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。**

**以采样和过滤理论为框架，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）的各种过滤技术结合起来，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过滤器，如上一节所述，它优于box过滤器。

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过滤器，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过滤[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过滤，从而模仿了像素覆盖率。为了得到颜色过滤器的效果，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。最好在执行过滤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），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]并乘以1/2.2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，发出的辐射不会呈现线性增长，而是（令人惊讶地）与该能级成正比地升高，例如假设功率为2，一个设置为50%的像素会发出四分之一的光，**比起被设置为1的像素，**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，通过测量阈值对比来，我们可以在很大范围的条件下检测到大约1%的亮度差异。当颜色存储在有限精度的显示容器中时，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，因此，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，大多数控制GPU的API可被设置为自动应用正确的sRGB转换，当从纹理中读取值或将值写入颜色缓冲区时[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]，关于它的一个好用法是我们用线性值来进行物理计算，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，将数据移到或移出其显示编码的形式。**

**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将值提高到一个幂，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“伽玛校正”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的倒数：**

|  |  |
| --- | --- |
|  | (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，然后用这个值乘以它自己求倒数，虽然这是一个粗略的近似，但这种转换比完全忽略这个问题要好。**

5.4.1 采样与滤波理论 Sampling and Filtering Theory

第5章 着色基础 Shading Basics