3.8 像素着色器

在顶点、曲面细分和几何着色器执行它们的操作后，图元被裁剪并设置为光栅化，如前一章所述。管线的这一部分在其处理步骤中相对固定，即不可编程但有些可配置。遍历每个三角形以确定它覆盖哪些像素。光栅化器还可以粗略计算三角形覆盖每个像素的单元格区域（第5.4.2节）。与三角形部分或完全重叠的像素区域称为片元。

三角形顶点处的值，包括z缓冲区中使用的z值，在三角形表面为每个像素进行插值。这些值被传递给像素着色器，然后像素着色器处理片元。在OpenGL中，像素着色器被称为片元着色器，这可能是一个更好的名称。我们在本书中使用“像素着色器”以保持一致性。沿管线发送的点和线图元也会为覆盖的像素创建片元。

跨三角形执行的插值类型由像素着色器程序指定。通常我们使用透视校正插值，这样像素表面位置之间的世界空间距离会随着物体距离的缩小而增加。一个例子是渲染延伸到地平线的铁轨。轨道距离越远，铁路枕木的间距就越近，因为每个接近地平线的连续像素都行进了更多的距离。其他插值选项可用，例如屏幕空间插值，其中不考虑透视投影。DirectX 11 进一步控制何时以及如何执行插值[530]。

在编程方面，顶点着色器程序的输出，在三角形（或线）上进行插值，有效地成为像素着色器程序的输入。随着GPU的发展，其他输入也暴露出来了。例如，片元的屏幕位置可用于着色器模型3.0及更高版本中的像素着色器。此外，三角形的哪一边可见是输入标志。这一点对于在单个通道中，三角形的正面和背面渲染不同的材质很重要。

有了输入，像素着色器通常会计算并输出片元的颜色。它还可能产生不透明度值并可选择修改其z深度。在合并阶段，这些值用于修改存储在像素中的内容。光栅化阶段生成的深度值也可以通过像素着色器进行修改。模板缓冲区值通常不可修改，而是传递到合并阶段。DirectX 11.3允许着色器更改此值。在SM 4.0[175]中，雾计算和alpha测试等操作已从合并操作转变为像素着色器计算。

像素着色器还具有丢弃传入片元的独特能力，即不生成输出。图3.14显示了如何使用片元丢弃的一个示例。裁剪平面功能曾经是固定功能管线中的可配置元素，后来在顶点着色器中指定。随着片元丢弃可用，此功能可以在像素着色器中以任何所需的方式实现，例如决定裁剪体的并和或操作。

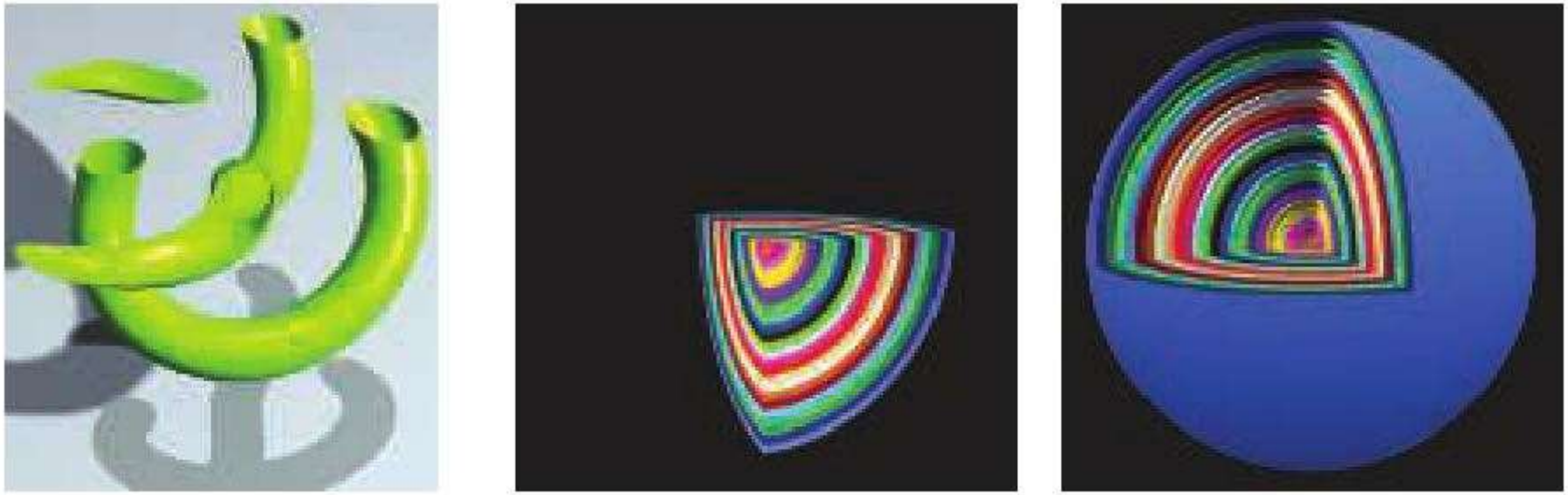

图3.14. 用户定义的剪裁平面。在左侧，单个水平剪切平面对对象进行切片。在中间，嵌套的球体被三个平面裁剪。在右侧，球体的表面仅在它们位于所有三个剪裁平面之外时才会被剪裁。（来自Three.js示例webgl裁剪和webgl裁剪交集[218]。）

最初，像素着色器只能输出到合并阶段，以供最终显示。像素着色器可以执行的指令数量随着时间的推移而显着增加。这种增加产生了多渲染目标 (MRT) 的想法。不是将像素着色器程序的结果仅发送到颜色和z缓冲区，而是可以为每个片元生成多组值并将其保存到不同的缓冲区，每个缓冲区称为渲染目标。渲染目标通常具有相同的x和y维度；一些API允许不同的大小，但渲染区域将是其中最小的。某些架构要求渲染目标具有相同的位深度，甚至可能具有相同的数据格式。根据GPU的不同，可用的渲染目标数量为四个或八个。

即使有这些限制，多渲染目标 (MRT) 功能仍然是更有效地执行渲染算法的有力助手。单个渲染通道可以在一个目标中生成彩色图像，在另一个目标中生成对象标识符，在第三个中生成世界空间距离。这种能力还产生了一种不同类型的渲染管管线，称为延迟着色，其中可见性和着色在单独的通道中完成。第一个通道存储在每个像素处有关对象位置和材质的数据。接下来的通道可以有效地应用照明和其他效果。此类渲染方法在第20.1节中描述。

像素着色器的局限性在于它通常只能在交给它的片元位置写入渲染目标，而不能从相邻像素读取当前结果。也就是说，当像素着色器程序执行时，它不能将其输出直接发送到相邻像素，也不能访问其他人最近的更改。相反，它计算的结果只影响它自己的像素。然而，这种限制并不像听起来那么严重。在一个通道中创建的输出图像可以让像素着色器在以后的通道中访问其任何数据。可以使用第12.1节中描述的图像处理技术处理相邻像素。

像素着色器无法知道或影响相邻像素结果的规则也有例外。一是像素着色器可以在计算梯度或导数信息期间立即访问相邻片段的信息（尽管是间接的）。像素着色器提供了任何内插值沿x和y屏幕轴每个像素的变化量。这些值对于各种计算和纹理寻址很有用。这些梯度对于诸如纹理过滤（第6.2.2

节）之类的操作特别重要，其过滤插值需要我们知道图像覆盖了多少像素。所有现代GPU通过以2×2为一组处理片元（称为四边形）来实现此功能。当像素着色器请求梯度值时，返回相邻片段之间的差异。参见图3.15。统一着色器核心具有访问相邻数据的能力——保存在同一warp的不同线程中——因此可以计算用于像素着色器的梯度。这种实现的一个结果是，在受动态流控制影响的着色器部分中无法访问梯度信息(动态流控制指的是具有可变迭代次数的“if”语句或循环）。一组中的所有片元必须使用相同的指令集进行处理，以便所有四个像素的结果对于计算梯度都有意义。这是一个基本限制，即使在离线渲染系统中也存在[64]。

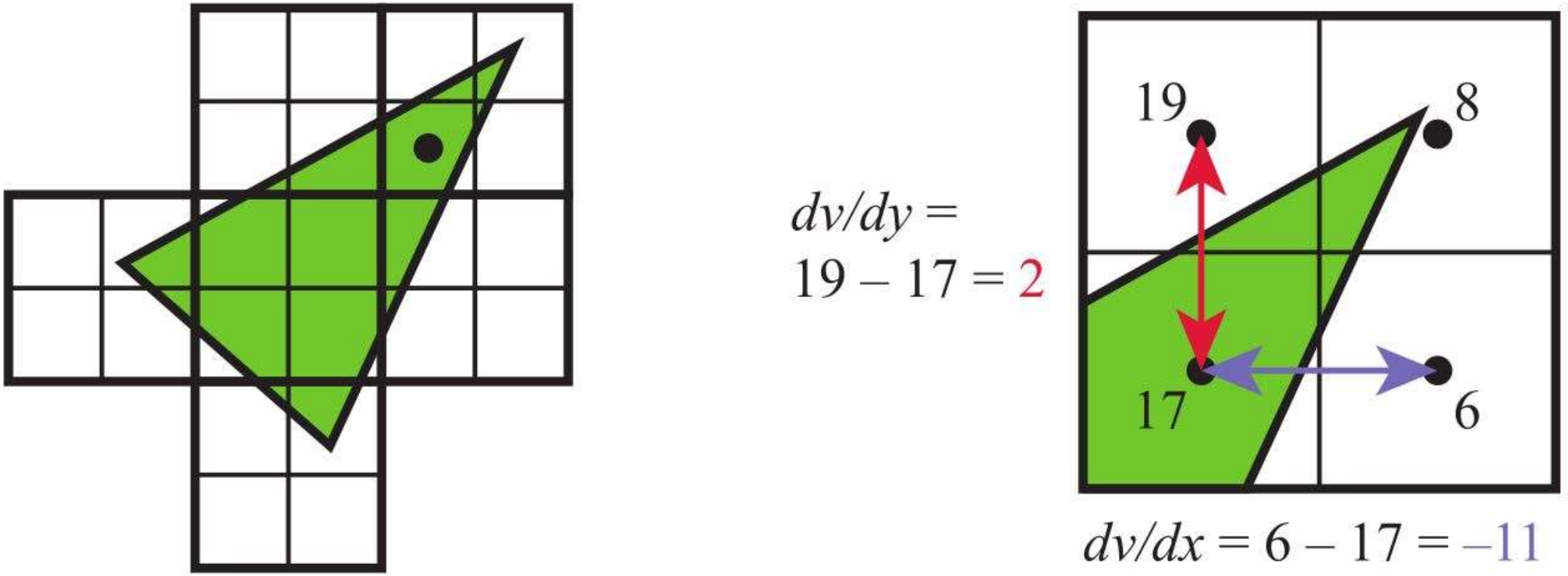

图3.15. 在左侧，一个三角形被光栅化为四边形，一组2×2像素。用黑点标记的像素的梯度计算显示在右侧。对于四边形中的四个像素位置中的每一个，都显示了v的值。注意三个像素是如何没有被三角形覆盖的，但它们仍然由GPU处理，以便可以找到梯度。x和y屏幕方向的梯度是通过使用其两个四边形邻居为左下像素计算的。

DirectX 11引入了一种允许对任何位置进行写访问的缓冲区类型，即无序访问视图(UAV)。最初仅用于像素和计算着色器，对UAV的访问扩展到DirectX 11.1 [146]中的所有着色器。OpenGL 4.3将此称为着色器存储缓冲区对象 (SSBO)。这两个名称都以自己的方式描述。像素着色器以任意顺序并行运行，并且该存储缓冲区在它们之间共享。

通常需要某种机制来避免数据竞争条件（又名数据风险），其中两个着色器程序都在“竞争”以影响相同的值，可能导致任意结果。例如，如果像素着色器的两次调用试图在大约同时添加到相同的检索值，则可能会发生错误。两者都会检索原始值，都会在本地修改它，但是无论哪个调用最后写入其结果都会消除另一个调用的贡献——只会发生一个添加。GPU通过具有着色器可以访问的专用原子单元来避免这个问题[530]。然而，原子操作意味着一些着色器可能会因为等待访问而停止，此时另一个着色器在读取/修改/写入相同的内存位置。

虽然原子可以避免数据风险，但许多算法需要特定的执行顺序。例如，你可能希望在用红色透明三角形覆盖之前绘制一个更远的透明蓝色三角形，将红色混合在蓝色之上。一个像素可能有两个像素着色器调用，每个三角形一个，以这样一种方式执行，即红色三角形的着色器在蓝色的着色器之前完成。在标准管线中，片元结果被处理之前，会在合并阶段进行排序。DirectX 11.3中引入了光栅化顺序视图(ROV)以强制执行顺序。这些就像UAV一样；它们可以由着色器以相同的方式读取和写入。关键区别在于ROV保证以正确的顺序访问数据。这大大增加了这些着色器可访问缓冲区的有用性[327,328]。例如，ROV使像素着色器可以编写自己的混合方法，因为它可以直接访问和写入ROV中的任何位置，因此不需要合并阶段[176]。代价是，如果检测到无序访问，像素着色器调用可能会停止，直到处理之前绘制的三角形。