基本体转换技术是将点基本体转换为一对三角形，形成一个四边形。这些四边形可以应用纹理，从而有效地将点几何体转换为精灵。此过程通常称为创建点精灵，通常用于在粒子中生成粒子e系统具有更引人注目的外观。创建点精灵的过程如图3.41所示。第12章中使用了该技术的一个示例，在粒子系统示例中为粒子添加纹理。

188                                                                                                                                                     3.渲染管道



图3.41.从点原语生成点精灵。

实例化几何体

我们将要检查的几何体着色器的最后一个功能是其执行实例化的能力。如上所述，要创建的每个基本体的实例数是使用实例属性静态声明的。然后，可以通过SV\_GSInstanceID系统值语义来识别各个实例。这是一个相当通用的Abiliity、 这可以应用于许多不同的情况。例如，可以实例化一个点精灵并从另一个点精灵偏移，以放大最终渲染中出现的粒子数，同时将系统中的粒子数保持在较小的数量。

如何使用此功能的另一个很好的示例是同时为多个渲染目标创建几何体。当两个渲染目标的几何体必须应用不同的变换矩阵时，此功能尤其有效，例如立方体映射环境映射或双抛物面环境映射。分开创建基本体的实例，可以将其转换并发送到相应的渲染目标。如果需要两个完整的渲染过程，这将消除重复几何体着色器阶段之前所需的所有处理。22

### 3.8.4几何体着色器管道输出

我们已经讨论了使用几何体着色器的输出顶点流的许多方法，并注意到这些输出流是几何体着色器沿管道传递其结果的机制。但是，我们没有详细讨论附加到输出流的顶点可以包含哪些内容。因为几何体着色器阶段发生在光栅化器阶段之前，它必须生成具有SV\_位置的顶点

22 该几何体着色器实例化技术在第13章的示例程序中实现，该程序使用双抛物面环境贴图方法。

|  |  |
| --- | --- |
| 3.8几何体着色器 | 189 |

系统值语义。此外，此属性的内容必须包含顶点的投影后剪辑空间位置，以便正确光栅化。几何体着色器阶段之前的阶段可以将此属性与其他形式的位置（例如对象空间、世界空间或视图空间位置）一起使用，但几何体着色器必须将最终剪辑空间位置传递给光栅化器阶段。

除了每个顶点的位置外，几何体着色器还可以利用管道中在此阶段之前不可用的一对系统值。这些系统值中的第一个是SV\_RenderTargetIndex。此系统值提供一个无符号整数值，该整数值标识此原基所在的渲染目标切片ve应渲染为。如第2章所述，可以创建包含纹理数组的2D纹理资源。这些纹理中的每一个都称为纹理的一个切片。如果渲染目标绑定为输出到管道并包含多个切片，则几何体着色器可以动态决定哪个渲染对象要将特定基元应用于的目标。此选择要应用的渲染目标的方法与本节前面介绍的单遍环境映射示例密切相关。SV\_GSInstanceID值将用于通过将基元的值复制到SV\_Ren中来确定将基元的哪个副本传递到每个渲染目标中derTargetIndex。当然，绑定的渲染目标必须具有适当数量的切片，以适应几何体着色器中指定的实例范围。我们将在“输出合并”阶段一节中更详细地讨论将渲染目标绑定到管道。此系统值语义的使用如图3.42所示。



图3.42.使用几何体着色器与“SV\_uRenderTargetIndex”系统值同时写入多个纹理切片。

190                                                                                                                                                     3.渲染管道

几何体着色器可以使用的第二个系统值是SV\_Viewport ArrayIndex属性。该属性的操作方式与SV\_RenderTargetIndex类似，只是它确定将应用于哪个视口，而不是纹理切片。视口位于光栅化器阶段，因此尚未讨论。但是，we可以泛化视口以表示渲染目标内的子区域。可以同时绑定多个视口，每个视口表示渲染目标的不同部分。这些视口的用途是可以将场景的不同表示渲染到每个视口中。这通常适用于分屏渲染ng，其中屏幕的每一侧显示一名玩家对场景的视图。几何体着色器阶段的实例化功能还可用于提供用作视口数组索引的索引，有效地允许将多组几何体传递到多个视口。

## 3.9流输出

在geometry shader阶段中，我们看到了如何使用其stream output对象生成传递到光栅化器阶段的基本体流，在光栅化器阶段最终用于修改渲染目标的内容。但是，这不是输出流的唯一用途。我们还看到了如何发送流到流输出阶段，其中输出原语可以流到缓冲区资源中供以后使用。流输出阶段的唯一目的是提供一种连接几何体着色器和输出缓冲区资源以保存顶点数据的机制。流输出阶段在管道中的位置在中突出显示图3.43。

如上所述，可以同时使用多个流输出缓冲区。这允许几何体着色器生成其输出的四个不同版本，并为实现多种算法提供了很大的自由度。流输出阶段可以被视为固定函数管道阶段，尽管如此gh在阶段本身内不执行任何实际处理。相反，应用程序使用专门创建的几何体着色器程序配置流输出阶段，然后将所需的缓冲区资源绑定到阶段。由于它用作管道的出口点，因此没有管道阶段连接到流输出o输出阶段。



图3.43.流输出阶段。

3.9流输出191

### 3.9.1流输出管道输入

流输出阶段只能以几何体着色器程序中声明的输出流的形式从几何体着色器阶段接收信息。几何体着色器最多可以使用四个流将数据传递到流输出阶段，每个流可以接收不同的顶点结构作为其输入。这允许同时对相同几何体的多个变体进行流式传输。例如，位置、法向量、切线和双切线向量或其他可选逐顶点数据可以拆分为多个流，然后在以后有选择地绑定到输入汇编阶段，从而提供一种非常灵活的方法，仅绑定特定效果所需的顶点数据。

这些顶点结构的限制是，每个结构最多允许128个标量，并且几何体着色器调用不能产生超过1024个标量。因此，如果使用四个输出流，并且每个输出流使用128个标量顶点的结构，那么所有四个流总共可以写入512个标量两次。实际上，这应该是足够的数据流，因为它是为几何体着色器接收的每个基本体执行的。

也可以产生完全不相交的数据流。例如，如果算法要求所有正面必须与背面分开渲染，则一组几何体的正面和背面可以传递到两个不同的流。这为从标准几何图形格式生成这些混合几何图形集提供了一种简单的机制。

### 3.9.2流输出状态配置

流输出阶段需要两种不同类型的配置。首先，应用程序必须通过ID3DllDeviceContext:：SOSetTargets（…）方法将适当数量的缓冲区资源绑定到此阶段。此方法的操作方式与其他资源绑定函数类似，并且它允许在同一方法调用中同时绑定多达四个缓冲区。清单3.18演示了如何使用此方法。还有一个对应的Get方法，可用于检索对当前绑定到stage的缓冲区资源的引用。

ID3DllBuffer\*pBuffers[l]={pBuffer1}；

UINT aOffsets[1]={0}；

pContext->SOSetTargets（1，pbuffer，offset）；

清单3.18。将缓冲区资源绑定到流输出阶段。

192                                                                                                                                                    3.渲染管道

除了指示如何将缓冲区用于顶点缓冲区使用的任何其他绑定标志外，必须使用D3D11\_BIND\_stream\_output BIND标志创建将绑定到流输出阶段输出的每个缓冲区资源。此阶段中存储的缓冲区资源引用在管道执行之间保持不变，这意味着它们必须由开发人员管理。这与其他资源绑定功能使用的原理相同，也可以使用相同的管理技术。正如我们前面所看到的，总是绑定所有四个缓冲区条目（未使用的插槽接收空指针），或者让应用程序管理插槽的当前状态，根据需要仅修改适当的插槽，都是有益的。

流输出级运行所需的第二种配置是指定将哪些数据流传输到哪个输出插槽。此信息以D3D11\_S0\_声明\_条目结构数组的形式提供给ID3DllDevice:：CreateGeometryShaderWithStreamOutput（…）方法。清单3.19中指定了此结构的元素。

结构D3D11\u S0\u声明项{

乌因特河；

LPCSTR语义名称；

UINT语义索引；

字节起始组件；

字节分量计数；

字节输出槽；

} ;

清单3.19。D3D11\u S0\u声明项结构的成员。

在这个结构中，我们可以看到几何体着色器输出的每个顶点的每一部分所需的信息，这些信息最终将出现在一个流输出缓冲区中。必须为每个输出属性提供这些结构中的一个。第一个参数Stream标识数据将来自几何体着色器中的哪个流输出对象。SemanticName属性为要流化的属性提供几何体着色器中定义的语义，Semanticlndex属性为共享相同语义名称的多个属性提供索引，以便唯一标识。这与我们在顶点缓冲区元素声明中看到的语义索引类型相同。

StartComponent和ComponentCount参数确定将对四组件属性的哪个部分进行流式传输。StartComponent的值可以是{0,1,2,3}，它分别对应于寄存器组件{x，y，z，w}。ComponentCount只是指示要流化的组件数量。例如，如果StartComponent

3.9流输出193

如果为1，ComponentCount为2，{y，z}组件将被流化。此结构的最后一个参数是OutputSlot，它确定哪个流输出级缓冲区资源将接收流数据。有效值的范围从0到3，最多对应四个输出缓冲区。

一旦应用程序正确填充了这些结构的数组以定义要输出的信息，就可以创建几何体着色器对象。清单3.20显示了用于创建此特殊版本的着色器对象的方法。

HRESULT Create6EOMetryShaderWithStream0Output(

const void\*pShaderBytecode，

大小\u T字节码长度，

常量D3D11\u SO\u声明\u条目\*pso声明，

国际货币基金组织；

康斯特\*巴弗斯迈步；

纽姆斯特里德斯酒店，

UINT光栅化流，

ID3D11ClassLinkage\*pClassLinkage，

ID3D11GeometryShader\*\*ppGeometryShader

**);**

清单3.20。ID3DllDevice接口的CreateGeometryShaderWithStreamOutput（）方法。

我们最感兴趣的是看这些参数中的哪一个不同于普通几何体着色器对象创建方法。第一个新参数pSODeclaration是指向上述声明数组的指针。后面是NumEntries，表示数组中有多少个条目。接下来的两个参数pBuffer Streads和NumStrides提供将流式传输到每个缓冲区的每个顶点的顶点步长。这允许每个缓冲区使用不同的顶点格式来存储所需的信息。最后一个新参数是光栅化数据流，它提供应发送到光栅化器级的输出流的索引。如果没有任何流将被光栅化，则该参数应设置为常量D3D11\_SO\_NO\_光栅化\_流。无论为光栅化指定哪个流，它仍然可以向其中一个流输出缓冲区提供数据。

使用此方法创建几何体着色器对象并随后绑定到几何体着色器阶段时，将激活流输出阶段。然后，当管道执行开始时，传递到几何体着色器中的输出流中的任何数据都将根据其语义名称和语义索引与输出配置匹配。一旦找到匹配项，该数据将流式传输到适当的缓冲区，以创建完成的输出顶点。

194                                                                                                                                                     3.渲染管道

### 3.9.3流输出级处理

自动绘图

使用流输出功能有两种通用方法。第一种方法是实际使用生成的缓冲区资源作为输入汇编器阶段的输入，用于为管道执行提供顶点数据。在这种情况下，使用一个特殊的draw调用来执行管道。缓冲区资源必须绑定到输入汇编程序阶段插槽0，并且应用程序还必须配置相应的输入布局对象。然后ID3DllDeviceContext:：DrawAuto（）方法将检查绑定到输入汇编程序阶段的流式输出缓冲区的内容。然后，它向输入汇编程序提供适当数量的顶点和基本体，以呈现缓冲区的全部内容，而不需要应用程序的交互。在这种情况下，几何体着色器对象生成流化输出顶点，并且基本体连接信息由输入汇编程序的基本体类型配置确定。选择的基元类型应与用于流式输出数据的基元流类型兼容。如果随后使用DrawAuto方法进行渲染，则会在应用程序不知道缓冲区内容的情况下执行完整的渲染。这是GPU变得更加自主的另一种方式，可以在某种程度上独立于CPU运行。23

这种操作的效用可能不会立即显现出来。毕竟，如果几何体到达几何体着色器，我们可以同样轻松地将其输出流传递到光栅化器阶段并直接使用它，将网格流出来然后重新渲染有什么好处？要考虑的重要点是，在输入着色器和几何着色器之间执行的完整操作集合包含在由几何着色器生成的输出流中。可以保存在顶点着色器中执行的任何顶点变换，从而允许需要相同几何体的任何进一步渲染过程跳过顶点变换并直接继续到下一阶段。执行昂贵的顶点级别操作（例如顶点蒙皮）时，这可能会导致渲染过程中执行的计算数量显著减少。

这对于使用细分时需要多次渲染过程的算法也特别有用。如果模型的几何体首先在顶点着色器中进行变换，然后在细分阶段进行细分，最后在几何体着色器阶段进行流化，则结果可用于将来的渲染过程，而无需再次执行所有这些计算。如果使用非常高的细分级别，则细分阶段的计算成本可能非常高，这意味着以这种方式缓存细分网格可以节省大量计算。处理节省如图3.44所示。

23 如果使用三角形流或线流生成输出顶点数据，则传递到这些流中的条带将转换为基本体列表。这为DrawAuto（）方法提供了一致的数据格式。

3.9流输出195



图3.44。为多个渲染过程缓存细分几何体的好处。

作为调试工具的流输出

流输出阶段的另一个潜在用途是在几何体着色器阶段挖掘管道数据，并检索部分数据以进行调试。这有助于检查最终渲染输出中通常不可用的中间值。在这种情况下，需要两种不同的缓冲区资源。第一个是流输出缓冲区，它必须使用默认的资源使用标志创建。应用程序无法直接访问这种类型的缓冲区，因为无法创建既可以通过管道写入也可以从CPU读取的缓冲区。相反，我们使用staging usage标志创建第二个缓冲区，然后将流数据从默认使用缓冲区复制到staging usage缓冲区。然后，应用程序可以映射暂存缓冲区并读取其内容。

一旦应用程序可以访问这些内容，就可以使用标识流顶点布局的结构来访问这些内容。这通常是事先就知道的，因为无论如何，在执行管道之前都需要流输出配置。可以使用D3D11\_query\_DATA\_S0\_STATISTICS类型的管道查询对象检索写入缓冲区的原语数量（有关如何执行管道查询的更多详细信息，请参见附录B）。这允许应用程序知道管道生成的数据在缓冲区中的什么位置结束，以及缓冲区的默认或无效内容从何处开始。

由于缓冲区数据可以以评估的形式访问，应用程序可以以任何需要的方式使用数据。这可能包括将数据写入文件以进行脱机检查、分析数据（例如查找新数据的轴对齐边界框），甚至实时将统计信息打印到屏幕，以便实时显示通过几何体着色器传递的模型数据。这些类型的技术通常不会用于面向用户的应用程序，但它们可以在调试过程中提供许多调试选项

196                                                                                                                                                      3.渲染管道

应用程序开发。Direct3D 11应用程序的另一个主要调试工具PIX工具主要用于数据的脱机分析。这使得管道数据流成为提供实时调试信息的合适方法。此外，使用流输出功能可以保存几何体着色器中的中间值，这些值通常不可在PIX中读取。

### 3.9.4蒸汽输出管道输出

由于流输出级没有附加到其输出的另一个级，所以我们可以将其输出视为流到缓冲器的数据组成。这只是从管道中提取数据的几种方法之一，但根据管道的配置，它可以用作主要输出。例如，管道可以用作基于软件的渲染系统的转换和细分加速器。在这种情况下，将要渲染的几何体将由GPU进行变换和细分，然后流式输出并读回CPU，在CPU中可以在软件渲染器中使用。几何体永远不会被Direct3D 11光栅化，而是可以使用光线跟踪引擎或GPU上不可用的其他高质量技术进行渲染。

## 3.10光栅化器

几何体着色器阶段标记管道中严格处理几何体数据的最后一点。无论是直接从顶点着色器阶段、域着色器阶段还是几何体着色器阶段接收数据，几何管道数据都会馈送到光栅化器阶段。光栅化器级的位置如图3.45所示。

此阶段的主要目的是将几何数据转换为定期采样的表示形式，以便稍后应用于渲染目标。这个取样过程



图3.45。光栅化阶段。

|  |  |
| --- | --- |
| 3.10光栅化器 | 197 |



图3.46。已光栅化的三角形。

被称为光栅化，它将单个基本体映射为适合存储在纹理资源中的格式。光栅化的结果是产生大量近似原始几何体的碎片。这是生成有助于输出图像的单个像素数据的第一步。图3.46显示了光栅化三角形的外观。

除了执行光栅化操作外，光栅化器阶段在生成片段之前还执行许多其他操作。第一个操作是基本体剔除，它消除了由于其在剪辑空间中的位置而不利于输出渲染的基本体。通过在光栅化之前消除基本体，减少了要处理的基本体的总数，从而提高了工作台的效率。

第二个操作是剪切基本体。此过程会将基本体“剪辑”到剪辑空间中可能影响渲染输出的部分。这实际上是将部分位于可视区域的基本体切割为完全位于可视区域内的基本体。此区域之外的部分不需要，因为它不会产生可能影响输出渲染目标的片段。然后将剪裁的基本体映射到活动视口。视口是一种结构，用于描述当前渲染目标的子集，以接收已处理的基本体。最后，在光栅化过程中应用了剪刀测试。这本质上允许应用程序指定一个矩形区域，在该区域中允许进行光栅化。任何生成的、落在剪切矩形之外的片段都将被丢弃，而不是沿着管道进一步传递。当只使用光栅化输出的一个子区域时，这也提高了总体管道效率。

由于要执行所有这些操作，光栅化器阶段为管道数据的处理提供了大量功能。它会对整个管道的性能产生深远的影响，因此了解上述所有过程如何工作以及如何影响其运行是很重要的。光栅化器级的输出设置其余的基于像素的管道级，这使其成为数据放大点。

198                                                                                                                                                      3.渲染管道

### 3.10.1光栅化器级管线输入

光栅化器阶段从管道接收单个原语作为其输入。基本体数据由定义其几何形状的顶点组成。每个顶点必须至少包含一个具有SV\_位置系统值语义的属性。这是一个四分量齐次坐标，表示顶点在剪辑空间中的位置。在继续考虑光栅化器在这些输入基元上执行的个别操作之前，我们将进行一次短途旅行，以阐明什么是剪辑空间，以及几何如何转化为这个坐标空间。也许更重要的是，我们将考虑这个坐标空间的性质，以及它如何在栅格化场景中使用。

剪辑空间和规格化设备坐标

术语剪辑空间指的是投影后坐标空间，可用于确定在当前观看条件下场景的哪些部分可见。该空间本质上是当前投影变换的结果，通常是透视投影或正交投影。我们主要关注透视投影，并将在本节中重点介绍它们。然而，许多相同的概念也适用于正交投影，因此下面的讨论在这两种情况下都是有效的。24

如上所述，SV\_位置系统值语义中提供的位置数据是由四个分量组成的齐次坐标。通常，此位置的前三个组件指定顶点当前所在三维空间中的X、Y和Z坐标。第四个分量称为W坐标，在对其应用投影变换之前，其值始终为1。因此，位置坐标始终采用[X，Y，Z，\]等形式。对这些点应用投影变换后，生成的形状将取决于执行的投影类型。如果使用透视投影，则生成的点将根据投影变换的近剪裁平面和远剪裁平面生成一个W值，该值等于投影前的Z分量，而Z分量是其自身的缩放版本。方程式（3.2）和（3.3）显示了基于方程式（8.7）中所示投影矩阵对输入Z（和W）分量进行投影变换的结果。这里我们假设输入坐标由下标1表示，并且后投影25

24 第8章“网格渲染”对一组几何体通过的各种变换进行了一般性讨论。其中包括对透视投影的讨论。

25 这假设使用的投影矩阵与DXSDK文档中描述的相同，也与第8章中描述的相同。

3.10光栅化器199

坐标用下标p表示。使用以下方程式：



(3.2)

(3.3)

方程（3.2）的结果在第一次检查时可能不明显，但我们可以通过在除以W后绘制z作为a来了解该方程。输入Z坐标的功能。如图3.47所示。这里我们可以看到，投影后的Z值可以位于三个一般区域之一。如果输入的Z坐标小于近剪裁平面Z，则生成的Z具有负值。如果输入Z坐标介于Z和Z之间，则结果Z的值介于0和Z之间。任何大于Z的输入Z坐标都会生成大于Z的Z*NPNFPFFPF*26

在投影后的某个点，必须通过除以W坐标来重新调整这些投影后坐标。通过将所有坐标除以W坐标，我们确保分割后的W坐标为1。我们还可以考虑通过简单地将Z所讨论的结果除以Z的z坐标的结果，当Z等于z时，结果是0。当Z等于Z时，结果为1。因此，在除以W之后，我们有一个有效的深度范围，其值在0和1之间，对应于Z和Z之间的输入值，我们不会在这里重复计算，但是X坐标和Y坐标会产生类似的行为，每个坐标都会产生-1到1之间的有效值*11N,1FNF*



图3.47。位置的投影后Z坐标。

26 所有这些方程都假定为左手坐标系，其中Z在视图方向上为正。这是Direct3D中使用的标准坐标系。

200                                                                                                                                                      3.渲染管道



图3.48。从一个坐标空间到另一个坐标空间的变化。

范围在后W分割范围内。因此，我们有两种不同的方法来表示W分割前后的投影后坐标。我们将把预分割坐标称为驻留在剪辑空间中的坐标，把后分割坐标简单地称为标准化设备坐标。这种区别非常重要，因为两个坐标描述的是同一点，但它们的值在这两个坐标之间会有很大差异。各种坐标空间及其产生的操作如图3.48所示。

如上所述，光栅化器级在SV\_位置系统值语义中接收到的位置应该是投影变换的结果，因此应该在剪辑空间中。光栅化器阶段稍后在其处理序列中执行W分割，因此执行投影变换的着色器程序不需要手动执行分割。在许多情况下，在考虑特定顶点是否可见时，通常更容易根据标准化设备坐标可视化投影后坐标。在这些标准化设备坐标中，顶点必须位于点[1,1,1]和[-1，-1,0]之间的立方体内。图3.49描述了坐标系的范围。27

如图3.49所示，正X轴指向右侧，正7轴指向上方，正Z轴指向上方

图3.49。顶点位于当前查看区域内的区域的可视化。

27 这并不意味着着色器无法执行W除法（如果需要进一步计算）。如果是这样，光栅化器阶段将简单地将其他坐标除以1，这当然会产生相同的结果。

3.10光栅化器201

进入现场。该区域通常被称为单位立方体，尽管由于其在Z方向上的范围，实际上它不是一个完整的单位立方体，但它仍然通常以这种方式被称为单位立方体。我们将在“光栅化器阶段处理”部分进一步探索剪辑空间和归一化设备坐标。

裁剪和剔除距离

现在我们已经了解了光栅化器接收到的位置数据应该如何看待，我们可以考虑它可能接收到的其他输入。我们将研究的下两个输入属性最初在“顶点着色器”部分中讨论。系统值属性SV\_ClipDistance和SV\_CullDistance都在光栅化器阶段中用作消隐和剪裁操作的附加用户定义输入。这些允许使用可编程着色器程序提供的自定义数据操纵剔除和剪裁过程。我们将在后面的“光栅化器阶段处理”部分更详细地讨论这些问题。

视口数组索引

除了剪辑和消隐距离外，光栅化器级还可以接收SV\_uViewPortarAyindex系统值。这是一个无符号整数，用于指定光栅化器在光栅化基本体时应使用的视口定义结构。这些单独视口如何影响输出的机制将在“光栅化器阶段处理”部分中详细描述。

渲染目标数组索引

我们还看到，几何体着色器可以指定SV\_RenderTargetArrayIndex系统值语义，以指示要写入渲染目标阵列的哪个纹理切片。这只是一个无符号整数值，用于选择纹理切片作为目标。本书中我们没有实现任何使用此系统值语义的示例程序，但DXSDK中有一个非常好的Direct3D 10示例，它演示了正在使用的此技术，称为CubeMapGS。此外，在（Zink）中还可以找到关于如何使用此功能的进一步说明。

附加输入属性

可以发送到光栅化器阶段的最后一组输入实际上是各种不同类型的信息。光栅化器接收到的顶点结构中的任何附加属性表示开发人员在输入原语的每个顶点定义的数据。对于三角形，每个三角形的每个顶点都有三个属性实例。因为光栅化过程会产生碎片

202                                                                                                                                                      3.渲染管道

如果表示这些顶点之间的常规采样模式，则必须对三个顶点属性值进行插值，以找到适当的中间值传递给每个片段。这是在实际的光栅化过程中执行的，将在本节后面讨论。

### 3.10.2光栅化器级状态配置

由于光栅化器级中有如此大量的不同功能可用，因此需要配置相应大量的状态以控制其操作。这些配置由应用程序通过三组不同的方法控制，这些方法可用于查询现有状态或用新状态覆盖现有状态。

光栅化状态

光栅化器阶段的主要配置使用光栅化器状态对象执行。此对象由ID3DllRasterizerState接口表示，它是一个不可变的状态对象，在创建时进行验证。与Direct3D 11中的所有对象一样，该对象是通过设备方法创建的，在本例中是通过ID3D11Device:：CreateRasterizer S t a t e（…）方法。与其他状态对象一样，此方法采用指向提供所需配置值的描述结构的指针。清单3.21提供了一个描述声明，随后创建了一个光栅化器状态对象。然后读取当前设备上下文中的现有状态，然后设置我们刚刚创建的新状态。

D3D11\_光栅化器\_状态rs；

rs.FillMode=D3D11\u FILL\u S0LID；

rs.CullMode=D3D11\u CULL\u BACK；

rs.正向逆时针=假；

rs.深度偏差=0；

rs.SlopeScaledDepthBias=0.0f；

rs.深度偏差放大器=0.0f；

rs.DepthClipEnable=真；

rs.ScissorEnable=false；

rs.MultisampleEnable=假；

rs.AntialiasedLineEnable=false；

ID3DllRasterizerState\*pState=0；

HRESULT hr=m\_pDevice->CreateRasterizerState（&rs，&pState）；

清单3.21。光栅化器状态对象的指定和创建。

***/***

3.10光栅化器



203

图3.50。线框和实体填充模式之间的差异。模型由放射性软件有限责任公司提供。由托马斯·德林诺夫斯基、丹尼·格林创作。www.radiotic-software.com

我们将分别检查光栅化器状态的每个元素。第一个设置是FillMode参数，它确定如何光栅化基本体。可用选项有实体填充或线框。在实体填充中，在基本体的顶点之间生成片段以完全填充其内部，而在线框模式中，仅对基本体的边进行栅格化。实际上，这只会影响三角形基本体，因为直线仅由一条边组成，而点仅由一个点组成，因此它们的填充模式是等效的。图3.50显示了三角形这两种模式之间差异的简单示例。

此结构中的第二个设置是CullMode。消隐模式控制光栅化器级的消隐操作。这可用于剔除面向查看器（正面）或背向查看器（背面）的基本体。还提供了完全禁用剔除操作的选项。通过检查三角形顶点到达的顺序来确定三角形是正面还是背面。如果顶点的顺序使按顺序遍历它们产生围绕渲染目标上三角形的顺时针移动，则它具有顺时针顶点顺序。否则，三角形称为逆时针缠绕。“向前-逆时针”设置确定应将前向三角形视为顺时针还是逆时针。实际上，此设置与消隐模式一起用于选择消隐操作的目标。

接下来的三个设置控制一个可选功能，为光栅化器生成的碎片提供深度偏移。某些算法要求以两种不同的方式将对象渲染到两个单独的渲染目标，以实现特定效果。阴影贴图可能是最常见的示例，其中从光源的角度渲染对象，以生成场景中哪些对象对灯光可见的贴图。此渲染生成灯光的深度贴图（由其确定）

204                                                                                                                                                    3.渲染管道



图3.51，从灯光角度和相机角度光栅化的场景不匹配。

每个像素与灯光的距离。在随后的渲染过程中，将从观察者的角度渲染对象，并从第一个角度对深度贴图进行采样，以确定该对象是否可以被灯光“看到”，从而从灯光接收灯光。理论上，这是一种非常明智的方法来确定哪些物体在阴影中，哪些不在阴影中。但实际上，由于两个渲染过程的有效采样模式不同，此方法可能会引入许多瑕疵。图3.51演示了这一点，它显示了对象的俯视图，以及如何通过灯光的深度贴图和场景渲染将其光栅化。28

为了对抗这种效果，光栅化器阶段提供了一种方法，将深度偏差引入生成的碎片中。DepthBias和SlopesCaledepthbias设置分别提供常量和基于坡度的偏移深度值。这两个参数以两种方式之一应用深度偏移，具体取决于所使用的深度缓冲区的类型。方程式（3.4）和方程式（3.5）提供了两种深度计算。

偏差=（浮动）深度偏差\*r+斜率刻度偏差\*MaxDepthSlope；

(3.4)

偏差=（浮动）深度偏差\*2（指数（原语中的最大z）-r）+

坡度CaledDepthBias\*最大深度坡度；(3.5)

28 在使用阴影贴图时，已经设计了很多很多算法来提高生成的图像质量。在学术文献中搜索将很容易返回100多篇关于该主题的不同论文。

3.10光栅化器205

当使用unorm格式的深度缓冲区时，或者如果没有深度缓冲区绑定到管道，则使用用于计算深度偏移的第一种技术。第二种方法在使用浮点深度缓冲区时使用。在第一个等式中，r参数表示可以表示的大于0的最小值，MaxDepthSlope是在基本体上发现的最大深度坡度。在第二个等式中，r参数表示深度缓冲区浮点类型中的尾数位数。在任何一种情况下，都会计算该偏差值，然后将其钳制到光栅化器状态描述的DepthBiasCamp参数。最后，在剪裁之后和执行插值设置之前，将偏移值添加到顶点的z值。添加偏移的效果可以有效地移动光栅化曲面，以确保计算中的微小差异不会产生伪影。

光栅化器状态描述的其余四个设置都是布尔参数，用于打开或关闭各种功能。我们将在“光栅化器阶段处理”一节中更详细地探讨这些参数中的每一个，但在这里我们将简要介绍它们。这些设置中的第一个是DepthClipEnable参数。这将根据基本体的深度值启用或禁用基本体的剪裁。这实际上是将几何体剪裁到近剪裁平面和远剪裁平面。接下来是ScissorEnable参数，它启用或禁用scissor测试的使用。剪刀测试用于剔除剪刀矩形之外生成的任何碎片。第三个布尔参数是multisampleEnable，它本质上切换光栅化器是否对MSAA渲染目标执行多个覆盖率测试。最后，AntialiasedLineEnable在光栅化线条图元时启用或禁用抗锯齿。

视口状态

除了主光栅化器状态外，应用程序还必须提供至少一个视图端口，以便在渲染操作中使用。视口结构由六个浮点参数组成，这些参数标识将光栅化到的当前渲染目标的子区域。清单3.22显示了该结构的内容。

结构D3D11\_VIEWP0RT{

浮动TopLeftX；

上下浮动；

浮动宽度；

浮动高度；

浮心；

浮动最大深度；

}

清单3.22。D3D11视口结构的成员。

206                                                                                                                                                     3.渲染管道

如清单3.22所示，视口结构定义了一个区域，规范化设备坐标将映射到该区域，以便从单位立方体的单位大小坐标转换为渲染目标的基于像素的坐标系。例如，如果视口应覆盖整个渲染目标，则“顶部”和“左侧”值将为0，“宽度”和“高度”参数将为所使用的渲染目标的宽度和高度。MinDepth和MaxDepth参数用于将深度值的范围缩放为整个常规范围的子集。

应用程序可以提供单个视口，也可以同时指定多个视口。如果设置了多个视口，则用于特定基本体光栅化的视口由SV\_ViewportArrayIndex系统值确定。视口的绑定使用ID3DllDeviceContext:：RSSetViewports方法执行。与前面讨论的状态数组不同，任何未在最近的set调用中设置的视口都将被清除。此外，还有用于检索当前设置的视口数组的常用获取对应方法。

剪切矩形状态

光栅化器阶段中可用的最后一种配置是能够指定一组剪式矩形。这些矩形在剪切测试期间用于指定渲染目标的特定区域，可以为该区域生成片段。这有效地消除了为当前剪切矩形之外的区域生成的任何片段，这可以减少不必要的计算。如果绑定了多个剪切矩形，则用于给定基本体的矩形由相同的SV\_uviewportarraylndex确定。这将确保同时使用的视口和剪刀形矩形是匹配的一对。剪式矩形阵列的管理方式也与视口阵列相同。可以使用ID3DllDeviceContext:：RSSet ScissorRects（）方法对其进行绑定，从而仅使在最近的set调用中指定的剪刀矩形保持活动状态。

### 3.10.3光栅化器级处理

在光栅化器阶段执行的所有处理都是将几何数据有效转换为适合存储在渲染目标中的基于图像的数据的重要组件。本节将更详细地探讨这些过程中的每一个过程所包含的内容，并试图深入了解如何在常见的实时渲染上下文中使用这些过程。图3.52显示了在光栅化器阶段内执行的功能操作的框图。这些过程可以在实际的GPU硬件中以不同的顺序实现，但是我们将在概念上考虑它们在所示序列中。

3.10光栅化器207



图3,52。光栅化器级执行的功能操作。

如图3.52所示，光栅化器接收完整的单个原语作为其输入。在管道中的这一点上，任何具有邻接关系的基本拓扑都将转换为标准基本拓扑。如果原语作为条带传递到管道中，则这些条带将被分开并分别发送到光栅化器阶段。然后以顺序管道方式对这些基本体进行处理，处理过程以生成片段结束，然后将片段传递到像素着色器阶段。以下各节将更详细地探讨在每个功能块中执行的操作。

挑选

对传入原语执行的第一个操作是剔除。此操作旨在从进一步处理中删除不会对最终渲染图像产生影响的基本体，以提高管道的效率。有两种不同类型的剔除：背面剔除和我们称之为基本剔除。

**背面剔除。**我们将研究的第一种剔除形式是背面剔除。顾名思义，此操作仅适用于三角形基本体，因为它们是唯一具有面概念的基本基本体。但是，三角形通常是最常用的基本体，因此此过程对于大多数渲染序列非常重要。当光栅化器级接收到输入基本体的顶点时，确定顶点的缠绕。图3.53显示了两个不同的三角形，其中顶点缠绕为顺时针或逆时针方向。“顶点着色器管道输出”部分介绍了执行此检查的一种技术，但硬件实现可能有所不同。

208                                                                                                                                                      3.渲染管道



图3.53。具有不同顶点缠绕的两个三角形。

我们已经看到，在光栅化器状态配置中，消隐操作有两种不同的设置：消隐模式和正向逆时针。指定哪个顶点缠绕表示前向基本体取决于在输入汇编程序阶段引入管道的输入数据。用于生成输入几何图形的数字内容创建工具可以使用任意一种缠绕，但通常可以选择反转缠绕，以使模型与最终用户应用程序的约定兼容。反转顶点缠绕顺序是通过从基本体顶点列表中的当前位置交换两个顶点来执行的。

一旦front逆时针参数指定了表示基本体正面的约定，CullMode参数将确定应剔除哪些面（如果有）。消隐用于通过消除背向当前视点的三角形来减少光栅化阶段所需的工作量。如果三角形背向当前视图，则视为背向。在传统的不透明渲染算法中，背面对最终图像没有影响，应该消除。事实上，进入此阶段的基本体中大约有一半将被删除，因为在一个帧内的单个视图中只能看到模型的一半。在这种情况下，CullMode参数应设置为D3D11\_CULL\_BACK。

同时，有许多算法需要使用不同的消隐模式多次渲染几何体以实现不同的效果。例如，在每个像素处查找对象厚度的一种简单技术是，首先使用剔除的前表面对其进行渲染，并将其写入渲染目标的红色通道，同时使用在写入渲染目标时选择最大值的混合状态。执行第二个渲染过程时，将剔除背面并将其写入渲染目标的绿色通道，同时使用混合状态，该状态在写入渲染目标时选择最小值。在这两个过程之后，渲染目标包含红色区域中最远的点29

29混合将在本章的“输出合并”一节中详细介绍。

3.10光栅化器209



图3.54。通过使用不同的剔除模式，对对象的最小和最大深度进行描述。

通道，以及绿色通道中最近的点。这两个值之间的差值共同提供了对象厚度的估计值。这一概念如图3.54所示。在这样的算法中，在一个过程中剔除一个方向的三角形，然后在第二个过程中剔除相反方向的三角形是必要且有用的。通过操纵剔除顺序可以实现许多特殊效果。30

**原始淘汰。**在光栅化器阶段执行的第二种消隐形式测试是否有任何原语完全位于归一化设备坐标中的单位立方体之外。这可以通过测试基本体的所有顶点是否位于同一剪裁平面之外保守地确定，其中剪裁平面由单位立方体的面定义。当在剪辑空间中执行此操作时，可以非常有效地执行此操作，因为平面是轴对齐的，并且始终位于与原点相同的距离处。例如，要测试三角形的三个顶点是否都在单位立方体的顶平面之外，测试仅包括检查其剪辑空间位置的F分量是否大于其W分量。可通过取W分量减去Y分量的差值来进行试验。如果所有结果均为负值，则可以安全地从进一步处理中丢弃原语。这31

50还有其他更新的技术可以在单个过程中执行此深度计算，但该示例仍然是一种有效的情况，其中需要使用切换消隐顺序进行多个渲染过程。

; 在哪个空间中执行此操作取决于硬件实现，并且可以在剪辑空间中执行，也可以不在剪辑空间中执行。

210                                                                                                                                                     3.渲染管道



图3.55。基本消隐测试的描述，其中三角形的所有三个顶点必须位于要消隐的同一平面之外。

对单元立方体的每一侧执行检查。该剔除过程如图3.55所示。

除了这种标准形式的剔除外，开发人员还可以使用自定义剔除算法。我们已经在顶点着色器部分看到了系统值属性SV\_CullDistance。如果一个基本体的所有顶点到达光栅化器阶段时，在同一个属性组件中使用SV\_uCullDistance语义标记负值，则该系统值语义将剔除该基本体。可以使用用户定义的平面方程生成该值，执行距离测试，然后将标量结果存储在SV\_CullDistance属性中。实际上，用于生成剔除值的计算类型没有限制。如果要应用更合适的测试，则可以根据需要实施该测试，并以相同的方式存储结果。例如，在生成抛物面环境贴图时，可以剔除与当前生成的相反半球中存在的几何体（第13章将详细讨论抛物面贴图）。

基本剪裁

光栅化器阶段执行的下一个操作是基本体剪裁。如果每个原语完全位于单元多维数据集内部，或者仅部分位于单元多维数据集内部，则此操作将获取在剔除操作和测试中幸存下来的每个原语。如果原语完全位于单元多维数据集之外，则可以直接丢弃它，而无需进一步处理。如果该图元部分位于内部，则该图元将拆分为完全位于单元立方体内的新图元，而外部图元将被丢弃。该程序如图3.56所示。由于单位立方体对应于当前投影矩阵的视锥体，因此此操作也称为视锥体剪裁。

3.10光栅化器211



图3.56。剪辑过程的描述。

此过程的实现取决于硬件，但它将在单位立方体的边界处生成新的顶点，以允许创建新的基本体。在这个过程中要考虑的一个重要点是这些生成的顶点接收来自原始顶点的插值属性值，这将随后用于执行实际的光栅化过程。

与剔除机制一样，开发人员也可以使用基本剪裁硬件来执行定制的逐片段剪裁。通过使用SV\_ClipDistance系统值属性，每个顶点可以指定其相对于用户定义的剪裁函数的位置。这可用于实现自定义剪裁平面，或提供适当剪裁结果的其他功能。这些属性在原语中插值，并执行逐片段剪裁测试。如果插值属性值为负值，则丢弃片段。可以通过使用顶点属性的多个组件来实现多个剪裁功能。例如，如果顶点属性声明为float4类型，则可以使用每个组件中的一个结果指定四个不同剪裁函数的结果。这允许同时执行大量测试。

同质分割

原语通过消隐和剪裁操作后，将执行同质分割。正如我们前面所讨论的，这个过程只是用投影点的W分量来划分投影点，从而生成形式为[X/W，Y/W，Z/W，1]的同质坐标。此格式称为标准化设备坐标，因为每个坐标的大小已按W坐标缩放。在此转换之前，坐标仍在剪辑空间中，这是由投影矩阵生成的形式。输入剪辑空间点的W坐标值可能为1，也可能不为1。

212                                                                                                                                                      3.渲染管道



图3.57。将归一化设备坐标中的单位立方体映射到渲染目标的屏幕空间坐标系。

视口变换

实际光栅化发生之前的最后一步是视口变换。基本体的顶点当前位于标准化设备坐标中，其中X和Y坐标范围在[-1,1]之间，Z坐标范围在[0,1]之间。视口提供将基本体从标准化设备坐标映射到像素坐标所需的信息。视口数据结构提供偏移（使用TopLeftX和TopLeftY参数）和比例（使用宽度和高度参数），用于将基本体定位在渲染目标的所需区域内。该映射过程如图3.57所示。

执行映射后，对于X组件，新坐标的范围为[TopLeftX，TopLeftX+宽度]，对于Y组件，新坐标的范围为[TopLeftY，TopLeftY+高度]，其中宽度和高度是要渲染到的渲染目标区域的尺寸。也可以缩放位置的Z分量，以操纵稍后在管道中使用的深度值。如前所述，可以使用多个视口来实现诸如分屏渲染之类的效果，其中场景被光栅化为渲染目标的两个不同区域，以指示从每个玩家的当前视点可以看到什么。但是，这也可以用于仅将场景渲染到用户界面元素未覆盖的区域。例如，在实时战略（RTS）游戏中，在屏幕底部有一个大的用户界面元素是很常见的。如果此用户界面不透明，则它可以显著减小要光栅化的区域的大小，并随后减少需要生成的片段/像素的数量。通常，如果有理由限制使用视口渲染的区域，那么最好这样做！

3.10光栅化器213



图3.58。单个渲染目标内的多个视口，允许将几何体栅格化到渲染目标的特定区域。

应用程序可以通过两种不同的方式选择所需的视口。如果光栅化器阶段的输入签名（由前一阶段声明的输出确定）包含SV\_ViewportArrayIndex系统值语义，则该值用于从当前绑定的视口数组中进行选择。如果输入签名中不存在此系统值语义，则系统默认为位于索引0处的视口。因此，如果只绑定了一个视口，并且不需要对视口进行进一步操作，开发人员只需从光栅化器输入中排除SV\_ViewportArrayIndex即可选择主视口。图3.58演示了多个视口的使用以及如何使用它们来选择多个区域进行光栅化。

视口变换完成后，顶点位置将拆分为其X坐标和7坐标，用于标识由基本体覆盖的渲染目标区域及其Z坐标，这是一个深度值，稍后在深度缓冲系统中用于确定单个片段的可见性。

光栅化

光栅化器阶段的最后一个操作是对管道中到达该点的基本体执行实际的光栅化。光栅化器的主要用途是将其接收的几何数据转换为离散采样数据，该数据近似于渲染目标内的几何数据。这可以被认为是一个采样过程，其中几何体可以被视为连续数据，而输出片段是几何体以规则间隔的数字表示。图3.59中显示了许多示例原语。

光栅化过程包括两个不同的任务，片段生成和属性插值。我们将分别讨论这些项目，然后

*Z&apos;~*                                                                                                                                                      3.渲染管道



图3.59。各种光栅化几何图元。

还可以更详细地了解在启用多采样消除混叠（MSAA）的情况下使用此过程的一些细节。

**片段生成。**光栅化的第一步是确定渲染目标的哪些像素被当前基本体覆盖。此操作通过使用一组光栅化规则来执行，这些规则根据要光栅化的基本体的类型以及渲染目标的类型而有所不同。如果渲染目标支持MSAA，则光栅化过程与使用标准渲染目标时不同。在引入标准概念之后，我们将在本节后面讨论MSAA的含义。对于表示渲染目标的像素网格，光栅化器阶段必须确定哪些像素被基本体“覆盖”。这里使用引号是因为在许多情况下，选择像素位置进行光栅化，但它并没有完全被原语覆盖。

图3.60。多个基本体边界处的像素。



为了理解这是如何发生的，我们首先需要探索片段生成是如何执行的。我们将从三角形光栅化过程开始。渲染目标的每个像素使用像素中心的一个点作为光栅化的关键点。由于它位于像素的中心，并且每个像素的大小为1个单位（在屏幕空间中），因此像素的中心距离像素边界的边缘为0.5个单位乘0.5个单位。当一个像素中心被一个原语完全覆盖时，它会被清晰地选择以产生一个片段。由于像素中心也可以落在基本体的一个边界边上，因此有些

|  |
| --- |
|  |

|  |
| --- |
| 图3.61。用于选择用于光栅化的像素的基本体边法向量的角度范围。 |

3.10光栅化器215



在测试中必须使用其他规则，以确保选中两个相邻三角形中的所有像素，并且相邻图元不会多次选中任何像素。图3.60展示了其中一些情况。

三角形光栅化规则指定从基本体上选择落在基本体上边缘或左边缘上的任何点进行光栅化。顶边是任何完全水平的边，而左边是任何角度向左的边。如果考虑允许选择像素的边缘范围的法线向量，则当边缘驻留在像素中心时会产生一个片段，如图3.61所示。

线光栅化与三角形光栅化有些不同。它可以遵循两条路径之一，即锯齿路径或抗锯齿路径。锯齿线光栅化使用相当简单的算法来确定线覆盖了哪些像素，而抗锯齿线光栅化则执行更复杂的算法来确定线覆盖了多少像素，然后将该像素的输出颜色乘以该覆盖因子。锯齿线光栅化或非锯齿线光栅化的选择由光栅化器状态的AntialiasedLineEnable参数确定。我们将检查锯齿光栅化的细节，同时保留反锯齿线。抗锯齿线算法的实现是特定于硬件的，因此在这里尝试详细解释是没有意义的。

锯齿线光栅化的执行如下。该过程不是使用基于点的算法，而是从在每个像素上刻一个菱形开始。如图3.62所示。如果直线的坡度在-1<=坡度<=1的范围内，则该直线被视为x-主线。在这种情况下，如果直线本身与菱形的左下角或右下角边缘或下角相交，则选择像素生成碎片。如果



图3.62。线光栅化的一些示例，右侧描述了两种菱形选择可能性。

216                                                                                                                                                      3.渲染管道

图3.63。将点光栅化为一对三角形的几个示例。



该线的坡度超出了上述定义的范围，因此被视为y-主线。在这种情况下，执行相同的测试，但右角也包括在测试中。这些交叉点图像如图3.62右侧所示。

可以光栅化的最后一种基本体类型是点基本体。点基本体栅格化实际上使用与三角形栅格化相同的规则，从概念上讲，它将自身扩展为两个三角形，在其点位置周围形成一个1 x 1的正方形。然后将这些三角形完全栅格化，就像它们是三角形基本体一样。该模式的几个示例如图3.63所示。

**剪刀试验。在里面**在所有这些情况下，除了满足光栅化规则外，原语还必须通过剪切测试。正如我们在“光栅化器状态配置”一节中所看到的，应用程序可以通过将一组剪切矩形绑定到光栅化器阶段来配置剪切测试。然后，光栅化器阶段根据用于选择视口的相同条件选择适当的剪切矩形。如果SV\_ViewportArrayIndex被声明为光栅化器阶段的输入属性之一，那么它将用于从数组中进行选择。否则，光栅化器默认为第一个条目。这种选择机制确保视口索引和剪刀形矩形索引始终相同，这使得应用程序可以轻松地同时引用这些对象对。

剪刀测试的工作原理是将碎片的X和Y分量与剪刀矩形进行比较。如果片段落在矩形之外，则将对其进行剔除，并且不会对渲染目标作出贡献。还需要注意的是，根据硬件实现，光栅化器操作序列中剪刀测试的准确位置可能会有所不同。这可能会对性能产生一些影响，因为从中剔除片段的位置可变。但是，应用程序可以确保剪刀矩形之外的像素不会写入渲染目标。

**属性插值。**在识别受当前原语影响的片段并通过剪切测试后，光栅化器阶段必须确定为其生成的每个片段生成哪些属性数据。要计算这些属性值，光栅化器将从其输入基本体顶点插值每个输入属性。每个基元的输入属性值根据距像素中心的距离贡献插值片段输出属性值。这意味着片段离基本体中的某个顶点越近，该顶点对在该位置生成的属性的影响就越大。图3.64给出了一个简单的示例。除了要插值的常规属性外，还将插值每个片段的深度，并将在深度测试的管道中稍后使用。

3.10光栅化器217



图3.64。栅格化三角形的几个像素，演示像素与每个顶点的接近程度如何影响像素。

默认情况下，用于生成这些每片段属性的插值类型是透视正确的线性插值。但是，可以通过在像素着色器输入属性的类型声明之前添加插值修改器来修改插值模式。表3.2列出了可用的插值模式，并对其行为进行了简要说明。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **插值模式** | **描述** | |  |  |  |  |  |
|  |  | | | | | |  |
| **线性的** | **提供线性、透视正确的插值。插值是基于** | | | | | |  |
| , |  |  | . | , |  |  |
|  | 像素的中心。 | | | | |  |  |
|  | 像素的中心。 | | | | |  |  |
| **质心** | **提供线性、透视正确的插值。插值是基于** | | | | | |  |
| , | ., | \_. , |  | , | .  , |  |
|  | 像素覆盖区域的质心。 | | | | | |  |
|  | 像素覆盖区域的质心。 | | | | | |  |
| **无极性** | **不提供插值。属性作为常量传递。** | | | | | |  |
|  |  | | | | | |  |
| **没有远见** | **提供线性插值，而不考虑透视效果。这个** | | | | | |  |
| . | , • | • , | , | , | .  , |  |
|  | 插值基于像素的中心。 | | | | | |  |
|  | 插值基于像素的中心。 | | | | | |  |
| 样品 | **提供线性、透视正确的插值。插值是基于** | | | | | |  |
| **MSA将显示采样位置，而不是像素的中心。** | | | | | |  |
|  |  |
|  |  |  |  |  |  |  |  |

表3.2。可用的属性插值模式。

多采样注意事项

Direct3D 11的光栅化管道是渲染3D几何体的有效方法，其性能适合实时应用。然而，光栅化的一个基本缺点是生成的图像可能会出现明显的锯齿伪影。混叠是信号处理领域的一个术语，指的是

218                                                                                                                                                      3.渲染管道



图3.65。用于非MSAA和MSAA渲染的光栅化采样点。

当连续信号的采样频率不足以再现原始信号时发生。如果我们用这样的术语描述光栅化，“信号”将是3D几何体的矢量表示，而“采样频率”是渲染目标的X和Y分辨率。在更一般的术语中，我们可以说出现混叠是因为我们的几何体最终必须使用二进制覆盖率测试渲染为离散的矩形网格。矩形网格永远无法完美地表示不完全水平或垂直的边，因此会出现锯齿状的“阶梯”图案。这种类型的锯齿伪影通常称为边锯齿，因为它发生在三角形边上。另一种类型的混叠是由于每个像素只执行一次像素着色器，因此会对材质和BRDF属性产生的曲面颜色进行离散采样。这种类型的锯齿称为着色器锯齿。如果图像的分辨率相对于显示器的大小很小，这些伪影可能会让眼睛非常不舒服。

对抗混叠的经典信号处理方法是以更高的频率采样，然后应用低通滤波器。在光栅化中，这大致相当于渲染到更高分辨率的渲染目标，对相邻像素进行平均，并将平均结果用作正常大小渲染目标的像素值。在实时3D图形中，此过程称为超级采样消除混叠（SSAA）。增加分辨率会增加用于光栅化和像素着色的采样频率，从而有效地减少两种类型的锯齿的出现。但是，渲染目标分辨率翻倍或翻两番所带来的额外性能成本通常会使该技术成本过高。因此，Direct3D 11包括一种更简化和优化的抗锯齿形式，称为多样本抗锯齿（MSAA）。

MSAA的基本前提是，用于光栅化和深度/模具测试的分辨率增加了一个整数因子，但像素着色器仍以渲染目标的正常分辨率执行。这允许它有效地处理边锯齿，但不能处理着色器锯齿。由于深度和模具测试以更高的分辨率执行，因此深度模具缓冲区必须每像素存储多个子样本，从而导致

3.10光栅化器219

内存占用将按MSAA系数增加。这同样适用于渲染目标，因为它必须为每个采样存储单独的颜色值，然后才能将其解析为单个值。因此，在创建渲染目标和深度模具资源时，必须显式指定MSAA样本计数，以便运行时能够分配足够的内存。此外，对于同时用于渲染的渲染目标和深度模具缓冲区，MSAA采样计数必须完全相同。有关创建MSAA所需资源的更多详细信息，请参见第2章。

**光栅化器与MSAA的交互。**启用MSAA进行渲染时，对管道的第一次修改处于光栅化阶段。通常情况下，在光栅化过程中，使用上述每个像素的单个点测试三角形的覆盖率。在Direct3D 11中，该点位于像素的精确中心，或在X和Y方向上距左上角0.5个单位。该点还用于插值上一阶段（顶点着色器、域着色器或几何体着色器，取决于使用的阶段）输出的所有顶点属性，包括用于执行深度测试的深度。启用MSAA时，将使用像素内的多个采样点测试覆盖率。这些采样点的位置是特定于实现的，尽管这些采样通常以旋转网格模式排列。MSAA和非MSAA光栅化的采样点示例如图3.65所示。

当深度模具目标和MSAA渲染目标已绑定到管道时，光栅化器行为由光栅化器状态的多采样成员确定。此参数本质上切换光栅化器是否对MSAA渲染目标执行多个覆盖率测试。如果它被禁用并且绑定了MSAA目标，则仅在像素中心执行单个覆盖测试。如果覆盖测试通过，像素着色器结果将写入所有子样本。如果它没有通过，那么它们都不会被写入。此可选MSAA操作允许使用多重采样功能渲染场景的一部分，而不需要抗锯齿的场景的其他部分（如用户界面的2D精灵）可以以正常方式渲染。这将为这些不使用抗锯齿的场景元素生成一致的外观，并通过减少所需的覆盖率测试数量来提高性能。



图3.66。启用和不启用MSAA的渲染之间的差异。

220                                                                                                                                                      3.渲染管道

图3.66显示了使用和不使用MSAA活动光栅化的基本体之间的差异。您可以清楚地看到，MSAA版本（如左图所示）与非MSAA版本（如右图所示）中基本体边缘的平滑程度有很大的不同。

### 3.10.4光栅化器级管线输出

片段生成

在整个光栅化器阶段的讨论中，我们已经看到了用于将原语转换为一组片段的每个过程。一些不同的测试，如剔除，旨在减少需要处理的原语数量。此外，还有一些测试，如剪裁、视口变换和剪刀测试，这些测试可能会减少原语生成的片段数量。似乎已经投入了大量的精力来减少光栅化器阶段生成的碎片数量。为什么这么多的测试都是为了降低这个阶段的产量？

原因是光栅化器级的输入和输出之间存在1对多关系。对于作为输入提交到光栅化器的每个原语，可能会生成许多片段。由于这种通用数据放大，管道中在此点之后执行的任何处理都将比之前执行的处理执行更多次。因此，显然最好在将未使用或不必要的原语分割成片段之前消除它们，并且最好在光栅化器阶段消除片段，然后再对其进行处理。这一思想也可以推广到一般的算法设计中。当您可以在光栅化之前或之后执行计算时，通常在生成多个片段之前执行计算更有效。在许多情况下，即使在光栅化器之前进行的计算只是之后进行的计算的近似值，这也可以在图像质量损失最小的情况下产生较大的效率增益。

片段数据

考虑到这一点，我们可以考虑光栅化阶段产生了什么类型的数据流。我们已经详细讨论了当渲染目标中的每个片段的位置被基元覆盖时，如何生成这些片段。当确定某个位置被基元覆盖时，使用像素的中心作为插值函数的输入，对传递到光栅化器中的每个属性进行插值。深度值也为32

32对于MSAA光栅化，采样位置也可用于插值。

|  |  |
| --- | --- |
| 3.11像素着色器 | 221 |

与每个片段一起插值并传递，以用于输出合并阶段的深度测试。

片段的位置可以添加到生成的片段中，也可以不添加到生成的片段中。这取决于像素着色器输入签名是否声明其输入属性之一具有SV\_位置系统值语义。在光栅化器阶段之前，SV\_位置值用于指示顶点位置的版本。传入光栅化器时，它必须包含剪辑空间顶点位置。但是，片段位置在光栅化器的输出中提供。这可以在像素着色器中用于查找其他纹理资源中的值，或者用于基于屏幕位置执行处理，例如在屏幕的外边缘将渲染目标颜色淡入黑色。

## 3.11像素着色器

光栅化器将基本体转换为片段后，片段将传递到像素着色器阶段。像素着色器阶段是渲染管道中的最后一个可编程着色器阶段。它通过调用其像素着色器程序分别处理每个片段。每个像素着色器调用都是独立操作的，处理的各个片段之间不可能进行直接通信。像素着色器完成后，结果是经过处理的输出片段，该片段被传递到输出合并阶段。管道内像素着色器的位置在图3.67中高亮显示。33,34

像素着色器阶段负责渲染基本体的主要外观。在光栅化之前，处理主要集中于操纵基本体的数量、大小、形状和属性。这些都是几何样式的操作。在光栅化过程中，这些原语用于选择受其影响的像素，并相应地生成片段。像素着色器只能接收传递给它的片段，因此无法从中更改片段的位置



图3.67。像素着色器阶段。

33 派生指令使用多个片段之间共享的数据。

34 无序访问视图改变了这种严格的隔离，但像素着色器调用之间仍然存在相当强的通信隔离。

222                                                                                                                                                      3.渲染管道

像素着色器阶段。相反，像素着色器根据片段的属性以及在各种资源中提供给舞台的信息，确定每个选定片段的显示方式。这使像素着色器在实时渲染中的图像生成中发挥了相对重要的作用。

像素着色器阶段还有一些在Direct3D 11之前不可用的新功能。它可以使用新的资源视图类型无序访问视图（UAV），对附加到视图的完整资源执行读写操作。这与像素着色器阶段的传统功能大不相同，后者只能将颜色和深度数据写入传递给它的像素位置。这提供了在渲染图像时直接在渲染管道中实现大量算法的可能性，例如直方图生成。35

### 3.11.1像素着色器管道输入

像素着色器阶段从光栅化器阶段接收其输入片段。这意味着像素着色器程序将使用的输入属性也由光栅化器生成。我们已经在“光栅化器”部分中看到，它根据要光栅化的原语中片段的采样位置生成插值属性数据，该位置通常是像素的中心（尽管有时来自其他采样位置）。我们还看到，可以在像素着色器程序中指定各种插值修改器关键字，该程序将指示光栅化器为每个输入属性使用特定的插值模式。这些插值模式在不同的场景中很有用，必须适当选择，以确保正确计算输入属性。我们将在这里更详细地探讨这些插值模式。

属性插值

插值属性的过程需要三条不同的信息。第一个是数据，该数据将根据要光栅化的基本体顶点的属性进行插值。这包括它们的位置和属性值。插值所需的第二条信息是需要插值属性数据的点的位置。这或多或少决定了插值将接收多少每个顶点属性。所需的最后一条信息是应该使用的插值技术。最后一项对上述前两项执行实际插值。插值模式由使用像素着色器输入签名声明的插值修改器确定。

33 资源视图只能用于公开资源的一个子部分，而不是完整的资源，如第2章所述。

|  |  |
| --- | --- |
| 3.11像素着色器 | 223 |



图3.68。使用线性插值模式的示例。

**线性模式。**使用线性插值模式时，为每个片段生成的属性将线性插值，并考虑透视效果。透视校正插值的需要源于透视投影不是线性操作，并且该插值在投影之后执行。这意味着屏幕空间中的顶点之间不能使用标准线性插值，但必须考虑每个顶点的深度。

在插值过程中，可以通过在插值之前将每个顶点属性除以其在查看器中的深度来说明透视效果。将为每个顶点找到深度值作为其顶点（W值，在投影后，但在转换为标准化设备坐标之前。W的倒数也会计算，然后连同这些修改的属性一起插值。在为每个光栅化片段插值所有这些值后，W的插值倒数将用于提取原始所需属性，而ch将生成透视校正插值值。这是最常见的插值模式，三维模型上的纹理坐标提供了一个完美的示例，说明何时需要它们。透视校正插值不仅需要用于纹理坐标，而且还需要用于位置或方向向量（或任何其他线性属性）从线性空间（如世界空间或视图空间）进入后投影阶段。图3.68中显示了线性插值模式的示例。

**非透视模式。**当使用非透视插值模式时，插值属性根据其在渲染目标上的二维位置严格插值。在本质上，可以在插值之前将几何图形视为投射到渲染目标上。基于顶点位置的顶点之间的插值。每当被插值的顶点都处于相同深度时，例如在构建屏幕用户界面渲染时，可以执行属性插值而不进行透视校正。图3.69演示了此插值模式的使用。

224                                                                                                                                                      3.渲染管道



图3.69使用非透视插值模式的示例。

无极化模式。顾名思义，“nointerpolation”模式不执行顶点属性的任何插值。这意味着每个基本体的第一个顶点的属性值将传递给给定基本体的所有片段，从而在基本体的整个曲面上生成一个常量值。当使用此插值模式修改与照明相关的属性时，可以使用该模式为模型提供镶嵌面外观。实际上，不使用插值通常会使渲染几何体的面更加明显，因为使用插值的全部目的是隐藏基于顶点的几何体近似于平滑曲面的事实。使用细分系统时，这可能会最小化，但也可能是可视化最终几何结果细分精细程度的好方法。图3.70演示了该插值模式的使用。

**质心模式。**质心插值模式旨在为MSAA渲染模式中的特殊情况提供更合适的插值模式。当一个像素是



图3.70。使用n o i n t e r p o l A t i o n插值模式的示例。

3.11像素着色器225



图3.71。质心采样的插值点。

部分由原语覆盖，并且启用了MSAA，光栅化器将为该像素生成一个片段。但是，如果原语仅覆盖像素内的少数样本，而不覆盖像素中心，则正常线性插值模式仍将使用像素中心作为插值函数的输入位置。因此，在仅覆盖部分采样位置的情况下，可以将顶点属性外推到三角形真实边之外。对于不需要外推的情况，可以通过向输入变量声明添加质心修饰符，在每个属性的基础上使用质心采样。使用质心采样时，质心采样会使受影响的属性内插到保证被三角形覆盖的点。所使用的位置是特定于实现的，但通常位于覆盖样本点的平均位置。质心采样模式的示例如图3.71所示。

**示例模式**. 采样插值模式在MSAA渲染模式中使用的每个采样点处提供插值。当为每个MSAA样本执行像素着色器以在每个单独的样本位置提供属性时，这非常有用。MSAA主题及其与像素着色器的关系将在下一节中进一步讨论。

多采样抗锯齿

可以将几个附加的系统值语义声明为像素着色器的输入，以便在启用MSAA时为控制其行为提供一些有趣的可能性。Direct3D 11提供了几种改变“标准”MSAA行为的方法。这些更改基于绑定到管道的像素着色器，以及它如何声明其输入和输出属性。

226                                                                                                                                                      3.渲染管道

**按样本执行。**第一种修改是可以为每个子样本运行像素着色器，而不是每个像素运行一次。当像素着色器获取附加了SV\_SampleIndex系统值语义的输入时，会触发此行为，该值提供当前正在处理的子样本的索引。如果输入属性被标记为采样插值模式，也会触发该模式。我们将在本节后面更详细地讨论SV\_SampleIndex的用法。

**子样本覆盖率。**与MSAA系统的第二种可能的交互涉及SV\_覆盖系统值语义的使用。使用此语义声明输入属性时，它提供一个无符号整数值，其中每个位对应于当前像素的一个子样本。这本质上向像素着色器指示在光栅化过程中检测到哪些子样本被光栅化器阶段覆盖。稍后，我们将看到如何将此系统值语义用作输出属性，以及如何对选定子样本执行自定义修改，这些子样本将随像素着色器的结果写入。这允许实现自定义覆盖掩码，这通常用于alpha-to-coverage算法。

片段定位

另外两个系统值语义可作为像素着色器的输入访问，以标识片段的位置，这是它最终将写入的位置。这些属性可用于帮助像素着色器程序处理片段。

**震源位置。**当像素着色器使用SV\_Position系统值语义声明输入属性时，像素着色器将接收光栅化器阶段生成的四个坐标片段位置。通常，该位置最重要的部分是X和Y坐标，它们指示片段在当前渲染目标中的位置。在大多数情况下，这些坐标将指示偏移0.5像素的像素中心位置。当使用多个纹理生成像素着色器结果时，这可用于标识其他纹理中需要采样的位置。一个常见的例子是使用多个纹理作为延迟渲染的G缓冲区（有关延迟渲染的更多详细信息，请参见第11章）。

除了像素中心位置外，还可以使用此系统值语义获取当前片段的质心位置。如上所述，质心位置保证位于基本体边界内，并且通常用于确保用于插值的坐标不会超出基本体的边缘。如果使用质心插值修饰符声明此语义值，则可以接收插值中使用的质心值。

光栅化器级生成的深度值也可从SV\_位置系统值语义中获得。这是标准化设备坐标深度，因此其值将在[0,1]范围内。这可用于在

3.11像素着色器227

使用此深度信息的像素着色器，也可用于使用自定义深度值修改光栅化器生成的深度值。我们将在本节后面看到像素着色器如何使用SV\_深度系统值语义修改其深度值。

**片段目的地。**除了能够接收生成片段的位置之外，片段的最终目的地也可以声明为输入。如果有一个标准的非数组渲染目标绑定用于接收来自管道的输出，则片段的输出位置或多或少由上面讨论的SV\_位置语义描述。但是，如果使用基于阵列的渲染目标来生成场景的多个同时渲染，则还必须使用渲染目标阵列索引来确定片段最终将写入的位置。这通常用于生成各种形式的环境地图，如第13章所示。

管道通过解释SV\_RenderTargetArrayIndex系统值语义来确定将基本体应用于哪个渲染目标切片。此参数还可以用作像素着色器的输入属性。这使像素着色器程序可以根据最终将接收片段的渲染目标切片对片段执行选择性处理。

几何定向

我们可以为像素着色器的输入声明的最后一个专用输入属性是SV\_IsFrontFace系统值语义。此参数指示生成片段的原语的方向。这是一个布尔属性，当片段由前向原语生成时为true，当片段由后向原语生成时为false。您可能还记得，光栅化器状态可用于确定哪个顶点缠绕定义了基本体的前向方向，以及要剔除哪些方向。如果两个方向都允许光栅化，则此系统值将指定光栅化器阶段检测到的方向。

由于点和线基本体没有方向的概念，因此从这些基本体类型生成的片段将始终为该输入属性生成一个true值。但是，如果在线框模式下对三角形基本体进行栅格化（该模式基本上从三角形的边生成线），则仍将使用三角形方向信息。

### 3.11.2像素着色器状态配置

随着对像素着色器程序所能接收到的数据的清楚理解，我们现在将考虑什么类型的状态配置可供应用程序使用。像素着色器阶段是一个可编程阶段，这意味着它可以访问我们在各个管道阶段中看到的标准通用着色器核心功能。我们再一次

228                                                                                                                                                       3.渲染管道

这里不会重复任何代码列表，因为它们与我们在顶点着色器部分中看到的基本相同。下面列出了这些常用方法，以供参考。

ID3DllDeviceContext:：PSSetShader（）

ID3DllDeviceContext:：PSSetConstantBuffers（）

ID3DllDeviceContext:：PSSetShaderResources（）

ID3DllDeviceContext:：PSSetSamplers（）

除了公共着色器核心的标准资源外，像素着色器还可以访问新的无序访问视图，从而提供直接从像素着色器读取和写入资源的随机访问。最后，可以使用名为earlydepthstencil的函数属性强制在像素着色器之前执行深度模具测试。我们将在以下章节中更详细地研究这些独特的配置。

无序访问视图

像素着色器阶段可以使用无序访问视图。与提供对资源的只读随机访问的着色器资源视图不同，无序访问视图允许读取和写入随机访问。这允许像素着色器程序执行分散写入操作，这意味着它可以通过编程方式决定将写入资源的数据存储在何处。在访问无人机之前，像素着色器只能在生成片段时由光栅化器阶段确定的位置写入输出渲染目标和深度模具渲染目标。

将无序访问视图绑定到管道以在像素着色器中使用的方法通过设备上下文执行，与所有其他管道配置一样。在这种情况下，UAV实际上绑定到输出合并阶段的管道，而不是像素着色器阶段。这样，渲染目标、深度模具目标和将从管道接收输出的任何UAV都将绑定为在同一方法调用中从像素着色器输出。这还将所有可能的管道输出资源分组到单个管道阶段，这在一定程度上简化了管道概念。在“输出合并”部分中，我们将看到如何将资源绑定到具有UAV的像素着色器中。

早期深度模板试验

像素着色器阶段的另一个独特配置是从像素着色器程序的源代码中执行的。可以在像素着色器程序之前声明earlydepthstencil函数属性，以指示应执行深度和模具测试

|  |  |
| --- | --- |
| 3.11像素着色器 | 229 |

在执行像素着色器之前执行（深度和模具测试将在“输出合并阶段”部分中详细介绍）。这些测试通常通过硬件实现在管道中尽可能早地执行，但此属性强制提前执行测试。

这旨在通过消除深度测试或模板测试失败的片段，从而提高特定算法的效率，从而丢弃这些片段。通过这样做，可以防止从深度或模具测试失败的片段调用任何像素着色器。当场景已经光栅化到深度缓冲区中并且需要第二个渲染过程时，这会特别有用。

另一种需要此属性的更微妙的情况是，渲染管道仅通过像素着色器向UAV输出数据。在这种情况下，将阻止数据写入输出合并，因此管道将永远不会执行深度或模具测试。但是，如果仍然需要这些测试，则必须使用此函数属性显式地强制执行这些测试。

### 3.11.3像素着色器阶段处理

我们现在已经了解了像素着色器阶段从光栅化器接收到的信息类型，以及可以为其提供哪些额外资源来增强可用数据。在这一节中，我们将首先考虑像素着色器阶段如何执行其职责的一些机制。接下来，我们将通过考虑像素着色器如何在传统渲染场景中表示对象的材质属性来进一步探索像素着色器。在此之后，我们将研究新UAV的一些可能用途，并考虑一些新的资源视图类型启用的一般操作。最后，我们将通过讨论MSAA如何与像素着色器阶段交互，以及如何控制各种MSAA功能来提高图像质量，同时仍保持可接受的性能水平来完成本节。

皮克斯！着色器力学

我们已经看到了数据是如何进入像素着色器阶段的。现在我们可以考虑像素着色器将执行的一些操作，以及它在给定流水线配置中必须产生的操作。

**像素着色器执行。**我们的像素着色器阶段的概念操作模型是，当它执行时，每个调用处理单个片段，然后计算输出颜色以传递到输出合并阶段。由于像素着色器在光栅化发生后运行，因此在像素着色器中执行的任何计算都将与对象覆盖的片段数成比例地执行。无论输入几何体有多复杂，都只处理几何体覆盖的视口中的片段

230                                                                                                                                                      3.渲染管道



图3.72。执行像素着色器以生成单个片段输出。

图3.73。同时执行多个像素着色器调用。

使用像素着色器。图3.72描述了为每个像素着色器调用处理单个片段的概念。

通过遵循此模型并限制线程之间的通信，可以在并行处理元素中同时计算像素着色器的许多不同调用。由于每个调用都从其输入片段接收自己的数据，并将自己的输出写入片段指定的位置，因此它不依赖于相邻的调用。图3.73显示了这一概念的示意图。

在较低的硬件级别上，这仍然基本正确，但像素着色器调用总是在至少2x2组片段中执行。这样做是为了确保通过执行离散差分，可以在X方向和Y方向的调用中计算屏幕空间派生指令。这些派生指令的计算依赖于多个调用同时运行这一事实，如果一个变量被传递到其中一个派生指令中，那么GPU将使用相邻调用变量的差异来发现该变量的值如何随屏幕空间变化。实际上，此行为对应用程序是透明的，像素着色器调用之间的边界保持不变。

**多个渲染目标。**我们已经提到，像素着色器计算并输出要传递到输出合并阶段的颜色。但是，这并不局限于单一颜色。我们将在输出合并阶段部分看到，最多可以绑定8个

3.11像素着色器231



图3.74。使用MRT和标准单渲染目标用基于场景的数据填充多个渲染目标之间的区别。

用于接收管道输出的不同渲染目标，只要它们满足一定的大小和格式限制。单像素着色器调用计算并输出要写入每个渲染目标的颜色。这种同时使用多个渲染目标的能力称为多个渲染目标（MRT），可以提高渲染算法的效率。

如果算法要求使用场景几何体生成的数据填充多个渲染目标，则可以使用传统的单个渲染目标配置一次填充一个渲染目标。这意味着在顶点着色器、细分阶段、几何体着色器和光栅化器中执行的所有计算都将针对每个过程重复。过程之间唯一不同的计算是由像素着色器执行的计算。MRT的使用允许将所有过程合并为一个过程，然后像素着色器分别写入所有输出。这样可以节省多次处理几何数据的成本，但仍然可以使用所需信息填充相同数量的渲染目标。图3.74显示了在该场景中使用MRT与进行多个渲染过程之间的区别。第11章“延迟渲染”中提供了使用MRT的一个很好的示例

**修改深度值。**除了负责为给定片段写入颜色值外，像素着色器还可以向SV\_深度系统值输出新的深度值。如果像素着色器未写入深度值，则在光栅化器阶段生成的深度将传递到输出合并阶段。但是，在某些情况下，像素着色器更适合指定深度值。当使用广告牌时，这将非常有用。广告牌通常用于模拟更复杂的几何体，其中两个三角形排列成四边形，然后对齐以垂直于当前视图方向。将纹理应用于四边形，并显示结果

232                                                                                                                                                      3.渲染管道



图3.75。使用广告牌模拟更复杂的几何体。模型由放射性软件有限责任公司提供。由托马斯·德林诺夫斯基、丹尼·格林创作。www.radiotic-software.com

仅渲染两个三角形仍然可以提供大量细节。广告牌的概念如图3.75所示。36

广告牌通过非常简单的几何体提供了更多的细节，这是一种将复杂性引入场景的有效方法。但是，由于广告牌本质上是平面的，当它用于与其他场景几何体相交的情况时，复杂几何体的幻觉会被破坏，因为广告牌在其表面上具有均匀的深度。在此场景中，如果模拟几何体的深度变化包含在布告牌纹理中（例如在alpha通道中），则像素着色器可以将修改的深度值写入片段，从而重新为布告牌几何体引入深度复杂性。然后，当场景几何体与广告牌相交时，它们实际上会更具说服力，部分遮挡而不是完全遮挡。图3.76显示了此类深度修改的示例。

**保守的深度输出。**但是，从像素着色器写入深度值确实有一些缺点。大多数现代GPU都实现了一种称为Hi-Z剔除（Hi-Z Culling）的效率改进技术。该技术背后的概念是GPU硬件执行一些简化形式的遮挡测试，以查看当前批次的几何体是否能够在最终渲染目标中看到，或者如果它将在渲染中显示在另一个对象后面。如果几何体将被遮挡，那么它只是

36广告牌用于第12章“模拟”中讨论的粒子系统示例

3.11像素着色器233



图3.76。如何将深度添加到广告牌以允许进行适当深度测试的俯视图。

在进一步处理之前丢弃。当场景中有许多对象重叠时，这将降低管道执行的总体计算成本。关于如何实现这一点的细节因GPU制造商而异，因此此处将不详细介绍。但是，在此分层测试期间使用的有关几何体大小的一些假设发生在执行像素着色器之前。因此，在像素着色器中修改深度值时，Hi-Z可能无法使用，其性能优势可能会丢失。

为了允许Hi-Z在需要深度输出的特定算法子集中保持活动状态，Direct3D 11引入了一种称为保守深度输出的新功能。该技术的工作原理是要求像素着色器指定一个不等式函数以及新的深度值。不等式有效地指定了深度输出的上限或下限，这允许Hi-Z继续识别可以被简单拒绝的片段，因此不需要执行像素着色器。

要使像素着色器使用保守的深度输出，着色器程序必须为用于输出深度的值指定四个新系统值语义之一。这些语义都将不等式指定为语义名称的一部分，深度值必须满足相对于光栅化器阶段计算的插值深度值的不等式。如果像素着色器输出的深度值未能满足不等式，则运行时将自动将该值钳制为适当的最小值或最大值。表3.3列出了这四种语义。

234                                                                                                                                                      3.渲染管道

|  |  |  |
| --- | --- | --- |
| **语义的** | **描述** |  |
|  |  |  |
| **SV\_深度更大** | **深度输出必须**heb大于插值的深度值 |  |
|  |  |  |
| **SV\_深度更大相等** | **深度输出必须大于或等于插值深度** |  |
| , |  |
|  | 价值 |  |
|  | 价值 |  |
| 无深度 | **深度输出必须小于插值的深度值** |  |
|  |  |  |
| **SV**\_深度相等 | **深度输出必须小于或等于插值深度** |  |
| , |  |
|  | 价值 |  |
|  | 价值 |  |



表3.3。保守深度输出的系统值语义。

保守深度是否允许Hi-Z拒绝碎片取决于不等式、光栅化器生成的插值深度值、深度模具状态中指定的深度测试函数以及深度缓冲区中的当前值。为了发生拒绝，当与深度缓冲区中的当前值进行比较时，所有可能满足不等式的深度值必须通过深度测试。作为一个简单的示例，假设生成的片段的插值深度为0.75，该像素的深度缓冲区中的当前值为0.5，并且D3D11\_C0MPARIS0N\_LESS被指定为当前深度比较函数。如果使用没有深度输出的像素着色器，Hi-Z硬件将确定0.75大于当前深度值，并且



图3.77。“SVDepthGreaterEqual”的保守深度输出

3.11像素着色器235

应该拒绝该片段。如果使用具有保守深度输出的像素着色器为深度输出属性指定SV\_DepthGreaterEqual，Hi-Z硬件仍将能够拒绝片段。这是因为不等式保证深度输出只能变得更大，因此，深度测试总是失败。图3.77说明了这种情况。

请注意，如果使用SV\_DepthLess或SV\_Depthl\_essEqual代替SV\_DepthGreaterEqual，Hi-Z装置将无法拒绝碎片。这是因为像素着色器输出的值小于当前深度缓冲区值0.5是合法的，从而导致片段通过深度测试。事实上，如果5V\_DepthLess或SV\_DepthLessEqual与D3D11\_c0n\_LESS一起使用，则Hi-Z装置不会拒绝碎片。这同样适用于使用SV\_DepthGreater或SV\_DepthGreater等于D3D11\_C00n\_更大。因此，为了获得最佳性能，应始终使用与深度测试方向相反的方向不相等的保守深度语义。

一个小例子

现在我们来考虑如何使用像素着色器阶段来实现渲染模型。此阶段的主要职责是生成输出颜色，该颜色将合并到管道末尾的渲染目标绑定中。因此，开发人员必须做出的最终决定是执行何种类型的计算，以及使用何种数据输入进行计算。算法的类型可以从渲染的所有像素的单一颜色输出，一直到执行最终渲染之前需要几个模拟步骤的完整全局照明系统。

为了探索如何实现各种算法，我们将首先研究一个简单的渲染示例，并了解如何使用像素着色器来形成渲染的颜色输出。在此过程中，我们将解释此示例场景如何表示像素着色器使用的基本方法。通过对这些概念的深入了解，我们将能够实现各种各样的算法，包括一些甚至不用于生成颜色数据（如阴影贴图）的算法。实现的细节不如阶段中如何执行处理的细节重要。

如上所述，像素着色器负责生成最终颜色值，该值将在管道末端合并到渲染目标中。在此示例场景中，颜色确定将是对象的材质属性以及材质与之交互的某些环境属性（如灯光的存在）的产物。在不同的照明条件下查看具有相同材质特性的对象时，其外观可能会大不相同，正如具有不同材质特性的模型即使在相同的照明条件下渲染，其外观也可能不同。因此，这两组属性都必须可供像素着色器程序使用，以正确确定要生成的颜色。

3.渲染管道

**材料特性。**对象的材质属性决定了它在基本标高上的外观。典型的材质属性包括对象的基色、一些用于描述曲面反射程度的系数，以及表示精细曲面颜色变化的纹理贴图。我们将假设像素着色器接收顶点法向量和纹理坐标作为插值输入属性。像素着色器程序将使用清单3.23中所示的函数计算对象材质的颜色。

浮动4 PSMAIN（在VS\_输出i n p u t中）：SV\_目标

{

//  从纹理float4 SurfaceColor=ColorTexture.Sample（LinearSampler，input.tex）确定曲面的颜色属性*);*

*二,*返回表面颜色

返回（SurfaceColor\*input.color）；

}

清单3.23。用于计算对象材质颜色的示例像素着色器。

在典型场景中，需要渲染许多不同的对象，并且每个对象都需要至少部分不同于其他对象的属性。在本例中，我们将假设像素着色器在渲染的每个对象中是通用的。考虑到这一点，我们可以通过一些对象渲染序列，并考虑每一个相对于像素着色器的不同之处。我们将假设场景中有三个对象，称为对象A、B和C。对于要渲染的每个对象，在使用其中一个绘制调用执行管道之前，必须使用对象属性配置管道。

当管道配置为渲染对象A时，对象的纹理通过着色器资源视图绑定到管道，其颜色通过将常量缓冲区绑定到包含适当颜色的像素着色器阶段来指定。生成的每个片段都被传递到像素着色器，在那里对纹理进行采样并乘以常量缓冲区中提供的颜色，如清单3.23所示。此管道执行完成后，应用程序将配置渲染对象B的管道。这需要绑定不同的纹理和颜色常量缓冲区，然后执行管道。对象C将遵循相同的模式，但在这种情况下，它将使用与对象B相同的纹理，并且只需要一个新的颜色常量缓冲区。

此短序列表明，对象的材质属性通常作为着色器程序本身外部的资源提供。模型的颜色和纹理都由操作像素着色器阶段状态的应用程序控制，而不是通过进出像素着色器程序来控制。此外，通过管道的几何图形对材质属性实际上并不重要。在所有三个

|  |  |
| --- | --- |
| 3.11像素着色器 | 237 |

在某些情况下，可以在对象之间交换输入几何体，并且材质的最终外观将保持不变。这是因为像素着色器在管道中的几何数据转换为光栅化形式后运行。它在光栅化后处理数据，因此不受几何体更改的影响。在考虑这些属性的同时，我们将考虑如何通过示例添加照明来使示例对象与环境交互。

照明属性。通过在场景中提供照明信息，我们可以显著提高其质量。照明是我们如何看待物理世界的一个非常基本的部分，使用它可以显著改进生成的场景渲染。现代实时渲染中使用了许多不同类型的照明表示，但在本示例中，我们将使用一个简单的定向照明模型。光将由方向向量和颜色来描述，这两者都将在第二恒定缓冲器中提供。到达给定曲面的光量通常通过曲面法向量和表示光的行进方向的向量的点积来近似。这将产生[0.0,1.0]范围内的标量值（只要法向量和光向量都已规格化），并可用于缩放应用于曲面的光量。清单3.24中提供了执行此操作的函数。

浮点4 PSMAIN（在VS\_输出输入中）：SV\_目标

{

//  规范化世界空间法线和光向量float3 n=规范化（input.normal）；

float3 1=标准化（input.light）；

//  计算到达该碎片的光量，4照度=最大值（点（n，l），0）+0.2f；

//  从纹理float4 SurfaceColor=ColorTexture.Sample（LinearSampler，input.tex）确定表面的颜色属性；

//  返回由照明调制的曲面颜色

返回（表面颜色\*输入颜色\*照明）；

}

清单3.24。执行简单照明方程式的函数。

现在，我们可以返回到对象A、B和C的样例渲染。渲染序列将重复，但照明信息必须在第二个常量缓冲区中提供给像素着色器。由于这些对象都位于同一场景中，因此它们都使用相同的灯光描述。这意味着相同的常量缓冲区可用于所有管道执行。为了执行该示例，将以与前面相同的方式为每个对象配置管道，并添加照明

238                                                                                                                                                      3.渲染管道

常量缓冲区也被绑定。执行像素着色器时，将以与之前相同的方式查找对象的材质颜色，并基于作为输入属性传递到像素着色器的法向量和灯光方向向量，计算每个像素位置处可见的光量。然后使用照明计算的结果来调节片段的颜色。当渲染每个附加对象时，该过程将使用其每个材质属性完全重复，并且生成的渲染现在除了材质颜色外还包含照明。

在本例中，我们看到整个场景中的照明信息是相同的，因此以相同的方式应用于三个对象中的每一个。共享同一环境的所有对象的环境数据都相同。我们还看到，尽管这三个对象的材质外观不同，但它们都以相同的方式与灯光交互，使用相同的计算，而不管它们是什么颜色。

**概括这个例子。**那么，我们从这个例子中学到了什么，我们如何应用这些简单实验的结果来理解使用像素着色器实现渲染技术的一般概念呢？像素着色器程序只是一个函数，它接受一定数量的输入参数并生成颜色。某些输入在每次管道执行时更改，例如材质属性，而某些输入在每个渲染帧中更改一次，例如照明属性。还有一些在应用程序的整个生命周期中根本不会改变，例如模型的顶点法向量。

像素着色器提供的灵活性现在变得非常清晰。计算生成片段颜色的函数是什么并不重要。只要生成的颜色结果在输入发生变化时会发生适当的变化，渲染模型就可以满足其目的。开发不同的渲染模型围绕着确定应该使用哪些输入来计算输出颜色，开发执行从输入到输出的映射的函数，然后生成适合渲染模型的几何内容。更复杂的渲染模型可能需要更多的输入，包括使用动态生成的输入（如附加渲染过程的结果）的可能性。但是，像素着色器本身始终解析为将输入数据转换为输出颜色的方法。

使用无序访问视图

上一个示例演示了如何使用像素着色器实现渲染模型。在这一节中，我们考虑通过将无序访问视图包含到像素着色器中来获得什么样的附加可能性。UAV允许通过像素着色器的任何调用在任何位置读取或写入各种资源类型。在添加无人机之前，像素着色器阶段仅限于从资源中读取，但将其输出颜色和深度写入片段位置除外。

3.11像素着色器

这种写入资源任何位置的新功能可用于从渲染过程中提取附加信息。例如，直方图的生成可以与场景的渲染同时执行。柱状图提供多个存储单元，其中每个存储单元表示具有特定范围内的值的像素数。直方图提供图像中值分布的指示。通过检查像素着色器的输出颜色，可以确定颜色所属的直方图区域。然后，可以使用UAV访问资源，并增加相应的bin值。在引入无人机之前，这是不可能的。

使用无人机可以实现更复杂的渲染算法。由于可以对资源进行读写访问，所以可以使用无人机实现通用数据结构。这种能力已经被用于从GPU资源中构建链表实现，然后可以将其用作其他算法的构建块。

多样本抗锯齿注意事项

由于像素着色器的操作非常接近最终渲染图像，因此在使用MSAA时应考虑几种特殊的像素着色器功能。对于非MSAA渲染，为每个像素的采样点执行的覆盖率测试确定是否应为该像素执行像素着色器阶段。使用普通MSAA渲染时，每个像素仍然只执行一次像素着色器。因此，只要至少一个采样点通过覆盖测试，就会执行像素着色器。执行着色器后，着色器输出的值将写入当前绑定的渲染目标中的相应子样本。

**在碎片中采样MSAA纹理。**在Direct3D 11中，可以为MSAA纹理创建着色器资源视图，以便将纹理绑定为各个着色器阶段的输入。这是通过将D3D11\_SRV\_DIMENSI0N\_TEXTURE2DMS设置为传递给ID3D11设备的D3D11\_SHADER\_resource\_VIEW\_DESC结构的ViewDimension成员来实现的

：：CreateShaderResourceView方法。但是，当视图绑定到某个阶段的管道时，该阶段无法使用normal Texture2D对象及其各种方法对纹理进行采样。相反，必须声明Texture2DMS对象。此对象支持load方法，该方法允许您指定要检索的子样本的XY位置和索引。它还支持用于查询XY维度和纹理中的子采样数的GetDimension方法，以及用于查询用于指定子采样索引的像素覆盖率采样点的GetSamplePosition方法。在像素着色器中使用这些方法可以实现自定义解析算法，而不是依赖硬件提供的内容。

**阿尔法覆盖率。**Alpha-to-coverage是一种修改像素着色器输出写入渲染目标的方式的技术。通过设置AlphaToCoverageEnabled启用时

240                                                                                                                                                      3.渲染管道

作为D3D11\_BLEND\_DESC结构的成员，如果设置为true，则像素着色器的输出不再仅基于光栅化阶段覆盖测试的结果写入各个子样本。相反，覆盖测试生成的掩码是逐位的，并使用从像素着色器输出的值的alpha分量生成的掩码进行加密。该结果使用屏幕空间抖动遮罩进行渲染，该遮罩使曲面在子采样上以固定的抖动模式进行渲染。此功能最常见的用途是实现简化版本的透明度，其中透明曲面根据其不透明度进行抖动，而不是混合。这提供了性能优势，因为像素输出不需要与渲染目标内容混合，也不需要根据透明曲面与摄影机的距离对其进行排序。

使用alpha覆盖并不明确要求启用MSAA。然而，当与MSAA结合使用时，质量显著提高，因为它允许在子样本级别而不是在像素级别发生抖动，从而提供更高的输出图像质量。

**像素着色器行为修改。**Direct3D 11提供了几种改变“标准”MSAA行为的方法。这些更改基于绑定到管道的像素着色器，以及它如何声明其输入和输出。

**按样本执行。**第一种修改是可以为每个子样本运行像素着色器，而不是每个像素运行一次。当像素着色器获取附加SV\_SampleIndex语义的输入时，会触发此行为，该语义提供当前着色的子样本的索引。此索引对应于传递给Texture2DM的索引。加载，使从另一个MSAA渲染目标检索适当的子样本变得简单。如果“采样”修改器附加到输入，也会触发每采样行为，这会导致属性插值到当前采样点，而不是像素中心。

让像素着色器以更高的频率运行意味着，除了边缘锯齿之外，还可以减少着色器锯齿，或者可以为非传统渲染管道实现特殊行为。但是，应该注意，以采样频率执行像素着色器意味着这些像素的性能实际上降低到超级采样的性能，因为管道基本相同。因此，仅对渲染的几何体子集使用每采样着色器，或使用模具缓冲区遮罩不需要每采样着色的像素通常是有益的。有关使用每采样像素着色器的示例，请参见第11章“延迟渲染”

**深度输出。**如前所述，像素着色器允许使用SV\_depth语义为深度测试和深度写入手动指定值。当启用MSAA且未对每个采样执行着色器时，输出到SV\_Depth会导致在测试每个采样点的深度时使用单个值。对于被多个三角形重叠的像素，这可能会导致MSAA无法正常工作。

3.12产出合并241

自定义覆盖口罩。在正常情况下，像素着色器输出的值基于在光栅化器阶段执行的覆盖测试生成的覆盖遮罩。但是，像素着色器可以使用SV\_覆盖语义输出自己的覆盖掩码。值输出是一个uint，其中每个位对应于渲染目标中的一个子样本。将位设置为1会导致将输出值写入子样本，而将其设置为0会阻止写入。该语义的明显用途是实现alpha到coverage透明度的自定义掩码，而不是依赖大多数硬件中实现的固定屏幕空间掩码。另一个可能的用途是实施MSAA，即延期投标，这将在第11章“延期提交”中进一步探讨

**3.11.4像素着色器管道输出**

由于像素着色器阶段将其输出发送到管道的末尾，即输出合并阶段，因此它可以写入的输出属性受到限制。输出合并只能处理上述颜色值和深度，并且不能接收附加信息。这是由着色器编译器强制执行的，它要求任何输出属性都限制为表示颜色和深度的系统值语义，即SV\_Target[n]和SV\_depth。

这些属性的唯一例外是SV\_DepthGreaterThan、SV\_depthlesshan和SV\_覆盖语义。前两种方法提供了在写入SV\_Depth属性时继续启用分层z剔除算法的机制。因此，如果未在像素着色器中手动修改深度，则写入这些语义属性中的任何一个都是无用的。SVCoverage语义使使用自定义子样本覆盖率掩码成为可能，如上所述。由于它指定了要在MSAA渲染目标中使用的模式，因此也应仅在使用MSAA渲染目标的情况下使用。

## 3.12产出合并

管道中的最后一站是输出合并阶段。这是一个固定的函数阶段，从像素着色器阶段接收颜色和深度结果，然后将这些结果合并到绑定到该阶段的渲染目标中进行输出。但是，此阶段提供的功能远远多于向资源写入颜色和深度值。输出合并还通过深度测试执行可见性确定，深度测试实现了传统的Z-buffer算法。此外，它还可以执行模具测试，以精确控制渲染目标的哪些区域被写入。我们将研究这两种方法是如何实现的

242                                                                                                                                                     3.渲染管道



图3.78。输出合并阶段。

本节后面将详细介绍测试功能。图3.78突出显示了输出合并阶段的位置。

输出合并还能够修改通过混合函数从像素着色器传递给它的颜色值。此可配置功能允许多种不同的混合模式，可用于将当前像素着色器结果与渲染目标的现有内容相结合。此混合功能，再加上处理MSAA渲染目标、多个渲染目标和渲染目标阵列资源的能力，使得输出合并在生成最终渲染图像时发挥重要作用。

### 3.12.1输出合并管道输入

因为它是管道中的最后一个阶段，所以输出合并阶段的输入是像素着色器阶段生成的完整数据集。输出合并的主要输入是由像素着色器程序生成的颜色值。接收到的颜色值的数量取决于像素着色器中声明的输出签名，但通常，输出颜色的数量将与已绑定到MRT配置的输出合并的渲染目标的数量相匹配。这有一个例外，即从像素着色器输出两种颜色以用于双源混合。这一概念将在“输出合并阶段处理”部分的混合部分进一步讨论。

输出合并阶段接收的第二个主要输入是片段的深度值。该值可以从光栅化器阶段接收，也可以从像素着色器阶段接收（如果在像素着色器程序中手动修改）。这些深度值是深度测试的输入，最终将用于更新深度缓冲区内容。

如“像素着色器管道输出”部分所述，使用MSAA时，每个子样本都会输出这两个值。在这种情况下，可以从像素着色器接收额外的输入系统值语义。SV\_覆盖语义指示像素着色器输出将写入哪些子样本。此子样本选择过程在输出合并阶段自动执行，不需要应用程序进行额外的工作，但了解输出数据最终如何在渲染目标中结束仍然很重要。

3.12产出合并243

### 3.12.2输出合并状态配置

输出合并阶段由一对状态对象（深度模具状态和混合状态）控制。除此之外，输出合并还可以绑定资源，这些资源最终表示渲染管道的最终输出。在本节中，我们将考虑如何配置这些属性中的每一个，以及处理每个可能的状态和资源的限制。

深度模板状态

深度测试和模具测试都使用相同的状态对象ID3DllDepthStencilState进行配置。与Direct3D 11中的所有资源一样，此对象是通过设备接口创建的。深度模具状态是使用ID3DllDevice:：Create DepthStencilStateQ方法创建的，该方法使用指向包含所需状态选项的描述结构的指针。清单3.25列出了深度模具状态描述的成员。

s t r u c t D3D11\_深度\_模板描述{

BOOL-DepthEnable；

D3D11深度写入掩码深度掩码；

D3D11比较函数深度函数；

BOOL S t e n c i l e n a b l e；

UINT8模板读取掩码；

UINT8 S t e n c i l W r i t e M a S k；

D3D11\_-DEPTH\_-STENCIL0P\_-DESC正面；

D3D11\_DEPTH\_STENCIL0P\_DESC BackFace；

}

清单3.25。D3D11\_DEPTH\_STENCIL\_DESC结构及其成员。

结构的前三个参数配置深度测试功能，其余参数用于配置模具测试。创建深度模具状态后，它是不可变的，不能更改。如果应用程序要求另一个状态处于活动状态，则必须创建一个单独的状态对象，并将其绑定到管道中以代替当前状态。这是通过ID3DllDeviceContext:：OMSet DepthStencilStateQ方法实现的。通常，有一个相应的get方法来检索绑定到管道的当前状态。“输出合并阶段处理”一节将更详细地讨论这些成员的工作细节。

混合状态

输出合并支持的第二个状态对象是ID3D11BlendState。此混合状态对象控制颜色值在写入之前如何混合在一起

244                                                                                                                                                      3.渲染管道

输出渲染目标。此阶段对象是使用ID3DllDevice:：Create BlendState（）方法创建的，该方法使用指向描述结构的指针，该描述结构指定状态对象将表示的配置。清单3.26显示了混合状态描述的内容。

结构D3D11\u混合\u描述{

BOOL-alphatocoverageable；

布尔独立不可更改；

D3D11\_RENDER\_TARGET\_BLEND\_DESC RENDER TARGET[8]；

}

清单3.26。D3D11\_BLEND\_DESC结构及其成员。

第一个参数确定是否使用输出颜色的alpha值来确定子采样覆盖率，而不是光栅化器覆盖率值，或者如果像素着色器写入SV\_uu覆盖系统值，则确定SV\_uu覆盖系统值语义。这可以用于实现一种透明渲染形式，而无需首先对场景中的对象进行排序。该结构的第二和第三构件相互配合使用。IndependentBlendEnable参数指示在使用多个渲染目标时，是否为每个渲染目标指定单独的混合模式。如果该参数为true，则RenderTarget数组中的八个元素中的每个元素都为其各自的渲染目标提供混合状态。如果“独立混合”为false，则索引0处的元素将用于所有渲染目标。创建混合状态对象后，它将通过ID3DllDeviceContext:：OMSetBlend state（）方法通过设备上下文绑定到管道。和往常一样，有一个相应的get方法来检索当前绑定到管道的状态。我们将在“输出合并阶段处理”部分探讨混合配置的细节。

渲染目标状态

输出合并的渲染目标状态表示接收整个管道中执行的所有渲染计算结果的实际资源。可以使用相对大量的不同配置来接收此输出，当然，对于可以一起使用的渲染目标组合也有限制。我们将首先讨论可以绑定到输出合并的各种类型的渲染目标，然后看看应用程序如何操纵它们。

输出合并有八个渲染目标槽和一个深度模具槽。每个渲染目标通过渲染目标视图（RTV）绑定到管道，深度模具目标通过深度模具视图（DSV）绑定到管道。在传统渲染配置中，单个渲染目标与单个深度目标绑定。然而，这不是一项要求。

3.12产出合并

应用程序可以同时绑定1到8个渲染目标，以生成同一光栅化场景的多个版本。这些渲染目标的类型（如Texture2D、TextureDarray等）和大小（包括宽度、高度、深度、数组大小和采样数）必须匹配，但格式可能不同。绑定多个渲染目标时，称为多渲染目标（MRT）配置。

此排列还使用单个深度模具目标，即使使用了多个渲染目标。由于深度模具目标用于执行深度和模具测试，因此它的类型和大小必须与作为渲染目标绑定的资源相同，但其格式将是深度模具格式之一。如上所述，这些大小匹配要求也适用于MSAA渲染目标。如果为输出绑定了MSAA渲染目标，则深度模具目标还必须是具有相同子采样数的MSAA资源。此外，资源的数组大小必须在渲染和深度模具目标之间匹配。您可能还记得，在第2章中，可以将资源创建为阵列资源，其中在同一资源中创建了多个纹理切片。如果渲染目标具有六个阵列切片以生成立方体贴图，则深度模具目标还必须具有六个阵列切片。

最后一种配置也可能根本不使用渲染目标。可以仅将深度模具视图绑定到管道。在这种情况下，通过将空值绑定到每个渲染目标槽，而将深度模具视图绑定到常规渲染配置中。这通常用于在继续进行其他渲染过程之前，使用场景的深度信息填充深度模具目标。

**MRT与渲染目标阵列。**应注意MRT和渲染目标阵列之间的区别。可以创建八个单独的纹理资源，然后将它们绑定到输出合并阶段，以便在MRT配置中使用。还可以创建具有八个纹理切片的单个纹理阵列资源，然后将该资源绑定到输出合并，以便在单个渲染目标配置中使用。尽管它们将提供相同数量的有效渲染目标，但在使用这些配置的机制和它们的功能方面存在一些差异。MRT配置使用多个渲染目标插槽，每个要使用的渲染目标插槽一个。另一方面，基于阵列的资源在输出合并中仅占用一个渲染目标槽。这会在两种配置之间产生另一个差异，即MRT设置同时写入其所有渲染目标，而基于阵列的设置一次只写入一个切片。由于像素着色器可以同时写入所有MRT渲染目标，因此只需调用单个像素着色器即可写入所有目标。由于像素着色器只能写入基于阵列的配置中的单个渲染目标，因此需要调用一个像素着色器才能写入其每个纹理切片。

这种情况似乎表明MRT配置是一个更好的选择，因为这可以减少像素着色器调用的数量，同时仍然写入相同的设置

246                                                                                                                                                      3.渲染管道



图3.79。使用MRT和基于阵列的资源对管道进行概念性拆分。

渲染目标的数量。但是，渲染目标阵列资源也有几个优点。在某种意义上，我们可以说MRT配置对其所有渲染目标使用相同的光栅化。这意味着，如果将三角形栅格化到渲染目标的右上角，则它将显示在写入的所有渲染目标的相同位置。渲染目标阵列每个都使用可以根据SV\_RenderTargetArrayIndex系统值进行自定义的单个光栅化。这允许在光栅化之前对几何体使用不同的视图变换，因此也允许将基本体光栅化到渲染目标中的不同位置。

我们可以将这两种配置视为拆分管道以最终写入多个渲染目标，但它们将管道拆分到不同的位置。MRT配置在像素着色器阶段分割管道，而基于阵列的配置在光栅化器阶段之前分割管道。这两种配置在不同的情况下都很有用，具体取决于需要哪种类型的管道输出。这种差异如图3.79所示。

关于渲染模板和深度模板目标的绑定还有一个方面需要考虑。我们在上面提到，这些资源与资源视图（RTV和DSV）绑定。由于资源的子资源可以由资源视图指定，因此可以通过使用较小部分资源的资源视图将较大的资源绑定到管道。当这与上面提到的所有其他选项和配置一起考虑时，开发人员可以使用大量不同的选项来创建专门的渲染算法。

**绑定渲染目标。**渲染目标和深度模具目标都绑定到单个设备上下文方法调用中的输出合并。ID3DllDeviceContext:：OMSetRenderTargets（）方法（如清单3.27所示）除了一个指定数组中目标数量的整数外，还接受一个指向指针数组的指针以呈现目标视图，以及一个指向深度模具视图的指针。管道将包含对

3.12产出合并247

绑定后渲染目标，它将保留引用，直到渲染目标替换为空引用。

另一种资源类型可以绑定到输出合并阶段。我们在pixel shader stage一节中看到，它可以使用无序访问视图（UAV）。但是，像素着色器阶段无法接受UAV进行绑定。相反，它们与渲染目标绑定到输出合并阶段的方式相同。使用无人机时，它们会与ID3D11DeviceContext:：OMSetRenderTargets和Unordered AccessView（）方法绑定。此方法的前三个参数与仅渲染目标版本相同，而其余四个参数用于绑定无人机。总共有八个无人机插槽可用，受此呼叫影响的插槽范围通过UAVStartSlot和NumUAVs参数选择。ppUnorderedAccessView参数是指向要绑定的UAV数组的指针。当然，该阵列中的无人机数量必须与NumUAVs参数中指定的视图数量相匹配。

此方法的最后一个参数是指向UINT值数组的另一个指针。这些值提供当前缓冲区计数器值，以便在追加/使用缓冲区中使用。如第2章所述，无人机使用的缓冲器包含一个内部计数器，指示缓冲器中存在的元件数量。此计数器隐藏在缓冲区中，由运行时维护。但是，绑定时存储在这些计数器中的值由传入该数组的内容控制。这使得应用程序可以在需要时有效地重置缓冲区中的数据，或者如果传递了-1的值，则保持当前的内部缓冲区计数器值。

绑定的渲染目标和无人机总数不得超过八个。但是，任何加起来等于或小于8个的组合都是允许的，包括使用8个无人机和不使用渲染目标。如果没有绑定到管道的渲染目标，则无人机表示整个管道的唯一输出。清单3.27显示了将渲染目标绑定到输出合并的两种方法。

无效OMSetRenderTargets(

国际新视点，

ID3D11RenderTargetView\*\*PPrenderTargetView，

ID3D11DepthStencilView\*pDepthStencilView

);

无效OMSetRenderTargetsandUnorderedAccessView(

国际新视点，

ID3D11RenderTargetView\*\*PPrenderTargetView，

ID3D11DepthStencilView\*ppDepthStencilView，

UINT UAVStartSlot，

乌因特·努穆阿夫斯，

ID3DllUnorderedAccessView\*\*ppUnorderedAccessView，const UINT\*pUAVInitialCounts

**);**

清单3.27。用于将资源绑定到输出合并阶段的设备上下文方法。

248                                                                                                                                                     3.渲染管道

只读深度模具视图

Direct3D 11中的另一个新功能是能够在输出合并阶段将深度模具资源用作深度/模具目标，并且还能够通过其中一个可编程着色器阶段中的着色器资源视图同时查看其内容。乍一看，这似乎违反了不能同时从不同资源读取和写入资源的规则，因为深度模具资源将在输出合并的深度测试部分写入。

但是，如果使用适当的只读标志创建深度模具视图，这将确保输出合并阶段不会写入资源。相反，它仅由输出合并读取，以确定深度测试是否通过或失败（如果深度测试当前已启用）。这将有效地使资源绑定到只读访问点的两个管道位置都保持同步读/写规则。该配置如图3.80所示。这非常有用，因为二次渲染过程可以使用深度缓冲区内容，而无需将其复制到另一个资源中，同时该资源也用于深度测试。37

### 3.12.3输出合并阶段处理

输出合并阶段执行两种类型的操作：可见性测试和混合操作。可见性测试提供可配置的操作，用于确定是否应将每个特定片段混合到附加到输出合并的输出资源中。如果片段通过了深度测试和模具测试，则会将其传递给混合函数，以便与渲染目标组合。混合功能也是可配置的，它允许多种方法将像素着色器输出应用于渲染目标。在本节中，我们将遵循片段在输出合并阶段中所遵循的路径。我们首先讨论两个可见性测试中的每一个，以便更好地理解测试如何运行以及可以对其进行哪些修改。接下来，我们将研究混合函数如何执行其职责，并检查可用的混合模式。

能见度测试

输出合并阶段对传递给它的每个片段同时执行两个可见性测试：深度测试和模具测试。这两个测试都使用深度模具资源，该资源通过深度模具视图（DSV）绑定到输出合并阶段，并且需要为深度模具资源选择适当的数据格式。这个

37 有两个可用标志，D3D11\_DSV\_只读\_深度和D3D11\_DSV\_只读\_模具，分别对应于深度/模具目标的深度和模具部分。



|  |  |
| --- | --- |
| 3.12产出合并 | 249 |

图3.80。在管道的多个位置使用的单个资源，包括输出合并中的深度模具视图。

250                                                                                                                                                      3.渲染管道

可用于此类资源的格式通常包含一部分用于存储当前深度值，另一部分用于存储当前模具值。这种格式的一个示例是DXGI\_F0RMAT\_D24\_UN0RM\_S8\_UINT，其中24位专用于保持深度值，8位专用于保持模板值。有些格式只包含深度值，没有专门用作模具缓冲区的数据。在这些情况下，模具测试将被禁用并始终通过。我们将看到如何在其各自的能见度测定测试中使用这些量。

启用后，将对发送到输出合并的每个片段执行深度测试和模具测试。如果启用MSAA，则对管道生成的每个子样本执行测试。这允许以光栅化器阶段使用的相同粒度执行可见性测试，并提高几何体相交区域的图像质量。

**模板测试。**我们将考虑的第一个可见性测试是模板测试。此测试允许应用程序对渲染目标执行掩蔽操作，以控制何时将特定片段写入目标渲染目标。测试本身是可配置的，具有相对较多的可能配置。此测试有两个部分。第一个是实际测试本身，第二个是用于更新模具缓冲区的更新机制。为了理解测试是如何工作的，我们将首先定义模具测试实现的等式。等式（3.6）提供了模板测试的伪代码。

（StencilRef和StencilMask）CompFunc（StencilBufferValue和StencilMask）

(3.6)

等式（3.6）的计算结果为真或假，其中真结果表示测试通过，假结果表示测试失败。模具测试的各个参数是“输出合并状态配置”一节中讨论的可配置状态的组合，我们将在这里对每个参数进行审查。从等式的左侧开始，我们得到了模具参考值，它与模具掩码按位AND。模具参考值是应用程序在ID3DllDeviceContext:：OMSetDepthStencilState（）方法中提供的无符号整数值，而模具掩码设置为深度模具状态描述的StencilReadMask成员。

如果我们花一点时间来考虑这个等式所代表的，我们会看到一个基于参数的左侧的参数，该参数可以用设备上下文方法来配置。右侧是基于当前模具缓冲区值的参数。这两个参数都被屏蔽，以允许选择其源变量中包含的位的子集。然后将这两个参数与API提供的几个比较函数之一进行比较。因此，该等式的所有三个主要部分都是可配置的，允许进行多种可能的测试。

3.12产出合并251



图3.81。模具测试的每个参数的位寄存器。

因为这个测试是高度可配置的，所以很难想象它是如何运行的。为了帮助阐明这一功能，我们将提供一个小的示例场景，用于通过模具测试的单个片段。在这种情况下，模具参考值将设置位0、1和3，而模具缓冲区设置位1和2。模板掩码设置了位0和位1。这些值如图3.81所示。

根据选择的比较功能，可以为模具测试找到不同的结果。例如，如果使用D3D11\u COMPARIS0N\u LESS，模具测试将失败，因为等式的左侧大于右侧。当然，如果使用D3D11\u比较0n\u更大的值，则此功能可以反转。可用的比较函数如清单3.28所示。

枚举D3Dll\u比较\u函数{

D3D11\u比较从未，

D3D11（小于等于0），

D3Dll\_比较\_相等，

D3Dll\_比较\_小于\_相等，

D3Dll\_比较\_更大，

D3Dll\u比较\u不相等，

D3D11比较大于等于，

D3D11\u比较始终

}

清单3.28。D3D11\_比较函数枚举。

考虑到能够指定模具参考值、模具掩码和比较函数，开发人员或多或少可以完全控制模具测试的操作方式。测试完成后，需要采取一些措施，具体取决于测试是通过还是失败。事实上，所采取的行动取决于模板测试结果和深度测试结果（我们将在下一节中讨论）。保持

252                                                                                                                                                      3.渲染管道

由于模具测试的可配置性质，可以在深度模具状态对象中指定各种可能性。可以为模板和深度测试结果的以下每个组合设置不同的操作：

1.        模板测试失败。

2.        模板测试通过，但深度测试失败。

3.        模板测试通过，深度测试通过。

要采取的操作可以从D3D11\_STENCIL\_uuOP枚举的一个成员中选择，如清单3.29所示。

枚举D3D11\u模具\u 0P{

D3D11\_模板\_0P\_保持，

D3D11\_模具\_0P\_ZER0，

D3D11\_模具\_0P\_更换，

D3Dll\_模板\_OP\_INCR\_SAT，

D3D11\_模板\_0P\_decru\_SAT，

D3D11\_模具\_0P\_倒置，

D3D11\_模板\_0P\_增量，

D3Dll模具操作说明

}

清单3.29。D3D11\_模具\_0P枚举。

这些选项确定测试完成后如何处理模具缓冲区值。有保留现有模具缓冲区值的选项；将其清除为零；将其替换为模板参考值；以及递增、递减或反转。同样，这些选项在为各种不同的算法使用模具缓冲区方面提供了很大的自由度，因为根据测试结果，可以采取不同的操作。在将模具值写入模具缓冲区之前执行最后一个操作。由模具操作生成的值在存储之前是按位的，并与模具写入掩码进行编码。

在本节中，我们看到了完整模具测试的配置。但是，还有一些其他配置需要讨论。可以使用“深度模具状态”对象的“正面”和“背面”成员分别为正面和背面配置上面显示的所有设置。这些成员实际上是封装一组模具测试配置的结构（比较函数和在每个测试结果案例中执行的操作）。这允许执行特殊测试，对正面和背面执行不同的选项。使用不同设置的最常见示例

3.12产出合并253

阴影体积算法就是以这种方式实现的。除了能够对正面和背面使用不同的配置外，还可以使用StencilEnable布尔参数启用或禁用整体测试。如果模具测试被禁用，则始终认为它已通过，并且不会因为模具测试而剔除片段。38

深度测试。在进行模板测试的同时，还进行了深度测试。该测试基本上实现了一个用于执行可见性测试的经典Z缓冲区算法（Williams，1978）。基本概念是保留与预期渲染目标大小相同的第二个缓冲区。但是，当一个基本体被光栅化时，它会存储每个片段的标准化设备坐标的Z分量，而不是将光栅化的颜色值存储在缓冲区中。这将生成一个缓冲区，其中包含片段位置的后分割Z分量，其值在[0.0,1.0]范围内。在输出合并阶段，该Z缓冲区被实现为深度模具缓冲区的深度部分，并且通常被称为深度缓冲区，因为Z分量表示与观看者的距离的度量。

当存储这些深度值时，任何被光栅化以创建片段的附加原语都可以将其自身的深度值与存储在深度缓冲区中的深度值进行比较。如果缓冲区中存储的深度值小于（更靠近查看器）新片段的值，则可以丢弃新片段，因为它位于场景中另一个对象的后面。如果存储的深度缓冲区值大于新片段（远离查看器），则新片段的颜色值将传递给管道的其余部分，并更新Z缓冲区值以表示新的可见性信息。这样，Z缓冲区提供了每像素（或者如果使用MSAA，则为每样本）可见性确定。

输出合并使用几个额外的配置实现Z-buffer算法，用于准备深度值，以及实际执行深度比较并将结果写入Z-buffer。这些配置主要包含在深度模具状态对象中，视口除外。我们将以与模板测试相同的方式跟踪深度测试中的片段，以更好地了解该过程是如何工作的。

深度测试可以使用深度模具状态对象的DepthEnable成员启用或禁用。此参数仅确定是否执行深度测试；它不控制深度写入功能是启用还是禁用。深度值直接从光栅化器阶段接收，或者如果像素着色器程序修改深度值，则从像素着色器阶段接收深度值。然后将该值钳制为用于生成片段的视口结构中指定的“最小深度”和“最大深度”值。该夹紧以深度缓冲区格式的适当方式执行。夹紧深度范围后，将从深度模具缓冲区读取深度值，并使用可选的深度比较功能（selected）比较这两个值

38 本章“光栅化器”一节首先介绍了该算法。

254                                                                                                                                                      3.渲染管道

使用深度模具状态对象的DepthFunc成员。清单3.30提供了所有可用的比较函数。

枚举D3D11比较函数{

D3D11\u比较从未，

D3D11\u比较\u少，

D3D11\_比较\_相等，

D3D11比较不相等，

D3D11比0大，

D3D11与同等标准相比，

D3D11比较大于等于，

D3D11\u比较始终

}

清单3.30。D3D11\_比较函数枚举。

这些比较与左侧的片段深度值和右侧的深度缓冲区值进行。如果比较结果为true，则深度测试已通过，并且片段在过程中继续。如果比较结果为false，则深度测试结束，并丢弃片段。例如，标准比较是使用D3Dll\_comparison\_LESS，它提供了与原始Z-buffer算法类似的功能。如果片段深度小于深度缓冲区值，它将通过深度缓冲区测试。这表明新片段更靠近查看器，并且应该可见。

如果深度测试失败，片段将被丢弃。如果通过，它将被传递到混合功能（将在下一节中介绍）。深度缓冲区可能会根据一些条件进行更新。如果深度测试和模具测试都已通过，则深度值的命运取决于是否启用深度写入。这是使用深度模具状态的DepthWriteMask成员指定的。如果该值设置为D3D11\_DEPTH\_WRITE\_MASK\_ALL，则深度值可在深度缓冲区中更新；否则，将丢弃深度值。

深度缓冲区可以用于深度测试，但仍然有一个不允许写入的配置，这似乎违反直觉。事实上，在许多情况下，这是现代渲染方案中的首选行为。使用上述标准Z缓冲区技术，使用第一个渲染过程用值填充深度缓冲区是很常见的。然后，任何后续过程都将使用深度缓冲区，因为之前所有几何体都已光栅化。实际上，可见性已经确定，因此不需要启用深度写入。禁用深度写入时，可以获得性能优势，因为深度缓冲区只需要读取，而不需要写入。

这种“只读深度缓冲区”的概念可以更进一步。您可能还记得，在第2章中，我们看到可以创建带有标志的深度模具视图，以表明

3.12产出合并255

深度组件或模具组件或两者都可以创建为只读。这意味着视图本身确保只能从中读取资源，这允许在管道中的多个位置使用资源。该资源可以绑定在深度测试中使用，同时用作着色器资源。当深度缓冲区用作后续渲染过程的输入时，这可能非常有用，但仍需要同时用于深度测试。

混合

如果一个片段在模板测试和深度测试中都幸存了下来，它将被传递给混合函数。混合功能可以在将两个可选值写入输出渲染目标之前将其组合，用于组合值的功能也是可选的。此功能传统上用于执行alpha混合以实现部分透明的渲染材质。然而，由于有大量可能的混合源、操作和写入选项，因此有许多附加功能可用于其他不太传统的技术。混合功能由混合状态对象中包含的配置控制。清单3.31提供了可以在混合状态下操作的列表成员。

S t r u c t D3D11混合描述{

BOOL-alphatocoverageable；

布尔独立不可更改；

D3D11\_渲染\_目标\_混合\_描述渲染目标[8]；

}

清单3.31。D3D11混合描述结构。

在这里，我们找到两个顶级配置，然后是八个渲染目标混合描述结构的数组。第一个顶级成员是alphatocoverageable参数，我们已经在本章的“像素着色器”一节中讨论过。第二个顶级成员是另一个布尔值，它确定是将独立混合输出合并阶段中为输出绑定的所有渲染目标，还是所有渲染目标都将使用相同的混合配置。正如您现在可能已经猜到的，混合配置存储在混合状态的八元素数组成员中，因为最多有八个渲染目标槽可用。如果通过将IndependentBlendEnable设置为true启用了单个混合，则这些数组元素中的每个元素都定义了用于相应渲染目标窗的混合模式。如果禁用独立混合，则所有渲染目标将使用阵列索引0中的混合配置。如果不需要，则应禁用独立混合以获得更好的性能。

256                                                                                                                                                    3.渲染管道

为了深入了解混合配置是如何使用的，我们将使用与模具和深度测试相同的范例，并遵循通过混合功能运行的片段路径。清单3.32显示了D3D11\_RENDER\_TARGET\_BLEND\_DESC结构的成员。

S t r u c t D3D11\_渲染\_目标\_混合\_描述{

布尔可调和；

D3D11\_混合料；

D3D11\_混合料和混合料；

D3D11\_混合物\_0P BlendOp；

D3D11\_biu\_END SrcBlendAlpha；

D3D11\_混合料，用于制备BLENDALPHA；

D3Dll\_BLEND\_OP BlendOpAlpha；

UINT8 RenderTargetWriteMask；

清单3.32.D3D11\_RENDER\_TARGET\_BLEND\_DESC结构。

第一个成员BlendEnable完全按照其名称所述执行。它启用或禁用混合功能。我们将假设在本例中启用了混合功能。接下来，我们将看到两组三种状态。BlendEnable后面的三个成员用于混合颜色值（片段颜色的RGB组件），接下来的三个用于混合alpha值（片段颜色的A组件）。这种混合状态的分离允许对alpha和颜色使用完全不同的混合模式，这可以让它们用于不同的目的。

为颜色和alpha提供的三个成员用于定义混合方程。混合方程的两个数据源由用于颜色混合的SrcBlend和DestBlend参数以及用于alpha混合的SrcBlendAlpha和DestBlendAlpha选择。这些参数将其数据源定义为t的一个成员他使用D3D11\_混合枚举，如清单3.33所示。

枚举D3D11\_混合{

D3D11\_混合物\_ZER0，

D3D11混合液，

D3D11\_混合物\_SRC\_C0L0R，

D3D11混合料库存SRC C0L0R，

D3D11\_混合物\_SRC\_α，

D3D11\_混合\_INV\_SRC\_ALPHA，

D3D11\_混合\_DEST\_ALPHAJ

D3D11混合料库存目的阿尔法，

D3D11\u混合物\u目的地\u C0L0R，

D3D11混合料库存目的地C0L0R，

D3D11混合型SRC阿尔法SAT，

D3D11混合混合混合工厂，

D3D11混合料库存混合料制造商，

|  |  |
| --- | --- |
| 3.12产出合并 | 257 |

D3D11\_混合物\_SRC1\_C0L0RJ

D3D11混合颜色，

D3D11\_混合物\_SRC1\_α，

D3D11\_混合\_INV\_SRC1\_ALPHA

}

清单3.33.D3D11\_混合枚举。

如您所见，有大量可能的数据源。此列表中的常规选项包括数据源，例如常量值和颜色/alpha值，以及几个可能的修饰符。颜色和alpha源都提供源修饰符和目标修饰符，其中源是来自当前片段和目标的值“目标”渲染目标中的值。每个值还提供一个“反向”值，该值是使用1减去该值生成的。例如，D3D11\_BLEND\_SRC\_C0L0R选择当前片段颜色作为其源。但是，D3D11\_BLEND\_INV\_DEST\_C0L0R选择目标渲染目标值的倒数作为其源。除了这些修改后的颜色和alpha值之外，还可以使用其中一个多个常量值。可以选择的常量值为0、1或由应用程序使用ID3DllDeviceContext:：0MSet BlendState（）方法设置的混合因子。混合因子还提供一个反向值。

您可能还注意到，名称中有几个参数带有SRC1修饰符，例如D3D11\_BLEND\_SRC1\_C0L0R和D3D11\_BLEND\_INV\_SRC1\_ALPHA。这些值实际上取自像素着色器中的第二个输出寄存器，可以与第一个输出寄存器同时用于混合操作。这使两个源q从当前片段读取数量，而不是使用目标缓冲区的内容。使用两个输出寄存器值称为双源颜色混合，只能与单个渲染目标输出一起使用（否则，第二个颜色值将传递给第二个渲染目标）。

一旦选择了数据源，就必须选择混合操作。此操作将两个数据源值合并为一个值，以写入输出渲染目标目标。清单3.34中提供了可用操作。可用操作非常简单，并且实现了它们的名称暗示。有三种算术运算（A+B、A-B、B-A），以及最小和最大选择选项。

枚举D3D11\u混合\u 0P{

D3D11\_混合\_0P\_添加，

D3D11\u混合\u 0P\u相减，

D3D11\u BLEI\ID\u 0P\u REV\u减法，

D3D11\u混合物\u 0P\u最小值，

D3D11\_混合\_0P\_最大值

}

清单3.34.D3D11混合操作枚举。

258                                                                                                                                                      3.渲染管道

如前所述，可以分别对颜色值和alpha值执行这些混合操作。一旦混合操作完成，就可以使用完整的四分量RGBA颜色进行写入。混合过程的最后一步是使用渲染目标写入掩码来确定输出颜色的哪个通道将l实际被写入。这是在混合状态结构的RenderTargetWriteMask成员中指定的。RGBA通道对应于掩码的位位置0、1、2和3。设置位后，通道将写入渲染目标。

### 3.12.4输出合并管线输出

在本节中，我们已经了解了在将管道数据写入绑定到输出合并阶段的资源之前对管道数据执行的操作。管道执行总共可以写入三种不同类型的资源：渲染目标、无序访问视图和深度模具目标。

渲染目标接收来自混合函数的输出。写入渲染目标的每个片段在混合并最终写入渲染目标之前必须经过模具测试和深度测试。如果正在使用MRT，则每个渲染目标将有一个从像素s写入的混合值hader。如果使用渲染目标数组，则在光栅化器阶段的输入处使用SV\_RenderTargetArrayIndex系统值语义确定适当的渲染目标切片。因此，当数据到达混合函数的输出时，已选择适当的纹理切片。

相反，写入UAV的数据直接由像素着色器阶段控制，无需通过任何测试即可成功写入输出资源。当像素着色器修改资源时，这些更改会立即提交到GPU，并在GPU的内存系统将值写入内存后立即生效因此，在设计算法时，不能假设其结果立即可用。

深度模具目标稍微复杂一些。输出数据必须通过深度测试，并且要写入到输出资源的深度部分的模具测试必须通过。此外，必须在深度状态对象中启用深度写入。根据深度测试和模具测试通过的组合，模具数据可以进行不同的更新。Si由于这些都是可配置的设置，所以或多或少由应用程序决定何时以及如何通过进入输出合并阶段的数据流更新模具数据。

管道执行完成后，这些资源中每个资源的修改内容可用于进一步的渲染过程和计算过程，甚至可用于CPU的直接操作。对于MSAA渲染，在使用资源之前还需要执行一个步骤。渲染完成后，通常需要设计可用于将MSAA渲染目标解析为非MSAA纹理。在Direct3D 11中，此操作已完成

3.13高级管道功能259

通过调用ID3DllDeviceContext:：ResolveSubresource（）方法。执行此解析将单个子样本的值组合为单个值，从而生成适合在屏幕上显示的纹理，或使用普通着色器采样方法进行采样。执行解析时应用的过滤器特定于硬件和驱动程序以及质量级别指定但是，在大多数情况下，使用框过滤器。对于框过滤器，所有子样本的权重相等，使其成为子样本值的简单算术平均值。

## 3.13高级管道功能

在逐步地跨过整个管道之后，我们已经详细地涵盖了大量的功能。事实上，本章中有太多的信息可以考虑所有可以执行的不同类型的计算。r从更高的层次上分析每个阶段的功能，试图阐明它们通常是如何使用的。这些分组是基于典型用途进行概括的，但它们不一定反映任何类型的需求。毕竟，管道的灵活性允许非常有创意和不寻常的用途，因此这些只是作为sta呈现供读者从中构建的rting点。

### 3.13.1顶点操作

管道的早期阶段通常用于构建和操作表示将渲染的几何曲面的顶点。这包括由输入汇编程序构建顶点，以及由顶点着色器执行的单个逐顶点操作。在最低级别，这两个阶段执行他介绍了在整个管道中看到的输入几何体的最基本操作。由于与原始输入几何体非常接近，因此在管道的早期对几何体进行修改是很自然的。传统上，变换矩阵应用于顶点着色器，而顶点着色器本质上是操纵几何体模型的度量属性。

一般来说，由于这两个阶段都是在逐顶点级别运行的，因此它们很可能在单个管道执行期间执行得最少。对于每个绘制调用，输入汇编程序将生成模型的每个顶点，并将设置每个基本体，以便稍后在管道中进行进一步处理。从这两个流中，顶点着色器w每个顶点只运行一次。与管道中的后期阶段不同，输入汇编程序和顶点着色器以相对较低的频率执行计算。虽然输入汇编程序是固定功能阶段，但顶点着色器是可编程的，可以

260                                                                                                                                                      3.渲染管道

用于实现自定义例程。通过在顶点着色器中执行可传递到后续阶段的计算，而不是在后续阶段直接计算所有内容，我们可以利用此自定义处理。

在顶点着色器中执行向几何体表面提供相对低频信息的任何其他计算也是合乎逻辑的。例如，在顶点着色器中计算环境照明计算通常是足够的，因为它不会在整个模型中快速变化。通过在在顶点级别，我们将在管道中稍后的每个着色器调用中释放少量计算。

### 3.13.2细分操作

下一组管道阶段将实现渲染管道的细分功能。外壳着色器、细分器和域着色器一起使用，以提供非常灵活和健壮的细分系统。这可以看作是以前基于顶点的过程的扩展，因为它最终也会生成顶点。在许多方面，域着色器可以被视为顶点着色器的重复，在处理顶点之前还需要承担一些创建顶点的额外责任。

但是，细分阶段旨在放大几何级别上的可用细节，这与输入汇编程序和顶点着色器的功能不同。这意味着域着色器的总调用次数将显著高于顶点着色器。因此对于需要在域着色器中执行的计算，应尽可能多地使用顶点着色器，而不是域着色器。如果所有条件相同，则应选择顶点着色器而不是域着色器。

即便如此，细分系统也为开发人员提供了一些独特的机会。通过放大可用于表示几何体的顶点数量，我们可以比Direct3D早期版本更早地在管道中执行更高频率的计算。在极端情况下，我们可以为每一个顶点生成一个顶点ixel，使逐顶点数据的视觉保真度与逐像素数据的视觉保真度相等。我们可以利用这一优势，将计算转移到管道中的早期阶段，远离基于光栅的阶段，或者通过在顶点级别执行更复杂的计算来简化后期阶段。

### 3.13.3几何操作

几何体着色器和流输出阶段是非常专门的阶段，位于基于顶点的阶段和基于光栅的阶段之间

3.13高级管道功能261

在栅格化几何体之前，执行比基于顶点的阶段更高级别的几何操作。虽然这在某些情况下非常有用，但在大多数算法中使用几何体着色器的情况不太常见。使用几何体着色器的大多数算法使用几何体着色器，因为其特殊功能不可用在任何其他阶段。将点扩展为四边形是在任何其他阶段都无法实现的功能的一个好例子。

这可能至少部分是由于几何体着色器在Direct3D 10中首次亮相时性能不佳，并且相应地缺乏针对它的开发工作。然而，由于大多数当前一代GPU使用的共享处理器体系结构，因此有足够的处理能力来使用ge度量着色器。最大的挑战是确保阶段的内存使用与正在执行的任务适当平衡。例如，如果每个几何体着色器调用用于生成100个输出三角形，则可能需要重新评估算法，以利用细分阶段，而不是几何体etry着色器。但是，对于小规模几何体操作，内存使用与计算之间有更好的平衡。这使得点精灵扩展成为一个有吸引力的目标—它执行一些计算，并进行少量的数据放大。至少，看看是否有更多的算法被开发用于使用ge在未来几年内，或者它是否将继续作为一个专业阶段在管道中。

### 3.13.4基于光栅的操作

管道的最后一部分是基于光栅的阶段。这些阶段包括光栅化器、像素着色器和输出合并。由于在光栅化器中执行的数据放大，这些阶段的操作频率比以前的阶段高很多。这允许在这些阶段中执行更高频率的操作，例如这就是为什么经常使用像素着色器将所有细节添加到渲染中的原因。光栅化器通常不是渲染算法中的瓶颈，而像素着色器可能是计算或内存带宽瓶颈，因为执行了大量调用。

输出合并可能会带来一些不太为人所认识的性能问题。由于它会读取和写入深度模具缓冲区，并且在启用混合时可能会读取和写入渲染目标，因此输出合并可能会大大增加带宽使用率。如果不需要使用混合功能，请确保禁用该功能同样，如果没有理由期望深度缓冲区更新（例如在第二次或第三次渲染过程中），请确保禁用深度写入功能。

除了这些考虑之外，一整类算法刚刚开始开发，它们使用像素着色器中的无序访问视图。在任意位置执行自定义资源读写的能力为

262                                                                                                                                                      3.渲染管道

使用以前不可用的位置的数据填充资源。GPU制造商正在制作一些全新的演示，以非常有趣的方式有效地使用这一新功能。随着我们进一步探索这一功能，我们将看到开发人员如何操纵它以提高性能或图像质量。