实时渲染 第4版

第5章 Shading Basics 基础着色

When you render images of three-dimensional objects, the models should not only

have the proper geometrical shape, they should also have the desired visual

appearance. Depending on the application, this can range from photorealism—an

appearance nearly identical to photographs of real objects—to various types of

stylized appearance chosen for creative reasons. See Figure 5.1 for examples of both.

This chapter will discuss those aspects of shading that are equally applicable to

photorealistic and stylized rendering. Chapter 15 is dedicated specifically to stylized

rendering, and a significant part of the book, Chapters 9 through 14, focuses on

physically based approaches commonly used for photorealistic rendering.

当你渲染三维对象图像时，模型不仅仅要有合适的几何外形，还要有理想的视觉外观。依据应用程序的不同，这可以包括从几乎与真实物体照片相同的光照到出于创造性原因而选择的各种类型的程序式外观。图5.1给出了两者的示例。这一章将主要从着色的角度，讨论真实和非真实渲染。第15章是专门为风格化渲染，书的重要部分，第9章到第14章，集中在物理基础上的方法，通常用于真实渲染。

5.1 Shading Models 着色器模型

The first step in determining the appearance of a rendered object is to choose a

shading model to describe how the object’s color should vary based on factors such

as surface orientation, view direction, and lighting.

决定渲染物体的呈现外观的第一步是选择一个着色器模型去描述物体的颜色应该如何根据表面方向，视角方向和光照因素而变化。

As an example, we will use a variation on the Gooch shading model [561]. This is a

form of non-photorealistic rendering, the subject of Chapter 15. The Gooch shading

model was designed to increase legibility of details in technical illustrations.

例如，我们将使用Gooch着色模型的一个变体[561]。这是非真实渲染的一种形式，是第15章的主题。Gooch着色模型是为了增加技术插图的细节而设计的。

The basic idea behind Gooch shading is to compare the surface normal to the light’s

location. If the normal points toward the light, a warmer tone is used to color the

surface; if it points away, a cooler tone is used. Angles in between interpolate

between these tones, which are based on a user-supplied surface color. In this

example, we add a stylized “highlight” effect to the model to give the surface a shiny

appearance. Figure 5.2 shows the shading model in action.

Gooch着色的基础思想是比较表面法线朝向光照的位置，如果法线点是朝向光照的话，就在表面使用暖色色调的颜色；如果偏离光照，会使用冷色色调的颜色。在色调之间使用角度插值。在这个例子中，我们增加一个非真实“高亮”效果在模型上，给表面一种光泽的表现。图5.2显示了着色器模型。

Shading models often have properties used to control appearance variation. Setting

the values of these properties is the next step in determining object appearance. Our

example model has just one property, surface color, as shown in the bottom image

of Figure 5.2.

着色器模型常常会提供属性来控制外观变化。设置这些属性的值是决定物体表面的下一步。我们的例子只有一个属性：表面颜色，显示在图5.2的底部图片。

Like most shading models, this example is affected by the surface orientation relative

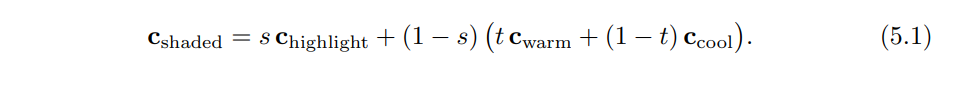
to the view and lighting directions. For shading purposes, these directions are

commonly expressed as normalized (unit-length) vectors, as illustrated in Figure 5.3.

像大部分着色器模型，这个例子受到了表面方向相对于视角以及光照方向的影响。出于着色目的，这些方向通常表示为标准化（单位长度）向量，如图5.3所示。

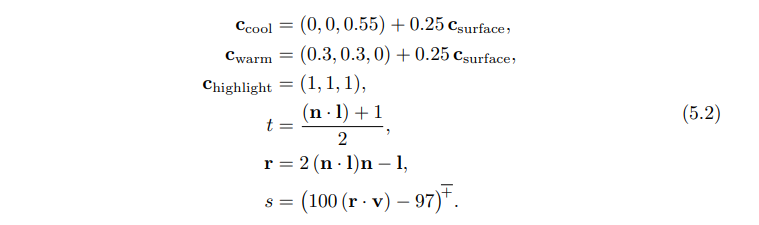
Now that we have defined all the inputs to our shading model, we can look at the

mathematical definition of the model itself:



现在我们定义了着色器模型的输出，我们可以看看模型本身的数学定义。

In this equation, we have used the following intermediate calculations:



在这个方程中个，我们可以使用下面中间计算过程。

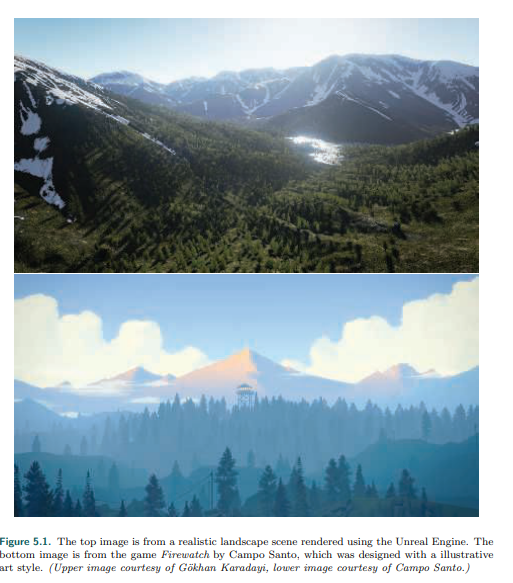


图5.1 顶部的照片是一张使用虚幻引擎的真实风景渲染。底部照片来自Campo Santo制作的游戏Firewatch，使用了插图艺术风格设计（上面图片来自G¨okhan Karadayi，下部图片来自Campo Santo）。

Several of the mathematical expressions in this definition are often found in other

shading models as well. Clamping operations, typically clamping to 0 or clamping

between 0 and 1, are common in shading. Here we use the x + notation, introduced

in Section 1.2, for the clamp between 0 and 1 used in the computation of the

highlight blend factor s. The dot product operator appears three times, in each case

between two unit-length vectors; this is an extremely common pattern. The dot

product of two vectors is the product of their lengths and the cosine of the angle

between them. So, the dot product of two unit-length vectors is simply the cosine,

which is a useful measure of the degree to which two vectors are aligned with each

other. Simple functions composed of cosines are often the most pleasing and

accurate mathematical expressions to account for the relationship between two

directions, e.g., light direction and surface normal, in a shading model.

在这个定义中的几个数学公式经常在其他的着色器模型中看到。钳位操作，在常规着色中，典型的是钳位到0或者钳位到0到1之间。在这里我们使用x(+-)符号，在章节1.2中介绍过，钳位到0到1之间用在计算高光混合因子s。点积操作出现了三次，每种情况都是在两个单位长度之间；这是极其常规的操作。两向量的点积是指的是他们的长度以及两者之间角度的cos之积。因此两个单位长度的向量积就可以简单的用cos表示，这对于衡量两个向量的角度是否一致非常有用。在着色器模型中，cos的简单的函数组合，通常是描述两个方向（例如光照方向的表面法线）之间关系的最令人愉快和准确的数学表达式。

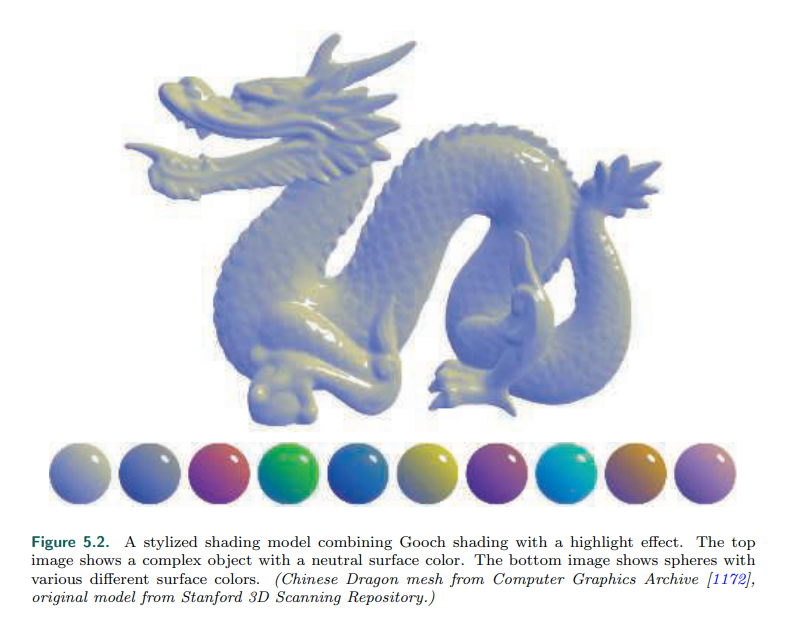


图5.2 一个风格化的着色模型联合了Gooch着色使用了高光影响。上部图像显示了混合物体使用了中性的表面颜色。底部的照片显示了使用了不同颜色表面颜色的球（中国龙网格来自计算机图形档案[1172]，原始模型来自斯坦福三维扫描库）。

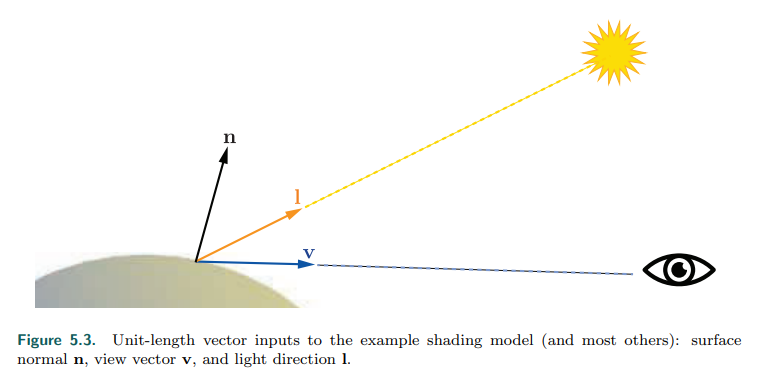


图5.3：单位长度向量输入着色器模型例子（其他大部分模型也是）：表面法线n，视角v，以及光照方向l。

Another common shading operation is to interpolate linearly between two colors

based on a scalar value between 0 and 1. This operation takes the form tca + (1 –

t)cb that interpolates between ca and cb as the value of t moves between 1 and 0,

respectively. This pattern appears twice in this shading model, first to interpolate

between cwarm and ccool and second to interpolate between the result of the

previous interpolation and chighlight. Linear interpolation appears so often in shaders

that it is a built-in function, called lerp or mix, in every shading language we have

seen.

另一个常规着色操作是两个基础颜色以0到1之间的标量进行线性插值。这种操作tca + (1 − t)cb通过t在0到1之间的移动，在ca和cb之间插值。这个模式在着色器模型中出现了两次，第一次在cwarm和ccool之前插值，第二次是在之前结果和chighlight之间插值。线性插值在着色器模型中的出现如此频繁，因此特定建立了函数，称作lerp或者mix，在我们看到的每种着色器语言中。

The line “r = 2 (n · l)n − l” computes the reflected light vector, reflecting l about n.

While not quite as common as the previous two operations, this is common enough

for most shading languages to have a built-in reflect function as well.

“r = 2 (n · l)n − l”这一行计算光向量的反射向量，关于n反射l。并不是如先前的两个操作那样常规，但是足够常规到大部分着色器语言都建立了反射函数。

By combining such operations in different ways with various mathematical

expressions and shading parameters, shading models can be defined for a huge

variety of stylized and realistic appearances.

通过联合各种各样的数学表达式和着色参数的此类操作，着色模型可以被各种风格和现实定义以及呈现。

5.2 Light Source 光源

The impact of lighting on our example shading model was quite simple; it provided a

dominant direction for shading. Of course, lighting in the real world can be quite

complex. There can be multiple light sources each with its own size, shape, color,

and intensity; indirect lighting adds even more variation. As we will see in Chapter 9,

physically based, photorealistic shading models need to take all these parameters

into account.

光照的影响在我们着色器模型例子中十分简单；他提供了一个显著的方向。当然，光照在真实世界中十分复杂。这里可能会有多重光源，每个都有自己的尺寸，形状，颜色以及强度；间接照明增加了更多的变种。我们会在第9章讲述基于物理的真实着色模型需要将这些所有参数考虑进去。

In contrast, stylized shading models may use lighting in many different ways,

depending on the needs of the application and visual style. Some highly stylized

models may have no concept of lighting at all, or (like our Gooch shading example)

may only use it to provide some simple directionality.

相反，非写实着色模型可能以很多不同的方式使用光照，这依据于程序和虚拟风格的需要而定。一些高度非写实模型可能没有光照的概念，或者（例如我们的Gooch着色例子）仅仅用它提供一些简单的方向。

The next step in lighting complexity is for the shading model to react to the presence

or absence of light in a binary way. A surface shaded with such a model would have

one appearance when lit and a different appearance when unaffected by light. This

implies some criteria for distinguishing the two cases: distance from light sources,

shadowing (which will be discussed in Chapter 7), whether the surface is facing away

from the light source (i.e., the angle between the surface normal n and the light

vector l is greater than 90◦ ), or some combination of these factors.

光照复杂性的下一步是对于着色模型以分叉的方式对光的存在或不存在做出反应。用这种模型的表面着色在光照下会有一种外观，在不受光照影响的情况下会有不同的外观。这意味着区分这两种情况的一些标准：光源的距离，阴影（将在第7章节讨论），表面是否朝向光源（即表面法线n和光向量l之间的角度是否大于90度），或者是这些因素的组合。

It is a small step from the binary presence or absence of light to a continuous scale

of light intensities. This could be expressed as a simple interpolation between

absence and full presence, which implies a bounded range for the intensity, perhaps

0 to 1, or as an unbounded quantity that affects the shading in some other way. A

common option for the latter is to factor the shading model into lit and unlit parts,

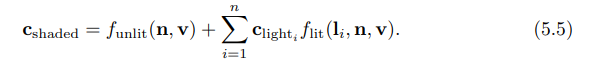
with the light intensity klight linearly scaling the lit part:



This easily extends to an RGB light color clight,



and to multiple light sources,



这是一个小的步骤从二元存在或者没有光到一个连续规模的光强度。这可以表示为一个简单的无光到满光的插值运算，这意味着一个有界的范围的强度，也许就是0到1，或者作为一个无界的数量，以某种方式影响着色。后者的一个常见的选择是将着色模型分解为光照的非光照部分，以及光照强度klight线性影响光照部分。

很容易拓展到RGB光照颜色。

以及拓展到多重光源。

The unlit part funlit(n, v) corresponds to the “appearance when unaffected by light”

of shading models that treat light as a binary. It can have various forms, depending

on the desired visual style and the needs of the application. For example, funlit() =

(0, 0, 0) will cause any surface unaffected by a light source to be colored pure black.

Alternately, the unlit part could express some form of stylized appearance for unlit

objects, similar to the Gooch model’s cool color for surfaces facing away from light.

Often, this part of the shading model expresses some form of lighting that does not

come directly from explicitly placed light sources, such as light from the sky or light

bounced from surrounding objects. These other forms of lighting will be discussed in

Chapters 10 and 11.

无光部分funlit(n, v)对应于将光视为二进制的着色模型的“不受光照影响时的外观”。他可以有多种格式，依据于所需的视觉风格和应用程序的需要。例如，funlit() = (0, 0, 0)将使任何不受光源影响的表面呈现纯黑色。另外，非光照部分可以为非真实对象表示某种形式的风格化外观，类似于Gooch模型为背向光的表面使用的冷色。通常，着色模型的这一部分表达了某种形式的照明并不直接来自于显示放置的光源，比如来自天空或者来自周围物体的反弹光。这种其他形式的照明将在第10章和第11章中讨论。

We mentioned earlier that a light source does not affect a surface point if the light

direction l is more than 90◦ from the surface normal n, in effect coming from

underneath the surface. This can be thought of as a special case of a more general

relationship between the light’s direction, relative to the surface, and its effect on

shading. Although physically based, this relationship can be derived from simple

geometrical principles and is useful for many types of non-physically based, stylized

shading models as well.

我们之前提到过光源并不会影响当表面法线n与光照方向l角度大于90度的情况。这可以看做是光照方向，相对于表面和他对着色影响之间更一般关系的一个特例。虽然是基于物理的，这种关系可以从简单几何法则中推导出来，并且对于许多类型是非基于物理的、非真实着色模型也是有用的。

The effect of light on a surface can be visualized as a set of rays, with the density of

rays hitting the surface corresponding to the light intensity for surface shading

purposes. See Figure 5.4, which shows a cross section of a lit surface. The spacing

between light rays hitting the surface along that cross section is inversely

proportional to the cosine of the angle between l and n. So, the overall density of

light rays hitting the surface is proportional to the cosine of the angle between l and

n, which, as we have seen earlier, is equal to the dot product between those two

unit-length vectors. Here we see why it is convenient to define the light vector l

opposite to the light’s direction of travel; otherwise we would have to negate it

before performing the dot product.

光照的影响可以形象化为一组射线，射线的密度击中表面对应于光强作用于表面着色。如图5.4，显示了光照表面的横截面。沿着横截面击中表面的光射线之间的空隙与l和n之间的夹角的cos值成负相关。因此，光射线击中表面的整体密度与l和n之间夹角的cos值成正相关，正如我们之前所见，与这两个单位长度向量的点积相等。这里我们可以看到为什么定义光向量l与光的运动方向相反是很方便的；否则我们做点积之前要先做负处理。

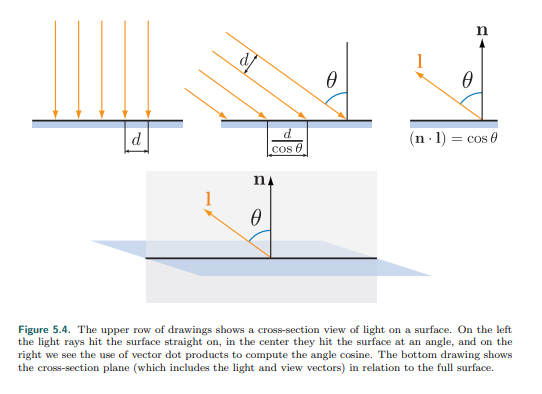


图5.4 上面一行显示的是光在表面的横截面处理。在左侧光射线垂直射向表面，在中间他们以一定的角度射向表面，在右侧我们看到计算l和n的点积来计算其角度。底部图片显示了整个平面的横截面（包括了光和视角向量）。

More precisely, the ray density (and thus the light’s contribution to shading) is

proportional to the dot product when it is positive. Negative values correspond to

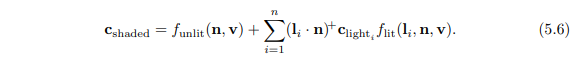
light rays coming from behind the surface, which have no effect. So, before

multiplying the light’s shading by the lighting dot product, we need to first clamp the

dot product to 0. Using the x + notation introduced in Section 1.2, which means

clamping negative values to zero, we have

更确切的说，射线密度（光对于着色的贡献）只有当点积为正值的时候才与点积成正相关。负值对应的光射线是来自表面的背部，这一部分光线并没有什么影响。因此，在通过点积增加光的着色影响时，我们需要映射点积到0，使用在1.2节介绍的x +符号，意味着将负值映射为0，因此公式如下。



Shading models that support multiple light sources will typically use one of the

structures from Equation 5.5, which is more general, or Equation 5.6, which is

required for physically based models. It can be advantageous for stylized models as

well, since it helps ensure an overall consistency to the lighting, especially for

surfaces that are facing away from the light or are shadowed. However, some

models are not a good fit for that structure; such models would use the structure in

Equation 5.5.

支持多重光源的着色器模型可以使用公式5.5中结构，这个更为普遍一些，或者使用公式5.6，常用在基于物理的模型。他同样适用于非真实渲染，因为他可以帮助确定表面与光的一致性，尤其是表面朝向光还是阴影。然而，一些模型并不适用这个公式；这些模型将使用公式5.5。

The simplest possible choice for the function flit() is to make it a constant color,



which results in the following shading model:



最简单的可能性选择是将函数flit()设置为常量颜色。那么着色模型的结果就如下所示。

The lit part of this model corresponds to the Lambertian shading model, after Johann

Heinrich Lambert [967], who published it in 1760! This model works in the context of

ideal diffusely reflecting surfaces, i.e., surfaces that are perfectly matte. We present

here a somewhat simplified explanation of Lambert’s model, which will be covered

with more rigor in Chapter 9. The Lambertian model can be used by itself for simple

shading, and it is a key building block in many shading models.

模型中光的部分对应于兰伯特着色模型，由Johann Heinrich Lambert[967]出版与1760年。这个模型是在理想漫反射表面的情况下工作，即表面是完美光泽。我们这里只对兰伯特模型做一个简单的介绍，第9章将会给出更严谨的说明。兰伯特模型可以使用在简单的着色情况，并且他也是很多着色模型的关键点。

We can see from Equations 5.3–5.6 that a light source interacts with the shading

model via two parameters: the vector l pointing toward the light and the light color

clight. There are various different types of light sources, which differ primarily in how

these two parameters vary over the scene.

从公式5.3-5.6，我们可以看出光源通过两个参数影响着色模型：向量l指向光的方向和光的颜色clight。有各种不同的光源类型，他们的主要区别在于如何使用这两个参数随场景而变化。

We will next discuss several popular types of light sources, which have one thing in

common: At a given surface location, each light source illuminates the surface from

only one direction l. In other words, the light source, as seen from the shaded

surface location, is an infinitesimally small point. This is not strictly true for real-world

lights, but most light sources are small relative to their distance from illuminated

surfaces, making this a reasonable approximation. In Sections 7.1.2 and 10.1, we

will discuss light sources that illuminate a surface location from a range of directions,

i.e., “area lights.”

我们接下来讨论几种常见的光源类型，他们有一点相同：在给定表面位置，每一个光源，只有一个方向l照亮表面。用其他话来说，光源从着色表面位置来看，他只是一个极小的点。这对于真实世界光来说，并不完全正确，但是大部分光源相对于他们照亮表面的距离来讲都很小，这似乎是一个合理的近似。在章节7.1.2和10.1，我们将讨论光源从一个范围方向照亮表面位置，即区域光。

5.2.1 Directional Lights 平行光

Directional light is the simplest model of a light source. Both l and clight are constant

over the scene, except that clight may be attenuated by shadowing. Directional lights

have no location. Of course, actual light sources do have specific locations in space.

Directional lights are abstractions, which work well when the distance to the light is

large relative to the scene size. For example, a floodlight 20 feet away illuminating a

small tabletop diorama could be represented as a directional light. Another example

is pretty much any scene lit by the sun, unless the scene in question is something

such as the inner planets of the solar system.

平行光是光源的最简单的模型。L和clight都是常量，除了clight可能被阴影衰减。平行光没有位置。当然，在空间中，真实的光源都有特定的位置。平行光是抽象的，当光源距离相对于场景尺寸非常大的时候，才工作良好。例如，一个照明灯在20尺以外的位置照明了一个小的桌面模型，此时可以称照明灯为平行光。另外一个例子在场景光中更常见就是太阳，除非这个场景是太阳系内行星之类的东西。

The concept of a directional light can be somewhat extended to allow varying the

value of clight while the light direction l remains constant. This is most often done to

bound the effect of the light to a particular part of the scene for performance or

creative reasons. For example, a region could be defined with two nested (one inside

the other) box-shaped volumes, where clight is equal to (0, 0, 0) (pure black)

outside the outer box, is equal to some constant value inside the inner box, and

smoothly interpolates between those extremes in the region between the two boxes.

平行光的概念可以稍微拓展，允许改变clight的值，而光的方向l保持不变。这通常是为了表现或创在性的原因，将灯光效果绑定到场景的特定部分。例如，可以定义一个区域有两个嵌套的盒子（一个在另一个内部）着色体，clight等于(0, 0, 0)（纯黑）以外的外框，等于一个常量在内盒内，两个盒子之间区域平滑插值。

5.2.2 Punctual Lights 精确光源

A punctual light is not one that is on time for its appointments, but rather a light that

has a location, unlike directional lights. Such lights also have no dimensions to them,

no shape or size, unlike real-world light sources. We use the term “punctual,” from

the Latin punctus meaning “point,” for the class consisting of all sources of

illumination that originate from a single, local position. We use the term “point light”

to mean a specific kind of emitter, one that shines light equally in all directions. So,

point and spotlight are two different forms of punctual lights. The light direction

vector l varies depending on the location of the currently shaded surface point p0

relative to the punctual light’s position plight:



精确光源并不是准时光源，而是一种有位置的光，不像平行光。这些光也没有尺寸，没有形状或者大小，不像真实世界的光源。我们使用术语“精确”，来自拉丁语的意思是“点”，指的是由一个单一的局部位置产生的所有光源组成的类。我们使用术语“点光源”来表示一种特殊的发射器，他在所有方向发射相等的光。因此，点光源和聚光灯是两种不同形式的精确光源。光方向向量I随着当前着色表面点p0相对于精确光源的位置plight而变化。

This equation is an example of vector normalization: dividing a vector by its length to

produce a unit-length vector pointing in the same direction. This is another common

shading operation, and, like the shading operations we have seen in the previous

section, it is a built-in function in most shading languages. However, sometimes an

intermediate result from this operation is needed, which requires performing the

normalization explicitly, in multiple steps, using more basic operations. Applying this

to the punctual light direction computation gives us the following:



这个方程式是向量标准化的一个例子：除以自身的长度得到一个指向相同位置的单位长度向量。这也是另一个常用的着色操作，并且，像之前我们看到的着色操作，在大部分的着色语言中有建立的有函数。然而，有时候需要操作的中间结果，需要使用更多的基础操作，多重步骤精确的执行标准化操作。应用到精确光源方向的计算如下。

Since the dot product of two vectors is equal to the product of the two vector’s

lengths with the cosine of the angle between them, and the cosine of 0◦ is 1.0, the

dot product of a vector with itself is the square of its length. So, to find the length of

any vector, we just dot it with itself and take the square root of the result.

由于两向量的点积等于两向量的长度以及他们之间角度的cos值得积，其中cos0度为1.0，一个向量与其自身的点积是其长度的平方。因此，为了得到向量的长度，我们可以仅仅使用它与自己的点积并计算它的平方根来计算。

The intermediate value that we need is r, the distance between the punctual light

source and the currently shaded point. Besides its use in normalizing the light vector,

the value of r is also needed to compute the attenuation (darkening) of the light

color clight as a function of distance. This will be discussed further in the following

section.

中间的值是r，是精确光源和当前着色点的距离。除了用在标准化光方向，r的值也被需要作为计算光颜色clight（变暗）随着距离的增加而衰减。这将会在接下来的部分更深入的讨论。

Point/Omni Lights 点/泛光源

Punctual lights that emit light uniformly in all directions are known as point lights or

omni lights. For point lights, clight varies as a function of the distance r, with the

only source of variation being the distance attenuation mentioned above. Figure 5.5

shows why this darkening occurs, using similar geometric reasoning as the

demonstration of the cosine factor in Figure 5.4. At a given surface, the spacing

between rays from a point light is proportional to the distance from the surface to

the light. Unlike the cosine factor in Figure 5.4, this spacing increase happens along

both dimensions of the surface, so the ray density (and thus the light color clight) is

proportional to the inverse square distance 1/r2 . This enables us to specify the

spatial variation in clight with a single light property, clight0 , which is defined as the

value of clight at a fixed reference distance r0:



精确光源朝着所有方向发射一致的光，正如点光源或泛光源。对于点光源，clight随着距离r变化，r是上面提到的唯一的衰减距离变化源。图5.5显示了为什么会发生变暗，使用了类似的几何推理来演示图5.4中的cos因子。在给定的表面，点光源的射线之间的距离，与表面到光的距离成正比。不像图5.4的cos因子，这种间距的增加沿着表面的尺寸进行，因此光线密度（也就是光颜色clight）与距离1/r2的平方反比成正相关。这使我们能够用一个光属性clight0指定clight中的空间变化，clight0定义为clight在固定参考距离r0处的值。

Equation 5.11 is often referred to as inverse-square light attenuation. Although

technically the correct distance attenuation for a point light, there are some issues

that make this equation less than ideal for practical shading use.

公式5.11经常成为平方反比光衰减。尽管从技术上讲，点光源的距离衰减是正确的，但是有一些问题使得这个方程对于实际的着色使用不是很理想。

The first issue occurs at relatively small distances. As the value of r tends to 0, the

value of clight will increase in an unbounded manner. When r reaches 0, we will

have a divide-by-zero singularity. To address this, one common modification is to

add a small value ǫ to the denominator [861]:



第一个问题出现在相对小的距离。在r趋近与0时，clight的值将增加为极大数。当r达到0，我们将有一个除以0的奇点。为了处理这种情况，一个常见的修正是在分母上增加一个小的值。

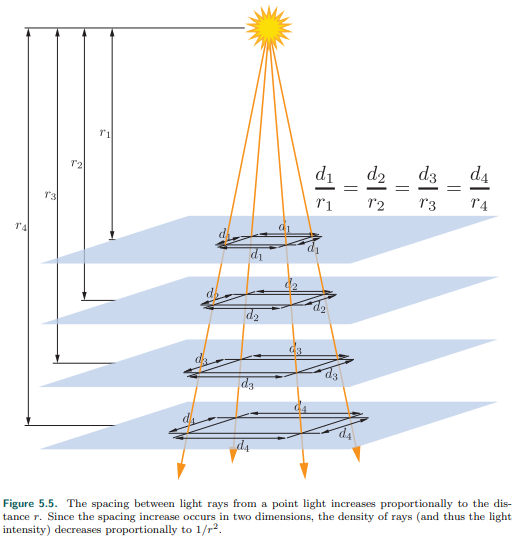


图5.5 点光源的射线之间的距离随着距离r正相关增长。由于距离增加发生在两个维度，射线的密度（也就是光强度）与1/r2成正相关。

The exact value used for ǫ depends on the application; for example, the Unreal

game engine uses ǫ = 1 cm [861].

使用ǫ的精准值根据应用程序而变化，比如虚幻引擎使用ǫ为1cm[861]。

An alternative modification, used in the CryEngine [1591] and Frostbite [960] game engines, is to clamp r to a minimum value rmin:



另一个可供选择的修正，使用在尖叫引擎和寒霜引擎，是映射r到一个最小值rmin。

Unlike the somewhat arbitrary ǫ value used in the previous method, the value of

rmin has a physical interpretation: the radius of the physical object emitting the light.

Values of r smaller than rmin correspond to the shaded surface penetrating inside

the physical light source, which is impossible.

不像之前的方法中使用的随意ǫ值，rmin的值具有物理解释；对象发射光的物理半径。R的值小于rmin对应于着色表面嵌入了物理光源的内部，是不可能发生的事。

In contrast, the second issue with inverse-square attenuation occurs at relatively

large distances. The problem is not with visuals but with performance. Although light

intensity keeps decreasing with distance it never goes to 0. For efficient rendering, it

is desirable for lights to reach 0 intensity at some finite distance (Chapter 20). There

are many different ways in which the inverse-square equation could be modified to

achieve this. Ideally the modification should introduce as little change as possible. To

avoid a sharp cutoff at the boundary of the light’s influence, it is also preferable for

the derivative and value of the modified function to reach 0 at the same distance.

One solution is to multiply the inverse-square equation by a windowing function with

the desired properties. One such function [860] is used by both the Unreal Engine

[861] and Frostbite [960] game engines:



与此相反，第二个问题是平方反比衰减出现在相对大的距离。问题不在于视觉效果，而在于性能。尽管光强度随着距离的增加而减少，但是他从来不趋近于0。为了有效的渲染，在有限的距离内，灯光最好达到0强度（第20章）。有很多不同的方法可以修改平方反比公式来实现这一点。理想情况下，修改应该引入尽可能少的更改。为了避免在光的影响便捷出现明显的截断，同样距离下修正函数的导数和值最好达到0。一种解决方法是将平方反比方唱乘以具有所需属性的窗口函数。虚幻引擎[861]和寒霜引擎[960]都使用了下面的方程[860]。

The +2 means to clamp the value, if negative, to 0 before squaring it. Figure 5.6

shows an example inverse-square curve, the windowing function from Equation 5.14,

and the result of multiplying the two.

+2的意思是钳位值，如果是负值，在平方前处理为0.在5.5图中显示了平方反比的曲线例子，窗函数是方程5.14,以及两者相乘的结果。

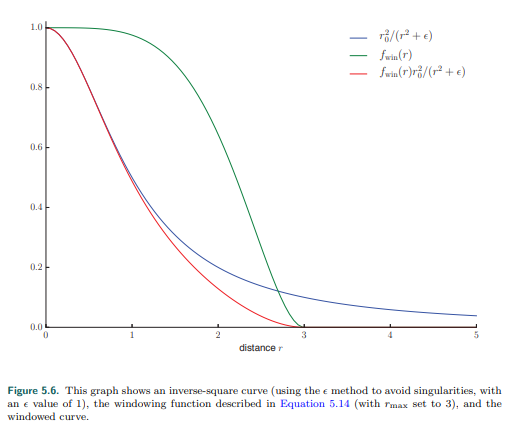


图5.6 图片显示了一个平方反比曲线（使用e方法避免奇点，并且e的值为1），窗函数是公式5.14（rmax设置为30），以及窗函数之后的曲线。

Application requirements will affect the choice of method used. For example, having

the derivative equal to 0 at rmax is particularly important when the distance

attenuation function is sampled at a relatively low spatial frequency (e.g., in light

maps or per-vertex). CryEngine does not use light maps or vertex lighting, so it

employs a simpler adjustment, switching to linear falloff in the range between

0.8rmax and rmax [1591].

程序需求影响着函数的选择。比如，当距离衰减函数以相对较低的空间频率采样时（例如光照贴图或逐顶点），rmax处的导数为0就显示尤为重要。尖叫引擎不使用光照贴图或顶点光照，因此他使用更简单的调整，切换0.8rmax和rmax之间的线性衰减[1591]。

For some applications, matching the inverse-square curve is not a priority, so some

other function entirely is used. This effectively generalizes Equations 5.11–5.14 to

the following:



在一些应用中，匹配平方反比曲线不是优先级，因此完全使用其他的函数。更有效的将公式5.11-5.14推广到。

where fdist(r) is some function of distance. Such functions are called distance falloff

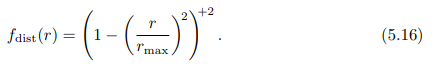
functions. In some cases, the use of non-inverse-square falloff functions is driven by

performance constraints. For example, the game Just Cause 2 needed lights that

were extremely inexpensive to compute. This dictated a falloff function that was

simple to compute, while also being smooth enough to avoid per-vertex lighting

artifacts [1379]:



这里的fdist(r)是一些距离的函数。这样的函数称为距离下降函数。在一些情况下，由于性能的限制，我们使用无平方反比的距离下降函数。比如，游戏Just Cause 2需要非常廉价的灯光计算。这就要求衰减函数计算简单，同时也足够光滑，以避免每个顶点光照伪影[1379]。

In other cases, the choice of falloff function may be driven by creative considerations.

For example, the Unreal Engine, used for both realistic and stylized games, has two

modes for light falloff: an inverse-square mode, as described in Equation 5.12, and

an exponential falloff mode that can be tweaked to create a variety of attenuation

curves [1802]. The developers of the game Tomb Raider (2013) used spline-editing

tools to author falloff curves [953], allowing for even greater control over the curve

shape.

在其他的情况下，衰减函数的选择可能是出于创造性的考虑。例如，虚幻引擎，用于现实和非现实的游戏，有两种模式的光衰减：一个平方反比模型，公式5.12所述，和指数衰减模型可以调整，以创造各种衰减曲线[1802]。游戏古墓丽影（2013）的开发者使用样条编辑工具来绘制衰减曲线，从而能够更好的控制曲线形状。

Spotlights 聚光灯

Unlike point lights, illumination from nearly all real-world light sources varies by

direction as well as distance. This variation can be expressed as a directional falloff

function fdir(l), which combines with the distance falloff function to define the overall

spatial variation in light intensity:



不像点光源，几乎所有的真实光源的照明都随方向和距离而变化。这种变化表示为方向衰减函数fdir(l)，结合距离衰减函数定义了光强的整体空间变化。

Different choices of fdir(l) can produce various lighting effects. One important type of

effect is the spotlight, which projects light in a circular cone. A spotlight’s directional

falloff function has rotational symmetry around a spotlight direction vector s, and

thus can be expressed as a function of the angle θs between s and the reversed light

vector −l to the surface. The light vector needs to be reversed because we define l

at the surface as pointing toward the light, and here we need the vector pointing

away from the light.

不同的fdir(l)选择能产生不同的光照效果。一个重要的效果就是聚光灯，光在一个圆锥体中照射。聚光灯的方向衰减函数绕着聚光灯方向向量s轴对称，因此可以表示为θs相反的光到表面向量-l与s之间角度的函数。光向量需要翻转是因为我们定义了l是由表面射向光，但是在这里，我们需要向量背离光。

Most spotlight functions use expressions composed of the cosine of θs, which (as we

have seen earlier) is the most common form for angles in shading. Spotlights

typically have an umbra angle θu, which bounds the light such that fdir(l) = 0 for all

θs ≥ θu. This angle can be used for culling in a similar manner to the maximum

falloff distance rmax seen earlier. It is also common for spotlights to have a

penumbra angle θp, which defines an inner cone where the light is at its full intensity.

See Figure 5.7.

大部分聚光灯函数是由cosθs组成的表达式，（正如我们前面所看到的）是最常见的角度着色。聚光灯通常有一个暗影角，像fdir(l) = 0这样的边界光为所有θs ≥ θu。这个角度可以用与前面看到的最大衰减距离rmax类似的方式进行剔除。这也是常见的聚光灯半影角θp，他定义了一个内层，光线的强度。如图5.7。

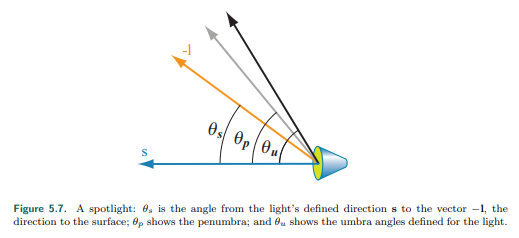


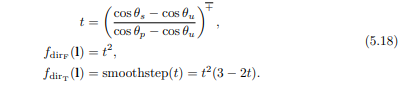
图5.7 聚光灯：θs是光照射方向s与向量-l的夹角，方向射向表面。Θp显示了半影角，θu显示了光的暗影角。

Various directional falloff functions are used for spotlights, but they tend to be

roughly similar. For example, the function fdirF (l) is used in the Frostbite game

engine [960], and the function fdirT (l) is used in the three.js browser graphics

library [218]:



各种方向衰减函数使用在聚光灯，但是他们大部分都相似。比如，函数fdirF (l)用在寒霜引擎[960]，函数fdirT (l)用在three.js浏览器图像库。

Recall that x + is our notation for clamping x between 0 and 1, as introduced in

Section 1.2. The smoothstep function is a cubic polynomial that is often used for

smooth interpolation in shading. It is a built-in function in most shading languages.

重述x+-意思是钳位x从0到1，在章节1.2中有介绍。Smoothstep函数是一个三次多项式经常用在着色中进行平滑插值。在大部分着色器语言中都有功能函数。

Figure 5.8 shows some of the light types we have discussed so far.

图5.8显示了我们目前为止讨论的光类型。

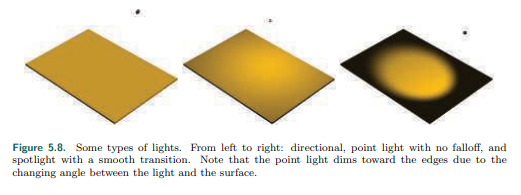


图5.8 光源类型。从左到右：平行光，没有衰减的点光源，平滑过渡的聚光灯。注意，由于光线与表面之间的角度变化，点光源会向边缘暗下来。

Other Punctual Lights 其他精确光源

There are many other ways in which the clight value of a punctual light can vary.

精确光源的clight值还有其他的变化方式。

The fdir(l) function is not limited to the simple spotlight falloff functions discussed

above; it can represent any type of directional variation, including complex tabulated

patterns measured from real-world light sources. The Illuminating Engineering

Society (IES) have defined a standard file format for such measurements. IES

profiles are available from many lighting manufacturers and have been used in the

game Killzone: Shadow Fall [379, 380], as well as the Unreal [861] and Frostbite

[960] game engines, among others. Lagarde gives a good summary [961] of issues

relating to parsing and using this file format.

fdir(l)函数不限于上面讨论的简单聚光灯衰减函数；他可以表示任何类型的方向变化，包括从真实光源测量的复杂表格模式。照明工程学会（IES）已经为这些测量定义了一个标准的文件格式。IES配置文件可以向很多照明制造商获得，并已被用于游戏杀戮地带Shadow Fall [379, 380]，以及虚幻引擎[861]和寒霜引擎[960]。Lagarde对于解析和使用这些文件格式有关的问题进行了很好的总结[961]。

The game Tomb Raider (2013) [953] has a type of punctual light that applies

independent falloff functions for distance along the x, y, and z world axes. In Tomb

Raider curves can also be applied to vary light intensity over time, e.g., to produce a

flickering torch.

游戏古墓丽影（2013）[953]有一种精确光，它对沿着x,y和z世界轴的距离应用独立的衰减函数。在古墓丽影中，曲线也可以用来随着时间改变光的强度，例如，产生一个闪烁的火炬。

In Section 6.9 we will discuss how light intensity and color can be varied via the use

of textures.

在章节6.9中，我们将讨论灯光强度以及颜色如何通过使用纹理进行改变。

5.2.3 Other Light Types 其他的光源类型

Directional and punctual lights are primarily characterized by how the light direction l

is computed. Different types of lights can be defined by using other methods to

compute the light direction. For example, in addition to the light types mentioned

earlier, Tomb Raider also has capsule lights that use a line segment as the source

instead of a point [953]. For each shaded pixel, the direction to the closest point on

the line segment is used as the light direction l.

平行光和精确光的主要特征是如何计算光的方向l。可以使用不同方法去定义不同类型的光源并计算光方向。例如，除了之前提到的光类型，古墓丽影有胶囊光使用线段作为光源而不是点[953]。对于每个着色像素，离线段上最近点的方向作为光的方向l。

As long as the shader has l and clight values for use in evaluating the shading

equation, any method can be used to compute those values.

只要着色器有l和clight值用于计算着色方程，任何方法可以用来计算这些值。

The types of light discussed so far are abstractions. In reality, light sources have size

and shape, and they illuminate surface points from multiple directions. In rendering,

such lights are called area lights, and their use in real-time applications is steadily

increasing. Area-light rendering techniques fall into two categories: those that

simulate the softening of shadow edges that results from the area light being

partially occluded (Section 7.1.2) and those that simulate the effect of the area light

on surface shading (Section 10.1). This second category of lighting is most

noticeable for smooth, mirror-like surfaces, where the light’s shape and size can be

clearly discerned in its reflection. Directional and punctual lights are unlikely to fall

into disuse, though they are no longer as ubiquitous as in the past. Approximations

accounting for a light’s area have been developed that are relatively inexpensive to

implement, and so are seeing wider use. Increased GPU performance also allows for

more elaborate techniques than in the past.

目前为止讨论的光的类型是抽象的。在现实中，光源有尺寸和形状，并且他们通过多重方向照亮表面。在渲染中，这样的光称为区域光，他们使用在实时应用中在稳固增加。区域光渲染技术分为两个类别：一类是模拟区域光部分遮挡导致阴影边缘软化的技术（章节7.1.2），另一类是模拟区域光在表面着色的影响（章节10.1）。第二类光源在平滑，镜像表面有显示效果，这种光的形状和尺寸在他的反射可以清楚辨认出的场景。方向光和精确光不太可能被废弃，尽管他们不再像过去一样普遍存在。计算区域光面积的近似方法已经开发出来，而且实现起来相对便宜，因此得到了更广泛的应用。GPU性能的提高也允许比过去更精细的技术。

5.3 Implementing Shading Models 实现着色模型

To be useful, these shading and lighting equations must of course be implemented in

code. In this section we will go over some key considerations for designing and

writing such implementations. We will also walk through a simple implementation

example.

为了有用，这些着色和光照方程当然比如用代码实现。在本节中，我们将讨论设计和编写此类实现的一些关键考虑事项。我们还将介绍一个简单的实现示例。

5.3.1 Frequency of Evaluation 评估的频率

When designing a shading implementation, the computations need to be divided

according to their frequency of evaluation. First, determine whether the result of a

given computation is always constant over an entire draw call. In this case, the

computation can be performed by the application, typically on the CPU, though a

GPU compute shader could be used for especially costly computations. The results

are passed to the graphics API via uniform shader inputs.

当设计一个着色器，应根据他们评估的频率分开计算。首先，决定是否计算的结果在整个绘制命令中保持不变。在这种情况下，计算可以由应用程序执行，通常在CPU中，尽管可以使用GPU计算着色器进行特别昂贵的计算。结果通过统一着色器输入传递给图形API。

Even within this category, there is a broad range of possible frequencies of

evaluation, starting from “once ever.” The simplest such case would be a constant

subexpression in the shading equation, but this could apply to any computation

based on rarely changing factors such as the hardware configuration and installation

options. Such shading computations might be resolved when the shader is compiled,

in which case there is no need to even set a uniform shader input. Alternatively, the

computation might be performed in an offline precomputation pass, at installation

time, or when the application is loaded.

即使在这一类别中，评估的可能频率范围也很广，从“一次”开始。最简单的例子是着色方程的常量子表达式，但是这可以应用于任何基于很少变化的因素计算，比如硬件配置和安装选项。这样的着色计算在着色器编译的过程就可能被解决，在这种情况下甚至都不需要设置统一着色器输入。或者，可以在安装时或加载应用程序时以离线预计算方式进行计算。

Another case is when the result of a shading computation changes over an

application run, but so slowly that updating it every frame is not necessary. For

example, lighting factors that depend on the time of day in a virtual game world. If

the computation is costly, it may be worthwhile to amortize it over multiple frames.

另一种情况是着色计算结果会在程序运行时改变，但是很慢并不是每一帧都更新一次。例如，依照与虚拟游戏世界时间的光照因素。如果计算是昂贵的，那么将其分摊到多个帧可能是值得的。

Other cases include computations that are performed once per frame, such as

concatenating the view and perspective matrices; or once per model, such as

updating model lighting parameters that depend on location; or once per draw call,

e.g., updating parameters for each material within a model. Grouping uniform shader

inputs by frequency of evaluation is useful for application efficiency, and can also

help GPU performance by minimizing constant updates [1165].

其他的情况是每一帧都执行的计算，比如连接的视图与投影矩阵；或者每个模型一次，比如根据位置更新模型光照参数；或者每个绘制命令一次，例如，更新一个模型中每个材质的参数。通过评估的频率分组统一着色输入对于应用效果有很大帮助，也可以通过极小常量更新提升GPU性能[1165]。

If the result of a shading computation changes within a draw call, it cannot be

passed to the shader through a uniform shader input. Instead, it must be computed

by one of the programmable shader stages described in Chapter 3 and, if needed,

passed to other stages via varying shader inputs. In theory, shading computations

can be performed on any of the programmable stages, each one corresponding to a

different evaluation frequency:

• Vertex shader—Evaluation per pre-tessellation vertex.

• Hull shader—Evaluation per surface patch.

• Domain shader—Evaluation per post-tessellation vertex.

• Geometry shader—Evaluation per primitive.

• Pixel shader—Evaluation per pixel.

如果在着色计算结果在一个绘制命令之间改变，那么他就不能通过统一着色器输入传递给着色器。取而代之的是他必须在一个可编程着色器阶段计算，并且，如果需要，通过各种着色器输入传递到其他阶段。理论上，着色计算可以在任意可编程阶段执行，每一个对应不同的评估频率：  
• 顶点着色器—评估每个曲面细分顶点

• 外壳着色器—评估每个面片

•域着色器—评估每个处理后的曲面细分顶点

•几何着色器—评估每个片元

• 像素着色器—评估每个像素

In practice most shading computations are performed per pixel. While these are

typically implemented in the pixel shader, compute shader implementations are

increasingly common; several examples will be discussed in Chapter 20. The other

stages are primarily used for geometric operations such as transformation and

deformation. To understand why this is the case, we will compare the results of per-

vertex and perpixel shading evaluations. In older texts, these are sometimes referred

to as Gouraud shading [578] and Phong shading [1414], respectively, though those

terms are not often used today. This comparison uses a shading model somewhat

similar to the one in Equation 5.1, but modified to work with multiple light sources.

The full model will be given a bit later, when we cover an example implementation in

detail.

在实际中，大部分着色计算是按每像素执行的。虽然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越来越普遍；第20章将讨论几个例子。其他阶段主要是用于几何运算，如变换和变形。为了理解为什么是这种情况，我们将比较每个顶点和每个像素着色计算的结果。在过去的书中，他们有时分别被称为高洛德着色[578]和phong着色[1414]，尽管这些术语在今天并不常用。这个比较使用了有点类似公式5.1中的着色模型，但是经过了修改，可以处理多个光源。稍后，当我们详细介绍一个示例实现时，将给出完整的模型。

Figure 5.9 shows the results of per-pixel and per-vertex shading on models with a

wide range of vertex densities. For the dragon, an extremely dense mesh, the

difference between the two is small. But on the teapot, vertex shading evaluation

causes visible errors such as angularly shaped highlights, and on the two-triangle

plane the vertex shaded version is clearly incorrect. The cause of these errors is that

parts of the shading equation, the highlight in particular, have values that vary

nonlinearly over the mesh surface. This makes them a poor fit for the vertex shader,

the results of which are interpolated linearly over the triangle before being fed to the

pixel shader.

图5.9显示了不同顶点密度模型的逐像素和逐顶点着色结果。对于龙来说，非常密集的网格，两者之间的差别很小。但是在茶壶中，顶点着色的计算会导致可见的错误，比如角度形状的高光，而在两个三角形的平面上，顶点着色的版本显示是不正确的。这些错误的原因是着色方程的某些部分，尤其是高光部分，其值在网格表面上是非线性的变化。这使得它们不适合顶点着色器，顶点着色器的结果在输入到像素着色器之前在三角形上进行线性插值。

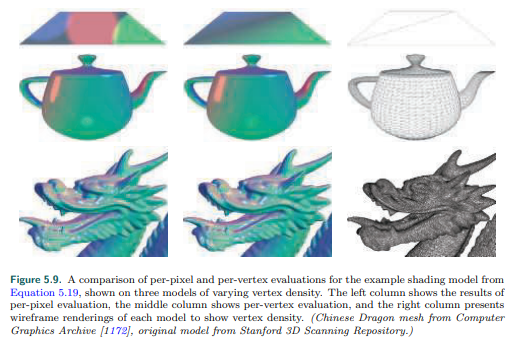


图5.9 根据方程5.19逐像素着色与逐顶点着色的对比，这些模型的顶点密度是不同的。左列是逐像素的结果，中间列是逐顶点的计算，左侧显示了线框渲染，每个模型的顶点密度。（中国龙来组计算图形结构[1172]，原始模型来自标准3D浏览库）

In principle, it would be possible to compute only the specular highlight part of the

shading model in the pixel shader, and calculate the rest in the vertex shader. This

would likely not result in visual artifacts and in theory would save some computation.

In practice, this kind of hybrid implementation is often not optimal. The linearly

varying parts of the shading model tend to be the least computationally costly, and

splitting up the shading computation in this way tends to add enough overhead, such

as duplicated computations and additional varying inputs, to outweigh any benefit.

大体上，可能仅仅在像素着色器上计算高光部分，而剩余的在顶点着色器中计算。这可能不会产生人为视觉效果，而且理论上可以节省一部分计算。实际上，这种混合实现通常不是最优的。着色模型的线性变化部分的计算成本往往是最低的，以这种方式分割着色计算往往会增加足够的开销，比如重复计算和额外的变化输入，从而超过任何好处。

As we mentioned earlier, in most implementations the vertex shader is responsible

for non-shading operations such as geometry transformation and deformation. The

resulting geometric surface properties, transformed into the appropriate coordinate

system, are written out by the vertex shader, linearly interpolated over the triangle,

and passed into the pixel shader as varying shader inputs. These properties typically

include the position of the surface, the surface normal, and optionally surface

tangent vectors, if needed for normal mapping.

正如我们之前提到的，在大部分顶点着色器的实现中都是处理非着色操作，比如几何转换和变形。最终得到的几个表面属性，转换到适当的坐标系，由顶点着色器编写出来，通过三角形进行线性插值，然后作为不同的着色器输入传递到像素着色器。这些属性通常包括表面的位置，表面法线，如果需要进行法线映射的话，还可以选择的表面切线向量。

Note that even if the vertex shader always generates unit-length surface normals,

interpolation can change their length. See the left side of Figure 5.10. For this reason

the normals need to be renormalized (scaled to length 1) in the pixel shader.

However, the length of the normals generated by the vertex shader still matters. If

the normal length varies significantly between vertices, e.g., as a side effect of

vertex blending, this will skew the interpolation. This can be seen in the right side of

Figure 5.10. Due to these two effects, implementations often normalize interpolated

vectors before and after interpolation, i.e., in both the vertex and pixel shaders.

注意尽管顶点着色器生成了单位长度的表面法线，但是插值可能改变他们的长度。见图5.10的左边。由于这种原因，法线需要在像素着色器中被重新归一化（缩放到长度1）。然而，顶点着色器生成的法线长度仍然很重要。如果法线长度在顶点之间有显著变化，比如作为顶点混合的一个副作用，将倾斜插值。这可以见图5.10的右侧。由于这两种影响，实现常常在插值前后对向量进行归一化，即在顶点着色器和像素着色器中。

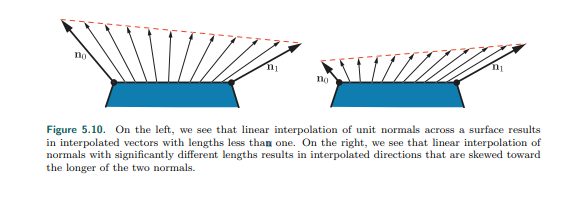


图5.10 在左侧，我们看到单位发现在表面的线性插值，插值后的结果向量长度小于1。在右侧，我们看到显著不同长度的法线之间进行线性插值，结果插值后的结果方向朝着两个法线的最长值倾斜了。

Unlike the surface normals, vectors that point toward specific locations, such as the

view vector and the light vector for punctual lights, are typically not interpolated.

Instead, the interpolated surface position is used to compute these vectors in the

pixel shader. Other than the normalization, which as we have seen needs to be

performed in the pixel shader in any case, each of these vectors is computed with a

vector subtraction, which is quick. If for some reason it is necessary to interpolate

these vectors, do not normalize them beforehand. This will yield incorrect results, as

shown in Figure 5.11.

不像表面法线，指向特定点的向量，比如视角向量和精确光的光向量，通常不进行插值运算。取而代之的是，在像素着色器中对表面位置进行插值去计算这些向量。除了我们已经看到的归一化（在任何情况下都需要在像素着色器中执行）之外，这些向量都是用向量减法计算的，这非常快。如果由于某种原因需要插值这些向量，不要预先对他们进行标准化。这将产生不正确的结果，如图5.11所示。

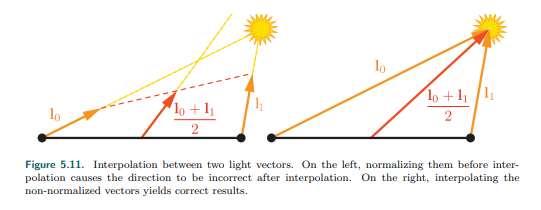


图5.11 在两个光向量之间的插值。在左侧，在插值之前进行归一化，导致了插值之后的方向错误。在右侧，插值非单位向量得到了正确的结果。

Earlier we mentioned that the vertex shader transforms the surface geometry into

“the appropriate coordinate system.” The camera and light positions, passed to the

pixel shader through uniform variables, are typically transformed by the application

into the same coordinate system. This minimizes work done by the pixel shader to

bring all the shading model vectors into the same coordinate space. But which

coordinate system is the “appropriate” one? Possibilities include the global world

space as well as the local coordinate system of the camera or, more rarely, that of

the currently rendered model. The choice is typically made for the rendering system

as a whole, based on systemic considerations such as performance, flexibility, and

simplicity. For example, if rendered scenes are expected to include huge numbers of

lights, world space might be chosen to avoid transforming the light positions.

Alternately, camera space might be preferred, to better optimize pixel shader

operations relating to the view vector and to possibly improve precision (Section

16.6).

正如我们之前提到的，在顶点着色器中，转换表面几何体到正确的坐标系统。摄像机和光位置通过一致的变量传递给像素着色器，通常由应用程序转换为相同的坐标系。这将最小化像素着色器所做的工作，从而将所有的着色器模型向量带入到相同的坐标空间。但是哪一个坐标系是合适的。可能性包括全局世界空间，以及摄像机的当地坐标系统，或者更少见的当前呈现的渲染模型坐标系。通常根据系统的考虑（如性能，灵活性和简单性）为渲染系统整体做出选择。例如，渲染的场景预计包含大量的光照，世界空间可以避免灯光位置的转换。另外摄像机空间可能是首选，以更好地优化与视图向量相关的着色器操作，并可能提高精度（第16.6节）。

Although most shader implementations, including the example implementation we

are about to discuss, follow the general outline described above, there are certainly

exceptions. For example, some applications choose the faceted appearance of per

primitive shading evaluation for stylistic reasons. This style is often referred to as flat

shading. Two examples are shown in Figure 5.12.

尽管大多数着色器实现，包括我们将要讨论的示例实现，都遵循上面描述的一般大纲，但是也有例外。比如，一些应用程序出于风格上的原因选择了逐像素着色的分面外观。这种样式通产称为平面着色。图5.12显示了两个示例。

In principle, flat shading could be performed in the geometry shader, but recent

implementations typically use the vertex shader. This is done by associating each

primitive’s properties with its first vertex and disabling vertex value interpolation.

Disabling interpolation (which can be done for each vertex value separately) causes

the value from the first vertex to be passed to all pixels in the primitive.

原则上，平面着色可以在几何着色器上执行，但是最近才开始使用顶点着色器实现。通过第一个顶点以及每个片元属性的联系和禁用顶点数值插值实现的。禁用插值（可以对每个顶点值单独执行）的原因是要传递第一个顶点的值到图元中的所有像素。

5.3.2 Implementation Example 实现例子

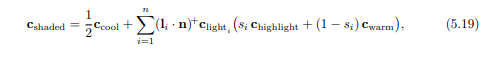
We will now present an example shading model implementation. As mentioned

earlier, the shading model we are implementing is similar to the extended Gooch

model from Equation 5.1, but modified to work with multiple light sources. It is

described by

我们现在呈现一个着色器实现的例子。如之前提到的，着色器模型与公式5.1扩展的Gooch模型相似，但是改进了可以与多重光源一起工作。描述如下：



with the following intermediate calculations:

以及中间计算如下：

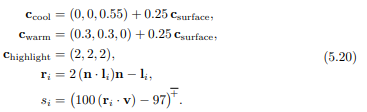




图5.12 两个游戏使用了平面着色作为风格化选择：上部的Kentucky Route Zero和下部的That Dragon, Cancer（上部图片来自Cardboard Computer，下部来自Numinous Games）。

This formulation fits the multi-light structure in Equation 5.6, repeated here for

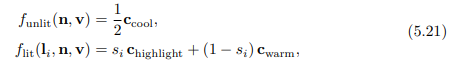
convenience：

制定符合公式5.6的多重光源结构，为了方便重述公式5.6：



The lit and unlit terms in this case are:

光照和无光的部分在这里是：



with the cool color’s unlit contribution adjusted to make results look more like the

original equation.

与冷色色调的无光照贡献调整，使结果看起来更像原始方程。

In most typical rendering applications, varying values for material properties such as

csurface would be stored in vertex data or, more commonly, in textures (Chapter 6).

However, to keep this example implementation simple, we will assume that csurface

is constant across the model.

在大多数渲染应用中，材质属性的不同值（比如csurface）可能储存在顶点数据中，或者更通常的，在纹理中（章节6）。然而，为了保证例子实现简单，我们假定csurface是一个常量值。

This implementation will use the shader’s dynamic branching capabilities to loop over

all light sources. While this straightforward approach can work well for reasonably

simple scenes, it does not scale well to large and geometrically complex scenes with

many light sources. Rendering techniques to efficiently handle large light counts will

be covered in Chapter 20. Also, in the interest of simplicity, we will only support one

type of light source: point lights. Although the implementation is quite simple, it

follows the best practices covered earlier.

这个实现将使用着色器动态分支能力去循环遍历所有的灯光源。尽管这种直接的方式在简单的场景可以合理的工作，但是它不能很好的缩放到大的和大量光源的几何复杂场景。有效的渲染大量光源的渲染技术参考20章。同样，为了简单起见，我们只支持其中一个光源类型：点光源。虽然实现非常简单，但是它遵循了前面介绍的最佳实践。

Shading models are not implemented in isolation, but in the context of a larger

rendering framework. This example is implemented inside a simple WebGL 2

application, modified from the “Phong-shaded Cube” WebGL 2 sample by Tarek

Sherif [1623], but the same principles apply to more complex frameworks as well.

着色器模型不是独立实现的，是在一个渲染框架中的上下文。这个例子是在简单的WebGL 2应用中实现的，由Tarek Sherif[1623]从phong着色立方体示例修改而来，但是同样的原则也适用于更复杂的框架。

We will be discussing some samples of GLSL shader code and JavaScript WebGL calls

from the application. The intent is not to teach the specifics of the WebGL API but to

show general implementation principles. We will go through the implementation in

“inside out” order, starting with the pixel shader, then the vertex shader, and finally

the application-side graphics API calls.

我们将会讨论一些来自应用程序GLSL着色器代码以及JavaScript WebGL的例子。其中目的不是教授WebGL API的细节，而是展示更一般的实现原则。我们将按照由内到外的顺序进行实现，首先是像素着色器，然后是顶点着色器，最后是应用程序端图形API调用。

Before the shader code proper, the shader source includes definitions of the shader

inputs and outputs. As discussed earlier in Section 3.3, using GLSL terminology,

shader inputs fall into two categories. One is the set of uniform inputs, which have

values set by the application and which remain constant over a draw call. The

second type consists of varying inputs, which have values that can change between

shader invocations (pixels or vertices). Here we see the definitions of the pixel

shader’s varying inputs, which in GLSL are marked in, as well as its outputs:

在着色器代码之前，在着色器开始要包括着色器输入和输出的定义。在章节3.3中讨论的，使用GLSL术语，着色器输入可以分为两类。其中之一是统一输入，由应用程序设置的值，在一个绘制命令中保持不变。第二种是由变化输入组成，在着色器（像素或者顶点）调用之间可以变化的值。这里看到像素着色器定义的变量，在GLSL中被标记出来，以及他的输出：



This pixel shader has a single output, which is the final shaded color. The pixel

shader inputs match the vertex shader outputs, which are interpolated over the

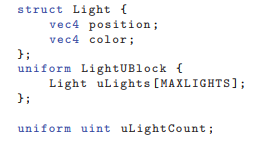
triangle before being fed into the pixel shader. This pixel shader has two varying

inputs: surface position and surface normal, both in the application’s world-space

coordinate system. The number of uniform inputs is much larger, so for brevity we

will only show the definitions of two, both related to light sources:

像素着色器有一个单一输出，最终的着色颜色。像素着色器的输入对应了顶点着色器的输出，这些值在传递到像素着色器之前通过三角形进行插值计算。像素着色器有两个变量输入：表面位置和表面法线，两个都是在应用世界空间坐标系统。统一输入的输入更大，为了简洁，我们将仅仅定义两个，两个都与光源有关：



Since these are point lights, the definition for each one includes a position and a

color. These are defined as vec4 instead of vec3 to conform to the restrictions of the

GLSL std140 data layout standard. Although, as in this case, the std140 layout can

lead to some wasted space, it simplifies the task of ensuring consistent data layout

between CPU and GPU, which is why we use it in this sample. The array of Light

structs is defined inside a named uniform block, which is a GLSL feature for binding a

group of uniform variables to a buffer object for faster data transfer. The array

length is defined to be equal to the maximum number of lights that the application

allows in a single draw call. As we will see later, the application replaces the

MAXLIGHTS string in the shader source with the correct value (10 in this case)

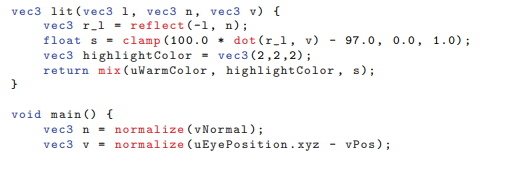
before shader compilation. The uniform integer uLightCount is the actual number of

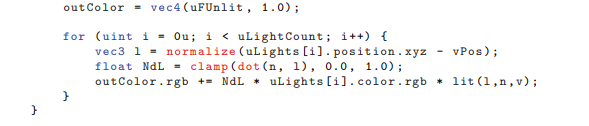
active lights in the draw call.

由于这些是点光源，每一个的定义都包括了位置和一个颜色。定义为vec4而不是vec3，是符合GLSL std140数据设计标准的限制。尽管，在这种情况下，std140设计可以导致一些浪费空间，他简化了CPU和GPU之间保持一致数据设计的任务，这也是这个例子我们使用的原因。光结构体的数组定义在一个统一块之内，在GLSL特征中绑定一组统一变量在一个缓冲对象中是为了更快的数据传输。数组长度定义为应用程序在单个绘制调用中允许最大的灯光数。稍后我们看到，应用程序在着色器编译之前用正确的值（本例中为10）替换着色器源码中的MAXLIGHTS字符串。统一整数uLightCount是绘制命令中调用活动灯光的实际数量。

Next, we will take a look at the pixel shader code:

接下来，我们将看一下像素着色器代码：





We have a function definition for the lit term, which is called by the main() function.

Overall, this is a straightforward GLSL implementation of Equations 5.20 and 5.21.

Note that the values of funlit() and cwarm are passed in as uniform variables. Since

these are constant over the entire draw call, the application can compute these

values, saving some GPU cycles.

我们有一个函数定义了光术语，会在main()函数中调用。总的来说，这是公式5.20和5.21中直接的GLSL实现。注意funlit()和cwarm的值是作为统一变量传递的。由于这些值在整个绘制调用期间都是常量，因此应用程序可以计算这些值，从而节省一些GPU周期。

This pixel shader uses several built-in GLSL functions. The reflect() function reflects

one vector, in this case the light vector, in the plane defined by a second vector, in

this case the surface normal. Since we want both the light vector and reflected

vector to point away from the surface, we need to negate the former before passing

it into reflect(). The clamp() function has three inputs. Two of them define a range

to which the third input is clamped. The special case of clamping to the range

between 0 and 1 (which corresponds to the HLSL saturate() function) is quick, often

effectively free, on most GPUs. This is why we use it here, although we only need to

clamp the value to 0, as we know it will not exceed 1. The function mix() also has

three inputs and linearly interpolates between two of them, the warm color and the

highlight color in this case, based on the value of the third, a mixing parameter

between 0 and 1. In HLSL this function is called lerp(), for “linear interpolation.”

Finally, normalize() divides a vector by its length, scaling it to a length of 1.

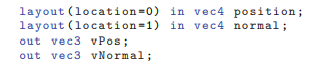
像素着色器使用了几个GLSL中建立的函数。Reflect()功能是反射一个向量，在这种情况下是光向量，在平面上被第二个向量定义，在这种情况下表面法线。由于我们需要光向量和和反射向量背离表面，我们需要在传递进Reflect()函数之前对光向量取负。Clamp()函数有3个输入。两个定义个第三个输入需要钳位的范围。特殊情况是钳位到范围0到1（相当于HLSL的saturate()函数），是非常快速的，通常有效无耗损的，在大多数GPU上。这也是为什么我们这里使用，尽管我们仅仅需要钳位到0，我们知道他们不可能超过1。函数mix()同样需要三个输入，在两者之间线性插值，在这种情况下暖色调颜色和高光颜色根据第三个值进行插值，混合参数在0到1之间。在HLSL中，这个函数称为lerp()，为了“线性插值”。最后，normalize()使一个向量除以他的长度，缩放到长度1。

Now let us look at the vertex shader. We will not show any of its uniform definitions

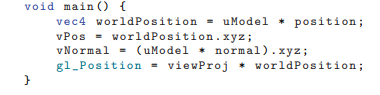
since we already saw some example uniform definitions for the pixel shader, but the

varying input and output definitions are worth examining:

现在，让我们看一下顶点着色器。由于我们已经在像素着色器看到的一些统一定义，这里我们将不再显示统一定义，但是变量输入和输出定义是值得注意的：



Note that, as mentioned earlier, the vertex shader outputs match the pixel shader varying inputs. The inputs include directives that specify how the data are laid out in the vertex array. The vertex shader code comes next:



注意，我们之前提到的，顶点着色器的输入与像素着色器变量输入匹配。输入包括制定数据如何在顶点数组中布局的指令。顶点着色器的代码如下。

These are common operations for a vertex shader. The shader transforms the

surface position and normal into world space and passes them to the pixel shader for

use in shading. Finally, the surface position is transformed into clip space and passed

into gl Position, a special system-defined variable used by the rasterizer. The gl

Position variable is the one required output from any vertex shader.

这是在顶点着色器中的常用操作。着色器转换表面位置和法线到世界空间并传递到像素着色器用来着色。最后，表面位置转换到裁减空间并传递到gl位置，一个被光栅化特殊系统定义的变量。Gl位置变量是任何顶点着色器所需的输出。

Note that the normal vectors are not normalized in the vertex shader. They do not

need to be normalized since they have a length of 1 in the original mesh data and

this application does not perform any operations, such as vertex blending or

nonuniform scaling, that could change their length unevenly. The model matrix could

have a uniform scale factor, but that would change the length of all normals

proportionally and thus not result in the problem shown on the right side of Figure

5.10.

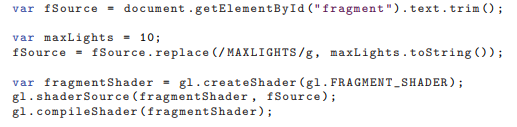
注意在顶点着色器中并没有对法线向量归一化。他们不需要被归一化，因为他们在原本的网格数据中就是单位1，而且应用并没有执行任何操作，比如顶点混合或者非均匀缩放，可能会非均衡的改变他们的长度。模型矩阵可能有一个统一缩放因子，但是会等比例改变法线的长度，从而不会导致图5.10右侧所示的问题。

The application uses the WebGL API for various rendering and shader setup. Each of

the programmable shader stages are set up individually, and then they are all bound

to a program object. Here is the pixel shader setup code:

应用使用WebGL API来各种渲染以及着色器设置。每个可编程着色器阶段单独设置，然后他们都绑定到一个程序对象。下面是像素着色器设置代码：



Note the “fragment shader” references. This term is used by WebGL (and OpenGL,

on which it is based). As noted earlier in this book, although “pixel shader” is less

precise in some ways, it is the more common usage, which we follow in this book.

This code is also where the MAXLIGHTS string is replaced with the appropriate

numerical value. Most rendering frameworks perform similar pre-compilation shader

manipulations.

注意“片段着色器”引用。术语是用在WebGL（基于OpenGL）。在书中之前提到的，尽管“像素着色器”以某种方式不够精确，他只是常用，以及在本书的后面。代码也以合适的数值替换了MAXLIGHTS字段。大部分渲染框架执行相同的逐编译着色器操作。

There is more application-side code for setting uniforms, initializing vertex arrays,

clearing, drawing, and so on, which you can view in the program [1623] and which

are explained by numerous API guides. Our goal here is to give a sense of how

shaders are treated as separate processors, with their own programming

environment. We thus end our walkthrough at this point.

这里有更多的应用阶段的代码设置统一变量，初始化顶点数组，清空缓冲，绘制等，你将在程序[1623]中看到，被很多API指南解释。我们这里的目标是让你了解如何将着色器作为单独的处理器来处理，并使用他们自己的编程环境。至此，我们将结束我们的演练。

5.3.3 Material Systems 材质系统

Rendering frameworks rarely implement just a single shader, as in our simple

example. Typically, a dedicated system is needed to handle the variety of materials,

shading models, and shaders used by the application.

渲染框架很少在一个着色器中实现，正如在我们简单的例子中。典型的，一个专用的系统需要去管理材质、着色器模型，以及应用程序所用的着色器的变化。

As explained in earlier chapters, a shader is a program for one of the GPU’s

programmable shader stages. As such, it is a low-level graphics API resource and not

something with which artists would interact directly. In contrast, a material is an

artist-facing encapsulation of the visual appearance of a surface. Materials

sometimes also describe non-visual aspects, such as collision properties, which we

will not discuss further because they are outside the scope of this book.

正如之前章节解释的，一个着色器是GPU可编程着色器阶段的一个程序。同样，他是一个低级的图形API资源，而不是艺术家可以直接与之交互的东西。相比之下，材质是表面视觉外观的一种个面向艺术家的封装。材料有时也描述非视觉方面，如碰撞属性，我们将不再进一步讨论，因为他们超出了本书的范围。

While materials are implemented via shaders, this is not a simple one-to-one

correspondence. In different rendering situations, the same material may use

different shaders. A shader can also be shared by multiple materials. The most

common case is parameterized materials. In its simplest form, material

parameterization requires two types of material entities: material templates and

material instances. Each material template describes a class of materials and has a

set of parameters that can be assigned numerical, color, or texture values depending

on the parameter type. Each material instance corresponds to a material template

plus a specific set of values for all of its parameters. Some rendering frameworks

such as the Unreal Engine [1802] allow for a more complex, hierarchical structure,

with material templates deriving from other templates at multiple levels.

材质通过着色器实现，但这不是一个简单的一对一的对应。在不同的渲染情况，相同材质可能使用不同的着色器。一个着色器也可以被多重材质共享。最常用的情况是参数化的材质。在他最简单的形式，材质参数化需要两种材质实体：材质模板和材质实例。每一个材质模板描述了一种材质和一系列的参数可以根据参数类别分配数值，颜色或者纹理。每一个材质实例对应于一个材质模板及其所有参数的特定值集。一些渲染框架比如虚幻引擎[1802]允许更复杂的层次结构，材质模板从其他模板派生到多个级别。

Parameters may be resolved at runtime, by passing uniform inputs to the shader

program, or at compile time, by substituting values before the shader is compiled. A

common type of compile-time parameter is a boolean switch that controls the

activation of a given material feature. This can be set by artists via a checkbox in the

material user interface or procedurally by the material system, e.g., to reduce shader

cost for distant objects where the visual effect of the feature is negligible.

参数可以在运行时被解决，通过传递统一输入到着色器程序，或者在编译时间，通过在着色器编译之前取代值。通常编译时间参数类型是布尔值开关，他控制给定材质特性的激活。这可以由艺术家通过材质用户界面中的复选框来设置，也可以由材质系统进行程序设置，例如，为那些视觉效果可以忽略的遥远对象来降低着色器消耗。

While the material parameters may correspond one-to-one with the parameters of

the shading model, this is not always the case. A material may fix the value of a

given shading model parameter, such as the surface color, to a constant value.

Alternately, a shading model parameter may be computed as the result of a complex

series of operations taking multiple material parameters, as well as interpolated

vertex or texture values, as inputs. In some cases, parameters such as surface

position, surface orientation, and even time may also factor into the calculation.

Shading based on surface position and orientation is especially common in terrain

materials. For example, the height and surface normal can be used to control a snow

effect, blending in a white surface color on high-altitude horizontal and almost-

horizontal surfaces. Time-based shading is common in animated materials, such as a

flickering neon sign.

尽管材质的参数可以与着色器模型的参数一一对应，但并不总是这种情况。材质可以固定一个给定着色器模型参数的值，比如表面颜色，设定为固定的值。另一方面，着色器模型参数可以通过一系列复杂的操作来计算，这些操作将多个材质参数以及插值顶点或纹理值作为输入。在某些情况下，表面位置，表面方向甚至是时间等参数也可以成为计算的因素。基于表面位置和方向的着色在地形材质中尤其常见。例如，高度和表面法线可以用来控制制造雪的特效，在高海拔水平和接近水平的表面混合白色的表面颜色。基于时间的着色在动画材质中很常见，比如闪烁的霓虹灯。

One of the most important tasks of a material system is dividing various shader

functions into separate elements and controlling how these are combined. There are

many cases where this type of composition is useful, including the following:

• Composing surface shading with geometric processing, such as rigid transforms,

vertex blending, morphing, tessellation, instancing, and clipping. These bits of

functionality vary independently: Surface shading depends on the material, and

geometry processing depends on the mesh. So, it is convenient to author them

separately and have the material system compose them as needed.

• Composing surface shading with compositing operations such as pixel discard and

blending. This is particularly relevant to mobile GPUs, where blending is typically

performed in the pixel shader. It is often desirable to select these operations

independently of the material used for surface shading.

• Composing the operations used to compute the shading model parameters with the

computation of the shading model itself. This allows authoring the shading model

implementation once and reusing it in combination with various different methods for

computing the shading model parameters.

• Composing individually selectable material features with each other, the selection

logic, and the rest of the shader. This enables writing the implementation of each

feature separately.

• Composing the shading model and computation of its parameters with light source

evaluation: computing the values of clight and l at the shaded point for each light

source. Techniques such as deferred rendering (discussed in Chapter 20) change the

structure of this composition. In rendering frameworks that support multiple such

techniques, this adds an additional layer of complexity.

材质系统最重要的任务之一是将各种着色器功能划分为单独的元素，并控制这些元素如何组合。在很多情况下这种组合是有用的，包括如下：

•组合表面着色与几何处理，比如严格的转换，顶点混合，变形，曲面细分，实例和裁减。这些功能独立变化：表面着色取决于材质，几何体处理取决于网格。因此可以方便的单独编写，并根据需要由材质系统组成。

•使用合成操作合成表面着色，如像素丢弃和混合。这与移动GPU特别相关，其中混合通常在像素着色器中执行。选择这些操作通常是可取的，独立于用于表面着色的材质。

•通过着色器模型自身的计算组合操作用来计算着色模型参数。这允许制作一次着色器模型的实现并且联合多种不同方法重复使用来计算着色模型参数。

•互相组合各自可选择的材质特性，选择逻辑和着色器的其他部分。这能够编写每个特性的实现。

•用光源模型评估组合着色模型以及其参数计算：为每一个光源计算clight值以及着色点的l。比如延迟渲染技术（20章将讨论）改变了这个组合的结构。在渲染支持多种此类技术的框架时，这又增加了一层复杂性。

It would be convenient if the graphics API provided this type of shader code

modularity as a core feature. Sadly, unlike CPU code, GPU shaders do not allow for

post-compilation linking of code fragments. The program for each shader stage is

compiled as a unit. The separation between shader stages does offer some limited

modularity, which somewhat fits the first item on our list: composing surface shading

(typically performed in the pixel shader) with geometric processing (typically

performed in other shader stages). But the fit is not perfect, since each shader

performs other operations as well, and the other types of composition still need to be

handled. Given these limitations, the only way that the material system can

implement all these types of composition is at the source-code level. This primarily

involves string operations such as concatenation and replacement, often performed

via C-style preprocessing directives such as #include, #if, and #define.

如果图形API提供了这种类型的着色器代码模块化作为核心特性，将会非常方便。悲伤的是，不像CPU代码，GPU着色器不允许代码片段的编译后链接。每个着色器阶段的程序作为一个单元编译。着色器阶段之间的分离确实提供了一些有限的模块性，这也在某种程度上符合我们列表中的第一项：用几何处理（通常在其他着色器阶段执行）组合表面着色（通常在像素着色器中执行）。但是这种匹配并不完美，因为每个着色器都执行了其他操作，而且其他类型的组合仍然需要处理。考虑到这些限制，材质系统实现所有的这类类型的组合的唯一方法是在源码级别。这主要包括了字符操作比如连接和替换，通常通过C风格的预处理指令执行，如#include, #if, 和#define。

Early rendering systems had a relatively small number of shader variants, and often

each one was written manually. This has some benefits. For example, each variant

can be optimized with full knowledge of the final shader program. However, this

approach quickly becomes impractical as the number of variants grows. When taking

all the different parts and options into account, the number of possible different

shader variants is huge. This is why modularity and composability are so crucial.

早期的渲染系统有相对较少的着色器变种，而且通常每个变种都是手工编写的。这有一些益处。比如，每个变种都可以在完全了解最终着色器程序的情况下进行优化。然而，随着变种数量的增加，这种方法很快变得不切实际。当考虑到所有不同的部分和选项时，可能的不同着色器变种的数量是巨大的。这也是模块化和组合性如此重要的原因。

The first question to be resolved when designing a system for handling shader

variants is whether selection between different options is performed at runtime via

dynamic branching, or at compile time via conditional preprocessing. On older

hardware, dynamic branching was often impossible or extremely slow, so runtime

selection was not an option. Variants were then all handled at compile time,

including all possible combinations of counts of the different light types [1193].

第一个需要解决的问题是当设计一个处理着色器变种系统时是通过运行时的动态批处理来选择不同选项的执行还是在编译阶段通过条件处理。在老的硬件上，动态批处理是不可能的或者及其慢的，因此运行时不是一个选择。变种都是在编译时间内处理的，包括所有可能的大量的不同类型的光组合[1193]。

In contrast, current GPUs handle dynamic branching quite well, especially when the

branch behaves the same for all pixels in a draw call. Today much of the

functionality variation, such as the number of lights, is handled at runtime. However,

adding a large amount of functional variation to a shader incurs a different cost: an

increase in register count and a corresponding reduction in occupancy, and thus

performance. See Section 18.4.5 for more details. So, compile-time variation is still

valuable. It avoids including complex logic that will never be executed.

相比之下，当前GPU处理动态分支非常好，特别是当分支在一个绘制命令中的所有像素表现相同时。现在，很多功能变化，比如灯光数量，都是在运行中处理的。然而，向着色器添加大量的功能变化会带来不同的成本：寄存器数量的增加和相应占有率的减少，从而降低性能。章节18.4.5有更详细的介绍。因此，编译时间变种仍然是有价值的。它避免包括永远不会执行的复杂逻辑。

As an example, let us imagine an application that supports three different types of

lights. Two light types are simple: point and directional. The third type is a

generalized spotlight that supports tabulated illumination patterns and other complex

features, requiring a significant amount of shader code to implement. However, say

the generalized spotlight is used relatively rarely, with less than 5% of the lights in

the application being this type. In the past, a separate shader variant would be

compiled for each possible combination of counts of the three light types, to avoid

dynamic branching. While this would not be needed today, it may still be beneficial

to compile two separate variants, one for the case when the count of generalized

spotlights is equal to or greater than 1, and one for the case where the count of such

lights is exactly 0. Due to its simpler code, the second variant (which is most

commonly used) is likely to have lower register occupancy and thus higher

performance.

举个例子，让我们想象一个应用支持三种不同类型的光。两种简单类型：点和平行光。第三种是一个泛化聚光灯支持平板照明模式和其他复杂特性，需要大量的着色器代码去执行。然而，泛化聚光灯使用相对较少，在应用中占所有灯光不足5%。在过去，会为三种光的每种可能的组合编译一个单独的着色器变种，以免动态分支。这在今天是不需要的，但是编译两个单独的变种仍然是有益的，一个用于泛化等数等于或大于1的情况，另一个用于泛化灯数恰好为0的情况。由于代码更简单，第二种变种（最常用）可能具有更低的寄存器占有率，从而具有更高的性能。

Modern material systems employ both runtime and compile-time shader variation.

Even though the full burden is no longer handled only at compile time, the overall

complexity and number of variations keep increasing, so a large number of shader

variants still need to be compiled. For example, in some areas of the game Destiny:

The Taken King, over 9000 compiled shader variations were used in a single frame

[1750]. The number of possible variations can be much larger, e.g., the Unity

rendering system has shaders with close to 100 billion possible variants. Only the

variants that are actually used are compiled, but the shader compilation system had

to be redesigned to handle the huge number of possible variants [1439].

现代材质系统有运行时和编译时间两种着色器变种。尽管全部的负担不再只在编译时处理，总体的复杂性和变种的数量仍在增加，所以大量的着色器变种仍然需要在编译时解决。例如，在《命运之神：夺去的国王》的一些领域里，在一帧中，超过9000编译的着色器变种被使用[1750]。可能的变化可能更多，例如，Unity渲染系统有接近1000亿个可能的变化着色器。只有实际使用的变种才会被编译，但是着色器编译系统必须重新设计以处理大量可能的变种[1439]。

Material-system designers employ different strategies to address these design goals.

Although these are sometimes presented as mutually exclusive system architectures

[342], these strategies can be—and usually are—combined in the same system.

These strategies include the following:

• Code reuse—Implementing functions in shared files, using #include preprocessor

directives to access those functions from any shader that needs them.

• Subtractive—A shader, often referred to as an ¨ubershader or supershader [1170,

1784], that aggregates a large set of functionality, using a combination of

compiletime preprocessor conditionals and dynamic branching to remove unused

parts and to switch between mutually exclusive alternatives.

• Additive—Various bits of functionality are defined as nodes with input and output

connectors, and these are composed together. This is similar to the code reuse

strategy but is more structured. The composition of nodes can be done via text [342]

or a visual graph editor. The latter is intended to make it easier for non-engineers,

such as technical artists, to author new material templates [1750, 1802]. Typically

only part of the shader is accessible to visual graph authoring. For example, in the

Unreal Engine the graph editor can only affect the computation of shading model

inputs [1802]. See Figure 5.13.

• Template-based—An interface is defined, into which different implementations can

be plugged as long as they conform to that interface. This is a bit more formal than

the additive strategy and is typically used for larger chunks of functionality. A

common example for such an interface is the separation between the calculation of

shading model parameters and the computation of the shading model itself. The

Unreal Engine [1802] has different “material domains,” including the Surface domain

for computing shading model parameters and the Light Function domain for

computing a scalar value that modulates clight for a given light source. A similar

“surface shader” structure also exists in Unity [1437]. Note that deferred shading

techniques (discussed in Chapter 20) enforce a similar structure, with the G-buffer

serving as the interface.

材质系统设计师采用不同的策略来实现这些设计目的。尽管这些策略有时表现为相互排斥的系统体系结构[342]，但他们可以——而且通常是——在同一个系统中组合使用。这些战略包括：

•代码复用——在共享文件中实现函数，任何着色器需要他们都可以用#include预处理指令去访问这些函数。

•减法——一个着色器，常常称为一个“父类着色器”或“超类着色器”[1170,1784]，聚集了很大的功能，使用编译时预处理条件和动态分支去移除掉未使用的部分，并在互斥替代之间切换。

•加法——各种功能位被定义为带有输入和输出连接器的节点，这些节点被组合在一起。这类似于代码复用策略，但是更加结构化。节点的组合可以通过文本[342]或可视化图形编辑器来完成。后者的目的是非工程师，如技术美术，更容易编写新的材质模板[1750,1802]。通常，只有部分着色器可用于可视化图形创作。例如，在虚幻引擎中，图形编辑器只能影响着色模型的计算[1802]。参考图5.13。

•基于模板——只要不同的实现符合该接口，就可以将它们插入其中。这比加法策略更正式一些，通常用于更大的功能块。这种接口的常用例子是分离着色器模型参数的计算和着色器模型本身的计算。虚幻引擎[1802]有不同的“材质域”，包括用于计算着色器模型参数的表面域和用于计算为给定光源调制clight标量值的光函数域。Unity中也存在类似的“表面着色器”结构。注意延迟着色技术（在第20章讨论）强制执行类似的结构，使用G-缓冲区作为接口。

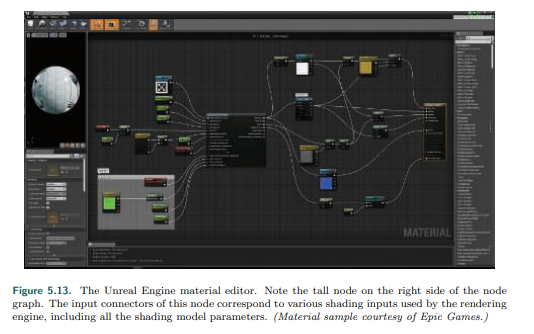


图5.13 虚幻引擎材质编辑器。注意节点图中右侧的高节点。此节点的输入连接器对应于各种渲染引擎使用的着色输入，包括所有的着色模型参数（材质样本由epic games提供）。

For more specific examples, several chapters in the (now free) book WebGL Insights

[301] discuss how a variety of engines control their shader pipelines. Besides

composition, there are several other important design considerations for modern

material systems, such as the need to support multiple platforms with minimal

duplication of shader code. This includes variations in functionality to account for

performance and capability differences among platforms, shading languages, and

APIs. The Destiny shader system [1750] is a representative solution to this type of

problem. It uses a proprietary preprocessor layer that takes shaders written in a

custom shading language dialect. This allows writing platform-independent materials

with automatic translation to different shading languages and implementations. The

Unreal Engine [1802] and Unity [1436] have similar systems.

更多特殊的例子，书籍（现在是免费的）WebGL Insights中几个章节讨论了各种引擎如何控制他们的着色器管道。除了组合，现代材质系统也有几种其他的重要设计理念，比如需要支持多个平台，同时最小化着色器代码的重复。这包括考虑平台、着色器语言和api之间性能和功能差异的功能变化。命运着色器系统有这种问题的典型解决方案。他使用一个专有的预处理层，该层使用用自定义着色语言编写的着色器。这允许编写独立于平台的材质，并自动转换为不同的着色语言和实现。虚幻引擎[1802]和Unity[1436]有相似的系统。

The material system also needs to ensure good performance. Besides specialized

compilation of shading variants, there are a few other common optimizations the

material system can perform. The Destiny shader system and the Unreal Engine

automatically detect computations that are constant across a draw call (such as the

warm and cool color computation in the earlier implementation example) and move it

outside of the shader. Another example is the scoping system used in Destiny to

differentiate between constants that are updated at different frequencies (e.g., once

per frame, once per light, once per object) and update each set of constants at the

appropriate times to reduce API overhead.

材质系统同样需要确保好的性能。除了专业的着色变种的编译，也有一些其他的共同的材质系统优化可以执行。命运着色器系统和虚幻引擎自动检测一个绘制命令中的常量计算（比如之前实现的例子中的暖和冷色调的计算）并他们移到着色器的外边。另一个例子是命运中使用的作用域系统，用于区分以不同频率更新的常量（例如，每帧一次，每光源一次，每对象一次），并在适当的时间更新每一组常量，以减少API开销。

As we have seen, implementing a shading equation is a matter of deciding what

parts can be simplified, how frequently to compute various expressions, and how the

user is able to modify and control the appearance. The ultimate output of the

rendering pipeline is a color and blend value. The remaining sections on antialiasing,

transparency, and image display detail how these values are combined and modified

for display.

正如我们之前看到的，实现着色方程是一个决定哪些部分可以简化，计算各种表达式的频率，以及用户如何修改和控制外观的问题。渲染管道的最终输出是一个颜色和混合值。关于抗锯齿、透明度和图像显示的其余部分详细介绍了如何组合和修改这些值以便显示。

5.4 Aliasing and Antialiasing 混叠和抗锯齿

Imagine a large black triangle moving slowly across a white background. As a screen

grid cell is covered by the triangle, the pixel value representing this cell should

smoothly drop in intensity. What typically happens in basic renderers of all sorts is

that the moment the grid cell’s center is covered, the pixel color immediately goes

from white to black. Standard GPU rendering is no exception. See the leftmost

column of Figure 5.14.

想象一下，一个大的黑色三角形在一个白色背景下缓慢移动。屏幕网格被三角形覆盖，表示该单元的像素值应该平滑地降低强度。在各种基本渲染器中通常发生的情况是，当网格单位格的中心被覆盖时，像素颜色立即从白色变为黑色。标准GPU渲染也不例外。参见图5.14中最左边的一列。

Triangles show up in pixels as either there or not there. Lines drawn have a similar

problem. The edges have a jagged look because of this, and so this visual artifact is

called “the jaggies,” which turn into “the crawlies” when animated. More formally,

this problem is called aliasing, and efforts to avoid it are called antialiasing

techniques.

三角形以像素表示，要么在这里，要么不在那里。画的线也有类似的问题。因为边缘有锯齿状的外观，所以这个视觉外观称为“锯齿”，当动画时就变成了“小爬虫”。更正式的说，这个问题称为锯齿，而努力去避免他的技术称为抗锯齿技术。

The subject of sampling theory and digital filtering is large enough to fill its own

book [559, 1447, 1729]. As this is a key area of rendering, the basic theory of

sampling and filtering will be presented. We will then focus on what currently can be

done in real time to alleviate aliasing artifacts.

采样理论和数字滤波这门学科的内容非常丰富，足以自称体系[559,1447,1729]。由于这是渲染的一个关键领域，我们将介绍采样和滤波的基本理论。然后我们将关注当前可以实时做什么来减轻锯齿工作。

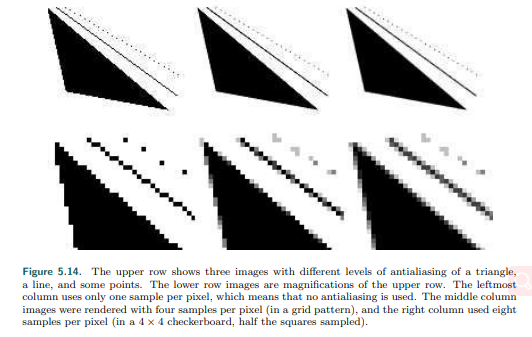


图5.14 上面一行显示了三角形，直线和一些点的不同级别抗锯齿。下面一行是上面行的放大品。最左侧的列每个像素只使用一个样本，这意味着不使用抗锯齿。中间一列图像每个像素有4个采样渲染（在网格模式中），右侧列使用了每个像素8个采样（在一个4x4的棋盘格中，采样的正方形的一半）。

5.4.1 Sampling and Filtering Theory 采用和滤波理论

The process of rendering images is inherently a sampling task. This is so since the

generation of an image is the process of sampling a three-dimensional scene in order

to obtain color values for each pixel in the image (an array of discrete pixels). To use

texture mapping (Chapter 6), texels have to be resampled to get good results under

varying conditions. To generate a sequence of images in an animation, the animation

is often sampled at uniform time intervals. This section is an introduction to the topic

of sampling, reconstruction, and filtering. For simplicity, most material will be

presented in one dimension. These concepts extend naturally to two dimensions as

well, and can thus be used when handling two-dimensional images.

渲染图像的过程是固定的采样任务。这是因为图像的生成是对三维场景进行采样的过程，以便获得图像中每个像素（离散像素数组）的颜色值。要使用纹理映射（第6章），必须对纹理进行重采样，以便在不同的条件下获得良好的结果。为了在动画中生成一个图像序列，动画通常以均匀的时间间隔采样。本节主要介绍采样、重构和过滤的主题。为了简单起见，大多数材料将以一维的形式呈现。这些概念自然扩展到二维，因此可以用来处理二维图像。

Figure 5.15 shows how a continuous signal is being sampled at uniformly spaced

intervals, that is, discretized. The goal of this sampling process is to represent

information digitally. In doing so, the amount of information is reduced. However,

the sampled signal needs to be reconstructed to recover the original signal. This is

done by filtering the sampled signal.

图5.15展示了连续信号如何以均匀间隔采样，即离散化。这个抽样过程的目标是用数字表示信息。这样做会减少信息量。然而，需要对采样信号进行重构才能恢复原始信号。这是通过过滤采样信号来实现的。

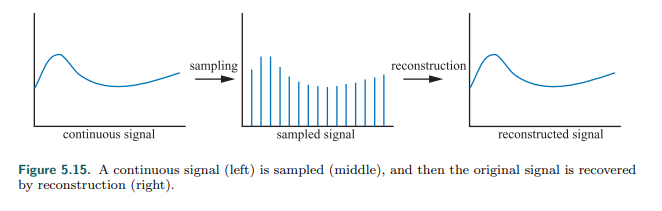


图5.15 连续信号（左侧）被采样（中间），然后原始的信号由重构恢复（右侧）。

Whenever sampling is done, aliasing may occur. This is an unwanted artifact, and we

need to battle aliasing to generate pleasing images. A classic example of aliasing

seen in old Westerns is a spinning wagon wheel filmed by a movie camera. Because

the spokes move much faster than the camera records images, the wheel may

appear to be spinning slowly (backward or forward), or may even look like it is not

rotating at all. This can be seen in Figure 5.16. The effect occurs because the images

of the wheel are taken in a series of time steps, and is called temporal aliasing.

当采样完成，混叠可能产生。这是一个不需要的制品，我们需要对抗混叠以生成令人满意的图像。在旧西部片中出现的一个典型的混叠例子是用电影摄像机拍摄的一个旋转的车轮。由于辐条的运动比相机记录的图像快的多，轮子可能看起来旋转的旋转的很慢（向后或者向前），甚至可能看起来根本不旋转。这可以在图5.16中看到。产生这种效果的原因是车轮的图像是在一系列的时间步骤中拍摄的，称为时间混叠。

Common examples of aliasing in computer graphics are the “jaggies” of a rasterized

line or triangle edge, flickering highlights known as “fireflies”, and when a texture

with a checker pattern is minified (Section 6.2.2).

计算机图形中常见的混叠例子是光栅化的“锯齿”线条或三角形边缘，闪烁的高光称为“萤火虫”，并当纹理使用检查器模式可以缩小（章节6.2.2）。

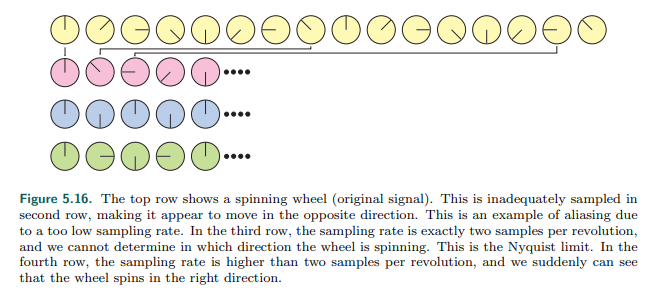


图5.16 最高处一行显示了快速旋转的轮子（原始信号）。第二行是不恰当的采样，使他出现了向相反方向移动的情况。这是一个由于采样率太低的混叠例子。第三行，采样率正好是每转两圈，我们无法确定轮子在向哪个方向旋转。这是Nyquist极限。在第四行，每转一圈的采样率大于两个，我们突然可以看到轮子在正确的方向旋转。

Aliasing occurs when a signal is being sampled at too low a frequency. The sampled

signal then appears to be a signal of lower frequency than the original. This is

illustrated in Figure 5.17. For a signal to be sampled properly (i.e., so that it is

possible to reconstruct the original signal from the samples), the sampling frequency

has to be more than twice the maximum frequency of the signal to be sampled. This

is often called the sampling theorem, and the sampling frequency is called the

Nyquist rate [1447] or Nyquist limit, after Harry Nyquist (1889–1976), a Swedish

scientist who discovered this in 1928. The Nyquist limit is also illustrated in Figure

5.16. The fact that the theorem uses the term “maximum frequency” implies that the

signal has to be band-limited, which just means that there are not any frequencies

above a certain limit. Put another way, the signal has to be smooth enough relative

to the spacing between neighboring samples.

采样频率太低导致了混叠的产生。采样后的信号出现了比原有信息更低的频率。见插图5.17。要正确的采样信号（即，这样有可能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。这通常称为抽样定理，抽样频率称为Nyquist速率[1447]或Nyquist极限，以1928年发现这一点的瑞典科学家Harry Nyquist（1889-1976）i命名。Nyquist极限如图5.16所示。该定律使用“最大频率”这一术语的事实意味着信号必须是带限的，这意味着没有任何频率高于某个极限。换句话说，相对于相邻样本之间的距离，信号必须足够平滑。

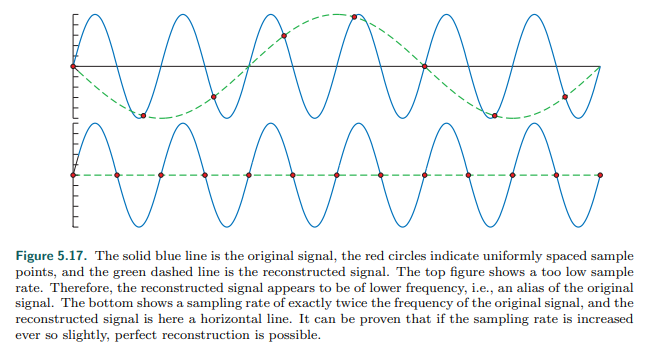


图5.17 蓝色实线是原始信号，红色圆圈表示均匀间隔的样本点，绿色虚线为重构信号。上部图片显示了一个太低的采样速率。因此，重构信号的频率较低，即一个原始信号的别名。底部显示的采样率正好是原始信号频率的两倍，重建信号在这里是一条水平线。可以证明，如果采样率增加即使那么一点点，完美的重建也是有可能的。

A three-dimensional scene is normally never band-limited when rendered with point

samples. Edges of triangles, shadow boundaries, and other phenomena produce a

signal that changes discontinuously and so produces frequencies that are infinite

[252]. Also, no matter how closely packed the samples are, objects can still be small

enough that they do not get sampled at all. Thus, it is impossible to entirely avoid

aliasing problems when using point samples to render a scene, and we almost

always use point sampling. However, at times it is possible to know when a signal is

band-limited. One example is when a texture is applied to a surface. It is possible to

compute the frequency of the texture samples compared to the sampling rate of the

pixel. If this frequency is lower than the Nyquist limit, then no special action is

needed to properly sample the texture. If the frequency is too high, then a variety of

algorithms are used to band-limit the texture (Section 6.2.2).

在使用点采样渲染三维场景时，通常不会带限。三角形的边缘、阴影的边缘和其他现象会产生无穷大的频率[252]。此外，无论样本的排列有多紧密，物体仍然可以小到根本不会被采样。因此，在使用点采样渲染场景时，不可能完全避免混叠问题，而且我们总是使用点采样。然而有时我们可以知道什么时候信号是带限的。一个例子就是当纹理应用到表面时。可以将纹理采样的频率与像素的采样率进行比较计算。如果这个频率低于Nyquist极限，那么就不需要特殊的操作来正确采样纹理。如果频率过高，则使用多种算法对纹理进行带限（章节6.2.2）。

Reconstruction 重构

Given a band-limited sampled signal, we will now discuss how the original signal can

be reconstructed from the sampled signal. To do this, a filter must be used. Three

commonly used filters are shown in Figure 5.18. Note that the area of the filter

should always be one, otherwise the reconstructed signal can appear to grow or

shrink.

给定一个带限的采样信号，我们将讨论如何从采样信号重构原始信号。为了达到目的，需要用到过滤。图5.18显示了三个常用的过滤器。注意，过滤器的面积应该始终为1，否则重构的信号可能出现增长或收缩。

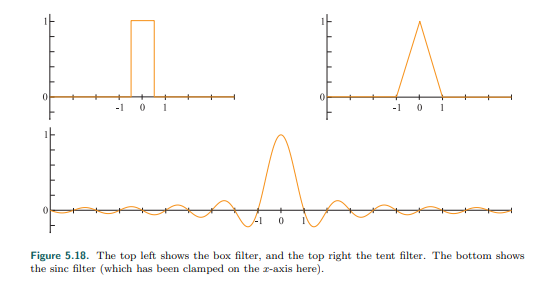


图5.18 上部左侧显示了盒型滤波器，上部右侧是帐篷滤波器。底部显示了正弦滤波器（这里已经钳位到x轴）。

In Figure 5.19, the box filter (nearest neighbor) is used to reconstruct a sampled

signal. This is the worst filter to use, as the resulting signal is a noncontinuous stair

case. Still, it is often used in computer graphics because of its simplicity. As can be

seen in the illustration, the box filter is placed over each sample point, and then

scaled so that the topmost point of the filter coincides with the sample point. The

sum of all these scaled and translated box functions is the reconstructed signal

shown to the right.

在图5.19中，盒型滤波器（最近邻）用来重构一个采样信号。这是一个最坏的滤波器，结果信号是一个间断的阶梯状情况。但是因为他的简单性，常常用在计算机图形上。可以在插图上看到的是，将盒型滤波器放置在每个采样点上，然后进行缩放，使滤波器的顶点与采样点重合。所有经过缩放和平移的盒型函数的和就是右边显示的重构信号。

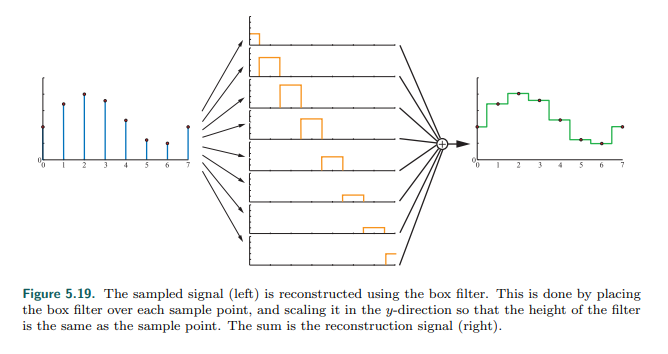


图5.19 采样信号（左侧）使用盒型滤波器进行重构。通过在每个采样点放置盒型滤波器，在y方向缩放使滤波器的高度与采样点一样高。总和就是重构后的信号（右侧）。

The box filter can be replaced with any other filter. In Figure 5.20, the tent filter,

also called the triangle filter, is used to reconstruct a sampled signal. Note that this

filter implements linear interpolation between neighboring sample points, and so it is

better than the box filter, as the reconstructed signal now is continuous.

盒型滤波器可以被任何其他的滤波器替换。在图5.20中，帐篷滤波器，同样称为三角形滤波器，用来重构一个采样信号。注意这个滤波器在相邻的采样点执行了线性插值，由于重构后的信号是连续的，所以它比盒型滤波器更好。

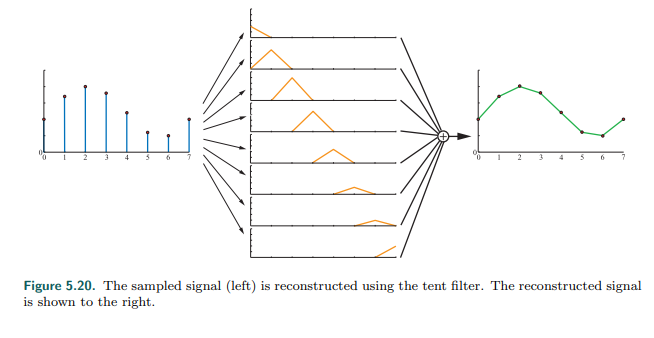


图5.20 采样信号（左侧）使用帐篷滤波器进行重构。重构后的信号显示在右侧。

However, the smoothness of the reconstructed signal using a tent filter is poor; there

are sudden slope changes at the sample points. This has to do with the fact that the

tent filter is not a perfect reconstruction filter. To get perfect reconstruction the ideal

low-pass filter has to be used. A frequency component of a signal is a sine wave:

sin(2πf), where f is the frequency of that component. Given this, a low-pass filter

removes all frequency components with frequencies higher than a certain frequency

defined by the filter. Intuitively, the low-pass filter removes sharp features of the

signal, i.e., the filter blurs it. The ideal low-pass filter is the sinc filter (Figure 5.18

bottom):



然而，使用帐篷滤波器重构的信号平滑性比较弱；在采样点有一个突然的倾斜改变。这与帐篷过滤器不是一个完美重构滤波器有关。为了得到完美的重构，必须使用理想的低通道滤波器。信号的滤波分量是正弦波：sin(2πf)，f是频率分量。在此情况下，低通道滤波器去除高于滤波器定义的特定频率的频率分量。直观的说，低通道滤波器去除信号的尖锐特征，即，过滤器使它变得模糊。理想的低通道滤波器是sin滤波器（图5.18底部）。

The theory of Fourier analysis [1447] explains why the sinc filter is the ideal lowpass

filter. Briefly, the reasoning is as follows. The ideal low-pass filter is a box filter in the

frequency domain, which removes all frequencies above the filter width when it is

multiplied with the signal. Transforming the box filter from the frequency domain to

the spatial domain gives a sinc function. At the same time, the multiplication

operation is transformed into the convolution function, which is what we have been

using in this section, without actually describing the term.

傅里叶分析理论[1447]解释了为什么正弦率滤波器是最理想的低通道滤波器。简明的说，理由如下。理想的低通道滤波器是频域内的盒型滤波器，当他与信号相乘时，移除滤波器宽度以上的所有频率。将盒型滤波器从频域转换到空间域，得到了一个正弦函数。同时，乘法运算被转换成了卷积函数，这是我们在这一节中一直使用的，没有实际描述这个术语。

Using the sinc filter to reconstruct the signal gives a smoother result, as shown in

Figure 5.21. The sampling process introduces high-frequency components (abrupt

changes) in the signal, and the task of the low-pass filter is to remove these. In fact,

the sinc filter eliminates all sine waves with frequencies higher than 1/2 the sampling

rate. The sinc function, as presented in Equation 5.22, is the perfect reconstruction

filter when the sampling frequency is 1.0 (i.e., the maximum frequency of the

sampled signal must be smaller than 1/2). More generally, assume the sampling

frequency is fs, that is, the interval between neighboring samples is 1/fs. For such a

case, the perfect reconstruction filter is sinc(fsx), and it eliminates all frequencies

higher than fs/2. This is useful when resampling the signal (next section). However,

the filter width of the sinc is infinite and is negative in some areas, so it is rarely

useful in practice.

如图5.21所示，使用正弦滤波器重构信号可以得到一个平滑地结果。采样处理会在信号上引入了高频率分量（突然的改变），低通道滤波的任务就是移除这些。实际上，正弦滤波器消除了频率高于采样率1/2的所有正弦波。如式5.22所示的正弦函数，当采样频率为1.0是完美的重构滤波（即，采样信号的最大频率必须小于1/2）。更一般的是，假定采样频率是fs，即相邻样本之间的间隔为1/fs。对于这种情况，最理想的重构滤波器是sinc(fsx)，它消除所有高于fs/2的频率。这在重新采样信号时非常有用（下一节）。然而，sinc的滤波器宽度是无限的，在某些区域是负的，因此在实际应用中很少有用。

There is a useful middle ground between the low-quality box and tent filters on one

hand, and the impractical sinc filter on the other. Most widely used filter functions

[1214, 1289, 1413, 1793] are between these extremes. All these filter functions have

some approximation to the sinc function, but with a limit on how many pixels they

influence. The filters that most closely approximate the sinc function have negative

values over part of their domain. For applications where negative filter values are

undesirable or impractical, filters with no negative lobes (often referred to generically

as Gaussian filters, since they either derive from or resemble a Gaussian curve) are

typically used [1402]. Section 12.1 discusses filter functions and their use in more

detail.

在低质量的盒型和帐篷过滤器和不实用的正弦滤波器之间，有一个有用的中间地带。最广泛使用的滤波函数[1214,1289,1413,1793]介于这两个极端之间。所有这些滤波函数都与正弦函数有一定的近似，但是他们对像素的影响是有限的。最接近正弦函数的滤波器在其定义域上有负值。对于不需要或不适用负滤波器值得应用，通常是没有负叶瓣的滤波器（通常称为高斯滤波器，因为他们要么派生自高斯曲线，要么类似于高斯曲线）[1402]。第12.1节更详细地讨论了过滤器函数及其使用。

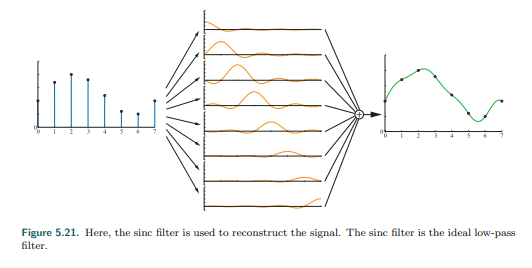


图5.21 这里，正弦滤波器用来重建信号。正弦滤波是理想的低通道滤波。

After using any filter, a continuous signal is obtained. However, in computer graphics

we cannot display continuous signals directly, but we can use them to resample the

continuous signal to another size, i.e., either enlarging the signal, or diminishing it.

This topic is discussed next.

使用任意滤波器后，得到连续信号。然而，在计算机图形学中，我们不能直接显示连续信号，但是我们可以使用他们来重新采样连续信号到另一个大小，即，要么放大信号，要么减弱信号。下面讨论这个主题。

Resampling 重采样

Resampling is used to magnify or minify a sampled signal. Assume that the original

sample points are located at integer coordinates (0, 1, 2, . . .), that is, with unit

intervals between samples. Furthermore, assume that after resampling we want the

new sample points to be located uniformly with an interval a between samples. For

a > 1, minification (downsampling) takes place, and for a < 1, magnification

(upsampling) occurs.

重采样用来放大或缩小一个采样信号。假定原始采样点位于整数坐标（0,1,2,…），即，使用单位间隔采样。更进一步，假定重新采样之后，我们希望新采样点的位置一致，采样点之间的间隔为a。对于a大于1，进行缩小（向下采样），对于小于1，进行放大（向上采样）。

Magnification is the simpler case of the two, so let us start with that. Assume the

sampled signal is reconstructed as shown in the previous section. Intuitively, since

the signal now is perfectly reconstructed and continuous, all that is needed is to

resample the reconstructed signal at the desired intervals. This process can be seen

in Figure 5.22.

放大是这两种情况中比较简单的，所以让我们从这里开始。假定采样信号如前一节所示被重构。直观的说，由于现在的信号是完全重构的，并且是连续的，所以所需要做的就是按期望的间隔重新采样重构信号。这个过程可以在图5.22中看到。

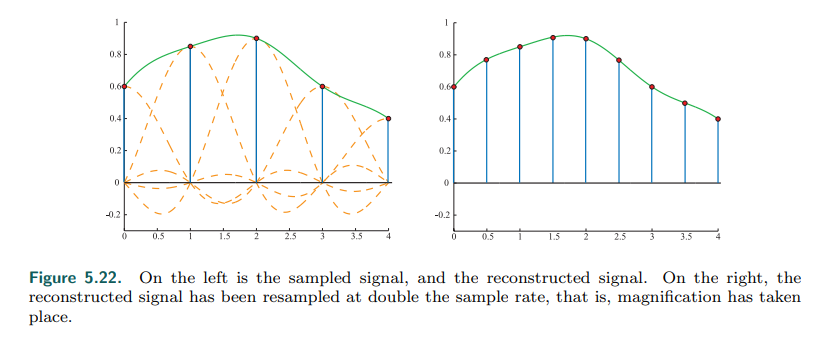


图 5.22 在左侧是采样后的信号和被重构后的信号。在右侧，重构后的信号以两倍的采样率进行重采样，即，放大采样。

However, this technique does not work when minification occurs. The frequency of

the original signal is too high for the sampling rate to avoid aliasing. Instead it has

been shown that a filter using sinc(x/a) should be used to create a continuous signal

from the sampled one [1447, 1661]. After that, resampling at the desired intervals

can take place. This can be seen in Figure 5.23. Said another way, by using sinc(x/a)

as a filter here, the width of the low-pass filter is increased, so that more of the

signal’s higher frequency content is removed. As shown in the figure, the filter width

(of the individual sinc’s) is doubled to decrease the resampling rate to half the

original sampling rate. Relating this to a digital image, this is similar to first blurring it

(to remove high frequencies) and then resampling the image at a lower resolution.

然而，这种技术在缩小采样时并不管用。原始信号的频率太高，采样率难以避免混叠。相反，使用sinc(x/a)滤波器可以从采样信号中产生连续信号[1447,1661]。在此之后，可以按所需的间隔重新取样。这可以在图5.23中看到。换句话说，在这里使用sinc(x/a)滤波器，增加了低通道滤波器的宽度，从而去除了更多的信号高频率内容。如图所示，将（单个sinc）滤波器宽度增加一倍，将重采样率降低到原始采样率的一半。将其与数字图像相关联，这类似于先将其模糊（去除高频），然后以低分辨率重新采样图像。

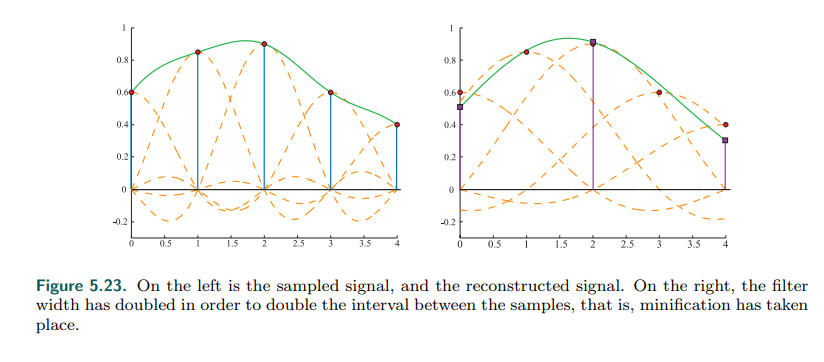


图5.23，在左侧是采样信号和重构信号。在右侧，滤波器宽度增加了一倍，使样品之间的间隔增加了一倍，即缩小采样。

With the theory of sampling and filtering available as a framework, the various

algorithms used in real-time rendering to reduce aliasing are now discussed.

随着采样理论和可用的滤波器作为框架，各种各样的算法用在实时渲染上减少混叠，我们接下来讨论。

5.4.2 Screen-Based Antialiasing 基于屏幕的抗锯齿

Edges of triangles produce noticeable artifacts if not sampled and filtered well.

Shadow boundaries, specular highlights, and other phenomena where the color is

changing rapidly can cause similar problems. The algorithms discussed in this section

help improve the rendering quality for these cases. They have the common thread

that they are screen based, i.e., that they operate only on the output samples of the

pipeline. There is no one best antialiasing technique, as each has different

advantages in terms of quality, ability to capture sharp details or other phenomena,

appearance during movement, memory cost, GPU requirements, and speed.

如果没有很好的采样和过滤，三角形的边缘会产生明显的伪影。阴影边界，高光亮度，和其他颜色迅速改变的现象可能导致相同的问题。在这个章节讨论的算法将帮助改善这种情况的渲染质量。他们有共同的思路就是他们是基于屏幕的，即，他们只对管道的输入样本进行操作。没有一种最佳的抗锯齿技术，因为每种技术在质量、捕捉尖锐细节或其他现象的能力，运动过程的外观，内存消耗，GPU需要和速度上都有不同的优势。

In the black triangle example in Figure 5.14, one problem is the low sampling rate. A

single sample is taken at the center of each pixel’s grid cell, so the most that is

known about the cell is whether or not the center is covered by the triangle. By using

more samples per screen grid cell and blending these in some fashion, a better pixel

color can be computed. This is illustrated in Figure 5.24.

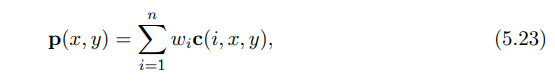
在图5.14中黑色三角形例子中，低的采样率是一个问题。在每个像素的网格中心取一个样本，因此对该单元所知最多的是该中心是否被三角形覆盖。通过对每个屏幕网格使用更多的样本并以某种方式混合这些样本，可以计算出更好的像素颜色。如图5.24所示。

The general strategy of screen-based antialiasing schemes is to use a sampling

pattern for the screen and then weight and sum the samples to produce a pixel color,

p:

基于屏幕的抗锯齿的一般策略是对屏幕使用采样模式，然后对采样进行加权和求和，产生像素颜色，p:



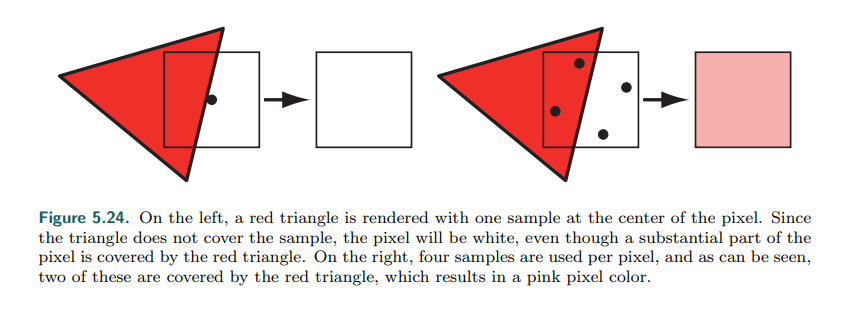


图5.24 在左侧，一个红色三角形被一个采样点（像素中心点）渲染。由于三角形并没有覆盖样本，像素因此是白色的，尽管一大部分的像素被红色三角形覆盖。在右侧，每个像素使用了4个采样点，可以看到，两个采样点被三角形覆盖，因此结果就是粉色像素颜色。

where n is the number of samples taken for a pixel. The function c(i, x, y) is a

sample color and wi is a weight, in the range [0, 1], that the sample will contribute

to the overall pixel color. The sample position is taken based on which sample it is in

the series 1, . . . , n, and the function optionally also uses the integer part of the

pixel location (x, y). In other words, where the sample is taken on the screen grid is

different for each sample, and optionally the sampling pattern can vary from pixel to

pixel. Samples are normally point samples in real-time rendering systems (and most

other rendering systems, for that matter). So, the function c can be thought of as

two functions. First, a function f(i, n) retrieves the floating point (xf , yf ) location on

the screen where a sample is needed. This location on the screen is then sampled,

i.e., the color at that precise point is retrieved. The sampling scheme is chosen and

the rendering pipeline configured to compute the samples at particular subpixel

locations, typically based on a per-frame (or per-application) setting.

n是一个像素采用的采样点数量。函数c(i, x, y)是一个采样颜色以及wi是权重，在范围[0,1]之间，即采样在所有像素颜色上的贡献。样本位置是根据他在连续1,…,n的哪个样本来确定的，函数还可以选择使用像素位置（x,y）的整数部分。换句话说，对于每个样品，在屏幕网格上采样的位置是不同的，并且可选择的采样模式随像素的不同而变化。在实时渲染系统中样品通常是点样品（以及大部分其他的渲染系统）。因此，函数c可以看成两个函数。首先，函数f(i, n)检索屏幕上需要样本的浮点（xf , yf）位置。然后对屏幕上的这个位置进行采样，即检索检索该精确点处的颜色。选择采样方案，并配置渲染管道来计算特定子像素位置的采样，通常基于每帧（或每个应用）设置。

The other variable in antialiasing is wi , the weight of each sample. These weights

sum to one. Most methods used in real-time rendering systems give a uniform

weight to their samples, i.e., wi = 1 n . The default mode for graphics hardware, a

single sample at the center of the pixel, is the simplest case of the antialiasing

equation above. There is only one term, the weight of this term is one, and the

sampling function f always returns the center of the pixel being sampled.

在抗锯齿中其他变量是wi，每个样品的权重。这些权重的总和为1。在实时渲染系统中，大部分方法给定了统一的采样权重，即，wi=1/n。图形硬件的默认模式是，像素中心点一个采样点，是上面抗锯齿方程的最简单情况。只有一项，这一项的权重是1，采样函数f总是返回被采样像素的中心。

Antialiasing algorithms that compute more than one full sample per pixel are called

supersampling (or oversampling) methods. Conceptually simplest, full-scene

antialiasing (FSAA), also known as “supersampling antialiasing” (SSAA), renders the

scene at a higher resolution and then filters neighboring samples to create an image.

For example, say an image of 1280 × 1024 pixels is desired. If you render an image

of 2560×2048 offscreen and then average each 2×2 pixel area on the screen, the

desired image is generated with four samples per pixel, filtered using a box filter.

Note that this corresponds to 2 × 2 grid sampling in Figure 5.25. This method is

costly, as all subsamples must be fully shaded and filled, with a z-buffer depth per

sample. FSAA’s main advantage is simplicity. Other, lower-quality versions of this

method sample at twice the rate on only one screen axis, and so are called 1 × 2 or

2 × 1 supersampling. Typically, powers-of-two resolution and a box filter are used

for simplicity. NVIDIA’s dynamic super resolution feature is a more elaborate form of

supersampling, where the scene is rendered at some higher resolution and a 13-

sample Gaussian filter is used to generate the displayed image [1848].

计算每个像素多个完整样本的抗锯齿算法称为超采样（或者过采样）方法。概念上最简单，全场景抗锯齿（FSAA），同样被称为“超级采样抗锯齿”(SSAA)，以更高的分辨率渲染场景，然后过滤相邻样本去创建图像。例如，预期一张1280x1024像素的图像。如果在屏幕外渲染一张2560x2048的图像，然后屏幕上每2x2像素区域平均，那么每个像素4个采样点使用盒型滤波器就生成了期望的图片。注意在图5.25中对应于2x2网格采样。这种方法是昂贵的，因为所有的子样本都必须完全着色和填充，每个样本都有z缓冲深度。FSAA的主要优点是简单。这种方法的其他低质量版本仅仅在一个屏幕轴上使用2倍采样率，因此成为1x2或2x1超采样。通常，为了简单起见，使用2次幂的分辨率和一个盒型滤波器。NVIDIA的动态超分辨率特性是一种更精细的超采样形式，其中场景以更高的分辨率渲染，使用13个样本高斯滤波器生成显示的图像[1848]。

A sampling method related to supersampling is based on the idea of the

accumulation buffer [637, 1115]. Instead of one large offscreen buffer, this method

uses a buffer that has the same resolution as the desired image, but with more bits

of color per channel. To obtain a 2 × 2 sampling of a scene, four images are

generated, with the view moved half a pixel in the screen x- or y-direction as needed.

Each image generated is based on a different sample position within the grid cell.

The additional costs of having to re-render the scene a few times per frame and

copy the result to the screen makes this algorithm costly for real-time rendering

systems. It is useful for generating higher-quality images when performance is not

critical, since any number of samples, placed anywhere, can be used per pixel [1679].

The accumulation buffer used to be a separate piece of hardware. It was supported

directly in the OpenGL API, but was deprecated in version 3.0. On modern GPUs the

accumulation buffer concept can be implemented in a pixel shader by using a higher-

precision color format for the output buffer.

一种与超采样相关的采样方法是基于累加缓冲区的思想[637,1115]。这种方法不是使用一个大的屏幕外的缓冲区，而是使用一个与所需图像具有相同分辨率的缓冲区，但是每个通道有更多的颜色位。为了获得一个场景的2x2采样，生成4张图片，视图根据需要在屏幕的x或y方向移动半个像素。基于网格单元内不同的采样位置生成每张图片。对于实时渲染系统来说，每帧重新渲染场景几次并将复制到屏幕上的额外成本使这种算法的成本很高。当性能不是很关键时，它对于生成高质量的图像很有用，因为任意数量的样本，放置在任何位置，每个像素都可以使用[1679]。累加缓冲区过去是一个单独的硬件。OpenGL API直接支持了它，但在3.0版本中不支持它。在现代GPU中，累加缓冲区的概念可以在像素着色器中实现，方法是为了输出缓冲使用更高精度的颜色格式。

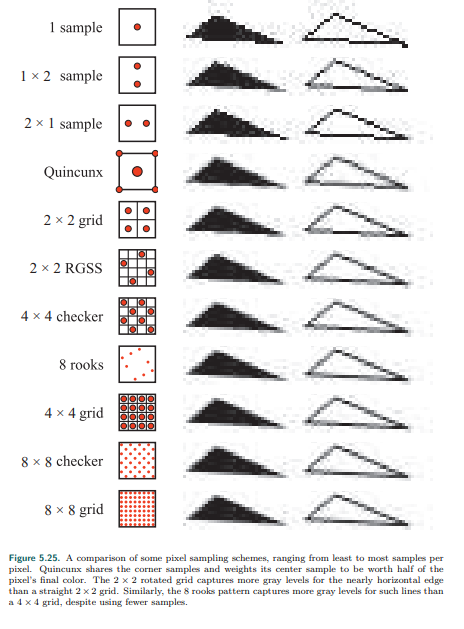


图5.25 一些像素采样方案的比较，范围从最小到最多像素。梅花形共享角落的样本，并将其中心样本的权重设置为像素的最终颜色。旋转后的2x2网格为接近水平的边缘捕获了更多的灰度而不是一个2x2的网格。类似的，8rooks模式捕捉到灰度级别相比于一个4x4的网格，使用了更少的样本。

Additional samples are needed when phenomena such as object edges, specular

highlights, and sharp shadows cause abrupt color changes. Shadows can often be

made softer and highlights smoother to avoid aliasing. Particular object types can be

increased in size, such as electrical wires, so that they are guaranteed to cover at

least one pixel at each location along their length [1384]. Aliasing of object edges

still remains as a major sampling problem. It is possible to use analytical methods,

where object edges are detected during rendering and their influence is factored in,

but these are often more expensive and less robust than simply taking more samples.

However, GPU features such as conservative rasterization and rasterizer order views

have opened up new possibilities [327].

当物体边缘，高光亮度，和尖锐阴影导致的颜色突然变化现象发生时，需要额外的样品。阴影常常柔软话以及亮度平滑化可以避免混叠。特定的对象类型可以在尺寸上增加，比如电线，这样他们就可以保证在其长度上每个位置覆盖至少一个像素[1384]。物体边缘的混叠仍然是主要的采样问题。可以使用分析方法，在渲染中检测对象边缘并考虑他们的影响，但是这些方法通常比简单的获取更多的样本更昂贵，也不那么健壮。然而，GPU的一些特性，如保守的格栅化和格栅化顺序视图，已经开辟可新的可能性[327]。

Techniques such as supersampling and accumulation buffering work by generating

samples that are fully specified with individually computed shades and depths. The

overall gains are relatively low and the cost is high, as each sample has to run

through a pixel shader.

超采样和累积缓冲等技术通过生成具有单独计算的着色器和深度的完全指定的样本来工作。总体增益相对较低，成本较高，因为每个样本都必须通过像素着色器。

Multisampling antialiasing (MSAA) lessens the high computational costs by computing

the surface’s shade once per pixel and sharing this result among the samples. Pixels

may have, say, four (x, y) sample locations per fragment, each with their own color

and z-depth, but the pixel shader is evaluated only once for each object fragment

applied to the pixel. If all MSAA positional samples are covered by the fragment, the

shading sample is evaluated at the center of the pixel. If instead the fragment covers

fewer positional samples, the shading sample’s position can be shifted to better

represent the positions covered. Doing so avoids shade sampling off the edge of a

texture, for example. This position adjustment is called centroid sampling or centroid

interpolation and is done automatically by the GPU, if enabled. Centroid sampling

avoids off-triangle problems but can cause derivative computations to return

incorrect values [530, 1041]. See Figure 5.26.

多重采样抗锯齿（MSAA）通过每像素计算一次表面着色，并在样本之间共享结果，降低了较高的计算成本。像素可能有，每个片段4个（x,y）样本位置，每个片元都有自己的颜色和深度缓冲，但是对于应用于像素的每个对象片段，像素着色器只评估一次。如果片元覆盖了所有MSAA位置样本，则在像素的中心评估着色样本。如果片元覆盖了更少的位置样本，则着色样本的位置可以移动，以更好的表示覆盖的位置。例如，这样做可以避免对纹理边缘进行着色采样。这种位置调整称为质心采样或质心插值，并由GPU自动完成，如果启用。质心采样避免了三角形外的问题，但会导致导数计算返回不正确的值[530,1041]。参见图5.26。

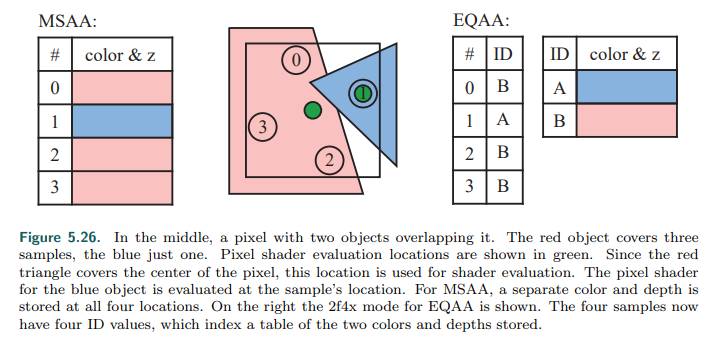


图5.26 在中间，有两个物体重叠的像素。红色的物体覆盖了三个物体样本，蓝色的只有一个。像素着色器评估位置显示为绿色。因为红色三角形覆盖了像素的中心，这个位置用于着色器评估。蓝色对象的像素着色器是在样本位置评估的。对于MSAA，单独的颜色和深度是存储在所有的四个位置。右边是EQAA的2f4x模式。四个样本由4个ID值，他们索引一个包含存储的两种颜色和深度的表。

MSAA is faster than a pure supersampling scheme because the fragment is shaded

only once. It focuses effort on sampling the fragment’s pixel coverage at a higher

rate and sharing the computed shade. It is possible to save more memory by further

decoupling sampling and coverage, which in turn can make antialiasing faster still—

the less memory touched, the quicker the render. NVIDIA introduced coverage

sampling antialiasing (CSAA) in 2006, and AMD followed suit with enhanced quality

antialiasing (EQAA). These techniques work by storing only the coverage for the

fragment at a higher sampling rate. For example, EQAA’s “2f4x” mode stores two

color and depth values, shared among four sample locations. The colors and depths

are no longer stored for particular locations but rather saved in a table. Each of the

four samples then needs just one bit to specify which of the two stored values is

associated with its location. See Figure 5.26. The coverage samples specify the

contribution of each fragment to the final pixel color. If the number of colors stored

is exceeded, a stored color is evicted and its samples are marked as unknown. These

samples do not contribute to the final color [382, 383]. For most scenes there are

relatively few pixels containing three or more visible opaque fragments that are

radically different in shade, so this scheme performs well in practice [1405]. However,

for highest quality, the game Forza Horizon 2 went with 4× MSAA, though EQAA had

a performance benefit [1002].

MSAA比纯超级采样方案更快因为片元只用着色一次。它着重于以更高的速率采样片元的像素覆盖率，并共享计算得到的着色。通过进一步解耦采样和覆盖可以节省更多的内存，这反过来又可以使抗锯齿速度更快——接触的内存越少，渲染的速度就越快。NVIDIA在2006年引入了覆盖采样抗锯齿（CSAA），AMD也紧随其后推出了增强质量抗锯齿（EQAA）。这些技术的工作原理是只以较高的采样率存储片元的覆盖率。例如，EQAA的“2f4x”模式存储两个颜色和深度值，在四个样本位置之间共享。颜色和深度不在存储在特定的位置上，而是保存在一个表中。四个样本仅仅只需要一个位就可以指定这两个存储值中哪个与它的位置相关联。参见图5.26。覆盖样本指定每个片元对最终像素颜色的贡献。如果超过存储的颜色数量，则删除存储的颜色，并将其样本标记为未知。这些样本不构成最终的颜色[382,383]。对于大部分场景来说，包含三个或更多可见的并且在着色上完全不同的不透明片段的像素相对较少，因此这种方案在实践中表现的很好[1405]。然而，在高质量游戏中，Forza Horizon 2使用了4xMSAA，尽管EQAA具有性能优势[1002]。

Once all geometry has been rendered to a multiple-sample buffer, a resolve

operation is then performed. This procedure averages the sample colors together to

determine the color for the pixel. It is worth noting that a problem can arise when

using multisampling with high dynamic range color values. In such cases, to avoid

artifacts you normally need to tone-map the values before the resolve [1375]. This

can be expensive, so a simpler approximation to the tone map function or other

methods can be used [862, 1405].

一旦所有的几何体渲染到了多重采样缓冲区，然后执行一次解析操作。这个过程平均采样颜色去决定像素的颜色。值得注意的是，当具有高动态范围颜色值的多重采样时，可能出现问题。在这种情况下，为了避免伪影，通常需要在解析之前进行色调映射[1375]。这可能是昂贵的，所以可以使用更简单的近似色调映射函数或其他方法[862,1405]。

By default, MSAA is resolved with a box filter. In 2007 ATI introduced custom filter

antialiasing (CFAA) [1625], with the capabilities of using narrow and wide tent filters

that extend slightly into other pixel cells. This mode has since been supplanted by

EQAA support. On modern GPUs pixel or compute shaders can access the MSAA

samples and use whatever reconstruction filter is desired, including one that samples

from the surrounding pixels’ samples. A wider filter can reduce aliasing, though at

the loss of sharp details. Pettineo [1402, 1405] found that the cubic smoothstep and

B-spline filters with a filter width of 2 or 3 pixels gave the best results overall. There

is also a performance cost, as even emulating the default box filter resolve will take

longer with a custom shader, and a wider filter kernel means increased sample

access costs.

默认的，MSAA被一个盒型滤波器解析。在2007年ATI引入了可定义滤波抗锯齿（CFAA）[1625]，具有使用窄和宽帐篷滤波器的能力，可以稍微扩展到其他像素单元。这种模式已经被EQAA支持所取代。在现代GPU上，像素或计算着色器可以访问MSAA采样，并使用所需的任何重构过滤器，包括从周围像素样本中提取样本的过滤器。一个更宽的滤波器可以减少混叠，尽管失去了清晰的细节。Pettineo [1402, 1405]发现立方体平滑和B样条的滤波器在滤波宽度为2或者3个像素时效果最好。还有一个性能成本，因为即使使用自定义着色器来模拟默认的盒型滤波器解析也需要更长的时间，而更广泛的滤波器核心意味着增加了样本访问成本。

NVIDIA’s built-in TXAA support similarly uses a better reconstruction filter over a

wider area than a single pixel to give a better result. It and the newer MFAA

(multiframe antialiasing) scheme both also use temporal antialiasing (TAA), a general

class of techniques that use results from previous frames to improve the image. In

part such techniques are made possible due to functionality that lets the programmer

set the MSAA sampling pattern per frame [1406]. Such techniques can attack

aliasing problems such as the spinning wagon wheel and can also improve edge

rendering quality.

NVIDIA内置的TXAA支持类似地比单个像素更宽的区域上使用更好的重建滤波器。他和最新的MFAA（多帧抗锯齿）方案都使用了时间抗锯齿（TAA），这是一种使用前几帧的结果来改进图像的一般技术。在某种程度上，这种技术之所以成为可能，是因为其功能允许程序员设置每帧MSAA采样模式[1406]。这种技术可以解决诸如旋转车轮之类的混叠问题，还可以提高边缘渲染质量。

Imagine performing a sampling pattern “manually” by generating a series of images

where each render uses a different location within the pixel for where the sample is

taken. This offsetting is done by appending a tiny translation on to the projection

matrix [1938]. The more images that are generated and averaged together, the

better the result. This concept of using multiple offset images is used in temporal

antialiasing algorithms. A single image is generated, possibly with MSAA or another

method, and the previous images are blended in. Usually just two to four frames are

used [382, 836, 1405]. Older images may be given exponentially less weight [862],

though this can have the effect of the frame shimmering if the viewer and scene do

not move, so often equal weighting of just the last and current frame is done. With

each frame’s samples in a different subpixel location, the weighted sum of these

samples gives a better coverage estimate of the edge than a single frame does. So,

a system using the latest two frames averaged together can give a better result. No

additional samples are needed for each frame, which is what makes this type of

approach so appealing. It is even possible to use temporal sampling to allow

generation of a lower-resolution image that is upscaled to the display’s resolution

[1110]. In addition, illumination methods or other techniques that require many

samples for a good result can instead use fewer samples each frame, since the

results will be blended over several frames [1938].

想象一下，通过生成一系列的图像手工执行采样模式，其中每个渲染都使用采样点所在像素内的不同位置。这种偏移通过在投影矩阵上附加一个微小的平移来实现[1938]。越多的图像生成然后一起平均，结果就越好。使用多重偏移图像的概念是使用时间抗锯齿算法。生成一个单独的图像，可能使用MSAA或其他方法，并混合以前的图像。通常只使用2到4帧[382,836,1405]。旧图像的权重可能会以指数形式减少[862]，尽管如果观众和场景不移动，这可能会产生帧闪烁的效果，因此通常只对最后一帧和当前帧进行相同的权重。对于每一帧的在不同子像素位置的样本，这些样本的权重之和合比单一帧具有更好的边缘覆盖率评估。因此，使用最新的两帧平均一起的系统可以得到更好的结果。每一帧不需要额外的样本，这正是这种方法如此吸引人的原因。甚至可以使用时间采样来生成分辨率较低的图像，并将其放大到显示器的分辨率[1110]。此外，每帧，光照方法或其他需要很多样本才能得到好的结果的技术可以使用更少的样本替代，因为结果将被混合在多个帧上[1938]。

While providing antialiasing for static scenes at no additional sampling cost, this type

of algorithm has a few problems when used for temporal antialiasing. If the frames

are not weighted equally, objects in a static scene can exhibit a shimmer. Rapidly

moving objects or quick camera moves can cause ghosting, i.e., trails left behind the

object due to the contributions of previous frames. One solution to ghosting is to

perform such antialiasing on only slow-moving objects [1110]. Another important

approach is to use reprojection (Section 12.2) to better correlate the previous and

current frames’ objects. In such schemes, objects generate motion vectors that are

stored in a separate “velocity buffer” (Section 12.5). These vectors are used to

correlate the previous frame with the current one, i.e., the vector is subtracted from

the current pixel location to find the previous frame’s color pixel for that object’s

surface location. Samples unlikely to be part of the surface in the current frame are

discarded [1912]. Because no extra samples, and so relatively little extra work, are

needed for temporal antialiasing, there has been a strong interest and wider

adoption of this type of algorithm in recent years. Some of this attention has been

because deferred shading techniques (Section 20.1) are not compatible with MSAA

and other multisampling support [1486]. Approaches vary and, depending on the

application’s content and goals, a range of techniques for avoiding artifacts and

improving quality have been developed [836, 1154, 1405, 1533, 1938]. Wihlidal’s

presentation [1885], for example, shows how EQAA, temporal antialiasing, and

various filtering techniques applied to a checkerboard sampling pattern can combine

to maintain quality while lowering the number of pixel shader invocations. Iglesias-

Guitian et al. [796] summarize previous work and present their scheme to use pixel

history and prediction to minimize filtering artifacts. Patney et al. [1357] extend TAA

work by Karis and Lottes on the Unreal Engine 4 implementation [862] for use in

virtual reality applications, adding variable-sized sampling along with compensation

for eye movement (Section 21.3.2).

在不增加采样成本的情况下为静态场景提供抗锯齿的同时，这种算法在用时间抗锯齿时存在一些问题。如果帧的权重不相等，静态场景中的物体就会闪烁。快速移动的物体或摄像机的快速移动会导致鬼影，即，由于前几帧的贡献留下了痕迹。重影的一种解决方案是仅对缓慢移动的对象执行这种抗锯齿[1110]。另一个重要的方法是使用二次投影（第12.2节）来更好的关联以前帧和当前帧的对象。在这类方案中，对象生成运动向量储存在独立的“速度缓冲区”（章节12.5）。这些向量用来重联系先前的帧与当前帧，即，从当前像素位置减去向量，得到该物体表面位置的前一帧颜色像素。在当前帧，不像表面的样本将被丢弃[1912]。由于时间抗锯齿不需要额外的样本，也不需要额外的工作，因此近年来这种算法得到了广泛的应用。这种关注的部分原因是延迟渲染技术（章节20.1）与MSAA和其他的多重采样支持不兼容[1486]。方法各不相同，并且根据应用程序的内容和目标，已经开发了一系列避免伪影和提高质量的技术[836,1154,1405,1533,1938]。例如，Wihlidal的展示，显示了EQAA，时间抗锯齿，和应用于棋盘采样模式的各种滤波技术如何结合使用来保持质量，同时降低像素着色器调用数量。Iglesias-Guitian等[796]总结了以前的工作，提出了使用像素历史和预测来最小化滤波伪影的方案。Patney等[1357]对Karis 和 Lottes在虚幻引擎4实现上的TAA工作进行了扩展[862]，将其用于虚拟现实应用程序，增加了可变大小的采样以及眼动补偿（章节21.3.2）。

Sampling Patterns 采样模式

Effective sampling patterns are a key element in reducing aliasing, temporal and

otherwise. Naiman [1257] shows that humans are most disturbed by aliasing on near

-horizontal and near-vertical edges. Edges with near 45 degrees slope are next most

disturbing. Rotated grid supersampling (RGSS) uses a rotated square pattern to give

more vertical and horizontal resolution within the pixel. Figure 5.25 shows an

example of this pattern.

有效的采样模式是降低混叠、时间和其他的一个关键因素。Naiman[1257]指出，人类最容易受到接近水平和接近垂直边缘混叠的干扰。边缘将近45度的倾斜是下一个困扰旋转网格超采样（RGSS）使用旋转的正方形模式来在像素内提供更多的垂直和水平分辨率。图5.25显示了这种模式的一个示例。

The RGSS pattern is a form of Latin hypercube or N-rooks sampling, in which n

samples are placed in an n×n grid, with one sample per row and column [1626].

With RGSS, the four samples are each in a separate row and column of the 4 × 4

subpixel grid. Such patterns are particularly good for capturing nearly horizontal and

vertical edges compared to a regular 2 × 2 sampling pattern, where such edges are

likely to cover an even number of samples, so giving fewer effective levels.

RGSS模式是来自Latin hypercube或N棍采样，一个nxn网格放置n个样本，每行和列只有一个样本[1626]。在RGSS下，4x4的子像素网格有4个样本，每个样本占独立的行列。与常规的2x2采样模式相比，这种模式尤其适用于捕捉接近水平和垂直的边缘，在常规的2x2采样模式中，这种边缘可能覆盖偶数个样本，因此提供的有效水平更低。

N-rooks is a start at creating a good sampling pattern, but it is not sufficient. For

example, the samples could all be places along the diagonal of a subpixel grid and so

give a poor result for edges that are nearly parallel to this diagonal. See Figure

5.27.For better sampling we want to avoid putting two samples near each other. We

also want a uniform distribution, spreading samples evenly over the area. To form

such patterns, stratified sampling techniques such as Latin hypercube sampling are

combined with other methods such as jittering, Halton sequences, and Poisson disk

sampling [1413, 1758].

N-rooks是创建良好抽样模式的起点，但它还不够。例如，样本可能位于子像素网格的对角线上，因此对于几乎平行于该对角线的边结果很差。见图5.27。为了更好的采样，我们将避免两个点相互挨着。我们也想要均匀分布，样本均匀的覆盖区域。为了形成这样模式，分层抽样技术（如拉丁超立方体抽样）与其他方法（如抖动，哈尔顿序列和泊松盘抽样）相结合[1413，1758]。

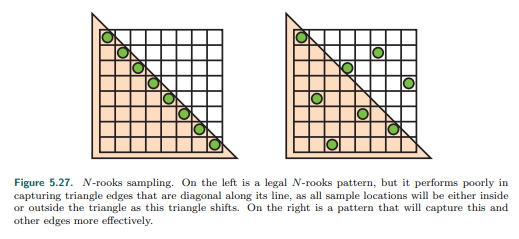


图5.27 N-rooks采样。在左侧是一个合法的N-rooks模式，但是他的执行很差，捕捉沿其直线对角线的三角形边缘，因为所有示例位置都在其中，或者三角形外，当这个三角形移动。右边是一个模式，可以捕捉到这个和其他优势更有效。

In practice GPU manufacturers usually hard-wire such sampling patterns into their

hardware for multisampling antialiasing. Figure 5.28 shows some MSAA patterns

used in practice. For temporal antialiasing, the coverage pattern is whatever the

programmer wants, as the sample locations can be varied frame to frame. For

example, Karis [862] finds that a basic Halton sequence works better than any MSAA

pattern provided by the GPU. A Halton sequence generates samples in space that

appear random but have low discrepancy, that is, they are well distributed over the

space and none are clustered [1413, 1938].

实际上，GPU制造商为了多重采样抗锯齿通常硬线连接这样的采样模式到硬件。图5.28显示了一些实际中使用的MSAA模式。对于时间抗锯齿，覆盖模式是程序员想要的任何东西，因为示例位置可以随着帧的不同而变化。例如，Karis[862]发现一个基本的Halton序列比GPU提供的任何MSAA模式工作的更好。Halton序列在空间中产生的样本看似随机，但差异很小，即他们在空间中分布良好，没有一个是聚类的[1413，1938]。

While a subpixel grid pattern results in a better approximation of how each triangle

covers a grid cell, it is not ideal. A scene can be made of objects that are arbitrarily

small on the screen, meaning that no sampling rate can ever perfectly capture them.

If these tiny objects or features form a pattern, sampling at constant intervals can

result in Moir´e fringes and other interference patterns. The grid pattern used in

supersampling is particularly likely to alias.

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络单元，但它并不理想。一个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捉到他们。如果这些小的物体或特征形成一种模式，在固定的时间间隔内采样会产生莫尔条纹或其他干涉模式。在超采样中使用的网格模式很可能是混叠的。

One solution is to use stochastic sampling, which gives a more randomized pattern.

Patterns such as those in Figure 5.28 certainly qualify. Imagine a fine-toothed comb

at a distance, with a few teeth covering each pixel. A regular pattern can give severe

artifacts as the sampling pattern goes in and out of phase with the tooth frequency.

Having a less ordered sampling pattern can break up these patterns. The

randomization tends to replace repetitive aliasing effects with noise, to which the

human visual system is much more forgiving [1413]. A pattern with less structure

helps, but it can still exhibit aliasing when repeated pixel to pixel. One solution is use

a different sampling pattern at each pixel, or to change each sampling location over

time. Interleaved sampling index sampling! interleaved, where each pixel of a set has

a different sampling pattern, has occasionally been supported in hardware over the

past decades. For example, ATI’s SMOOTHVISION allowed up to 16 samples per

pixel and up to 16 different user-defined sampling patterns that could be

intermingled in a repeating pattern (e.g., in a 4 × 4 pixel tile). Molnar [1234], as well

as Keller and Heidrich [880], found that using interleaved stochastic sampling

minimizes the aliasing artifacts formed when using the same pattern for every pixel.

一种解决方案是使用随机采样，是一种更随机化的模式。图5.28中所示的模式符合条件。想象一下，在远处有一个细齿梳子，每个像素都有一些细齿。常规模式下，当采样模式随着细齿频率而进进出出时，会发生严重的伪影。如果抽样模式的顺序较低，则会破快这些模式，随机化倾向于用噪声替代重复的混叠效果，而人类的视觉系统对噪声的容忍度要高得多[1413]。较少结果的模式帮助，但是当重复像素到像素时，它仍然可以显示混叠。一种解决方案是在每个像素上使用不同的采样模式，或者随着时间的推移改变每个采样位置。交错抽样指数抽样！交叉采样是指一组图像中的每个像素都有不同的采样模式，在过去几十年里，硬件偶尔也支持交叉采样。例如，ATI的平滑视觉允许每个像素最多16个样本，以及最多16个不同的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4x4像素瓦片中）。Molnar [1234]以及Keller 和 Heidrich [880]法线，当对每个像素使用相同的模式时，使用交叉随机抽样可以最小化产生的混叠伪影。

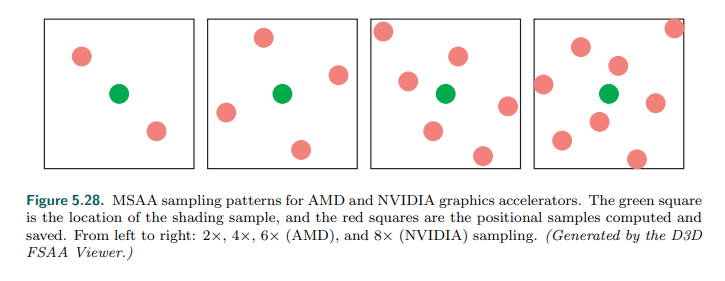


图5.28 MSAA采样模式AMD和NVIDIA图形加速器。绿色中心是着色样本的位置，红色中心是位置样本的计算和保存。从左到右2x，4x（AMD），8x（NVIDIA）采样。（由D3D生成FASS查看器）。

A few other GPU-supported algorithms are worth noting. One real-time antialiasing

scheme that lets samples affect more than one pixel is NVIDIA’s older Quincunx

method [365]. “Quincunx” means an arrangement of five objects, four in a square

and the fifth in the center, such as the pattern of five dots on a six-sided die.

Quincunx multisampling antialiasing uses this pattern, putting the four outer samples

at the corners of the pixel. See Figure 5.25. Each corner sample value is distributed

to its four neighboring pixels. Instead of weighting each sample equally (as most

other real-time schemes do), the center sample is given a weight of 1/2 , and each

corner sample has a weight of 1/8 . Because of this sharing, an average of only two

samples are needed per pixel, and the results are considerably better than two-

sample FSAA methods [1678]. This pattern approximates a two-dimensional tent

filter, which, as discussed in the previous section, is superior to the box filter.

其他的一些GPU支持的算法值得注意。一种让样本影响多个像素的实时抗锯齿方案是NVIDIA的较老的梅花形方法[365]。“梅花形”意味着五个物体的排泄，四个在一个正方形中，第5个在中间，比如六面骰子上五个点的模式。梅花形多重采样抗锯齿采用这种模式，将四个外部样本放在像素的拐角。参看图5.25.每个拐角样本值被分配到他的四个相邻像素。不同于其他实时方案对每个样本的权重相同（大多数其他实时方案都是这样），中心样本的权重为1/2，每个拐角样本权重为1/8。由于这种共享，平均每个像素只需要两个样本，其结果明显优于双采样FSAA[1678]。这种模式近似于二维帐篷滤波器，正如前面所讨论的，它优于盒型滤波器。

Quincunx sampling can also be applied to temporal antialiasing by using a single

sample per pixel [836, 1677]. Each frame is offset half a pixel in each axis from the

frame before, with the offset direction alternating between frames. The previous

frame provides the pixel corner samples, and bilinear interpolation is used to rapidly

compute the contribution per pixel. The result is averaged with the current frame.

Equal weighting of each frame means there are no shimmer artifacts for a static view.

The issue of aligning moving objects is still present, but the scheme itself is simple to

code and gives a much better look while using only one sample per pixel per frame.

梅花形采样也可以通过使用每个像素一个样本来应用于时间抗锯齿[836,1677]。每一帧在每个轴上距前一帧偏移半像素，偏移方向在帧与帧之间交替。前一帧提供像素拐角采样，采用线性插值快速计算每个像素的贡献。计算结果与当前帧平均。每个帧权重相等意味着静态视图没有闪烁的伪影。对齐移动对象的问题仍然存在，但是该方案本身易于编写代码，而且在每帧仅适用一个像素样本的情况下，效果会好得多。

When used in a single frame, Quincunx has a low cost of only two samples by

sharing samples at the pixel boundaries. The RGSS pattern is better at capturing

more gradations of nearly horizontal and vertical edges. First developed for mobile

graphics, the FLIPQUAD pattern combines both of these desirable features [22]. Its

advantages are that the cost is only two samples per pixel, and the quality is similar

to RGSS (which costs four samples per pixel). This sampling pattern is shown in

Figure 5.29. Other inexpensive sampling patterns that exploit sample sharing are

explored by Hasselgren et al. [677].

在每帧中使用梅花形方案时，通过像素边界共享样本，只有两个样本，成本很低。RGSS模式更擅长捕捉几乎水平和垂直边缘的更多渐变。FLIPQUAD模式最初是为了移动图形开发的，它结合了这两个理想的特性[22]。他的优点是每像素只需要两个样本，质量与RGSS类似（每像素4个样本）。这个抽样模式如图5.29。Hasselgren等人探索了其他利用样本共享的廉价抽样模式[677]。

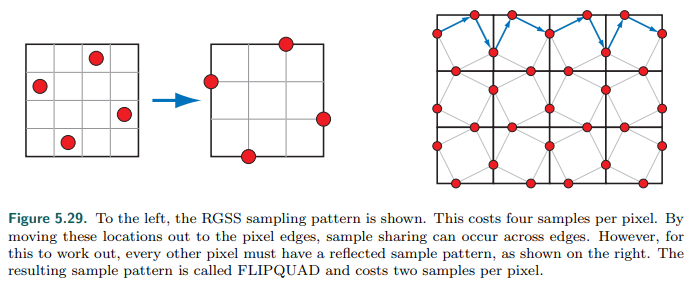


图 5.29 左侧，显示了RGSS采样模式。每像素耗费4个采样。通过移动这些位置到像素边缘，在边缘就出现了采样分享。然而，对于要做到这一点，每一个其他像素必须有一个反射的样本模式，如图所示。得到的样本模式成为FLIPQUAD，每个像素需要两个样本。

Like Quincunx, the two-sample FLIPQUAD pattern can also be used with temporal

antialiasing and spread over two frames. Drobot [382, 383, 1154] tackles the

question of which two-sample pattern is best in his hybrid reconstruction antialiasing

(HRAA) work. He explores different sampling patterns for temporal antialiasing,

finding the FLIPQUAD pattern to be the best of the five tested. A checkerboard

pattern has also seen use with temporal antialiasing. El Mansouri [415] discusses

using two sample MSAA to create a checkerboard render to reduce shader costs

while addressing aliasing issues. Jimenez [836] uses SMAA, temporal antialiasing,

and a variety of other techniques to provide a solution where antialiasing quality can

be changed in response to rendering engine load. Carpentier and Ishiyama [231]

sample on edges, rotating the sampling grid by 45◦ . They combine this temporal

antialiasing scheme with FXAA (discussed later) to efficiently render on higher-

resolution displays.

和梅花形方案一样，双样本的FLIPQUAD模式也可以用于时间抗锯齿以及扩展到两帧。Drobot在他的混合重建抗锯齿（HRAA）工作中解决了哪种双样本模式最好的问题。他探索了不同的采样模式的时间抗锯齿，发现FLIPQUAD模式是5个测试中最好的。棋盘格式也可以用于时间抗锯齿。El Mansouri[415]讨论了使用两个样本的MSAA来创建一个棋盘渲染以减少着色器的成本，同时解决了混叠问题。Jimenez[836]使用SMAA、时间抗锯齿和多种其他技术提供了一种解决方案，其中抗锯齿质量可以随着渲染引擎负载而改变。Carpentier 和 Ishiyama在边缘采样，旋转采样网格45度。他们将这种时间抗锯齿方案和FXAA（稍后讨论）相结合，以高效的渲染在高分辨率显示器上。

Morphological Methods 形态学的方法

Aliasing often results from edges, such as those formed by geometry, sharp shadows,

or bright highlights. The knowledge that aliasing has a structure associated with it

can be exploited to give a better antialiased result. In 2009 Reshetov [1483]

presented an algorithm along these lines, calling it morphological antialiasing (MLAA).

“Morphological” means “relating to structure or shape.” Earlier work had been done

in this area [830], as far back as 1983 by Bloomenthal [170]. Reshetov’s paper

reinvigorated research into alternatives to multisampling approaches, emphasizing

searching for and reconstructing edges [1486].

混叠经常是边缘的结果，比如由几何体，尖锐阴影，或者高亮。关于混叠相关结构的知识可以用来提供更好的抗锯齿结果。在2009年Reshetov[1483]提供了一种类似的算法，称为形态学抗锯齿（MLAA）。“形态学”的意思是“与结构或形状有关的”。早在1983年Bloomenthal[170]就在这一领域做过研究[830]。Reshetov的论文重振了对多采样方法替代方法的研究，强调寻找和重构边缘[1486]。

This form of antialiasing is performed as a post-process. That is, rendering is done in

the usual fashion, then the results are fed to a process that generates the antialiased

result. A wide range of techniques have been developed since 2009. Those that rely

on additional buffers such as depths and normals can provide better results, such as

subpixel reconstruction antialiasing (SRAA) [43, 829], but are then applicable for

antialiasing only geometric edges. Analytical approaches, such as geometry buffer

antialiasing (GBAA) and distance-to-edge antialiasing (DEAA), have the renderer

compute additional information about where triangle edges are located, e.g., how far

the edge is from the center of the pixel [829].

这种形式的抗锯齿是作为一个后处理来执行的。也就是说，渲染以通常的方式完成，然后将结果提供给生成抗锯齿结果的进程。自2009年以来，已经开发了广泛的技术。那些依赖于额外缓冲区（如深度和法线）的方法可以提供更好的结果，比如子像素重构抗锯齿（SRAA）[43,829]，但是只适用于几何边缘的抗锯齿。分析方法，比如几何缓冲区抗锯齿（GBAA）和距离到边缘抗锯齿（DEAA），使渲染器计算三角形边缘位于何处的附加信息，例如，边缘距离像素中心有多远[829]。

The most general schemes need only the color buffer, meaning they can also

improve edges from shadows, highlights, or various previously applied post-

processing techniques, such as silhouette edge rendering (Section 15.2.3). For

example, directionally localized antialiasing (DLAA) [52, 829] is based on the

observation that an edge which is nearly vertical should be blurred horizontally, and

likewise nearly horizontal edges should be blurred vertically with their neighbors.

大多数通用的方案只需要颜色缓冲，意味着他们还可以从阴影、高光、或者各种以前应用的后处理技术（如剪影边缘渲染）中改善边缘（章节15.2.3）。例如指向局部抗锯齿[52,829]是基于这样一种观察近乎垂直的边缘应该在水平方向上进行模糊处理，同样，近乎水平的边缘也应该与相邻的边缘在垂直方向上进行模糊处理。

More elaborate forms of edge detection attempt to find pixels likely to contain an

edge at any angle and determine its coverage. The neighborhoods around potential

edges are examined, with the goal of reconstructing as possible where the original

edge was located. The edge’s effect on the pixel can then be used to blend in

neighboring pixels’ colors. See Figure 5.30 for a conceptual view of the process.

更精细的边缘检测形式试图找到可能包含任何角度边缘的像素，并确定其覆盖范围。研究潜在边缘周围的领域，目的是尽可能重构原始边缘所在的位置。边缘对像素的影响可以用来混合相邻像素的颜色。有关流程的概念视图，请参见图5.30。

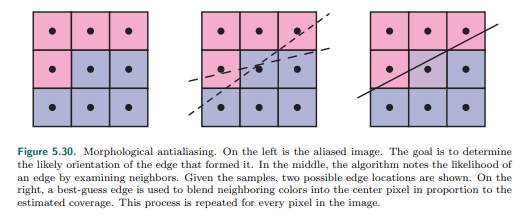


图5.30 形态学抗锯齿。在左侧是锯齿图片。目标是确定形成他的边缘的可能方向。在中间，算法通过检查邻居计算边缘的可能性。给定样本，显示了两个可能的边缘位置。在右侧，使用最佳猜测边缘将相邻颜色按比例混合到中心像素中估计覆盖。对图像中的每个像素重复这个过程。

Iourcha et al. [798] improve edge-finding by examine the MSAA samples in pixels to

compute a better result. Note that edge prediction and blending can give a higher

precision result than sample-based algorithms. For example, a technique that uses

four samples per pixel can give only five levels of blending for an object’s edge: no

samples covered, one covered, two, three, and four. The estimated edge location

can have more locations and so provide better results.

Iourcha等[798]通过以像素为单位检测MSAA样本来改进边缘检测，从而计算出更好的结果。注意，与基于样本的算法相比，边缘预测和混合可以得到更高的精度。例如，每个像素使用4个样本的技术只能为对象的边缘提供5个层次的混合：没有覆盖的样本，一个、两个、三个和四个覆盖的样本。估计的边缘位置可以有更多的位置，从而提供更好的结果。

There are several ways image-based algorithms can go astray. First, the edge may

not be detected if the color difference between two objects is lower than the

algorithm’s threshold. Pixels where there are three or more distinct surfaces

overlapping are difficult to interpret. Surfaces with high-contrast or high-frequency

elements, where the color is changing rapidly from pixel to pixel, can cause

algorithms to miss edges. In particular, text quality usually suffers when

morphological antialiasing is applied to it. Object corners can be a challenge, with

some algorithms giving them a rounded appearance. Curved lines can also be

adversely affected by the assumption that edges are straight. A single pixel change

can cause a large shift in how the edge is reconstructed, which can create noticeable

artifacts frame to frame. One approach to ameliorate this problem is to use MSAA

coverage masks to improve edge determination [1484].

有几种基于图像的算法可能会误入歧途。首先，如果两个物体之间的色差小于算法的阈值，则不能检测到边缘。有三个或多个不同表面重叠的像素很难解释。具有高对比度或者高频元素的表面，颜色在像素之间快速变化，可能导致算法遗漏边缘。特别的是，当形态学抗锯齿应用时，文本质量通常会收到影响。对象圆角可能是一个挑战，一些算法给他们一个圆形的外观。假定边是直的，曲线也会收到不利的影响。单个像素的改变会导致重构边缘的方式发生很大的变化，从而在帧与帧之间创建明显的伪影。改善这一问题的一种方法是使用MSAA覆盖遮罩来改进边缘确定[1484]。

Morphological antialiasing schemes use only the information that is provided. For

example, an object thinner than a pixel in width, such as an electrical wire or rope,

will have gaps on the screen wherever it does not happen to cover the center

location of a pixel. Taking more samples can improve the quality in such situations;

image-based antialiasing alone cannot. In addition, execution time can be variable

depending on what content is viewed. For example, a view of a field of grass can

take three times as long to antialias as a view of the sky [231].

形态学抗锯齿方案仅使用所提供的信息。例如，宽的小于一个像素的物体，如电线或绳子，只要没有恰好覆盖像素的中心位置，屏幕上就会有间隙。在这种情况下，多采样可以提高质量：仅仅基于图像的抗锯齿是不行的。此外，执行时间可以根据查看的内容而变化。例如，一块草地的视图的抗锯齿可能是天空视图的三倍[231]。

All this said, image-based methods can provide antialiasing support for modest

memory and processing costs, so they are used in many applications. The color-only

versions are also decoupled from the rendering pipeline, making them easy to modify

or disable, and can even be exposed as GPU driver options. The two most popular

algorithms are fast approximate antialiasing (FXAA) [1079, 1080, 1084], and subpixel

morphological antialiasing (SMAA) [828, 830, 834], in part because both provide

solid (and free) source code implementations for a variety of machines. Both

algorithms use color-only input, with SMAA having the advantage of being able to

access MSAA samples. Each has its own variety of settings available, trading off

between speed and quality. Costs are generally in the range of 1 to 2 milliseconds

per frame, mainly because that is what video games are willing to spend. Finally,

both algorithms can also take advantage of temporal antialiasing [1812]. Jimenez

[836] presents an improved SMAA implementation, faster than FXAA, and describes

a temporal antialiasing scheme. To conclude, we recommend the reader to the wide-

ranging review by Reshetov and Jimenez [1486] of morphological techniques and

their use in video games.

尽管如此，基于图像方法可以为有限的内存和处理成本提供抗锯齿支持，因此它们在许多用应用程序中得到应用。只要颜色的版本也与渲染管道解耦，使它们更容易修改或禁用，甚至可以作为GPU驱动程序选项公开。两种最流行的算法是快速近似抗锯齿（FXAA）[1079,1080,1084]，和子像素形态学抗锯齿（SMAA）[828,830,834]，部分原因是他们都为各种机器提供了可靠的（和免费的）源代码实现。两种算法都使用纯颜色输入，SMAA具有访问MSAA样本的优势。每个都有自己的设置，在速度和质量上做出权衡。成本通常是每帧1到2毫秒之间，主要是因为这是视频游戏愿意花费的时间。最后，两种算法都可以利用时间抗锯齿[1812]。Jimenez[836]提供了一种改进的SMAA实现，比FXAA更快，并描述了一种时间抗锯齿方案。最后，我们推荐阅读者Reshetov和Jimenez[1486]对形态学技术及其在电子游戏中的应用所作的广泛综述。

5.5 Transparency, Alpha, and Compositing 透明，透明度和合成

There are many different ways in which semitransparent objects can allow light to

pass through them. For rendering algorithms, these can be roughly divided into light-

based and view-based effects. Light-based effects are those in which the object

causes light to be attenuated or diverted, causing other objects in the scene to be lit

and rendered differently. View-based effects are those in which the semitransparent

object itself is being rendered.

半透明物体允许光通过他们的方式有很多种。对于渲染算法，可以大致分为基于光和基于视图的效果，基于光的效果是指物体使光线减弱或转移，导致场景中的其他物体被照亮并渲染出不同的效果。基于视图的效果是那些渲染半透明对象本身的效果。

In this section we will deal with the simplest form of view-based transparency, in

which the semitransparent object acts as an attenuator of the colors of the objects

behind it. More elaborate view- and light-based effects such as frosted glass, the

bending of light (refraction), attenuation of light due to the thickness of the

transparent object, and reflectivity and transmission changes due to the viewing

angle are discussed in later chapters.

在本节中，我们将处理基于视图的透明的最简单形式，其中半透明对象充当其背后对象颜色的衰减器。更精细的基于视图和光的效果，如磨砂玻璃，光的弯曲（折射），由于透明物体的厚度光的衰减，由于观察视角反射率和透射率的变化等将在后面的章节中讨论。

One method for giving the illusion of transparency is called screen-door transparency

[1244]. The idea is to render the transparent triangle with a pixel-aligned

checkerboard fill pattern. That is, every other pixel of the triangle is rendered,

thereby leaving the object behind it partially visible. Usually the pixels on the screen

are close enough together that the checkerboard pattern itself is not visible. A major

drawback of this method is that often only one transparent object can be

convincingly rendered on one area of the screen. For example, if a transparent red

object and transparent green object are rendered atop a blue object, only two of the

three colors can appear on the checkerboard pattern. Also, the 50% checkerboard is

limiting. Other larger pixel masks could be used to give other percentages, but these

tend to create detectable patterns [1245].

一种给人透明错觉的方法叫纱门透明[1244]。其想法是用像素对齐的棋盘填充模式渲染透明三角形。也就是说，三角形的其他每个像素都被渲染，从而使后面的对象部分可见。通常，屏幕上的像素距离足够近，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的渲染一个透明对象。例如，如果透明的红色对象的透明的绿色对象渲染在蓝色对象之上，那么这三种颜色中只有两种可以出现在棋盘格模式中。另外，50%的棋盘是有限的。其他较大的像素遮罩可以用来给出其他百分比，但是这些遮罩往往会创建可检测的模式[1245]。

That said, one advantage of this technique is its simplicity. Transparent objects can

be rendered at any time, in any order, and no special hardware is needed. The

transparency problem goes away by making all objects opaque at the pixels they

cover. This same idea is used for antialiasing edges of cutout textures, but at a

subpixel level, using a feature called alpha to coverage (Section 6.6).

也就是说，这项技术的一大优点是简单。透明物体可以在任何时间，任何顺序渲染并且不要特殊硬件。透明度问题通过使所有对象在其覆盖的像素处不透明来解决。同样的想法也适用于裁剪纹理的抗锯齿边缘，但是在亚像素极，使用透明度覆盖（章节6.6）。

Introduced by Enderton et al. [423], stochastic transparency uses subpixel screen

door masks combined with stochastic sampling. A reasonable, though noisy, image is

created by using random stipple patterns to represent the alpha coverage of the

fragment. See Figure 5.31. A large number of samples per pixel is needed for the

result to look reasonable, as well as a sizable amount of memory for all the subpixel

samples. What is appealing is that no blending is needed, and antialiasing,

transparency, and any other phenomena that creates partially covered pixels are

covered by a single mechanism.

由Enderton等[423]提出，随机透明使用亚像素纱门遮罩结合随机采样。一个合理的，虽然有噪声，图像是通过使用随机点画模式来表示片元的透明度覆盖创建的。参见图5.31。为了使结果看起来合理，每个像素需要大量的样本，并且所有的亚像素样本都需要相当大的内存。吸引人的是，不需要混合，而且抗锯齿，透明度和其他任何创建部分覆盖像素的现象都是由一个机制覆盖。

Most transparency algorithms blend the transparent object’s color with the color of

the object behind it. For this, the concept of alpha blending is needed [199, 387,

1429]. When an object is rendered on the screen, an RGB color and a z-buffer depth

are associated with each pixel. Another component, called alpha (α), can also be

defined for each pixel the object covers. Alpha is a value describing the degree of

opacity and coverage of an object fragment for a given pixel. An alpha of 1.0 means

the object is opaque and entirely covers the pixel’s area of interest; 0.0 means the

pixel is not obscured at all, i.e., the fragment is entirely transparent.

大部分透明算法混合透明物体颜色和他后面对象的颜色。对于此，透明度混合的概念是必须的[199,387,1429]。当一个物体渲染在屏幕上，一个RGB颜色和z深度缓冲与每个像素联系起来。另外一个组件，称为透明度（α），也可以为每个像素定义对象覆盖。透明度是一个值描述了给定像素的对象片元的不透明度和覆盖率。Alpha值为1.0意味着对象是不透明的，并且完全覆盖了像素感兴趣的区域；0.0意味着像素完全不被遮挡，即，片元是完全透明的。

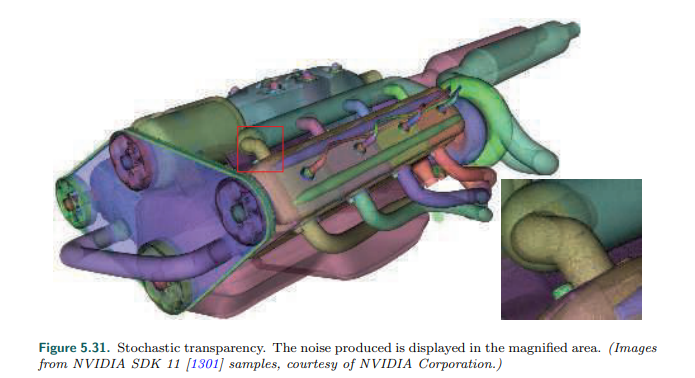


图5.31 随机透明。产生的噪声显示在放大的区域。（图像来自英伟达Sdk11[1301]样本，由英伟达公司提供。）

A pixel’s alpha can represent either opacity, coverage, or both, depending on the

circumstances. For example, the edge of a soap bubble may cover three-quarters of

the pixel, 0.75, and may be nearly transparent, letting nine-tenths of the light

through to the eye, so it is one-tenth opaque, 0.1. Its alpha would then be 0.75 ×

0.1 = 0.075. However, if we were using MSAA or similar antialiasing schemes, the

coverage would be taken into account by the samples themselves. Three-quarters of

the samples would be affected by the soap bubble. At each of these samples we

would then use the 0.1 opacity value as the alpha.

像素的alpha可以代表不透明度、覆盖率，也可以同时表示两者，取决于对应的情况。比如，肥皂泡的边缘可能覆盖像素的四分之三，0.75，而且可能几乎是透明的，让十分之9的光线进入眼睛，所以他是十分之一不透明，0.1。他的alpha可以是0.75x0.1=0.075。然而，如果我们使用MSAA或者相似的抗锯齿方案，覆盖率将有样本本身考虑。四分之三的样本收到了肥皂泡的影响。在每个样本中，我们将使用0.1的不透明度值作为alpha值。

5.5.1 Blending Order 混合顺序

To make an object appear transparent, it is rendered on top of the existing scene

with an alpha of less than 1.0. Each pixel covered by the object will receive a

resulting RGBα (also called RGBA) from the pixel shader. Blending this fragment’s

value with the original pixel color is usually done using the over operator, as follows:

为了使物体出现透明，alpha值小于1.0可以在现有场景的顶部渲染。对象覆盖的每个像素都将从像素着色器中收到一个结果RGBα（也成为RGBA）。将该片元的值与原始颜色混合，通常使用over操作符，如下所示：



where cs is the color of the transparent object (called the source), αs is the object’s

alpha, cd is the pixel color before blending (called the destination), and co is the

resulting color due to placing the transparent object over the existing scene. In the

case of the rendering pipeline sending in cs and αs, the pixel’s original color cd gets

replaced by the result co. If the incoming RGBα is, in fact, opaque (αs = 1.0), the

equation simplifies to the full replacement of the pixel’s color by the object’s color.

Cs是透明物体的颜色（也成为源颜色），αs是物体的alpha，cd是混合之前的像素颜色（也成为目标颜色），以及co是将透明物体放置在当前场景上的结果颜色。在渲染管道的情况下，发送cs 和 αs，像素原始颜色cd将会被结果co代替。如果传入的RGBA，实际上，不透明（as=1.0），这个方程简化为像素的颜色将完全被物体的颜色替换。

Example: Blending. A red semitransparent object is rendered onto a blue background.

Say that at some pixel the RGB shade of the object is (0.9, 0.2, 0.1), the background

is (0.1, 0.1, 0.9), and the object’s opacity is set at 0.6. The blend of these two colors

is then which gives a color of (0.58, 0.16, 0.42).

例子：混合。红色半透明物体在蓝色背景上渲染。假设在某个像素处，对象的RGB着色为（0.9,0.2,0.1），背景是（0.1,0.1,0.9），对象的不透明度为0.6。两种颜色混合之后的结果是： 颜色（0.58,0.16,0.42）。

The over operator gives a semitransparent look to the object being rendered.

Transparency done this way works, in the sense that we perceive something as

transparent whenever the objects behind can be seen through it [754]. Using over

simulates the real-world effect of a gauzy fabric. The view of the objects behind the

fabric are partially obscured—the fabric’s threads are opaque. In practice, loose

fabric has an alpha coverage that varies with angle [386]. Our point here is that

alpha simulates how much the material covers the pixel.

Over操作符给渲染的对象一个半透明的外观。透明性就是这样工作的，在某种意义上，当我们看到后面的物体时，我们就认为它是透明的[754]。使用over模拟薄纱织物的真实效果。织物后面的对象视图部分是模糊的——织物线程是不透明的。在实践中，松散的织物具有随角度变化的alpha覆盖[386]。我们的这里的重点是alpha模拟了材质覆盖像素的程度。

The over operator is less convincing simulating other transparent effects, most

notably viewing through colored glass or plastic. A red filter held in front of a blue

object in the real world usually makes the blue object look dark, as this object

reflects little light that can pass through the red filter. See Figure 5.32. When over is

used for blending, the result is a portion of the red and the blue added together. It

would be better to multiply the two colors together, as well as adding in any

reflection off the transparent object itself. This type of physical transmittance is

discussed in Sections 14.5.1 and 14.5.2.

Over操作在模拟其他透明效果方面的说服力较差，尤其是通过彩色玻璃或塑料进行观察。在真实世界中，蓝色物体前面覆盖着红色滤镜会使蓝色物体看起来是黑色的，因为物体反射少量的光可以通过红色滤镜。见图5.32。当使用over进行混合时，结果是红色和蓝色的一部分加在了一起。最好是将两种颜色相乘，并添加透明物体本身的反射。这种物理透过率在章节14.5.1和14.5.2节中讨论。

Of the basic blend stage operators, over is the one commonly used for a

transparency effect [199, 1429]. Another operation that sees some use is additive

blending, where pixel values are simply summed. That is,

在基本的混合阶段操作中，over通常用于透明效果[199,1429]。另一种有一定用途的操作是加法混合，他简单的对像素值求和，即，



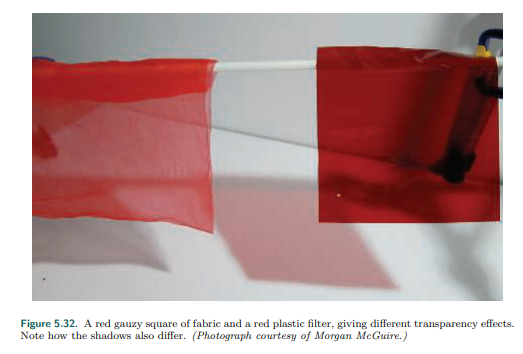


图5.32 红色薄纱块和红色玻璃滤镜，得到不同的透明效果。注意阴影也是不同的。（相片由Morgan McGuire提供）。

This blending mode can work well for glowing effects such as lightning or sparks that

do not attenuate the pixels behind but instead only brighten them [1813]. However,

this mode does not look correct for transparency, as the opaque surfaces do not

appear filtered [1192]. For several layered semitransparent surfaces, such as smoke

or fire, additive blending has the effect of saturating the colors of the phenomenon

[1273].

这种混合模式可以很好的用于发光效果，如闪电或火花，这些发光效果不会减弱后面的像素，而只会使它们变亮[1813]。然而，这种模式在透明度方面看起来并不正确，因为不透明的表面没有经过过滤[1192]。具有多层半透明表面，比如烟或者火，添加混合具有饱和现象颜色的效果[1273]。

To render transparent objects properly, we need to draw them after the opaque

objects. This is done by rendering all opaque objects first with blending off, then

rendering the transparent objects with over turned on. In theory we could always

have over on, since an opaque alpha of 1.0 would give the source color and hide the

destination color, but doing so is more expensive, for no real gain.

为了正确的渲染透明物体，我们需要在不透明物体的后面渲染他们。首先渲染所有不透明物体并关闭混合，然后渲染透明物体并开启over。理论上，我们应该一直开启over，因为一个不透明物体的alpha为1.0，将给出源颜色而隐藏目标颜色，并且这么做是昂贵的，对真实性没有一点增长。

A limitation of the z-buffer is that only one object is stored per pixel. If several

transparent objects overlap the same pixel, the z-buffer alone cannot hold and later

resolve the effect of all the visible objects. When using over the transparent surfaces

at any given pixel generally need to be rendered in back-to-front order. Not doing so

can give incorrect perceptual cues. One way to achieve this ordering is to sort

individual objects by, say, the distance of their centroids along the view direction.

This rough sorting can work reasonably well, but has a number of problems under

various circumstances. First, the order is just an approximation, so objects classified

as more distant may be in front of objects considered nearer. Objects that

interpenetrate are impossible to resolve on a per-mesh basis for all view angles,

short of breaking each mesh into separate pieces. See the left image in Figure 5.33

for an example. Even a single mesh with concavities can exhibit sorting problems for

view directions where it overlaps itself on the screen.

Z缓冲区的一个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个像素上，则z缓冲区无法单独保存并在稍后解析所有可见对象的效果。当在任何给定像素的透明表面上使用over时，通常需要按前后顺序渲染。不这样做会给出错误的感知提示。实现这种排序的一种方法是对单个对象进行排序，例如，根据其中心点沿视图方向的距离。这种粗略的排序可以很好地工作，但是在不同的情况下会有很多问题。首先，这个顺序只是一个近似值，所以被分为较远的物体可能在被认为较近的物体前面。相互渗透的对象不可能在每个网络的基础上解决所有视角，除非将每个网络分割成单独的部分。有关示例，请参见图5.33中的左边图像。即使是一个带有空洞的网格，在屏幕上与自身重叠的视图方向上也会出现排序问题。

Nonetheless, because of its simplicity and speed, as well as needing no additional

memory or special GPU support, performing a rough sort for transparency is still

commonly used. If implemented, it is usually best to turn off z-depth replacement

when performing transparency. That is, the z-buffer is still tested normally, but

surviving surfaces do not change the z-depth stored; the closest opaque surface’s

depth is left intact. In this way, all transparent objects will at least appear in some

form, versus suddenly appearing or disappearing when a camera rotation changes

the sort order. Other techniques can also help improve the appearance, such as

drawing each transparent mesh twice as you go, first rendering backfaces and then

frontfaces [1192, 1255].

尽管如此，由于它的简单和速度，同样不需要增加内存或者特殊的GPU支持，执行一个粗略的透明排序仍然是十分常见的。如果实现了，通产最好在执行透明时关闭z深度替换。即，z缓冲区仍然正常测试，但存活的表面不会改变存储的z深度值；最接近不透明表面的深度保持不变。这样，所有透明的物体至少都会以某种形式出现，而不是在相机旋转改变排序顺序时突然出现或消失。其他技术也可以帮助改善外观，比如每次绘制两次透明网格，首先绘制背面，然后绘制正面[1192,1255]。

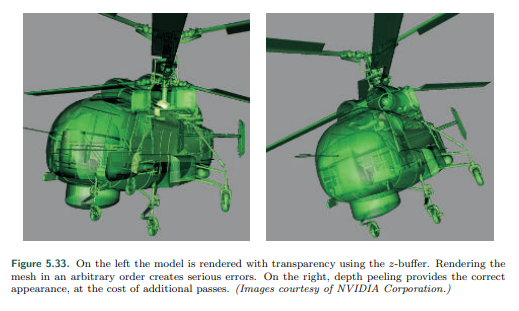


图5.33.左侧模型通过使用z缓冲区渲染。呈现在任意顺序的网格会产生严重的错误。在右边，深度剥离提供了正确的选项外观，以额外的通道为代价（图片由英伟达公司提供）。

The over equation can also be modified so that blending front to back gives the

same result. This blending mode is called the under operator:

Over方程式也可以修改，使混合前后得到相同的结果。这种混合模式称为under操作：



Note that under requires the destination to maintain an alpha value, which over does

not. In other words, the destination—the closer transparent surface being blended

under—is not opaque and so needs to have an alpha value. The under formulation is

like over, but with source and destination swapped. Also, notice that the formula for

computing alpha is order-independent, in that the source and destination alphas can

be swapped, with the same final alpha being the result.

注意，Under要求目标要维持alpha值，而over不需要。换句话说，目标——正在混合的较近透明表面——不是不透明的，因此需要一个alpha值。Under公式就像over，但是源和目标交换了。另外，注意计算alpha的公式是与顺序无关的，因为可以交换源和目标的的alpha值，最终alpha是相同的结果。

The equation for alpha comes from considering the fragment’s alphas as coverages.

Porter and Duff [1429] note that since we do not know the shape of the coverage

area for either fragment, we assume that each fragment covers the other in

proportion to its alpha. For example, if αs = 0.7, the pixel is somehow divided into

two areas, with 0.7 covered by the source fragment and 0.3 not. Barring any other

knowledge, the destination fragment covering, say, αd = 0.6 will be proportionally

overlapped by the source fragment. This formula has a geometric interpretation,

shown in Figure 5.34.

Alpha方程来自考虑碎片的alpha作为覆盖率。Porter 和 Duff[1429]指出，由于我们不知道任何一个片段的覆盖区域的形状，我们假定每个片元覆盖另一个片元的面积与它的alpha值成正比。例如，如果as=0.7，像素分为两个区域，0.7被源片元覆盖而0.3没有。除去其他的认知，目标片元覆盖，即，ad=0.6将成比例的被源片元覆盖。这个公式具有几个解释，如图5.34所示。

5.5.2 Order-Independent Transparency 顺序无关透明

The under equations are used by drawing all transparent objects to a separate color

buffer, then merging this color buffer atop the opaque view of the scene using over.

Another use of the under operator is for performing an order-independent

transparency (OIT) algorithm known as depth peeling [449, 1115]. Order-

independent means that the application does not need to perform sorting. The idea

behind depth peeling is to use two z-buffers and multiple passes. First, a rendering

pass is made so that all surfaces’ z-depths, including transparent surfaces, are in the

first z-buffer. In the second pass all transparent objects are rendered. If the z-depth

of an object matches the value in the first z-buffer, we know this is the closest

transparent object and save its RGBα to a separate color buffer. We also “peel” this

layer away by saving the z-depth of whichever transparent object, if any, is beyond

the first z-depth and is closest. This z-depth is the distance of the second-closest

transparent object. Successive passes continue to peel and add transparent layers

using under. We stop after some number of passes and then blend the transparent

image atop the opaque image. See Figure 5.35.

Under方程式用途是通过将所有透明的物体绘制在独立的颜色缓冲区，然后使用over操作将这个颜色缓冲区和场景中顶部的不透明视图合并。Under另一种操作是执行顺序无关的透明（OIT）算法，称为深度剥离[449，1115]。顺序无关意味着应用程序不需要执行排序。深度剥离后面的理念是使用2个z缓冲区和多重通道。首先，生成一个渲染通道以便所有表面的z深度，包括透明表面都位于第一个z缓冲区中。第二个通道，所有透明物体进行渲染。如果物体的z深度与第一个z缓冲区的值匹配，我们知道，这是最靠近的透明物体，然后保存他的RGBa到一个独立的颜色缓冲区。我们还通过保存任何透明对象的z深度（如果有法人话）来“剥离”这一层，该透明对象超过第一个z深度且距离最近。这个z深度是第二个最近的透明物体的距离。连续通道继续剥离和使用under添加透明层。我们在经历一些通道后停止，然后将透明图像混合到不透明图像之上。参见图5.35。

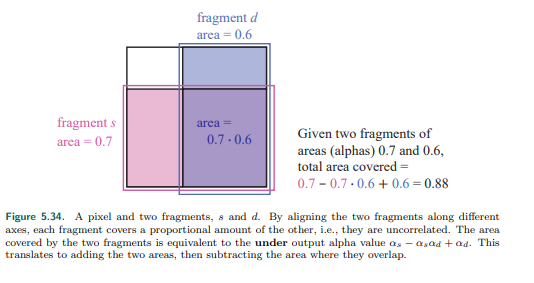


图5.34.一个像素和两个片元。S和d。通过沿着不同的轴调整两个片元，每个片元覆盖了另一个片元的一定比例，即，他们是不想关的。被两个片元覆盖的区域等于under输出alpha值as-asad+ad。翻译过来就是增加两个区域，然后减去重叠的地方。

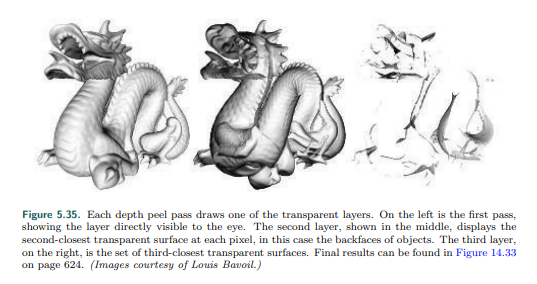


图5.35 每个深度剥离通道绘制一个透明层。左侧是第一个通道，显示了该层直接可见的部分。第二层（显示在中间）显示了第二个最接近每个像素的透明表面，在本例中为物体的背面。第三层，右侧是第三个最近的透明表面。最终结果如图14.33所示，在624页（图片由louis bavoil提供）。

Several variants on this scheme have been developed. For example, Thibieroz [1763]

gives an algorithm that works back to front, which has the advantage of being able

to blend the transparent values immediately, meaning that no separate alpha

channel is needed. One problem with depth peeling is knowing how many passes are

sufficient to capture all the transparent layers. One hardware solution is to provide a

pixel draw counter, which tells how many pixels were written during rendering; when

no pixels are rendered by a pass, rendering is done. The advantage of using under is

that the most important transparent layers—those the eye first sees—are rendered

early on. Each transparent surface always increases the alpha value of the pixel it

covers. If the alpha value for a pixel nears 1.0, the blended contributions have made

the pixel almost opaque, and so more distant objects will have a negligible effect

[394]. Front-to-back peeling can be cut short when the number of pixels rendered by

a pass falls below some minimum, or a fixed number of passes can be specified. This

does not work as well with back-to-front peeling, as the closest (and usually most

important) layers are drawn last and so may be lost by early termination.

已经开发了该方案的几个变体。例如，Thibieroz[1763]给出了一种算法可以从后向前工作，他的优点是可以立即混合透明度值，意味着不需要独立的alpha通道。深度剥离的一个问题是知道需要多少通道才能捕获所有的透明层。一种硬件解决方案是提供一个像素绘制计数器，它告诉绘制过程中写入了多少像素；当一个通道没有像素渲染时，就结束渲染。使用under的好处是最重要的透明层——眼睛最先看到的层——最早渲染。每个透明的表面总是增加他所覆盖像素的alpha值。如果一个像素的alpha值接近1.0，混合的贡献使得像素几乎不透明，因此距离更远的物体将产生微不足道的影响[394]。当一个通道渲染的像素低于某个最小值，或者指定一个固定的通道数时，可以通断从前到后的剥离。这对于从后向前剥离不太好，因为最近的（通常也是最重要的）层是最后绘制的，因此可能会在早期终止时丢失。

While depth peeling is effective, it can be slow, as each layer peeled is a separate

rendering pass of all transparent objects. Bavoil and Myers [118] presented dual

depth peeling, where two depth peel layers, the closest and the farthest remaining,

are stripped off in each pass, thus cutting the number of rendering passes in half. Liu

et al. [1056] explore a bucket sort method that captures up to 32 layers in a single

pass. One drawback of this type of approach is that it needs considerable memory to

keep a sorted order for all layers. Antialiasing via MSAA or similar would increase the

costs astronomically.

尽管深度剥离是有效的，它可以是缓慢的，因为所有的透明对象来说每一层剥离都是一个独立的渲染通道。Bavoil 和 Myers[118]提出双重深度剥离，指的是两个深度剥离层，距离最近和最远的保留，每个通道去掉，从而渲染通道的数量减半。Liu等[1056]探索了一种桶式排序方法，在一个通道中一次可以捕获多达32层。这种方法的一个缺点是，他需要相当大的内存去保存所有层的排序顺序。通过MSAA类似的抗锯齿将巨额增加成本。

The problem of blending transparent objects together properly at interactive rates is

not one in which we are lacking algorithms, it is one of efficiently mapping those

algorithms to the GPU. In 1984 Carpenter presented the A-buffer [230], another

form of multisampling. In the A-buffer, each triangle rendered creates a coverage

mask for each screen grid cell it fully or partially covers. Each pixel stores a list of all

relevant fragments. Opaque fragments can cull out fragments behind them, similar

to the z-buffer. All the fragments are stored for transparent surfaces. Once all lists

are formed, a final result is produced by walking through the fragments and

resolving each sample.

以交互速率将透明对象正确地混合在一起额问题并不是我们缺少算法，而是将这些算法有效地映射到GPU的问题之一。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，渲染的每个三角形为他完全或部分覆盖的每个屏幕网格单元创建一个覆盖掩码。每个像素存储所有相关片段的列表。不透明的片段可以剔除它们后面的片段，类似于z缓冲区。所有的片元都存储在透明的表面。一旦所有列表都形成，通过遍历片元并解析每个示例就会产生最终的结果。

The idea of creating linked lists of fragments on the GPU was made possible through

new functionality exposed in DirectX 11 [611, 1765]. The features used include

unordered access views (UAVs) and atomic operations, described in Section 3.8.

Antialiasing via MSAA is enabled by the ability to access the coverage mask and to

evaluate the pixel shader at every sample. This algorithm works by rasterizing each

transparent surface and inserting the fragments generated in a long array. Along

with the colors and depths, a separate pointer structure is generated that links each

fragment to the previous fragment stored for the pixel. A separate pass is then

performed, where a screen-filling quadrilateral is rendered so that a pixel shader is

evaluated at every pixel. This shader retrieves all the transparent fragments at each

pixel by following the links. Each fragment retrieved is sorted in turn with the

previous fragments. This sorted list is then blended back to front to give the final

pixel color. Because blending is performed by the pixel shader, different blend modes

can be specified per pixel, if desired. Continuing evolution of the GPU and APIs have

improved performance by reducing the cost of using atomic operators [914].

在GPU上创建片元链表的想法是通过directX11[611,1765]中公开的新功能实现的。所使用的特性包括无序访问视图（UAVs）和原子操作，见第3.8节。通过MSAA的抗锯齿功能是通过访问覆盖掩码和在每个样本中评估像素着色器。该算法对每个透明表面进行光栅化，并将生成的片元插入到一个长数组中。除了颜色和深度之外，还生成一个单独的指针结构，为像素将每个片元与之前的片元进行连接。然后执行一个单独的通道，其中渲染一个填充屏幕的四边形，以便在每个像素处计算像素着色器。该着色器通过跟踪连接检索每个像素处的所有透明片元。检索到的每个片元依次与前面的片元进行排序。然后将这个排序后的列表从后往前混合，以得到最后的颜色。由于混合是由像素着色器执行的，如果需要，可以为每个像素指定不同的混合模式。GPU和api的不断发展通过降低使用原子操作符的成本提高了性能[914]。

The A-buffer has the advantage that only the fragments needed for each pixel are

allocated, as does the linked list implementation on the GPU. This in a sense can also

be a disadvantage, as the amount of storage required is not known before rendering

of a frame begins. A scene with hair, smoke, or other objects with a potential for

many overlapping transparent surfaces can produce a huge number of fragments.

Andersson [46] notes that, for complex game scenes, up to 50 transparent meshes

of objects such as foliage and up to 200 semitransparent particles may overlap.

A-缓冲区具有优点是，当GPU链接链表执行时，只有每个像素上的片元会分配。这有一种感觉，也可能是一种缺点，因为在开始渲染帧之前，还不知道需要多少存储空间。一个有毛发、烟雾或其他物体的场景，可能会产生多重叠的透明表面，从而产生大量的碎片。Andersson [46]指出，对于复杂的游戏场景，多达50个透明网格的物体（如树叶）和多达200个半透明粒子可能会重叠。

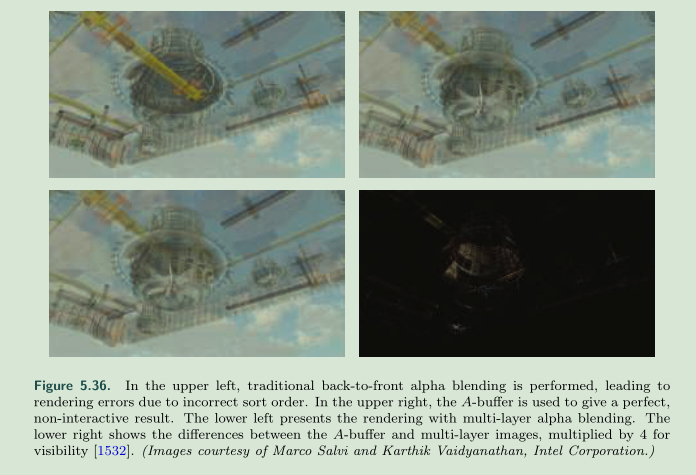


图5.36 在上左方，执行了传统的从后往前的透明混合，由于排序顺序不正确导致了渲染错误。在上部右侧，a-缓冲区用来表示完美，非交互式的结果。左下角是多层透明混合的渲染图。右下角不同于a-缓冲和多层图像，为了可见性乘以4倍[1532]。

GPUs normally have memory resources such as buffers and arrays allocated in

advance, and linked-list approaches are no exception. Users need to decide how

much memory is enough, and running out of memory causes noticeable artifacts.

Salvi and Vaidyanathan [1532] present an approach tackling this problem, multi-layer

alpha blending, using a GPU feature introduced by Intel called pixel synchronization.

See Figure 5.36. This capability provides programmable blending with less overhead

than atomics. Their approach reformulates storage and blending so that it gracefully

degrades if memory runs out. A rough sort order can benefit their scheme. DirectX

11.3 introduced rasterizer order views (Section 3.8), a type of buffer that allows this

transparency method to be implemented on any GPU supporting this feature [327,

328]. Mobile devices have a similar technology called tile local storage that permits

them to implement multi-layer alpha blending [153]. Such mechanisms have a

performance cost, however, so this type of algorithm can be expensive [1931].

Gpu通常有内存资源，比如缓冲区和高级方法数组分配，和链表方法也不例外。用户需要决定多少内存已经足够了，内存不足会导致明显的伪影。Salvi 和Vaidyanathan[1532]提供了一种方法处理这种问题，多层透明混合，使用因特尔引入的称为像素同步方法的gpu功能。见图5.36。该功能提供了可编程的混合和更少的院子开销。他们的方法重新定义了存储和混合，以便在内存耗尽时可以优雅地降级。粗略的排序顺序对他们的方案有利。Dx11.3引入了光栅化序列视图（章节3.8），一种允许这种透明方法在任何支持该特性的gpu上实现的缓冲区类型[327.328]。移动设备有相似的技术称为平铺本地存储，实现多层透明混合[153]。这样的机制有一定的性能消耗，因此这种算法类型可能是昂贵的。

This approach builds on the idea of the k-buffer, introduced by Bavoil et al. [115],

where the first few visible layers are saved and sorted as possible, with deeper layers

discarded and merged as possible. Maule et al. [1142] use a k-buffer and account for

these more distant deep layers by using weighted averaging. Weighted sum [1202]

and weighted average [118] transparency techniques are order-independent, are

single-pass, and run on almost every GPU. The problem is that they do not take into

account the ordering of the objects. So, for example, using alpha to represent

coverage, a gauzy red scarf atop a gauzy blue scarf gives a violet color, versus

properly seeing a red scarf with a little blue showing through. While nearly opaque

objects give poor results, this class of algorithms is useful for visualization and works

well for highly transparent surfaces and particles. See Figure 5.37.

这种方法建立在k-缓冲区的思想上，由Bavoil等人[115]提出，在k-缓冲区中，前几层可见的层被尽可能的保存和排序，更深处的层被尽可能的丢弃和合并。Maule等人[1142]使用k-缓冲区，通过加权平均来解释这些较远的深层。加权和[1202]和加权平均[118]透明技术是独立与顺序的，是单通道的，并且几乎可以在所有的gpu上运行。问题是他们没有考虑队形的顺序，例如，使用透明度来表示覆盖范围，薄纱的红色围巾盖在薄纱的蓝色围巾上，就会产生紫罗兰色，而在正常情况下，红色围巾会透出一点蓝色。虽然几乎不透明的物体给出的结果很差，但是这类算法对可视化很有用，并且在高度透明的表面和粒子也很有效果。参见图5.37。

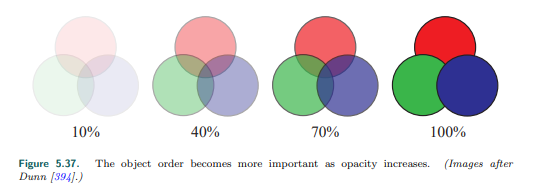


图5.37 随着不透明度的增加，物体的顺序变得更重要。

In weighted sum transparency the formula is



在加权和透明度中，公式如下。

where n is the number of transparent surfaces, ci and αi represent the set of

transparency values, and cd is the color of the opaque portion of the scene. The two

sums are accumulated and stored separately as transparent surfaces are rendered,

and at the end of the transparency pass, the equation is evaluated at each pixel.

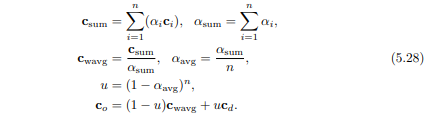
Problems with this method are that the first sum saturates, i.e., generates color

values greater than (1.0, 1.0, 1.0), and that the background color can have a

negative effect, since the sum of the alphas can surpass 1.0.

n是透明表面的个数，ci和αi代表着一系列透明值，cd是场景中不透明部分的颜色。当透明表面被渲染时，这两个和被分别累加和存储，在透明通道结束时，在每个像素处对方程求值，这个方法的问题是第一个值饱和，即会产生大于（1.0，1.0，1.0）的颜色值，而且背景颜色可能会有负面影响，因为不透明通道的总和可以超过1.0。

The weighted average equation is usually preferred because it avoids these problems:



加权平均方程通常更好，因为他避免了这些问题。

The first line represents the results in the two separate buffers generated during

transparency rendering. Each surface contributing to csum is given an influence

weighted by its alpha; nearly opaque surfaces contribute more of their color, and

nearly transparent surfaces have little influence. By dividing csum by αsum we get a

weighted average transparency color. The value αavg is the average of all alpha

values. The value u is the estimated visibility of the destination (the opaque scene)

after this average alpha is applied n times, for n transparent surfaces. The final line

is effectively the over operator, with (1 − u) representing the source’s alpha.

第一行表示在透明渲染期间生成的两个独立缓冲区中的结果。每一个对csum有贡献的表面都被赋予一个加权的影响，接近不透明的颜色贡献了更多的颜色，而接近透明的表面几乎没有影响。Csum除以αsum，我们得到一个加权平均的颜色。Αavg值是所有不透明度值的平均值。值u是对n个透明表面应用这个平均不透明值n次方后对目标（不透明场景）的可见性的评估。最后一行实际上是over运算符，（1-u）表示源的不透明值。

One limitation with weighted average is that, for identical alphas, it blends all colors equally, regardless of order. McGuire and Bavoil [1176, 1180] introduced weighted blended order-independent transparency to give a more convincing result. In their formulation, the distance to the surface also affects the weight, with closer surfaces given more influence. Also, rather than averaging the alphas, u is computed by multiplying the terms (1 − αi) together and subtracting from one, giving the true alpha coverage of the set of surfaces. This method produces more visually convincing results, as seen in Figure 5.38.

加权平均算法的一个限制是：对于相同的不透明度值，他不管顺序，混合相同的颜色值。McGuire和 Bavoil [1176, 1180]