第5章 着色基础 Shading Basics

5.3 实现着色模型 Implementing Shading Models

当然，这些着色和光照方程必须在代码中实现才能发挥作用。在本节中，我们将讨论设计和编写此类实现时的一些关键考虑因素，我们还将介绍一个简单的实现示例。

5.3.1 计算频率 Frequency of Evaluation

（译注：frequency of evaluation应是特殊名词，暂未查到正式翻译，联系上下文后，发现译为“计算频率”更易理解。）

当设计一个着色实现时，需要根据它们的计算频率对计算量进行划分。首先，确定给定计算的结果在整个绘制调用中是否始终是常量，在这种情况下，计算一般可由应用程序（通常是在CPU上）执行，即便GPU计算着色器可以用于特别昂贵的计算，计算结果通过统一着色器输入（uniform shader inputs）传递给图形API。

即使在这个类别中，可能也有一个广泛的计算频率范围，从“一次”开始（starting from "once ever."），最简单的一个例子是着色方程中可能存在的常量子表达式，这可应用于任意很少更改参数（如硬件配置和安装选项）的计算中，这样的着色计算可能在着色器编译的时候就完成了，这种情况下甚至不需要设置一个统一着色器输入，可以在安装阶段或应用加载时，用一个离线的预计算Pass执行计算。

另一种情况是，着色计算的结果在应用程序执行时不断变化，但是变化很慢，以至于不需要在每一帧都进行更新，例如虚拟游戏世界内基于时间的光照参数，如果计算花费昂贵，将其平摊到多个帧是很划算的。

其他情况有逐帧执行计算，例如连接视图矩阵和透视矩阵，或逐模型执行，例如更新依赖于位置的模型光照参数，或逐绘制调用执行，例如更新模型中各材质的参数，通过计算频率，我们将统一着色器输入分组，这样有助于提高应用程序的效率，且也可通过使更新常量最小来提升GPU性能[1165]。

如果着色计算的结果在一次绘制调用中改变，它就不能通过统一着色器输入传递给着色器，相反，它必须通过第3章中描述的一个可编程着色器阶段来计算，需要的话可通过不同的着色器输入传递到其他阶段，理论上，着色计算能在任意可编程阶段上执行，其中每个都对应着不同的计算频率：

* 顶点着色器（Vertex shader）——逐曲面细分前的顶点计算。
* 外壳着色器（Hull shader）**——**逐**表面补丁计算。**
* **域着色器（**Domain shader**）——**逐**曲面细分后的顶点计算。**
* **几何着色器（**Geometry shader**）——**逐**图元计算。**
* **像素着色器（**Pixel shader**）——**逐**像素计算。**

**图片包含 形状

描述已自动生成**

**图 5.9. 公式5.19所示着色模型的逐像素和逐顶点的计算结果的比较，显示了三个不同顶点密度的模型。左侧一列显示逐像素计算的结果，中间一列显示逐顶点计算的结果，右边呈现了每个模型的线框渲染以展示顶点的密度。（来自计算机图形学档案的中国龙网格模型[1172]，原模型来自斯坦福3D扫描存储库）**

**实际上，大部分着色计算是逐像素执行的，尽管这些通常是在像素着色器中实现的，但是计算着色器的实现正变得越来越普遍，一些相关例子会在第20章中讨论，其他阶段主要用于几何操作，例如变换和变形。为了理解为什么是这种情况，我们会对比逐顶点和逐像素着色计算的结果，在旧版的文本里，它们有时会被称作Gouraud着色（Gouraud shading）[578]和Phong着色（Phong Shading）[1414]，尽管这些术语如今已不常使用，对比中使用的着色模型在某些方面与公式5.1中的较为相似，但它经过修改，可以与多个光源一起使用，完整的模型将在稍后详细介绍示例实现时给出。**

**图5.9展示了不同顶点密度的模型的逐像素和逐顶点着色结果，对于龙这个顶点密度极高的网格模型，逐顶点与逐像素之间的区别很小，但在茶壶上，顶点着色计算会导致视觉错误，如棱角分明的高光，且在两个三角形平面上，顶点着色的版本显然是不正确的，造成这些错误的原因是着色方程的部分，尤其是高光部分，在网格表面有着非线性变化的值，这使得它们不适合用于顶点着色器，其计算结果会在递交给像素着色器之前在三角形上进行线性插值。**

**原则上来说，可以仅在像素着色器中计算着色模型的镜面高光部分（specular highlight），而在顶点着色器中计算其余部分，这可能不会导致视觉伪像（visual artifacts），并且理论上将节省一些计算，但在实践中，这种混合实现通常不是最佳的，着色模型的线性变化部分往往是计算成本最低的，且以这种方式拆分着色计算往往会增加相当多的开销，比如重复的计算和额外的可变输入，从而导致弊大于利。**

**正如我们之前提到的，在大部分实现中，顶点着色器负责非着色操作，例如几何变换和变形，生成的几何表面属性，转换到合适的坐标系统中，并被顶点着色器写入，在三角面上进行线性插值，然后作为可变着色器输入传入像素着色器，这些属性通常包括表面的位置，表面法线，以及可选的表面切线向量（如果需要法线贴图的话）。**

**注意，即使顶点着色器总是生成单位长度表面法线，插值也是能改变其长度的，见图5.10左侧。基于这个原因，法线需要在像素着色器中重新归一化（缩放至长度为1），但是顶点着色器生成的法线的长度仍然很重要，如果法线长度在顶点间明显不同的话，比如顶点混合的副作用会使插值倾斜，此情况可见图5.10右侧。由于这两个副作用，实现通常会在插值之前与之后，即在顶点着色器和像素着色器中，都进行插值向量的归一化。**

图示

描述已自动生成

**图 5.10. 在左侧，我们看到跨越表面的单位法线的线性插值将导致插值后的向量长度小于1。在右侧，我们看到线性插值的法线有着明显不同的长度，这导致了插值后的方向朝着两个法线中较长者的倾斜。**

**与表面法线不同，指向特殊位置的向量，例如视图向量（view vector）和精确光的光线向量（light vector），通常是不进行插值的，取而代之的是，在像素着色器中插值后的表面位置将被用来计算这些向量。除了在任何情况下都需要在像素着色器中执行的归一化操作外，每个向量都会用到向量减法运算，这是很快的。如果因为某些原因，需要对这些向量进行插值的话，不要事先对它们进行归一化，这会导致错误的结果，如图5.11所示。**

图示

描述已自动生成

**图 5.11. 两个光线向量间的插值。在左侧，在插值前将它们归一化将导致插值后方向不正确。在右侧，对未归一化向量插值，得到了正确的结果。**

**之前我们提到顶点着色器变换表面几何体到“合适的坐标系”，通过统一变量传递到像素着色器的相机与光源位置，通常被应用程序变换到相同的坐标系，这样可以最大程度减少像素着色器将所有的着色模型向量带入相同的坐标空间的工作，但究竟哪个坐标系是“合适”的呢？可能的答案包括全局世界空间以及相机的局部坐标系，或者更罕见的，当前渲染模型的局部坐标系。通常是基于系统性的考虑，例如性能，灵活性和简单性，为整个渲染系统做出选择，举个例子，如果渲染的场景预计包含大量的光源，那么可能选择世界空间以避免变换光源位置，但相机空间可能更好，这样可以更好地优化与视图向量相关的像素着色器操作，并且可能提高精确度。（第16.6节）**

**虽然大部分着色器实现，包括我们将要讨论的案例实现，都遵循上述一般概述，但是总有例外，比如，一些应用程序出于美术风格的原因选择了基于逐图元着色计算的多面外观，这种风格常被称为平面着色（flat shading），图5.12中展示了两个例子。**

电视萤幕画面

低可信度描述已自动生成

人在房间里

低可信度描述已自动生成

**图 5.12. 两款使用平面着色作为风格选择的游戏：肯塔基0号路（Kentucky Route Zero，上图）与癌症似龙（That Dragon, Cancer，下图）**

**原则上，可以在几何着色器中执行平面着色，但是近年来相关的实现通常使用顶点着色器，这是通过将每个图元的属性与其第一个顶点相关联并禁用顶点数值插值来完成的，禁用插值（可以对每个顶点值单独操作）将导致第一个顶点的值传递到图元中的所有像素。**

5.3.2 实现案例 Implementation Example

现在我们会展示一个着色模型实现的案例，正如之前所提，我们正在实现的着色模型与来自公式5.1的扩展的Gooch模型是相似的，但经过了修改以使其能够支持多个光源，它可以被描述为：

|  |  |
| --- | --- |
|  | (5.19) |

通过以下中间计算：

|  |  |
| --- | --- |
|  | (5.20) |

此公式适合公式5.6中的多光源结构，为方便起见，在此重复：

|  |  |
| --- | --- |
|  |  |

此例中，照亮项（lit）和非照亮项（unlit）为：

（译注：照亮表示被着色器中设置的光源照亮的部分）

|  |  |
| --- | --- |
|  | (5.21) |

通过调整冷色的非照亮项贡献值，可使结果看起来更像原始方程。

在大部分典型的渲染应用程序中，材质属性的变化值例如会被存储在顶点数据中，或者更普遍的，存储在纹理中（第六章）。然而为了让这个案例的实现保持简单，我们会假设在整个模型中是一个常数。

该实现方案会使用着色器的动态分支功能去循环处理所有的光源，尽管这种直接的方法可以很好地处理还算简单的场景，但是它并不合适庞大、具有复杂几何体，且拥有许多光源的场景，有效处理大量光源的渲染技术将会在第20章详细介绍。为了简单起见，我们只支持一种类型的光源：点光源，虽然这个实现相当简单，但是它遵循了我们之前提到的最佳实践。

（译注：“最佳实践”应该是指前文中“但在实践中，这种混合实现通常不是最佳的，着色模型的线性变化部分往往是计算成本最低的，且以这种方式拆分着色计算往往会增加相当多的开销，比如重复的计算和额外的可变输入，从而导致弊大于利。”）

着色模型并不是单独实现的，而是在一个更大的渲染框架上下文（context）中实现，该案例在一个简单的WebGL 2应用程序中实现，修改自Tarek Sherif [1623]的“Phong-shaded Cube”WebGL 2案例，但是相同的原则也适用于其他更复杂的框架。

我们将讨论一些GLSL着色器代码和JavaScript WebGL调用的应用程序的示例，目的并不是讲述WebGL API的细节，而是展示一般性的实现原理。我们将以“由内到外”的顺序来讲解实现过程，首先是像素着色器，然后是顶点着色器，最后是应用程序端的图形API调用。

着色器源文件应包含着色器输入与输出的定义，这样的着色器代码才是正确的，正如先前在第3.3节讨论的那样。使用GLSL术语，着色器输入会分为两类，其中之一是统一变量输入集，它们有着应用程序所设置的值，并且在一个绘制调用中保持不变，第二种类型由可变的输入（varying inputs）组成，它们的值可以在着色器调用（像素或顶点）之间改变。这里我们可以看到像素着色器的可变输入的定义，在GLSL中用in来标记，同时还有其输出：

in vec3 vPos ;

in vec3 vNormal ;

out vec4 outColor ;

像素着色器有单独的输出，表示最终的着色颜色，像素着色器的输入与顶点着色器的输出相匹配，其在被输入到像素着色器之前在三角形上进行插值。像素着色器有两个不同的输入：表面位置与表面法线，两者都在应用程序的世界空间坐标系内。统一变量输入的数量非常多，为了简洁，我们仅展示这两个的定义，且这两者都是与光源相关的：

struct Light {

vec4 position ;

vec4 color ;

};

uniform LightUBlock {

Light uLights [MAXLIGHTS];

};

uniform uint uLightCount ;

由于这些是点光源，因此每个光源的定义都包含位置和颜色，它们被定义为vec4而不是vec3，以符合GLSL std140数据布局标准的限制，尽管在这种情况下，std140布局可能导致一些空间的浪费，但是它简化了确保CPU与GPU之间的数据布局一致的任务，这也是我们为什么在该例中使用它的原因。Light结构体的数组是在一个已命名的统一代码块（uniform block）内定义的，这是GLSL的功能，用于将一组统一变量绑定到缓冲区对象，以加快数据传输，数组长度被定义为应用程序允许的在单个绘制调用中光源的最大数量，稍后我们将看到，在编译着色器之前，应用程序将着色器源码中的MAXLIGHTS字符串替换为正确的值(在本例中是10)。统一整数（uniform integer）uLightCount是在绘制调用中的实际的活动光源数。

接下来，我们来看一下像素着色器的代码：

vec3 lit( vec3 l, vec3 n, vec3 v) {

vec3 r\_l = reflect (-l, n);

float s = clamp (100.0 \* dot(r\_l , v) - 97.0 , 0.0 , 1.0) ;

vec3 highlightColor = vec3 (2 ,2 ,2);

return mix( uWarmColor , highlightColor , s);

}

void main () {

vec3 n = normalize ( vNormal );

vec3 v = normalize ( uEyePosition .xyz - vPos );

outColor = vec4 ( uFUnlit , 1.0) ;

for ( uint i = 0u; i < uLightCount ; i++) {

vec3 l = normalize ( uLights [i]. position .xyz - vPos );

float NdL = clamp (dot(n, l), 0.0 , 1.0) ;

outColor .rgb += NdL \* uLights [i]. color .rgb \* lit(l,n,v);

}

}

我们有一个照亮项的函数定义，它被main()函数调用，总的来说，这是公式5.20与公式5.21的GLSL直接实现，需要注意，和是作为统一变量被传入，因为这些值在整个绘制调用中是恒定不变的，因此应用程序可以计算这些值（译注：而不是着色器），从而节省一些GPU的周期。

该像素着色器使用了一些的内置的GLSL函数，reflect()函数反射一个向量（此例中为光线向量），反射位于第二个向量（此例中为表面法线）定义的平面上，由于我们想要光线向量和反射向量都指向远离表面的位置，我们需要在将前者传递给reflect()之前对其取反。clamp()函数有三个输入值，其中的两个输入值定义了第三个值被约束的范围，一个特别的例子是约束范围在0和1之间（这时就相当于HLSL的saturate()函数），该特例速度很快，在大部分GPU上没什么消耗，这也是我们使用它的原因，虽然我们只需要约束值到0，因为我们知道它不会超过1。mix()函数也有三个输入值，并基于第三个值（一个0到1之间的混合参数）在其中两个值（此例中，指的是暖色与高光色）之间进行线性插值，在HLSL中这个函数称为lerp()，意思是“线性插值”（linear interpolation）。最后是normalize()函数，它会将向量除以其长度，将其长度缩放为1。

现在让我们看一下顶点着色器，我们不会展示它的任何统一变量定义，因为我们已经在像素着色器里看过一些例子了，但是可变输入和输出的定义还是值得查看一下的：

layout ( location =0) in vec4 position ;

layout ( location =1) in vec4 normal ;

out vec3 vPos ;

out vec3 vNormal ;

注意如前所提，顶点着色器的输出对应像素着色器的可变输入，顶点着色器的输入中包含指定了数据在顶点数组中如何布局的指令，顶点着色器的代码如下：

void main () {

vec4 worldPosition = uModel \* position ;

vPos = worldPosition .xyz;

vNormal = ( uModel \* normal ).xyz;

gl\_Position = viewProj \* worldPosition ;

}

这些是顶点着色器的常规操作，着色器变换表面位置与法线到世界空间，并将它们传入像素着色器以便在着色过程中使用。最终，表面位置被变换到裁剪空间并且传入gl\_Position，一个被光栅器使用的特殊系统定义变量，gl Position变量是所有顶点着色器必需的输出。

注意，法线向量在顶点着色器中并没有归一化，它们不需要被归一化是因为它们在原始网格模型中的长度为1，并且应用程序没有执行任何可能会非均匀地改变它们长度的操作，例如顶点混合或非均匀缩放。模型矩阵可以有一个统一的缩放因子，但那会按比例地改变所有法线的长度，因此不会导致图5.10右侧中显示的问题。

应用程序为了各种不同的渲染和着色器设置而使用WebGL API，每个可编程着色器阶段都被单独地设置，并且它们都被绑定到程序对象上。以下是像素着色器的设置代码：

var fSource = document . getElementById (" fragment "). text . trim ();

var maxLights = 10;

fSource = fSource . replace (/ MAXLIGHTS /g, maxLights . toString ());

var fragmentShader = gl. createShader (gl. FRAGMENT\_SHADER );

gl. shaderSource ( fragmentShader , fSource );

gl. compileShader ( fragmentShader );

请注意提到的“片元着色器”，是WebGL（以及它所基于的OpenGL）中使用的术语，正如我们之前在本书中提到的，虽然“像素着色器”在某些方面描述不够精确，但它是更普遍的称呼方式，所以我们将继续在本书中使用这个称呼。这段代码也是将MAXLIGHTS字符串替换成合适数值的地方，大部分渲染框架执行类似的预编译着色器操作。

这里还有很多应用程序端代码用于设置统一变量，初始化顶点数组，清除，绘制等，你可以在程序[1623]中查看这些代码，并且许多API指南对此进行了说明。我们在此的目标是通过它们自身的编程环境，了解着色器是怎样被当作单独的处理器，因此我们在此结束本小节。

5.3.3 材质系统 Material Systems

正如我们的简单案例里，渲染框架很少只实现单个着色器，通常来说，需要一个专用的系统来处理大量的材质，着色模型，以及应用程序所使用的着色器。

正如之前章节所述，着色器是用于GPU的可编程着色阶段之一的一个程序，因此，着色器是低级的图形API资源，并且不是美术人员会直接接触的，与之相反，材质是面向美术人员封装的表面的视觉表现，材质有时也描述非视觉部分，比如碰撞属性，我们不会继续深入此话题，因为它已经超出了本书的范围。

虽然材质通过着色器实现，但不是简单的一对一对应关系，在不同的渲染情况下，相同的材质可能使用不同的着色器，一个着色器也可能被多种材质共用。最普遍的情况是材质参数化，在它的最简单形式中，材质参数化需要两种类型的材质实体：材质模板（material templates）与材质实例（material instances），每个材质模板描述一类材质，并且有一组参数，可以根据参数类型去分配数值，颜色，或者纹理的值，每个材质实例对应着一个材质模板与所有参数的一组特定值。一些渲染框架例如虚幻引擎[1802]允许更复杂的、分层的结构，其中材质模板派生自多层次的其他模板。

参数可以在运行时被解析，通过统一变量输入传递到着色器程序，或者也可以在编译时，通过在着色器编译前替换值来解析参数。一个常见的编译时参数类型是布尔开关量，用来控制激活给定材质的特征，这可由美术人员通过材质UI的复选框去设置，或者由材质系统在程序上设置，例如对于远处的物体，当它们的视觉效果特征可以忽略不计时，可减少着色器消耗。

尽管材质参数可以与着色模型参数一对一匹配，但我们不是总会遇到这种情况，一个材质可能会修改一个给定着色模型参数的值，例如表面颜色，可修改为一个常量，或者可以将多个材质参数以及插值的顶点或纹理值作为输入，通过一系列复杂的操作来计算着色模型参数。在某些情况下，表面位置，表面方向，甚至时间等参数也可能是计算的因素，基于表面位置和方向的着色在地形材质中尤为常见。举个例子，高度与表面法线可以被用来控制积雪特效，做法是在高处的水平面和接近水平面的表面以白色表面颜色做混合。基于时间的着色通常用于动画材质，例如闪烁的霓虹灯标志。

材质系统的最重要任务之一是将多种着色器函数划分为单独的元素，并控制这些元素的组合方式，在许多情况下，这种组合是很有用的，包括以下几种情况：

* 将表面着色与几何处理组合在一起，例如刚体变换，顶点混合，变形，曲面细分，实例化，以及裁剪，这些功能都是各不相同的：表面着色器依赖于材质，几何处理依赖于模型网格，所以，分开编写它们并让材质系统根据需求组合它们是很方便的。
* 将表面着色与一些组合操作例如像素丢弃（discard）与混合（blending）组合在一起，这与移动端GPU是尤为相关的，在移动端GPU的像素着色器里，混合是一种普遍执行的操作，通常我们希望独立于用于表面着色用的材质来选择这些操作。
* 将用来计算着色模型参数的操作与着色模型自身的计算组合在一起，这种方式允许编写一次着色模型的实现，并以与多种计算着色模型参数的方法组合一起的方式重用。
* 将独立可选的材质特征相互之间，以及与选择逻辑，剩余的着色器组合在一起，这种方式使得可以分别编写每个特征的实现。
* 将着色模型和用光源计算法（light source evaluation）其参数的计算组合在一起：在每个光源的着色点计算和的值，有些技术例如延迟渲（将在第20章讨论）改变了这个组合的结构，在支持该技术的渲染框架里，这种方式将增加额外的杂度。

如果图形API提供这种类型的着色器代码模块作为核心功能，将会很方便，可惜的是，不像CPU代码，GPU着色器不允许对代码片段进行后编译链接，每个着色器阶段的程序都作为一个单元被编译。着色器阶段之间的分离确实提供了一些有限的的模块化，这在某种程度上适合我们列表上的第一项：将表面着色（通常在像素着色器上执行）与几何处理（通常在其他着色阶段执行）组合在一起，但这个匹配并不完美，因为每个着色器也会执行其他操作，并且其他类型的组合仍然需要被处理。由于存在这些限制，材质系统能实现所有组合类型的唯一办法在源代码层级，主要包括如并列和替换的字符串操作，通常通过C风格的预处理指令例如#include，#if和#define去执行。

早期的渲染系统有着相对较小数量的着色器变体（shader variants，译注：这里提到的着色器变体很突兀，搜索了一下并结合上下文，其表示一个shader特定的变种。包含预编译宏等方式以控制不同分支的shader源代码，通过使用不同关键字编译，生成的不同shader就是变体），并且通常是手动去编写每个变体，这也有一些好处的，例如可以在充分了解最终的着色器程序的基础上去优化每个变体，然而这种手动编写的方法随着着色器变体数量的增加而变得不切实际。当我们将所有不同部分和选项都纳入考虑时，可能的不同着色器变体数量是巨大的，这就是为什么模块化和可组合性如此关键。

当设计一个系统用来处理着色器变体时，第一个需要解决的问题是，不同选项间的选择是否是通过动态分支（dynamic branching）在运行时执行，或者是在编译时通过条件预处理（conditional preprocessing）执行。在较老的硬件上，动态分支通常是不可能的或者是速度极慢的，所以运行时的选择是不可行的，所以所有变体都在编译阶段被处理，包括不同光源类型计数的所有可能的组合[1193]。

不过，如今的GPU能够很好地处理动态分支，尤其是在一个绘制调用中分支对所有像素有相同行为的情况下。现在许多功能性变体，例如光源的数量，都在运行时被处理，然而，为一个着色器添加大量的功能变体将产生一个不同的消耗：寄存器计数（register count）的增加和占用率的相应降低，进而导致性能下降，详情可见第18.4.5节。所以编译时变体仍然是有价值的，它能避免包含那些从不执行的复杂逻辑。

举个例子，让我们想象一个支持三种不同类型光源的应用程序，其中两种光源类型很简单：点光源与方向光，第三种类型是通用的聚光灯，它支持表格式照明模式（tabulated illumination patterns）以及其他复杂的功能，这需要大量的着色器代码去实现，但是这个通用的聚光灯使用率相对较小，此应用程序中只有不到5%的光源是这种类型。在过去，一个单独的着色器变体会为每个可能的三种光源的计数组合去编译，以避免动态分支，尽管如今已不再需要这种方式，但是编译两个单独的变体仍然是有好处的，一个变体适用于通用聚光灯数量大于等于1时，另一个变体适用于此类聚光灯数量正好为0时。由于第二个变体的代码更简单，它（更常被使用）可能有着更低的寄存器占用率，并且因此有着更高的性能表现。

现代材质系统同时使用了运行时着色器变体和编译时着色器变体，即使完整的负载已经不会仅在编译时处理，但总体的复杂度与变体的数量仍然保持增长，所以还是需要编译大量的着色器变体。举个例子，在游戏《命运：被夺走的国王》（Destiny: The Taken King）的某些场景中，单帧内使用了超过9000个编译期着色器变体[1750]，可能的变体数量还可以变得更为巨大，例如Unity渲染系统有着接近1000亿可能的变体，只有确确实实被使用的变体才会被编译，但是着色器编译系统必须被重新设计以处理大量可能的变体[1439]。

材质系统设计者使用不同的策略去解决这些设计目标，虽然这些策略有时候表现为互斥的系统体系结构[342]，但这些战略能够被——也通常是——结合到相同的系统中，这些策略包含以下内容：

* 代码重用——在共享的文件中实现函数，使用#include预处理指令去访问那些任意着色器的所需函数。
* 减法——使用一个通常被称为超级着色器（üershader或supershader）[1170,1784]的着色器，它聚集了一大批功能，使用编译时预处理器与动态分支的组合去移除无用的部分并在互斥的备选方案之间切换。
* 加法——将各种功能定义为具有输入输出连接器的节点，并将它们组合到一起，这与代码重用策略相似，但是更结构化。节点的组合可以通过文本[342]或一个可视图形编辑器来完成，后者旨在使非工程师（例如技术美术）更容易编写新的材质模板[1750，1802]，通常来说，可视化图形编写只能访问到着色器的一部分，例如在虚幻引擎的可视化图形编辑器中，只能作用到着色模型输入相关的计算，如图5.13所示。

电脑主机

中度可信度描述已自动生成

**图 5.13. 虚幻引擎材质编辑器，注意在节点图右侧的高的节点，这个节点的输入连接器对应着渲染引擎使用的多种着色输入，包括所有的着色模型参数。（材质示例由Epic Games提供）**

* 基于模板——定义了一个接口，不同的实现只要符合该接口，就可以接入该接口，这比加法策略更加正式，因此通常被用于更大的功能块中。基于接口的一个普遍案例是着色模型参数计算与着色模型自身计算的分离，虚幻引擎[1802]有着不同的“材质域”，包括计算着色模型参数的表面域，以及计算给定光源用来调整的标量的光照函数域，一个与此相似的“表面着色器”结构也存在于Unity中[1437]。注意延迟着色技术（将在第20章讨论）强制采用了一个类似的结构，采用G缓冲区用作接口。

更具体的案例，书籍《WebGL Insights》（现在免费）中的一些章节讨论了各种引擎是怎样控制它们的渲染管线。除了组合外，现代材质系统还有一些重要的设计注意事项，例如如何以最少的着色器代码重复去支持多平台，这包括函数性的变体，以解决平台，着色语言和API之间的性能和功能差异。游戏《命运》的着色器系统（The Destiny shader system）[1750]是对这种问题的最具代表性的解决方案，它采用了一个专有的预处理器层从而使着色器能够由一个自定义的着色器语言书写，这允许我们编写与平台无关的材质，然后自动翻译成不同的着色语言与具体实现，虚幻引擎[1802]与Unity[1436]有着相似的系统。

材质系统也需确保具有好的性能表现，除了着色变体的专门编译外，材质系统还能执行一些其他常见的优化。《命运》着色器系统以及虚幻引擎会自动检测那些在一次绘制调用中保持恒定的计算（例如在之前实现案例中提到的暖色与冷色计算），并且将它们移到着色器之外。另一个例子是在《命运》中使用的作用域系统（scoping system），它用来区分以不同频率更新的常数（例如每帧一次，每个光源一次，每个物体一次），并在合适的时间更新每组常数以减少API的开销。

正如我们所见过的，实现一个着色方程的关键是决定哪个部分可被简化，各种表达式计算的频率，以及用户能够如何修改和控制其表现，渲染管线的最终输出是颜色和混合值。剩余的部分是反走样，透明度，与图像显示的部分详细介绍了这些值将怎样合并与修改以用于显示。

5.4 走样与反走样 Aliasing and Antialiasing

**（译注：aliasing可译为走样，锯齿，混叠，失真）**

**想象一个巨大的黑色三角形缓缓地穿过一个白色背景，当屏幕网格单元格（screen grid cell）被三角形覆盖，代表这个单元格的像素值强度应该平滑地下降，然而通常发生在所有类型的基础渲染器中的情况是，当网格单元的中心被覆盖的那一刻，像素颜色立即从白色变为黑色，标准GPU的渲染也不例外，见图5.14的最左列。**

**三角形在像素里的显示是要么存在，要么不存在，线的绘制也有类似的问题，边缘有着锯齿状的外观，因此这种视觉伪像被称作“锯齿”（the jaggies），当物体运动时则被称作“爬虫”（crawlies），此问题更正式的称呼为“走样”（aliasing），同时我们称旨在避免这个问题的相关技术为“反走样”（antialiasing）。**

**关于采样理论与数字滤波的话题已经足够另外写一本书了[559，1447,1729]，因为这是渲染的关键领域，我们会在这里阐述采样和滤波的基础理论，接下来我们会专注于当前在实时渲染中我们能做一些什么以减轻走样伪像（aliasing artifacts）。**

图片包含 游戏机, 画, 钟表

描述已自动生成

**图 5.14. 上排图像显示了三个不同反走样级别的三角形、线、点，下排图像是上排图像放大后的样子。最左列每个像素仅用一个采样，这意味着没有使用反走样技术，中间列的图像以每像素四个采样（以网格模式）的方式渲染，最右列则是使用每像素八个采样（在4×4 的棋盘格中，一半的正方形被采样）。**

5.4.1 采样与滤波理论 Sampling and Filtering Theory

**渲染图像的处理本身便是一个采样任务，之所以这样是因为图像的生成就是三维场景采样的处理过程，其目的是获取图像中每个像素（一个离散的像素数组）的颜色值。为了使用纹理映射（第6章），纹素（texels）必须能被重采样以在各种条件下获得好的结果，为了在动画文件中生成一系列图像，动画文件通常以均匀的时间间隔采样。本节将介绍采样，重建，以及滤波的话题，为了简单起见，大部分材质将以一维呈现，这些概念也可自然地扩展到二维，因此能够在处理二维图像时使用。**

**（注：纹素（英语：Texel，即textureelement或texture pixel的合成字）是纹理元素的简称，它是计算机图形纹理空间中的基本单元，如同图像是由像素排列而成，纹理是由纹素排列表示的。）**

**图5.15展示了一个连续的信号是怎样以均匀的间隔被采样的，这称之为离散化（discretized），采样处理的目标是数字化地呈现信息，这样做是为了减少信息量。然后采样的信号需要被重建（reconstructed）以恢复原始信号，这是通过对采样信号进行滤波（filtering）来完成的。**

图表, 条形图, 直方图

描述已自动生成

**图 5.15. 一个连续信号（左图）被采样（中图），接下来原始信号通过重建被恢复（右图）。**

**无论何时进行采样，都可能出现走样，这是我们不想造成的伪像，我们需要与走样进行战斗以生成令人满意的图像。西方老年人见过的一个关于走样的经典案例，是电影摄像机拍摄的一个旋转的马车车轮，由于车轮辐条移动得比摄像机记录图像的速度快得多，车轮看起来像是在向后或向前缓慢旋转，或者甚至有可能看起来根本没有转动，如图5.16所示，之所以出现这种现象，是因为车轮的图像是以一系列时间步长被记录的，这被称作时间走样（temporal aliasing）。**

形状

中度可信度描述已自动生成

**图 5.16. 最上面一行显示一个纺车的车轮(原始信号)。在第二行采样不足，使它似乎在相反的方向移动，这是一个由于采样率太低导致走样的例子。在第三行中，采样率正好是每转两个样本，我们不能确定轮子旋转的方向，这就是奈奎斯特极限（Nyquist limit）。在第四行中，采样率高于每转两个样本，我们突然可以看到轮子在正确的方向上旋转。**

**在计算机图形中走样的普遍案例有光栅化后的线与三角形边缘的“锯齿”，被称为“萤火虫”（fireflies）的闪烁的高光，以及缩小具有方格模式的纹理时发生的走样（见6.2.2节）。**

**当一个信号被以过慢的频率采样时，就会出现走样，采样后的信号看起来是一个频率比原始信号低的信号，如图5.17所示。为了使一个信号被合适地采样（换句话说，这样就能够从样本中重建原始信号），采样频率必须大于被采样信号最大频率的两倍，这通常被称作采样定理（sampling theorem），并且该采样频率被称为奈奎斯特率（Nyquist rate）[1447]或奈奎斯特极限（Nyquist limit），以一位在1928年发现此频率的瑞典科学家哈里·奈奎斯特（Harry Nyquist）(1889–1976)命名，奈奎斯特极限也在图5.16进行了描述。该定理使用术语“最大频率”的这一事实暗示着信号应该受到频带限制（band-limited），而这意味着任何频率都不能超过特定限制，换句话说，信号相对于相邻样本间的间隔应该足够平滑。**

图表, 折线图

描述已自动生成

**图 5.17. 蓝色实线为原始信号，红色圆圈表示间距均匀的采样点，绿色虚线为重构信号。上图显示了过低的样本率，导致重构信号看上去频率更低，即原始信号的混叠信号。下图显示了采样率正好是原始信号频率的两倍，重构出来的信号是一条水平线。可以证明，如果采样率稍微提高一点，完美的重建是可能的。**

**当一个三维场景以点样本渲染时，正常情况下是不会有频带限制的，但三角形的边缘，阴影边界，以及其他现象会产生不连续变化的信号，因此会产生无限的频率[252]，同时无论样本被打包得多紧密，物体仍然能足够小，以至于它们根本不能被采样，因此当我们使用点样本渲染场景时，完全避免走样的问题是不可能的，而我们几乎总是使用点采样。不过有时候我们可以知道信号在何时有频带限制，其中一个例子是当纹理被应用到表面的时候，这时计算纹理采样频率与像素采样率之比是可能的，如果该频率低于奈奎斯特极限，那就不用做特殊的操作以对纹理进行合适的采样，如果频率过高的话，就要使用各种算法对纹理进行频带限制（第6.2.2节）。**

**重建 Reconstruction**

**我们现在来讨论给定一个频带受限的采样信号，原始信号如何从采样信号中重建，为做到这点，我们必须用到一个滤波器，三个普遍使用的滤波器如图5.18所示。需要注意的是，滤波器的面积应该总是为1，否则重建的信号可能会扩大或收缩。**

图表, 折线图

描述已自动生成

**图 5.18. 左上为box滤波器（box filter），右上为tent滤波器（tent filter），底部为sinc滤波器（sinc filter）（这里已经clamped在x轴上）**

电脑屏幕的照片

中度可信度描述已自动生成

**图 5.19. 采样后的信号（左侧）使用box滤波器进行重建，这是通过以下步骤完成的：首先在每个采样点上放置box滤波器，并且在y方向上将其缩放，使滤波器的高度与采样点相同，之后求出的和就是重建后的信号（右侧）。**

**在图5.19中，相邻的box滤波器被用于重建一个采样信号，这是所使用的最糟糕的滤波器，因为产生的结果信号是一非连续的阶梯状，然而由于它很简单，所以仍然经常在计算机图形学中使用。如图所示，box滤波器被放置在每个采样点上，之后会被缩放，这样滤波器最上方的点就可以与样本上的点重合，所有这些缩放与平移后的box函数之和就是右侧所示的重建后的信号。**

**box滤波器可以替换成任意其他滤波器，在图5.20中，tent滤波器，也被称作三角形滤波器，被用来重建采样后的信号，注意这个滤波器在相邻采样点之间实施了线性插值，所以它比box滤波器要更好，因为重建的信号现在是连续的。**

卡通人物

中度可信度描述已自动生成

**图 5.20. 采样后的信号（左侧）使用tent滤波器进行重建，重建后的信号如右侧所示。**

**然而使用tent滤波器重建的信号的平滑程度并不好，在采样点有突然的斜率改变，这与tent滤波器并非是完美的重建滤波器这一事实有关。为了得到完美的重建，必须使用理想的低通滤波器，其中信号的频率分量是正弦波：，f是该分量的频率，考虑到这一情况，低通滤波器将去除频率高于滤波器定义的某个频率的所有频率分量，直观的，低通滤波器移除了信号的尖锐特征，即滤波器对信号做了模糊处理。理想的低通滤波器是sinc滤波器（见图5.18底部）：**

|  |  |
| --- | --- |
|  | (5.22) |

**傅里叶分析[1447]的理论解释了为什么sinc滤波器是理想的低通滤波器，简单来说，理由如下：理想的低通滤波器是频率域的box滤波器，当它与信号相乘时，移除了所有高于滤波器宽度的频率，将box滤波器从频域转到空间域就得到了sinc函数，与此同时，乘法操作被转换为了卷积（convolution）函数，卷积是我们在本节中一直使用，但却没有实际描述过的术语。**

图表, 图示, 雷达图

描述已自动生成

**图 5.21. 此处，sinc滤波器被用来重建信号，sinc滤波器是理想的低通滤波器。**

**使用sinc滤波器重建信号能得到更平滑的结果，如图5.21所示。采样过程在信号中引入了高频部分（突变），而低频滤波器的任务就是移除这些高频部分，事实上，sinc滤波器消除了所有频率大于1/2采样率的正弦波，sinc函数如公式5.22所示，当采样频率是1.0时（即采样信号的最大频率必须小于1/2），它是完美的重建滤波器。更一般地，假设采样频率是****，也就是说，相邻样本的间隔为，对这种情况来说，****是完美的重建滤波器，它消除了所有高于的频率，这在重采样中是很有用的（下一节），然而sinc滤波器的宽度是无限的，并且在某些区域是负值，所以它在实践中很少使用。**

**一方面在低品质的box与tent滤波器之间存在有用的中间区域，另一方面是不切实际的sinc滤波器，大部分广泛使用的滤波器函数[1214,1289,1413,1793]处于这些极端情况之间，所有这些滤波器函数都有一点类似sinc函数，但是对它们影响的像素数量有所限制。最接近sinc函数的滤波器在它们的部分域上有负值，对于应用程序而言，负滤波器值是不需要也不实用的，我们通常使用有着非负瓣（no negative lobes）的滤波器[1402]（通常被称作高斯滤波器，因为它们源于或类似于高斯曲线），第12.1节更详细地讨论了滤波器函数以及它们的使用。**

**在使用任意滤波器之后，便得到了一个连续的信号，然而在计算机图形学中我们不能直接显示一个连续的信号，但是我们可以重采样连续信号使其变为另一个大小，即放大或缩小信号，这个话题将在接下来讨论。**

**重采样 Resampling**

**重采样被用来放大或缩小一个被采样信号，假设原采样点位于整数坐标系内（0, 1, 2, ...），即样本间的间隔是单位整数，更进一步，假设在重采样后，我们想要新的采样点以样本的间隔a均匀地放置，如果a > 1，产生缩小（下采样），对于a < 1，产生放大（上采样）。**

**放大是两种情况中较为简单的一个，让我们从放大开始。假设采样信号如上一节所示那样被重建，直观地，因为信号现在已经被完美重建并且是连续的，所需要的便是以我们期望的间隔去重采样重建后的信号，该过程如图5.22所示。**

图表, 图示

描述已自动生成

**图 5.22. 左边是采样信号和重建信号，****右边是重建信号以两倍的采样频率进行重采样，即发生了放大。（译注：频率放大，间隔缩小）**

**然而当缩小时，这个技术不起作用，原始信号的频率对采样率来说过高，以至于无法避免走样，不过已经证明了使用的滤波器可被用来从采样信号中创建连续信号[1447,1661]，之后便可以用期望的间隔进行重采样，如图5.23所示。换句话说，通过使用作为滤波器，低通滤波器的宽度增加了，以至于更多的信号高频率内容被移除了，正如图中所示，滤波器（sinc函数专属的）宽度被翻倍以减少重采样率到原采样率的一半，将此与数字图像联系起来，这类似于先模糊它(去除高频)，然后以较低的分辨率对图像进行重采样。**

图表, 图示

描述已自动生成

**图 5.23. 左边是采样信号和重建信号，右边滤波器宽度放大为原来的两倍，以使样本之间的间隔变为原来的两倍，即发生了缩小。**

**以采样和滤波理论为框架，现在讨论在实时渲染中用于减少走样的各种算法。**

5.4.2 基于屏幕空间的反走样 Screen-Based Antialiasing

如果采样与滤波的效果不好，三角形的边缘会产生明显的伪像，阴影边缘，高光，以及颜色迅速变化的其他现象都可能导致类似的问题，在本小节中讨论的算法会帮助前述的这些案例提升渲染品质，它们的共同点是，它们都是基于屏幕空间的，即它们只在管线输出的采样样本上进行操作。没有最好的反采样技术，每一种技术都有不同的质量优势，如捕捉敏锐的细节或者其他现象的能力，运动时的表现，内存消耗，GPU硬件要求，以及速度等等。

在如图5.14所示的黑色三角形案例中，存在一个问题是采样率较低，在每个像素的网格单元中心进行单个采样，因此对于该网格单元，最多被了解的是它是否被三角形所覆盖（译注：0 or all）。通过在每个屏幕网格单元使用更多的采样并以一些方式将它们混合，就能计算出更好的像素颜色，如图5.24中所示。

形状

描述已自动生成

**图 5.24. 左侧，以像素中心采样一次的方式渲染一个红色三角形，因为三角形并没有覆盖样本，像素是白色，即使像素的大部分已经被红色三角形覆盖。右侧，对每个像素采样四次，可以看到其中两个采样点被红色三角形所覆盖，因此像素为粉红色。**

基于屏幕的反走样方案的一般策略是使用一个针对屏幕的采样模式，并对这些样本进行加权与求和，得到像素的颜色，p：

|  |  |
| --- | --- |
|  | (5.23) |

式中是用于单个像素的采样数，函数是一个采样颜色，是权重，范围是[0,1]，样本对整个像素的颜色有所贡献，样本的位置根据其在序列中的顺序来确定，如1, …, n，并且可选函数也是使用像素位置(x,y)的整数部分，换句话说，每个样本在屏幕网格的采样位置都是不同的，并且每个像素可以选用不同的采样模式。在实时渲染系统（以及大多数其他渲染系统）中，样本通常是点样本，所以，函数c可以被认为是两个函数，首先函数f(i,n)检索所需样本在屏幕上的位置浮点数()，然后对屏幕上的该位置进行采样，即检索该精确点处的颜色，选择采样方案，并且配置渲染管线以计算特定子像素位置的采样，这通常是基于逐帧（或逐应用）设置的。

在反走样中的另一个变量是，每个样本的权重，这些权重的和为1，大部分用于实时渲染系统的方法都对它们的样本给出了统一权重，即。图形硬件的默认模式，像素中心的单个采样，是上述反走样方程的最简单情况，只有一个项，该项的权重为1，并且采样函数f总是返回被采样像素的中心。

每个像素计算超过一个完整采样以上的反走样算法，被称作超级采样（supersampling）方法（或过采样方法（oversampling)））。简单地从概念上说，全场景反走样（full-scene antialiasing, FSAA），又名“超级采样反走样”（supersampling antialiasing, SSAA），以更高的分辨率渲染场景，然后对相邻的样本进行滤波以得到图像。举个例子，假设我们需要一张1280×1024像素的图像，如果你在屏幕外渲染一个2560×2048像素的图像，然后对每2×2的像素区域取平均值，我们需要的图像就会以每像素采样四次的方式生成，并使用box滤波器进行滤波，需要注意的是，这相当于图5.25中的2×2网格采样，此方法比较消耗性能，因为所有的子采样必须被完整地着色与填充，其中每个样本都具有z缓冲区的深度信息。FSAA的主要优点在于简单，此外，这种方法的低质量版本只在一个屏幕轴向上以两倍的速率采样，因此被称为1×2或2×1超级采样，通常为了简便，使用二次幂分辨率和box滤波器。英伟达（NVIDIA）的动态超分辨率（dynamic super resolution）功能是一个更加复杂的超级采样形式，其中以更高的分辨率渲染场景，并且使用13个采样的高斯滤波器（13-sample Gaussian filter）去生成显示的图像[1848]。

图片包含 信件

描述已自动生成

**图 5.25. 一些像素采样方案的对比，按照单个像素采样数从少到多排列，Quincunx共享角落的样本以及对中心样本进行加权，以使其值达到像素最终颜色的一半。2×2旋转网格比2×2直形网格在几乎水平的边缘上会捕获更多的灰度级。类似地，8rooks模式为此类线条捕获的灰度级别比4×4网格更多，尽管其使用的样本更少。**

有一个与超级采样相关的采样方法基于累积缓冲区（accumulation buffer）[637,1115]，该方法不使用一个大的离屏缓冲区，而是使用一个与最终期望图像具有相同分辨率的缓冲区，但是每个颜色槽（channel）使用更多的字节位，为得到一个场景的2×2采样，生成了四幅图像，视图根据需要在屏幕x轴或y轴上移动半个像素，每个生成的图像都是基于网格单元内的不同采样位置，每帧必须重新渲染场景几次，并将结果复制到屏幕上，这种额外花费使该算法在实时渲染系统中成本很高，当性能不是关键时，这种方法对生成高质量的图像来说是很有用的，因为每个像素可以使用任何数量的样本，并且可以放置在任何地方[1679]。它曾直接被OpenGL API支持，但是在3.0版本中被弃用，在现代GPU中，累积缓冲区这个概念可通过在输出缓冲区使用高精度的颜色格式，从而在像素着色器中实现。

当物体边缘、镜面高光和锐利阴影等现象引起突变的颜色变化时，需要额外的采样样本，这样阴影通常能够变得更软，高光也可以变得更平滑以避免走样，可以增加特定类型对象的大小，例如电线，以保证它们在长度上每个位置至少覆盖一个像素[1384]。物体边缘的走样仍然是一个主要的采样问题，在渲染中当物体边缘被检测以及它们的影响被考虑在内时，可以使用分析方法（analytical methods），但是这些方法通常更为昂贵，并且与简单地进行更多的采样相比，健壮性更低。不过，GPU的功能例如保守光栅化和光栅化顺序视图开启了新的可能性[327]。

像超级采样与累积缓冲区等技术，它们通过生成完全由单独计算的阴影和深度指定的样本来工作，由于每个样本都必须通过像素着色器，因此总体收益相对较低，性能消耗也较高。

多重采样反走样（Multisampling antialiasing, MSAA）通过每像素一次的表面着色计算，并在样本间共享计算结果，从而降低了高额的计算成本。Pixels may have, say,（译注：不会译）每个片元有四个(x,y)的样本位置，每个都有它们自己的颜色与z深度值，但是对于应用到像素的每个对象的片元，像素着色器只进行一次计算。如果所有的MSAA位置样本都被片元覆盖，那么着色样本就会在像素的中心被计算，相反如果片元覆盖较少的位置样本，则着色样本的位置可以移动，以更好地表示所覆盖的位置，这么做可以避免例如纹理边缘的着色采样，这种位置调整方法被称作质心采样（centroid sampling）或质心插值（centroid interpolation），如果开启此功能，GPU将自动完成。质心采样能避免出现三角形外的问题（off-triangle problems），但会导致导数计算返回不正确的值[530, 1041]，见图5.26。

形状

描述已自动生成

**图 5.26. 中间图，一个像素中有两个物体重叠，红色物体覆盖了三个样本，蓝色物体只有一个，像素着色器计算该位置以绿色显示。因为红色三角形覆盖了像素的中心，这个位置被用于着色器计算，用于蓝色物体的像素着色器在此样本位置进行计算。对于MSAA来说，所有四个位置中存储了单独的颜色与深度值，在右侧展示了EQAA的2f4x模式（the 2f4x mode for EQAA），四个样本现在有四个ID值，这些ID索引了一张存储起来的表，表内有两种颜色和深度的信息。**

MSAA比纯粹的超级采样方案快，因为片元只进行一次着色，它致力于以高频率对片元的像素覆盖区域进行采样，以及共享计算出的着色数据。通过进一步分离采样和覆盖范围（coverage），可以节省更多的内存，这反过来又可以使反走样的速度更快——使用的内存越少，渲染速度就越快。英伟达（NVIDIA）在2006年推出了覆盖采样反走样（coverage sampling antialiasing, CSAA），AMD随后推出了增强质量反走样（enhanced quality antialiasing, EQAA），这些技术的工作原理是，以更高的采样率存储仅被覆盖的片段。例如，EQAA的“2f4x”模式存储了两个颜色与深度值，在四个样本位置之间共享，颜色与深度值信息不再储存在特定的位置里，而是储存在一张表中，四个样本每个只需要一位（bit）空间用来指定两个存储值中的哪个与其位置相关联，见图5.26。覆盖样本明确规定了每个片元对最终像素颜色的贡献，如果储存的颜色数量超出限制，一个储存的颜色就会被移除并且对应的样本会被标记为未知，这些样本对最终颜色不产生贡献[382,383]，对大多数场景来说，相对较少的像素会包含三个或更多的在着色上完全不同的可见不透明片元，所以这个方案在实践中表现良好[1405]。然而对于最高品质来说，游戏极限竞速：地平线2（Forza Horizon 2）运行时会使用4倍MSAA，即便EQAA有更好的性能优势[1002]。

一旦所有的几何体被渲染到一个多重采样缓冲区，将会执行一个解析（resolve）操作，这段程序会将总体样本颜色进行平均以决定像素的颜色，值得注意的是，当使用具有高动态范围的颜色值进行多重采样时，可能会出现一个问题，在这种情况下，为了避免伪像，在进行解析操作前，你通常需要对颜色值进行色调映射[1375]，这个开销可能很昂贵，所以可以使用更简单的色调映射函数的近似函数或者其他方法[862,1405]。

默认的情况下，MSAA通过box滤波器进行解析，2007年，ATI推出了自定义滤波器反走样（custom filter antialiasing, CFAA）[1625]，它能够使用更狭窄或更宽的tent滤波器并且稍微拓展到其他像素格，这种模式已经被EQAA支持所取代。在现代GPU上，像素或者计算着色器能够访问MSAA的样本并且使用任何我们所期望的重建滤波器，包括从像素周围样本中得到的样本，一个更宽的滤波器能减少走样，虽然丢失锐利的细节，Pettineo[1402, 1405]发现立方体的smoothstep以及有着2或3像素宽度的B样条滤波器总体上得出了最好的结果。（译注：Smoothstep is a scalar interpolation function commonly used in computer graphics Smoothstep at Microsoft Developer NetworkGLSL Language Specification, Version 1.40 and video game engines. 维基百科）当然还有性能消耗，因为即使使用自定义着色器模拟默认的box滤波器解析也会花费很长的时间，而一个更宽的滤波器核心意味着增加了样本的访问成本。

英伟达（NVIDIA）内建的TXAA支持与之类似，在比单个像素更大的区域上使用了更好的重建滤波器，以获得更好的结果，它和更新的多帧反走样（multiframe antialiasing, MFAA）方案都使用了时间性反走样（temporal antialiasing, TAA），这是一类通用技术，它可以使用之前帧的结果用来改进图像，在某种程度上，这种技术之所以成为可能，是因为程序员可以设置每帧MSAA采样模式[1406]，这种技术可以解决例如旋转的马车车轮等反走样问题，并且能够改进边缘渲染质量。

想象通过生成一系列图像来“手动”执行采样模式，其中每次渲染使用不同的位置进行采样，这种偏移是通过在投影矩阵上附加一个微小的平移来完成的[1938]，生成和取平均的图像越多，结果就越好，这种使用多个偏移图像的概念被用于时间性反走样算法。可能使用MSAA或其他方法生成单个图像，然后与之前的图像做混合，通常只有2~4帧被使用[382, 836, 1405]，较旧的图像被赋予的权重可能呈指数减小[862]，尽管如果观看者和场景不移动，这可能会导致帧闪烁（shimmering），所以通常情况下，最后一帧和当前帧的权重相等。由于每帧的样本位于不同的子像素位置，这些样本的加权和比单个帧能更好地估计边缘的覆盖范围，因此，使用最新的两帧平均的系统可以给出更好的结果，每一帧不需要额外的样本，这是这种方法如此吸引人的原因，甚至可以使用时间性采样来生成放大到显示器分辨率的低分辨率图像。此外，需要很多样本才能得到好结果的照明方法或其他技术，可以改为每帧使用更少的样本，因为结果将在数帧中混合。

在不增加额外采样成本的情况下为静态场景提供反走样功能时，这种类型的算法在使用时间性反走样时会遇到一些问题，如果没有对帧进行均等的加权，则静态场景中的对象可能会出现闪烁，快速移动物体或快速的摄像机移动会导致重影（ghosting），即由于先前帧的影响而在物体后方留下轨迹。重影的一个解决方案是只在缓慢移动的物体上执行这种反走样[1110]，另一个重要的方法是使用重投影（reprojection）（第12.2节）来更好地关联先前和当前帧的对象，在这种方案中，对象生成运动向量，并将其存储在单独的“速度缓冲区”（velocity buffer）中（第12.5节），这些向量用于将前一帧与当前帧相关联，即从当前像素位置减去该向量，以找到该对象表面位置前一帧的颜色像素，在当前帧中不可能成为表面一部分的样本被丢弃[1912]。由于时间性反走样不需要额外的样本，所以额外的工作相对较少，因此近年来人们对这种算法产生了浓厚的兴趣，并且该算法也得到了广泛的应用，其中一些关注是因为延迟着色技术（第20.1节）与MSAA等其他多采样支持不兼容[1486]。时间性反走样的实现方法各不相同，并且根据应用程序的内容和目标，已经开发了避免伪像和提高质量的一系列技术[836, 1154, 1405, 1533, 1938]。以Wihlidal的演讲[1885]为例，它展示了如何将EQAA，时间性反走样和应用于棋盘采样模式（checkerboard sampling pattern）的各种滤波技术结合起来，以保持画面质量，同时减少像素着色器的调用次数。Iglesias-Guitian等[796]总结了以前的工作，并提出了他们的方案，使用像素的历史信息并预测，从而最小化滤波伪像（filtering artifacts），Patney等人[1357]扩展了Karis和Lottes在虚幻4引擎的实现中[862]用于虚拟现实应用程序的TAA工作，增加了可变大小的采样以及对眼睛运动的补偿（第21.3.2节）。

**采样模式 Sampling Patterns**

有效的采样模式是减少走样、时间及其他消耗的关键要素。Naiman[1257]表明，人类最容易受到水平和垂直边缘附近走样的干扰，斜面接近45度的边缘是第二大的干扰。旋转网格超级采样（Rotated grid supersampling, RGSS）使用旋转的正方形模式来在像素内提供更多垂直和水平分辨率，图5.25显示了此模式的一个例子。

RGSS模式是一种拉丁超立方体（Latin hypercube）或N-rooks采样的形式，其中n个样本放置在n×n的网格中，每行和每列一个样本[1626]，使用RGSS时，这四个样本分别位于4×4的子像素网格中的单独行和列中，与常规2×2的采样模式相比，这种模式特别适合捕获接近水平和垂直的边缘，后者的边缘可能覆盖偶数个样本，因此有效程度较低。

N-rooks是创建良好采样模式的开始，但这还不够，比如样本可能都位于子像素网格的对角线上，所以对于几乎平行于该对角线的边缘，结果会很差，见图5.27。为了获得更好的采样，我们要避免将两个样本靠得很近，我们还希望分布均匀，将样本均匀分布在整个区域，为了形成这样的模式，我们会将如拉丁超立方体采样的分层采样（stratified sampling）技术，与其他例如抖动（jittering）、霍尔顿序列（Halton sequences）和泊松磁盘采样（Poisson disk sampling）的方法相结合[1413，1758]。

游戏机里面的人物

低可信度描述已自动生成

**图 5.27. N-rooks采样，左侧是一个符合规则的N-rooks模式，但是它在捕捉沿对角线的三角形边缘上表现较差。**

实际中，GPU制造商通常将这种采样模式硬连接到他们的硬件中，用于多重采样反走样，图5.28显示了在实践中使用的一些MSAA模式。对于时间性抗锯齿来说，覆盖模式是程序员想要的任何模式，因为样本位置可以在帧与帧之间变化，例如，Karis[862]发现基本的霍尔顿序列比GPU提供的任何MSAA模式效果都要好，霍尔顿序列在空间中产生看似随机但差异很小的样本，也就是说，它们很好地分布在空间上，没有聚集在一起[1413, 1938]。

图表, 散点图

描述已自动生成

**图 5.28. 用于AMD和NVIDIA图形加速器的MSAA采样模式，绿色的方块是阴影样本的位置，红色的方块是计算和保存的位置样本，从左到右：2×，4×，6×（AMD），和8×（NVIDIA）采样。（由D3D FSAA查看器生成）**

虽然子像素网格模式可以更好地近似每个三角形如何覆盖网格单元，但它并不理想，场景可以由屏幕上任意小的物体组成，这意味着没有采样率可以完美地捕获它们，如果这些微小的物体或特征形成图案，则以恒定间隔进行采样可能会导致莫尔条纹（Moiré fringes）和其他干涉图案，超级采样中使用的网格模式特别容易产生走样。

一种解决方案是使用随机采样（stochastic sampling），这样可以提供更加随机的模式，如图5.28所示的模式肯定可以使用。想象一下，远处有一个拥有漂亮梳齿的梳子，每个像素覆盖几根梳齿，当采样模式与梳齿频率异相时，规则模式就会产生严重的伪像，不那么有序的采样模式可以分解这些模式。（译注：意译。原文Having a less ordered sampling pattern can break up these patterns.）随机化倾向于用噪声代替重复的走样效果，因为人类视觉系统对此更宽容[1413]，这对结构较少的模式是有益的，但当像素间重复时，它仍会出现走样。一种解决方法是在每个像素上使用不同的采样模式，或者随时更改每个采样位置，在交错采样（Interleaved sampling）和索引采样（index sampling）方法中，一组像素的每个像素都有不同的采样模式，（译注：原文中这一句是乱码，Interleaved samplingindexsampling!interleaved,）过去几十年中偶尔会在硬件中提供支持。例如，ATI的SMOOTHVISION允许每个像素最多有16个样本和最多16个用户定义的采样模式，这些模式可以混合在一个重复的模式中(例如，在一个4-4像素贴图中)。Molnar[1234]以及Keller和Heidrich[880]发现，对于每个像素使用相同的模式时，使用交错式随机采样可以最大程度地减少走样伪像。

值得注意的是其他一些GPU支持的算法，NVIDIA的较早的Quincunx方法[365]是一种使样本影响一个以上像素的实时反走样方案，“Quincunx”是指五个对象的排列，四个在正方形上，第五个在中心，例如在骰子（six-sided die）上五个点那一面的图案。Quincunx多重采样反走样使用此模式，将四个外部采样置于像素的角落，见图5.25，每个角落的采样值被分发给它相邻的四个像素，与其他大多数实时方案对每个样本进行平均加权不同，中心样本赋予1/2的权重，每个角落样本赋予1/8的权重，由于这种共享，每个像素平均只需要两个样本，其结果比两个样本的FSAA方法要好得多[1678]。这种模式近似于二维tent滤波器，如上一节所述，它优于box滤波器。

通过每像素使用单个样本的方法，也可以将Quincunx采样应用于时间性反走样上[836, 1677]，其中每帧在每个轴上都比之前的帧偏移半个像素，偏移方向在帧与帧之间交替，前一帧提供像素角落的点样本，并且使用双线性插值来快速计算每个像素的贡献值，将结果与当前帧取平均值。每个帧的权重相等意味着静态视图没有闪烁的伪像，对齐移动中物体的问题仍然存在，但是该方案本身很容易编写代码，并且在每帧每像素只使用一个样本的情况下会产生更好的效果。

当在单帧中使用时，Quincunx通过在像素边界共享样本而仅具有两样本的低成本消耗，RGSS模式更擅长捕捉更多的接近水平和垂直边缘的渐变。FLIPQUAD模式最初是为移动图形设备开发的，结合了这两种理想的功能[22]，它的优点是成本仅为每像素两样本，并且质量类似于RGSS（每像素四样本），这种采样模式如图5.29所示。另外，Hasselgren等人[677]探索了其他一些利用样本共享方式的廉价采样模式。

图表, 图示

描述已自动生成

**图 5.29. 左侧显示了RGSS采样模式，每像素花费四个样本，通过将这些位置移到像素边缘，可以跨边缘进行样本共享，但是要做到这样，必须使每个其他像素具有镜像的采样模式，如右图所示，所得的样本模式称为FLIPQUAD，每个像素花费两个样本。**

与Quincunx一样，双样本的FLIPQUAD模式也可以与时间性反走样一起使用，并分布在两个帧上。Drobot[382，383，1154]解决了在他关于混合重建反走样（hybrid reconstruction antialiasing, HRAA）的研究工作中哪种双样本模式最好的问题，他探索了用于时间性反走样的不同采样模式，并发现FLIPQUAD模式是所测试的五种模式中最好的。棋盘格模式也被用于时间性反走样，El Mansouri[415]讨论了使用两个样本MSAA创建棋盘渲染，以减少着色器成本，同时解决走样问题。Jimenez[836]使用SMAA，时间性反走样技术以及多种其他技术来提供一种解决方案，其中反走样质量可以根据渲染引擎负载来改变。Carpentier和Ishiyama[231]在边缘采样，将采样网格旋转了45°，他们将此时间性反走样方案与FXAA（稍后讨论）结合在一起，以在更高分辨率的显示器上进行有效的渲染。

**形态学方法 Morphological Methods**

走样通常是由边缘引起的，例如由几何形状，尖锐阴影或明亮高光形成的边缘，走样具有与边缘相关的结构，可以加以利用来提供更好的反走样结果。2009年，Reshetov[1483]沿着这些思路提出了一种算法，称其为形态学反走样（morphological antialiasing, MLAA），其中“形态”是指“与结构或形状有关”。早在1983年，Bloomenthal[170]就在这一领域做了较早的工作[830]，Reshetov的论文重新激活对了多重采样替代方法的研究，强调寻找和重建边缘[1486]。

这种反走样形式是在后处理（post-process）中执行的，也就是说以通常的方式进行渲染，然后将结果反馈到生成反走样结果的过程中去。自2009年以来，已经开发出了多种技术，那些依赖于其他缓冲区（例如深度和法线）的分析方法可以提供更好的结果，例如子像素重建反走样（subpixel reconstruction antialiasing，SRAA）[43, 829]，但仅适用于对几何边缘进行反走样。诸如几何缓冲区反走样（geometry buffer antialiasing, GBAA）和距离边缘反走样（distance-to-edge antialiasing, DEAA）之类的分析方法，会使渲染器计算有关三角形边缘位于何处的附加信息，例如边缘距像素中心的距离有多少[829]。

最通用的方案只需要颜色缓冲区，这意味着它们还可以从阴影，高光或之前应用的各种后处理技术，如轮廓边缘渲染（silhouette edge rendering）中改善边缘（见15.2.3节），例如方向局部反走样（directionally localized antialiasing, DLAA）[52, 829]是基于以下观察结果，接近垂直的边缘应水平模糊，同样接近水平的边缘也应与其相邻像素垂直模糊。

边缘检测的更复杂形式尝试寻找可能包含任意角度的边缘的像素并确定其覆盖范围，检查潜在边缘周围的邻域，目标是尽可能地重建原始边缘所在的位置，然后可以使用边缘对像素的效果来融合相邻像素的颜色，有关过程的概念视图，请参见图5.30。

卡通人物

中度可信度描述已自动生成

**图 5.30. 形态学反走样，左侧是走样图像，我们的目的是确定形成边缘的可能方向。中间表示，该算法通过检查相邻像素来记录其为边缘的可能性，给定样本后，显示了两个可能的边缘位置。右侧表示，利用最佳猜测边缘将相邻颜色按估计覆盖率成比例的混合到中心像素中。图像中的每个像素重复会这个过程。**

Iourcha等人[798]通过检查像素中的MSAA样本来计算更好的结果，从而改善了边缘查找。请注意，边缘预测和融合可以比基于样本的算法提供更高的精度，例如，一种使用每像素四样本的技术只能为对象的边缘提供以下五个混合级别：无样本覆盖，一个样本覆盖，两个样本，三个样本和四个样本。估计的边缘位置可以有更多的位置，因此提供了更好的结果。

基于图像的算法中有几种可能会误入歧途：首先，如果两个对象之间的色差低于算法的阈值，则可能无法检测到边缘；具有三个或更多不同表面重叠的像素很难进行转换；在具有高对比度或高频率元素的表面，颜色在像素间快速变化会导致算法错过边缘；特别地，当对文本应用形态学反走样时，其显示质量通常会受到影响；对象的角落部分可能是一个挑战，有些算法可能会给它们圆润的外观，假设边缘是直的，曲线也会受到其不好的影响；单个像素变化可能会导致边缘重建方式发生很大变化，从而在帧与帧之间产生明显的伪像，改善此问题的一种方法是使用MSAA覆盖遮挡（MSAA coverage masks，译注：mask在上下文中多次提到，Unity官方文档中对其有如下解释：mask不是可见的UI控件，而是一种修改控件子元素外观的方法，mask将子元素限制（即“遮挡”）为父元素的形状。其使用GPU模板缓冲区实现）来改进边缘确定性。

形态学反走样方案仅使用所提供的信息，举个例子，宽度小于像素的物体，如电线或绳索，只要它没有覆盖一个像素的中心位置，就将在屏幕上出现间隙，在这种情况下，采集更多的样本可以提高质量，但仅基于图像的反走样是不行的。此外，执行时间会根据展示的内容而变化，例如，一片草地的景色所需的反走样时间是天空的三倍[231]。

综上所述，基于图像的方法可以为较小的内存和处理成本提供反走样支持，因此它们被用于许多应用程序中，仅颜色的版本还与渲染管线分离，使其更容易修改或禁用，并且甚至可以公开为GPU的驱动程序选项。两种最流行的算法是快速近似反走样（fast approximate antialiasing, FXAA）[1079, 1080, 1084]和子像素形态反走样（subpixel morphological antialiasing, SMAA）[828, 830, 834]，部分原因是它们都为各种设备提供了可靠的（以及免费的）源代码实现，两种算法都仅使用颜色的输入，SMAA具有能够访问MSAA样本的优势，每个算法都有自己的各种可配置选项，以便在速度和质量之间进行权衡，每帧消耗通常在1-2毫秒的范围内，主要因为这是视频游戏所愿意花费的时间，最后，两种算法都可以应用时间性反走样[1812]，Jimenez[836]提出了一种改进的SMAA实现，比FXAA更快，并描述了一种时间性反走样方案。最后，我们向读者推荐Reshetov和Jimenez[1486]对形态学技术及其在视频游戏中的使用的广泛评论。

5.5 透明度，Alpha值，与合成 Transparency, Alpha, and Compositing

**半透明物体允许光通过的方法有许多，对于渲染算法而言，这些方法大致可以分为基于光线的效果及基于观察的效果，基于光线的效果指那些物体使光线衰弱或转移，导致场景中的其他物体被不同地照亮和渲染的效果，基于观察的效果指的是半透明物体本身如何被渲染的效果。**

**在本节中，我们将讨论基于视线的透明度的最简单形式，其中半透明的物体作为它后面的物体颜色的衰减器。更详细的基于视线和光线的效果，例如毛玻璃（frosted glass），光的弯曲（折射），由于透明物体的厚度导致的光线衰减，以及由于视角导致的反射率和透射率变化等，将在后面的章节中讨论。**

**一种制造透明假象的方法叫做屏幕门透明（screen-door transparency）[1244]，其思路是用像素对齐的棋盘格填充模式渲染透明三角形，也就是说三角形的像素每隔一个（every other）被渲染，从而使其后面的对象部分可见，通常情况下屏幕上的像素距离很近，以至于棋盘图案本身是不可见的。这种方法的一个主要缺点是，通常只能在屏幕的一个区域上令人信服地渲染仅一个透明对象，举个例子，如果透明的红色对象和透明的绿色对象在蓝色对象之上渲染，则三种颜色中只有两种可以出现在棋盘格图案上。此外，50%的棋盘格效果是很有限的，其他更大的像素遮掩可以用来给出其他百分比，但那往往产生可使人察觉（是透明假象）的模式。**

**之前说过，这种技术的一个优势是它比较简单，透明对象可在任意时间，以任何顺序渲染，且不需特殊的硬件支持，通过使所有对象在它们覆盖的像素处不透明，透明度问题就解决了。同样的想法也用于裁剪纹理的边缘反走样，但在子像素级别，使用了一种称为通道印射（alpha to coverage）的特性（第6.6节）。**

**由Enderton等人介绍[423]，随机透明（stochastic transparency）的方法使用子像素屏幕门遮挡与随机采样相结合而成，一个合理但充满噪声的图像，可以通过使用随机点画模式（random stipple patterns）表示片元通道印射而生成，见图5.31。为了让结果看起来合理，每个像素需要大量的样本，以及为所有子像素样本准备相当数量的内存。此方法吸引人的是不需要混合，和反走样，透明度，以及任何其他由单一机制产生的部分覆盖像素的现象。**

图示

描述已自动生成

**图 5.31. 随机透明，产生的噪声显示在放大区域中。图片来自NVIDIA SDK11 [1301]样本，由NVIDIA公司提供。）**

**大多数透明度算法会将透明对象的颜色与其后面对象的颜色混合在一起，为此，需要阿尔法混合（alpha blending）的概念[199, 387, 1429]。当一个对象在屏幕上渲染，每个像素都关联有一个RGB颜色和一个z缓冲区深度，也可以为对象覆盖的每个像素定义，另一个称为alpha(****α)的组成部分，alpha是一个值，描述给定像素的对象片段的不透明度和覆盖率。alpha为1.0意味着物体是不透明的，并且完全覆盖了对应的像素区域，为0.0表示像素完全不被遮挡，即片段是完全透明的。**

**像素的alpha可以表示不透明度或覆盖率，或同时表示两者，这取决于环境。例如，肥皂泡的边缘可能会覆盖像素的四分之三，即0.75，并且可能几乎是透明的，从而使十分之九的光线直达眼睛，所以它是十分之一不透明的，即0.1，所以它的alpha值是0.75×0.1=0.075。但是，如果我们使用MSAA或类似的反走样方案，覆盖率将通过样本自身而被考虑在内，四分之三的样本将受到肥皂泡的影响。因此在每个这样的样本中，我们将使用0.1不透明度作为alpha值。**

5.5.1 混合顺序 Blending Order

**为了使对象看起来透明，它得以小于1.0的alpha值渲染到现有场景的顶部，对象覆盖的每个像素都将从像素着色器收到一个结果****RGBα(也叫RGBA)，通常使用over操作将此片段的值与原始像素颜色混合，如下所示：**

|  |  |
| --- | --- |
|  | (5.24) |

**其中是透明对象的颜色（称作来源），是对象的alpha值，是混合前的像素颜色（称作目标），是将透明对象over在现有场景上而产生的最终颜色。在渲染管线传入和的情况下，像素的原始颜色被结果所取代。如果传入的RGBα实际上是不透明的（=1.0），则该公式简化为用对象的颜色完全替换像素的颜色。**

**示例：混合。红色的半透明对象被渲染到蓝色背景上，假设对象在某个像素处的RGB着色为（0.9**, **0.2**, **0.1），背景为（0.1**, **0.1**, **0.9），并且对象的不透明度设置为0.6，然后将这两种颜色混合**

|  |  |
| --- | --- |
|  |  |

**得到颜色为（0.58**, **0.16**, **0.42）。**

**over操作符为被渲染的对象提供半透明外观，这样做的透明度是有效的，在某种意义上，只要可以透过物体看到它背后的物体，我们就认为它是透明的[754]。使用over模拟现实世界中薄纱织物的效果时，织物后面的物体被部分遮蔽，因为织物的线是不透明的。在实际中，宽松织物的alpha覆盖率随（观察）角度变化而变化[386]，我们这里的重点是alpha模拟了有多少材质覆盖像素。**

图片包含 游戏机, 房间, 桌子

描述已自动生成

**图 5.32. 红色薄纱正方形的织物与红色的塑料滤波器，具有不同的透明效果。注意，它们的阴影也不同。（照片由Morgan McGuire提供。）**

**over操作符模拟其他透明物体时没什么说服力，尤其是模拟通过有色玻璃或塑料观察时，在现实世界中，放置在蓝色物体前面的红色滤镜通常会使蓝色物体看起来很暗，因为该物体反射的可以穿过红色滤镜光线很少，见图5.32。当使用over进行混合时，结果是红色的一部分和蓝色部分相加在一起，更好的方法应该是将这两种颜色相乘，也可以添加对透明对象本身的任意反射，第14.5.1节和第14.5.2节中讨论了这种类型的物理透射率。**

**在基本的混合阶段运算符中，over是通常用于透明效果的运算符[199，1429]，另一种有用的操作是加法混合（additive blending），即将像素值简单地求和，如下所示，**

|  |  |
| --- | --- |
|  | (5.25) |

**这种混合模式可以很好地用于发光效果，如闪电或火花，这些效果不会使对象后的像素衰减，只会使它们变得更亮[1813]。然而此模式的透明度看起来不正确，因为不透明的表面似乎没有被滤波[1192]，对于几种分层的半透明表面，例如烟或火焰之类，加法混合具有使结果的颜色更饱和的效果[1273]。**

**为了正确渲染透明对象，我们需要在不透明对象之后绘制它们，这通过首先关闭混合以渲染所有不透明对象，然后开启over以渲染透明对象。从理论上讲，我们总是可以让over开启，因为不透明的alpha为1.0会给出源颜色并隐藏目标颜色，但是这样做成本更高，没什么收益。**

**z缓冲区的限制是每个像素只能存储一个对象，如果多个透明对象重叠在同一像素，仅z缓冲区无法存储且在之后无法解决所有可见对象的影响。当使用over时，任意给定像素处的透明表面通常都需要以从后向前的顺序进行渲染，不这样做的话会给出错误的知觉暗示，一种实现这种排序的方法是，按单个对象的形心沿视线方向的距离对其进行排序，这种粗略的分类可以很好地工作，但是在各种情况下都有许多问题。首先这里的顺序只是一个近似值，因此被归类为较远的物体可能在被认为较近的物体前面；互相穿透的对象无法在每个网格的基础上对所有视角进行解析，除非把每个网格都拆成单独的部件，相关案例，请参见图5.33的左图；甚至具有凹面的单个网格，也会在屏幕上自重叠的视线方向上出现排序问题。**

图示

描述已自动生成

**图 5.33. 左侧，使用z缓冲区以透明方式渲染模型，以任意顺序渲染网格会产生严重的错误。右侧，深度剥离（depth peeling）可提供正确的外观，但要消耗额外的pass数。（图片由NVIDIA Corporation提供。）**

**尽管如此，由于其简单性与速度，以及它不需要额外的内存或特殊GPU支持，我们仍然经常使用这种对透明度进行粗糙排序的方法。如果应用这种方法，通常最好在执行透明度时关闭z深度替换功能，也就是说，z缓冲区仍被正常测试，但保留下来的表面不会改变已存储的z深度，最接近的不透明表面的深度保持不变。这样，所有透明物体至少会以某种形式出现，而不是在照相机旋转导致的排序顺序更改时突然出现或消失。其他技术也有助于改善外观表现，比如你可以把每个透明网格画两遍，首先渲染背面，然后渲染正面[1192**, **1255]。**

**over方程也可被修改，使得从前向后混合能得到相同的结果，这种混合模式称为under操作符：**

|  |  |
| --- | --- |
|  | (5.26) |

**注意，under要求目标保持alpha值，而over则不需要，换句话说，目标——在它之下混合了更近的透明表面——并不是不透明的，因此需要有alpha值，under的公式和over相似，但是交换了源和目标。另外注意，用于计算alpha值的公式与顺序无关，因为源alpha值和目标alpha值可以交换，结果都是相同的最终alpha值。**

**alpha的方程来自于将片元的alpha视为覆盖率，Porter和Duff[1429]注意到，由于我们不知道每个片元覆盖区域的形状，因此我们假设每个片元都按其alpha值比例去覆盖其他片元。例如如果，则以某种方式将像素分为两个区域，其中0.7被源片元覆盖，而另外0.3不是。在没有任何其他条件的情况下，假设目标片元覆盖率，将会被源片元按比例重叠，（译注：整个源片元，而不是只有0.7的那部分）该公式具有几何解释，如图5.34所示。**

图示

描述已自动生成

**图 5.34. 一个像素和两个片元s和d，通过将两个片元沿不同轴线对齐，每个片元会按一定比例覆盖另一个，也就是说，它们是不相关的。被两个片元覆盖的面积等于under输出的alpha值，这相当于把两个区域相加，然后减去它们重叠的区域。**

5.4.1 与顺序无关的透明度 Order-Independent Transparency

under操作符用于将所有透明物体绘制到单独的颜色缓冲区中，然后使用over将这个颜色缓冲区合并到在不透明的场景之上。under操作符的另一种用途是执行称为深度剥离（depth peeling）[449, 1115]的与顺序无关的透明度（order-independent transparency，OIT）算法，顺序无关意味着应用程序不需要执行排序，深度剥离背后的思想是使用两个z缓冲区和多个pass。首先，渲染一个pass，以使所有表面的z深度（包括透明表面）都位于第一个z缓冲区中，在第二个pass中，将会渲染所有透明物体，如果一个物体的z深度与第一个z缓冲区中的值匹配，则我们知道这是最近的透明物体，然后将它的**RGBα保存到单独的颜色缓冲中。我们还通过保存超出第一个z深度并且最接近的透明对象（如果有的话）的z深度，来“剥离”该层。这个z深度是第二近的透明物体的距离，接下来的一系列pass继续使用under剥离和添加透明层，经过一定数量的pass渲染之后停下来，然后将透明图像混合在不透明图像之上，见图5.35。**

画里面的卡通人物

中度可信度描述已自动生成

**图 5.35. 每个深度剥离pass都绘制一个透明层。左侧是第一遍pass，显示了直接可见的图层。中间图显示的第二层，在每个像素处展示了距离第二近的透明表面，在这种情况下为对象的背面。右侧的第三层是一组距离第三近的透明表面。最终结果可以在图14.33中找到。（图片由Louis Bavoil提供。）**

该方案的几种变体已经研发出来了，例如Thibieroz[1763]给出了一种从后向前计算的算法，其优点是能够立即混合透明值，这意味着不需要单独的alpha channel。深度剥离的一个问题是需要知道究竟多少pass足以捕获所有透明层，一种硬件上的解决方案是提供一个像素绘制计数器，它告知渲染过程中写入了多少像素，当pass未渲染任何像素时，渲染就完成了，使用under的好处是，最重要的透明层——眼睛首先看到的那些——会在早期渲染。每个透明表面总是增加它覆盖的像素的alpha值，如果像素的alpha值接近1.0，则混合的贡献会使像素几乎不透明，所以更远的物体会有一个可以忽略不计的效果[394]。当一个pass渲染的像素数低于某个最小值，或可以指定固定数量的pass时，可以缩短从前向后的剥离过程，然而这对于从后向前的剥离效果不佳，因为距离最近（并且通常是最重要）的层是最后绘制的，因此可能会因提前终止而丧失。

尽管深度剥离是有效的，但速度可能很慢，因为每一层剥离都是针对所有透明对象的一个独立渲染pass，Bavoil和Myers[118]提出了双重深度剥离（dual depth peeling），其中每个pass剥离和保存最近最远两个深度剥离层，从而将渲染pass的数量减少一半。Liu等人[1056]探索了一种桶排序方法（bucket sort method），该方法单个pass可捕获多达32层，这种方法的一个缺点是，它需要大量内存才能为所有层保持排序顺序，通过MSAA或类似方法进行反走样将极大增加成本。

将透明物体以交互速率正确地混合在一起的难点不是我们缺乏算法，而是如何将这些算法有效地映射到GPU。1984年，Carpenter提出了A缓冲区[230]，这是多重采样的另一种形式，在A缓冲区中，渲染的每个三角形都会为其完全或部分覆盖的每个屏幕网格创建一个覆盖遮挡（coverage mask），每个像素存储所有相关片元的列表（list），不透明的片元可以清除它们后面的片元，类似于z缓冲区，对于透明表面，所有片元都被存储，一旦所有列表形成，就可以通过遍历片元和解析每个样本来产生最终结果。

在GPU上创建片元的链表的想法通过DirectX 11[611, 1765]中公开的新功能成为可能，使用的特性包括无序访问视图(unordered access views, UAVs)和原子操作（atomic operations），描述于第3.8节。基于MSAA的反走样可以通过访问覆盖遮挡和计算每个样本的像素着色器的方式来实现，该算法通过对每个透明表面进行光栅化，并将生成的片元插入长数组从而实现，连同颜色和深度一起，生成一个单独的指针结构，用于将像素的每个片元与之前存储的片元相链接，然后执行单独的pass，在此渲染屏幕填充四边形，以便在每个像素处计算像素着色器，着色器通过追踪链接来检索每个像素处的所有透明片元，检索到的每个片元都与先前的片元依次排序，然后将排序后的链表从后向前混合，从而得出最终的像素颜色，由于混合由像素着色器执行，如果需要，可以为每个像素指定不同的混合模式。随着GPU和API的不断发展，性能表现也不断通过减少原子运算符的使用消耗来提高[914]。

A缓冲区有只分配每个像素所需的片段的优点，GPU上的链表实现也是如此，从某种意义上讲，这也可能是一种缺点，因为在开始渲染帧之前所需的存储量是未知的，具有头发、烟雾或其他有可能具有许多重叠的透明表面的物体，会产生巨量的片元。Andersson[46]指出，对于复杂的游戏场景，最多可以重叠50个物体（例如树叶）的透明网格和最多200个半透明粒子。

GPU通常有预先分配好的内存资源，例如缓冲区和数组，链表方法也不例外，用户需要决定需要多少内存，而内存耗尽会导致明显的伪像。Salvi和Vaidyanathan[1532]提出了一种解决此问题的方法，多层alpha混合（multi-layer alpha blending），使用英特尔推出的称为像素同步的GPU特性，见图5.36。该功能提供可编程混合模式，且开销比原子操作要少，他们的方法重新定义了存储和混合，因此如果内存耗尽，它会适当的降低性能，粗略的排序顺序有利于他们的方案。DirectX 11.3引入了光栅化程序顺序视图（rasterizer order views）（第3.8节），这是缓冲区的一种类型，它允许在支持该功能的任何GPU上实现这个透明方法[327，328]。移动设备也具有类似的技术，称为图块本地存储（tile local storage），这允许它们实现多层alpha混合[153]，然而这种机制具有性能成本，所以这种类型的算法的消耗可能很大[1931]。

墙上挂着一幅画

低可信度描述已自动生成

**图 5.36. 在左上方，执行传统的从后到前的alpha混合，由于排序顺序不正确，导致渲染错误。在右上方，A缓冲区用于提供完美的非交互结果。左下方展示了具有多层alpha混合的渲染。右下方显示了A缓冲区和多层alpha混合之间图像的差异，乘以4以增加可见度[1532]。（图片由英特尔公司的Marco Salvi和Karthik Vaidyanathan提供。）**

这种方法（多层alpha混合）建立在Bavoil等人[115]提出的k缓冲区的概念上，其中保存了前几层可见图层并尽可能地进行了排序，而更深的图层则被丢弃并尽可能地进行了合并。Maule等人[1142]使用k缓冲区，和加权平均（weighted averaging）来计算这些更远的深层图层，加权和[1202]与加权平均[118]透明技术都是顺序无关的，且都是单pass，并且几乎可以在任意GPU上运行，但它们的问题在于没有考虑对象的顺序，例如使用alpha表示覆盖率，在一条淡蓝色薄纱围巾上的一条淡红色薄纱围巾呈现出紫罗兰色，而不是正确地看到带有一点蓝色的红色围巾。虽然对于接近不透明的对象得到的结果很差，但这类算法对于可视化是有用的，并且对高度透明的表面和粒子可以很好地工作，见图5.37。

图表, 气泡图

描述已自动生成

**图 5.37. 随着不透明度的增加，对象的顺序变得越来越重要。（图片由Dunn提供[394]。）**

在加权和透明度中，公式为

|  |  |
| --- | --- |
|  | (5.27) |

其中是透明表面的数量，和表示透明值的集合，是场景中不透明部分的颜色，渲染透明表面时将两个和累加并分别存储，并在处理透明的pass结尾处，在每个像素上计算该公式。该方法的问题在于，第一个求和是饱和的，即会得到大于（1.0, 1.0, 1.0）的颜色值，并且因为alpha的总和可能超过1.0，所以背景颜色可能会产生负面影响。

通常我们首选加权平均公式，因为它避免了这些问题：

|  |  |
| --- | --- |
|  | (5.28) |

第一行表示透明渲染过程中在两个缓冲区独自生成的结果，每个对有贡献的表面都获得一个由其alpha提供的影响权重，几乎不透明的表面提供了更多的颜色，几乎透明的表面则几乎没有影响，通过将除以，我们得到了加权的平均透明度颜色，值是所有alpha值的平均值，值是对个透明表面应用此平均alpha次数次后，目标（不透明场景）的可见性估计值，最后一行实际上是over运算符，其中代表源的alpha。

加权平均值的一个限制是对于相同的alpha，它均等地混合了所有颜色，而不论顺序如何。McGuire和Bavoil[1176, 1180]引入了加权混合（weighted blended）的顺序无关的透明方案，以提供更具说服力的结果，在他们的模拟中，到表面的距离也会影响权重，越近的表面会受到更大的影响，同时比起对alpha进行平均，是通过将项相乘在一起，并用1减去它，从而获得一组表面的真实alpha覆盖率。这种方法在视觉上产生更令人信服的结果，如图5.38所示。

图示

中度可信度描述已自动生成

**图 5.38. 查看同一引擎模型的两个不同的摄像机位置，均使用加权混合的顺序无关的透明方案进行渲染。按距离加权有助于弄清哪些表面更靠近观察者[1185]。（图片由Morgan McGuire提供。）**

该方案的一个缺点是，在一个大环境中，彼此靠近的对象的距离权重几乎是相等的，使结果与加权平均值几乎没有区别。另外，随着相机到透明物体距离的改变，深度权重实际上可能会发生变化，但是这种变化是逐渐发生的。

McGuire和Mara[1181, 1185]扩展了该方法，使其包含了合理的颜色传输效果（transmission color effect），如前所述，本节中讨论的所有透明度算法都将各种颜色混合在一起而不是对其进行滤波，从而模仿了像素覆盖率。为了得到颜色滤波器的效果，不透明场景由像素着色器读取，每个透明表面将它在场景中覆盖的像素乘以自身的颜色，并将结果保存到第三个缓冲区，这个缓冲区，其中不透明的对象现在被半透明的对象染色，在接下来解析透明缓冲区时，会用来代替不透明的场景。该方法之所以有效是因为，与透明度造成覆盖不同，颜色的传输是顺序无关的。

还有其他算法用到了这里所介绍的技术中的部分内容，例如，Wyman[1931]通过内存需求，插入和合并方法对之前的这些工作进行了分类，无论是用到的alpha值或几何覆盖率，还是如何处理丢弃的片元。通过寻找先前研究中的漏洞，他展示了两种发现的新方法，他的随机分层alpha混合方法使用了k缓冲区，加权平均值和随机透明度，另一种算法是Salvi和Vaidyanathan方法（多层alpha）的一种变体，其中使用覆盖遮挡代替alpha值。

尽管我们给出了种类繁多的透明相关内容、渲染方法和GPU功能，但是并没有完美的解决方案来渲染透明对象。我们推荐感兴趣的读者阅读Wyman的论文[1931]和Maule等人的关于交互式透明性算法的更详细的研究[1141]，McGuire的演讲[1182]提供了该领域的更广阔视野，它融会贯通了其他相关现象，例如体积照明，颜色传输和折射等，这些在本书后面会有更深入的讨论。

5.5.3 预乘Alpha与合成 Premultiplied Alphas and Compositing

**over操作符还用于将照片或对象的合成渲染混合到一起，该过程称之为合成（compositing）[199, 1662]，在这种情况下，每个像素的alpha值将与对象的RGB颜色值一起存储，由alpha channel形成的图像有时称为无光粗糙层（matte），它显示了对象的轮廓形状，相关示例参见图6.27。这个RGBα图像可以用来混合其他类似的内容或与背景做对比。**

**使用合成RGBα数据的一种方法是使用预乘alpha（premultiplied alphas，也称为关联的alpha），即在被使用之前将RGB值乘以alpha值，这使合成over公式的效率更高：**

|  |  |
| --- | --- |
|  | (5.29) |

**其中**表示**预乘的源通道，代替公式5.25中的**，预乘alpha也可以在不更改混合状态的情况下使用over和叠加混合，因为现在在混合过程中添加了源颜色[394]，请注意使用预乘的**RGBα**值，RGB分量通常不大于alpha值，尽管可以将它们创建为特别明亮的半透明值。

**合成图像的渲染很自然地与预乘alpha相吻合，默认情况下，在黑色背景上渲染的反走样不透明对象会提供预乘值，假设白色（1, 1, 1）三角形沿其边缘覆盖了某些像素的40％，使用（极精确的）反走样功能，这些像素值将设置成值为0.4的灰色，即我们将为此像素保存颜色（0.4, 0.4, 0.4），如果存储的是alpha值，那么也将为0.4，因为这是三角形覆盖的区域。最终****RGBα值为（0.4, 0.4, 0.4, 0.4），这是一个预乘值。**

**图像存储的另一种方式是使用未相乘的alpha（unmultiplied alphas），它也被称为未关联的alpha（unassociated alphas），甚至一个奇怪的术语——非预乘的alpha（nonpremultiplied alphas），未相乘的alpha就是它的字面意思：RGB值不乘以alpha值。对于白色三角形的示例，未相乘的颜色为（1, 1, 1****, 0.4），这种表示形式的优点是可以存储三角形的原始颜色，但是在显示之前，始终需要将该颜色乘以存储的alpha值。最好在执行滤波和混合操作时使用预乘数据，因为使用未相乘的alpha不能正确执行线性插值之类的操作[108, 164]，会产生诸如围绕对象边缘的黑色条纹之类的伪像[295, 648]，第6.6节的末尾有更多讨论。预乘alpha也可进行更清晰的理论处理[1662]。**

**对于图像处理应用程序来说，未关联的alpha值有利于遮挡（mask）照片而不影响基础图像的原始数据，而且未关联的alpha意味着可以使用颜色channel的全部精度范围，也就是说，必须小心地将未相乘的RGBα值正确地转换到用于计算机图形计算的线性空间，或转换回来。比如没有浏览器能正确地完成这一点，它们也不可能这样做，因为不正确的行为现在是满足预期的[649]，支持alpha的图像文件格式包括PNG（仅非关联的alpha），OpenEXR（仅关联）和TIFF（支持两种alpha类型）。**

**一个与alpha channel相关的概念是色度键控（chroma-keying）[199]，这是视频制作中的一个术语，表示演员是在绿色或蓝色屏幕上拍摄并与背景融合在一起的，在电影工业中，此过程称为绿幕（green-screening）或蓝幕（blue-screening）。这里的思路是，将特定的色调（用于胶卷）或精确值（用于计算机图形）指定为透明，当它被检测到时就会显示背景，这允许图像只使用RGB颜色来给出一个轮廓形状，无需存储任何alpha。该方案的一个缺点是，对象在任意像素处要么完全不透明，要么完全透明，即alpha实际仅为1.0或0.0，比如GIF格式允许一种颜色被指定为透明。**

5.6 显示编码 Display Encoding

**当我们计算照明效果，纹理或其他操作时，假定使用的值是线性（linear）的，非正式情况下，这意味着加法和乘法按预期执行，但是为了避免各种视觉伪像，显示缓冲区和纹理使用非线性编码，这是我们必须要考虑的。关于该问题，简短而粗略的答案如下：取着色器输出颜色范围为[0, 1]并以1/2.2作为它的幂，进行所谓的伽马校正（gamma correction），同时对传入的纹理和颜色执行相反的操作。大多数情况下，你可以告诉GPU让它为你执行这些操作，本节将快速总结这么做的方法和原因。**

**我们从阴极射线管（cathode-ray tube, CRT）开始，在数字成像的早期，CRT显示器是标准设备。这些设备在输入电压和显示辐射率之间表现出指数关系（power law），当施加于像素的能量级别增加时，发出的辐射率不会呈现线性增长，而是（令人惊讶地）以大于1的幂比例增长，例如假设指数为2，**比起被设置为1的像素，**一个设置为50%的像素会发出四分之一的光，即**。虽然LCD和其他显示技术具有与CRT不同的固有色调响应曲线（intrinsic tone response curves），但它们是通过转换电路制造的，这使它们与CRT的响应类似。

**该幂函数几乎与人类版本的亮度灵敏度相反[1431]，这种幸运巧合的结果是，编码在感知上大致是一致，也就是说，在可显示范围内，一对编码值N和N+1之间的感知差异大致是恒定的，通过测量阈值反衬（threshold contrast），我们可以在很大范围的条件下检测到大约1%的亮度差异。当颜色存储在精度有限的显示缓冲区时，这种近乎最佳的值分布最小化了条带伪像（banding artifacts），（第23.6节），同样的优点也适用于纹理，它们通常使用相同的编码。**

**显示传递函数（display transfer function），描述了显示缓冲区中的数字值与从显示器发出的辐射等级之间的关系，因为这个理由，它也被称为电光传递函数（electrical optical transfer function, EOTF）。显示传输功能是硬件的一部分，并且对于计算机显示器，电视和电影放映机有着不同的标准，还有一个标准传递函数，用于过程的另一端：图像和视频捕获设备，该函数被称为光电传递函数（optical electric transfer function, OETF）[672]。**

**在编码用于显示的线性颜色值时，我们的目标是消除显示传递函数的影响，使得无论我们计算出什么值，都会发出相应的辐射级别，比如如果计算值翻倍，则希望输出亮度翻倍，为了保持这种关系，我们使用了显示传递函数的逆函数来抵消它的非线性影响，这个使显示器响应曲线无效的过程也称为伽马校正，我们很快就会知道为什么要这样做。在解码纹理值时，我们需要应用显示传递函数来生成一个用于着色的线性值，图5.39显示了解码和编码在显示过程中的使用。**

图示

描述已自动生成

**图 5.39. 在左侧，一个PNG颜色纹理被GPU着色器访问，并且它的非线性编码值被转换（蓝色）到线性值，在着色和色调映射（tone mapping）之后（8.2.2节），最终计算值被编码（绿色）并且存储在帧缓冲区中，该值与显示传递函数决定了发射的辐射量（红色）。绿色和红色功能的组合抵消了，因此发出的辐射与线性计算值成正比。**

**个人计算机显示器的标准传递函数由被称为sRGB的颜色空间规范来定义，当从纹理中读取值或将值写入颜色缓冲区时，大多数控制GPU的API可被设置为自动应用正确的sRGB转换[491]。如6.2.2节所述，mipmap生成时也将考虑sRGB编码，通过首先转换为线性值，然后执行插值，纹理值之间的双线性插值将正确工作。alpha混合是通过将存储的值解码回线性值，混合新值，然后对结果进行编码来正确完成的。**

**在渲染的最后阶段应用这种转换很重要，此时值被写入帧缓冲区以进行显示，如果在显示编码之后应用后处理，这些结果将在非线性值上计算，这通常是不正确的，并将经常导致伪像。显示编码可以被认为是一种压缩形式，它可以最好地保留值的感知效果[491]，认知它的一个好用法是我们用线性值来执行物理计算，当我们想要显示结果或访问可显示的图像(如颜色纹理)时，我们需要使用适当的编码或解码转换，将数据移到或移自其显示编码的形式。**

**如果你确实需要手动应用sRGB，这里有标准转换公式或一些简化版本可供使用，实际上显示是由每个颜色通道的位数控制的，例如对于消费者级别的显示器，该位数为8，并提供一组在[0, 255]范围内的级别，这里我们将显示编码的级别表示为[0.0, 1.0]范围，而忽略位数，线性值也在[0.0, 1.0]范围内，表示浮点数，我们用x表示这些线性值，用y表示存储在帧缓冲区中的非线性编码值。要将线性值转换为sRGB非线性编码值，我们应用sRGB显示传递函数的逆函数：**

|  |  |
| --- | --- |
|  | (5.30) |

**其中代表线性RGB三元组通道，该公式适用于每个通道，并且这三个生成的值会用来显示。如果你手动应用转换功能，请当心，错误的一个来源是使用编码颜色而不是其线性形式，另一个来源是解码或编码颜色两次。**

**这两个变换表达式的底部是一个简单的乘法，这是因为数字硬件需要使变换完全可逆[1431]。上面的表达式涉及到一个幂，几乎应用于整个[0.0, 1.0]范围的输入值x。考虑到偏移量和比例，此函数非常近似于一个更简单的公式[491]：**

|  |  |
| --- | --- |
|  | (5.31) |

**其中**，这个希腊字母（读音伽马）是名称“伽马校正”的来源。

**就像必须对计算值进行编码以便显示一样，静态或视频摄像机捕捉到的图像必须转换成线性值才能用于计算。你在显示器或电视上看到的任何颜色都有一些显示编码的RGB三元组，可以从屏幕捕获或颜色选择器中获取它们，这些值是以PNG，JPEG和GIF等文件格式存储的，这些是可以直接发送到帧缓冲区并在屏幕上显示而不用进行转换的格式，换句话说，你在屏幕上看到的所有内容都是被定义为显示编码的数据，在着色计算中使用这些颜色之前，我们必须将这种编码形式转换回线性值，从显示编码到线性值所需的sRGB转换是**

|  |  |
| --- | --- |
|  | (5.32) |

**其中代表标准化的显示道通值，即存储在图像或帧缓冲区中的，表示为[0.0, 1.0]范围内的值，这个解码函数与我们之前的sRGB公式相反，这意味着如果纹理被着色器访问且不更改的情况下输出，它将与处理之前有相同的显示，一如预期。解码函数与显示传递函数相同，因为存储在纹理中的值已经被编码以正确显示，相比转换为给出线性响应显示，我们转换为给出线性值。**

**更简单的伽马显示传递函数是公式5.31的逆函数：**

|  |  |
| --- | --- |
|  | (5.33) |

**有时你会看到一个更简单的转换对，特别是在手机和浏览器应用上[1666]：**

|  |  |
| --- | --- |
|  | (5.34) |

**也就是说，取线性值的平方根进行转换显示，然后仅用这个值乘以它自己求逆函数，虽然这是一个粗略的近似，但这种转换也比完全忽略这个问题要好。**

徽标

描述已自动生成

**图 5.40. 两个重叠的聚光灯照亮一个平面，左侧图像中，在将亮度值0.6和0.4相加后不执行伽马校正，有效地对非线性值执行加法操作，从而导致错误，注意左侧的光看起来比右侧的光要明亮得多，且重叠部分的亮度表现出不真实的明亮。在右图中，相加后的值将进行伽马校正，灯光本身会成比例地变亮，且它们在重叠的地方正确组合。**

**如果我们不注意伽马，较低的线性值将在屏幕上显示得太暗，一个相关的错误是，如果没有伽马校正，一些颜色的色调会偏移，假设我们的**，我们想从显示的像素中发出与计算值成线性比例的辐射率，意味着我们必须将线性值的幂设为（1/2.2），线性值0.1给出0.351，0.2给出0.481，而0.5给出0.730。如果未编码，按原样使用这些值将导致显示器发出比所需更少的辐射率，请注意，任何转换都始终不会更改0.0和1.0。在使用伽马校正之前，场景建模人员通常会人为地增加深色表面的颜色，并在逆变换中折叠。

忽略伽马校正的另一个问题是，对物理线性辐射值正确的阴影计算在非线性值上执行，图5.40给出了一个实例。

图片包含 游戏机, 家具, 桌子

描述已自动生成

**图 5.41. 在左侧，黑色（显示为灰色）背景上的白色三角形边缘覆盖了四个像素，并显示了真实的区域覆盖率。如果不执行伽马校正，则中间色调的变暗会导致边缘的感知失真，如右图所示。**

忽略伽马校正也会影响反走样边缘的质量，例如，假设一个三角形的边缘覆盖了四个屏幕网格单元（图5.41），三角形的标准化辐射率为1（白色），背景为0（黑色），从左到右，单元格的覆盖率为、、、，因此如果我们使用box滤波器，我们希望将像素的归一化线性辐射度表示为0.125、0.375、0.625和0.875，正确的方法是对线性值执行反走样操作，然后将编码函数应用于四个结果值。如果不这么做，像素表示的辐射将太暗，造成如图右侧所示的边缘的感知变形，这种伪像被称为扭绳（roping），因为边缘看起来有点像扭曲的绳子[167, 1265]，图5.42展示了这种效果。

图片包含 游戏机, 梳子, 猫, 体育

描述已自动生成

**图 5.42. 在左侧，一组反走样线条经过了伽玛校正；在中间，这组线条被部分校正；在右侧，没有线条被校正。（图片由Scott R. Nelson提供。）**

sRGB标准创建于1996年，已经成为大多数电脑显示器的标准，不过从那时起，显示技术不断发展，显示器更亮，且可以显示更广泛的颜色范围已经开发出来，颜色显示和亮度将在8.1.3节中讨论，用于高动态范围显示器的显示编码在第8.2.1节中介绍，Hart的文章[672]是关于高级显示的详细信息的一个特别详尽的来源。

进一步阅读和资源 Further Reading and Resources

Pharr等[1413]更深入地讨论了采样模式和反走样，Teschner的课程笔记[1758]展示了各种采样模式的生成方法，Drobot[382，383]融会贯通了前面对实时反走样的研究，解释了各种技术的属性和性能表现，我们可以在SIGGRAPH相关课程[829]的注释中找到有关各种形态学反走样方法的信息，Reshetov和Jimenez[1486]提供了游戏中使用的形态学和相关时间性反走样工作的最新回顾。

对于透明度研究，我们再次向感兴趣的读者推荐McGuire的陈述[1182]和Wyman的工作[1931]，Blinn的文章“像素是什么？”(What Is a Pixel?)[169]在讨论不同定义的同时，提供了对计算机图形学多个领域的绝佳浏览，Blinn的书籍“脏像素”(Dirty Pixels)和“符号，符号，符号”（Notation, Notation, Notation）[166，168]包含一些介绍性文章，涉及滤波和反走样，alpha，合成和伽马校正，Jimenez的演讲[836]给出了用于反走样的最先进技术的详细信息。

Gritz和d'Eon[607]对伽马校正问题有很好的总结，Poynton的书[1431]对各类传媒中的伽马校正以及其他与颜色相关的主题提供了详尽的介绍，Selan的白皮书[1602]是更新的资料，它解释了显示编码及其在电影行业中的用途，以及许多其他相关信息。

第6章 纹理化 Texturing

“所需要的是使渲染图像看起来正确。”——吉姆·布林

表面的纹理就是外观和感觉——只要想想油画的质感就明白了，在计算机图形学中，纹理化是获取一个表面，然后使用某些图片、函数或其他数据源在每个位置修改其外观的过程。举个例子，对于砖墙显示，一般是将砖墙的彩色图片应用在两个三角形组成的矩形上，而不是精确的显示它的几何形状，当矩形被查看时，彩色图片会显示在矩形所在的位置，除非观察者很靠近墙壁，否则缺少的几何细节是不会引起注意的。

然而，某些带纹理的砖墙可能会因为缺乏几何形状而无法令人信服，例如，如果砂浆应该是哑光的（matte），而砖块是光滑的（glossy），则观察者会注意到两种材料的粗糙度相同。为了产生更令人信服的体验，第二个图片纹理可以被应用到表面上，根据表面上的位置，这个纹理可以更改墙壁的粗糙度，而不是更改表面的颜色。现在，砖块和砂浆从图片纹理获取颜色，而从新纹理获取粗糙度。

现在观看者可能会看到所有砖块都是光滑的，而砂浆不是，每个砖块表面看起来都非常平坦，这看起来是不对的，因为砖的表面通常会有点不规则，通过应用凹凸贴图（bump mapping），砖块的着色法线就可以被改变，因此渲染时，它们看起来并不十分平滑，这种类型的纹理会摆动矩形原始表面法线的方向，以用于计算光照。

以低视角观察时，这种凹凸不平的错觉可能会被拆穿，砖块应突出在砂浆上方，以遮挡住它，即使从正面观察，砖块也应该将阴影投射到砂浆上，视差贴图（Parallax mapping）使用一张纹理让平面在渲染时似乎变形了，而视差遮蔽贴图（parallax occlusion mapping）将射线投射到高度场纹理上以提高真实感，位移贴图（Displacement mapping）通过修改形成模型的三角形高度来真正位移曲面，图6.1显示了带有颜色纹理和凹凸贴图的示例。

这些是可以通过纹理解决的问题类型的示例，使用了越来越复杂的算法。在本章中将详细介绍纹理化技术，首先介绍纹理化过程的一般框架；接下来，我们聚焦于使用图片对表面进行纹理化处理，因为这是实时工作中最流行的纹理化形式；接下来简要讨论了程序纹理，然后说明了使纹理影响表面的一些常用方法。

红色的鱼

低可信度描述已自动生成

**图 6.1. 纹理化，将颜色和凹凸贴图应用于此鱼以增加其视觉细节水平。（图片由Elinor Quittner提供。）**

6.1 纹理化管线 The Texturing Pipeline

纹理化是一种对表面材质的变化有效地建模与完善的技术，认知纹理化的一种方法是去思考单个着色像素在此过程中会发生什么。如上一章所述，我们通过考虑材质和灯光的颜色以及其他因素来计算着色，如果透明物体存在，其也能影响采样，我们通过修改着色方程中使用的值来进行纹理化，这些值的更改方式通常基于表面上的位置。因此，对于砖墙的示例，我们会基于表面位置，用砖墙图片中的相应颜色替换表面上任意点的颜色，图片纹理中的像素通常称为纹理像素（texels，简称纹素），以区别于屏幕上的像素，粗糙度纹理会修改粗糙度值，而凹凸纹理会更改着色法线的方向，因此它们每个都会改变着色方程的结果。

纹理化过程可以通过概括的纹理管线来描述，稍后我们将介绍许多术语，但请打起精神：管线的每个部分都将被详细介绍。

空间中的位置是纹理化过程的起点，该位置可以位于世界空间中，但通常位于模型的参照系中，因此随着模型的移动，纹理也随之移动，用Kershaw的术语[884]来描述，空间上此点已应用了投影函数（projector function），以便获得一组被称为纹理坐标（texture coordinates）的数值，这些数值将用于访问纹理，此过程称为映射（mapping），这引出了短语纹理映射（texture mapping），有时纹理图片本身就被称为纹理贴图（texture map），尽管严格来说这并非正确。

在这些新值可能被用于访问纹理之前，可以使用一个或多个匹配函数（corresponder function）将纹理坐标转换到纹理空间，这些纹理空间的位置用于从纹理中获取值，例如，它们可能是用于检索图像纹理中像素的数组索引，然后，这些取回的值可能会通过值转换函数（value transform function）再次进行转换，最后，这些新值将用于修改表面的某些属性，比如材质或着色法线。图6.2详细展示了这个应用单个纹理的过程，管线复杂的原因是每个步骤都为用户提供了有用的控制权，应该注意的是，并非所有步骤都需要始终保持激活。

图示

描述已自动生成

**图 6.2. 单个纹理的纹理管线概括。**

使用该管线，当具有砖墙纹理的三角形在其表面上生成一个样本时，具体发生的事情如图（见图6.3）。找到对象的局部参照系中的位置，假设是，然后将投影函数应用到此位置，正如世界地图是将三维对象投影到二维中一样，此处的投影函数通常将向量变为二维向量，此例中使用的投影函数等效于正交投影（第2.3.1节），其作用类似于幻灯片投影仪，将砖墙图片照在三角形的表面上。回到墙这边，其表面上的一个点可以被转换为范围从0到1之间的一对值，假设这里得到的值为，这些纹理坐标将用于查找图像中位于此处的颜色，假设我们的砖墙纹理分辨率为256×256，因此匹配函数会将分别乘以256，得到，舍弃小数点，在砖墙图片中找到的对应像素位置为，且它的颜色为，纹理颜色位于sRGB颜色空间中，因此如果要在着色方程式中使用该颜色，需要将其转换到线性空间，得到（第5.6节）。

图示

描述已自动生成

**图 6.3. 砖墙的管线。**

6.1.1 投影函数 The Projector Function

纹理处理的第一步是获取表面的位置并将其投影到纹理坐标空间中，通常是二维空间，建模包（Modeling packages）通常允许美术师定义每个顶点的坐标，这些可以从投影函数或网格展开算法（mesh unwrapping algorithms）来初始化，美工可以用与编辑顶点位置相同的方法来编辑坐标，投影函数通常通过将空间中的三维点转换为纹理坐标来工作，建模程序中常用的投影函数包括球形投影，圆柱形投影和平面投影[141, 884, 970]。

图片包含 游戏机

描述已自动生成

**图 6.4. 不同的纹理投影，从左到右显示了球形，圆柱形，平面和自然（u, v）投影，底部行表示每个投影应用于单个对象（其中没有自然投影）。**

其他的输入也可用于投影函数，比如可以使用表面法线来选择将六个平面投影方向中的哪个用于该表面，面部相接处的接缝会出现纹理匹配问题，Geiss[521, 522]讨论了一种将它们混合的技术，Tarini等人[1740]描述了多立方体贴图（polycube maps），其中一个模型被映射到一组立方体投影，用不同体积的空间映射到不同的立方体。

其他投影函数则根本不是投影，但它们是表面创建和细分的固有部分，例如参数化曲面具有一组自然的值集作为它们定义的一部分，见图6.4。纹理坐标也可以由各种不同的参数生成，例如视图方向，表面温度或其他任何你能想到的参数，投影函数的目标是生成纹理坐标，推导出位置的函数只是其中一种方法。

非交互式渲染器通常将这些投影函数称为渲染过程本身的一部分，单个投影函数可能就足以满足整个模型的需要，但美术人员通常必须使用工具来划分模型并分别应用各种投影函数[1345]，参见图6.5。

图示

描述已自动生成

**图 6.5. 各种纹理投影是如何被应用在单个模型上，盒式映射（Box mapping）由六个平面映射组成，每个盒子面都有一个映射。（图片由Tito Pagn提供。）**

在实时渲染中，通常在建模阶段应用投影函数，并将投影结果存储在顶点上，然而并非总是如此，有时在顶点或像素着色器中应用投影函数也是有好处的，这么做可以提高精度，并有助于启用各种效果，包括动画（第6.4节）。一些渲染方法，比如环境映射（environment mapping）（第10.4节），具有自己的专有的逐像素进行计算的投影函数。

球形投影（图6.4的左侧）将点投射到以某个点为中心的假想球体上，此投影与Blinn和Newell的环境映射方案（第10.4.1节）中使用的投影相同，因此公式10.30描述了此函数，这种投影方法也遭遇了本节所述的相同的顶点插值问题。

圆柱形投影计算的u纹理坐标与球形投影相同，而v纹理坐标作为沿圆柱轴的距离计算，此投影对于具有自然轴的对象，例如旋转表面（surfaces of revolution），是很有用的，此外当表面几乎垂直于圆柱体的轴时，会发生变形。

平面投影就像X射线一样，沿着一个方向进行平行投影，并将纹理应用于所有表面，它使用正交投影（第4.7.1节），举个例子，这种类型的投影可用于贴花（decals）（第20.2节）。

由于投影方向边缘的曲面存在严重的变形，因此美术人员通常需要手动将模型分解为近似平面的部分。有一些工具可通过展开网格，或是创建一组接近最佳的平面投影来帮助最大程度地减少变形，也可以通过其他方式帮助此过程。我们的目标是让每个多边形在纹理区域中占有更公平的份额，同时也保持尽可能多的网格连接，连接性很重要，因为采样伪像（sampling artifacts）可以沿着纹理的各个单独部分接触的边缘出现，具有良好展开效果的网格也可以简化美术人员的工作[970, 1345]，16.2.1节讨论了纹理变形如何对渲染产生不利影响，图6.6展示了用于创建图6.5中的雕像的工作空间，这种展开过程是网格参数化（mesh parameterization）这一较大研究领域的一个方面，有兴趣的读者可以参考Hormann等人的SIGGRAPH课程笔记[774]。

卡通人物

中度可信度描述已自动生成

**图 6.6. 雕像模型的几个较小的纹理，保存在两个较大的纹理中，右图显示了三角形网格如何展开并显示在纹理上以帮助其创建。（图片由Tito Pagn提供。）**

纹理坐标空间并不总是二维平面，有时它是三维体积，在这种情况下，纹理坐标表示为三元向量，其中w为沿投影方向的深度，其他系统最多使用四个坐标，通常指定为[885]，q用作齐次坐标中的第四个值，它的作用类似于电影或幻灯片的投影仪，投影纹理的大小随距离而增加。举个例子，对于将装饰性聚光图案（称为gobo）投影到舞台或其他表面上来说，它是很有用的[1597]。

纹理坐标空间的另一种重要类型是方向型，其空间中的每个点都可以通过输入方向访问，可视化这种空间的一种方法是将其作为单位球面上的点，每个点的法线表示用于访问该位置的纹理的方向，使用方向参数化的最常见纹理类型是立方体贴图（cube map）（第6.2.4节）。

另外还值得注意的是，一维纹理图像和函数也有其用途，例如，在地形模型上，可以通过其高度确定颜色，比如低地是绿色，山峰是白色，线条也可以进行纹理化，其中一种用法是将雨水渲染为一组带有半透明图像的长线，此外，这种纹理对于从一个值转换为另一值，即作为查找表，也是有用的。

由于可以将多个纹理应用于一个表面，因此可能需要定义多组纹理坐标，如果是应用坐标值的话，思路是相同的：这些纹理坐标在表面上插值并用于检索纹理值，然而在插值之前，这些纹理坐标会由匹配函数转换。

6.1.2 匹配函数 The Corresponder Function

匹配函数将纹理坐标转换为纹理空间位置，它们提供了将纹理应用于表面的灵活性，匹配函数的一个应用案例就是使用API选择现有纹理的一部分进行显示，只有该子图像才会被后续操作流程使用。

匹配函数的另一种类型是矩阵变换，可以将其应用于顶点或像素着色器，这样就可以在表面上平移，旋转，缩放，剪切或投影纹理，正如第4.1.5节所述。转换的顺序很重要，但令人惊讶的是，纹理转换的顺序必须与预期的顺序相反，这是因为纹理变换实际上影响确定被查看图像位置的空间，图像本身并不是要转换的对象，定义图像位置的空间被更改。

还有另一类匹配函数控制应用图像的方式，我们知道图像将出现在处于范围内的表面上，但是超出此范围会发生什么？匹配函数确定其具体行为。在OpenGL中，这种类型的匹配函数被称为“包装模式”（wrapping mode）；在DirectX中，它被称为“纹理寻址模式”（texture addressing mode），这种类型的常见匹配函数有：

* 包裹（wrap）（DirectX），重复（repeat）（OpenGL），或图块（tile）——图像在整个表面上重复，算法上将丢弃纹理坐标的整数部分。此函数对于使材质的图像重复并覆盖表面很有用，并且通常是默认设置。
* 镜像（mirror）——图像在整个表面上重复，但在其他每个重复图像上都被镜像（翻转），例如，图像通常从0到1正常出现，然后在1和2之间反转，然后在2和3之间正常，然后反转，依此类推。这可以让纹理边缘带有一定连续性。
* 夹取（clamp）（DirectX）或夹取到边缘（clamp to edge）（OpenGL）——超出[0, 1]范围的值都将被clamp到该范围，这会导致图像纹理的边缘不断重复，此函数对于在纹理的边缘附近采用双线性插值时，避免意外地从纹理的对边进行采样[885]很有用。
* 边框（border）（DirectX）或夹取到边框（clamp to border）（OpenGL）——[0, 1]以外的纹理坐标由单独定义的边框颜色渲染，例如，由于纹理的边缘会与边框颜色平滑融合，这种函数可以很好地将贴花渲染到单色表面上。

图片包含 游戏机

描述已自动生成

**图 6.7. 图像纹理的包裹、镜像、夹取、边框函数的效果。**

见图6.7，可以为每个纹理轴分配不同的这些匹配函数，例如纹理可以沿u轴重复同时clamp在v轴上，在DirectX中，还有一个单次镜像模式（mirror once mode），该模式沿着纹理坐标的零值镜像一次纹理，然后clamp，这对于对称贴花来说很有用。

重复平铺纹理（Repeated tiling of a texture）是向场景添加更多视觉细节的便宜方法，但是，这种技术在重复大约三遍纹理后通常看起来并不可信，因为眼睛会挑选出重复图案。避免此类周期性问题（periodicity problem）的常见解决方案是将纹理值与另一个非平铺纹理组合，这种方法可以被大大地扩展，如在Andersson[40]描述的商业地形绘制系统中所描述的那样。在该系统中，会基于地形类型，高度，坡度和其他因素组合多个纹理，纹理图像还会与场景中放置几何模型的位置相关联，例如灌木和岩石。

避免周期性的另一种选择是使用着色器程序来实现专门的匹配函数，随机地重新组合纹理图案或图块，王图块（Wang tiles）是这种方法的一个例子，一个王图块集是一小组具有匹配边缘的正方形图块，在纹理化过程中[1860]会随机选择图块，Lefebvre和Neyret[1016]使用有依赖关系的纹理读取和制表，以避图案重复，实现了相似类型的匹配函数。

应用的最后一个匹配函数边框（border）是隐式的，它是从图像的大小得出的，一个纹理通常在u和v的范围内应用，如砖墙案例所示，通过在该范围内将纹理坐标乘以图像的分辨率，可以获取像素位置。能够在范围内指定值的优点在于，可以交换具有不同分辨率的图像纹理，而不必修改存储在模型顶点处的值。

6.1.3 纹理值 Texture Values

在使用匹配函数生成纹理空间坐标之后，会使用坐标去获取纹理值。对于图像纹理，这是通过访问纹理，并从图像中检索纹素信息来完成的，此过程将在6.2节中广泛处理，图像纹理构成了实时渲染中绝大多数使用的纹理，但程序化函数（procedural functions）也是可以使用的，在程序纹理化的情况下，从纹理空间位置获取纹理值的过程不涉及存储器查找，而是函数的计算，程序纹理化将在6.3节中进一步描述。

最直接的纹理值是用于替换或修改表面颜色的RGB三元组，类似的可以返回单个灰度值，如5.5节所述，另一种可返回的数据是**RGB****α**，**α（**alpha）值通常是颜色的不透明度，决定了颜色可能影响像素的程度，也就是说，任何其他值都可以被存储，比如表面粗糙度。还有许多其他类型的数据可以存储在图像纹理中，我们将在详细讨论凹凸映射时看到这一点（第6.7节）。

从纹理返回的值可以在使用前进行转换，这些转换可以在着色器程序中执行，一个常见的例子是将数据从无符号范围（0.0到1.0）重映射到有符号范围（-1.0到1.0），用于将着色法线存储在颜色纹理中。

6.2 图像纹理化 Image Texturing

图像纹理化中，一个二维图像会有效地粘贴到一个或多个三角形的表面上，我们已经完成了计算纹理空间位置的过程，现在，我们将解决在给定位置信息后，从图像纹理获取纹理值的相关问题和算法。在本章的其余部分中，图像纹理将简称为纹理，另外，当我们提到像素单元（pixel’s cell）时，我们指的是围绕像素的屏幕网格单元，如第5.4.1节所述，像素实际上是显示的颜色值，它可能（且应该，为了更好的质量）受到与其关联的网格单元外部的样本的影响。

在本节中，我们特别关注快速采样和纹理图像的滤波方法，第5.4.2节讨论了反走样的问题，特别是在渲染对象的边缘时，纹理也可能存在采样问题，但它们发生在被渲染的三角形内部。

像素着色器通过将纹理坐标值传递给诸如texture2D之类的调用来访问纹理，这些值在纹理坐标中，由匹配函数映射到范围，GPU负责将该值转换为纹素坐标，不同API中的纹理坐标系之间有两个主要区别，在DirectX中，纹理的左上角为，右下角为，这与多种存储数据的图像类型相匹配，顶行是文件中的第一行。在OpenGL中，纹素位于左下角，是DirectX的y轴翻转。像素具有整数坐标，但是我们经常想要访问像素之间的位置，并在它们之间进行混合，这引出了一个问题：像素中心的浮点坐标是什么，Heckbert[692]讨论了为何有两种可能的系统：截断（truncating）和四舍五入（rounding）。DirectX 9将每个中心定义为——使用四舍五入，这个系统有些令人困惑，因为作为左上角像素的左上角，DirectX的原点的值为。DirectX 10则改变为OpenGL的系统，纹理元素的中心具有小数值——即截断，或者更准确地讲，是向下取整（flooring），小数部分被丢弃，向下取整是一个更自然的系统，可以很好地映射到语言，例如像素为u坐标定义了5.0到6.0的范围，为v定义了9.0到10.0的范围。

这里一个值得解释的术语是依赖纹理读取（dependent texture read），它有两个定义，第一种特别适用于移动设备，当通过texture2D或类似方法访问纹理时，当像素着色器计算纹理坐标而不是使用从顶点着色器传入的未修改纹理坐标时，就会产生依赖纹理读取[66]。请注意，这意味着对传入的纹理坐标的任何更改，甚至包括交换u和v值之类的简单操作。那些不支持OpenGL ES 3.0的较早的移动GPU，着色器没有依赖纹理读取时可以更有效地运行，因为可以预先获取纹理像素数据。该术语的另一个较老的定义对于早期的台式机GPU特别重要，当一个纹理的坐标取决于某些先前纹理的值的结果时，就会发生依赖纹理读取。例如，一种纹理可能会更改着色法线，进而改变用于访问立方体贴图的坐标，这种功能在早期的GPU上受到限制甚至不存在，如今这类读取可能会影响性能，具体取决于批处理中计算的像素数量以及其他因素。更多信息参见第23.8节。

GPU中使用的纹理图像大小通常为纹素，其中m和n为非负整数，这些被称为二次幂（power-of-two, POT）纹理，现代GPU可以处理任意大小的非二次幂（non-power-of-two, NPOT）纹理，它允许生成的图像被视为纹理，但是，某些较旧的移动GPU可能不支持NPOT纹理的mipmapping（第6.2.2节）。图形加速器对纹理大小有不同的上限，例如，DirectX 12最多允许个纹理像素。

假设我们有一个256×256纹素的纹理，我们想把它用作一个正方形的纹理，只要投影在屏幕上的正方形与纹理的大小大致相同，正方形上的纹理看起来就几乎与原始图像相同，但是，如果投影的正方形覆盖的像素是原始图像的十倍（称为放大，magnification），或者投影的正方形只覆盖屏幕的一小部分（缩小，minification），会发生什么呢？答案是，这取决于你决定对这两个不同的情况使用哪种采样和滤波方法。

本章中讨论的图像采样和滤波方法应用于从每个纹理中读取的值，但是，期望的结果是防止在最终渲染的图像中出现走样，这在理论上需要对最终像素的颜色进行采样和滤波，这里的区别是滤波着色方程的输入，还是滤波它的输出，只要输入和输出是线性相关的（对于颜色之类的输入来说就是如此），那滤波单个纹理值等价于最终颜色的滤波，然而许多存储在纹理中的着色器输入值，如表面法线和粗糙度值，与输出有非线性关系。标准的纹理滤波方法可能不能很好地处理这些纹理，导致走样，用于滤波这些纹理的改进方法将在9.13节中讨论。

6.2.1 放大 Magnification

在图6.8中，将大小为48×48纹理像素的纹理贴到一个正方形上，相对于纹理大小而言，该正方形看起来相当紧密（也就是很糊），因此基础图形系统必须放大该纹理。放大最常见的滤波技术是最近邻（实际滤波器称为box滤波器，参见第5.4.1节，nearest neighbor）和双线性插值（bilinear interpolation），还有三次卷积（cubic convolution），其使用4×4或5×5纹素阵列的加权和，这使得放大质量更高，尽管对三次卷积（也称为双三次插值，bicubic interpolation）的原生硬件支持目前并不普遍，但它可以在着色器程序中执行。

在图6.8的左侧，我们使用的是最近邻法，这种放大技术的一个特点是，单个的纹理可能会变得明显，这种效果被称为像素化，因为该方法在放大时取用离每个像素中心最近的纹理像素值，从而导致块状的外观，虽然这种方法的质量有时很差，但它只需要每个像素提取一个纹理像素值。

人不同表情的照片

描述已自动生成

**图 6.8.48×48图像放大率为320×320像素，左边：最近邻滤波，其中每像素选择最近的纹理像素。中间：使用四个最近的纹理像素的加权平均值进行双线性滤波。右边：使用5×5最近的纹理像素的加权平均值进行三次滤波。**

在中间的图像中，使用了双线性插值（有时称为线性插值），对于每个像素，这种滤波方法找到四个相邻的纹素，并在二维上进行线性插值，以找到像素的混合值，其结果更加模糊，使用最近邻法产生的锯齿也消失了很多。这里做个实验，尝试看向左图的同时眯着眼睛，能看到更多的脸部，这和低通滤波的效果差不多。

回到前述的砖块纹理示例：在不丢弃小数的情况下，我们得到了。我们在这里使用OpenGL的左下原点纹理像素坐标系，因为它与标准笛卡尔坐标系相匹配。我们的目标是在四个最接近的纹素之间插值，使用它们的纹素中心定义一个纹素大小的坐标系统，参见图6.9，为了找到四个最近的像素，我们从我们的样本位置减去像素中心分数，得到。丢弃小数部分，四个最接近的像素范围从到，在我们的示例中，小数部分是样本相对于由四个纹理像素中心形成的坐标系的位置，我们将此位置表示为。

图示, 工程绘图

描述已自动生成

**图 6.9. 双线性插值，涉及的四个纹理像素由左侧的四个正方形表示，纹理像素中心为蓝色，右边是由四个纹理像素的中心形成的坐标系。**

将纹理访问函数（texture access function）定义为，其中和为整数，并返回纹素的颜色，任何位置的双线性插值颜色的计算可分为两步。首先，底部的纹素和，将被水平插值（使用），对顶部的两个纹素和也做同样的操作。对底部纹素，我们得到（图6.9中的底部绿色圆圈），对顶部纹素，我们得到（上方的绿色圆圈），然后将这两个值垂直插值（使用），因此在处的双线性插值颜色**b**为

|  |  |
| --- | --- |
|  | (6.1) |

直观地，接近我们样本位置的纹理像素将对最终值产生更大的影响，这确实是我们在公式中看到的。在处的右上纹理像素具有对的影响，请注意对称性：右上角的影响等同于由左下角和采样点组成的矩形的面积。回到我们的示例，这意味着从该纹理像素检索的值将乘以0.42×0.74，即0.3108，从该纹理像素的顺时针方向，其他乘数为0.42×0.26，0.58×0.26和0.58×0.74，这四个权重之和为1.0。

一种解决放大倍率模糊的常见方法是使用细节纹理（detail textures），这些纹理代表了精细的表面细节，从手机上的划痕到地形上的灌木丛，这样的细节作为单独的纹理以不同的比例覆盖在放大的纹理上，细节纹理的高频重复图案与低频放大的纹理相结合，具有类似于使用单个高分辨率纹理的视觉效果。

双线性插值在两个方向上进行线性插值，但是，线性插值并不是我们想要的，假设纹理由棋盘图案中的黑白像素组成，使用双线性插值会得出整个纹理上变化的灰度样本，通过重新映射，例如，所有低于0.4的灰度都是黑色，所有高于0.6的灰度都是白色，并且中间的灰度都被拉伸以填充间隙，同时还使纹理像素之间进行一些混合，这样就可以让纹理看起来更像是棋盘，见图6.10。

图片包含 游戏机, 厨具, 过滤网

描述已自动生成

**图 6.10. 最近邻，双线性插值，以及通过重新映射介于两者之间的情况，使用相同的2×2棋盘纹理。注意由于纹理和图像网格不完全匹配，最近邻采样给出的正方形大小略有不同。**

使用更高分辨率的纹理将具有相似的效果，例如，假设每个方格由4×4像素组成，而不是1×1。在每个方格的中心周围，插值的颜色将完全是黑色或白色。

在图6.8的右侧，采用了双三次滤波器，剩余的块状现象已被大大消除，需要注意的是，双三次滤波器比双线性滤波器更昂贵，但是，许多高阶滤波器可以表示为重复线性插值（repeated linear interpolations）[1518]（见第17.1.1节），因此可以通过多次查找来利用纹理单元中用于线性插值的GPU硬件。

如果认为双三次滤波器太昂贵，Quílez[1451]提出了一种简单技术，使用平滑曲线在一组2×2纹素之间进行插值，我们首先描述曲线，然后介绍技术，两条常用的曲线是平滑阶跃曲线（smoothstep curve）和五阶曲线（quintic curve）[1372]：

|  |  |
| --- | --- |
|  | (6.2) |

当您想要平滑地从一个值插入到另一个值时，这些方法非常有用，平滑阶跃曲线具有的性质，且它在0到1之间是平滑的，五阶曲线也有同样的性质，但同时，即曲线起点和终点的二阶导数也为0，两种曲线如图6.11所示。

一些文字和图案

低可信度描述已自动生成

**图 6.11. 平滑阶跃曲线（左）和五阶曲线（右）**

该技术首先通过将样本乘以纹理尺寸并加上0.5来计算（与公式6.1和图6.9中所示相同），保留整数部分以备后来使用，并将分数存储在和中，它们在范围内，然后将变换为，仍在范围内，最后减去0.5，再将整数部分加回来，然后将得到的坐标除以纹理宽度，对也做同样的操作。此时，新的纹理坐标与GPU提供的双线性插值查找一起使用，注意，这个方法将在每个纹素处达到稳定状态，这意味着如果纹素位于RGB空间的一个平面上，那么这种插值会得到一个平滑的，但仍然呈阶梯状的外观，这可能并不总是我们所希望的，见图6.12。

图片包含 游戏机, 物体, 衣架, 钟表

描述已自动生成

**图 6.12. 放大一维纹理的四种不同方法，橙色的圆圈表示纹理的中心以及纹理的值(高度)。从左到右：最近邻，线性，以及在每对相邻纹理之间使用五阶曲线，和使用三次插值**

6.2.2 缩小 Minification

当一个纹理被缩小时，多个纹素可能会覆盖一个像素的单元格，如图6.13所示。为使每个像素获得正确的颜色值，你应该整合影响像素的所有纹素的效果，但是，很难精确地确定一个特定像素附近的所有纹素的确切影响，实际上，要完美地实时做到这一点是不可能的。

围栏里的雪地上滑雪

中度可信度描述已自动生成

**图 6.13. 缩小：通过一排像素单元，看一个棋盘纹理的正方形，大致显示若干像素如何作用于每个像素。**

由于这种限制，在GPU上使用了几种不同的方法，一种方法是使用最近邻，它的工作原理与相应的放大滤波器完全相同，即选择在像素单元中心可见的纹素，该滤波器可能会导致严重的走样问题。在图6.14中，最上面的图使用了最近邻方法，朝向地平线时会出现伪像，因为我们只选择影响像素的众多纹素中的一个来表示表面，当表面相对于观看者移动时，这样的伪像甚至更明显，这是所谓的时间走样（temporal aliasing）的一种表现形式。

图片包含 户外, 建筑, 笔记本, 屋顶

描述已自动生成

**图 6.14. 上面的图像是用点采样（最近邻）渲染的，中间是mipmapping渲染的，底部是求和区域表（summed area tables）渲染的。**

另一种常用的滤波器是双线性插值，它的工作原理也与放大滤波器完全相同，在缩小方面，这种滤波器只比最近邻方法好一点点，它混合了4个纹素而不是只用一个，但是当一个像素受到超过4个纹素的影响时，滤波器会迅速失效并产生走样。

更好的解决方案是可能的，如5.4.1节所述，走样问题可以通过采样和滤波技术来解决。纹理的信号频率取决于它的纹理在屏幕上的间距，由于奈奎斯特极限，我们需要确保纹理的信号频率不大于样本频率的一半。例如，假设图像是由交替的黑白线组成的，相隔一个纹理，然后波长为两个像素宽（从黑线到黑线），因此频率为，为了在屏幕上正确地显示这个纹理，频率必须至少为，即至少每纹素一个像素，因此，对于一般的纹理，每个像素最多应该有一个纹素，以避免走样。

为了达到这一目的，像素的采样频率必须增加或纹理频率必须降低，前一章讨论的反走样方法提供了提高像素采样率的方法，然而，这只能有限地增加采样频率，为了更充分地解决这个问题，各种纹理缩小算法已经被开发出来。

所有纹理抗锯齿算法背后的基本思想是相同的：对纹理进行预处理并创建数据结构，以帮助计算一组纹理在一个像素上的效果的快速近似值。对于实时渲染，这些算法具有使用固定的时间和资源以执行的特点，用这种方式，每个像素取一个固定数量的样本，并结合起来计算（可能是巨量的）大量纹素的效果。

**多级渐远纹理 Mipmapping**

最流行的纹理反走样方法叫做多级渐远纹理（mipmapping）[1889]，目前生产的所有图形加速器上都以某种形式实现了它，“Mip”代表拉丁文multum in parvo，意思是“小地方有很多东西”——这是一个很好的名字，用来描述一个过程，在这个过程中，原始纹理被反复滤波成更小的图像。

当使用mipmapping缩小滤波器时，在进行实际渲染之前，使用一组较小版本的纹理来扩展原始纹理。纹理（在级别零）被下采样到原始区域的四分之一，其中每个新的纹素值通常计算为原始纹理中四个相邻纹素的平均值，新的一级纹理有时被称为原始纹理的子纹理（subtexture），递归地执行缩减，直到纹理的一个或两个维度等于一个纹素，这个过程如图6.15所示，作为一个整体的图像集通常被称为一个mipmap链（mipmap chain）。

在屋顶上

低可信度描述已自动生成

**图 6.15. mipmap是通过以原始图像（第0层）作为金字塔底部，将其中的每个2×2区域平均为下一个级别的纹素值而形成的，垂直轴是第三纹理坐标d。在该图中，d不是线性的，它用于衡量一个样本用来插值的两个纹理级别。**

形成高质量mipmap的两个重要元素是良好的滤波和伽马校正，形成mipmap级别的常用方法是将每组2×2纹素取平均值，得到mip纹素值，这时使用的滤波器是一个box滤波器，它可能是最坏的滤波器之一，这可能导致较差的质量，因为它不必要的模糊低频带来的影响，同时保留一些导致走样的高频[172]，最好使用高斯、Lanczos、Kaiser或类似的滤波器。有快速、免费的源代码可用于任务[172, 1592]，并且某些API在GPU身上支持更好的滤波，在纹理的边缘附近滤波时，必须注意纹理是重复的还是单一的副本。

对于在非线性空间中编码的纹理（如大多数颜色纹理），在滤波时忽略伽马校正将修改mipmap级别的感知亮度[173, 607]，使用未校正的mipmap时，当你离物体更远，物体整体看起来会更暗，对比度和细节也会受到影响，因此将这种纹理从sRGB转换为线性空间是很重要的（第5.6节），在该空间中执行所有的mipmap滤波，并将最终结果转换回sRGB空间中存储。大多数API都支持sRGB纹理，因此可以在线性空间中正确生成mipmap并将结果存储在sRGB中，当访问sRGB纹理时，它们的值首先被转换到线性空间，以便正确地执行放大和缩小。

正如前面提到的，一些纹理与最终的着色颜色有基本的非线性关系，这给一般的滤波带来了一个问题，但mipmap生成对这个问题特别敏感，因为要滤波数百或数千个像素。通常需要专门的mipmap生成方法来获得最佳结果，这些方法在第9.13节中详细介绍。

电脑萤幕画面

描述已自动生成

**图 6.16. 左边是一个正方形像素单元格及其纹理视图，右边是像素单元格在纹理本身上的投影。**

纹理化时访问这个结构的基本过程很简单，一个屏幕像素在纹理上围出来一个区域，当像素的区域投影到纹理上时（图6.16），它包含一个或多个纹理像素，严格来说使用像素的单元边界并不是正确的，但在这里使用它以简化演示，单元格外的纹素可以影响像素的颜色，参见第5.4.1节，我们的目标是大致确定纹理对像素的影响程度。有两种常用方法来计算d（OpenGL称之为λ，它也被称为纹理细节级别（texture level of detail）），一种方法是使用像素单元形成的四边形的长边来近似像素的覆盖范围[1889]，另一种方法是用，，和这四个微分的最大绝对值作为度量[901, 1411]，每个微分是纹理坐标相对于屏幕轴的变化量的度量，例如是一个像素的u纹理值沿x屏幕轴的变化量，参见Williams的原创文章[1889]，或Flavell[473]或Pharr[1411]的文章了解更多关于这些方程的内容，McCormack等人[1160]讨论了用最大绝对值法引入走样的问题，并提出了一个替代公式，Ewins等人[454]分析了几种质量相当的算法的硬件开销。

使用Shader Model 3.0或更高版本的像素着色器程序可以使用这些梯度值，因为它们是基于相邻像素值之间的差异，它们在受动态流控制影响的像素着色器部分中不可访问（第3.8节），对于在这样的部分中（例如，在循环中）执行的纹理读取，导数必须被提前计算。注意，因为顶点着色器不能访问梯度信息，梯度或细节级别需要在顶点着色器本身中计算，并在使用顶点纹理时提供给GPU。

计算坐标d的目的是确定沿mipmap的金字塔轴采样的位置，见图6.15，我们的目标是像素与纹理的比例至少为1:1，以达到奈奎斯特率，这里的重要原则是，随着像素单元包含更多的纹素以及d的增加，将访问一个更小，更模糊的纹理版本，三元组用于访问mipmap，值d与纹理级别类似，但d不是一个整数值，而是等级之间距离的小数值，高于d位置的纹理级别和低于d位置的纹理级别被采样，位置用于从这两个纹理级别中逐个检索双线性插值的样本，然后根据从每个纹理级别到d的距离，对得到的样本进行线性插值，这整个过程被称为三线性插值（trilinear interpolation），每像素执行一次。

d坐标上的一个用户控制参数是细节偏差级别（level of detail bias，简称LOD偏差），这是一个增加到d的值，因此它影响纹理的相对感知清晰度，如果我们向上移动金字塔到顶（增加d），则纹理看起来会很模糊。对于任意给定的纹理，良好的LOD偏差将随图像类型和使用方式而变化，例如一开始就有些模糊的图像可以使用负偏差，而滤波不良（走样）的合成图像用于纹理可以使用正偏差，可以为整个纹理或像素着色器中每个像素指定偏差，用户可以提供d坐标或用于计算d的导数，以提供更好的控制。

mipmapping的好处在于，它不是试图单独地将影响一个像素的所有纹素相加，而是访问和插值了预先组合的纹素集，这个过程需要花费固定的时间，无论缩小的量是多少。然而mipmapping有几个缺陷[473]，一个主要缺陷是过度模糊（overblurring），想象一个像素单元，它在u方向上覆盖了大量的纹素，而在v方向上只覆盖了少量的纹素，这种情况通常发生在观察者几乎侧着沿纹理表面看时。实际上，有可能需要沿着纹理的一个轴缩小和沿着另一个轴放大，访问mipmap的效果是纹理上的正方形区域被检索，而不能检索矩形区域，为了避免走样，我们选择纹理上像素单元近似覆盖的最大量，这导致检索的样本通常是相对模糊的，这种效果可以在图6.14的mipmap图中看到，右边的线条向远处移动时，显示出过度模糊。

**求和区域表 Summed-Area Table**

另一种避免过度模糊的方法是求和区域（summed-area table, SAT）[312]，要使用此方法，首先创建一个与纹理大小相同但包含更高精度的颜色存储的数组（例如，红色、绿色和蓝色分别为16位或更多），在这个数组中的每个位置，必须计算所有对应纹理的纹素的总和，并存储在由该位置和纹素（原点）组成的矩形中。纹理化过程中，像素单元格在纹理上的投影被一个矩形限制，然后访问求和区域表以确定该矩形的平均颜色，该颜色作为像素的纹理颜色传回，使用矩形的纹理坐标计算平均值的过程如图6.17所示，它通过公式6.3来实现：

|  |  |
| --- | --- |
|  | (6.3) |

在这里，x和y是矩形的纹素坐标，而是该纹素的面积总和值，这个方程的原理是取从右上角到原点的整个面积和，然后通过减去相邻角的贡献来减去区域A和B，区域C已经被减去了两次，所以它被添加回左下角，注意是C区域的右上角，即是边界框的左下角。

使用求和区域表的结果如图6.14所示，接近右边缘地平线的线更加清晰，但中间对角交叉的线仍然过于模糊，问题在于，当沿着纹理的对角线观察时，会生成一个大矩形，其中许多纹素不在被计算的像素附近，例如，想象一个长而细的矩形表示像素单元的反向投影，按对角线横贯整个纹理（如图6.17所示），整个纹理矩形的平均值将被返回，而不仅仅是像素单元内的平均值。

图示

描述已自动生成

**图 6.17. 像素单元反向投影到纹理上，并以矩形为界，矩形的四个角用于访问求和区域表。**

求和区域表是所谓各向异性滤波（anisotropic filtering）算法的一个例子[691]，这样的算法在非正方形区域上检索纹理像素值，不过SAT主要在水平和垂直方向上能够最有效地做到这一点。还需要注意的是，对于大小为16×16或更小的纹理，求和区域表至少需要两倍的内存，而对于更大的纹理则需要更高的精度。

求和区域表以合理的整体内存成本提供更高的质量，它可以在现代GPU上实现[585]，改进的滤波对高级渲染技术的质量至关重要，例如Hensley等人[718, 719]提供了一种有效的实现，展示了求和区域采样如何改善光滑反射。还可以通过SAT改进其他使用区域采样的算法，例如景深（depth of field）[585, 719]、阴影图（shadow maps）[988]和模糊反射（blurry reflections）[718]。

**无约束各向异性滤波 Unconstrained Anisotropic Filtering**

对于当前的图形硬件，进一步改进纹理滤波的最常用方法是重用现有的mipmap硬件，其基本思想是对像素单元反向投影，纹理上的这个四边形被采样几次，然后样本被组合。如上所述，每个mipmap样本都有一个位置和一个与之关联的正方形区域，该算法不是使用一个单一的mipmap样本来近似这个四边形的覆盖范围，而是使用几个正方形来覆盖四边形。可以用四边形中较短的边来确定d（不像在mipmapping中，其通常使用较长的边），这使得每个mipmap样本的平均面积更小（因此更不模糊），四边形的较长的边被用来创建一条平行于较长的边并穿过四边形中间的各向异性线（ine of anisotropy），当各向异性的量在1:1和2:1之间时，沿这条线取两个样本（见图6.18），当各向异性比率越高时，沿轴取的样本越多。

图示

描述已自动生成

**图 6.18. 各向异性滤波，像素单元的反向投影创建一个四边形，在较长的边之间形成了一条各向异性线。**

这种方案允许各向异性线向任何方向延申，因此没有求和区域表的限制，它也不需要比mipmap更多的纹理内存，因为它使用mipmap算法来进行采样，各向异性滤波的一个例子如图6.19所示。

在屋顶上

中度可信度描述已自动生成

**图 6.19. Mipmap与各向异性滤波对比，左侧完成了三线性mipmapping，右侧进行了16：1各向异性滤波。朝向水平线方向，各向异性滤波提供了更清晰的结果，走样最小。（图片来自three.js示例webgl materials texture anisotropy [218]。）**

这种沿轴采样的想法最早是由Schilling等人提出的，同时提出的还有他们的Texram动态存储设备[1564]，Barkans描述了该算法在Talisman系统中的应用[103]，McCormack等人[1161]提出了一个名为猫（Feline）的类似系统，Texram的最初方案是对沿着各向异性轴的样本（也称为探针，probes）给予相等的权重，Talisman给予轴两端相对的两个探针一半权值，Feline使用高斯滤波内核对探针集合进行加权。这些算法接近高质量的软件采样算法，如椭圆加权平均（Elliptical Weighted Average, EWA）滤波器，其将像素的影响区域转换为纹理上的椭圆，并通过滤波器内核对椭圆内的纹素进行加权[691]，Mavridis和Papaioannou提出了几种在GPU上使用着色器代码实现EWA滤波的方法[1143]。

6.2.3 体积纹理 Volume Textures

图像纹理的直接延伸是由（或值）访问的三维图像数据，例如，可以将医学成像数据生成为三维网格，通过在网格中移动多边形，可以查看这些数据的二维切片，一个相关的想法是用这种形式表示体积灯光，一个表面上一个点的亮度是通过找到它在这个体积内的位置值，结合光的方向来得到的。

大多数GPU支持体积纹理的mipmapping，因为在体积纹理的单个mipmap级别的滤波涉及到三线性插值，所以mipmap级别之间的滤波需要四线性插值（quadrilinear interpolation）。因为这涉及到来自16个纹素的结果的平均，可能会产生精度问题，这可以通过使用更高精度的体积纹理来解决，Sigg和Hadwiger[1638]讨论了这个以及其他与体积纹理相关的问题，并提供了执行滤波和其他操作的有效方法。

尽管体积纹理对存储的要求明显更高，滤波成本也更高，但它们确实有一些独特的优势，为三维网格寻找一个好的二维参数化的复杂过程可以被跳过，因为三维位置可以直接用作纹理坐标，这避免了二维参数化通常会出现的失真和接缝问题，体积纹理也可以用来表示诸如木材或大理石等材质的体积结构，一个具有这种纹理的模型似乎是用这种材料雕刻出来的。

使用体积纹理的表面纹理化是非常低效的，因为绝大多数的样本没有使用，Benson和Davis[133]以及DeBry等人[334]讨论了用松散八叉树（sparse octree）结构存储纹理数据，这种方案非常适合交互式三维绘画系统，因为在创建时，不需要分配明确的纹理坐标给表面，而且八叉树可以将纹理细节保持在所需的任何级别，Lefebvre等人[1017]讨论了在现代GPU上实现八叉树纹理的细节，Lefebvre和Hoppe[1018]讨论了一种将稀疏的体积数据打包到一个明显更小的纹理中的方法。

6.2.4 立方体贴图 Cube Maps

另一种类型的纹理是立方体纹理（cube texture）或立方体贴图（cube map），它有六个正方形纹理，每个纹理都与立方体的一个面相关联，一个立方体贴图是通过一个三分量的纹理坐标向量来访问的，它指定了从立方体中心向外的射线的方向，射线与立方体相交点的表达式如下所示。具有最大量级的纹理坐标选择相应的面（例如，向量选择−z面），剩下的两个坐标除以最大量级坐标的绝对值，即8.4，它们现在的范围是−1到1，之后简单地重新映射到以计算纹理坐标，例如坐标映射为，立方体映射用于表示方向函数的值，它们最常用于环境映射（第10.4.3节）。

6.2.5 纹理表示 Texture Representation

在处理应用程序中的多个纹理时，有几种方法可以提高性能，纹理压缩在第6.2.6节中描述，本节的重点是纹理图集（texture atlases）、纹理数组（texture arrays）和无绑定纹理（bindless textures），所有这些都是为了避免在渲染时改变纹理造成的成本，在第19.10.1和19.10.2节中，描述了纹理流和转码。

为了能够让GPU批处理尽可能多的工作，通常最好尽可能少地改变状态（第18.4.2节），为此，可以将多个图像放入一个较大的纹理中，称为纹理图集，如图6.20左侧所示，注意，子纹理的形状可以是任意的，如图6.6所示。Nöll和Stricker[1286]描述了子纹理放置图集的优化，还需要注意mipmap的生成和访问，因为mipmap的上层可能包含几个独立的、不相关的形状，Manson和Schaefer[1119]提出了一种通过考虑表面参数化来优化mipmap的创建的方法，可以产生更好的结果，Burley和Lacewell[213]提出了一种称为Ptex的系统，其中细分曲面中的每个四边形都有自己的小纹理，这样做的好处是避免了在一个网格上分配唯一的纹理坐标，并且在纹理图集中不连接的部分的接缝上没有伪像，为了能够跨四边形进行滤波，Ptex使用了邻接（adjacency）的数据结构，虽然最初的目标是产品渲染化，Hillesland[746]提出了打包的Ptex（packed Ptex），它将每个面的子纹理放入纹理图集，并使用相邻面的填充，以避免滤波时的不连续性，Yuksel[1955]提出了网格颜色纹理（mesh color textures），它改进了Ptex，Toth[1780]通过实现一种如果滤波器抽头（tap）超出的范围则丢弃它们的方法，从而为类似ptex的系统提供了高质量的跨面滤波。

图形用户界面

中度可信度描述已自动生成

**图 6.20. 左边：一个纹理图集，其中九个较小的图像被合成为一个单一的大纹理，右边：一种更现代的方法，将较小的图像设置为纹理数组，这是大多数API中的一个概念。**

使用图集的一个困难是包装（wrapping）/重复（repeat）和镜像模式（mirror modes），这将会不正确地影响子纹理，而不只影响整个纹理，另一个问题是在为图集生成mipmap时，一个子纹理可能会渗透到另一个子纹理中，不过这可以通过为每个子纹理分别生成mipmap层次结构来避免，之后将它们放入一个大的纹理图集中，并对子纹理使用二次方分辨率[1293]。

针对这些问题的一个更简单的解决方案是使用一个叫做纹理数组的API结构，它完全避免了mipmapping和重复模式的所有问题[452]，参见图6.20的右边部分，纹理数组中的所有子纹理都需要具有相同的维度、格式、mipmap层次结构和MSAA设置，与纹理图集一样，纹理数组的设置只需要完成一次，然后可以使用着色器中的索引访问任何数组元素，这比绑定每个子纹理要快5倍[452]。

API对无绑定纹理的支持[1407]也是有助于避免状态改变成本的一个特性。如果没有无绑定的纹理，纹理将使用API绑定到特定的纹理单元，一个问题是纹理单元数量的上限，这让程序员的工作变得复杂，驱动程序确保纹理驻留在GPU端。对于无绑定的纹理，没有纹理数量的上限，因为每个纹理都由一个64位指针（有时称为句柄（handle））关联到它的数据结构，这些句柄可以通过多种不同的方式访问，例如通过统一变量，通过可变的数据，来自其他纹理或来自着色器存储缓冲对象（SSBO）。应用程序需要确保纹理驻留在GPU端，无绑定纹理避免了驱动程序中任何类型的绑定成本，这使得渲染更快。

6.2.6 纹理压缩 Texture Compression

一个直接解决内存问题、带宽问题以及缓存问题的解决方案是固定速率纹理压缩（fixed-rate texture compression）[127]。通过让GPU动态解码压缩的纹理，纹理可以只需更少的纹理内存，从而增加有效缓存大小，至少同样重要的是，这样的纹理使用起来更有效，因为它们在访问时消耗更少的内存带宽，一个相关但不同的用例是添加压缩以提供更大的纹理（译注：意思是原理相同，都是压缩可以减少带宽，但目的不同，一个是为了减少消耗、一个为了增加质量）。例如一个非压缩纹理每个纹素使用3字节，分辨率为，将占用768kB，使用纹理压缩，以6:1的压缩比，分辨率的纹理只占用512kB。

在图像文件格式中使用了多种图像压缩方法，例如JPEG和PNG，但在硬件中实现解码这些是昂贵的（参见19.10.1节关于纹理转码的信息），S3开发了一个称为S3纹理压缩（S3 Texture Compression, S3TC）的方案[1524]，该方案被选为DirectX 10的标准并被称为DXTC，在DirectX 10中它被称为BC（Block Compression，块压缩），此外它是OpenGL中事实上的标准，因为几乎所有GPU都支持它，它的优点是可以创建固定大小的压缩图像，具有独立的编码片段，而且解码简单（因此也很快）。图像的每个压缩部分都可以独立处理，没有共享查找表或其他依赖项，这简化了解码。

DXTC/BC压缩方案有七个变体，它们具有一些共同的属性，编码是在4×4的纹素块上完成的，也称为铺砖（tile），每个块单独编码，编码是基于插值的，对于每个编码的数量，存储两个参考值（例如颜色），一个插值因子被保存到块中16个纹素的每一中，它沿着两个参考值之间的直线选择一个值，例如一种已存储颜色或从两种已存储颜色中插入的颜色。压缩来自于每个像素通过短索引值只存储两种颜色。

表格

描述已自动生成

**表 6.1. 纹理压缩格式，所有这些压缩块均为4×4纹素，存储列显示每个块的字节数（B）和每个纹素的位数（bpt）。参考颜色的符号首先是通道，然后是每个通道的比特数，例如RGB565表示红蓝通道5位，绿色通道6位。**

确切的编码在七个变体之间是不同的，表6.1将它们总结，

6.1.3 匹配函数 The Corresponder Function

5.4.1 采样与滤波理论 Sampling and Filtering Theory

第5章 着色基础 Shading Basics